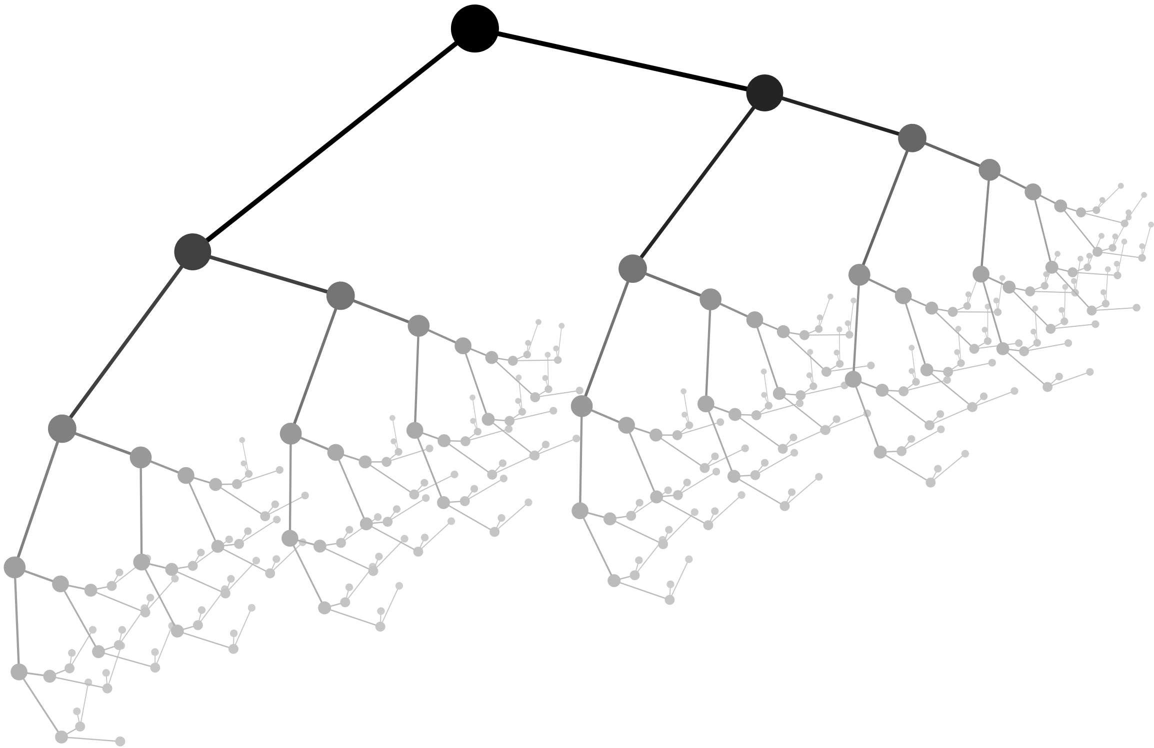
An Introduction to Binary Search Trees and Balanced Trees

LIBAVL Binary Search Tree Library
Volume 1: Source Code
Version 2.0.2



by Ben Pfaff

The author may be contacted as `blp@gnu.org` on the Internet, or write to:

Ben Pfaff
Stanford University
Computer Science Dept.
353 Serra Mall
Stanford CA 94305
UNITED STATES

Brief Contents

Table of Contents

Preface

Early in 1998, I wanted an AVL tree library for use in writing GNU PSPP. At the time, few of these were available on the Internet. Those that were had licenses that were not entirely satisfactory for inclusion in GNU software. I resolved to write my own. I sat down with Knuth's *The Art of Computer Programming* and did so. The result was the earliest version of Libavl. As I wrote it, I learned valuable lessons about implementing algorithms for binary search trees, and covered many notebook pages with scribbled diagrams.

Later, I decided that what I really wanted was a similar library for threaded AVL trees, so I added an implementation to Libavl. Along the way, I ended up having to relearn many of the lessons I'd already painstakingly uncovered in my earlier work. Even later, I had much the same experience in writing code for right-threaded AVL trees and red-black trees, which was done as much for my own education as any intention of using the code in real software.

In late 1999, I contributed a chapter on binary search trees and balanced trees to a book on programming in C. This again required a good deal of duplication of effort as I rediscovered old techniques. By now I was beginning to see the pattern, so I decided to document once and for all the algorithms I had chosen and the tradeoffs I had made. Along the way, the project expanded in scope several times.

You are looking at the results. I hope you find that it is as useful for reading and reference as I found that writing it was enjoyable for me. As I wrote later chapters, I referred less and less to my other reference books and more and more to my own earlier chapters, so I already know that it can come in handy for me.

Please feel free to copy and distribute this book, in accordance with the license agreement. If you make multiple printed copies, consider contacting me by email first to check whether there are any late-breaking corrections or new editions in the pipeline.

Acknowledgements

Libavl has grown into its current state over a period of years. During that time, many people have contributed advice, bug reports, and occasional code fragments. I have attempted to individually acknowledge all of these people, along with their contributions, in the ‘`NEWS`’ and ‘`ChangeLog`’ files included with the Libavl source distribution. Without their help, Libavl would not be what it is today. If you believe that you should be listed in one of these files, but are not, please contact me.

Many people have indirectly contributed by providing computer science background and software infrastructure, without which Libavl would not have been possible at all. For a partial list, please see ‘`THANKS`’ in the Libavl source distribution.

Special thanks are due to Erik Goodman of the A. H. Case Center for Computer-Aided Engineering and Manufacturing at Michigan State University for making it possible for me to receive MSU honors credit for rewriting Libavl as a literate program, and to Dann Corbit for his invaluable suggestions during development.

Contacting the Author

Libavl, including this book, the source code, the TexiWEB software, and related programs, was written by Ben Pfaff, who welcomes your feedback. Please send Libavl-related correspondence, including bug reports and suggestions for improvement, to him at `blp@gnu.org`.

Ben received his B.S. in electrical engineering from Michigan State University in May 2001. He is now studying for a Ph.D. in computer science at Stanford University as a Stanford Graduate Fellow.

Ben's personal webpage is at `http://benpfaff.org/`, where you can find a list of his current projects, including the status of Libavl test releases. You can also find him hanging out in the Internet newsgroup `comp.lang.c`.

1 Introduction

Libavl is a library in ANSI C for manipulation of various types of binary trees. This book provides an introduction to binary tree techniques and presents all of Libavl's source code, along with annotations and exercises for the reader. It also includes practical information on how to use Libavl in your programs and discussion of the larger issues of how to choose efficient data structures and libraries. The book concludes with suggestions for further reading, answers to all the exercises, glossary, and index.

1.1 Audience

This book is intended both for novices interested in finding out about binary search trees and practicing programmers looking for a cookbook of algorithms. It has several features that will be appreciated by both groups:

- *Tested code*: With the exception of code presented as counterexamples, which are clearly marked, all code presented has been tested. Most code comes with a working program for testing or demonstrating it.
- *No pseudo-code*: Pseudo-code can be confusing, so it is not used.
- *Motivation*: An important goal is to demonstrate general methods for programming, not just the particular algorithms being examined. As a result, the rationale for design choices is explained carefully.
- *Exercises and answers*: To clarify issues raised within the text, many sections conclude with exercises. All exercises come with complete answers in an appendix at the back of the book.

  Some exercises are marked with one or more stars (*). Exercises without stars are recommended for all readers, but starred exercises deal with particularly obscure topics or make reference to topics covered later.

  Experienced programmers should find the exercises particularly interesting, because many of them present alternatives to choices made in the main text.
- *Asides*: Occasionally a section is marked as an "aside". Like exercises, asides often highlight alternatives to techniques in the main text, but asides are more extensive than most exercises. Asides are not essential to comprehension of the main text, so readers not interested may safely skip over them to the following section.
- *Minimal C knowledge assumed*: Basic familiarity with the C language is assumed, but obscure constructions are briefly explained the first time they occur.

  Those who wish for a review of C language features before beginning should consult [Summit 1999]. This is especially recommended for novices who feel uncomfortable with pointer and array concepts.
- *References*: When appropriate, other texts that cover the same or related material are referenced at the end of sections.
- *Glossary*: Terms are **emphasized** and defined the first time they are used. Definitions for these terms and more are collected into a glossary at the back of the book.
- *Catalogue of algorithms*: See Appendix E [Catalogue of Algorithms], page 405, for a handy list of all the algorithms implemented in this book.

1.2 Reading the Code

This book contains all the source code to Libavl. Conversely, much of the source code presented in this book is part of Libavl.

Libavl is written in ANSI/ISO C89 using TexiWEB, a **literate programming** system. Literate programming is a philosophy that regards software as a kind of literature. The ideas behind literate programming have been around for a long time, but the term itself was invented by computer scientist Donald Knuth in 1984, who wrote two of his most famous programs (TeX and METAFONT) with a literate programming system of his own design. That system, called `WEB`, inspired the form and much of the syntax of TexiWEB.

A TexiWEB document is a C program that has been cut into sections, rearranged, and annotated, with the goal to make the program as a whole as comprehensible as possible to a reader who starts at the beginning and reads the entire program in order. Of course, understanding large, complex programs cannot be trivial, but TexiWEB tries to make it as easy as possible.

Each section of a TexiWEB program is assigned both a number and a name. Section numbers are assigned sequentially, starting from 1 with the first section, and they are used for cross-references between sections. Section names are words or phrases assigned by the TexiWEB program's author to describe the role of the section's code.

Here's a sample TexiWEB section:

§19 ⟨ Clear hash table entries 19 ⟩ ≡
for ($i = 0$; $i < hash{\rightarrow}m$; i++)
    $hash{\rightarrow}entry[i]$ = `NULL`;

This code is included in §15.

The first line of a section, as shown here, gives the section's name and its number within angle brackets. The section number is also printed in the left margin to make individual sections easy to find. Looking farther down, at the code itself, the C operator `->` has been replaced by the nicer-looking arrow →. TexiWEB makes an attempt to "prettify" C in a few ways like this. The table below lists most of these substitutions:

`->`	becomes	→
0x12ab	becomes	`0x12ab`
0377	becomes	*0377*
1.2e34	becomes	$1.2{\cdot}10^{34}$

In addition, − and + are written as superscripts when used to indicate sign, as in $^{-}5$ or $^{+}10$.

In TexiWEB, C's reserved words are shown like this: **int**, **struct**, **while**. . . . Types defined with **typedef** or with **struct**, **union**, and **enum** tags are shown the same way. Identifiers in all capital letters (often names of macros) are shown like this: `BUFSIZ`, `EOF`, `ERANGE`. . . . Other identifiers are shown like this: *getc*, *argv*, *strlen*. . . .

Sometimes it is desirable to talk about mathematical expressions, as opposed to C expressions. When this is done, mathematical operators (≤, ≥) instead of C operators (<=, >=) are used. In particular, mathematical equality is indicated with ≡ instead of = in order to minimize potential confusion.

Code segments often contain references to other code segments, shown as a section name and number within angle brackets. These act something like macros, in that they stand for the corresponding replacement text. For instance, consider the following segment:

§15 $\langle$ Initialize hash table 15 $\rangle \equiv$
$hash \rightarrow m = 13$;
$\langle$ Clear hash table entries 19 $\rangle$

See also §16.

This means that the code for 'Clear hash table entries' should be inserted as part of 'Initialize hash table'. Because the name of a section explains what it does, it's often unnecessary to know anything more. If you do want more detail, the section number 19 in $\langle$ Clear hash table entries 19 $\rangle$ can easily be used to find the full text and annotations for 'Clear hash table entries'. At the bottom of section 19 you will find a note reading 'This code is included in §15.', making it easy to move back to section 15 that includes it.

There's also a note following the code in the section above: 'See also §16.'. This demonstrates how TexiWEB handles multiple sections that have the same name. When a name that corresponds to multiple sections is referenced, code from all the sections with that name is substituted, in order of appearance. The first section with the name ends with a note listing the numbers of all other same-named sections. Later sections show their own numbers in the left margin, but the number of the first section within angle brackets, to make the first section easy to find. For example, here's another line of code for $\langle$ Clear hash table entries 15 $\rangle$:

§16 $\langle$ Initialize hash table 15 $\rangle \mathrel{+}\equiv$
$hash \rightarrow n = 0$;

Code segment references have one more feature: the ability to do special macro replacements within the referenced code. These replacements are made on all words within the code segment referenced and recursively within code segments that the segment references, and so on. Word prefixes as well as full words are replaced, as are even occurrences within comments in the referenced code. Replacements take place regardless of case, and the case of the replacement mirrors the case of the replaced text. This odd feature is useful for adapting a section of code written for one library having a particular identifier prefix for use in a different library with another identifier prefix. For instance, the reference '$\langle$ BST types; bst $\Rightarrow$ avl $\rangle$' inserts the contents of the segment named 'BST types', replacing 'bst' by 'avl' wherever the former appears at the beginning of a word.

When a TexiWEB program is converted to C, conversion conceptually begins from sections named for files; e.g., $\langle$ '`foo.c`' 37 $\rangle$. Within these sections, all section references are expanded, then references within those sections are expanded, and so on. When expansion is complete, the specified files are written out.

A final resource in reading a TexiWEB is the index, which contains an entry for the points of declaration of every section name, function, type, structure, union, global variable, and macro. Declarations within functions are not indexed.

See also: [Knuth 1992], "How to read a `WEB`".

1.3 Code Conventions

Where possible, the Libavl source code complies to the requirements imposed by ANSI/ISO C89 and C99. Features present only in C99 are not used. In addition, most of the GNU Coding Standards are followed. Indentation style is an exception to the latter: in print, to conserve vertical space, K&R indentation style is used instead of GNU style.

See also: [ISO 1990]; [ISO 1999]; [FSF 2001], "Writing C".

1.4 License

This book, including the code in it, is subject to the following license:

§1 ⟨ License 1 ⟩ ≡

```
/* GNU Libavl - library for manipulation of binary trees.
   Copyright © 1998–2002, 2004 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
   See the GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
   02111-1307, USA.

   The author may be contacted at <blp@gnu.org> on the Internet, or
   write to Ben Pfaff, Stanford University, Computer Science Dept., 353
   Serra Mall, Stanford CA 94305, USA.
   */
```

This code is included in §24, §25, §97, §98, §99, §142, §143, §186, §192, §193, §238, §247, §248, §290, §297, §298, §330, §333, §334, §368, §372, §373, §411, §415, §416, §449, §452, §453, §482, §486, §487, §515, §519, §520, §548, §551, §552, §583, §595, §599, §617, and §649.

2 The Table ADT

Most of the chapters in this book implement a table structure as some kind of binary tree, so it is important to understand what a table is before we begin. That is this chapter's purpose.

This chapter begins with a brief definition of the meaning of "table" for the purposes of this book, then moves on to describe in a more formal way the interface of a table used by all of the tables in this book. The next chapter motivates the basic idea of a binary tree starting from simple, everyday concepts. Experienced programmers may skip these chapters after skimming through the definitions below.

2.1 Informal Definition

If you've written even a few programs, you've probably noticed the necessity for searchable collections of data. Compilers search their symbol tables for identifiers and network servers often search tables to match up data with users. Many applications with graphical user interfaces deal with mouse and keyboard activity by searching a table of possible actions. In fact, just about every nontrivial program, regardless of application domain, needs to maintain and search tables of some kind.

In this book, the term "table" does not refer to any particular data structure. Rather, it is the name for a abstract data structure or ADT, defined in terms of the operations that can be performed on it. A table ADT can be implemented in any number of ways. Later chapters will show how to implement tables in terms of various binary tree data structures.

The purpose of a table is to keep track of a collection of items, all of the same type. Items can be inserted into and deleted from a table, with no arbitrary limit on the number of items in the table. We can also search a table for items that match a given item.

Other operations are supported, too. Traversal is the most important of these: all of the items in a table can be visited, in sorted order from smallest to largest, or from largest to smallest. Traversals can also start from an item in the middle, or a newly inserted item, and move in either direction.

The data in a table may be of any C type, but all the items in a table must be of the same type. Structure types are common. Often, only part of each data item is used in item lookup, with the rest for storage of auxiliary information. A table that contains two-part data items like this is called a "dictionary" or an "associative array". The part of table data used for lookup, whether the table is a dictionary or not, is the **key**. In a dictionary, the remainder is the **value**.

Our tables cannot contain duplicates. An attempt to insert an item into a table that already contains a matching item will fail.

Exercises:

1. Suggest a way to simulate the ability to insert duplicate items in a table.

2.2 Identifiers

In C programming it is necessary to be careful if we expect to avoid clashes between our own names and those used by others. Any identifiers that we pick might also be used by others. The usual solution is to adopt a prefix that is applied to the beginning of every identifier that can be visible in code outside a single source file. In particular, most identifiers in a library's public header files must be prefixed.

LIBAVL is a collection of mostly independent modules, each of which implements the table ADT. Each module has its own, different identifier prefix. Identifiers that begin with this prefix are reserved for any use in source files that #**include** the module header file. Also reserved (for use as macro names) are identifiers that begin with the all-uppercase version of the prefix. Both sets of identifiers are also reserved as external names[1] throughout any program that uses the module.

In addition, all identifiers that begin with *libavl_* or `LIBAVL_` are reserved for any use in source files that #**include** any LIBAVL module. Likewise, these identifiers are reserved as external names in any program that uses any LIBAVL module. This is primarily to allow for future expansion, but see Section 2.5 [Memory Allocation], page 11 and Exercise 2.5-1 for a sample use.

The prefix used in code samples in this chapter is *tbl_*, short for "table". This can be considered a generic substitute for the prefix used by any of the table implementation. All of the statements about these functions here apply equally to all of the table implementation in later chapters, except that the *tbl_* prefix must be replaced by the prefix used by the chapter's table implementation.

Exercises:

1. The following kinds of identifiers are among those that might appear in a header file. Which of them can be safely appear unprefixed? Why?

a. Parameter names within function prototypes.
b. Macro parameter names.
c. Structure and union tags.
d. Structure and union member names.

2. Suppose that we create a module for reporting errors. Why is *err_* a poorly chosen prefix for the module's identifiers?

2.3 Comparison Function

The C language provides the **void** $*$ generic pointer for dealing with data of unknown type. We will use this type to allow our tables to contain a wide range of data types. This flexibility does keep the table from working directly with its data. Instead, the table's user must provide means to operate on data items. This section describes the user-provided functions for comparing items, and the next section describes two other kinds of user-provided functions.

[1] External names are identifiers visible outside a single source file. These are, mainly, non-**static** functions and variables declared outside a function.

There is more than one kind of generic algorithm for searching. We can search by comparison of keys, by digital properties of the keys, or by computing a function of the keys. In this book, we are only interested in the first possibility, so we need a way to compare data items. This is done with a user-provided function compatible with *tbl_comparison_func*, declared as follows:

§2 ⟨ Table function types 2 ⟩ ≡

```
/* Function types. */
typedef int tbl_comparison_func (const void *tbl_a, const void *tbl_b, void *tbl_param);
```

See also §4.
This code is included in §14.

A comparison function takes two pointers to data items, here called a and b, and compares their keys. It returns a negative value if $a < b$, zero if $a == b$, or a positive value if $a > b$. It takes a third parameter, here called *param*, which is user-provided.

A comparison function must work more or less like an arithmetic comparison within the domain of the data. This could be alphabetical ordering for strings, a set of nested sort orders (e.g., sort first by last name, with duplicates by first name), or any other comparison function that behaves in a "natural" way. A comparison function in the exact class of those acceptable is called a **strict weak ordering**, for which the exact rules are explained in Exercise 5.

Here's a function that can be used as a comparison function for the case that the **void** * pointers point to single **int**s:

§3 ⟨ Comparison function for **int**s 3 ⟩ ≡

```
/* Comparison function for pointers to ints. param is not used. */
int compare_ints (const void *pa, const void *pb, void *param) {
    const int *a = pa;
    const int *b = pb;
    if (*a < *b) return -1;
    else if (*a > *b) return +1;
    else return 0;
}
```

This code is included in §134.

Here's another comparison function for data items that point to ordinary C strings:

```
/* Comparison function for strings. param is not used. */
int compare_strings (const void *pa, const void *pb, void *param) {
    return strcmp (pa, pb);
}
```

See also: [FSF 1999], node "Defining the Comparison Function"; [ISO 1998], section 25.3, "Sorting and related operations"; [SGI 1993], section "Strict Weak Ordering".

Exercises:

1. In C, integers may be cast to pointers, including **void** *, and vice versa. Explain why it is not a good idea to use an integer cast to **void** * as a data item. When would such a technique would be acceptable?

2. When would the following be an acceptable alternate definition for *compare_ints*()?

```
int compare_ints (const void *pa, const void *pb, void *param) {
    return *((int *) pa) - *((int *) pb);
}
```

3. Could *strcmp*(), suitably cast, be used in place of *compare_strings*()?

4. Write a comparison function for data items that, in any particular table, are character arrays of fixed length. Among different tables, the length may differ, so the third parameter to the function points to a **size_t** specifying the length for a given table.

***5.** For a comparison function $f()$ to be a strict weak ordering, the following must hold for all possible data items a, b, and c:

- *Irreflexivity:* For every a, $f(a,\ a) == 0$.
- *Antisymmetry*: If $f(a,\ b) > 0$, then $f(b,\ a) < 0$.
- *Transitivity*: If $f(a,\ b) > 0$ and $f(b,\ c) > 0$, then $f(a,\ c) > 0$.
- *Transitivity of equivalence*: If $f(a,\ b) == 0$ and $f(b,\ c) == 0$, then $f(a,\ c) == 0$.

Consider the following questions that explore the definition of a strict weak ordering.

a. Explain how *compare_ints*() above satisfies each point of the definition.

b. Can the standard C library function *strcmp*() be used for a strict weak ordering?

c. Propose an irreflexive, antisymmetric, transitive function that lacks transitivity of equivalence.

***6.** Libavl uses a ternary comparison function that returns a negative value for $<$, zero for $\equiv$, positive for $>$. Other libraries use binary comparison functions that return nonzero for $<$ or zero for $\geq$. Consider these questions about the differences:

a. Write a C expression, in terms of a binary comparison function $f()$ and two items a and b, that is nonzero if and only if $a == b$ as defined by $f()$. Write a similar expression for $a > b$.

b. Write a binary comparison function "wrapper" for a Libavl comparison function.

c. Rewrite *bst_find*() based on a binary comparison function. (You can use the wrapper from above to simulate a binary comparison function.)

2.4 Item and Copy Functions

Besides **tbl_comparison_func**, there are two kinds of functions used in Libavl to manipulate item data:

§4 ⟨ Table function types 2 ⟩ +≡

```
typedef void tbl_item_func (void *tbl_item, void *tbl_param);
typedef void *tbl_copy_func (void *tbl_item, void *tbl_param);
```

Both of these function types receive a table item as their first argument *tbl_item* and the *tbl_param* associated with the table as their second argument. This *tbl_param* is the same one passed as the third argument to **tbl_comparison_func**. Libavl will never pass a null pointer as *tbl_item* to either kind of function.

A **tbl_item_func** performs some kind of action on *tbl_item*. The particular action that it should perform depends on the context in which it is used and the needs of the calling program.

A **tbl_copy_func** creates and returns a new copy of *tbl_item*. If copying fails, then it returns a null pointer.

2.5 Memory Allocation

The standard C library functions *malloc*() and *free*() are the usual way to obtain and release memory for dynamic data structures like tables. Most users will be satisfied if LIBAVL uses these routines for memory management. On the other hand, some users will want to supply their own methods for allocating and freeing memory, perhaps even different methods from table to table. For these users' benefit, each table is associated with a memory allocator, which provides functions for memory allocation and deallocation. This allocator has the same form in each table implementation. It looks like this:

§5 ⟨ Memory allocator 5 ⟩ ≡

```
#ifndef LIBAVL_ALLOCATOR
#define LIBAVL_ALLOCATOR
/* Memory allocator. */
struct libavl_allocator {
    void *(*libavl_malloc) (struct libavl_allocator *, size_t libavl_size);
    void (*libavl_free) (struct libavl_allocator *, void *libavl_block);
};
#endif
```

This code is included in §14, §99, and §649.

Members of **struct libavl_allocator** have the same interfaces as the like-named standard C library functions, except that they are each additionally passed a pointer to the **struct libavl_allocator** * itself as their first argument. The table implementations never call *tbl_malloc*() with a zero size or *tbl_free*() with a null pointer block.

The **struct libavl_allocator** type is shared between all of LIBAVL's modules, so its name begins with *libavl_*, not with the specific module prefix that we've been representing generically here as *tbl_*. This makes it possible for a program to use a single allocator with multiple LIBAVL table modules, without the need to declare instances of different structures.

The default allocator is just a wrapper around *malloc*() and *free*(). Here it is:

§6 ⟨ Default memory allocation functions 6 ⟩ ≡

```
/* Allocates size bytes of space using malloc(). Returns a null pointer if allocation fails. */
void *tbl_malloc (struct libavl_allocator *allocator, size_t size) {
    assert (allocator != NULL && size > 0);
    return malloc (size);
}
/* Frees block. */
void tbl_free (struct libavl_allocator *allocator, void *block) {
    assert (allocator != NULL && block != NULL);
    free (block);
}
/* Default memory allocator that uses malloc() and free(). */
struct libavl_allocator tbl_allocator_default = {tbl_malloc, tbl_free};
```

This code is included in §29, §145, §196, §251, §300, §336, §375, §418, §455, §489, §522, §554, and §649.

The default allocator comes along with header file declarations:

§7 ⟨ Default memory allocator header 7 ⟩ ≡

```
/* Default memory allocator. */
extern struct libavl_allocator tbl_allocator_default;
void *tbl_malloc (struct libavl_allocator *, size_t);
void tbl_free (struct libavl_allocator *, void *);
```

This code is included in §14 and §649.

See also: [FSF 1999], nodes "Malloc Examples" and "Changing Block Size".

Exercises:

1. This structure is named with a *libavl_* prefix because it is shared among all of LIBAVL's module. Other types are shared among LIBAVL modules, too, such as **tbl_item_func**. Why don't the names of these other types also begin with *libavl_*?

2. Supply an alternate allocator, still using *malloc*() and *free*(), that prints an error message to *stderr* and aborts program execution when memory allocation fails.

***3.** Some kinds of allocators may need additional arguments. For instance, if memory for each table is taken from a separate Apache-style "memory pool", then a pointer to the pool structure is needed. Show how this can be done without modifying existing types.

2.6 Creation and Destruction

This section describes the functions that create and destroy tables.

§8 ⟨ Table creation function prototypes 8 ⟩ ≡

```
/* Table functions. */
struct tbl_table *tbl_create (tbl_comparison_func *, void *, struct libavl_allocator *);
struct tbl_table *tbl_copy (const struct tbl_table *, tbl_copy_func *,
                            tbl_item_func *, struct libavl_allocator *);
void tbl_destroy (struct tbl_table *, tbl_item_func *);
```

This code is included in §15.

- *tbl_create*(): Creates and returns a new, empty table as a **struct tbl_table** *. The table is associated with the given arguments. The **void** * argument is passed as the third argument to the comparison function when it is called. If the allocator is a null pointer, then *tbl_allocator_default* is used.
- *tbl_destroy*(): Destroys a table. During destruction, the **tbl_item_func** provided, if non-null, is called once for every item in the table, in no particular order. The function, if provided, must not invoke any table function or macro on the table being destroyed.
- *tbl_copy*(): Creates and returns a new table with the same contents as the existing table passed as its first argument. Its other three arguments may all be null pointers.

  If a **tbl_copy_func** is provided, then it is used to make a copy of each table item as it is inserted into the new table, in no particular order (a **deep copy**). Otherwise, the **void** * table items are copied verbatim (a **shallow copy**).

  If the table copy fails, either due to memory allocation failure or a null pointer returned by the **tbl_copy_func**, *tbl_copy*() returns a null pointer. In this case, any provided **tbl_item_func** is called once for each new item already copied, in no particular order.

By default, the new table uses the same memory allocator as the existing one. If non-null, the **struct libavl_allocator** * given is used instead as the new memory allocator. To use the *tbl_allocator_default* allocator, specify &*tbl_allocator_default* explicitly.

2.7 Count

This function returns the number of items currently in a table.

§9 ⟨ Table count function prototype 9 ⟩ ≡
size_t *tbl_count* (**const struct tbl_table** *);

The actual tables instead use a macro for implementation.

Exercises:

1. Implement *tbl_count*() as a macro, on the assumption that **struct tbl_table** keeps the number of items in the table in a **size_t** member named *tbl_count*.

2.8 Insertion and Deletion

These functions insert and delete items in tables. There is also a function for searching a table without modifying it.

The design behind the insertion functions takes into account a couple of important issues:

- What should happen if there is a matching item already in the tree? If the items contain only keys and no values, then there's no point in doing anything. If the items do contain values, then we might want to leave the existing item or replace it, depending on the particular circumstances. The *tbl_insert*() and *tbl_replace*() functions are handy in simple cases like these.
- Occasionally it is convenient to insert one item into a table, then immediately replace it by a different item that has identical key data. For instance, if there is a good chance that a data item already exists within a table, then it might make sense to insert data allocated as a local variable into a table, then replace it by a dynamically allocated copy if it turned out that the item wasn't already in the table. That way, we save the time required to make an additional copy of the item to insert. The *tbl_probe*() function allows for this kind of flexibility.

§10 ⟨ Table insertion and deletion function prototypes 10 ⟩ ≡
void ***tbl_probe* (**struct tbl_table** *, **void** *);
void **tbl_insert* (**struct tbl_table** *, **void** *);
void **tbl_replace* (**struct tbl_table** *, **void** *);
void **tbl_delete* (**struct tbl_table** *, **const void** *);
void **tbl_find* (**const struct tbl_table** *, **const void** *);

This code is included in §15.

Each of these functions takes a table to manipulate as its first argument and a table item as its second argument, here called *table* and *item*, respectively. Both arguments must be non-null in all cases. All but *tbl_probe*() return a table item or a null pointer.

- *tbl_probe*(): Searches in *table* for an item matching *item*. If found, a pointer to the **void** * data item is returned. Otherwise, *item* is inserted into the table and a pointer to the copy within the table is returned. Memory allocation failure causes a null pointer to be returned.

The pointer returned can be used to replace the item found or inserted by a different item. This must only be done if the replacement item has the same position relative to the other items in the table as did the original item. That is, for existing item e, replacement item r, and the table's comparison function $f()$, the return values of $f(e, x)$ and $f(r, x)$ must have the same sign for every other item x currently in the table. Calling any other table function invalidates the pointer returned and it must not be referenced subsequently.

- *tbl_insert*(): Inserts *item* into *table*, but not if a matching item exists. Returns a null pointer if successful or if a memory allocation error occurs. If a matching item already exists in the table, returns that item.
- *tbl_replace*(): Inserts *item* into *table*, replacing and returning any matching item. Returns a null pointer if the item was inserted but there was no matching item to replace, or if a memory allocation error occurs.
- *tbl_delete*(): Removes from *table* and returns an item matching *item*. Returns a null pointer if no matching item exists in the table.
- *tbl_find*(): Searches *table* for an item matching *item* and returns any item found. Returns a null pointer if no matching item exists in the table.

Exercises:

1. Functions *tbl_insert*() and *tbl_replace*() return `NULL` in two very different situations: an error or successful insertion. Why is this not necessarily a design mistake?

2. Suggest a reason for disallowing insertion of a null item.

3. Write generic implementations of *tbl_insert*() and *tbl_replace*() in terms of *tbl_probe*().

2.9 Assertions

Sometimes an insertion or deletion must succeed because it is known in advance that there is no way that it can fail. For instance, we might be inserting into a table from a list of items known to be unique, using a memory allocator that cannot return a null pointer. In this case, we want to make sure that the operation succeeded, and abort if not, because that indicates a program bug. We also would like to be able to turn off these tests for success in our production versions, because we don't want them slowing down the code.

§11 ⟨ Table assertion function prototypes 11 ⟩ ≡
void *tbl_assert_insert* (**struct tbl_table** ∗, **void** ∗);
void ∗*tbl_assert_delete* (**struct tbl_table** ∗, **void** ∗);

This code is included in §15.

These functions provide assertions for *tbl_insert*() and *tbl_delete*(). They expand, via macros, directly into calls to those functions when `NDEBUG`, the same symbol used to turn off *assert*() checks, is declared. As for the standard C header ⟨ assert.h ⟩, header files for tables may be included multiple times in order to turn these assertions on or off.

Exercises:

1. Write a set of preprocessor directives for a table header file that implement the behavior described in the final paragraph above.

2. Write a generic implementation of *tbl_assert_insert*() and *tbl_assert_delete*() in terms of existing table functions. Consider the base functions carefully. Why must we make sure that assertions are always enabled for these functions?

3. Why must *tbl_assert_insert*() not be used if the table's memory allocator can fail? (See also Exercise 2.8-1.)

2.10 Traversers

A **struct tbl_traverser** is a table "traverser" that allows the items in a table to be examined. With a traverser, the items within a table can be enumerated in sorted ascending or descending order, starting from either end or from somewhere in the middle.

The user of the traverser declares its own instance of **struct tbl_traverser**, typically as a local variable. One of the traverser constructor functions described below can be used to initialize it. Until then, the traverser is invalid. An invalid traverser must not be passed to any traverser function other than a constructor.

Seen from the viewpoint of a table user, a traverser has only one attribute: the current item. The current item is either an item in the table or the "null item", represented by a null pointer and not associated with any item.

Traversers continue to work when their tables are modified. Any number of insertions and deletions may occur in the table without affecting the current item selected by a traverser, with only a few exceptions:

- Deleting a traverser's current item from its table invalidates the traverser (even if the item is later re-inserted).
- Using the return value of *tbl_probe*() to replace an item in the table invalidates all traversers with that item current, unless the replacement item has the same key data as the original item (that is, the table's comparison function returns 0 when the two items are compared).
- Similarly, *tbl_t_replace*() invalidates all *other* traversers with the same item selected, unless the replacement item has the same key data.
- Destroying a table with *tbl_destroy*() invalidates all of that table's traversers.

There is no need to destroy a traverser that is no longer needed. An unneeded traverser can simply be abandoned.

2.10.1 Constructors

These functions initialize traversers. A traverser must be initialized with one of these functions before it is passed to any other traverser function.

§12 ⟨ Traverser constructor function prototypes 12 ⟩ ≡
/∗ Table traverser functions. ∗/
void *tbl_t_init* (**struct tbl_traverser** ∗, **struct tbl_table** ∗);
void ∗*tbl_t_first* (**struct tbl_traverser** ∗, **struct tbl_table** ∗);
void ∗*tbl_t_last* (**struct tbl_traverser** ∗, **struct tbl_table** ∗);

void ∗*tbl_t_find* (**struct tbl_traverser** ∗, **struct tbl_table** ∗, **void** ∗);
void ∗*tbl_t_insert* (**struct tbl_traverser** ∗, **struct tbl_table** ∗, **void** ∗);
void ∗*tbl_t_copy* (**struct tbl_traverser** ∗, **const struct tbl_traverser** ∗);

This code is included in §15.

All of these functions take a traverser to initialize as their first argument, and most take a table to associate the traverser with as their second argument. These arguments are here called *trav* and *table*. All, except *tbl_t_init*(), return the item to which *trav* is initialized, using a null pointer to represent the null item. None of the arguments to these functions may ever be a null pointer.

- *tbl_t_init*(): Initializes *trav* to the null item in *table*.
- *tbl_t_first*(): Initializes *trav* to the least-valued item in *table*. If the table is empty, then *trav* is initialized to the null item.
- *tbl_t_last*(): Same as *tbl_t_first*(), for the greatest-valued item in *table*.
- *tbl_t_find*(): Searches *table* for an item matching the one given. If one is found, initializes *trav* with it. If none is found, initializes *trav* to the null item.
- *tbl_t_insert*(): Attempts to insert the given item into *table*. If it is inserted succesfully, *trav* is initialized to its location. If it cannot be inserted because of a duplicate, the duplicate item is set as *trav*'s current item. If there is a memory allocation error, *trav* is initialized to the null item.
- *tbl_t_copy*(): Initializes *trav* to the same table and item as a second valid traverser. Both arguments pointing to the same valid traverser is valid and causes no change in either.

2.10.2 Manipulators

These functions manipulate valid traversers.

§13 ⟨ Traverser manipulator function prototypes 13 ⟩ ≡
void ∗*tbl_t_next* (**struct tbl_traverser** ∗);
void ∗*tbl_t_prev* (**struct tbl_traverser** ∗);
void ∗*tbl_t_cur* (**struct tbl_traverser** ∗);
void ∗*tbl_t_replace* (**struct tbl_traverser** ∗, **void** ∗);

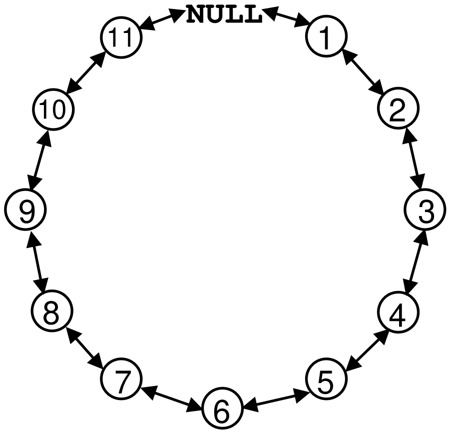
This code is included in §15.

Each of these functions takes a valid traverser, here called *trav*, as its first argument, and returns a data item. All but *tbl_t_replace*() can also return a null pointer that represents the null item. All arguments to these functions must be non-null pointers.

- *tbl_t_next*(): Advances *trav* to the next larger item in its table. If *trav* was at the null item in a nonempty table, then the smallest item in the table becomes current. If *trav* was already at the greatest item in its table or the table is empty, the null item becomes current. Returns the new current item.
- *tbl_t_prev*(): Advances *trav* to the next smaller item in its table. If *trav* was at the null item in a nonempty table, then the greatest item in the table becomes current. If *trav* was already at the lowest item in the table or the table is empty, the null item becomes current. Returns the new current item.
- *tbl_t_cur*(): Returns *trav*'s current item.

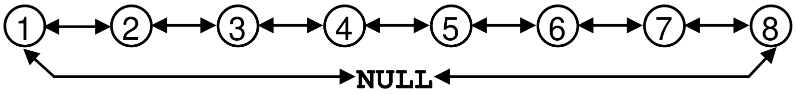
- *tbl_t_replace*(): Replaces the data item currently selected in *trav* by the one provided. The replacement item is subject to the same restrictions as for the same replacement using *tbl_probe*(). The item replaced is returned. If the null item is current, the behavior is undefined.

Seen from the outside, the traverser treats the table as a circular arrangement of items:



Moving clockwise in the circle is equivalent, under our traverser, to moving to the next item with *tbl_t_next*(). Moving counterclockwise is equivalent to moving to the previous item with *tbl_t_prev*().

An equivalent view is that the traverser treats the table as a linear arrangement of nodes:



From this perspective, nodes are arranged from least to greatest in left to right order, and the null node lies in the middle as a connection between the least and greatest nodes. Moving to the next node is the same as moving to the right and moving to the previous node is motion to the left, except where the null node is concerned.

2.11 Table Headers

Here we gather together in one place all of the types and prototypes for a generic table.

§14 ⟨ Table types 14 ⟩ ≡
⟨ Table function types 2 ⟩
⟨ Memory allocator 5 ⟩
⟨ Default memory allocator header 7 ⟩

This code is included in §24, §142, §192, §247, §297, §333, §372, §415, §452, §486, §519, and §551.

§15 ⟨ Table function prototypes 15 ⟩ ≡
⟨ Table creation function prototypes 8 ⟩
⟨ Table insertion and deletion function prototypes 10 ⟩
⟨ Table assertion function prototypes 11 ⟩
⟨ Table count macro 591 ⟩
⟨ Traverser constructor function prototypes 12 ⟩
⟨ Traverser manipulator function prototypes 13 ⟩

This code is included in §24, §142, §192, §247, §297, §333, §372, §415, §452, §486, §519, and §551.

All of our tables fit the specification given in Exercise 2.7-1, so ⟨ Table count macro 591 ⟩ is directly included above.

2.12 Additional Exercises

Exercises:

***1.** Compare and contrast the design of LIBAVL's tables with that of the *set* container in the C++ Standard Template Library.

2. What is the smallest set of table routines such that all of the other routines can be implemented in terms of the interfaces of that set as defined above?

3 Search Algorithms

In LIBAVL, we are primarily concerned with binary search trees and balanced binary trees. If you're already familiar with these concepts, then you can move right into the code, starting from the next chapter. But if you're not, then a little motivation and an explanation of exactly what a binary search tree is can't hurt. That's the goal of this chapter.

More particularly, this chapter concerns itself with algorithms for searching. Searching is one of the core problems in organizing a table. As it will turn out, arranging a table for fast searching also facilitates some other table features.

3.1 Sequential Search

Suppose that you have a bunch of things (books, magazines, CDs, ...) in a pile, and you're looking for one of them. You'd probably start by looking at the item at the top of the pile to check whether it was the one you were looking for. If it wasn't, you'd check the next item down the pile, and so on, until you either found the one you wanted or ran out of items.

In computer science terminology, this is a **sequential search**. It is easy to implement sequential search for an array or a linked list. If, for the moment, we limit ourselves to items of type **int**, we can write a function to sequentially search an array like this:

§16 ⟨ Sequentially search an array of **int**s 16 ⟩ ≡
```
/* Returns the smallest i such that array[i] == key, or -1 if key is not in array[].
   array[] must be an array of n ints. */
int seq_search (int array[], int n, int key) {
    int i;
    for (i = 0; i < n; i++)
        if (array[i] == key)
            return i;
    return -1;
}
```
This code is included in §595 and §600.

We can hardly hope to improve on the data requirements, space, or complexity of simple sequential search, as they're about as good as we can want. But the speed of sequential search leaves something to be desired. The next section describes a simple modification of the sequential search algorithm that can sometimes lead to big improvements in performance.

See also: [Knuth 1998b], algorithm 6.1S; [Kernighan 1976], section 8.2; [Cormen 1990], section 11.2; [Bentley 2000], sections 9.2 and 13.2, appendix 1.

Exercises:

1. Write a simple test framework for *seq_search*(). It should read sample data from *stdin* and collect them into an array, then search for each item in the array in turn and compare the results to those expected, reporting any discrepancies on *stdout* and exiting with an appropriate return value. You need not allow for the possibility of duplicate input values and may limit the maximum number of input values.

3.2 Sequential Search with Sentinel

Try to think of some ways to improve the speed of sequential search. It should be clear that, to speed up a program, it pays to concentrate on the parts that use the most time to begin with. In this case, it's the loop.

Consider what happens each time through the loop:

1. The loop counter *i* is incremented and compared against *n*.
2. *array*[*i*] is compared against *key*.

If we could somehow eliminate one of these comparisons, the loop might be a lot faster. So, let's try… why do we need step 1? It's because, otherwise, we might run off the end of *array*[], causing undefined behavior, which is in turn because we aren't sure that *key* is in *array*[]. If we knew that *key* was in *array*[], then we could skip step 1.

But, hey! we *can* ensure that the item we're looking for is in the array. How? By putting a copy of it at the end of the array. This copy is called a **sentinel**, and the search technique as a whole is called **sequential search with sentinel**. Here's the code:

§17 ⟨ Sequentially search an array of **int**s using a sentinel 17 ⟩ ≡

```
/* Returns the smallest i such that array[i] == key, or -1 if key is not in array[].
   array[] must be an modifiable array of n ints with room for a (n + 1)th element. */
int seq_sentinel_search (int array[], int n, int key) {
  int *p;
  array[n] = key;
  for (p = array; *p != key; p++)
    /* Nothing to do. */;
  return p - array < n ? p - array : -1;
}
```

This code is included in §600.

Notice how the code above uses a pointer, **int** $*p$, rather than a counter *i* as in ⟨ Sequentially search an array of **int**s 16 ⟩ earlier. For the most part, this is simply a style preference: for iterating through an array, C programmers usually prefer pointers to array indexes. Under older compilers, code using pointers often compiled into faster code as well, but modern C compilers usually produce the same code whether pointers or indexes are used.

The **return** statement in this function uses two somewhat advanced features of C: the conditional or "ternary" operator ?: and pointer arithmetic. The former is a bit like an expression form of an **if** statement. The expression a ? b : c first evaluates a. Then, if a != 0, b is evaluated and the expression takes that value. Otherwise, a == 0, c is evaluated, and the result is the expression's value.

Pointer arithmetic is used in two ways here. First, the expression p++ acts to advance p to point to the next **int** in *array*. This is analogous to the way that i++ would increase the value of an integer or floating point variable i by one. Second, the expression $p - array$ results in the "difference" between p and *array*, i.e., the number of **int** elements between the locations to which they point. For more information on these topics, please consult a good C reference, such as [Kernighan 1988].

Searching with a sentinel requires that the array be modifiable and large enough to hold an extra element. Sometimes these are inherently problematic—the array may not be modifiable or it might be too small—and sometimes they are problems because of external circumstances. For instance, a program with more than one concurrent **thread** cannot modify a shared array for sentinel search without expensive locking.

Sequential sentinel search is an improvement on ordinary sequential search, but as it turns out there's still room for improvement—especially in the runtime for unsuccessful searches, which still always take n comparisons. In the next section, we'll see one technique that can reduce the time required for unsuccessful searches, at the cost of longer runtime for successful searches.

See also: [Knuth 1998b], algorithm 6.1Q; [Cormen 1990], section 11.2; [Bentley 2000], section 9.2.

3.3 Sequential Search of Ordered Array

Let's jump back to the pile-of-things analogy from the beginning of this chapter (see Section 3.1 [Sequential Search], page 19). This time, suppose that instead of being in random order, the pile you're searching through is ordered on the property that you're examining; e.g., magazines sorted by publication date, if you're looking for, say, the July 1988 issue.

Think about how this would simplify searching through the pile. Now you can sometimes tell that the magazine you're looking for isn't in the pile before you get to the bottom, because it's not between the magazines that it otherwise would be. On the other hand, you still might have to go through the entire pile if the magazine you're looking for is newer than the newest magazine in the pile (or older than the oldest, depending on the ordering that you chose).

Back in the world of computers, we can apply the same idea to searching a sorted array:

§18 ⟨ Sequentially search a sorted array of **int**s 18 ⟩ ≡

```
/* Returns the smallest i such that array[i] == key, or −1 if key is not in array[].
   array[] must be an array of n ints sorted in ascending order. */
int seq_sorted_search (int array[], int n, int key) {
    int i;

    for (i = 0; i < n; i++)
        if (key <= array[i])
            return key == array[i] ? i : −1;

    return −1;
}
```

This code is included in §600.

At first it might be a little tricky to see exactly how *seq_sorted_search*() works, so we'll work through a few examples. Suppose that *array*[] has the four elements $\{3, 5, 6, 8\}$, so that n is 4. If *key* is 6, then the first time through the loop the **if** condition is $6 <= 3$, or false, so the loop repeats with $i == 1$. The second time through the loop we again have a false condition, $6 <= 5$, and the loop repeats again. The third time the **if** condition, $6 <= 6$, is true, so control passes to the **if** statement's dependent **return**. This **return** verifies that $6 == 6$ and returns i, or 2, as the function's value.

On the other hand, suppose *key* is 4, a value not in *array*[]. For the first iteration, when i is 0, the **if** condition, 4 <= 3, is false, but in the second iteration we have 4 <= 5, which is true. However, this time *key* == *array*[i] is 4 == 5, or false, so $^-1$ is returned.

See also: [Sedgewick 1998], program 12.4.

3.4 Sequential Search of Ordered Array with Sentinel

When we implemented sequential search in a sorted array, we lost the benefits of having a sentinel. But we can reintroduce a sentinel in the same way we did before, and obtain some of the same benefits. It's pretty clear how to proceed:

§19 ⟨ Sequentially search a sorted array of **int**s using a sentinel 19 ⟩ ≡

```
/* Returns the smallest i such that array[i] == key, or -1 if key is not in array[].
   array[] must be an modifiable array of n ints, sorted in ascending order,
   with room for a (n + 1)th element at the end. */
int seq_sorted_sentinel_search (int array[], int n, int key) {
    int *p;

    array[n] = key;
    for (p = array; *p < key; p++)
        /* Nothing to do. */;
    return p - array < n && *p == key ? p - array : -1;
}
```

This code is included in §600.

With a bit of additional cleverness we can eliminate one objection to this sentinel approach. Suppose that instead of using the value being searched for as the sentinel value, we used the maximum possible value for the type in question. If we did this, then we could use almost the same code for searching the array.

The advantage of this approach is that there would be no need to modify the array in order to search for different values, because the sentinel is the same value for all searches. This eliminates the potential problem of searching an array in multiple contexts, due to nested searches, threads, or signals, for instance. (In the code below, we will still put the sentinel into the array, because our generic test program won't know to put it in for us in advance, but in real-world code we could avoid the assignment.)

We can easily write code for implementation of this technique:

§20 ⟨ Sequentially search a sorted array of **int**s using a sentinel (2) 20 ⟩ ≡

```
/* Returns the smallest i such that array[i] == key, or -1 if key is not in array[].
   array[] must be an array of n ints, sorted in ascending order,
   with room for an (n + 1)th element to set to INT_MAX. */
int seq_sorted_sentinel_search_2 (int array[], int n, int key) {
    int *p;

    array[n] = INT_MAX;
    for (p = array; *p < key; p++)
        /* Nothing to do. */;
    return p - array < n && *p == key ? p - array : -1;
}
```

This code is included in §600.

Exercises:

1. When can't the largest possible value for the type be used as a sentinel?

3.5 Binary Search of Ordered Array

At this point we've squeezed just about all the performance we can out of sequential search in portable C. For an algorithm that searches faster than our final refinement of sequential search, we'll have to reconsider our entire approach.

What's the fundamental idea behind sequential search? It's that we examine array elements in order. That's a fundamental limitation: if we're looking for an element in the middle of the array, we have to examine every element that comes before it. If a search algorithm is going to be faster than sequential search, it will have to look at fewer elements.

One way to look at search algorithms based on repeated comparisons is to consider what we learn about the array's content at each step. Suppose that *array*[] has n elements in sorted order, without duplicates, that *array*[j] contains *key*, and that we are trying to learn the value j. In sequential search, we learn only a little about the data set from each comparison with *array*[i]: either *key* == *array*[i] so that $i == j$, or *key* != *array*[i] so that i != j and therefore $j > i$. As a result, we eliminate only one possibility at each step.

Suppose that we haven't made any comparisons yet, so that we know nothing about the contents of *array*[]. If we compare *key* to *array*[i] for arbitrary i such that $0 \le i < n$, what do we learn? There are three possibilities:

- *key* < *array*[i]: Now we know that *key* < *array*[i] < *array*[i + 1] < $\cdots$ < *array*[n − 1].[1] Therefore, $0 \le j < i$.
- *key* == *array*[i]: We're done: $j == i$.
- *key* > *array*[i]: Now we know that *key* > *array*[i] > *array*[i − 1] > $\cdots$ > *array*[0]. Therefore, $i < j < n$.

So, after one step, if we're not done, we know that $j > i$ or that $j < i$. If we're equally likely to be looking for each element in *array*[], then the best choice of i is n / 2: for that value, we eliminate about half of the possibilities either way. (If n is odd, we'll round down.)

After the first step, we're back to essentially the same situation: we know that *key* is in *array*[j] for some j in a range of about n / 2. So we can repeat the same process. Eventually, we will either find *key* and thus j, or we will eliminate all the possibilities.

Let's try an example. For simplicity, let *array*[] contain the values 100 through 114 in numerical order, so that *array*[i] is 100 + i and n is 15. Suppose further that *key* is 110. The steps that we'd go through to find j are described below. At each step, the facts are listed: the known range that j can take, the selected value of i, the results of comparing *key* to *array*[i], and what was learned from the comparison.

1. $0 \le j \le 14$: i becomes $(0 + 14) / 2 \equiv 7$. $110 >$ *array*[i] $\equiv 107$, so now we know that $j > 7$.

[1] This sort of notation means very different things in C and mathematics. In mathematics, writing $a < b < c$ asserts both of the relations $a < b$ and $b < c$, whereas in C, it expresses the evaluation of $a < b$, then the comparison of the 0 or 1 result to the value of c. In mathematics this notation is invaluable, but in C it is rarely meaningful. As a result, this book uses this notation only in the mathematical sense.

2. $8 \le j \le 14$: i becomes $(8 + 14) / 2 \equiv 11$. $110 < array[i] \equiv 111$, so now we know that $j < 11$.
3. $8 \le j \le 10$: i becomes $(8 + 10) / 2 \equiv 9$. $110 > array[i] \equiv 109$, so now we know that $j > 9$.
4. $10 \le j \le 10$: i becomes $(10 + 10) / 2 \equiv 10$. $110 \equiv array[i] \equiv 110$, so we're done and $i \equiv j \equiv 10$.

In case you hadn't yet figured it out, this technique is called **binary search**. We can make an initial C implementation pretty easily:

§21 ⟨ Binary search of ordered array 21 ⟩ ≡

```
/* Returns the offset within array[] of an element equal to key, or −1 if key is not in array[].
   array[] must be an array of n ints sorted in ascending order. */
int binary_search (int array[], int n, int key) {
  int min = 0;
  int max = n − 1;

  while (max >= min) {
    int i = (min + max) / 2;
    if (key < array[i]) max = i − 1;
    else if (key > array[i]) min = i + 1;
    else return i;
  }

  return −1;
}
```

This code is included in §600.

The maximum number of comparisons for a binary search in an array of n elements is about $\log_2 n$, as opposed to a maximum of n comparisons for sequential search. For moderate to large values of n, this is a lot better.

On the other hand, for small values of n, binary search may actually be slower because it is more complicated than sequential search. We also have to put our array in sorted order before we can use binary search. Efficiently sorting an n-element array takes time proportional to $n \log_2 n$ for large n. So binary search is preferred if n is large enough (see the answer to Exercise 4 for one typical value) and if we are going to do enough searches to justify the cost of the initial sort.

Further small refinements are possible on binary search of an ordered array. Try some of the exercises below for more information.

See also: [Knuth 1998b], algorithm 6.2.1B; [Kernighan 1988], section 3.3; [Bentley 2000], chapters 4 and 5, section 9.3, appendix 1; [Sedgewick 1998], program 12.6.

Exercises:

1. Function *binary_search*() above uses three local variables: *min* and *max* for the ends of the remaining search range and *i* for its midpoint. Write and test a binary search function that uses only two variables: *i* for the midpoint as before and *m* representing the width of the range on either side of *i*. You may require the existence of a dummy element just before the beginning of the array. Be sure, if so, to specify what its value should be.

2. The standard C library provides a function, *bsearch*(), for searching ordered arrays. Commonly, *bsearch*() is implemented as a binary search, though ANSI C does not require it. Do the following:

a. Write a function compatible with the interface for *binary_search*() that uses *bsearch*() "under the hood." You'll also have to write an additional callback function for use by *bsearch*().

b. Write and test your own version of *bsearch*(), implementing it using a binary search. (Use a different name to avoid conflicts with the C library.)

3. An earlier exercise presented a simple test framework for *seq_search*(), but now we have more search functions. Write a test framework that will handle all of them presented so far. Add code for timing successful and unsuccessful searches. Let the user specify, on the command line, the algorithm to use, the size of the array to search, and the number of search iterations to run.

4. Run the test framework from the previous exercise on your own system for each algorithm. Try different array sizes and compiler optimization levels. Be sure to use enough iterations to make the searches take at least a few seconds each. Analyze the results: do they make sense? Try to explain any apparent discrepancies.

3.6 Binary Search Tree in Array

Binary search is pretty fast. Suppose that we wish to speed it up anyhow. Then, the obvious speed-up targets in ⟨ Binary search of ordered array 21 ⟩ above are the **while** condition and the calculations determining values of i, *min*, and *max*. If we could eliminate these, we'd have an incrementally faster technique, all else being equal. And, as it turns out, we *can* eliminate both of them, the former by use of a sentinel and the latter by precalculation.

Let's consider precalculating i, *min*, and *max* first. Think about the nature of the choices that binary search makes at each step. Specifically, in ⟨ Binary search of ordered array 21 ⟩ above, consider the dependence of *min* and *max* upon i. Is it ever possible for *min* and *max* to have different values for the same i and n?

The answer is no. For any given i and n, *min* and *max* are fixed. This is important because it means that we can represent the entire "state" of a binary search of an n-element array by the single variable i. In other words, if we know i and n, we know all the choices that have been made to this point and we know the two possible choices of i for the next step.

This is the key insight in eliminating calculations. We can use an array in which the items are labeled with the next two possible choices.

An example is indicated. Let's continue with our example of an array containing the 16 integers 100 to 115. We define an entry in the array to contain the item value and the array index of the item to examine next for search values smaller and larger than the item:

§22 ⟨ Binary search tree entry 22 ⟩ ≡

```
/* One entry in a binary search tree stored in an array. */
struct binary_tree_entry {
    int value; /* This item in the binary search tree. */
```

```
    int smaller; /* Array index of next item for smaller targets. */
    int larger; /* Array index of next item for larger targets. */
};
```

This code is included in §617.

Of course, it's necessary to fill in the values for *smaller* and *larger*. A few moments' reflection should allow you to figure out one method for doing so. Here's the full array, for reference:

```
const struct binary_tree_entry bins[16] = {
    {100, 15, 15}, {101, 0, 2}, {102, 15, 15}, {103, 1, 5}, {104, 15, 15},
    {105, 4, 6}, {106, 15, 15}, {107, 3, 11}, {108, 15, 15}, {109, 8, 10},
    {110, 15, 15}, {111, 9, 13}, {112, 15, 15}, {113, 12, 14}, {114, 15, 15},
    {0, 0, 0},
};
```

For now, consider only *bins*[]'s first 15 rows. Within these rows, the first column is *value*, the item value, and the second and third columns are *smaller* and *larger*, respectively. Values 0 through 14 for *smaller* and *larger* indicate the index of the next element of *bins*[] to examine. Value 15 indicates "element not found". Element *array*[15] is not used for storing data.

Try searching for *key* == 110 in *bins*[], starting from element 7, the midpoint:

1. $i == 7$: $110 > bins[i].value == 107$, so let $i = bins[i].larger$, or 11.
2. $i == 11$: $110 < bins[i].value == 111$, so let $i = bins[i].smaller$, or 10.
3. $i == 10$: $110 == bins[i].value == 110$, so we're done.

We can implement this search in C code. The function uses the common C idiom of writing **for** (;;) for an "infinite" loop:

§23 ⟨Search of binary search tree stored as array 23⟩ ≡

```
/* Returns i such that array[i].value == key, or -1 if key is not in array[].
   array[] is an array of n elements forming a binary search tree,
   with its root at array[n / 2], and space for an (n + 1)th value at the end. */
int binary_search_tree_array (struct binary_tree_entry array[], int n, int key) {
    int i = n / 2;

    array[n].value = key;
    for (;;)
        if (key > array[i].value) i = array[i].larger;
        else if (key < array[i].value) i = array[i].smaller;
        else return i != n ? i : -1;
}
```

This code is included in §617.

Examination of the code above should reveal the purpose of *bins*[15]. It is used as a sentinel value, allowing the search to always terminate without the use of an extra test on each loop iteration.

The result of augmenting binary search with "pointer" values like *smaller* and *larger* is called a **binary search tree**.

Exercises:

1. Write a function to automatically initialize *smaller* and *larger* within *bins*[].

2. Write a simple automatic test program for *binary_search_tree_array*(). Let the user specify the size of the array to test on the command line. You may want to use your results from the previous exercise.

3.7 Dynamic Lists

Up until now, we've considered only lists whose contents are fixed and unchanging, that is, **static** lists. But in real programs, many lists are **dynamic**, with their contents changing rapidly and unpredictably. For the case of dynamic lists, we need to reconsider some of the attributes of the types of lists that we've examined.[2]

Specifically, we want to know how long it takes to insert a new element into a list and to remove an existing element from a list. Think about it for each type of list examined so far:

Unordered array

Adding items to the list is easy and fast, unless the array grows too large for the block and has to be copied into a new area of memory. Just copy the new item to the end of the list and increase the size by one.

Removing an item from the list is almost as simple. If the item to delete happens to be located at the very end of the array, just reduce the size of the list by one. If it's located at any other spot, you must also copy the element that is located at the very end onto the location that the deleted element used to occupy.

Ordered array

In terms of inserting and removing elements, ordered arrays are mechanically the same as unordered arrays. The difference is that insertions and deletions can only be at one end of the array if the item in question is the largest or smallest in the list. The practical upshot is that dynamic ordered arrays are only efficient if items are added and removed in sorted order.

Binary search tree

Insertions and deletions are where binary search trees have their chance to shine. Insertions and deletions are efficient in binary search trees whether they're made at the beginning, middle, or end of the lists.

Clearly, binary search trees are superior to ordered or unordered arrays in situations that require insertion and deletion in random positions. But insertion and deletion operations in binary search trees require a bit of explanation if you've never seen them before. This is what the next chapter is for, so read on.

---

[2] These uses of the words "static" and "dynamic" are different from their meanings in the phrases "static allocation" and "dynamic allocation." See Appendix C [Glossary], page 331, for more details.

4 Binary Search Trees

The previous chapter motivated the need for binary search trees. This chapter implements a table ADT backed by a binary search tree. Along the way, we'll see how binary search trees are constructed and manipulated in abstract terms as well as in concrete C code.

The library includes a header file ⟨ `bst.h` 24 ⟩ and an implementation file ⟨ `bst.c` 25 ⟩, outlined below. We borrow most of the header file from the generic table headers designed a couple of chapters back, simply replacing *tbl* by *bst*, the prefix used in this table module.

§24 ⟨ `bst.h` 24 ⟩ ≡
⟨ License 1 ⟩
#ifndef `BST_H`
#define `BST_H` 1

#include ⟨ stddef.h ⟩

⟨ Table types; tbl ⇒ bst 14 ⟩
⟨ BST maximum height 28 ⟩
⟨ BST table structure 27 ⟩
⟨ BST node structure 26 ⟩
⟨ BST traverser structure 61 ⟩
⟨ Table function prototypes; tbl ⇒ bst 15 ⟩
⟨ BST extra function prototypes 88 ⟩

#endif /∗ bst.h ∗/

⟨ Table assertion function control directives; tbl ⇒ bst 593 ⟩

§25 ⟨ `bst.c` 25 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include ⟨ string.h ⟩
#include "bst.h"

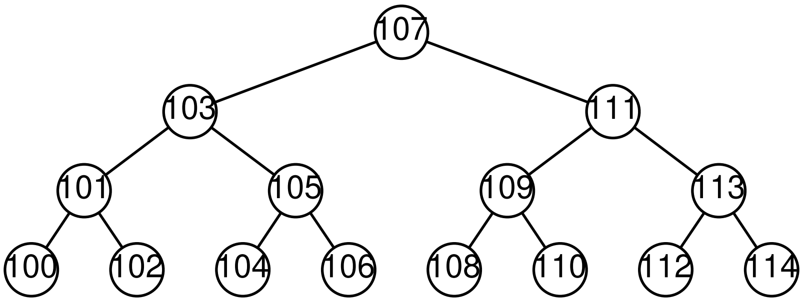
⟨ BST operations 29 ⟩

Exercises:

1. What is the purpose of **#ifndef** `BST_H` . . . **#endif** in ⟨ `bst.h` 24 ⟩ above?

4.1 Vocabulary

When binary search trees, or BSTs, were introduced in the previous chapter, the reason that they were called binary search trees wasn't explained. The diagram below should help to clear up matters, and incidentally let us define some BST-related vocabulary:



This diagram illustrates the binary search tree example from the previous chapter. The circle or **node** at the top, labeled 107, is the starting point of any search. As such, it is called the **root** of the tree. The node connected to it below to the left, labeled 103, is the root's **left child**, and node 111 to its lower right is its **right child**. A node's left child corresponds to *smaller* from the array-based BST of the previous chapter, and a right child corresponds to *larger*.

Some nodes, such as 106 here, don't have any children. Such a node is called a **leaf** or **terminal node**. Although not shown here, it's also possible for a node to have only one child, either on the left or the right side. A node with at least one child is called a **nonterminal node**.

Each node in a binary search tree is, conceptually, the root of its own tree. Such a tree is called a **subtree** of the tree that contains it. The left child of a node and recursively all of that child's children is a subtree of the node, called the **left subtree** of the node. The term **right subtree** is defined similarly for the right side of the node. For instance, above, nodes 104, 105, and 106 are the right subtree of node 103, with 105 as the subtree's root.

A BST without any nodes is called an **empty tree**. Both subtrees of all even-numbered nodes in the BST above are empty trees.

In a binary search tree, the left child of a node, if it exists, has a smaller value than the node, and the right child of a node has a larger value. The more general term **binary tree**, on the other hand, refers to a data structure with the same form as a binary search tree, but which does not necessarily share this property. There are also related, but different, structures simply called "trees".

In this book, all our binary trees are binary search trees, and this book will not discuss plain trees at all. As a result, we will often be a bit loose in terminology and use the term "binary tree" or "tree" when "binary search tree" is the proper term.

Although this book discusses binary search trees exclusively, it is instructive to occasionally display, as a counterexample, a diagram of a binary tree whose nodes are out of order and therefore not a BST. Such diagrams are marked $**$ to reinforce their non-BST nature to the casual browser.

See also: [Knuth 1997], section 2.3; [Knuth 1998b], section 6.2.2; [Cormen 1990], section 13.1; [Sedgewick 1998], section 5.4.

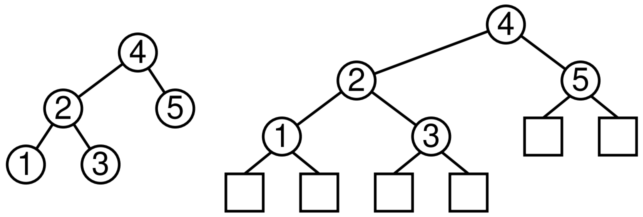
4.1.1 Aside: Differing Definitions

The definitions in the previous section are the ones used in this book. They are the definitions that programmers often use in designing and implementing real programs. However, they are slightly different from the definitions used in formal computer science textbooks. This section gives these formal definitions and contrasts them against our own.

The most important difference is in the definition of a binary tree itself. Formally, a binary tree is either an "external node" or an "internal node" connected to a pair of binary trees called the internal node's left subtree and right subtree. Internal nodes correspond to our notion of nodes, and external nodes correspond roughly to nodes' empty left or right subtrees. The generic term "node" includes both internal and external nodes.

Every internal node always has exactly two children, although those children may be external nodes, so we must also revise definitions that depend on a node's number of children. Then, a "leaf" is an internal node with two external node children and a "nonterminal node" is an internal node at least one of whose children is an internal node. Finally, an "empty tree" is a binary tree that contains of only an external node.

Tree diagrams in books that use these formal definitions show both internal and external nodes. Typically, internal nodes are shown as circles, external nodes as square boxes. Here's an example BST in the format used in this book, shown alongside an identical BST in the format used in formal computer science books:



See also: [Sedgewick 1998], section 5.4.

4.2 Data Types

The types for memory allocation and managing data as **void** ∗ pointers were discussed previously (see Chapter 2 [The Table ADT], page 7), but to build a table implementation using BSTs we must define some additional types. In particular, we need **struct bst_node** to represent an individual node and **struct bst_table** to represent an entire table. The following sections take care of this.

4.2.1 Node Structure

When binary search trees were introduced in the last chapter, we used indexes into an array to reference items' *smaller* and *larger* values. But in C, BSTs are usually constructed using pointers. This is a more general technique, because pointers aren't restricted to references within a single array.

§26 ⟨ BST node structure 26 ⟩ ≡

```
/* A binary search tree node. */
struct bst_node {
    struct bst_node *bst_link[2]; /* Subtrees. */
    void *bst_data; /* Pointer to data. */
};
```

This code is included in §24.

In **struct bst_node**, *bst_link*[0] takes the place of *smaller*, and *bst_link*[1] takes the place of *larger*. If, in our array implementation of binary search trees, either of these would have pointed to the sentinel, it instead is assigned `NULL`, the null pointer constant.

In addition, *bst_data* replaces *value*. We use a **void** ∗ generic pointer here, instead of **int** as used in the last chapter, to let any kind of data be stored in the BST. See Section 2.3 [Comparison Function], page 8, for more information on **void** ∗ pointers.

4.2.2 Tree Structure

The **struct bst_table** structure ties together all of the data needed to keep track of a table implemented as a binary search tree:

§27 ⟨ BST table structure 27 ⟩ ≡

```
/* Tree data structure. */
struct bst_table {
    struct bst_node *bst_root; /* Tree's root. */
    bst_comparison_func *bst_compare; /* Comparison function. */
    void *bst_param; /* Extra argument to bst_compare. */
    struct libavl_allocator *bst_alloc; /* Memory allocator. */
    size_t bst_count; /* Number of items in tree. */
    unsigned long bst_generation; /* Generation number. */
};
```

This code is included in §24, §142, and §192.

Most of **struct bst_table**'s members should be familiar. Member *bst_root* points to the root node of the BST. Together, *bst_compare* and *bst_param* specify how items are compared (see Section 2.4 [Item and Copy Functions], page 10). The members of *bst_alloc* specify how to allocate memory for the BST (see Section 2.5 [Memory Allocation], page 11). The number of items in the BST is stored in *bst_count* (see Section 2.7 [Count], page 13).

The final member, *bst_generation*, is a **generation number**. When a tree is created, it starts out at zero. After that, it is incremented every time the tree is modified in a way that might disturb a traverser. We'll talk more about the generation number later (see Section 4.9.3 [Better Iterative Traversal], page 53).

Exercises:

***1.** Why is it a good idea to include *bst_count* in **struct bst_table**? Under what circumstances would it be better to omit it?

4.2.3 Maximum Height

For efficiency, some of the BST routines use a stack of a fixed maximum height. This maximum height affects the maximum number of nodes that can be fully supported by Libavl in any given tree, because a binary tree of height n contains at most $2^n - 1$ nodes.

The `BST_MAX_HEIGHT` macro sets the maximum height of a BST. The default value of 32 allows for trees with up to $2^{32} - 1 = 4{,}294{,}967{,}295$ nodes. On today's common 32-bit computers that support only 4 GB of memory at most, this is hardly a limit, because memory would be exhausted long before the tree became too big.

The BST routines that use fixed stacks also detect stack overflow and call a routine to "balance" or restructure the tree in order to reduce its height to the permissible range. The limit on the BST height is therefore not a severe restriction.

§28 ⟨ BST maximum height 28 ⟩ ≡

```
/* Maximum BST height. */
#ifndef BST_MAX_HEIGHT
#define BST_MAX_HEIGHT 32
#endif
```

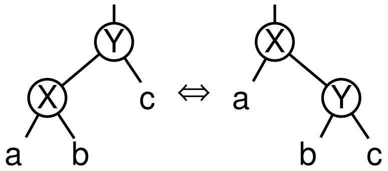
This code is included in §24, §142, §297, §415, and §519.

Exercises:

1. Suggest a reason why the `BST_MAX_HEIGHT` macro is defined conditionally. Are there any potential pitfalls?

4.3 Rotations

Soon we'll jump right in and start implementing the table functions for BSTs. But before that, there's one more topic to discuss, because they'll keep coming up from time to time throughout the rest of the book. This topic is the concept of a **rotation**. A rotation is a simple transformation of a binary tree that looks like this:



In this diagram, X and Y represent nodes and a, b, and c are arbitrary binary trees that may be empty. A rotation that changes a binary tree of the form shown on the left to the form shown on the right is called a **right rotation** on Y. Going the other way, it is a **left rotation** on X.

This figure also introduces new graphical conventions. First, the line leading vertically down to the root explicitly shows that the BST may be a subtree of a larger tree. Also, (possible empty) subtrees, as opposed to individual nodes, are indicated by lowercase letters not enclosed by circles.

A rotation changes the local structure of a binary tree without changing its ordering as seen through inorder traversal. That's a subtle statement, so let's dissect it bit by bit. Rotations have the following properties:

Rotations change the structure of a binary tree.
: In particular, rotations can often, depending on the tree's shape, be used to change the height of a part of a binary tree.

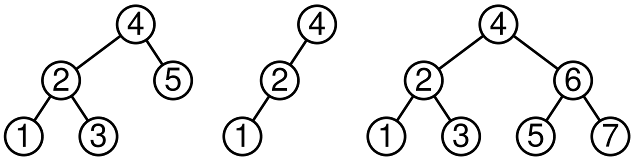
Rotations change the local structure of a binary tree.
: Any given rotation only affects the node rotated and its immediate children. The node's ancestors and its children's children are unchanged.

Rotations do not change the ordering of a binary tree.
: If a binary tree is a binary search tree before a rotation, it is a binary search tree after a rotation. So, we can safely use rotations to rearrange a BST-based structure, without concerns about upsetting its ordering.

See also: [Cormen 1990], section 14.2; [Sedgewick 1998], section 12.8.

Exercises:

1. For each of the binary search trees below, perform a right rotation at node 4.



2. Write a pair of functions, one to perform a right rotation at a given BST node, one to perform a left rotation. What should be the type of the functions' parameter?

4.4 Operations

Now can start to implement the operations that we'll want to perform on BSTs. Here's the outline of the functions we'll implement. We use the generic table insertion convenience functions from Exercise 2.8-3 to implement *bst_insert*() and *bst_replace*(), as well the generic assertion function implementations from Exercise 2.9-2 to implement *tbl_assert_insert*() and *tbl_assert_delete*(). We also include a copy of the default memory allocation functions for use with BSTs:

§29 ⟨ BST operations 29 ⟩ ≡
⟨ BST creation function 30 ⟩
⟨ BST search function 31 ⟩
⟨ BST item insertion function 32 ⟩
⟨ Table insertion convenience functions; tbl ⇒ bst 592 ⟩
⟨ BST item deletion function 37 ⟩
⟨ BST traversal functions 63 ⟩
⟨ BST copy function 83 ⟩
⟨ BST destruction function 84 ⟩
⟨ BST balance function 87 ⟩
⟨ Default memory allocation functions; tbl ⇒ bst 6 ⟩
⟨ Table assertion functions; tbl ⇒ bst 594 ⟩

This code is included in §25.

4.5 Creation

We need to write *bst_create*() to create an empty BST. All it takes is a little bit of memory allocation and initialization:

§30 ⟨ BST creation function 30 ⟩ ≡

```
struct bst_table *bst_create (bst_comparison_func *compare, void *param,
                              struct libavl_allocator *allocator) {
  struct bst_table *tree;
  assert (compare != NULL);
  if (allocator == NULL)
    allocator = &bst_allocator_default;
  tree = allocator->libavl_malloc (allocator, sizeof *tree);
  if (tree == NULL)
```

```
        return NULL;
    tree->bst_root = NULL;
    tree->bst_compare = compare;
    tree->bst_param = param;
    tree->bst_alloc = allocator;
    tree->bst_count = 0;
    tree->bst_generation = 0;
    return tree;
}
```

This code is included in §29, §145, and §196.

4.6 Search

Searching a binary search tree works just the same way as it did before when we were doing it inside an array. We can implement *bst_find*() immediately:

§31 ⟨ BST search function 31 ⟩ ≡

```
void *bst_find (const struct bst_table *tree, const void *item) {
    const struct bst_node *p;
    assert (tree != NULL && item != NULL);
    for (p = tree->bst_root; p != NULL; ) {
        int cmp = tree->bst_compare (item, p->bst_data, tree->bst_param);
        if (cmp < 0) p = p->bst_link[0];
        else if (cmp > 0) p = p->bst_link[1];
        else /* cmp == 0 */ return p->bst_data;
    }
    return NULL;
}
```

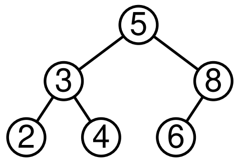
This code is included in §29, §145, §196, §489, §522, and §554.

See also: [Knuth 1998b], section 6.2.2; [Cormen 1990], section 13.2; [Kernighan 1988], section 3.3; [Bentley 2000], chapters 4 and 5, section 9.3, appendix 1; [Sedgewick 1998], program 12.7.

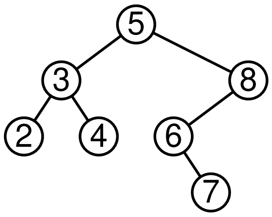
4.7 Insertion

Inserting new nodes into a binary search tree is easy. To start out, we work the same way as in a search, traversing the tree from the top down, as if we were searching for the item that we're inserting. If we find one, the item is already in the tree, and we need not insert it again. But if the new item is not in the tree, eventually we "fall off" the bottom of the tree. At this point we graft the new item as a child of the node that we last examined.

An example is in order. Consider this binary search tree:



Suppose that we wish to insert a new item, 7, into the tree. 7 is greater than 5, so examine 5's right child, 8. 7 is less than 8, so examine 8's left child, 6. 7 is greater than 6, but 6 has no right child. So, make 7 the right child of 6:



We cast this in a form compatible with the abstract description as follows:

§32 ⟨BST item insertion function 32⟩ ≡

```
void **bst_probe (struct bst_table *tree, void *item) {
    struct bst_node *p, *q; /* Current node in search and its parent. */
    int dir; /* Side of q on which p is located. */
    struct bst_node *n; /* Newly inserted node. */

    assert (tree != NULL && item != NULL);

    for (q = NULL, p = tree->bst_root; p != NULL; q = p, p = p->bst_link[dir]) {
        int cmp = tree->bst_compare (item, p->bst_data, tree->bst_param);
        if (cmp == 0)
            return &p->bst_data;
        dir = cmp > 0;
    }

    n = tree->bst_alloc->libavl_malloc (tree->bst_alloc, sizeof *p);
    if (n == NULL)
        return NULL;

    tree->bst_count++;
    n->bst_link[0] = n->bst_link[1] = NULL;
    n->bst_data = item;
    if (q != NULL)
        q->bst_link[dir] = n;
    else tree->bst_root = n;

    return &n->bst_data;
}
```

This code is included in §29.

See also: [Knuth 1998b], algorithm 6.2.2T; [Cormen 1990], section 13.3; [Bentley 2000], section 13.3; [Sedgewick 1998], program 12.7.

Exercises:

1. Explain the expression p = (**struct bst_node** $*$) &*tree*→*bst_root*. Suggest an alternative.

2. Rewrite *bst_probe*() to use only a single local variable of type **struct bst_node** $**$.

3. Suppose we want to make a new copy of an existing binary search tree, preserving the original tree's shape, by inserting items into a new, currently empty tree. What constraints are there on the order of item insertion?

4. Write a function that calls a provided **bst_item_func** for each node in a provided BST in an order suitable for reproducing the original BST, as discussed in Exercise 3.

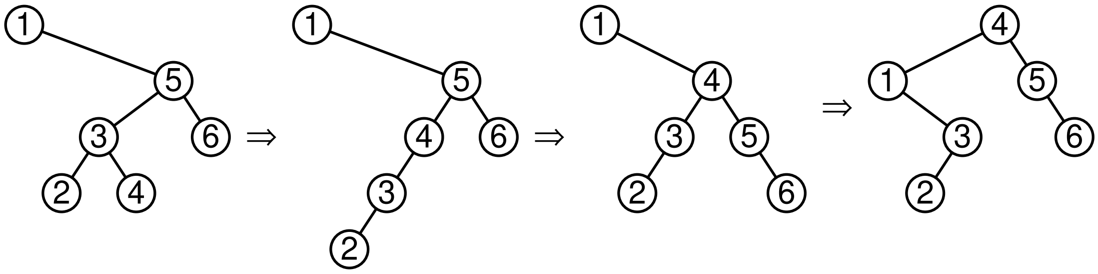
4.7.1 Aside: Root Insertion

One side effect of the usual method for BST insertion, implemented in the previous section, is that items inserted more recently tend to be farther from the root, and therefore it takes longer to find them than items inserted longer ago. If all items are equally likely to be requested in a search, this is unimportant, but this is regrettable for some common usage patterns, where recently inserted items tend to be searched for more often than older items.

In this section, we examine an alternative scheme for insertion that addresses this problem, called "insertion at the root" or "root insertion". An insertion with this algorithm always places the new node at the root of the tree. Following a series of such insertions, nodes inserted more recently tend to be nearer the root than other nodes.

As a first attempt at implementing this idea, we might try simply making the new node the root and assigning the old root as one of its children. Unfortunately, this and similar approaches will not work because there is no guarantee that nodes in the existing tree have values all less than or all greater than the new node.

An approach that will work is to perform a conventional insertion as a leaf node, then use a series of rotations to move the new node to the root. For example, the diagram below illustrates rotations to move node 4 to the root. A left rotation on 3 changes the first tree into the second, a right rotation on 5 changes the second into the third, and finally a left rotation on 1 moves 4 into the root position:



The general rule follows the pattern above. If we moved down to the left from a node x during the insertion search, we rotate right at x. If we moved down to the right, we rotate left.

The implementation is straightforward. As we search for the insertion point we keep track of the nodes we've passed through, then after the insertion we return to each of them in reverse order and perform a rotation:

§33 ⟨ BST item insertion function, root insertion version 33 ⟩ ≡
void ∗∗*bst_probe* (**struct bst_table** ∗*tree*, **void** ∗*item*) {
  ⟨ *rb_probe*() local variables; rb ⇒ bst 198 ⟩
  ⟨ Step 1: Search BST for insertion point, root insertion version 34 ⟩
  ⟨ Step 2: Insert new BST node, root insertion version 35 ⟩
  ⟨ Step 3: Move BST node to root 36 ⟩
  return &*n*→*bst_data*;

```
}
```

§34 ⟨ Step 1: Search BST for insertion point, root insertion version 34 ⟩ ≡

```
pa[0] = (struct bst_node *) &tree->bst_root;
da[0] = 0;
k = 1;
for (p = tree->bst_root; p != NULL; p = p->bst_link[da[k - 1]]) {
    int cmp = tree->bst_compare (item, p->bst_data, tree->bst_param);
    if (cmp == 0)
        return &p->bst_data;
    if (k >= BST_MAX_HEIGHT) {
        bst_balance (tree);
        return bst_probe (tree, item);
    }
    pa[k] = p;
    da[k++] = cmp > 0;
}
```

This code is included in §33.

§35 ⟨ Step 2: Insert new BST node, root insertion version 35 ⟩ ≡

```
n = pa[k - 1]->bst_link[da[k - 1]] =
    tree->bst_alloc->libavl_malloc (tree->bst_alloc, sizeof *n);
if (n == NULL)
    return NULL;
n->bst_link[0] = n->bst_link[1] = NULL;
n->bst_data = item;
tree->bst_count++;
tree->bst_generation++;
```

This code is included in §33.

§36 ⟨ Step 3: Move BST node to root 36 ⟩ ≡

```
for (; k > 1; k--) {
    struct bst_node *q = pa[k - 1];
    if (da[k - 1] == 0) {
        q->bst_link[0] = n->bst_link[1];
        n->bst_link[1] = q;
    } else /* da[k - 1] == 1 */ {
        q->bst_link[1] = n->bst_link[0];
        n->bst_link[0] = q;
    }
    pa[k - 2]->bst_link[da[k - 2]] = n;
}
```

This code is included in §33, §622, and §627.

See also: [Sedgewick 1998], section 12.8.

Exercises:

1. Root insertion will prove useful later when we write a function to join a pair of disjoint BSTs (see Section 4.13 [Joining BSTs], page 78). For that purpose, we need to be able to insert a preallocated node as the root of an arbitrary tree that may be a subtree of some other tree. Write a function to do this matching the following prototype:

static int *root_insert* (**struct bst_table** **tree*, **struct bst_node** ***root*,
struct bst_node **new_node*);

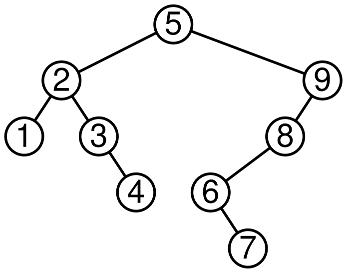
Your function should insert *new_node* at **root* using root insertion, storing *new_node* into **root*, and return nonzero only if successful. The subtree at **root* is in *tree*. You may assume that no node matching *new_node* exists within subtree *root*.

2. Now implement a root insertion as in Exercise 1, except that the function is not allowed to fail, and rebalancing the tree is not acceptable either. Use the same prototype with the return type changed to **void**.

***3.** Suppose that we perform a series of root insertions in an initially empty BST. What kinds of insertion orders require a large amount of stack?

4.8 Deletion

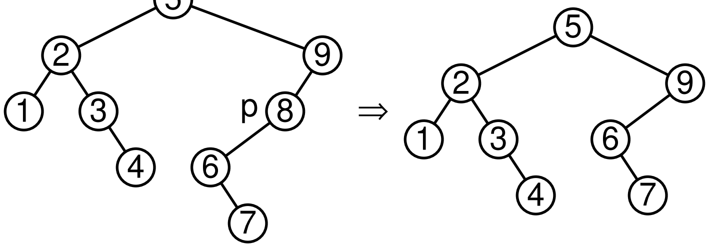
Deleting an item from a binary search tree is a little harder than inserting one. Before we write any code, let's consider how to delete nodes from a binary search tree in an abstract fashion. Here's a BST from which we can draw examples during the discussion:



It is more difficult to remove some nodes from this tree than to remove others. Here, we recognize three distinct cases (Exercise 4.8-1 offers a fourth), described in detail below in terms of the deletion of a node designated p.

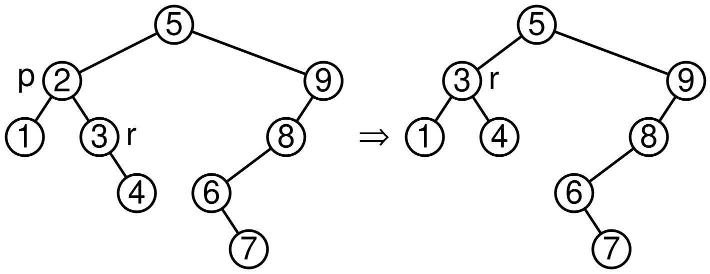
Case 1: *p* has no right child

It is trivial to delete a node with no right child, such as node 1, 4, 7, or 8 above. We replace the pointer leading to p by p's left child, if it has one, or by a null pointer, if not. In other words, we replace the deleted node by its left child. For example, the process of deleting node 8 looks like this:



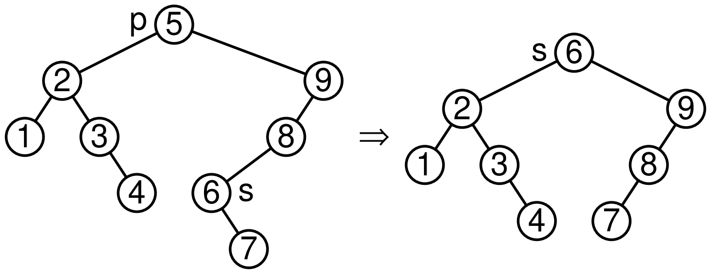
Case 2: *p*'s right child has no left child

This case deletes any node *p* with a right child *r* that itself has no left child. Nodes 2, 3, and 6 in the tree above are examples. In this case, we move *r* into *p*'s place, attaching *p*'s former left subtree, if any, as the new left subtree of *r*. For instance, to delete node 2 in the tree above, we can replace it by its right child 3, giving node 2's left child 1 to node 3 as its new left child. The process looks like this:



Case 3: *p*'s right child has a left child

This is the "hard" case, where *p*'s right child *r* has a left child. but if we approach it properly we can make it make sense. Let *p*'s **inorder successor**, that is, the node with the smallest value greater than *p*, be *s*. Then, our strategy is to detach *s* from its position in the tree, which is always an easy thing to do, and put it into the spot formerly occupied by *p*, which disappears from the tree. In our example, to delete node 5, we move inorder successor node 6 into its place, like this:



But how do we know that node *s* exists and that we can delete it easily? We know that it exists because otherwise this would be case 1 or case 2 (consider their conditions). We can easily detach from its position for a more subtle reason: *s* is the inorder successor of *p* and is therefore has the smallest value in *p*'s right subtree, so *s* cannot have a left child. (If it did, then this left child would have a smaller value than *s*, so it, rather than *s*, would be *p*'s inorder successor.) Because *s* doesn't have a left child, we can simply replace it by its right child, if any. This is the mirror image of case 1.

Implementation

The code for BST deletion closely follows the above discussion. Let's start with an outline of the function:

§37 ⟨ BST item deletion function 37 ⟩ ≡

```
void *bst_delete (struct bst_table *tree, const void *item) {
    struct bst_node *p, *q; /* Node to delete and its parent. */
```

```
  int cmp; /* Comparison between p→bst_data and item. */
  int dir; /* Side of q on which p is located. */
  assert (tree != NULL && item != NULL);
  ⟨Step 1: Find BST node to delete 38⟩
  ⟨Step 2: Delete BST node 39⟩
  ⟨Step 3: Finish up after deleting BST node 44⟩
}
```

This code is included in §29.

We begin by finding the node to delete, in much the same way that *bst_find*() did. But, in every case above, we replace the link leading to the node being deleted by another node or a null pointer. To do so, we have to keep track of the pointer that led to the node to be deleted. This is the purpose of *q* and *dir* in the code below.

§38 ⟨Step 1: Find BST node to delete 38⟩ ≡

```
p = (struct bst_node *) &tree→bst_root;
for (cmp = -1; cmp != 0; cmp = tree→bst_compare (item, p→bst_data, tree→bst_param)) {
  dir = cmp > 0;
  q = p;
  p = p→bst_link[dir];
  if (p == NULL)
    return NULL;
}
item = p→bst_data;
```

This code is included in §37.

Now we can actually delete the node. Here is the code to distinguish between the three cases:

§39 ⟨Step 2: Delete BST node 39⟩ ≡

```
if (p→bst_link[1] == NULL) { ⟨Case 1 in BST deletion 40⟩ }
else {
  struct bst_node *r = p→bst_link[1];
  if (r→bst_link[0] == NULL) {
    ⟨Case 2 in BST deletion 41⟩
  } else {
    ⟨Case 3 in BST deletion 42⟩
  }
}
```

This code is included in §37.

In case 1, we simply replace the node by its left subtree:

§40 ⟨Case 1 in BST deletion 40⟩ ≡

```
q→bst_link[dir] = p→bst_link[0];
```

This code is included in §39.

In case 2, we attach the node's left subtree as its right child *r*'s left subtree, then replace the node by *r*:

§41 ⟨Case 2 in BST deletion 41⟩ ≡

```
r→bst_link[0] = p→bst_link[0];
```

```
q→bst_link[dir] = r;
```

This code is included in §39.

We begin case 3 by finding p's inorder successor as s, and the parent of s as r. Node p's inorder successor is the smallest value in p's right subtree and that the smallest value in a tree can be found by descending to the left until a node with no left child is found:

§42 ⟨ Case 3 in BST deletion 42 ⟩ ≡

```
struct bst_node *s;
for (;;) {
    s = r→bst_link[0];
    if (s→bst_link[0] == NULL)
        break;
    r = s;
}
```

See also §43.

This code is included in §39.

Case 3 wraps up by adjusting pointers so that s moves into p's place:

§43 ⟨ Case 3 in BST deletion 42 ⟩ +≡

```
r→bst_link[0] = s→bst_link[1];
s→bst_link[0] = p→bst_link[0];
s→bst_link[1] = p→bst_link[1];
q→bst_link[dir] = s;
```

As the final step, we decrement the number of nodes in the tree, free the node, and return its data:

§44 ⟨ Step 3: Finish up after deleting BST node 44 ⟩ ≡

```
tree→bst_alloc→libavl_free (tree→bst_alloc, p);
tree→bst_count−−;
tree→bst_generation++;
return (void *) item;
```

This code is included in §37.

See also: [Knuth 1998b], algorithm 6.2.2D; [Cormen 1990], section 13.3.

Exercises:

1. Write code for a case 1.5 which handles deletion of nodes with no left child.

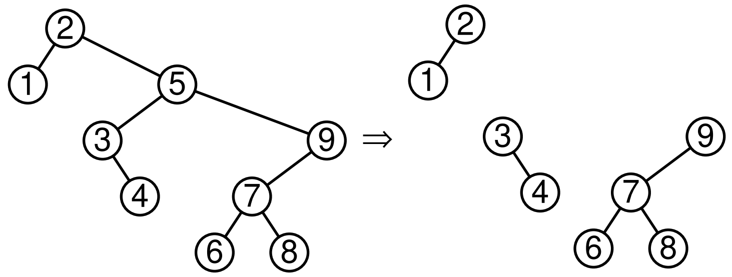
2. In the code presented above for case 3, we update pointers to move s into p's position, then free p. An alternate approach is to replace p's data by s's and delete s. Write code to use this approach. Can a similar modification be made to either of the other cases?

***3.** The code in the previous exercise is a few lines shorter than that in the main text, so it would seem to be preferable. Explain why the revised code, and other code based on the same idea, cannot be used in Libavl. (Hint: consider the semantics of Libavl traversers.)

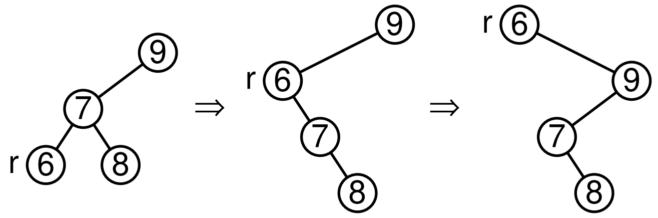
4.8.1 Aside: Deletion by Merging

The Libavl algorithm for deletion is commonly used, but it is also seemingly ad-hoc and arbitrary in its approach. In this section we'll take a look at another algorithm that may seem a little more uniform. Unfortunately, though it is conceptually simpler in some ways, in practice this algorithm is both slower and more difficult to properly implement.

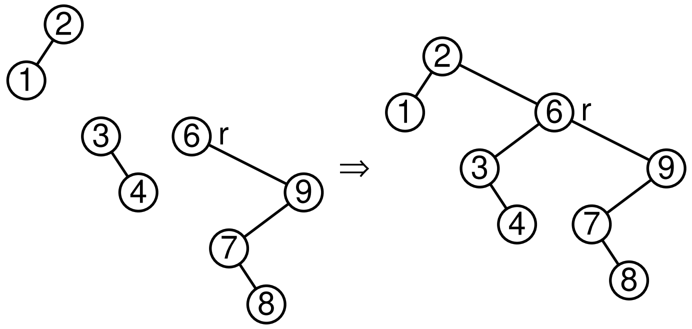
The idea behind this algorithm is to consider deletion as breaking the links between the deleted node and its parent and children. In the most general case, we end up with three disconnected BSTs, one that contains the deleted node's parent and two corresponding to the deleted node's former subtrees. The diagram below shows how this idea works out for the deletion of node 5 from the tree on the left:



Of course, the problem then becomes to reassemble the pieces into a single binary search tree. We can do this by merging the two former subtrees of the deleted node and attaching them as the right child of the parent subtree. As the first step in merging the subtrees, we take the minimum node r in the former right subtree and repeatedly perform a right rotation on its parent, until it is the root of its subtree. The process up to this point looks like this for our example, showing only the subtree containing r:



Now, because r is the root and the minimum node in its subtree, r has no left child. Also, all the nodes in the opposite subtree are smaller than r. So to merge these subtrees, we simply link the opposite subtree as r's left child and connect r in place of the deleted node:



The function outline is straightforward:

§45 ⟨ BST item deletion function, by merging 45 ⟩ ≡
void ∗*bst_delete* (**struct bst_table** ∗*tree*, **const void** ∗*item*) {
  struct bst_node ∗*p*; /∗ The node to delete, or a node part way to it. ∗/

```
    struct bst_node *q; /* Parent of p. */
    int cmp, dir; /* Result of comparison between item and p. */
    assert (tree != NULL && item != NULL);
    ⟨Step 1: Find BST node to delete by merging 46⟩
    ⟨Step 2: Delete BST node by merging 47⟩
    ⟨Step 3: Finish up after BST deletion by merging 48⟩
    return (void *) item;
}
```

First we search for the node to delete, storing it as p and its parent as q:

§46 ⟨Step 1: Find BST node to delete by merging 46⟩ ≡

```
p = (struct bst_node *) &tree->bst_root;
for (cmp = -1; cmp != 0; cmp = tree->bst_compare (item, p->bst_data, tree->bst_param)) {
    dir = cmp > 0;
    q = p;
    p = p->bst_link[dir];
    if (p == NULL)
        return NULL;
}
```

This code is included in §45.

The actual deletion process is not as simple. We handle specially the case where p has no right child. This is unfortunate for uniformity, but simplifies the rest of the code considerably. The main case starts off with a loop on variable r to build a stack of the nodes in the right subtree of p that will need to be rotated. After the loop, r is the minimum value in p's right subtree. This will be the new root of the merged subtrees after the rotations, so we set r as q's child on the appropriate side and r's left child as p's former left child. After that the only remaining task is the rotations themselves, so we perform them and we're done:

§47 ⟨Step 2: Delete BST node by merging 47⟩ ≡

```
if (p->bst_link[1] != NULL) {
    struct bst_node *pa[BST_MAX_HEIGHT]; /* Nodes on stack. */
    unsigned char da[BST_MAX_HEIGHT]; /* Directions moved from stack nodes. */
    int k = 0; /* Stack height. */
    struct bst_node *r; /* Iterator; final value is minimum node in subtree. */
    for (r = p->bst_link[1]; r->bst_link[0] != NULL; r = r->bst_link[0]) {
        if (k >= BST_MAX_HEIGHT) {
            bst_balance (tree);
            return bst_delete (tree, item);
        }
        pa[k] = r;
        da[k++] = 0;
    }
    q->bst_link[dir] = r;
    r->bst_link[0] = p->bst_link[0];
    for (; k > 0; k--) {
```

```
            struct bst_node *y = pa[k - 1];
            struct bst_node *x = y->bst_link[0];
            y->bst_link[0] = x->bst_link[1];
            x->bst_link[1] = y;
            if (k > 1)
                pa[k - 2]->bst_link[da[k - 2]] = x;
        }
    }
else q->bst_link[dir] = p->bst_link[0];
```

This code is included in §45.

Finally, there's a bit of obligatory bookkeeping:

§48 ⟨ Step 3: Finish up after BST deletion by merging 48 ⟩ ≡

```
item = p->bst_data;
tree->bst_alloc->libavl_free (tree->bst_alloc, p);
tree->bst_count--;
tree->bst_generation++;
```

This code is included in §45.

See also: [Sedgewick 1998], section 12.9.

4.9 Traversal

After we've been manipulating a binary search tree for a while, we will want to know what items are in it. The process of enumerating the items in a binary search tree is called **traversal**. LIBAVL provides the *bst_t_** functions for a particular kind of traversal called **inorder traversal**, so-called because items are enumerated in sorted order.

In this section we'll implement three algorithms for traversal. Each of these algorithms is based on and in some way improves upon the previous algorithm. The final implementation is the one used in LIBAVL, so we will implement all of the *bst_t_** functions for it.

Before we start looking at particular algorithms, let's consider some criteria for evaluating traversal algorithms. The following are not the only criteria that could be used, but they are indeed important:[1]

complexity
: Is it difficult to describe or to correctly implement the algorithm? Complex algorithms also tend to take more code than simple ones.

efficiency
: Does the algorithm make good use of time and memory? The ideal traversal algorithm would require time proportional to the number of nodes traversed and a constant amount of space. In this chapter we will meet this ideal time criterion and come close on the space criterion for the average case. In future chapters we will be able to do better even in the worst case.

[1] Some of these terms are not generic BST vocabulary. Rather, they have been adopted for these particular uses in this text. You can consider the above to be our working definitions of these terms.

convenience
Is it easy to integrate the traversal functions into other code? Callback functions are not as easy to use as other methods that can be used from **for** loops (see Section 4.9.2.1 [Improving Convenience], page 50).

reliability Are there pathological cases where the algorithm breaks down? If so, is it possible to fix these problems using additional time or space?

generality Does the algorithm only allow iteration in a single direction? Can we begin traversal at an arbitrary node, or just at the least or greatest node?

resilience If the tree is modified during a traversal, is it possible to continue traversal, or does the modification invalidate the traverser?

The first algorithm we will consider uses recursion. This algorithm is worthwhile primarily for its simplicity. In C, such an algorithm cannot be made as efficient, convenient, or general as other algorithms without unacceptable compromises. It is possible to make it both reliable and resilient, but we won't bother because of its other drawbacks.

We arrive at our second algorithm through a literal transformation of the recursion in the first algorithm into iteration. The use of iteration lets us improve the algorithm's memory efficiency, and, on many machines, its time efficiency as well. The iterative algorithm also lets us improve the convenience of using the traverser. We could also add reliability and resilience to an implementation of this algorithm, but we'll save that for later. The only problem with this algorithm, in fact, lies in its generality: it works best for moving only in one direction and starting from the least or greatest node.

The importance of generality is what draws us to the third algorithm. This algorithm is based on ideas from the previous iterative algorithm along with some simple observations. This algorithm is no more complex than the previous one, but it is more general, allowing easily for iteration in either direction starting anywhere in the tree. This is the algorithm used in Libavl, so we build an efficient, convenient, reliable, general, resilient implementation.

4.9.1 Traversal by Recursion

To figure out how to traverse a binary search tree in inorder, think about a BST's structure. A BST consists of a root, a left subtree, and right subtree. All the items in the left subtree have smaller values than the root and all the items in the right subtree have larger values than the root.

That's good enough right there: we can traverse a BST in inorder by dealing with its left subtree, then doing with the root whatever it is we want to do with each node in the tree (generically, **visit** the root node), then dealing with its right subtree. But how do we deal with the subtrees? Well, they're BSTs too, so we can do the same thing: traverse its left subtree, then visit its root, then traverse its right subtree, and so on. Eventually the process terminates because at some point the subtrees are null pointers, and nothing needs to be done to traverse an empty tree.

Writing the traversal function is almost trivial. We use **bst_item_func** to visit a node (see Section 2.4 [Item and Copy Functions], page 10):

§49 ⟨ Recursive traversal of BST 49 ⟩ ≡

```
static void traverse_recursive (struct bst_node *node, bst_item_func *action, void *param) {
  if (node != NULL) {
    traverse_recursive (node->bst_link[0], action, param);
    action (node->bst_data, param);
    traverse_recursive (node->bst_link[1], action, param);
  }
}
```

See also §50.

We also want a wrapper function to insulate callers from the existence of individual tree nodes:

§50 ⟨ Recursive traversal of BST 49 ⟩ +≡

```
void walk (struct bst_table *tree, bst_item_func *action, void *param) {
  assert (tree != NULL && action != NULL);
  traverse_recursive (tree->bst_root, action, param);
}
```

See also: [Knuth 1997], section 2.3.1; [Cormen 1990], section 13.1; [Sedgewick 1998], program 12.8.

Exercises:

1. Instead of checking for a null *node* at the top of *traverse_recursive*(), would it be better to check before calling in each place that the function is called? Why or why not?

2. Some languages, such as Pascal, support the concept of **nested functions**, that is, functions within functions, but C does not. Some algorithms, including recursive tree traversal, can be expressed much more naturally with this feature. Rewrite *walk*(), in a hypothetical C-like language that supports nested functions, as a function that calls an inner, recursively defined function. The nested function should only take a single parameter. (The GNU C compiler supports nested functions as a language extension, so you may want to use it to check your code.)

4.9.2 Traversal by Iteration

The recursive approach of the previous section is one valid way to traverse a binary search tree in sorted order. This method has the advantages of being simple and "obviously correct". But it does have problems with efficiency, because each call to *traverse_recursive*() receives its own duplicate copies of arguments *action* and *param*, and with convenience, because writing a new callback function for each traversal is unpleasant. It has other problems, too, as already discussed, but these are the ones to be addressed immediately.

Unfortunately, neither problem can be solved acceptably in C using a recursive method, the first because the traversal function has to somehow know the action function and the parameter to pass to it, and the second because there is simply no way to jump out of and then back into recursive calls in C.[2] Our only option is to use an algorithm that does not involve recursion.

[2] This is possible in some other languages, such as Scheme, that support "coroutines" as well as subroutines.

The simplest way to eliminate recursion is by a literal conversion of the recursion to iteration. This is the topic of this section. Later, we will consider a slightly different, and in some ways superior, iterative solution.

Converting recursion into iteration is an interesting problem. There are two main ways to do it:

tail recursion elimination
: If a recursive call is the last action taken in a function, then it is equivalent to a **goto** back to the beginning of the function, possibly after modifying argument values. (If the function has a return value then the recursive call must be a **return** statement returning the value received from the nested call.) This form of recursion is called **tail recursion**.

save-and-restore recursion elimination
: In effect, a recursive function call saves a copy of argument values and local variables, modifies the arguments, then executes a **goto** to the beginning of the function. Accordingly, the return from the nested call is equivalent to restoring the saved arguments and local variables, then executing a **goto** back to the point where the call was made.

We can make use of both of these rules in converting *traverse_recursive*() to iterative form. First, does *traverse_recursive*() ever call itself as its last action? The answer is yes, so we can convert that to an assignment plus a **goto** statement:

§51 ⟨ Iterative traversal of BST, take 1 51 ⟩ ≡

```
static void traverse_iterative (struct bst_node *node, bst_item_func *action, void *param) {
start:
    if (node != NULL) {
        traverse_iterative (node->bst_link[0], action, param);
        action (node->bst_data, param);
        node = node->bst_link[1];
        goto start;
    }
}
```

Sensible programmers are not fond of **goto**. Fortunately, it is easy to eliminate by rephrasing in terms of a **while** loop:

§52 ⟨ Iterative traversal of BST, take 2 52 ⟩ ≡

```
static void traverse_iterative (struct bst_node *node, bst_item_func *action, void *param) {
    while (node != NULL) {
        traverse_iterative (node->bst_link[0], action, param);
        action (node->bst_data, param);
        node = node->bst_link[1];
    }
}
```

This still leaves another recursive call, one that is not tail recursive. This one must be eliminated by saving and restoring values. A stack is ideal for this purpose. For now, we use a stack of fixed size `BST_MAX_HEIGHT` and deal with stack overflow by aborting. Later, we'll handle overflow more gracefully. Here's the code:

§53 ⟨Iterative traversal of BST, take 3 53⟩ ≡

```
static void traverse_iterative (struct bst_node *node, bst_item_func *action, void *param) {
    struct bst_node *stack[BST_MAX_HEIGHT];
    size_t height = 0;
start:
    while (node != NULL) {
        if (height >= BST_MAX_HEIGHT) {
            fprintf (stderr, "tree too deep\n");
            exit (EXIT_FAILURE);
        }
        stack[height++] = node;
        node = node->bst_link[0];
        goto start;
    resume:
        action (node->bst_data, param);
        node = node->bst_link[1];
    }
    if (height > 0) {
        node = stack[--height];
        goto resume;
    }
}
```

This code, an ugly mash of statements, is a prime example of why **goto** statements are discouraged, but its relationship with the earlier code is clear. To make it acceptable for real use, we must rephrase it. First, we can eliminate label *resume* by recognizing that it can only be reached from the corresponding **goto** statement, then moving its code appropriately:

§54 ⟨Iterative traversal of BST, take 4 54⟩ ≡

```
static void traverse_iterative (struct bst_node *node, bst_item_func *action, void *param) {
    struct bst_node *stack[BST_MAX_HEIGHT];
    size_t height = 0;
start:
    while (node != NULL) {
        if (height >= BST_MAX_HEIGHT) {
            fprintf (stderr, "tree too deep\n");
            exit (EXIT_FAILURE);
        }
        stack[height++] = node;
        node = node->bst_link[0];
        goto start;
    }
    if (height > 0) {
        node = stack[--height];
        action (node->bst_data, param);
        node = node->bst_link[1];
        goto start;
```

```
    }
}
```

The first remaining **goto** statement can be eliminated without any other change, because it is redundant; the second, by enclosing the whole function body in an "infinite loop":

§55 ⟨ Iterative traversal of BST, take 5 55 ⟩ ≡

```
static void traverse_iterative (struct bst_node *node, bst_item_func *action, void *param) {
    struct bst_node *stack[BST_MAX_HEIGHT];
    size_t height = 0;
    for (;;) {
        while (node != NULL) {
            if (height >= BST_MAX_HEIGHT) {
                fprintf (stderr, "tree too deep\n");
                exit (EXIT_FAILURE);
            }
            stack[height++] = node;
            node = node->bst_link[0];
        }
        if (height == 0)
            break;
        node = stack[--height];
        action (node->bst_data, param);
        node = node->bst_link[1];
    }
}
```

This initial iterative version takes care of the efficiency problem.

Exercises:

1. Function *traverse_iterative*() relies on *stack*[], a stack of nodes yet to be visited, which as allocated can hold up to BST_MAX_HEIGHT nodes. Consider the following questions concerning *stack*[]:

a. What is the maximum height this stack will attain in traversing a binary search tree containing n nodes, if the binary tree has minimum possible height?

b. What is the maximum height this stack can attain in traversing any binary tree of n nodes? The minimum height?

c. Under what circumstances is it acceptable to use a fixed-size stack as in the example code?

d. Rewrite *traverse_iterative*() to dynamically expand *stack*[] in case of overflow.

e. Does *traverse_recursive*() also have potential for running out of "stack" or "memory"? If so, more or less than *traverse_iterative*() as modified by the previous part?

4.9.2.1 Improving Convenience

Now we can work on improving the convenience of our traversal function. But, first, perhaps it's worthwhile to demonstrate how inconvenient it really can be to use *walk*(), regardless of how it's implemented internally.

Suppose that we have a BST of character strings and, for whatever reason, want to know the total length of all the strings in it. We could do it like this using *walk*():

§56 ⟨ Summing string lengths with *walk*() 56 ⟩ ≡

```
static void process_node (void *data, void *param) {
    const char *string = data;
    size_t *total = param;

    *total += strlen (string);
}

size_t total_length (struct bst_table *tree) {
    size_t total = 0;
    walk (tree, process_node, &total);
    return total;
}
```

With the functions *first_item*() and *next_item*() that we'll write in this section, we can rewrite these functions as the single function below:

§57 ⟨ Summing string lengths with *next_item*() 57 ⟩ ≡

```
size_t total_length (struct bst_table *tree) {
    struct traverser t;
    const char *string;
    size_t total = 0;

    for (string = first_item (tree, &t); string != NULL; string = next_item (&t))
        total += strlen (string);
    return total;
}
```

You're free to make your own assessment, of course, but many programmers prefer the latter because of its greater brevity and fewer "unsafe" conversions to and from **void** pointers.

Now to actually write the code. Our task is to modify *traverse_iterative*() so that, instead of calling *action*, it returns *node*→*bst_data*. But first, some infrastructure. We define a structure to contain the state of the traversal, equivalent to the relevant argument and local variables in *traverse_iterative*(). To emphasize that this is not our final version of this structure or the related code, we will call it **struct traverser**, without any name prefix:

§58 ⟨ Iterative traversal of BST, take 6 58 ⟩ ≡

```
struct traverser {
    struct bst_table *table; /* Tree being traversed. */
    struct bst_node *node; /* Current node in tree. */
    struct bst_node *stack[BST_MAX_HEIGHT]; /* Parent nodes to revisit. */
    size_t height; /* Number of nodes in stack. */
};
```

See also §59 and §60.

Function *first_item*() just initializes a **struct traverser** and returns the first item in the tree, deferring most of its work to *next_item*():

§59 ⟨ Iterative traversal of BST, take 6 58 ⟩ +≡

```
/* Initializes trav for tree.
```

```
   Returns data item in tree with the smallest value, or NULL if tree is empty.
   In the former case, next_item() may be called with trav
   to retrieve additional data items. */
void *first_item (struct bst_table *tree, struct traverser *trav) {
  assert (tree != NULL && trav != NULL);
  trav->table = tree;
  trav->node = tree->bst_root;
  trav->height = 0;
  return next_item (trav);
}
```

Function *next_item*() is, for the most part, a simple modification of *traverse_iterative*():

§60 ⟨ Iterative traversal of BST, take 6 58 ⟩ +≡

```
/* Returns the next data item in inorder within the tree being traversed with trav,
   or if there are no more data items returns NULL.
   In the former case next_item() may be called again to retrieve the next item. */
void *next_item (struct traverser *trav) {
  struct bst_node *node;

  assert (trav != NULL);
  node = trav->node;
  while (node != NULL) {
    if (trav->height >= BST_MAX_HEIGHT) {
      fprintf (stderr, "tree too deep\n");
      exit (EXIT_FAILURE);
    }
    trav->stack[trav->height++] = node;
    node = node->bst_link[0];
  }
  if (trav->height == 0)
    return NULL;

  node = trav->stack[--trav->height];
  trav->node = node->bst_link[1];
  return node->bst_data;
}
```

See also: [Knuth 1997], algorithm 2.3.1T; [Knuth 1992], p. 50–54, section "Recursion Elimination" within article "Structured Programming with **go to** statements".

Exercises:

1. Make *next_item*() reliable by providing alternate code to execute on stack overflow. This code will work by calling *bst_balance*() to "balance" the tree, reducing its height such that it can be traversed with the small stack that we use. We will develop *bst_balance*() later. For now, consider it a "black box" that simply needs to be invoked with the tree to balance as an argument. Don't forget to adjust the traverser structure so that later calls will work properly, too.

2. Without modifying *next_item*() or *first_item*(), can a function *prev_item*() be written that will move to and return the previous item in the tree in inorder?

4.9.3 Better Iterative Traversal

We have developed an efficient, convenient function for traversing a binary tree. In the exercises, we made it reliable, and it is possible to make it resilient as well. But its algorithm makes it difficult to add generality. In order to do that in a practical way, we will have to use a new algorithm.

Let us start by considering how to understand how to find the successor or predecessor of any node in general, as opposed to just blindly transforming code as we did in the previous section. Back when we wrote *bst_delete*(), we already solved half of the problem, by figuring out how to find the successor of a node that has a right child: take the least-valued node in the right subtree of the node (see [Deletion Case 3], page 40).

The other half is the successor of a node that doesn't have a right child. Take a look at the code for one of the previous traversal functions—recursive or iterative, whichever you better understand—and mentally work out the relationship between the current node and its successor for a node without a right child. What happens is that we move up the tree, from a node to its parent, one node at a time, until it turns out that we moved up to the right (as opposed to up to the left) and that is the successor node. Think of it this way: if we move up to the left, then the node we started at has a lesser value than where we ended up, so we've already visited it, but if we move up to the right, then we're moving to a node with a greater value, so we've found the successor.

Using these instructions, we can find the predecessor of a node, too, just by exchanging "left" and "right". This suggests that all we have to do in order to generalize our traversal function is to keep track of all the nodes above the current node, not just the ones that are up and to the left. This in turn suggests our final implementation of **struct bst_traverser**, with appropriate comments:

§61 ⟨ BST traverser structure 61 ⟩ ≡

```
/* BST traverser structure. */
struct bst_traverser {
    struct bst_table *bst_table; /* Tree being traversed. */
    struct bst_node *bst_node; /* Current node in tree. */
    struct bst_node *bst_stack[BST_MAX_HEIGHT]; /* All the nodes above bst_node. */
    size_t bst_height; /* Number of nodes in bst_parent. */
    unsigned long bst_generation; /* Generation number. */
};
```

This code is included in §24, §142, and §192.

Because user code is expected to declare actual instances of **struct bst_traverser**, **struct bst_traverser** must be defined in ⟨ `bst.h` 24 ⟩ and therefore all of its member names are prefixed by *bst_* for safety.

The only surprise in **struct bst_traverser** is member *bst_generation*, the traverser's generation number. This member is set equal to its namesake in **struct bst_table** when a traverser is initialized. After that, the two values are compared whenever the stack of parent pointers must be accessed. Any change in the tree that could disturb the action of a traverser will cause their generation numbers to differ, which in turn triggers an update to the stack. This is what allows this final implementation to be resilient.

We need a utility function to actually update the stack of parent pointers when differing generation numbers are detected. This is easy to write:

§62 ⟨ BST traverser refresher 62 ⟩ ≡
```
/* Refreshes the stack of parent pointers in trav
   and updates its generation number. */
static void trav_refresh (struct bst_traverser *trav) {
  assert (trav != NULL);

  trav->bst_generation = trav->bst_table->bst_generation;

  if (trav->bst_node != NULL) {
    bst_comparison_func *cmp = trav->bst_table->bst_compare;
    void *param = trav->bst_table->bst_param;
    struct bst_node *node = trav->bst_node;
    struct bst_node *i;

    trav->bst_height = 0;
    for (i = trav->bst_table->bst_root; i != node; ) {
      assert (trav->bst_height < BST_MAX_HEIGHT);
      assert (i != NULL);

      trav->bst_stack[trav->bst_height++] = i;
      i = i->bst_link[cmp (node->bst_data, i->bst_data, param) > 0];
    }
  }
}
```
This code is included in §63 and §178.

The following sections will implement all of the traverser functions *bst_t_**(). See Section 2.10 [Traversers], page 15, for descriptions of the purpose of each of these functions.

The traversal functions are collected together into ⟨ BST traversal functions 63 ⟩:

§63 ⟨ BST traversal functions 63 ⟩ ≡
```
⟨ BST traverser refresher 62 ⟩
⟨ BST traverser null initializer 64 ⟩
⟨ BST traverser least-item initializer 65 ⟩
⟨ BST traverser greatest-item initializer 66 ⟩
⟨ BST traverser search initializer 67 ⟩
⟨ BST traverser insertion initializer 68 ⟩
⟨ BST traverser copy initializer 69 ⟩
⟨ BST traverser advance function 70 ⟩
⟨ BST traverser back up function 73 ⟩
⟨ BST traverser current item function 74 ⟩
⟨ BST traverser replacement function 75 ⟩
```
This code is included in §29.

Exercises:

1. The *bst_probe*() function doesn't change the tree's generation number. Why not?

***2.** The main loop in *trav_refresh*() contains the assertion

```
assert (trav->bst_height < BST_MAX_HEIGHT);
```

Prove that this assertion is always true.

3. In *trav_refresh*(), it is tempting to avoid calls to the user-supplied comparison function by comparing the nodes on the stack to the current state of the tree; e.g., move up the stack, starting from the bottom, and for each node verify that it is a child of the previous one on the stack, falling back to the general algorithm at the first mismatch. Why won't this work?

4.9.3.1 Starting at the Null Node

The *trav_t_init*() function just initializes a traverser to the null item, indicated by a null pointer for *bst_node*.

§64 ⟨ BST traverser null initializer 64 ⟩ ≡

```
void bst_t_init (struct bst_traverser *trav, struct bst_table *tree) {
    trav->bst_table = tree;
    trav->bst_node = NULL;
    trav->bst_height = 0;
    trav->bst_generation = tree->bst_generation;
}
```

This code is included in §63 and §178.

4.9.3.2 Starting at the First Node

To initialize a traverser to start at the least valued node, we simply descend from the root as far down and left as possible, recording the parent pointers on the stack as we go. If the stack overflows, then we balance the tree and start over.

§65 ⟨ BST traverser least-item initializer 65 ⟩ ≡

```
void *bst_t_first (struct bst_traverser *trav, struct bst_table *tree) {
    struct bst_node *x;

    assert (tree != NULL && trav != NULL);

    trav->bst_table = tree;
    trav->bst_height = 0;
    trav->bst_generation = tree->bst_generation;

    x = tree->bst_root;
    if (x != NULL)
        while (x->bst_link[0] != NULL) {
            if (trav->bst_height >= BST_MAX_HEIGHT) {
                bst_balance (tree);
                return bst_t_first (trav, tree);
            }
            trav->bst_stack[trav->bst_height++] = x;
            x = x->bst_link[0];
        }
    trav->bst_node = x;

    return x != NULL ? x->bst_data : NULL;
}
```

This code is included in §63.

Exercises:

***1.** Show that *bst_t_first*() will never make more than one recursive call to itself at a time.

4.9.3.3 Starting at the Last Node

The code to start from the greatest node in the tree is analogous to that for starting from the least node. The only difference is that we descend to the right instead:

§66 ⟨ BST traverser greatest-item initializer 66 ⟩ ≡

```
void *bst_t_last (struct bst_traverser *trav, struct bst_table *tree) {
  struct bst_node *x;
  assert (tree != NULL && trav != NULL);
  trav->bst_table = tree;
  trav->bst_height = 0;
  trav->bst_generation = tree->bst_generation;
  x = tree->bst_root;
  if (x != NULL)
    while (x->bst_link[1] != NULL) {
      if (trav->bst_height >= BST_MAX_HEIGHT) {
        bst_balance (tree);
        return bst_t_last (trav, tree);
      }
      trav->bst_stack[trav->bst_height++] = x;
      x = x->bst_link[1];
    }
  trav->bst_node = x;
  return x != NULL ? x->bst_data : NULL;
}
```

This code is included in §63.

4.9.3.4 Starting at a Found Node

Sometimes it is convenient to begin a traversal at a particular item in a tree. This function works in the same was as *bst_find*(), but records parent pointers in the traverser structure as it descends the tree.

§67 ⟨ BST traverser search initializer 67 ⟩ ≡

```
void *bst_t_find (struct bst_traverser *trav, struct bst_table *tree, void *item) {
  struct bst_node *p, *q;
  assert (trav != NULL && tree != NULL && item != NULL);
  trav->bst_table = tree;
  trav->bst_height = 0;
  trav->bst_generation = tree->bst_generation;
  for (p = tree->bst_root; p != NULL; p = q) {
    int cmp = tree->bst_compare (item, p->bst_data, tree->bst_param);
    if (cmp < 0) q = p->bst_link[0];
    else if (cmp > 0) q = p->bst_link[1];
```

```
        else /* cmp == 0 */ {
            trav→bst_node = p;
            return p→bst_data;
        }
        if (trav→bst_height >= BST_MAX_HEIGHT) {
            bst_balance (trav→bst_table);
            return bst_t_find (trav, tree, item);
        }
        trav→bst_stack[trav→bst_height++] = p;
    }
    trav→bst_height = 0;
    trav→bst_node = NULL;
    return NULL;
}
```

This code is included in §63.

4.9.3.5 Starting at an Inserted Node

Another operation that can be useful is to insert a new node and construct a traverser to the inserted node in a single operation. The following code does this:

§68 ⟨ BST traverser insertion initializer 68 ⟩ ≡

```
void *bst_t_insert (struct bst_traverser *trav, struct bst_table *tree, void *item) {
    struct bst_node **q;
    assert (tree != NULL && item != NULL);
    trav→bst_table = tree;
    trav→bst_height = 0;
    q = &tree→bst_root;
    while (*q != NULL) {
        int cmp = tree→bst_compare (item, (*q)→bst_data, tree→bst_param);
        if (cmp == 0) {
            trav→bst_node = *q;
            trav→bst_generation = tree→bst_generation;
            return (*q)→bst_data;
        }
        if (trav→bst_height >= BST_MAX_HEIGHT) {
            bst_balance (tree);
            return bst_t_insert (trav, tree, item);
        }
        trav→bst_stack[trav→bst_height++] = *q;
        q = &(*q)→bst_link[cmp > 0];
    }
    trav→bst_node = *q = tree→bst_alloc→libavl_malloc (tree→bst_alloc, sizeof **q);
    if (*q == NULL) {
        trav→bst_node = NULL;
        trav→bst_generation = tree→bst_generation;
```

```
        return NULL;
    }
    (*q)->bst_link[0] = (*q)->bst_link[1] = NULL;
    (*q)->bst_data = item;
    tree->bst_count++;
    trav->bst_generation = tree->bst_generation;
    return (*q)->bst_data;
}
```

This code is included in §63.

4.9.3.6 Initialization by Copying

This function copies one traverser to another. It only copies the stack of parent pointers if they are up-to-date:

§69 ⟨ BST traverser copy initializer 69 ⟩ ≡

```
void *bst_t_copy (struct bst_traverser *trav, const struct bst_traverser *src) {
    assert (trav != NULL && src != NULL);
    if (trav != src) {
        trav->bst_table = src->bst_table;
        trav->bst_node = src->bst_node;
        trav->bst_generation = src->bst_generation;
        if (trav->bst_generation == trav->bst_table->bst_generation) {
            trav->bst_height = src->bst_height;
            memcpy (trav->bst_stack, (const void *) src->bst_stack,
                    sizeof *trav->bst_stack * trav->bst_height);
        }
    }
    return trav->bst_node != NULL ? trav->bst_node->bst_data : NULL;
}
```

This code is included in §63 and §178.

Exercises:

1. Without the check that *trav* != *src* before copying *src* into *trav*, what might happen?

4.9.3.7 Advancing to the Next Node

The algorithm of *bst_t_next*(), the function for finding a successor, divides neatly into three cases. Two of these are the ones that we discussed earlier in the introduction to this kind of traverser (see Section 4.9.3 [Better Iterative Traversal], page 53). The third case occurs when the last node returned was NULL, in which case we return the least node in the table, in accordance with the semantics for LIBAVL tables. The function outline is this:

§70 ⟨ BST traverser advance function 70 ⟩ ≡

```
void *bst_t_next (struct bst_traverser *trav) {
    struct bst_node *x;
    assert (trav != NULL);
```

```
    if (trav→bst_generation != trav→bst_table→bst_generation)
        trav_refresh (trav);
    x = trav→bst_node;
    if (x == NULL) {
        return bst_t_first (trav, trav→bst_table);
    } else if (x→bst_link[1] != NULL) {
        ⟨ Handle case where x has a right child 71 ⟩
    } else {
        ⟨ Handle case where x has no right child 72 ⟩
    }
    trav→bst_node = x;
    return x→bst_data;
}
```

This code is included in §63.

The case where the current node has a right child is accomplished by stepping to the right, then to the left until we can't go any farther, as discussed in detail earlier. The only difference is that we must check for stack overflow. When stack overflow does occur, we recover by calling *trav_balance*(), then restarting *bst_t_next*() using a tail-recursive call. The tail recursion will never happen more than once, because *trav_balance*() ensures that the tree's height is small enough that the stack cannot overflow again:

§71 ⟨ Handle case where x has a right child 71 ⟩ ≡

```
if (trav→bst_height >= BST_MAX_HEIGHT) {
    bst_balance (trav→bst_table);
    return bst_t_next (trav);
}
trav→bst_stack[trav→bst_height++] = x;
x = x→bst_link[1];
while (x→bst_link[0] != NULL) {
    if (trav→bst_height >= BST_MAX_HEIGHT) {
        bst_balance (trav→bst_table);
        return bst_t_next (trav);
    }
    trav→bst_stack[trav→bst_height++] = x;
    x = x→bst_link[0];
}
```

This code is included in §70.

In the case where the current node has no right child, we move upward in the tree based on the stack of parent pointers that we saved, as described before. When the stack underflows, we know that we've run out of nodes in the tree:

§72 ⟨ Handle case where x has no right child 72 ⟩ ≡

```
struct bst_node *y;
do {
    if (trav→bst_height == 0) {
        trav→bst_node = NULL;
```

```
        return NULL;
    }
    y = x;
    x = trav->bst_stack[--trav->bst_height];
} while (y == x->bst_link[1]);
```

This code is included in §70.

4.9.3.8 Backing Up to the Previous Node

Moving to the previous node is analogous to moving to the next node. The only difference, in fact, is that directions are reversed from left to right.

§73 ⟨BST traverser back up function 73⟩ ≡

```
void *bst_t_prev (struct bst_traverser *trav) {
    struct bst_node *x;
    assert (trav != NULL);
    if (trav->bst_generation != trav->bst_table->bst_generation)
        trav_refresh (trav);
    x = trav->bst_node;
    if (x == NULL) {
        return bst_t_last (trav, trav->bst_table);
    } else if (x->bst_link[0] != NULL) {
        if (trav->bst_height >= BST_MAX_HEIGHT) {
            bst_balance (trav->bst_table);
            return bst_t_prev (trav);
        }
        trav->bst_stack[trav->bst_height++] = x;
        x = x->bst_link[0];
        while (x->bst_link[1] != NULL) {
            if (trav->bst_height >= BST_MAX_HEIGHT) {
                bst_balance (trav->bst_table);
                return bst_t_prev (trav);
            }
            trav->bst_stack[trav->bst_height++] = x;
            x = x->bst_link[1];
        }
    } else {
        struct bst_node *y;
        do {
            if (trav->bst_height == 0) {
                trav->bst_node = NULL;
                return NULL;
            }
            y = x;
            x = trav->bst_stack[--trav->bst_height];
        } while (y == x->bst_link[0]);
```

```
    }
    trav→bst_node = x;
    return x→bst_data;
}
```

This code is included in §63.

4.9.3.9 Getting the Current Item

§74 ⟨ BST traverser current item function 74 ⟩ ≡

```
void *bst_t_cur (struct bst_traverser *trav) {
    assert (trav != NULL);
    return trav→bst_node != NULL ? trav→bst_node→bst_data : NULL;
}
```

This code is included in §63, §178, §268, §395, §502, and §546.

4.9.3.10 Replacing the Current Item

§75 ⟨ BST traverser replacement function 75 ⟩ ≡

```
void *bst_t_replace (struct bst_traverser *trav, void *new) {
    void *old;
    assert (trav != NULL && trav→bst_node != NULL && new != NULL);
    old = trav→bst_node→bst_data;
    trav→bst_node→bst_data = new;
    return old;
}
```

This code is included in §63, §178, §268, §395, §502, and §546.

4.10 Copying

In this section, we're going to write function *bst_copy*() to make a copy of a binary tree. This is the most complicated function of all those needed for BST functionality, so pay careful attention as we proceed.

4.10.1 Recursive Copying

The "obvious" way to copy a binary tree is recursive. Here's a basic recursive copy, hard-wired to allocate memory with *malloc*() for simplicity:

§76 ⟨ Recursive copy of BST, take 1 76 ⟩ ≡

```
/* Makes and returns a new copy of tree rooted at x. */
static struct bst_node *bst_copy_recursive_1 (struct bst_node *x) {
    struct bst_node *y;
    if (x == NULL)
        return NULL;
    y = malloc (sizeof *y);
    if (y == NULL)
```

```
        return NULL;
    y->bst_data = x->bst_data;
    y->bst_link[0] = bst_copy_recursive_1 (x->bst_link[0]);
    y->bst_link[1] = bst_copy_recursive_1 (x->bst_link[1]);
    return y;
}
```

But, again, it would be nice to rewrite this iteratively, both because the iterative version is likely to be faster and for the sheer mental exercise of it. Recall, from our earlier discussion of inorder traversal, that tail recursion (recursion where a function calls itself as its last action) is easier to convert to iteration than other types. Unfortunately, neither of the recursive calls above are tail-recursive.

Fortunately, we can rewrite it so that it is, if we change the way we allocate data:

§77 ⟨ Recursive copy of BST, take 2 77 ⟩ ≡

```
/* Copies tree rooted at x to y, which latter is allocated but not yet initialized. */
static void bst_copy_recursive_2 (struct bst_node *x, struct bst_node *y) {
    y->bst_data = x->bst_data;
    if (x->bst_link[0] != NULL) {
        y->bst_link[0] = malloc (sizeof *y->bst_link[0]);
        bst_copy_recursive_2 (x->bst_link[0], y->bst_link[0]);
    }
    else y->bst_link[0] = NULL;
    if (x->bst_link[1] != NULL) {
        y->bst_link[1] = malloc (sizeof *y->bst_link[1]);
        bst_copy_recursive_2 (x->bst_link[1], y->bst_link[1]);
    }
    else y->bst_link[1] = NULL;
}
```

Exercises:

1. When *malloc*() returns a null pointer, *bst_copy_recursive_1*() fails "silently", that is, without notifying its caller about the error, and the output is a partial copy of the original tree. Without removing the recursion, implement two different ways to propagate such errors upward to the function's caller:

a. Change the function's prototype to:

   static int *bst_robust_copy_recursive_1* (**struct bst_node** *, **struct bst_node** **);

b. Without changing the function's prototype. (Hint: use a **static**ally declared **struct bst_node**).

In each case make sure that any allocated memory is safely freed if an allocation error occurs.

2. *bst_copy_recursive_2*() is even worse than *bst_copy_recursive_1*() at handling allocation failure. It actually invokes undefined behavior when an allocation fails. Fix this, changing it to return an **int**, with nonzero return values indicating success. Be careful not to leak memory.

4.10.2 Iterative Copying

Now we can factor out the recursion, starting with the tail recursion. This process is very similar to what we did with the traversal code, so the details are left for Exercise 1. Let's look at the results part by part:

§78 ⟨ Iterative copy of BST 78 ⟩ ≡
```
/* Copies org to a newly created tree, which is returned. */
struct bst_table *bst_copy_iterative (const struct bst_table *org) {
  struct bst_node *stack[2 * (BST_MAX_HEIGHT + 1)]; /* Stack. */
  int height = 0; /* Stack height. */
```
See also §79, §80, and§81.

This time, our stack will have two pointers added to it at a time, one from the original tree and one from the copy. Thus, the stack needs to be twice as big. In addition, we'll see below that there'll be an extra item on the stack representing the pointer to the tree's root, so our stack needs room for an extra pair of items, which is the reason for the "+ 1" in *stack*[]'s size.

§79 ⟨ Iterative copy of BST 78 ⟩ +≡
```
  struct bst_table *new; /* New tree. */
  const struct bst_node *x; /* Node currently being copied. */
  struct bst_node *y; /* New node being copied from x. */

  new = bst_create (org->bst_compare, org->bst_param, org->bst_alloc);
  new->bst_count = org->bst_count;
  if (new->bst_count == 0)
    return new;

  x = (const struct bst_node *) &org->bst_root;
  y = (struct bst_node *) &new->bst_root;
```
This is the same kind of "dirty trick" already described in Exercise 4.7-1.

§80 ⟨ Iterative copy of BST 78 ⟩ +≡
```
  for (;;) {
    while (x->bst_link[0] != NULL) {
      y->bst_link[0] = org->bst_alloc->libavl_malloc (org->bst_alloc,
                          sizeof *y->bst_link[0]);
      stack[height++] = (struct bst_node *) x;
      stack[height++] = y;
      x = x->bst_link[0];
      y = y->bst_link[0];
    }
    y->bst_link[0] = NULL;
```
This code moves x down and to the left in the tree until it runs out of nodes, allocating space in the new tree for left children and pushing nodes from the original tree and the copy onto the stack as it goes. The cast on x suppresses a warning or error due to x, a pointer to a **const** structure, being stored into a non-constant pointer in *stack*[]. We won't ever try to store into the pointer that we store in there, so this is legitimate.

We've switched from using *malloc*() to using the allocation function provided by the user. This is easy now because we have the tree structure to work with. To do this earlier, we would have had to somehow pass the tree structure to each recursive call of the copy function, wasting time and space.

§81 ⟨Iterative copy of BST 78⟩ +≡

```
        for (;;) {
            y->bst_data = x->bst_data;
            if (x->bst_link[1] != NULL) {
                y->bst_link[1] = org->bst_alloc->libavl_malloc (org->bst_alloc,
                                    sizeof *y->bst_link[1]);
                x = x->bst_link[1];
                y = y->bst_link[1];
                break;
            }
            else y->bst_link[1] = NULL;
            if (height <= 2)
                return new;
            y = stack[--height];
            x = stack[--height];
        }
    }
}
```

We do not pop the bottommost pair of items off the stack because these items contain the fake **struct bst_node** pointer that is actually the address of *bst_root*. When we get down to these items, we're done copying and can return the new tree.

See also: [Knuth 1997], algorithm 2.3.1C; [ISO 1990], section 6.5.2.1.

Exercises:

1. Suggest a step between *bst_copy_recursive_2*() and *bst_copy_iterative*().

4.10.3 Error Handling

So far, outside the exercises, we've ignored the question of handling memory allocation errors during copying. In our other routines, we've been careful to implement to handle allocation failures by cleaning up and returning an error indication to the caller. Now we will apply this same policy to tree copying, as Libavl semantics require (see Section 2.6 [Creation and Destruction], page 12): a memory allocation error causes the partially copied tree to be destroyed and returns a null pointer to the caller.

This is a little harder to do than recovering after a single operation, because there are potentially many nodes that have to be freed, and each node might include additional user data that also has to be freed. The new BST might have as-yet-uninitialized pointer fields as well, and we must be careful to avoid reading from these fields as we destroy the tree.

We could use a number of strategies to destroy the partially copied tree while avoiding uninitialized pointers. The strategy that we will actually use is to initialize these pointers to NULL, then call the general tree destruction routine *bst_destroy*(). We haven't yet written *bst_destroy*(), so for now we'll treat it as a **black box** that does what we want, even if we don't understand how.

Next question: *which* pointers in the tree are not initialized? The answer is simple: during the copy, we will not revisit nodes not currently on the stack, so only pointers in the current node (y) and on the stack can be uninitialized. For its part, depending on what we're doing to it, y might not have any of its fields initialized. As for the stack, nodes are pushed onto it because we have to come back later and build their right subtrees, so we must set their right child pointers to `NULL`.

We will need this error recovery code in a number of places, so it is worth making it into a small helper function:

§82 ⟨BST copy error helper function 82⟩ ≡

```
static void copy_error_recovery (struct bst_node **stack, int height,
                                 struct bst_table *new, bst_item_func *destroy) {
  assert (stack != NULL && height >= 0 && new != NULL);
  for (; height > 2; height -= 2)
    stack[height - 1]->bst_link[1] = NULL;
  bst_destroy (new, destroy);
}
```

This code is included in §83 and §185.

Another problem that can arise in copying a binary tree is stack overflow. We will handle stack overflow by destroying the partial copy, balancing the original tree, and then restarting the copy. The balanced tree is guaranteed to have small enough height that it will not overflow the stack.

The code below for our final version of *bst_copy*() takes three new parameters: two function pointers and a memory allocator. The meaning of these parameters was explained earlier (see Section 2.6 [Creation and Destruction], page 12). Their use within the function should be self-explanatory.

§83 ⟨BST copy function 83⟩ ≡

```
⟨BST copy error helper function 82⟩
struct bst_table *bst_copy (const struct bst_table *org, bst_copy_func *copy,
                            bst_item_func *destroy, struct libavl_allocator *allocator) {
  struct bst_node *stack[2 * (BST_MAX_HEIGHT + 1)];
  int height = 0;

  struct bst_table *new;
  const struct bst_node *x;
  struct bst_node *y;

  assert (org != NULL);
  new = bst_create (org->bst_compare, org->bst_param,
                    allocator != NULL ? allocator : org->bst_alloc);
  if (new == NULL)
    return NULL;
  new->bst_count = org->bst_count;
  if (new->bst_count == 0)
    return new;

  x = (const struct bst_node *) &org->bst_root;
  y = (struct bst_node *) &new->bst_root;
```

```
for (;;) {
  while (x->bst_link[0] != NULL) {
    if (height >= 2 * (BST_MAX_HEIGHT + 1)) {
      y->bst_data = NULL;
      y->bst_link[0] = y->bst_link[1] = NULL;
      copy_error_recovery (stack, height, new, destroy);

      bst_balance ((struct bst_table *) org);
      return bst_copy (org, copy, destroy, allocator);
    }
    y->bst_link[0] = new->bst_alloc->libavl_malloc (new->bst_alloc,
                         sizeof *y->bst_link[0]);
    if (y->bst_link[0] == NULL) {
      if (y != (struct bst_node *) &new->bst_root) {
        y->bst_data = NULL;
        y->bst_link[1] = NULL;
      }
      copy_error_recovery (stack, height, new, destroy);
      return NULL;
    }
    stack[height++] = (struct bst_node *) x;
    stack[height++] = y;
    x = x->bst_link[0];
    y = y->bst_link[0];
  }
  y->bst_link[0] = NULL;
  for (;;) {
    if (copy == NULL)
      y->bst_data = x->bst_data;
    else {
      y->bst_data = copy (x->bst_data, org->bst_param);
      if (y->bst_data == NULL) {
        y->bst_link[1] = NULL;
        copy_error_recovery (stack, height, new, destroy);
        return NULL;
      }
    }
    if (x->bst_link[1] != NULL) {
      y->bst_link[1] = new->bst_alloc->libavl_malloc (new->bst_alloc,
                           sizeof *y->bst_link[1]);
      if (y->bst_link[1] == NULL) {
        copy_error_recovery (stack, height, new, destroy);
        return NULL;
      }
      x = x->bst_link[1];
      y = y->bst_link[1];
```

```
                break;
            }
            else y→bst_link[1] = NULL;
            if (height <= 2)
                return new;
            y = stack[−−height];
            x = stack[−−height];
        }
    }
}
```

This code is included in §29.

4.11 Destruction

Eventually, we'll want to get rid of the trees we've spent all this time constructing. When this happens, it's time to destroy them by freeing their memory.

4.11.1 Destruction by Rotation

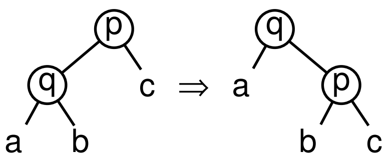
The method actually used in LIBAVL for destruction of binary trees is somewhat novel. This section will cover this method. Later sections will cover more conventional techniques using recursive or iterative **postorder traversal**.

To destroy a binary tree, we must visit and free each node. We have already covered one way to traverse a tree (inorder traversal) and used this technique for traversing and copying a binary tree. But, both times before, we were subject to both the explicit constraint that we had to visit the nodes in sorted order and the implicit constraint that we were not to change the structure of the tree, or at least not to change it for the worse.

Neither of these constraints holds for destruction of a binary tree. As long as the tree finally ends up freed, it doesn't matter how much it is mangled in the process. In this case, "the end justifies the means" and we are free to do it however we like.

So let's consider why we needed a stack before. It was to keep track of nodes whose left subtree we were currently visiting, in order to go back later and visit them and their right subtrees. Hmm. . . what if we rearranged nodes so that they *didn't have* any left subtrees? Then we could just descend to the right, without need to keep track of anything on a stack.

We can do this. For the case where the current node p has a left child q, consider the transformation below where we rotate right at p:



where a, b, and c are arbitrary subtrees or even empty trees. This transformation shifts nodes from the left to the right side of the root (which is now q). If it is performed enough times, the root node will no longer have a left child. After the transformation, q becomes the current node.

For the case where the current node has no left child, we can just destroy the current node and descend to its right. Because the transformation used does not change the tree's ordering, we end up destroying nodes in inorder. It is instructive to verify this by simulating with paper and pencil the destruction of a few trees this way.

The code to implement destruction in this manner is brief and straightforward:

§84 ⟨ BST destruction function 84 ⟩ ≡
```
void bst_destroy (struct bst_table *tree, bst_item_func *destroy) {
  struct bst_node *p, *q;

  assert (tree != NULL);

  for (p = tree->bst_root; p != NULL; p = q)
    if (p->bst_link[0] == NULL) {
      q = p->bst_link[1];
      if (destroy != NULL && p->bst_data != NULL)
        destroy (p->bst_data, tree->bst_param);
      tree->bst_alloc->libavl_free (tree->bst_alloc, p);
    } else {
      q = p->bst_link[0];
      p->bst_link[0] = q->bst_link[1];
      q->bst_link[1] = p;
    }

  tree->bst_alloc->libavl_free (tree->bst_alloc, tree);
}
```
This code is included in §29, §145, §196, §489, §522, and §554.

See also: [Stout 1986], *tree_to_vine* procedure.

Exercises:

1. Before calling *destroy*() above, we first test that we are not passing it a NULL pointer, because we do not want *destroy*() to have to deal with this case. How can such a pointer get into the tree in the first place, since *bst_probe*() refuses to insert such a pointer into a tree?

4.11.2 Aside: Recursive Destruction

The algorithm used in the previous section is easy and fast, but it is not the most common method for destroying a tree. The usual way is to perform a traversal of the tree, in much the same way we did for tree traversal and copying. Once again, we'll start from a recursive implementation, because these are so easy to write. The only tricky part is that subtrees have to be freed *before* the root. This code is hard-wired to use *free*() for simplicity:

§85 ⟨ Destroy a BST recursively 85 ⟩ ≡
```
static void bst_destroy_recursive (struct bst_node *node) {
  if (node == NULL)
    return;

  bst_destroy_recursive (node->bst_link[0]);
  bst_destroy_recursive (node->bst_link[1]);
```

```
    free (node);
}
```

4.11.3 Aside: Iterative Destruction

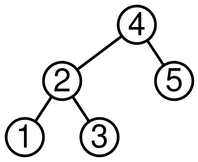
As we've done before for other algorithms, we can factor the recursive destruction algorithm into an equivalent iteration. In this case, neither recursive call is tail recursive, and we can't easily modify the code so that it is. We could still factor out the recursion by our usual methods, although it would be more difficult, but this problem is simple enough to figure out from first principles. Let's do it that way, instead, this time.

The idea is that, for the tree's root, we traverse its left subtree, then its right subtree, then free the root. This pattern is called a **postorder traversal**.

Let's think about how much state we need to keep track of. When we're traversing the root's left subtree, we still need to remember the root, in order to come back to it later. The same is true while traversing the root's right subtree, because we still need to come back to free the root. What's more, we need to keep track of what state we're in: have we traversed the root's left subtree or not, have we traversed the root's right subtree or not?

This naturally suggests a stack that holds two-part items (*root*, *state*), where *root* is the root of the tree or subtree and *state* is the state of the traversal at that node. We start by selecting the tree's root as our current node p, then pushing (p, 0) onto the stack and moving down to the left as far as we can, pushing as we go. Then we start popping off the stack into (p, *state*) and notice that *state* is 0, which tells us that we've traversed p's left subtree but not its right. So, we push (p, 1) back onto the stack, then we traverse p's right subtree. When, later, we pop off that same node back off the stack, the 1 tells us that we've already traversed both subtrees, so we free the node and keep popping. The pattern follows as we continue back up the tree.

That sounds pretty complicated, so let's work through an example to help clarify. Consider this binary search tree:



Abstractly speaking, we start with 4 as p and an empty stack. First, we work our way down the left-child pointers, pushing onto the stack as we go. We push (4, 0), then (2, 0), then (1, 0), and then p is `NULL` and we've fallen off the bottom of the tree. We pop the top item off the stack into (p, *state*), getting (1, 0). Noticing that we have 0 for *state*, we push (1, 1) on the stack and traverse 1's right subtree, but it is empty so there is nothing to do. We pop again and notice that *state* is 1, meaning that we've fully traversed 1's subtrees, so we free node 1. We pop again, getting 2 for p and 0 for *state*. Because *state* is 0, we push (2, 1) and traverse 2's right subtree, which means that we push (3, 0). We traverse 3's null right subtree (again, it is empty so there is nothing to do), pushing and popping (3, 1), then free node 3, then move back up to 2. Because we've traversed 2's right subtree, *state* is 1 and p is 2, and we free node 2. You should be able to figure out how 4 and 5 get freed.

A straightforward implementation of this approach looks like this:

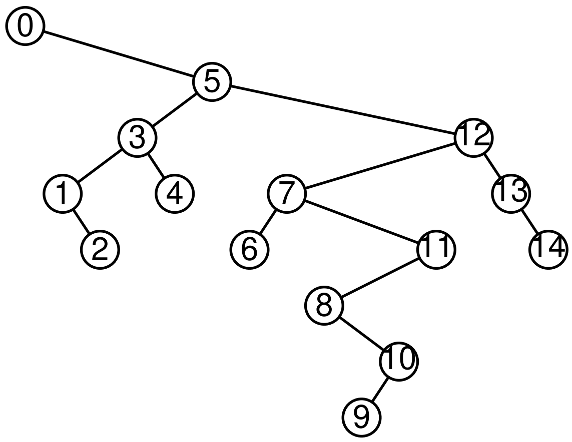
§86 ⟨ Destroy a BST iteratively 86 ⟩ ≡

```
void bst_destroy (struct bst_table *tree, bst_item_func *destroy) {
  struct bst_node *stack[BST_MAX_HEIGHT];
  unsigned char state[BST_MAX_HEIGHT];
  int height = 0;

  struct bst_node *p;

  assert (tree != NULL);
  p = tree->bst_root;
  for (;;) {
    while (p != NULL) {
      if (height >= BST_MAX_HEIGHT) {
        fprintf (stderr, "tree too deep\n");
        exit (EXIT_FAILURE);
      }
      stack[height] = p;
      state[height] = 0;
      height++;

      p = p->bst_link[0];
    }

    for (;;) {
      if (height == 0) {
        tree->bst_alloc->libavl_free (tree->bst_alloc, tree);
        return;
      }

      height--;
      p = stack[height];
      if (state[height] == 0) {
        state[height++] = 1;
        p = p->bst_link[1];
        break;
      } else {
        if (destroy != NULL && p->bst_data != NULL)
          destroy (p->bst_data, tree->bst_param);
        tree->bst_alloc->libavl_free (tree->bst_alloc, p);
      }
    }
  }
}
```

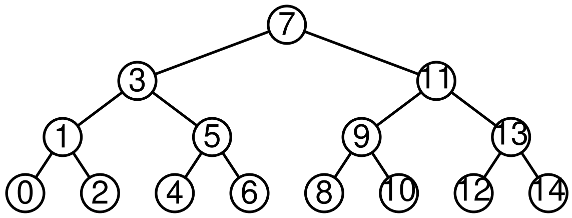
See also: [Knuth 1997], exercise 13 in section 2.3.1.

4.12 Balance

Sometimes binary trees can grow to become much taller than their optimum height. For example, the following binary tree was one of the tallest from a sample of 100 15-node trees built by inserting nodes in random order:



The average number of comparisons required to find a random node in this tree is $(1 + 2 + (3\times 2) + (4\times 4) + (5\times 4) + 6 + 7 + 8)/15 = 4.4$ comparisons. In contrast, the corresponding optimal binary tree, shown below, requires only $(1 + (2 \times 2) + (3 \times 4) + (4 \times 8))/15 = 3.3$ comparisons, on average. Moreover, the optimal tree requires a maximum of 4, as opposed to 8, comparisons for any search:



Besides this inefficiency in time, trees that grow too tall can cause inefficiency in space, leading to an overflow of the stack in *bst_t_next*(), *bst_copy*(), or other functions. For both reasons, it is helpful to have a routine to rearrange a tree to its minimum possible height, that is, to **balance** the tree.

The algorithm we will use for balancing proceeds in two stages. In the first stage, the binary tree is “flattened” into a pathological, linear binary tree, called a “vine.” In the second stage, binary tree structure is restored by repeatedly “compressing” the vine into a minimal-height binary tree.

Here’s a top-level view of the balancing function:

§87 ⟨ BST balance function 87 ⟩ ≡

```
⟨ BST to vine function 89 ⟩
⟨ Vine to balanced BST function 90 ⟩
void bst_balance (struct bst_table *tree) {
    assert (tree != NULL);
    tree_to_vine (tree);
    vine_to_tree (tree);
    tree→bst_generation++;
}
```

This code is included in §29.

§88 ⟨ BST extra function prototypes 88 ⟩ ≡

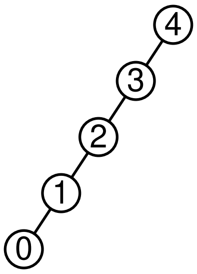
```
/* Special BST functions. */
void bst_balance (struct bst_table *tree);
```

This code is included in §24, §247, §372, and §486.

See also: [Stout 1986], *rebalance* procedure.

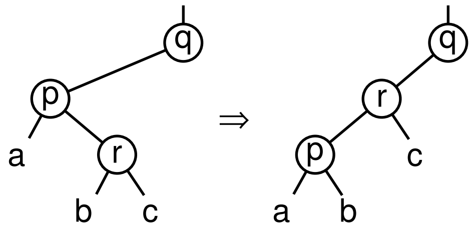
4.12.1 From Tree to Vine

The first stage of balancing converts a binary tree into a linear structure resembling a linked list, called a **vine**. The vines we will create have the greatest value in the binary tree at the root and decrease descending to the left. Any binary search tree that contains a particular set of values, no matter its shape, corresponds to the same vine of this type. For instance, all binary search trees of the integers 0...4 will be transformed into the following vine:



The method for transforming a tree into a vine of this type is similar to that used for destroying a tree by rotation (see Section 4.11.1 [Destroying a BST by Rotation], page 67). We step pointer p through the tree, starting at the root of the tree, maintaining pointer q as p's parent. (Because we're building a vine, p is always the left child of q.) At each step, we do one of two things:

- If p has no right child, then this part of the tree is already the shape we want it to be. We step p and q down to the left and continue.
- If p has a right child r, then we rotate left at p, performing the following transformation:



where a, b, and c are arbitrary subtrees or empty trees. Node r then becomes the new p. If c is an empty tree, then, in the next step, we will continue down the tree. Otherwise, the right subtree of p is smaller (contains fewer nodes) than previously, so we're on the right track.

This is all it takes:

§89 ⟨ BST to vine function 89 ⟩ ≡

```
/* Converts tree into a vine. */
static void tree_to_vine (struct bst_table *tree) {
    struct bst_node *q, *p;
    q = (struct bst_node *) &tree->bst_root;
    p = tree->bst_root;
```

```
    while (p != NULL)
        if (p→bst_link[1] == NULL) {
            q = p;
            p = p→bst_link[0];
        }
        else {
            struct bst_node *r = p→bst_link[1];
            p→bst_link[1] = r→bst_link[0];
            r→bst_link[0] = p;
            p = r;
            q→bst_link[0] = r;
        }
}
```

This code is included in §87, §511, and §679.

See also: [Stout 1986], *tree_to_vine* procedure.

4.12.2 From Vine to Balanced Tree

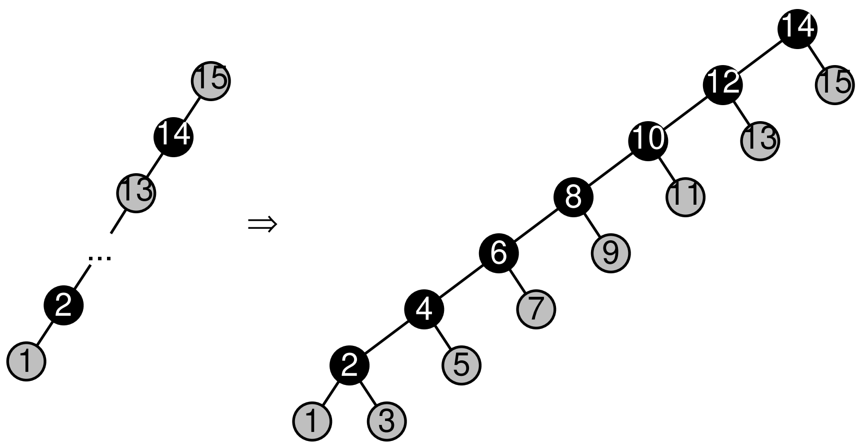
Converting the vine, once we have it, into a balanced tree is the interesting and clever part of the balancing operation. However, at first it may be somewhat less than obvious how this is actually done. We will tackle the subject by presenting an example, then the generalized form.

Suppose we have a vine, as above, with $2^n - 1$ nodes for positive integer n. For the sake of example, take $n = 4$, corresponding to a tree with 15 nodes. We convert this vine into a balanced tree by performing three successive **compression** operations.

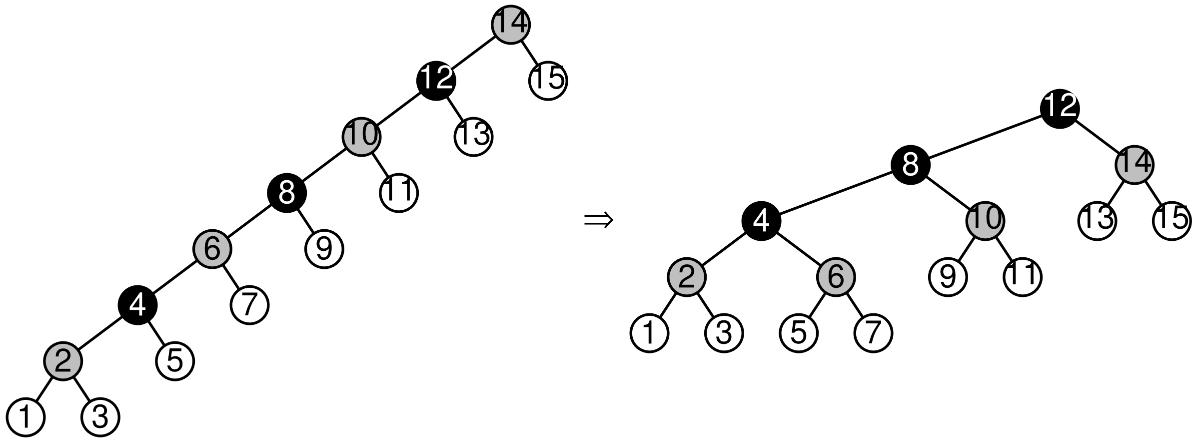
To perform the first compression, move down the vine, starting at the root. Conceptually assign each node a "color", alternating between red and black and starting with red at the root.[3] Then, take each red node, except the bottommost, and remove it from the vine, making it the child of its black former child node.

After this transformation, we have something that looks a little more like a tree. Instead of a 15-node vine, we have a 7-node black vine with a 7-node red vine as its right children and a single red node as its left child. Graphically, this first compression step on a 15-node vine looks like this:

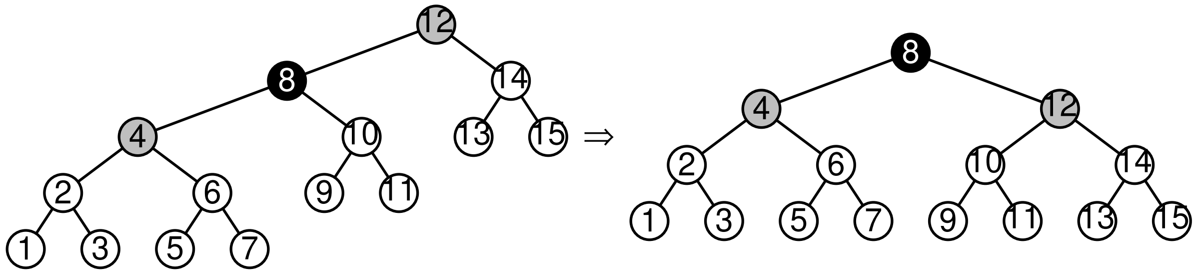
[3] These colors are for the purpose of illustration only. They are not stored in the nodes and are not related to those used in a **red-black tree**.



To perform the second compression, recolor all the red nodes to white, then change the color of alternate black nodes to red, starting at the root. As before, extract each red node, except the bottommost, and reattach it as the child of its black former child node. Attach each black node's right subtree as the left subtree of the corresponding red node. Thus, we have the following:



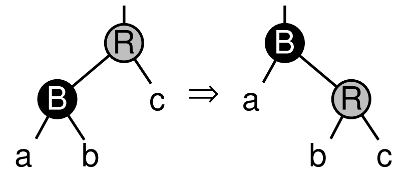
The third compression is the same as the first two. Nodes 12 and 4 are recolored red, then node 12 is removed and reattached as the right child of its black former child node 8, receiving node 8's right subtree as its left subtree:



The result is a fully balanced tree.

4.12.2.1 General Trees

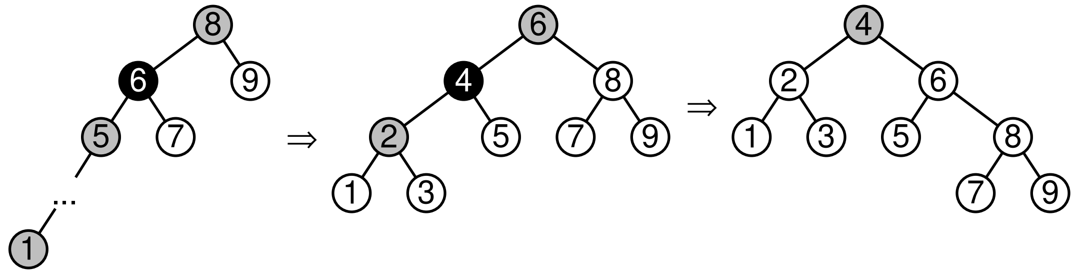
A compression is the repeated application of a right rotation, called in this context a "compression transformation", once for each black node, like so:



So far, all of the compressions we've performed have involved all $2^k - 1$ nodes composing the "main vine." This works out well for an initial vine of exactly $2^n - 1$ nodes. In this case, a total of $n-1$ compressions are required, where for successive compressions $k = n, n-1, \ldots, 2$.

For trees that do not have exactly one fewer than a power of two nodes, we need to begin with a compression that does not involve all of the nodes in the vine. Suppose that our vine has m nodes, where $2^n - 1 < m < 2^{n+1} - 1$ for some value of n. Then, by applying the compression transformation shown above $m - (2^n - 1)$ times, we reduce the length of the main vine to exactly $2^n - 1$ nodes. After that, we can treat the problem in the same way as the former case. The result is a balanced tree with n full levels of nodes, and a bottom level containing $m - (2^n - 1)$ nodes and $(2^{n+1} - 1) - m$ vacancies.

An example is indicated. Suppose that the vine contains $m \equiv 9$ nodes numbered from 1 to 9. Then $n \equiv 3$ since we have $2^3 - 1 \equiv 7 < 9 < 15 \equiv 2^4 - 1$, and we must perform the compression transformation shown above $9 - (2^3 - 1) \equiv 2$ times initially, reducing the main vine's length to 7 nodes. Afterward, we treat the problem the same way as for a tree that started off with only 7 nodes, performing one compression with $k \equiv 3$ and one with $k \equiv 2$. The entire sequence, omitting the initial vine, looks like this:



Now we have a general technique that can be applied to a vine of any size.

4.12.2.2 Implementation

Implementing this algorithm is more or less straightforward. Let's start from an outline:

§90 ⟨ Vine to balanced BST function 90 ⟩ ≡
⟨ BST compression function 95 ⟩

/∗ Converts *tree*, which must be in the shape of a vine, into a balanced tree. ∗/
static void *vine_to_tree* (**struct bst_table** ∗*tree*) {
    unsigned long *vine*; /∗ Number of nodes in main vine. ∗/
    unsigned long *leaves*; /∗ Nodes in incomplete bottom level, if any. ∗/
    int *height*; /∗ Height of produced balanced tree. ∗/

    ⟨ Calculate *leaves* 91 ⟩
    ⟨ Reduce vine general case to special case 92 ⟩
    ⟨ Make special case vine into balanced tree and count height 93 ⟩
    ⟨ Check for tree height in range 94 ⟩
}

This code is included in §87.

The first step is to calculate the number of compression transformations necessary to reduce the general case of a tree with m nodes to the special case of exactly $2^n - 1$ nodes, i.e., calculate $m - (2^n - 1)$, and store it in variable *leaves*. We are given only the value of m, as *tree→bst_count*. Rewriting the calculation as the equivalent $m + 1 - 2^n$, one way to calculate it is evident from looking at the pattern in binary:

m	n	$m+1$	2^n	$m+1-2^n$
1	1	$2 = 00010_2$	$2 = 00010_2$	$0 = 00000_2$
2	1	$3 = 00011_2$	$2 = 00010_2$	$1 = 00001_2$
3	2	$4 = 00100_2$	$4 = 00100_2$	$0 = 00000_2$
4	2	$5 = 00101_2$	$4 = 00100_2$	$1 = 00001_2$
5	2	$6 = 00110_2$	$4 = 00100_2$	$2 = 00010_2$
6	2	$7 = 00111_2$	$4 = 00100_2$	$3 = 00011_2$
7	3	$8 = 01000_2$	$8 = 01000_2$	$0 = 00000_2$
8	3	$9 = 01001_2$	$8 = 01000_2$	$1 = 00000_2$
9	3	$10 = 01001_2$	$8 = 01000_2$	$2 = 00000_2$

See the pattern? It's simply that $m + 1 - 2^n$ is m with the leftmost 1-bit turned off. So, if we can find the leftmost 1-bit in $m + 1$ we can figure out the number of leaves.

In turn, there are numerous ways to find the leftmost 1-bit in a number. The one used here is based on the principle that, if x is a positive integer, then x & $(x - 1)$ is x with its rightmost 1-bit turned off.

Here's the code that calculates the number of leaves and stores it in *leaves*:

§91 ⟨ Calculate *leaves* 91 ⟩ ≡
```
leaves = tree→bst_count + 1;
for (;;) {
    unsigned long next = leaves & (leaves − 1);
    if (next == 0)
        break;
    leaves = next;
}
leaves = tree→bst_count + 1 − leaves;
```
This code is included in §90, §285, §512, and §680.

Once we have the number of leaves, we perform a compression composed of *leaves* compression transformations. That's all it takes to reduce the general case to the $2^n - 1$ special case. We'll write the *compress*() function itself later:

§92 ⟨ Reduce vine general case to special case 92 ⟩ ≡
```
compress ((struct bst_node *) &tree→bst_root, leaves);
```
This code is included in §90, §512, and §680.

The heart of the function is the compression of the vine into the tree. Before each compression, *vine* contains the number of nodes in the main vine of the tree. The number of compression transformations necessary for the compression is *vine* / 2; e.g., when the main vine contains 7 nodes, $7/2 = 3$ transformations are necessary. The number of nodes in the vine afterward is the same number (see page 73).

At the same time, we keep track of the height of the balanced tree. The final tree always has height at least 1. Each compression step means that it is one level taller than that. If the tree needed general-to-special-case transformations, that is, *leaves* > 0, then it's one more than that.

§93 ⟨ Make special case vine into balanced tree and count height 93 ⟩ ≡

```
vine = tree→bst_count − leaves;
height = 1 + (leaves > 0);
while (vine > 1) {
    compress ((struct bst_node *) &tree→bst_root, vine / 2);
    vine /= 2;
    height++;
}
```

This code is included in §90, §512, and §680.

Finally, we make sure that the height of the tree is within range for what the functions that use stacks can handle. Otherwise, we could end up with an infinite loop, with *bst_t_next*() (for example) calling *bst_balance*() repeatedly to balance the tree in order to reduce its height to the acceptable range.

§94 ⟨ Check for tree height in range 94 ⟩ ≡

```
if (height > BST_MAX_HEIGHT) {
    fprintf (stderr, "libavl: Tree too big (%lu nodes) to handle.",
             (unsigned long) tree→bst_count);
    exit (EXIT_FAILURE);
}
```

This code is included in §90.

4.12.2.3 Implementing Compression

The final bit of code we need is that for performing a compression. The following code performs a compression consisting of *count* applications of the compression transformation starting at *root*:

§95 ⟨ BST compression function 95 ⟩ ≡

```
/* Performs a compression transformation count times, starting at root. */
static void compress (struct bst_node *root, unsigned long count) {
    assert (root != NULL);

    while (count−−) {
        struct bst_node *red = root→bst_link[0];
        struct bst_node *black = red→bst_link[0];

        root→bst_link[0] = black;
        red→bst_link[0] = black→bst_link[1];
        black→bst_link[1] = red;
        root = black;
    }
```

}

This code is included in §90 and §512.

The operation of *compress*() should be obvious, given the discussion earlier. See Section 4.12.2.1 [Balancing General Trees], page 74, above, for a review.

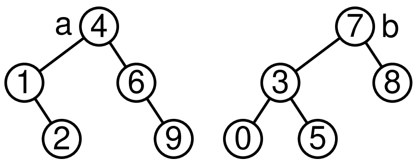
See also: [Stout 1986], *vine_to_tree* procedure.

4.13 Aside: Joining BSTs

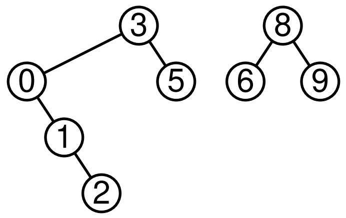
Occasionally we may want to take a pair of BSTs and merge or "join" their contents, forming a single BST that contains all the items in the two original BSTs. It's easy to do this with a series of calls to *bst_insert*(), but we can optimize the process if we write a function exclusively for the purpose. We'll write such a function in this section.

There are two restrictions on the trees to be joined. First, the BSTs' contents must be disjoint. That is, no item in one may match any item in the other. Second, the BSTs must have compatible comparison functions. Typically, they are the same. Speaking more precisely, if $f()$ and $g()$ are the comparison functions, p and q are nodes in either BST, and r and s are the BSTs' user-provided extra comparison parameters, then the expressions $f(p,\ q,\ r)$, $f(p,\ q,\ s)$, $g(p,\ q,\ r)$, and $g(p,\ q,\ s)$ must all have the same value for all possible choices of p and q.

Suppose we're trying to join the trees shown below:

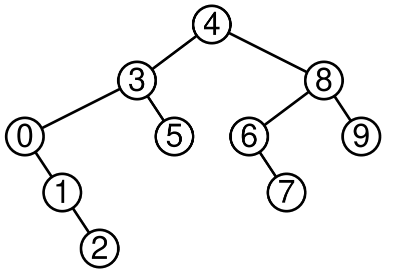


Our first inclination is to try a "divide and conquer" approach by reducing the problem of joining a and b to the subproblems of joining a's left subtree with b's left subtree and joining a's right subtree with b's right subtree. Let us postulate for the moment that we are able to solve these subproblems and that the solutions that we come up with are the following:



To convert this partial solution into a full solution we must combine these two subtrees into a single tree and at the same time reintroduce the nodes a and b into the combined tree. It is easy enough to do this by making a (or b) the root of the combined tree with these two subtrees as its children, then inserting b (or a) into the combined tree. Unfortunately, in neither case will this actually work out properly for our example. The diagram below illustrates one possibility, the result of combining the two subtrees as the child of node 4, then inserting node 7 into the final tree. As you can see, nodes 4 and 5 are out of order:[4]

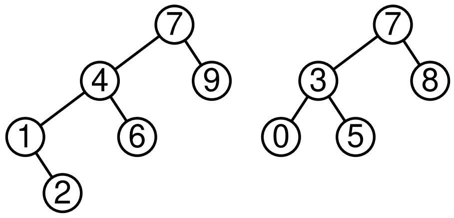
[4] The ** notation in the diagram emphasizes that this is a counterexample.



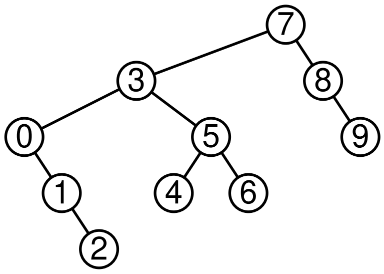
**

Now let's step back and analyze why this attempt failed. It was essentially because, when we recombined the subtrees, a node in the combined tree's left subtree had a value larger than the root. If we trace it back to the original trees to be joined, we see that this was because node 5 in the left subtree of b is greater than a. (If we had chosen 7 as the root of the combined tree we would have found instead node 6 in the right subtree of b to be the culprit.)

On the other hand, if every node in the left subtree of a had a value less than b's value, and every node in the right subtree of a had a value greater than b's value, there would be no problem. Hey, wait a second... we can force that condition. If we perform a root insertion (see Section 4.7.1 [Root Insertion in a BST], page 37) of b into subtree a, then we end up with one pair of subtrees whose node values are all less than 7 (the new and former left subtrees of node 7) and one pair of subtrees whose node values are all greater than 7 (the new and former right subtrees of node 7). Conceptually it looks like this, although in reality we would need to remove node 7 from the tree on the right as we inserted it into the tree on the left:



We can then combine the two subtrees with values less than 7 with each other, and similarly for the ones with values greater than 7, using the same algorithm recursively, and safely set the resulting subtrees as the left and right subtrees of node 7, respectively. The final product is a correctly joined binary tree:



Of course, since we've defined a join recursively in terms of itself, there must be some maximum depth to the recursion, some simple case that can be defined without further recursion. This is easy: the join of an empty tree with another tree is the second tree.

Implementation

It's easy to implement this algorithm recursively. The only nonobvious part of the code below is the treatment of node *b*. We want to insert node *b*, but not *b*'s children, into the subtree rooted at *a*. However, we still need to keep track of *b*'s children. So we temporarily save *b*'s children as *b0* and *b1* and set its child pointers to `NULL` before the root insertion.

This code makes use of *root_insert*() from ⟨Robust root insertion of existing node in arbitrary subtree 625⟩.

§96 ⟨BST join function, recursive version 96⟩ ≡

```
/* Joins a and b, which are subtrees of tree, and returns the resulting tree. */
static struct bst_node *join (struct bst_table *tree, struct bst_node *a, struct bst_node *b) {
  if (b == NULL)
    return a;
  else if (a == NULL)
    return b;
  else {
    struct bst_node *b0 = b->bst_link[0];
    struct bst_node *b1 = b->bst_link[1];
    b->bst_link[0] = b->bst_link[1] = NULL;
    root_insert (tree, &a, b);
    a->bst_link[0] = join (tree, b0, a->bst_link[0]);
    a->bst_link[1] = join (tree, b1, a->bst_link[1]);
    return a;
  }
}

/* Joins a and b, which must be disjoint and have compatible comparison functions.
   b is destroyed in the process. */
void bst_join (struct bst_table *a, struct bst_table *b) {
  a->bst_root = join (a, a->bst_root, b->bst_root);
  a->bst_count += b->bst_count;
  free (b);
}
```

See also: [Sedgewick 1998], program 12.16.

Exercises:

1. Rewrite *bst_join*() to avoid use of recursion.

4.14 Testing

Whew! We're finally done with building functions for performing BST operations. But we haven't tested any of our code. Testing is an essential step in writing programs, because untested software cannot be assumed to work.

Let's build a test program that exercises all of the functions we wrote. We'll also do our best to make parts of it generic, so that we can reuse test code in later chapters when we want to test other BST-based structures.

The first step is to figure out how to test the code. One goal in testing is to exercise as much of the code as possible. Ideally, every line of code would be executed sometime during testing. Often, this is difficult or impossible, but the principle remains valid, with the goal modified to testing as much of the code as possible.

In applying this principle to the BST code, we have to consider why each line of code is executed. If we look at the code for most functions in ⟨`bst.c` 25⟩, we can see that, if we execute them for any BST of reasonable size, most or all of their code will be tested.

This is encouraging. It means that we can just construct some trees and try out the BST functions on them, check that the results make sense, and have a pretty good idea that they work. Moreover, if we build trees in a random fashion, and delete their nodes in a random order, and do it several times, we'll even have a good idea that the *bst_probe*() and *bst_delete*() cases have all come up and worked properly. (If you want to be sure, then you can insert *printf*() calls for each case to record when they trip.) This is not the same as a proof of correctness, but proofs of correctness can only be constructed by computer scientists with fancy degrees, not by mere clever programmers.

There are three notably missing pieces of code coverage if we just do the above. These are stack overflow handling, memory allocation failure handling, and traverser code to deal with modified trees. But we can mop up these extra problems with a little extra effort:[5]

- Stack overflow handling can be tested by forcing the stack to overflow. Stack overflow can occur in many places, so for best effect we must test each possible spot. We will write special tests for these problems.
- Memory allocation failure handling can be tested by simulating memory allocation failures. We will write a replacement memory allocator that "fails" after a specified number of calls. This allocator will also allow for memory leak detection.
- Traverser code to deal with modified trees. This can be tested by modifying trees during traversal and making sure that the traversal functions still work as expected.

The testing code can be broken into the following groups of functions:

Testing and verification
These functions actually try out the BST routines and do their best to make sure that their results are correct.

Test set generation
Generates the order of node insertion and deletion, for use during testing.

Memory manager
Handles memory issues, including memory leak detection and failure simulation.

User interaction
Figures out what the user wants to test in this run.

Main program
Glues everything else together by calling functions in the proper order.

Utilities
Miscellaneous routines that don't fit comfortably into another category.

[5] Some might scoff at this amount of detail, calling it wasted effort, but this thorough testing in fact revealed a number of subtle bugs during development of Libavl that had otherwise gone unnoticed.

Most of the test code will also work nicely for testing other binary tree-based structures. This code is grouped into a single file, ⟨ `test.c` 97 ⟩, which has the following structure:

§97 ⟨ `test.c` 97 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ limits.h ⟩
#include ⟨ stdarg.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include ⟨ string.h ⟩
#include ⟨ time.h ⟩
#include "test.h"

⟨ Test declarations 121 ⟩
⟨ Test utility functions 134 ⟩
⟨ Memory tracker 126 ⟩
⟨ Option parser 586 ⟩
⟨ Command line parser 589 ⟩
⟨ Insertion and deletion order generation 642 ⟩
⟨ Random number seeding 643 ⟩
⟨ Test main program 140 ⟩

The code specifically for testing BSTs goes into ⟨ `bst-test.c` 98 ⟩, outlined like this:

§98 ⟨ `bst-test.c` 98 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ limits.h ⟩
#include ⟨ stdio.h ⟩
#include "bst.h"
#include "test.h"

⟨ BST print function 119 ⟩
⟨ BST traverser check function 104 ⟩
⟨ Compare two BSTs for structure and content 106 ⟩
⟨ Recursively verify BST structure 113 ⟩
⟨ BST verify function 109 ⟩
⟨ BST test function 100 ⟩
⟨ BST overflow test function 122 ⟩

The interface between ⟨ `test.c` 97 ⟩ and ⟨ `bst-test.c` 98 ⟩ is contained in ⟨ `test.h` 99 ⟩:

§99 ⟨ `test.h` 99 ⟩ ≡
⟨ License 1 ⟩
#ifndef `TEST_H`
#define `TEST_H` 1

⟨ Memory allocator 5 ⟩
⟨ Test prototypes 101 ⟩

#endif /* test.h */

Although much of the test program code is nontrivial, only some of the interesting parts fall within the scope of this book. The remainder will be listed without comment or relegated to the exercises. The most tedious code is listed in an appendix (see Appendix B [Supplementary Code], page 323).

4.14.1 Testing BSTs

As suggested above, the main way we will test the BST routines is by using them and checking the results, with checks performed by slow but simple routines. The idea is that bugs in the BST routines are unlikely to be mirrored in the check routines, and vice versa. This way, identical results from the BST and checks tend to indicate that both implementations are correct.

The main test routine is designed to exercise as many of the BST functions as possible. It starts by creating a BST and inserting nodes into it, then deleting the nodes. Midway, various traversals are tested, including the ability to traverse a tree while its content is changing. After each operation that modifies the tree, its structure and content are verified for correspondence with expectations. The function for copying a BST is also tested. This function, *test*(), has the following outline:

§100 ⟨ BST test function 100 ⟩ ≡
```
/* Tests tree functions.
   insert[] and delete[] must contain some permutation of values 0...n − 1.
   Uses allocator as the allocator for tree and node data.
   Higher values of verbosity produce more debug output. */
int test_correctness (struct libavl_allocator *allocator,
                      int insert[], int delete[], int n, int verbosity) {
  struct bst_table *tree;
  int okay = 1;
  int i;
  ⟨ Test creating a BST and inserting into it 102 ⟩
  ⟨ Test BST traversal during modifications 103 ⟩
  ⟨ Test deleting nodes from the BST and making copies of it 105 ⟩
  ⟨ Test deleting from an empty tree 107 ⟩
  ⟨ Test destroying the tree 108 ⟩
  return okay;
}
```
This code is included in §98, §186, §238, §330, §368, §449, §482, §548, and §583.

§101 ⟨ Test prototypes 101 ⟩ ≡
```
int test_correctness (struct libavl_allocator *allocator,
                      int insert[], int delete[], int n, int verbosity);
```
See also §123 and §135.
This code is included in §99.

The first step is to create a BST and insert items into it in the order specified by the caller. We use the comparison function *compare_ints*() from ⟨ Comparison function for **int**s 3 ⟩ to put the tree's items into ordinary numerical order. After each insertion we call *verify_tree*(), which we'll write later and which checks that the tree actually contains the items that it should:

§102 ⟨ Test creating a BST and inserting into it 102 ⟩ ≡

```
tree = bst_create (compare_ints, NULL, allocator);
if (tree == NULL) {
    if (verbosity >= 0) printf ("␣␣Out␣of␣memory␣creating␣tree.\n");
    return 1;
}
for (i = 0; i < n; i++) {
    if (verbosity >= 2) printf ("␣␣Inserting␣%d...\n", insert[i]);
    /* Add the ith element to the tree. */
    {
        void **p = bst_probe (tree, &insert[i]);
        if (p == NULL) {
            if (verbosity >= 0) printf ("␣␣␣␣Out␣of␣memory␣in␣insertion.\n");
            bst_destroy (tree, NULL);
            return 1;
        }
        if (*p != &insert[i]) printf ("␣␣␣␣Duplicate␣item␣in␣tree!\n");
    }
    if (verbosity >= 3) print_whole_tree (tree, "␣␣␣␣Afterward");
    if (!verify_tree (tree, insert, i + 1))
        return 0;
}
```

This code is included in §100 and §295.

If the tree is being modified during traversal, that causes a little more stress on the tree routines, so we should test this specially. We initialize one traverser, x, at a selected item, then delete and reinsert a different item in order to invalidate that traverser. We make a copy, y, of the traverser in order to check that *bst_t_copy*() works properly and initialize a third traverser, z, with the inserted item. After the deletion and reinsertion we check that all three of the traversers behave properly.

§103 ⟨Test BST traversal during modifications 103⟩ ≡

```
for (i = 0; i < n; i++) {
    struct bst_traverser x, y, z;
    int *deleted;
    if (insert[i] == delete[i])
        continue;
    if (verbosity >= 2)
        printf ("␣␣␣Checking␣traversal␣from␣item␣%d...\n", insert[i]);
    if (bst_t_find (&x, tree, &insert[i]) == NULL) {
        printf ("␣␣␣␣Can't␣find␣item␣%d␣in␣tree!\n", insert[i]);
        continue;
    }
    okay &= check_traverser (&x, insert[i], n, "Predeletion");
    if (verbosity >= 3) printf ("␣␣␣␣Deleting␣item␣%d.\n", delete[i]);
    deleted = bst_delete (tree, &delete[i]);
    if (deleted == NULL || *deleted != delete[i]) {
```

```
        okay = 0;
        if (deleted == NULL)
            printf ("␣␣␣␣Deletion␣failed.\n");
        else printf ("␣␣␣␣Wrong␣node␣%d␣returned.\n", *deleted);
    }
    bst_t_copy (&y, &x);
    if (verbosity >= 3) printf ("␣␣␣␣Reinserting␣item␣%d.\n", delete[i]);
    if (bst_t_insert (&z, tree, &delete[i]) == NULL) {
        if (verbosity >= 0) printf ("␣␣␣␣Out␣of␣memory␣reinserting␣item.\n");
        bst_destroy (tree, NULL);
        return 1;
    }
    okay &= check_traverser (&x, insert[i], n, "Postdeletion");
    okay &= check_traverser (&y, insert[i], n, "Copied");
    okay &= check_traverser (&z, delete[i], n, "Insertion");
    if (!verify_tree (tree, insert, n))
        return 0;
}
```

This code is included in §100 and §295.

The *check_traverser*() function used above checks that a traverser behaves properly, by checking that the traverser is at the correct item and that the previous and next items are correct as well.

§104 ⟨ BST traverser check function 104 ⟩ ≡

```
/* Checks that the current item at trav is i
   and that its previous and next items are as they should be.
   label is a name for the traverser used in reporting messages.
   There should be n items in the tree numbered 0...n − 1.
   Returns nonzero only if there is an error. */
static int check_traverser (struct bst_traverser *trav, int i, int n, const char *label) {
    int okay = 1;
    int *cur, *prev, *next;
    prev = bst_t_prev (trav);
    if ((i == 0 && prev != NULL) || (i > 0 && (prev == NULL || *prev != i − 1))) {
        printf ("␣␣␣%s␣traverser␣ahead␣of␣%d,␣but␣should␣be␣ahead␣of␣%d.\n",
                label, prev != NULL ? *prev : −1, i == 0 ? −1 : i − 1);
        okay = 0;
    }
    bst_t_next (trav);
    cur = bst_t_cur (trav);
    if (cur == NULL || *cur != i) {
        printf ("␣␣␣%s␣traverser␣at␣%d,␣but␣should␣be␣at␣%d.\n",
                label, cur != NULL ? *cur : −1, i);
        okay = 0;
    }
    next = bst_t_next (trav);
```

```
    if ((i == n - 1 && next != NULL)
        || (i != n - 1 && (next == NULL || *next != i + 1))) {
      printf ("   %s traverser behind %d, but should be behind %d.\n",
              label, next != NULL ? *next : -1, i == n - 1 ? -1 : i + 1);
      okay = 0;
    }
    bst_t_prev (trav);

    return okay;
}
```

This code is included in §98, §186, §238, §290, §330, §368, §411, §449, §482, §515, §548, and §583.

We also need to test deleting nodes from the tree and making copies of a tree. Here's the code to do that:

§105 ⟨ Test deleting nodes from the BST and making copies of it 105 ⟩ ≡

```
for (i = 0; i < n; i++) {
    int *deleted;

    if (verbosity >= 2) printf ("  Deleting %d...\n", delete[i]);

    deleted = bst_delete (tree, &delete[i]);
    if (deleted == NULL || *deleted != delete[i]) {
      okay = 0;
      if (deleted == NULL)
        printf ("    Deletion failed.\n");
      else printf ("    Wrong node %d returned.\n", *deleted);
    }

    if (verbosity >= 3) print_whole_tree (tree, "    Afterward");

    if (!verify_tree (tree, delete + i + 1, n - i - 1))
      return 0;

    if (verbosity >= 2) printf ("  Copying tree and comparing...\n");

    /* Copy the tree and make sure it's identical. */
    {
      struct bst_table *copy = bst_copy (tree, NULL, NULL, NULL);
      if (copy == NULL) {
        if (verbosity >= 0) printf ("  Out of memory in copy\n");
        bst_destroy (tree, NULL);
        return 1;
      }

      okay &= compare_trees (tree->bst_root, copy->bst_root);
      bst_destroy (copy, NULL);
    }
}
```

This code is included in §100 and §295.

The actual comparison of trees is done recursively for simplicity:

§106 ⟨ Compare two BSTs for structure and content 106 ⟩ ≡

```
/* Compares binary trees rooted at a and b, making sure that they are identical. */
static int compare_trees (struct bst_node *a, struct bst_node *b) {
```

```
    int okay;
    if (a == NULL || b == NULL) {
        assert (a == NULL && b == NULL);
        return 1;
    }
    if (*(int *) a->bst_data != *(int *) b->bst_data
        || ((a->bst_link[0] != NULL) != (b->bst_link[0] != NULL))
        || ((a->bst_link[1] != NULL) != (b->bst_link[1] != NULL))) {
        printf (" Copied nodes differ: a=%d b=%d a:",
                *(int *) a->bst_data, *(int *) b->bst_data);
        if (a->bst_link[0] != NULL) printf ("l");
        if (a->bst_link[1] != NULL) printf ("r");
        printf (" b:");
        if (b->bst_link[0] != NULL) printf ("l");
        if (b->bst_link[1] != NULL) printf ("r");
        printf ("\n");
        return 0;
    }
    okay = 1;
    if (a->bst_link[0] != NULL) okay &= compare_trees (a->bst_link[0], b->bst_link[0]);
    if (a->bst_link[1] != NULL) okay &= compare_trees (a->bst_link[1], b->bst_link[1]);
    return okay;
}
```

This code is included in §98.

As a simple extra check, we make sure that attempting to delete from an empty tree fails in the expected way:

§107 ⟨ Test deleting from an empty tree 107 ⟩ ≡

```
if (bst_delete (tree, &insert[0]) != NULL) {
    printf (" Deletion from empty tree succeeded.\n");
    okay = 0;
}
```

This code is included in §100.

Finally, we're done with the tree and can get rid of it.

§108 ⟨ Test destroying the tree 108 ⟩ ≡

```
/* Test destroying the tree. */
bst_destroy (tree, NULL);
```

This code is included in §100 and §295.

Exercises:

1. Which functions in ⟨ bst.c 25 ⟩ are not exercised by *test*()?

2. Some errors within *test*() just set the *okay* flag to zero, whereas others cause an immediate unsuccessful return to the caller without performing any cleanup. A third class of errors causes cleanup followed by a successful return. Why and how are these distinguished?

4.14.1.1 BST Verification

After each change to the tree in the testing program, we call *verify_tree*() to check that the tree's structure and content are what we think they should be. This function runs through a full gamut of checks, with the following outline:

```
§109 ⟨ BST verify function 109 ⟩ ≡
/* Checks that tree is well-formed
   and verifies that the values in array[] are actually in tree.
   There must be n elements in array[] and tree.
   Returns nonzero only if no errors detected. */
static int verify_tree (struct bst_table *tree, int array[], size_t n) {
    int okay = 1;
    ⟨ Check tree→bst_count is correct 110 ⟩
    if (okay) { ⟨ Check BST structure 111 ⟩ }
    if (okay) { ⟨ Check that the tree contains all the elements it should 115 ⟩ }
    if (okay) { ⟨ Check that forward traversal works 116 ⟩ }
    if (okay) { ⟨ Check that backward traversal works 117 ⟩ }
    if (okay) { ⟨ Check that traversal from the null element works 118 ⟩ }
    return okay;
}
```

This code is included in §98, §411, and §515.

The first step just checks that the number of items passed in as n is the same as *tree→bst_count*.

```
§110 ⟨ Check tree→bst_count is correct 110 ⟩ ≡
/* Check tree's bst_count against that supplied. */
if (bst_count (tree) != n) {
    printf (" Tree count is %lu, but should be %lu.\n",
            (unsigned long) bst_count (tree), (unsigned long) n);
    okay = 0;
}
```

This code is included in §109, §190, §244, and §294.

Next, we verify that the BST has proper structure and that it has the proper number of items. We'll do this recursively because that's easiest and most obviously correct way. Function *recurse_verify_tree*() for this returns the number of nodes in the BST. After it returns, we verify that this is the expected number.

```
§111 ⟨ Check BST structure 111 ⟩ ≡
/* Recursively verify tree structure. */
size_t count;
recurse_verify_tree (tree→bst_root, &okay, &count, 0, INT_MAX);
⟨ Check counted nodes 112 ⟩
```

This code is included in §109 and §294.

```
§112 ⟨ Check counted nodes 112 ⟩ ≡
if (count != n) {
```

```
    printf (" Tree has %lu nodes, but should have %lu.\n",
            (unsigned long) count, (unsigned long) n);
    okay = 0;
}
```

This code is included in §111, §191, and §246.

The function *recurse_verify_tree*() does the recursive verification. It checks that nodes' values increase down to the right and decrease down to the left. We also use it to count the number of nodes actually in the tree:

§113 ⟨ Recursively verify BST structure 113 ⟩ ≡

```
/* Examines the binary tree rooted at node.
   Zeroes *okay if an error occurs. Otherwise, does not modify *okay.
   Sets *count to the number of nodes in that tree, including node itself if node != NULL.
   All the nodes in the tree are verified to be at least min but no greater than max. */
static void recurse_verify_tree (struct bst_node *node, int *okay, size_t *count,
                                 int min, int max) {
    int d; /* Value of this node's data. */
    size_t subcount[2]; /* Number of nodes in subtrees. */
    if (node == NULL) {
        *count = 0;
        return;
    }
    d = *(int *) node->bst_data;
    ⟨ Verify binary search tree ordering 114 ⟩
    recurse_verify_tree (node->bst_link[0], okay, &subcount[0], min, d - 1);
    recurse_verify_tree (node->bst_link[1], okay, &subcount[1], d + 1, max);
    *count = 1 + subcount[0] + subcount[1];
}
```

This code is included in §98.

§114 ⟨ Verify binary search tree ordering 114 ⟩ ≡

```
if (min > max) {
    printf (" Parents of node %d constrain it to empty range %d...%d.\n",
            d, min, max);
    *okay = 0;
} else if (d < min || d > max) {
    printf (" Node %d is not in range %d...%d implied by its parents.\n",
            d, min, max);
    *okay = 0;
}
```

This code is included in §113, §188, §240, §293, §332, §370, §414, §451, §484, §517, §550, and §585.

The third step is to check that the BST indeed contains all of the items that it should:

§115 ⟨ Check that the tree contains all the elements it should 115 ⟩ ≡

```
/* Check that all the values in array[] are in tree. */
size_t i;
for (i = 0; i < n; i++)
```

```
    if (bst_find (tree, &array[i]) == NULL) {
        printf (" Tree does not contain expected value %d.\n", array[i]);
        okay = 0;
    }
```

This code is included in §109, §190, §244, and §294.

The final steps all check traversal of the BST, first by traversing in forward order from the beginning to the end, then in reverse order, then by checking that the null item behaves correctly. The forward traversal checks that the proper number of items are in the BST. It could appear to have too few items if the tree's pointers are screwed up in one way, or it could appear to have too many items if they are screwed up in another way. We try to figure out how many items actually appear in the tree during traversal, but give up if the count gets to be more than twice that expected, assuming that this indicates a "loop" that will cause traversal to never terminate.

§116 ⟨Check that forward traversal works 116⟩ ≡

```
/* Check that bst_t_first() and bst_t_next() work properly. */
struct bst_traverser trav;
size_t i;
int prev = -1;
int *item;

for (i = 0, item = bst_t_first (&trav, tree); i < 2 * n && item != NULL;
    i++, item = bst_t_next (&trav)) {
    if (*item <= prev) {
        printf (" Tree out of order: %d follows %d in traversal\n", *item, prev);
        okay = 0;
    }
    prev = *item;
}
if (i != n) {
    printf (" Tree should have %lu items, but has %lu in traversal\n",
            (unsigned long) n, (unsigned long) i);
    okay = 0;
}
```

This code is included in §109, §190, §244, and §294.

We do a similar traversal in the reverse order:

§117 ⟨Check that backward traversal works 117⟩ ≡

```
/* Check that bst_t_last() and bst_t_prev() work properly. */
struct bst_traverser trav;
size_t i;
int next = INT_MAX;
int *item;

for (i = 0, item = bst_t_last (&trav, tree); i < 2 * n && item != NULL;
    i++, item = bst_t_prev (&trav)) {
    if (*item >= next) {
        printf (" Tree out of order: %d precedes %d in traversal\n", *item, next);
        okay = 0;
```

```
    }
    next = *item;
}
if (i != n) {
    printf (" Tree should have %lu items, but has %lu in reverse\n",
            (unsigned long) n, (unsigned long) i);
    okay = 0;
}
```

This code is included in §109, §190, §244, and §294.

The final check to perform on the traverser is to make sure that the traverser null item works properly. We start out a traverser at the null item with *bst_t_init*(), then make sure that the next item after that, as reported by *bst_t_next*(), is the same as the item returned by *bst_t_init*(), and similarly for the previous item:

§118 ⟨ Check that traversal from the null element works 118 ⟩ ≡

```
/* Check that bst_t_init() works properly. */
struct bst_traverser init, first, last;
int *cur, *prev, *next;
bst_t_init (&init, tree);
bst_t_first (&first, tree);
bst_t_last (&last, tree);
cur = bst_t_cur (&init);
if (cur != NULL) {
    printf (" Inited traverser should be null, but is actually %d.\n", *cur);
    okay = 0;
}
next = bst_t_next (&init);
if (next != bst_t_cur (&first)) {
    printf (" Next after null should be %d, but is actually %d.\n",
            *(int *) bst_t_cur (&first), *next);
    okay = 0;
}
bst_t_prev (&init);
prev = bst_t_prev (&init);
if (prev != bst_t_cur (&last)) {
    printf (" Previous before null should be %d, but is actually %d.\n",
            *(int *) bst_t_cur (&last), *prev);
    okay = 0;
}
bst_t_next (&init);
```

This code is included in §109, §190, §244, and §294.

Exercises:

1. Many of the segments of code in this section cast **size_t** arguments to *printf*() to **unsigned long**. Why?

2. Does *test*() work properly for testing trees with only one item in them? Zero items?

4.14.1.2 Displaying BST Structures

The *print_tree_structure*() function below can be useful for debugging, but it is not used very much by the testing code. It prints out the structure of a tree, with the root first, then its children in parentheses separated by a comma, and their children in inner parentheses, and so on. This format is easy to print but difficult to visualize, so it's a good idea to have a notebook on hand to sketch out the shape of the tree. Alternatively, this output is in the right format to feed directly into the `texitree` program used to draw the tree diagrams in this book, which can produce output in plain text or PostScript form.

§119 ⟨BST print function 119⟩ ≡
```
/* Prints the structure of node, which is level levels from the top of the tree. */
static void print_tree_structure (const struct bst_node *node, int level) {
  /* You can set the maximum level as high as you like.
     Most of the time, you'll want to debug code using small trees,
     so that a large level indicates a "loop", which is a bug. */
  if (level > 16) {
    printf ("[...]");
    return;
  }

  if (node == NULL)
    return;

  printf ("%d", *(int *) node->bst_data);
  if (node->bst_link[0] != NULL || node->bst_link[1] != NULL) {
    putchar ('(');

    print_tree_structure (node->bst_link[0], level + 1);
    if (node->bst_link[1] != NULL) {
      putchar (',');
      print_tree_structure (node->bst_link[1], level + 1);
    }

    putchar (')');
  }
}
```
See also §120.
This code is included in §98, §186, §238, §515, §548, and §583.

A function *print_whole_tree*() is also provided as a convenient wrapper for printing an entire BST's structure.

§120 ⟨BST print function 119⟩ +≡
```
/* Prints the entire structure of tree with the given title. */
void print_whole_tree (const struct bst_table *tree, const char *title) {
  printf ("%s: ", title);
  print_tree_structure (tree->bst_root, 0);
  putchar ('\n');
}
```

4.14.2 Test Set Generation

We need code to generate a random permutation of numbers to order insertion and deletion of items. We will support some other orders besides random permutation as well for completeness and to allow for overflow testing. Here is the complete list:

§121 ⟨ Test declarations 121 ⟩ ≡
```
/* Insertion order. */
enum insert_order {
    INS_RANDOM, /* Random order. */
    INS_ASCENDING, /* Ascending order. */
    INS_DESCENDING, /* Descending order. */
    INS_BALANCED, /* Balanced tree order. */
    INS_ZIGZAG, /* Zig-zag order. */
    INS_ASCENDING_SHIFTED, /* Ascending from middle, then beginning. */
    INS_CUSTOM, /* Custom order. */

    INS_CNT /* Number of insertion orders. */
};

/* Deletion order. */
enum delete_order {
    DEL_RANDOM, /* Random order. */
    DEL_REVERSE, /* Reverse of insertion order. */
    DEL_SAME, /* Same as insertion order. */
    DEL_CUSTOM, /* Custom order. */

    DEL_CNT /* Number of deletion orders. */
};
```

See also §125, §133, §138, §139, and§141.
This code is included in §97.

The code to actually generate these orderings is left to the exercises.

Exercises:

1. Write a function to generate a random permutation of the n **int**s between 0 and $n - 1$ into a provided array.

***2.** Write a function to generate an ordering of **int**s that, when inserted into a binary tree, produces a balanced tree of the integers from *min* to *max* inclusive. (Hint: what kind of recursive traversal makes this easy?)

3. Write one function to generate an insertion order of n integers into a provided array based on an **enum insert_order** and the functions written in the previous two exercises. Write a second function to generate a deletion order using similar parameters plus the order of insertion.

***4.** By default, the C random number generator produces the same sequence every time the program is run. In order to generate different sequences, it has to be "seeded" using *srand*() with a unique value. Write a function to select a random number seed based on the current time.

4.14.3 Testing Overflow

Testing for overflow requires an entirely different set of test functions. The idea is to create a too-tall tree using one of the pathological insertion orders (ascending, descending, zig-zag, shifted ascending), then try out each of the functions that can overflow on it and make sure that they behave as they should.

There is a separate test function for each function that can overflow a stack but which is not tested by *test*(). These functions are called by driver function *test_overflow*(), which also takes care of creating, populating, and destroying the tree.

§122 ⟨ BST overflow test function 122 ⟩ ≡

```
⟨ Overflow testers 124 ⟩
/* Tests the tree routines for proper handling of overflows.
   Inserting the n elements of order[] should produce a tree
   with height greater than BST_MAX_HEIGHT.
   Uses allocator as the allocator for tree and node data.
   Use verbosity to set the level of chatter on stdout. */
int test_overflow (struct libavl_allocator *allocator, int order[], int n, int verbosity) {
    /* An overflow tester function. */
    typedef int test_func (struct bst_table *, int n);

    /* An overflow tester. */
    struct test {
        test_func *func; /* Tester function. */
        const char *name; /* Test name. */
    };

    /* All the overflow testers. */
    static const struct test test[] = {
        {test_bst_t_first, "first item"},
        {test_bst_t_last, "last item"},
        {test_bst_t_find, "find item"},
        {test_bst_t_insert, "insert item"},
        {test_bst_t_next, "next item"},
        {test_bst_t_prev, "previous item"},
        {test_bst_copy, "copy tree"},
    };

    const struct test *i; /* Iterator. */

    /* Run all the overflow testers. */
    for (i = test; i < test + sizeof test / sizeof *test; i++) {
        struct bst_table *tree;
        int j;

        if (verbosity >= 2) printf ("  Running %s test...\n", i->name);

        tree = bst_create (compare_ints, NULL, allocator);
        if (tree == NULL) {
            printf ("    Out of memory creating tree.\n");
            return 1;
        }
```

```
        for (j = 0; j < n; j++) {
            void **p = bst_probe (tree, &order[j]);
            if (p == NULL || *p != &order[j]) {
                if (p == NULL && verbosity >= 0)
                    printf ("␣␣␣␣Out␣of␣memory␣in␣insertion.\n");
                else if (p != NULL) printf ("␣␣␣␣Duplicate␣item␣in␣tree!\n");
                bst_destroy (tree, NULL);
                return p == NULL;
            }
        }
        if (i→func (tree, n) == 0)
            return 0;
        if (verify_tree (tree, order, n) == 0)
            return 0;
        bst_destroy (tree, NULL);
    }
    return 1;
}
```

This code is included in §98, §186, §238, §290, §330, §368, §411, §449, §482, §515, §548, and §583.

§123 ⟨ Test prototypes 101 ⟩ +≡

```
int test_overflow (struct libavl_allocator *, int order[], int n, int verbosity);
```

There is an overflow tester for almost every function that can overflow. Here is one example:

§124 ⟨ Overflow testers 124 ⟩ ≡

```
static int test_bst_t_first (struct bst_table *tree, int n) {
    struct bst_traverser trav;
    int *first;
    first = bst_t_first (&trav, tree);
    if (first == NULL || *first != 0) {
        printf ("␣␣␣␣First␣item␣test␣failed:␣expected␣0,␣got␣%d\n",
                first != NULL ? *first : −1);
        return 0;
    }
    return 1;
}
```

See also §644.

This code is included in §122.

Exercises:

1. Write the rest of the overflow tester functions. (The *test_overflow*() function lists all of them.)

4.14.4 Memory Manager

We want to test our code to make sure that it always releases allocated memory and that it behaves robustly when memory allocations fail. We can do the former by building our own memory manager that keeps tracks of blocks as they are allocated and freed. The memory manager can also disallow allocations according to a policy set by the user, taking care of the latter.

The available policies are:

§125 ⟨ Test declarations 121 ⟩ +≡

```
/* Memory tracking policy. */
enum mt_policy {
    MT_TRACK, /* Track allocation for leak detection. */
    MT_NO_TRACK, /* No leak detection. */
    MT_FAIL_COUNT, /* Fail allocations after a while. */
    MT_FAIL_PERCENT, /* Fail allocations randomly. */
    MT_SUBALLOC /* Suballocate from larger blocks. */
};
```

`MT_TRACK` and `MT_NO_TRACK` should be self-explanatory. `MT_FAIL_COUNT` takes an argument specifying after how many allocations further allocations should always fail. `MT_FAIL_PERCENT` takes an argument specifying an integer percentage of allocations to randomly fail.

`MT_SUBALLOC` causes small blocks to be carved out of larger ones allocated with *malloc*(). This is a good idea for two reasons: *malloc*() can be slow and *malloc*() can waste a lot of space dealing with the small blocks that Libavl uses for its node. Suballocation cannot be implemented in an entirely portable way because of alignment issues, but the test program here requires the user to specify the alignment needed, and its use is optional anyhow.

The memory manager keeps track of allocated blocks using **struct block**:

§126 ⟨ Memory tracker 126 ⟩ ≡

```
/* Memory tracking allocator. */

/* A memory block. */
struct block {
    struct block *next; /* Next in linked list. */

    int idx; /* Allocation order index number. */
    size_t size; /* Size in bytes. */
    size_t used; /* MT_SUBALLOC: amount used so far. */
    void *content; /* Allocated region. */
};
```

See also §127, §128, §129, §130, §131, and§132.
This code is included in §97.

The *next* member of **struct block** is used to keep a linked list of all the currently allocated blocks. Searching this list is inefficient, but there are at least two reasons to do it this way, instead of using a more efficient data structure, such as a binary tree. First, this code is for testing binary tree routines—using a binary tree data structure to do it is a strange idea! Second, the ISO C standard says that, with few exceptions, using the relational operators (<, <=, >, >=) to compare pointers that do not point inside the same array produces undefined behavior, but allows use of the equality operators (==, !=) for a larger class of pointers.

We also need a data structure to keep track of settings and a list of blocks. This memory manager uses the technique discussed in Exercise 2.5-3 to provide this structure to the allocator.

§127 ⟨ Memory tracker 126 ⟩ +≡

```
/* Indexes into arg[] within struct mt_allocator. */
enum mt_arg_index {
    MT_COUNT = 0, /* MT_FAIL_COUNT: Remaining successful allocations. */
    MT_PERCENT = 0, /* MT_FAIL_PERCENT: Failure percentage. */
    MT_BLOCK_SIZE = 0, /* MT_SUBALLOC: Size of block to suballocate. */
    MT_ALIGN = 1 /* MT_SUBALLOC: Alignment of suballocated blocks. */
};
/* Memory tracking allocator. */
struct mt_allocator {
    struct libavl_allocator allocator; /* Allocator. Must be first member. */

    /* Settings. */
    enum mt_policy policy; /* Allocation policy. */
    int arg[2]; /* Policy arguments. */
    int verbosity; /* Message verbosity level. */

    /* Current state. */
    struct block *head, *tail; /* Head and tail of block list. */
    int alloc_idx; /* Number of allocations so far. */
    int block_cnt; /* Number of still-allocated blocks. */
};
```

Function *mt_create*() creates a new instance of the memory tracker. It takes an allocation policy and policy argument, as well as a number specifying how verbose it should be in reporting information. It uses utility function *xmalloc*(), a simple wrapper for *malloc*() that aborts the program on failure. Here it is:

§128 ⟨ Memory tracker 126 ⟩ +≡

```
static void *mt_allocate (struct libavl_allocator *, size_t);
static void mt_free (struct libavl_allocator *, void *);

/* Initializes the memory manager for use
   with allocation policy policy and policy arguments arg[],
   at verbosity level verbosity, where 0 is a "normal" value. */
struct mt_allocator *mt_create (enum mt_policy policy, int arg[2], int verbosity) {
    struct mt_allocator *mt = xmalloc (sizeof *mt);

    mt->allocator.libavl_malloc = mt_allocate;
    mt->allocator.libavl_free = mt_free;

    mt->policy = policy;
    mt->arg[0] = arg[0];
    mt->arg[1] = arg[1];
    mt->verbosity = verbosity;

    mt->head = mt->tail = NULL;
    mt->alloc_idx = 0;
    mt->block_cnt = 0;

    return mt;
```

```
}
```

After allocations and deallocations are done, the memory manager must be freed with *mt_destroy*(), which also reports any memory leaks. Blocks are removed from the block list as they are freed, so any remaining blocks must be leaked memory:

§129 ⟨ Memory tracker 126 ⟩ +≡

```
/* Frees and destroys memory tracker mt, reporting any memory leaks. */
void mt_destroy (struct mt_allocator *mt) {
  assert (mt != NULL);
  if (mt->block_cnt == 0) {
    if (mt->policy != MT_NO_TRACK && mt->verbosity >= 1)
      printf ("  No memory leaks.\n");
  } else {
    struct block *iter, *next;
    if (mt->policy != MT_SUBALLOC) printf ("  Memory leaks detected:\n");
    for (iter = mt->head; iter != NULL; iter = next) {
      if (mt->policy != MT_SUBALLOC)
        printf ("    block #%d: %lu bytes\n",
                iter->idx, (unsigned long) iter->size);
      next = iter->next;
      free (iter->content);
      free (iter);
    }
  }
  free (mt);
}
```

For the sake of good encapsulation, *mt_allocator*() returns the **struct libavl_allocator** associated with a given memory tracker:

§130 ⟨ Memory tracker 126 ⟩ +≡

```
/* Returns the struct libavl_allocator associated with mt. */
void *mt_allocator (struct mt_allocator *mt) {
  return &mt->allocator;
}
```

The allocator function *mt_allocate*() is in charge of implementing the selected allocation policy. It delegates most of the work to a pair of helper functions *new_block*() and *reject_request*() and makes use of utility function *xmalloc*(), a simple wrapper for *malloc*() that aborts the program on failure. The implementation is straightforward:

§131 ⟨ Memory tracker 126 ⟩ +≡

```
/* Creates a new struct block containing size bytes of content
   and returns a pointer to content. */
static void *new_block (struct mt_allocator *mt, size_t size) {
  struct block *new;
  /* Allocate and initialize new struct block. */
  new = xmalloc (sizeof *new);
  new->next = NULL;
```

```
  new->idx = mt->alloc_idx++;
  new->size = size;
  new->used = 0;
  new->content = xmalloc (size);
  /* Add block to linked list. */
  if (mt->head == NULL)
    mt->head = new;
  else mt->tail->next = new;
  mt->tail = new;
  /* Alert user. */
  if (mt->verbosity >= 3)
    printf ("    block #%d: allocated %lu bytes\n",
            new->idx, (unsigned long) size);
  /* Finish up and return. */
  mt->block_cnt++;
  return new->content;
}
/* Prints a message about a rejected allocation if appropriate. */
static void reject_request (struct mt_allocator *mt, size_t size) {
  if (mt->verbosity >= 2)
    printf ("    block #%d: rejected request for %lu bytes\n",
            mt->alloc_idx++, (unsigned long) size);
}
/* Allocates and returns a block of size bytes. */
static void *mt_allocate (struct libavl_allocator *allocator, size_t size) {
  struct mt_allocator *mt = (struct mt_allocator *) allocator;
  /* Special case. */
  if (size == 0)
    return NULL;
  switch (mt->policy) {
    case MT_TRACK: return new_block (mt, size);
    case MT_NO_TRACK: return xmalloc (size);
    case MT_FAIL_COUNT:
      if (mt->arg[MT_COUNT] == 0) {
        reject_request (mt, size);
        return NULL;
      }
      mt->arg[MT_COUNT]--;
      return new_block (mt, size);
    case MT_FAIL_PERCENT:
      if (rand () / (RAND_MAX / 100 + 1) < mt->arg[MT_PERCENT]) {
        reject_request (mt, size);
        return NULL;
      }
      else return new_block (mt, size);
```

```
    case MT_SUBALLOC:
      if (mt->tail == NULL
          || mt->tail->used + size > (size_t) mt->arg[MT_BLOCK_SIZE])
        new_block (mt, mt->arg[MT_BLOCK_SIZE]);
      if (mt->tail->used + size <= (size_t) mt->arg[MT_BLOCK_SIZE]) {
        void *p = (char *) mt->tail->content + mt->tail->used;
        size = ((size + mt->arg[MT_ALIGN] - 1)
                / mt->arg[MT_ALIGN] * mt->arg[MT_ALIGN]);
        mt->tail->used += size;
        if (mt->verbosity >= 3)
          printf ("    block #%d: suballocated %lu bytes\n",
                  mt->tail->idx, (unsigned long) size);
        return p;
      }
      else fail ("blocksize %lu too small for %lubyte allocation",
                 (unsigned long) mt->tail->size, (unsigned long) size);

    default: assert (0);
    }
}
```

The corresponding function *mt_free*() searches the block list for the specified block, removes it, and frees the associated memory. It reports an error if the block is not in the list:

§132 ⟨ Memory tracker 126 ⟩ +≡

```
/* Releases block previously returned by mt_allocate(). */
static void mt_free (struct libavl_allocator *allocator, void *block) {
  struct mt_allocator *mt = (struct mt_allocator *) allocator;
  struct block *iter, *prev;

  /* Special cases. */
  if (block == NULL || mt->policy == MT_NO_TRACK) {
    free (block);
    return;
  }
  if (mt->policy == MT_SUBALLOC)
    return;

  /* Search for block within the list of allocated blocks. */
  for (prev = NULL, iter = mt->head; iter; prev = iter, iter = iter->next) {
    if (iter->content == block) {
      /* Block found. Remove it from the list. */
      struct block *next = iter->next;

      if (prev == NULL)
        mt->head = next;
      else prev->next = next;
      if (next == NULL) mt->tail = prev;

      /* Alert user. */
      if (mt->verbosity >= 4)
```

```
            printf ("␣␣␣␣block␣#%d:␣freed␣%lu␣bytes\n",
                    iter→idx, (unsigned long) iter→size);
          /* Free block. */
          free (iter→content);
          free (iter);
          /* Finish up and return. */
          mt→block_cnt−−;
          return;
      }
  }
  /* Block not in list. */
  printf ("␣␣␣␣attempt␣to␣free␣unknown␣block␣%p␣(already␣freed?)\n", block);
}
```

See also: [ISO 1990], sections 6.3.8 and 6.3.9.

Exercises:

1. As its first action, *mt_allocate*() checks for and special-cases a *size* of 0. Why?

4.14.5 User Interaction

This section briefly discusses LIBAVL's data structures and functions for parsing command-line arguments. For more information on the command-line arguments accepted by the testing program, refer to the LIBAVL reference manual.

The main way that the test program receives instructions from the user is through the set of arguments passed to *main*(). The program assumes that these arguments can be controlled easily by the user, presumably through some kind of command-based "shell" program. It allows for two kinds of options: traditional UNIX "short options" that take the form '`-o`' and GNU-style "long options" of the form '`--option`'. Either kind of option may take an argument.

Options are specified using an array of **struct option**, terminated by an all-zero structure:

§133 ⟨ Test declarations 121 ⟩ +≡

```
/* A single command-line option. */
struct option {
    const char *long_name; /* Long name ("-name"). */
    int short_name; /* Short name ("n"); value returned. */
    int has_arg; /* Has a required argument? */
};
```

There are two public functions in the option parser:

struct option_state **option_init* (**struct option** **options*, **char** ***args*)
: Creates and returns a **struct option_state**, initializing it based on the array of arguments passed in. This structure is used to keep track of the option parsing state. Sets *options* as the set of options to parse.

int *option_get* (**struct option_state** **state*, **char** ***argp*)
: Parses the next option from *state* and returns the value of the *short_name* member from its **struct option**. Sets **argp* to the option's argument or `NULL` if none. Returns $^-1$ and destroys *state* if no options remain.

These functions' implementation are not too interesting for our purposes, so they are relegated to an appendix. See Section B.1 [Option Parser], page 323, for the full story.

The option parser provides a lot of support for parsing the command line, but of course the individual options have to be handled once they are retrieved by *option_get*(). The *parse_command_line*() function takes care of the whole process:

void *parse_command_line* (**char** ∗∗*args*, **struct test_options** ∗*options*)

> Parses the command-line arguments in *args*[], which must be terminated with an element set to all zeros, using *option_init*() and *option_get*(). Sets up *options* appropriately to correspond.

See Section B.2 [Command-Line Parser], page 326, for source code. The **struct test_options** initialized by *parse_command_line*() is described in detail below.

4.14.6 Utility Functions

The first utility function is *compare_ints*(). This function is not used by ⟨ `test.c` 97 ⟩ but it is included there because it is used by the test modules for all the individual tree structures.

§134 ⟨ Test utility functions 134 ⟩ ≡

```
/* Utility functions. */
⟨ Comparison function for ints 3 ⟩
```

See also §136 and §137.
This code is included in §97.

It is prototyped in ⟨ `test.h` 99 ⟩:

§135 ⟨ Test prototypes 101 ⟩ +≡

```
int compare_ints (const void *pa, const void *pb, void *param);
```

The *fail*() function prints a provided error message to *stderr*, formatting it as with *printf*(), and terminates the program unsuccessfully:

§136 ⟨ Test utility functions 134 ⟩ +≡

```
/* Prints message on stderr, which is formatted as for printf(),
   and terminates the program unsuccessfully. */
static void fail (const char *message, ...) {
    va_list args;

    fprintf (stderr, "%s: ", pgm_name);

    va_start (args, message);
    vfprintf (stderr, message, args);
    va_end (args);

    putchar ('\n');

    exit (EXIT_FAILURE);
}
```

Finally, the *xmalloc*() function is a *malloc*() wrapper that aborts the program if allocation fails:

§137 ⟨ Test utility functions 134 ⟩ +≡

```
/* Allocates and returns a pointer to size bytes of memory.
```

```
   Aborts if allocation fails. */
static void *xmalloc (size_t size) {
    void *block = malloc (size);
    if (block == NULL && size != 0)
        fail ("out of memory");
    return block;
}
```

4.14.7 Main Program

Everything comes together in the main program. The test itself (default or overflow) is selected with **enum test**:

§138 ⟨ Test declarations 121 ⟩ +≡

```
/* Test to perform. */
enum test {
    TST_CORRECTNESS, /* Default tests. */
    TST_OVERFLOW, /* Stack overflow test. */
    TST_NULL /* No test, just overhead. */
};
```

The program's entire behavior is controlled by **struct test_options**, defined as follows:

§139 ⟨ Test declarations 121 ⟩ +≡

```
/* Program options. */
struct test_options {
    enum test test; /* Test to perform. */
    enum insert_order insert_order; /* Insertion order. */
    enum delete_order delete_order; /* Deletion order. */

    enum mt_policy alloc_policy; /* Allocation policy. */
    int alloc_arg[2]; /* Policy arguments. */
    int alloc_incr; /* Amount to increment alloc_arg each iteration. */

    int node_cnt; /* Number of nodes in tree. */
    int iter_cnt; /* Number of runs. */

    int seed_given; /* Seed provided on command line? */
    unsigned seed; /* Random number seed. */

    int verbosity; /* Verbosity level, 0=default. */
    int nonstop; /* Don't stop after one error? */
};
```

The *main*() function for the test program is perhaps a bit long, but simple. It begins by parsing the command line and allocating memory, then repeats a loop once for each repetition of the test. Within the loop, an insertion and a deletion order are selected, the memory tracker is set up, and test function (either *test*() or *test_overflow*()) is called.

§140 ⟨ Test main program 140 ⟩ ≡

```
int main (int argc, char *argv[]) {
    struct test_options opts; /* Command-line options. */
    int *insert, *delete; /* Insertion and deletion orders. */
    int success; /* Everything okay so far? */
```

```
/* Initialize pgm_name, using argv[0] if sensible. */
pgm_name = argv[0] != NULL && argv[0][0] != '\0' ? argv[0] : "bsttest";
/* Parse command line into options. */
parse_command_line (argv, &opts);
if (opts.verbosity >= 0)
    fputs ("bsttest for GNU libavl 2.0.2; use -help to get help.\n", stdout);
if (!opts.seed_given) opts.seed = time_seed () % 32768u;
insert = xmalloc (sizeof *insert * opts.node_cnt);
delete = xmalloc (sizeof *delete * opts.node_cnt);
/* Run the tests. */
success = 1;
while (opts.iter_cnt--) {
    struct mt_allocator *alloc;
    if (opts.verbosity >= 0) {
        printf ("Testing seed=%u", opts.seed);
        if (opts.alloc_incr) printf (", alloc arg=%d", opts.alloc_arg[0]);
        printf ("...\n");
        fflush (stdout);
    }
    /* Generate insertion and deletion order.
       Seed them separately to ensure deletion order is
       independent of insertion order. */
    srand (opts.seed);
    gen_insertions (opts.node_cnt, opts.insert_order, insert);
    srand (++opts.seed);
    gen_deletions (opts.node_cnt, opts.delete_order, insert, delete);
    if (opts.verbosity >= 1) {
        int i;
        printf ("  Insertion order:");
        for (i = 0; i < opts.node_cnt; i++)
            printf (" %d", insert[i]);
        printf (".\n");
        if (opts.test == TST_CORRECTNESS) {
            printf ("Deletion order:");
            for (i = 0; i < opts.node_cnt; i++)
                printf (" %d", delete[i]);
            printf (".\n");
        }
    }
    alloc = mt_create (opts.alloc_policy, opts.alloc_arg, opts.verbosity);
    {
        int okay;
        struct libavl_allocator *a = mt_allocator (alloc);
        switch (opts.test) {
```

```
        case TST_CORRECTNESS:
          okay = test_correctness (a, insert, delete, opts.node_cnt, opts.verbosity);
          break;
        case TST_OVERFLOW:
          okay = test_overflow (a, insert, opts.node_cnt, opts.verbosity);
          break;
        case TST_NULL: okay = 1; break;
        default: assert (0);
        }
      if (okay) {
        if (opts.verbosity >= 1)
          printf ("  No errors.\n");
      } else {
        success = 0;
        printf ("  Error!\n");
      }
    }
    mt_destroy (alloc);
    opts.alloc_arg[0] += opts.alloc_incr;
    if (!success && !opts.nonstop)
      break;
  }
  free (delete);
  free (insert);
  return success ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

This code is included in §97.

The main program initializes our single global variable, *pgm_name*, which receives the name of the program at start of execution:

§141 ⟨ Test declarations 121 ⟩ +≡

```
/* Program name. */
char *pgm_name;
```

4.15 Additional Exercises

Exercises:

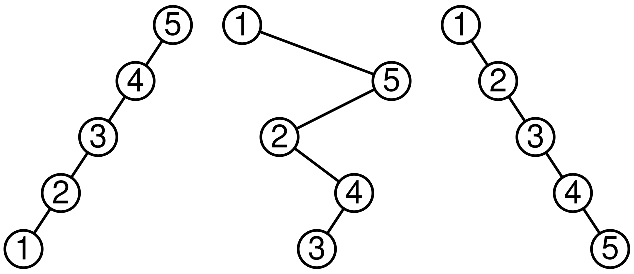
1. Sentinels were a main theme of the chapter before this one. Figure out how to apply sentinel techniques to binary search trees. Write routines for search and insertion in such a binary search tree with sentinel. Test your functions. (You need not make your code fully generic; e.g., it is acceptable to "hard-code" the data type stored in the tree.)

5 AVL Trees

In the last chapter, we designed and implemented a table ADT using binary search trees. We were interested in binary trees from the beginning because of their promise of speed compared to linear lists.

But we only get these speed improvements if our binary trees are arranged more or less optimally, with the tree's height as small as possible. If we insert and delete items in the tree in random order, then chances are that we'll come pretty close to this optimal tree.[1]

In "pathological" cases, search within binary search trees can be as slow as sequential search, or even slower when the extra bookkeeping needed for a binary tree is taken into account. For example, after inserting items into a BST in sorted order, we get something like the vines on the left and the right below. The BST in the middle below illustrates a more unusual case, a "zig-zag" BST that results from inserting items from alternating ends of an ordered list.



Unfortunately, these pathological cases can easily come up in practice, because sorted data in the input to a program is common. We could periodically balance the tree using some heuristic to detect that it is "too tall". In the last chapter, in fact, we used a weak version of this idea, rebalancing when a stack overflow force it. We could abandon the idea of a binary search tree, using some other data structure. Finally, we could adopt some modifications to binary search trees that prevent the pathological case from occurring.

For the remainder of this book, we're only interested in the latter choice. We'll look at two sets of rules that, when applied to the basic structure of a binary search tree, ensure that the tree's height is kept within a constant factor of the minimum value. Although this is not as good as keeping the BST's height at its minimum, it comes pretty close, and the required operations are much faster. A tree arranged to rules such as these is called a **balanced tree**. The operations used for minimizing tree height are said to **rebalance** the tree, even though this is different from the sort of rebalancing we did in the previous chapter, and are said to maintain the tree's "balance."

A balanced tree arranged according to the first set of rebalancing rules that we'll examine is called an **AVL tree**, after its inventors, G. M. Adel'son-Vel'skiĭ and E. M. Landis. AVL trees are the subject of this chapter, and the next chapter will discuss red-black trees, another type of balanced tree.

In the following sections, we'll construct a table implementation based on AVL trees. Here's an outline of the AVL code:

§142 ⟨ `avl.h` 142 ⟩ ≡

[1] This seems true intuitively, but there are some difficult mathematics in this area. For details, refer to [Knuth 1998b] theorem 6.2.2H, [Knuth 1977], and [Knuth 1978].

⟨ License 1 ⟩
#ifndef AVL_H
#define AVL_H 1
#include ⟨ stddef.h ⟩
⟨ Table types; tbl ⇒ avl 14 ⟩
⟨ BST maximum height; bst ⇒ avl 28 ⟩
⟨ BST table structure; bst ⇒ avl 27 ⟩
⟨ AVL node structure 144 ⟩
⟨ BST traverser structure; bst ⇒ avl 61 ⟩
⟨ Table function prototypes; tbl ⇒ avl 15 ⟩
#endif /* avl.h */

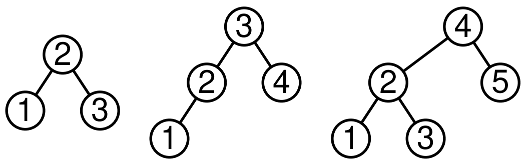
§143 ⟨ `avl.c` 143 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include ⟨ string.h ⟩
#include "avl.h"
⟨ AVL functions 145 ⟩

See also: [Knuth 1998b], sections 6.2.2 and 6.2.3; [Cormen 1990], section 13.4.

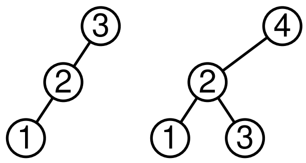
5.1 Balancing Rule

A binary search tree is an AVL tree if the difference in height between the subtrees of each of its nodes is between $^-1$ and $^+1$. Said another way, a BST is an AVL tree if it is an empty tree or if its subtrees are AVL trees and the difference in height between its left and right subtree is between $^-1$ and $^+1$.

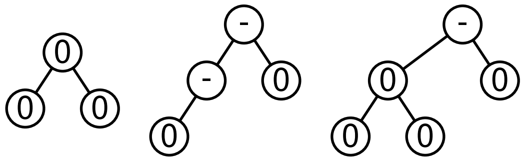
Here are some AVL trees:



These binary search trees are not AVL trees:



In an AVL tree, the height of a node's right subtree minus the height of its left subtree is called the node's **balance factor**. Balance factors are always $^-1$, 0, or $^+1$. They are often represented as one of the single characters $-$, 0, or $+$. Because of their importance in AVL trees, balance factors will often be shown in this chapter in AVL tree diagrams along with or instead of data items. Here are the AVL trees from above, but with balance factors shown in place of data values:



See also: [Knuth 1998b], section 6.2.3.

5.1.1 Analysis

How good is the AVL balancing rule? That is, before we consider how much complication it adds to BST operations, what does this balancing rule guarantee about performance? This is a simple question only if you're familiar with the mathematics behind computer science. For our purposes, it suffices to state the results:

> An AVL tree with n nodes has height between $\log_2(n+1)$ and $1.44\log_2(n+2)-.328$. An AVL tree with height h has between $2^{h+.328}/1.44$ and 2^h-1 nodes.
>
> For comparison, an optimally balanced BST with n nodes has height $\lceil\log_2(n+1)\rceil$. An optimally balanced BST with height h has between 2^{h-1} and 2^h-1 nodes.

The average speed of a search in a binary tree depends on the tree's height, so the results above are quite encouraging: an AVL tree will never be more than about 50% taller than the corresponding optimally balanced tree. Thus, we have a guarantee of good performance even in the worst case, and optimal performance in the best case.

See also: [Knuth 1998b], theorem 6.2.3A.

5.2 Data Types

We need to define data types for AVL trees like we did for BSTs. AVL tree nodes contain all the fields that a BST node does, plus a field recording its balance factor:

§144 ⟨ AVL node structure 144 ⟩ ≡

```
/* An AVL tree node. */
struct avl_node {
    struct avl_node *avl_link[2]; /* Subtrees. */
    void *avl_data; /* Pointer to data. */
    signed char avl_balance; /* Balance factor. */
};
```

This code is included in §142.

We're using *avl_* as the prefix for all AVL-related identifiers.

The other data structures for AVL trees are the same as for BSTs.

5.3 Operations

Now we'll implement for AVL trees all the operations that we did for BSTs. Here's the outline. Creation and search of AVL trees is exactly like that for plain BSTs, and the generic table functions for insertion convenience, assertion, and memory allocation are still relevant, so we just reuse the code. Of the remaining functions, we will write new implementations of the insertion and deletion functions and revise the traversal and copy functions.

§145 ⟨ AVL functions 145 ⟩ ≡

⟨ BST creation function; bst ⇒ avl 30 ⟩
⟨ BST search function; bst ⇒ avl 31 ⟩
⟨ AVL item insertion function 146 ⟩
⟨ Table insertion convenience functions; tbl ⇒ avl 592 ⟩
⟨ AVL item deletion function 164 ⟩
⟨ AVL traversal functions 178 ⟩
⟨ AVL copy function 185 ⟩
⟨ BST destruction function; bst ⇒ avl 84 ⟩
⟨ Default memory allocation functions; tbl ⇒ avl 6 ⟩
⟨ Table assertion functions; tbl ⇒ avl 594 ⟩

This code is included in §143.

5.4 Insertion

The insertion function for unbalanced BSTs does not maintain the AVL balancing rule, so we have to write a new insertion function. But before we get into the nitty-gritty details, let's talk in generalities. This is time well spent because we will be able to apply many of the same insights to AVL deletion and insertion and deletion in red-black trees.

Conceptually, there are two stages to any insertion or deletion operation in a balanced tree. The first stage may lead to violation of the tree's balancing rule. If so, we fix it in the second stage. The insertion or deletion itself is done in the first stage, in much the same way as in an unbalanced BST, and we may also do a bit of additional bookkeeping work, such as updating balance factors in an AVL tree, or swapping node "colors" in red-black trees.

If the first stage of the operation does not lead to a violation of the tree's balancing rule, nothing further needs to be done. But if it does, the second stage rearranges nodes and modifies their attributes to restore the tree's balance. This process is said to **rebalance** the tree. The kinds of rebalancing that might be necessary depend on the way the operation is performed and the tree's balancing rule. A well-chosen balancing rule helps to minimize the necessity for rebalancing.

When rebalancing does become necessary in an AVL or red-black tree, its effects are limited to the nodes along or near the direct path from the inserted or deleted node up to the root of the tree. Usually, only one or two of these nodes are affected, but, at most, one simple manipulation is performed at each of the nodes along this path. This property ensures that balanced tree operations are efficient (see Exercise 1 for details).

That's enough theory for now. Let's return to discussing the details of AVL insertion. There are four steps in Libavl's implementation of AVL insertion:

1. **Search** for the location to insert the new item.
2. **Insert** the item as a new leaf.
3. **Update** balance factors in the tree that were changed by the insertion.
4. **Rebalance** the tree, if necessary.

Steps 1 and 2 are the same as for insertion into a BST. Step 3 performs the additional bookkeeping alluded to above in the general description of balanced tree operations. Finally, step 4 rebalances the tree, if necessary, to restore the AVL balancing rule.

The following sections will cover all the details of AVL insertion. For now, here's an outline of *avl_probe*():

§146 ⟨ AVL item insertion function 146 ⟩ ≡
```
void **avl_probe (struct avl_table *tree, void *item) {
  ⟨ avl_probe() local variables 147 ⟩
  assert (tree != NULL && item != NULL);
  ⟨ Step 1: Search AVL tree for insertion point 148 ⟩
  ⟨ Step 2: Insert AVL node 149 ⟩
  ⟨ Step 3: Update balance factors after AVL insertion 150 ⟩
  ⟨ Step 4: Rebalance after AVL insertion 151 ⟩
}
```
This code is included in §145.

§147 ⟨ *avl_probe*() local variables 147 ⟩ ≡
```
struct avl_node *y, *z; /* Top node to update balance factor, and parent. */
struct avl_node *p, *q; /* Iterator, and parent. */
struct avl_node *n; /* Newly inserted node. */
struct avl_node *w; /* New root of rebalanced subtree. */
int dir; /* Direction to descend. */
unsigned char da[AVL_MAX_HEIGHT]; /* Cached comparison results. */
int k = 0; /* Number of cached results. */
```
This code is included in §146, §301, and §419.

See also: [Knuth 1998b], algorithm 6.2.3A.

Exercises:

***1.** When rebalancing manipulations are performed on the chain of nodes from the inserted or deleted node to the root, no manipulation takes more than a fixed amount of time. In other words, individual manipulations do not involve any kind of iteration or loop. What can you conclude about the speed of an individual insertion or deletion in a large balanced tree, compared to the best-case speed of an operation for unbalanced BSTs?

5.4.1 Step 1: Search

The search step is an extended version of the corresponding code for BST insertion in ⟨ BST item insertion function 32 ⟩. The earlier code had only two variables to maintain: the current node the direction to descend from p. The AVL code does this, but it maintains some other variables, too. During each iteration of the **for** loop, p is the node we are examining, q is p's parent, y is the most recently examined node with nonzero balance factor, z is y's parent, and elements $0 \ldots k-1$ of array da[] record each direction descended, starting from z, in order to arrive at p. The purposes for many of these variables are surely uncertain right now, but they will become clear later.

§148 ⟨ Step 1: Search AVL tree for insertion point 148 ⟩ ≡
```
z = (struct avl_node *) &tree->avl_root;
y = tree->avl_root;
dir = 0;
for (q = z, p = y; p != NULL; q = p, p = p->avl_link[dir]) {
```

```
    int cmp = tree→avl_compare (item, p→avl_data, tree→avl_param);
    if (cmp == 0)
        return &p→avl_data;
    if (p→avl_balance != 0)
        z = q, y = p, k = 0;
    da[k++] = dir = cmp > 0;
}
```

This code is included in §146.

5.4.2 Step 2: Insert

Following the search loop, q is the last non-null node examined, so it is the parent of the node to be inserted. The code below creates and initializes a new node as a child of q on side *dir*, and stores a pointer to it into n. Compare this code for insertion to that within ⟨ BST item insertion function 32 ⟩.

§149 ⟨ Step 2: Insert AVL node 149 ⟩ ≡

```
n = q→avl_link[dir] = tree→avl_alloc→libavl_malloc (tree→avl_alloc, sizeof *n);
if (n == NULL)
    return NULL;
tree→avl_count++;
n→avl_data = item;
n→avl_link[0] = n→avl_link[1] = NULL;
n→avl_balance = 0;
if (y == NULL)
    return &n→avl_data;
```

This code is included in §146.

Exercises:

1. How can y be `NULL`? Why is this special-cased?

5.4.3 Step 3: Update Balance Factors

When we add a new node n to an AVL tree, the balance factor of n's parent must change, because the new node increases the height of one of the parent's subtrees. The balance factor of n's parent's parent may need to change, too, depending on the parent's balance factor, and in fact the change can propagate all the way up the tree to its root.

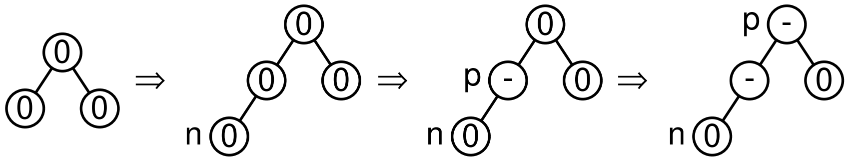
At each stage of updating balance factors, we are in a similar situation. First, we are examining a particular node p that is one of n's direct ancestors. The first time around, p is n's parent, the next time, if necessary, p is n's grandparent, and so on. Second, the height of one of p's subtrees has increased, and which one can be determined using $da[]$.

In general, if the height of p's left subtree increases, p's balance factor decreases. On the other hand, if the right subtree's height increases, p's balance factor increases. If we account for the three possible starting balance factors and the two possible sides, there are six possibilities. The three of these corresponding to an increase in one subtree's height are symmetric with the others that go along with an increase in the other subtree's height. We treat these three cases below.

Case 1: *p* has balance factor 0

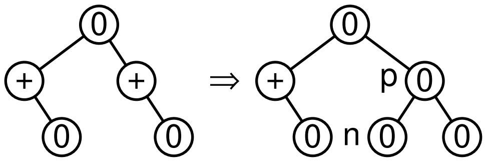
If p had balance factor 0, its new balance factor is $-$ or $+$, depending on the side of the root to which the node was added. After that, the change in height propagates up the tree to p's parent (unless p is the tree's root) because the height of the subtree rooted at p's parent has also increased.

The example below shows a new node n inserted as the left child of a node with balance factor 0. On the far left is the original tree before insertion; in the middle left is the tree after insertion but before any balance factors are adjusted; in the middle right is the tree after the first adjustment, with p as n's parent; on the far right is the tree after the second adjustment, with p as n's grandparent. Only in the trees on the far left and far right are all of the balance factors correct.



Case 2: *p*'s shorter subtree has increased in height

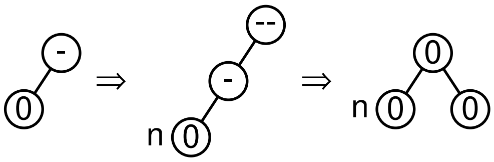
If the new node was added to p's shorter subtree, then the subtree has become more balanced and its balance factor becomes 0. If p started out with balance factor $+$, this means the new node is in p's left subtree. If p had a $-$ balance factor, this means the new node is in the right subtree. Since tree p has the same height as it did before, the change does not propagate up the tree any farther, and we are done. Here's an example that shows pre-insertion and post-balance factor updating views:



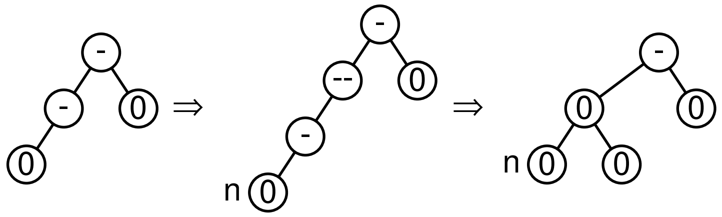
Case 3: *p*'s taller subtree has increased in height

If the new node was added on the taller side of a subtree with nonzero balance factor, the balance factor becomes $^{+}2$ or $^{-}2$. This is a problem, because balance factors in AVL trees must be between $^{-}1$ and $^{+}1$. We have to rebalance the tree in this case. We will cover rebalancing later. For now, take it on faith that rebalancing does not increase the height of subtree p as a whole, so there is no need to propagate changes any farther up the tree.

Here's an example of an insertion that leads to rebalancing. On the left is the tree before insertion; in the middle is the tree after insertion and updating balance factors; on the right is the tree after rebalancing to. The $^{-}2$ balance factor is shown as two minus signs ($--$). The rebalanced tree is the same height as the original tree before insertion.



As another demonstration that the height of a rebalanced subtree does not change after insertion, here's a similar example that has one more layer of nodes. The trees below follow the same pattern as the ones above, but the rebalanced subtree has a parent. Even though the tree's root has the wrong balance factor in the middle diagram, it turns out to be correct after rebalancing.



Implementation

Looking at the rules above, we can see that only in case 1, where p's balance factor is 0, do changes to balance factors continue to propagate upward in the tree. So we can start from n's parent and move upward in the tree, handling case 1 each time, until we hit a nonzero balance factor, handle case 2 or case 3 at that node, and we're done (except for possible rebalancing afterward).

Wait a second—there is no efficient way to move upward in a binary search tree![2] Fortunately, there is another approach we can use. Remember the extra code we put into ⟨ Step 1: Search AVL tree for insertion point 148 ⟩? This code kept track of the last node we'd passed through that had a nonzero balance factor as s. We can use s to move downward, instead of upward, through the nodes whose balance factors are to be updated.

Node s itself is the topmost node to be updated; when we arrive at node n, we know we're done. We also kept track of the directions we moved downward in $da[]$. Suppose that we've got a node p whose balance factor is to be updated and a direction d that we moved from it. We know that if we moved down to the left ($d == 0$) then the balance factor must be decreased, and that if we moved down to the right ($d == 1$) then the balance factor must be increased.

Now we have enough knowledge to write the code to update balance factors. The results are almost embarrassingly short:

§150 ⟨ Step 3: Update balance factors after AVL insertion 150 ⟩ ≡

```
for (p = y, k = 0; p != n; p = p→avl_link[da[k]], k++)
    if (da[k] == 0)
        p→avl_balance−−;
    else p→avl_balance++;
```

This code is included in §146, §301, and §419.

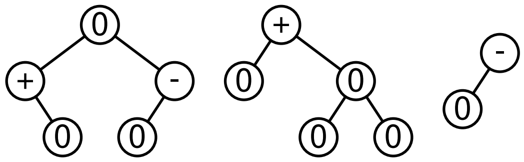
Now p points to the new node as a consequence of the loop's exit condition. Variable p will not be modified again in this function, so it is used in the function's final **return** statement to take the address of the new node's *avl_data* member (see ⟨ AVL item insertion function 146 ⟩ above).

[2] We could make a list of the nodes as we move down the tree and reuse it on the way back up. We'll do that for deletion, but there's a simpler way for insertion, so keep reading.

Exercises:

1. Can case 3 be applied to the parent of the newly inserted node?

2. For each of the AVL trees below, add a new node with a value smaller than any already in the tree and update the balance factors of the existing nodes. For each balance factor that changes, indicate the numbered case above that applies. Which of the trees require rebalancing after the insertion?



3. Earlier versions of LIBAVL used **char**s, not **unsigned char**s, to cache the results of comparisons, as the elements of *da*[] are used here. At some warning levels, this caused the GNU C compiler to emit the warning "array subscript has type 'char'" when it encountered expressions like $q \rightarrow avl\_link[da[k]]$. Explain why this can be a useful warning message.

4. If our AVL trees won't ever have a height greater than 32, then we can portably use the bits in a single **unsigned long** to compactly store what the entire *da*[] array does. Write a new version of step 3 to use this form, along with any necessary modifications to other steps and *avl_probe*()'s local variables.

5.4.4 Step 4: Rebalance

We've covered steps 1 through 3 so far. Step 4, rebalancing, is somewhat complicated, but it's the key to the entire insertion procedure. It is also similar to, but simpler than, other rebalancing procedures we'll see later. As a result, we're going to discuss it in detail. Follow along carefully and it should all make sense.

Before proceeding, let's briefly review the circumstances under which we need to rebalance. Looking back a few sections, we see that there is only one case where this is required: case 3, when the new node is added in the taller subtree of a node with nonzero balance factor.

Case 3 is the case where y has a $^-2$ or $^+2$ balance factor after insertion. For now, we'll just consider the $^-2$ case, because we can write code for the $^+2$ case later in a mechanical way by applying the principle of symmetry. In accordance with this idea, step 4 branches into three cases immediately, one for each rebalancing case and a third that just returns from the function if no rebalancing is necessary:

§151 ⟨ Step 4: Rebalance after AVL insertion 151 ⟩ ≡
if ($y \rightarrow avl\_balance == {}^-2$)
    { ⟨ Rebalance AVL tree after insertion in left subtree 152 ⟩ }
else if ($y \rightarrow avl\_balance == {}^+2$)
    { ⟨ Rebalance AVL tree after insertion in right subtree 157 ⟩ }
else return &$n \rightarrow avl\_data$;

See also §153 and §154.
This code is included in §146.

We will call y's left child x. The new node is somewhere in the subtrees of x. There are now only two cases of interest, distinguished on whether x has a + or − balance factor. These cases are almost entirely separate:

§152 ⟨ Rebalance AVL tree after insertion in left subtree 152 ⟩ ≡
struct avl_node $*x = y \rightarrow avl\_link[0]$;
if ($x \rightarrow avl\_balance$ == $^-1$)
  { ⟨ Rotate right at y in AVL tree 155 ⟩ }
else { ⟨ Rotate left at x then right at y in AVL tree 156 ⟩ }

This code is included in §151 and §162.

In either case, w receives the root of the rebalanced subtree, which is used to update the parent's pointer to the subtree root (recall that z is the parent of y):

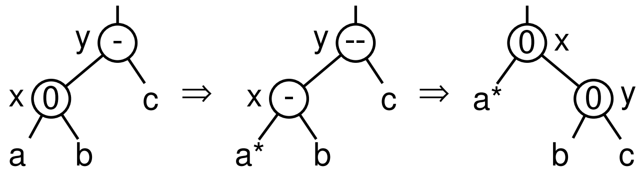
§153 ⟨ Step 4: Rebalance after AVL insertion 151 ⟩ +≡
$z \rightarrow avl\_link[y$!= $z \rightarrow avl\_link[0]] = w$;

Finally, we increment the generation number, because the tree's structure has changed. Then we're done and we return to the caller:

§154 ⟨ Step 4: Rebalance after AVL insertion 151 ⟩ +≡
$tree \rightarrow avl\_generation$++;
return &$n \rightarrow avl\_data$;

Case 1: *x* has − balance factor

For a − balance factor, we just rotate right at y. Then the entire process, including insertion and rebalancing, looks like this:



This figure also introduces some new graphical conventions. When both balance factors and node labels are shown in a figure, node labels are shown beside the node circles, instead of inside them. Second, the change in subtree a between the first and second diagrams is indicated by an asterisk (*).[3] In this case, it indicates that the new node was inserted in subtree a.

The code here is similar to *rotate_right*() in the solution to Exercise 4.3-2:

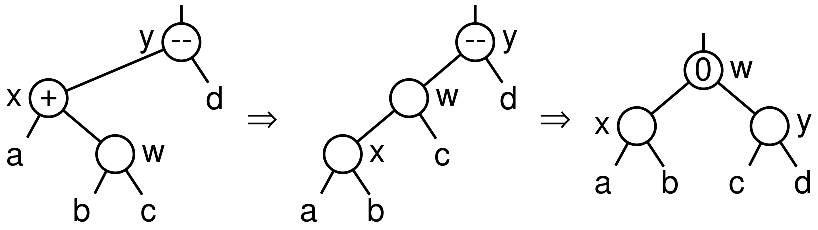
§155 ⟨ Rotate right at y in AVL tree 155 ⟩ ≡
$w = x$;
$y \rightarrow avl\_link[0] = x \rightarrow avl\_link[1]$;
$x \rightarrow avl\_link[1] = y$;
$x \rightarrow avl\_balance = y \rightarrow avl\_balance = 0$;

This code is included in §152 and §529.

[3] A "prime" (′) is traditional, but primes are easy to overlook.

Case 2: x has + balance factor

This case is just a little more intricate. First, let x's right child be w. Either w is the new node, or the new node is in one of w's subtrees. To restore balance, we rotate left at x, then rotate right at y (this is a kind of "double rotation"). The process, starting just after the insertion and showing the results of each rotation, looks like this:



At the beginning, the figure does not show the balance factor of w. This is because there are three possibilities:

Case 2.1: w has balance factor 0.
This means that w is the new node. a, b, c, and d have height 0. After the rotations, x and y have balance factor 0.

Case 2.2: w has balance factor $-$.
a, b, and d have height $h > 0$, and c has height $h - 1$.

Case 2.3: w has balance factor +.
a, c, and d have height $h > 0$, and b has height $h - 1$.

§156 ⟨ Rotate left at x then right at y in AVL tree 156 ⟩ ≡

```
assert (x→avl_balance == +1);
w = x→avl_link[1];
x→avl_link[1] = w→avl_link[0];
w→avl_link[0] = x;
y→avl_link[0] = w→avl_link[1];
w→avl_link[1] = y;
if (w→avl_balance == −1) x→avl_balance = 0, y→avl_balance = +1;
else if (w→avl_balance == 0) x→avl_balance = y→avl_balance = 0;
else /* w→avl_balance == +1 */ x→avl_balance = −1, y→avl_balance = 0;
w→avl_balance = 0;
```

This code is included in §152, §177, §307, §427, and §530.

Exercises:

1. Why can't the new node be x rather than a node in x's subtrees?

2. Why can't x have a 0 balance factor?

3. For each subcase of case 2, draw a figure like that given for generic case 2 that shows the specific balance factors at each step.

4. Explain the expression *z→avl_link*[y != *z→avl_link*[0]] = w in the second part of ⟨ Step 4: Rebalance after AVL insertion 151 ⟩ above. Why would it be a bad idea to substitute the apparent equivalent *z→avl_link*[y == *z→avl_link*[1]] = w?

5. Suppose that we wish to make a copy of an AVL tree, preserving the original tree's shape, by inserting nodes from the original tree into a new tree, using *avl_probe*(). Will inserting the original tree's nodes in level order (see the answer to Exercise 4.7-4) have the desired effect?

5.4.5 Symmetric Case

Finally, we need to write code for the case that we chose not to discuss earlier, where the insertion occurs in the right subtree of y. All we have to do is invert the signs of balance factors and switch *avl_link*[] indexes between 0 and 1. The results are this:

§157 ⟨ Rebalance AVL tree after insertion in right subtree 157 ⟩ ≡
struct avl_node $*x$ = y→*avl_link*[1];
if (x→*avl_balance* == $^+1$)
    { ⟨ Rotate left at y in AVL tree 158 ⟩ }
else { ⟨ Rotate right at x then left at y in AVL tree 159 ⟩ }

This code is included in §151 and §162.

§158 ⟨ Rotate left at y in AVL tree 158 ⟩ ≡
w = x;
y→*avl_link*[1] = x→*avl_link*[0];
x→*avl_link*[0] = y;
x→*avl_balance* = y→*avl_balance* = 0;

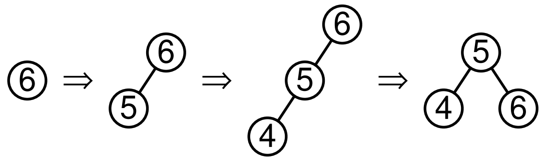
This code is included in §157 and §532.

§159 ⟨ Rotate right at x then left at y in AVL tree 159 ⟩ ≡
assert (x→*avl_balance* == $^-1$);
w = x→*avl_link*[0];
x→*avl_link*[0] = w→*avl_link*[1];
w→*avl_link*[1] = x;
y→*avl_link*[1] = w→*avl_link*[0];
w→*avl_link*[0] = y;
if (w→*avl_balance* == $^+1$) x→*avl_balance* = 0, y→*avl_balance* = $^-1$;
else if (w→*avl_balance* == 0) x→*avl_balance* = y→*avl_balance* = 0;
else /* w→*avl_balance* == $^-1$ */ x→*avl_balance* = $^+1$, y→*avl_balance* = 0;
w→*avl_balance* = 0;

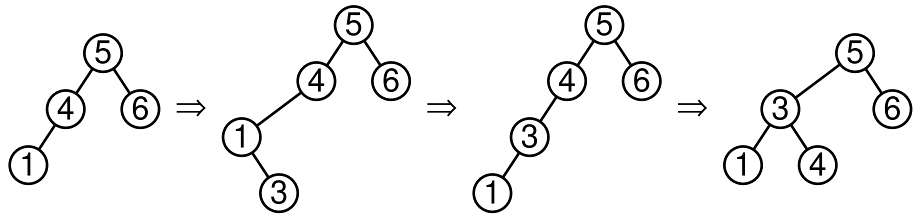
This code is included in §157, §174, §310, §428, and §533.

5.4.6 Example

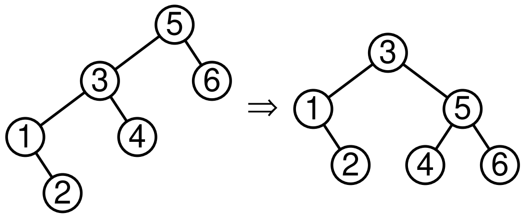
We're done with writing the code. Now, for clarification, let's run through an example designed to need lots of rebalancing along the way. Suppose that, starting with an empty AVL tree, we insert 6, 5, and 4, in that order. The first two insertions do not require rebalancing. After inserting 4, rebalancing is needed because the balance factor of node 6 would otherwise become $^-2$, an invalid value. This is case 1, so we perform a right rotation on 6. So far, the AVL tree has evolved this way:



If we now insert 1, then 3, a double rotation (case 2.1) becomes necessary, in which we rotate left at 1, then rotate right at 4:



Inserting a final item, 2, requires a right rotation (case 1) on 5:



5.4.7 Aside: Recursive Insertion

In previous sections we first looked at recursive approaches because they were simpler and more elegant than iterative solutions. As it happens, the reverse is true for insertion into an AVL tree. But just for completeness, we will now design a recursive implementation of *avl_probe*().

Our first task in such a design is to figure out what arguments and return value the recursive core of the insertion function will have. We'll begin by considering AVL insertion in the abstract. Our existing function *avl_probe*() works by first moving down the tree, from the root to a leaf, then back up the tree, from leaf to root, as necessary to adjust balance factors or rebalance. In the existing iterative version, down and up movement are implemented by pushing nodes onto and popping them off from a stack. In a recursive version, moving down the tree becomes a recursive call, and moving up the tree becomes a function return.

While descending the tree, the important pieces of information are the tree itself (to allow for comparisons to be made), the current node, and the data item we're inserting. The latter two items need to be modifiable by the function, the former because the tree rooted at the node may need to be rearranged during a rebalance, and the latter because of *avl_probe*()'s return value.

While ascending the tree, we'll still have access to all of this information, but, to allow for adjustment of balance factors and rebalancing, we also need to know whether the subtree visited in a nested call became taller. We can use the function's return value for this.

Finally, we know to stop moving down and start moving up when we find a null pointer in the tree, which is the place for the new node to be inserted. This suggests itself naturally as the test used to stop the recursion.

Here is an outline of a recursive insertion function directly corresponding to these considerations:

§160 ⟨ Recursive insertion into AVL tree 160 ⟩ ≡

```
static int probe (struct avl_table *tree, struct avl_node **p, void ***data) {
  struct avl_node *y; /* The current node; shorthand for *p. */
  assert (tree != NULL && p != NULL && data != NULL);
  y = *p;
  if (y == NULL)
    { ⟨ Found insertion point in recursive AVL insertion 161 ⟩ }
  else /* y != NULL */ { ⟨ Move down then up in recursive AVL insertion 162 ⟩ }
}
```

See also §163.

Parameter *p* is declared as a double pointer (**struct avl_node** ∗∗) and *data* as a triple pointer (**void** ∗∗∗). In both cases, this is because C passes arguments by value, so that a function modifying one of its arguments produces no change in the value seen in the caller. As a result, to allow a function to modify a scalar, a pointer to it must be passed as an argument; to modify a pointer, a double pointer must be passed; to modify a double pointer, a triple pointer must be passed. This can result in difficult-to-understand code, so it is often advisable to copy the dereferenced argument into a local variable for read-only use, as ∗*p* is copied into *y* here.

When the insertion point is found, a new node is created and a pointer to it stored into ∗*p*. Because the insertion causes the subtree to increase in height (from 0 to 1), a value of 1 is then returned:

§161 ⟨ Found insertion point in recursive AVL insertion 161 ⟩ ≡

```
y = *p = tree->avl_alloc->libavl_malloc (tree->avl_alloc, sizeof *y);
if (y == NULL) {
  *data = NULL;
  return 0;
}
y->avl_data = **data;
*data = &y->avl_data;
y->avl_link[0] = y->avl_link[1] = NULL;
y->avl_balance = 0;
tree->avl_count++;
tree->avl_generation++;
return 1;
```

This code is included in §160.

When we're not at the insertion point, we move down, then back up. Whether to move down to the left or the right depends on the value of the item to insert relative to the value in the current node *y*. Moving down is the domain of the recursive call to *probe*(). If the recursive call doesn't increase the height of a subtree of *y*, then there's nothing further to do, so we return immediately. Otherwise, on the way back up, it is necessary to at least adjust *y*'s balance factor, and possibly to rebalance as well. If only adjustment of the balance factor is necessary, it is done and the return value is based on whether this subtree has changed height in the process. Rebalancing is accomplished using the same code used in iterative insertion. A rebalanced subtree has the same height as before insertion, so the value returned is 0. The details are in the code itself:

§162 ⟨ Move down then up in recursive AVL insertion 162 ⟩ ≡
```
struct avl_node *w; /* New root of this subtree; replaces *p. */
int cmp;
cmp = tree→avl_compare (**data, y→avl_data, tree→avl_param);
if (cmp < 0) {
    if (probe (tree, &y→avl_link[0], data) == 0)
        return 0;
    if (y→avl_balance == +1) {
        y→avl_balance = 0;
        return 0;
    }
    else if (y→avl_balance == 0) {
        y→avl_balance = -1;
        return 1;
    } else { ⟨ Rebalance AVL tree after insertion in left subtree 152 ⟩ }
} else if (cmp > 0) {
    struct avl_node *r; /* Right child of y, for rebalancing. */
    if (probe (tree, &y→avl_link[1], data) == 0)
        return 0;
    if (y→avl_balance == -1) {
        y→avl_balance = 0;
        return 0;
    }
    else if (y→avl_balance == 0) {
        y→avl_balance = +1;
        return 1;
    } else { ⟨ Rebalance AVL tree after insertion in right subtree 157 ⟩ }
} else /* cmp == 0 */ {
    *data = &y→avl_data;
    return 0;
}
*p = w;
return 0;
```
This code is included in §160.

Finally, we need a wrapper function to start the recursion off correctly and deal with passing back the results:

§163 ⟨ Recursive insertion into AVL tree 160 ⟩ +≡
```
/* Inserts item into tree and returns a pointer to item's address.
   If a duplicate item is found in the tree,
   returns a pointer to the duplicate without inserting item.
   Returns NULL in case of memory allocation failure. */
void **avl_probe (struct avl_table *tree, void *item) {
```

```
    void **ret = &item;
    probe (tree, &tree→avl_root, &ret);
    return ret;
}
```

5.5 Deletion

Deletion in an AVL tree is remarkably similar to insertion. The steps that we go through are analogous:

1. **Search** for the item to delete.
2. **Delete** the item.
3. **Update** balance factors.
4. **Rebalance** the tree, if necessary.
5. **Finish up** and return.

The main difference is that, after a deletion, we may have to rebalance at more than one level of a tree, starting from the bottom up. This is a bit painful, because it means that we have to keep track of all the nodes that we visit as we search for the node to delete, so that we can then move back up the tree. The actual updating of balance factors and rebalancing steps are similar to those used for insertion.

The following sections cover deletion from an AVL tree in detail. Before we get started, here's an outline of the function.

§164 ⟨ AVL item deletion function 164 ⟩ ≡

```
void *avl_delete (struct avl_table *tree, const void *item) {
    /* Stack of nodes. */
    struct avl_node *pa[AVL_MAX_HEIGHT]; /* Nodes. */
    unsigned char da[AVL_MAX_HEIGHT]; /* avl_link[] indexes. */
    int k; /* Stack pointer. */
    struct avl_node *p; /* Traverses tree to find node to delete. */
    int cmp; /* Result of comparison between item and p. */
    assert (tree != NULL && item != NULL);
    ⟨ Step 1: Search AVL tree for item to delete 165 ⟩
    ⟨ Step 2: Delete item from AVL tree 166 ⟩
    ⟨ Steps 3–4: Update balance factors and rebalance after AVL deletion 171 ⟩
    ⟨ Step 5: Finish up and return after AVL deletion 176 ⟩
}
```

This code is included in §145.

See also: [Knuth 1998b], pages 473–474; [Pfaff 1998].

5.5.1 Step 1: Search

The only difference between this search and an ordinary search in a BST is that we have to keep track of the nodes above the one we're deleting. We do this by pushing them onto the stack defined above. Each iteration through the loop compares *item* to *p*'s data, pushes the node onto the stack, moves down in the proper direction. The first trip through the loop is something of an exception: we hard-code the comparison result to $^{-}1$ so that the pseudo-root node is always the topmost node on the stack. When we find a match, we set *item* to the actual data item found, so that we can return it later.

§165 ⟨ Step 1: Search AVL tree for item to delete 165 ⟩ ≡
```
k = 0;
p = (struct avl_node *) &tree->avl_root;
for (cmp = -1; cmp != 0; cmp = tree->avl_compare (item, p->avl_data, tree->avl_param)) {
    int dir = cmp > 0;

    pa[k] = p;
    da[k++] = dir;

    p = p->avl_link[dir];
    if (p == NULL)
        return NULL;
}
item = p->avl_data;
```
This code is included in §164 and §220.

5.5.2 Step 2: Delete

At this point, we've identified p as the node to delete. The node on the top of the stack, $da[k - 1]$, is p's parent node. There are the same three cases we saw in deletion from an ordinary BST (see Section 4.8 [Deleting from a BST], page 39), with the addition of code to copy balance factors and update the stack.

The code for selecting cases is the same as for BSTs:

§166 ⟨ Step 2: Delete item from AVL tree 166 ⟩ ≡
```
if (p->avl_link[1] == NULL)
    { ⟨ Case 1 in AVL deletion 168 ⟩ }
else {
    struct avl_node *r = p->avl_link[1];
    if (r->avl_link[0] == NULL)
        { ⟨ Case 2 in AVL deletion 169 ⟩ }
    else { ⟨ Case 3 in AVL deletion 170 ⟩ }
}
```
See also §167.
This code is included in §164.

Regardless of the case, we are in the same situation after the deletion: node p has been removed from the tree and the stack contains k nodes at which rebalancing may be necessary. Later code may change p to point elsewhere, so we free the node immediately. A pointer to the item data has already been saved in *item* (see page 123):

§167 ⟨ Step 2: Delete item from AVL tree 166 ⟩ +≡
```
tree->avl_alloc->libavl_free (tree->avl_alloc, p);
```

Case 1: p has no right child

If p has no right child, then we can replace it with its left child, the same as for BSTs (see page 39).

§168 ⟨ Case 1 in AVL deletion 168 ⟩ ≡

```
pa[k - 1]->avl_link[da[k - 1]] = p->avl_link[0];
```

This code is included in §166.

Case 2: p's right child has no left child

If p has a right child r, which in turn has no left child, then we replace p by r, attaching p's left child to r, as we would in an unbalanced BST (see page 40). In addition, r acquires p's balance factor, and r must be added to the stack of nodes above the deleted node.

§169 ⟨ Case 2 in AVL deletion 169 ⟩ ≡

```
r->avl_link[0] = p->avl_link[0];
r->avl_balance = p->avl_balance;
pa[k - 1]->avl_link[da[k - 1]] = r;
da[k] = 1;
pa[k++] = r;
```

This code is included in §166.

Case 3: p's right child has a left child

If p's right child has a left child, then this is the third and most complicated case. On the other hand, as a modification from the third case in an ordinary BST deletion (see page 40), it is rather simple. We're deleting the inorder successor of p, so we push the nodes above it onto the stack. The only trickery is that we do not know in advance the node that will replace p, so we reserve a spot on the stack for it ($da[j]$) and fill it in later:

§170 ⟨ Case 3 in AVL deletion 170 ⟩ ≡

```
struct avl_node *s;
int j = k++;
for (;;) {
    da[k] = 0;
    pa[k++] = r;
    s = r->avl_link[0];
    if (s->avl_link[0] == NULL)
        break;

    r = s;
}
s->avl_link[0] = p->avl_link[0];
r->avl_link[0] = s->avl_link[1];
s->avl_link[1] = p->avl_link[1];
s->avl_balance = p->avl_balance;
pa[j - 1]->avl_link[da[j - 1]] = s;
da[j] = 1;
pa[j] = s;
```

This code is included in §166.

Exercises:

1. Write an alternate version of ⟨ Case 3 in AVL deletion 170 ⟩ that moves data instead of pointers, as in Exercise 4.8-2.

2. Why is it important that the item data was saved earlier? (Why couldn't we save it just before freeing the node?)

5.5.3 Step 3: Update Balance Factors

When we updated balance factors in insertion, we were lucky enough to know in advance which ones we'd need to update. Moreover, we never needed to rebalance at more than one level in the tree for any one insertion. These two factors conspired in our favor to let us do all the updating of balance factors at once from the top down.

Everything is not quite so simple in AVL deletion. We don't have any easy way to figure out during the search process which balance factors will need to be updated, and for that matter we may need to perform rebalancing at multiple levels. Our strategy must change.

This new approach is not fundamentally different from the previous one. We work from the bottom up instead of from the top down. We potentially look at each of the nodes along the direct path from the deleted node to the tree's root, starting at $pa[k-1]$, the parent of the deleted node. For each of these nodes, we adjust its balance factor and possibly perform rebalancing. After that, if we're lucky, this was enough to restore the tree's balancing rule, and we are finished with updating balance factors and rebalancing. Otherwise, we look at the next node, repeating the process.

Here is the loop itself with the details abstracted out:

§171 ⟨ Steps 3–4: Update balance factors and rebalance after AVL deletion 171 ⟩ ≡
```
assert (k > 0);
while (--k > 0) {
    struct avl_node *y = pa[k];
    if (da[k] == 0)
        { ⟨ Update y's balance factor after left-side AVL deletion 172 ⟩ }
    else { ⟨ Update y's balance factor after right-side AVL deletion 177 ⟩ }
}
```
This code is included in §164.

The reason this works is the loop invariants. That is, because each time we look at a node in order to update its balance factor, the situation is the same. In particular, if we're looking at a node $pa[k]$, then we know that it's because the height of its subtree on side $da[k]$ decreased, so that the balance factor of node $pa[k]$ needs to be updated. The rebalancing operations we choose reflect this invariant: there are sometimes multiple valid ways to rebalance at a given node and propagate the results up the tree, but only one way to do this while maintaining the invariant. (This is especially true in red-black trees, for which we will develop code for two possible invariants under insertion and deletion.)

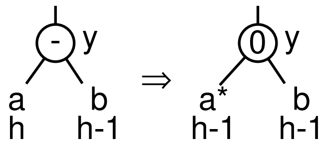
Updating the balance factor of a node after deletion from its left side and right side are symmetric, so we'll discuss only the left-side case here and construct the code for the right-side case later. Suppose we have a node y whose left subtree has decreased in height.

In general, this increases its balance factor, because the balance factor of a node is the height of its right subtree minus the height of its left subtree. More specifically, there are three cases, treated individually below.

Case 1: y has − balance factor

If y started with a − balance factor, then its left subtree was taller than its right subtree. Its left subtree has decreased in height, so the two subtrees must now be the same height and we set y's balance factor to 0. This is between $^{-}1$ and $^{+}1$, so there is no need to rebalance at y. However, binary tree y has itself decreased in height, so that means that we must rebalance the AVL tree above y as well, so we continue to the next iteration of the loop.

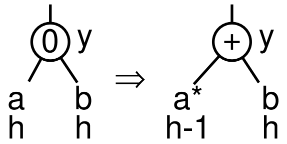
The diagram below may help in visualization. On the left is shown the original configuration of a subtree, where subtree a has height h and subtree b has height $h - 1$. The height of a nonempty binary tree is one plus the larger of its subtrees' heights, so tree y has height $h + 1$. The diagram on the right shows the situation after a node has been deleted from a, reducing that subtree's height. The new height of tree y is $(h - 1) + 1 \equiv h$.



Case 2: y has 0 balance factor

If y started with a 0 balance factor, and its left subtree decreased in height, then the result is that its right subtree is now taller than its left subtree, so the new balance factor is +. However, the overall height of binary tree y has not changed, so no balance factors above y need to be changed, and we are done, hence we **break** to exit the loop.

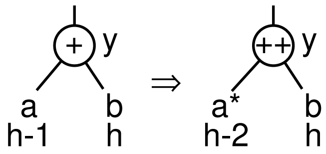
Here's the corresponding diagram, similar to the one for the previous case. The height of tree y on both sides of the diagram is $h + 1$, since y's taller subtree in both cases has height h.



Case 3: y has + balance factor

Otherwise, y started with a + balance factor, so the decrease in height of its left subtree, which was already shorter than its right subtree, causes a violation of the AVL constraint with a $^{+}2$ balance factor. We need to rebalance. After rebalancing, we may or may not have to rebalance further up the tree.

Here's a diagram of what happens to forcing rebalancing:



Implementation

The implementation is straightforward:

§172 ⟨ Update y's balance factor after left-side AVL deletion 172 ⟩ ≡
$y \rightarrow avl\_balance$++;
if ($y \rightarrow avl\_balance$ == $^+1$)
    break;
else if ($y \rightarrow avl\_balance$ == $^+2$)
    { ⟨ Step 4: Rebalance after AVL deletion 173 ⟩ }

This code is included in §171.

5.5.4 Step 4: Rebalance

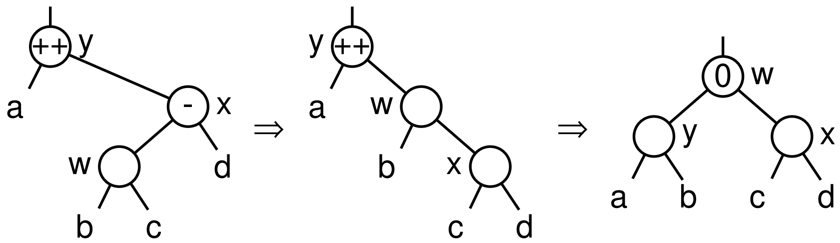
Now we have to write code to rebalance when it becomes necessary. We'll use rotations to do this, as before. Again, we'll distinguish the cases on the basis of x's balance factor, where x is y's right child:

§173 ⟨ Step 4: Rebalance after AVL deletion 173 ⟩ ≡
struct avl_node $*x = y \rightarrow avl\_link[1]$;
if ($x \rightarrow avl\_balance$ == $^-1$)
    { ⟨ Left-side rebalancing case 1 in AVL deletion 174 ⟩ }
else { ⟨ Left-side rebalancing case 2 in AVL deletion 175 ⟩ }

This code is included in §172.

Case 1: x has $-$ balance factor

If x has a $-$ balance factor, we handle rebalancing in a manner analogous to case 2 for insertion. In fact, we reuse the code. We rotate right at x, then left at y. w is the left child of x. The two rotations look like this:

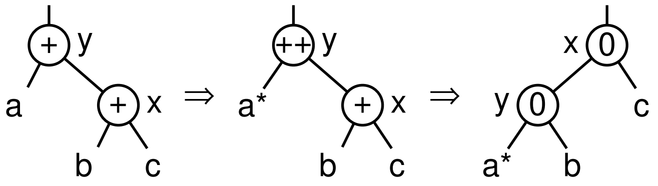


§174 ⟨ Left-side rebalancing case 1 in AVL deletion 174 ⟩ ≡
struct avl_node $*w$;
⟨ Rotate right at x then left at y in AVL tree 159 ⟩
$pa[k-1] \rightarrow avl\_link[da[k-1]] = w$;

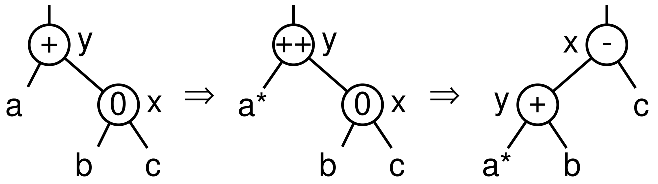
This code is included in §173.

Case 2: *x* has + or 0 balance factor

When x's balance factor is +, the needed treatment is analogous to Case 1 for insertion. We simply rotate left at y and update the pointer to the subtree, then update balance factors. The deletion and rebalancing then look like this:



When x's balance factor is 0, we perform the same rotation, but the height of the overall subtree does not change, so we're done and can exit the loop with **break**. Here's what the deletion and rebalancing look like for this subcase:



§175 ⟨ Left-side rebalancing case 2 in AVL deletion 175 ⟩ ≡

```
y->avl_link[1] = x->avl_link[0];
x->avl_link[0] = y;
pa[k - 1]->avl_link[da[k - 1]] = x;
if (x->avl_balance == 0) {
    x->avl_balance = -1;
    y->avl_balance = +1;
    break;
}
else x->avl_balance = y->avl_balance = 0;
```

This code is included in §173.

Exercises:

1. In ⟨ Step 4: Rebalance after AVL deletion 173 ⟩, we refer to fields in x, the right child of y, without checking that y has a non-null right child. Why can we assume that node x is non-null?

2. Describe the shape of a tree that might require rebalancing at every level above a particular node. Give an example.

5.5.5 Step 5: Finish Up

§176 ⟨ Step 5: Finish up and return after AVL deletion 176 ⟩ ≡

```
tree->avl_count--;
tree->avl_generation++;
return (void *) item;
```

This code is included in §164.

5.5.6 Symmetric Case

Here's the code for the symmetric case, where the deleted node was in the right subtree of its parent.

§177 ⟨ Update y's balance factor after right-side AVL deletion 177 ⟩ ≡

```
y→avl_balance−−;
if (y→avl_balance == −1)
    break;
else if (y→avl_balance == −2) {
    struct avl_node *x = y→avl_link[0];
    if (x→avl_balance == +1) {
        struct avl_node *w;
        ⟨ Rotate left at x then right at y in AVL tree 156 ⟩
        pa[k − 1]→avl_link[da[k − 1]] = w;
    } else {
        y→avl_link[0] = x→avl_link[1];
        x→avl_link[1] = y;
        pa[k − 1]→avl_link[da[k − 1]] = x;
        if (x→avl_balance == 0) {
            x→avl_balance = +1;
            y→avl_balance = −1;
            break;
        }
        else x→avl_balance = y→avl_balance = 0;
    }
}
```

This code is included in §171.

5.6 Traversal

Traversal is largely unchanged from BSTs. However, we can be confident that the tree won't easily exceed the maximum stack height, because of the AVL balance condition, so we can omit checking for stack overflow.

§178 ⟨ AVL traversal functions 178 ⟩ ≡

```
⟨ BST traverser refresher; bst ⇒ avl 62 ⟩
⟨ BST traverser null initializer; bst ⇒ avl 64 ⟩
⟨ AVL traverser least-item initializer 180 ⟩
⟨ AVL traverser greatest-item initializer 181 ⟩
⟨ AVL traverser search initializer 182 ⟩
⟨ AVL traverser insertion initializer 179 ⟩
⟨ BST traverser copy initializer; bst ⇒ avl 69 ⟩
⟨ AVL traverser advance function 183 ⟩
⟨ AVL traverser back up function 184 ⟩
⟨ BST traverser current item function; bst ⇒ avl 74 ⟩
⟨ BST traverser replacement function; bst ⇒ avl 75 ⟩
```

This code is included in §145 and §196.

We do need to make a new implementation of the insertion traverser initializer. Because insertion into an AVL tree is so complicated, we just write this as a wrapper to *avl_probe*(). There probably wouldn't be much of a speed improvement by inlining the code anyhow:

§179 ⟨ AVL traverser insertion initializer 179 ⟩ ≡

```
void *avl_t_insert (struct avl_traverser *trav, struct avl_table *tree, void *item) {
  void **p;
  assert (trav != NULL && tree != NULL && item != NULL);
  p = avl_probe (tree, item);
  if (p != NULL) {
    trav->avl_table = tree;
    trav->avl_node =
      ((struct avl_node *) ((char *) p - offsetof (struct avl_node, avl_data)));
    trav->avl_generation = tree->avl_generation - 1;
    return *p;
  } else {
    avl_t_init (trav, tree);
    return NULL;
  }
}
```

This code is included in §178.

We will present the rest of the modified functions without further comment.

§180 ⟨ AVL traverser least-item initializer 180 ⟩ ≡

```
void *avl_t_first (struct avl_traverser *trav, struct avl_table *tree) {
  struct avl_node *x;
  assert (tree != NULL && trav != NULL);
  trav->avl_table = tree;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;
  x = tree->avl_root;
  if (x != NULL)
    while (x->avl_link[0] != NULL) {
      assert (trav->avl_height < AVL_MAX_HEIGHT);
      trav->avl_stack[trav->avl_height++] = x;
      x = x->avl_link[0];
    }
  trav->avl_node = x;
  return x != NULL ? x->avl_data : NULL;
}
```

This code is included in §178.

§181 ⟨ AVL traverser greatest-item initializer 181 ⟩ ≡

```
void *avl_t_last (struct avl_traverser *trav, struct avl_table *tree) {
  struct avl_node *x;
  assert (tree != NULL && trav != NULL);
  trav->avl_table = tree;
```

```
    trav->avl_height = 0;
    trav->avl_generation = tree->avl_generation;

    x = tree->avl_root;
    if (x != NULL)
        while (x->avl_link[1] != NULL) {
            assert (trav->avl_height < AVL_MAX_HEIGHT);
            trav->avl_stack[trav->avl_height++] = x;
            x = x->avl_link[1];
        }
    trav->avl_node = x;

    return x != NULL ? x->avl_data : NULL;
}
```

This code is included in §178.

§182 ⟨ AVL traverser search initializer 182 ⟩ ≡

```
void *avl_t_find (struct avl_traverser *trav, struct avl_table *tree, void *item) {
    struct avl_node *p, *q;

    assert (trav != NULL && tree != NULL && item != NULL);
    trav->avl_table = tree;
    trav->avl_height = 0;
    trav->avl_generation = tree->avl_generation;
    for (p = tree->avl_root; p != NULL; p = q) {
        int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);

        if (cmp < 0) q = p->avl_link[0];
        else if (cmp > 0) q = p->avl_link[1];
        else /* cmp == 0 */ {
            trav->avl_node = p;
            return p->avl_data;
        }

        assert (trav->avl_height < AVL_MAX_HEIGHT);
        trav->avl_stack[trav->avl_height++] = p;
    }

    trav->avl_height = 0;
    trav->avl_node = NULL;
    return NULL;
}
```

This code is included in §178.

§183 ⟨ AVL traverser advance function 183 ⟩ ≡

```
void *avl_t_next (struct avl_traverser *trav) {
    struct avl_node *x;

    assert (trav != NULL);

    if (trav->avl_generation != trav->avl_table->avl_generation)
        trav_refresh (trav);

    x = trav->avl_node;
    if (x == NULL) {
```

```
        return avl_t_first (trav, trav->avl_table);
    } else if (x->avl_link[1] != NULL) {
        assert (trav->avl_height < AVL_MAX_HEIGHT);
        trav->avl_stack[trav->avl_height++] = x;
        x = x->avl_link[1];
        while (x->avl_link[0] != NULL) {
            assert (trav->avl_height < AVL_MAX_HEIGHT);
            trav->avl_stack[trav->avl_height++] = x;
            x = x->avl_link[0];
        }
    } else {
        struct avl_node *y;
        do {
            if (trav->avl_height == 0) {
                trav->avl_node = NULL;
                return NULL;
            }
            y = x;
            x = trav->avl_stack[--trav->avl_height];
        } while (y == x->avl_link[1]);
    }
    trav->avl_node = x;
    return x->avl_data;
}
```

This code is included in §178.

§184 ⟨ AVL traverser back up function 184 ⟩ ≡

```
void *avl_t_prev (struct avl_traverser *trav) {
    struct avl_node *x;
    assert (trav != NULL);
    if (trav->avl_generation != trav->avl_table->avl_generation)
        trav_refresh (trav);
    x = trav->avl_node;
    if (x == NULL) {
        return avl_t_last (trav, trav->avl_table);
    } else if (x->avl_link[0] != NULL) {
        assert (trav->avl_height < AVL_MAX_HEIGHT);
        trav->avl_stack[trav->avl_height++] = x;
        x = x->avl_link[0];
        while (x->avl_link[1] != NULL) {
            assert (trav->avl_height < AVL_MAX_HEIGHT);
            trav->avl_stack[trav->avl_height++] = x;
            x = x->avl_link[1];
        }
    } else {
        struct avl_node *y;
```

```
        do {
            if (trav→avl_height == 0) {
                trav→avl_node = NULL;
                return NULL;
            }
            y = x;
            x = trav→avl_stack[−−trav→avl_height];
        } while (y == x→avl_link[0]);
    }
    trav→avl_node = x;
    return x→avl_data;
}
```

This code is included in §178.

Exercises:

1. Explain the meaning of this ugly expression, used in *avl_t_insert*():

(**struct avl_node** ∗) ((**char** ∗) *p* − *offsetof* (**struct avl_node**, *avl_data*))

5.7 Copying

Copying an AVL tree is similar to copying a BST. The only important difference is that we have to copy the AVL balance factor between nodes as well as node data. We don't check our stack height here, either.

§185 ⟨ AVL copy function 185 ⟩ ≡

```
⟨ BST copy error helper function; bst ⇒ avl 82 ⟩
struct avl_table *avl_copy (const struct avl_table *org, avl_copy_func *copy,
                            avl_item_func *destroy, struct libavl_allocator *allocator) {
    struct avl_node *stack[2 * (AVL_MAX_HEIGHT + 1)];
    int height = 0;
    struct avl_table *new;
    const struct avl_node *x;
    struct avl_node *y;
    assert (org != NULL);
    new = avl_create (org→avl_compare, org→avl_param,
                      allocator != NULL ? allocator : org→avl_alloc);
    if (new == NULL)
        return NULL;
    new→avl_count = org→avl_count;
    if (new→avl_count == 0)
        return new;
    x = (const struct avl_node *) &org→avl_root;
    y = (struct avl_node *) &new→avl_root;
    for (;;) {
        while (x→avl_link[0] != NULL) {
            assert (height < 2 * (AVL_MAX_HEIGHT + 1));
```

```
        y->avl_link[0] = new->avl_alloc->libavl_malloc (new->avl_alloc,
                          sizeof *y->avl_link[0]);
        if (y->avl_link[0] == NULL) {
            if (y != (struct avl_node *) &new->avl_root) {
                y->avl_data = NULL;
                y->avl_link[1] = NULL;
            }
            copy_error_recovery (stack, height, new, destroy);
            return NULL;
        }
        stack[height++] = (struct avl_node *) x;
        stack[height++] = y;
        x = x->avl_link[0];
        y = y->avl_link[0];
    }
    y->avl_link[0] = NULL;
    for (;;) {
        y->avl_balance = x->avl_balance;
        if (copy == NULL)
            y->avl_data = x->avl_data;
        else {
            y->avl_data = copy (x->avl_data, org->avl_param);
            if (y->avl_data == NULL) {
                y->avl_link[1] = NULL;
                copy_error_recovery (stack, height, new, destroy);
                return NULL;
            }
        }
        if (x->avl_link[1] != NULL) {
            y->avl_link[1] = new->avl_alloc->libavl_malloc (new->avl_alloc,
                              sizeof *y->avl_link[1]);
            if (y->avl_link[1] == NULL) {
                copy_error_recovery (stack, height, new, destroy);
                return NULL;
            }
            x = x->avl_link[1];
            y = y->avl_link[1];
            break;
        }
        else y->avl_link[1] = NULL;
        if (height <= 2)
            return new;
        y = stack[--height];
        x = stack[--height];
    }
```

```
    }
}
```

This code is included in §145 and §196.

5.8 Testing

Our job isn't done until we can demonstrate that our code works. We'll do this with a test program built using the framework from the previous chapter (see Section 4.14 [Testing BST Functions], page 80). All we have to do is produce functions for AVL trees that correspond to each of those in ⟨ bst-test.c 98 ⟩. This just involves making small changes to the functions used there. They are presented below without additional comment.

§186 ⟨ avl-test.c 186 ⟩ ≡

```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ limits.h ⟩
#include ⟨ stdio.h ⟩
#include "avl.h"
#include "test.h"
⟨ BST print function; bst ⇒ avl 119 ⟩
⟨ BST traverser check function; bst ⇒ avl 104 ⟩
⟨ Compare two AVL trees for structure and content 187 ⟩
⟨ Recursively verify AVL tree structure 188 ⟩
⟨ AVL tree verify function 190 ⟩
⟨ BST test function; bst ⇒ avl 100 ⟩
⟨ BST overflow test function; bst ⇒ avl 122 ⟩
```

§187 ⟨ Compare two AVL trees for structure and content 187 ⟩ ≡

```
static int compare_trees (struct avl_node *a, struct avl_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
        assert (a == NULL && b == NULL);
        return 1;
    }
    if (*(int *) a->avl_data != *(int *) b->avl_data
        || ((a->avl_link[0] != NULL) != (b->avl_link[0] != NULL))
        || ((a->avl_link[1] != NULL) != (b->avl_link[1] != NULL))
        || a->avl_balance != b->avl_balance) {
        printf (" Copied nodes differ: a=%d (bal=%d) b=%d (bal=%d) a:",
                *(int *) a->avl_data, a->avl_balance,
                *(int *) b->avl_data, b->avl_balance);
        if (a->avl_link[0] != NULL) printf ("l");
        if (a->avl_link[1] != NULL) printf ("r");
        printf (" b:");
        if (b->avl_link[0] != NULL) printf ("l");
        if (b->avl_link[1] != NULL) printf ("r");
        printf ("\n");
```

```
        return 0;
    }
    okay = 1;
    if (a->avl_link[0] != NULL) okay &= compare_trees (a->avl_link[0], b->avl_link[0]);
    if (a->avl_link[1] != NULL) okay &= compare_trees (a->avl_link[1], b->avl_link[1]);
    return okay;
}
```

This code is included in §186.

§188 ⟨ Recursively verify AVL tree structure 188 ⟩ ≡

```
/* Examines the binary tree rooted at node.
   Zeroes *okay if an error occurs. Otherwise, does not modify *okay.
   Sets *count to the number of nodes in that tree, including node itself if node != NULL.
   Sets *height to the tree's height.
   All the nodes in the tree are verified to be at least min but no greater than max. */
static void recurse_verify_tree (struct avl_node *node, int *okay, size_t *count,
                                 int min, int max, int *height) {
    int d; /* Value of this node's data. */
    size_t subcount[2]; /* Number of nodes in subtrees. */
    int subheight[2]; /* Heights of subtrees. */
    if (node == NULL) {
        *count = 0;
        *height = 0;
        return;
    }
    d = *(int *) node->avl_data;
    ⟨ Verify binary search tree ordering 114 ⟩
    recurse_verify_tree (node->avl_link[0], okay, &subcount[0],
                         min, d - 1, &subheight[0]);
    recurse_verify_tree (node->avl_link[1], okay, &subcount[1],
                         d + 1, max, &subheight[1]);
    *count = 1 + subcount[0] + subcount[1];
    *height = 1 + (subheight[0] > subheight[1] ? subheight[0] : subheight[1]);
    ⟨ Verify AVL node balance factor 189 ⟩
}
```

This code is included in §186.

§189 ⟨ Verify AVL node balance factor 189 ⟩ ≡

```
if (subheight[1] - subheight[0] != node->avl_balance) {
    printf (" Balance factor of node %d is %d, but should be %d.\n",
            d, node->avl_balance, subheight[1] - subheight[0]);
    *okay = 0;
}
else if (node->avl_balance < -1 || node->avl_balance > +1) {
    printf (" Balance factor of node %d is %d.\n", d, node->avl_balance);
    *okay = 0;
}
```

This code is included in §188, §332, §451, and §550.

§190 ⟨ AVL tree verify function 190 ⟩ ≡

```
static int verify_tree (struct avl_table *tree, int array[], size_t n) {
    int okay = 1;
    ⟨ Check tree→bst_count is correct; bst ⇒ avl 110 ⟩
    if (okay) { ⟨ Check AVL tree structure 191 ⟩ }
    if (okay) { ⟨ Check that the tree contains all the elements it should; bst ⇒ avl 115 ⟩ }
    if (okay) { ⟨ Check that forward traversal works; bst ⇒ avl 116 ⟩ }
    if (okay) { ⟨ Check that backward traversal works; bst ⇒ avl 117 ⟩ }
    if (okay) { ⟨ Check that traversal from the null element works; bst ⇒ avl 118 ⟩ }
    return okay;
}
```

This code is included in §186, §330, §449, and §548.

§191 ⟨ Check AVL tree structure 191 ⟩ ≡

```
/* Recursively verify tree structure. */
size_t count;
int height;
recurse_verify_tree (tree→avl_root, &okay, &count, 0, INT_MAX, &height);
⟨ Check counted nodes 112 ⟩
```

This code is included in §190.

6 Red-Black Trees

The last chapter saw us implementing a library for one particular type of balanced trees. Red-black trees were invented by R. Bayer and studied at length by L. J. Guibas and R. Sedgewick. This chapter will implement a library for another kind of balanced tree, called a **red-black tree**. For brevity, we'll often abbreviate "red-black" to RB.

Insertion and deletion operations on red-black trees are more complex to describe or to code than the same operations on AVL trees. Red-black trees also have a higher maximum height than AVL trees for a given number of nodes. The primary advantage of red-black trees is that, in AVL trees, deleting one node from a tree containing n nodes may require $\log_2 n$ rotations, but deletion in a red-black tree never requires more than three rotations.

The functions for RB trees in this chapter are analogous to those that we developed for use with AVL trees in the previous chapter. Here's an outline of the red-black code:

§192 ⟨ `rb.h` 192 ⟩ ≡
⟨ License 1 ⟩
#ifndef `RB_H`
#define `RB_H` 1
#include ⟨ stddef.h ⟩
⟨ Table types; tbl ⇒ rb 14 ⟩
⟨ RB maximum height 195 ⟩
⟨ BST table structure; bst ⇒ rb 27 ⟩
⟨ RB node structure 194 ⟩
⟨ BST traverser structure; bst ⇒ rb 61 ⟩
⟨ Table function prototypes; tbl ⇒ rb 15 ⟩
#endif /∗ rb.h ∗/

§193 ⟨ `rb.c` 193 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include ⟨ string.h ⟩
#include "rb.h"
⟨ RB functions 196 ⟩

See also: [Cormen 1990], chapter 14, "Chapter notes."

6.1 Balancing Rule

To most clearly express the red-black balancing rule, we need a few new vocabulary terms. First, define a **non-branching node** as a node that does not "branch" the binary tree in different directions, i.e., a node with exactly zero or one children.

Second, a **path** is a list of one or more nodes in a binary tree where every node in the list (except the last node, of course) is **adjacent** in the tree to the one after it. Two nodes in a tree are considered to be adjacent for this purpose if one is the child of the other. Furthermore, a **simple path** is a path that does not contain any given node more than once.

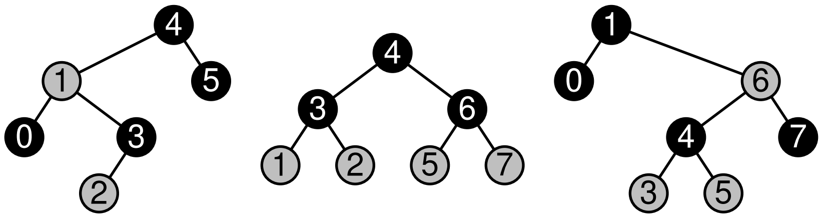
Finally, a node p is a **descendant** of a second node q if both p and q are the same node, or if p is located in one of the subtrees of q.

With these definitions in mind, a red-black tree is a binary search tree in which every node has been labeled with a **color**, either "red" or "black", with those colors distributed according to these two simple rules, which are called the "red-black balancing rules" and often referenced by number:

1. No red node has a red child.
2. Every simple path from a given node to one of its non-branching node descendants contains the same number of black nodes.

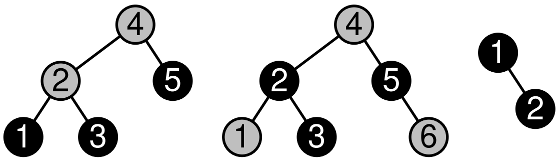
Any binary search tree that conforms to these rules is a red-black tree. Additionally, all red-black trees in Libavl share a simple additional property: their roots are black. This property is not essential, but it does slightly simplify insertion and deletion operations.

To aid in digestion of all these definitions, here are some red-black trees that might be produced by Libavl:



In this book, black nodes are colored black and red nodes are colored gray, as shown here.

The three colored BSTs below are **not** red-black trees. The one on the left violates rule 1, because red node 2 is a child of red node 4. The one in the middle violates rule 2, because one path from the root has two black nodes (4-2-3) and the other paths from the root down to a non-branching node (4-2-1, 4-5, 4-5-6) have only one black node. The one on the right violates rule 2, because the path consisting of only node 1 has only one black node but path 1-2 has two black nodes.



See also: [Cormen 1990], section 14.1; [Sedgewick 1998], definitions 13.3 and 13.4.

Exercises:

***1.** A red-black tree contains only black nodes. Describe the tree's shape.

2. Suppose that a red-black tree's root is red. How can it be transformed into a equivalent red-black tree with a black root? Does a similar procedure work for changing a RB's root from black to red?

3. Suppose we have a perfectly balanced red-black tree with exactly $2^n - 1$ nodes and a black root. Is it possible there is another way to arrange colors in a tree of the same shape that obeys the red-black rules while keeping the root black? Is it possible if we drop the requirement that the tree be balanced?

6.1.1 Analysis

As we were for AVL trees, we're interested in what the red-black balancing rule guarantees about performance. Again, we'll simply state the results:

> A red-black tree with n nodes has height at least $\log_2(n+1)$ but no more than $2\log_2(n+1)$. A red-black tree with height h has at least $2^{h/2}-1$ nodes but no more than 2^h-1.
>
> For comparison, an optimally balanced BST with n nodes has height $\lceil \log_2(n+1) \rceil$. An optimally balanced BST with height h has between 2^{h-1} and 2^h-1 nodes.

See also: [Cormen 1990], lemma 14.1; [Sedgewick 1998], property 13.8.

6.2 Data Types

Red-black trees need their own data structure. Otherwise, there's no appropriate place to store each node's color. Here's a C type for a color and a structure for an RB node, using the *rb_* prefix that we've adopted for this module:

§194 ⟨RB node structure 194⟩ ≡
```
/* Color of a red-black node. */
enum rb_color {
    RB_BLACK, /* Black. */
    RB_RED /* Red. */
};

/* A red-black tree node. */
struct rb_node {
    struct rb_node *rb_link[2]; /* Subtrees. */
    void *rb_data; /* Pointer to data. */
    unsigned char rb_color; /* Color. */
};
```
This code is included in §192.

The maximum height for an RB tree is higher than for an AVL tree, because in the worst case RB trees store nodes less efficiently:

§195 ⟨RB maximum height 195⟩ ≡
```
/* Maximum RB height. */
#ifndef RB_MAX_HEIGHT
#define RB_MAX_HEIGHT 48
#endif
```
This code is included in §192, §333, §452, and §551.

The other data structures for RB trees are the same as for BSTs or AVL trees.

Exercises:

1. Why is it okay to have both an enumeration type and a structure member named *rb_color*?

6.3 Operations

Now we'll implement for RB trees all the operations that we did for BSTs. Everything but the insertion and deletion function can be borrowed either from our BST or AVL tree functions. The copy function is an unusual case: we need it to copy colors, instead of balance factors, between nodes, so we replace *avl_balance* by *rb_color* in the macro expansion.

§196 ⟨ RB functions 196 ⟩ ≡
⟨ BST creation function; bst ⇒ rb 30 ⟩
⟨ BST search function; bst ⇒ rb 31 ⟩
⟨ RB item insertion function 197 ⟩
⟨ Table insertion convenience functions; tbl ⇒ rb 592 ⟩
⟨ RB item deletion function 220 ⟩
⟨ AVL traversal functions; avl ⇒ rb 178 ⟩
⟨ AVL copy function; avl ⇒ rb; avl_balance ⇒ rb_color 185 ⟩
⟨ BST destruction function; bst ⇒ rb 84 ⟩
⟨ Default memory allocation functions; tbl ⇒ rb 6 ⟩
⟨ Table assertion functions; tbl ⇒ rb 594 ⟩

This code is included in §193.

6.4 Insertion

The steps for insertion into a red-black tree are similar to those for insertion into an AVL tree:

1. **Search** for the location to insert the new item.
2. **Insert** the item.
3. **Rebalance** the tree as necessary to satisfy the red-black balance condition.

Red-black node colors don't need to be updated in the way that AVL balance factors do, so there is no separate step for updating colors.

Here's the outline of the function, expressed as code:

§197 ⟨ RB item insertion function 197 ⟩ ≡

```
void **rb_probe (struct rb_table *tree, void *item) {
  ⟨ rb_probe() local variables 198 ⟩
  ⟨ Step 1: Search RB tree for insertion point 199 ⟩
  ⟨ Step 2: Insert RB node 200 ⟩
  ⟨ Step 3: Rebalance after RB insertion 201 ⟩
  return &n->rb_data;
}
```

This code is included in §196.

§198 ⟨ rb_probe() local variables 198 ⟩ ≡

```
struct rb_node *pa[RB_MAX_HEIGHT]; /* Nodes on stack. */
unsigned char da[RB_MAX_HEIGHT]; /* Directions moved from stack nodes. */
int k; /* Stack height. */

struct rb_node *p; /* Traverses tree looking for insertion point. */
struct rb_node *n; /* Newly inserted node. */
```

```
assert (tree != NULL && item != NULL);
```

This code is included in §33, §197, and §210.

See also: [Cormen 1990], section 14.3; [Sedgewick 1998], program 13.6.

6.4.1 Step 1: Search

The first thing to do is to search for the point to insert the new node. In a manner similar to AVL deletion, we keep a stack of nodes tracking the path followed to arrive at the insertion point, so that later we can move up the tree in rebalancing.

§199 ⟨ Step 1: Search RB tree for insertion point 199 ⟩ ≡

```
pa[0] = (struct rb_node *) &tree->rb_root;
da[0] = 0;
k = 1;
for (p = tree->rb_root; p != NULL; p = p->rb_link[da[k - 1]]) {
    int cmp = tree->rb_compare (item, p->rb_data, tree->rb_param);
    if (cmp == 0)
        return &p->rb_data;

    pa[k] = p;
    da[k++] = cmp > 0;
}
```

This code is included in §197 and §210.

6.4.2 Step 2: Insert

§200 ⟨ Step 2: Insert RB node 200 ⟩ ≡

```
n = pa[k - 1]->rb_link[da[k - 1]] =
    tree->rb_alloc->libavl_malloc (tree->rb_alloc, sizeof *n);
if (n == NULL)
    return NULL;

n->rb_data = item;
n->rb_link[0] = n->rb_link[1] = NULL;
n->rb_color = RB_RED;
tree->rb_count++;
tree->rb_generation++;
```

This code is included in §197 and §210.

Exercises:

1. Why are new nodes colored red, instead of black?

6.4.3 Step 3: Rebalance

The code in step 2 that inserts a node always colors the new node red. This means that rule 2 is always satisfied afterward (as long as it was satisfied before we began). On the other hand, rule 1 is broken if the newly inserted node's parent was red. In this latter case we must rearrange or recolor the BST so that it is again an RB tree.

This is what rebalancing does. At each step in rebalancing, we have the invariant that we just colored a node p red and that p's parent, the node at the top of the stack, is also red, a rule 1 violation. The rebalancing step may either clear up the violation entirely, without introducing any other violations, in which case we are done, or, if that is not possible, it reduces the violation to a similar violation of rule 1 higher up in the tree, in which case we go around again.

In no case can we allow the rebalancing step to introduce a rule 2 violation, because the loop is not prepared to repair that kind of problem: it does not fit the invariant. If we allowed rule 2 violations to be introduced, we would have to write additional code to recognize and repair those violations. This extra code would be a waste of space, because we can do just fine without it. (Incidentally, there is nothing magical about using a rule 1 violation as our rebalancing invariant. We could use a rule 2 violation as our invariant instead, and in fact we will later write an alternate implementation that does that, in order to show how it would be done.)

Here is the rebalancing loop. At each rebalancing step, it checks that we have a rule 1 violation by checking the color of $pa[k-1]$, the node on the top of the stack, and then divides into two cases, one for rebalancing an insertion in $pa[k-1]$'s left subtree and a symmetric case for the right subtree. After rebalancing it recolors the root of the tree black just in case the loop changed it to red:

§201 ⟨ Step 3: Rebalance after RB insertion 201 ⟩ ≡
```
while (k >= 3 && pa[k − 1]→rb_color == RB_RED) {
    if (da[k − 2] == 0)
        { ⟨ Left-side rebalancing after RB insertion 202 ⟩ }
    else { ⟨ Right-side rebalancing after RB insertion 206 ⟩ }
}
tree→rb_root→rb_color = RB_BLACK;
```
This code is included in §197.

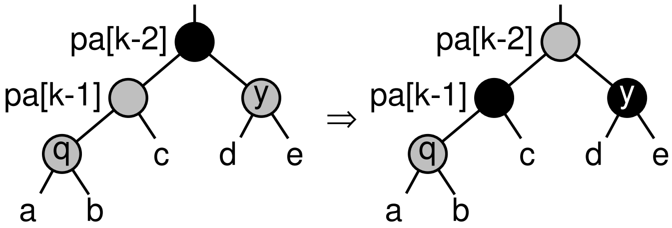
Now for the real work. We'll look at the left-side insertion case only. Consider the node that was just recolored red in the last rebalancing step, or if this is the first rebalancing step, the newly inserted node n. The code does not name this node, but we will refer to it here as q. We know that q is red and, because the loop condition was met, that its parent $pa[k-1]$ is red. Therefore, due to rule 1, q's grandparent, $pa[k-2]$, must be black. After this, we have three cases, distinguished by the following code:

§202 ⟨ Left-side rebalancing after RB insertion 202 ⟩ ≡
```
struct rb_node *y = pa[k − 2]→rb_link[1];
if (y != NULL && y→rb_color == RB_RED)
    { ⟨ Case 1 in left-side RB insertion rebalancing 203 ⟩ }
else {
    struct rb_node *x;
    if (da[k − 1] == 0)
        y = pa[k − 1];
    else { ⟨ Case 3 in left-side RB insertion rebalancing 205 ⟩ }
    ⟨ Case 2 in left-side RB insertion rebalancing 204 ⟩
    break;
}
```

This code is included in §201.

Case 1: q's uncle is red

If q has an "uncle" y, that is, its grandparent has a child on the side opposite q, and y is red, then rearranging the tree's color scheme is all that needs to be done, like this:



Notice the neat way that this preserves the **black-height**, or the number of black nodes in any simple path from a given node down to a node with 0 or 1 children, at $pa[k-2]$. This ensures that rule 2 is not violated.

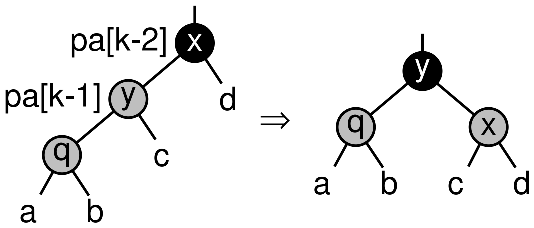
After the transformation, if node $pa[k-2]$'s parent exists and is red, then we have to move up the tree and try again. The **while** loop condition takes care of this test, so adjusting the stack is all that has to be done in this code segment:

§203 ⟨ Case 1 in left-side RB insertion rebalancing 203 ⟩ ≡
$pa[k-1]\to rb\_color = y\to rb\_color =$ `RB_BLACK`;
$pa[k-2]\to rb\_color =$ `RB_RED`;
k −= 2;

This code is included in §202, §207, §342, and §462.

Case 2: q is the left child of $pa[k-1]$

If q is the left child of its parent, then we can perform a right rotation at q's grandparent, which we'll call x, and recolor a couple of nodes. Then we're all done, because we've satisfied both rules. Here's a diagram of what's happened:



There's no need to progress farther up the tree, because neither the subtree's black-height nor its root's color have changed. Here's the corresponding code. Bear in mind that the **break** statement is in the enclosing code segment:

§204 ⟨ Case 2 in left-side RB insertion rebalancing 204 ⟩ ≡
$x = pa[k-2]$;
$x\to rb\_color =$ `RB_RED`;
$y\to rb\_color =$ `RB_BLACK`;
$x\to rb\_link[0] = y\to rb\_link[1]$;

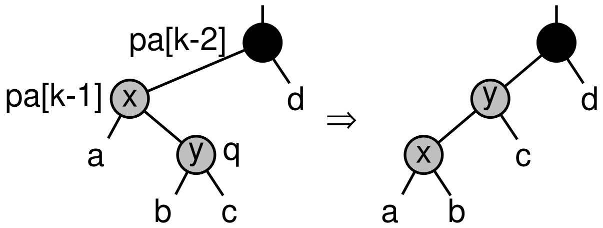
$y \to rb\_link[1] = x$;
$pa[k - 3] \to rb\_link[da[k - 3]] = y$;

This code is included in §202, §343, and §464.

Case 3: *q* is the right child of *pa*[*k* − 1]

The final case, where q is a right child, is really just a small variant of case 2, so we can handle it by transforming it into case 2 and sharing code for that case. To transform case 2 to case 3, we just rotate left at q's parent, which is then treated as q.

The diagram below shows the transformation from case 3 into case 2. After this transformation, x is relabeled q and y's parent is labeled x, then rebalancing continues as shown in the diagram for case 2, with the exception that $pa[k-1]$ is not updated to correspond to y as shown in that diagram. That's okay because variable y has already been set to point to the proper node.



§205 ⟨Case 3 in left-side RB insertion rebalancing 205⟩ ≡
$x = pa[k - 1]$;
$y = x \to rb\_link[1]$;
$x \to rb\_link[1] = y \to rb\_link[0]$;
$y \to rb\_link[0] = x$;
$pa[k - 2] \to rb\_link[0] = y$;

This code is included in §202, §344, and §466.

Exercises:

1. Why is the test $k >= 3$ on the **while** loop valid? (Hint: read the code for step 4, below, first.)

2. Consider rebalancing case 2 and, in particular, what would happen if the root of subtree d were red. Wouldn't the rebalancing transformation recolor x as red and thus cause a rule 1 violation?

6.4.4 Symmetric Case

§206 ⟨Right-side rebalancing after RB insertion 206⟩ ≡
struct rb_node $*y = pa[k - 2] \to rb\_link[0]$;
if (y != `NULL` && $y \to rb\_color$ == `RB_RED`)
    { ⟨Case 1 in right-side RB insertion rebalancing 207⟩ }
else {
    struct rb_node $*x$;
    if ($da[k - 1]$ == 1)
        $y = pa[k - 1]$;

else { ⟨ Case 3 in right-side RB insertion rebalancing 209 ⟩ }
⟨ Case 2 in right-side RB insertion rebalancing 208 ⟩
break;
}

This code is included in §201.

§207 ⟨ Case 1 in right-side RB insertion rebalancing 207 ⟩ ≡
⟨ Case 1 in left-side RB insertion rebalancing 203 ⟩

This code is included in §206, §346, and §463.

§208 ⟨ Case 2 in right-side RB insertion rebalancing 208 ⟩ ≡
$x = pa[k - 2]$;
$x \to rb\_color$ = `RB_RED`;
$y \to rb\_color$ = `RB_BLACK`;
$x \to rb\_link[1] = y \to rb\_link[0]$;
$y \to rb\_link[0] = x$;
$pa[k - 3] \to rb\_link[da[k - 3]] = y$;

This code is included in §206, §347, and §465.

§209 ⟨ Case 3 in right-side RB insertion rebalancing 209 ⟩ ≡
$x = pa[k - 1]$;
$y = x \to rb\_link[0]$;
$x \to rb\_link[0] = y \to rb\_link[1]$;
$y \to rb\_link[1] = x$;
$pa[k - 2] \to rb\_link[1] = y$;

This code is included in §206, §348, and §467.

6.4.5 Aside: Initial Black Insertion

The traditional algorithm for insertion in an RB tree colors new nodes red. This is a good choice, because it often means that no rebalancing is necessary, but it is not the only possible choice. This section implements an alternate algorithm for insertion into an RB tree that colors new nodes black.

The outline is the same as for initial-red insertion. We change the newly inserted node from red to black and replace the rebalancing algorithm:

§210 ⟨ RB item insertion function, initial black 210 ⟩ ≡
void $**rb\_probe$ (**struct rb_table** $*tree$, **void** $*item$) {
⟨ $rb\_probe()$ local variables 198 ⟩
⟨ Step 1: Search RB tree for insertion point 199 ⟩
⟨ Step 2: Insert RB node; RB_RED ⇒ RB_BLACK 200 ⟩
⟨ Step 3: Rebalance after initial-black RB insertion 211 ⟩
return $\&n \to rb\_data$;
}

The remaining task is to devise the rebalancing algorithm. Rebalancing is always necessary, unless the tree was empty before insertion, because insertion of a black node into a nonempty tree always violates rule 2. Thus, our invariant is that we have a rule 2 violation to fix.

More specifically, the invariant, as implemented, is that at the top of each trip through the loop, stack *pa*[] contains the chain of ancestors of a node that is the black root of a subtree whose black-height is 1 more than it should be. We give that node the name q. There is one easy rebalancing special case: if node q has a black parent, we can just recolor q as red, and we're done. Here's the loop:

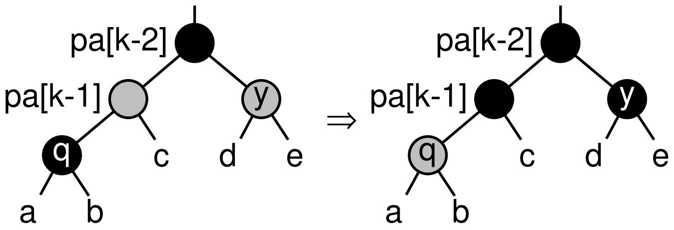
§211 ⟨ Step 3: Rebalance after initial-black RB insertion 211 ⟩ ≡
```
while (k >= 2) {
    struct rb_node *q = pa[k - 1]->rb_link[da[k - 1]];

    if (pa[k - 1]->rb_color == RB_BLACK) {
        q->rb_color = RB_RED;
        break;
    }

    if (da[k - 2] == 0)
        { ⟨Left-side rebalancing after initial-black RB insertion 212⟩ }
    else { ⟨Right-side rebalancing after initial-black RB insertion 216⟩ }
}
```
This code is included in §210.

Consider rebalancing where insertion was on the left side of q's grandparent. We know that q is black and its parent $pa[k-1]$ is red. Then, we can divide rebalancing into three cases, described below in detail. (For additional insight, compare these cases to the corresponding cases for initial-red insertion.)

§212 ⟨ Left-side rebalancing after initial-black RB insertion 212 ⟩ ≡
```
struct rb_node *y = pa[k - 2]->rb_link[1];

if (y != NULL && y->rb_color == RB_RED)
    { ⟨Case 1 in left-side initial-black RB insertion rebalancing 213⟩ }
else {
    struct rb_node *x;

    if (da[k - 1] == 0)
        y = pa[k - 1];
    else { ⟨Case 3 in left-side initial-black RB insertion rebalancing 215⟩ }

    ⟨Case 2 in left-side initial-black RB insertion rebalancing 214⟩
}
```
This code is included in §211.

Case 1: *q*'s uncle is red

If q has an red "uncle" y, then we recolor q red and $pa[k-1]$ and y black. This fixes the immediate problem, making the black-height of q equal to its sibling's, but increases the black-height of $pa[k-2]$, so we must repeat the rebalancing process farther up the tree:

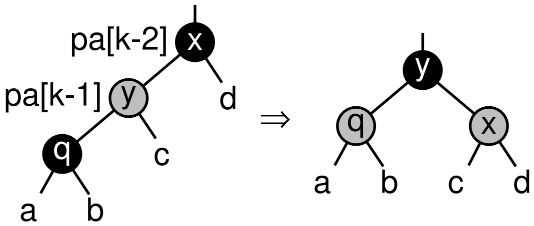


§213 ⟨ Case 1 in left-side initial-black RB insertion rebalancing 213 ⟩ ≡
$pa[k-1] \rightarrow rb\_color = y \rightarrow rb\_color =$ `RB_BLACK`;
$q \rightarrow rb\_color =$ `RB_RED`;
k −= 2;

This code is included in §212 and §217.

Case 2: *q* is the left child of *pa*[*k* − 1]

If q is a left child, then call q's parent y and its grandparent x, rotate right at x, and recolor q, y, and x. The effect is that the black-heights of all three subtrees is the same as before q was inserted, so we're done, and **break** out of the loop.

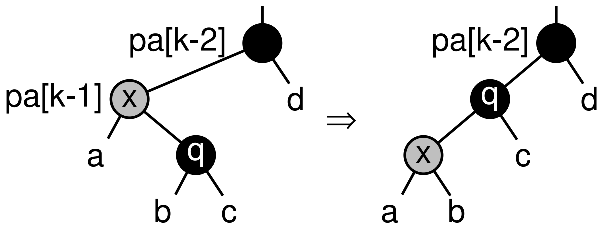


§214 ⟨ Case 2 in left-side initial-black RB insertion rebalancing 214 ⟩ ≡
$x = pa[k-2]$;
$x \rightarrow rb\_color = q \rightarrow rb\_color =$ `RB_RED`;
$y \rightarrow rb\_color =$ `RB_BLACK`;
$x \rightarrow rb\_link[0] = y \rightarrow rb\_link[1]$;
$y \rightarrow rb\_link[1] = x$;
$pa[k-3] \rightarrow rb\_link[da[k-3]] = y$;
break;

This code is included in §212.

Case 3: *q* is the right child of *pa*[*k* − 1]

If q is a right child, then we rotate left at its parent, which we here call x. The result is in the form for application of case 2, so after the rotation, we relabel the nodes to be consistent with that case.



§215 ⟨ Case 3 in left-side initial-black RB insertion rebalancing 215 ⟩ ≡
$x = pa[k - 1]$;
$y = pa[k - 2] \rightarrow rb\_link[0] = q$;
$x \rightarrow rb\_link[1] = y \rightarrow rb\_link[0]$;
$q = y \rightarrow rb\_link[0] = x$;

This code is included in §212.

6.4.5.1 Symmetric Case

§216 ⟨ Right-side rebalancing after initial-black RB insertion 216 ⟩ ≡
struct rb_node $*y = pa[k - 2] \rightarrow rb\_link[0]$;
if (y != `NULL` && $y \rightarrow rb\_color$ == `RB_RED`)
  { ⟨ Case 1 in right-side initial-black RB insertion rebalancing 217 ⟩ }
else {
  struct rb_node $*x$;
  if ($da[k - 1]$ == 1)
    $y = pa[k - 1]$;
  else { ⟨ Case 3 in right-side initial-black RB insertion rebalancing 219 ⟩ }
  ⟨ Case 2 in right-side initial-black RB insertion rebalancing 218 ⟩
}

This code is included in §211.

§217 ⟨ Case 1 in right-side initial-black RB insertion rebalancing 217 ⟩ ≡
⟨ Case 1 in left-side initial-black RB insertion rebalancing 213 ⟩

This code is included in §216.

§218 ⟨ Case 2 in right-side initial-black RB insertion rebalancing 218 ⟩ ≡
$x = pa[k - 2]$;
$x \rightarrow rb\_color = q \rightarrow rb\_color =$ `RB_RED`;
$y \rightarrow rb\_color =$ `RB_BLACK`;
$x \rightarrow rb\_link[1] = y \rightarrow rb\_link[0]$;
$y \rightarrow rb\_link[0] = x$;
$pa[k - 3] \rightarrow rb\_link[da[k - 3]] = y$;
break;

This code is included in §216.

§219 ⟨ Case 3 in right-side initial-black RB insertion rebalancing 219 ⟩ ≡
$x = pa[k - 1]$;
$y = pa[k - 2] \rightarrow rb\_link[1] = q$;
$x \rightarrow rb\_link[0] = y \rightarrow rb\_link[1]$;
$q = y \rightarrow rb\_link[1] = x$;

This code is included in §216.

6.5 Deletion

The process of deletion from an RB tree is very much in line with the other algorithms for balanced trees that we've looked at already. This time, the steps are:

1. **Search** for the item to delete.
2. **Delete** the item.
3. **Rebalance** the tree as necessary.
4. **Finish up** and return.

Here's an outline of the code. Step 1 is already done for us, because we can reuse the search code from AVL deletion.

§220 ⟨ RB item deletion function 220 ⟩ ≡

```
void *rb_delete (struct rb_table *tree, const void *item) {
    struct rb_node *pa[RB_MAX_HEIGHT]; /* Nodes on stack. */
    unsigned char da[RB_MAX_HEIGHT]; /* Directions moved from stack nodes. */
    int k; /* Stack height. */

    struct rb_node *p; /* The node to delete, or a node part way to it. */
    int cmp; /* Result of comparison between item and p. */

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search AVL tree for item to delete; avl ⇒ rb 165 ⟩
    ⟨ Step 2: Delete item from RB tree 221 ⟩
    ⟨ Step 3: Rebalance tree after RB deletion 225 ⟩
    ⟨ Step 4: Finish up after RB deletion 232 ⟩
}
```

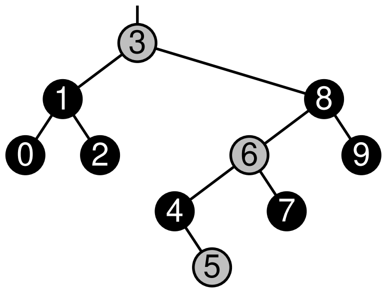
This code is included in §196.

See also: [Cormen 1990], section 14.4.

6.5.1 Step 2: Delete

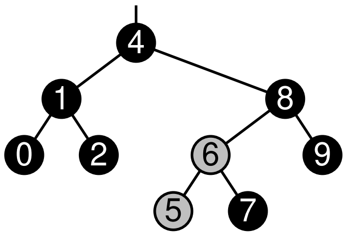
At this point, p is the node to be deleted and the stack contains all of the nodes on the simple path from the tree's root down to p. The immediate task is to delete p. We break deletion down into the familiar three cases (see Section 4.8 [Deleting from a BST], page 39), but before we dive into the code, let's think about the situation.

In red-black insertion, we were able to limit the kinds of violation that could occur to rule 1 or rule 2, at our option, by choosing the new node's color. No such luxury is available in deletion, because colors have already been assigned to all of the nodes. In fact, a naive approach to deletion can lead to multiple violations in widely separated parts of a tree. Consider the effects of deletion of node 3 from the following red-black tree tree, supposing that it is a subtree of some larger tree:

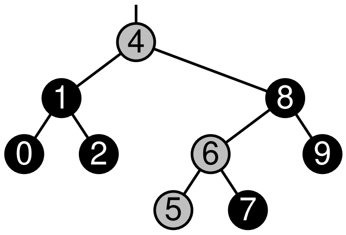


If we performed this deletion in a literal-minded fashion, we would end up with the tree below, with the following violations: rule 1, between node 6 and its child; rule 2, at node 6;

rule 2, at node 4, because the black-height of the subtree as a whole has increased (ignoring the rule 2 violation at node 6); and rule 1, at node 4, only if the subtree's parent is red. The result is difficult to rebalance in general because we have two problem areas to deal with, one at node 4, one at node 6.



Fortunately, we can make things easier for ourselves. We can eliminate the problem area at node 4 simply by recoloring it red, the same color as the node it replaced, as shown below. Then all we have to deal with are the violations at node 6:



This idea holds in general. So, when we replace the deleted node p by a different node q, we set q's color to p's. Besides that, as an implementation detail, we need to keep track of the color of the node that was moved, i.e., node q's former color. We do this here by saving it temporarily in p. In other words, when we replace one node by another during deletion, we swap their colors.

Now we know enough to begin the implementation. While reading this code, keep in mind that after deletion, regardless of the case selected, the stack contains a list of the nodes where rebalancing may be required, and $da[k-1]$ indicates the side of $pa[k-1]$ from which a node of color $p \rightarrow rb\_color$ was deleted. Here's an outline of the meat of the code:

§221 ⟨ Step 2: Delete item from RB tree 221 ⟩ ≡

```
if (p→rb_link[1] == NULL)
    { ⟨ Case 1 in RB deletion 222 ⟩ }
else {
    enum rb_color t;
    struct rb_node *r = p→rb_link[1];
    if (r→rb_link[0] == NULL)
        { ⟨ Case 2 in RB deletion 223 ⟩ }
    else { ⟨ Case 3 in RB deletion 224 ⟩ }
}
```

This code is included in §220.

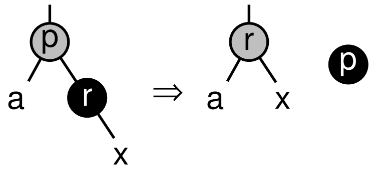
Case 1: *p* has no right child

In case 1, p has no right child, so we replace it by its left subtree. As a very special case, there is no need to do any swapping of colors (see Exercise 1 for details).

§222 ⟨ Case 1 in RB deletion 222 ⟩ ≡
```
pa[k - 1]->rb_link[da[k - 1]] = p->rb_link[0];
```
This code is included in §221.

Case 2: *p*'s right child has no left child

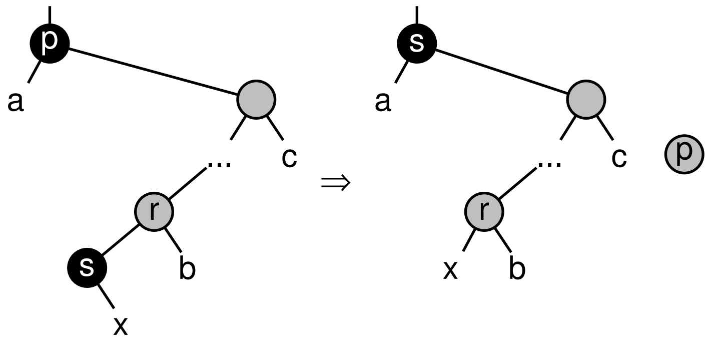
In this case, p has a right child r, which in turn has no left child. We replace p by r, swap the colors of nodes p and r, and add r to the stack because we may need to rebalance there. Here's a pre- and post-deletion diagram that shows one possible set of colors out of the possibilities. Node p is shown detached after deletion to make it clear that the colors are swapped:



§223 ⟨ Case 2 in RB deletion 223 ⟩ ≡
```
r->rb_link[0] = p->rb_link[0];
t = r->rb_color;
r->rb_color = p->rb_color;
p->rb_color = t;
pa[k - 1]->rb_link[da[k - 1]] = r;
da[k] = 1;
pa[k++] = r;
```
This code is included in §221.

Case 3: *p*'s right child has a left child

In this case, p's right child has a left child. The code here is basically the same as for AVL deletion. We replace p by its inorder successor s and swap their node colors. Because they may require rebalancing, we also add all of the nodes we visit to the stack. Here's a diagram to clear up matters, again with arbitrary colors:



§224 ⟨ Case 3 in RB deletion 224 ⟩ ≡

```
struct rb_node *s;
int j = k++;
for (;;) {
    da[k] = 0;
    pa[k++] = r;
    s = r->rb_link[0];
    if (s->rb_link[0] == NULL)
        break;
    r = s;
}
da[j] = 1;
pa[j] = s;
pa[j - 1]->rb_link[da[j - 1]] = s;
s->rb_link[0] = p->rb_link[0];
r->rb_link[0] = s->rb_link[1];
s->rb_link[1] = p->rb_link[1];
t = s->rb_color;
s->rb_color = p->rb_color;
p->rb_color = t;
```

This code is included in §221.

Exercises:

***1.** In case 1, why is it unnecessary to swap the colors of p and the node that replaces it?

2. Rewrite ⟨ Step 2: Delete item from RB tree 221 ⟩ to replace the deleted node's *rb_data* by its successor, then delete the successor, instead of shuffling pointers. (Refer back to Exercise 4.8-3 for an explanation of why this approach cannot be used in Libavl.)

6.5.2 Step 3: Rebalance

At this point, node p has been removed from *tree* and $p \rightarrow rb\_color$ indicates the color of the node that was removed from the tree. Our first step is to handle one common special case: if we deleted a red node, no rebalancing is necessary, because deletion of a red node cannot violate either rule. Here is the code to avoid rebalancing in this special case:

§225 ⟨ Step 3: Rebalance tree after RB deletion 225 ⟩ ≡

```
if (p->rb_color == RB_BLACK)
    { ⟨ Rebalance after RB deletion 226 ⟩ }
```

This code is included in §220.

On the other hand, if a black node was deleted, then we have more work to do. At the least, we have a violation of rule 2. If the deletion brought together two red nodes, as happened in the example in the previous section, there is also a violation of rule 1.

We must now fix both of these problems by rebalancing. This time, the rebalancing loop invariant is that the black-height of $pa[k-1]$'s subtree on side $da[k-1]$ is 1 less than the black-height of its other subtree, a rule 2 violation.

There may also be a rule 2 violation, such $pa[k-1]$ and its child on side $da[k-1]$, which we will call x, are both red. (In the first iteration of the rebalancing loop, node x is the node labeled as such in the diagrams in the previous section.) If this is the case, then the fix for rule 2 is simple: just recolor x black. This increases the black-height and fixes any rule 1 violation as well. If we can do this, we're all done. Otherwise, we have more work to do.

Here's the rebalancing loop:

§226 ⟨ Rebalance after RB deletion 226 ⟩ ≡

```
for (;;) {
    struct rb_node *x = pa[k - 1]->rb_link[da[k - 1]];
    if (x != NULL && x->rb_color == RB_RED) {
        x->rb_color = RB_BLACK;
        break;
    }
    if (k < 2)
        break;
    if (da[k - 1] == 0)
        { ⟨ Left-side rebalancing after RB deletion 227 ⟩ }
    else { ⟨ Right-side rebalancing after RB deletion 233 ⟩ }
    k--;
}
```

This code is included in §225.

Now we'll take a detailed look at the rebalancing algorithm. As before, we'll only examine the case where the deleted node was in its parent's left subtree, that is, where $da[k-1]$ is 0. The other case is similar.

Recall that x is $pa[k-1]\to rb\_link[da[k-1]]$ and that it may be a null pointer. In the left-side deletion case, x is $pa[k-1]$'s left child. We now designate x's "sibling", the right child of $pa[k-1]$, as w. Jumping right in, here's an outline of the rebalancing code:

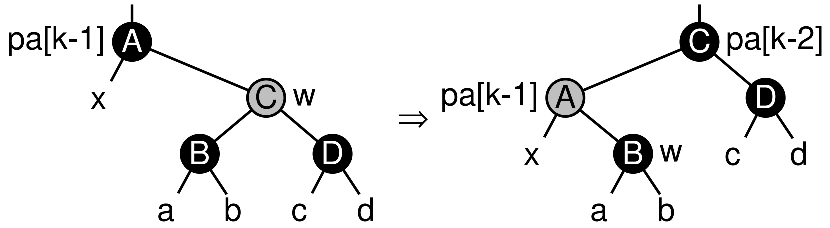
§227 ⟨ Left-side rebalancing after RB deletion 227 ⟩ ≡

```
struct rb_node *w = pa[k - 1]->rb_link[1];
if (w->rb_color == RB_RED)
    { ⟨ Ensure w is black in left-side RB deletion rebalancing 228 ⟩ }
if ((w->rb_link[0] == NULL || w->rb_link[0]->rb_color == RB_BLACK)
  && (w->rb_link[1] == NULL || w->rb_link[1]->rb_color == RB_BLACK))
    { ⟨ Case 1 in left-side RB deletion rebalancing 229 ⟩ }
else {
    if (w->rb_link[1] == NULL || w->rb_link[1]->rb_color == RB_BLACK)
        { ⟨ Transform left-side RB deletion rebalancing case 3 into case 2 231 ⟩ }
    ⟨ Case 2 in left-side RB deletion rebalancing 230 ⟩
    break;
}
```

This code is included in §226.

Case Reduction: Ensure w is black

We know, at this point, that x is a black node or an empty tree. Node w may be red or black. If w is red, we perform a left rotation at the common parent of x and w, labeled A in the diagram below, and recolor A and its own newly acquired parent C. Then we reassign w as the new sibling of x. The effect is to ensure that w is also black, in order to reduce the number of cases:



Node w must have children because x is black, in order to satisfy rule 2, and w's children must be black because of rule 1.

Here is the code corresponding to this transformation. Because the ancestors of node x change, pa[] and da[] are updated as well as w.

§228 ⟨ Ensure w is black in left-side RB deletion rebalancing 228 ⟩ ≡

```
w->rb_color = RB_BLACK;
pa[k - 1]->rb_color = RB_RED;

pa[k - 1]->rb_link[1] = w->rb_link[0];
w->rb_link[0] = pa[k - 1];
pa[k - 2]->rb_link[da[k - 2]] = w;

pa[k] = pa[k - 1];
da[k] = 0;
pa[k - 1] = w;
k++;

w = pa[k - 1]->rb_link[1];
```

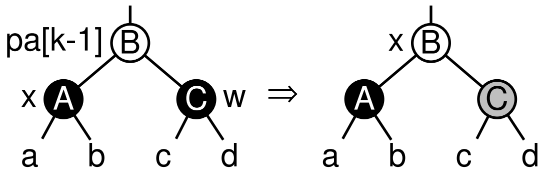
This code is included in §227, §358, and §475.

Now we can take care of the three rebalancing cases one by one. Remember that the situation is a deleted black node in the subtree designated x and the goal is to correct a rule 2 violation. Although subtree x may be an empty tree, the diagrams below show it as a black node. That's okay because the code itself never refers to x. The label is supplied for the reader's benefit only.

Case 1: w has no red children

If w doesn't have any red children, then it can be recolored red. When we do that, the black-height of the subtree rooted at w has decreased, so we must move up the tree, with $pa[k - 1]$ becoming the new x, to rebalance at w and x's parent.

The parent, labeled B in the diagram below, may be red or black. Its color is not changed within the code for this case. If it is red, then the next iteration of the rebalancing loop will recolor it as red immediately and exit. In particular, B will be red if the transformation to make x black was performed earlier. If, on the other hand, B is black, the loop will continue as usual.

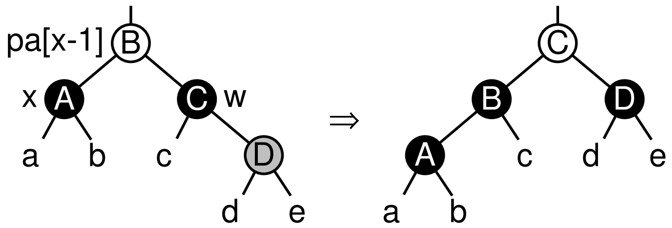


§229 ⟨ Case 1 in left-side RB deletion rebalancing 229 ⟩ ≡
$w \rightarrow rb\_color$ = `RB_RED`;

This code is included in §227, §359, §475, and §574.

Case 2: *w*'s right child is red

If w's right child is red, we can perform a left rotation at $pa[k-1]$ and recolor some nodes, and thereby satisfy both of the red-black rules. The loop is then complete. The transformation looks like this:



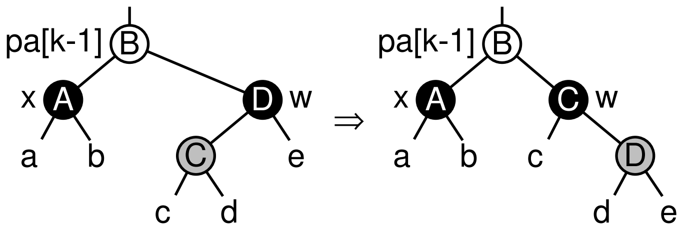
The corresponding code is below. The **break** is supplied by the enclosing code segment ⟨ Left-side rebalancing after RB deletion 227 ⟩:

§230 ⟨ Case 2 in left-side RB deletion rebalancing 230 ⟩ ≡
$w \rightarrow rb\_color = pa[k-1] \rightarrow rb\_color$;
$pa[k-1] \rightarrow rb\_color$ = `RB_BLACK`;
$w \rightarrow rb\_link[1] \rightarrow rb\_color$ = `RB_BLACK`;
$pa[k-1] \rightarrow rb\_link[1] = w \rightarrow rb\_link[0]$;
$w \rightarrow rb\_link[0] = pa[k-1]$;
$pa[k-2] \rightarrow rb\_link[da[k-2]] = w$;

This code is included in §227, §360, and §477.

Case 3: *w*'s left child is red

Because the conditions for neither case 1 nor case 2 apply, the only remaining possibility is that w has a red left child. When this is the case, we can transform it into case 2 by rotating right at w. This causes w to move to the node that was previously w's left child, in this way:



§231 ⟨ Transform left-side RB deletion rebalancing case 3 into case 2 231 ⟩ ≡
struct rb_node $*y = w \rightarrow rb\_link[0]$;

```
y->rb_color = RB_BLACK;
w->rb_color = RB_RED;
w->rb_link[0] = y->rb_link[1];
y->rb_link[1] = w;
w = pa[k - 1]->rb_link[1] = y;
```

This code is included in §227, §361, and §479.

6.5.3 Step 4: Finish Up

All that's left to do is free the node, update counters, and return the deleted item:

§232 ⟨ Step 4: Finish up after RB deletion 232 ⟩ ≡

```
tree->rb_alloc->libavl_free (tree->rb_alloc, p);
tree->rb_count--;
tree->rb_generation++;
return (void *) item;
```

This code is included in §220.

6.5.4 Symmetric Case

§233 ⟨ Right-side rebalancing after RB deletion 233 ⟩ ≡

```
struct rb_node *w = pa[k - 1]->rb_link[0];
if (w->rb_color == RB_RED)
    { ⟨ Ensure w is black in right-side RB deletion rebalancing 234 ⟩ }
if ((w->rb_link[0] == NULL || w->rb_link[0]->rb_color == RB_BLACK)
  && (w->rb_link[1] == NULL || w->rb_link[1]->rb_color == RB_BLACK))
    { ⟨ Case 1 in right-side RB deletion rebalancing 235 ⟩ }
else {
    if (w->rb_link[0] == NULL || w->rb_link[0]->rb_color == RB_BLACK)
        { ⟨ Transform right-side RB deletion rebalancing case 3 into case 2 236 ⟩ }
    ⟨ Case 2 in right-side RB deletion rebalancing 237 ⟩
    break;
}
```

This code is included in §226.

§234 ⟨ Ensure w is black in right-side RB deletion rebalancing 234 ⟩ ≡

```
w->rb_color = RB_BLACK;
pa[k - 1]->rb_color = RB_RED;
pa[k - 1]->rb_link[0] = w->rb_link[1];
w->rb_link[1] = pa[k - 1];
pa[k - 2]->rb_link[da[k - 2]] = w;
pa[k] = pa[k - 1];
da[k] = 1;
pa[k - 1] = w;
k++;
w = pa[k - 1]->rb_link[0];
```

This code is included in §233, §364, and §476.

§235 ⟨ Case 1 in right-side RB deletion rebalancing 235 ⟩ ≡
```
w->rb_color = RB_RED;
```
This code is included in §233, §365, and §476.

§236 ⟨ Transform right-side RB deletion rebalancing case 3 into case 2 236 ⟩ ≡
```
struct rb_node *y = w->rb_link[1];
y->rb_color = RB_BLACK;
w->rb_color = RB_RED;
w->rb_link[1] = y->rb_link[0];
y->rb_link[0] = w;
w = pa[k - 1]->rb_link[0] = y;
```
This code is included in §233, §367, and §480.

§237 ⟨ Case 2 in right-side RB deletion rebalancing 237 ⟩ ≡
```
w->rb_color = pa[k - 1]->rb_color;
pa[k - 1]->rb_color = RB_BLACK;
w->rb_link[0]->rb_color = RB_BLACK;
pa[k - 1]->rb_link[0] = w->rb_link[1];
w->rb_link[1] = pa[k - 1];
pa[k - 2]->rb_link[da[k - 2]] = w;
```
This code is included in §233, §366, and §478.

6.6 Testing

Now we'll present a test program to demonstrate that our code works, using the same framework that has been used in past chapters. The additional code needed is straightforward:

§238 ⟨ rb-test.c 238 ⟩ ≡
```
⟨ License 1 ⟩
#include <assert.h>
#include <limits.h>
#include <stdio.h>
#include "rb.h"
#include "test.h"
⟨ BST print function; bst ⇒ rb 119 ⟩
⟨ BST traverser check function; bst ⇒ rb 104 ⟩
⟨ Compare two RB trees for structure and content 239 ⟩
⟨ Recursively verify RB tree structure 240 ⟩
⟨ RB tree verify function 244 ⟩
⟨ BST test function; bst ⇒ rb 100 ⟩
⟨ BST overflow test function; bst ⇒ rb 122 ⟩
```

§239 ⟨ Compare two RB trees for structure and content 239 ⟩ ≡
```
static int compare_trees (struct rb_node *a, struct rb_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
        assert (a == NULL && b == NULL);
        return 1;
```

```
  }
  if (*(int *) a->rb_data != *(int *) b->rb_data
      || ((a->rb_link[0] != NULL) != (b->rb_link[0] != NULL))
      || ((a->rb_link[1] != NULL) != (b->rb_link[1] != NULL))
      || a->rb_color != b->rb_color) {
    printf (" Copied nodes differ: a=%d%c b=%d%c a:",
            *(int *) a->rb_data, a->rb_color == RB_RED ? 'r' : 'b',
            *(int *) b->rb_data, b->rb_color == RB_RED ? 'r' : 'b');
    if (a->rb_link[0] != NULL) printf ("l");
    if (a->rb_link[1] != NULL) printf ("r");
    printf (" b:");
    if (b->rb_link[0] != NULL) printf ("l");
    if (b->rb_link[1] != NULL) printf ("r");
    printf ("\n");
    return 0;
  }
  okay = 1;
  if (a->rb_link[0] != NULL) okay &= compare_trees (a->rb_link[0], b->rb_link[0]);
  if (a->rb_link[1] != NULL) okay &= compare_trees (a->rb_link[1], b->rb_link[1]);
  return okay;
}
```

This code is included in §238.

§240 ⟨ Recursively verify RB tree structure 240 ⟩ ≡

```
/* Examines the binary tree rooted at node.
   Zeroes *okay if an error occurs. Otherwise, does not modify *okay.
   Sets *count to the number of nodes in that tree, including node itself if node != NULL.
   Sets *bh to the tree's black-height.
   All the nodes in the tree are verified to be at least min but no greater than max. */
static void recurse_verify_tree (struct rb_node *node, int *okay, size_t *count,
                                 int min, int max, int *bh) {
  int d;                /* Value of this node's data. */
  size_t subcount[2];   /* Number of nodes in subtrees. */
  int subbh[2];         /* Black-heights of subtrees. */
  if (node == NULL) {
    *count = 0;
    *bh = 0;
    return;
  }
  d = *(int *) node->rb_data;
  ⟨ Verify binary search tree ordering 114 ⟩
  recurse_verify_tree (node->rb_link[0], okay, &subcount[0],
                       min, d - 1, &subbh[0]);
  recurse_verify_tree (node->rb_link[1], okay, &subcount[1],
                       d + 1, max, &subbh[1]);
  *count = 1 + subcount[0] + subcount[1];
```

```
    *bh = (node->rb_color == RB_BLACK) + subbh[0];
    ⟨ Verify RB node color 241 ⟩
    ⟨ Verify RB node rule 1 compliance 242 ⟩
    ⟨ Verify RB node rule 2 compliance 243 ⟩
}
```

This code is included in §238.

§241 ⟨ Verify RB node color 241 ⟩ ≡

```
if (node->rb_color != RB_RED && node->rb_color != RB_BLACK) {
    printf ("␣Node␣%d␣is␣neither␣red␣nor␣black␣(%d).\n", d, node->rb_color);
    *okay = 0;
}
```

This code is included in §240, §370, §484, and §585.

§242 ⟨ Verify RB node rule 1 compliance 242 ⟩ ≡

```
/* Verify compliance with rule 1. */
if (node->rb_color == RB_RED) {
    if (node->rb_link[0] != NULL && node->rb_link[0]->rb_color == RB_RED) {
        printf ("␣Red␣node␣%d␣has␣red␣left␣child␣%d\n",
                d, *(int *) node->rb_link[0]->rb_data);
        *okay = 0;
    }
    if (node->rb_link[1] != NULL && node->rb_link[1]->rb_color == RB_RED) {
        printf ("␣Red␣node␣%d␣has␣red␣right␣child␣%d\n",
                d, *(int *) node->rb_link[1]->rb_data);
        *okay = 0;
    }
}
```

This code is included in §240 and §585.

§243 ⟨ Verify RB node rule 2 compliance 243 ⟩ ≡

```
/* Verify compliance with rule 2. */
if (subbh[0] != subbh[1]) {
    printf ("␣Node␣%d␣has␣two␣different␣blackheights:␣left␣bh=%d,␣"
            "right␣bh=%d\n", d, subbh[0], subbh[1]);
    *okay = 0;
}
```

This code is included in §240, §370, §484, and §585.

§244 ⟨ RB tree verify function 244 ⟩ ≡

```
static int verify_tree (struct rb_table *tree, int array[], size_t n) {
    int okay = 1;
    ⟨ Check tree->bst_count is correct; bst ⇒ rb 110 ⟩
    if (okay) { ⟨ Check root is black 245 ⟩ }
    if (okay) { ⟨ Check RB tree structure 246 ⟩ }
    if (okay) { ⟨ Check that the tree contains all the elements it should; bst ⇒ rb 115 ⟩ }
    if (okay) { ⟨ Check that forward traversal works; bst ⇒ rb 116 ⟩ }
```

```
    if (okay) { ⟨Check that backward traversal works; bst ⇒ rb 117⟩ }
    if (okay) { ⟨Check that traversal from the null element works; bst ⇒ rb 118⟩ }
    return okay;
}
```

This code is included in §238, §368, §482, and §583.

§245 ⟨Check root is black 245⟩ ≡

```
if (tree→rb_root != NULL && tree→rb_root→rb_color != RB_BLACK) {
    printf (" Tree's root is not black.\n");
    okay = 0;
}
```

This code is included in §244.

§246 ⟨Check RB tree structure 246⟩ ≡

```
/* Recursively verify tree structure. */
size_t count;
int bh;

recurse_verify_tree (tree→rb_root, &okay, &count, 0, INT_MAX, &bh);
⟨Check counted nodes 112⟩
```

This code is included in §244.

7 Threaded Binary Search Trees

Traversal in inorder, as done by LIBAVL traversers, is a common operation in a binary tree. To do this efficiently in an ordinary binary search tree or balanced tree, we need to maintain a list of the nodes above the current node, or at least a list of nodes still to be visited. This need leads to the stack used in **struct bst_traverser** and friends.

It's really too bad that we need such stacks for traversal. First, they take up space. Second, they're fragile: if an item is inserted into or deleted from the tree during traversal, or if the tree is balanced, we have to rebuild the traverser's stack. In addition, it can sometimes be difficult to know in advance how tall the stack will need to be, as demonstrated by the code that we wrote to handle stack overflow.

These problems are important enough that, in this book, we'll look at two different solutions. This chapter looks at the first of these, which adds special pointers, each called a **thread**, to nodes, producing what is called a threaded binary search tree, **threaded tree**, or simply a TBST.[1] Later in the book, we'll examine an alternate and more general solution using a **parent pointer** in each node.

Here's the outline of the TBST code. We're using the prefix *tbst_* this time:

§247 ⟨ `tbst.h` 247 ⟩ ≡
⟨ License 1 ⟩
#ifndef `TBST_H`
#define `TBST_H` 1

#include ⟨ stddef.h ⟩

⟨ Table types; tbl ⇒ tbst 14 ⟩
⟨ TBST table structure 250 ⟩
⟨ TBST node structure 249 ⟩
⟨ TBST traverser structure 267 ⟩
⟨ Table function prototypes; tbl ⇒ tbst 15 ⟩
⟨ BST extra function prototypes; bst ⇒ tbst 88 ⟩

#endif /∗ tbst.h ∗/

§248 ⟨ `tbst.c` 248 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "tbst.h"

⟨ TBST functions 251 ⟩

7.1 Threads

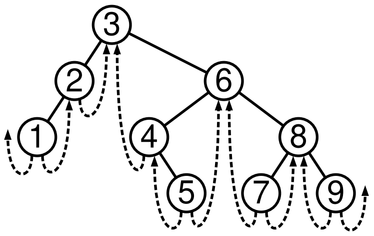
In an ordinary binary search tree or balanced tree, a lot of the pointer fields go more-or-less unused. Instead of pointing to somewhere useful, they are used to store null pointers.

[1] This usage of "thread" has nothing to do with the idea of a program with multiple "threads of excecution", a form of multitasking within a single program.

In a sense, they're wasted. What if we were to instead use these fields to point elsewhere in the tree?

This is the idea behind a threaded tree. In a threaded tree, a node's left child pointer field, if it would otherwise be a null pointer, is used to point to the node's inorder predecessor. An otherwise-null right child pointer field points to the node's successor. The least-valued node in a threaded tree has a null pointer for its left thread, and the greatest-valued node similarly has a null right thread. These two are the only null pointers in a threaded tree.

Here's a sample threaded tree:



This diagram illustrates the convention used for threads in this book, arrowheads attached to dotted lines. Null threads in the least and greatest nodes are shown as arrows pointing into space. This kind of arrow is also used to show threads that point to nodes not shown in the diagram.

There are some disadvantages to threaded trees. Each node in an unthreaded tree has only one pointer that leads to it, either from the tree structure or its parent node, but in a threaded tree some nodes have as many as three pointers leading to them: one from the root or parent, one from its predecessor's right thread, and one from its successor's left thread. This means that, although traversing a threaded tree is simpler, building and maintaining a threaded tree is more complicated.

As we learned earlier, any node that has a right child has a successor in its right subtree, and that successor has no left child. So, a node in an threaded tree has a left thread pointing back to it if and only if the node has a right child. Similarly, a node has a right thread pointing to it if and only if the node has a left child. Take a look at the sample tree above and check these statements for yourself for some of its nodes.

See also: [Knuth 1997], section 2.3.1.

7.2 Data Types

We need two extra fields in the node structure to keep track of whether each link is a child pointer or a thread. Each of these fields is called a **tag**. The revised **struct tbst_node**, along with **enum tbst_tag** for tags, looks like this:

§249 ⟨ TBST node structure 249 ⟩ ≡

```
/* Characterizes a link as a child pointer or a thread. */
enum tbst_tag {
    TBST_CHILD, /* Child pointer. */
    TBST_THREAD /* Thread. */
};
/* A threaded binary search tree node. */
struct tbst_node {
```

```
    struct tbst_node *tbst_link[2]; /* Subtrees. */
    void *tbst_data; /* Pointer to data. */
    unsigned char tbst_tag[2]; /* Tag fields. */
};
```

This code is included in §247.

Each element of *tbst_tag*[] is set to `TBST_CHILD` if the corresponding *tbst_link*[] element is a child pointer, or to `TBST_THREAD` if it is a thread. The other members of **struct tbst_node** should be familiar.

We also want a revised table structure, because traversers in threaded trees do not need a generation number:

§250 ⟨ TBST table structure 250 ⟩ ≡

```
/* Tree data structure. */
struct tbst_table {
    struct tbst_node *tbst_root; /* Tree's root. */
    tbst_comparison_func *tbst_compare; /* Comparison function. */
    void *tbst_param; /* Extra argument to tbst_compare. */
    struct libavl_allocator *tbst_alloc; /* Memory allocator. */
    size_t tbst_count; /* Number of items in tree. */
};
```

This code is included in §247, §297, §333, §372, §415, §452, §486, §519, and §551.

There is no need to define a maximum height for TBST trees because none of the TBST functions use a stack.

Exercises:

1. We defined **enum tbst_tag** for distinguishing threads from child pointers, but declared the actual tag members as **unsigned char** instead. Why?

7.3 Operations

Now that we've changed the basic form of our binary trees, we have to rewrite most of the tree functions. A function designed for use with unthreaded trees will get hopelessly lost in a threaded tree, because it will follow threads that it thinks are child pointers. The only functions we can keep are the totally generic functions defined in terms of other table functions.

§251 ⟨ TBST functions 251 ⟩ ≡

⟨ TBST creation function 252 ⟩
⟨ TBST search function 253 ⟩
⟨ TBST item insertion function 254 ⟩
⟨ Table insertion convenience functions; tbl ⇒ tbst 592 ⟩
⟨ TBST item deletion function 257 ⟩
⟨ TBST traversal functions 268 ⟩
⟨ TBST copy function 278 ⟩
⟨ TBST destruction function 281 ⟩
⟨ TBST balance function 282 ⟩
⟨ Default memory allocation functions; tbl ⇒ tbst 6 ⟩

⟨ Table assertion functions; tbl ⇒ tbst 594 ⟩

This code is included in §248.

7.4 Creation

Function *tbst_create*() is the same as *bst_create*() except that a **struct tbst_table** has no generation number to fill in.

§252 ⟨ TBST creation function 252 ⟩ ≡

```
struct tbst_table *tbst_create (tbst_comparison_func *compare, void *param,
                                struct libavl_allocator *allocator) {
  struct tbst_table *tree;

  assert (compare != NULL);

  if (allocator == NULL)
    allocator = &tbst_allocator_default;

  tree = allocator->libavl_malloc (allocator, sizeof *tree);
  if (tree == NULL)
    return NULL;

  tree->tbst_root = NULL;
  tree->tbst_compare = compare;
  tree->tbst_param = param;
  tree->tbst_alloc = allocator;
  tree->tbst_count = 0;

  return tree;
}
```

This code is included in §251, §300, §336, §375, §418, §455, §489, §522, and §554.

7.5 Search

In searching a TBST we just have to be careful to distinguish threads from child pointers. If we hit a thread link, then we've run off the bottom of the tree and the search is unsuccessful. Other that that, a search in a TBST works the same as in any other binary search tree.

§253 ⟨ TBST search function 253 ⟩ ≡

```
void *tbst_find (const struct tbst_table *tree, const void *item) {
  const struct tbst_node *p;

  assert (tree != NULL && item != NULL);

  p = tree->tbst_root;
  if (p == NULL)
    return NULL;

  for (;;) {
    int cmp, dir;

    cmp = tree->tbst_compare (item, p->tbst_data, tree->tbst_param);
    if (cmp == 0)
      return p->tbst_data;
```

```
        dir = cmp > 0;
        if (p->tbst_tag[dir] == TBST_CHILD)
            p = p->tbst_link[dir];
        else return NULL;
    }
}
```

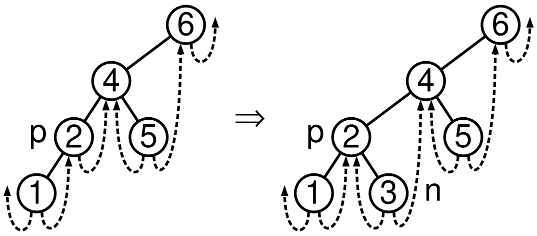
This code is included in §251, §300, and §336.

7.6 Insertion

It take a little more effort to insert a new node into a threaded BST than into an unthreaded one, but not much more. The only difference is that we now have to set up the new node's left and right threads to point to its predecessor and successor, respectively.

Fortunately, these are easy to figure out. Suppose that new node *n* is the right child of its parent *p* (the other case is symmetric). This means that *p* is *n*'s predecessor, because *n* is the least node in *p*'s right subtree. Moreover, *n*'s successor is the node that was *p*'s successor before *n* was inserted, that is to say, it is the same as *p*'s former right thread.

Here's an example that may help to clear up the description. When new node 3 is inserted as the right child of 2, its left thread points to 2 and its right thread points where 2's right thread formerly did, to 4:



The following code unifies the left-side and right-side cases using *dir*, which takes the value 1 for a right-side insertion, 0 for a left-side insertion. The side opposite *dir* can then be expressed simply as !*dir*.

§254 ⟨TBST item insertion function 254⟩ ≡

```
void **tbst_probe (struct tbst_table *tree, void *item) {
    struct tbst_node *p; /* Traverses tree to find insertion point. */
    struct tbst_node *n; /* New node. */
    int dir; /* Side of p on which n is inserted. */

    assert (tree != NULL && item != NULL);

    ⟨Step 1: Search TBST for insertion point 255⟩
    ⟨Step 2: Insert TBST node 256⟩

    return &n->tbst_data;
}
```

This code is included in §251.

§255 ⟨Step 1: Search TBST for insertion point 255⟩ ≡

```
if (tree->tbst_root != NULL)
    for (p = tree->tbst_root; ; p = p->tbst_link[dir]) {
        int cmp = tree->tbst_compare (item, p->tbst_data, tree->tbst_param);
```

```
            if (cmp == 0)
                return &p->tbst_data;
            dir = cmp > 0;
            if (p->tbst_tag[dir] == TBST_THREAD)
                break;
        }
    else {
        p = (struct tbst_node *) &tree->tbst_root;
        dir = 0;
    }
```

This code is included in §254 and §668.

§256 ⟨ Step 2: Insert TBST node 256 ⟩ ≡

```
n = tree->tbst_alloc->libavl_malloc (tree->tbst_alloc, sizeof *n);
if (n == NULL)
    return NULL;
tree->tbst_count++;
n->tbst_data = item;
n->tbst_tag[0] = n->tbst_tag[1] = TBST_THREAD;
n->tbst_link[dir] = p->tbst_link[dir];
if (tree->tbst_root != NULL) {
    p->tbst_tag[dir] = TBST_CHILD;
    n->tbst_link[!dir] = p;
}
else n->tbst_link[1] = NULL;
p->tbst_link[dir] = n;
```

This code is included in §254, §303, and §339.

See also: [Knuth 1997], algorithm 2.3.1I.

Exercises:

1. What happens if we reverse the order of the final **if** statement above and the following assignment?

7.7 Deletion

When we delete a node from a threaded tree, we have to update one or two more pointers than if it were an unthreaded BST. What's more, we sometimes have to go to a bit of effort to track down what pointers these are, because they are in the predecessor and successor of the node being deleted.

The outline is the same as for deleting a BST node:

§257 ⟨ TBST item deletion function 257 ⟩ ≡

```
void *tbst_delete (struct tbst_table *tree, const void *item) {
    struct tbst_node *p; /* Node to delete. */
    struct tbst_node *q; /* Parent of p. */
    int dir; /* Index into q->tbst_link[] that leads to p. */
    assert (tree != NULL && item != NULL);
```

```
    ⟨ Find TBST node to delete 258 ⟩
    ⟨ Delete TBST node 259 ⟩
    ⟨ Finish up after deleting TBST node 266 ⟩
}
```

This code is included in §251.

We search down the tree to find the item to delete, p. As we do it we keep track of its parent q and the direction *dir* that we descended from it. The initial value of q and *dir* use the trick seen originally in copying a BST (see Section 4.10.2 [Copying a BST Iteratively], page 63).

There are nicer ways to do the same thing, though they are not necessarily as efficient. See the exercises for one possibility.

§258 ⟨ Find TBST node to delete 258 ⟩ ≡

```
if (tree→tbst_root == NULL)
    return NULL;
p = tree→tbst_root;
q = (struct tbst_node *) &tree→tbst_root;
dir = 0;
for (;;) {
    int cmp = tree→tbst_compare (item, p→tbst_data, tree→tbst_param);
    if (cmp == 0)
        break;
    dir = cmp > 0;
    if (p→tbst_tag[dir] == TBST_THREAD)
        return NULL;
    q = p;
    p = p→tbst_link[dir];
}
item = p→tbst_data;
```

This code is included in §257.

The cases for deletion from a threaded tree are a bit different from those for an unthreaded tree. The key point to keep in mind is that a node with n children has n threads pointing to it that must be updated when it is deleted. Let's look at the cases in detail now.

Here's the outline:

§259 ⟨ Delete TBST node 259 ⟩ ≡

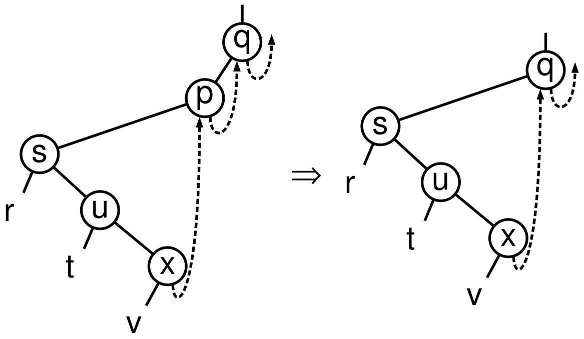
```
if (p→tbst_tag[1] == TBST_THREAD) {
    if (p→tbst_tag[0] == TBST_CHILD)
        { ⟨ Case 1 in TBST deletion 260 ⟩ }
    else { ⟨ Case 2 in TBST deletion 261 ⟩ }
} else {
    struct tbst_node *r = p→tbst_link[1];
    if (r→tbst_tag[0] == TBST_THREAD)
        { ⟨ Case 3 in TBST deletion 262 ⟩ }
    else { ⟨ Case 4 in TBST deletion 263 ⟩ }
```

}

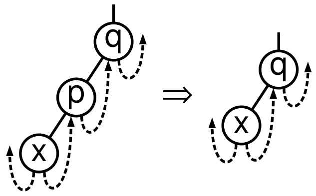
This code is included in §257.

Case 1: *p* has a right thread and a left child

If *p* has a right thread and a left child, then we replace it by its left child. We also replace its predecessor *t*'s right thread by *p*'s right thread. In the most general subcase, the whole operation looks something like this:



On the other hand, it can be as simple as this:



Both of these subcases, and subcases in between them in complication, are handled by the same code:

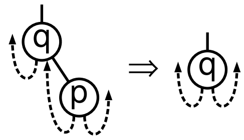
§260 ⟨ Case 1 in TBST deletion 260 ⟩ ≡

```
struct tbst_node *t = p→tbst_link[0];
while (t→tbst_tag[1] == TBST_CHILD)
    t = t→tbst_link[1];
t→tbst_link[1] = p→tbst_link[1];
q→tbst_link[dir] = p→tbst_link[0];
```

This code is included in §259 and §314.

Case 2: *p* has a right thread and a left thread

If *p* is a leaf, then no threads point to it, but we must change its parent *q*'s pointer to *p* to a thread, pointing to the same place that the corresponding thread of *p* pointed. This is easy, and typically looks something like this:



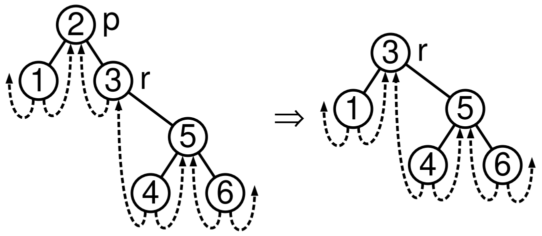
There is one special case, which comes up when *q* is the pseudo-node used for the parent of the root. We can't access *tbst_tag*[] in this "node". Here's the code:

§261 ⟨ Case 2 in TBST deletion 261 ⟩ ≡
```
q→tbst_link[dir] = p→tbst_link[dir];
if (q != (struct tbst_node *) &tree→tbst_root)
    q→tbst_tag[dir] = TBST_THREAD;
```

This code is included in §259 and §315.

Case 3: *p*'s right child has a left thread

If p has a right child r, and r itself has a left thread, then we delete p by moving r into its place. Here's an example where the root node is deleted:



This just involves changing q's right link to point to r, copying p's left link and tag into r, and fixing any thread that pointed to p so that it now points to r. The code is straightforward:

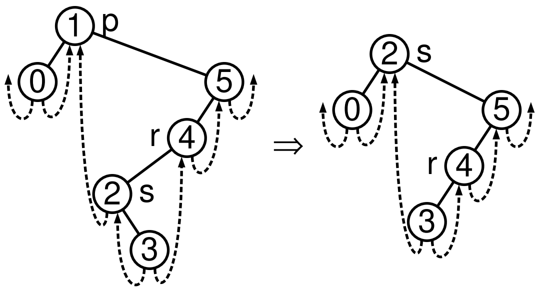
§262 ⟨ Case 3 in TBST deletion 262 ⟩ ≡
```
r→tbst_link[0] = p→tbst_link[0];
r→tbst_tag[0] = p→tbst_tag[0];
if (r→tbst_tag[0] == TBST_CHILD) {
    struct tbst_node *t = r→tbst_link[0];
    while (t→tbst_tag[1] == TBST_CHILD)
        t = t→tbst_link[1];
    t→tbst_link[1] = r;
}
q→tbst_link[dir] = r;
```

This code is included in §259 and §316.

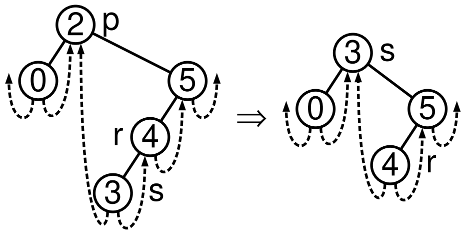
Case 4: *p*'s right child has a left child

If p has a right child, which in turn has a left child, we arrive at the most complicated case. It corresponds to case 3 in deletion from an unthreaded BST. The solution is to find p's successor s and move it in place of p. In this case, r is s's parent node, not necessarily p's right child.

There are two subcases here. In the first, s has a right child. In that subcase, s's own successor's left thread already points to s, so we need not adjust any threads. Here's an example of this subcase. Notice how the left thread of node 3, s's successor, already points to s.



The second subcase comes up when s has a right thread. Because s also has a left thread, this means that s is a leaf. This subcase requires us to change r's left link to a thread to its predecessor, which is now s. Here's a continuation of the previous example, showing deletion of the new root, node 2:



The first part of the code handles finding r and s:

§263 ⟨ Case 4 in TBST deletion 263 ⟩ ≡
```
struct tbst_node *s;
for (;;) {
    s = r->tbst_link[0];
    if (s->tbst_tag[0] == TBST_THREAD)
        break;
    r = s;
}
```
See also §264 and §265.
This code is included in §259 and §317.

Next, we update r, handling each of the subcases:

§264 ⟨ Case 4 in TBST deletion 263 ⟩ +≡
```
if (s->tbst_tag[1] == TBST_CHILD)
    r->tbst_link[0] = s->tbst_link[1];
else {
    r->tbst_link[0] = s;
    r->tbst_tag[0] = TBST_THREAD;
}
```

Finally, we copy p's links and tags into s and chase down and update any right thread in s's left subtree, then replace the pointer from q down to s:

§265 ⟨ Case 4 in TBST deletion 263 ⟩ +≡
```
s->tbst_link[0] = p->tbst_link[0];
if (p->tbst_tag[0] == TBST_CHILD) {
    struct tbst_node *t = p->tbst_link[0];
    while (t->tbst_tag[1] == TBST_CHILD)
        t = t->tbst_link[1];
```

```
    t→tbst_link[1] = s;
    s→tbst_tag[0] = TBST_CHILD;
}
s→tbst_link[1] = p→tbst_link[1];
s→tbst_tag[1] = TBST_CHILD;
q→tbst_link[dir] = s;
```

We finish up by deallocating the node, decrementing the tree's item count, and returning the deleted item's data:

§266 ⟨ Finish up after deleting TBST node 266 ⟩ ≡

```
tree→tbst_alloc→libavl_free (tree→tbst_alloc, p);
tree→tbst_count−−;
return (void *) item;
```

This code is included in §257.

Exercises:

***1.** In a threaded BST, there is an efficient algorithm to find the parent of a given node. Use this algorithm to reimplement ⟨ Find TBST node to delete 258 ⟩.

2. In case 2, we must handle q as the pseudo-root as a special case. Can we rearrange the TBST data structures to avoid this?

3. Rewrite case 4 to replace the deleted node's *tbst_data* by its successor and actually delete the successor, instead of moving around pointers. (Refer back to Exercise 4.8-3 for an explanation of why this approach cannot be used in Libavl.)

***4.** Many of the cases in deletion from a TBST require searching down the tree for the nodes with threads to the deleted node. Show that this adds only a constant number of operations to the deletion of a randomly selected node, compared to a similar deletion in an unthreaded tree.

7.8 Traversal

Traversal in a threaded BST is much simpler than in an unthreaded one. This is, indeed, much of the point to threading our trees. This section implements all of the Libavl traverser functions for threaded trees.

Suppose we wish to find the successor of an arbitrary node in a threaded tree. If the node has a right child, then the successor is the smallest item in the node's right subtree. Otherwise, the node has a right thread, and its sucessor is simply the node to which the right thread points. If the right thread is a null pointer, then the node is the largest in the tree. We can find the node's predecessor in a similar manner.

We don't ever need to know the parent of a node to traverse the threaded tree, so there's no need to keep a stack. Moreover, because a traverser has no stack to be corrupted by changes to its tree, there is no need to keep or compare generation numbers. Therefore, this is all we need for a TBST traverser structure:

§267 ⟨ TBST traverser structure 267 ⟩ ≡

```
/* TBST traverser structure. */
struct tbst_traverser {
```

```
    struct tbst_table *tbst_table; /* Tree being traversed. */
    struct tbst_node *tbst_node; /* Current node in tree. */
};
```

This code is included in §247, §297, §333, §372, §415, §452, §486, §519, and §551.

The traversal functions are collected together here. A few of the functions are implemented directly in terms of their unthreaded BST counterparts, but most must be reimplemented:

§268 ⟨ TBST traversal functions 268 ⟩ ≡

```
⟨ TBST traverser null initializer 269 ⟩
⟨ TBST traverser first initializer 270 ⟩
⟨ TBST traverser last initializer 271 ⟩
⟨ TBST traverser search initializer 272 ⟩
⟨ TBST traverser insertion initializer 273 ⟩
⟨ TBST traverser copy initializer 274 ⟩
⟨ TBST traverser advance function 275 ⟩
⟨ TBST traverser back up function 276 ⟩
⟨ BST traverser current item function; bst ⇒ tbst 74 ⟩
⟨ BST traverser replacement function; bst ⇒ tbst 75 ⟩
```

This code is included in §251, §300, and §336.

See also: [Knuth 1997], algorithm 2.3.1S.

7.8.1 Starting at the Null Node

§269 ⟨ TBST traverser null initializer 269 ⟩ ≡

```
void tbst_t_init (struct tbst_traverser *trav, struct tbst_table *tree) {
    trav->tbst_table = tree;
    trav->tbst_node = NULL;
}
```

This code is included in §268, §395, §502, and §546.

7.8.2 Starting at the First Node

§270 ⟨ TBST traverser first initializer 270 ⟩ ≡

```
void *tbst_t_first (struct tbst_traverser *trav, struct tbst_table *tree) {
    assert (tree != NULL && trav != NULL);

    trav->tbst_table = tree;
    trav->tbst_node = tree->tbst_root;
    if (trav->tbst_node != NULL) {
        while (trav->tbst_node->tbst_tag[0] == TBST_CHILD)
            trav->tbst_node = trav->tbst_node->tbst_link[0];
        return trav->tbst_node->tbst_data;
    }
    else return NULL;
}
```

This code is included in §268.

7.8.3 Starting at the Last Node

§271 ⟨ TBST traverser last initializer 271 ⟩ ≡

```
void *tbst_t_last (struct tbst_traverser *trav, struct tbst_table *tree) {
  assert (tree != NULL && trav != NULL);

  trav->tbst_table = tree;
  trav->tbst_node = tree->tbst_root;
  if (trav->tbst_node != NULL) {
    while (trav->tbst_node->tbst_tag[1] == TBST_CHILD)
      trav->tbst_node = trav->tbst_node->tbst_link[1];
    return trav->tbst_node->tbst_data;
  }
  else return NULL;
}
```

This code is included in §268.

7.8.4 Starting at a Found Node

The code for this function is derived with few changes from ⟨ TBST search function 253 ⟩.

§272 ⟨ TBST traverser search initializer 272 ⟩ ≡

```
void *tbst_t_find (struct tbst_traverser *trav, struct tbst_table *tree, void *item) {
  struct tbst_node *p;

  assert (trav != NULL && tree != NULL && item != NULL);

  trav->tbst_table = tree;
  trav->tbst_node = NULL;

  p = tree->tbst_root;
  if (p == NULL)
    return NULL;

  for (;;) {
    int cmp, dir;

    cmp = tree->tbst_compare (item, p->tbst_data, tree->tbst_param);
    if (cmp == 0) {
      trav->tbst_node = p;
      return p->tbst_data;
    }

    dir = cmp > 0;
    if (p->tbst_tag[dir] == TBST_CHILD)
      p = p->tbst_link[dir];
    else return NULL;
  }
}
```

This code is included in §268.

7.8.5 Starting at an Inserted Node

This implementation is a trivial adaptation of ⟨ AVL traverser insertion initializer 179 ⟩. In particular, management of generation numbers has been removed.

§273 ⟨ TBST traverser insertion initializer 273 ⟩ ≡
```
void *tbst_t_insert (struct tbst_traverser *trav, struct tbst_table *tree, void *item) {
    void **p;
    assert (trav != NULL && tree != NULL && item != NULL);
    p = tbst_probe (tree, item);
    if (p != NULL) {
        trav->tbst_table = tree;
        trav->tbst_node =
            ((struct tbst_node *) ((char *) p - offsetof (struct tbst_node, tbst_data)));
        return *p;
    } else {
        tbst_t_init (trav, tree);
        return NULL;
    }
}
```
This code is included in §268, §395, and §546.

7.8.6 Initialization by Copying

§274 ⟨ TBST traverser copy initializer 274 ⟩ ≡
```
void *tbst_t_copy (struct tbst_traverser *trav, const struct tbst_traverser *src) {
    assert (trav != NULL && src != NULL);
    trav->tbst_table = src->tbst_table;
    trav->tbst_node = src->tbst_node;
    return trav->tbst_node != NULL ? trav->tbst_node->tbst_data : NULL;
}
```
This code is included in §268, §395, §502, and §546.

7.8.7 Advancing to the Next Node

Despite the earlier discussion (see Section 7.8 [Traversing a TBST], page 173), there are actually three cases, not two, in advancing within a threaded binary tree. The extra case turns up when the current node is the null item. We deal with that case by calling out to *tbst_t_first*().

Notice also that, below, in the case of following a thread we must check for a null node, but not in the case of following a child pointer.

§275 ⟨ TBST traverser advance function 275 ⟩ ≡
```
void *tbst_t_next (struct tbst_traverser *trav) {
    assert (trav != NULL);
    if (trav->tbst_node == NULL)
        return tbst_t_first (trav, trav->tbst_table);
```

```
    else if (trav→tbst_node→tbst_tag[1] == TBST_THREAD) {
        trav→tbst_node = trav→tbst_node→tbst_link[1];
        return trav→tbst_node != NULL ? trav→tbst_node→tbst_data : NULL;
    } else {
        trav→tbst_node = trav→tbst_node→tbst_link[1];
        while (trav→tbst_node→tbst_tag[0] == TBST_CHILD)
            trav→tbst_node = trav→tbst_node→tbst_link[0];
        return trav→tbst_node→tbst_data;
    }
}
```

This code is included in §268.

See also: [Knuth 1997], algorithm 2.3.1S.

7.8.8 Backing Up to the Previous Node

§276 ⟨ TBST traverser back up function 276 ⟩ ≡

```
void *tbst_t_prev (struct tbst_traverser *trav) {
    assert (trav != NULL);
    if (trav→tbst_node == NULL)
        return tbst_t_last (trav, trav→tbst_table);
    else if (trav→tbst_node→tbst_tag[0] == TBST_THREAD) {
        trav→tbst_node = trav→tbst_node→tbst_link[0];
        return trav→tbst_node != NULL ? trav→tbst_node→tbst_data : NULL;
    } else {
        trav→tbst_node = trav→tbst_node→tbst_link[0];
        while (trav→tbst_node→tbst_tag[1] == TBST_CHILD)
            trav→tbst_node = trav→tbst_node→tbst_link[1];
        return trav→tbst_node→tbst_data;
    }
}
```

This code is included in §268.

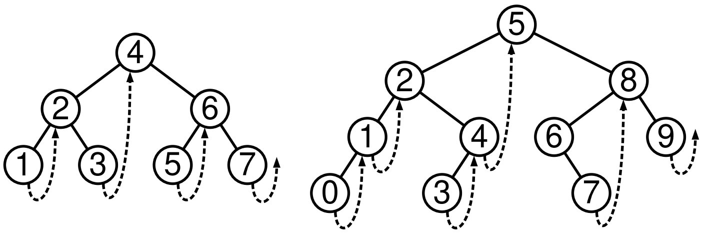
7.9 Copying

We can use essentially the same algorithm to copy threaded BSTs as unthreaded (see ⟨ BST copy function 83 ⟩). Some modifications are necessary, of course. The most obvious change is that the threads must be set up. This is not hard. We can do it the same way that *tbst_probe*() does.

Less obvious is the way to get rid of the stack. In *bst_copy*(), the stack was used to keep track of as yet incompletely processed parents of the current node. When we came back to one of these nodes, we did the actual copy of the node data, then visited the node's right subtree, if non-empty.

In a threaded tree, we can replace the use of the stack by the use of threads. Instead of popping an item off the stack when we can't move down in the tree any further, we follow the node's right thread. This brings us up to an ancestor (parent, grandparent, . . .) of the node, which we can then deal with in the same way as before.

This diagram shows the threads that would be followed to find parents in copying a couple of different threaded binary trees. Of course, the TBSTs would have complete sets of threads, but only the ones that are followed are shown:



Why does following the right thread from a node bring us to one of the node's ancestors? Consider the algorithm for finding the successor of a node with no right child, described earlier (see Section 4.9.3 [Better Iterative Traversal], page 53). This algorithm just moves up the tree from a node to its parent, grandparent, etc., guaranteeing that the successor will be a ancestor of the original node.

How do we know that following the right thread won't take us too far up the tree and skip copying some subtree? Because we only move up to the right one time using that same algorithm. When we move up to the left, we're going back to some binary tree whose right subtree we've already dealt with (we are currently in the right subtree of that binary tree, so of course we've dealt with it).

In conclusion, following the right thread always takes us to just the node whose right subtree we want to copy next. Of course, if that node happens to have an empty right subtree, then there is nothing to do, so we just continue along the next right thread, and so on.

The first step is to build a function to copy a single node. The following function *copy_node*() does this, creating a new node as the child of an existing node:

§277 ⟨ TBST node copy function 277 ⟩ ≡

```
/* Creates a new node as a child of dst on side dir.
   Copies data from src into the new node, applying copy(), if non-null.
   Returns nonzero only if fully successful.
   Regardless of success, integrity of the tree structure is assured,
   though failure may leave a null pointer in a tbst_data member. */
static int copy_node (struct tbst_table *tree, struct tbst_node *dst, int dir,
                      const struct tbst_node *src, tbst_copy_func *copy) {
  struct tbst_node *new = tree->tbst_alloc->libavl_malloc (tree->tbst_alloc, sizeof *new);
  if (new == NULL)
    return 0;

  new->tbst_link[dir] = dst->tbst_link[dir];
  new->tbst_tag[dir] = TBST_THREAD;
  new->tbst_link[!dir] = dst;
  new->tbst_tag[!dir] = TBST_THREAD;
  dst->tbst_link[dir] = new;
  dst->tbst_tag[dir] = TBST_CHILD;

  if (copy == NULL)
    new->tbst_data = src->tbst_data;
```

```
    else {
        new->tbst_data = copy (src->tbst_data, tree->tbst_param);
        if (new->tbst_data == NULL)
            return 0;
    }
    return 1;
}
```

This code is included in §278.

Using the node copy function above, constructing the tree copy function is easy. In fact, the code is considerably easier to read than our original function to iteratively copy an unthreaded binary tree (see Section 4.10.3 [Handling Errors in Iterative BST Copying], page 64), because this function is not as heavily optimized.

One tricky part is getting the copy started. We can't use the dirty trick from *bst_copy*() of casting the address of a *bst_root* to a node pointer, because we need access to the first tag as well as the first link (see Exercise 2 for a way to sidestep this problem). So instead we use a couple of "pseudo-root" nodes *rp* and *rq*, allocated locally.

§278 ⟨ TBST copy function 278 ⟩ ≡

```
⟨ TBST node copy function 277 ⟩
⟨ TBST copy error helper function 280 ⟩
⟨ TBST main copy function 279 ⟩
```

This code is included in §251.

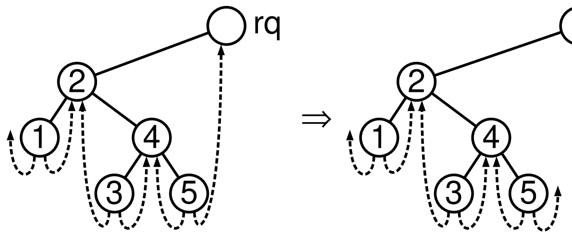
§279 ⟨ TBST main copy function 279 ⟩ ≡

```
struct tbst_table *tbst_copy (const struct tbst_table *org, tbst_copy_func *copy,
                              tbst_item_func *destroy, struct libavl_allocator *allocator) {
    struct tbst_table *new;
    const struct tbst_node *p;
    struct tbst_node *q;
    struct tbst_node rp, rq;
    assert (org != NULL);
    new = tbst_create (org->tbst_compare, org->tbst_param,
                       allocator != NULL ? allocator : org->tbst_alloc);
    if (new == NULL)
        return NULL;
    new->tbst_count = org->tbst_count;
    if (new->tbst_count == 0)
        return new;
    p = &rp;
    rp.tbst_link[0] = org->tbst_root;
    rp.tbst_tag[0] = TBST_CHILD;
    q = &rq;
    rq.tbst_link[0] = NULL;
    rq.tbst_tag[0] = TBST_THREAD;
    for (;;) {
        if (p->tbst_tag[0] == TBST_CHILD) {
```

```
            if (!copy_node (new, q, 0, p→tbst_link[0], copy)) {
                copy_error_recovery (rq.tbst_link[0], new, destroy);
                return NULL;
            }
            p = p→tbst_link[0];
            q = q→tbst_link[0];
        } else {
            while (p→tbst_tag[1] == TBST_THREAD) {
                p = p→tbst_link[1];
                if (p == NULL) {
                    q→tbst_link[1] = NULL;
                    new→tbst_root = rq.tbst_link[0];
                    return new;
                }
                q = q→tbst_link[1];
            }
            p = p→tbst_link[1];
            q = q→tbst_link[1];
        }
        if (p→tbst_tag[1] == TBST_CHILD)
            if (!copy_node (new, q, 1, p→tbst_link[1], copy)) {
                copy_error_recovery (rq.tbst_link[0], new, destroy);
                return NULL;
            }
    }
}
```

This code is included in §278 and §329.

A sensitive issue in the code above is treatment of the final thread. The initial call to *copy_node*() causes a right thread to point to *rq*, but it needs to be a null pointer. We need to perform this kind of transformation:



When the copy is successful, this is just a matter of setting the final *q*'s right child pointer to NULL, but when it is unsuccessful we have to find the pointer in question, which is in the greatest node in the tree so far (to see this, try constructing a few threaded BSTs by hand on paper). Function *copy_error_recovery*() does this, as well as destroying the tree. It also handles the case of failure when no nodes have yet been added to the tree:

§280 ⟨ TBST copy error helper function 280 ⟩ ≡

```
static void copy_error_recovery (struct tbst_node *p,
                                 struct tbst_table *new, tbst_item_func *destroy) {
```

```
    new→tbst_root = p;
    if (p != NULL) {
        while (p→tbst_tag[1] == TBST_CHILD)
            p = p→tbst_link[1];
        p→tbst_link[1] = NULL;
    }
    tbst_destroy (new, destroy);
}
```

This code is included in §278 and §329.

Exercises:

1. In the diagram above that shows examples of threads followed while copying a TBST, all right threads in the TBSTs are shown. Explain how this is not just a coincidence.

2. Suggest some optimization possibilities for *tbst_copy*().

7.10 Destruction

Destroying a threaded binary tree is easy. We can simply traverse the tree in inorder in the usual way. We always have a way to get to the next node without having to go back up to any of the nodes we've already destroyed. (We do, however, have to make sure to go find the next node before destroying the current one, in order to avoid reading data from freed memory.) Here's all it takes:

§281 ⟨ TBST destruction function 281 ⟩ ≡

```
void tbst_destroy (struct tbst_table *tree, tbst_item_func *destroy) {
    struct tbst_node *p; /* Current node. */
    struct tbst_node *n; /* Next node. */

    p = tree→tbst_root;
    if (p != NULL)
        while (p→tbst_tag[0] == TBST_CHILD)
            p = p→tbst_link[0];

    while (p != NULL) {
        n = p→tbst_link[1];
        if (p→tbst_tag[1] == TBST_CHILD)
            while (n→tbst_tag[0] == TBST_CHILD)
                n = n→tbst_link[0];

        if (destroy != NULL && p→tbst_data != NULL)
            destroy (p→tbst_data, tree→tbst_param);
        tree→tbst_alloc→libavl_free (tree→tbst_alloc, p);

        p = n;
    }

    tree→tbst_alloc→libavl_free (tree→tbst_alloc, tree);
}
```

This code is included in §251, §300, and §336.

7.11 Balance

Just like their unthreaded cousins, threaded binary trees can become degenerate, leaving their good performance characteristics behind. When this happened in a unthreaded BST, stack overflow often made it necessary to rebalance the tree. This doesn't happen in our implementation of threaded BSTs, because none of the routines uses a stack. It is still useful to have a rebalance routine for performance reasons, so we will implement one, in this section, anyway.

There is no need to change the basic algorithm. As before, we convert the tree to a linear "vine", then the vine to a balanced binary search tree. See Section 4.12 [Balancing a BST], page 70, for a review of the balancing algorithm.

Here is the outline and prototype for *tbst_balance*().

§282 ⟨ TBST balance function 282 ⟩ ≡
```
⟨ TBST tree-to-vine function 284 ⟩
⟨ TBST vine compression function 286 ⟩
⟨ TBST vine-to-tree function 285 ⟩
⟨ TBST main balance function 283 ⟩
```
This code is included in §251.

§283 ⟨ TBST main balance function 283 ⟩ ≡
```
/* Balances tree. */
void tbst_balance (struct tbst_table *tree) {
    assert (tree != NULL);

    tree_to_vine (tree);
    vine_to_tree (tree);
}
```
This code is included in §282 and §408.

7.11.1 From Tree to Vine

We could transform a threaded binary tree into a vine in the same way we did for unthreaded binary trees, by use of rotations (see Section 4.12.1 [Transforming a BST into a Vine], page 72). But one of the reasons we did it that way was to avoid use of a stack, which is no longer a problem. It's now simpler to rearrange nodes by inorder traversal.

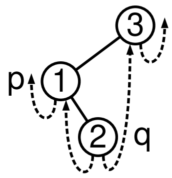
We start by finding the minimum node in the tree as p, which will step through the tree in inorder. During each trip through the main loop, we find p's successor as q and make p the left child of q. We also have to make sure that p's right thread points to q. That's all there is to it.

§284 ⟨ TBST tree-to-vine function 284 ⟩ ≡
```
static void tree_to_vine (struct tbst_table *tree) {
    struct tbst_node *p;

    if (tree→tbst_root == NULL)
        return;

    p = tree→tbst_root;
    while (p→tbst_tag[0] == TBST_CHILD)
        p = p→tbst_link[0];
```

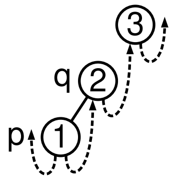
```
    for (;;) {
        struct tbst_node *q = p->tbst_link[1];
        if (p->tbst_tag[1] == TBST_CHILD) {
            while (q->tbst_tag[0] == TBST_CHILD)
                q = q->tbst_link[0];
            p->tbst_tag[1] = TBST_THREAD;
            p->tbst_link[1] = q;
        }
        if (q == NULL)
            break;
        q->tbst_tag[0] = TBST_CHILD;
        q->tbst_link[0] = p;
        p = q;
    }
    tree->tbst_root = p;
}
```

This code is included in §282.

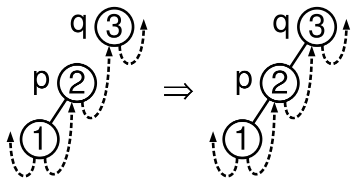
Sometimes one trip through the main loop above will put the TBST into an inconsistent state, where two different nodes are the parent of a third node. Such an inconsistency is always corrected in the next trip through the loop. An example is warranted. Suppose the original threaded binary tree looks like this, with nodes p and q for the initial iteration of the loop as marked:



The first trip through the loop makes p, 1, the child of q, 2, but p's former parent's left child pointer still points to p. We now have a situation where node 1 has two parents: both 2 and 3. This diagram tries to show the situation by omitting the line that would otherwise lead down from 3 to 2:



On the other hand, node 2's right thread still points to 3, so on the next trip through the loop there is no trouble finding the new p's successor. Node 3 is made the parent of 2 and all is well. This diagram shows the new p and q, then the fixed-up vine. The only difference is that node 3 now, correctly, has 2 as its left child:



7.11.2 From Vine to Balanced Tree

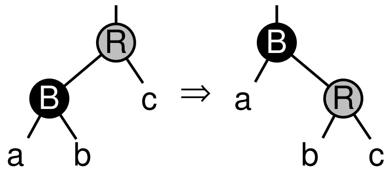
Transforming a vine into a balanced threaded BST is similar to the same operation on an unthreaded BST. We can use the same algorithm, adjusting it for presence of the threads. The following outline is similar to ⟨ BST balance function 87 ⟩. In fact, we entirely reuse ⟨ Calculate *leaves* 91 ⟩, just changing *bst* to *tbst*. We omit the final check on the tree's height, because none of the TBST functions are height-limited.

§285 ⟨ TBST vine-to-tree function 285 ⟩ ≡

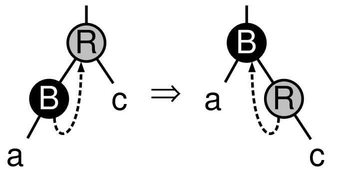
```
static void vine_to_tree (struct tbst_table *tree) {
    unsigned long vine; /* Number of nodes in main vine. */
    unsigned long leaves; /* Nodes in incomplete bottom level, if any. */
    int height; /* Height of produced balanced tree. */
    ⟨ Calculate leaves; bst ⇒ tbst 91 ⟩
    ⟨ Reduce TBST vine general case to special case 287 ⟩
    ⟨ Make special case TBST vine into balanced tree and count height 288 ⟩
}
```

This code is included in §282 and §408.

Not many changes are needed to adapt the algorithm to handle threads. Consider the basic right rotation transformation used during a compression:



The rotation does not disturb *a* or *c*, so the only node that can cause trouble is *b*. If *b* is a real child node, then there's no need to do anything differently. But if *b* is a thread, then we have to swap around the direction of the thread, like this:



After a rotation that involves a thread, the next rotation on *B* will not involve a thread. So after we perform a rotation that adjusts a thread in one place, the next one in the same place will not require a thread adjustment.

Every node in the vine we start with has a thread as its right link. This means that during the first pass along the main vine we must perform thread adjustments at every node, but subsequent passes along the vine must not perform any adjustments.

This simple idea is complicated by the initial partial compression pass in trees that do not have exactly one fewer than a power of two nodes. After a partial compression pass, the nodes at the top of the main vine no longer have right threads, but the ones farther down still do.

We deal with this complication by defining the *compress*() function so it can handle a mixture of rotations with and without right threads. The rotations that need thread adjustments will always be below the ones that do not, so this function simply takes a pair of parameters, the first specifying how many rotations without thread adjustment to perform, the next how many with thread adjustment. Compare this code to that for unthreaded BSTs:

§286 ⟨ TBST vine compression function 286 ⟩ ≡

```
/* Performs a nonthreaded compression operation nonthread times,
   then a threaded compression operation thread times, starting at root. */
static void compress (struct tbst_node *root,
                      unsigned long nonthread, unsigned long thread) {
  assert (root != NULL);
  while (nonthread--) {
    struct tbst_node *red = root→tbst_link[0];
    struct tbst_node *black = red→tbst_link[0];
    root→tbst_link[0] = black;
    red→tbst_link[0] = black→tbst_link[1];
    black→tbst_link[1] = red;
    root = black;
  }
  while (thread--) {
    struct tbst_node *red = root→tbst_link[0];
    struct tbst_node *black = red→tbst_link[0];
    root→tbst_link[0] = black;
    red→tbst_link[0] = black;
    red→tbst_tag[0] = TBST_THREAD;
    black→tbst_tag[1] = TBST_CHILD;
    root = black;
  }
}
```

This code is included in §282.

When we reduce the general case to the $2^n - 1$ special case, all of the rotations adjust threads:

§287 ⟨ Reduce TBST vine general case to special case 287 ⟩ ≡

```
compress ((struct tbst_node *) &tree→tbst_root, 0, leaves);
```

This code is included in §285.

We deal with the first compression specially, in order to clean up any remaining unadjusted threads:

§288 ⟨ Make special case TBST vine into balanced tree and count height 288 ⟩ ≡

```
vine = tree→tbst_count - leaves;
height = 1 + (leaves > 0);
if (vine > 1) {
  unsigned long nonleaves = vine / 2;
  leaves /= 2;
  if (leaves > nonleaves) {
    leaves = nonleaves;
```

```
        nonleaves = 0;
    }
    else nonleaves -= leaves;
    compress ((struct tbst_node *) &tree→tbst_root, leaves, nonleaves);
    vine /= 2;
    height++;
}
```

See also §289.
This code is included in §285.

After this, all the remaining compressions use only rotations without thread adjustment, and we're done:

§289 ⟨ Make special case TBST vine into balanced tree and count height 288 ⟩ +≡

```
while (vine > 1) {
    compress ((struct tbst_node *) &tree→tbst_root, vine / 2, 0);
    vine /= 2;
    height++;
}
```

7.12 Testing

There's little new in the testing code. We do add an test for *tbst_balance*(), because none of the existing tests exercise it. This test doesn't check that *tbst_balance*() actually balances the tree, it just verifies that afterwards the tree contains the items it should, so to be certain that balancing is correct, turn up the verbosity and look at the trees printed.

Function *print_tree_structure*() prints thread node numbers preceded by '`>`', with null threads indicated by '`>>`'. This notation is compatible with the plain text output format of the `texitree` program used to draw the binary trees in this book. (It will cause errors for PostScript output because it omits node names.)

§290 ⟨ `tbst-test.c` 290 ⟩ ≡

```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ limits.h ⟩
#include ⟨ stdio.h ⟩
#include "tbst.h"
#include "test.h"
⟨ TBST print function 291 ⟩
⟨ BST traverser check function; bst ⇒ tbst 104 ⟩
⟨ Compare two TBSTs for structure and content 292 ⟩
⟨ Recursively verify TBST structure 293 ⟩
⟨ TBST verify function 294 ⟩
⟨ TBST test function 295 ⟩
⟨ BST overflow test function; bst ⇒ tbst 122 ⟩
```

§291 ⟨ TBST print function 291 ⟩ ≡

```
void print_tree_structure (struct tbst_node *node, int level) {
    int i;
```

```
    if (level > 16) {
        printf ("[...]");
        return;
    }
    if (node == NULL) {
        printf ("<nil>");
        return;
    }
    printf ("%d(", node→tbst_data ? *(int *) node→tbst_data : -1);
    for (i = 0; i <= 1; i++) {
        if (node→tbst_tag[i] == TBST_CHILD) {
            if (node→tbst_link[i] == node) printf ("loop");
            else print_tree_structure (node→tbst_link[i], level + 1);
        }
        else if (node→tbst_link[i] != NULL)
            printf (">%d", (node→tbst_link[i]→tbst_data
                            ? *(int *) node→tbst_link[i]→tbst_data : -1));
        else printf (">>");
        if (i == 0) fputs (",␣", stdout);
    }
    putchar (')');
}
void print_whole_tree (const struct tbst_table *tree, const char *title) {
    printf ("%s:␣", title);
    print_tree_structure (tree→tbst_root, 0);
    putchar ('\n');
}
```

This code is included in §290, §330, and §368.

§292 ⟨ Compare two TBSTs for structure and content 292 ⟩ ≡

```
static int compare_trees (struct tbst_node *a, struct tbst_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
        if (a != NULL || b != NULL) {
            printf ("␣a=%d␣b=%d\n",
                    a ? *(int *) a→tbst_data : -1, b ? *(int *) b→tbst_data : -1);
            assert (0);
        }
        return 1;
    }
    assert (a != b);
    if (*(int *) a→tbst_data != *(int *) b→tbst_data
        || a→tbst_tag[0] != b→tbst_tag[0] || a→tbst_tag[1] != b→tbst_tag[1]) {
        printf ("␣Copied␣nodes␣differ:␣a=%d␣b=%d␣a:",
                *(int *) a→tbst_data, *(int *) b→tbst_data);
        if (a→tbst_tag[0] == TBST_CHILD) printf ("l");
```

```
        if (a->tbst_tag[1] == TBST_CHILD) printf ("r");
        printf (" b:");
        if (b->tbst_tag[0] == TBST_CHILD) printf ("l");
        if (b->tbst_tag[1] == TBST_CHILD) printf ("r");
        printf ("\n");
        return 0;
    }
    if (a->tbst_tag[0] == TBST_THREAD)
        assert ((a->tbst_link[0] == NULL) != (a->tbst_link[0] != b->tbst_link[0]));
    if (a->tbst_tag[1] == TBST_THREAD)
        assert ((a->tbst_link[1] == NULL) != (a->tbst_link[1] != b->tbst_link[1]));
    okay = 1;
    if (a->tbst_tag[0] == TBST_CHILD)
        okay &= compare_trees (a->tbst_link[0], b->tbst_link[0]);
    if (a->tbst_tag[1] == TBST_CHILD)
        okay &= compare_trees (a->tbst_link[1], b->tbst_link[1]);
    return okay;
}
```

This code is included in §290.

§293 ⟨ Recursively verify TBST structure 293 ⟩ ≡

```
static void recurse_verify_tree (struct tbst_node *node, int *okay, size_t *count,
                                 int min, int max) {
    int d; /* Value of this node's data. */
    size_t subcount[2]; /* Number of nodes in subtrees. */
    if (node == NULL) {
        *count = 0;
        return;
    }
    d = *(int *) node->tbst_data;
    ⟨ Verify binary search tree ordering 114 ⟩
    subcount[0] = subcount[1] = 0;
    if (node->tbst_tag[0] == TBST_CHILD)
        recurse_verify_tree (node->tbst_link[0], okay, &subcount[0], min, d - 1);
    if (node->tbst_tag[1] == TBST_CHILD)
        recurse_verify_tree (node->tbst_link[1], okay, &subcount[1], d + 1, max);
    *count = 1 + subcount[0] + subcount[1];
}
```

This code is included in §290.

§294 ⟨ TBST verify function 294 ⟩ ≡

```
static int verify_tree (struct tbst_table *tree, int array[], size_t n) {
    int okay = 1;
    ⟨ Check tree->bst_count is correct; bst ⇒ tbst 110 ⟩
    if (okay) { ⟨ Check BST structure; bst ⇒ tbst 111 ⟩ }
    if (okay) { ⟨ Check that the tree contains all the elements it should; bst ⇒ tbst 115 ⟩ }
```

```
    if (okay) { ⟨Check that forward traversal works; bst ⇒ tbst 116⟩ }
    if (okay) { ⟨Check that backward traversal works; bst ⇒ tbst 117⟩ }
    if (okay) { ⟨Check that traversal from the null element works; bst ⇒ tbst 118⟩ }
    return okay;
}
```

This code is included in §290.

§295 ⟨TBST test function 295⟩ ≡

```
int test_correctness (struct libavl_allocator *allocator,
                      int insert[], int delete[], int n, int verbosity) {
    struct tbst_table *tree;
    int okay = 1;
    int i;
    ⟨Test creating a BST and inserting into it; bst ⇒ tbst 102⟩
    ⟨Test BST traversal during modifications; bst ⇒ tbst 103⟩
    ⟨Test deleting nodes from the BST and making copies of it; bst ⇒ tbst 105⟩
    ⟨Test destroying the tree; bst ⇒ tbst 108⟩
    ⟨Test TBST balancing 296⟩
    return okay;
}
```

This code is included in §290, §411, and §515.

§296 ⟨Test TBST balancing 296⟩ ≡

```
/* Test tbst_balance(). */
if (verbosity >= 2) printf ("  Testing balancing...\n");
tree = tbst_create (compare_ints, NULL, allocator);
if (tree == NULL) {
    if (verbosity >= 0) printf ("  Out of memory creating tree.\n");
    return 1;
}
for (i = 0; i < n; i++) {
    void **p = tbst_probe (tree, &insert[i]);
    if (p == NULL) {
        if (verbosity >= 0) printf ("    Out of memory in insertion.\n");
        tbst_destroy (tree, NULL);
        return 1;
    }
    if (*p != &insert[i]) printf ("    Duplicate item in tree!\n");
}
if (verbosity >= 4) print_whole_tree (tree, "    Prebalance");
tbst_balance (tree);
if (verbosity >= 4) print_whole_tree (tree, "    Postbalance");
if (!verify_tree (tree, insert, n))
    return 0;
tbst_destroy (tree, NULL);
```

This code is included in §295.

8 Threaded AVL Trees

The previous chapter introduced a new concept in BSTs, the idea of threads. Threads allowed us to simplify traversals and eliminate the use of stacks. On the other hand, threaded trees can still grow tall enough that they reduce the program's performance unacceptably, the problem that balanced trees were meant to solve. Ideally, we'd like to add threads to balanced trees, to produce threaded balanced trees that combine the best of both worlds.

We can do this, and it's not even very difficult. This chapter will show how to add threads to AVL trees. The next will show how to add them to red-black trees.

Here's an outline of the table implementation for threaded AVL or "TAVL" trees that we'll develop in this chapter. Note the usage of prefix *tavl_* for these functions.

§297 ⟨ `tavl.h` 297 ⟩ ≡
```
⟨ License 1 ⟩
#ifndef TAVL_H
#define TAVL_H 1

#include ⟨ stddef.h ⟩

⟨ Table types; tbl ⇒ tavl 14 ⟩
⟨ BST maximum height; bst ⇒ tavl 28 ⟩
⟨ TBST table structure; tbst ⇒ tavl 250 ⟩
⟨ TAVL node structure 299 ⟩
⟨ TBST traverser structure; tbst ⇒ tavl 267 ⟩
⟨ Table function prototypes; tbl ⇒ tavl 15 ⟩

#endif /* tavl.h */
```

§298 ⟨ `tavl.c` 298 ⟩ ≡
```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "tavl.h"

⟨ TAVL functions 300 ⟩
```

8.1 Data Types

The TAVL node structure takes the basic fields for a BST and adds a balance factor for AVL balancing and a pair of tag fields to allow for threading.

§299 ⟨ TAVL node structure 299 ⟩ ≡
```
/* Characterizes a link as a child pointer or a thread. */
enum tavl_tag {
    TAVL_CHILD, /* Child pointer. */
    TAVL_THREAD /* Thread. */
};

/* An TAVL tree node. */
struct tavl_node {
    struct tavl_node *tavl_link[2]; /* Subtrees. */
```

```
    void *tavl_data; /* Pointer to data. */
    unsigned char tavl_tag[2]; /* Tag fields. */
    signed char tavl_balance; /* Balance factor. */
};
```

This code is included in §297.

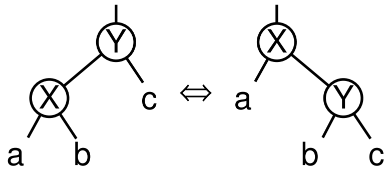
Exercises:

1. **struct avl_node** contains three pointer members and a single character member, whereas **struct tavl_node** additionally contains an array of two characters. Is **struct tavl_node** necessarily larger than **struct avl_node**?

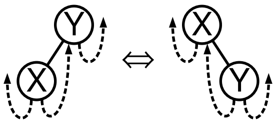
8.2 Rotations

Rotations are just as useful in threaded BSTs as they are in unthreaded ones. We do need to re-examine the idea, though, to see how the presence of threads affect rotations.

A generic rotation looks like this diagram taken from Section 4.3 [BST Rotations], page 33:

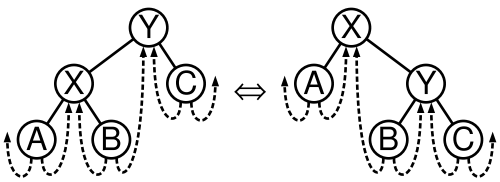


Any of the subtrees labeled a, b, and c may be in fact threads. In the most extreme case, all of them are threads, and the rotation looks like this:



As you can see, the thread from X to Y, represented by subtree b, reverses direction and becomes a thread from Y to X following a right rotation. This has to be handled as a special case in code for rotation. See Exercise 1 for details.

On the other hand, there is no need to do anything special with threads originating in subtrees of a rotated node. This is a direct consequence of the locality and order-preserving properties of a rotation (see Section 4.3 [BST Rotations], page 33). Here's an example diagram to demonstrate. Note in particular that the threads from A, B, and C point to the same nodes in both trees:



Exercises:

1. Write functions for right and left rotations in threaded BSTs, analogous to those for unthreaded BSTs developed in Exercise 4.3-2.

8.3 Operations

Now we'll implement all the usual operations for TAVL trees. We can reuse everything from TBSTs except insertion, deletion, and copy functions. Most of the copy function code will in fact be reused also. Here's the outline:

§300 ⟨ TAVL functions 300 ⟩ ≡
⟨ TBST creation function; tbst ⇒ tavl 252 ⟩
⟨ TBST search function; tbst ⇒ tavl 253 ⟩
⟨ TAVL item insertion function 301 ⟩
⟨ Table insertion convenience functions; tbl ⇒ tavl 592 ⟩
⟨ TAVL item deletion function 311 ⟩
⟨ TBST traversal functions; tbst ⇒ tavl 268 ⟩
⟨ TAVL copy function 329 ⟩
⟨ TBST destruction function; tbst ⇒ tavl 281 ⟩
⟨ Default memory allocation functions; tbl ⇒ tavl 6 ⟩
⟨ Table assertion functions; tbl ⇒ tavl 594 ⟩

This code is included in §298.

8.4 Insertion

Insertion into an AVL tree is not complicated much by the need to update threads. The outline is the same as before, and the code for step 3 and the local variable declarations can be reused entirely:

§301 ⟨ TAVL item insertion function 301 ⟩ ≡
void ***tavl_probe* (**struct tavl_table** **tree*, **void** **item*) {
    ⟨ *avl_probe*() local variables; avl ⇒ tavl 147 ⟩
    assert (*tree* != `NULL` && *item* != `NULL`);
    ⟨ Step 1: Search TAVL tree for insertion point 302 ⟩
    ⟨ Step 2: Insert TAVL node 303 ⟩
    ⟨ Step 3: Update balance factors after AVL insertion; avl ⇒ tavl 150 ⟩
    ⟨ Step 4: Rebalance after TAVL insertion 304 ⟩
}

This code is included in §300.

8.4.1 Steps 1 and 2: Search and Insert

The first step is a lot like the unthreaded AVL version in ⟨ Step 1: Search AVL tree for insertion point 148 ⟩. There is an unfortunate special case for an empty tree, because a null pointer for *tavl_root* indicates an empty tree but in a nonempty tree we must seek a thread link. After we're done, p, not q as before, is the node below which a new node should be inserted, because the test for stepping outside the binary tree now comes before advancing p.

§302 ⟨ Step 1: Search TAVL tree for insertion point 302 ⟩ ≡
z = (**struct tavl_node** *) &*tree*→*tavl_root*;
y = *tree*→*tavl_root*;
if (y != `NULL`) {

```
    for (q = z, p = y; ; q = p, p = p->tavl_link[dir]) {
        int cmp = tree->tavl_compare (item, p->tavl_data, tree->tavl_param);
        if (cmp == 0)
            return &p->tavl_data;
        if (p->tavl_balance != 0)
            z = q, y = p, k = 0;
        da[k++] = dir = cmp > 0;
        if (p->tavl_tag[dir] == TAVL_THREAD)
            break;
    }
} else {
    p = z;
    dir = 0;
}
```

This code is included in §301.

The insertion adds to the TBST code by setting the balance factor of the new node and handling the first insertion into an empty tree as a special case:

§303 ⟨ Step 2: Insert TAVL node 303 ⟩ ≡

```
⟨ Step 2: Insert TBST node; tbst ⇒ tavl 256 ⟩
n->tavl_balance = 0;
if (tree->tavl_root == n)
    return &n->tavl_data;
```

This code is included in §301.

8.4.2 Step 4: Rebalance

Now we're finally to the interesting part, the rebalancing step. We can tell whether rebalancing is necessary based on the balance factor of y, the same as in unthreaded AVL insertion:

§304 ⟨ Step 4: Rebalance after TAVL insertion 304 ⟩ ≡

```
if (y->tavl_balance == -2)
    { ⟨ Rebalance TAVL tree after insertion in left subtree 305 ⟩ }
else if (y->tavl_balance == +2)
    { ⟨ Rebalance TAVL tree after insertion in right subtree 308 ⟩ }
else return &n->tavl_data;
z->tavl_link[y != z->tavl_link[0]] = w;
return &n->tavl_data;
```

This code is included in §301.

We will examine the case of insertion in the left subtree of y, the node at which we must rebalance. We take x as y's child on the side of the new node, then, as for unthreaded AVL insertion, we distinguish two cases based on the balance factor of x:

§305 ⟨ Rebalance TAVL tree after insertion in left subtree 305 ⟩ ≡

```
struct tavl_node *x = y->tavl_link[0];
if (x->tavl_balance == -1)
    { ⟨ Rebalance for - balance factor in TAVL insertion in left subtree 306 ⟩ }
```

else { ⟨ Rebalance for + balance factor in TAVL insertion in left subtree 307 ⟩ }

This code is included in §304.

Case 1: x has − balance factor

As for unthreaded insertion, we rotate right at y (see Section 5.4.4 [Rebalancing AVL Trees], page 115). Notice the resemblance of the following code to *rotate_right*() in the solution to Exercise 8.2-1.

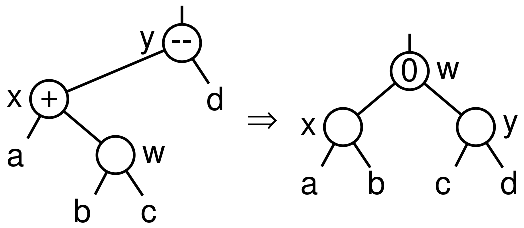
§306 ⟨ Rebalance for − balance factor in TAVL insertion in left subtree 306 ⟩ ≡

```
w = x;
if (x→tavl_tag[1] == TAVL_THREAD) {
    x→tavl_tag[1] = TAVL_CHILD;
    y→tavl_tag[0] = TAVL_THREAD;
    y→tavl_link[0] = x;
}
else y→tavl_link[0] = x→tavl_link[1];
x→tavl_link[1] = y;
x→tavl_balance = y→tavl_balance = 0;
```

This code is included in §305.

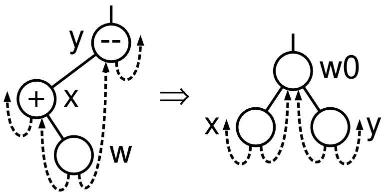
Case 2: x has + balance factor

When x has a + balance factor, we perform the transformation shown below, which consists of a left rotation at x followed by a right rotation at y. This is the same transformation used in unthreaded insertion:



We could simply apply the standard code from Exercise 8.2-1 in each rotation (see Exercise 1), but it is just as straightforward to do both of the rotations together, then clean up any threads. Subtrees a and d cannot cause thread-related trouble, because they are not disturbed during the transformation: a remains x's left child and d remains y's right child. The children of w, subtrees b and c, do require handling. If subtree b is a thread, then after the rotation and before fix-up x's right link points to itself, and, similarly, if c is a thread then y's left link points to itself. These links must be changed into threads to w instead, and w's links must be tagged as child pointers.

If both b and c are threads then the transformation looks like the diagram below, showing pre-rebalancing and post-rebalancing, post-fix-up views. The AVL balance rule implies that if b and c are threads then a and d are also:



The required code is heavily based on the corresponding code for unthreaded AVL rebalancing:

§307 ⟨ Rebalance for + balance factor in TAVL insertion in left subtree 307 ⟩ ≡
⟨ Rotate left at x then right at y in AVL tree; avl ⇒ tavl 156 ⟩
```
if (w→tavl_tag[0] == TAVL_THREAD) {
    x→tavl_tag[1] = TAVL_THREAD;
    x→tavl_link[1] = w;
    w→tavl_tag[0] = TAVL_CHILD;
}
if (w→tavl_tag[1] == TAVL_THREAD) {
    y→tavl_tag[0] = TAVL_THREAD;
    y→tavl_link[0] = w;
    w→tavl_tag[1] = TAVL_CHILD;
}
```
This code is included in §305, §324, and §667.

Exercises:

1. Rewrite ⟨ Rebalance for + balance factor in TAVL insertion in left subtree 307 ⟩ in terms of the routines from Exercise 8.2-1.

8.4.3 Symmetric Case

Here is the corresponding code for the case where insertion occurs in the right subtree of y.

§308 ⟨ Rebalance TAVL tree after insertion in right subtree 308 ⟩ ≡
```
struct tavl_node *x = y→tavl_link[1];
if (x→tavl_balance == +1)
    { ⟨ Rebalance for + balance factor in TAVL insertion in right subtree 309 ⟩ }
else { ⟨ Rebalance for − balance factor in TAVL insertion in right subtree 310 ⟩ }
```
This code is included in §304.

§309 ⟨ Rebalance for + balance factor in TAVL insertion in right subtree 309 ⟩ ≡
```
w = x;
if (x→tavl_tag[0] == TAVL_THREAD) {
    x→tavl_tag[0] = TAVL_CHILD;
    y→tavl_tag[1] = TAVL_THREAD;
    y→tavl_link[1] = x;
}
else y→tavl_link[1] = x→tavl_link[0];
x→tavl_link[0] = y;
x→tavl_balance = y→tavl_balance = 0;
```
This code is included in §308.

§310 ⟨ Rebalance for − balance factor in TAVL insertion in right subtree 310 ⟩ ≡

```
⟨ Rotate right at x then left at y in AVL tree; avl ⇒ tavl 159 ⟩
if (w→tavl_tag[0] == TAVL_THREAD) {
    y→tavl_tag[1] = TAVL_THREAD;
    y→tavl_link[1] = w;
    w→tavl_tag[0] = TAVL_CHILD;
}
if (w→tavl_tag[1] == TAVL_THREAD) {
    x→tavl_tag[0] = TAVL_THREAD;
    x→tavl_link[0] = w;
    w→tavl_tag[1] = TAVL_CHILD;
}
```

This code is included in §308, §320, and §666.

8.5 Deletion

Deletion from a TAVL tree can be accomplished by combining our knowledge about AVL trees and threaded trees. From one perspective, we add rebalancing to TBST deletion. From the other perspective, we add thread handling to AVL tree deletion.

The function outline is about the same as usual. We do add a helper function for finding the parent of a TAVL node:

§311 ⟨ TAVL item deletion function 311 ⟩ ≡

```
⟨ Find parent of a TBST node; tbst ⇒ tavl 327 ⟩
void *tavl_delete (struct tavl_table *tree, const void *item) {
    struct tavl_node *p; /* Traverses tree to find node to delete. */
    struct tavl_node *q; /* Parent of p. */
    int dir; /* Index into q→tavl_link[] to get p. */
    int cmp; /* Result of comparison between item and p. */

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search TAVL tree for item to delete 312 ⟩
    ⟨ Step 2: Delete item from TAVL tree 313 ⟩
    ⟨ Steps 3 and 4: Update balance factors and rebalance after TAVL deletion 318 ⟩
}
```

This code is included in §300.

8.5.1 Step 1: Search

We use p to search down the tree and keep track of p's parent with q. We keep the invariant at the beginning of the loop here that $q\to tavl\_link[dir] == p$. As the final step, we record the item deleted and update the tree's item count.

§312 ⟨ Step 1: Search TAVL tree for item to delete 312 ⟩ ≡

```
if (tree→tavl_root == NULL)
    return NULL;

q = (struct tavl_node *) &tree→tavl_root;
p = tree→tavl_root;
dir = 0;
```

```
for (;;)
  {
  cmp = tree→tavl_compare (item, p→tavl_data, tree→tavl_param);
  if (cmp == 0)
    break;
  dir = cmp > 0;

  q = p;
  if (p→tavl_tag[dir] == TAVL_THREAD)
    return NULL;
  p = p→tavl_link[dir];
}
item = p→tavl_data;
```

This code is included in §311 and §670.

8.5.2 Step 2: Delete

The cases for deletion are the same as for a TBST (see Section 7.7 [Deleting from a TBST], page 168). The difference is that we have to copy around balance factors and keep track of where balancing needs to start. After the deletion, q is the node at which balance factors must be updated and possible rebalancing occurs and *dir* is the side of q from which the node was deleted. For cases 1 and 2, q need not change from its current value as the parent of the deleted node. For cases 3 and 4, q will need to be changed.

§313 ⟨Step 2: Delete item from TAVL tree 313⟩ ≡

```
if (p→tavl_tag[1] == TAVL_THREAD) {
  if (p→tavl_tag[0] == TAVL_CHILD)
    { ⟨Case 1 in TAVL deletion 314⟩ }
  else { ⟨Case 2 in TAVL deletion 315⟩ }
} else {
  struct tavl_node *r = p→tavl_link[1];
  if (r→tavl_tag[0] == TAVL_THREAD)
    { ⟨Case 3 in TAVL deletion 316⟩ }
  else { ⟨Case 4 in TAVL deletion 317⟩ }
}
tree→tavl_alloc→libavl_free (tree→tavl_alloc, p);
```

This code is included in §311.

Case 1: *p* has a right thread and a left child

If p has a right thread and a left child, then we replace it by its left child. Rebalancing must begin right above p, which is already set as q. There's no need to change the TBST code:

§314 ⟨Case 1 in TAVL deletion 314⟩ ≡

```
⟨Case 1 in TBST deletion; tbst ⇒ tavl 260⟩
```

This code is included in §313.

Case 2: *p* has a right thread and a left thread

If p is a leaf, then we change q's pointer to p into a thread. Again, rebalancing must begin at the node that's already set up as q and there's no need to change the TBST code:

§315 ⟨ Case 2 in TAVL deletion 315 ⟩ ≡
⟨ Case 2 in TBST deletion; tbst ⇒ tavl 261 ⟩

This code is included in §313.

Case 3: *p*'s right child has a left thread

If p has a right child r, which in turn has no left child, then we move r in place of p. In this case r, having replaced p, acquires p's former balance factor and rebalancing must start from there. The deletion in this case is always on the right side of the node.

§316 ⟨ Case 3 in TAVL deletion 316 ⟩ ≡
⟨ Case 3 in TBST deletion; tbst ⇒ tavl 262 ⟩

```
r->tavl_balance = p->tavl_balance;
q = r;
dir = 1;
```

This code is included in §313.

Case 4: *p*'s right child has a left child

The most general case comes up when p's right child has a left child, where we replace p by its successor s. In that case s acquires p's former balance factor and rebalancing begins from s's parent r. Node s is always the left child of r.

§317 ⟨ Case 4 in TAVL deletion 317 ⟩ ≡
⟨ Case 4 in TBST deletion; tbst ⇒ tavl 263 ⟩

```
s->tavl_balance = p->tavl_balance;
q = r;
dir = 0;
```

This code is included in §313.

Exercises:

1. Rewrite ⟨ Case 4 in TAVL deletion 317 ⟩ to replace the deleted node's *tavl_data* by its successor, then delete the successor, instead of shuffling pointers. (Refer back to Exercise 4.8-3 for an explanation of why this approach cannot be used in Libavl.)

8.5.3 Step 3: Update Balance Factors

Rebalancing begins from node q, from whose side *dir* a node was deleted. Node q at the beginning of the iteration becomes node y, the root of the balance factor update and rebalancing, and *dir* at the beginning of the iteration is used to separate the left-side and right-side deletion cases.

The loop also updates the values of q and *dir* for rebalancing and for use in the next iteration of the loop, if any. These new values can only be assigned after the old ones are no longer needed, but must be assigned before any rebalancing so that the parent link to y

can be changed. For q this is after y receives q's old value and before rebalancing. For *dir*, it is after the branch point that separates the left-side and right-side deletion cases, so the *dir* assignment is duplicated in each branch. The code used to update q is discussed later.

§318 ⟨ Steps 3 and 4: Update balance factors and rebalance after TAVL deletion 318 ⟩ ≡

```
while (q != (struct tavl_node *) &tree→tavl_root) {
    struct tavl_node *y = q;
    q = find_parent (tree, y);
    if (dir == 0) {
        dir = q→tavl_link[0] != y;
        y→tavl_balance++;
        if (y→tavl_balance == +1)
            break;
        else if (y→tavl_balance == +2)
            { ⟨ Step 4: Rebalance after TAVL deletion 319 ⟩ }
    }
    else { ⟨ Steps 3 and 4: Symmetric case in TAVL deletion 323 ⟩ }
}
tree→tavl_count−−;
return (void *) item;
```

This code is included in §311.

8.5.4 Step 4: Rebalance

Rebalancing after deletion in a TAVL tree divides into three cases. The first of these is analogous to case 1 in unthreaded AVL deletion, the other two to case 2 (see Section 7.6 [Inserting into a TBST], page 167). The cases are distinguished, as usual, based on the balance factor of right child x of the node y at which rebalancing occurs:

§319 ⟨ Step 4: Rebalance after TAVL deletion 319 ⟩ ≡

```
struct tavl_node *x = y→tavl_link[1];
assert (x != NULL);
if (x→tavl_balance == −1) {
    ⟨ Rebalance for − balance factor after TAVL deletion in left subtree 320 ⟩
} else {
    q→tavl_link[dir] = x;
    if (x→tavl_balance == 0) {
        ⟨ Rebalance for 0 balance factor after TAVL deletion in left subtree 321 ⟩
        break;
    } else /* x→tavl_balance == +1 */ {
        ⟨ Rebalance for + balance factor after TAVL deletion in left subtree 322 ⟩
    }
}
```

This code is included in §318.

Case 1: x has − balance factor

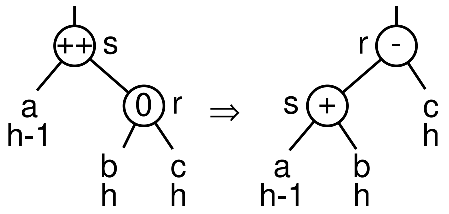
This case is just like case 2 in TAVL insertion. In fact, we can even reuse the code:

§320 ⟨ Rebalance for − balance factor after TAVL deletion in left subtree 320 ⟩ ≡
struct tavl_node $*w$;
⟨ Rebalance for − balance factor in TAVL insertion in right subtree 310 ⟩
$q \rightarrow tavl\_link[dir] = w$;

This code is included in §319.

Case 2: *x* has 0 balance factor

If x has a 0 balance factor, then we perform a left rotation at y. The transformation looks like this, with subtree heights listed under their labels:



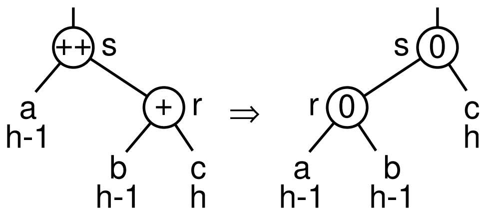
Subtree b is taller than subtree a, so even if h takes its minimum value of 1, then subtree b has height $h \equiv 1$ and, therefore, it must contain at least one node and there is no need to do any checking for threads. The code is simple:

§321 ⟨ Rebalance for 0 balance factor after TAVL deletion in left subtree 321 ⟩ ≡
$y \rightarrow tavl\_link[1] = x \rightarrow tavl\_link[0]$;
$x \rightarrow tavl\_link[0] = y$;
$x \rightarrow tavl\_balance = {}^{-}1$;
$y \rightarrow tavl\_balance = {}^{+}1$;

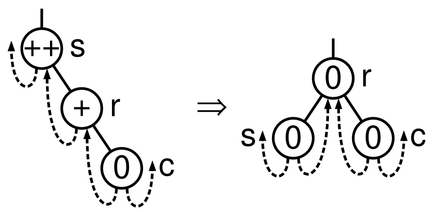
This code is included in §319 and §443.

Case 3: *x* has + balance factor

If x has a + balance factor, we perform a left rotation at y, same as for case 2, and the transformation looks like this:



One difference from case 2 is in the resulting balance factors. The other is that if $h \equiv 1$, then subtrees a and b have height $h - 1 \equiv 0$, so a and b may actually be threads. In that case, the transformation must be done this way:



This code handles both possibilities:

§322 ⟨ Rebalance for + balance factor after TAVL deletion in left subtree 322 ⟩ ≡

```
if (x→tavl_tag[0] == TAVL_CHILD)
      y→tavl_link[1] = x→tavl_link[0];
else {
      y→tavl_tag[1] = TAVL_THREAD;
      x→tavl_tag[0] = TAVL_CHILD;
}
x→tavl_link[0] = y;
y→tavl_balance = x→tavl_balance = 0;
```

This code is included in §319.

8.5.5 Symmetric Case

Here's the code for the symmetric case.

§323 ⟨ Steps 3 and 4: Symmetric case in TAVL deletion 323 ⟩ ≡

```
dir = q→tavl_link[0] != y;
y→tavl_balance−−;
if (y→tavl_balance == −1) break;
else if (y→tavl_balance == −2) {
      struct tavl_node *x = y→tavl_link[0];
      assert (x != NULL);
      if (x→tavl_balance == +1) {
            ⟨ Rebalance for + balance factor after TAVL deletion in right subtree 324 ⟩
      } else {
            q→tavl_link[dir] = x;
            if (x→tavl_balance == 0) {
                  ⟨ Rebalance for 0 balance factor after TAVL deletion in right subtree 325 ⟩
                  break;
            } else /* x→tavl_balance == −1 */ {
                  ⟨ Rebalance for − balance factor after TAVL deletion in right subtree 326 ⟩
            }
      }
}
```

This code is included in §318.

§324 ⟨ Rebalance for + balance factor after TAVL deletion in right subtree 324 ⟩ ≡

```
struct tavl_node *w;
⟨ Rebalance for + balance factor in TAVL insertion in left subtree 307 ⟩
q→tavl_link[dir] = w;
```

This code is included in §323.

§325 ⟨ Rebalance for 0 balance factor after TAVL deletion in right subtree 325 ⟩ ≡

```
y→tavl_link[0] = x→tavl_link[1];
x→tavl_link[1] = y;
x→tavl_balance = +1;
y→tavl_balance = −1;
```

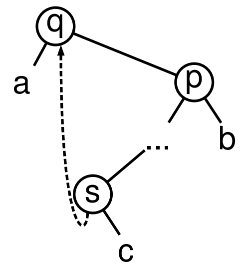
This code is included in §323 and §444.

§326 ⟨ Rebalance for − balance factor after TAVL deletion in right subtree 326 ⟩ ≡
```
if (x→tavl_tag[1] == TAVL_CHILD)
    y→tavl_link[0] = x→tavl_link[1];
else {
    y→tavl_tag[0] = TAVL_THREAD;
    x→tavl_tag[1] = TAVL_CHILD;
}
x→tavl_link[1] = y;
y→tavl_balance = x→tavl_balance = 0;
```
This code is included in §323.

8.5.6 Finding the Parent of a Node

The last component of *tavl_delete*() left undiscussed is the implementation of its helper function *find_parent*(), which requires an algorithm for finding the parent of an arbitrary node in a TAVL tree. If there were no efficient algorithm for this purpose, we would have to keep a stack of parent nodes as we did for unthreaded AVL trees. (This is still an option, as shown in Exercise 3.) We are fortunate that such an algorithm does exist. Let's discover it.

Because child pointers always lead downward in a BST, the only way that we're going to get from one node to another one above it is by following a thread. Almost directly from our definition of threads, we know that if a node q has a right child p, then there is a left thread in the subtree rooted at p that points back to q. Because a left thread points from a node to its predecessor, this left thread to q must come from q's successor, which we'll call s. The situation looks like this:



This leads immediately to an algorithm to find q given p, if p is q's right child. We simply follow left links starting at p until we we reach a thread, then we follow that thread. On the other hand, it doesn't help if p is q's left child, but there's an analogous situation with q's predecessor in that case.

Will this algorithm work for any node in a TBST? It won't work for the root node, because no thread points above the root (see Exercise 2). It will work for any other node, because any node other than the root has its successor or predecessor as its parent.

Here is the actual code, which finds and returns the parent of *node*. It traverses both the left and right subtrees of *node* at once, using x to move down to the left and y to move down to the right. When it hits a thread on one side, it checks whether it leads to *node*'s parent. If it does, then we're done. If it doesn't, then we continue traversing along the other side, which is guaranteed to lead to *node*'s parent.

§327 ⟨ Find parent of a TBST node 327 ⟩ ≡

```
/* Returns the parent of node within tree,
   or a pointer to tbst_root if s is the root of the tree. */
static struct tbst_node *find_parent (struct tbst_table *tree, struct tbst_node *node) {
    if (node != tree→tbst_root) {
        struct tbst_node *x, *y;

        for (x = y = node; ; x = x→tbst_link[0], y = y→tbst_link[1])
            if (y→tbst_tag[1] == TBST_THREAD) {
                struct tbst_node *p = y→tbst_link[1];
                if (p == NULL || p→tbst_link[0] != node) {
                    while (x→tbst_tag[0] == TBST_CHILD)
                        x = x→tbst_link[0];
                    p = x→tbst_link[0];
                }
                return p;
            }
            else if (x→tbst_tag[0] == TBST_THREAD) {
                struct tbst_node *p = x→tbst_link[0];
                if (p == NULL || p→tbst_link[1] != node) {
                    while (y→tbst_tag[1] == TBST_CHILD)
                        y = y→tbst_link[1];
                    p = y→tbst_link[1];
                }
                return p;
            }
    }
    else return (struct tbst_node *) &tree→tbst_root;
}
```

This code is included in §311, §668, and §670.

See also: [Knuth 1997], exercise 2.3.1-19.

Exercises:

***1.** Show that finding the parent of a given node using this algorithm, averaged over all the node within a TBST, requires only a constant number of links to be followed.

2. The structure of threads in our TBSTs force finding the parent of the root node to be special-cased. Suggest a modification to the tree structure to avoid this.

3. It can take several steps to find the parent of an arbitrary node in a TBST, even though the operation is "efficient" in the sense of Exercise 7.7-4. On the other hand, finding the parent of a node is very fast with a stack, but it costs time to construct the stack. Rewrite *tavl_delete*() to use a stack instead of the parent node algorithm.

8.6 Copying

We can use the tree copy function for TBSTs almost verbatim here. The one necessary change is that *copy_node*() must copy node balance factors. Here's the new version:

§328 ⟨ TAVL node copy function 328 ⟩ ≡

```
static int copy_node (struct tavl_table *tree, struct tavl_node *dst, int dir,
                      const struct tavl_node *src, tavl_copy_func *copy) {
  struct tavl_node *new = tree->tavl_alloc->libavl_malloc (tree->tavl_alloc, sizeof *new);
  if (new == NULL)
    return 0;

  new->tavl_link[dir] = dst->tavl_link[dir];
  new->tavl_tag[dir] = TAVL_THREAD;
  new->tavl_link[!dir] = dst;
  new->tavl_tag[!dir] = TAVL_THREAD;
  dst->tavl_link[dir] = new;
  dst->tavl_tag[dir] = TAVL_CHILD;

  new->tavl_balance = src->tavl_balance;
  if (copy == NULL)
    new->tavl_data = src->tavl_data;
  else {
    new->tavl_data = copy (src->tavl_data, tree->tavl_param);
    if (new->tavl_data == NULL)
      return 0;
  }

  return 1;
}
```

This code is included in §329.

§329 ⟨ TAVL copy function 329 ⟩ ≡

```
⟨ TAVL node copy function 328 ⟩
⟨ TBST copy error helper function; tbst ⇒ tavl 280 ⟩
⟨ TBST main copy function; tbst ⇒ tavl 279 ⟩
```

This code is included in §300 and §336.

8.7 Testing

The testing code harbors no surprises.

§330 ⟨ `tavl-test.c` 330 ⟩ ≡

```
⟨ License 1 ⟩
#include <assert.h>
#include <limits.h>
#include <stdio.h>
#include "tavl.h"
#include "test.h"

⟨ TBST print function; tbst ⇒ tavl 291 ⟩
⟨ BST traverser check function; bst ⇒ tavl 104 ⟩
⟨ Compare two TAVL trees for structure and content 331 ⟩
⟨ Recursively verify TAVL tree structure 332 ⟩
⟨ AVL tree verify function; avl ⇒ tavl 190 ⟩
⟨ BST test function; bst ⇒ tavl 100 ⟩
```

⟨ BST overflow test function; bst ⇒ tavl 122 ⟩

§331 ⟨ Compare two TAVL trees for structure and content 331 ⟩ ≡

```
static int compare_trees (struct tavl_node *a, struct tavl_node *b) {
  int okay;
  if (a == NULL || b == NULL) {
    if (a != NULL || b != NULL) {
      printf (" a=%d b=%d\n",
              a ? *(int *) a->tavl_data : -1, b ? *(int *) b->tavl_data : -1);
      assert (0);
    }
    return 1;
  }
  assert (a != b);
  if (*(int *) a->tavl_data != *(int *) b->tavl_data
      || a->tavl_tag[0] != b->tavl_tag[0] || a->tavl_tag[1] != b->tavl_tag[1]
      || a->tavl_balance != b->tavl_balance) {
    printf (" Copied nodes differ: a=%d (bal=%d) b=%d (bal=%d) a:",
            *(int *) a->tavl_data, a->tavl_balance,
            *(int *) b->tavl_data, b->tavl_balance);
    if (a->tavl_tag[0] == TAVL_CHILD) printf ("l");
    if (a->tavl_tag[1] == TAVL_CHILD) printf ("r");
    printf (" b:");
    if (b->tavl_tag[0] == TAVL_CHILD) printf ("l");
    if (b->tavl_tag[1] == TAVL_CHILD) printf ("r");
    printf ("\n");
    return 0;
  }
  if (a->tavl_tag[0] == TAVL_THREAD)
    assert ((a->tavl_link[0] == NULL) != (a->tavl_link[0] != b->tavl_link[0]));
  if (a->tavl_tag[1] == TAVL_THREAD)
    assert ((a->tavl_link[1] == NULL) != (a->tavl_link[1] != b->tavl_link[1]));
  okay = 1;
  if (a->tavl_tag[0] == TAVL_CHILD)
    okay &= compare_trees (a->tavl_link[0], b->tavl_link[0]);
  if (a->tavl_tag[1] == TAVL_CHILD)
    okay &= compare_trees (a->tavl_link[1], b->tavl_link[1]);
  return okay;
}
```

This code is included in §330.

§332 ⟨ Recursively verify TAVL tree structure 332 ⟩ ≡

```
static void recurse_verify_tree (struct tavl_node *node, int *okay, size_t *count,
                                 int min, int max, int *height) {
  int d; /* Value of this node's data. */
  size_t subcount[2]; /* Number of nodes in subtrees. */
  int subheight[2]; /* Heights of subtrees. */
```

```
    if (node == NULL) {
        *count = 0;
        *height = 0;
        return;
    }
    d = *(int *) node→tavl_data;
    ⟨ Verify binary search tree ordering 114 ⟩
    subcount[0] = subcount[1] = 0;
    subheight[0] = subheight[1] = 0;
    if (node→tavl_tag[0] == TAVL_CHILD)
        recurse_verify_tree (node→tavl_link[0], okay, &subcount[0],
                             min, d − 1, &subheight[0]);
    if (node→tavl_tag[1] == TAVL_CHILD)
        recurse_verify_tree (node→tavl_link[1], okay, &subcount[1],
                             d + 1, max, &subheight[1]);
    *count = 1 + subcount[0] + subcount[1];
    *height = 1 + (subheight[0] > subheight[1] ? subheight[0] : subheight[1]);
    ⟨ Verify AVL node balance factor; avl ⇒ tavl 189 ⟩
}
```

This code is included in §330.

9 Threaded Red-Black Trees

In the last two chapters, we introduced the idea of a threaded binary search tree, then applied that idea to AVL trees to produce threaded AVL trees. In this chapter, we will apply the idea of threading to red-black trees, resulting in threaded red-black or "TRB" trees.

Here's an outline of the table implementation for threaded RB trees, which use a *trb_* prefix.

§333 ⟨ trb.h 333 ⟩ ≡
```
⟨ License 1 ⟩
#ifndef TRB_H
#define TRB_H 1
#include ⟨ stddef.h ⟩
⟨ Table types; tbl ⇒ trb 14 ⟩
⟨ RB maximum height; rb ⇒ trb 195 ⟩
⟨ TBST table structure; tbst ⇒ trb 250 ⟩
⟨ TRB node structure 335 ⟩
⟨ TBST traverser structure; tbst ⇒ trb 267 ⟩
⟨ Table function prototypes; tbl ⇒ trb 15 ⟩
#endif /* trb.h */
```

§334 ⟨ trb.c 334 ⟩ ≡
```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "trb.h"
⟨ TRB functions 336 ⟩
```

9.1 Data Types

To make a RB tree node structure into a threaded RB tree node structure, we just add a pair of tag fields. We also reintroduce a maximum height definition here. It is not used by traversers, only by by the default versions of *trb_probe*() and *trb_delete*(), for maximum efficiency.

§335 ⟨ TRB node structure 335 ⟩ ≡
```
/* Color of a red-black node. */
enum trb_color {
    TRB_BLACK, /* Black. */
    TRB_RED /* Red. */
};
/* Characterizes a link as a child pointer or a thread. */
enum trb_tag {
    TRB_CHILD, /* Child pointer. */
    TRB_THREAD /* Thread. */
};
```

```
/* An TRB tree node. */
struct trb_node {
    struct trb_node *trb_link[2]; /* Subtrees. */
    void *trb_data; /* Pointer to data. */
    unsigned char trb_color; /* Color. */
    unsigned char trb_tag[2]; /* Tag fields. */
};
```

This code is included in §333.

9.2 Operations

Now we'll implement all the usual operations for TRB trees. Here's the outline. We can reuse everything from TBSTs except insertion, deletion, and copy functions. The copy function is implemented by reusing the version for TAVL trees, but copying colors instead of balance factors.

§336 ⟨ TRB functions 336 ⟩ ≡

```
⟨ TBST creation function; tbst ⇒ trb 252 ⟩
⟨ TBST search function; tbst ⇒ trb 253 ⟩
⟨ TRB item insertion function 337 ⟩
⟨ Table insertion convenience functions; tbl ⇒ trb 592 ⟩
⟨ TRB item deletion function 349 ⟩
⟨ TBST traversal functions; tbst ⇒ trb 268 ⟩
⟨ TAVL copy function; tavl ⇒ trb; tavl_balance ⇒ trb_color 329 ⟩
⟨ TBST destruction function; tbst ⇒ trb 281 ⟩
⟨ Default memory allocation functions; tbl ⇒ trb 6 ⟩
⟨ Table assertion functions; tbl ⇒ trb 594 ⟩
```

This code is included in §334.

9.3 Insertion

The structure of the insertion routine is predictable:

§337 ⟨ TRB item insertion function 337 ⟩ ≡

```
void **trb_probe (struct trb_table *tree, void *item) {
    struct trb_node *pa[TRB_MAX_HEIGHT]; /* Nodes on stack. */
    unsigned char da[TRB_MAX_HEIGHT]; /* Directions moved from stack nodes. */
    int k; /* Stack height. */
    struct trb_node *p; /* Traverses tree looking for insertion point. */
    struct trb_node *n; /* Newly inserted node. */
    int dir; /* Side of p on which n is inserted. */
    assert (tree != NULL && item != NULL);
    ⟨ Step 1: Search TRB tree for insertion point 338 ⟩
    ⟨ Step 2: Insert TRB node 339 ⟩
    ⟨ Step 3: Rebalance after TRB insertion 340 ⟩
    return &n->trb_data;
}
```

This code is included in §336.

9.3.1 Steps 1 and 2: Search and Insert

As usual, we search the tree from the root and record parents as we go.

§338 ⟨ Step 1: Search TRB tree for insertion point 338 ⟩ ≡
```
da[0] = 0;
pa[0] = (struct trb_node *) &tree→trb_root;
k = 1;
if (tree→trb_root != NULL) {
    for (p = tree→trb_root; ; p = p→trb_link[dir]) {
        int cmp = tree→trb_compare (item, p→trb_data, tree→trb_param);
        if (cmp == 0)
            return &p→trb_data;

        pa[k] = p;
        da[k++] = dir = cmp > 0;

        if (p→trb_tag[dir] == TRB_THREAD)
            break;
    }
} else {
    p = (struct trb_node *) &tree→trb_root;
    dir = 0;
}
```
This code is included in §337.

The code for insertion is included within the loop for easy access to the *dir* variable.

§339 ⟨ Step 2: Insert TRB node 339 ⟩ ≡
```
⟨ Step 2: Insert TBST node; tbst ⇒ trb 256 ⟩
n→trb_color = TRB_RED;
```
This code is included in §337 and §668.

9.3.2 Step 3: Rebalance

The basic rebalancing loop is unchanged from ⟨ Step 3: Rebalance after RB insertion 201 ⟩.

§340 ⟨ Step 3: Rebalance after TRB insertion 340 ⟩ ≡
```
while (k >= 3 && pa[k − 1]→trb_color == TRB_RED) {
    if (da[k − 2] == 0)
        { ⟨ Left-side rebalancing after TRB insertion 341 ⟩ }
    else { ⟨ Right-side rebalancing after TRB insertion 345 ⟩ }
}
tree→trb_root→trb_color = TRB_BLACK;
```
This code is included in §337.

The cases for rebalancing are the same as in ⟨ Left-side rebalancing after RB insertion 202 ⟩, too. We do need to check for threads, instead of null pointers.

§341 ⟨ Left-side rebalancing after TRB insertion 341 ⟩ ≡
```
struct trb_node *y = pa[k − 2]→trb_link[1];
if (pa[k − 2]→trb_tag[1] == TRB_CHILD && y→trb_color == TRB_RED)
```

```
    { ⟨Case 1 in left-side TRB insertion rebalancing 342⟩ }
else {
    struct trb_node *x;

    if (da[k - 1] == 0)
        y = pa[k - 1];
    else { ⟨Case 3 in left-side TRB insertion rebalancing 344⟩ }

    ⟨Case 2 in left-side TRB insertion rebalancing 343⟩
    break;
}
```

This code is included in §340.

The rest of this section deals with the individual rebalancing cases, the same as in unthreaded RB insertion (see Section 6.4.3 [Inserting an RB Node Step 3 - Rebalance], page 143). Each iteration deals with a node whose color has just been changed to red, which is the newly inserted node n in the first trip through the loop. In the discussion, we'll call this node q.

Case 1: *q*'s uncle is red

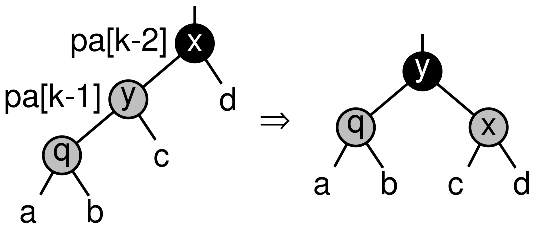
If node q has an red "uncle", then only recoloring is required. Because no links are changed, no threads need to be updated, and we can reuse the code for RB insertion without change:

§342 ⟨Case 1 in left-side TRB insertion rebalancing 342⟩ ≡
⟨Case 1 in left-side RB insertion rebalancing; rb ⇒ trb 203⟩

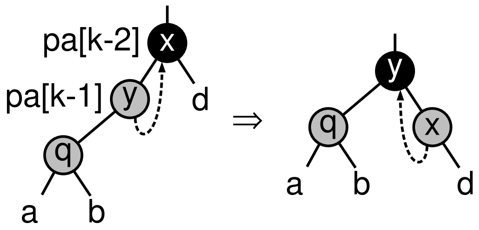
This code is included in §341.

Case 2: *q* is the left child of its parent

If q is the left child of its parent, we rotate right at q's grandparent, and recolor a few nodes. Here's the transformation:



This transformation can only cause thread problems with subtree c, since the other subtrees stay firmly in place. If c is a thread, then we need to make adjustments after the transformation to account for the difference between threaded and unthreaded rotation, so that the final operation looks like this:



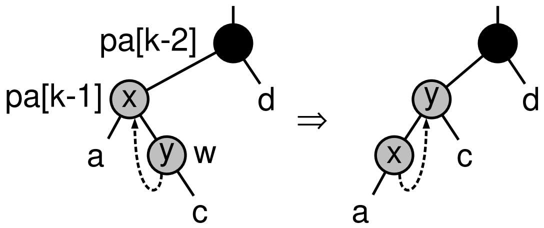
§343 ⟨ Case 2 in left-side TRB insertion rebalancing 343 ⟩ ≡
⟨ Case 2 in left-side RB insertion rebalancing; rb ⇒ trb 204 ⟩

```
if (y→trb_tag[1] == TRB_THREAD) {
    y→trb_tag[1] = TRB_CHILD;
    x→trb_tag[0] = TRB_THREAD;
    x→trb_link[0] = y;
}
```

This code is included in §341.

Case 3: *q* is the right child of its parent

The modification to case 3 is the same as the modification to case 2, but it applies to a left rotation instead of a right rotation. The adjusted case looks like this:



§344 ⟨ Case 3 in left-side TRB insertion rebalancing 344 ⟩ ≡
⟨ Case 3 in left-side RB insertion rebalancing; rb ⇒ trb 205 ⟩

```
if (y→trb_tag[0] == TRB_THREAD) {
    y→trb_tag[0] = TRB_CHILD;
    x→trb_tag[1] = TRB_THREAD;
    x→trb_link[1] = y;
}
```

This code is included in §341.

9.3.3 Symmetric Case

§345 ⟨ Right-side rebalancing after TRB insertion 345 ⟩ ≡

```
struct trb_node *y = pa[k − 2]→trb_link[0];
if (pa[k − 2]→trb_tag[0] == TRB_CHILD && y→trb_color == TRB_RED)
    { ⟨ Case 1 in right-side TRB insertion rebalancing 346 ⟩ }
else {
    struct trb_node *x;
    if (da[k − 1] == 1)
        y = pa[k − 1];
    else { ⟨ Case 3 in right-side TRB insertion rebalancing 348 ⟩ }
```

```
    ⟨ Case 2 in right-side TRB insertion rebalancing 347 ⟩
    break;
}
```

This code is included in §340.

§346 ⟨ Case 1 in right-side TRB insertion rebalancing 346 ⟩ ≡
⟨ Case 1 in right-side RB insertion rebalancing; rb ⇒ trb 207 ⟩

This code is included in §345.

§347 ⟨ Case 2 in right-side TRB insertion rebalancing 347 ⟩ ≡
⟨ Case 2 in right-side RB insertion rebalancing; rb ⇒ trb 208 ⟩

```
if (y->trb_tag[0] == TRB_THREAD) {
    y->trb_tag[0] = TRB_CHILD;
    x->trb_tag[1] = TRB_THREAD;
    x->trb_link[1] = y;
}
```

This code is included in §345.

§348 ⟨ Case 3 in right-side TRB insertion rebalancing 348 ⟩ ≡
⟨ Case 3 in right-side RB insertion rebalancing; rb ⇒ trb 209 ⟩

```
if (y->trb_tag[1] == TRB_THREAD) {
    y->trb_tag[1] = TRB_CHILD;
    x->trb_tag[0] = TRB_THREAD;
    x->trb_link[0] = y;
}
```

This code is included in §345.

Exercises:

1. It could be argued that the algorithm here is "impure" because it uses a stack, when elimination of the need for a stack is one of the reasons originally given for using threaded trees. Write a version of *trb_probe*() that avoids the use of a stack. You can use *find_parent*() from ⟨ Find parent of a TBST node 327 ⟩ as a substitute.

9.4 Deletion

The outline for the deletion function follows the usual pattern.

§349 ⟨ TRB item deletion function 349 ⟩ ≡

```
void *trb_delete (struct trb_table *tree, const void *item) {
    struct trb_node *pa[TRB_MAX_HEIGHT]; /* Nodes on stack. */
    unsigned char da[TRB_MAX_HEIGHT]; /* Directions moved from stack nodes. */
    int k = 0; /* Stack height. */

    struct trb_node *p;
    int cmp, dir;

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search TRB tree for item to delete 350 ⟩
    ⟨ Step 2: Delete item from TRB tree 351 ⟩
    ⟨ Step 3: Rebalance tree after TRB deletion 356 ⟩
```

```
    ⟨Step 4: Finish up after TRB deletion 362⟩
}
```

This code is included in §336.

9.4.1 Step 1: Search

There's nothing new or interesting in the search code.

§350 ⟨Step 1: Search TRB tree for item to delete 350⟩ ≡

```
if (tree→trb_root == NULL)
    return NULL;
p = (struct trb_node *) &tree→trb_root;
for (cmp = -1; cmp != 0; cmp = tree→trb_compare (item, p→trb_data, tree→trb_param)) {
    dir = cmp > 0;
    pa[k] = p;
    da[k++] = dir;
    if (p→trb_tag[dir] == TRB_THREAD)
        return NULL;
    p = p→trb_link[dir];
}
item = p→trb_data;
```

This code is included in §349 and §659.

9.4.2 Step 2: Delete

The code for node deletion is a combination of RB deletion (see Section 6.5.1 [Deleting an RB Node Step 2 - Delete], page 151) and TBST deletion (see Section 7.7 [Deleting from a TBST], page 168). The node to delete is p, and after deletion the stack contains all the nodes down to where rebalancing begins. The cases are the same as for TBST deletion:

§351 ⟨Step 2: Delete item from TRB tree 351⟩ ≡

```
if (p→trb_tag[1] == TRB_THREAD) {
    if (p→trb_tag[0] == TRB_CHILD)
        { ⟨Case 1 in TRB deletion 352⟩ }
    else { ⟨Case 2 in TRB deletion 353⟩ }
} else {
    enum trb_color t;
    struct trb_node *r = p→trb_link[1];
    if (r→trb_tag[0] == TRB_THREAD)
        { ⟨Case 3 in TRB deletion 354⟩ }
    else { ⟨Case 4 in TRB deletion 355⟩ }
}
```

This code is included in §349.

Case 1: *p* has a right thread and a left child

If the node to delete p has a right thread and a left child, then we replace it by its left child. We also have to chase down the right thread that pointed to p. The code is almost the same as ⟨Case 1 in TBST deletion 260⟩, but we use the stack here instead of a single parent pointer.

§352 ⟨Case 1 in TRB deletion 352⟩ ≡
```
struct trb_node *t = p->trb_link[0];
while (t->trb_tag[1] == TRB_CHILD)
    t = t->trb_link[1];
t->trb_link[1] = p->trb_link[1];
pa[k - 1]->trb_link[da[k - 1]] = p->trb_link[0];
```
This code is included in §351.

Case 2: *p* has a right thread and a left thread

Deleting a leaf node is the same process as for a TBST. The changes from ⟨Case 2 in TBST deletion 261⟩ are again due to the use of a stack.

§353 ⟨Case 2 in TRB deletion 353⟩ ≡
```
pa[k - 1]->trb_link[da[k - 1]] = p->trb_link[da[k - 1]];
if (pa[k - 1] != (struct trb_node *) &tree->trb_root)
    pa[k - 1]->trb_tag[da[k - 1]] = TRB_THREAD;
```
This code is included in §351.

Case 3: *p*'s right child has a left thread

The code for case 3 merges ⟨Case 3 in TBST deletion 262⟩ with ⟨Case 2 in RB deletion 223⟩. First, the node is deleted in the same way used for a TBST. Then the colors of p and r are swapped, and r is added to the stack, in the same way as for RB deletion.

§354 ⟨Case 3 in TRB deletion 354⟩ ≡
```
r->trb_link[0] = p->trb_link[0];
r->trb_tag[0] = p->trb_tag[0];
if (r->trb_tag[0] == TRB_CHILD) {
    struct trb_node *t = r->trb_link[0];
    while (t->trb_tag[1] == TRB_CHILD)
        t = t->trb_link[1];
    t->trb_link[1] = r;
}
pa[k - 1]->trb_link[da[k - 1]] = r;
t = r->trb_color;
r->trb_color = p->trb_color;
p->trb_color = t;
da[k] = 1;
pa[k++] = r;
```
This code is included in §351.

Case 4: *p*'s right child has a left child

Case 4 is a mix of ⟨Case 4 in TBST deletion 263⟩ and ⟨Case 3 in RB deletion 224⟩. It follows the outline of TBST deletion, but updates the stack. After the deletion it also swaps the colors of p and s as in RB deletion.

§355 ⟨ Case 4 in TRB deletion 355 ⟩ ≡

```
struct trb_node *s;
int j = k++;
for (;;) {
    da[k] = 0;
    pa[k++] = r;
    s = r->trb_link[0];
    if (s->trb_tag[0] == TRB_THREAD)
        break;
    r = s;
}
da[j] = 1;
pa[j] = s;
if (s->trb_tag[1] == TRB_CHILD)
    r->trb_link[0] = s->trb_link[1];
else {
    r->trb_link[0] = s;
    r->trb_tag[0] = TRB_THREAD;
}
s->trb_link[0] = p->trb_link[0];
if (p->trb_tag[0] == TRB_CHILD) {
    struct trb_node *t = p->trb_link[0];
    while (t->trb_tag[1] == TRB_CHILD)
        t = t->trb_link[1];
    t->trb_link[1] = s;
    s->trb_tag[0] = TRB_CHILD;
}
s->trb_link[1] = p->trb_link[1];
s->trb_tag[1] = TRB_CHILD;
t = s->trb_color;
s->trb_color = p->trb_color;
p->trb_color = t;
pa[j - 1]->trb_link[da[j - 1]] = s;
```

This code is included in §351.

Exercises:

1. Rewrite ⟨ Case 4 in TAVL deletion 317 ⟩ to replace the deleted node's *tavl_data* by its successor, then delete the successor, instead of shuffling pointers. (Refer back to Exercise 4.8-3 for an explanation of why this approach cannot be used in Libavl.)

9.4.3 Step 3: Rebalance

The outline for rebalancing after threaded RB deletion is the same as for the unthreaded case (see Section 6.5.2 [Deleting an RB Node Step 3 - Rebalance], page 154):

§356 ⟨ Step 3: Rebalance tree after TRB deletion 356 ⟩ ≡

```
if (p→trb_color == TRB_BLACK) {
    for (; k > 1; k−−) {
        if (pa[k − 1]→trb_tag[da[k − 1]] == TRB_CHILD) {
            struct trb_node *x = pa[k − 1]→trb_link[da[k − 1]];
            if (x→trb_color == TRB_RED) {
                x→trb_color = TRB_BLACK;
                break;
            }
        }
        if (da[k − 1] == 0)
            { ⟨Left-side rebalancing after TRB deletion 357⟩ }
        else { ⟨Right-side rebalancing after TRB deletion 363⟩ }
    }
    if (tree→trb_root != NULL)
        tree→trb_root→trb_color = TRB_BLACK;
}
```

This code is included in §349.

The rebalancing cases are the same, too. We need to check for thread tags, not for null pointers, though, in some places:

§357 ⟨Left-side rebalancing after TRB deletion 357⟩ ≡

```
struct trb_node *w = pa[k − 1]→trb_link[1];
if (w→trb_color == TRB_RED)
    { ⟨Ensure w is black in left-side TRB deletion rebalancing 358⟩ }
if ((w→trb_tag[0] == TRB_THREAD || w→trb_link[0]→trb_color == TRB_BLACK)
   && (w→trb_tag[1] == TRB_THREAD || w→trb_link[1]→trb_color == TRB_BLACK))
    { ⟨Case 1 in left-side TRB deletion rebalancing 359⟩ }
else {
    if (w→trb_tag[1] == TRB_THREAD || w→trb_link[1]→trb_color == TRB_BLACK)
        { ⟨Transform left-side TRB deletion rebalancing case 3 into case 2 361⟩ }
    ⟨Case 2 in left-side TRB deletion rebalancing 360⟩
    break;
}
```

This code is included in §356.

Case Reduction: Ensure *w* is black

This transformation does not move around any subtrees that might be threads, so there is no need for it to change.

§358 ⟨Ensure w is black in left-side TRB deletion rebalancing 358⟩ ≡
⟨Ensure w is black in left-side RB deletion rebalancing; rb ⇒ trb 228⟩

This code is included in §357.

Case 1: *w* has no red children

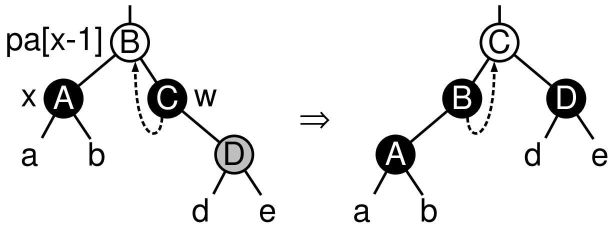
This transformation just recolors nodes, so it also does not need any changes.

§359 ⟨ Case 1 in left-side TRB deletion rebalancing 359 ⟩ ≡
⟨ Case 1 in left-side RB deletion rebalancing; rb ⇒ trb 229 ⟩

This code is included in §357.

Case 2: *w*'s right child is red

If w has a red right child and a left thread, then it is necessary to adjust tags and links after the left rotation at w and recoloring, as shown in this diagram:



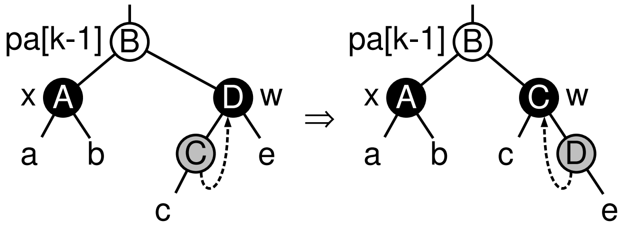
§360 ⟨ Case 2 in left-side TRB deletion rebalancing 360 ⟩ ≡
⟨ Case 2 in left-side RB deletion rebalancing; rb ⇒ trb 230 ⟩

```
if (w→trb_tag[0] == TRB_THREAD) {
    w→trb_tag[0] = TRB_CHILD;
    pa[k − 1]→trb_tag[1] = TRB_THREAD;
    pa[k − 1]→trb_link[1] = w;
}
```

This code is included in §357.

Case 3: *w*'s left child is red

If w has a red left child, which has a right thread, then we again need to adjust tags and links after right rotation at w and recoloring, as shown here:



§361 ⟨ Transform left-side TRB deletion rebalancing case 3 into case 2 361 ⟩ ≡
⟨ Transform left-side RB deletion rebalancing case 3 into case 2; rb ⇒ trb 231 ⟩

```
if (w→trb_tag[1] == TRB_THREAD) {
    w→trb_tag[1] = TRB_CHILD;
    w→trb_link[1]→trb_tag[0] = TRB_THREAD;
    w→trb_link[1]→trb_link[0] = w;
}
```

This code is included in §357.

9.4.4 Step 4: Finish Up

All that's left to do is free the node, update the count, and return the deleted item:

§362 ⟨ Step 4: Finish up after TRB deletion 362 ⟩ ≡
```
tree→trb_alloc→libavl_free (tree→trb_alloc, p);
tree→trb_count−−;
return (void *) item;
```
This code is included in §349.

9.4.5 Symmetric Case

§363 ⟨ Right-side rebalancing after TRB deletion 363 ⟩ ≡
```
struct trb_node *w = pa[k − 1]→trb_link[0];
if (w→trb_color == TRB_RED)
    { ⟨ Ensure w is black in right-side TRB deletion rebalancing 364 ⟩ }
if ((w→trb_tag[0] == TRB_THREAD || w→trb_link[0]→trb_color == TRB_BLACK)
  && (w→trb_tag[1] == TRB_THREAD || w→trb_link[1]→trb_color == TRB_BLACK))
    { ⟨ Case 1 in right-side TRB deletion rebalancing 365 ⟩ }
else {
    if (w→trb_tag[0] == TRB_THREAD || w→trb_link[0]→trb_color == TRB_BLACK)
        { ⟨ Transform right-side TRB deletion rebalancing case 3 into case 2 367 ⟩ }
    ⟨ Case 2 in right-side TRB deletion rebalancing 366 ⟩
    break;
}
```
This code is included in §356.

§364 ⟨ Ensure w is black in right-side TRB deletion rebalancing 364 ⟩ ≡
```
⟨ Ensure w is black in right-side RB deletion rebalancing; rb ⇒ trb 234 ⟩
```
This code is included in §363.

§365 ⟨ Case 1 in right-side TRB deletion rebalancing 365 ⟩ ≡
```
⟨ Case 1 in right-side RB deletion rebalancing; rb ⇒ trb 235 ⟩
```
This code is included in §363.

§366 ⟨ Case 2 in right-side TRB deletion rebalancing 366 ⟩ ≡
```
⟨ Case 2 in right-side RB deletion rebalancing; rb ⇒ trb 237 ⟩
if (w→trb_tag[1] == TRB_THREAD) {
    w→trb_tag[1] = TRB_CHILD;
    pa[k − 1]→trb_tag[0] = TRB_THREAD;
    pa[k − 1]→trb_link[0] = w;
}
```
This code is included in §363.

§367 ⟨ Transform right-side TRB deletion rebalancing case 3 into case 2 367 ⟩ ≡
```
⟨ Transform right-side RB deletion rebalancing case 3 into case 2; rb ⇒ trb 236 ⟩
if (w→trb_tag[0] == TRB_THREAD) {
    w→trb_tag[0] = TRB_CHILD;
    w→trb_link[0]→trb_tag[1] = TRB_THREAD;
```

```
    w→trb_link[0]→trb_link[1] = w;
}
```

This code is included in §363.

Exercises:

1. Write another version of *trb_delete*() that does not use a stack. You can use ⟨ Find parent of a TBST node 327 ⟩ to find the parent of a node.

9.5 Testing

The testing code harbors no surprises.

§368 ⟨ trb-test.c 368 ⟩ ≡

```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ limits.h ⟩
#include ⟨ stdio.h ⟩
#include "trb.h"
#include "test.h"
⟨ TBST print function; tbst ⇒ trb 291 ⟩
⟨ BST traverser check function; bst ⇒ trb 104 ⟩
⟨ Compare two TRB trees for structure and content 369 ⟩
⟨ Recursively verify TRB tree structure 370 ⟩
⟨ RB tree verify function; rb ⇒ trb 244 ⟩
⟨ BST test function; bst ⇒ trb 100 ⟩
⟨ BST overflow test function; bst ⇒ trb 122 ⟩
```

§369 ⟨ Compare two TRB trees for structure and content 369 ⟩ ≡

```
static int compare_trees (struct trb_node *a, struct trb_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
        if (a != NULL || b != NULL) {
            printf (" a=%d b=%d\n",
                    a ? *(int *) a→trb_data : -1, b ? *(int *) b→trb_data : -1);
            assert (0);
        }
        return 1;
    }
    assert (a != b);
    if (*(int *) a→trb_data != *(int *) b→trb_data
        || a→trb_tag[0] != b→trb_tag[0] || a→trb_tag[1] != b→trb_tag[1]
        || a→trb_color != b→trb_color) {
        printf (" Copied nodes differ: a=%d%c b=%d%c a:",
                *(int *) a→trb_data, a→trb_color == TRB_RED ? 'r' : 'b',
                *(int *) b→trb_data, b→trb_color == TRB_RED ? 'r' : 'b');
        if (a→trb_tag[0] == TRB_CHILD) printf ("l");
        if (a→trb_tag[1] == TRB_CHILD) printf ("r");
```

```
    printf (" b:");
    if (b->trb_tag[0] == TRB_CHILD) printf ("l");
    if (b->trb_tag[1] == TRB_CHILD) printf ("r");
    printf ("\n");
    return 0;
  }
  if (a->trb_tag[0] == TRB_THREAD)
    assert ((a->trb_link[0] == NULL) != (a->trb_link[0] != b->trb_link[0]));
  if (a->trb_tag[1] == TRB_THREAD)
    assert ((a->trb_link[1] == NULL) != (a->trb_link[1] != b->trb_link[1]));
  okay = 1;
  if (a->trb_tag[0] == TRB_CHILD)
    okay &= compare_trees (a->trb_link[0], b->trb_link[0]);
  if (a->trb_tag[1] == TRB_CHILD)
    okay &= compare_trees (a->trb_link[1], b->trb_link[1]);
  return okay;
}
```

This code is included in §368.

§370 ⟨ Recursively verify TRB tree structure 370 ⟩ ≡

```
static void recurse_verify_tree (struct trb_node *node, int *okay, size_t *count,
                                 int min, int max, int *bh) {
  int d; /* Value of this node's data. */
  size_t subcount[2]; /* Number of nodes in subtrees. */
  int subbh[2]; /* Black-heights of subtrees. */
  if (node == NULL) {
    *count = 0;
    *bh = 0;
    return;
  }
  d = *(int *) node->trb_data;
  ⟨ Verify binary search tree ordering 114 ⟩
  subcount[0] = subcount[1] = 0;
  subbh[0] = subbh[1] = 0;
  if (node->trb_tag[0] == TRB_CHILD)
    recurse_verify_tree (node->trb_link[0], okay, &subcount[0],
                         min, d - 1, &subbh[0]);
  if (node->trb_tag[1] == TRB_CHILD)
    recurse_verify_tree (node->trb_link[1], okay, &subcount[1],
                         d + 1, max, &subbh[1]);
  *count = 1 + subcount[0] + subcount[1];
  *bh = (node->trb_color == TRB_BLACK) + subbh[0];
  ⟨ Verify RB node color; rb ⇒ trb 241 ⟩
  ⟨ Verify TRB node rule 1 compliance 371 ⟩
  ⟨ Verify RB node rule 2 compliance; rb ⇒ trb 243 ⟩
}
```

This code is included in §368.

§371 ⟨ Verify TRB node rule 1 compliance 371 ⟩ ≡

```
/* Verify compliance with rule 1. */
if (node->trb_color == TRB_RED) {
    if (node->trb_tag[0] == TRB_CHILD && node->trb_link[0]->trb_color == TRB_RED) {
        printf (" Red node %d has red left child %d\n",
                d, *(int *) node->trb_link[0]->trb_data);
        *okay = 0;
    }
    if (node->trb_tag[1] == TRB_CHILD && node->trb_link[1]->trb_color == TRB_RED) {
        printf (" Red node %d has red right child %d\n",
                d, *(int *) node->trb_link[1]->trb_data);
        *okay = 0;
    }
}
```

This code is included in §370.

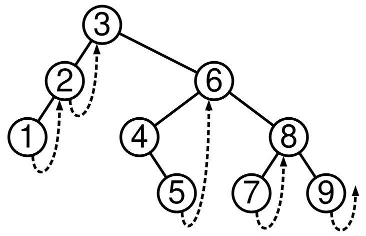
10 Right-Threaded Binary Search Trees

We originally introduced threaded trees to allow for traversal without maintaining a stack explicitly. This worked out well, so we implemented tables using threaded BSTs and AVL and RB trees. However, maintaining the threads can take some time. It would be nice if we could have the advantages of threads without so much of the overhead.

In one common special case, we can. Threaded trees are symmetric: there are left threads for moving to node predecessors and right threads for move to node successors. But traversals are not symmetric: many algorithms that traverse table entries only from least to greatest, never backing up. This suggests a matching asymmetric tree structure that has only right threads.

We can do this. In this chapter, we will develop a table implementation for a new kind of binary tree, called a right-threaded binary search tree, **right-threaded tree**, or simply "RTBST", that has threads only on the right side of nodes. Construction and modification of such trees can be faster and simpler than threaded trees because there is no need to maintain the left threads.

There isn't anything fundamentally new here, but just for completeness, here's an example of a right-threaded tree:



Keep in mind that although it is not efficient, it is still possible to traverse a right-threaded tree in order from greatest to least.[1] If it were not possible at all, then we could not build a complete table implementation based on right-threaded trees, because the definition of a table includes the ability to traverse it in either direction (see Section 2.10.2 [Manipulators], page 16).

Here's the outline of the RTBST code, which uses the prefix *rtbst_*:

§372 ⟨ `rtbst.h` 372 ⟩ ≡
⟨ License 1 ⟩
#ifndef `RTBST_H`
#define `RTBST_H` 1
#include ⟨ stddef.h ⟩
⟨ Table types; tbl ⇒ rtbst 14 ⟩
⟨ TBST table structure; tbst ⇒ rtbst 250 ⟩
⟨ RTBST node structure 374 ⟩
⟨ TBST traverser structure; tbst ⇒ rtbst 267 ⟩
⟨ Table function prototypes; tbl ⇒ rtbst 15 ⟩
⟨ BST extra function prototypes; bst ⇒ rtbst 88 ⟩

---

[1] It can be efficient if we use a stack to do it, but that kills the advantage of threading the tree. It would be possible to implement two sets of traversers for right-threaded trees, one with a stack, one without, but in that case it's probably better to just use a threaded tree.

```
#endif /* rtbst.h */
```

§373 ⟨ `rtbst.c` 373 ⟩ ≡

```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "rtbst.h"
⟨ RTBST functions 375 ⟩
```

See also: [Knuth 1997], section 2.3.1.

Exercises:

1. We can define a **left-threaded tree** in a way analogous to a right-threaded tree, as a binary search tree with threads only on the left sides of nodes. Is this a useful thing to do?

10.1 Data Types

§374 ⟨ RTBST node structure 374 ⟩ ≡

```
/* Characterizes a link as a child pointer or a thread. */
enum rtbst_tag {
    RTBST_CHILD, /* Child pointer. */
    RTBST_THREAD /* Thread. */
};

/* A threaded binary search tree node. */
struct rtbst_node {
    struct rtbst_node *rtbst_link[2]; /* Subtrees. */
    void *rtbst_data; /* Pointer to data. */
    unsigned char rtbst_rtag; /* Tag field. */
};
```

This code is included in §372.

10.2 Operations

§375 ⟨ RTBST functions 375 ⟩ ≡

```
⟨ TBST creation function; tbst ⇒ rtbst 252 ⟩
⟨ RTBST search function 376 ⟩
⟨ RTBST item insertion function 377 ⟩
⟨ Table insertion convenience functions; tbl ⇒ rtbst 592 ⟩
⟨ RTBST item deletion function 380 ⟩
⟨ RTBST traversal functions 395 ⟩
⟨ RTBST copy function 406 ⟩
⟨ RTBST destruction function 407 ⟩
⟨ RTBST balance function 408 ⟩
⟨ Default memory allocation functions; tbl ⇒ rtbst 6 ⟩
⟨ Table assertion functions; tbl ⇒ rtbst 594 ⟩
```

This code is included in §373.

10.3 Search

A right-threaded tree is inherently asymmetric, so many of the algorithms on it will necessarily be asymmetric as well. The search function is the simplest demonstration of this. For descent to the left, we test for a null left child with *rtbst_link*[0]; for descent to the right, we test for a right thread with *rtbst_rtag*. Otherwise, the code is familiar:

§376 ⟨ RTBST search function 376 ⟩ ≡
```
void *rtbst_find (const struct rtbst_table *tree, const void *item) {
    const struct rtbst_node *p;
    int dir;
    assert (tree != NULL && item != NULL);
    if (tree→rtbst_root == NULL)
        return NULL;
    for (p = tree→rtbst_root; ; p = p→rtbst_link[dir]) {
        int cmp = tree→rtbst_compare (item, p→rtbst_data, tree→rtbst_param);
        if (cmp == 0)
            return p→rtbst_data;
        dir = cmp > 0;
        if (dir == 0) {
            if (p→rtbst_link[0] == NULL)
                return NULL;
        } else /* dir == 1 */ {
            if (p→rtbst_rtag == RTBST_THREAD)
                return NULL;
        }
    }
}
```
This code is included in §375, §418, and §455.

10.4 Insertion

Regardless of the kind of binary tree we're dealing with, adding a new node requires setting three pointer fields: the parent pointer and the two child pointers of the new node. On the other hand, we do save a tiny bit on tags: we set either 1 or 2 tags here as opposed to a constant of 3 in ⟨ TBST item insertion function 254 ⟩.

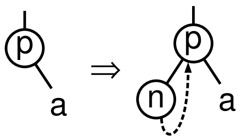
Here is the outline:

§377 ⟨ RTBST item insertion function 377 ⟩ ≡
```
void **rtbst_probe (struct rtbst_table *tree, void *item) {
    struct rtbst_node *p; /* Current node in search. */
    int dir; /* Side of p on which to insert the new node. */
    struct rtbst_node *n; /* New node. */
    ⟨ Step 1: Search RTBST for insertion point 378 ⟩
    ⟨ Step 2: Insert new node into RTBST tree 379 ⟩
}
```
This code is included in §375.

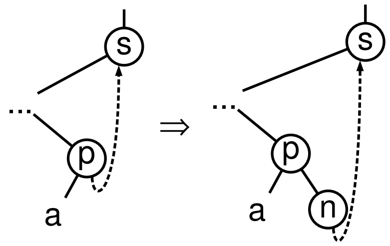
The code to search for the insertion point is not unusual:

§378 ⟨ Step 1: Search RTBST for insertion point 378 ⟩ ≡
```
if (tree→rtbst_root != NULL)
  for (p = tree→rtbst_root; ; p = p→rtbst_link[dir]) {
    int cmp = tree→rtbst_compare (item, p→rtbst_data, tree→rtbst_param);
    if (cmp == 0)
      return &p→rtbst_data;
    dir = cmp > 0;

    if (dir == 0) {
      if (p→rtbst_link[0] == NULL)
        break;
    } else /* dir == 1 */ {
      if (p→rtbst_rtag == RTBST_THREAD)
        break;
    }
  }
else {
  p = (struct rtbst_node *) &tree→rtbst_root;
  dir = 0;
}
```
This code is included in §377.

Now for the insertion code. An insertion to the left of a node p in a right-threaded tree replaces the left link by the new node n. The new node in turn has a null left child and a right thread pointing back to p:



An insertion to the right of p replaces the right thread by the new child node n. The new node has a null left child and a right thread that points where p's right thread formerly pointed:



We can handle both of these cases in one code segment. The difference is in the treatment of n's right child and p's right tag. Insertion into an empty tree is handled as a special case as well:

§379 ⟨ Step 2: Insert new node into RTBST tree 379 ⟩ ≡
```
n = tree→rtbst_alloc→libavl_malloc (tree→rtbst_alloc, sizeof *n);
if (n == NULL)
```

```
    return NULL;
tree→rtbst_count++;
n→rtbst_data = item;
n→rtbst_link[0] = NULL;
if (dir == 0) {
    if (tree→rtbst_root != NULL)
        n→rtbst_link[1] = p;
    else n→rtbst_link[1] = NULL;
} else /* dir == 1 */ {
    p→rtbst_rtag = RTBST_CHILD;
    n→rtbst_link[1] = p→rtbst_link[1];
}
n→rtbst_rtag = RTBST_THREAD;
p→rtbst_link[dir] = n;
return &n→rtbst_data;
```

This code is included in §377.

10.5 Deletion

Deleting a node from an RTBST can be done using the same ideas as for other kinds of trees we've seen. However, as it turns out, a variant of this usual technique allows for faster code. In this section, we will implement the usual method, then the improved version. The latter is actually used in Libavl.

Here is the outline of the function. Step 2 is the only part that varies between versions:

§380 ⟨ RTBST item deletion function 380 ⟩ ≡

```
void *rtbst_delete (struct rtbst_table *tree, const void *item) {
    struct rtbst_node *p; /* Node to delete. */
    struct rtbst_node *q; /* Parent of p. */
    int dir; /* Index into q→rtbst_link[] that leads to p. */
    assert (tree != NULL && item != NULL);
    ⟨ Step 1: Find RTBST node to delete 381 ⟩
    ⟨ Step 2: Delete RTBST node, left-looking 388 ⟩
    ⟨ Step 3: Finish up after deleting RTBST node 382 ⟩
}
```

This code is included in §375.

The first step just finds the node to delete. After it executes, p is the node to delete and q and *dir* are set such that q→*rtbst_link*[*dir*] == p.

§381 ⟨ Step 1: Find RTBST node to delete 381 ⟩ ≡

```
if (tree→rtbst_root == NULL)
    return NULL;
p = tree→rtbst_root;
q = (struct rtbst_node *) &tree→rtbst_root;
dir = 0;
if (p == NULL)
```

```
        return NULL;
    for (;;) {
        int cmp = tree→rtbst_compare (item, p→rtbst_data, tree→rtbst_param);
        if (cmp == 0)
            break;

        dir = cmp > 0;
        if (dir == 0) {
            if (p→rtbst_link[0] == NULL)
                return NULL;
        } else /* dir == 1 */ {
            if (p→rtbst_rtag == RTBST_THREAD)
                return NULL;
        }

        q = p;
        p = p→rtbst_link[dir];
    }
    item = p→rtbst_data;
```

This code is included in §380.

The final step is also common. We just clean up and return:

§382 ⟨ Step 3: Finish up after deleting RTBST node 382 ⟩ ≡

```
tree→rtbst_alloc→libavl_free (tree→rtbst_alloc, p);
tree→rtbst_count−−;
return (void *) item;
```

This code is included in §380.

10.5.1 Right-Looking Deletion

Our usual algorithm for deletion looks at the right subtree of the node to be deleted, so we call it "right-looking." The outline for this kind of deletion is the same as in TBST deletion (see Section 7.7 [Deleting from a TBST], page 168):

§383 ⟨ Step 2: Delete RTBST node, right-looking 383 ⟩ ≡

```
if (p→rtbst_rtag == RTBST_THREAD) {
    if (p→rtbst_link[0] != NULL)
        { ⟨ Case 1 in right-looking RTBST deletion 384 ⟩ }
    else { ⟨ Case 2 in right-looking RTBST deletion 385 ⟩ }
} else {
    struct rtbst_node *r = p→rtbst_link[1];
    if (r→rtbst_link[0] == NULL)
        { ⟨ Case 3 in right-looking RTBST deletion 386 ⟩ }
    else { ⟨ Case 4 in right-looking RTBST deletion 387 ⟩ }
}
```

Each of the four cases, presented below, is closely analogous to the same case in TBST deletion.

Case 1: *p* has a right thread and a left child

In this case, node p has a right thread and a left child. As in a TBST, this means that after deleting p we must update the right thread in p's former left subtree to point to p's replacement. The only difference from ⟨ Case 1 in TBST deletion 260 ⟩ is in structure members:

§384 ⟨ Case 1 in right-looking RTBST deletion 384 ⟩ ≡
```
struct rtbst_node *t = p→rtbst_link[0];
while (t→rtbst_rtag == RTBST_CHILD)
    t = t→rtbst_link[1];
t→rtbst_link[1] = p→rtbst_link[1];
q→rtbst_link[dir] = p→rtbst_link[0];
```
This code is included in §383.

Case 2: *p* has a right thread and no left child

If node p is a leaf, then there are two subcases, according to whether p is a left child or a right child of its parent q. If *dir* is 0, then p is a left child and the pointer from its parent must be set to NULL. If *dir* is 1, then p is a right child and the link from its parent must be changed to a thread to its successor.

In either of these cases we must set $q\rightarrow rtbst\_link[dir]$: if *dir* is 0, we set it to NULL, otherwise *dir* is 1 and we set it to $p\rightarrow rtbst\_link[1]$. However, we know that $p\rightarrow rtbst\_link[0]$ is NULL, because p is a leaf, so we can instead unconditionally assign $p\rightarrow rtbst\_link[dir]$. In addition, if *dir* is 1, then we must tag q's right link as a thread.

If q is the pseudo-root, then *dir* is 0 and everything works out fine with no need for a special case.

§385 ⟨ Case 2 in right-looking RTBST deletion 385 ⟩ ≡
```
q→rtbst_link[dir] = p→rtbst_link[dir];
if (dir == 1)
    q→rtbst_rtag = RTBST_THREAD;
```
This code is included in §383.

Case 3: *p*'s right child has no left child

Code for this case, where p has a right child r that itself has no left child, is almost identical to ⟨ Case 3 in TBST deletion 262 ⟩. There is no left tag to copy, but it is still necessary to chase down the right thread in r's new left subtree (the same as p's former left subtree):

§386 ⟨ Case 3 in right-looking RTBST deletion 386 ⟩ ≡
```
r→rtbst_link[0] = p→rtbst_link[0];
if (r→rtbst_link[0] != NULL) {
    struct rtbst_node *t = r→rtbst_link[0];
    while (t→rtbst_rtag == RTBST_CHILD)
        t = t→rtbst_link[1];
    t→rtbst_link[1] = r;
}
```

$q \rightarrow rtbst\_link[dir] = r;$

This code is included in §383.

Case 4: *p*'s right child has a left child

Code for case 4, the most general case, is very similar to ⟨ Case 4 in TBST deletion 263 ⟩. The only notable difference is in the subcase where s has a right thread: in that case we just set r's left link to NULL instead of having to set it up as a thread.

§387 ⟨ Case 4 in right-looking RTBST deletion 387 ⟩ ≡

```
struct rtbst_node *s;
for (;;) {
    s = r->rtbst_link[0];
    if (s->rtbst_link[0] == NULL)
        break;
    r = s;
}
if (s->rtbst_rtag == RTBST_CHILD)
    r->rtbst_link[0] = s->rtbst_link[1];
else r->rtbst_link[0] = NULL;
s->rtbst_link[0] = p->rtbst_link[0];
if (p->rtbst_link[0] != NULL) {
    struct rtbst_node *t = p->rtbst_link[0];
    while (t->rtbst_rtag == RTBST_CHILD)
        t = t->rtbst_link[1];
    t->rtbst_link[1] = s;
}
s->rtbst_link[1] = p->rtbst_link[1];
s->rtbst_rtag = RTBST_CHILD;
q->rtbst_link[dir] = s;
```

This code is included in §383.

Exercises:

1. Rewrite ⟨ Case 4 in right-looking RTBST deletion 387 ⟩ to replace the deleted node's *rtavl_data* by its successor, then delete the successor, instead of shuffling pointers. (Refer back to Exercise 4.8-3 for an explanation of why this approach cannot be used in Libavl.)

10.5.2 Left-Looking Deletion

The previous section implemented the "right-looking" form of deletion used elsewhere in Libavl. Compared to deletion in a fully threaded binary tree, the benefits to using an RTBST with this kind of deletion are minimal:

- Cases 1 and 2 are similar code in both TBST and RTBST deletion.
- Case 3 in an RTBST avoids one tag copy required in TBST deletion.
- One subcase of case 4 in an RTBST avoids one tag assignment required in the same subcase of TBST deletion.

This is hardly worth it. We saved at most one assignment per call. We need something better if it's ever going to be worthwhile to use right-threaded trees.

Fortunately, there is a way that we can save a little more. This is by changing our right-looking deletion into left-looking deletion, by switching the use of left and right children in the algorithm. In a BST or TBST, this symmetrical change in the algorithm would have no effect, because the BST and TBST node structures are themselves symmetric. But in an asymmetric RTBST even a symmetric change can have a significant effect on an algorithm, as we'll see.

The cases for left-looking deletion are outlined in the same way as for right-looking deletion:

§388 ⟨ Step 2: Delete RTBST node, left-looking 388 ⟩ ≡

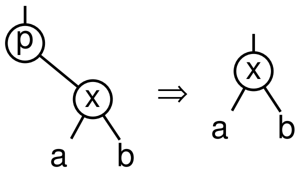
```
if (p→rtbst_link[0] == NULL) {
    if (p→rtbst_rtag == RTBST_CHILD)
        { ⟨ Case 1 in left-looking RTBST deletion 389 ⟩ }
    else { ⟨ Case 2 in left-looking RTBST deletion 390 ⟩ }
} else {
    struct rtbst_node *r = p→rtbst_link[0];
    if (r→rtbst_rtag == RTBST_THREAD)
        { ⟨ Case 3 in left-looking RTBST deletion 391 ⟩ }
    else { ⟨ Case 4 in left-looking RTBST deletion 392 ⟩ }
}
```

This code is included in §380.

Case 1: *p* has a right child but no left child

If the node to delete p has a right child but no left child, we can just replace it by its right child. There is no right thread to update in p's left subtree because p has no left child, and there is no left thread to update because a right-threaded tree has no left threads.

The deletion looks like this if p's right child is designated x:



§389 ⟨ Case 1 in left-looking RTBST deletion 389 ⟩ ≡

```
q→rtbst_link[dir] = p→rtbst_link[1];
```

This code is included in §388.

Case 2: *p* has a right thread and no left child

This case is analogous to case 2 in right-looking deletion covered earlier. The same discussion applies.

§390 ⟨ Case 2 in left-looking RTBST deletion 390 ⟩ ≡

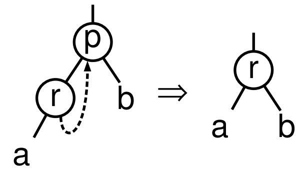
```
q→rtbst_link[dir] = p→rtbst_link[dir];
if (dir == 1)
```

```
q->rtbst_rtag = RTBST_THREAD;
```

This code is included in §388.

Case 3: *p*'s left child has a right thread

If p has a left child r that itself has a right thread, then we replace p by r. Node r receives p's former right link, as shown here:



There is no need to fiddle with threads. If r has a right thread then it gets replaced by p's right child or thread anyhow. Any right thread within r's left subtree either points within that subtree or to r. Finally, r's right subtree cannot cause problems.

§391 ⟨ Case 3 in left-looking RTBST deletion 391 ⟩ ≡

```
r->rtbst_link[1] = p->rtbst_link[1];
r->rtbst_rtag = p->rtbst_rtag;
q->rtbst_link[dir] = r;
```

This code is included in §388.

Case 4: *p*'s left child has a right child

The final case handles deletion of a node p with a left child r that in turn has a right child. The code here follows the same pattern as ⟨ Case 4 in TBST deletion 263 ⟩ (see the discussion there for details). The first step is to find the predecessor s of node p:

§392 ⟨ Case 4 in left-looking RTBST deletion 392 ⟩ ≡

```
struct rtbst_node *s;
for (;;) {
    s = r->rtbst_link[1];
    if (s->rtbst_rtag == RTBST_THREAD)
        break;
    r = s;
}
```

See also §393 and §394.
This code is included in §388.

Next, we update r, handling two subcases depending on whether s has a left child:

§393 ⟨ Case 4 in left-looking RTBST deletion 392 ⟩ +≡

```
if (s->rtbst_link[0] != NULL)
    r->rtbst_link[1] = s->rtbst_link[0];
else {
    r->rtbst_link[1] = s;
    r->rtbst_rtag = RTBST_THREAD;
}
```

The final step is to copy p's fields into s, then set q's child pointer to point to s instead of p. There is no need to chase down any threads.

§394 ⟨ Case 4 in left-looking RTBST deletion 392 ⟩ +≡
$s{\rightarrow}rtbst\_link[0] = p{\rightarrow}rtbst\_link[0];$
$s{\rightarrow}rtbst\_link[1] = p{\rightarrow}rtbst\_link[1];$
$s{\rightarrow}rtbst\_rtag = p{\rightarrow}rtbst\_rtag;$
$q{\rightarrow}rtbst\_link[dir] = s;$

Exercises:

1. Rewrite ⟨ Case 4 in left-looking RTBST deletion 392 ⟩ to replace the deleted node's *rtavl_data* by its predecessor, then delete the predecessor, instead of shuffling pointers. (Refer back to Exercise 4.8-3 for an explanation of why this approach cannot be used in Libavl.)

10.5.3 Aside: Comparison of Deletion Algorithms

This book has presented algorithms for deletion from BSTs, TBSTs, and RTBSTs. In fact, we implemented two algorithms for RTBSTs. Each of these four algorithms has slightly different performance characteristics. The following table summarizes the behavior of all of the cases in these algorithms. Each cell describes the actions that take place: "link" is the number of link fields set, "tag" the number of tag fields set, and "succ/pred" the number of general successor or predecessors found during the case.

	BST*	TBST	Right-Looking TBST	Left-Looking TBST
Case 1	1 link	2 links 1 succ/pred	2 links 1 succ/pred	1 link
Case 2	1 link	1 link 1 tag	1 link 1 tag	1 link 1 tag
Case 3	2 links	3 links 1 tag 1 succ/pred	3 links 1 succ/pred	2 links 1 tag
Case 4 subcase 1	4 links 1 succ/pred	5 links 2 tags 2 succ/pred	5 links 1 tag 2 succ/pred	4 links 1 tag 1 succ/pred
Case 4 subcase 2	4 links 1 succ/pred	5 links 2 tags 2 succ/pred	5 links 1 tag 2 succ/pred	4 links 1 tag 1 succ/pred

* Listed cases 1 and 2 both correspond to BST deletion case 1, and listed cases 3 and 4 to BST deletion cases 2 and 3, respectively. BST deletion does not have any subcases in its case 3 (listed case 4), so it also saves a test to distinguish subcases.

As you can see, the penalty for left-looking deletion from a RTBST, compared to a plain BST, is at most one tag assignment in any given case, except for the need to distinguish subcases of case 4. In this sense at least, left-looking deletion from an RTBST is considerably faster than deletion from a TBST or right-looking deletion from a RTBST. This means that it can indeed be worthwhile to implement right-threaded trees instead of BSTs or TBSTs.

10.6 Traversal

Traversal in an RTBST is unusual due to its asymmetry. Moving from smaller nodes to larger nodes is easy: we do it with the same algorithm used in a TBST. Moving the other way is more difficult and inefficient besides: we have neither a stack of parent nodes to fall back on nor left threads to short-circuit.

RTBSTs use the same traversal structure as TBSTs, so we can reuse some of the functions from TBST traversers. We also get a few directly from the implementations for BSTs. Other than that, everything has to be written anew here:

§395 ⟨ RTBST traversal functions 395 ⟩ ≡
⟨ TBST traverser null initializer; tbst ⇒ rtbst 269 ⟩
⟨ RTBST traverser first initializer 396 ⟩
⟨ RTBST traverser last initializer 397 ⟩
⟨ RTBST traverser search initializer 398 ⟩
⟨ TBST traverser insertion initializer; tbst ⇒ rtbst 273 ⟩
⟨ TBST traverser copy initializer; tbst ⇒ rtbst 274 ⟩
⟨ RTBST traverser advance function 399 ⟩
⟨ RTBST traverser back up function 400 ⟩
⟨ BST traverser current item function; bst ⇒ rtbst 74 ⟩
⟨ BST traverser replacement function; bst ⇒ rtbst 75 ⟩

This code is included in §375, §418, and §455.

10.6.1 Starting at the First Node

To find the first (least) item in the tree, we just descend all the way to the left, as usual. In an RTBST, as in a BST, this involves checking for null pointers.

§396 ⟨ RTBST traverser first initializer 396 ⟩ ≡
```
void *rtbst_t_first (struct rtbst_traverser *trav, struct rtbst_table *tree) {
    assert (tree != NULL && trav != NULL);

    trav->rtbst_table = tree;
    trav->rtbst_node = tree->rtbst_root;
    if (trav->rtbst_node != NULL) {
        while (trav->rtbst_node->rtbst_link[0] != NULL)
            trav->rtbst_node = trav->rtbst_node->rtbst_link[0];
        return trav->rtbst_node->rtbst_data;
    }
    else return NULL;
}
```

This code is included in §395.

10.6.2 Starting at the Last Node

To start at the last (greatest) item in the tree, we descend all the way to the right. In an RTBST, as in a TBST, this involves checking for thread links.

§397 ⟨ RTBST traverser last initializer 397 ⟩ ≡

```
void *rtbst_t_last (struct rtbst_traverser *trav, struct rtbst_table *tree) {
  assert (tree != NULL && trav != NULL);

  trav->rtbst_table = tree;
  trav->rtbst_node = tree->rtbst_root;
  if (trav->rtbst_node != NULL) {
    while (trav->rtbst_node->rtbst_rtag == RTBST_CHILD)
      trav->rtbst_node = trav->rtbst_node->rtbst_link[1];
    return trav->rtbst_node->rtbst_data;
  }
  else return NULL;
}
```

This code is included in §395.

10.6.3 Starting at a Found Node

To start from an item found in the tree, we use the same algorithm as *rtbst_find*().

§398 ⟨ RTBST traverser search initializer 398 ⟩ ≡

```
void *rtbst_t_find (struct rtbst_traverser *trav, struct rtbst_table *tree, void *item) {
  struct rtbst_node *p;

  assert (trav != NULL && tree != NULL && item != NULL);

  trav->rtbst_table = tree;
  trav->rtbst_node = NULL;

  p = tree->rtbst_root;
  if (p == NULL)
    return NULL;

  for (;;) {
    int cmp = tree->rtbst_compare (item, p->rtbst_data, tree->rtbst_param);
    if (cmp == 0) {
      trav->rtbst_node = p;
      return p->rtbst_data;
    }

    if (cmp < 0) {
      p = p->rtbst_link[0];
      if (p == NULL)
        return NULL;
    } else {
      if (p->rtbst_rtag == RTBST_THREAD)
        return NULL;
      p = p->rtbst_link[1];
    }
```

```
    }
}
```

This code is included in §395.

10.6.4 Advancing to the Next Node

We use the same algorithm to advance an RTBST traverser as for TBST traversers. The only important difference between this code and ⟨ TBST traverser advance function 275 ⟩ is the substitution of *rtbst_rtag* for *tbst_tag*[1].

§399 ⟨ RTBST traverser advance function 399 ⟩ ≡

```
void *rtbst_t_next (struct rtbst_traverser *trav) {
    assert (trav != NULL);
    if (trav→rtbst_node == NULL)
        return rtbst_t_first (trav, trav→rtbst_table);
    else if (trav→rtbst_node→rtbst_rtag == RTBST_THREAD) {
        trav→rtbst_node = trav→rtbst_node→rtbst_link[1];
        return trav→rtbst_node != NULL ? trav→rtbst_node→rtbst_data : NULL;
    } else {
        trav→rtbst_node = trav→rtbst_node→rtbst_link[1];
        while (trav→rtbst_node→rtbst_link[0] != NULL)
            trav→rtbst_node = trav→rtbst_node→rtbst_link[0];
        return trav→rtbst_node→rtbst_data;
    }
}
```

This code is included in §395.

10.6.5 Backing Up to the Previous Node

Moving an RTBST traverser backward has the same cases as in the other ways of finding an inorder predecessor that we've already discussed. The two main cases are distinguished on whether the current item has a left child; the third case comes up when there is no current item, implemented simply by delegation to *rtbst_t_last*():

§400 ⟨ RTBST traverser back up function 400 ⟩ ≡

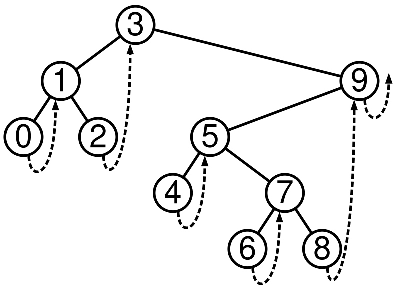
```
void *rtbst_t_prev (struct rtbst_traverser *trav) {
    assert (trav != NULL);
    if (trav→rtbst_node == NULL)
        return rtbst_t_last (trav, trav→rtbst_table);
    else if (trav→rtbst_node→rtbst_link[0] == NULL) {
        ⟨ Find predecessor of RTBST node with no left child 401 ⟩
    } else {
        ⟨ Find predecessor of RTBST node with left child 402 ⟩
    }
}
```

This code is included in §395.

The novel case is where the node *p* whose predecessor we want has no left child. In this case, we use a modified version of the algorithm originally specified for finding a node's successor in an unthreaded tree (see Section 4.9.3 [Better Iterative Traversal], page 53). We take the idea of moving up until we've moved up to the left, and turn it upside down (to avoid need for a parent stack) and reverse it (to find the predecessor instead of the successor).

The idea here is to trace p's entire direct ancestral line. Starting from the root of the tree, we repeatedly compare each node's data with p's and use the result to move downward, until we encounter node p itself. Each time we move down from a node x to its right child, we record x as the potential predecessor of p. When we finally arrive at p, the last node so selected is the actual predecessor, or if none was selected then p is the least node in the tree and we select the null item as its predecessor.

Consider this algorithm in the context of the tree shown here:



To find the predecessor of node 8, we trace the path from the root down to it: 3-9-5-7-8. The last time we move down to the right is from 7 to 8, so 7 is node 8's predecessor. To find the predecessor of node 6, we trace the path 3-9-5-7-6 and notice that we last move down to the right from 5 to 7, so 5 is node 6's predecessor. Finally, node 0 has the null item as its predecessor because path 3-1-0 does not involve any rightward movement.

Here is the code to implement this case:

§401 ⟨Find predecessor of RTBST node with no left child 401⟩ ≡

```
rtbst_comparison_func *cmp = trav->rtbst_table->rtbst_compare;
void *param = trav->rtbst_table->rtbst_param;
struct rtbst_node *node = trav->rtbst_node;
struct rtbst_node *i;

trav->rtbst_node = NULL;
for (i = trav->rtbst_table->rtbst_root; i != node; ) {
    int dir = cmp (node->rtbst_data, i->rtbst_data, param) > 0;
    if (dir == 1)
        trav->rtbst_node = i;
    i = i->rtbst_link[dir];
}
return trav->rtbst_node != NULL ? trav->rtbst_node->rtbst_data : NULL;
```

This code is included in §400.

The other case, where the node whose predecessor we want has a left child, is nothing new. We just find the largest node in the node's left subtree:

§402 ⟨Find predecessor of RTBST node with left child 402⟩ ≡

```
trav->rtbst_node = trav->rtbst_node->rtbst_link[0];
while (trav->rtbst_node->rtbst_rtag == RTBST_CHILD)
```

```
    trav->rtbst_node = trav->rtbst_node->rtbst_link[1];
return trav->rtbst_node->rtbst_data;
```

This code is included in §400.

10.7 Copying

The algorithm that we used for copying a TBST makes use of threads, but only right threads, so we can apply this algorithm essentially unmodified to RTBSTs.

We will make one change that superficially simplifies and improves the elegance of the algorithm. Function *tbst_copy*() in ⟨ TBST main copy function 279 ⟩ uses a pair of local variables *rp* and *rq* to store pointers to the original and new tree's root, because accessing the tag field of a cast "pseudo-root" pointer produces undefined behavior. However, in an RTBST there is no tag for a node's left subtree. During a TBST copy, only the left tags of the root nodes are accessed, so this means that we can use the pseudo-roots in the RTBST copy, with no need for *rp* or *rq*.

§403 ⟨ RTBST main copy function 403 ⟩ ≡

```
struct rtbst_table *rtbst_copy (const struct rtbst_table *org, rtbst_copy_func *copy,
                                rtbst_item_func *destroy, struct libavl_allocator *allocator)
{
    struct rtbst_table *new;
    const struct rtbst_node *p;
    struct rtbst_node *q;
    assert (org != NULL);
    new = rtbst_create (org->rtbst_compare, org->rtbst_param,
                        allocator != NULL ? allocator : org->rtbst_alloc);
    if (new == NULL)
        return NULL;
    new->rtbst_count = org->rtbst_count;
    if (new->rtbst_count == 0)
        return new;
    p = (struct rtbst_node *) &org->rtbst_root;
    q = (struct rtbst_node *) &new->rtbst_root;
    for (;;) {
        if (p->rtbst_link[0] != NULL) {
            if (!copy_node (new, q, 0, p->rtbst_link[0], copy)) {
                copy_error_recovery (new, destroy);
                return NULL;
            }
            p = p->rtbst_link[0];
            q = q->rtbst_link[0];
        } else {
            while (p->rtbst_rtag == RTBST_THREAD) {
                p = p->rtbst_link[1];
                if (p == NULL) {
                    q->rtbst_link[1] = NULL;
```

```
                    return new;
                }
                q = q->rtbst_link[1];
            }
            p = p->rtbst_link[1];
            q = q->rtbst_link[1];
        }
        if (p->rtbst_rtag == RTBST_CHILD)
            if (!copy_node (new, q, 1, p->rtbst_link[1], copy)) {
                copy_error_recovery (new, destroy);
                return NULL;
            }
    }
}
```

This code is included in §406 and §447.

The code to copy a node must be modified to deal with the asymmetrical nature of insertion in an RTBST:

§404 ⟨ RTBST node copy function 404 ⟩ ≡

```
static int copy_node (struct rtbst_table *tree, struct rtbst_node *dst, int dir,
                      const struct rtbst_node *src, rtbst_copy_func *copy) {
    struct rtbst_node *new = tree->rtbst_alloc->libavl_malloc (tree->rtbst_alloc, sizeof *new);
    if (new == NULL)
        return 0;
    new->rtbst_link[0] = NULL;
    new->rtbst_rtag = RTBST_THREAD;
    if (dir == 0)
        new->rtbst_link[1] = dst;
    else {
        new->rtbst_link[1] = dst->rtbst_link[1];
        dst->rtbst_rtag = RTBST_CHILD;
    }
    dst->rtbst_link[dir] = new;
    if (copy == NULL)
        new->rtbst_data = src->rtbst_data;
    else {
        new->rtbst_data = copy (src->rtbst_data, tree->rtbst_param);
        if (new->rtbst_data == NULL)
            return 0;
    }
    return 1;
}
```

This code is included in §406.

The error recovery function for copying is a bit simpler now, because the use of the pseudo-root means that no assignment to the new tree's root need take place, eliminating the need for one of the function's parameters:

§405 ⟨ RTBST copy error helper function 405 ⟩ ≡

```
static void copy_error_recovery (struct rtbst_table *new, rtbst_item_func *destroy) {
  struct rtbst_node *p = new->rtbst_root;
  if (p != NULL) {
    while (p->rtbst_rtag == RTBST_CHILD)
      p = p->rtbst_link[1];
    p->rtbst_link[1] = NULL;
  }
  rtbst_destroy (new, destroy);
}
```

This code is included in §406 and §447.

§406 ⟨ RTBST copy function 406 ⟩ ≡

⟨ RTBST node copy function 404 ⟩
⟨ RTBST copy error helper function 405 ⟩
⟨ RTBST main copy function 403 ⟩

This code is included in §375.

10.8 Destruction

The destruction algorithm for TBSTs makes use only of right threads, so we can easily adapt it for RTBSTs.

§407 ⟨ RTBST destruction function 407 ⟩ ≡

```
void rtbst_destroy (struct rtbst_table *tree, rtbst_item_func *destroy) {
  struct rtbst_node *p; /* Current node. */
  struct rtbst_node *n; /* Next node. */

  p = tree->rtbst_root;
  if (p != NULL)
    while (p->rtbst_link[0] != NULL)
      p = p->rtbst_link[0];

  while (p != NULL) {
    n = p->rtbst_link[1];
    if (p->rtbst_rtag == RTBST_CHILD)
      while (n->rtbst_link[0] != NULL)
        n = n->rtbst_link[0];

    if (destroy != NULL && p->rtbst_data != NULL)
      destroy (p->rtbst_data, tree->rtbst_param);
    tree->rtbst_alloc->libavl_free (tree->rtbst_alloc, p);

    p = n;
  }

  tree->rtbst_alloc->libavl_free (tree->rtbst_alloc, tree);
}
```

This code is included in §375, §418, and §455.

10.9 Balance

As for so many other operations, we can reuse most of the TBST balancing code to rebalance RTBSTs. Some of the helper functions can be completely recycled:

§408 ⟨ RTBST balance function 408 ⟩ ≡
⟨ RTBST tree-to-vine function 409 ⟩
⟨ RTBST vine compression function 410 ⟩
⟨ TBST vine-to-tree function; tbst ⇒ rtbst 285 ⟩
⟨ TBST main balance function; tbst ⇒ rtbst 283 ⟩

This code is included in §375.

The only substantative difference for the remaining two functions is that there is no need to set nodes' left tags (since they don't have any):

§409 ⟨ RTBST tree-to-vine function 409 ⟩ ≡

```
static void tree_to_vine (struct rtbst_table *tree) {
    struct rtbst_node *p;
    if (tree->rtbst_root == NULL)
        return;
    p = tree->rtbst_root;
    while (p->rtbst_link[0] != NULL)
        p = p->rtbst_link[0];
    for (;;) {
        struct rtbst_node *q = p->rtbst_link[1];
        if (p->rtbst_rtag == RTBST_CHILD) {
            while (q->rtbst_link[0] != NULL)
                q = q->rtbst_link[0];
            p->rtbst_rtag = RTBST_THREAD;
            p->rtbst_link[1] = q;
        }
        if (q == NULL)
            break;
        q->rtbst_link[0] = p;
        p = q;
    }
    tree->rtbst_root = p;
}
```

This code is included in §408.

§410 ⟨ RTBST vine compression function 410 ⟩ ≡

```
/* Performs a compression transformation count times, starting at root. */
static void compress (struct rtbst_node *root,
                      unsigned long nonthread, unsigned long thread) {
    assert (root != NULL);
    while (nonthread--) {
        struct rtbst_node *red = root->rtbst_link[0];
        struct rtbst_node *black = red->rtbst_link[0];
```

```
        root→rtbst_link[0] = black;
        red→rtbst_link[0] = black→rtbst_link[1];
        black→rtbst_link[1] = red;
        root = black;
    }
    while (thread−−) {
        struct rtbst_node *red = root→rtbst_link[0];
        struct rtbst_node *black = red→rtbst_link[0];
        root→rtbst_link[0] = black;
        red→rtbst_link[0] = NULL;
        black→rtbst_rtag = RTBST_CHILD;
        root = black;
    }
}
```

This code is included in §408.

10.10 Testing

There's nothing new or interesting in the test code.

§411 ⟨ rtbst-test.c 411 ⟩ ≡

```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ limits.h ⟩
#include ⟨ stdio.h ⟩
#include "rtbst.h"
#include "test.h"
⟨ RTBST print function 412 ⟩
⟨ BST traverser check function; bst ⇒ rtbst 104 ⟩
⟨ Compare two RTBSTs for structure and content 413 ⟩
⟨ Recursively verify RTBST structure 414 ⟩
⟨ BST verify function; bst ⇒ rtbst 109 ⟩
⟨ TBST test function; tbst ⇒ rtbst 295 ⟩
⟨ BST overflow test function; bst ⇒ rtbst 122 ⟩
```

§412 ⟨ RTBST print function 412 ⟩ ≡

```
void print_tree_structure (struct rtbst_node *node, int level) {
    if (level > 16) {
        printf ("[...]");
        return;
    }
    if (node == NULL) {
        printf ("<nil>");
        return;
    }
    printf ("%d(", node→rtbst_data ? *(int *) node→rtbst_data : −1);
    if (node→rtbst_link[0] != NULL)
```

```
        print_tree_structure (node->rtbst_link[0], level + 1);
    fputs (", ", stdout);
    if (node->rtbst_rtag == RTBST_CHILD) {
        if (node->rtbst_link[1] == node)
            printf ("loop");
        else print_tree_structure (node->rtbst_link[1], level + 1);
    }
    else if (node->rtbst_link[1] != NULL) printf (">%d", (node->rtbst_link[1]->rtbst_data
                                ? *(int *) node->rtbst_link[1]->rtbst_data : -1));
    else printf (">>");
    putchar (')');
}
void print_whole_tree (const struct rtbst_table *tree, const char *title) {
    printf ("%s: ", title);
    print_tree_structure (tree->rtbst_root, 0);
    putchar ('\n');
}
```

This code is included in §411, §449, and §482.

§413 ⟨ Compare two RTBSTs for structure and content 413 ⟩ ≡

```
static int compare_trees (struct rtbst_node *a, struct rtbst_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
        if (a != NULL || b != NULL) {
            printf (" a=%d b=%d\n",
                    a ? *(int *) a->rtbst_data : -1,
                    b ? *(int *) b->rtbst_data : -1);
            assert (0);
        }
        return 1;
    }
    assert (a != b);
    if (*(int *) a->rtbst_data != *(int *) b->rtbst_data
      || a->rtbst_rtag != b->rtbst_rtag) {
        printf (" Copied nodes differ: a=%d b=%d a:",
                *(int *) a->rtbst_data, *(int *) b->rtbst_data);
        if (a->rtbst_rtag == RTBST_CHILD) printf ("r");
        printf (" b:");
        if (b->rtbst_rtag == RTBST_CHILD) printf ("r");
        printf ("\n");
        return 0;
    }
    if (a->rtbst_rtag == RTBST_THREAD)
        assert ((a->rtbst_link[1] == NULL)
                != (a->rtbst_link[1] != b->rtbst_link[1]));
```

```
    okay = compare_trees (a->rtbst_link[0], b->rtbst_link[0]);
    if (a->rtbst_rtag == RTBST_CHILD)
        okay &= compare_trees (a->rtbst_link[1], b->rtbst_link[1]);
    return okay;
}
```

This code is included in §411.

§414 ⟨ Recursively verify RTBST structure 414 ⟩ ≡

```
static void recurse_verify_tree (struct rtbst_node *node, int *okay, size_t *count,
                                 int min, int max) {
    int d;  /* Value of this node's data. */
    size_t subcount[2];  /* Number of nodes in subtrees. */

    if (node == NULL) {
        *count = 0;
        return;
    }
    d = *(int *) node->rtbst_data;

    ⟨ Verify binary search tree ordering 114 ⟩

    subcount[0] = subcount[1] = 0;
    recurse_verify_tree (node->rtbst_link[0], okay, &subcount[0], min, d - 1);
    if (node->rtbst_rtag == RTBST_CHILD)
        recurse_verify_tree (node->rtbst_link[1], okay, &subcount[1], d + 1, max);
    *count = 1 + subcount[0] + subcount[1];
}
```

This code is included in §411.

11 Right-Threaded AVL Trees

In the same way that we can combine threaded trees with AVL trees to produce threaded AVL trees, we can combine right-threaded trees with AVL trees to produce right-threaded AVL trees. This chapter explores this combination, producing another table implementation.

Here's the form of the source and header files. Notice the use of *rtavl_* as the identifier prefix. Likewise, we will often refer to right-threaded AVL trees as "RTAVL trees".

§415
```
⟨ rtavl.h 415 ⟩ ≡
⟨ License 1 ⟩
#ifndef RTAVL_H
#define RTAVL_H 1
#include ⟨ stddef.h ⟩
⟨ Table types; tbl ⇒ rtavl 14 ⟩
⟨ BST maximum height; bst ⇒ rtavl 28 ⟩
⟨ TBST table structure; tbst ⇒ rtavl 250 ⟩
⟨ RTAVL node structure 417 ⟩
⟨ TBST traverser structure; tbst ⇒ rtavl 267 ⟩
⟨ Table function prototypes; tbl ⇒ rtavl 15 ⟩
#endif /* rtavl.h */
```

§416
```
⟨ rtavl.c 416 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "rtavl.h"
⟨ RTAVL functions 418 ⟩
```

11.1 Data Types

Besides the members needed for any BST, an RTAVL node structure needs a tag to indicate whether the right link is a child pointer or a thread, and a balance factor to facilitate AVL balancing. Here's what we end up with:

§417
```
⟨ RTAVL node structure 417 ⟩ ≡
/* Characterizes a link as a child pointer or a thread. */
enum rtavl_tag {
    RTAVL_CHILD, /* Child pointer. */
    RTAVL_THREAD /* Thread. */
};
/* A threaded binary search tree node. */
struct rtavl_node {
    struct rtavl_node *rtavl_link[2]; /* Subtrees. */
    void *rtavl_data; /* Pointer to data. */
    unsigned char rtavl_rtag; /* Tag field. */
    signed char rtavl_balance; /* Balance factor. */
```

};

This code is included in §415.

11.2 Operations

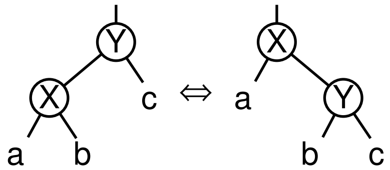
Most of the operations for RTAVL trees can come directly from their RTBST implementations. The notable exceptions are, as usual, the insertion and deletion functions. The copy function will also need a small tweak. Here's the list of operations:

§418 ⟨ RTAVL functions 418 ⟩ ≡
⟨ TBST creation function; tbst ⇒ rtavl 252 ⟩
⟨ RTBST search function; rtbst ⇒ rtavl 376 ⟩
⟨ RTAVL item insertion function 419 ⟩
⟨ Table insertion convenience functions; tbl ⇒ rtavl 592 ⟩
⟨ RTAVL item deletion function 429 ⟩
⟨ RTBST traversal functions; rtbst ⇒ rtavl 395 ⟩
⟨ RTAVL copy function 447 ⟩
⟨ RTBST destruction function; rtbst ⇒ rtavl 407 ⟩
⟨ Default memory allocation functions; tbl ⇒ rtavl 6 ⟩
⟨ Table assertion functions; tbl ⇒ rtavl 594 ⟩

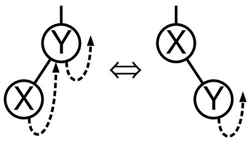
This code is included in §416.

11.3 Rotations

We will use rotations in right-threaded trees in the same way as for other kinds of trees that we have already examined. As always, a generic rotation looks like this:



On the left side of this diagram, a may be an empty subtree and b and c may be threads. On the right side, a and b may be empty subtrees and c may be a thread. If none of them in fact represent actual nodes, then we end up with the following pathological case:



Notice the asymmetry here: in a right rotation the right thread from X to Y becomes a null left child of Y, but in a left rotation this is reversed and a null subtree b becomes a right thread from X to Y. Contrast this to the correponding rotation in a threaded tree (see Section 8.2 [TBST Rotations], page 192), where either way the same kind of change occurs: the thread from X to Y, or vice versa, simply reverses direction.

As with other kinds of rotations we've seen, there is no need to make any changes in subtrees of a, b, or c, because of rotations' locality and order-preserving properties (see

Section 4.3 [BST Rotations], page 33). In particular, nodes a and c, if they exist, need no adjustments, as implied by the diagram above, which shows no changes to these subtrees on opposite sides.

Exercises:

1. Write functions for right and left rotations in right-threaded BSTs, analogous to those for unthreaded BSTs developed in Exercise 4.3-2.

11.4 Insertion

Insertion into an RTAVL tree follows the same pattern as insertion into other kinds of balanced tree. The outline is straightforward:

§419 ⟨ RTAVL item insertion function 419 ⟩ ≡

```
void **rtavl_probe (struct rtavl_table *tree, void *item) {
    ⟨ avl_probe() local variables; avl ⇒ rtavl 147 ⟩
    assert (tree != NULL && item != NULL);
    ⟨ Step 1: Search RTAVL tree for insertion point 420 ⟩
    ⟨ Step 2: Insert RTAVL node 421 ⟩
    ⟨ Step 3: Update balance factors after AVL insertion; avl ⇒ rtavl 150 ⟩
    ⟨ Step 4: Rebalance after RTAVL insertion 422 ⟩
}
```

This code is included in §418.

11.4.1 Steps 1–2: Search and Insert

The basic insertion step itself follows the same steps as ⟨ RTBST item insertion function 377 ⟩ does for a plain RTBST. We do keep track of the directions moved on stack *da*[] and the last-seen node with nonzero balance factor, in the same way as ⟨ Step 1: Search AVL tree for insertion point 148 ⟩ for unthreaded AVL trees.

§420 ⟨ Step 1: Search RTAVL tree for insertion point 420 ⟩ ≡

```
z = (struct rtavl_node *) &tree→rtavl_root;
y = tree→rtavl_root;
if (tree→rtavl_root != NULL)
    for (q = z, p = y; ; q = p, p = p→rtavl_link[dir]) {
        int cmp = tree→rtavl_compare (item, p→rtavl_data, tree→rtavl_param);
        if (cmp == 0)
            return &p→rtavl_data;
        if (p→rtavl_balance != 0)
            z = q, y = p, k = 0;
        da[k++] = dir = cmp > 0;
        if (dir == 0) {
            if (p→rtavl_link[0] == NULL)
                break;
        } else /* dir == 1 */ {
            if (p→rtavl_rtag == RTAVL_THREAD)
                break;
```

```
        }
    }
else {
    p = (struct rtavl_node *) &tree→rtavl_root;
    dir = 0;
}
```

This code is included in §419.

§421 ⟨ Step 2: Insert RTAVL node 421 ⟩ ≡

```
n = tree→rtavl_alloc→libavl_malloc (tree→rtavl_alloc, sizeof *n);
if (n == NULL)
    return NULL;
tree→rtavl_count++;
n→rtavl_data = item;
n→rtavl_link[0] = NULL;
if (dir == 0)
    n→rtavl_link[1] = p;
else /* dir == 1 */ {
    p→rtavl_rtag = RTAVL_CHILD;
    n→rtavl_link[1] = p→rtavl_link[1];
}
n→rtavl_rtag = RTAVL_THREAD;
n→rtavl_balance = 0;
p→rtavl_link[dir] = n;
if (y == NULL) {
    n→rtavl_link[1] = NULL;
    return &n→rtavl_data;
}
```

This code is included in §419.

11.4.2 Step 4: Rebalance

Unlike all of the AVL rebalancing algorithms we've seen so far, rebalancing of a right-threaded AVL tree is not symmetric. This means that we cannot single out left-side rebalancing or right-side rebalancing as we did before, hand-waving the rest of it as a symmetric case. But both cases are very similar, if not exactly symmetric, so we will present the corresponding cases together. The theory is exactly the same as before (see Section 5.4.4 [Rebalancing AVL Trees], page 115). Here is the code to choose between left-side and right-side rebalancing:

§422 ⟨ Step 4: Rebalance after RTAVL insertion 422 ⟩ ≡

```
if (y→rtavl_balance == -2)
    { ⟨ Step 4: Rebalance RTAVL tree after insertion to left 423 ⟩ }
else if (y→rtavl_balance == +2)
    { ⟨ Step 4: Rebalance RTAVL tree after insertion to right 424 ⟩ }
else return &n→rtavl_data;
z→rtavl_link[y != z→rtavl_link[0]] = w;
return &n→rtavl_data;
```

This code is included in §419.

The code to choose between the two subcases within the left-side and right-side rebalancing cases follows below. As usual during rebalancing, y is the node at which rebalancing occurs, x is its child on the same side as the inserted node, and cases are distinguished on the basis of x's balance factor:

§423 ⟨ Step 4: Rebalance RTAVL tree after insertion to left 423 ⟩ ≡
struct rtavl_node $*x = y \rightarrow rtavl\_link[0]$;
if ($x \rightarrow rtavl\_balance == {}^{-}1$)
  { ⟨ Rebalance for − balance factor in RTAVL insertion in left subtree 425 ⟩ }
else { ⟨ Rebalance for + balance factor in RTAVL insertion in left subtree 427 ⟩ }

This code is included in §422.

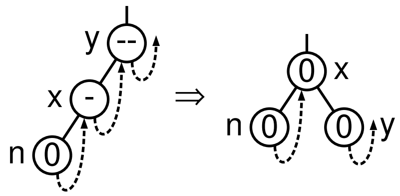
§424 ⟨ Step 4: Rebalance RTAVL tree after insertion to right 424 ⟩ ≡
struct rtavl_node $*x = y \rightarrow rtavl\_link[1]$;
if ($x \rightarrow rtavl\_balance == {}^{+}1$)
  { ⟨ Rebalance for + balance factor in RTAVL insertion in right subtree 426 ⟩ }
else { ⟨ Rebalance for − balance factor in RTAVL insertion in right subtree 428 ⟩ }

This code is included in §422.

Case 1: *x* has taller subtree on side of insertion

If node x's taller subtree is on the same side as the inserted node, then we perform a rotation at y in the opposite direction. That is, if the insertion occurred in the left subtree of y and x has a − balance factor, we rotate right at y, and if the insertion was to the right and x has a + balance factor, we rotate left at y. This changes the balance of both x and y to zero. None of this is a change from unthreaded or fully threaded rebalancing. The difference is in the handling of empty subtrees, that is, in the rotation itself (see Section 11.3 [RTBST Rotations], page 248).

Here is a diagram of left-side rebalancing for the interesting case where x has a right thread. Taken along with x's − balance factor, this means that n, the newly inserted node, must be x's left child. Therefore, subtree x has height 2, so y has no right child (because it has a ${}^{-}2$ balance factor). This chain of logic means that we know exactly what the tree looks like in this particular subcase:

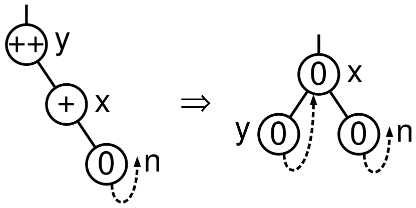


§425 ⟨ Rebalance for − balance factor in RTAVL insertion in left subtree 425 ⟩ ≡
$w = x$;
if ($x \rightarrow rtavl\_rtag ==$ `RTAVL_THREAD`) {
  $x \rightarrow rtavl\_rtag =$ `RTAVL_CHILD`;
  $y \rightarrow rtavl\_link[0] =$ `NULL`;
}
else $y \rightarrow rtavl\_link[0] = x \rightarrow rtavl\_link[1]$;
$x \rightarrow rtavl\_link[1] = y$;

$x \to rtavl\_balance = y \to rtavl\_balance = 0;$

This code is included in §423.

Here is the diagram and code for the similar right-side case:



§426 ⟨Rebalance for + balance factor in RTAVL insertion in right subtree 426⟩ ≡

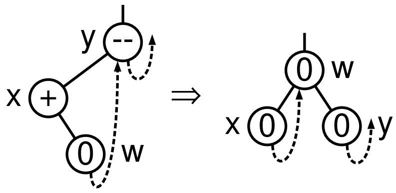
$w = x;$
if $(x \to rtavl\_link[0] ==$ `NULL`$)$ {
    $y \to rtavl\_rtag =$ `RTAVL_THREAD`;
    $y \to rtavl\_link[1] = x;$
}
else $y \to rtavl\_link[1] = x \to rtavl\_link[0];$
$x \to rtavl\_link[0] = y;$
$x \to rtavl\_balance = y \to rtavl\_balance = 0;$

This code is included in §424.

Case 2: *x* has taller subtree on side opposite insertion

If node x's taller subtree is on the side opposite the newly inserted node, then we perform a double rotation: first rotate at x in the same direction as the inserted node, then in the opposite direction at y. This is the same as in a threaded or unthreaded tree, and indeed we can reuse much of the code.

The case where the details differ is, as usual, where threads or null child pointers are moved around. In the most extreme case for insertion to the left, where w is a leaf, we know that x has no left child and s no right child, and the situation looks like the diagram below before and after the rebalancing step:

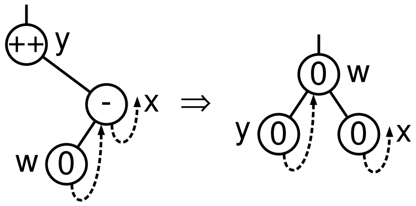


§427 ⟨Rebalance for + balance factor in RTAVL insertion in left subtree 427⟩ ≡

⟨Rotate left at x then right at y in AVL tree; avl ⇒ rtavl 156⟩
if $(x \to rtavl\_link[1] ==$ `NULL`$)$ {
    $x \to rtavl\_rtag =$ `RTAVL_THREAD`;
    $x \to rtavl\_link[1] = w;$
}
if $(w \to rtavl\_rtag ==$ `RTAVL_THREAD`$)$ {
    $y \to rtavl\_link[0] =$ `NULL`;
    $w \to rtavl\_rtag =$ `RTAVL_CHILD`;
}

This code is included in §423 and §442.

Here is the code and diagram for right-side insertion rebalancing:



§428 ⟨ Rebalance for − balance factor in RTAVL insertion in right subtree 428 ⟩ ≡

```
⟨ Rotate right at x then left at y in AVL tree; avl ⇒ rtavl 159 ⟩
if (y→rtavl_link[1] == NULL) {
    y→rtavl_rtag = RTAVL_THREAD;
    y→rtavl_link[1] = w;
}
if (w→rtavl_rtag == RTAVL_THREAD) {
    x→rtavl_link[0] = NULL;
    w→rtavl_rtag = RTAVL_CHILD;
}
```

This code is included in §424 and §441.

11.5 Deletion

Deletion in an RTAVL tree takes the usual pattern.

§429 ⟨ RTAVL item deletion function 429 ⟩ ≡

```
void *rtavl_delete (struct rtavl_table *tree, const void *item) {
    /* Stack of nodes. */
    struct rtavl_node *pa[RTAVL_MAX_HEIGHT]; /* Nodes. */
    unsigned char da[RTAVL_MAX_HEIGHT]; /* rtavl_link[] indexes. */
    int k; /* Stack pointer. */

    struct rtavl_node *p; /* Traverses tree to find node to delete. */

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search RTAVL tree for item to delete 430 ⟩
    ⟨ Step 2: Delete RTAVL node 431 ⟩
    ⟨ Steps 3 and 4: Update balance factors and rebalance after RTAVL deletion 438 ⟩

    return (void *) item;
}
```

This code is included in §418.

11.5.1 Step 1: Search

There's nothing new in searching an RTAVL tree for a node to delete. We use p to search the tree, and push its chain of parent nodes onto stack $pa[]$ along with the directions $da[]$ moved down from them, including the pseudo-root node at the top.

§430 ⟨ Step 1: Search RTAVL tree for item to delete 430 ⟩ ≡

```
k = 1;
da[0] = 0;
pa[0] = (struct rtavl_node *) &tree→rtavl_root;
p = tree→rtavl_root;
if (p == NULL)
      return NULL;
for (;;) {
      int cmp, dir;
      cmp = tree→rtavl_compare (item, p→rtavl_data, tree→rtavl_param);
      if (cmp == 0)
            break;
      dir = cmp > 0;
      if (dir == 0) {
            if (p→rtavl_link[0] == NULL)
                  return NULL;
      } else /* dir == 1 */ {
            if (p→rtavl_rtag == RTAVL_THREAD)
                  return NULL;
      }
      pa[k] = p;
      da[k++] = dir;
      p = p→rtavl_link[dir];
}
tree→rtavl_count−−;
item = p→rtavl_data;
```

This code is included in §429 and §468.

11.5.2 Step 2: Delete

As demonstrated in the previous chapter, left-looking deletion, where we examine the left subtree of the node to be deleted, is more efficient than right-looking deletion in an RTBST (see Section 10.5.2 [Left-Looking Deletion in an RTBST], page 232). This holds true in an RTAVL tree, too.

§431 ⟨ Step 2: Delete RTAVL node 431 ⟩ ≡

```
if (p→rtavl_link[0] == NULL) {
      if (p→rtavl_rtag == RTAVL_CHILD)
            { ⟨ Case 1 in RTAVL deletion 432 ⟩ }
      else { ⟨ Case 2 in RTAVL deletion 433 ⟩ }
} else {
      struct rtavl_node *r = p→rtavl_link[0];
      if (r→rtavl_rtag == RTAVL_THREAD)
            { ⟨ Case 3 in RTAVL deletion 434 ⟩ }
      else { ⟨ Case 4 in RTAVL deletion 435 ⟩ }
}
tree→rtavl_alloc→libavl_free (tree→rtavl_alloc, p);
```

This code is included in §429.

Case 1: *p* has a right child but no left child

If the node to be deleted, p, has a right child but not a left child, then we replace it by its right child.

§432 ⟨ Case 1 in RTAVL deletion 432 ⟩ ≡
$pa[k-1] \to rtavl\_link[da[k-1]] = p \to rtavl\_link[1];$

This code is included in §431 and §470.

Case 2: *p* has a right thread and no left child

If we are deleting a leaf, then we replace it by a null pointer if it's a left child, or by a pointer to its own former right thread if it's a right child. Refer back to the commentary on ⟨ Case 2 in right-looking RTBST deletion 385 ⟩ for further explanation.

§433 ⟨ Case 2 in RTAVL deletion 433 ⟩ ≡
```
pa[k - 1]->rtavl_link[da[k - 1]] = p->rtavl_link[da[k - 1]];
if (da[k - 1] == 1)
    pa[k - 1]->rtavl_rtag = RTAVL_THREAD;
```

This code is included in §431 and §471.

Case 3: *p*'s left child has a right thread

If p has a left child r, and r has a right thread, then we replace p by r and transfer p's former right link to r. Node r also receives p's balance factor.

§434 ⟨ Case 3 in RTAVL deletion 434 ⟩ ≡
```
r->rtavl_link[1] = p->rtavl_link[1];
r->rtavl_rtag = p->rtavl_rtag;
r->rtavl_balance = p->rtavl_balance;
pa[k - 1]->rtavl_link[da[k - 1]] = r;
da[k] = 0;
pa[k++] = r;
```

This code is included in §431.

Case 4: *p*'s left child has a right child

The final case, where node p's left child r has a right child, is also the most complicated. We find p's predecessor s first:

§435 ⟨ Case 4 in RTAVL deletion 435 ⟩ ≡
```
struct rtavl_node *s;
int j = k++;
for (;;) {
    da[k] = 1;
    pa[k++] = r;
    s = r->rtavl_link[1];
    if (s->rtavl_rtag == RTAVL_THREAD)
        break;
    r = s;
```

```
}
```

See also §436 and §437.
This code is included in §431.

Then we move s into p's place, not forgetting to update links and tags as necessary:

§436 ⟨ Case 4 in RTAVL deletion 435 ⟩ +≡

```
da[j] = 0;
pa[j] = pa[j − 1]→rtavl_link[da[j − 1]] = s;
if (s→rtavl_link[0] != NULL)
    r→rtavl_link[1] = s→rtavl_link[0];
else {
    r→rtavl_rtag = RTAVL_THREAD;
    r→rtavl_link[1] = s;
}
```

Finally, we copy p's old information into s, except for the actual data:

§437 ⟨ Case 4 in RTAVL deletion 435 ⟩ +≡

```
s→rtavl_balance = p→rtavl_balance;
s→rtavl_link[0] = p→rtavl_link[0];
s→rtavl_link[1] = p→rtavl_link[1];
s→rtavl_rtag = p→rtavl_rtag;
```

11.5.3 Step 3: Update Balance Factors

Updating balance factors works exactly the same way as in unthreaded AVL deletion (see Section 5.5.3 [Deleting an AVL Node Step 3 - Update], page 125).

§438 ⟨ Steps 3 and 4: Update balance factors and rebalance after RTAVL deletion 438 ⟩ ≡

```
assert (k > 0);
while (−−k > 0) {
    struct rtavl_node *y = pa[k];
    if (da[k] == 0) {
        y→rtavl_balance++;
        if (y→rtavl_balance == +1)
            break;
        else if (y→rtavl_balance == +2) {
            ⟨ Step 4: Rebalance after RTAVL deletion in left subtree 439 ⟩
        }
    } else {
        y→rtavl_balance−−;
        if (y→rtavl_balance == −1)
            break;
        else if (y→rtavl_balance == −2) {
            ⟨ Step 4: Rebalance after RTAVL deletion in right subtree 440 ⟩
        }
    }
}
```

This code is included in §429.

11.5.4 Step 4: Rebalance

Rebalancing in an RTAVL tree after deletion is not completely symmetric between left-side and right-side rebalancing, but there are pairs of similar subcases on each side. The outlines are similar, too. Either way, rebalancing occurs at node y, and cases are distinguished based on the balance factor of x, the child of y on the side opposite the deletion.

§439 ⟨ Step 4: Rebalance after RTAVL deletion in left subtree 439 ⟩ ≡

```
struct rtavl_node *x = y→rtavl_link[1];
assert (x != NULL);
if (x→rtavl_balance == -1) {
    ⟨ Rebalance for − balance factor after left-side RTAVL deletion 441 ⟩
} else {
    pa[k - 1]→rtavl_link[da[k - 1]] = x;
    if (x→rtavl_balance == 0) {
        ⟨ Rebalance for 0 balance factor after left-side RTAVL deletion 443 ⟩
        break;
    }
    else /* x→rtavl_balance == +1 */ {
        ⟨ Rebalance for + balance factor after left-side RTAVL deletion 445 ⟩
    }
}
```

This code is included in §438.

§440 ⟨ Step 4: Rebalance after RTAVL deletion in right subtree 440 ⟩ ≡

```
struct rtavl_node *x = y→rtavl_link[0];
assert (x != NULL);
if (x→rtavl_balance == +1) {
    ⟨ Rebalance for + balance factor after right-side RTAVL deletion 442 ⟩
} else {
    pa[k - 1]→rtavl_link[da[k - 1]] = x;
    if (x→rtavl_balance == 0) {
        ⟨ Rebalance for 0 balance factor after right-side RTAVL deletion 444 ⟩
        break;
    }
    else /* x→rtavl_balance == -1 */ {
        ⟨ Rebalance for − balance factor after right-side RTAVL deletion 446 ⟩
    }
}
```

This code is included in §438.

Case 1: x has taller subtree on same side as deletion

If the taller subtree of x is on the same side as the deletion, then we rotate at x in the opposite direction from the deletion, then at y in the same direction as the deletion. This is the same as case 2 for RTAVL insertion (see page 252), which in turn performs the general transformation described for AVL deletion case 1 (see page 127), and we can reuse the code.

§441 ⟨ Rebalance for − balance factor after left-side RTAVL deletion 441 ⟩ ≡

struct rtavl_node $*w$;

⟨ Rebalance for − balance factor in RTAVL insertion in right subtree 428 ⟩
$pa[k-1] \rightarrow rtavl\_link[da[k-1]] = w$;

This code is included in §439.

§442 ⟨ Rebalance for + balance factor after right-side RTAVL deletion 442 ⟩ ≡
struct rtavl_node $*w$;

⟨ Rebalance for + balance factor in RTAVL insertion in left subtree 427 ⟩
$pa[k-1] \rightarrow rtavl\_link[da[k-1]] = w$;

This code is included in §440.

Case 2: *x*'s subtrees are equal height

If x's two subtrees are of equal height, then we perform a rotation at y toward the deletion. This rotation cannot be troublesome, for the same reason discussed for rebalancing in TAVL trees (see page 201). We can even reuse the code:

§443 ⟨ Rebalance for 0 balance factor after left-side RTAVL deletion 443 ⟩ ≡
⟨ Rebalance for 0 balance factor after TAVL deletion in left subtree; tavl ⇒ rtavl 321 ⟩

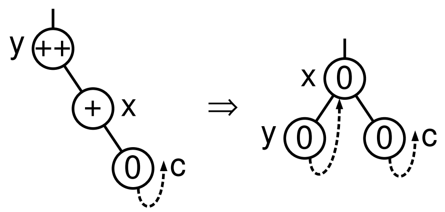
This code is included in §439.

§444 ⟨ Rebalance for 0 balance factor after right-side RTAVL deletion 444 ⟩ ≡
⟨ Rebalance for 0 balance factor after TAVL deletion in right subtree; tavl ⇒ rtavl 325 ⟩

This code is included in §440.

Case 3: *x* has taller subtree on side opposite deletion

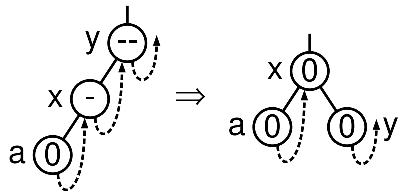
When x's taller subtree is on the side opposite the deletion, we rotate at y toward the deletion, same as case 2. If the deletion was on the left side of y, then the general form is the same as for TAVL deletion (see page 201). The special case for left-side deletion, where x lacks a left child, and the general form of the code, are shown here:



§445 ⟨ Rebalance for + balance factor after left-side RTAVL deletion 445 ⟩ ≡
if $(x \rightarrow rtavl\_link[0]$!= `NULL`)
    $y \rightarrow rtavl\_link[1] = x \rightarrow rtavl\_link[0]$;
else $y \rightarrow rtavl\_rtag$ = `RTAVL_THREAD`;
$x \rightarrow rtavl\_link[0] = y$;
$y \rightarrow rtavl\_balance = x \rightarrow rtavl\_balance = 0$;

This code is included in §439.

The special case for right-side deletion, where x lacks a right child, and the general form of the code, are shown here:



§446 ⟨ Rebalance for − balance factor after right-side RTAVL deletion 446 ⟩ ≡

```
if (x→rtavl_rtag == RTAVL_CHILD)
    y→rtavl_link[0] = x→rtavl_link[1];
else {
    y→rtavl_link[0] = NULL;
    x→rtavl_rtag = RTAVL_CHILD;
}
x→rtavl_link[1] = y;
y→rtavl_balance = x→rtavl_balance = 0;
```

This code is included in §440.

Exercises:

1. In the chapter about TAVL deletion, we offered two implementations of deletion: one using a stack (⟨ TAVL item deletion function, with stack 659 ⟩) and one using an algorithm to find node parents (⟨ TAVL item deletion function 311 ⟩). For RTAVL deletion, we offer only a stack-based implementation. Why?

2. The introduction to this section states that left-looking deletion is more efficient than right-looking deletion in an RTAVL tree. Confirm this by writing a right-looking alternate implementation of ⟨ Step 2: Delete RTAVL node 431 ⟩ and comparing the two sets of code.

3. Rewrite ⟨ Case 4 in RTAVL deletion 435 ⟩ to replace the deleted node's *rtavl_data* by its successor, then delete the successor, instead of shuffling pointers. (Refer back to Exercise 4.8-3 for an explanation of why this approach cannot be used in Libavl.)

11.6 Copying

We can reuse most of the RTBST copying functionality for copying RTAVL trees, but we must modify the node copy function to copy the balance factor into the new node as well.

§447 ⟨ RTAVL copy function 447 ⟩ ≡

⟨ RTAVL node copy function 448 ⟩
⟨ RTBST copy error helper function; rtbst ⇒ rtavl 405 ⟩
⟨ RTBST main copy function; rtbst ⇒ rtavl 403 ⟩

This code is included in §418 and §455.

§448 ⟨ RTAVL node copy function 448 ⟩ ≡

```
static int copy_node (struct rtavl_table *tree, struct rtavl_node *dst, int dir,
                      const struct rtavl_node *src, rtavl_copy_func *copy) {
  struct rtavl_node *new = tree→rtavl_alloc→libavl_malloc (tree→rtavl_alloc,
                                                            sizeof *new);
  if (new == NULL)
    return 0;
  new→rtavl_link[0] = NULL;
```

```
    new->rtavl_rtag = RTAVL_THREAD;
    if (dir == 0)
        new->rtavl_link[1] = dst;
    else {
        new->rtavl_link[1] = dst->rtavl_link[1];
        dst->rtavl_rtag = RTAVL_CHILD;
    }
    dst->rtavl_link[dir] = new;
    new->rtavl_balance = src->rtavl_balance;
    if (copy == NULL)
        new->rtavl_data = src->rtavl_data;
    else {
        new->rtavl_data = copy (src->rtavl_data, tree->rtavl_param);
        if (new->rtavl_data == NULL)
            return 0;
    }
    return 1;
}
```

This code is included in §447.

11.7 Testing

§449 ⟨ rtavl-test.c 449 ⟩ ≡

```
⟨ License 1 ⟩
#include <assert.h>
#include <limits.h>
#include <stdio.h>
#include "rtavl.h"
#include "test.h"
⟨ RTBST print function; rtbst ⇒ rtavl 412 ⟩
⟨ BST traverser check function; bst ⇒ rtavl 104 ⟩
⟨ Compare two RTAVL trees for structure and content 450 ⟩
⟨ Recursively verify RTAVL tree structure 451 ⟩
⟨ AVL tree verify function; avl ⇒ rtavl 190 ⟩
⟨ BST test function; bst ⇒ rtavl 100 ⟩
⟨ BST overflow test function; bst ⇒ rtavl 122 ⟩
```

§450 ⟨ Compare two RTAVL trees for structure and content 450 ⟩ ≡

```
static int compare_trees (struct rtavl_node *a, struct rtavl_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
        if (a != NULL || b != NULL) {
            printf (" a=%d b=%d\n",
                    a ? *(int *) a->rtavl_data : -1, b ? *(int *) b->rtavl_data : -1);
            assert (0);
        }
```

```
        return 1;
    }
    assert (a != b);
    if (*(int *) a->rtavl_data != *(int *) b->rtavl_data
        || a->rtavl_rtag != b->rtavl_rtag
        || a->rtavl_balance != b->rtavl_balance) {
        printf (" Copied nodes differ: a=%d (bal=%d) b=%d (bal=%d) a:",
                *(int *) a->rtavl_data, a->rtavl_balance,
                *(int *) b->rtavl_data, b->rtavl_balance);
        if (a->rtavl_rtag == RTAVL_CHILD) printf ("r");
        printf (" b:");
        if (b->rtavl_rtag == RTAVL_CHILD) printf ("r");
        printf ("\n");
        return 0;
    }
    if (a->rtavl_rtag == RTAVL_THREAD)
        assert ((a->rtavl_link[1] == NULL) != (a->rtavl_link[1] != b->rtavl_link[1]));
    okay = compare_trees (a->rtavl_link[0], b->rtavl_link[0]);
    if (a->rtavl_rtag == RTAVL_CHILD)
        okay &= compare_trees (a->rtavl_link[1], b->rtavl_link[1]);
    return okay;
}
```

This code is included in §449.

§451 ⟨ Recursively verify RTAVL tree structure 451 ⟩ ≡

```
static void recurse_verify_tree (struct rtavl_node *node, int *okay, size_t *count,
                                 int min, int max, int *height) {
    int d; /* Value of this node's data. */
    size_t subcount[2]; /* Number of nodes in subtrees. */
    int subheight[2]; /* Heights of subtrees. */
    if (node == NULL) {
        *count = 0;
        *height = 0;
        return;
    }
    d = *(int *) node->rtavl_data;
    ⟨ Verify binary search tree ordering 114 ⟩
    subcount[0] = subcount[1] = 0;
    subheight[0] = subheight[1] = 0;
    recurse_verify_tree (node->rtavl_link[0], okay, &subcount[0],
                         min, d - 1, &subheight[0]);
    if (node->rtavl_rtag == RTAVL_CHILD)
        recurse_verify_tree (node->rtavl_link[1], okay, &subcount[1],
                             d + 1, max, &subheight[1]);
    *count = 1 + subcount[0] + subcount[1];
```

$*height = 1 + (subheight[0] > subheight[1] \ ? \ subheight[0] : subheight[1]);$
⟨ Verify AVL node balance factor; avl ⇒ rtavl 189 ⟩
}

This code is included in §449.

12 Right-Threaded Red-Black Trees

This chapter is this book's final demonstration of right-threaded trees, carried out by using them in a red-black tree implementation of tables. The chapter, and the code, follow the pattern that should now be familiar, using *rtrb_* as the naming prefix and often referring to right-threaded right-black trees as "RTRB trees".

§452
```
⟨ rtrb.h 452 ⟩ ≡
⟨ License 1 ⟩
#ifndef RTRB_H
#define RTRB_H 1
#include ⟨ stddef.h ⟩
⟨ Table types; tbl ⇒ rtrb 14 ⟩
⟨ RB maximum height; rb ⇒ rtrb 195 ⟩
⟨ TBST table structure; tbst ⇒ rtrb 250 ⟩
⟨ RTRB node structure 454 ⟩
⟨ TBST traverser structure; tbst ⇒ rtrb 267 ⟩
⟨ Table function prototypes; tbl ⇒ rtrb 15 ⟩
#endif /* rtrb.h */
```

§453
```
⟨ rtrb.c 453 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "rtrb.h"
⟨ RTRB functions 455 ⟩
```

12.1 Data Types

Like any right-threaded tree node, an RTRB node has a right tag, and like any red-black tree node, an RTRB node has a color, either red or black. The combination is straightforward, as shown here.

§454
```
⟨ RTRB node structure 454 ⟩ ≡
/* Color of a red-black node. */
enum rtrb_color {
    RTRB_BLACK, /* Black. */
    RTRB_RED /* Red. */
};
/* Characterizes a link as a child pointer or a thread. */
enum rtrb_tag {
    RTRB_CHILD, /* Child pointer. */
    RTRB_THREAD /* Thread. */
};
/* A threaded binary search tree node. */
struct rtrb_node {
    struct rtrb_node *rtrb_link[2]; /* Subtrees. */
```

```
    void *rtrb_data; /* Pointer to data. */
    unsigned char rtrb_color; /* Color. */
    unsigned char rtrb_rtag; /* Tag field. */
};
```

This code is included in §452.

12.2 Operations

Most of the operations on RTRB trees can be borrowed from the corresponding operations on TBSTs, RTBSTs, or RTAVL trees, as shown below.

§455 ⟨ RTRB functions 455 ⟩ ≡

```
⟨ TBST creation function; tbst ⇒ rtrb 252 ⟩
⟨ RTBST search function; rtbst ⇒ rtrb 376 ⟩
⟨ RTRB item insertion function 456 ⟩
⟨ Table insertion convenience functions; tbl ⇒ rtrb 592 ⟩
⟨ RTRB item deletion function 468 ⟩
⟨ RTBST traversal functions; rtbst ⇒ rtrb 395 ⟩
⟨ RTAVL copy function; rtavl ⇒ rtrb; rtavl_balance ⇒ rtrb_color 447 ⟩
⟨ RTBST destruction function; rtbst ⇒ rtrb 407 ⟩
⟨ Default memory allocation functions; tbl ⇒ rtrb 6 ⟩
⟨ Table assertion functions; tbl ⇒ rtrb 594 ⟩
```

This code is included in §453.

12.3 Insertion

Insertion is, as usual, one of the operations that must be newly implemented for our new type of tree. There is nothing surprising in the function's outline:

§456 ⟨ RTRB item insertion function 456 ⟩ ≡

```
void **rtrb_probe (struct rtrb_table *tree, void *item) {
    struct rtrb_node *pa[RTRB_MAX_HEIGHT]; /* Nodes on stack. */
    unsigned char da[RTRB_MAX_HEIGHT]; /* Directions moved from stack nodes. */
    int k; /* Stack height. */

    struct rtrb_node *p; /* Current node in search. */
    struct rtrb_node *n; /* New node. */
    int dir; /* Side of p on which p is located. */

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search RTRB tree for insertion point 457 ⟩
    ⟨ Step 2: Insert RTRB node 458 ⟩
    ⟨ Step 3: Rebalance after RTRB insertion 459 ⟩

    return &n->rtrb_data;
}
```

This code is included in §455.

12.3.1 Steps 1 and 2: Search and Insert

The process of search and insertion proceeds as usual. Stack *pa*[], with $pa[k-1]$ at top of stack, records the parents of the node *p* currently under consideration, with corresponding stack *da*[] indicating the direction moved. We use the standard code for insertion into an RTBST. When the loop exits, *p* is the node under which a new node should be inserted on side *dir*.

§457 ⟨Step 1: Search RTRB tree for insertion point 457⟩ ≡

```
da[0] = 0;
pa[0] = (struct rtrb_node *) &tree→rtrb_root;
k = 1;
if (tree→rtrb_root != NULL)
  for (p = tree→rtrb_root; ; p = p→rtrb_link[dir]) {
    int cmp = tree→rtrb_compare (item, p→rtrb_data, tree→rtrb_param);
    if (cmp == 0)
      return &p→rtrb_data;

    pa[k] = p;
    da[k++] = dir = cmp > 0;

    if (dir == 0) {
      if (p→rtrb_link[0] == NULL)
        break;
    } else /* dir == 1 */ {
      if (p→rtrb_rtag == RTRB_THREAD)
        break;
    }
  }
else {
  p = (struct rtrb_node *) &tree→rtrb_root;
  dir = 0;
}
```

This code is included in §456.

§458 ⟨Step 2: Insert RTRB node 458⟩ ≡

```
n = tree→rtrb_alloc→libavl_malloc (tree→rtrb_alloc, sizeof *n);
if (n == NULL)
  return NULL;

tree→rtrb_count++;
n→rtrb_data = item;
n→rtrb_link[0] = NULL;
if (dir == 0) {
  if (tree→rtrb_root != NULL)
    n→rtrb_link[1] = p;
  else n→rtrb_link[1] = NULL;
} else /* dir == 1 */ {
  p→rtrb_rtag = RTRB_CHILD;
  n→rtrb_link[1] = p→rtrb_link[1];
}
```

```
n→rtrb_rtag = RTRB_THREAD;
n→rtrb_color = RTRB_RED;
p→rtrb_link[dir] = n;
```

This code is included in §456.

12.3.2 Step 3: Rebalance

The rebalancing outline follows ⟨ Step 3: Rebalance after RB insertion 201 ⟩.

§459 ⟨ Step 3: Rebalance after RTRB insertion 459 ⟩ ≡

```
while (k >= 3 && pa[k − 1]→rtrb_color == RTRB_RED) {
    if (da[k − 2] == 0)
        { ⟨ Left-side rebalancing after RTRB insertion 460 ⟩ }
    else { ⟨ Right-side rebalancing after RTRB insertion 461 ⟩ }
}
tree→rtrb_root→rtrb_color = RTRB_BLACK;
```

This code is included in §456.

The choice of case for insertion on the left side is made in the same way as in ⟨ Left-side rebalancing after RB insertion 202 ⟩, except that of course right-side tests for non-empty subtrees are made using *rtrb_rtag* instead of *rtrb_link*[1], and similarly for insertion on the right side. In short, we take q (which is not a real variable) as the new node n if this is the first time through the loop, or a node whose color has just been changed to red otherwise. We know that both q and its parent $pa[k-1]$ are red, violating rule 1 for red-black trees, and that q's grandparent $pa[k-2]$ is black. Here is the code to distinguish cases:

§460 ⟨ Left-side rebalancing after RTRB insertion 460 ⟩ ≡

```
struct rtrb_node *y = pa[k − 2]→rtrb_link[1];
if (pa[k − 2]→rtrb_rtag == RTRB_CHILD && y→rtrb_color == RTRB_RED)
    { ⟨ Case 1 in left-side RTRB insertion rebalancing 462 ⟩ }
else {
    struct rtrb_node *x;
    if (da[k − 1] == 0)
        y = pa[k − 1];
    else { ⟨ Case 3 in left-side RTRB insertion rebalancing 466 ⟩ }
    ⟨ Case 2 in left-side RTRB insertion rebalancing 464 ⟩
    break;
}
```

This code is included in §459.

§461 ⟨ Right-side rebalancing after RTRB insertion 461 ⟩ ≡

```
struct rtrb_node *y = pa[k − 2]→rtrb_link[0];
if (pa[k − 2]→rtrb_link[0] != NULL && y→rtrb_color == RTRB_RED)
    { ⟨ Case 1 in right-side RTRB insertion rebalancing 463 ⟩ }
else {
    struct rtrb_node *x;
    if (da[k − 1] == 1)
        y = pa[k − 1];
    else { ⟨ Case 3 in right-side RTRB insertion rebalancing 467 ⟩ }
```

⟨ Case 2 in right-side RTRB insertion rebalancing 465 ⟩
break;
}

This code is included in §459.

Case 1: *q*'s uncle is red

If node q's uncle is red, then no links need be changed. Instead, we will just recolor nodes. We reuse the code for RB insertion (see page 145):

§462 ⟨ Case 1 in left-side RTRB insertion rebalancing 462 ⟩ ≡
⟨ Case 1 in left-side RB insertion rebalancing; rb ⇒ rtrb 203 ⟩

This code is included in §460.

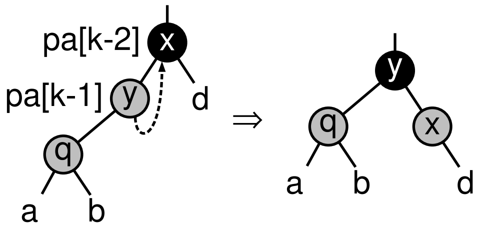
§463 ⟨ Case 1 in right-side RTRB insertion rebalancing 463 ⟩ ≡
⟨ Case 1 in right-side RB insertion rebalancing; rb ⇒ rtrb 207 ⟩

This code is included in §461.

Case 2: *q* is on same side of parent as parent is of grandparent

If q is a left child of its parent y and y is a left child of its own parent x, or if both q and y are right children, then we rotate at x away from y. This is the same that we would do in an unthreaded RB tree (see page 145).

However, as usual, we must make sure that threads are fixed up properly in the rotation. In particular, for case 2 in left-side rebalancing, we must convert a right thread of y, after rotation, into a null left child pointer of x, like this:

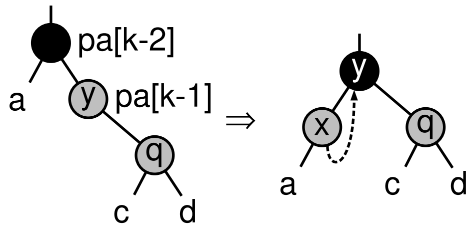


§464 ⟨ Case 2 in left-side RTRB insertion rebalancing 464 ⟩ ≡
⟨ Case 2 in left-side RB insertion rebalancing; rb ⇒ rtrb 204 ⟩

```
if (y→rtrb_rtag == RTRB_THREAD) {
    y→rtrb_rtag = RTRB_CHILD;
    x→rtrb_link[0] = NULL;
}
```

This code is included in §460.

For the right-side rebalancing case, we must convert a null left child of y, after rotation, into a right thread of x:



§465 ⟨ Case 2 in right-side RTRB insertion rebalancing 465 ⟩ ≡
⟨ Case 2 in right-side RB insertion rebalancing; rb ⇒ rtrb 208 ⟩

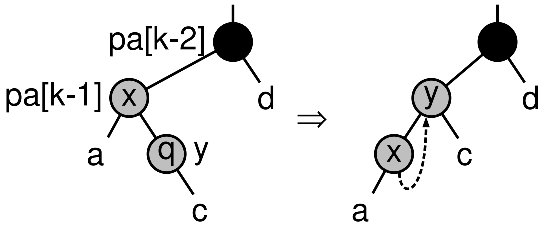
```
if (x→rtrb_link[1] == NULL) {
    x→rtrb_rtag = RTRB_THREAD;
    x→rtrb_link[1] = y;
}
```

This code is included in §461.

Case 3: *q* is on opposite side of parent as parent is of grandparent

If q is a left child and its parent is a right child, or vice versa, then we have an instance of case 3, and we rotate at q's parent in the direction from q to its parent. We handle this case as seen before for unthreaded RB trees (see page 146), with the addition of fix-ups for threads during rotation.

The left-side fix-up and the code to do it look like this:

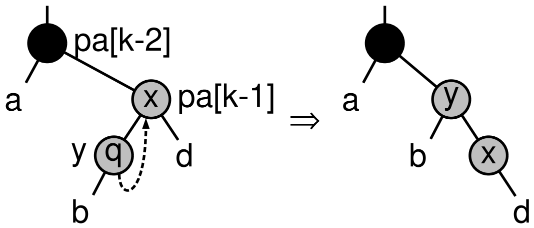


§466 ⟨ Case 3 in left-side RTRB insertion rebalancing 466 ⟩ ≡
⟨ Case 3 in left-side RB insertion rebalancing; rb ⇒ rtrb 205 ⟩

```
if (x→rtrb_link[1] == NULL) {
    x→rtrb_rtag = RTRB_THREAD;
    x→rtrb_link[1] = y;
}
```

This code is included in §460.

Here's the right-side fix-up and code:



§467 ⟨ Case 3 in right-side RTRB insertion rebalancing 467 ⟩ ≡
⟨ Case 3 in right-side RB insertion rebalancing; rb ⇒ rtrb 209 ⟩

```
if (y→rtrb_rtag == RTRB_THREAD) {
    y→rtrb_rtag = RTRB_CHILD;
    x→rtrb_link[0] = NULL;
}
```

This code is included in §461.

12.4 Deletion

The process of deletion from an RTRB tree is the same that we've seen many times now. Code for the first step is borrowed from RTAVL deletion:

§468 ⟨ RTRB item deletion function 468 ⟩ ≡

```
void *rtrb_delete (struct rtrb_table *tree, const void *item) {
    struct rtrb_node *pa[RTRB_MAX_HEIGHT]; /* Nodes on stack. */
    unsigned char da[RTRB_MAX_HEIGHT]; /* Directions moved from stack nodes. */
    int k; /* Stack height. */

    struct rtrb_node *p;

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search RTAVL tree for item to delete; rtavl ⇒ rtrb 430 ⟩
    ⟨ Step 2: Delete RTRB node 469 ⟩
    ⟨ Step 3: Rebalance after RTRB deletion 474 ⟩
    ⟨ Step 4: Finish up after RTRB deletion 481 ⟩
}
```

This code is included in §455.

12.4.1 Step 2: Delete

We use left-looking deletion. At this point, p is the node to delete. After the deletion, x is the node that replaced p, or a null pointer if the node was deleted without replacement. The cases are distinguished in the usual way:

§469 ⟨ Step 2: Delete RTRB node 469 ⟩ ≡

```
if (p→rtrb_link[0] == NULL) {
    if (p→rtrb_rtag == RTRB_CHILD)
        { ⟨ Case 1 in RTRB deletion 470 ⟩ }
    else { ⟨ Case 2 in RTRB deletion 471 ⟩ }
} else {
    enum rtrb_color t;
    struct rtrb_node *r = p→rtrb_link[0];

    if (r→rtrb_rtag == RTRB_THREAD)
        { ⟨ Case 3 in RTRB deletion 472 ⟩ }
    else { ⟨ Case 4 in RTRB deletion 473 ⟩ }
}
```

This code is included in §468.

Case 1: *p* has a right child but no left child

If p, the node to be deleted, has a right child but no left child, then we replace it by its right child. This is the same as ⟨ Case 1 in RTAVL deletion 432 ⟩.

§470 ⟨ Case 1 in RTRB deletion 470 ⟩ ≡
⟨ Case 1 in RTAVL deletion; rtavl ⇒ rtrb 432 ⟩

This code is included in §469.

Case 2: *p* has a right thread and no left child

Similarly, case 2 is the same as ⟨ Case 2 in RTAVL deletion 433 ⟩, with the addition of an assignment to x.

§471 ⟨ Case 2 in RTRB deletion 471 ⟩ ≡
⟨ Case 2 in RTAVL deletion; rtavl ⇒ rtrb 433 ⟩

This code is included in §469.

Case 3: *p*'s left child has a right thread

If p has a left child r, and r has a right thread, then we replace p by r and transfer p's former right link to r. Node r also receives p's balance factor.

§472 ⟨ Case 3 in RTRB deletion 472 ⟩ ≡

```
r->rtrb_link[1] = p->rtrb_link[1];
r->rtrb_rtag = p->rtrb_rtag;
t = r->rtrb_color;
r->rtrb_color = p->rtrb_color;
p->rtrb_color = t;
pa[k - 1]->rtrb_link[da[k - 1]] = r;
da[k] = 0;
pa[k++] = r;
```

This code is included in §469.

Case 4: *p*'s left child has a right child

The fourth case, where p has a left child that itself has a right child, uses the same algorithm as ⟨ Case 4 in RTAVL deletion 435 ⟩, except that instead of setting the balance factor of s, we swap the colors of t and s as in ⟨ Case 3 in RB deletion 224 ⟩.

§473 ⟨ Case 4 in RTRB deletion 473 ⟩ ≡

```
struct rtrb_node *s;
int j = k++;
for (;;) {
    da[k] = 1;
    pa[k++] = r;
    s = r->rtrb_link[1];
    if (s->rtrb_rtag == RTRB_THREAD)
        break;
    r = s;
```

```
}
da[j] = 0;
pa[j] = pa[j - 1]→rtrb_link[da[j - 1]] = s;
if (s→rtrb_link[0] != NULL)
    r→rtrb_link[1] = s→rtrb_link[0];
else {
    r→rtrb_rtag = RTRB_THREAD;
    r→rtrb_link[1] = s;
}
s→rtrb_link[0] = p→rtrb_link[0];
s→rtrb_link[1] = p→rtrb_link[1];
s→rtrb_rtag = p→rtrb_rtag;
t = s→rtrb_color;
s→rtrb_color = p→rtrb_color;
p→rtrb_color = t;
```

This code is included in §469.

12.4.2 Step 3: Rebalance

The rebalancing step's outline is much like that for deletion in a symmetrically threaded tree, except that we must check for a null child pointer on the left side of x versus a thread on the right side:

§474 ⟨ Step 3: Rebalance after RTRB deletion 474 ⟩ ≡

```
if (p→rtrb_color == RTRB_BLACK) {
    for (; k > 1; k−−) {
        struct rtrb_node *x;
        if (da[k - 1] == 0 || pa[k - 1]→rtrb_rtag == RTRB_CHILD)
            x = pa[k - 1]→rtrb_link[da[k - 1]];
        else x = NULL;
        if (x != NULL && x→rtrb_color == RTRB_RED) {
            x→rtrb_color = RTRB_BLACK;
            break;
        }
        if (da[k - 1] == 0)
            { ⟨ Left-side rebalancing after RTRB deletion 475 ⟩ }
        else { ⟨ Right-side rebalancing after RTRB deletion 476 ⟩ }
    }
    if (tree→rtrb_root != NULL) tree→rtrb_root→rtrb_color = RTRB_BLACK;
}
```

This code is included in §468.

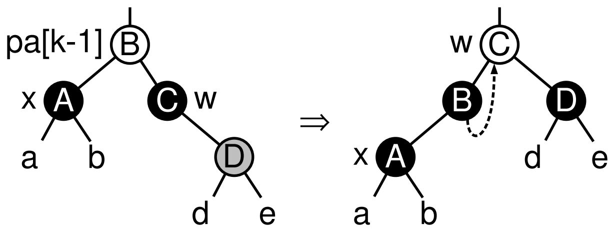
As for RTRB insertion, rebalancing on either side of the root is not symmetric because the tree structure itself is not symmetric, but again the rebalancing steps are very similar. The outlines of the left-side and right-side rebalancing code are below. The code for ensuring that w is black and for case 1 on each side are the same as the corresponding unthreaded RB code, because none of that code needs to check for empty trees:

§475
```
⟨Left-side rebalancing after RTRB deletion 475⟩ ≡
struct rtrb_node *w = pa[k - 1]→rtrb_link[1];
if (w→rtrb_color == RTRB_RED)
  { ⟨Ensure w is black in left-side RB deletion rebalancing; rb ⇒ rtrb 228⟩ }
if ((w→rtrb_link[0] == NULL || w→rtrb_link[0]→rtrb_color == RTRB_BLACK)
    && (w→rtrb_rtag == RTRB_THREAD || w→rtrb_link[1]→rtrb_color == RTRB_BLACK))
  { ⟨Case 1 in left-side RB deletion rebalancing; rb ⇒ rtrb 229⟩ }
else {
  if (w→rtrb_rtag == RTRB_THREAD || w→rtrb_link[1]→rtrb_color == RTRB_BLACK)
    { ⟨Transform left-side RTRB deletion rebalancing case 3 into case 2 479⟩ }
  ⟨Case 2 in left-side RTRB deletion rebalancing 477⟩
  break;
}
```
This code is included in §474.

§476
```
⟨Right-side rebalancing after RTRB deletion 476⟩ ≡
struct rtrb_node *w = pa[k - 1]→rtrb_link[0];
if (w→rtrb_color == RTRB_RED)
  { ⟨Ensure w is black in right-side RB deletion rebalancing; rb ⇒ rtrb 234⟩ }
if ((w→rtrb_link[0] == NULL || w→rtrb_link[0]→rtrb_color == RTRB_BLACK)
    && (w→rtrb_rtag == RTRB_THREAD || w→rtrb_link[1]→rtrb_color == RTRB_BLACK))
  { ⟨Case 1 in right-side RB deletion rebalancing; rb ⇒ rtrb 235⟩ }
else {
  if (w→rtrb_link[0] == NULL || w→rtrb_link[0]→rtrb_color == RTRB_BLACK)
    { ⟨Transform right-side RTRB deletion rebalancing case 3 into case 2 480⟩ }
  ⟨Case 2 in right-side RTRB deletion rebalancing 478⟩
  break;
}
```
This code is included in §474.

Case 2: *w*'s child opposite the deletion is red

If the deletion was on the left side of w and w's right child is red, we rotate left at $pa[k-1]$ and perform some recolorings, as we did for unthreaded RB trees (see page 157). There is a special case when w has no left child. This must be transformed into a thread from leading to w following the rotation:

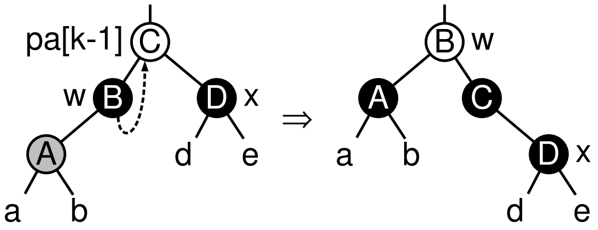


§477
```
⟨Case 2 in left-side RTRB deletion rebalancing 477⟩ ≡
⟨Case 2 in left-side RB deletion rebalancing; rb ⇒ rtrb 230⟩
if (w→rtrb_link[0]→rtrb_link[1] == NULL) {
```

```
    w→rtrb_link[0]→rtrb_rtag = RTRB_THREAD;
    w→rtrb_link[0]→rtrb_link[1] = w;
}
```

This code is included in §475.

Alternately, if the deletion was on the right side of w and w's left child is right, we rotate right at $pa[k-1]$ and recolor. There is an analogous special case:



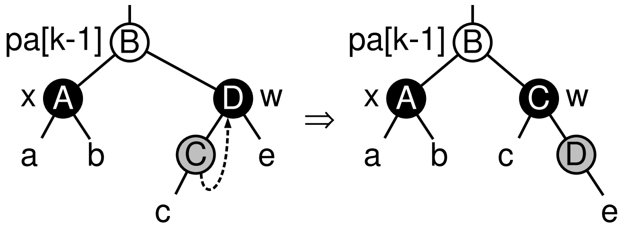
§478 ⟨ Case 2 in right-side RTRB deletion rebalancing 478 ⟩ ≡
⟨ Case 2 in right-side RB deletion rebalancing; rb ⇒ rtrb 237 ⟩

```
if (w→rtrb_rtag == RTRB_THREAD) {
    w→rtrb_rtag = RTRB_CHILD;
    pa[k − 1]→rtrb_link[0] = NULL;
}
```

This code is included in §476.

Case 3: *w*'s child on the side of the deletion is red

If the deletion was on the left side of w and w's left child is red, then we rotate right at w and recolor, as in case 3 for unthreaded RB trees (see page 157). There is a special case when w's left child has a right thread. This must be transformed into a null left child of w's right child following the rotation:

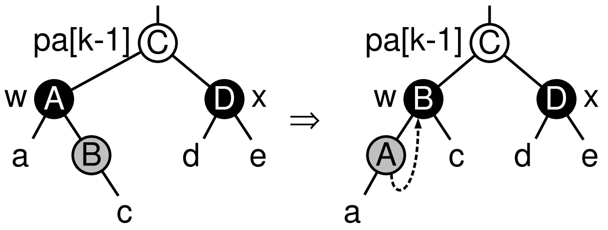


§479 ⟨ Transform left-side RTRB deletion rebalancing case 3 into case 2 479 ⟩ ≡
⟨ Transform left-side RB deletion rebalancing case 3 into case 2; rb ⇒ rtrb 231 ⟩

```
if (w→rtrb_rtag == RTRB_THREAD) {
    w→rtrb_rtag = RTRB_CHILD;
    w→rtrb_link[1]→rtrb_link[0] = NULL;
}
```

This code is included in §475.

Alternately, if the deletion was on the right side of w and w's right child is red, we rotate left at w and recolor. There is an analogous special case:



§480 ⟨ Transform right-side RTRB deletion rebalancing case 3 into case 2 480 ⟩ ≡
⟨ Transform right-side RB deletion rebalancing case 3 into case 2; rb ⇒ rtrb 236 ⟩

```
if (w→rtrb_link[0]→rtrb_link[1] == NULL) {
    w→rtrb_link[0]→rtrb_rtag = RTRB_THREAD;
    w→rtrb_link[0]→rtrb_link[1] = w;
}
```

This code is included in §476.

12.4.3 Step 4: Finish Up

§481 ⟨ Step 4: Finish up after RTRB deletion 481 ⟩ ≡

```
tree→rtrb_alloc→libavl_free (tree→rtrb_alloc, p);
return (void *) item;
```

This code is included in §468.

12.5 Testing

§482 ⟨ rtrb-test.c 482 ⟩ ≡

```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ limits.h ⟩
#include ⟨ stdio.h ⟩
#include "rtrb.h"
#include "test.h"
⟨ RTBST print function; rtbst ⇒ rtrb 412 ⟩
⟨ BST traverser check function; bst ⇒ rtrb 104 ⟩
⟨ Compare two RTRB trees for structure and content 483 ⟩
⟨ Recursively verify RTRB tree structure 484 ⟩
⟨ RB tree verify function; rb ⇒ rtrb 244 ⟩
⟨ BST test function; bst ⇒ rtrb 100 ⟩
⟨ BST overflow test function; bst ⇒ rtrb 122 ⟩
```

§483 ⟨ Compare two RTRB trees for structure and content 483 ⟩ ≡

```
static int compare_trees (struct rtrb_node *a, struct rtrb_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
        if (a != NULL || b != NULL) {
            printf ("␣a=%d␣b=%d\n",
                    a ? *(int *) a→rtrb_data : -1, b ? *(int *) b→rtrb_data : -1);
            assert (0);
        }
```

```
        return 1;
    }
    assert (a != b);
    if (*(int *) a->rtrb_data != *(int *) b->rtrb_data
        || a->rtrb_rtag != b->rtrb_rtag
        || a->rtrb_color != b->rtrb_color) {
        printf (" Copied nodes differ: a=%d%c b=%d%c a:",
                *(int *) a->rtrb_data, a->rtrb_color == RTRB_RED ? 'r' : 'b',
                *(int *) b->rtrb_data, b->rtrb_color == RTRB_RED ? 'r' : 'b');
        if (a->rtrb_rtag == RTRB_CHILD) printf ("r");
        printf (" b:");
        if (b->rtrb_rtag == RTRB_CHILD) printf ("r");
        printf ("\n");
        return 0;
    }
    if (a->rtrb_rtag == RTRB_THREAD)
        assert ((a->rtrb_link[1] == NULL) != (a->rtrb_link[1] != b->rtrb_link[1]));
    okay = compare_trees (a->rtrb_link[0], b->rtrb_link[0]);
    if (a->rtrb_rtag == RTRB_CHILD)
        okay &= compare_trees (a->rtrb_link[1], b->rtrb_link[1]);
    return okay;
}
```

This code is included in §482.

§484 ⟨ Recursively verify RTRB tree structure 484 ⟩ ≡

```
static void recurse_verify_tree (struct rtrb_node *node, int *okay, size_t *count,
                                 int min, int max, int *bh) {
    int d; /* Value of this node's data. */
    size_t subcount[2]; /* Number of nodes in subtrees. */
    int subbh[2]; /* Black-heights of subtrees. */
    if (node == NULL) {
        *count = 0;
        *bh = 0;
        return;
    }
    d = *(int *) node->rtrb_data;
    ⟨ Verify binary search tree ordering 114 ⟩
    subcount[0] = subcount[1] = 0;
    subbh[0] = subbh[1] = 0;
    recurse_verify_tree (node->rtrb_link[0], okay, &subcount[0],
                         min, d - 1, &subbh[0]);
    if (node->rtrb_rtag == RTRB_CHILD)
        recurse_verify_tree (node->rtrb_link[1], okay, &subcount[1],
                             d + 1, max, &subbh[1]);
    *count = 1 + subcount[0] + subcount[1];
```

```
    *bh = (node->rtrb_color == RTRB_BLACK) + subbh[0];
    ⟨ Verify RB node color; rb ⇒ rtrb 241 ⟩
    ⟨ Verify RTRB node rule 1 compliance 485 ⟩
    ⟨ Verify RB node rule 2 compliance; rb ⇒ rtrb 243 ⟩
}
```

This code is included in §482.

§485 ⟨ Verify RTRB node rule 1 compliance 485 ⟩ ≡

```
/* Verify compliance with rule 1. */
if (node->rtrb_color == RTRB_RED) {
    if (node->rtrb_link[0] != NULL && node->rtrb_link[0]->rtrb_color == RTRB_RED) {
        printf (" Red node %d has red left child %d\n",
                d, *(int *) node->rtrb_link[0]->rtrb_data);
        *okay = 0;
    }
    if (node->rtrb_rtag == RTRB_CHILD && node->rtrb_link[1]->rtrb_color == RTRB_RED) {
        printf (" Red node %d has red right child %d\n",
                d, *(int *) node->rtrb_link[1]->rtrb_data);
        *okay = 0;
    }
}
```

This code is included in §484.

13 BSTs with Parent Pointers

The preceding six chapters introduced two different forms of threaded trees, which simplified traversal by eliminating the need for a stack. There is another way to accomplish the same purpose: add to each node a **parent pointer**, a link from the node to its parent. A binary search tree so augmented is called a BST with parent pointers, or PBST for short.[1] In this chapter, we show how to add parent pointers to binary trees. The next two chapters will add them to AVL trees and red-black trees.

Parent pointers and threads have equivalent power. That is, given a node within a threaded tree, we can find the node's parent, and given a node within a tree with parent pointers, we can determine the targets of any threads that the node would have in a similar threaded tree.

Parent pointers have some advantages over threads. In particular, parent pointers let us more efficiently eliminate the stack for insertion and deletion in balanced trees. Rebalancing during these operations requires us to locate the parents of nodes. In our implementations of threaded balanced trees, we wrote code to do this, but it took a relatively complicated and slow helper function. Parent pointers make it much faster and easier. It is also easier to search a tree with parent pointers than a threaded tree, because there is no need to check tags. Outside of purely technical issues, many people find the use of parent pointers more intuitive than threads.

On the other hand, to traverse a tree with parent pointers in inorder we may have to follow several parent pointers instead of a single thread. What's more, parent pointers take extra space for a third pointer field in every node, whereas the tag fields in threaded balanced trees often fit into node structures without taking up additional room (see Exercise 8.1-1). Finally, maintaining parent pointers on insertion and deletion takes time. In fact, we'll see that it takes more operations (and thus, all else being equal, time) than maintaining threads.

In conclusion, a general comparison of parent pointers with threads reveals no clear winner. Further discussion of the merits of parent pointers versus those of threads will be postponed until later in this book. For now, we'll stick to the problems of parent pointer implementation.

Here's the outline of the PBST code. We're using the prefix *pbst_* this time:

§486 ⟨ `pbst.h` 486 ⟩ ≡
⟨ License 1 ⟩
#ifndef `PBST_H`
#define `PBST_H` 1
#include ⟨ stddef.h ⟩
⟨ Table types; tbl ⇒ pbst 14 ⟩
⟨ TBST table structure; tbst ⇒ pbst 250 ⟩
⟨ PBST node structure 488 ⟩
⟨ TBST traverser structure; tbst ⇒ pbst 267 ⟩
⟨ Table function prototypes; tbl ⇒ pbst 15 ⟩

[1] This abbreviation might be thought of as expanding to "parented BST" or "parental BST", but those are not proper terms.

⟨ BST extra function prototypes; bst ⇒ pbst 88 ⟩
#**endif** /* pbst.h */

§487 ⟨ `pbst.c` 487 ⟩ ≡
⟨ License 1 ⟩
#**include** ⟨ assert.h ⟩
#**include** ⟨ stdio.h ⟩
#**include** ⟨ stdlib.h ⟩
#**include** "pbst.h"
⟨ PBST functions 489 ⟩

13.1 Data Types

For PBSTs we reuse TBST table and traverser structures. In fact, the only data type that needs revision is the node structure. We take the basic form of a node and add a member *pbst_parent* to point to its parent node:

§488 ⟨ PBST node structure 488 ⟩ ≡

```
/* A binary search tree with parent pointers node. */
struct pbst_node {
    struct pbst_node *pbst_link[2]; /* Subtrees. */
    struct pbst_node *pbst_parent; /* Parent. */
    void *pbst_data; /* Pointer to data. */
};
```

This code is included in §486.

There is one special case: what should be the value of *pbst_parent* for a node that has no parent, that is, in the tree's root? There are two reasonable choices.

First, *pbst_parent* could be `NULL` in the root. This makes it easy to check whether a node is the tree's root. On the other hand, we often follow a parent pointer in order to change the link down from the parent, and `NULL` as the root node's *pbst_parent* requires a special case.

We can eliminate this special case if the root's *pbst_parent* is the tree's pseudo-root node, that is, (**struct pbst_node** *) &*tree*→*pbst_root*. The downside of this choice is that it becomes uglier, and perhaps slower, to check whether a node is the tree's root, because a comparison must be made against a non-constant expression instead of simply `NULL`.

In this book, we make the former choice, so *pbst_parent* is `NULL` in the tree's root node.

See also: [Cormen 1990], section 11.4.

13.2 Operations

When we added parent pointers to BST nodes, we did not change the interpretation of any of the node members. This means that any function that examines PBSTs without modifying them will work without change. We take advantage of that for tree search. We also get away with it for destruction, since there's no problem with failing to update parent pointers in that case. Although we could, technically, do the same for traversal, that would negate much of the advantage of parent pointers, so we reimplement them. Here is the overall outline:

§489 ⟨ PBST functions 489 ⟩ ≡
⟨ TBST creation function; tbst ⇒ pbst 252 ⟩
⟨ BST search function; bst ⇒ pbst 31 ⟩
⟨ PBST item insertion function 490 ⟩
⟨ Table insertion convenience functions; tbl ⇒ pbst 592 ⟩
⟨ PBST item deletion function 493 ⟩
⟨ PBST traversal functions 502 ⟩
⟨ PBST copy function 509 ⟩
⟨ BST destruction function; bst ⇒ pbst 84 ⟩
⟨ PBST balance function 511 ⟩
⟨ Default memory allocation functions; tbl ⇒ pbst 6 ⟩
⟨ Table assertion functions; tbl ⇒ pbst 594 ⟩

This code is included in §487.

13.3 Insertion

The only difference between this code and ⟨ BST item insertion function 32 ⟩ is that we set *n*'s parent pointer after insertion.

§490 ⟨ PBST item insertion function 490 ⟩ ≡

```
void **pbst_probe (struct pbst_table *tree, void *item) {
    struct pbst_node *p, *q; /* Current node in search and its parent. */
    int dir; /* Side of q on which p is located. */
    struct pbst_node *n; /* Newly inserted node. */
    assert (tree != NULL && item != NULL);
    ⟨ Step 1: Search PBST tree for insertion point 491 ⟩
    ⟨ Step 2: Insert PBST node 492 ⟩
    return &n->pbst_data;
}
```

This code is included in §489.

§491 ⟨ Step 1: Search PBST tree for insertion point 491 ⟩ ≡

```
for (q = NULL, p = tree->pbst_root; p != NULL; q = p, p = p->pbst_link[dir]) {
    int cmp = tree->pbst_compare (item, p->pbst_data, tree->pbst_param);
    if (cmp == 0)
        return &p->pbst_data;
    dir = cmp > 0;
}
```

This code is included in §490 and §555.

§492 ⟨ Step 2: Insert PBST node 492 ⟩ ≡

```
n = tree->pbst_alloc->libavl_malloc (tree->pbst_alloc, sizeof *p);
if (n == NULL)
    return NULL;
tree->pbst_count++;
n->pbst_link[0] = n->pbst_link[1] = NULL;
n->pbst_parent = q;
n->pbst_data = item;
```

```
if (q != NULL)
    q->pbst_link[dir] = n;
else tree->pbst_root = n;
```

This code is included in §490, §525, and §556.

See also: [Cormen 1990], section 13.3.

13.4 Deletion

The new aspect of deletion in a PBST is that we must properly adjust parent pointers. The outline is the same as usual:

§493 ⟨ PBST item deletion function 493 ⟩ ≡
```
void *pbst_delete (struct pbst_table *tree, const void *item) {
    struct pbst_node *p; /* Traverses tree to find node to delete. */
    struct pbst_node *q; /* Parent of p. */
    int dir; /* Side of q on which p is linked. */

    assert (tree != NULL && item != NULL);

    ⟨Step 1: Find PBST node to delete 494⟩
    ⟨Step 2: Delete PBST node 496⟩
    ⟨Step 3: Finish up after deleting PBST node 501⟩
}
```

This code is included in §489.

We find the node to delete by using p to search for *item*. For the first time in implementing a deletion routine, we do not keep track of the current node's parent, because we can always find it out later with little effort:

§494 ⟨ Step 1: Find PBST node to delete 494 ⟩ ≡
```
if (tree->pbst_root == NULL)
    return NULL;

p = tree->pbst_root;
for (;;) {
    int cmp = tree->pbst_compare (item, p->pbst_data, tree->pbst_param);
    if (cmp == 0)
        break;

    dir = cmp > 0;
    p = p->pbst_link[dir];
    if (p == NULL)
        return NULL;
}
item = p->pbst_data;
```

See also §495.

This code is included in §493, §534, and §566.

Now we've found the node to delete, p. The first step in deletion is to find the parent of p as q. Node p is q's child on side *dir*. Deletion of the root is a special case:

§495 ⟨ Step 1: Find PBST node to delete 494 ⟩ +≡
```
q = p->pbst_parent;
```

```
if (q == NULL) {
    q = (struct pbst_node *) &tree→pbst_root;
    dir = 0;
}
```

The remainder of the deletion follows the usual outline:

§496 ⟨ Step 2: Delete PBST node 496 ⟩ ≡

```
if (p→pbst_link[1] == NULL)
    { ⟨ Case 1 in PBST deletion 497 ⟩ }
else {
    struct pbst_node *r = p→pbst_link[1];
    if (r→pbst_link[0] == NULL)
        { ⟨ Case 2 in PBST deletion 498 ⟩ }
    else { ⟨ Case 3 in PBST deletion 499 ⟩ }
}
```

This code is included in §493.

Case 1: *p* has no right child

If p has no right child, then we can replace it by its left child, if any. If p does have a left child then we must update its parent to be p's former parent.

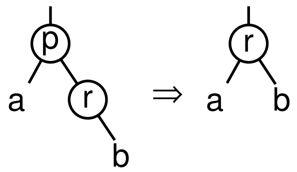
§497 ⟨ Case 1 in PBST deletion 497 ⟩ ≡

```
q→pbst_link[dir] = p→pbst_link[0];
if (q→pbst_link[dir] != NULL)
    q→pbst_link[dir]→pbst_parent = p→pbst_parent;
```

This code is included in §496, §536, and §568.

Case 2: *p*'s right child has no left child

When we delete a node with a right child that in turn has no left child, the operation looks like this:



The key points to notice are that node r's parent changes and so does the parent of r's new left child, if there is one. We update these in deletion:

§498 ⟨ Case 2 in PBST deletion 498 ⟩ ≡

```
r→pbst_link[0] = p→pbst_link[0];
q→pbst_link[dir] = r;
r→pbst_parent = p→pbst_parent;
if (r→pbst_link[0] != NULL)
    r→pbst_link[0]→pbst_parent = r;
```

This code is included in §496, §537, and §569.

Case 3: *p*'s right child has a left child

If p's right child has a left child, then we replace p by its successor, as usual. Finding the successor s and its parent r is a little simpler than usual, because we can move up the tree so easily. We know that s has a non-null parent so there is no need to handle that special case:

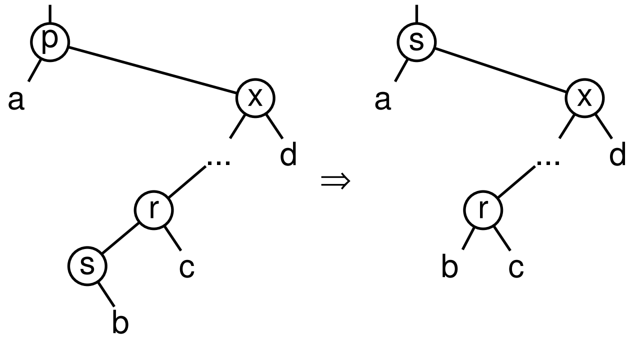
§499 ⟨ Case 3 in PBST deletion 499 ⟩ ≡

```
struct pbst_node *s = r->pbst_link[0];
while (s->pbst_link[0] != NULL)
  s = s->pbst_link[0];
r = s->pbst_parent;
```

See also §500.

This code is included in §496, §538, and §570.

The only other change here is that we must update parent pointers. It is easy to pick out the ones that must be changed by looking at a diagram of the deletion:



Node s's parent changes, as do the parents of its new right child x and, if it has one, its left child a. Perhaps less obviously, if s originally had a right child, it becomes the new left child of r, so its new parent is r:

§500 ⟨ Case 3 in PBST deletion 499 ⟩ +≡

```
r->pbst_link[0] = s->pbst_link[1];
s->pbst_link[0] = p->pbst_link[0];
s->pbst_link[1] = p->pbst_link[1];
q->pbst_link[dir] = s;
if (s->pbst_link[0] != NULL)
  s->pbst_link[0]->pbst_parent = s;
s->pbst_link[1]->pbst_parent = s;
s->pbst_parent = p->pbst_parent;
if (r->pbst_link[0] != NULL)
  r->pbst_link[0]->pbst_parent = r;
```

Finally, we free the deleted node p and return its data:

§501 ⟨ Step 3: Finish up after deleting PBST node 501 ⟩ ≡

```
tree->pbst_alloc->libavl_free (tree->pbst_alloc, p);
tree->pbst_count--;
return (void *) item;
```

This code is included in §493.

See also: [Cormen 1990], section 13.3.

Exercises:

1. In case 1, can we change the right side of the assignment in the **if** statement's consequent from *p→pbst_parent* to *q*?

13.5 Traversal

The traverser for a PBST is just like that for a TBST, so we can reuse a couple of the TBST functions. Besides that and a couple of completely generic functions, we have to reimplement the traversal functions.

§502 ⟨ PBST traversal functions 502 ⟩ ≡
⟨ TBST traverser null initializer; tbst ⇒ pbst 269 ⟩
⟨ PBST traverser first initializer 503 ⟩
⟨ PBST traverser last initializer 504 ⟩
⟨ PBST traverser search initializer 505 ⟩
⟨ PBST traverser insertion initializer 506 ⟩
⟨ TBST traverser copy initializer; tbst ⇒ pbst 274 ⟩
⟨ PBST traverser advance function 507 ⟩
⟨ PBST traverser back up function 508 ⟩
⟨ BST traverser current item function; bst ⇒ pbst 74 ⟩
⟨ BST traverser replacement function; bst ⇒ pbst 75 ⟩

This code is included in §489.

13.5.1 Starting at the First Node

Finding the smallest node in the tree is just a matter of starting from the root and descending as far to the left as we can.

§503 ⟨ PBST traverser first initializer 503 ⟩ ≡

```
void *pbst_t_first (struct pbst_traverser *trav, struct pbst_table *tree) {
    assert (tree != NULL && trav != NULL);
    trav→pbst_table = tree;
    trav→pbst_node = tree→pbst_root;
    if (trav→pbst_node != NULL) {
        while (trav→pbst_node→pbst_link[0] != NULL)
            trav→pbst_node = trav→pbst_node→pbst_link[0];
        return trav→pbst_node→pbst_data;
    }
    else return NULL;
}
```

This code is included in §502 and §546.

13.5.2 Starting at the Last Node

This is the same as starting from the least item, except that we descend to the right.

§504 ⟨ PBST traverser last initializer 504 ⟩ ≡

```
void *pbst_t_last (struct pbst_traverser *trav, struct pbst_table *tree) {
  assert (tree != NULL && trav != NULL);
  trav->pbst_table = tree;
  trav->pbst_node = tree->pbst_root;
  if (trav->pbst_node != NULL) {
    while (trav->pbst_node->pbst_link[1] != NULL)
      trav->pbst_node = trav->pbst_node->pbst_link[1];
    return trav->pbst_node->pbst_data;
  }
  else return NULL;
}
```

This code is included in §502 and §546.

13.5.3 Starting at a Found Node

To start from a particular item, we search for it in the tree. If it exists then we initialize the traverser to it. Otherwise, we initialize the traverser to the null item and return a null pointer. There are no surprises here.

§505 ⟨ PBST traverser search initializer 505 ⟩ ≡

```
void *pbst_t_find (struct pbst_traverser *trav, struct pbst_table *tree, void *item) {
  struct pbst_node *p;
  int dir;
  assert (trav != NULL && tree != NULL && item != NULL);
  trav->pbst_table = tree;
  for (p = tree->pbst_root; p != NULL; p = p->pbst_link[dir]) {
    int cmp = tree->pbst_compare (item, p->pbst_data, tree->pbst_param);
    if (cmp == 0) {
      trav->pbst_node = p;
      return p->pbst_data;
    }
    dir = cmp > 0;
  }
  trav->pbst_node = NULL;
  return NULL;
}
```

This code is included in §502 and §546.

13.5.4 Starting at an Inserted Node

This function combines the functionality of search and insertion with initialization of a traverser.

§506 ⟨ PBST traverser insertion initializer 506 ⟩ ≡

```
void *pbst_t_insert (struct pbst_traverser *trav, struct pbst_table *tree, void *item) {
  struct pbst_node *p, *q; /* Current node in search and its parent. */
  int dir; /* Side of q on which p is located. */
```

```
    struct pbst_node *n; /* Newly inserted node. */
    assert (trav != NULL && tree != NULL && item != NULL);
    trav->pbst_table = tree;
    for (q = NULL, p = tree->pbst_root; p != NULL; q = p, p = p->pbst_link[dir]) {
        int cmp = tree->pbst_compare (item, p->pbst_data, tree->pbst_param);
        if (cmp == 0) {
            trav->pbst_node = p;
            return p->pbst_data;
        }
        dir = cmp > 0;
    }
    trav->pbst_node = n = tree->pbst_alloc->libavl_malloc (tree->pbst_alloc, sizeof *p);
    if (n == NULL) return NULL;
    tree->pbst_count++;
    n->pbst_link[0] = n->pbst_link[1] = NULL;
    n->pbst_parent = q;
    n->pbst_data = item;
    if (q != NULL)
        q->pbst_link[dir] = n;
    else tree->pbst_root = n;
    return item;
}
```

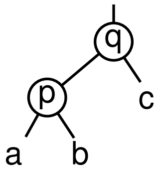
This code is included in §502.

13.5.5 Advancing to the Next Node

There are the same three cases for advancing a traverser as the other types of binary trees that we've already looked at. Two of the cases, the ones where we're starting from the null item or a node that has a right child, are unchanged.

The third case, where the node that we're starting from has no right child, is the case that must be revised. We can use the same algorithm that we did for ordinary BSTs without threads or parent pointers, described earlier (see Section 4.9.3 [Better Iterative Traversal], page 53). Simply put, we move upward in the tree until we move up to the right (or until we move off the top of the tree).

The code uses *q* to move up the tree and *p* as *q*'s child, so the termination condition is when *p* is *q*'s left child or *q* becomes a null pointer. There is a non-null successor in the former case, where the situation looks like this:



§507 ⟨PBST traverser advance function 507⟩ ≡

```
void *pbst_t_next (struct pbst_traverser *trav) {
```

```
    assert (trav != NULL);

    if (trav->pbst_node == NULL)
        return pbst_t_first (trav, trav->pbst_table);
    else if (trav->pbst_node->pbst_link[1] == NULL) {
        struct pbst_node *q, *p; /* Current node and its child. */
        for (p = trav->pbst_node, q = p->pbst_parent; ; p = q, q = q->pbst_parent)
            if (q == NULL || p == q->pbst_link[0]) {
                trav->pbst_node = q;
                return trav->pbst_node != NULL ? trav->pbst_node->pbst_data : NULL;
            }
    } else {
        trav->pbst_node = trav->pbst_node->pbst_link[1];
        while (trav->pbst_node->pbst_link[0] != NULL)
            trav->pbst_node = trav->pbst_node->pbst_link[0];
        return trav->pbst_node->pbst_data;
    }
}
```

This code is included in §502 and §546.

See also: [Cormen 1990], section 13.2.

13.5.6 Backing Up to the Previous Node

This is the same as advancing a traverser, except that we reverse the directions.

§508 ⟨PBST traverser back up function 508⟩ ≡

```
void *pbst_t_prev (struct pbst_traverser *trav) {
    assert (trav != NULL);

    if (trav->pbst_node == NULL)
        return pbst_t_last (trav, trav->pbst_table);
    else if (trav->pbst_node->pbst_link[0] == NULL) {
        struct pbst_node *q, *p; /* Current node and its child. */
        for (p = trav->pbst_node, q = p->pbst_parent; ; p = q, q = q->pbst_parent)
            if (q == NULL || p == q->pbst_link[1]) {
                trav->pbst_node = q;
                return trav->pbst_node != NULL ? trav->pbst_node->pbst_data : NULL;
            }
    } else {
        trav->pbst_node = trav->pbst_node->pbst_link[0];
        while (trav->pbst_node->pbst_link[1] != NULL)
            trav->pbst_node = trav->pbst_node->pbst_link[1];
        return trav->pbst_node->pbst_data;
    }
}
```

This code is included in §502 and §546.

See also: [Cormen 1990], section 13.2.

13.6 Copying

To copy BSTs with parent pointers, we use a simple adaptation of our original algorithm for copying BSTs, as implemented in ⟨ BST copy function 83 ⟩. That function used a stack to keep track of the nodes that need to be revisited to have their right subtrees copies. We can eliminate that by using the parent pointers. Instead of popping a pair of nodes off the stack, we ascend the tree until we moved up to the left:

§509 ⟨ PBST copy function 509 ⟩ ≡

```
⟨ PBST copy error helper function 510 ⟩
struct pbst_table *pbst_copy (const struct pbst_table *org, pbst_copy_func *copy,
                              pbst_item_func *destroy, struct libavl_allocator *allocator) {
  struct pbst_table *new;
  const struct pbst_node *x;
  struct pbst_node *y;
  assert (org != NULL);
  new = pbst_create (org→pbst_compare, org→pbst_param,
                     allocator != NULL ? allocator : org→pbst_alloc);
  if (new == NULL)
    return NULL;
  new→pbst_count = org→pbst_count;
  if (new→pbst_count == 0)
    return new;
  x = (const struct pbst_node *) &org→pbst_root;
  y = (struct pbst_node *) &new→pbst_root;
  for (;;) {
    while (x→pbst_link[0] != NULL) {
      y→pbst_link[0] = new→pbst_alloc→libavl_malloc (new→pbst_alloc,
                         sizeof *y→pbst_link[0]);
      if (y→pbst_link[0] == NULL) {
        if (y != (struct pbst_node *) &new→pbst_root) {
          y→pbst_data = NULL;
          y→pbst_link[1] = NULL;
        }
        copy_error_recovery (y, new, destroy);
        return NULL;
      }
      y→pbst_link[0]→pbst_parent = y;
      x = x→pbst_link[0];
      y = y→pbst_link[0];
    }
    y→pbst_link[0] = NULL;
    for (;;) {
      if (copy == NULL)
        y→pbst_data = x→pbst_data;
      else {
        y→pbst_data = copy (x→pbst_data, org→pbst_param);
```

```
          if (y->pbst_data == NULL) {
            y->pbst_link[1] = NULL;
            copy_error_recovery (y, new, destroy);
            return NULL;
          }
        }
        if (x->pbst_link[1] != NULL) {
          y->pbst_link[1] = new->pbst_alloc->libavl_malloc (new->pbst_alloc,
                        sizeof *y->pbst_link[1]);
          if (y->pbst_link[1] == NULL) {
            copy_error_recovery (y, new, destroy);
            return NULL;
          }
          y->pbst_link[1]->pbst_parent = y;
          x = x->pbst_link[1];
          y = y->pbst_link[1];
          break;
        }
        else y->pbst_link[1] = NULL;
        for (;;) {
          const struct pbst_node *w = x;
          x = x->pbst_parent;
          if (x == NULL) {
            new->pbst_root->pbst_parent = NULL;
            return new;
          }
          y = y->pbst_parent;
          if (w == x->pbst_link[0])
            break;
        }
      }
    }
}
```

This code is included in §489.

Recovering from an error changes in the same way. We ascend from the node where we were copying when memory ran out and set the right children of the nodes where we ascended to the right to null pointers, then destroy the fixed-up tree:

§510 ⟨ PBST copy error helper function 510 ⟩ ≡

```
static void copy_error_recovery (struct pbst_node *q,
                                 struct pbst_table *new, pbst_item_func *destroy) {
  assert (q != NULL && new != NULL);
  for (;;) {
    struct pbst_node *p = q;
    q = q->pbst_parent;
    if (q == NULL)
```

```
            break;
        if (p == q→pbst_link[0])
            q→pbst_link[1] = NULL;
    }
    pbst_destroy (new, destroy);
}
```

This code is included in §509 and §547.

13.7 Balance

We can balance a PBST in the same way that we would balance a BST without parent pointers. In fact, we'll use the same code, with the only change omitting only the maximum height check. This code doesn't set parent pointers, so afterward we traverse the tree to take care of that.

Here are the pieces of the core code that need to be repeated:

§511 ⟨ PBST balance function 511 ⟩ ≡

```
⟨ BST to vine function; bst ⇒ pbst 89 ⟩
⟨ Vine to balanced PBST function 512 ⟩
⟨ Update parent pointers function 514 ⟩
void pbst_balance (struct pbst_table *tree) {
    assert (tree != NULL);
    tree_to_vine (tree);
    vine_to_tree (tree);
    update_parents (tree);
}
```

This code is included in §489.

§512 ⟨ Vine to balanced PBST function 512 ⟩ ≡

```
⟨ BST compression function; bst ⇒ pbst 95 ⟩
static void vine_to_tree (struct pbst_table *tree) {
    unsigned long vine; /* Number of nodes in main vine. */
    unsigned long leaves; /* Nodes in incomplete bottom level, if any. */
    int height; /* Height of produced balanced tree. */
    ⟨ Calculate leaves; bst ⇒ pbst 91 ⟩
    ⟨ Reduce vine general case to special case; bst ⇒ pbst 92 ⟩
    ⟨ Make special case vine into balanced tree and count height; bst ⇒ pbst 93 ⟩
}
```

This code is included in §511.

§513 ⟨ PBST extra function prototypes 513 ⟩ ≡

```
/* Special PBST functions. */
void pbst_balance (struct pbst_table *tree);
```

Updating Parent Pointers

The procedure for rebalancing a binary tree leaves the nodes' parent pointers pointing every which way. Now we'll fix them. Incidentally, this is a general procedure, so the same code could be used in other situations where we have a tree to which we want to add parent pointers.

The procedure takes the same form as an inorder traversal, except that there is nothing to do in the place where we would normally visit the node. Instead, every time we move down to the left or the right, we set the parent pointer of the node we move to.

The code is straightforward enough. The basic strategy is to always move down to the left when possible; otherwise, move down to the right if possible; otherwise, repeatedly move up until we've moved up to the left to arrive at a node with a right child, then move to that right child.

§514 ⟨ Update parent pointers function 514 ⟩ ≡
```
static void update_parents (struct pbst_table *tree) {
  struct pbst_node *p;
  if (tree→pbst_root == NULL)
    return;
  tree→pbst_root→pbst_parent = NULL;
  for (p = tree→pbst_root; ; p = p→pbst_link[1]) {
    for (; p→pbst_link[0] != NULL; p = p→pbst_link[0])
      p→pbst_link[0]→pbst_parent = p;
    for (; p→pbst_link[1] == NULL; p = p→pbst_parent) {
      for (;;) {
        if (p→pbst_parent == NULL)
          return;
        if (p == p→pbst_parent→pbst_link[0])
          break;
        p = p→pbst_parent;
      }
    }
    p→pbst_link[1]→pbst_parent = p;
  }
}
```
This code is included in §511.

Exercises:

1. There is another approach to updating parent pointers: we can do it during the compressions. Implement this approach. Make sure not to miss any pointers.

13.8 Testing

§515 ⟨ `pbst-test.c` 515 ⟩ ≡
```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ limits.h ⟩
#include ⟨ stdio.h ⟩
#include "pbst.h"
#include "test.h"
```

⟨ BST print function; bst ⇒ pbst 119 ⟩
⟨ BST traverser check function; bst ⇒ pbst 104 ⟩
⟨ Compare two PBSTs for structure and content 516 ⟩
⟨ Recursively verify PBST structure 517 ⟩
⟨ BST verify function; bst ⇒ pbst 109 ⟩
⟨ TBST test function; tbst ⇒ pbst 295 ⟩
⟨ BST overflow test function; bst ⇒ pbst 122 ⟩

§516 ⟨ Compare two PBSTs for structure and content 516 ⟩ ≡

```
static int compare_trees (struct pbst_node *a, struct pbst_node *b) {
  int okay;
  if (a == NULL || b == NULL) {
    assert (a == NULL && b == NULL);
    return 1;
  }
  if (*(int *) a->pbst_data != *(int *) b->pbst_data
      || ((a->pbst_link[0] != NULL) != (b->pbst_link[0] != NULL))
      || ((a->pbst_link[1] != NULL) != (b->pbst_link[1] != NULL))
      || ((a->pbst_parent != NULL) != (b->pbst_parent != NULL))
      || (a->pbst_parent != NULL && b->pbst_parent != NULL
        && a->pbst_parent->pbst_data != b->pbst_parent->pbst_data)) {
    printf (" Copied nodes differ:\n"
            "  a: %d, parent %d, %s left child, %s right child\n"
            "  b: %d, parent %d, %s left child, %s right child\n",
            *(int *) a->pbst_data,
            a->pbst_parent != NULL ? *(int *) a->pbst_parent : -1,
            a->pbst_link[0] != NULL ? "has" : "no",
            a->pbst_link[1] != NULL ? "has" : "no",
            *(int *) b->pbst_data,
            b->pbst_parent != NULL ? *(int *) b->pbst_parent : -1,
            b->pbst_link[0] != NULL ? "has" : "no",
            b->pbst_link[1] != NULL ? "has" : "no");
    return 0;
  }
  okay = 1;
  if (a->pbst_link[0] != NULL)
    okay &= compare_trees (a->pbst_link[0], b->pbst_link[0]);
  if (a->pbst_link[1] != NULL)
    okay &= compare_trees (a->pbst_link[1], b->pbst_link[1]);
  return okay;
}
```

This code is included in §515.

§517 ⟨ Recursively verify PBST structure 517 ⟩ ≡

```
static void recurse_verify_tree (struct pbst_node *node, int *okay, size_t *count,
                                 int min, int max) {
  int d; /* Value of this node's data. */
  size_t subcount[2]; /* Number of nodes in subtrees. */
```

```
    int i;
    if (node == NULL) {
        *count = 0;
        return;
    }
    d = *(int *) node->pbst_data;
    ⟨ Verify binary search tree ordering 114 ⟩
    recurse_verify_tree (node->pbst_link[0], okay, &subcount[0], min, d − 1);
    recurse_verify_tree (node->pbst_link[1], okay, &subcount[1], d + 1, max);
    *count = 1 + subcount[0] + subcount[1];
    ⟨ Verify PBST node parent pointers 518 ⟩
}
```

This code is included in §515.

§518 ⟨ Verify PBST node parent pointers 518 ⟩ ≡

```
for (i = 0; i < 2; i++) {
    if (node->pbst_link[i] != NULL && node->pbst_link[i]->pbst_parent != node) {
        printf (" Node %d has parent %d (should be %d).\n",
                *(int *) node->pbst_link[i]->pbst_data,
                (node->pbst_link[i]->pbst_parent != NULL
                 ? *(int *) node->pbst_link[i]->pbst_parent->pbst_data : −1),
                d);
        *okay = 0;
    }
}
```

This code is included in §517, §550, and §585.

14 AVL Trees with Parent Pointers

This chapter adds parent pointers to AVL trees. The result is a data structure that combines the strengths of AVL trees and trees with parent pointers. Of course, there's no free lunch: it combines their disadvantages, too.

The abbreviation we'll use for the term "AVL tree with parent pointers" is "PAVL tree", with corresponding prefix *pavl_*. Here's the outline for the PAVL table implementation:

§519 ⟨ pavl.h 519 ⟩ ≡
⟨ License 1 ⟩
#ifndef PAVL_H
#define PAVL_H 1

#include ⟨ stddef.h ⟩

⟨ Table types; tbl ⇒ pavl 14 ⟩
⟨ BST maximum height; bst ⇒ pavl 28 ⟩
⟨ TBST table structure; tbst ⇒ pavl 250 ⟩
⟨ PAVL node structure 521 ⟩
⟨ TBST traverser structure; tbst ⇒ pavl 267 ⟩
⟨ Table function prototypes; tbl ⇒ pavl 15 ⟩

#endif /∗ pavl.h ∗/

§520 ⟨ pavl.c 520 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "pavl.h"

⟨ PAVL functions 522 ⟩

14.1 Data Types

A PAVL tree node has a parent pointer and an AVL balance field in addition to the usual members needed for any binary search tree:

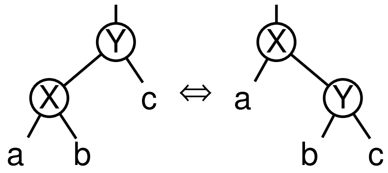
§521 ⟨ PAVL node structure 521 ⟩ ≡
```
/* An PAVL tree node. */
struct pavl_node {
    struct pavl_node *pavl_link[2]; /* Subtrees. */
    struct pavl_node *pavl_parent; /* Parent node. */
    void *pavl_data; /* Pointer to data. */
    signed char pavl_balance; /* Balance factor. */
};
```

This code is included in §519.

The other data structures are the same as the corresponding ones for TBSTs.

14.2 Rotations

Let's consider how rotations work in PBSTs. Here's the usual illustration of a rotation:



As we move from the left side to the right side, rotating right at Y, the parents of up to three nodes change. In any case, Y's former parent becomes X's new parent and X becomes Y's new parent. In addition, if b is not an empty subtree, then the parent of subtree b's root node becomes Y. Moving from right to left, the situation is reversed.

See also: [Cormen 1990], section 14.2.

Exercises:

1. Write functions for right and left rotations in BSTs with parent pointers, analogous to those for plain BSTs developed in Exercise 4.3-2.

14.3 Operations

As usual, we must reimplement the item insertion and deletion functions. The tree copy function and some of the traversal functions also need to be rewritten.

§522 ⟨ PAVL functions 522 ⟩ ≡
⟨ TBST creation function; tbst ⇒ pavl 252 ⟩
⟨ BST search function; bst ⇒ pavl 31 ⟩
⟨ PAVL item insertion function 523 ⟩
⟨ Table insertion convenience functions; tbl ⇒ pavl 592 ⟩
⟨ PAVL item deletion function 534 ⟩
⟨ PAVL traversal functions 546 ⟩
⟨ PAVL copy function 547 ⟩
⟨ BST destruction function; bst ⇒ pavl 84 ⟩
⟨ Default memory allocation functions; tbl ⇒ pavl 6 ⟩
⟨ Table assertion functions; tbl ⇒ pavl 594 ⟩

This code is included in §520.

14.4 Insertion

The same basic algorithm has been used for insertion in all of our AVL tree variants so far. (In fact, all three functions share the same set of local variables.) For PAVL trees, we will slightly modify our approach. In particular, until now we have cached comparison results on the way down in order to quickly adjust balance factors after the insertion. Parent pointers let us avoid this caching but still efficiently update balance factors.

Before we look closer, here is the function's outline:

§523 ⟨ PAVL item insertion function 523 ⟩ ≡
void ***pavl_probe* (**struct pavl_table** **tree*, **void** **item*) {

```
    struct pavl_node *y; /* Top node to update balance factor, and parent. */
    struct pavl_node *p, *q; /* Iterator, and parent. */
    struct pavl_node *n; /* Newly inserted node. */
    struct pavl_node *w; /* New root of rebalanced subtree. */
    int dir; /* Direction to descend. */

    assert (tree != NULL && item != NULL);

    〈 Step 1: Search PAVL tree for insertion point 524 〉
    〈 Step 2: Insert PAVL node 525 〉
    〈 Step 3: Update balance factors after PAVL insertion 526 〉
    〈 Step 4: Rebalance after PAVL insertion 527 〉
}
```

This code is included in §522.

14.4.1 Steps 1 and 2: Search and Insert

We search much as before. Despite use of the parent pointers, we preserve the use of q as the parent of p because the termination condition is a value of NULL for p, and NULL has no parent. (Thus, q is not, strictly speaking, always p's parent, but rather the last node examined before p.)

Because of parent pointers, there is no need for variable z, used in earlier implementations of AVL insertion to maintain y's parent.

§524 〈 Step 1: Search PAVL tree for insertion point 524 〉 ≡

```
y = tree->pavl_root;
for (q = NULL, p = tree->pavl_root; p != NULL; q = p, p = p->pavl_link[dir]) {
    int cmp = tree->pavl_compare (item, p->pavl_data, tree->pavl_param);
    if (cmp == 0)
        return &p->pavl_data;
    dir = cmp > 0;

    if (p->pavl_balance != 0)
        y = p;
}
```

This code is included in §523.

The node to create and insert the new node is based on that for PBSTs. There is a special case for a node inserted into an empty tree:

§525 〈 Step 2: Insert PAVL node 525 〉 ≡

```
〈 Step 2: Insert PBST node; pbst ⇒ pavl 492 〉
n->pavl_balance = 0;
if (tree->pavl_root == n)
    return &n->pavl_data;
```

This code is included in §523.

14.4.2 Step 3: Update Balance Factors

Until now, in step 3 of insertion into AVL trees we've always updated balance factors from the top down, starting at y and working our way down to n (see, e.g., 〈 Step 3: Update balance factors after AVL insertion 150⟩). This approach was somewhat unnatural, but it worked. The original reason we did it this way was that it was either impossible, as for AVL and RTAVL trees, or slow, as for TAVL trees, to efficiently move upward in a tree. That's not a consideration anymore, so we can do it from the bottom up and in the process eliminate the cache used before.

At each step, we need to know the node to update and, for that node, on which side of its parent it is a child. In the code below, *q* is the node and *dir* is the side.

§526 ⟨Step 3: Update balance factors after PAVL insertion 526⟩ ≡

```
for (p = n; p != y; p = q) {
    q = p→pavl_parent;
    dir = q→pavl_link[0] != p;
    if (dir == 0)
        q→pavl_balance−−;
    else q→pavl_balance++;
}
```

This code is included in §523.

Exercises:

1. Does this step 3 update the same set of balance factors as would a literal adaptation of ⟨Step 3: Update balance factors after AVL insertion 150⟩?

2. Would it be acceptable to substitute *q→pavl_link*[1] == *p* for *q→pavl_link*[0] != *p* in the code segment above?

14.4.3 Step 4: Rebalance

The changes needed to the rebalancing code for parent pointers resemble the changes for threads in that we can reuse most of the code from plain AVL trees. We just need to add a few new statements to each rebalancing case to adjust the parent pointers of nodes whose parents have changed.

The outline of the rebalancing code should be familiar by now. The code to update the link to the root of the rebalanced subtree is the only change. It needs a special case for the root, because the parent pointer of the root node is a null pointer, not the pseudo-root node. The other choice would simplify this piece of code, but complicate other pieces (see Section 13.1 [PBST Data Types], page 278).

§527 ⟨Step 4: Rebalance after PAVL insertion 527⟩ ≡

```
if (y→pavl_balance == −2)
    { ⟨Rebalance PAVL tree after insertion in left subtree 528⟩ }
else if (y→pavl_balance == +2)
    { ⟨Rebalance PAVL tree after insertion in right subtree 531⟩ }
else return &n→pavl_data;
if (w→pavl_parent != NULL)
    w→pavl_parent→pavl_link[y != w→pavl_parent→pavl_link[0]] = w;
else tree→pavl_root = w;

return &n→pavl_data;
```

This code is included in §523.

As usual, the cases for rebalancing are distinguished based on the balance factor of the child of the unbalanced node on its taller side:

§528 ⟨ Rebalance PAVL tree after insertion in left subtree 528 ⟩ ≡
struct pavl_node $*x = y \rightarrow pavl\_link[0]$;
if $(x \rightarrow pavl\_balance == {}^-1)$
  { ⟨ Rebalance for − balance factor in PAVL insertion in left subtree 529 ⟩ }
else { ⟨ Rebalance for + balance factor in PAVL insertion in left subtree 530 ⟩ }

This code is included in §527.

Case 1: *x* has − balance factor

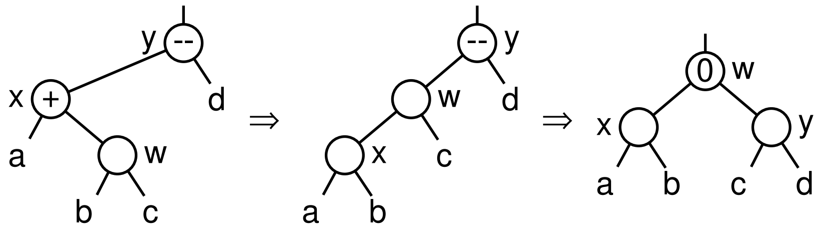
The added code here is exactly the same as that added to BST rotation to handle parent pointers (in Exercise 14.2-1), and for good reason since this case simply performs a right rotation in the PAVL tree.

§529 ⟨ Rebalance for − balance factor in PAVL insertion in left subtree 529 ⟩ ≡
⟨ Rotate right at y in AVL tree; avl ⇒ pavl 155 ⟩
$x \rightarrow pavl\_parent = y \rightarrow pavl\_parent$;
$y \rightarrow pavl\_parent = x$;
if $(y \rightarrow pavl\_link[0]$!= `NULL`)
  $y \rightarrow pavl\_link[0] \rightarrow pavl\_parent = y$;

This code is included in §528.

Case 2: *x* has + balance factor

When x has a + balance factor, we need a double rotation, composed of a right rotation at x followed by a left rotation at y. The diagram below show the effect of each of the rotations:



Along with this double rotation comes a small bulk discount in parent pointer assignments. The parent of w changes in both rotations, but we only need assign to it its final value once, ignoring the intermediate value.

§530 ⟨ Rebalance for + balance factor in PAVL insertion in left subtree 530 ⟩ ≡
⟨ Rotate left at x then right at y in AVL tree; avl ⇒ pavl 156 ⟩
$w \rightarrow pavl\_parent = y \rightarrow pavl\_parent$;
$x \rightarrow pavl\_parent = y \rightarrow pavl\_parent = w$;
if $(x \rightarrow pavl\_link[1]$!= `NULL`)
  $x \rightarrow pavl\_link[1] \rightarrow pavl\_parent = x$;
if $(y \rightarrow pavl\_link[0]$!= `NULL`)
  $y \rightarrow pavl\_link[0] \rightarrow pavl\_parent = y$;

This code is included in §528 and §544.

14.4.4 Symmetric Case

§531 ⟨ Rebalance PAVL tree after insertion in right subtree 531 ⟩ ≡
```
struct pavl_node *x = y→pavl_link[1];
if (x→pavl_balance == +1)
    { ⟨ Rebalance for + balance factor in PAVL insertion in right subtree 532 ⟩ }
else { ⟨ Rebalance for − balance factor in PAVL insertion in right subtree 533 ⟩ }
```
This code is included in §527.

§532 ⟨ Rebalance for + balance factor in PAVL insertion in right subtree 532 ⟩ ≡
```
⟨ Rotate left at y in AVL tree; avl ⇒ pavl 158 ⟩
x→pavl_parent = y→pavl_parent;
y→pavl_parent = x;
if (y→pavl_link[1] != NULL)
    y→pavl_link[1]→pavl_parent = y;
```
This code is included in §531.

§533 ⟨ Rebalance for − balance factor in PAVL insertion in right subtree 533 ⟩ ≡
```
⟨ Rotate right at x then left at y in AVL tree; avl ⇒ pavl 159 ⟩
w→pavl_parent = y→pavl_parent;
x→pavl_parent = y→pavl_parent = w;
if (x→pavl_link[0] != NULL)
    x→pavl_link[0]→pavl_parent = x;
if (y→pavl_link[1] != NULL)
    y→pavl_link[1]→pavl_parent = y;
```
This code is included in §531 and §541.

14.5 Deletion

Deletion from a PAVL tree is a natural outgrowth of algorithms we have already implemented. The basic algorithm is the one originally used for plain AVL trees. The search step is taken verbatim from PBST deletion. The deletion step combines PBST and TAVL tree code. Finally, the rebalancing strategy is the same as used in TAVL deletion.

The function outline is below. As noted above, step 1 is borrowed from PBST deletion. The other steps are implemented in the following sections.

§534 ⟨ PAVL item deletion function 534 ⟩ ≡
```
void *pavl_delete (struct pavl_table *tree, const void *item) {
    struct pavl_node *p; /* Traverses tree to find node to delete. */
    struct pavl_node *q; /* Parent of p. */
    int dir; /* Side of q on which p is linked. */

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Find PBST node to delete; pbst ⇒ pavl 494 ⟩
    ⟨ Step 2: Delete item from PAVL tree 535 ⟩
    ⟨ Steps 3 and 4: Update balance factors and rebalance after PAVL deletion 539 ⟩
}
```
This code is included in §522.

14.5.1 Step 2: Delete

The actual deletion step is derived from that for PBSTs. We add code to modify balance factors and set up for rebalancing. After the deletion, q is the node at which balance factors must be updated and possible rebalancing occurs and *dir* is the side of q from which the node was deleted. This follows the pattern already seen in TAVL deletion (see Section 8.5.2 [Deleting a TAVL Node Step 2 - Delete], page 198).

§535 ⟨ Step 2: Delete item from PAVL tree 535 ⟩ ≡
```
if (p→pavl_link[1] == NULL)
    { ⟨ Case 1 in PAVL deletion 536 ⟩ }
else {
    struct pavl_node *r = p→pavl_link[1];
    if (r→pavl_link[0] == NULL)
        { ⟨ Case 2 in PAVL deletion 537 ⟩ }
    else { ⟨ Case 3 in PAVL deletion 538 ⟩ }
}
tree→pavl_alloc→libavl_free (tree→pavl_alloc, p);
```
This code is included in §534.

Case 1: *p* has no right child

No changes are needed for case 1. No balance factors need change and q and *dir* are already set up correctly.

§536 ⟨ Case 1 in PAVL deletion 536 ⟩ ≡
```
⟨ Case 1 in PBST deletion; pbst ⇒ pavl 497 ⟩
```
This code is included in §535.

Case 2: *p*'s right child has no left child

See the commentary on ⟨ Case 3 in TAVL deletion 316 ⟩ for details.

§537 ⟨ Case 2 in PAVL deletion 537 ⟩ ≡
```
⟨ Case 2 in PBST deletion; pbst ⇒ pavl 498 ⟩
r→pavl_balance = p→pavl_balance;
q = r;
dir = 1;
```
This code is included in §535.

Case 3: *p*'s right child has a left child

See the commentary on ⟨ Case 4 in TAVL deletion 317 ⟩ for details.

§538 ⟨ Case 3 in PAVL deletion 538 ⟩ ≡
```
⟨ Case 3 in PBST deletion; pbst ⇒ pavl 499 ⟩
s→pavl_balance = p→pavl_balance;
q = r;
dir = 0;
```
This code is included in §535.

14.5.2 Step 3: Update Balance Factors

Step 3, updating balance factors, is taken straight from TAVL deletion (see Section 8.5.3 [Deleting a TAVL Node Step 3 - Update], page 199), with the call to *find_parent*() replaced by inline code that uses *pavl_parent*.

§539 ⟨ Steps 3 and 4: Update balance factors and rebalance after PAVL deletion 539 ⟩ ≡

```
while (q != (struct pavl_node *) &tree→pavl_root) {
    struct pavl_node *y = q;
    if (y→pavl_parent != NULL)
        q = y→pavl_parent;
    else q = (struct pavl_node *) &tree→pavl_root;
    if (dir == 0) {
        dir = q→pavl_link[0] != y;
        y→pavl_balance++;
        if (y→pavl_balance == +1)
            break;
        else if (y→pavl_balance == +2)
            { ⟨ Step 4: Rebalance after PAVL deletion 540 ⟩ }
    }
    else { ⟨ Steps 3 and 4: Symmetric case in PAVL deletion 543 ⟩ }
}
tree→pavl_count−−;
return (void *) item;
```

This code is included in §534.

14.5.3 Step 4: Rebalance

The two cases for PAVL deletion are distinguished based on x's balance factor, as always:

§540 ⟨ Step 4: Rebalance after PAVL deletion 540 ⟩ ≡

```
struct pavl_node *x = y→pavl_link[1];
if (x→pavl_balance == −1)
    { ⟨ Left-side rebalancing case 1 in PAVL deletion 541 ⟩ }
else { ⟨ Left-side rebalancing case 2 in PAVL deletion 542 ⟩ }
```

This code is included in §539.

Case 1: x has − balance factor

The same rebalancing is needed here as for a − balance factor in PAVL insertion, and the same code is used.

§541 ⟨ Left-side rebalancing case 1 in PAVL deletion 541 ⟩ ≡

```
struct pavl_node *w;
⟨ Rebalance for − balance factor in PAVL insertion in right subtree 533 ⟩
q→pavl_link[dir] = w;
```

This code is included in §540.

Case 2: x has + or 0 balance factor

If x has a + or 0 balance factor, we rotate left at y and update parent pointers as for any left rotation (see Section 14.2 [PBST Rotations], page 294). We also update balance factors. If x started with balance factor 0, then we're done. Otherwise, x becomes the new y for the next loop iteration, and rebalancing continues. See [avldel2], page 128, for details on this rebalancing case.

§542 ⟨ Left-side rebalancing case 2 in PAVL deletion 542 ⟩ ≡
```
y->pavl_link[1] = x->pavl_link[0];
x->pavl_link[0] = y;
x->pavl_parent = y->pavl_parent;
y->pavl_parent = x;
if (y->pavl_link[1] != NULL)
    y->pavl_link[1]->pavl_parent = y;
q->pavl_link[dir] = x;
if (x->pavl_balance == 0) {
    x->pavl_balance = -1;
    y->pavl_balance = +1;
    break;
} else {
    x->pavl_balance = y->pavl_balance = 0;
    y = x;
}
```
This code is included in §540.

14.5.4 Symmetric Case

§543 ⟨ Steps 3 and 4: Symmetric case in PAVL deletion 543 ⟩ ≡
```
dir = q->pavl_link[0] != y;
y->pavl_balance--;
if (y->pavl_balance == -1)
    break;
else if (y->pavl_balance == -2) {
    struct pavl_node *x = y->pavl_link[0];
    if (x->pavl_balance == +1)
        { ⟨ Right-side rebalancing case 1 in PAVL deletion 544 ⟩ }
    else { ⟨ Right-side rebalancing case 2 in PAVL deletion 545 ⟩ }
}
```
This code is included in §539.

§544 ⟨ Right-side rebalancing case 1 in PAVL deletion 544 ⟩ ≡
```
struct pavl_node *w;
⟨ Rebalance for + balance factor in PAVL insertion in left subtree 530 ⟩
q->pavl_link[dir] = w;
```
This code is included in §543.

§545 ⟨ Right-side rebalancing case 2 in PAVL deletion 545 ⟩ ≡
```
y->pavl_link[0] = x->pavl_link[1];
```

```
x→pavl_link[1] = y;
x→pavl_parent = y→pavl_parent;
y→pavl_parent = x;
if (y→pavl_link[0] != NULL)
    y→pavl_link[0]→pavl_parent = y;
q→pavl_link[dir] = x;
if (x→pavl_balance == 0) {
    x→pavl_balance = +1;
    y→pavl_balance = -1;
    break;
} else {
    x→pavl_balance = y→pavl_balance = 0;
    y = x;
}
```

This code is included in §543.

14.6 Traversal

The only difference between PAVL and PBST traversal functions is the insertion initializer. We use the TBST implementation here, which performs a call to *pavl_probe*(), instead of the PBST implementation, which inserts the node directly without handling node colors.

§546 ⟨ PAVL traversal functions 546 ⟩ ≡

```
⟨ TBST traverser null initializer; tbst ⇒ pavl 269 ⟩
⟨ PBST traverser first initializer; pbst ⇒ pavl 503 ⟩
⟨ PBST traverser last initializer; pbst ⇒ pavl 504 ⟩
⟨ PBST traverser search initializer; pbst ⇒ pavl 505 ⟩
⟨ TBST traverser insertion initializer; tbst ⇒ pavl 273 ⟩
⟨ TBST traverser copy initializer; tbst ⇒ pavl 274 ⟩
⟨ PBST traverser advance function; pbst ⇒ pavl 507 ⟩
⟨ PBST traverser back up function; pbst ⇒ pavl 508 ⟩
⟨ BST traverser current item function; bst ⇒ pavl 74 ⟩
⟨ BST traverser replacement function; bst ⇒ pavl 75 ⟩
```

This code is included in §522 and §554.

14.7 Copying

The copy function is the same as ⟨ PBST copy function 509 ⟩, except that it copies *pavl_balance* between copied nodes.

§547 ⟨ PAVL copy function 547 ⟩ ≡

```
⟨ PBST copy error helper function; pbst ⇒ pavl 510 ⟩
struct pavl_table *pavl_copy (const struct pavl_table *org, pavl_copy_func *copy,
                              pavl_item_func *destroy, struct libavl_allocator *allocator) {
    struct pavl_table *new;
    const struct pavl_node *x;
    struct pavl_node *y;
```

```
assert (org != NULL);
new = pavl_create (org->pavl_compare, org->pavl_param,
                   allocator != NULL ? allocator : org->pavl_alloc);
if (new == NULL)
    return NULL;
new->pavl_count = org->pavl_count;
if (new->pavl_count == 0)
    return new;
x = (const struct pavl_node *) &org->pavl_root;
y = (struct pavl_node *) &new->pavl_root;
for (;;) {
    while (x->pavl_link[0] != NULL) {
        y->pavl_link[0] = new->pavl_alloc->libavl_malloc (new->pavl_alloc,
                    sizeof *y->pavl_link[0]);
        if (y->pavl_link[0] == NULL) {
            if (y != (struct pavl_node *) &new->pavl_root) {
                y->pavl_data = NULL;
                y->pavl_link[1] = NULL;
            }
            copy_error_recovery (y, new, destroy);
            return NULL;
        }
        y->pavl_link[0]->pavl_parent = y;
        x = x->pavl_link[0];
        y = y->pavl_link[0];
    }
    y->pavl_link[0] = NULL;
    for (;;) {
        y->pavl_balance = x->pavl_balance;
        if (copy == NULL)
            y->pavl_data = x->pavl_data;
        else {
            y->pavl_data = copy (x->pavl_data, org->pavl_param);
            if (y->pavl_data == NULL) {
                y->pavl_link[1] = NULL;
                copy_error_recovery (y, new, destroy);
                return NULL;
            }
        }
        if (x->pavl_link[1] != NULL) {
            y->pavl_link[1] = new->pavl_alloc->libavl_malloc (new->pavl_alloc,
                        sizeof *y->pavl_link[1]);
            if (y->pavl_link[1] == NULL) {
                copy_error_recovery (y, new, destroy);
                return NULL;
            }
```

```
                y->pavl_link[1]->pavl_parent = y;
                x = x->pavl_link[1];
                y = y->pavl_link[1];
                break;
            }
            else y->pavl_link[1] = NULL;
            for (;;) {
                const struct pavl_node *w = x;
                x = x->pavl_parent;
                if (x == NULL) {
                    new->pavl_root->pavl_parent = NULL;
                    return new;
                }
                y = y->pavl_parent;
                if (w == x->pavl_link[0])
                    break;
            }
        }
    }
}
```

This code is included in §522 and §554.

14.8 Testing

The testing code harbors no surprises.

§548 ⟨ pavl-test.c 548 ⟩ ≡

```
⟨ License 1 ⟩
#include <assert.h>
#include <limits.h>
#include <stdio.h>
#include "pavl.h"
#include "test.h"
⟨ BST print function; bst ⇒ pavl 119 ⟩
⟨ BST traverser check function; bst ⇒ pavl 104 ⟩
⟨ Compare two PAVL trees for structure and content 549 ⟩
⟨ Recursively verify PAVL tree structure 550 ⟩
⟨ AVL tree verify function; avl ⇒ pavl 190 ⟩
⟨ BST test function; bst ⇒ pavl 100 ⟩
⟨ BST overflow test function; bst ⇒ pavl 122 ⟩
```

§549 ⟨ Compare two PAVL trees for structure and content 549 ⟩ ≡

```
/* Compares binary trees rooted at a and b,
   making sure that they are identical. */
static int compare_trees (struct pavl_node *a, struct pavl_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
```

```
    assert (a == NULL && b == NULL);
    return 1;
  }
  if (*(int *) a→pavl_data != *(int *) b→pavl_data
      || ((a→pavl_link[0] != NULL) != (b→pavl_link[0] != NULL))
      || ((a→pavl_link[1] != NULL) != (b→pavl_link[1] != NULL))
      || ((a→pavl_parent != NULL) != (b→pavl_parent != NULL))
      || (a→pavl_parent != NULL && b→pavl_parent != NULL
        && a→pavl_parent→pavl_data != b→pavl_parent→pavl_data)
      || a→pavl_balance != b→pavl_balance) {
    printf ("␣Copied␣nodes␣differ:\n"
            "␣␣a:␣%d,␣bal␣%+d,␣parent␣%d,␣%s␣left␣child,␣%s␣right␣child\n"
            "␣␣b:␣%d,␣bal␣%+d,␣parent␣%d,␣%s␣left␣child,␣%s␣right␣child\n",
            *(int *) a→pavl_data, a→pavl_balance,
            a→pavl_parent != NULL ? *(int *) a→pavl_parent : −1,
            a→pavl_link[0] != NULL ? "has" : "no",
            a→pavl_link[1] != NULL ? "has" : "no",
            *(int *) b→pavl_data, b→pavl_balance,
            b→pavl_parent != NULL ? *(int *) b→pavl_parent : −1,
            b→pavl_link[0] != NULL ? "has" : "no",
            b→pavl_link[1] != NULL ? "has" : "no");
    return 0;
  }
  okay = 1;
  if (a→pavl_link[0] != NULL)
    okay &= compare_trees (a→pavl_link[0], b→pavl_link[0]);
  if (a→pavl_link[1] != NULL)
    okay &= compare_trees (a→pavl_link[1], b→pavl_link[1]);
  return okay;
}
```

This code is included in §548.

§550 ⟨ Recursively verify PAVL tree structure 550 ⟩ ≡

```
static void recurse_verify_tree (struct pavl_node *node, int *okay, size_t *count,
                                 int min, int max, int *height) {
  int d; /* Value of this node's data. */
  size_t subcount[2]; /* Number of nodes in subtrees. */
  int subheight[2]; /* Heights of subtrees. */
  int i;

  if (node == NULL) {
    *count = 0;
    *height = 0;
    return;
  }
  d = *(int *) node→pavl_data;

  ⟨ Verify binary search tree ordering 114 ⟩
```

```
    recurse_verify_tree (node→pavl_link[0], okay, &subcount[0],
                         min, d − 1, &subheight[0]);
    recurse_verify_tree (node→pavl_link[1], okay, &subcount[1],
                         d + 1, max, &subheight[1]);
    *count = 1 + subcount[0] + subcount[1];
    *height = 1 + (subheight[0] > subheight[1] ? subheight[0] : subheight[1]);
    ⟨ Verify AVL node balance factor; avl ⇒ pavl 189 ⟩
    ⟨ Verify PBST node parent pointers; pbst ⇒ pavl 518 ⟩
}
```

This code is included in §548.

15 Red-Black Trees with Parent Pointers

As our twelfth and final example of a table data structure, this chapter will implement a table as a red-black tree with parent pointers, or "PRB" tree for short. We use *prb_* as the prefix for identifiers. Here's the outline:

§551 ⟨ prb.h 551 ⟩ ≡
⟨ License 1 ⟩
#ifndef PRB_H
#define PRB_H 1

#include ⟨ stddef.h ⟩

⟨ Table types; tbl ⇒ prb 14 ⟩
⟨ RB maximum height; rb ⇒ prb 195 ⟩
⟨ TBST table structure; tbst ⇒ prb 250 ⟩
⟨ PRB node structure 553 ⟩
⟨ TBST traverser structure; tbst ⇒ prb 267 ⟩
⟨ Table function prototypes; tbl ⇒ prb 15 ⟩

#endif /∗ prb.h ∗/

§552 ⟨ prb.c 552 ⟩ ≡
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "prb.h"

⟨ PRB functions 554 ⟩

15.1 Data Types

The PRB node structure adds a color and a parent pointer to the basic binary tree data structure. The other PRB data structures are the same as the ones used for TBSTs.

§553 ⟨ PRB node structure 553 ⟩ ≡
```
/* Color of a red-black node. */
enum prb_color {
    PRB_BLACK, /* Black. */
    PRB_RED /* Red. */
};

/* A red-black tree with parent pointers node. */
struct prb_node {
    struct prb_node *prb_link[2]; /* Subtrees. */
    struct prb_node *prb_parent; /* Parent. */
    void *prb_data; /* Pointer to data. */
    unsigned char prb_color; /* Color. */
};
```
This code is included in §551.

See also: [Cormen 1990], section 14.1.

15.2 Operations

Most of the PRB operations use the same implementations as did PAVL trees in the last chapter. The PAVL copy function is modified to copy colors instead of balance factors. The item insertion and deletion functions must be newly written, of course.

§554 ⟨ PRB functions 554 ⟩ ≡
⟨ TBST creation function; tbst ⇒ prb 252 ⟩
⟨ BST search function; bst ⇒ prb 31 ⟩
⟨ PRB item insertion function 555 ⟩
⟨ Table insertion convenience functions; tbl ⇒ prb 592 ⟩
⟨ PRB item deletion function 566 ⟩
⟨ PAVL traversal functions; pavl ⇒ prb 546 ⟩
⟨ PAVL copy function; pavl ⇒ prb; pavl_balance ⇒ prb_color 547 ⟩
⟨ BST destruction function; bst ⇒ prb 84 ⟩
⟨ Default memory allocation functions; tbl ⇒ prb 6 ⟩
⟨ Table assertion functions; tbl ⇒ prb 594 ⟩

This code is included in §552.

15.3 Insertion

Inserting into a red-black tree is a problem whose form of solution should by now be familiar to the reader. We must now update parent pointers, of course, but the major difference here is that it is fast and easy to find the parent of any given node, eliminating any need for a stack.

Here's the function outline. The code for finding the insertion point is taken directly from the PBST code:

§555 ⟨ PRB item insertion function 555 ⟩ ≡

```
void **prb_probe (struct prb_table *tree, void *item) {
    struct prb_node *p; /* Traverses tree looking for insertion point. */
    struct prb_node *q; /* Parent of p; node at which we are rebalancing. */
    struct prb_node *n; /* Newly inserted node. */
    int dir; /* Side of q on which n is inserted. */

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search PBST tree for insertion point; pbst ⇒ prb 491 ⟩
    ⟨ Step 2: Insert PRB node 556 ⟩
    ⟨ Step 3: Rebalance after PRB insertion 557 ⟩

    return &n->prb_data;
}
```

This code is included in §554.

See also: [Cormen 1990], section 14.3.

15.3.1 Step 2: Insert

The code to do the insertion is based on that for PBSTs. We need only add initialization of the new node's color.

§556 ⟨ Step 2: Insert PRB node 556 ⟩ ≡
⟨ Step 2: Insert PBST node; pbst ⇒ prb 492 ⟩
$n \rightarrow prb\_color$ = `PRB_RED`;

This code is included in §555.

15.3.2 Step 3: Rebalance

When we rebalanced ordinary RB trees, we used the expressions $pa[k-1]$ and $pa[k-2]$ to refer to the parent and grandparent, respectively, of the node at which we were rebalancing, and we called that node q, though that wasn't a variable name (see Section 6.4.3 [Inserting an RB Node Step 3 - Rebalance], page 143). Now that we have parent pointers, we use a real variable q to refer to the node where we're rebalancing.

This means that we could refer to its parent and grandparent as $q \rightarrow prb\_parent$ and $q \rightarrow prb\_parent \rightarrow prb\_parent$, respectively, but there's a small problem with that. During rebalancing, we will need to move nodes around and modify parent pointers. That means that $q \rightarrow prb\_parent$ and $q \rightarrow prb\_parent \rightarrow prb\_parent$ will be changing under us as we work. This makes writing correct code hard, and reading it even harder. It is much easier to use a pair of new variables to hold q's parent and grandparent.

That's exactly the role that f and g, respectively, play in the code below. If you compare this code to ⟨ Step 3: Rebalance after RB insertion 201 ⟩, you'll also notice the way that checking that f and g are non-null corresponds to checking that the stack height is at least 3 (see Exercise 6.4.3-1 for an explanation of the reason this is a valid test).

§557 ⟨ Step 3: Rebalance after PRB insertion 557 ⟩ ≡
```
q = n;
for (;;) {
    struct prb_node *f; /* Parent of q. */
    struct prb_node *g; /* Grandparent of q. */
    f = q->prb_parent;
    if (f == NULL || f->prb_color == PRB_BLACK)
        break;
    g = f->prb_parent;
    if (g == NULL)
        break;
    if (g->prb_link[0] == f)
        { ⟨ Left-side rebalancing after PRB insertion 558 ⟩ }
    else { ⟨ Right-side rebalancing after PRB insertion 562 ⟩ } }
tree->prb_root->prb_color = PRB_BLACK;
```
This code is included in §555.

After replacing $pa[k-1]$ by f and $pa[k-2]$ by g, the cases for PRB rebalancing are distinguished on the same basis as those for RB rebalancing (see ⟨ Left-side rebalancing after RB insertion 202 ⟩). One addition: cases 2 and 3 need to work with q's great-grandparent, so they stash it into a new variable h.

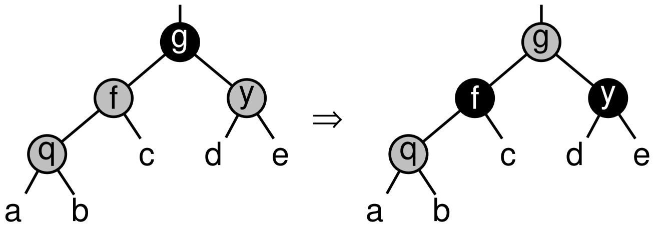
§558 ⟨ Left-side rebalancing after PRB insertion 558 ⟩ ≡
```
struct prb_node *y = g->prb_link[1];
if (y != NULL && y->prb_color == PRB_RED)
```

```
    { ⟨ Case 1 in left-side PRB insertion rebalancing 559 ⟩ }
else {
    struct prb_node *h; /* Great-grandparent of q. */
    h = g→prb_parent;
    if (h == NULL)
        h = (struct prb_node *) &tree→prb_root;
    if (f→prb_link[1] == q)
        { ⟨ Case 3 in left-side PRB insertion rebalancing 561 ⟩ }
    ⟨ Case 2 in left-side PRB insertion rebalancing 560 ⟩
    break;
}
```

This code is included in §557.

Case 1: *q*'s uncle is red

In this case, as before, we need only rearrange colors (see page 145). Instead of popping the top two items off the stack, we directly set up q, the next node at which to rebalance, to be the (former) grandparent of the original q.



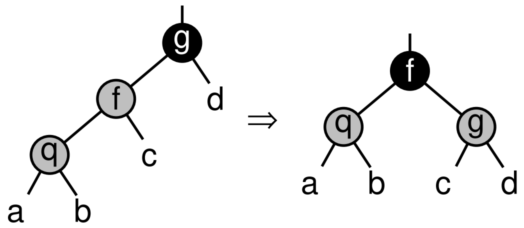
§559 ⟨ Case 1 in left-side PRB insertion rebalancing 559 ⟩ ≡

```
f→prb_color = y→prb_color = PRB_BLACK;
g→prb_color = PRB_RED;
q = g;
```

This code is included in §558.

Case 2: *q* is the left child of its parent

If q is the left child of its parent, we rotate right at g:



The result satisfies both RB balancing rules. Refer back to the discussion of the same case in ordinary RB trees for more details (see page 145).

§560 ⟨ Case 2 in left-side PRB insertion rebalancing 560 ⟩ ≡

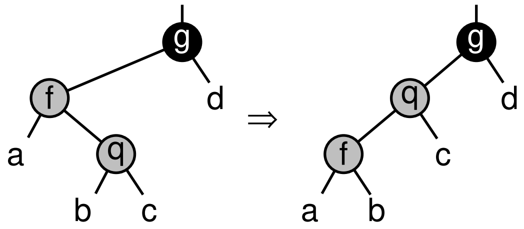
```
g→prb_color = PRB_RED;
```

```
f→prb_color = PRB_BLACK;
g→prb_link[0] = f→prb_link[1];
f→prb_link[1] = g;
h→prb_link[h→prb_link[0] != g] = f;
f→prb_parent = g→prb_parent;
g→prb_parent = f;
if (g→prb_link[0] != NULL)
    g→prb_link[0]→prb_parent = g;
```

This code is included in §558.

Case 3: *q* is the right child of its parent

If q is a right child, then we transform it into case 2 by rotating left at f:



Afterward we relabel q as f and treat the result as case 2. There is no need to properly set q itself because case 2 never uses variable q. For more details, refer back to case 3 in ordinary RB trees (see page 146).

§561 ⟨ Case 3 in left-side PRB insertion rebalancing 561 ⟩ ≡

```
f→prb_link[1] = q→prb_link[0];
q→prb_link[0] = f;
g→prb_link[0] = q;
f→prb_parent = q;
if (f→prb_link[1] != NULL)
    f→prb_link[1]→prb_parent = f;
f = q;
```

This code is included in §558.

15.3.3 Symmetric Case

§562 ⟨ Right-side rebalancing after PRB insertion 562 ⟩ ≡

```
struct prb_node *y = g→prb_link[0];
if (y != NULL && y→prb_color == PRB_RED)
    { ⟨ Case 1 in right-side PRB insertion rebalancing 563 ⟩ }
else {
    struct prb_node *h; /* Great-grandparent of q. */
    h = g→prb_parent;
    if (h == NULL)
        h = (struct prb_node *) &tree→prb_root;
    if (f→prb_link[0] == q)
```

```
        { ⟨Case 3 in right-side PRB insertion rebalancing 565⟩ }
    ⟨Case 2 in right-side PRB insertion rebalancing 564⟩
    break;
}
```

This code is included in §557.

§563 ⟨Case 1 in right-side PRB insertion rebalancing 563⟩ ≡

```
f->prb_color = y->prb_color = PRB_BLACK;
g->prb_color = PRB_RED;
q = g;
```

This code is included in §562.

§564 ⟨Case 2 in right-side PRB insertion rebalancing 564⟩ ≡

```
g->prb_color = PRB_RED;
f->prb_color = PRB_BLACK;

g->prb_link[1] = f->prb_link[0];
f->prb_link[0] = g;
h->prb_link[h->prb_link[0] != g] = f;

f->prb_parent = g->prb_parent;
g->prb_parent = f;
if (g->prb_link[1] != NULL)
    g->prb_link[1]->prb_parent = g;
```

This code is included in §562.

§565 ⟨Case 3 in right-side PRB insertion rebalancing 565⟩ ≡

```
f->prb_link[0] = q->prb_link[1];
q->prb_link[1] = f;
g->prb_link[1] = q;
f->prb_parent = q;
if (f->prb_link[0] != NULL)
    f->prb_link[0]->prb_parent = f;

f = q;
```

This code is included in §562.

15.4 Deletion

The RB item deletion algorithm needs the same kind of changes to handle parent pointers that the RB item insertion algorithm did. We can reuse the code from PBST trees for finding the node to delete. The rest of the code will be presented in the following sections.

§566 ⟨PRB item deletion function 566⟩ ≡

```
void *prb_delete (struct prb_table *tree, const void *item) {
    struct prb_node *p; /* Node to delete. */
    struct prb_node *q; /* Parent of p. */
    struct prb_node *f; /* Node at which we are rebalancing. */
    int dir; /* Side of q on which p is a child;
                side of f from which node was deleted. */

    assert (tree != NULL && item != NULL);
```

```
    ⟨ Step 1: Find PBST node to delete; pbst ⇒ prb 494 ⟩
    ⟨ Step 2: Delete item from PRB tree 567 ⟩
    ⟨ Step 3: Rebalance tree after PRB deletion 571 ⟩
    ⟨ Step 4: Finish up after PRB deletion 577 ⟩
}
```

This code is included in §554.

See also: [Cormen 1990], section 14.4.

15.4.1 Step 2: Delete

The goal of this step is to remove p from the tree and set up f as the node where rebalancing should start. Secondarily, we set *dir* as the side of f from which the node was deleted. Together, f and *dir* fill the role that the top-of-stack entries in *pa*[] and *da*[] took in ordinary RB deletion.

§567 ⟨ Step 2: Delete item from PRB tree 567 ⟩ ≡

```
if (p→prb_link[1] == NULL)
    { ⟨ Case 1 in PRB deletion 568 ⟩ }
else {
    enum prb_color t;
    struct prb_node *r = p→prb_link[1];
    if (r→prb_link[0] == NULL)
        { ⟨ Case 2 in PRB deletion 569 ⟩ }
    else { ⟨ Case 3 in PRB deletion 570 ⟩ }
}
```

This code is included in §566.

Case 1: *p* has no right child

If p has no right child, then rebalancing should start at its parent, q, and *dir* is already the side that p is on. The rest is the same as PBST deletion (see page 281).

§568 ⟨ Case 1 in PRB deletion 568 ⟩ ≡

```
⟨ Case 1 in PBST deletion; pbst ⇒ prb 497 ⟩
f = q;
```

This code is included in §567.

Case 2: *p*'s right child has no left child

In case 2, we swap the colors of p and r as for ordinary RB deletion (see page 152). We set up f and *dir* in the same way that ⟨ Case 2 in RB deletion 223 ⟩ set up the top of stack. The rest is the same as PBST deletion (see page 281).

§569 ⟨ Case 2 in PRB deletion 569 ⟩ ≡

```
⟨ Case 2 in PBST deletion; pbst ⇒ prb 498 ⟩
t = p→prb_color;
p→prb_color = r→prb_color;
r→prb_color = t;
```

```
f = r;
dir = 1;
```

This code is included in §567.

Case 3: *p*'s right child has a left child

Case 2 swaps the colors of p and s the same way as in ordinary RB deletion (see page 152), and sets up f and *dir* in the same way that ⟨ Case 3 in RB deletion 224 ⟩ set up the stack. The rest is borrowed from PBST deletion (see page 282).

§570 ⟨ Case 3 in PRB deletion 570 ⟩ ≡

```
⟨ Case 3 in PBST deletion; pbst ⇒ prb 499 ⟩
t = p->prb_color;
p->prb_color = s->prb_color;
s->prb_color = t;
f = r;
dir = 0;
```

This code is included in §567.

15.4.2 Step 3: Rebalance

The rebalancing code is easily related to the analogous code for ordinary RB trees in ⟨ Rebalance after RB deletion 226 ⟩. As we carefully set up in step 2, we use f as the top of stack node and *dir* as the side of f from which a node was deleted. These variables f and *dir* were formerly represented by $pa[k-1]$ and $da[k-1]$, respectively. Additionally, variable g is used to represent the parent of f. Formerly the same node was referred to as $pa[k-2]$.

The code at the end of the loop simply moves f and *dir* up one level in the tree. It has the same effect as did popping the stack with $k--$.

§571 ⟨ Step 3: Rebalance tree after PRB deletion 571 ⟩ ≡

```
if (p->prb_color == PRB_BLACK) {
    for (;;) {
        struct prb_node *x; /* Node we want to recolor black if possible. */
        struct prb_node *g; /* Parent of f. */
        struct prb_node *t; /* Temporary for use in finding parent. */
        x = f->prb_link[dir];
        if (x != NULL && x->prb_color == PRB_RED)
            {
                x->prb_color = PRB_BLACK;
                break;
            }
        if (f == (struct prb_node *) &tree->prb_root)
            break;
        g = f->prb_parent;
        if (g == NULL)
            g = (struct prb_node *) &tree->prb_root;
```

```
            if (dir == 0)
                { ⟨Left-side rebalancing after PRB deletion 572⟩ }
            else { ⟨Right-side rebalancing after PRB deletion 578⟩ }
            t = f;
            f = f→prb_parent;
            if (f == NULL)
                f = (struct prb_node *) &tree→prb_root;
            dir = f→prb_link[0] != t;
        }
    }
```

This code is included in §566.

The code to distinguish rebalancing cases in PRB trees is almost identical to ⟨Left-side rebalancing after RB deletion 227⟩.

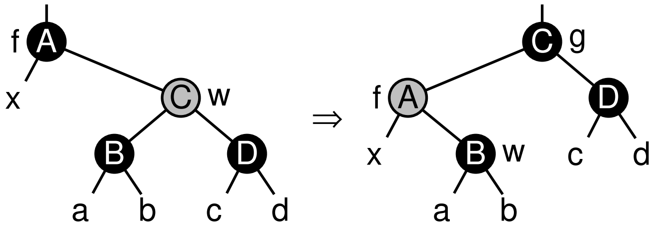
§572 ⟨Left-side rebalancing after PRB deletion 572⟩ ≡

```
struct prb_node *w = f→prb_link[1];
if (w→prb_color == PRB_RED)
    { ⟨Ensure w is black in left-side PRB deletion rebalancing 573⟩ }
if ((w→prb_link[0] == NULL || w→prb_link[0]→prb_color == PRB_BLACK)
    && (w→prb_link[1] == NULL || w→prb_link[1]→prb_color == PRB_BLACK))
    { ⟨Case 1 in left-side PRB deletion rebalancing 574⟩ }
else {
    if (w→prb_link[1] == NULL || w→prb_link[1]→prb_color == PRB_BLACK)
        { ⟨Transform left-side PRB deletion rebalancing case 3 into case 2 576⟩ }
    ⟨Case 2 in left-side PRB deletion rebalancing 575⟩
    break;
}
```

This code is included in §571.

Case Reduction: Ensure *w* is black

The case reduction code is much like that for plain RB trees (see page 155), with $pa[k-1]$ replaced by f and $pa[k-2]$ replaced by g. Instead of updating the stack, we change g. Node f need not change because it's already what we want it to be. We also need to update parent pointers for the rotation.



§573 ⟨Ensure w is black in left-side PRB deletion rebalancing 573⟩ ≡

```
w→prb_color = PRB_BLACK;
f→prb_color = PRB_RED;
f→prb_link[1] = w→prb_link[0];
```

$w \rightarrow prb\_link[0] = f;$
$g \rightarrow prb\_link[g \rightarrow prb\_link[0] \mathrel{!=} f] = w;$
$w \rightarrow prb\_parent = f \rightarrow prb\_parent;$
$f \rightarrow prb\_parent = w;$
$g = w;$
$w = f \rightarrow prb\_link[1];$
$w \rightarrow prb\_parent = f;$

This code is included in §572.

Case 1: *w* has no red children

Case 1 is trivial. No changes from ordinary RB trees are necessary (see page 156).

§574 ⟨ Case 1 in left-side PRB deletion rebalancing 574 ⟩ ≡
⟨ Case 1 in left-side RB deletion rebalancing; rb ⇒ prb 229 ⟩

This code is included in §572.

Case 2: *w*'s right child is red

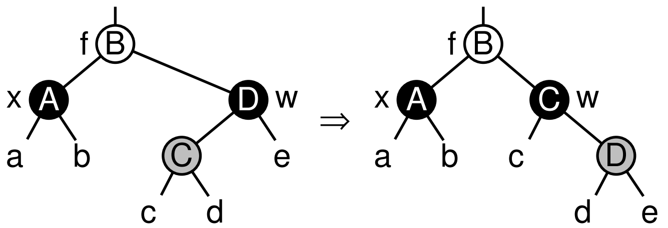
The changes from ordinary RB trees (see page 157) for case 2 follow the same pattern.

§575 ⟨ Case 2 in left-side PRB deletion rebalancing 575 ⟩ ≡
$w \rightarrow prb\_color = f \rightarrow prb\_color;$
$f \rightarrow prb\_color =$ `PRB_BLACK`;
$w \rightarrow prb\_link[1] \rightarrow prb\_color =$ `PRB_BLACK`;
$f \rightarrow prb\_link[1] = w \rightarrow prb\_link[0];$
$w \rightarrow prb\_link[0] = f;$
$g \rightarrow prb\_link[g \rightarrow prb\_link[0] \mathrel{!=} f] = w;$
$w \rightarrow prb\_parent = f \rightarrow prb\_parent;$
$f \rightarrow prb\_parent = w;$
if ($f \rightarrow prb\_link[1] \mathrel{!=}$ `NULL`)
    $f \rightarrow prb\_link[1] \rightarrow prb\_parent = f;$

This code is included in §572.

Case 3: *w*'s left child is red

The code for case 3 in ordinary RB trees (see page 157) needs slightly more intricate changes than case 1 or case 2, so the diagram below may help to clarify:



§576 ⟨ Transform left-side PRB deletion rebalancing case 3 into case 2 576 ⟩ ≡
struct prb_node $*y = w \rightarrow prb\_link[0];$

```
y→prb_color = PRB_BLACK;
w→prb_color = PRB_RED;
w→prb_link[0] = y→prb_link[1];
y→prb_link[1] = w;
if (w→prb_link[0] != NULL)
    w→prb_link[0]→prb_parent = w;
w = f→prb_link[1] = y;
w→prb_link[1]→prb_parent = w;
```

This code is included in §572.

15.4.3 Step 4: Finish Up

§577 ⟨ Step 4: Finish up after PRB deletion 577 ⟩ ≡

```
tree→prb_alloc→libavl_free (tree→prb_alloc, p);
tree→prb_count−−;
return (void *) item;
```

This code is included in §566.

15.4.4 Symmetric Case

§578 ⟨ Right-side rebalancing after PRB deletion 578 ⟩ ≡

```
struct prb_node *w = f→prb_link[0];
if (w→prb_color == PRB_RED)
    { ⟨ Ensure w is black in right-side PRB deletion rebalancing 579 ⟩ }
if ((w→prb_link[0] == NULL || w→prb_link[0]→prb_color == PRB_BLACK)
  && (w→prb_link[1] == NULL || w→prb_link[1]→prb_color == PRB_BLACK))
    { ⟨ Case 1 in right-side PRB deletion rebalancing 580 ⟩ }
else {
    if (w→prb_link[0] == NULL || w→prb_link[0]→prb_color == PRB_BLACK)
        { ⟨ Transform right-side PRB deletion rebalancing case 3 into case 2 582 ⟩ }
    ⟨ Case 2 in right-side PRB deletion rebalancing 581 ⟩
    break;
}
```

This code is included in §571.

§579 ⟨ Ensure w is black in right-side PRB deletion rebalancing 579 ⟩ ≡

```
w→prb_color = PRB_BLACK;
f→prb_color = PRB_RED;
f→prb_link[0] = w→prb_link[1];
w→prb_link[1] = f;
g→prb_link[g→prb_link[0] != f] = w;
w→prb_parent = f→prb_parent;
f→prb_parent = w;
g = w;
w = f→prb_link[0];
w→prb_parent = f;
```

This code is included in §578.

§580 ⟨ Case 1 in right-side PRB deletion rebalancing 580 ⟩ ≡
```
w->prb_color = PRB_RED;
```
This code is included in §578.

§581 ⟨ Case 2 in right-side PRB deletion rebalancing 581 ⟩ ≡
```
w->prb_color = f->prb_color;
f->prb_color = PRB_BLACK;
w->prb_link[0]->prb_color = PRB_BLACK;
f->prb_link[0] = w->prb_link[1];
w->prb_link[1] = f;
g->prb_link[g->prb_link[0] != f] = w;
w->prb_parent = f->prb_parent;
f->prb_parent = w;
if (f->prb_link[0] != NULL)
    f->prb_link[0]->prb_parent = f;
```
This code is included in §578.

§582 ⟨ Transform right-side PRB deletion rebalancing case 3 into case 2 582 ⟩ ≡
```
struct prb_node *y = w->prb_link[1];
y->prb_color = PRB_BLACK;
w->prb_color = PRB_RED;
w->prb_link[1] = y->prb_link[0];
y->prb_link[0] = w;
if (w->prb_link[1] != NULL)
    w->prb_link[1]->prb_parent = w;
w = f->prb_link[0] = y;
w->prb_link[0]->prb_parent = w;
```
This code is included in §578.

15.5 Testing

No comment is necessary.

§583 ⟨ prb-test.c 583 ⟩ ≡
```
⟨ License 1 ⟩
#include <assert.h>
#include <limits.h>
#include <stdio.h>
#include "prb.h"
#include "test.h"
⟨ BST print function; bst ⇒ prb 119 ⟩
⟨ BST traverser check function; bst ⇒ prb 104 ⟩
⟨ Compare two PRB trees for structure and content 584 ⟩
⟨ Recursively verify PRB tree structure 585 ⟩
⟨ RB tree verify function; rb ⇒ prb 244 ⟩
⟨ BST test function; bst ⇒ prb 100 ⟩
⟨ BST overflow test function; bst ⇒ prb 122 ⟩
```

§584 ⟨ Compare two PRB trees for structure and content 584 ⟩ ≡

```
static int compare_trees (struct prb_node *a, struct prb_node *b) {
    int okay;
    if (a == NULL || b == NULL) {
        assert (a == NULL && b == NULL);
        return 1;
    }
    if (*(int *) a->prb_data != *(int *) b->prb_data
        || ((a->prb_link[0] != NULL) != (b->prb_link[0] != NULL))
        || ((a->prb_link[1] != NULL) != (b->prb_link[1] != NULL))
        || a->prb_color != b->prb_color) {
        printf (" Copied nodes differ: a=%d%c b=%d%c a:",
                *(int *) a->prb_data, a->prb_color == PRB_RED ? 'r' : 'b',
                *(int *) b->prb_data, b->prb_color == PRB_RED ? 'r' : 'b');
        if (a->prb_link[0] != NULL) printf ("l");
        if (a->prb_link[1] != NULL) printf ("r");
        printf (" b:");
        if (b->prb_link[0] != NULL) printf ("l");
        if (b->prb_link[1] != NULL) printf ("r");
        printf ("\n");
        return 0;
    }
    okay = 1;
    if (a->prb_link[0] != NULL)
        okay &= compare_trees (a->prb_link[0], b->prb_link[0]);
    if (a->prb_link[1] != NULL)
        okay &= compare_trees (a->prb_link[1], b->prb_link[1]);
    return okay;
}
```

This code is included in §583.

§585 ⟨ Recursively verify PRB tree structure 585 ⟩ ≡

```
/* Examines the binary tree rooted at node.
   Zeroes *okay if an error occurs. Otherwise, does not modify *okay.
   Sets *count to the number of nodes in that tree, including node itself if node != NULL.
   Sets *bh to the tree's black-height.
   All the nodes in the tree are verified to be at least min but no greater than max. */
static void recurse_verify_tree (struct prb_node *node, int *okay, size_t *count,
                                 int min, int max, int *bh) {
    int d; /* Value of this node's data. */
    size_t subcount[2]; /* Number of nodes in subtrees. */
    int subbh[2]; /* Black-heights of subtrees. */
    int i;
    if (node == NULL) {
        *count = 0;
        *bh = 0;
```

```
        return;
    }
    d = *(int *) node->prb_data;
    ⟨ Verify binary search tree ordering 114 ⟩
    recurse_verify_tree (node->prb_link[0], okay, &subcount[0],
                         min, d - 1, &subbh[0]);
    recurse_verify_tree (node->prb_link[1], okay, &subcount[1],
                         d + 1, max, &subbh[1]);
    *count = 1 + subcount[0] + subcount[1];
    *bh = (node->prb_color == PRB_BLACK) + subbh[0];
    ⟨ Verify RB node color; rb ⇒ prb 241 ⟩
    ⟨ Verify RB node rule 1 compliance; rb ⇒ prb 242 ⟩
    ⟨ Verify RB node rule 2 compliance; rb ⇒ prb 243 ⟩
    ⟨ Verify PBST node parent pointers; pbst ⇒ prb 518 ⟩
}
```

This code is included in §583.

Appendix A References

[Aho 1986]. Aho, A. V., R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

[Bentley 2000]. Bentley, J., *Programming Pearls*, 2nd ed. Addison-Wesley, 2000. ISBN 0-201-65788-0.

[Brown 2001]. Brown, S., "Identifiers NOT To Use in C Programs". Oak Road Systems, Feb. 15, 2001. `http://www.oakroadsystems.com/tech/c-predef.htm`.

[Cormen 1990]. Cormen, C. H., C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. McGraw-Hill, 1990. ISBN 0-262-03141-8.

[FSF 1999]. Free Software Foundation, *GNU C Library Reference Manual*, version 0.08, 1999.

[FSF 2001]. Free Software Foundation, "GNU Coding Standards", ed. of March 23, 2001.

[ISO 1990]. International Organization for Standardization, *ANSI/ISO 9899-1990: American National Standard for Programming Languages—C*, 1990. Reprinted in *The Annotated ANSI C Standard*, ISBN 0-07-881952-0.

[ISO 1998]. International Organization for Standardization, *ISO/IEC 14882:1998(E): Programming languages—C++*, 1998.

[ISO 1999]. International Orgnaization for Standardization, *ISO/IEC 9899:1999: Programming Languages—C*, 2nd ed., 1999.

[Kernighan 1976]. Kernighan, B. W., and P. J. Plauger, *Software Tools*. Addison-Wesley, 1976. ISBN 0-201-03669-X.

[Kernighan 1988]. Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice-Hall, 1988. ISBN 0-13-110362-8.

[Knuth 1997]. Knuth, D. E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Addison-Wesley, 1997. ISBN 0-201-89683-4.

[Knuth 1998a]. Knuth, D. E., *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1998. ISBN 0-201-89684-2.

[Knuth 1998b]. Knuth, D. E., *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Addison-Wesley, 1998. ISBN 0-201-89685-0.

[Knuth 1977]. Knuth, D. E., "Deletions that Preserve Randomness", *IEEE Trans. on Software Eng.* SE-3 (1977), pp. 351–9. Reprinted in [Knuth 2000].

[Knuth 1978]. Knuth, D. E., "A Trivial Algorithm Whose Analysis Isn't", *Journal of Algorithms* 6 (1985), pp. 301–22. Reprinted in [Knuth 2000].

[Knuth 1992]. Knuth, D. E., *Literate Programming*, CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Leland Stanford Junior University, 1992. ISBN 0-9370-7380-6.

[Knuth 2000]. Knuth, D. E., *Selected Papers on Analysis of Algorithms*, CSLI Lecture Notes Number 102. Center for the Study of Language and Information, Leland Stanford Junior University, 2000. ISBN 1-57586-212-3.

[Pfaff 1998]. Pfaff, B. L., "An Iterative Algorithm for Deletion from AVL-Balanced Binary Trees". Presented July 1998, annual meeting of Pi Mu Epsilon, Toronto, Canada. `http://benpfaff.org/avl/`.

[Sedgewick 1998]. Sedgewick, R., *Algorithms in C, Parts 1-4*, 3rd ed. Addison-Wesley, 1998. ISBN 0-201-31452-5.

[SGI 1993]. Silicon Graphics, Inc., *Standard Template Library Programmer's Guide*. `http://www.sgi.com/tech/stl/`.

[Stout 1986]. Stout, F. S. and B. L. Warren, "Tree Rebalancing in Optimal Time and Space", *Communications of the ACM* 29 (1986), pp. 902–908.

[Summit 1999]. Summit, S., "comp.lang.c Answers to Frequently Asked Questions", version 3.5. `http://www.eskimo.com/~scs/C-faq/top.html`. ISBN 0-201-84519-9.

Appendix B Supplementary Code

This appendix contains code too long for the exposition or too far from the main topic of the book.

B.1 Option Parser

The BST test program contains an option parser for handling command-line options. See Section 4.14.5 [User Interaction], page 101, for an introduction to its public interface. This section describes the option parser's implementation.

The option parsing state is kept in **struct option_state**:

§586 ⟨ Option parser 586 ⟩ ≡
```
/* Option parsing state. */
struct option_state {
    const struct option *options; /* List of options. */
    char **arg_next; /* Remaining arguments. */
    char *short_next; /* When non-null, unparsed short options. */
};
```
See also §587 and §588.
This code is included in §97.

The initialization function just creates and returns one of these structures:

§587 ⟨ Option parser 586 ⟩ +≡
```
/* Creates and returns a command-line options parser.
   args is a null-terminated array of command-line arguments, not
   including program name. */
static struct option_state *option_init (const struct option *options, char **args) {
    struct option_state *state;

    assert (options != NULL && args != NULL);

    state = xmalloc (sizeof *state);
    state->options = options;
    state->arg_next = args;
    state->short_next = NULL;

    return state;
}
```

The option retrieval function uses a couple of helper functions. The code is lengthy, but not complicated:

§588 ⟨ Option parser 586 ⟩ +≡
```
/* Parses a short option whose single-character name is pointed to by
   state->short_next. Advances past the option so that the next one
   will be parsed in the next call to option_get(). Sets *argp to
   the option's argument, if any. Returns the option's short name. */
static int handle_short_option (struct option_state *state, char **argp) {
    const struct option *o;

    assert (state != NULL && state->short_next != NULL && *state->short_next != '\0'
```

```
          && state→options != NULL);
    /* Find option in o. */
    for (o = state→options; ; o++)
        if (o→long_name == NULL)
            fail ("unknown option `%c'; use -help for help", *state→short_next);
        else if (o→short_name == *state→short_next)
            break;
    state→short_next++;
    /* Handle argument. */
    if (o→has_arg) {
        if (*state→arg_next == NULL || **state→arg_next == '')
            fail ("`%c' requires an argument; use -help for help");
        *argp = *state→arg_next++;
    }
    return o→short_name;
}
/* Parses a long option whose command-line argument is pointed to by
   *state→arg_next. Advances past the option so that the next one
   will be parsed in the next call to option_get(). Sets *argp to
   the option's argument, if any. Returns the option's identifier. */
static int handle_long_option (struct option_state *state, char **argp) {
    const struct option *o; /* Iterator on options. */
    char name[16]; /* Option name. */
    const char *arg; /* Option argument. */
    assert (state != NULL && state→arg_next != NULL && *state→arg_next != NULL
          && state→options != NULL && argp != NULL);
    /* Copy the option name into name
       and put a pointer to its argument, or NULL if none, into arg. */
    {
        const char *p = *state→arg_next + 2;
        const char *q = p + strcspn (p, "=");
        size_t name_len = q - p;
        if (name_len > (sizeof name) - 1)
            name_len = (sizeof name) - 1;
        memcpy (name, p, name_len);
        name[name_len] = '\0';
        arg = (*q == '=') ? q + 1 : NULL;
    }
    /* Find option in o. */
    for (o = state→options; ; o++)
        if (o→long_name == NULL)
            fail ("unknown option -%s; use -help for help", name);
        else if (!strcmp (name, o→long_name))
            break;
    /* Make sure option has an argument if it should. */
```

```
    if ((arg != NULL) != (o->has_arg != 0)) {
        if (arg != NULL)
            fail ("-%s can't take an argument; use -help for help", name);
        else fail ("-%s requires an argument; use -help for help", name);
    }

    /* Advance and return. */
    state->arg_next++;
    *argp = (char *) arg;
    return o->short_name;
}

/* Retrieves the next option in the state vector state.
   Returns the option's identifier, or -1 if out of options.
   Stores the option's argument, or NULL if none, into *argp. */
static int option_get (struct option_state *state, char **argp) {
    assert (state != NULL && argp != NULL);

    /* No argument by default. */
    *argp = NULL;

    /* Deal with left-over short options. */
    if (state->short_next != NULL) {
        if (*state->short_next != '\0')
            return handle_short_option (state, argp);
        else state->short_next = NULL;
    }

    /* Out of options? */
    if (*state->arg_next == NULL) {
        free (state);
        return -1;
    }

    /* Non-option arguments not supported. */
    if ((*state->arg_next)[0] != '')
        fail ("nonoption arguments encountered; use -help for help");
    if ((*state->arg_next)[1] == '\0')
        fail ("unknown option `'; use -help for help");

    /* Handle the option. */
    if ((*state->arg_next)[1] == '')
        return handle_long_option (state, argp);
    else {
        state->short_next = *state->arg_next + 1;
        state->arg_next++;
        return handle_short_option (state, argp);
    }
}
```

B.2 Command-Line Parser

The option parser in the previous section handles the general form of command-line options. The code in this section applies that option parser to the specific options used by the BST test program. It has helper functions for argument parsing and advice to users. Here is all of it together:

§589 ⟨ Command line parser 589 ⟩ ≡

```
/* Command line parser. */
/* If a is a prefix for b or vice versa, returns the length of the match.
   Otherwise, returns 0. */
size_t match_len (const char *a, const char *b) {
    size_t cnt;
    for (cnt = 0; *a == *b && *a != '\0'; a++, b++)
        cnt++;
    return (*a != *b && *a != '\0' && *b != '\0') ? 0 : cnt;
}
/* s should point to a decimal representation of an integer.
   Returns the value of s, if successful, or 0 on failure. */
static int stoi (const char *s) {
    long x = strtol (s, NULL, 10);
    return x >= INT_MIN && x <= INT_MAX ? x : 0;
}
/* Print helpful syntax message and exit. */
static void usage (void) {
    static const char *help[] = {
    "bsttest,␣unit␣test␣for␣GNU␣libavl.\n\n",
    "Usage:␣%s␣[OPTION]...\n\n",
    "In␣the␣option␣descriptions␣below,␣CAPITAL␣denote␣arguments.\n",
    "If␣a␣long␣option␣shows␣an␣argument␣as␣mandatory,␣then␣it␣is\n",
    "mandatory␣for␣the␣equivalent␣short␣option␣also.␣␣See␣the␣GNU\n",
    "libavl␣manual␣for␣more␣information.\n\n",
    "t,␣-test=TEST␣␣␣␣␣Sets␣test␣to␣perform.␣␣TEST␣is␣one␣of:\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣correctness␣insert/delete/...␣(default)\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣overflow␣␣␣␣stack␣overflow␣test\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣benchmark␣␣␣benchmark␣test\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣null␣␣␣␣␣␣␣␣no␣test\n",
    "s,␣-size=TREESIZE␣␣Sets␣tree␣size␣in␣nodes␣(default␣16).\n",
    "r,␣-repeat=COUNT␣␣Repeats␣operation␣COUNT␣times␣(default␣16).\n",
    "i,␣-insert=ORDER␣␣Sets␣the␣insertion␣order.␣␣ORDER␣is␣one␣of:\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣random␣␣␣␣␣␣random␣permutation␣(default)\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ascending␣␣␣ascending␣order␣0...n1\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣descending␣␣descending␣order␣n1...0\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣balanced␣␣␣␣balanced␣tree␣order\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zigzag␣␣␣␣␣␣zigzag␣tree\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ascshifted␣n/2...n1,␣0...n/21\n",
    "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣custom␣␣␣␣␣␣custom,␣read␣from␣stdin\n",
```

```
    "d, -delete=ORDER  Sets the deletion order.  ORDER is one of:\n",
    "                      random   random permutation (default)\n",
    "                      reverse  reverse order of insertion\n",
    "                      same     same as insertion order\n",
    "                      custom   custom, read from stdin\n",
    "a, -alloc=POLICY  Sets allocation policy.  POLICY is one of:\n",
    "                      track     track memory leaks (default)\n",
    "                      notrack  turn off leak detection\n",
    "                      failCNT  fail after CNT allocations\n",
    "                      fail%%PCT  fail random PCT%% of allocations\n",
    "                      subB,A   divide Bbyte blocks in Abyte units\n",
    "                    (Ignored for ‘benchmark’ test.)\n",
    "A, -incr=INC      Fail policies: arg increment per repetition.\n",
    "S, -seed=SEED     Sets initial number seed to SEED.\n",
    "                    (default based on system time)\n",
    "n, -nonstop       Don’t stop after a single error.\n",
    "q, -quiet         Turns down verbosity level.\n",
    "v, -verbose       Turns up verbosity level.\n",
    "h, -help          Displays this help screen.\n",
    "V, -version       Reports version and copyright information.\n",
    NULL,
};
    const char **p;
    for (p = help; *p != NULL; p++)
        printf (*p, pgm_name);
    exit (EXIT_SUCCESS);
}
/* Parses command-line arguments from null-terminated array args.
   Sets up options appropriately to correspond. */
static void parse_command_line (char **args, struct test_options *options) {
    static const struct option option_tab[] = {
        {"test", 't', 1}, {"insert", 'i', 1}, {"delete", 'd', 1},
        {"alloc", 'a', 1}, {"incr", 'A', 1}, {"size", 's', 1},
        {"repeat", 'r', 1}, {"operation", 'o', 1}, {"seed", 'S', 1},
        {"nonstop", 'n', 0}, {"quiet", 'q', 0}, {"verbose", 'v', 0},
        {"help", 'h', 0}, {"version", 'V', 0}, {NULL, 0, 0},
    };
    struct option_state *state;
    /* Default options. */
    options->test = TST_CORRECTNESS; options->insert_order = INS_RANDOM;
    options->delete_order = DEL_RANDOM; options->alloc_policy = MT_TRACK;
    options->alloc_arg[0] = 0; options->alloc_arg[1] = 0;
    options->alloc_incr = 0; options->node_cnt = 15;
    options->iter_cnt = 15; options->seed_given = 0;
    options->verbosity = 0; options->nonstop = 0;
    if (*args == NULL)
```

```
        return;
    state = option_init (option_tab, args + 1);
    for (;;) {
        char *arg;
        int id = option_get (state, &arg);
        if (id == -1)
            break;
        switch (id) {
            case 't':
                if (match_len (arg, "correctness") >= 3)
                    options->test = TST_CORRECTNESS;
                else if (match_len (arg, "overflow") >= 3)
                    options->test = TST_OVERFLOW;
                else if (match_len (arg, "null") >= 3)
                    options->test = TST_NULL;
                else
                    fail ("unknown test \"%s\"", arg);
                break;
            case 'i': {
                    static const char *orders[INS_CNT] = {
                        "random", "ascending", "descending",
                        "balanced", "zigzag", "ascshifted", "custom",
                    };
                    const char **iter;
                    assert (sizeof orders / sizeof *orders == INS_CNT);
                    for (iter = orders; ; iter++)
                        if (iter >= orders + INS_CNT)
                            fail ("unknown order \"%s\"", arg);
                        else if (match_len (*iter, arg) >= 3) {
                            options->insert_order = iter - orders;
                            break;
                        }
                }
                break;
            case 'd': {
                    static const char *orders[DEL_CNT] = {
                        "random", "reverse", "same", "custom",
                    };
                    const char **iter;
                    assert (sizeof orders / sizeof *orders == DEL_CNT);
                    for (iter = orders; ; iter++)
                        if (iter >= orders + DEL_CNT)
                            fail ("unknown order \"%s\"", arg);
                        else if (match_len (*iter, arg) >= 3) {
                            options->delete_order = iter - orders;
```

```
                break;
            }
        }
        break;
    case 'a':
        if (match_len (arg, "track") >= 3)
            options->alloc_policy = MT_TRACK;
        else if (match_len (arg, "notrack") >= 3)
            options->alloc_policy = MT_NO_TRACK;
        else if (!strncmp (arg, "fail", 3)) {
            const char *p = arg + strcspn (arg, "%");
            if (*p == '') options->alloc_policy = MT_FAIL_COUNT;
            else if (*p == '%') options->alloc_policy = MT_FAIL_PERCENT;
            else fail ("invalid allocation policy \"%s\"", arg);
            options->alloc_arg[0] = stoi (p + 1);
        }
        else if (!strncmp (arg, "suballoc", 3)) {
            const char *p = strchr (arg, '');
            const char *q = strchr (arg, ',');
            if (p == NULL || q == NULL)
                fail ("invalid allocation policy \"%s\"", arg);
            options->alloc_policy = MT_SUBALLOC;
            options->alloc_arg[0] = stoi (p + 1);
            options->alloc_arg[1] = stoi (q + 1);
            if (options->alloc_arg[MT_BLOCK_SIZE] < 32)
                fail ("block size too small");
            else if (options->alloc_arg[MT_ALIGN]
                    > options->alloc_arg[MT_BLOCK_SIZE])
                fail ("alignment cannot be greater than block size");
            else if (options->alloc_arg[MT_ALIGN] < 1)
                fail ("alignment must be at least 1");
        }
        break;
    case 'A': options->alloc_incr = stoi (arg); break;
    case 's':
        options->node_cnt = stoi (arg);
        if (options->node_cnt < 1)
            fail ("bad tree size \"%s\"", arg);
        break;
    case 'r':
        options->iter_cnt = stoi (arg);
        if (options->iter_cnt < 1)
            fail ("bad repeat count \"%s\"", arg);
        break;
    case 'S':
```

```
            options->seed_given = 1;
            options->seed = strtoul (arg, NULL, 0);
            break;
        case 'n': options->nonstop = 1; break;
        case 'q': options->verbosity--; break;
        case 'v': options->verbosity++; break;
        case 'h': usage (); break;
        case 'V':
            fputs ("GNU libavl 2.0.2\n"
                   "Copyright (C) 1998-2002, 2004 "
                   "Free Software Foundation, Inc.\n"
                   "This program comes with NO WARRANTY, not even for\n"
                   "MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.\n"
                   "You may redistribute copies under the terms of the\n"
                   "GNU General Public License.  For more information on\n"
                   "these matters, see the file named COPYING.\n",
                   stdout);
            exit (EXIT_SUCCESS);
        default: assert (0);
        }
    }
}
```

This code is included in §97.

Appendix C Glossary

adjacent: Two nodes in a **binary tree** are adjacent if one is the child of the other.

AVL tree: A type of **balanced tree**, where the AVL **balance factor** of each node is limited to $^-1$, 0, or $^+1$.

balance: To rearrange a **binary search tree** so that it has its minimum possible **height**, approximately the binary logarithm of its number of nodes.

balance condition: In a **balanced tree**, the additional rule or rules that limit the tree's height.

balance factor: For any node in an **AVL tree**, the difference between the **height** of the node's **right subtree** and **left subtree**.

balanced tree: A **binary search tree** along with a rule that limits the tree's height in order to avoid a **pathological case**. Types of balanced trees: **AVL tree**, **red-black tree**.

binary search: A technique for searching by comparison of keys, in which the search space roughly halves in size after each comparison step.

binary search tree: A **binary tree** with the additional property that the key in each node's left child is less than the node's key, and that the key in each node's right child is greater than the node's key. In **inorder traversal**, the items in a BST are visited in sorted order of their keys.

binary tree: A data structure that is either an **empty tree** or consists of a **root**, a **left subtree**, and a **right subtree**.

black box: Conceptually, a device whose input and output are defined but whose principles of internal operation is not specified.

black-height: In a **red-black tree**, the number of black nodes along a simple path from a given node down to a non-branching node. Due to **rule 2**, this is the same regardless of the path chosen.

BST: See **binary search tree**.

child: In a **binary tree**, a **left child** or **right child** of a node.

children: More than one **child**.

color: In a **red-black tree**, a property of a node, either red or black. Node colors in a red-black tree are constrained by **rule 1** and **rule 2**

complete binary tree: A **binary tree** in which every **simple path** from the root down to a leaf has the same length and every non-leaf node has two children.

compression: A transformation on a binary search tree used to **rebalance** (sense 2).

deep copy: In making a copy of a complex data structure, it is often possible to copy upper levels of data without copying lower levels. If all levels are copied nonetheless, it is a deep copy. See also **shallow copy**.

dynamic: 1. When speaking of data, data that can change or (in some contexts) varies quickly. 2. In C, memory allocation with *malloc*() and related functions. See also **static**.

empty tree: A binary tree without any nodes.

height: In a binary tree, the maximum number of nodes that can be visited starting at the tree's root and moving only downward. An an empty tree has height 0.

idempotent: Having the same effect as if used only once, even if used multiple times. C header files are usually designed to be idempotent.

inorder predecessor: The node preceding a given node in an **inorder traversal**.

inorder successor: The node following a given node in an **inorder traversal**.

inorder traversal: A type of binary tree **traversal** where the root's left subtree is traversed, then the root is visited, then the root's right subtree is traversed.

iteration: In C, repeating a sequence of statements without using recursive function calls, most typically accomplished using a **for** or **while** loop. Oppose **recursion**.

key: In a binary search tree, data stored in a **node** and used to order nodes.

leaf: A **node** whose **children** are empty.

left child: In a **binary tree**, the root of a node's left subtree, if that subtree is non-empty. A node that has an empty left subtree may be said to have no left child.

left rotation: See [rotation], page 333.

left subtree: Part of a non-empty **binary tree**.

left-threaded tree: A **binary search tree** augmented to simplify and speed up traversal in reverse of **inorder traversal**, but not traversal in the forward direction.

literate programming: A philosophy of programming that regards software as a type of literature, popularized by Donald Knuth through his works such as [Knuth 1992].

node: The basic element of a binary tree, consisting of a **key**, a **left child**, and a **right child**.

non-branching node: A node in a **binary tree** that has exactly zero or one non-empty children.

nonterminal node: A **node** with at least one nonempty **subtree**.

parent: When one node in a **binary tree** is the child of another, the first node. A node that is not the child of any other node has no parent.

parent pointer: A pointer within a node to its **parent** node.

pathological case: In a **binary search tree** context, a BST whose **height** is much greater than the minimum possible. Avoidable through use of **balanced tree** techniques.

path: In a **binary tree**, a list of nodes such that, for each pair of nodes appearing adjacent in the list, one of the nodes is the parent of the other.

postorder traversal: A type of binary tree **traversal** where the root's left subtree is traversed, then the root's right subtree is traversed, then the root is visited.

preorder traversal: A type of binary tree **traversal** where the root is visited, then the root's left subtree is traversed, then the root's right subtree is traversed.

rebalance: 1. After an operation that modifies a **balanced tree**, to restore the tree's **balance condition**, typically by **rotation** or, in a **red-black tree**, changing the **color** of one or more nodes. 2. To reorganize a **binary search tree** so that its shape more closely approximates that of a **complete binary tree**.

recursion: In C, describes a function that calls itself directly or indirectly. See also **tail recursion**. Oppose **iteration**.

red-black tree: A form of **balanced tree** where each node has a **color** and these colors are laid out such that they satisfy **rule 1** and **rule 2** for red-black trees.

right child: In a **binary tree**, the root of a node's right subtree, if that subtree is non-empty. A node that has an empty right subtree may be said to have no right child.

right rotation: See [rotation], page 333.

right subtree: Part of a non-empty **binary tree**.

right-threaded tree: A **binary search tree** augmented to simplify and speed up **inorder traversal**, but not traversal in the reverse order.

rotation: A particular type of simple transformation on a **binary search tree** that changes local structure without changing **inorder traversal** ordering. See Section 4.3 [BST Rotations], page 33, Section 8.2 [TBST Rotations], page 192, Section 11.3 [RTBST Rotations], page 248, and Section 14.2 [PBST Rotations], page 294, for more details.

root: A **node** taken as a **binary tree** in its own right. Every node is the root of a binary tree, but "root" is most often used to refer to a node that is not a **child** of any other node.

rule 1: One of the rules governing layout of node colors in a **red-black tree**: no red node may have a red child. See Section 6.1 [RB Balancing Rule], page 139.

rule 2: One of the rules governing layout of node colors in a **red-black tree**: every **simple path** from a given node to one of its **non-branching node** descendants contains the same number of black nodes. See Section 6.1 [RB Balancing Rule], page 139.

sentinel: In the context of searching in a data structure, a piece of data used to avoid an explicit test for a null pointer, the end of an array, etc., typically by setting its value to that of the looked-for data item.

sequential search: A technique for searching by comparison of keys, in which the search space is typically reduced only by one item for each comparison.

sequential search with sentinel: A **sequential search** in a search space set up with a **sentinel**.

shallow copy: In making a copy of a complex data structure, it is often possible to copy upper levels of data without copying lower levels. If lower levels are indeed shared, it is a shallow copy. See also **deep copy**.

simple path: A **path** that does not include any node more than once.

static: 1. When speaking of data, data that is invariant or (in some contexts) changes rarely. 2. In C, memory allocation other than that done with *malloc*() and related functions. 3. In C, a keyword used for a variety of purposes, some of which are related to sense 2. See also **dynamic**.

subtree: A **binary tree** that is itself a child of some **node**.

symmetric traversal: **inorder traversal**.

tag: A field in a **threaded tree** node used to distinguish a **thread** from a **child** pointer.

tail recursion: A form of **recursion** where a function calls itself as its last action. If the function is non-**void**, the outer call must also return to its caller the value returned by the inner call in order to be tail recursive.

terminal node: A node with no **left child** or **right child**.

thread: In a **threaded tree**, a pointer to the predecessor or successor of a **node**, replacing a child pointer that would otherwise be null. Distinguished from an ordinary child pointer using a **tag**.

threaded tree: A form of **binary search tree** augmented to simplify **inorder traversal**. See also **thread**, **tag**.

traversal: To **visit** each of the nodes in a **binary tree** according to some scheme based on the tree's structure. See **inorder traversal**, **preorder traversal**, **postorder traversal**.

undefined behavior: In C, a situation to which the computer's response is unpredictable. It is frequently noted that, when undefined behavior is invoked, it is legal for the compiler to "make demons fly out of your nose."

value: Often kept in a **node** along with the **key**, a value is auxiliary data not used to determine ordering of nodes.

vine: A degenerate **binary tree**, resembling a linked list, in which each node has at most one child.

visit: During **traversal**, to perform an operation on a node, such as to display its value or free its associated memory.

Appendix D Answers to All the Exercises

Chapter 2

Section 2.1

1. If the table is not a dictionary, then we can just include a count along with each item recording the number of copies of it that would otherwise be included in the table. If the table is a dictionary, then each data item can include a single key and possibly multiple values.

Section 2.2

1. Only macro parameter names can safely appear prefixless. Macro parameter names are significant only in a scope from their declaration to the end of the macro definition. Macro parameters may even be named as otherwise reserved C keywords such as **int** and **while**, although this is a bad idea.

The main reason that the other kinds of identifiers must be prefixed is the possibility of a macro having the same name. A surprise macro expansion in the midst of a function prototype can lead to puzzling compiler diagnostics.

2. The capitalized equivalent is `ERR_`, which is a reserved identifier. All identifiers that begin with an uppercase '`E`' followed by a digit or capital letter are reserved in many contexts. It is best to avoid them entirely. There are other identifiers to avoid, too. The article cited below has a handy list.

See also: [Brown 2001].

Section 2.3

1. C does not guarantee that an integer cast to a pointer and back retains its value. In addition, there's a chance that an integer cast to a pointer becomes the null pointer value. This latter is not limited to integers with value 0. On the other hand, a nonconstant integer with value 0 is not guaranteed to become a null pointer when cast.

Such a technique is only acceptable when the machine that the code is to run on is known in advance. At best it is inelegant. At worst, it will cause erroneous behavior.

See also: [Summit 1999], section 5; [ISO 1990], sections 6.2.2.3 and 6.3.4; [ISO 1999], section 6.3.2.3.

2. This definition would only cause problems if the subtraction overflowed. It would be acceptable if it was known that the values to be compared would always be in a small enough range that overflow would never occur.

Here are two more "clever" definitions for *compare_ints*() that work in all cases:

```
/* Credit: GNU C library reference manual. */
int compare_ints (const void *pa, const void *pb, void *param) {
    const int *a = pa;
```

```
    const int *b = pb;
    return (*a > *b) - (*a < *b);
}
int compare_ints (const void *pa, const void *pb, void *param) {
    const int *a = pa;
    const int *b = pb;
    return (*a < *b) ? -1 : (*a > *b);
}
```

3. No. Not only does *strcmp*() take parameters of different types (**const char** ∗s instead of **const void** ∗s), our comparison functions take an additional parameter. Functions *strcmp*() and *compare_strings*() are not compatible.

4.

```
int compare_fixed_strings (const void *pa, const void *pb, void *param) {
    return memcmp (pa, pb, *(size_t *) param);
}
```

5a. Here's the blow-by-blow rundown:

- Irreflexivity: $a == a$ is always true for integers.
- Antisymmetry: If $a > b$ then $b < a$ for integers.
- Transitivity: If $a > b$ and $b > c$ then $a > c$ for integers.
- Transitivity of equivalence: If $a == b$ and $b == c$, then $a == c$ for integers.

5b. Yes, *strcmp*() satisfies all of the points above.

5c. Consider the domain of pairs of integers ($x0$,$x1$) with $x1 \geq x0$. Pair x, composed of ($x0$,$x1$), is less than pair y, composed of ($y0$,$y1$), if $x1 < y0$. Alternatively, pair x is greater than pair y if $x0 > y1$. Otherwise, the pairs are equal.

This rule is irreflexive: for any given pair a, neither $a1 < a0$ nor $a0 > a1$, so $a == a$. It is antisymmetic: $a > b$ implies $a0 > b1$, therefore $b1 < a0$, and therefore $b < a$. It is transitive: $a > b$ implies $a0 > b1$, $b > c$ implies $b0 > c1$, and we know that $b1 > b0$, so $a0 > b1 > b0 > c1$ and $a > c$. It does not have transitivity of equivalence: suppose that we have $a \equiv$ (1,2), $b \equiv$ (2,3), $c \equiv$ (3,4). Then, $a == b$ and $b == c$, but not $a == c$.

A form of augmented binary search tree, called an "interval tree", *can* be used to efficiently handle this data type. The references have more details.

See also: [Cormen 1990], section 15.3.

6a. $!f(a, b)$ && $!f(b, a)$ and $!f(a, b)$ && $f(b, a)$.

6b.

```
static int bin_cmp (const void *a, const void *b, void *param, bst_comparison_func tern) {
    return tern (a, b, param) < 0;
}
```

6c. This problem presents an interesting tradeoff. We must choose between sometimes calling the comparison function twice per item to convert our $\geq$ knowledge into $>$ or $\equiv$, or always traversing all the way to a leaf node, then making a final call to decide on equality. The former choice doesn't provide any new insight, so we choose the latter here.

In the code below, p traverses the tree and q keeps track of the current candidate for a match to *item*. If the item in p is less than *item*, then the matching item, if any, must be in the left subtree of p, and we leave q as it was. Otherwise, the item in p is greater than or equal to p and then matching item, if any, is either p itself or in its right subtree, so we set q to the potential match. When we run off the bottom of the tree, we check whether q is really a match by making one additional comparison.

```
void *bst_find (const struct bst_table *tree, const void *item) {
    const struct bst_node *p;
    void *q;
    assert (tree != NULL && item != NULL);
    p = tree→bst_root;
    q = NULL;
    while (p != NULL)
        if (!bin_cmp (p→bst_data, item, tree→bst_param, tree→bst_compare)) {
            q = p→bst_data;
            p = p→bst_link[0];
        }
        else p = p→bst_link[1];
    if (q != NULL && !bin_cmp (item, q, tree→bst_param, tree→bst_compare))
        return q;
    else return NULL;
}
```

Section 2.5

1. It's not necessary, for reasons of the C definition of type compatibility. Within a C source file (more technically, a "translation unit"), two structures are compatible only if they are the same structure, regardless of how similar their members may be, so hypothetical structures **struct bst_allocator** and **struct avl_allocator** couldn't be mixed together without nasty-smelling casts. On the other hand, prototyped function types are compatible if they have compatible return types and compatible parameter types, so **bst_item_func** and **avl_item_func** (say) are interchangeable.

2. This allocator uses the same function *tbl_free*() as *tbl_allocator_default*.

§590 ⟨ Aborting allocator 590 ⟩ ≡

```
/* Allocates size bytes of space using malloc().
   Aborts if out of memory. */
void *tbl_malloc_abort (struct libavl_allocator *allocator, size_t size) {
    void *block;
    assert (allocator != NULL && size > 0);
    block = malloc (size);
    if (block != NULL)
        return block;
    fprintf (stderr, "out of memory\n");
    exit (EXIT_FAILURE);
}
```

```
struct libavl_allocator tbl_allocator_abort = {
        tbl_malloc_abort, tbl_free};
```

3. Define a wrapper structure with **struct libavl_allocator** as its first member. For instance, a hypothetical pool allocator might look like this:

```
struct pool_allocator {
    struct libavl_allocator suballocator;
    struct pool *pool;
};
```

Because a pointer to the first member of a structure is a pointer to the structure itself, and vice versa, the allocate and free functions can use a cast to access the larger **struct pool_allocator** given a pointer to **struct libavl_allocator**. If we assume the existence of functions *pool_malloc*() and *pool_free*() to allocate and free memory within a pool, then we can define the functions for **struct pool_allocator**'s *suballocator* like this:

```
void *pool_allocator_malloc (struct libavl_allocator *allocator, size_t size) {
    struct pool_allocator *pa = (struct pool_allocator *) allocator;
    return pool_malloc (pa->pool, size);
}
void pool_allocator_free (struct libavl_allocator *allocator, void *ptr) {
    struct pool_allocator *pa = (struct pool_allocator *) allocator;
    pool_free (pa->pool, ptr);
}
```

Finally, we want to actually allocate a table inside a pool. The following function does this. Notice the way that it uses the pool to store the **struct pool_allocator** as well; this trick comes in handy sometimes.

```
struct tbl_table *pool_allocator_tbl_create (struct tbl_pool *pool) {
    struct pool_allocator *pa = pool_malloc (pool, sizeof *pa);
    if (pa == NULL)
        return NULL;
    pa->suballocator.tbl_malloc = pool_allocator_malloc;
    pa->suballocator.tbl_free = pool_allocator_free;
    pa->pool = pool;
    return tbl_create (compare_ints, NULL, &pa->suballocator);
}
```

Section 2.7

1. Notice the cast to **size_t** in the macro definition below. This prevents the result of *tbl_count*() from being used as an lvalue (that is, on the left side of an assignment operator), because the result of a cast is never an lvalue.

§591 ⟨ Table count macro 591 ⟩ ≡

```
#define tbl_count(table) ((size_t) (table)->tbl_count)
```

This code is included in §15.

Another way to get the same effect is to use the unary + operator, like this:

```
#define tbl_count(table) (+(table)->tbl_count)
```

See also: [ISO 1990], section 6.3.4; [Kernighan 1988], section A7.5.

Section 2.8

1. If a memory allocation function that never returns a null pointer is used, then it is reasonable to use these functions. For instance, *tbl_allocator_abort* from Exercise 2.5-2 is such an allocator.

2. Among other reasons, *tbl_find*() returns a null pointer to indicate that no matching item was found in the table. Null pointers in the table could therefore lead to confusing results. It is better to entirely prevent them from being inserted.

3.

§592 ⟨ Table insertion convenience functions 592 ⟩ ≡

```
void *tbl_insert (struct tbl_table *table, void *item) {
    void **p = tbl_probe (table, item);
    return p == NULL || *p == item ? NULL : *p;
}
void *tbl_replace (struct tbl_table *table, void *item) {
    void **p = tbl_probe (table, item);
    if (p == NULL || *p == item)
        return NULL;
    else {
        void *r = *p;
        *p = item;
        return r;
    }
}
```

This code is included in §29, §145, §196, §251, §300, §336, §375, §418, §455, §489, §522, and §554.

Section 2.9

1. Keep in mind that these directives have to be processed every time the header file is included. (Typical header file are designed to be "idempotent", i.e., processed by the compiler only on first inclusion and skipped on any later inclusions, because some C constructs cause errors if they are encountered twice during a compilation.)

§593 ⟨ Table assertion function control directives 593 ⟩ ≡

```
/* Table assertion functions. */
#ifndef NDEBUG
#undef tbl_assert_insert
#undef tbl_assert_delete
#else
#define tbl_assert_insert(table, item) tbl_insert (table, item)
#define tbl_assert_delete(table, item) tbl_delete (table, item)
#endif
```

This code is included in §24.

See also: [Summit 1999], section 10.7.

2. *tbl_assert_insert*() must be based on *tbl_probe*(), because *tbl_insert*() does not distinguish in its return value between successful insertion and memory allocation errors.

Assertions must be enabled for these functions because we want them to verify success if assertions were enabled at the point from which they were called, not if assertions were enabled when the table was compiled.

Notice the parentheses around the assertion function names before. The parentheses prevent the macros by the same name from being expanded. A function-like macro is only expanded when its name is followed by a left parenthesis, and the extra set of parentheses prevents this from being the case. Alternatively **#undef** directives could be used to achieve the same effect.

§594 ⟨ Table assertion functions 594 ⟩ ≡
```
#undef NDEBUG
#include <assert.h>
void (tbl_assert_insert) (struct tbl_table *table, void *item) {
    void **p = tbl_probe (table, item);
    assert (p != NULL && *p == item);
}
void *(tbl_assert_delete) (struct tbl_table *table, void *item) {
    void *p = tbl_delete (table, item);
    assert (p != NULL);
    return p;
}
```
This code is included in §29, §145, §196, §251, §300, §336, §375, §418, §455, §489, §522, and §554.

3. The *assert*() macro is meant for testing for design errors and "impossible" conditions, not runtime errors like disk input/output errors or memory allocation failures. If the memory allocator can fail, then the *assert*() call in *tbl_assert_insert*() effectively does this.

See also: [Summit 1999], section 20.24b.

Section 2.12

1. Both tables and *set*s store sorted arrangements of unique items. Both require a strict weak ordering on the items that they contain. Libavl uses ternary comparison functions whereas the STL uses binary comparison functions (see Exercise 2.3-6).

The description of tables here doesn't list any particular speed requirements for operations, whereas STL *set*s are constrained in the complexity of their operations. It's worth noting, however, that the Libavl implementation of AVL and RB trees meet all of the STL complexity requirements, for their equivalent operations, except one. The exception is that *set* methods *begin*() and *rbegin*() must have constant-time complexity, whereas the equivalent Libavl functions **_t_first*() and **_t_last*() on AVL and RB trees have logarithmic complexity.

Libavl traversers and STL iterators have similar semantics. Both remain valid if new items are inserted, and both remain valid if old items are deleted, unless it's the iterator's current item that's deleted.

The STL has a more complete selection of methods than Libavl does of table functions, but many of the additional ones (e.g., *distance*() or *erase*() each with two iterators as arguments) can be implemented easily in terms of existing Libavl functions. These might benefit from optimization possible with specialized implementations, but may not be worth it. The SGI/HP implementation of the STL does not contain any such optimization.

See also: [ISO 1998], sections 23.1, 23.1.2, and 23.3.3.

2. The nonessential functions are:

- *tbl_probe*(), *tbl_insert*(), and *tbl_replace*(), which can be implemented in terms of *tbl_t_insert*() and *tbl_t_replace*().
- *tbl_find*(), which can be implemented in terms of *tbl_t_find*().
- *tbl_assert_insert*() and *tbl_assert_delete*().
- *tbl_t_first*() and *tbl_t_last*(), which can be implemented with *tbl_t_init*() and *tbl_t_next*().

If we allow it to know what allocator was used for the original table, which is, strictly speaking, cheating, then we can also implement *tbl_copy*() in terms of *tbl_create*(), *tbl_t_insert*(), and *tbl_destroy*(). Under similar restrictions we can also implement *tbl_t_prev*() and *tbl_t_copy*() in terms of *tbl_t_init*() and *tbl_t_next*(), though in a very inefficient way.

Chapter 3

Section 3.1

1. The following program can be improved in many ways. However, we will implement a much better testing framework later, so this is fine for now.

```
§595 ⟨seq-test.c 595⟩ ≡
⟨License 1⟩
#include ⟨stdio.h⟩
#define MAX_INPUT 1024
⟨Sequentially search an array of ints 16⟩
int main (void) {
    int array[MAX_INPUT];
    int n, i;
    for (n = 0; n < MAX_INPUT; n++)
        if (scanf ("%d", &array[n]) != 1)
            break;
    for (i = 0; i < n; i++) {
        int result = seq_search (array, n, array[i]);
        if (result != i)
            printf ("seq_search() returned %d looking for %d  expected %d\n",
                    result, array[i], i);
    }
    return 0;
}
```

Section 3.4

1. Some types don't have a largest possible value; e.g., arbitrary-length strings.

Section 3.5

1. Knuth's name for this procedure is "uniform binary search." The code below is an almost-literal implementation of his Algorithm U. The fact that Knuth's arrays are 1-based, but C arrays are 0-based, accounts for most of the differences.

The code below uses **for** (;;) to assemble an "infinite" loop, a common C idiom.

§596 ⟨ Uniform binary search of ordered array 596 ⟩ ≡

```
/* Returns the offset within array[] of an element equal to key, or −1 if key is not in array[].
   array[] must be an array of n ints sorted in ascending order, with array[−1] modifiable. */
int uniform_binary_search (int array[], int n, int key) {
  int i = (n + 1) / 2 − 1;
  int m = n / 2;
  array[−1] = INT_MIN;
  for (;;) {
    if (key < array[i]) {
      if (m == 0)
        return −1;
      i −= (m + 1) / 2;
      m /= 2;
    }
    else if (key > array[i]) {
      if (m == 0)
        return −1;
      i += (m + 1) / 2;
      m /= 2;
    }
    else return i >= 0 ? i : −1;
  }
}
```

This code is included in §600.

See also: [Knuth 1998b], section 6.2.1, Algorithm U.

2a. This actually uses *blp_bsearch*(), implemented in part (b) below, in order to allow that function to be tested. You can replace the reference to *blp_bsearch*() by *bsearch*() without problem.

§597 ⟨ Binary search using *bsearch*() 597 ⟩ ≡

```
⟨ blp's implementation of bsearch() 598 ⟩
/* Compares the ints pointed to by pa and pb and returns positive
   if *pa > *pb, negative if *pa < *pb, or zero if *pa == *pb. */
static int compare_ints (const void *pa, const void *pb) {
  const int *a = pa;
  const int *b = pb;
  if (*a > *b) return 1;
  else if (*a < *b) return −1;
  else return 0;
}
```

```
/* Returns the offset within array[] of an element equal to key, or -1 if key is not in array[].
   array[] must be an array of n ints sorted in ascending order. */
static int binary_search_bsearch (int array[], int n, int key) {
    int *p = blp_bsearch (&key, array, n, sizeof *array, compare_ints);
    return p != NULL ? p - array : -1;
}
```

This code is included in §600.

2b. This function is named using the author of this book's initials. Note that the implementation below assumes that *count*, a **size_t**, won't exceed the range of an **int**. Some systems provide a type called **ssize_t** for this purpose, but we won't assume that here. (**long** is perhaps a better choice than **int**.)

§598 ⟨ blp's implementation of *bsearch*() 598 ⟩ ≡

```
/* Plug-compatible with standard C library bsearch(). */
static void *blp_bsearch (const void *key, const void *array, size_t count,
                          size_t size, int (*compare) (const void *, const void *)) {
    int min = 0;
    int max = count;

    while (max >= min) {
        int i = (min + max) / 2;
        void *item = ((char *) array) + size * i;
        int cmp = compare (key, item);

        if (cmp < 0) max = i - 1;
        else if (cmp > 0) min = i + 1;
        else return item;
    }

    return NULL;
}
```

This code is included in §597.

3. Here's an outline of the entire program:

§599 ⟨ `srch-test.c` 599 ⟩ ≡

```
⟨ License 1 ⟩
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

⟨ Search functions 600 ⟩
⟨ Array of search functions 601 ⟩

⟨ Timer functions 604 ⟩
⟨ Search test functions 606 ⟩
⟨ Search test main program 609 ⟩
```

We need to include all the search functions we're going to use:

§600 ⟨ Search functions 600 ⟩ ≡

```
⟨ Sequentially search an array of ints 16 ⟩
⟨ Sequentially search an array of ints using a sentinel 17 ⟩
```

⟨ Sequentially search a sorted array of **int**s 18 ⟩
⟨ Sequentially search a sorted array of **int**s using a sentinel 19 ⟩
⟨ Sequentially search a sorted array of **int**s using a sentinel (2) 20 ⟩
⟨ Binary search of ordered array 21 ⟩
⟨ Uniform binary search of ordered array 596 ⟩
⟨ Binary search using *bsearch*() 597 ⟩
⟨ Cheating search 603 ⟩

This code is included in §599.

We need to make a list of the search functions. We start by defining the array's element type:

§601 ⟨ Array of search functions 601 ⟩ ≡

```
/* Description of a search function. */
struct search_func {
    const char *name;
    int (*search) (int array[], int n, int key);
};
```

See also §602.
This code is included in §599.

Then we define the list as an array:

§602 ⟨ Array of search functions 601 ⟩ +≡

```
/* Array of all the search functions we know. */
struct search_func search_func_tab[] = {
    {"seq_search()", seq_search},
    {"seq_sentinel_search()", seq_sentinel_search},
    {"seq_sorted_search()", seq_sorted_search},
    {"seq_sorted_sentinel_search()", seq_sorted_sentinel_search},
    {"seq_sorted_sentinel_search_2()", seq_sorted_sentinel_search_2},
    {"binary_search()", binary_search},
    {"uniform_binary_search()", uniform_binary_search},
    {"binary_search_bsearch()", binary_search_bsearch},
    {"cheat_search()", cheat_search},
};
/* Number of search functions. */
const size_t n_search_func = sizeof search_func_tab / sizeof *search_func_tab;
```

We've added previously unseen function *cheat_search*() to the array. This is a function that "cheats" on the search because it knows that we are only going to search in a array such that *array*[*i*] == *i*. The purpose of *cheat_search*() is to allow us to find out how much of the search time is overhead imposed by the framework and the function calls and how much is actual search time. Here's *cheat_search*():

§603 ⟨ Cheating search 603 ⟩ ≡

```
/* Cheating search function that knows that array[i] == i.
   n must be the array size and key the item to search for.
   array[] is not used.
   Returns the index in array[] where key is found, or −1 if key is not in array[]. */
int cheat_search (int array[], int n, int key) {
```

```
    return key >= 0 && key < n ? key : -1;
}
```

This code is included in §600.

We're going to need some functions for timing operations. First, a function to "start" a timer:

§604 ⟨ Timer functions 604 ⟩ ≡

```
/* "Starts" a timer by recording the current time in *t. */
static void start_timer (clock_t *t) {
    clock_t now = clock ();
    while (now == clock ())
        /* Do nothing. */;
    *t = clock ();
}
```

See also §605.

This code is included in §599.

Function *start_timer*() waits for the value returned by *clock*() to change before it records the value. On systems with a slow timer (such as PCs running MS-DOS, where the clock ticks only 18.2 times per second), this gives more stable timing results because it means that timing always starts near the beginning of a clock tick.

We also need a function to "stop" the timer and report the results:

§605 ⟨ Timer functions 604 ⟩ +≡

```
/* Prints the elapsed time since start, set by start_timer(). */
static void stop_timer (clock_t start) {
    clock_t end = clock ();
    printf ("%.2f seconds\n", ((double) (end - start)) / CLOCKS_PER_SEC);
}
```

The value reported by *clock*() can "wrap around" to zero from a large value. *stop_timer*() does not allow for this possibility.

We will write three tests for the search functions. The first of these just checks that the search function works properly:

§606 ⟨ Search test functions 606 ⟩ ≡

```
/* Tests that f->search returns expect when called to search for key within array[],
   which has n elements such that array[i] == i. */
static void test_search_func_at (struct search_func *f, int array[], int n,
                                 int key, int expect) {
    int result = f->search (array, n, key);
    if (result != expect)
        printf ("%s returned %d looking for %d  expected %d\n",
                f->name, result, key, expect);
}

/* Tests searches for each element in array[] having n elements such that array[i] == i,
   and some unsuccessful searches too, all using function f->search. */
static void test_search_func (struct search_func *f, int array[], int n) {
    static const int shouldnt_find[] = {INT_MIN, -20, -1, INT_MAX};
```

```
    int i;
    printf ("Testing␣integrity␣of␣%s...␣␣", f→name);
    fflush (stdout);
    /∗ Verify that the function finds values that it should. ∗/
    for (i = 0; i < n; i++)
        test_search_func_at (f, array, n, i, i);
    /∗ Verify that the function doesn't find values it shouldn't. ∗/
    for (i = 0; i < (int) (sizeof shouldnt_find / sizeof ∗shouldnt_find); i++)
        test_search_func_at (f, array, n, shouldnt_find[i], −1);
    printf ("done\n");
}
```

See also §607 and §608.
This code is included in §599.

The second test function finds the time required for searching for elements in the array:

§607 ⟨ Search test functions 606 ⟩ +≡

```
/∗ Times a search for each element in array[] having n elements such that
   array[i] == i, repeated n_iter times, using function f→search. ∗/
static void time_successful_search (struct search_func ∗f, int array[], int n, int n_iter) {
    clock_t timer;
    printf ("Timing␣%d␣sets␣of␣successful␣searches...␣␣", n_iter);
    fflush (stdout);
    start_timer (&timer);
    while (n_iter−− > 0) {
        int i;
        for (i = 0; i < n; i++)
            f→search (array, n, i);
    }
    stop_timer (timer);
}
```

The last test function finds the time required for searching for values that don't appear in the array:

§608 ⟨ Search test functions 606 ⟩ +≡

```
/∗ Times n search for elements not in array[] having n elements such that
   array[i] == i, repeated n_iter times, using function f→search. ∗/
static void time_unsuccessful_search (struct search_func ∗f, int array[], int n, int n_iter) {
    clock_t timer;
    printf ("Timing␣%d␣sets␣of␣unsuccessful␣searches...␣␣", n_iter);
    fflush (stdout);
    start_timer (&timer);
    while (n_iter−− > 0) {
        int i;
        for (i = 0; i < n; i++)
            f→search (array, n, −i);
    }
```

```
    stop_timer (timer);
}
```

Here's the main program:

§609 ⟨ Search test main program 609 ⟩ ≡

```
⟨ Usage printer for search test program 615 ⟩
⟨ String to integer function stoi() 611 ⟩
int main (int argc, char *argv[]) {
    struct search_func *f; /* Search function. */
    int *array, n; /* Array and its size. */
    int n_iter; /* Number of iterations. */
    ⟨ Parse search test command line 610 ⟩
    ⟨ Initialize search test array 612 ⟩
    ⟨ Run search tests 613 ⟩
    ⟨ Clean up after search tests 614 ⟩
    return 0;
}
```

This code is included in §599.

§610 ⟨ Parse search test command line 610 ⟩ ≡

```
if (argc != 4) usage ();
{
    long algorithm = stoi (argv[1]) - 1;
    if (algorithm < 0 || algorithm > (long) n_search_func) usage ();
    f = &search_func_tab[algorithm];
}
n = stoi (argv[2]);
n_iter = stoi (argv[3]);
if (n < 1 || n_iter < 1) usage ();
```

This code is included in §609.

§611 ⟨ String to integer function stoi() 611 ⟩ ≡

```
/* s should point to a decimal representation of an integer.
   Returns the value of s, if successful, or 0 on failure. */
static int stoi (const char *s) {
    long x = strtol (s, NULL, 10);
    return x >= INT_MIN && x <= INT_MAX ? x : 0;
}
```

This code is included in §609 and §617.

When reading the code below, keep in mind that some of our algorithms use a sentinel at the end and some use a sentinel at the beginning, so we allocate two extra integers and take the middle part.

§612 ⟨ Initialize search test array 612 ⟩ ≡

```
array = malloc ((n + 2) * sizeof *array);
if (array == NULL) {
    fprintf (stderr, "out of memory\n");
    exit (EXIT_FAILURE);
```

```
}
array++;
{
    int i;
    for (i = 0; i < n; i++)
        array[i] = i;
}
```

This code is included in §609.

§613 ⟨ Run search tests 613 ⟩ ≡

```
test_search_func (f, array, n);
time_successful_search (f, array, n, n_iter);
time_unsuccessful_search (f, array, n, n_iter);
```

This code is included in §609.

§614 ⟨ Clean up after search tests 614 ⟩ ≡

```
free (array − 1);
```

This code is included in §609.

§615 ⟨ Usage printer for search test program 615 ⟩ ≡

```
/* Prints a message to the console explaining how to use this program. */
static void usage (void) {
    size_t i;
    fputs ("usage: srchtest <algorithm> <arraysize> <niterations>\n"
           "where <algorithm> is one of the following:\n", stdout);
    for (i = 0; i < n_search_func; i++)
        printf ("        %u for %s\n", (unsigned) i + 1, search_func_tab[i].name);
    fputs ("      <arraysize> is the size of the array to search, and\n"
           "      <niterations> is the number of times to iterate.\n", stdout);
    exit (EXIT_FAILURE);
}
```

This code is included in §609.

4. Here are the results on the author's computer, a Pentium II at 233 MHz, using GNU C 2.95.2, for 1024 iterations using arrays of size 1024 with no optimization. All values are given in seconds rounded to tenths.

Function	Successful searches	Unsuccessful searches
seq_search()	18.4	36.3
seq_sentinel_search()	16.5	32.8
seq_sorted_search()	18.6	0.1
seq_sorted_sentinel_search()	16.4	0.2
seq_sorted_sentinel_search_2()	16.6	0.2
binary_search()	1.3	1.2
uniform_binary_search()	1.1	1.1
binary_search_bsearch()	2.6	2.4
cheat_search()	0.1	0.1

Results of similar tests using full optimization were as follows:

Function	**Successful searches**	**Unsuccessful searches**
seq_search()	6.3	12.4
seq_sentinel_search()	4.8	9.4
seq_sorted_search()	9.3	0.1
seq_sorted_sentinel_search()	4.8	0.2
seq_sorted_sentinel_search_2()	4.8	0.2
binary_search()	0.7	0.5
uniform_binary_search()	0.7	0.6
binary_search_bsearch()	1.5	1.2
cheat_search()	0.1	0.1

Observations:

- In general, the times above are about what we might expect them to be: they decrease as we go down the table.
- Within sequential searches, the sentinel-based searches have better search times than non-sentinel searches, and other search characteristics (whether the array was sorted, for instance) had little impact on performance.
- Unsuccessful searches were very fast for sorted sequential searches, but the particular test set used always allowed such searches to terminate after a single comparison. For other test sets one might expect these numbers to be similar to those for unordered sequential search.
- Either of the first two forms of binary search had the best overall performance. They also have the best performance for successful searches and might be expected to have the best performance for unsuccessful searches in other test sets, for the reason given before.
- Binary search using the general interface *bsearch*() was significantly slower than either of the other binary searches, probably because of the cost of the extra function calls. Items that are more expensive to compare (for instance, long text strings) might be expected to show less of a penalty.

Here are the results on the same machine for 1,048,576 iterations on arrays of size 8 with full optimization:

Function	**Successful searches**	**Unsuccessful searches**
seq_search()	1.7	2.0
seq_sentinel_search()	1.7	2.0
seq_sorted_search()	2.0	1.1
seq_sorted_sentinel_search()	1.9	1.1
seq_sorted_sentinel_search_2()	1.8	1.2
binary_search()	2.5	1.9
uniform_binary_search()	2.4	2.3
binary_search_bsearch()	4.5	3.9
cheat_search()	0.7	0.7

For arrays this small, simple algorithms are the clear winners. The additional complications of binary search make it slower. Similar patterns can be expected on most architectures, although the "break even" array size where binary search and sequential search are equally fast can be expected to differ.

Section 3.6

1. Here is one easy way to do it:

§616 ⟨ Initialize *smaller* and *larger* within binary search tree 616 ⟩ ≡

```
/* Initializes larger and smaller within range min...max of array[],
   which has n real elements plus a (n + 1)th sentinel element. */
int init_binary_tree_array (struct binary_tree_entry array[], int n, int min, int max) {
  if (min <= max) {
    /* The '+ 1' is necessary because the tree root must be at n / 2,
       and on the first call we have min == 0 and max == n - 1. */
    int i = (min + max + 1) / 2;
    array[i].larger = init_binary_tree_array (array, n, i + 1, max);
    array[i].smaller = init_binary_tree_array (array, n, min, i - 1);
    return i;
  }
  else return n;
}
```

This code is included in §617.

2.

§617 ⟨ `bin-ary-test.c` 617 ⟩ ≡

```
⟨ License 1 ⟩
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
⟨ Binary search tree entry 22 ⟩
⟨ Search of binary search tree stored as array 23 ⟩
⟨ Initialize smaller and larger within binary search tree 616 ⟩
⟨ Show 'bin-ary-test' usage message 619 ⟩
⟨ String to integer function stoi() 611 ⟩
⟨ Main program to test binary_search_tree_array() 618 ⟩
```

§618 ⟨ Main program to test *binary_search_tree_array*() 618 ⟩ ≡

```
int main (int argc, char *argv[]) {
  struct binary_tree_entry *array;
  int n, i;
  /* Parse command line. */
  if (argc != 2) usage ();
  n = stoi (argv[1]);
  if (n < 1) usage ();
  /* Allocate memory. */
  array = malloc ((n + 1) * sizeof *array);
  if (array == NULL) {
    fprintf (stderr, "out of memory\n");
    return EXIT_FAILURE;
  }
  /* Initialize array. */
```

```
    for (i = 0; i < n; i++)
        array[i].value = i;
    init_binary_tree_array (array, n, 0, n − 1);
    /* Test successful and unsuccessful searches. */
    for (i = −1; i < n; i++) {
        int result = binary_search_tree_array (array, n, i);
        if (result != i)
            printf ("Searching␣for␣%d:␣expected␣%d,␣but␣received␣%d\n",
                    i, i, result);
    }
    /* Clean up. */
    free (array);
    return EXIT_SUCCESS;
}
```

This code is included in §617.

§619 ⟨ Show 'bin-ary-test' usage message 619 ⟩ ≡

```
/* Print a helpful usage message and abort execution. */
static void usage (void) {
    fputs ("Usage:␣binarytest␣<arraysize>\n"
           "where␣<arraysize>␣is␣the␣size␣of␣the␣array␣to␣test.\n",
           stdout);
    exit (EXIT_FAILURE);
}
```

This code is included in §617.

Chapter 4

1. This construct makes ⟨ `bst.h` 24 ⟩ **idempotent**, that is, including it many times has the same effect as including it once. This is important because some C constructs, such as type definitions with **typedef**, are erroneous if included in a program multiple times.

Of course, ⟨ Table assertion function control directives 593 ⟩ is included outside the **#ifndef**-protected part of ⟨ `bst.h` 24 ⟩. This is intentional (see Exercise 2.9-1 for details).

Section 4.2.2

1. Under many circumstances we often want to know how many items are in a binary tree. In these cases it's cheaper to keep track of item counts as we go instead of counting them each time, which requires a full binary tree traversal.

It would be better to omit it if we never needed to know how many items were in the tree, or if we only needed to know very seldom.

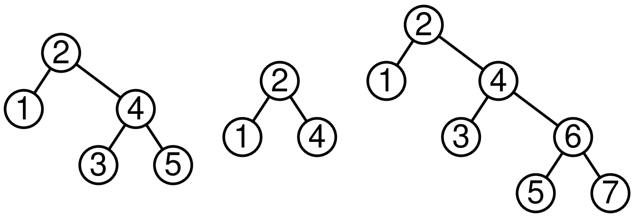
Section 4.2.3

1. The purpose for conditional definition of `BST_MAX_HEIGHT` is not to keep it from being redefined if the header file is included multiple times. There's a higher-level "include guard" for that (see Exercise 4-1), and, besides, identical definitions of a macro are okay in C. Instead, it is to allow the user to set the maximum height of binary trees by defining that macro before ⟨ bst.h 24 ⟩ is #**include**d. The limit can be adjusted upward for larger computers or downward for smaller ones.

The main pitfall is that a user program will use different values of `BST_MAX_HEIGHT` in different source files. This leads to undefined behavior. Less of a problem are definitions to invalid values, which will be caught at compile time by the compiler.

Section 4.3

1.



2. The functions need to adjust the pointer from the rotated subtree's parent, so they take a double-pointer **struct bst_node** $**$. An alternative would be to accept two parameters: the rotated subtree's parent node and the *bst_link*[] index of the subtree.

```
/* Rotates right at *yp. */
static void rotate_right (struct bst_node **yp) {
    struct bst_node *y = *yp;
    struct bst_node *x = y->bst_link[0];
    y->bst_link[0] = x->bst_link[1];
    x->bst_link[1] = y;
    *yp = x;
}
/* Rotates left at *xp. */
static void rotate_left (struct bst_node **xp) {
    struct bst_node *x = *xp;
    struct bst_node *y = x->bst_link[1];
    x->bst_link[1] = y->bst_link[0];
    y->bst_link[0] = x;
    *xp = y;
}
```

Section 4.7

1. This is a dirty trick. The *bst_root* member of **struct bst_table** is not a **struct bst_node**, but we are pretending that it is by casting its address to **struct bst_node** $*$. We can get away with this only because the first member of **struct bst_node** $*$ is *bst_link*, whose first element *bst_link*[0] is a **struct bst_node** $*$, the same type as *bst_root*. ANSI C guarantees that a pointer to a structure is a pointer to the structure's first member, so this is fine as long as we never try to access any member of $*p$ except *bst_link*[0]. Trying to access other members would result in undefined behavior.

The reason that we want to do this at all is that it means that the tree's root is not a special case. Otherwise, we have to deal with the root separately from the rest of the nodes in the tree, because of its special status as the only node in the tree not pointed to by the *bst_link*[] member of a **struct bst_node**.

It is a good idea to get used to these kinds of pointer cast, because they are common in LIBAVL.

As an alternative, we can declare an actual instance of **struct bst_node**, store the tree's *bst_root* into its *bst_link*[0], and copy its possibly updated value back into *bst_root* when done. This isn't very elegant, but it works. This technique is used much later in this book, in ⟨ TBST main copy function 279 ⟩. A different kind of alternative approach is used in Exercise 2.

2. Here, pointer-to-pointer q traverses the tree, starting with a pointer to the root, comparing each node found against *item* while looking for a null pointer. If an item equal to *item* is found, it returns a pointer to the item's data. Otherwise, q receives the address of the `NULL` pointer that becomes the new node, the new node is created, and a pointer to its data is returned.

§620 ⟨ BST item insertion function, alternate version 620 ⟩ ≡

```
void **bst_probe (struct bst_table *tree, void *item) {
    struct bst_node **q;
    int cmp;
    assert (tree != NULL && item != NULL);
    for (q = &tree→bst_root; *q != NULL; q = &(*q)→bst_link[cmp > 0]) {
        cmp = tree→bst_compare (item, (*q)→bst_data, tree→bst_param);
        if (cmp == 0)
            return &(*q)→bst_data;
    }
    *q = tree→bst_alloc→libavl_malloc (tree→bst_alloc, sizeof **q);
    if (*q == NULL)
        return NULL;
    (*q)→bst_link[0] = (*q)→bst_link[1] = NULL;
    (*q)→bst_data = item;
    tree→bst_count++;
    return &(*q)→bst_data;
}
```

3. The first item to be inserted have the value of the original tree's root. After that, at each step, we can insert either an item with the value of either child x of any node in the original tree corresponding to a node y already in the copy tree, as long as x's value is not already in the copy tree.

4. The function below traverses *tree* in "level order". That is, it visits the root, then the root's children, then the children of the root's children, and so on, so that all the nodes at a particular level in the tree are visited in sequence.

See also: [Sedgewick 1998], Program 5.16.

§621 ⟨ Level-order traversal 621 ⟩ ≡

```
/* Calls visit for each of the nodes in tree in level order.
```

```
   Returns nonzero if successful, zero if out of memory. */
static int bst_traverse_level_order (struct bst_table *tree, bst_item_func *visit) {
  struct bst_node **queue;
  size_t head, tail;

  if (tree->bst_count == 0)
    return 1;

  queue = tree->bst_alloc->libavl_malloc (tree->bst_alloc, sizeof *queue * tree->bst_count);
  if (queue == NULL)
    return 0;

  head = tail = 0;
  queue[head++] = tree->bst_root;
  while (head != tail) {
    struct bst_node *cur = queue[tail++];
    visit (cur->bst_data, tree->bst_param);
    if (cur->bst_link[0] != NULL)
      queue[head++] = cur->bst_link[0];
    if (cur->bst_link[1] != NULL)
      queue[head++] = cur->bst_link[1];
  }
  tree->bst_alloc->libavl_free (tree->bst_alloc, queue);

  return 1;
}
```

Section 4.7.1

1.

§622 ⟨ Root insertion of existing node in arbitrary subtree 622 ⟩ ≡

```
/* Performs root insertion of n at root within tree.
   Subtree root must not contain a node matching n.
   Returns nonzero only if successful. */
static int root_insert (struct bst_table *tree, struct bst_node **root,
                        struct bst_node *n) {
  struct bst_node *pa[BST_MAX_HEIGHT]; /* Nodes on stack. */
  unsigned char da[BST_MAX_HEIGHT]; /* Directions moved from stack nodes. */
  int k; /* Stack height. */

  struct bst_node *p; /* Traverses tree looking for insertion point. */

  assert (tree != NULL && n != NULL);

  ⟨Step 1: Search for insertion point in arbitrary subtree 623⟩
  ⟨Step 2: Insert n into arbitrary subtree 624⟩
  ⟨Step 3: Move BST node to root 36⟩

  return 1;
}
```

§623 ⟨ Step 1: Search for insertion point in arbitrary subtree 623 ⟩ ≡

```
pa[0] = (struct bst_node *) root;
da[0] = 0;
```

```
k = 1;
for (p = *root; p != NULL; p = p->bst_link[da[k - 1]]) {
    int cmp = tree->bst_compare (n->bst_data, p->bst_data, tree->bst_param);
    assert (cmp != 0);
    if (k >= BST_MAX_HEIGHT)
        return 0;
    pa[k] = p;
    da[k++] = cmp > 0;
}
```

This code is included in §622.

§624 ⟨ Step 2: Insert n into arbitrary subtree 624 ⟩ ≡

```
pa[k - 1]->bst_link[da[k - 1]] = n;
```

This code is included in §622 and §625.

2. The idea is to optimize for the common case but allow for fallback to a slower algorithm that doesn't require a stack when necessary.

§625 ⟨ Robust root insertion of existing node in arbitrary subtree 625 ⟩ ≡

```
/* Performs root insertion of n at root within tree.
   Subtree root must not contain a node matching n.
   Never fails and will not rebalance tree. */
static void root_insert (struct bst_table *tree, struct bst_node **root,
                         struct bst_node *n) {
    struct bst_node *pa[BST_MAX_HEIGHT]; /* Nodes on stack. */
    unsigned char da[BST_MAX_HEIGHT]; /* Directions moved from stack nodes. */
    int k; /* Stack height. */
    int overflow = 0; /* Set nonzero if stack overflowed. */
    struct bst_node *p; /* Traverses tree looking for insertion point. */
    assert (tree != NULL && n != NULL);
    ⟨ Step 1: Robustly search for insertion point in arbitrary subtree 626 ⟩
    ⟨ Step 2: Insert n into arbitrary subtree 624 ⟩
    ⟨ Step 3: Robustly move BST node to root 627 ⟩
}
```

If the stack overflows while we're searching for the insertion point, we stop keeping track of any nodes but the last one and set *overflow* so that later we know that overflow occurred:

§626 ⟨ Step 1: Robustly search for insertion point in arbitrary subtree 626 ⟩ ≡

```
pa[0] = (struct bst_node *) root;
da[0] = 0;
k = 1;
for (p = *root; p != NULL; p = p->bst_link[da[k - 1]]) {
    int cmp = tree->bst_compare (n->bst_data, p->bst_data, tree->bst_param);
    assert (cmp != 0);
    if (k >= BST_MAX_HEIGHT) {
        overflow = 1;
        k--;
    }
```

```
    pa[k] = p;
    da[k++] = cmp > 0;
}
```

This code is included in §625.

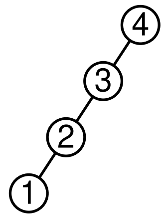
Once we've inserted the node, we deal with the rotation in the same way as before if there was no overflow. If overflow occurred, we instead do the rotations one by one, with a full traversal from **root* every time:

§627 ⟨Step 3: Robustly move BST node to root 627⟩ ≡

```
if (!overflow)
    { ⟨Step 3: Move BST node to root 36⟩ }
else {
    while (*root != n) {
        struct bst_node **r; /* Link to node to rotate. */
        struct bst_node *q; /* Node to rotate. */
        int dir;
        for (r = root; ; r = &q→bst_link[dir]) {
            q = *r;
            dir = 0 < tree→bst_compare (n→bst_data, q→bst_data, tree→bst_param);
            if (q→bst_link[dir] == n)
                break;
        }
        if (dir == 0) {
            q→bst_link[0] = n→bst_link[1];
            n→bst_link[1] = q;
        } else {
            q→bst_link[1] = n→bst_link[0];
            n→bst_link[0] = q;
        }
        *r = n;
    }
}
```

This code is included in §625.

3. One insertion order that does *not* require much stack is ascending order. If we insert 1. . . 4 at the root in ascending order, for instance, we get a BST that looks like this:



If we then insert node 5, it will immediately be inserted as the right child of 4, and then a left rotation will make it the root, and we're back where we started without ever using more than one stack entry. Other obvious pathological orders such as descending order and "zig-zag" order behave similarly.

One insertion order that does require an arbitrary amount of stack space is to first insert $1 \ldots n$ in ascending order, then the single item 0. Each of the first group of insertions requires only one stack entry (except the first, which does not use any), but the final insertion uses $n - 1$.

If we're interested in high average consumption of stack space, the pattern consisting of a series of ascending insertions $(n / 2 + 1) \ldots n$ followed by a second ascending series $1 \ldots (n / 2)$, for even n, is most effective. For instance, each insertion for insertion order 6, 7, 8, 9, 10, 1, 2, 3, 4, 5 requires 0, 1, 1, 1, 1, 5, 6, 6, 6, 6 stack entries, respectively, for a total of 33.

These are, incidentally, the best possible results in each category, as determined by exhaustive search over the $10! \equiv 3{,}628{,}800$ possible root insertion orders for trees of 10 nodes. (Thanks to Richard Heathfield for suggesting exhaustive search.)

Section 4.8

1. Add this before the top-level **else** clause in ⟨ Step 2: Delete BST node 39 ⟩:

§628 ⟨ Case 1.5 in BST deletion 628 ⟩ ≡

```
else if (p->bst_link[0] == NULL) q->bst_link[dir] = p->bst_link[1];
```

2. Be sure to look at Exercise 3 before actually making this change.

§629 ⟨ Case 3 in BST deletion, alternate version 629 ⟩ ≡

```
struct bst_node *s = r->bst_link[0];
while (s->bst_link[0] != NULL) {
    r = s;
    s = r->bst_link[0];
}
p->bst_data = s->bst_data;
r->bst_link[0] = s->bst_link[1];
p = s;
```

We could, indeed, make similar changes to the other cases, but for these cases the code would become more complicated, not simpler.

3. The semantics for Libavl traversers only invalidate traversers with the deleted item selected, but the revised code would actually free the node of the successor to that item. Because **struct bst_traverser** keeps a pointer to the **struct bst_node** of the current item, attempts to use a traverser that had selected the successor of the deleted item would result in undefined behavior.

Some other binary tree libraries have looser semantics on their traversers, so they can afford to use this technique.

Section 4.9.1

1. It would probably be faster to check before each call rather than after, because this way many calls would be avoided. However, it might be more difficult to maintain the code, because we would have to remember to check for a null pointer before every call. For instance, the call to *traverse_recursive*() within *walk*() might easily be overlooked. Which is “better” is therefore a toss-up, dependent on a program’s goals and the programmer’s esthetic sense.

2.

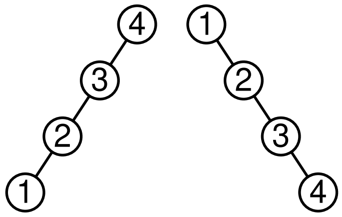
§630 ⟨ Recursive traversal of BST, using nested function 630 ⟩ ≡

```
void walk (struct bst_table *tree, bst_item_func *action, void *param) {
  void traverse_recursive (struct bst_node *node) {
    if (node != NULL) {
      traverse_recursive (node->bst_link[0]);
      action (node->bst_data, param);
      traverse_recursive (node->bst_link[1]);
    }
  }
  assert (tree != NULL && action != NULL);
  traverse_recursive (tree->bst_root);
}
```

Section 4.9.2

1a. First of all, a minimal-height binary tree of n nodes has a **height** of about $\log_2 n$, that is, starting from the root and moving only downward, you can visit at most n nodes (including the root) without running out of nodes. Examination of the code should reveal to you that only moving down to the left pushes nodes on the stack and only moving upward pops nodes off. What’s more, the first thing the code does is move as far down to the left as it can. So, the maximum height of the stack in a minimum-height binary tree of n nodes is the binary tree’s height, or, again, about $\log_2 n$.

1b. If a binary tree has only left children, as does the BST on the left below, the stack will grow as tall as the tree, to a height of n. Conversely, if a binary tree has only right children, as does the BST on the right below, no nodes will be pushed onto the stack at all.



1c. It’s only acceptable if it’s known that the stack will not exceed the fixed maximum height (or if the program aborting with an error is itself acceptable). Otherwise, you should use a recursive method (but see part (e) below), or a dynamically extended stack, or a balanced binary tree library.

1d. Keep in mind this is not the only way or necessarily the best way to handle stack overflow. Our final code for tree traversal will rebalance the tree when it grows too tall.

§631 ⟨ Iterative traversal of BST, with dynamically allocated stack 631 ⟩ ≡

```
static void traverse_iterative (struct bst_node *node, bst_item_func *action, void *param) {
  struct bst_node **stack = NULL;
  size_t height = 0;
```

```
    size_t max_height = 0;
    for (;;) {
        while (node != NULL) {
            if (height >= max_height) {
                max_height = max_height * 2 + 8;
                stack = realloc (stack, sizeof *stack * max_height);
                if (stack == NULL) {
                    fprintf (stderr, "out of memory\n");
                    exit (EXIT_FAILURE);
                }
            }
            stack[height++] = node;
            node = node->bst_link[0];
        }
        if (height == 0)
            break;
        node = stack[--height];
        action (node->bst_data, param);
        node = node->bst_link[1];
    }
    free (stack);
}
```

1e. Yes, *traverse_recursive*() can run out of memory, because its arguments must be stored somewhere by the compiler. Given typical compilers, it will consume more memory per call than *traverse_iterative*() will per item on the stack, because each call includes two arguments not pushed on *traverse_iterative*()'s stack, plus any needed compiler-specific bookkeeping information.

Section 4.9.2.1

1. After calling *bst_balance*(), the structure of the binary tree may have changed completely, so we need to "find our place" again by setting up the traverser structure as if the traversal had been done on the rebalanced tree all along. Specifically, members *node*, *stack*[], and *height* of **struct traverser** need to be updated.

It is easy to set up **struct traverser** in this way, given the previous node in inorder traversal, which we'll call *prev*. Simply search the tree from the new root to find this node. Along the way, because the stack is used to record nodes whose left subtree we are examining, push nodes onto the stack as we move left down the tree. Member *node* receives *prev*→*bst_link*[1], just as it would have if no overflow had occurred.

A small problem with this approach is that it requires knowing the previous node in inorder, which is neither explicitly noted in **struct traverser** nor easy to find out. But it *is* easy to find out the next node: it is the smallest-valued node in the binary tree rooted at the node we were considering when the stack overflowed. (If you need convincing, refer to the code for *next_item*() above: the **while** loop descends to the left, pushing nodes as it goes, until it hits a `NULL` pointer, then the node pushed last is popped and returned.) So we can return this as the next node in inorder while setting up the traverser to return the nodes after it.

Here's the code:

§632 ⟨ Handle stack overflow during BST traversal 632 ⟩ ≡
```
struct bst_node *prev, *iter;
prev = node;
while (prev->bst_link[0] != NULL)
    prev = prev->bst_link[0];
bst_balance (trav->table);
trav->height = 0;
for (iter = trav->table->bst_root; iter != prev; )
    if (trav->table->bst_compare (prev->bst_data, iter->bst_data, trav->table->bst_param) < 0) {
        trav->stack[trav->height++] = iter;
        iter = iter->bst_link[0];
    }
    else iter = iter->bst_link[1];
trav->node = iter->bst_link[1];
return prev->bst_data;
```

Without this code, it is not necessary to have member *table* in **struct traverser**.

2. It is possible to write *prev_item*() given our current *next_item*(), but the result is not very efficient, for two reasons, both related to the way that **struct traverser** is used. First, the structure doesn't contain a pointer to the current item. Second, its stack doesn't contain pointers to trees that must be descended to the left to find a predecessor node, only those that must be descended to the right to find a successor node.

The next section will develop an alternate, more general method for traversal that avoids these problems.

Section 4.9.3

1. The *bst_probe*() function can't disturb any traversals. A change in the tree is only problematic for a traverser if it deletes the currently selected node (which is explicitly undefined: see Section 2.10 [Traversers], page 15) or if it shuffles around any of the nodes that are on the traverser's stack. An insertion into a tree only creates new leaves, so it can't cause either of those problems, and there's no need to increment the generation number.

The same logic applies to *bst_t_insert*(), presented later.

On the other hand, an insertion into the AVL and red-black trees discussed in the next two chapters can cause restructuring of the tree and thus potentially disturb ongoing traversals. For this reason, the insertion functions for AVL and red-black trees *will* increment the tree's generation number.

2. First, *trav_refresh*() is only called from *bst_t_next*() and *bst_t_prev*(), and these functions are mirrors of each other, so we need only show it for one of them.

Second, all of the traverser functions check the stack height, so these will not cause an item to be initialized at too high a height, nor will *bst_t_next*() or *bst_t_prev*() increase the stack height above its limit.

Since the traverser functions won't force a too-tall stack directly, this leaves the other functions. Only functions that modify the tree could cause problems, by pushing an item farther down in the tree.

There are only four functions that modify a tree. The insertion functions *bst_probe*() and *bst_t_insert*() can't cause problems, because they add leaves but never move around nodes. The deletion function *bst_delete*() does move around nodes in case 3, but it always moves them higher in the tree, never lower. Finally, *bst_balance*() always ensures that all nodes in the resultant tree are within the tree's height limit.

3. This won't work because the stack may contain pointers to nodes that have been deleted and whose memory have been freed. In ANSI C89 and C99, any use of a pointer to an object after the end of its lifetime results in undefined behavior, even seemingly innocuous uses such as pointer comparisons. What's worse, the memory for the node may already have been recycled for use for another, different node elsewhere in the tree.

This approach does work if there are never any deletions in the tree, or if we use some kind of generation number for each node that we store along with each stack entry. The latter would be overkill unless comparisons are very expensive and the traversals in changing trees are common. Another possibility would be to somehow only select this behavior if there have been no deletions in the binary tree since the traverser was last used. This could be done, for instance, with a second generation number in the binary tree incremented only on deletions, with a corresponding number kept in the traverser.

The following reimplements *trav_refresh*() to include this optimization. As noted, it will not work if there are any deletions in the tree. It does work for traversers that must be refreshed due to, e.g., rebalancing.

§633 ⟨BST traverser refresher, with caching 633⟩ ≡

```
/* Refreshes the stack of parent pointers in trav
   and updates its generation number.
   Will *not* work if any deletions have occurred in the tree. */
static void trav_refresh (struct bst_traverser *trav) {
  assert (trav != NULL);
  trav->bst_generation = trav->bst_table->bst_generation;
  if (trav->bst_node != NULL) {
    bst_comparison_func *cmp = trav->bst_table->bst_compare;
    void *param = trav->bst_table->bst_param;
    struct bst_node *node = trav->bst_node;
    struct bst_node *i = trav->bst_table->bst_root;
    size_t height = 0;
    if (trav->bst_height > 0 && i == trav->bst_stack[0])
      for (; height < trav->bst_height; height++) {
        struct bst_node *next = trav->bst_stack[height + 1];
        if (i->bst_link[0] != next && i->bst_link[1] != next)
          break;
        i = next;
      }
    while (i != node) {
      assert (height < BST_MAX_HEIGHT);
```

```
            assert (i != NULL);
            trav→bst_stack[height++] = i;
            i = i→bst_link[cmp (node→bst_data, i→bst_data, param) > 0];
        }
        trav→bst_height = height;
    }
}
```

Section 4.9.3.2

1. It only calls itself if it runs out of stack space. Its call to *bst_balance*() right before the recursive call ensures that the tree is short enough to fit within the stack, so the recursive call cannot overflow.

Section 4.9.3.6

1. The assignment statements are harmless, but *memcpy*() of overlapping regions produces undefined behavior.

Section 4.10.1

1a. Notice the use of & instead of && below. This ensures that both link fields get initialized, so that deallocation can be done in a simple way. If && were used instead then we wouldn't have any way to tell whether (*y)→*bst_link*[1] had been initialized.

§634 ⟨ Robust recursive copy of BST, take 1 634 ⟩ ≡

```
/* Stores in *y a new copy of tree rooted at x.
   Returns nonzero if successful, or zero if memory was exhausted.*/
static int bst_robust_copy_recursive_1 (struct bst_node *x, struct bst_node **y) {
    if (x != NULL) {
        *y = malloc (sizeof **y);
        if (*y == NULL)
            return 0;
        (*y)→bst_data = x→bst_data;
        if (!(bst_robust_copy_recursive_1 (x→bst_link[0], &(*y)→bst_link[0])
            & bst_robust_copy_recursive_1 (x→bst_link[1], &(*y)→bst_link[1]))) {
            bst_deallocate_recursive (*y);
            *y = NULL;
            return 0;
        }
    }
    else *y = NULL;
    return 1;
}
```

Here's a needed auxiliary function:

§635 ⟨ Recursive deallocation function 635 ⟩ ≡

```
static void bst_deallocate_recursive (struct bst_node *node) {
    if (node == NULL)
        return;
    bst_deallocate_recursive (node→bst_link[0]);
    bst_deallocate_recursive (node→bst_link[1]);
    free (node);
}
```

1b.

§636 ⟨ Robust recursive copy of BST, take 2 636 ⟩ ≡

```
static struct bst_node error_node;
/* Makes and returns a new copy of tree rooted at x.
   If an allocation error occurs, returns &error_node. */
static struct bst_node *bst_robust_copy_recursive_2 (struct bst_node *x) {
    struct bst_node *y;
    if (x == NULL)
        return NULL;
    y = malloc (sizeof *y);
    if (y == NULL)
        return &error_node;
    y→bst_data = x→bst_data;
    y→bst_link[0] = bst_robust_copy_recursive_2 (x→bst_link[0]);
    y→bst_link[1] = bst_robust_copy_recursive_2 (x→bst_link[1]);
    if (y→bst_link[0] == &error_node || y→bst_link[1] == &error_node) {
        bst_deallocate_recursive (y);
        return &error_node;
    }
    return y;
}
```

2. Here's one way to do it, which is simple but perhaps not the fastest possible.

§637 ⟨ Robust recursive copy of BST, take 3 637 ⟩ ≡

```
/* Copies tree rooted at x to y, which latter is allocated but not yet initialized.
   Returns one if successful, zero if memory was exhausted.
   In the latter case y is not freed but any partially allocated
   subtrees are. */
static int bst_robust_copy_recursive_3 (struct bst_node *x, struct bst_node *y) {
    y→bst_data = x→bst_data;
    if (x→bst_link[0] != NULL) {
        y→bst_link[0] = malloc (sizeof *y→bst_link[0]);
        if (y→bst_link[0] == NULL)
            return 0;
        if (!bst_robust_copy_recursive_3 (x→bst_link[0], y→bst_link[0])) {
            free (y→bst_link[0]);
            return 0;
        }
```

```
    }
    else y->bst_link[0] = NULL;
    if (x->bst_link[1] != NULL) {
        y->bst_link[1] = malloc (sizeof *y->bst_link[1]);
        if (y->bst_link[1] == NULL)
            return 0;
        if (!bst_robust_copy_recursive_3 (x->bst_link[1], y->bst_link[1])) {
            bst_deallocate_recursive (y->bst_link[0]);
            free (y->bst_link[1]);
            return 0;
        }
    }
    else y->bst_link[1] = NULL;
    return 1;
}
```

Section 4.10.2

1. Here is one possibility.

§638 ⟨Intermediate step between *bst_copy_recursive_2*() and *bst_copy_iterative*() 638⟩ ≡

```
/* Copies org to a newly created tree, which is returned. */
struct bst_table *bst_copy_iterative (const struct bst_table *org) {
    struct bst_node *stack[2 * (BST_MAX_HEIGHT + 1)];
    int height = 0;
    struct bst_table *new;
    const struct bst_node *x;
    struct bst_node *y;
    new = bst_create (org->bst_compare, org->bst_param, org->bst_alloc);
    new->bst_count = org->bst_count;
    if (new->bst_count == 0)
        return new;
    x = (const struct bst_node *) &org->bst_root;
    y = (struct bst_node *) &new->bst_root;
    for (;;) {
        while (x->bst_link[0] != NULL) {
            y->bst_link[0] = org->bst_alloc->libavl_malloc (org->bst_alloc,
                                sizeof *y->bst_link[0]);
            stack[height++] = (struct bst_node *) x;
            stack[height++] = y;
            x = x->bst_link[0];
            y = y->bst_link[0];
        }
        y->bst_link[0] = NULL;
        for (;;) {
            y->bst_data = x->bst_data;
```

```
            if (x→bst_link[1] != NULL) {
                y→bst_link[1] = org→bst_alloc→libavl_malloc (org→bst_alloc,
                            sizeof *y→bst_link[1]);
                x = x→bst_link[1];
                y = y→bst_link[1];
                break;
            }
            else y→bst_link[1] = NULL;
            if (height <= 2)
                return new;
            y = stack[−−height];
            x = stack[−−height];
        }
    }
}
```

Section 4.11.1

1. *bst_copy*() can set *bst_data* to `NULL` when memory allocation fails.

Section 4.13

1. Factoring out recursion is troublesome in this case. Writing the loop with an explicit stack exposes more explicitly the issue of stack overflow. Failure on stack overflow is not acceptable, because it would leave both trees in disarray, so we handle it by dropping back to a slower algorithm that does not require a stack.

This code also makes use of *root_insert*() from ⟨ Robust root insertion of existing node in arbitrary subtree 625 ⟩.

§639 ⟨ BST join function, iterative version 639 ⟩ ≡

```
/* Adds to tree all the nodes in the tree rooted at p. */
static void fallback_join (struct bst_table *tree, struct bst_node *p) {
    struct bst_node *q;
    for (; p != NULL; p = q)
        if (p→bst_link[0] == NULL) {
            q = p→bst_link[1];
            p→bst_link[0] = p→bst_link[1] = NULL;
            root_insert (tree, &tree→bst_root, p);
        }
        else {
            q = p→bst_link[0];
            p→bst_link[0] = q→bst_link[1];
            q→bst_link[1] = p;
        }
}
/* Joins a and b, which must be disjoint and have compatible comparison functions.
   b is destroyed in the process. */
```

```
void bst_join (struct bst_table *ta, struct bst_table *tb) {
  size_t count = ta->bst_count + tb->bst_count;

  if (ta->bst_root == NULL)
    ta->bst_root = tb->bst_root;
  else if (tb->bst_root != NULL) {
    struct bst_node **pa[BST_MAX_HEIGHT];
    struct bst_node *qa[BST_MAX_HEIGHT];
    int k = 0;

    pa[k] = &ta->bst_root;
    qa[k++] = tb->bst_root;
    while (k > 0) {
      struct bst_node **a = pa[--k];
      struct bst_node *b = qa[k];

      for (;;) {
        struct bst_node *b0 = b->bst_link[0];
        struct bst_node *b1 = b->bst_link[1];
        b->bst_link[0] = b->bst_link[1] = NULL;
        root_insert (ta, a, b);

        if (b1 != NULL) {
          if (k < BST_MAX_HEIGHT) {
            pa[k] = &(*a)->bst_link[1];
            qa[k] = b1;
            if (*pa[k] != NULL)
              k++;
            else *pa[k] = qa[k];
          } else {
            int j;

            fallback_join (ta, b0);
            fallback_join (ta, b1);
            for (j = 0; j < k; j++)
              fallback_join (ta, qa[j]);

            ta->bst_count = count;
            free (tb);
            bst_balance (ta);
            return;
          }
        }

        a = &(*a)->bst_link[0];
        b = b0;
        if (*a == NULL) {
          *a = b;
          break;
        } else if (b == NULL)
          break;
      }
```

```
        }
    }
    ta→bst_count = count;
    free (tb);
}
```

Section 4.14.1

1. Functions not used at all are *bst_insert*(), *bst_replace*(), *bst_t_replace*(), *bst_malloc*(), and *bst_free*().

Functions used explicitly within *test*() or functions that it calls are *bst_create*(), *bst_find*(), *bst_probe*(), *bst_delete*(), *bst_t_init*(), *bst_t_first*(), *bst_t_last*(), *bst_t_insert*(), *bst_t_find*(), *bst_t_copy*(), *bst_t_next*(), *bst_t_prev*(), *bst_t_cur*(), *bst_copy*(), and *bst_destroy*().

The *trav_refresh*() function is called indirectly by modifying the tree during traversal.

The *copy_error_recovery*() function is called if a memory allocation error occurs during *bst_copy*(). The *bst_balance*() function, and therefore also *tree_to_vine*(), *vine_to_tree*(), and *compress*(), are called if a stack overflow occurs. It is possible to force both these behaviors with command-line options to the test program.

2. Some kinds of errors mean that we can keep going and test other parts of the code. Other kinds of errors mean that something is deeply wrong, and returning without cleanup is the safest action short of terminating the program entirely. The third category is memory allocation errors. In our test program these are always caused intentionally in order to test out the BST functions' error recovery abilities, so a memory allocation error is not really an error at all, and we clean up and return successfully. (A real memory allocation error will cause the program to abort in the memory allocator. See the definition of *mt_allocate*() within ⟨ Memory tracker 126 ⟩.)

Section 4.14.1.1

1. The definition of **size_t** differs from one compiler to the next. All we know about it for sure is that it's an unsigned type appropriate for representing the size of an object. So we must convert it to some known type in order to pass it to *printf*(), because *printf*(), having a variable number of arguments, does not know what type to convert it into.

Incidentally, C99 solves this problem by providing a '`z`' modifier for *printf*() conversions, so that we could use `"%zu"` to print out **size_t** values without the need for a cast.

See also: [ISO 1999], section 7.19.6.1.

2. Yes.

Section 4.14.2

1.

§640 ⟨ Generate random permutation of integers 640 ⟩ ≡

```
/* Fills the n elements of array[] with a random permutation of the
   integers between 0 and n − 1. */
```

```
static void permuted_integers (int array[], size_t n) {
  size_t i;
  for (i = 0; i < n; i++)
    array[i] = i;
  for (i = 0; i < n; i++) {
    size_t j = i + (unsigned) rand () / (RAND_MAX / (n − i) + 1);
    int t = array[j];
    array[j] = array[i];
    array[i] = t;
  }
}
```

This code is included in §642.

2. All it takes is a preorder traversal. If the code below is confusing, try looking back at ⟨Initialize *smaller* and *larger* within binary search tree 616⟩.

§641 ⟨Generate permutation for balanced tree 641⟩ ≡

```
/* Generates a list of integers that produce a balanced tree when
   inserted in order into a binary tree in the usual way.
   min and max inclusively bound the values to be inserted.
   Output is deposited starting at *array. */
static void gen_balanced_tree (int min, int max, int **array) {
  int i;
  if (min > max)
    return;
  i = (min + max + 1) / 2;
  *(*array)++ = i;
  gen_balanced_tree (min, i − 1, array);
  gen_balanced_tree (i + 1, max, array);
}
```

This code is included in §642.

3.

§642 ⟨Insertion and deletion order generation 642⟩ ≡

```
⟨Generate random permutation of integers 640⟩
⟨Generate permutation for balanced tree 641⟩
/* Generates a permutation of the integers 0 to n − 1 into
   insert[] according to insert_order. */
static void gen_insertions (size_t n, enum insert_order insert_order, int insert[]) {
  size_t i;
  switch (insert_order) {
    case INS_RANDOM:
      permuted_integers (insert, n);
      break;
    case INS_ASCENDING:
      for (i = 0; i < n; i++)
        insert[i] = i;
```

```
        break;
    case INS_DESCENDING:
        for (i = 0; i < n; i++)
            insert[i] = n - i - 1;
        break;
    case INS_BALANCED:
        gen_balanced_tree (0, n - 1, &insert);
        break;
    case INS_ZIGZAG:
        for (i = 0; i < n; i++)
            if (i % 2 == 0) insert[i] = i / 2;
            else insert[i] = n - i / 2 - 1;
        break;
    case INS_ASCENDING_SHIFTED:
        for (i = 0; i < n; i++) {
            insert[i] = i + n / 2;
            if ((size_t) insert[i] >= n)
                insert[i] -= n;
        }
        break;
    case INS_CUSTOM:
        for (i = 0; i < n; i++)
            if (scanf ("%d", &insert[i]) == 0)
                fail ("error reading insertion order from stdin");
        break;
    default:
        assert (0);
    }
}
/* Generates a permutation of the integers 0 to n - 1 into
   delete[] according to delete_order and insert[]. */
static void gen_deletions (size_t n, enum delete_order delete_order,
                           const int *insert, int *delete) {
  size_t i;
  switch (delete_order) {
    case DEL_RANDOM:
        permuted_integers (delete, n);
        break;
    case DEL_REVERSE:
        for (i = 0; i < n; i++)
            delete[i] = insert[n - i - 1];
        break;
    case DEL_SAME:
        for (i = 0; i < n; i++)
            delete[i] = insert[i];
```

```
            break;
        case DEL_CUSTOM:
            for (i = 0; i < n; i++)
                if (scanf ("%d", &delete[i]) == 0)
                    fail ("error reading deletion order from stdin");
            break;
        default:
            assert (0);
        }
}
```

This code is included in §97.

4. The function below is carefully designed. It uses *time*() to obtain the current time. The alternative *clock*() is a poor choice because it measures CPU time used, which is often more or less constant among runs. The actual value of a **time_t** is not portable, so it computes a "hash" of the bytes in it using a multiply-and-add technique. The factor used for multiplication normally comes out as 257, a prime and therefore a good candidate.

See also: [Knuth 1998a], section 3.2.1; [Aho 1986], section 7.6.

§643 ⟨ Random number seeding 643 ⟩ ≡

```
/* Choose and return an initial random seed based on the current time.
   Based on code by Lawrence Kirby <fred@genesis.demon.co.uk>. */
unsigned time_seed (void) {
    time_t timeval; /* Current time. */
    unsigned char *ptr; /* Type punned pointed into timeval. */
    unsigned seed; /* Generated seed. */
    size_t i;

    timeval = time (NULL);
    ptr = (unsigned char *) &timeval;

    seed = 0;
    for (i = 0; i < sizeof timeval; i++)
        seed = seed * (UCHAR_MAX + 2u) + ptr[i];

    return seed;
}
```

This code is included in §97.

Section 4.14.3

1.

§644 ⟨ Overflow testers 124 ⟩ +≡

```
static int test_bst_t_last (struct bst_table *tree, int n) {
    struct bst_traverser trav;
    int *last;

    last = bst_t_last (&trav, tree);
    if (last == NULL || *last != n - 1) {
        printf ("    Last item test failed: expected %d, got %d\n",
```

```
            n - 1, last != NULL ? *last : -1);
      return 0;
    }
  return 1;
}
static int test_bst_t_find (struct bst_table *tree, int n) {
  int i;
  for (i = 0; i < n; i++) {
      struct bst_traverser trav;
      int *iter;
      iter = bst_t_find (&trav, tree, &i);
      if (iter == NULL || *iter != i) {
          printf ("    Find item test failed: looked for %d, got %d\n",
                  i, iter != NULL ? *iter : -1);
          return 0;
        }
    }
  return 1;
}
static int test_bst_t_insert (struct bst_table *tree, int n) {
  int i;
  for (i = 0; i < n; i++) {
      struct bst_traverser trav;
      int *iter;
      iter = bst_t_insert (&trav, tree, &i);
      if (iter == NULL || iter == &i || *iter != i) {
          printf ("    Insert item test failed: inserted dup %d, got %d\n",
                  i, iter != NULL ? *iter : -1);
          return 0;
        }
    }
  return 1;
}
static int test_bst_t_next (struct bst_table *tree, int n) {
  struct bst_traverser trav;
  int i;
  bst_t_init (&trav, tree);
  for (i = 0; i < n; i++) {
      int *iter = bst_t_next (&trav);
      if (iter == NULL || *iter != i) {
          printf ("    Next item test failed: expected %d, got %d\n",
                  i, iter != NULL ? *iter : -1);
          return 0;
        }
    }
```

```
    return 1;
}
static int test_bst_t_prev (struct bst_table *tree, int n) {
    struct bst_traverser trav;
    int i;
    bst_t_init (&trav, tree);
    for (i = n - 1; i >= 0; i--) {
        int *iter = bst_t_prev (&trav);
        if (iter == NULL || *iter != i) {
            printf ("    Previous item test failed: expected %d, got %d\n",
                    i, iter != NULL ? *iter : -1);
            return 0;
        }
    }
    return 1;
}
static int test_bst_copy (struct bst_table *tree, int n) {
    struct bst_table *copy = bst_copy (tree, NULL, NULL, NULL);
    int okay = compare_trees (tree->bst_root, copy->bst_root);
    bst_destroy (copy, NULL);
    return okay;
}
```

Section 4.14.4

1. Attempting to apply an allocation policy to allocations of zero-byte blocks is silly. How could a failure be indicated, given that one of the successful results for an allocation of 0 bytes is `NULL`? At any rate, Libavl never calls *bst_allocate*() with a *size* argument of 0.

See also: [ISO 1990], section 7.10.3.

Section 4.15

1. We'll use *bsts_*, short for "binary search tree with sentinel", as the prefix for these functions. First, we need node and tree structures:

§645 ⟨ BSTS structures 645 ⟩ ≡

```
/* Node for binary search tree with sentinel. */
struct bsts_node {
    struct bsts_node *link[2];
    int data;
};
/* Binary search tree with sentinel. */
struct bsts_tree {
    struct bsts_node *root;
    struct bsts_node sentinel;
    struct libavl_allocator *alloc;
```

```
};
```

This code is included in §649.

Searching is simple:

§646 ⟨ BSTS functions 646 ⟩ ≡

```
/* Returns nonzero only if item is in tree. */
int bsts_find (struct bsts_tree *tree, int item) {
    const struct bsts_node *node;

    tree->sentinel.data = item;
    node = tree->root;
    while (item != node->data)
        if (item < node->data) node = node->link[0];
        else node = node->link[1];
    return node != &tree->sentinel;
}
```

See also §647.
This code is included in §649.

Insertion is just a little more complex, because we have to keep track of the link that we just came from (alternately, we could divide the function into multiple cases):

§647 ⟨ BSTS functions 646 ⟩ +≡

```
/* Inserts item into tree, if it is not already present. */
void bsts_insert (struct bsts_tree *tree, int item) {
    struct bsts_node **q = &tree->root;
    struct bsts_node *p = tree->root;

    tree->sentinel.data = item;
    while (item != p->data) {
        int dir = item > p->data;
        q = &p->link[dir];
        p = p->link[dir];
    }
    if (p == &tree->sentinel) {
        *q = tree->alloc->libavl_malloc (tree->alloc, sizeof **q);
        if (*q == NULL) {
            fprintf (stderr, "out of memory\n");
            exit (EXIT_FAILURE);
        }
        (*q)->link[0] = (*q)->link[1] = &tree->sentinel;
        (*q)->data = item;
    }
}
```

Our test function will just insert a collection of integers, then make sure that all of them are in the resulting tree. This is not as thorough as it could be, and it doesn't bother to free what it allocates, but it is good enough for now:

§648 ⟨ BSTS test 648 ⟩ ≡

```
/* Tests BSTS functions.
```

```
   insert and delete must contain some permutation of values
   0...n − 1. */
int test_correctness (struct libavl_allocator *alloc, int *insert,
                      int *delete, int n, int verbosity) {
  struct bsts_tree tree;
  int okay = 1;
  int i;

  tree.root = &tree.sentinel;
  tree.alloc = alloc;

  for (i = 0; i < n; i++)
    bsts_insert (&tree, insert[i]);

  for (i = 0; i < n; i++)
    if (!bsts_find (&tree, i)) {
      printf ("%d should be in tree, but isn't\n", i);
      okay = 0;
    }

  return okay;
}

/* Not supported. */
int test_overflow (struct libavl_allocator *alloc, int order[], int n, int verbosity) {
  return 0;
}
```

This code is included in §649.

Function *test*() doesn't free allocated nodes, resulting in a memory leak. You should fix this if you are concerned about it.

Here's the whole program:

§649 ⟨ bsts.c 649 ⟩ ≡
```
⟨ License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ stdlib.h ⟩
#include "test.h"

⟨ BSTS structures 645 ⟩
⟨ Memory allocator; tbl ⇒ bsts 5 ⟩
⟨ Default memory allocator header; tbl ⇒ bsts 7 ⟩
⟨ Default memory allocation functions; tbl ⇒ bsts 6 ⟩
⟨ BSTS functions 646 ⟩
⟨ BSTS test 648 ⟩
```

See also: [Bentley 2000], exercise 7 in chapter 13.

Chapter 5

Section 5.4

1. In a BST, the time for an insertion or deletion is the time required to visit each node from the root down to the node of interest, plus some time to perform the operation itself. Functions *bst_probe*() and *bst_delete*() contain only a single loop each, which iterates once for each node examined. As the tree grows, the time for the actual operation loses significance and the total time for the operation becomes essentially proportional to the height of the tree, which is approximately $\log_2 n$ in the best case (see Section 5.1.1 [Analysis of AVL Balancing Rule], page 109).

We were given that the additional work for rebalancing an AVL or red-black tree is at most a constant amount multiplied by the height of the tree. Furthermore, the maximum height of an AVL tree is 1.44 times the maximum height for the corresponding perfectly balanced binary tree, and a red-black tree has a similar bound on its height. Therefore, for trees with many nodes, the worst-case time required to insert or delete an item in a balanced tree is a constant multiple of the time required for the same operation on an unbalanced BST in the best case. In the formal terms of computer science, insertion and deletion in a balanced tree are $O(log\ n)$ operations, where n is the number of nodes in the tree.

In practice, operations on balanced trees of reasonable size are, at worst, not much slower than operations on unbalanced binary trees and, at best, much faster.

Section 5.4.2

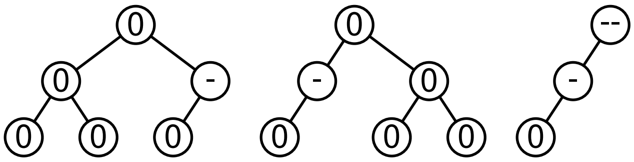
1. Variable y is only modified within ⟨ Step 1: Search AVL tree for insertion point 148 ⟩. If y is set during the loop, it is set to p, which is always a non-null pointer within the loop. So y can only be `NULL` if it is last set before the loop begins. If that is true, it will be `NULL` only if *tree*→*avl_root* == `NULL`. So, variable y can only be `NULL` if the AVL tree was empty before the insertion.

A `NULL` value for y is a special case because later code assumes that y points to a node.

Section 5.4.3

1. No. Suppose that n is the new node, that p is its parent, and that p has a $-$ balance factor before n's insertion (a similar argument applies if p's balance factor is $+$). Then, for n's insertion to decrease p's balance factor to $^-2$, n would have to be the left child of p. But if p had a $-$ balance factor before the insertion, it already had a left child, so n cannot be the new left of p. This is a contradiction, so case 3 will never be applied to the parent of a newly inserted node.

2.



In the leftmost tree, case 2 applies to the root's left child and the root's balance factor does not change. In the middle tree, case 1 applies to the root's left child and case 2 applies to the root. In the rightmost tree, case 1 applies to the root's left child and case 3 applies to the root. The tree on the right requires rebalancing, and the others do not.

3. Type **char** may be signed or unsigned, depending on the C compiler and/or how the C compiler is run. Also, a common use for subscripting an array with a character type is to translate an arbitrary character to another character or a set of properties. For example, this is a common way to implement the standard C functions from *ctype.h*. This means that subscripting such an array with a **char** value can have different behavior when **char** changes between signed and unsigned with different compilers (or with the same compiler invoked with different options).

See also: [ISO 1990], section 6.1.2.5; [Kernighan 1988], section A4.2.

4. Here is one possibility:

§650 ⟨ Step 3: Update balance factors after AVL insertion, with bitmasks 650 ⟩ ≡

```
for (p = y; p != n; p = p->avl_link[cache & 1], cache >>= 1)
  if ((cache & 1) == 0)
    p->avl_balance--;
  else p->avl_balance++;
```

Also, replace the declarations of *da*[] and *k* by these:

```
unsigned long cache = 0; /* Cached comparison results. */
int k = 0; /* Number of cached comparison results. */
```

and replace the second paragraph of code within the loop in step 1 by this:

```
if (p->avl_balance != 0)
  z = q, y = p, cache = 0, k = 0;
dir = cmp > 0;
if (dir)
  cache |= 1ul << k;
k++;
```

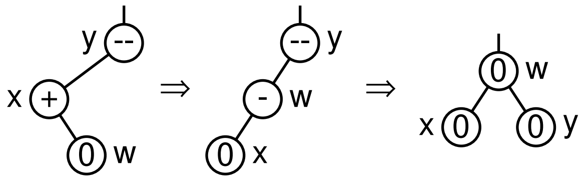
It is interesting to note that the speed difference between this version and the standard version was found to be negligible, when compiled with full optimization under GCC (both 2.95.4 and 3.0.3) on x86.

Section 5.4.4

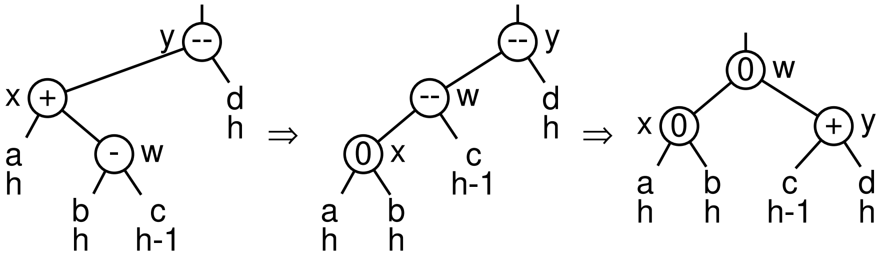
1. Because then y's right subtree would have height 1, so there's no way that y could have a $^{+}2$ balance factor.

2. The value of y is set during the search for *item* to point to the closest node above the insertion point that has a nonzero balance factor, so any node below y along this search path, including x, must have had a 0 balance factor originally. All such nodes are updated to have a nonzero balance factor later, during step 3. So x must have either a $-$ or $+$ balance factor at the time of rebalancing.

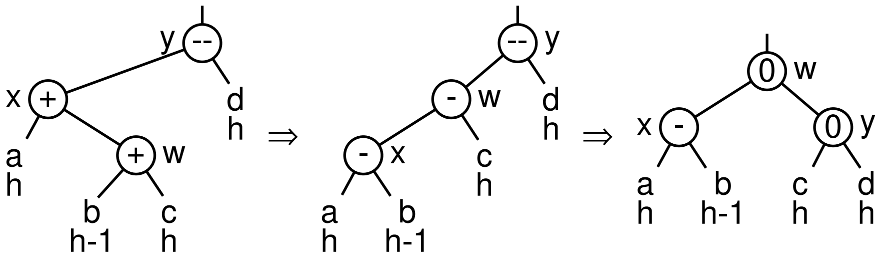
3.1.



3.2.



3.3.



4. w should replace y as the left or right child of z. y != z→*avl_link*[0] has the value 1 if y is the right child of z, or 0 if y is the left child. So the overall expression replaces y with w as a child of z.

The suggested substitution is a poor choice because if z == (**struct avl_node** ∗) &*tree*→*root*, z→*avl_link*[1] is undefined.

5. Yes.

Section 5.5.2

1. This approach cannot be used in Libavl (see Exercise 4.8-3).

§651 ⟨ Case 3 in AVL deletion, alternate version 651 ⟩ ≡

```
struct avl_node *s;
da[k] = 1;
pa[k++] = p;
for (;;) {
    da[k] = 0;
    pa[k++] = r;
    s = r->avl_link[0];
    if (s->avl_link[0] == NULL)
        break;
    r = s;
}
p->avl_data = s->avl_data;
```

$r \rightarrow avl\_link[0] = s \rightarrow avl\_link[1]$;
$p = s$;

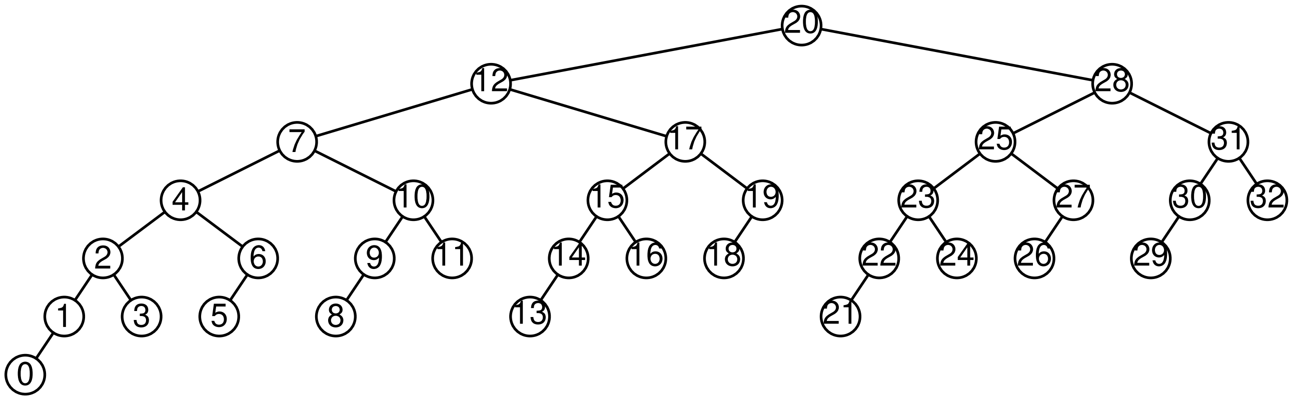
2. We could, if we use the standard LIBAVL code for deletion case 3. The alternate version in Exercise 1 modifies item data, which would cause the wrong value to be returned later.

Section 5.5.4

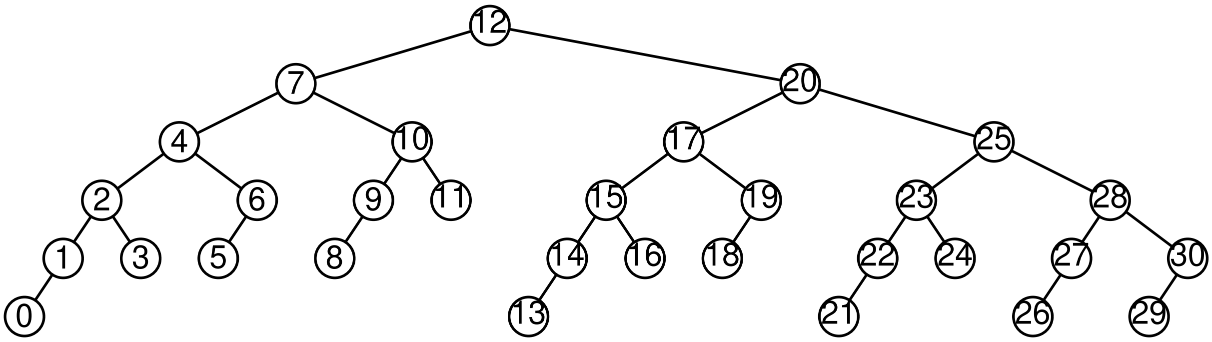
1. Tree y started out with a + balance factor, meaning that its right subtree is taller than its left. So, even if y's left subtree had height 0, its right subtree has at least height 1, meaning that y must have at least one right child.

2. Rebalancing is required at each level if, at every level of the tree, the deletion causes a $^+2$ or $^-2$ balance factor at a node p while there is a $^+1$ or $^-1$ balance factor at p's child opposite the deletion.

For example, consider the AVL tree below:



Deletion of node 32 in this tree leads to a $^-2$ balance factor on the left side of node 31, causing a right rotation at node 31. This shortens the right subtree of node 28, causing it to have a $^-2$ balance factor, leading to a right rotation there. This shortens the right subtree of node 20, causing it to have a $^-2$ balance factor, forcing a right rotation there, too. Here is the final tree:



Incidentally, our original tree was an example of a "Fibonacci tree", a kind of binary tree whose form is defined recursively, as follows. A Fibonacci tree of order 0 is an empty tree and a Fibonacci tree of order 1 is a single node. A Fibonacci tree of order $n \geq 2$ is a node whose left subtree is a Fibonacci tree of order $n - 1$ and whose right subtree is a Fibonacci tree of order $n - 2$. Our example is a Fibonacci tree of order 7. Any big-enough

Fibonacci tree will exhibit this pathological behavior upon AVL deletion of its maximum node.

Section 5.6

1. At this point in the code, p points to the *avl_data* member of an **struct avl_node**. We want a pointer to the **struct avl_node** itself. To do this, we just subtract the offset of the *avl_data* member within the structure. A cast to **char** $*$ is necessary before the subtraction, because *offsetof* returns a count of bytes, and a cast to **struct avl_node** $*$ afterward, to make the result the right type.

Chapter 6

Section 6.1

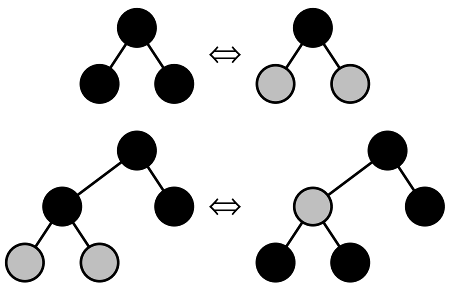
1. It must be a **complete binary tree** of exactly $2^n - 1$ nodes.

If a red-black tree contains only red nodes, on the other hand, it cannot have more than one node, because of rule 1.

2. If a red-black tree's root is red, then we can transform it into an equivalent red-black tree with a black root simply by recoloring the root. This cannot violate rule 1, because it does not introduce a red node. It cannot violate rule 2 because it only affects the number of black nodes along paths that pass through the root, and it affects all of those paths equally, by increasing the number of black nodes along them by one.

If, on the other hand, a red-black tree has a black root, we cannot in general recolor it to red, because this causes a violation of rule 1 if the root has a red child.

3. Yes and yes:



Section 6.2

1. C has a number of different namespaces. One of these is the namespace that contains **struct**, **union**, and **enum** tags. Names of structure members are in a namespace separate from this tag namespace, so it is okay to give an **enum** and a structure member the same name. On the other hand, it would be an error to give, e.g., a **struct** and an **enum** the same name.

Section 6.4.2

1. Inserting a red node can sometimes be done without breaking any rules. Inserting a black node will always break rule 2.

Section 6.4.3

1. We can't have $k == 1$, because then the new node would be the root, and the root doesn't have a parent that could be red. We don't need to rebalance $k == 2$, because the new node is a direct child of the root, and the root is always black.

2. Yes, it would, but if d has a red node as its root, case 1 will be selected instead.

Section 6.5.1

1. If p has no left child, that is, it is a leaf, then obviously we cannot swap colors. Now consider only the case where p does have a non-null left child x. Clearly, x must be red, because otherwise rule 2 would be violated at p. This means that p must be black to avoid a rule 1 violation. So the deletion will eliminate a black node, causing a rule 2 violation. This is exactly the sort of problem that the rebalancing step is designed to deal with, so we can rebalance starting from node x.

2. There are two cases in this algorithm, which uses a new **struct avl_node** $*$ variable named x. Regardless of which one is chosen, x has the same meaning afterward: it is the node that replaced one of the children of the node at top of stack, and may be `NULL` if the node removed was a leaf.

Case 1: If one of p's child pointers is `NULL`, then p can be replaced by the other child, or by `NULL` if both children are `NULL`:

§652 ⟨Step 2: Delete item from RB tree, alternate version 652⟩ ≡

```
if (p->rb_link[0] == NULL || p->rb_link[1] == NULL) {
    x = p->rb_link[0];
    if (x == NULL)
        x = p->rb_link[1];
}
```

See also §653 and §654.

Case 2: If both of p's child pointers are non-null, then we find p's successor and replace p's data by the successor's data, then delete the successor instead:

§653 ⟨Step 2: Delete item from RB tree, alternate version 652⟩ +≡

```
else {
    struct rb_node *y;

    pa[k] = p;
    da[k++] = 1;

    y = p->rb_link[1];
    while (y->rb_link[0] != NULL) {
        pa[k] = y;
        da[k++] = 0;
```

```
        y = y→rb_link[0];
    }
    x = y→rb_link[1];
    p→rb_data = y→rb_data;
    p = y;
}
```

In either case, we need to update the node above the deleted node to point to x.

§654 ⟨ Step 2: Delete item from RB tree, alternate version 652 ⟩ +≡

```
pa[k − 1]→rb_link[da[k − 1]] = x;
```

See also: [Cormen 1990], section 14.4.

Chapter 7

Section 7.2

1. An enumerated type is compatible with some C integer type, but the particular type is up to the C compiler. Many C compilers will always pick **int** as the type of an enumeration type. But we want to conserve space in the structure (see Exercise 1), so we specify **unsigned char** explicitly as the type.

See also: [ISO 1990], section 6.5.2.2; [ISO 1999], section 6.7.2.2.

Section 7.6

1. When we add a node to a formerly empty tree, this statement will set *tree→tbst_root*, thereby breaking the **if** statement's test.

Section 7.7

1. See Section 8.5.6 [Finding the Parent of a TBST Node], page 203. Function *find_parent*() is implemented in ⟨ Find parent of a TBST node 327 ⟩.

§655 ⟨ Find TBST node to delete, with parent node algorithm 655 ⟩ ≡

```
p = tree→tbst_root;
if (p == NULL)
    return NULL;
for (;;) {
    int cmp = tree→tbst_compare (item, p→tbst_data, tree→tbst_param);
    if (cmp == 0)
        break;
    p = p→tbst_link[cmp > 0];
}
q = find_parent (tree, p);
dir = q→tbst_link[0] != p;
```

See also: [Knuth 1997], exercise 2.3.1-19.

2. Yes. We can bind a pointer and a tag into a single structure, then use that structure for our links and for the root in the table structure.

```
/* A tagged link. */
struct tbst_link {
    struct tbst_node *tbst_ptr; /* Child pointer or thread. */
    unsigned char tbst_tag; /* Tag. */
};

/* A threaded binary search tree node. */
struct tbst_node {
    struct tbst_link tbst_link[2]; /* Links. */
    void *tbst_data; /* Pointer to data. */
};

/* Tree data structure. */
struct tbst_table {
    struct tbst_link tbst_root; /* Tree's root; tag is unused. */
    tbst_comparison_func *tbst_compare; /* Comparison function. */
    void *tbst_param; /* Extra argument to tbst_compare. */
    struct libavl_allocator *tbst_alloc; /* Memory allocator. */
    size_t tbst_count; /* Number of items in tree. */
};
```

The main disadvantage of this approach is in storage space: many machines have alignment restrictions for pointers, so the nonadjacent **unsigned char**s cause space to be wasted. Alternatively, we could keep the current arrangement of the node structure and change *tbst_root* in **struct tbst_table** from a pointer to an instance of **struct tbst_node**.

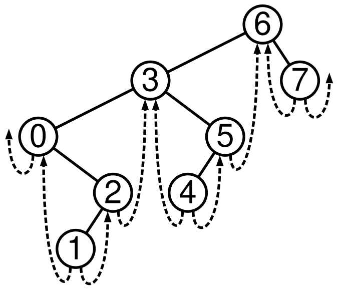
3. Much simpler than the implementation given before:

§656 ⟨ Case 4 in TBST deletion, alternate version 656 ⟩ ≡

```
struct tbst_node *s = r->tbst_link[0];
while (s->tbst_tag[0] == TBST_CHILD) {
    r = s;
    s = r->tbst_link[0];
}
p->tbst_data = s->tbst_data;
if (s->tbst_tag[1] == TBST_THREAD) {
    r->tbst_tag[0] = TBST_THREAD;
    r->tbst_link[0] = p;
} else {
    q = r->tbst_link[0] = s->tbst_link[1];
    while (q->tbst_tag[0] == TBST_CHILD)
        q = q->tbst_link[0];
    q->tbst_link[0] = p;
}
p = s;
```

This code is included in §658.

4. If all the possible deletions from a given TBST are considered, then no link will be followed more than once to update a left thread, and similarly for right threads. Averaged over all the possible deletions, this is a constant. For example, take the following TBST:



Consider right threads that must be updated on deletion. Nodes 2, 3, 5, and 6 have right threads pointing to them. To update the right thread to node 2, we follow the link to node 1; to update node 3's, we move to 0, then 2; for node 5, we move to node 4; and for node 6, we move to 3, then 5. No link is followed more than once. Here's a summary table:

Node	Right Thread Follows	Left Thread Follows
0:	(none)	2, 1
1:	(none)	(none)
2:	1	(none)
3:	0, 2	5, 4
4:	(none)	(none)
5:	4	(none)
6:	3, 5	7
7:	(none)	(none)

The important point here is that no number appears twice within a column.

Section 7.9

1. Suppose a node has a right thread. If the node has no left subtree, then the thread will be followed immediately when the node is reached. If the node does have a left subtree, then the left subtree will be traversed, and when the traversal is finished the node's predecessor's right thread will be followed back to the node, then its right thread will be followed. The node cannot be skipped, because all the nodes in its left subtree are less than it, so none of the right threads in its left subtree can skip beyond it.

2. The biggest potential for optimization probably comes from *tbst_copy*()'s habit of always keeping the TBST fully consistent as it builds it, which causes repeated assignments to link fields in order to keep threads correct at all times. The unthreaded BST copy function *bst_copy*() waited to initialize fields until it was ready for them. It may be possible, though difficult, to do this in *tbst_copy*() as well.

Inlining and specializing *copy_node*() is a cheaper potential speedup.

Chapter 8

Section 8.1

1. No: the compiler may insert padding between or after structure members. For example, today (2002) the most common desktop computers have 32-bit pointers and and 8-bit **char**s. On these systems, most compilers will pad out structures to a multiple of 32 bits. Under these circumstances, **struct tavl_node** is no larger than **struct avl_node**, because (32 + 32 + 8) and (32 + 32 + 8 + 8 + 8) both round up to the same multiple of 32 bits, or 96 bits.

Section 8.2

1. We just have to special-case the possibility that subtree b is a thread.

```
/* Rotates right at *yp. */
static void rotate_right (struct tavl_node **yp) {
    struct tavl_node *y = *yp;
    struct tavl_node *x = y->tavl_link[0];
    if (x->tavl_tag[1] == TAVL_THREAD) {
        x->tavl_tag[1] = TAVL_CHILD;
        y->tavl_tag[0] = TAVL_THREAD;
        y->tavl_link[0] = x;
    }
    else y->tavl_link[0] = x->tavl_link[1];
    x->tavl_link[1] = y;
    *yp = x;
}
/* Rotates left at *xp. */
static void rotate_left (struct tavl_node **xp) {
    struct tavl_node *x = *xp;
    struct tavl_node *y = x->tavl_link[1];
    if (y->tavl_tag[0] == TAVL_THREAD) {
        y->tavl_tag[0] = TAVL_CHILD;
        x->tavl_tag[1] = TAVL_THREAD;
        x->tavl_link[1] = y;
    }
    else x->tavl_link[1] = y->tavl_link[0];
    y->tavl_link[0] = x;
    *xp = y;
}
```

Section 8.4.2

1. Besides this change, the statement

```
z->tavl_link[y != z->tavl_link[0]] = w;
```

must be removed from ⟨ Step 4: Rebalance after TAVL insertion 304 ⟩, and copies added to the end of ⟨ Rebalance TAVL tree after insertion in right subtree 308 ⟩ and ⟨ Rebalance for − balance factor in TAVL insertion in left subtree 306 ⟩.

§657 ⟨ Rebalance + balance in TAVL insertion in left subtree, alternate version 657 ⟩ ≡
$w = x \rightarrow tavl\_link[1]$;
$rotate\_left$ ($\&y \rightarrow tavl\_link[0]$);
$rotate\_right$ ($\&z \rightarrow tavl\_link[y \mathrel{!=} z \rightarrow tavl\_link[0]]$);
if ($w \rightarrow tavl\_balance == {}^-1$) $x \rightarrow tavl\_balance = 0$, $y \rightarrow tavl\_balance = {}^+1$;
else if ($w \rightarrow tavl\_balance == 0$) $x \rightarrow tavl\_balance = y \rightarrow tavl\_balance = 0$;
else /* $w \rightarrow tavl\_balance == {}^+1$ */ $x \rightarrow tavl\_balance = {}^-1$, $y \rightarrow tavl\_balance = 0$;
$w \rightarrow tavl\_balance = 0$;

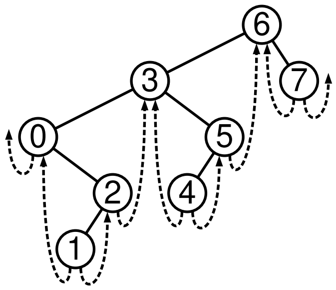
Section 8.5.2

1. We can just reuse the alternate implementation of case 4 for TBST deletion, following it by setting up q and dir as the rebalancing step expects them to be.

§658 ⟨ Case 4 in TAVL deletion, alternate version 658 ⟩ ≡
⟨ Case 4 in TBST deletion, alternate version; tbst ⇒ tavl 656 ⟩
$q = r$;
$dir = 0$;

Section 8.5.6

1. Our argument here is similar to that in Exercise 7.7-4. Consider the links that are traversed to successfully find the parent of each node, besides the root, in the tree shown below. Do not include links followed on the side that does not lead to the node's parent. Because there are never more of these than on the successful side, they add only a constant time to the algorithm and can be ignored.



The table below lists the links followed. The important point is that no link is listed twice.

Node	Links Followed to Node's Parent
0	0→2, 2→3
1	1→2
2	2→1, 1→0
3	3→5, 5→6

4	4→5
5	5→4, 4→3
6	(root)
7	7→6

This generalizes to all TBSTs. Because a TBST with n nodes contains only $2n$ links, this means we have an upper bound on finding the parent of every node in a TBST of at most $2n$ successful link traversals plus $2n$ unsuccessful link traversals. Averaging $4n$ over n nodes, we get an upper bound of $4n/n \equiv 4$ link traversals, on average, to find the parent of a given node.

This upper bound applies only to the average case, not to the case of any individual node. In particular, it does not say that the usage of the algorithm in *tavl_delete*() will exhibit average behavior. In practice, however, the performance of this algorithm in *tavl_delete*() seems quite acceptable. See Exercise 3 for an alternative with more certain behavior.

2. Instead of storing a null pointer in the left thread of the least node and the right thread of the greatest node, store a pointer to a node "above the root". To make this work properly, *tavl_root* will have to become an actual node, not just a node pointer, because otherwise trying to find its right child would invoke undefined behavior. Also, both of *tavl_root*'s children would have to be the root node.

This is probably not worth it. On the surface it seems like a good idea but ugliness lurks beneath.

3. The necessary changes are pervasive, so the complete code for the modified function is presented below. The search step is borrowed from TRB deletion, presented in the next chapter.

§659 ⟨ TAVL item deletion function, with stack 659 ⟩ ≡
```
void *tavl_delete (struct tavl_table *tree, const void *item) {
    /* Stack of nodes. */
    struct tavl_node *pa[TAVL_MAX_HEIGHT]; /* Nodes. */
    unsigned char da[TAVL_MAX_HEIGHT]; /* tavl_link[] indexes. */
    int k = 0; /* Stack pointer. */

    struct tavl_node *p; /* Traverses tree to find node to delete. */
    int cmp; /* Result of comparison between item and p. */
    int dir; /* Child of p to visit next. */

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search TRB tree for item to delete; trb ⇒ tavl 350 ⟩
    ⟨ Step 2: Delete item from TAVL tree, with stack 660 ⟩
    ⟨ Steps 3 and 4: Update balance factors and rebalance after TAVL deletion, with stack 665 ⟩

    return (void *) item;
}
```

§660 ⟨ Step 2: Delete item from TAVL tree, with stack 660 ⟩ ≡
```
if (p→tavl_tag[1] == TAVL_THREAD) {
    if (p→tavl_tag[0] == TAVL_CHILD)
        { ⟨ Case 1 in TAVL deletion, with stack 661 ⟩ }
```

```
    else { ⟨Case 2 in TAVL deletion, with stack 662⟩ }
} else {
    struct tavl_node *r = p->tavl_link[1];
    if (r->tavl_tag[0] == TAVL_THREAD)
        { ⟨Case 3 in TAVL deletion, with stack 663⟩ }
    else { ⟨Case 4 in TAVL deletion, with stack 664⟩ }
}
tree->tavl_count--;
tree->tavl_alloc->libavl_free (tree->tavl_alloc, p);
```

This code is included in §659.

§661 ⟨Case 1 in TAVL deletion, with stack 661⟩ ≡

```
struct tavl_node *r = p->tavl_link[0];
while (r->tavl_tag[1] == TAVL_CHILD)
    r = r->tavl_link[1];
r->tavl_link[1] = p->tavl_link[1];
pa[k - 1]->tavl_link[da[k - 1]] = p->tavl_link[0];
```

This code is included in §660.

§662 ⟨Case 2 in TAVL deletion, with stack 662⟩ ≡

```
pa[k - 1]->tavl_link[da[k - 1]] = p->tavl_link[da[k - 1]];
if (pa[k - 1] != (struct tavl_node *) &tree->tavl_root)
    pa[k - 1]->tavl_tag[da[k - 1]] = TAVL_THREAD;
```

This code is included in §660.

§663 ⟨Case 3 in TAVL deletion, with stack 663⟩ ≡

```
r->tavl_link[0] = p->tavl_link[0];
r->tavl_tag[0] = p->tavl_tag[0];
r->tavl_balance = p->tavl_balance;
if (r->tavl_tag[0] == TAVL_CHILD) {
    struct tavl_node *x = r->tavl_link[0];
    while (x->tavl_tag[1] == TAVL_CHILD)
        x = x->tavl_link[1];
    x->tavl_link[1] = r;
}
pa[k - 1]->tavl_link[da[k - 1]] = r;
da[k] = 1;
pa[k++] = r;
```

This code is included in §660.

§664 ⟨Case 4 in TAVL deletion, with stack 664⟩ ≡

```
struct tavl_node *s;
int j = k++;
for (;;) {
    da[k] = 0;
    pa[k++] = r;
    s = r->tavl_link[0];
    if (s->tavl_tag[0] == TAVL_THREAD)
        break;
```

```
    r = s;
}
da[j] = 1;
pa[j] = pa[j - 1]->tavl_link[da[j - 1]] = s;
if (s->tavl_tag[1] == TAVL_CHILD)
    r->tavl_link[0] = s->tavl_link[1];
else {
    r->tavl_link[0] = s;
    r->tavl_tag[0] = TAVL_THREAD;
}
s->tavl_balance = p->tavl_balance;
s->tavl_link[0] = p->tavl_link[0];
if (p->tavl_tag[0] == TAVL_CHILD) {
    struct tavl_node *x = p->tavl_link[0];
    while (x->tavl_tag[1] == TAVL_CHILD)
        x = x->tavl_link[1];
    x->tavl_link[1] = s;
    s->tavl_tag[0] = TAVL_CHILD;
}
s->tavl_link[1] = p->tavl_link[1];
s->tavl_tag[1] = TAVL_CHILD;
```

This code is included in §660.

§665 ⟨Steps 3 and 4: Update balance factors and rebalance after TAVL deletion, with stack 665⟩ ≡

```
assert (k > 0);
while (--k > 0) {
    struct tavl_node *y = pa[k];
    if (da[k] == 0) {
        y->tavl_balance++;
        if (y->tavl_balance == +1) break;
        else if (y->tavl_balance == +2) {
            ⟨Step 4: Rebalance after TAVL deletion, with stack 666⟩
        }
    } else {
        ⟨Steps 3 and 4: Symmetric case in TAVL deletion, with stack 667⟩
    }
}
```

This code is included in §659.

§666 ⟨Step 4: Rebalance after TAVL deletion, with stack 666⟩ ≡

```
struct tavl_node *x = y->tavl_link[1];
assert (x != NULL);
if (x->tavl_balance == -1) {
    struct tavl_node *w;
    ⟨Rebalance for - balance factor in TAVL insertion in right subtree 310⟩
    pa[k - 1]->tavl_link[da[k - 1]] = w;
```

```
}
else if (x->tavl_balance == 0) {
    y->tavl_link[1] = x->tavl_link[0];
    x->tavl_link[0] = y;
    x->tavl_balance = -1;
    y->tavl_balance = +1;
    pa[k - 1]->tavl_link[da[k - 1]] = x;
    break;
}
else /* x->tavl_balance == +1 */ {
    if (x->tavl_tag[0] == TAVL_CHILD)
        y->tavl_link[1] = x->tavl_link[0];
    else {
        y->tavl_tag[1] = TAVL_THREAD;
        x->tavl_tag[0] = TAVL_CHILD;
    }
    x->tavl_link[0] = y;
    x->tavl_balance = y->tavl_balance = 0;
    pa[k - 1]->tavl_link[da[k - 1]] = x;
}
```

This code is included in §665.

§667 ⟨Steps 3 and 4: Symmetric case in TAVL deletion, with stack 667⟩ ≡

```
y->tavl_balance--;
if (y->tavl_balance == -1) break;
else if (y->tavl_balance == -2) {
    struct tavl_node *x = y->tavl_link[0];
    assert (x != NULL);
    if (x->tavl_balance == +1) {
        struct tavl_node *w;
        ⟨Rebalance for + balance factor in TAVL insertion in left subtree 307⟩
        pa[k - 1]->tavl_link[da[k - 1]] = w;
    }
    else if (x->tavl_balance == 0) {
        y->tavl_link[0] = x->tavl_link[1];
        x->tavl_link[1] = y;
        x->tavl_balance = +1;
        y->tavl_balance = -1;
        pa[k - 1]->tavl_link[da[k - 1]] = x;
        break;
    }
    else /* x->tavl_balance == -1 */ {
        if (x->tavl_tag[1] == TAVL_CHILD)
            y->tavl_link[0] = x->tavl_link[1];
        else {
            y->tavl_tag[0] = TAVL_THREAD;
            x->tavl_tag[1] = TAVL_CHILD;
```

```
        }
        x->tavl_link[1] = y;
        x->tavl_balance = y->tavl_balance = 0;
        pa[k - 1]->tavl_link[da[k - 1]] = x;
    }
}
```

This code is included in §665.

Chapter 9

Section 9.3.3

1. For a brief explanation of an algorithm similar to the one here, see Section 15.3 [Inserting into a PRB Tree], page 308.

§668 ⟨ TRB item insertion function, without stack 668 ⟩ ≡

```
⟨ Find parent of a TBST node; tbst ⇒ trb 327 ⟩
void **trb_probe (struct trb_table *tree, void *item) {
    struct trb_node *p; /* Traverses tree looking for insertion point. */
    struct trb_node *n; /* Newly inserted node. */
    int dir; /* Side of p on which n is inserted. */

    assert (tree != NULL && item != NULL);

    ⟨ Step 1: Search TBST for insertion point; tbst ⇒ trb 255 ⟩
    ⟨ Step 2: Insert TRB node 339 ⟩
    p = n;
    for (;;) {
        struct trb_node *f, *g;

        f = find_parent (tree, p);
        if (f == (struct trb_node *) &tree->trb_root || f->trb_color == TRB_BLACK)
            break;

        g = find_parent (tree, f);
        if (g == (struct trb_node *) &tree->trb_root)
            break;

        if (g->trb_link[0] == f) {
            struct trb_node *y = g->trb_link[1];
            if (g->trb_tag[1] == TRB_CHILD && y->trb_color == TRB_RED) {
                f->trb_color = y->trb_color = TRB_BLACK;
                g->trb_color = TRB_RED;
                p = g;
            } else {
                struct trb_node *c, *x;

                if (f->trb_link[0] == p)
                    y = f;
                else {
                    x = f;
```

```
                y = x→trb_link[1];
                x→trb_link[1] = y→trb_link[0];
                y→trb_link[0] = x;
                g→trb_link[0] = y;
                if (y→trb_tag[0] == TRB_THREAD) {
                    y→trb_tag[0] = TRB_CHILD;
                    x→trb_tag[1] = TRB_THREAD;
                    x→trb_link[1] = y;
                }
            }
            c = find_parent (tree, g);
            c→trb_link[c→trb_link[0] != g] = y;
            x = g;
            x→trb_color = TRB_RED;
            y→trb_color = TRB_BLACK;
            x→trb_link[0] = y→trb_link[1];
            y→trb_link[1] = x;
            if (y→trb_tag[1] == TRB_THREAD) {
                y→trb_tag[1] = TRB_CHILD;
                x→trb_tag[0] = TRB_THREAD;
                x→trb_link[0] = y;
            }
            break;
        }
    } else {
        struct trb_node *y = g→trb_link[0];
        if (g→trb_tag[0] == TRB_CHILD && y→trb_color == TRB_RED) {
            f→trb_color = y→trb_color = TRB_BLACK;
            g→trb_color = TRB_RED;
            p = g;
        } else {
            struct trb_node *c, *x;
            if (f→trb_link[1] == p)
                y = f;
            else {
                x = f;
                y = x→trb_link[0];
                x→trb_link[0] = y→trb_link[1];
                y→trb_link[1] = x;
                g→trb_link[1] = y;
                if (y→trb_tag[1] == TRB_THREAD) {
                    y→trb_tag[1] = TRB_CHILD;
                    x→trb_tag[0] = TRB_THREAD;
                    x→trb_link[0] = y;
                }
```

```
                }
                c = find_parent (tree, g);
                c->trb_link[c->trb_link[0] != g] = y;
                x = g;
                x->trb_color = TRB_RED;
                y->trb_color = TRB_BLACK;
                x->trb_link[1] = y->trb_link[0];
                y->trb_link[0] = x;
                if (y->trb_tag[0] == TRB_THREAD) {
                    y->trb_tag[0] = TRB_CHILD;
                    x->trb_tag[1] = TRB_THREAD;
                    x->trb_link[1] = y;
                }
                break;
            }
        }
    }
    tree->trb_root->trb_color = TRB_BLACK;
    return &n->trb_data;
}
```

Section 9.4.2

1.

§669 ⟨Case 4 in TRB deletion, alternate version 669⟩ ≡

```
struct trb_node *s;
da[k] = 1;
pa[k++] = p;
for (;;) {
    da[k] = 0;
    pa[k++] = r;
    s = r->trb_link[0];
    if (s->trb_tag[0] == TRB_THREAD)
        break;
    r = s;
}
p->trb_data = s->trb_data;
if (s->trb_tag[1] == TRB_THREAD) {
    r->trb_tag[0] = TRB_THREAD;
    r->trb_link[0] = p;
} else {
    struct trb_node *t = r->trb_link[0] = s->trb_link[1];
    while (t->trb_tag[0] == TRB_CHILD)
        t = t->trb_link[0];
    t->trb_link[0] = p;
```

```
}
p = s;
```

Section 9.4.5

1. The code used in the rebalancing loop is related to ⟨Step 3: Rebalance tree after PRB deletion 571⟩. Variable x is initialized by step 2 here, though, because otherwise the pseudo-root node would be required to have a *trb_tag*[] member.

§670 ⟨TRB item deletion function, without stack 670⟩ ≡
⟨Find parent of a TBST node; tbst ⇒ trb 327⟩

```
void *trb_delete (struct trb_table *tree, const void *item) {
    struct trb_node *p; /* Node to delete. */
    struct trb_node *q; /* Parent of p. */
    struct trb_node *x; /* Node we might want to recolor red (maybe NULL). */
    struct trb_node *f; /* Parent of x. */
    struct trb_node *g; /* Parent of f. */
    int dir, cmp;
    assert (tree != NULL && item != NULL);
    ⟨Step 1: Search TAVL tree for item to delete; tavl ⇒ trb 312⟩
    if (p→trb_tag[1] == TRB_THREAD) {
        if (p→trb_tag[0] == TRB_CHILD) {
            struct trb_node *t = p→trb_link[0];
            while (t→trb_tag[1] == TRB_CHILD)
                t = t→trb_link[1];
            t→trb_link[1] = p→trb_link[1];
            x = q→trb_link[dir] = p→trb_link[0];
        } else {
            q→trb_link[dir] = p→trb_link[dir];
            if (q != (struct trb_node *) &tree→trb_root)
                q→trb_tag[dir] = TRB_THREAD;
            x = NULL;
        }
        f = q;
    } else {
        enum trb_color t;
        struct trb_node *r = p→trb_link[1];
        if (r→trb_tag[0] == TRB_THREAD) {
            r→trb_link[0] = p→trb_link[0];
            r→trb_tag[0] = p→trb_tag[0];
            if (r→trb_tag[0] == TRB_CHILD) {
                struct trb_node *t = r→trb_link[0];
                while (t→trb_tag[1] == TRB_CHILD)
                    t = t→trb_link[1];
                t→trb_link[1] = r;
            }
```

```
            q->trb_link[dir] = r;
            x = r->trb_tag[1] == TRB_CHILD ? r->trb_link[1] : NULL;
            t = r->trb_color;
            r->trb_color = p->trb_color;
            p->trb_color = t;
            f = r;
            dir = 1;
        } else {
            struct trb_node *s;
            for (;;) {
                s = r->trb_link[0];
                if (s->trb_tag[0] == TRB_THREAD)
                    break;
                r = s;
            }
            if (s->trb_tag[1] == TRB_CHILD)
                x = r->trb_link[0] = s->trb_link[1];
            else {
                r->trb_link[0] = s;
                r->trb_tag[0] = TRB_THREAD;
                x = NULL;
            }
            s->trb_link[0] = p->trb_link[0];
            if (p->trb_tag[0] == TRB_CHILD) {
                struct trb_node *t = p->trb_link[0];
                while (t->trb_tag[1] == TRB_CHILD)
                    t = t->trb_link[1];
                t->trb_link[1] = s;
                s->trb_tag[0] = TRB_CHILD;
            }
            s->trb_link[1] = p->trb_link[1];
            s->trb_tag[1] = TRB_CHILD;
            t = s->trb_color;
            s->trb_color = p->trb_color;
            p->trb_color = t;
            q->trb_link[dir] = s;
            f = r;
            dir = 0;
        }
    }
    if (p->trb_color == TRB_BLACK) {
        for (;;)
            {
            if (x != NULL && x->trb_color == TRB_RED) {
                x->trb_color = TRB_BLACK;
```

```
        break;
    }
    if (f == (struct trb_node *) &tree->trb_root)
        break;
    g = find_parent (tree, f);
    if (dir == 0) {
        struct trb_node *w = f->trb_link[1];
        if (w->trb_color == TRB_RED) {
            w->trb_color = TRB_BLACK;
            f->trb_color = TRB_RED;
            f->trb_link[1] = w->trb_link[0];
            w->trb_link[0] = f;
            g->trb_link[g->trb_link[0] != f] = w;
            g = w;
            w = f->trb_link[1];
        }
        if ((w->trb_tag[0] == TRB_THREAD
             || w->trb_link[0]->trb_color == TRB_BLACK)
            && (w->trb_tag[1] == TRB_THREAD
                || w->trb_link[1]->trb_color == TRB_BLACK)) w->trb_color = TRB_RED;
        else {
            if (w->trb_tag[1] == TRB_THREAD
                || w->trb_link[1]->trb_color == TRB_BLACK) {
                struct trb_node *y = w->trb_link[0];
                y->trb_color = TRB_BLACK;
                w->trb_color = TRB_RED;
                w->trb_link[0] = y->trb_link[1];
                y->trb_link[1] = w;
                w = f->trb_link[1] = y;
                if (w->trb_tag[1] == TRB_THREAD) {
                    w->trb_tag[1] = TRB_CHILD;
                    w->trb_link[1]->trb_tag[0] = TRB_THREAD;
                    w->trb_link[1]->trb_link[0] = w;
                }
            }
            w->trb_color = f->trb_color;
            f->trb_color = TRB_BLACK;
            w->trb_link[1]->trb_color = TRB_BLACK;
            f->trb_link[1] = w->trb_link[0];
            w->trb_link[0] = f;
            g->trb_link[g->trb_link[0] != f] = w;
            if (w->trb_tag[0] == TRB_THREAD) {
                w->trb_tag[0] = TRB_CHILD;
                f->trb_tag[1] = TRB_THREAD;
                f->trb_link[1] = w;
```

```
            }
            break;
        }
    } else {
        struct trb_node *w = f->trb_link[0];
        if (w->trb_color == TRB_RED) {
            w->trb_color = TRB_BLACK;
            f->trb_color = TRB_RED;
            f->trb_link[0] = w->trb_link[1];
            w->trb_link[1] = f;
            g->trb_link[g->trb_link[0] != f] = w;
            g = w;
            w = f->trb_link[0];
        }
        if ((w->trb_tag[0] == TRB_THREAD
             || w->trb_link[0]->trb_color == TRB_BLACK)
            && (w->trb_tag[1] == TRB_THREAD
                || w->trb_link[1]->trb_color == TRB_BLACK)) w->trb_color = TRB_RED;
        else {
            if (w->trb_tag[0] == TRB_THREAD
                || w->trb_link[0]->trb_color == TRB_BLACK) {
                struct trb_node *y = w->trb_link[1];
                y->trb_color = TRB_BLACK;
                w->trb_color = TRB_RED;
                w->trb_link[1] = y->trb_link[0];
                y->trb_link[0] = w;
                w = f->trb_link[0] = y;
                if (w->trb_tag[0] == TRB_THREAD) {
                    w->trb_tag[0] = TRB_CHILD;
                    w->trb_link[0]->trb_tag[1] = TRB_THREAD;
                    w->trb_link[0]->trb_link[1] = w;
                }
            }
            w->trb_color = f->trb_color;
            f->trb_color = TRB_BLACK;
            w->trb_link[0]->trb_color = TRB_BLACK;
            f->trb_link[0] = w->trb_link[1];
            w->trb_link[1] = f;
            g->trb_link[g->trb_link[0] != f] = w;
            if (w->trb_tag[1] == TRB_THREAD) {
                w->trb_tag[1] = TRB_CHILD;
                f->trb_tag[0] = TRB_THREAD;
                f->trb_link[0] = w;
            }
            break;
```

```
                }
            }
            x = f;
            f = find_parent (tree, x);
            if (f == (struct trb_node *) &tree->trb_root)
                break;
            dir = f->trb_link[0] != x;
        }
    }
    tree->trb_alloc->libavl_free (tree->trb_alloc, p);
    tree->trb_count--;
    return (void *) item;
}
```

Chapter 10

1. If we already have right-threaded trees, then we can get the benefits of a left-threaded tree just by reversing the sense of the comparison function, so there is no additional benefit to left-threaded trees.

Section 10.5.1

1.

§671 ⟨ Case 4 in right-looking RTBST deletion, alternate version 671 ⟩ ≡

```
struct rtbst_node *s = r->rtbst_link[0];
while (s->rtbst_link[0] != NULL) {
    r = s;
    s = r->rtbst_link[0];
}
p->rtbst_data = s->rtbst_data;
if (s->rtbst_rtag == RTBST_THREAD)
    r->rtbst_link[0] = NULL;
else r->rtbst_link[0] = s->rtbst_link[1];
p = s;
```

Section 10.5.2

1. This alternate version is not really an improvement: it runs up against the same problem as right-looking deletion, so it sometimes needs to search for a predecessor.

§672 ⟨ Case 4 in left-looking RTBST deletion, alternate version 672 ⟩ ≡

```
struct rtbst_node *s = r->rtbst_link[1];
while (s->rtbst_rtag == RTBST_CHILD) {
    r = s;
    s = r->rtbst_link[1];
}
```

```
p->rtbst_data = s->rtbst_data;
if (s->rtbst_link[0] != NULL) {
    struct rtbst_node *t = s->rtbst_link[0];
    while (t->rtbst_rtag == RTBST_CHILD)
        t = t->rtbst_link[1];
    t->rtbst_link[1] = p;
    r->rtbst_link[1] = s->rtbst_link[0];
} else {
    r->rtbst_link[1] = p;
    r->rtbst_rtag = RTBST_THREAD;
}
p = s;
```

Chapter 11

Section 11.3

1.

```
/* Rotates right at *yp. */
static void rotate_right (struct rtbst_node **yp) {
    struct rtbst_node *y = *yp;
    struct rtbst_node *x = y->rtbst_link[0];
    if (x->rtbst_rtag[1] == RTBST_THREAD) {
        x->rtbst_rtag = RTBST_CHILD;
        y->rtbst_link[0] = NULL;
    }
    else y->rtbst_link[0] = x->rtbst_link[1];
    x->rtbst_link[1] = y;
    *yp = x;
}
/* Rotates left at *xp. */
static void rotate_left (struct rtbst_node **xp) {
    struct rtbst_node *x = *xp;
    struct rtbst_node *y = x->rtbst_link[1];
    if (y->rtbst_link[0] == NULL) {
        x->rtbst_rtag = RTBST_THREAD;
        x->rtbst_link[1] = y;
    }
    else x->rtbst_link[1] = y->rtbst_link[0];
    y->rtbst_link[0] = x;
    *xp = y;
}
```

Section 11.5.4

1. There is no general efficient algorithm to find the parent of a node in an RTAVL tree. The lack of left threads means that half the time we must do a full search from the top of the tree. This would increase the execution time for deletion unacceptably.

2.

§673 ⟨ Step 2: Delete RTAVL node, right-looking 673 ⟩ ≡

```
if (p→rtavl_rtag == RTAVL_THREAD) {
    if (p→rtavl_link[0] != NULL)
        { ⟨ Case 1 in RTAVL deletion, right-looking 674 ⟩ }
    else { ⟨ Case 2 in RTAVL deletion, right-looking 675 ⟩ }
} else {
    struct rtavl_node *r = p→rtavl_link[1];
    if (r→rtavl_link[0] == NULL)
        { ⟨ Case 3 in RTAVL deletion, right-looking 676 ⟩ }
    else { ⟨ Case 4 in RTAVL deletion, right-looking 677 ⟩ }
}
tree→rtavl_alloc→libavl_free (tree→rtavl_alloc, p);
```

§674 ⟨ Case 1 in RTAVL deletion, right-looking 674 ⟩ ≡

```
struct rtavl_node *t = p→rtavl_link[0];
while (t→rtavl_rtag == RTAVL_CHILD)
    t = t→rtavl_link[1];
t→rtavl_link[1] = p→rtavl_link[1];
pa[k − 1]→rtavl_link[da[k − 1]] = p→rtavl_link[0];
```

This code is included in §673.

§675 ⟨ Case 2 in RTAVL deletion, right-looking 675 ⟩ ≡

```
pa[k − 1]→rtavl_link[da[k − 1]] = p→rtavl_link[da[k − 1]];
if (da[k − 1] == 1)
    pa[k − 1]→rtavl_rtag = RTAVL_THREAD;
```

This code is included in §673.

§676 ⟨ Case 3 in RTAVL deletion, right-looking 676 ⟩ ≡

```
r→rtavl_link[0] = p→rtavl_link[0];
if (r→rtavl_link[0] != NULL) {
    struct rtavl_node *t = r→rtavl_link[0];
    while (t→rtavl_rtag == RTAVL_CHILD)
        t = t→rtavl_link[1];
    t→rtavl_link[1] = r;
}
pa[k − 1]→rtavl_link[da[k − 1]] = r;
r→rtavl_balance = p→rtavl_balance;
da[k] = 1;
pa[k++] = r;
```

This code is included in §673.

§677 ⟨ Case 4 in RTAVL deletion, right-looking 677 ⟩ ≡

```
struct rtavl_node *s;
int j = k++;
```

```
for (;;) {
    da[k] = 0;
    pa[k++] = r;
    s = r->rtavl_link[0];
    if (s->rtavl_link[0] == NULL)
        break;

    r = s;
}
da[j] = 1;
pa[j] = pa[j - 1]->rtavl_link[da[j - 1]] = s;
if (s->rtavl_rtag == RTAVL_CHILD)
    r->rtavl_link[0] = s->rtavl_link[1];
else r->rtavl_link[0] = NULL;
if (p->rtavl_link[0] != NULL) {
    struct rtavl_node *t = p->rtavl_link[0];
    while (t->rtavl_rtag == RTAVL_CHILD)
        t = t->rtavl_link[1];
    t->rtavl_link[1] = s;
}
s->rtavl_link[0] = p->rtavl_link[0];
s->rtavl_link[1] = p->rtavl_link[1];
s->rtavl_rtag = RTAVL_CHILD;
s->rtavl_balance = p->rtavl_balance;
```

This code is included in §673.

3.

§678 ⟨ Case 4 in RTAVL deletion, alternate version 678 ⟩ ≡

```
struct rtavl_node *s;
da[k] = 0;
pa[k++] = p;
for (;;) {
    da[k] = 1;
    pa[k++] = r;
    s = r->rtavl_link[1];
    if (s->rtavl_rtag == RTAVL_THREAD)
        break;
    r = s;
}
if (s->rtavl_link[0] != NULL) {
    struct rtavl_node *t = s->rtavl_link[0];
    while (t->rtavl_rtag == RTAVL_CHILD)
        t = t->rtavl_link[1];
    t->rtavl_link[1] = p;
}
p->rtavl_data = s->rtavl_data;
```

```
if (s→rtavl_link[0] != NULL)
      r→rtavl_link[1] = s→rtavl_link[0];
else {
      r→rtavl_rtag = RTAVL_THREAD;
      r→rtavl_link[1] = p;
}
p = s;
```

Chapter 13

Section 13.4

1. No. It would work, except for the important special case where q is the pseudo-root but *p→pbst_parent* is `NULL`.

Section 13.7

1.

§679 ⟨ PBST balance function, with integrated parent updates 679 ⟩ ≡

```
⟨ BST to vine function; bst ⇒ pbst 89 ⟩
⟨ Vine to balanced PBST function, with parent updates 680 ⟩
void pbst_balance (struct pbst_table *tree) {
      assert (tree != NULL);
      tree_to_vine (tree);
      vine_to_tree (tree);
}
```

§680 ⟨ Vine to balanced PBST function, with parent updates 680 ⟩ ≡

```
⟨ PBST compression function 682 ⟩
static void vine_to_tree (struct pbst_table *tree) {
      unsigned long vine; /* Number of nodes in main vine. */
      unsigned long leaves; /* Nodes in incomplete bottom level, if any. */
      int height; /* Height of produced balanced tree. */
      struct pbst_node *p, *q; /* Current visited node and its parent. */
      ⟨ Calculate leaves; bst ⇒ pbst 91 ⟩
      ⟨ Reduce vine general case to special case; bst ⇒ pbst 92 ⟩
      ⟨ Make special case vine into balanced tree and count height; bst ⇒ pbst 93 ⟩
      ⟨ Set parents of main vine 681 ⟩
}
```

This code is included in §679.

§681 ⟨ Set parents of main vine 681 ⟩ ≡

```
for (q = NULL, p = tree→pbst_root; p != NULL; q = p, p = p→pbst_link[0])
      p→pbst_parent = q;
```

This code is included in §680.

§682 ⟨ PBST compression function 682 ⟩ ≡

```
static void compress (struct pbst_node *root, unsigned long count) {
    assert (root != NULL);
    while (count--) {
        struct pbst_node *red = root->pbst_link[0];
        struct pbst_node *black = red->pbst_link[0];
        root->pbst_link[0] = black;
        red->pbst_link[0] = black->pbst_link[1];
        black->pbst_link[1] = red;
        red->pbst_parent = black;
        if (red->pbst_link[0] != NULL)
            red->pbst_link[0]->pbst_parent = red;
        root = black;
    }
}
```

This code is included in §680.

Chapter 14

Section 14.2

1.

```
/* Rotates right at *yp. */
static void rotate_right (struct pbst_node **yp) {
    struct pbst_node *y = *yp;
    struct pbst_node *x = y->pbst_link[0];
    y->pbst_link[0] = x->pbst_link[1];
    x->pbst_link[1] = y;
    *yp = x;
    x->pbst_parent = y->pbst_parent;
    y->pbst_parent = x;
    if (y->pbst_link[0] != NULL)
        y->pbst_link[0]->pbst_parent = y;
}
/* Rotates left at *xp. */
static void rotate_left (struct pbst_node **xp) {
    struct pbst_node *x = *xp;
    struct pbst_node *y = x->pbst_link[1];
    x->pbst_link[1] = y->pbst_link[0];
    y->pbst_link[0] = x;
    *xp = y;
    y->pbst_parent = x->pbst_parent;
    x->pbst_parent = y;
    if (x->pbst_link[1] != NULL)
        x->pbst_link[1]->pbst_parent = x;
}
```

Section 14.4.2

1. Yes. Both code segments update the nodes along the direct path from y down to n, including node y but not node n. The plain AVL code excluded node n by updating nodes as it moved down to them and making arrival at node n the loop's termination condition. The PAVL code excludes node n by starting at it but updating the parent of each visited node instead of the node itself.

There still could be a problem at the edge case where no nodes' balance factors were to be updated, but there is no such case. There is always at least one balance factor to update, because every inserted node has a parent whose balance factor is affected by its insertion. The one exception would be the first node inserted into an empty tree, but that was already handled as a special case.

2. Sure. There is no parallel to Exercise 5.4.4-4 because q is never the pseudo-root.

Appendix E Catalogue of Algorithms

This appendix lists all of the algorithms described and implemented in this book, along with page number references. Each algorithm is listed under the least-specific type of tree to which it applies, which is not always the same as the place where it is introduced. For instance, rotations on threaded trees can be used in any threaded tree, so they appear under "Threaded Binary Search Tree Algorithms", despite their formal introduction later within the threaded AVL tree chapter.

Sometimes multiple algorithms for accomplishing the same task are listed. In this case, the different algorithms are qualified by a few descriptive words. For the algorithm used in Libavl, the description is enclosed by parentheses, and the description of each alternative algorithm is set off by a comma.

Binary Search Tree Algorithms

AVL Tree Algorithms

Red-Black Tree Algorithms

Threaded Binary Search Tree Algorithms

Threaded AVL Tree Algorithms

Threaded Red-Black Tree Algorithms

Right-Threaded Binary Search Tree Algorithms

Right-Threaded AVL Tree Algorithms

Right-Threaded Red-Black Tree Algorithms

Binary Search Tree with Parent Pointers Algorithms

AVL Tree with Parent Pointers Algorithms

Red-Black Tree with Parent Pointers Algorithms

Appendix F Index

A

B

C

D

E

F

G

H

I

L

R

S

T

U

V

W

X