

10.5.4 Hashing with Chains

The Method

We may eliminate the overflow problem altogether if each position of the hash table is able to hold an unlimited number of pairs. One way to accomplish this is to place a linear list in each position of the hash table. Now each pair may be stored in its home-bucket linear list. We consider the case when each bucket of the hash table is a sorted chain. Figure 10.3 shows an example of such a hash table. As in our earlier example, the hash function divisor is 11.

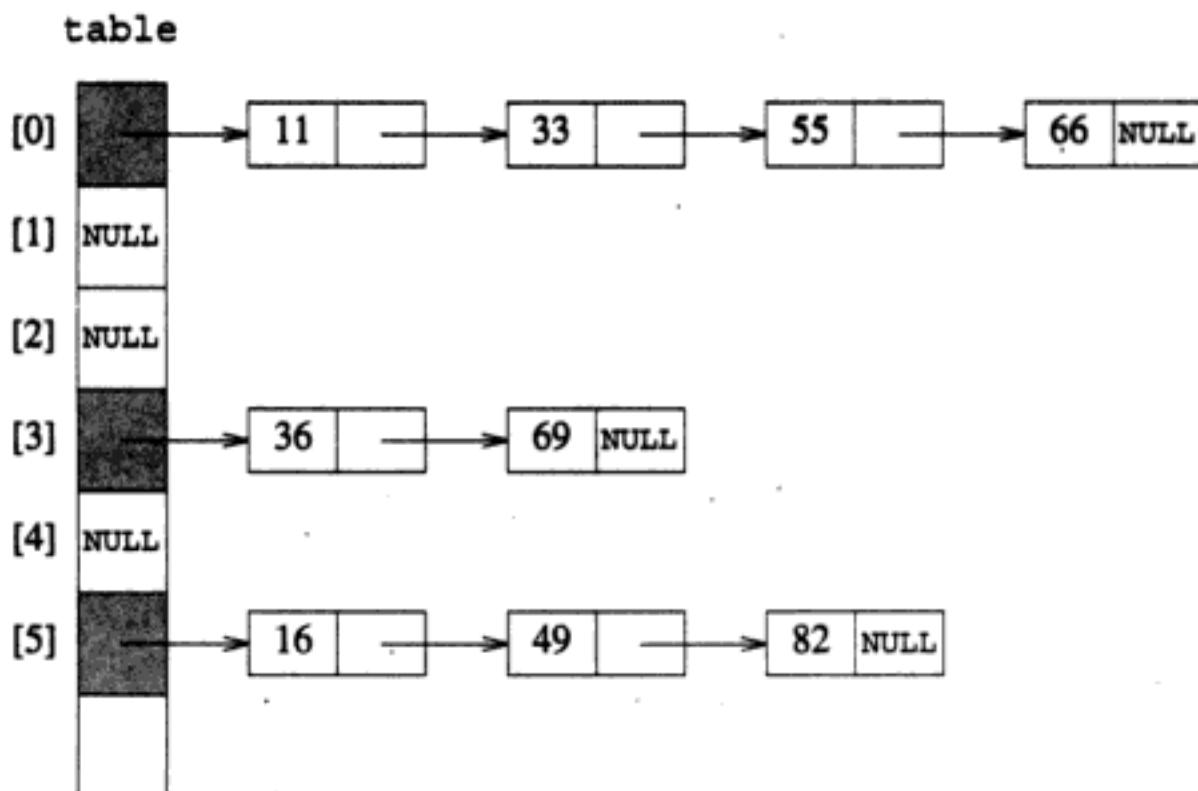


Figure 10.3 A chained hash table

To search for a pair with key k , we first compute the home bucket, $k \% D$, for the key and then search the chain in this bucket. To insert a pair, we need to first verify that the table does not already have a pair with the same key. This search can, of course, be limited to the chain in the home bucket of the new pair. Finally, to erase the pair with key k , we access the home bucket chain, search this chain for the pair with the given key, and then erase the pair.

Hidden page

Hidden page

$$\frac{1}{i+1}(i + \sum_{j=1}^i j) = \frac{1}{i+1}\left(i + \frac{i(i+1)}{2}\right) = \frac{i(i+3)}{2(i+1)}$$

when $i \geq 1$. When $i = 0$, the average number of nodes examined is 0. For chained hash tables we expect the length of a chain to be $n/b = \alpha$ on average. When $\alpha \geq 1$, we may substitute α for i in the above expression to get

$$U_n \approx \frac{\alpha(\alpha+3)}{2(\alpha+1)}, \quad \alpha \geq 1 \quad (10.7)$$

When $\alpha < 1$, $U_n \leq \alpha$ because the average chain length is α and no search requires us to examine more nodes than on a chain.

For S_n we need to know the expected distance of each of the n identifiers from the head of its chain. To determine this distance, assume that the identifiers are inserted in increasing order. This assumption does not affect the positioning of identifiers on their respective chains. When the i th identifier is inserted, its chain is expected to have a length of $(i-1)/b$. The i th identifier gets added to the end of the chain as identifiers are inserted in increasing order. Hence a search for this identifier will require us to examine $1 + (i-1)/b$ nodes. Note also that when identifiers are inserted in increasing order, their distance from the chain head does not change as a result of further insertions. Assuming that each of the n identifiers is searched for with equal probability, we get

$$S_n = \frac{1}{n} \sum_{i=1}^n \{1 + (i-1)/b\} = 1 + \frac{n-1}{2b} \approx 1 + \frac{\alpha}{2} \quad (10.8)$$

Comparing the formulas for chaining with those for linear and random probing, we see that, on average, the number of nodes examined when chaining is used is smaller than the number of buckets examined when linear and random probing are used. For instance, when $\alpha = 0.9$, an unsuccessful search in a chained hash table is expected to examine 0.9 node and a successful search, 1.45 nodes. On the other hand, when linear probing is used, 50.5 buckets are expected to be examined if the search is unsuccessful and 5.5 if it is successful!

Comparison with Skip Lists

Both skip lists and hashing utilize a randomization process to improve the expected performance of the dictionary operations. In the case of skip lists, randomization is used to assign a level to a pair at the time of insertion. This level assignment is done without examining the key of the pair being inserted. In the case of hashing, the hash function assigns a bucket so as to randomly distribute the bucket assignments

Hidden page

Hidden page

23. Determine a suitable value for the hash function divisor D when linear probing is used. Do this for each of the following situations:
 - (a) $n = 50, S_n \leq 3, U_n \leq 20.$
 - (b) $n = 500, S_n \leq 5, U_n \leq 60.$
 - (c) $n = 10, S_n \leq 2, U_n \leq 10.$
24. For each of the following conditions, obtain a suitable value for the hash function divisor D . For this value of D determine S_n and U_n as a function of n . Assume that linear probing is used.
 - (a) $\text{MaxElements} \leq 530.$
 - (b) $\text{MaxElements} \leq 130.$
 - (c) $\text{MaxElements} \leq 150.$
25. Write a version of the method `hashTable<K,E>::insert` (Program 10.19) in which the table size is approximately doubled whenever the loading density exceeds a user-specified amount. This loading density is specified along with the initial capacity when the hash table is constructed. Specifically, we limit ourselves to odd table sizes (and so to odd divisors); and each time the loading density is exceeded, the new table size is $2 * (\text{oldtablesize}) + 1$. Although this method does not pick divisors according to the rule we have specified, this approach avoids even divisors and is easy to implement.
26. Write code for the method `hashTable<K,E>::erase`. Do not change any existing members of `hashTable`. What is the worst-case time complexity of your code to erase a pair? Use suitable data to test its correctness.
27. Develop a class for hash tables using linear probing and the `neverUsed` concept to handle an erase operation. Write complete C++ code for all methods. Include a method to reorganize the table when (say) 60 percent of the empty buckets have `neverUsed` equal to `false`. The reorganization should move pairs around as necessary and leave a properly configured hash table in which `neverUsed` is `true` for every empty bucket. Test the correctness of your code.
28.
 - (a) Implement a hash table class that uses quadratic probing instead of linear probing. You need not implement a method to erase pairs. See the Web site for a description of quadratic probing.
 - (b) Compare the performance of your class with that of the class `hashTable` that was developed in Section 10.5.3. Do this comparison by experimentally measuring the average number of key comparisons made in successful and unsuccessful searches as well as by measuring the actual run time.
29. Do Exercise 28 using double hashing rather than quadratic probing. See the Web site for a description of double hashing.

30. Comment on the difficulty of providing sequential access when (a) linear probing is used and (b) when a chained hash table is used.
31. Do Exercise 17 for a chained hash table.
32. Do Exercise 20 for a chained hash table.
33. Develop a new class `sortedChainWithTail` in which the sorted chain has a tail node. Use the tail node to simplify your code by placing the pair or key being searched for, inserted, or deleted into the tail at the start of the operation. Compare the run-time performance of sorted chains with and without tail nodes.
34. Develop the class `chainedHashTable` that implements all methods of `dictionary`. The class should be developed from scratch and should make insertions and removals from its chains without invoking any method of any chain class. Test your code.
35. Develop a chained hash table class `hashChainsWithTails` in which the chains are instances of the class `sortedChainWithTail` (see Exercise 33). Compare the run-time performance of hash tables that are instances of `hashChains` and those that are instances of `hashChainsWithTails`.
36. Develop a class `hashChainsWithTail` in which each hash table chain is a sorted chain with a tail node. The tail node for all chains is the same physical node. The class should be developed from scratch and should make insertions and removals from its chains without invoking any method of any chain class. Compare the run-time performance of this class with that of `hashChains` (Program 10.20).
37. In an effort to simplify the insert and erase codes for chained hash tables, we might consider adding a header node to each chain. The header node is in addition to a tail node as discussed in the text. All insertions and removals now take place between the header and tail of a chain. As a result, the case of insertion and erase at/from the front of a chain is eliminated.
 - (a) Is it possible to use the same header node for all chains? Why?
 - (b) Is it desirable to set the key field(s) of the header node(s) to a particular value? Why?
 - (c) Develop and test the class `hashChainsWithHeadersAndTail` in which each chain has a header node and a tail node. The class should be developed from scratch and should make insertions and removals from its chains without invoking any method of any chain class. Include all methods included in `hashChains`.

Hidden page

Hidden page

code	0	1
key	a	b

0	1	2
a	b	aa

a a a b b b b b b a a b a a b a

Compressed string = null
(a) Initial configuration

a a a b b b b b b a a b a a b a

Compressed string = 0
(b) a has been compressed

0	1	2	3
a	b	aa	aab

0	1	2	3	4
a	b	aa	aab	bb

a a a b b b b b b a a b a a b a

Compressed string = 02
(c) aaa has been compressed

a a a b b b b b b a a b a a b a

Compressed string = 021
(d) aaab has been compressed

0	1	2	3	4	5
a	b	aa	aab	bb	bbb

0	1	2	3	4	5	6
a	b	aa	aab	bb	bbb	bbba

a a a b b b b b b a a b a a b a

Compressed string = 0214
(e) aaabb has been compressed

a a a b b b b b b a a b a a b a

Compressed string = 02145
(f) aaabb has been compressed

0	1	2	3	4	5	6	7
a	b	aa	aab	bb	bbb	bbba	aaba

a a a b b b b b a a b a a b a

Compressed string = 021453
(g) aaabb has been compressed

Figure 10.5 LZW compression

Hidden page

```
void setFiles(int argc, char* argv[])
{// Create input and output streams.
    char outputFile[50], inputFile[54];
    // see if file name provided
    if (argc >= 2)
        strcpy(inputFile, argv[1]);
    else
        {// name not provided, ask for it
            cout << "Enter name of file to compress" << endl;
            cin >> inputFile;
        }

    // open files in binary mode
    in.open(inputFile, ios::binary);
    if (in.fail())
    {
        cerr << "Cannot open " << inputFile << endl;
        exit(1);
    }
    strcpy(outputFile, inputFile);
    strcat(outputFile, ".zzz");
    out.open(outputFile, ios::binary);
}
```

Program 10.21 Establish input and output streams

To simplify decoding the compressed file, we will write each code using a fixed number of bits. In further development we will assume that each code is 12 bits long. Hence we can assign at most $2^{12} = 4096$ codes. Under this assumption, the encoding 0214537 for our 16-character sample string S is written out as $12 * 7 = 84$ bits, which rounds to 11 bytes.

Since each character is 8 bits long (we are assuming each character is one of the 256 ASCII characters), a key is 20 bits long (12 bits for the code of the prefix and 8 bits for the last character in the key) and can be represented using a long integer (32 bits). The least significant 8 bits are used for the last character in the key, and the next 12 bits for the code of its prefix. The dictionary itself may be represented as a chained hash table. If the prime number DIVISOR = 4099 is used as the hash function divisor, the loading density will be less than 1, as we can have at most 4096 pairs in the dictionary. The declaration

```
hashChains<long, int> h(DIVISOR);
```

suffices to create the table.

Hidden page

we have used all 4096 codes).

```

void compress()
{// Lempel-Ziv-Welch compressor.
    // define and initialize the code dictionary
    hashChains<long, int> h(DIVISOR);
    for (int i = 0; i < ALPHA; i++)
        h.insert(pairType(i, i));
    int codesUsed = ALPHA;

    // input and compress
    int c = in.get(); // first character of input file
    if (c != EOF)
    {// input file is not empty
        long pcode = c; // prefix code
        while ((c = in.get()) != EOF)
        {// process character c
            long theKey = (pcode << BYTE_SIZE) + c;
            // see if code for theKey is in the dictionary
            pairType* thePair = h.find(theKey);
            if (thePair == NULL)
            {// theKey is not in the table
                output(pcode);
                if (codesUsed < MAX_CODES) // create new code
                    h.insert(pairType((pcode << BYTE_SIZE) | c, codesUsed++));
                pcode = c;
            }
            else pcode = thePair->second; // theKey is in table
        }

        // output last code(s)
        output(pcode);
        if (bitsLeftOver)
            out.put(leftOver << EXCESS);
    }

    out.close();
    in.close();
}

```

Program 10.23 LZW compressor

Constants, Global Variables and the Function main

Program 10.24 gives the constants and global variables as well as the function `main`.

```
// constants
const DIVISOR = 4099,           // hash function DIVISOR
      MAX_CODES = 4096,          // 2^12
      BYTE_SIZE = 8,
      EXCESS = 4,                // 12 - BYTE_SIZE
      ALPHA = 256,               // 2^BYTE_SIZE
      MASK1 = 255,               // ALPHA - 1
      MASK2 = 15;                // 2^EXCESS - 1

typedef pair<const long, int> pairType ;
    // pair.first = key, pair.second = code

// globals
int leftOver;                  // code bits yet to be output
bool bitsLeftOver = false;      // false means no bits in leftOver
ifstream in;                    // input file
ofstream out;                  // output file

void main(int argc, char* argv[])
{
    setFiles(argc, argv);
    compress();
}
```

Program 10.24 Constants, global variables, and `main`

10.6.3 LZW Decompression

For decompression we input the codes one at a time and replace them by the texts they denote. The code-to-text mapping can be reconstructed in the following way. The codes assigned for single-character texts are entered into the dictionary. As before, the dictionary entries are code-text pairs. This time, however, the dictionary is searched for an entry with a given code (rather than with a given text). Therefore the dictionary pairs (*key, value*) are constructed so that *key* is a code and *value* is the text denoted by the code. The first code in the compressed file corresponds to a single character and so may be replaced by the corresponding character. For all other codes *p* in the compressed file, we have two cases to consider: (1) the code *p* is in the dictionary, and (2) it is not.

Case When Code p Is in the Dictionary

When p is in the dictionary, the text $\text{text}(p)$ to which it corresponds is extracted from the dictionary and output. Also, from the working of the compressor, we know that if the code that precedes p in the compressed file is q and $\text{text}(q)$ is the corresponding text, then the compressor would have created a new code for the text $\text{text}(q)$ followed by the first character $\text{fc}(p)$ of $\text{text}(p)$. So we enter the pair (next code, $\text{text}(q)\text{fc}(p)$) into the directory.

Case When Code p Is Not in the Dictionary

This case arises only when the current text segment has the form $\text{text}(q)\text{text}(q)\text{fc}(q)$ and $\text{text}(p) = \text{text}(q)\text{fc}(q)$. The corresponding compressed file segment is qp . During compression, $\text{text}(q)\text{fc}(q)$ is assigned the code p , and the code p is output for the text $\text{text}(q)\text{fc}(q)$. During decompression, after q is replaced by $\text{text}(q)$, we encounter the code p . However, there is no code-to-text mapping for p in our table. We are able to decode p by knowing that this situation arises only when the decompressed text segment is $\text{text}(q)\text{text}(q)\text{fc}(q)$. So when we encounter a code p for which the code-to-text mapping is undefined, the code-to-text mapping for p is $\text{text}(q)\text{fc}(q)$ where q is the code that precedes p .

An Example

Let us try this decompression scheme on our earlier sample string

aaabbbbbbaabaaba

which was compressed into the coded string 0214537. To begin, we initialize the dictionary with the pairs (0, a) and (1, b) and obtain the first two entries in the dictionary of Figure 10.5. The first code in the compressed file is 0. It is replaced by the text a. The next code 2 is undefined. Since the previous code, 0, has $\text{text}(0) = a$, $\text{fc}(0) = a$ and $\text{text}(2) = \text{text}(0)\text{fc}(0) = aa$. So the code 2 is replaced by aa, and (2, aa) is entered into the dictionary. The next code, 1, is replaced by $\text{text}(1) = b$, and (3, $\text{text}(2)\text{fc}(1)$) = (3, aab) is entered into the dictionary. The next code, 4, is not in the dictionary. The code preceding it is 1, and so $\text{text}(4) = \text{text}(1)\text{fc}(1) = bb$. The pair (4, bb) is entered into the dictionary, and bb is output to the decompressed file. When the next code, 5, is encountered, (5, bbb) is entered into the directory; bbb is output to the decompressed file. The next code is 3. $\text{text}(3) = aab$ is output to the decompressed file, and the pair (6, $\text{text}(5)\text{fc}(3)$) = (6, bbba) is entered into the dictionary. Finally, when the code 7 is encountered, (7, $\text{text}(3)\text{fc}(3)$) = (7, aaba) is entered into the dictionary and aaba output.

10.6.4 Implementation of LZW Decompression

As was the case for the implementation of the compression algorithm, the decompression task is decomposed into several subtasks, and each is implemented by a different function. Since the decompression function `setFiles`, which establishes the input and output streams, is very similar to the corresponding function for compression, we do not discuss this function further.

Dictionary Organization

Because we will be querying the dictionary by providing a code and since the number of codes is 4096, we can use an array `ht[4096]` and store `text(p)` in `ht[p]`. Using array `ht` in this way corresponds to ideal hashing with $f(k) = k$. `text(p)` may be compactly stored by using the code for the prefix of `text(p)` and the last character (suffix) of `text(p)` as in Figure 10.6. For our decompression application it is convenient to store the prefix code and suffix using the two fields of a pair. The prefix code is stored in first field of a pair and the suffix is stored in the second field. The declaration

```
typedef pair<int, char> pairType;
pairType ht[MAX_CODES];
```

may be used to define our dictionary. So if $text(p) = text(q)c$, then `ht[p].second` equals the character `c` and `ht[p].first` equals `q`.

When this dictionary organization is used, `text(p)` may be constructed from right to left beginning with the last character `ht[p].second`, as is shown in Program 10.25. This code obtains suffix values of codes $\geq \text{ALPHA}$ from the table `ht`, and for codes $< \text{ALPHA}$ it uses the knowledge that the code is just the integer representation of the corresponding character. `text(p)` is assembled into the array `s[]` and then output. Since `text(p)` is assembled from right to left, the first character of `text(p)` is in `s[size]`.

Input of Codes

Since the sequence of 12-bit codes is represented as a sequence of 8-bit bytes in the compressed file, we need to reverse the process employed by the `output` function of the LZW compressor (Program 10.22). This reversal is done by the function `getCode` (Program 10.26). The only new constant here is `MASK`. Its value, 15, enables us to extract the low-order 4 bits of a byte.

Decompression

Program 10.27 gives the LZW decompressor. The first code in the compressed file is decoded outside the `while` loop by doing a type conversion to the type `char`, and the remaining codes are decoded inside this loop. *At the start of each iteration of*

Hidden page

the while loop, `s[size]` contains the first character of the last decoded text that was output. To establish this condition for the first iteration, we set `size` to 0 and `s[0]` to the first and only character corresponding to the first code in the compressed file.

The while loop repeatedly obtains a code `ccode` from the compressed file and decodes it. There are two cases for `ccode`—(1) `ccode` is in the dictionary, and (2) it is not. `ccode` is in the dictionary iff `ccode < codesUsed` where `ht[0:codesUsed-1]` is the defined part of table `h`. In this case the code is decoded using the decompression function `output`, and following the LZW rule, a new code is created with suffix being the first character of the text just output for `ccode`. When `ccode` is not defined, we are in the special case discussed at the beginning of this section and `ccode` is `text(pcode)s[size]`. This information is used to create a table entry for `code` and to output the decoded text to which it corresponds.

Constants, Global Variables, and the Function main

Program 10.28 gives the constants, global variables, and `main` function for LZW decompression.

10.6.5 Performance Evaluation

Well, how good is our compressor compared to, say, the popular compression program `zip`? Our program compressed a 33,772-byte ASCII file to 18,765 bytes, achieving a compression ratio of $33,772/18,765 = 1.8$; `zip` did much better—it compressed the same file to 11,041 bytes, achieving a compression ratio of 3.1. This disappointing showing by our compression program should not be a cause for concern, since commercial compression programs such as `zip` couple methods such as LZW compression with good coding techniques such as Huffman coding (see Section 12.6.3) to obtain a higher compression ratio. So we should not expect a raw LZW compressor to match the performance of a commercial compressor.

EXERCISES

42. Start with an LZW compression dictionary that has the entries (a, 0) and (b, 1).
 - (a) Draw figures similar to Figure 10.5 for the LZW compression dictionary following the processing of each character of the string babababbabba.
 - (b) Give the code sequence for the compressed form of babababbabba.
 - (c) Now decompress the code sequence of part (b). For each code encountered, explain how it is decoded. Draw the decode table following the decoding of each code.

Hidden page

Hidden page

Use 8 bits per code. Test the correctness of your program. Is it possible for the compressed file to be longer than the original file?

47. Modify the LZW compress and decompress programs so that the code table is reinitialized after every x kilobytes of the text file have been compressed / decompressed. Experiment with the modified compression code using text files that are 100K to 200K bytes long and $x = 10, 20, 30, 40$, and 50. Which value of x gives the best compression?
48. A concordance is an alphabetized list of all the words in a text. Together with each word is a sorted list of all the lines of the text that contain this word. That is, each entry of a concordance is a pair of the form (word, sorted list of all line numbers where this word occurs). Notice that a concordance is similar to a book index; however, unlike a book index, which lists the page numbers on which only some of the words in the book occur, a concordance includes every word and lists line numbers rather than page numbers. The objective of this exercise is to write a program that uses a hash table to create a concordance. Each hash table entry is a pair (`key, list`) = (word, sorted list of line numbers where this word occurs).
 - (a) Develop a C++ class for the hash table pairs. Justify your selection of data types for `key` and `list`. When selecting a data type for `list`, consider the types `vector`, `arrayList`, `chain`, `arrayQueue`, `linkedQueue`, and any customized type(s) that may be appropriate.
 - (b) Develop two C++ programs to input text and output its concordance. The first program should use the STL class `hash_map` to construct the concordance entries and then use a suitable sort method to sort these entries by their `key` fields. The second program should use the class `hashChains` to construct the concordance entries and then should merge the hash table chains to obtain a sorted list of concordance entries.
 - (c) Compare the run-time performance of the two programs you have developed.

10.7 REFERENCES AND SELECTED READINGS

Skip lists were proposed by William Pugh. An analysis of their expected complexity can be found in the paper "Skip Lists: A Probabilistic Alternative to Balanced Trees" by W. Pugh, *Communications of the ACM*, 33, 6, 1990, 668–676.

Visit this book's Web site to learn more about hash functions and overflow-handling mechanisms. To find out everything you ever wanted to know about hashing, see the book *The Art of Computer Programming: Sorting and Searching*, Volume 3, Second Edition, by D. Knuth, Addison-Wesley, Menlo Park, CA, 1998.

Our description of the Lempel-Ziv-Welch compression method is based on the paper "A Technique for High-Performance Data Compression" by T. Welch, *IEEE Computer*, June 1994, 8–19. For more on data compression, see the survey article "Data Compression" by D. Lelewer and D. Hirschberg, *ACM Computing Surveys*, 19, 3, 1987, 261–296.

CHAPTER 11

BINARY AND OTHER TREES

BIRD'S-EYE VIEW

Yes, it's a jungle out there. The jungle is populated with many varieties of trees, plants, and animals. The world of data structures also has a wide variety of trees, too many for us to discuss in this book. In the present chapter we study two basic varieties: general trees (or simply trees) and binary trees. Chapters 12 through 15 consider the more popular of the remaining varieties—heaps, leftist trees, tournament trees, binary search trees, AVL trees, red-black trees, splay trees, and B-trees. Chapters 12 through 14 are fairly independent and may be read in any order. However, Chapter 15 should be read only after you have assimilated Chapter 14. If you are still hungry for trees when you are done with these chapters, you can find additional tree varieties—pairing heaps, interval heaps, tree structures for double-ended priority queues, tries, and suffix trees—on the Web site for this book.

Two applications of trees are developed in the applications section. The first concerns the placement of signal boosters in a tree distribution network. The second revisits the online equivalence problem introduced in Section 6.5.4. This problem is also known as the union/find problem. By using trees to represent the sets, we can obtain improved run-time performance over the chain representation developed in Section 6.5.4.

In addition, this chapter covers the following topics:

- Tree and binary tree terminology such as height, depth, level, root, leaf, child, parent, and sibling.
- Array and linked representations of binary trees.
- The four common ways to traverse a binary tree: preorder, inorder, postorder, and level order.

11.1 TREES

So far in this text we have seen data structures for linear and tabular data. These data structures are generally not suitable for the representation of hierarchical data. In hierarchical data we have an ancestor-descendant, superior-subordinate, whole-part, or similar relationship among the data elements.

Example 11.1 [Joe's Descendants] Figure 11.1 shows Joe and his descendants arranged in a hierarchical manner, beginning with Joe at the top of the hierarchy. Joe's children (Ann, Mary, and John) are listed next in the hierarchy, and a line or edge joins Joe and his children. Ann has no children, while Mary has two and John has one. Mary's children are listed below her, and John's child is listed below him. There is an edge between each parent and his/her children. From this hierarchical representation, it is easy to identify Ann's siblings, Joe's descendants, Chris's ancestors, and so on. ■

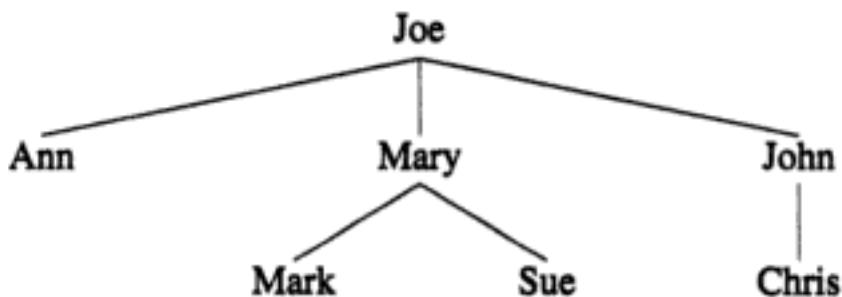


Figure 11.1 Descendants of Joe

Example 11.2 [Corporate Structure] As an example of hierarchical data, consider the administrative structure of the corporation of Figure 11.2. The person (in this case the president) highest in the hierarchy appears at the top of the diagram. Those who are next in the hierarchy (i.e., the vice presidents) are shown below the president and so on. The vice presidents are the president's subordinates, and the president is their superior. Each vice president, in turn, has his/her subordinates who may themselves have subordinates. In the diagram we have drawn a line or edge between each person and his/her direct subordinates or superior. ■

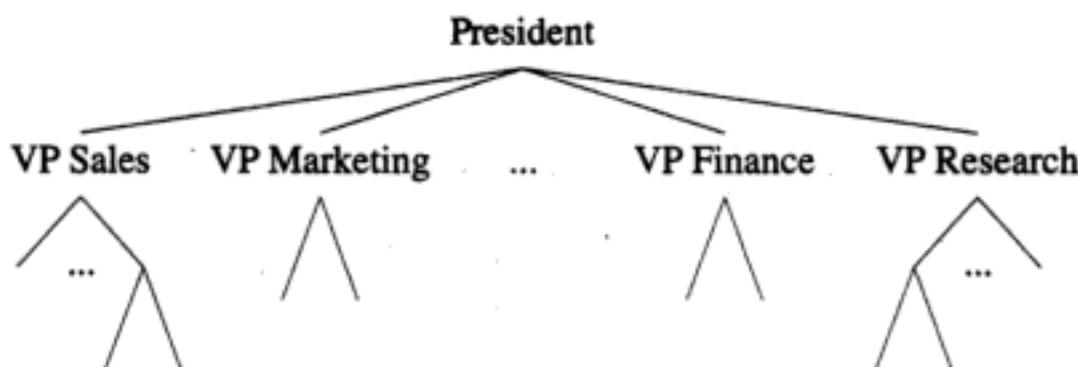


Figure 11.2 Hierarchical administrative structure of a corporation

Example 11.3 [Governmental Subdivisions] Figure 11.3 is a hierarchical drawing of the branches of the federal government. At the top of the hierarchy, we have the entire federal government. At the next level of the hierarchy, we have drawn its major subdivisions (i.e., the different departments). Each department may be further subdivided. These subdivisions are drawn at the next level of the hierarchy. For example, the Department of Defense has been subdivided into the Army, Navy, Air Force, and Marines. A line runs between each element and its components. The data of Figure 11.3 are an example of whole-part relationships. ■

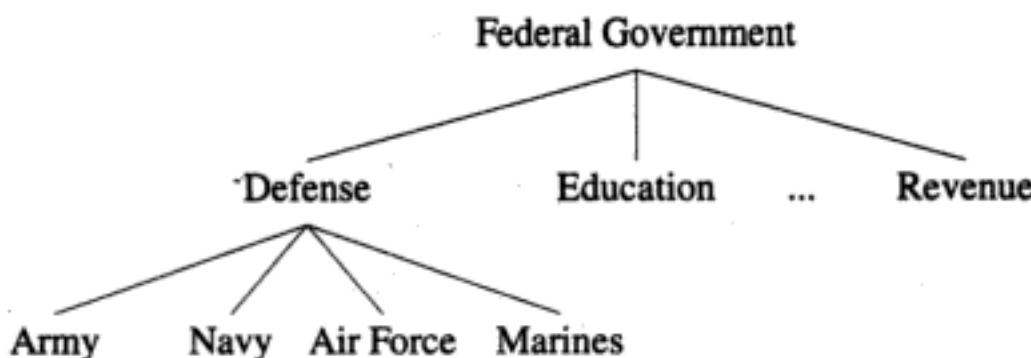


Figure 11.3 Modules of the federal government

Example 11.4 [Software Engineering] For another example of hierarchical data, consider the software-engineering technique referred to as modularization. In modularization we decompose a large and complex task into a collection of smaller, less complex tasks. The objective is to divide the software system into many functionally independent parts or **modules** so that each can be developed relatively independently. This decision reduces the overall software development time, as it is much easier to solve several small problems than one large one. Additionally, different

programmers can develop different modules at the same time. If necessary, each module may be further decomposed so as to obtain a hierarchy of modules as shown by the tree of Figure 11.4. This tree represents a possible modular decomposition of a text processor.

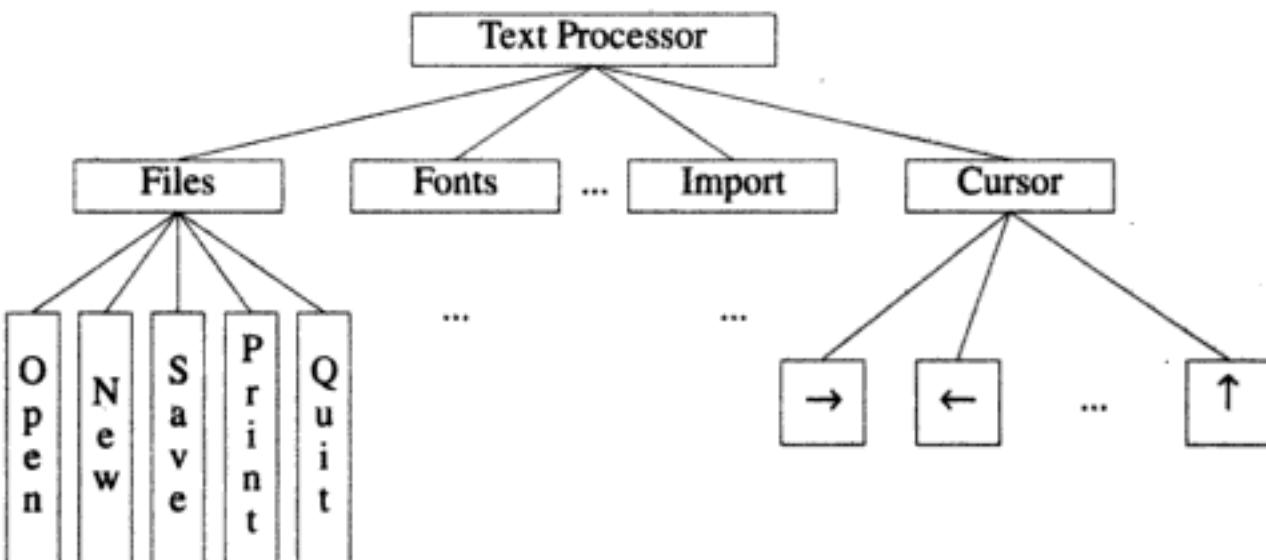


Figure 11.4 Module hierarchy for text processor

At the top level the text processor has been split into several modules. Only four are shown in the figure. The Files module performs functions related to text files such as opening an existing file, opening a new file, saving a file, printing a file, and exiting from the text processor (exiting requires saving files if the user so desires). Each function is represented by a module at the next level of the hierarchy. The Fonts module handles all functions related to the font in use. These functions include changing the font, its size, color, and so on. If modules for these functions were shown in the figure, they would appear below the module Fonts. The Import module handles functions associated with the import of material such as graphics, tables, and text in a format not native to this text processor. The Cursor module handles the movement of the cursor on the screen. Its subordinate modules correspond to various cursor motions. Programmers can carry out the specification, design, and development of each module in a relatively independent manner after the interfaces are fully specified.

When the software system is specified and designed in a modular fashion, it is natural to develop the system itself in this way. The resulting software system will have as many modules as there are nodes in the module hierarchy. Modularization improves the intellectual manageability of a problem. By systematically dividing a large problem into smaller relatively independent problems, we can solve the large problem with much less effort. The independent problems can be assigned to

Hidden page

of the department subtree. The remaining employees of a department could be partitioned into projects and so on.

The vice presidents are children of the president; department heads are children of their vice president, and so on. The president is the parent of the vice presidents, and each vice president is the parent of the department heads in his/her division.

In Figure 11.3 the root is the element Federal Government. Its subtrees have the roots Defense, Education, ..., and Revenue, which are the children of Federal Government. Federal Government is the parent of its children. Defense has the children Army, Navy, Air Force, and Marines. The children of Defense are siblings and are also leaves. ■

Another commonly used tree term is **level**. By definition the tree root is at level 1; its children (if any) are at level 2; their children (if any) are at level 3; and so on.¹ In the tree of Figure 11.3, Federal Government is at level 1; Defense, Education, and Revenue are at level 2; and Army, Navy, Air Force, and Marines are at level 3.

The **height** (or **depth**) of a tree is the number of levels in it. The trees of Figures 11.1, 11.3, and 11.4 have a height of 3.

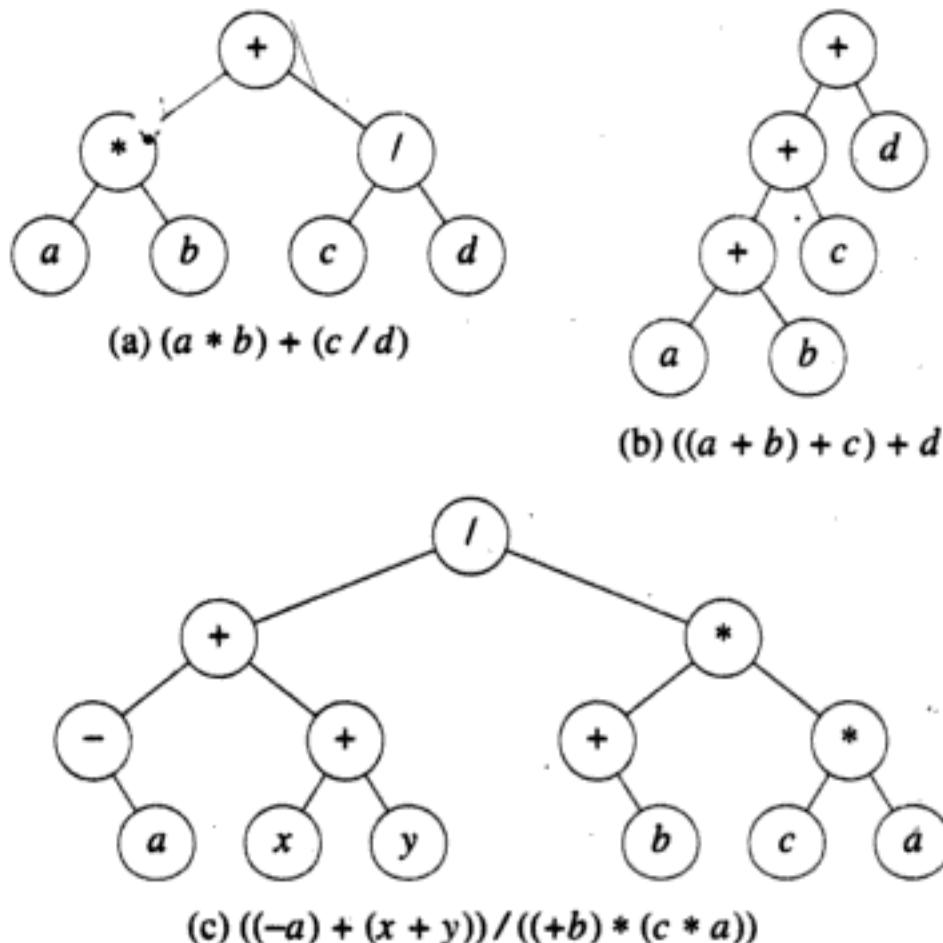
The **degree of an element** is the number of children it has. The degree of a leaf is 0. The degree of Files in Figure 11.4 is 5. The **degree of a tree** is the maximum of its element degrees.

EXERCISES

1. Explain why the diagram of Figure 11.1 is a tree. Label the root node and mark each node with its level number and degree. What is the depth of this tree?
2. Do Exercise 1 for the recursion diagram of Figure 1.3.
3. Develop a tree representation for the major elements (whole book, chapters, sections, and subsections) of this text.
 - (a) What is the total number of elements in your tree?
 - (b) Identify the leaf elements.
 - (c) Identify the elements on level 3.
 - (d) List the degree of each element.
4. Access the World Wide Web home page for your department. (Alternatively, access <http://www.cise.ufl.edu>.)
 - (a) Follow some of the links to lower-level pages and draw the resulting structure. In your drawing the Web pages should be represented by nodes, and the links by edges that join pairs of nodes.

¹Some authors number tree levels beginning at 0 rather than 1. In this case the root of a tree is at level 0.

Hidden page

**Figure 11.5** Expression trees

6. Draw the binary expression trees corresponding to each of the following expressions:

- $(a + b)/(c - d) + e + g * h/a$
- $-x - y * z + (a + b + c/d * e)$
- $((a + b) > (c - e)) \mid\mid a < b \&\& (x < y \mid\mid y > z)$

11.3 PROPERTIES OF BINARY TREES

Property 11.1 *The drawing of every binary tree with n elements, $n > 0$, has exactly $n - 1$ edges.*

Proof Every element in a binary tree (except the root) has exactly one parent. There is exactly one edge between each child and its parent. So the number of edges is $n - 1$. ■

Property 11.2 A binary tree of height h , $h \geq 0$, has at least h and at most $2^h - 1$ elements in it.

Proof Since each level has at least one element, the number of elements is at least h . As each element can have at most two children, the number of elements at level i is at most 2^{i-1} , $i > 0$. For $h = 0$, the total number of elements is 0, which equals $2^0 - 1$. For $h > 0$, the number of elements cannot exceed $\sum_{i=1}^h 2^{i-1} = 2^h - 1$. ■

Property 11.3 The height of a binary tree that contains n , $n \geq 0$, elements is at most n and at least $\lceil \log_2(n + 1) \rceil$.

Proof Since there must be at least one element at each level, the height cannot exceed n . From Property 11.2 we know that a binary tree of height h can have no more than $2^h - 1$ elements. So $n \leq 2^h - 1$. Hence $h \geq \log_2(n + 1)$. Since h is an integer, we get $h \geq \lceil \log_2(n + 1) \rceil$. ■

A binary tree of height h that contains exactly $2^h - 1$ elements is called a **full binary tree**. The binary tree of Figure 11.5(a) is a full binary tree of height 3. The binary trees of Figures 11.5(b) and (c) are not full binary trees. Figure 11.6 shows a full binary tree of height 4.

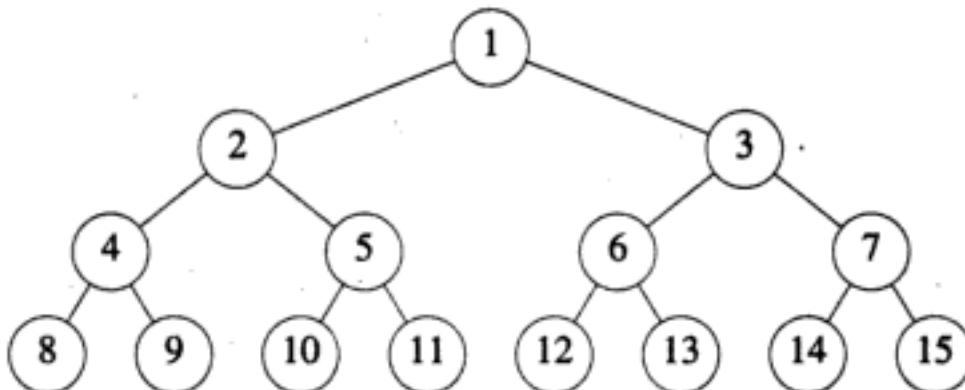
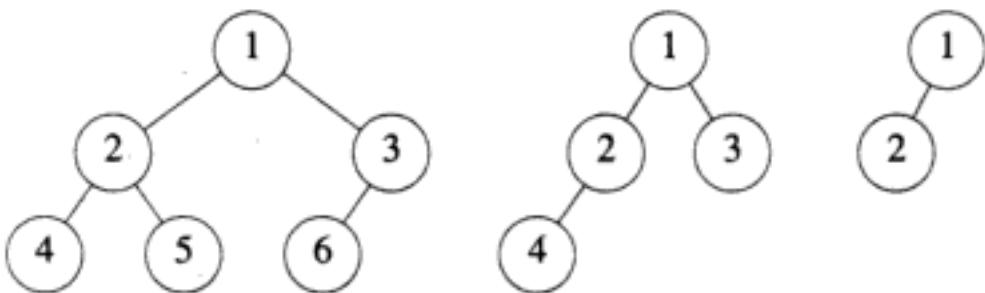


Figure 11.6 Full binary tree of height 4

Suppose we number the elements in a full binary tree of height h using the numbers 1 through $2^h - 1$. We begin at level 1 and go down to level h . Within levels the elements are numbered left to right. The elements of the full binary tree of Figure 11.6 have been numbered in this way. Now suppose we delete the k elements numbered $2^h - i$, $1 \leq i \leq k < 2^h$. The resulting binary tree is called a **complete binary tree**. Figure 11.7 gives some examples. Note that a full binary tree is a special case of a complete binary tree. Also, note that the height of a complete binary tree that contains n elements is $\lceil \log_2(n + 1) \rceil$.

There is a very nice relationship among the numbers assigned to an element and its children in a complete binary tree, as given by Property 11.4.

**Figure 11.7** Complete binary trees

Property 11.4 Let i , $1 \leq i \leq n$, be the number assigned to an element of a complete binary tree. The following are true:

1. If $i = 1$, then this element is the root of the binary tree. If $i > 1$, then the parent of this element has been assigned the number $\lfloor i/2 \rfloor$.
2. If $2i > n$, then this element has no left child. Otherwise, its left child has been assigned the number $2i$.
3. If $2i + 1 > n$, then this element has no right child. Otherwise, its right child has been assigned the number $2i + 1$.

Proof Can be established by induction on i . ■

EXERCISES

7. Prove Property 11.4.
8. In a k -ary tree, $k > 1$, each node may have up to k children. These children are called, respectively, the first, second, ..., k th child of the node. A 2-ary tree is a binary tree.
 - (a) Determine the analogue of Property 11.1 for k -ary trees.
 - (b) Determine the analogue of Property 11.2 for k -ary trees.
 - (c) Determine the analogue of Property 11.3 for k -ary trees.
 - (d) Determine the analogue of Property 11.4 for k -ary trees.
9. What is the maximum number of nodes in a binary tree that has m leaves?

11.4 REPRESENTATION OF BINARY TREES

11.4.1 Array-Based Representation

The array representation of a binary tree utilizes Property 11.4. The binary tree to be represented is regarded as a complete binary tree with some missing elements. Figure 11.8 shows two sample binary trees. The first binary tree has three elements (A, B, and C), and the second has five elements (A, B, C, D, and E). Neither is complete. Unshaded circles represent missing elements. All elements (including the missing ones) are numbered as described in the previous section.

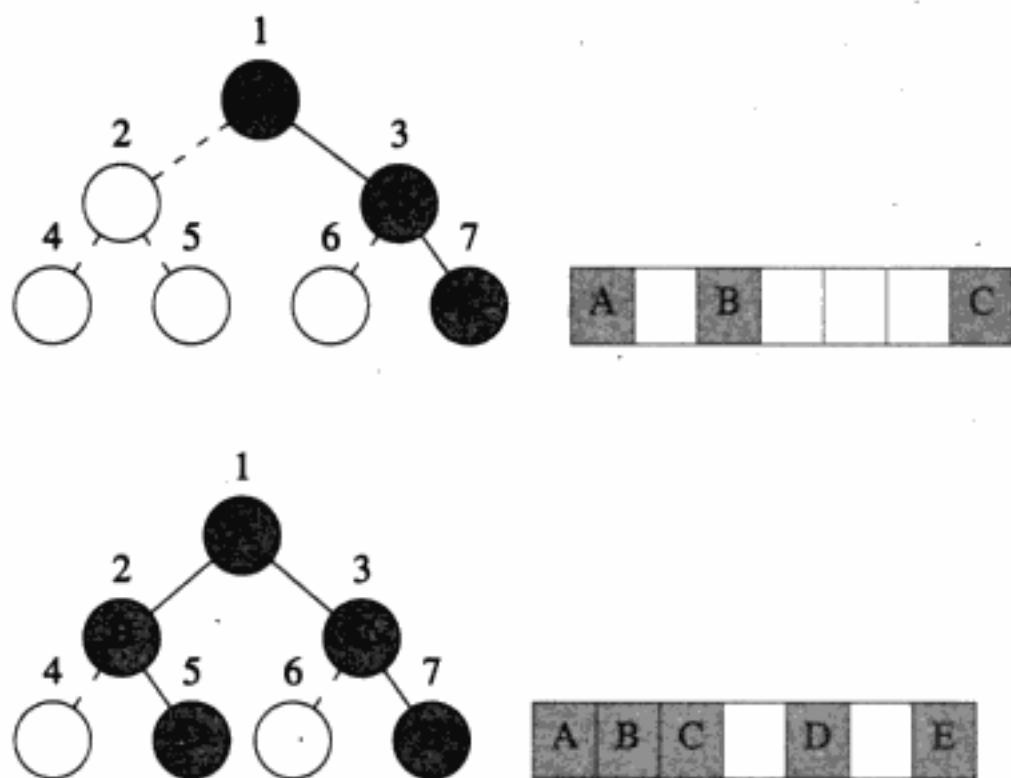
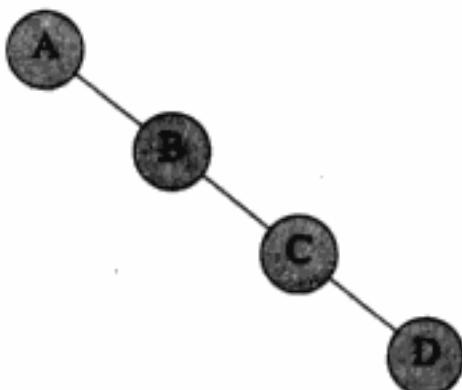


Figure 11.8 Incomplete binary trees

In an array representation, the binary tree is represented in an array by storing each element at the array position corresponding to the number assigned to it. Figure 11.8 also shows the array representations for its binary trees (position 0 of the array is not shown). Missing elements are represented by white boxes. As can be seen, this representation scheme is quite wasteful of space when many elements are missing. In fact, a binary tree that has n elements may require an array of size up to 2^n (including position 0) for its representation. This maximum size is needed when each element (except the root) of the n -element binary tree is the right child of its parent. Figure 11.9 shows such a binary tree with four elements. Binary trees of this type are called **right-skewed** binary trees. Note that the worst-case space

required may be reduced to $2^n - 1$ by numbering the binary tree nodes beginning at 0 rather than at 1.



(a) Right-skewed tree



(b) Array representation

Figure 11.9 Right-skewed binary tree

The array representation is useful only when the number of missing elements is small.

11.4.2 Linked Representation

The most popular way to represent a binary tree is by using links or pointers. A node that has exactly two pointer fields represents each element. Let us call these pointer fields `leftChild` and `rightChild`. In addition to these two pointer fields, each node has a field named `element`. This node structure is implemented by the C++ struct `binaryTreeNode` (Program 11.1). We have provided three constructors for a binary tree node. The first takes no parameters and sets the two children fields to `NULL`; the second takes one parameter and uses it to initialize `element`, and the child fields are set to `NULL`; the third takes three parameters and uses these to initialize all three fields of the node.

Each pointer from a parent node to a child node represents an edge in the drawing of a binary tree. Since an n -element binary tree has exactly $n - 1$ edges, we are left with $2n - (n - 1) = n + 1$ pointer fields that have no value. These pointer fields are set to `NULL`. Figure 11.10 shows the linked representations of the binary trees of Figure 11.8.

We can access all nodes in a binary tree by starting at the root and following `leftChild` and `rightChild` pointers. The absence of a parent pointer from the

Hidden page

Hidden page

Hidden page

The first three traversal methods are best described recursively as in Programs 11.2, 11.3, and 11.4. These codes assume that the binary tree being traversed is represented with the linked scheme of the previous section.

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among the three orders comes from the difference in the time at which a node is visited. In the case of a preorder traversal, each node is visited before its left and right subtrees are traversed. In an inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. In a postorder traversal, each root is visited after its left and right subtrees have been traversed.

Figure 11.11 shows the output generated by Programs 11.2—11.4 when `visit(t)` is as shown in Program 11.5. The input binary trees are those of Figure 11.5.

```
template <class T>
void visit(binaryTreeNode<T> *x)
{// visit node *x, just output element field.
    cout << x->element;
}
```

Program 11.5 A visit function

Preorder	$+ * ab/cd$	$++ +abcd$	$/ + -a + xy * +b * ca$
Inorder	$a * b + c/d$	$a + b + c + d$	$-a + x + y / + b * c * a$
Postorder	$ab * cd/+$ (a)	$ab + c + d+$ (b)	$a - xy + +b + ca * */$ (c)

Figure 11.11 Elements of a binary tree listed in pre-, in-, and postorder

When an expression tree is output in pre-, in-, or postorder, we get the prefix, infix, and postfix forms of the expression, respectively. The **infix** form of an expression is the form in which we normally write an expression. In this form each binary operator (i.e., operator with two operands) appears just after the infix form of its left operand and just before the infix form of its right operand. An expression presented as a binary tree is unambiguous in the sense that the association between operators and operands is uniquely determined by the representation. This association is not uniquely determined by the representation when the infix form is used. For example, is $x + y * z$ to be interpreted as $(x + y) * z$ or $x + (y * z)$? To resolve this ambiguity, one assigns priorities to operators and employs priority rules. Further, delimiters such as parentheses are used to override these rules if necessary. In a *fully parenthesized* infix representation, each operator and its operands are en-

closed in a pair of parentheses. Furthermore, each of the operands of the operator are in fully parenthesized form. Some representations of this type are $((x) + (y))$, $((x) + ((y) * (z)))$, and $((((x) + (y)) * ((y) + (z))) * (w))$. This form of the expression is obtained by modifying inorder traversal as in Program 11.6.

```
template <class T>
void infix(binaryTreeNode<T> *t)
{// Output infix form of expression.
    if (t != NULL)
    {
        cout << '(';
        infix(t->leftChild); // left operand
        cout << t->element; // operator
        infix(t->rightChild); // right operand
        cout << ')';
    }
}
```

Program 11.6 Output fully parenthesized infix form

In the **postfix form** each operator comes immediately after the postfix form of its operands. The operands themselves appear in left-to-right order. In the **prefix form** each operator comes immediately before the prefix form of its operands. The operands themselves appear in left-to-right order. Like the binary tree representation, the prefix and postfix representations are unambiguous. As a result, neither the prefix nor the postfix representation employs parentheses or operator priorities. The association between operators and operands is easily determined by scanning the expression from right to left or from left to right and by employing a stack of operands. If an operand is encountered during this scan, it is stacked. If an operator is encountered, it is associated with the correct number of operands from the top of the stack. These operands are deleted from the stack and replaced by an operand that represents the result produced by the operator.

In a level-order traversal, elements are visited by level from top to bottom. Within levels, elements are visited from left to right. It is quite difficult to write a recursive function for level-order traversal, as the correct data structure to use here is a queue and not a stack. Program 11.7 traverses a binary tree in level order.

Program 11.7 enters the **while** loop only if the tree is not empty. The root is visited, and its children, if any, are added to the queue. Following the addition of the children of *t* to the queue, we attempt to access the front element of the queue. When the queue is empty, *front()* throws an exception of type *queueEmpty*; and when the queue is not empty, *front* returns a pointer to the front element. This element is the next node that is to be visited.

```
template <class T>
void levelOrder(binaryTreeNode<T> *t)
{// Level-order traversal of *t.
    arrayQueue<binaryTreeNode<T>*> q;
    while (t != NULL)
    {
        visit(t); // visit t

        // put t's children on queue
        if (t->leftChild != NULL)
            q.push(t->leftChild);
        if (t->rightChild != NULL)
            q.push(t->rightChild);

        // get next node to visit
        try {t = q.front();}
        catch (queueEmpty) {return;}
        q.pop();
    }
}
```

Program 11.7 Level-order traversal

Let n be the number of nodes in a binary tree. The space and time complexity of each of the four traversal programs is $O(n)$. To verify this claim, observe that the recursion stack space needed by pre-, in-, and postorder traversal is $\Theta(n)$ when the tree height is n (as is the case for a right-skewed binary tree (Figure 11.9)); the queue space needed by level-order traversal is $\Theta(n)$ when the tree is a full binary tree. For the time complexity observe that each of the traversal methods spends $\Theta(1)$ time at each node of the tree (assuming the time needed to visit a node is $\Theta(1)$).

EXERCISES

12. List the nodes of the binary trees of Figure 11.10 in pre-, in-, post-, and level order.
13. Do Exercise 12 for the full binary tree of Figure 11.6.
14. List the nodes of the binary trees for the expressions of Exercise 6 in pre-, in-, post-, and level order.

15. The nodes of a binary tree are labeled $a - h$. The preorder listing is $abcdefgh$, and the inorder listing is $cdbagfeh$. Draw the binary tree. Also, list the nodes of your binary tree in postorder and level order.
16. Do Exercise 15 for a binary tree with node labels $a - l$, preorder listing $abcde-fghijkl$, and inorder listing $aefdcgihjklb$.
17. The nodes of a binary tree are labeled $a - h$. The postorder listing is $abcdefgh$, and the inorder listing is $aedbchgf$. Draw the binary tree. Also, list the nodes of your binary tree in preorder and level order.
18. Do Exercise 17 for a binary tree with node labels $a - l$, postorder listing $abcdefghijkl$, and inorder listing $backdejfghl$.
19. Draw two binary trees whose preorder listing is $abcdefgh$ and whose postorder listing is $dcbgfhea$. Also, list the nodes of your binary trees in inorder and level order.
20. Write a function to perform a preorder traversal on a binary tree represented as an array. Assume that the elements of the binary tree are stored in array `a` and that `last` is the position of the last element of the tree. The array is of type `pair<bool, T>`, where `a[i].first` is `true` iff there is an element (`a[i].second`) at position `i`. What is the time complexity of your function?
21. Do Exercise 20 for inorder.
22. Do Exercise 20 for postorder.
23. Do Exercise 20 for level order.
24. Write a C++ function to make a copy of a binary tree represented as an array.
25. Write two C++ functions to make a copy of a binary tree that is represented using nodes of type `binaryTreeNode`. The first function should traverse the tree in postorder, and the second in preorder. What is the difference (if any) in the recursion stack space needed by these two functions?
26. Write a function to evaluate an expression tree that is represented using nodes of type `binaryTreeNode`. Develop a suitable data type for the elements in the tree.
27. Write a function to determine the height of a linked binary tree. What is the time complexity of your function?
28. Write a function to determine the number of nodes in a linked binary tree. What is the time complexity of your function?

Hidden page

11.7 THE ADT *BinaryTree*

Now that we have some understanding of what a binary tree is, we can specify it as an abstract data type (ADT 11.1). Since the number of operations we may wish to perform on a binary tree is quite large, we list only some of the commonly performed ones.

AbstractDataType *binaryTree*

{

instances

collection of elements; if not empty, the collection is partitioned into a root, left subtree, and right subtree; each subtree is also a binary tree;

operations

empty() : return **true** if empty, return **false** otherwise;

size() : return number of elements/nodes in the tree;

preOrder(visit) : preorder traversal of binary tree; *visit* is the visit function to use;

inOrder(visit) : inorder traversal of binary tree;

postOrder(visit) : postorder traversal of binary tree;

levelOrder(visit) : level-order traversal of binary tree;

}

ADT 11.1 The abstract data type *binary tree*

Program 11.8 gives the C++ abstract class *binaryTree* that corresponds to the ADT *binaryTree*. The class *T* refers to the data type of the nodes in the binary tree. The data type

void () (T *)*

used to specify the data type of the single parameter of the binary tree traversal methods specifies a function whose return type is *void* and which takes a single parameter of type *T**.

11.8 THE CLASS *linkedBinaryTree*

The class *linkedBinaryTree* derives from the abstract class *binaryTree* and uses nodes of the type *binaryTreeNode* (Program 11.1). Program 11.9 gives the data members and some of the public and private methods of of *linkedBinaryTree*.

```
template<class T>
class binaryTree
{
public:
    virtual ~binaryTree() {}
    virtual bool empty() const = 0;
    virtual int size() const = 0;
    virtual void preOrder(void (*) (T *)) = 0;
    virtual void inOrder(void (*) (T *)) = 0;
    virtual void postOrder(void (*) (T *)) = 0;
    virtual void levelOrder(void (*) (T *)) = 0;
};
```

Program 11.8 Binary tree abstract class

`linkedBinaryTree` has two instance data members `root` and `treeSize`. `root` is a pointer to the root node of the binary tree and `treeSize` is the number of nodes in the binary tree. The class `linkedBinaryTree` has a static data member `visit`, which is (a pointer to) a function whose return type is `void` and which has a single parameter that is a pointer to a `binaryTreeNode`. The static method `dispose` deletes a single node, and the method `erase` uses the static method `dispose` as the visit method for a postorder traversal to delete all nodes in a binary tree.

Program 11.10 gives the code for the preorder traversal method. The public method `preOrder` sets the class data member `visit` so that the desired function is used during the visit step. After setting the class data member `visit`, `preOrder` invokes the private recursive method `preOrder`, which does the actual preorder traversal. The corresponding inorder and postorder methods are similar.

The code for level-order traversal is quite similar to that given in Program 11.7. We may add a method to output a binary tree in preorder by adding the line

```
void preOrderOutput() {preOrder(output); cout << endl;}
```

to the public part of Program 11.9 and the lines

```
static void output(binaryTreeNode<E> *t)
    {cout << t->element << ' ' ;}
```

to the private part. Methods for inorder, postorder, and level-order output may be defined similarly.

Program 11.11 gives an additional public member method of `linkedBinaryTree` together with its accompanying private static recursive method. This recursive method determines the height of a binary tree by performing a postorder traversal of the binary tree. The method first determines the height of the left subtree, then

Hidden page

```
template<class E>
void linkedBinaryTree<E>::preOrder(binaryTreeNode<E> *t)
{// Preorder traversal.
    if (t != NULL)
    {
        linkedBinaryTree<E>::visit(t);
        preOrder(t->leftChild);
        preOrder(t->rightChild);
    }
}
```

Program 11.10 Private preorder method of `linkedBinaryTree`.

```
int height() const {return height(root);}

template <class E>
int linkedBinaryTree<E>::height(binaryTreeNode<E> *t)
{// Return height of tree rooted at *t.
    if (t == NULL)
        return 0; // empty tree
    int hl = height(t->leftChild); // height of left
    int hr = height(t->rightChild); // height of right
    if (hl > hr)
        return ++hl;
    else
        return ++hr;
}
```

Program 11.11 Determining the height of a binary tree

binary tree `*this` with the binary tree `x`. The method returns `true` iff the two binary trees are identical. Test your code. What is the complexity of your code?

46. Write the method `linkedBinaryTree<E>::swapTrees()`, which swaps the left and right subtrees of each node of a binary tree. Test your code. What is the complexity of your code?
47. Let $heightDifference(x)$ be the absolute value of the difference in heights of the left and right subtrees of node x . Let $maxHeightDifference(t)$ be $\max\{heightDifference(x) | x \text{ is a node of the binary tree } t\}$. Write the method

Hidden page

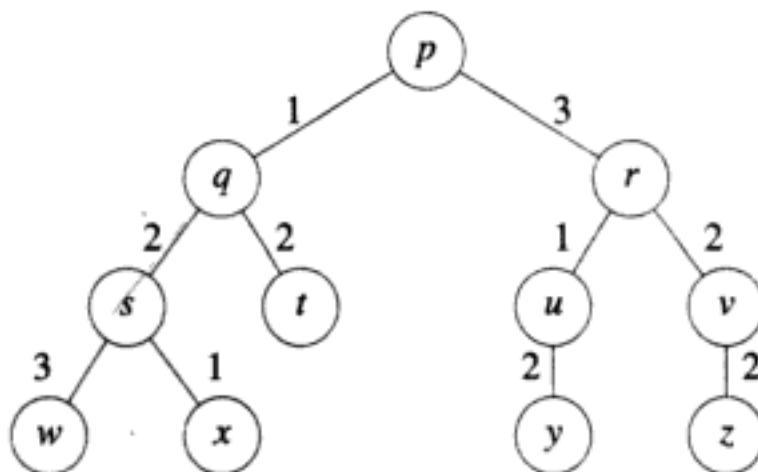
of pipes from the source of the petroleum/natural gas to the consumption sites. Similarly, electrical power may be distributed through a network of wires from the power plant to the points of consumption. We will use the term **signal** to refer to the resource (petroleum, natural gas, power, etc.) that is to be distributed. While the signal is being transported through the distribution network, it may experience a loss in or degradation of one or more of its characteristics. For example, there may be a pressure drop along a natural gas pipeline or a voltage drop along an electrical transmission line. In other situations noise may enter the signal as it moves along the network. Between the signal source and point of consumption, we can tolerate only a certain amount, *tolerance*, of signal degradation. To guarantee a degradation that does not exceed this amount, **signal boosters** are placed at strategic places in the network. A signal booster might, for example, increase the signal pressure or voltage so that it equals that at the source or may enhance the signal so that the signal-to-noise ratio is the same as that at the source. In this section we develop an algorithm to determine where to place signal boosters. Our objective is to minimize the number of boosters in use while ensuring that the degradation in signal (relative to that at the source) does not exceed the given tolerance.

To simplify the problem, we assume that the distribution network is a tree with the source as the root. Each node in the tree (other than the root) represents a substation where we can place a booster. Some of these nodes also represent points of consumption. The signal flows from a node to its children. Figure 11.12 shows a distribution network that is a tree. Each edge is labeled by the amount of signal degradation that takes place when a signal flows between the corresponding parent and child. The units of degradation are assumed to be additive. That is, when a signal flows from node p to node v in Figure 11.12, the degradation is 5; the degradation from node q to node x is 3. When a signal booster is placed at node r , the strength of the signal that arrives at r is three units less than that of the signal that leaves source p ; however, the signal that leaves r has the same strength as the signal that left p . Therefore, the signal that arrives at v has degraded by two units relative to the signal at the source; and the signal that arrives at z is four units weaker than the signal that left p . Without the booster at r , the arriving signal at z will be seven units weaker than the signal that left p .

Solution Strategy

Let $degradeFromParent(i)$ denote the degradation between node i and its parent. Therefore, in Figure 11.12, $degradeFromParent(w) = 2$, $degradeFromParent(p) = 0$, and $degradeFromParent(r) = 3$. Since signal boosters can be placed only at nodes of the distribution tree, the presence of a node i with $degradeFromParent(i) > tolerance$ implies that no placement of boosters can prevent signal degradation from exceeding *tolerance*. For example, if $tolerance = 2$, then there is no way to place signal boosters so that the degradation between p and r is $\leq tolerance = 2$ in Figure 11.12.

For any node i let $degradeToLeaf(i)$ denote the maximum signal degradation from node i to any leaf in the subtree rooted at i . If i is a leaf node, then

**Figure 11.12** Tree distribution network

$\text{degradeToLeaf}(i) = 0$. For the example of Figure 11.12, $\text{degradeToLeaf}(i) = 0$ for $i \in \{w, x, t, y, z\}$. For the remaining nodes $\text{degradeToLeaf}(i)$ may be computed using the following equality:

$$\text{degradeToLeaf}(i) = \max_{j \text{ is a child of } i} \{\text{degradeToLeaf}(j) + \text{degradeFromParent}(j)\}$$

So $\text{degradeToLeaf}(s) = 3$. To use this equation, we must compute the degradeToLeaf value of a node after computing that of its children. Therefore, we must traverse the tree so that we visit a node after we visit its children. We can compute the degradeToLeaf value of a node when we visit it. This traversal order is a natural extension of postorder traversal to trees of degree (possibly) more than 2.

Suppose that during the computation of degradeToLeaf as described above, we encounter a node i with a child j such that

$$\text{degradeToLeaf}(j) + \text{degradeFromParent}(j) > \text{tolerance}$$

If we do not place a booster at j , then the signal degradation from i to a leaf will exceed tolerance even if a booster is placed at i . For example, in Figure 11.12, when computing $\text{degradeToLeaf}(q)$, we compute

$$\text{degradeToLeaf}(s) + \text{degradeFromParent}(s) = 5$$

If $\text{tolerance} = 3$, then placing a booster at q or at any of q 's ancestors doesn't reduce signal degradation between q and its descendants. We need to place a booster at s . If a booster is placed at s , then $\text{degradeToLeaf}(q) = 3$.

Hidden page

Proof The proof is by induction on the number n of nodes in the distribution tree. If $n = 1$, the theorem is trivially valid. Assume that the theorem is valid for $n \leq m$ where m is an arbitrary natural number. Let t be a tree with $n + 1$ nodes. Let X be the set of vertices at which the outlined procedure places boosters and let W be a minimum cardinality placement of boosters that satisfies the tolerance requirements. We need to show that $|X| = |W|$.

If $|X| = 0$, then $|X| = |W|$. If $|X| > 0$, then let z be the first vertex at which a booster is placed by the outlined procedure. Let t_z be the subtree of t rooted at z . Since $\text{degradeToLeaf}(z) + \text{degradeFromParent}(z) > \text{tolerance}$, W must contain at least one vertex u that is in t_z . If W contains more than one such u , then W cannot be of minimum cardinality because by placing boosters at $W - \{\text{all such } u\} + \{z\}$, we can satisfy the tolerance requirement. Hence W contains exactly one such u . Let $W' = W - \{u\}$. Let t' be the tree that results from the removal of t_z from t except z . We see that W' is a minimum cardinality booster placement for t' that satisfies the tolerance requirement. Also, $X' = X - \{z\}$ satisfies the tolerance requirement for t' and is the booster placement generated by our outlined procedure on the tree t' . Since the number of vertices in t' is less than $m + 1$, $|X'| = |W'|$. Hence $|X| = |X'| + 1 = |W'| + 1 = |W|$. ■

C++ Implementation

When no node of the distribution tree has more than two children, the distribution tree may be represented as a binary tree by using the class `linkedBinaryTree` (Program 11.9) and the struct `booster` (Program 11.12). The field `boosterHere` is used to differentiate between nodes at which a booster is placed and those where it is not. The `element` fields of our binary tree will be of type `booster`.

We can compute the `degradeToLeaf` values for the nodes and the location of a minimum set of boosters by performing a postorder traversal of the binary distribution tree. During the visit step, we execute the code of Program 11.13. The code for `placeBoosters` assumes that `tolerance` is a global variable.

If t is a `linkedBinaryTree` whose `degradeFromParent` fields have been set to the degradation values and whose `boosterHere` fields have been set to `false`, then the invocation `t.postOrder(placeBoosters)` will reset the `degradeToLeaf` and `boosterHere` fields correctly. Since the complexity of `placeBoosters` is $\Theta(1)$, the invocation `t.postOrder(placeBoosters)` takes $O(n)$ time where n is the number of nodes in the distribution tree.

Binary Tree Representation of a Tree

When the distribution tree t contains nodes that have more than two children, we can still represent the tree as a binary tree. This time, for each node x of the tree t , we link its children into a chain using the `rightChild` fields of the children nodes. The `leftChild` field of x points to the first node in this chain. The `rightChild` field of x is used for the chain of x 's siblings. Figure 11.15 shows a tree and its

```

struct booster
{
    int degradeToLeaf,           // degradation to leaf
        degradeFromParent;      // degradation from parent
    bool boosterHere;           // true iff booster here

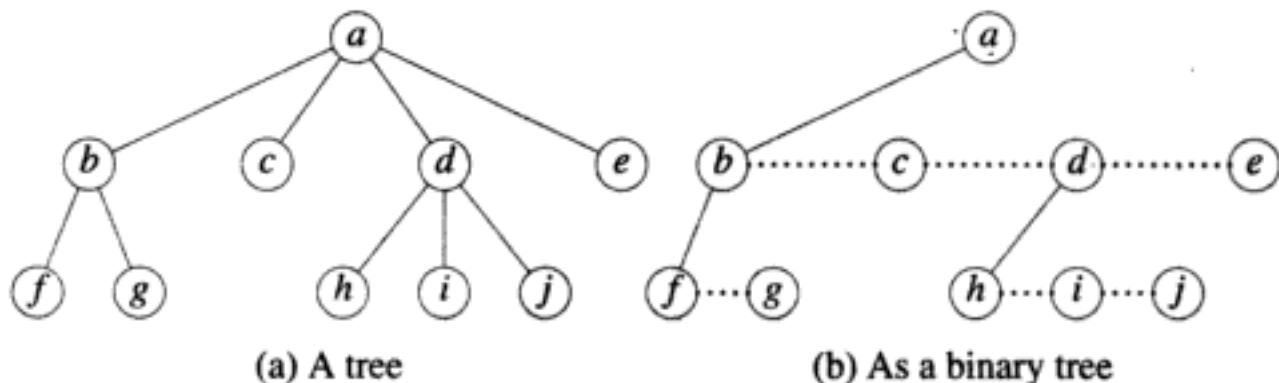
    void output(ostream& out) const
    {out << boosterHere << ' ' << degradeToLeaf << ' '
     << degradeFromParent << ' ';}
};

// overload <<
ostream& operator<<(ostream& out, booster x)
{x.output(out); return out;}

```

Program 11.12 The struct booster

binary tree representation. Solid lines represent left-child pointers, and right-child pointers are shown as dotted lines.

**Figure 11.15** A tree and its binary tree representation

When the binary tree representation of a tree is used, the invocation `t.postOrder-(placeBoosters)` does not have the desired effect. The development of the new function to compute `degradeToLeaf` and `boosterHere` is considered in Exercise 57.

```
void placeBoosters(binaryTreeNode<booster> *x)
{// Compute degradation at *x. Place booster(s) at children
 // of x if degradation exceeds tolerance.

    x->element.degradeToLeaf = 0; // initialize degradation at x

    // compute degradation from left subtree of x and
    // place a booster at the left child of x if needed
    binaryTreeNode<booster> *y = x->leftChild;
    if (y != NULL)
        {// x has a nonempty left subtree
            int degradation = y->element.degradeToLeaf +
                y->element.degradeFromParent;
            if (degradation > tolerance)
                {// place a booster at y
                    y->element.boosterHere = true;
                    x->element.degradeToLeaf = y->element.degradeFromParent;
                }
            else // no booster needed at y
                x->element.degradeToLeaf = degradation;
        }

    // compute degradation from right subtree of x and
    // place a booster at the right child of x if needed
    y = x->rightChild;
    if (y != NULL)
        {// x has a nonempty right subtree
            int degradation = y->element.degradeToLeaf +
                y->element.degradeFromParent;
            if (degradation > tolerance)
                {// place booster at y
                    y->element.boosterHere = true;
                    degradation = y->element.degradeFromParent;
                }
            if (x->element.degradeToLeaf < degradation)
                x->element.degradeToLeaf = degradation;
        }
    }
}
```

Program 11.13 Place boosters and determine degradeToLeaf for binary distribution trees

11.9.2 Union-Find Problem

Problem Description

The union-find problem was introduced in Section 6.5.4. Basically, we begin with n elements numbered 1 through n ; initially, each is in a class of its own, and we perform a sequence of find and combine operations. The operation `find(theElement)` returns a unique characteristic of the class that `theElement` is in, and `combine(a,b)` combines the classes that contain the elements `a` and `b`. In Section 6.5.4 we saw that `combine(a,b)` is usually implemented by using the union operation `unite(classA, classB)` where `classA = find(a)`, `classB = find(b)`, and `classA ≠ classB`. The solution provided in Section 6.5.4 used chains and had a complexity $O(n + u \log u + f)$ where u is the number of union operations and f is the number of find operations performed. In this section we explore an alternative solution in which each set (or class) is represented as a tree.

Representing a Set as a Tree

Any set S may be represented as a tree with $|S|$ nodes, one node per element. Any element of S may be selected as the root element; any subset of the remaining elements could be the children of the root; any subset of the elements that remain could be the grandchildren of the root; and so on.

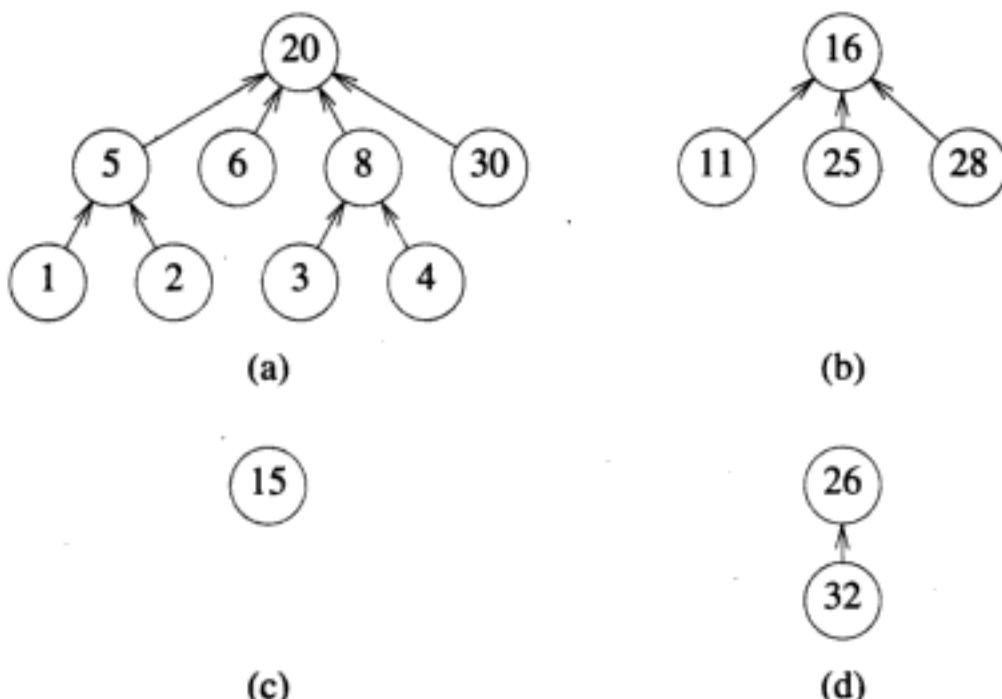
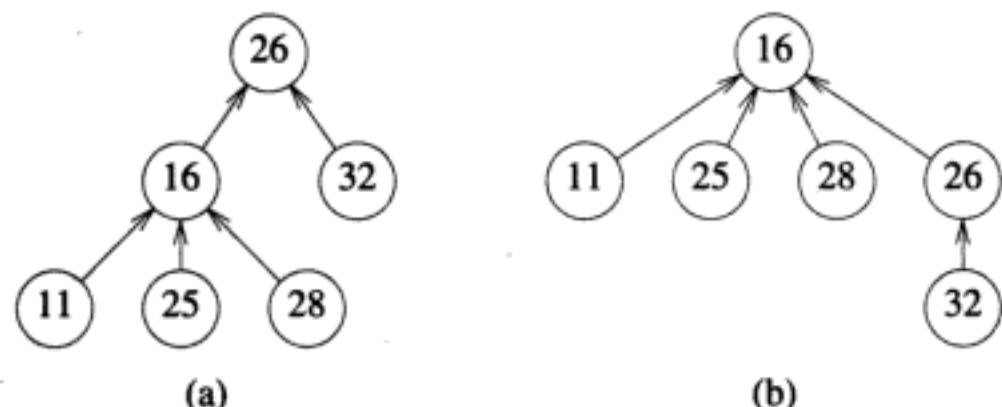
Figure 11.16 shows some sets represented as trees. Notice that each node that is not a root has a pointer to its parent in the tree. Our pointers go from a node to its parent because the find operation will require us to move up a tree. Neither the find nor union operations requires us to move down a tree.

We say that the elements 1, 2, 20, 30, and so on are in the set with root 20; the elements 11, 16, 25, and 28 are in the set with root 16; the element 15 is in the set with root 15; and the elements 26 and 32 are in the set with root 26 (or simply the set 26).

Solution Strategy

Our strategy to solve the union-find problem is to represent each set as a tree. For the find operation we use the element in the root as the set identifier. So `find(3)` returns the value 20 (see Figure 11.16); `find(1)` returns 20; and `find(26)` returns 26. Since each tree has a unique root, `find(i) = find(j)` iff i and j are in the same set. To find the set that contains `theElement`, we begin at the node for element `theElement` and move up the tree until we reach the root.

For the union operation we assume that the invocation `unite(classA, classB)` is done with `classA` and `classB` being the elements in the roots of two different trees (i.e., `classA ≠ classB`). To unite the two trees (or sets) of elements, we make one tree a subtree of the other. For instance, if `classA = 16` and `classB = 26` (Figure 11.16), the tree of Figure 11.17(a) results if `classA` is made a subtree of `classB`, whereas the result is Figure 11.17(b) if `classB` is made a subtree of `classA`.

**Figure 11.16** Tree representation of disjoint sets**Figure 11.17** Union

C++ Implementation

The solution to the union-find problem is a good example of the use of simulated pointers. A linked representation of the trees is needed. Each node must have a **parent** field. Children fields are, however, not needed. We also have a need to make direct access to nodes. To find the set containing element 10, we need to determine which node represents the element 10 and then follow a sequence of **parent** pointers

Hidden page

```

void initialize(int numberOfElements)
{ // Initialize numberOfElements trees, 1 element per set/class/tree.
    parent = new int[numberOfElements + 1];
    for (int e = 1; e <= numberOfElements; e++)
        parent[e] = 0;
}

int find(int theElement)
{ // Return root of tree that contains theElement.
    while (parent[theElement] != 0)
        theElement = parent[theElement]; // move up one level
    return theElement;
}

void unite(int rootA, int rootB)
{ // Combine trees with different roots rootA and rootB.
    parent[rootB] = rootA;
}

```

Program 11.14 Simple tree solution to the union-find problem

check `rootA ≠ rootB` is performed externally. `rootB` is always made a subtree of `rootA`.

Performance Analysis

The time complexity of the constructor is $O(\text{numberOfElements})$; the complexity of `find(theElement)` is $O(h)$ where h is the height of the tree that contains element `theElement`; and the complexity of `unite(rootA, rootB)` is $\Theta(1)$.

In a typical application of the union-find problem, many union and find operations are performed. Furthermore, in typical applications we do not care how much time an individual operation takes; we are concerned with the time taken for all operations collectively. Assume that u unions and f finds are to be performed. Since each union is necessarily preceded by two finds (these finds determine the roots of the trees to be united), we may assume that $f > u$. Each union takes $\Theta(1)$ time. The time for each find depends on the height of the trees that get created. In the worst case a tree with m elements can have a height of m . This worst case happens, for example, when the following sequence of unions is performed:

`unite(2, 1), unite(3, 2), unite(4, 3), unite(5, 4), ...`

Hence each find can take as much as $\Theta(q)$ time where q is the number of unions

that have been performed before the find. Therefore, the time for a sequence of operations becomes $O(fu)$. The worst-case time needed for a sequence of operations can be reduced to almost $O(f+u) = O(f)$ by enhancing the union and find methods as described next.

Enhancing the Union Function

We can enhance the performance of the union-find algorithms by using either the **weight** or the **height** rule when performing a union of the trees with roots i and j .

Definition 11.3 [Weight rule] *If the number of nodes in the tree with root i is less than the number in the tree with root j , then make i a child of j ; otherwise, make j a child of i .* ■

Definition 11.4 [Height rule] *If the height of tree i is less than that of tree j , then make j the parent of i ; otherwise, make i the parent of j .* ■

If we perform a union on the trees of Figure 11.16 (a) and (b), then the tree with root 16 becomes a subtree of the tree with root 20 regardless of whether the weight or the height rule is used. When we perform a union on the trees of Figure 11.19 (a) and (b), the tree with root 16 becomes a subtree of the tree with root 20 in case the weight rule is used. However, when the height rule is used, the tree with root 20 becomes a subtree of the one with root 16.

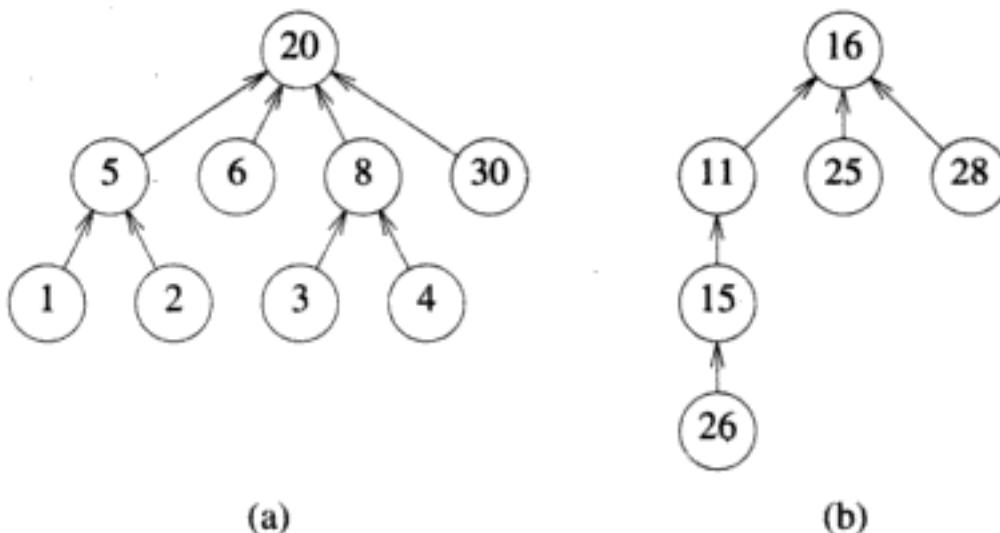


Figure 11.19 Two sample trees

To incorporate the weight rule into the function for a union, we add a Boolean field `root` to each node. The `root` field of a node is `true` iff the node is currently

a root node. The `parent` field of each root node keeps a count of the total number of nodes in the tree. For the trees of Figure 11.16, we have `node[i].root = true` iff $i = 20, 16, 15$, or 26 . Also, `node[i].parent = 9, 4, 1`, and 2 for $i = 20, 16, 15$, and 26 , respectively. The remaining `parent` fields are unchanged.

To implement the weighting rule, we define the struct `unionFindNode` that defines the data type of the nodes in a tree. Program 11.15 gives the code for this struct.

```
struct unionFindNode
{
    int parent; // if true then tree weight
                 // otherwise pointer to parent in tree
    bool root; // true iff root

    unionFindNode()
        {parent = 1; root = true;}
};
```

Program 11.15 Struct used when implementing the weighting rule

The `initialize`, `find` and `unite` functions now take the form given in Program 11.16.

The time required to perform a union has increased somewhat but is still bounded by a constant; it is $\Theta(1)$. Lemma 11.1 determines the maximum time to perform a find.

Lemma 11.1 [Weight rule lemma] *Assume that we start with singleton sets and perform unions, using the weight rule (as in Program 11.16). Let t be a tree with p nodes created in this way. The height of t is at most $\lfloor \log_2 p \rfloor + 1$.*

Proof The lemma is clearly true for $p = 1$. Assume it is true for all trees with i nodes, $i \leq p - 1$. We will show that it is also true for $i = p$. Consider the last union operation, `union(k, j)`, performed to create tree t . Let m be the number of nodes in tree j and let $p - m$ be the number of nodes in k . Without loss of generality, we may assume $1 \leq m \leq p/2$. So j is made a subtree of tree k . Therefore, the height of t either is the same as that of k or is one more than that of j . If the former is the case, then the height of t is $\leq \lfloor \log_2(p - m) \rfloor + 1 \leq \lfloor \log_2 p \rfloor + 1$. If the latter is the case then the height of t is $\leq \lfloor \log_2 m \rfloor + 2 \leq \lfloor \log_2 p/2 \rfloor + 2 \leq \lfloor \log_2 p \rfloor + 1$. ■

If we start with singleton sets and perform an intermixed sequence of u unions and f finds, no set will have more than $u + 1$ elements in it. From Lemma 11.1 it follows that when the weight rule is used, the cost of the sequence of union and find operations (excluding the initialization time) is $O(u + f \log u) = O(f \log u)$ (since we assume $f > u$).

```
void initialize(int numberElements)
{ // Initialize numberElements trees, 1 element per set/class/tree.
    node = new unionFindNode[numberElements+1];
}

int find(int theElement)
{ // Return root of tree containing theElement.
    while (!node[theElement].root)
        theElement = node[theElement].parent; // move up one level
    return theElement;
}

void unite(int rootA, int rootB)
{ // Combine trees with different roots rootA and rootB.
    // Use the weighting rule.
    if (node[rootA].parent < node[rootB].parent)
        { // rootA becomes subtree of rootB
            node[rootB].parent += node[rootA].parent;
            node[rootA].root = false;
            node[rootA].parent = rootB;
        }
    else
        { // rootB becomes subtree of rootA
            node[rootA].parent += node[rootB].parent;
            node[rootB].root = false;
            node[rootB].parent = rootA;
        }
}
```

Program 11.16 Functions using the weighting rule

When the weight rule is replaced by the height rule in Program 11.16, the bound of Lemma 11.1 still governs the height of the resulting trees. Exercises 60, 61, and 62 explore the use of the height rule.

Enhancing the Find Function

Further improvement in the worst-case performance is possible by modifying the find function of Program 11.14 so as to reduce the length of the path from the find element e to the root. This reduction in path length is obtained by using a process called **path compression**, which we can do in at least three different ways—path compaction, path splitting, and path halving.

In **path compaction** we change the pointers in all nodes on the path from the element being searched to the root so that these nodes point directly to the root. As an example, consider the tree of Figure 11.20. When we perform a `find(10)`, the nodes 10, 15, and 3 are determined to be on the path from 10 to the root. Their parent fields are changed to 2, and the tree of Figure 11.21 is obtained. (Since node 3 already points to 2, its field doesn't have to be changed; when writing the program, it turns out to be easier to include this node in the set of nodes whose parent field is to be changed.)

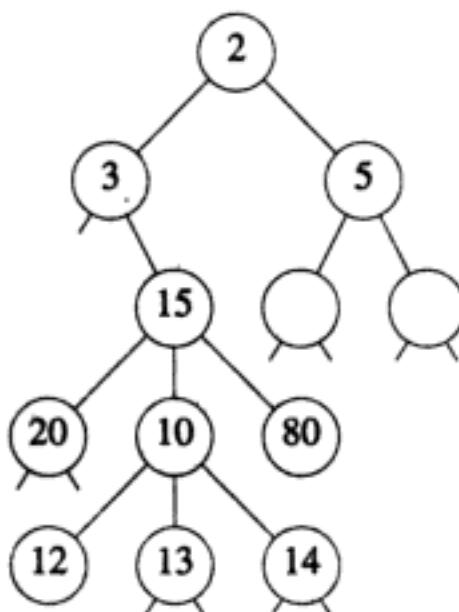


Figure 11.20 Sample tree

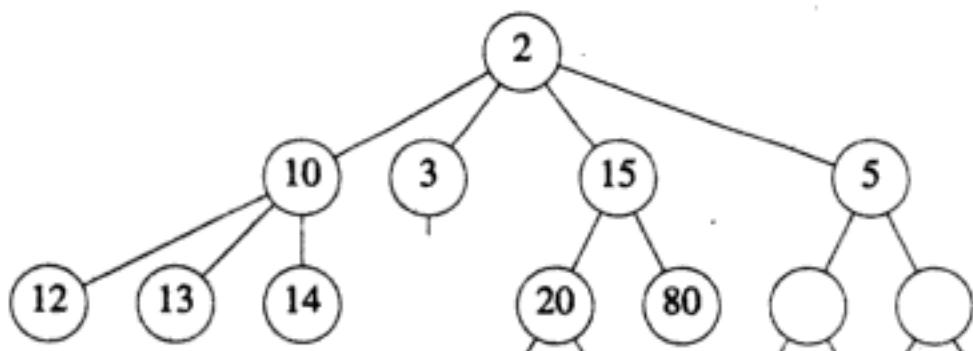


Figure 11.21 Path compaction

Although path compaction increases the time needed for an individual find, it

reduces the cost of future finds. For instance, finding the elements in the subtrees of 10 and 15 is quicker in the compacted tree of Figure 11.21. Program 11.17 implements the compaction rule.

```

int find(int theElement)
{ // Return root of tree containing theElement.
  // Compact path from theElement to root.

  // theRoot will eventually be the root of the tree
  int theRoot = theElement;
  while (!node[theRoot].root)
    theRoot = node[theRoot].parent;

  // compact path from theElement to theRoot
  int currentNode = theElement; // start at theElement
  while (currentNode != theRoot)
  {
    int parentNode = node[currentNode].parent;
    node[currentNode].parent = theRoot; // move to level 2
    currentNode = parentNode;           // moves to old parent
  }

  return theRoot;
}

```

Program 11.17 Find an element using path compaction

In **path splitting** we change the parent pointer in each node (except the root and its child) on the path from e to the root to point to the node's original grandparent. In the tree of Figure 11.20, path splitting beginning at node 13 results in the tree of Figure 11.22. Note that when we use path splitting, a single pass from e to the root suffices.

In **path halving** we change the parent pointer of every other node (except the root and its child) on the path from e to the root to point to the node's grandparent. As a result, in path halving only half as many pointers are changed as in path splitting. As in the case of path splitting, a single pass from e to the root suffices. Figure 11.23 shows the result of path halving beginning at node 13 of Figure 11.20.

Enhancing Both the Union and the Find Methods

Path compression may change the height of a tree, but not its weight. Determining the new tree height following path compression requires considerable effort. So while it is relatively easy to use any of the path-compression schemes in conjunction

Hidden page

process an intermixed sequence of unions and finds is almost linear in the number of unions and finds. Not only is the complexity analysis exceptionally difficult, but stating the result is also quite difficult.

We first define the explosively growing Ackermann's function $A(i, j)$ and its inverse $\alpha(p, q)$ (which grows at a snail's pace) as follows:

$$A(i, j) = \begin{cases} 2^j & i = 1 \text{ and } j \geq 1 \\ A(i - 1, 2) & i \geq 2 \text{ and } j = 1 \\ A(i - 1, A(i, j - 1)) & i, j \geq 2 \end{cases}$$

$$\alpha(p, q) = \min\{z \geq 1 | A(z, \lfloor p/q \rfloor) > \log_2 q\}, \quad p \geq q \geq 1$$

Exercise 67 shows that $A(i, j)$ is an increasing function of i and j ; that is, $A(i, j) > A(i - 1, j)$, and $A(i, j) > A(i, j - 1)$. Actually, $A(i, j)$ is a very rapidly growing function of i and j . Consequently, its inverse α grows very slowly as p and q are increased.

Example 11.7 To get a feel for how fast Ackermann's function grows, let us evaluate $A(i, j)$ for a few small values of i and j . From the definition we get $A(2, 1) = A(1, 2) = 2^2 = 4$. Further, for $j \geq 2$, $A(2, j) = A(1, A(2, j - 1)) = 2^{A(2, j - 1)}$. So $A(2, 2) = 2^{A(2, 1)} = 2^4 = 16$; $A(2, 3) = 2^{A(2, 2)} = 2^{16} = 65,536$; $A(2, 4) = 2^{A(2, 3)} = 2^{65,536}$ (which is toooooooo big to write here); and

$$A(2, j) = 2^{2^{2^{\dots^{\dots}}}}$$

where the stack of twos on the right side is made up of $j + 1$ twos. $A(2, j)$ is quite a frightening number for $j > 3$.

$A(3, 1) = A(2, 2) = 16$ is not a number to be frightened by. But $A(4, 1) = A(3, 2) = A(2, A(3, 1)) = A(2, 16)$. If $A(2, 4)$ is toooooooo big, then how big is $A(4, 1) = A(2, 16)$?

The intent of this example is not to impress you with how fast $A(i, j)$ grows, but with how slow $\alpha(p, q)$ grows. When $q = 65,535 = 2^{16} - 1$, $\log q < 16$. Since $A(3, 1) = 16$, $\alpha(65,535, 65,535) = 3$ and $\alpha(p, 65,535) \leq 3$ for $p \geq 65,535$. $\alpha(65,536, 65,536) = 4$ and $\alpha(p, 65,536) \leq 4$ for $p \geq 65,536$. $\alpha(q, q)$ does not become 5 until $q = 2^{A(4,1)}$, an amazingly large number. Therefore, $\alpha(p, q)$, $p \geq q$, does not become 5 until $q = 2^{A(4,1)}$.

In the complexity analysis of the enhanced union-find algorithms, q is the number of elements, and p is the sum of the number of finds and the number of elements. Therefore, for all practical purposes, we can assume that $\alpha(p, q) \leq 4$. ■

Hidden page

56. A **forest** is a collection of zero or more trees. In the binary tree representation of a tree, the root has no right child. We may use this observation to arrive at a binary tree representation for a forest with m trees. First we obtain the binary tree representation of each tree in the forest. Next the i th tree is made the right subtree of the $(i - 1)$ th, $2 \leq i \leq m$. Draw the binary tree representation of the four-tree forest of Figure 11.16, the two-tree forest of Figure 11.17, and the two-tree forest of Figure 11.19.
57. Let t be an instance of `linkedBinaryTree`. Assume that t is the binary tree representation of a distribution tree (see Figure 11.15). Develop a program to compute the `degradeToLeaf` and `boosterHere` values of each node in t . Your program should also output these values by invoking `t.postOrderOutput()`. Use suitable distribution trees to test your program.
58. Suppose we start with n sets, each containing a distinct element.
 - (a) Show that if u unions are performed, then no set contains more than $u + 1$ elements.
 - (b) Show that at most $n - 1$ unions can be performed before the number of sets becomes 1.
 - (c) Show that if fewer than $\lceil n/2 \rceil$ unions are performed, then at least one set with a single element in it remains.
 - (d) Show that if u unions are performed, then at least $\max\{n - 2u, 0\}$ singleton sets remain.
59. Give an example of a sequence of unions that start with singleton sets and create trees whose height equals the upper bound given in Lemma 11.1. Assume that each union is performed with the weight rule.
60. Write a version of the method `union` (Program 11.14) that uses the height rule instead of the weight rule.
61. Prove Lemma 11.1 for the case when the height rule is used instead of the weight rule.
62. Give an example of a sequence of unions that start with singleton sets and create trees whose height equals the upper bound given in Lemma 11.1. Assume that each union is performed with the height rule.
63. Compare the average performance of the simple union/find solution of Programs 11.14 and the solution that uses the weight rule as well as path compaction (Programs 11.16 and 11.17). Do this comparison for different values of n . For each value of n , generate a random sequence of pairs (i, j) . Replace each pair by two finds (one for i and the other for j). If the two are in different sets, then a union is to be performed. Repeat the experiment with

many different random sequences. Measure the total time taken over these sequences. It is left to you to take this basic description of the experiment and plan a meaningful experiment to compare the average performance of the two sets of programs. Write a report that describes your experiment and your conclusions. Include program listings, a table of average times, and graphs in your report.

64. Write code for the find operation that uses path halving instead of path compaction (as used in Program 11.17).
65. Write code for the find operation that uses path splitting instead of path compaction (as used in Program 11.17).
66. Program the six ways in which you can achieve the performance stated in Theorem 11.2. Conduct experiments to evaluate these six solutions to determine which performs best.
67. Show that
 - (a) $A(i, j) > A(i - 1, j)$ for $i > 1$ and $j \geq 1$.
 - (b) $A(i, j) > A(i, j - 1)$ for $i \geq 1$ and $j > 1$.
 - (c) $\alpha(r, q) \geq \alpha(p, q)$ for $r > p \geq q \geq 1$.

11.10 REFERENCES AND SELECTED READINGS

The book *The Art of Computer Programming: Fundamental Algorithms*, Volume 1, Third Edition, by D. Knuth, Addison-Wesley, Reading, MA, 1997, is a good reference for material on binary trees. The problem of placing boosters is studied in the following papers: "Deleting Vertices in Dags to Bound Path Lengths" by D. Paik,, S. Reddy, and S. Sahni, *IEEE Transactions on Computers*, 43, 9, 1994, 1091–1096, and "Heuristics for the Placement of Flip-Flops in Partial Scan Designs and for the Placement of Signal Boosters in Lossy Circuits" by D. Paik, S. Reddy, and S. Sahni, *Sixth International Conference on VLSI Design*, 1993, 45–50.

A complete analysis of the tree representations for the inline equivalence problem appears in the paper "Worst Case Analysis of Set Union Algorithms" by R. Tarjan and J. Leeuwen, *Journal of the ACM*, 31, 2, 1984, 245–281.

The Web site for this book develops some tree varieties not covered in the text—pairing heaps, interval heaps, tree structures for double-ended priority queues, tries, and suffix trees. You should look at these only after you have read Chapters 12 through 15.

CHAPTER 12

PRIORITY QUEUES

BIRD'S-EYE VIEW

Unlike the queues of Chapter 9, which are FIFO structures, the order of deletion from a priority queue is determined by the element priority. Elements are removed/deleted either in increasing or decreasing order of priority rather than in the order in which they arrived in the queue.

A priority queue is efficiently implemented with the heap data structure, which is a complete binary tree that is most efficiently stored by using the array representation described in Section 11.4.1. Linked data structures suitable for the implementation of a priority queue include height- and weight-biased leftist trees. This chapter covers both heaps and leftist trees. An additional priority queue data structure—pairing heaps—is developed in the Web site. The Web site also includes data structures for double-ended priority queues that allow you to delete both in increasing and in decreasing order of priority. The C++ STL class `priority_queue` implements a priority queue as a heap.

In the applications section at the end of the chapter, we use heaps to develop an $O(n \log n)$ sorting method called heap sort. The sort methods of Chapter 2 take $O(n^2)$ to sort n elements. Even though the bin sort and radix sort methods of Chapter 6 run in $O(n)$ time, they are limited to elements with keys in an appropriate range. So heap sort is the first general-purpose sort we are seeing that has a complexity better than $O(n^2)$. Chapter 18 discusses other sort methods with this

complexity. From the asymptotic-complexity point of view, heap sort is an optimal sorting method because we can show that every general-purpose sorting method that relies on comparing pairs of elements has a complexity that is $\Omega(n \log n)$ (Section 18.4.2).

The other applications considered in this chapter are machine scheduling and the generation of Huffman codes. The machine-scheduling application allows us to introduce the NP-hard class of problems. This class includes problems for which no polynomial-time algorithms are known. As noted in Chapter 3, for large instances only polynomial-time algorithms are practical. As a result, NP-hard problems are often solved by approximation algorithms or heuristics that complete in a reasonable amount of computer time, but do not guarantee to find the best answer. For the machine-scheduling application, we use the heap data structure to obtain an efficient implementation of a much-studied machine-scheduling approximation algorithm.

12.1 DEFINITION AND APPLICATIONS

A **priority queue** is a collection of zero or more elements. Each element has a priority or value. The operations performed on a priority queue are (1) find an element, (2) insert a new element, and (3) remove (or delete) an element. The functions that correspond to the find, insert, and remove operations of a priority queue are, respectively, named *top*, *push* and *pop*. In a **min priority queue**, the find operation finds the element with minimum priority, and the remove operation removes this element. In a **max priority queue**, the find operation finds the element with maximum priority, and the remove operation removes this element. The elements in a priority queue need not have distinct priorities. The find and remove operations may break ties in any manner.

Example 12.1 Suppose that we are selling the services of a machine over a fixed period of time (say, a day or a month). Although each user pays a fixed amount per use, the time needed by each user is different. To maximize the earning from this machine (under the assumption that the machine is not to be kept idle unless no user is available), we maintain a min priority queue of all users waiting for the machine. The priority of a user is the amount of time he/she needs. When a new user requests the machine, his/her request is put into the priority queue. Whenever the machine becomes available, the user with the smallest time requirement (i.e., priority) is selected.

If each user needs the same amount of time on the machine but users are willing to pay different amounts for the service, then we can use a priority queue in which the element priorities are the amount of payment. Whenever the machine becomes available, the user paying the most is selected. This selection requires a max priority queue. ■

Example 12.2 [Event List] The machine shop simulation problem was introduced in Section 9.5.4. The operations performed on the event queue are (1) find the machine with minimum finish time and (2) change the finish time of this machine. Suppose we set up a min priority queue in which each element represents a machine, and an element's priority is the finish time of the machine it represents. The *top* operation of a min priority queue gives us the machine with the smallest finish time. To change a machine's finish time, we may first do a *top* and a *pop*, and then *push* the removed element/machine with its priority changed to the new finish time. Actually, for event list applications it is desirable for priority queues to include an additional operation to change the priority of the top element. Such an operation would collapse the three step process to change a machine's finish time (*top* followed by *pop* followed by *push*) into one.

Max priority queues may also be used in the machine shop simulation problem. In the simulation programs of Section 9.5.4, each machine served its set of waiting jobs in a first-come-first-served manner. Therefore, we used a FIFO queue at each

machine. If the service discipline is changed to "when a machine is ready for a new job, it selects the waiting job with maximum priority," we need a max priority queue at each machine. The operations that are to be performed at each machine are (1) when a new job arrives at the machine, it is inserted into the max priority queue for that machine, and (2) when a machine is ready to work on a new job, a job with maximum priority is removed from its queue and made active.

When the service discipline at each machine is changed as above, the simulation problem of Section 9.5.4 requires a min priority queue for the event list and a max priority queue at each machine. ■

In this chapter we develop efficient representations for priority queues. Since the implementations for min and max priority queues are very similar, we explicitly develop only those for max priority queues.

12.2 THE ABSTRACT DATA TYPE

The abstract data type specification for a max priority queue is given in ADT 12.1. The specification for a min priority queue is the same except that *top*' and *pop*, respectively, find and remove the element with minimum priority.

```
AbstractDataType maxPriorityQueue
{
    instances
        finite collection of elements, each has a priority
    operations
        empty() : return true iff the queue is empty
        size() : return number of elements in the queue
        top() : return element with maximum priority
        pop() : remove the element with largest priority from the queue;
        push(x) : insert the element x into the queue
}
```

ADT 12.1 Abstract data type specification of a max priority queue

Program 12.1 gives the C++ abstract class that corresponds to the ADT *maxPriorityQueue*. We make the assumption that when two elements of type *T* are compared using relational operators such as *<* and *<=* the element priorities are compared.

```
template<class T>
class maxPriorityQueue
{
public:
    virtual ~maxPriorityQueue() {}
    virtual bool empty() const = 0;
        // return true iff queue is empty
    virtual int size() const = 0;
        // return number of elements in queue
    virtual const T& top() = 0;
        // return reference to the max element
    virtual void pop() = 0;
        // remove the top element
    virtual void push(const T& theElement) = 0;
        // add theElement to the queue
};
```

Program 12.1 The abstract class `maxPriorityQueue`

12.3 LINEAR LISTS

The simplest way to represent a max priority queue is as an unordered linear list. Suppose that we have a priority queue with n elements. If Equation 5.1 is used, new elements are most easily inserted at the right end of this list. Hence the insert or *push* time is $\Theta(1)$. A *pop* operation requires a search for the element with largest priority followed by the removal of this element. Since it takes $\Theta(n)$ time to find the largest element in an n -element unordered list, the removal time is $\Theta(n)$. If a chain is used, *pushes* can be performed at the front of the chain in $\Theta(1)$ time. Each *pop* takes $\Theta(n)$ time.

An alternative is to use an ordered linear list. The elements are in nondecreasing order when we use Equation 5.1 and in nonincreasing order when we use an ordered chain. The *pop* time for each representation is $\Theta(1)$, and the *push* time is $O(n)$.

EXERCISES

1. Develop a C++ class for the ADT *maxPriorityQueue*, using an array linear list (see Section 5.3). The *push* time should be $\Theta(1)$, and the *top* and *pop* times should be $O(n)$, where n is the number of elements in the priority queue.
2. Do Exercise 1 using a chain (i.e., use the class *chain* of Section 6.1).
3. Do Exercise 1 using a sorted array linear list. This time the *push* time should be $O(n)$, and the *top* and *pop* times should be $\Theta(1)$.

Hidden page

Hidden page

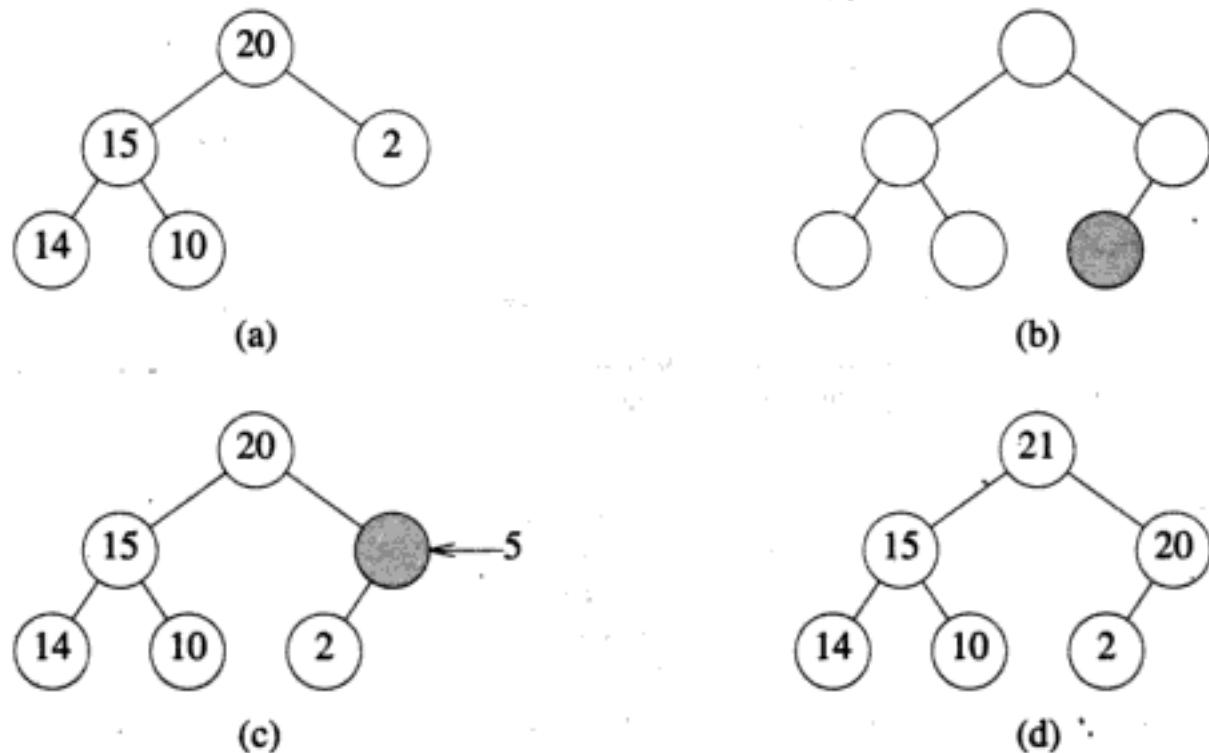


Figure 12.3 Insertion into a max heap

toward the root. At each level we do $\Theta(1)$ work, so we should be able to implement the strategy to have complexity $O(\text{height}) = O(\log n)$.

12.4.3 Deletion from a Max Heap

When an element is to be removed from a max heap, it is taken from the root of the heap. For instance, a removal from the max heap of Figure 12.3(d) results in the deletion of the element 21. Since the resulting max heap has only five elements, the binary tree of Figure 12.3(d) needs to be *reheapedified* (i.e., restructured to correspond to a complete binary tree, which is a min tree with five elements). To do this reheapification, we remove the element in position 6, that is, the element 2. Now we have the right structure (Figure 12.4(a)), but the root is vacant, and the element 2 is not in the heap. If the 2 is put into the root, the resulting binary tree is not a max tree. The larger of the two children of the root (i.e., the element 20) is moved into the root, thereby creating a vacancy in position 3. Since this position has no children, the 2 may be put here. Figure 12.3(a) shows the resulting max heap.

Now suppose we wish to remove 20. Following this removal the heap has the binary tree structure shown in Figure 12.4(b). To get this structure, the 10 is removed from position 5. If we put the 10 into the root, the result is not a max heap. The larger of the two children of the root (15 and 2) is moved to the root,

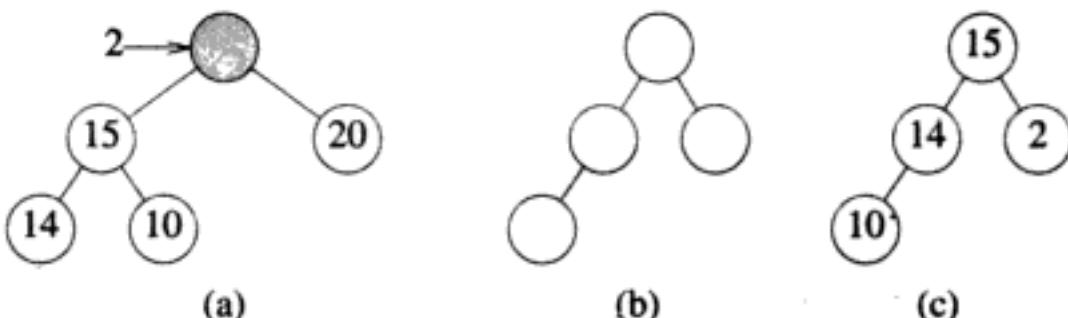


Figure 12.4 Removing the max element from a max heap

and we attempt to insert the 10 into position 2. If the 10 is put here, the result is, again, not a max heap. So the 14 is moved up, and the 10 is put into position 4. The resulting heap is shown in Figure 12.4(c).

The deletion strategy just outlined makes a single pass from the heap root down toward a leaf. At each level $\Theta(1)$ work is done, so we should be able to implement the strategy to have complexity $O(\text{height}) = O(\log n)$.

12.4.4 Max Heap Initialization

In several max heap applications, including the event list of the machine shop scheduling problem of Example 12.2, we begin with a heap that contains $n > 0$ elements. We can construct this initial nonempty heap by performing n inserts into an initially empty heap. The total time taken by these n inserts is $O(n \log n)$. We may initialize the heap in $\Theta(n)$ time by using a different strategy.

Suppose we begin with an array a of n elements. Assume that $n = 10$ and the priority of the elements in $a[1:10]$ is [20, 12, 35, 15, 10, 80, 30, 17, 2, 1]. This array may be interpreted as representing a complete binary tree as shown in Figure 12.5(a). This complete binary tree is not a max heap.

To *heapify* (i.e., make into a max heap) the complete binary tree of Figure 12.5(a), we begin with the last element that has a child (i.e., 10). This element is at position $i = \lfloor n/2 \rfloor$ of the array. If the subtree that is rooted at this position is a max heap, then no work is done here. If this subtree is not a max heap, then we adjust the subtree so that it is a heap. Following this adjustment, we examine the subtree whose root is at $i - 1$, then $i - 2$, and so on until we have examined the root of the entire binary tree, which is at position 1.

Let us try this process on the binary tree of Figure 12.5(a). Initially, $i = 5$. The subtree with root at i is a max heap, as $10 > 1$. Next we examine the subtree rooted at position 4. This subtree is not a max heap, as $15 < 17$. To convert this subtree into a max heap, the 15 and 17 are interchanged to get the tree of Figure 12.5(b). The next subtree examined has its root at position 3. To make this subtree into a max heap, we interchange the 80 and 35. Next we examine the subtree with its

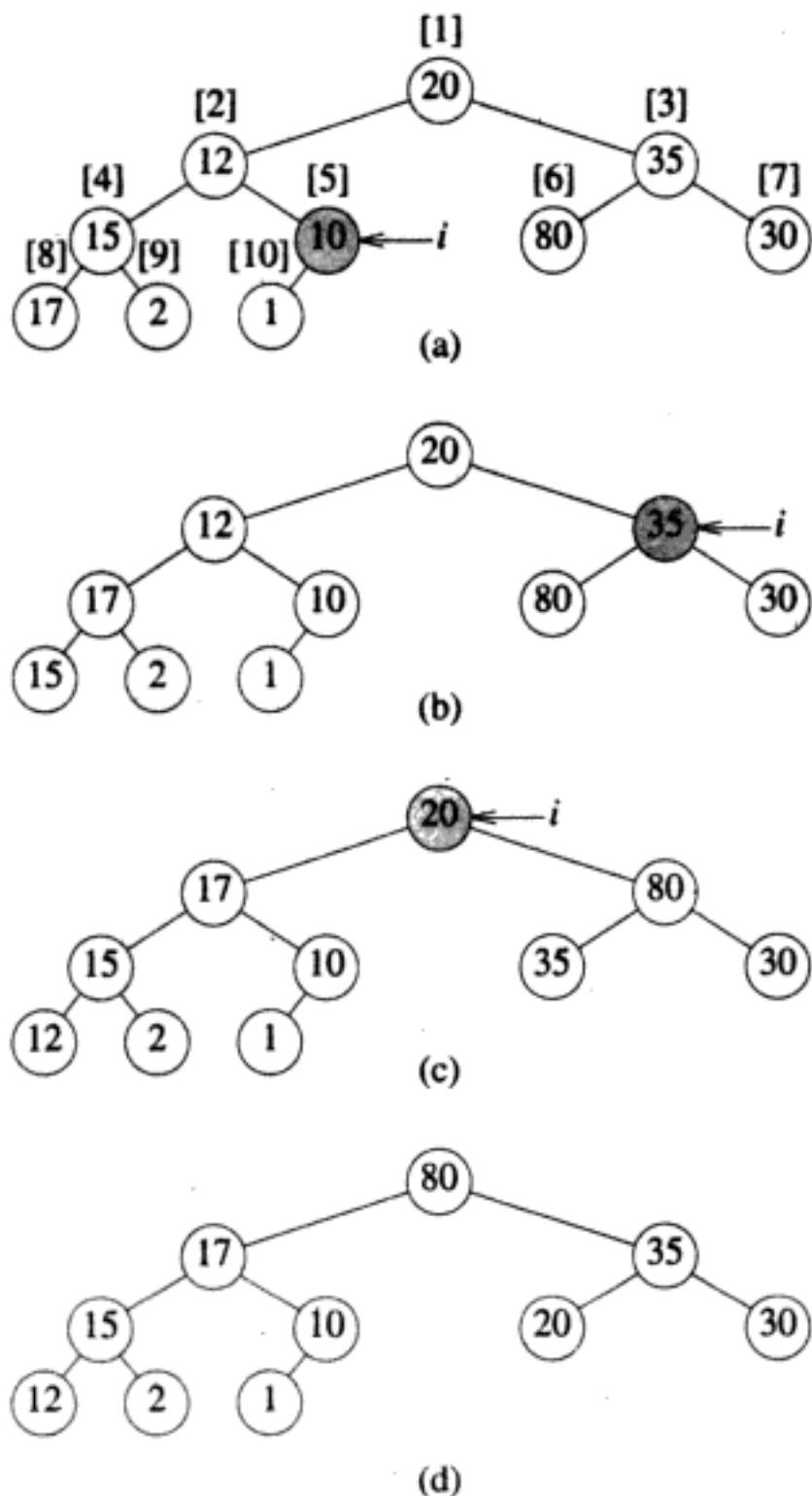


Figure 12.5 Initializing a max heap

root at position 2. From the way the restructuring progresses, the subtrees of this element are guaranteed to be max heaps. So restructuring this subtree into a max heap involves determining the larger of its two children, 17. As $12 < 17$, the 17 should be the root of the restructured subtree. Next we compare 12 with the larger of the two children of position 4. As $12 < 15$, the 15 is moved to position 4. The vacant position 8 has no children, and the 12 is inserted here. The resulting binary tree appears in Figure 12.5(c). Finally, we examine position 1. The subtrees with roots at positions 2 and 3 are max heaps at this time. However, $20 < \max\{17, 80\}$. So the 80 should be in the root of the max heap. When the 80 is moved there, it leaves a vacancy at position 3. Since $20 < \max\{35, 30\}$, position 3 is to be occupied by the 35. The 20 can now occupy position 6. Figure 12.5(d) shows the resulting max heap.

12.4.5 The Class `maxHeap`

The class `maxHeap` implements a max priority queue. Its data members are `heap` (a one-dimensional array of type `T`), `arrayLength` (the capacity of the array `heap`) and `heapSize` (the number of elements in the heap). The `top` method throws a `queueEmpty` exception if the max heap is empty; otherwise, it returns the element `heap[1]`. The code for this method is omitted. The codes for `push` (Program 12.2) and `pop` (Program 12.3) mirror the discussion of Sections 12.4.2 and 12.4.3.

In the code to insert an element into the max heap, we start by verifying that we have space in the array `heap` to accommodate the new element. If not, we double the capacity of this array. Next, we start `currentNode` at the position, `heapSize`, of the newly created leaf of the heap `heap` and traverse the path from here to the root. We essentially bubble `theElement` up the tree until it reaches its proper place. At each positioning of `currentNode`, we check to see whether we are at the root (`currentNode = 1`) or whether the insertion of the new element `theElement` at `currentNode` does not violate the max tree property (`theElement ≤ heap[currentNode/2]`). If either of these conditions holds, we can put `theElement` at position `currentNode`. Otherwise, we enter the body of the `while` loop, move the element at `currentNode/2` down to `currentNode`, and advance `currentNode` up to its parent (`currentNode/2`). For a max heap with n elements (i.e., `size = n`), the number of iterations of the `while` loop is $O(\text{height}) = O(\log n)$, and each iteration takes $\Theta(1)$ time. Therefore, the complexity of `push` (exclusive of the time needed to resize the array `heap`) is $O(\log n)$.

For a `pop` operation the maximum element, which is in the root (`heap[1]`), is deleted; the element in the last heap position (`heap[size]`) is saved in `lastElement`; and the heap size (`size`) is reduced by 1. In the `while` loop we reheapify the array. The reheapification process requires us to find the proper place to reinsert `lastElement`. The search for this proper place begins at the root and proceeds down the heap. For an n -element heap, the number of iterations of the `while` loop is $O(\log n)$, and each iteration takes $\Theta(1)$ time. Therefore, the overall complexity of `pop` is $O(\log n)$. Notice that the code works correctly even when the size of the

```

template<class T>
void maxHeap<T>::push(const T& theElement)
{// Add theElement to heap.

// increase array length if necessary
if (heapSize == arrayLength - 1)
{// double array length
    changeLength1D(heap, arrayLength, 2 * arrayLength);
    arrayLength *= 2;
}

// find place for theElement
// currentNode starts at new leaf and moves up tree
int currentNode = ++heapSize;
while (currentNode != 1 && heap[currentNode / 2] < theElement)
{
    // cannot put theElement in heap[currentNode]
    heap[currentNode] = heap[currentNode / 2]; // move element down
    currentNode /= 2;                         // move to parent
}

heap[currentNode] = theElement;
}

```

Program 12.2 Inserting an element into a max heap

heap following the deletion is 0. In this case the while loop is not entered, and a redundant assignment to position 1 of the heap is made.

The method `initialize` (Program 12.4) heapifies the array `theHeap` by first assigning `theHeap` to `heap`. `theSize` is the number of elements in `theHeap`. In the for loop of Program 12.4, we begin at the last node in the binary tree interpretation of the array `heap` (now equivalent to the array `theHeap`) that has a child and work our way to the root. At each positioning of the variable `root`, the embedded while loop ensures that the subtree rooted at `root` is a max heap. Notice the similarity between the body of the for loop and the code for `pop` (Program 12.3).

Complexity of initialize

If the number of elements is n (i.e., `theSize = n`), each iteration of the for loop of `initialize` (Program 12.4) takes $O(\log n)$ time and the number of iterations is $n/2$. So the complexity of `initialize` is $O(n \log n)$. Recall that the big oh notation provides only an upper bound on the complexity of an algorithm. Consequently,

```
template<class T>
void maxHeap<T>::pop()
{// Remove max element.
    // if heap is empty return null
    if (heapSize == 0) // heap empty
        throw queueEmpty();

    // Delete max element
    heap[1].~T();

    // Remove last element and reheapify
    T lastElement = heap[heapSize--];

    // find place for lastElement starting at root
    int currentNode = 1,
        child = 2; // child of currentNode
    while (child <= heapSize)
    {
        // heap[child] should be larger child of currentNode
        if (child < heapSize && heap[child] < heap[child + 1])
            child++;

        // can we put lastElement in heap[currentNode]?
        if (lastElement >= heap[child])
            break; // yes

        // no
        heap[currentNode] = heap[child]; // move child up
        currentNode = child; // move down a level
        child *= 2;
    }
    heap[currentNode] = lastElement;
}
```

Program 12.3 Removing the max element from a max heap

the complexity of `initialize` could be better than suggested by this upper bound. A more careful analysis allows us to conclude that the complexity is actually $\Theta(n)$.

Each iteration of the `while` loop of `initialize` takes $O(h_i)$ time where h_i is the height of the subtree with root i . The complete binary tree `heap[1 : n]` has height $h = \lceil \log_2(n+1) \rceil$. It has at most 2^{j-1} nodes at level j . Hence at most 2^{j-1} of the

Hidden page

$$\sum_{k=1}^m \frac{k}{2^k} = 2 - \frac{m+2}{2^m} \quad (12.1)$$

Since the `for` loop goes through $n/2$ iterations, the complexity is also $\Omega(n)$. Combining these two bounds, we get $\Theta(n)$ as the complexity of `initialize`.

12.4.6 Heaps and the STL

The STL class `priority_queue` employs a vector-based heap to implement a max priority queue. However, since the STL class permits the user to specify the function used to compare priorities, the class may also be used for min priority queues.

EXERCISES

6. Consider the array `theHeap` = [-, 3, 5, 6, 7, 20, 8, 2, 9, 12, 15, 30, 17].
 - (a) Draw the corresponding complete binary tree.
 - (b) Heapify the tree by using the method of Program 12.4. Show the result in both tree and array format.
 - (c) Now insert the elements 15, 20, and 45 (in this order) using the bubbling up process of Program 12.2. Show the max heap following each insert.
 - (d) Perform four remove max operations on the max heap of part (c). Use the remove method of Program 12.3. Show the max heap following each remove.
7. Do Exercise 6 beginning with the array [-, 10, 2, 7, 6, 5, 9, 12, 35, 22, 15, 1, 3, 4].
8. Do Exercise 6 beginning with the array [-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 22, 35].
9. Do Exercise 6 under the assumption that you are to have a min heap. The initial array is [-, 30, 17, 20, 15, 10, 12, 5, 7, 8, 5, 2, 9]. For part (b) insert 1, 10, 6, and 4. For part (d) perform three remove min operations.
10. Use induction on m to prove Equation 12.1.
11. Write a copy constructor for `maxHeap`. Test the correctness of your code.
12. Extend the class `maxHeap` by adding a public member `changeMax(newElement)` that replaces the current maximum element with `newElement`. `newElement` may have a value that is either smaller or larger than the current priority of the element in `heap[1]`. Your code should follow a downward path from the root, as is done in the `pop` method. The complexity of your code should be $O(\log n)$ where n is the number of elements in the max heap. Show that this is the case. Test the correctness of your code.

13. Extend the class `maxHeap` by adding a public member `remove(i)` that removes and returns the element in `heap[i]`. The complexity of your code should be $O(\log n)$ where n is the number of elements in the max heap. Show that this is the case. Test the correctness of your code.
14. Develop an iterator for the class `maxHeap`. You may enumerate the elements in any order. The time taken to enumerate the elements of an n -element max heap should be $O(n)$. Show that this is the case. Test the correctness of your code.
15. Since the element `lastElement` that is reinserted into the max heap during a `pop` (see Program 12.3) was removed from the bottom of the heap, we expect to reinsert it near the bottom. Write a new version of `pop` in which the vacancy in the root is first moved down to a leaf, and then the place for `lastElement` is determined making an upward pass from this leaf. Experiment with the new code to see whether it works faster than the old one.
16. Rewrite the methods of `maxHeap` under the following assumptions:
 - (a) When a heap is created, the creator provides two elements `maxElement` and `minElement`; no element in the heap is larger than `maxElement` or smaller than `minElement`.
 - (b) A heap with n elements requires an array `heap[0:2n+1]`.
 - (c) The n elements are stored in `heap[1:n]` as described in this section.
 - (d) `maxElement` is stored in `heap[0]`.
 - (e) `minElement` is stored in `heap[n+1:2n+1]`.

These assumptions should simplify the codes for `push` and `pop`. Conduct experiments to compare the implementation of this section with the one of this exercise.

17. A d -heap is a complete tree whose degree is d . Develop the concrete class `maxDHeap` which extends the abstract class `maxPriorityQueue` using a d -heap. Compare the performance of your implementation with that of `maxHeap` for $d = 2, 3$, and 4 .
18. Do Exercise 12 for the class `maxDHeap` (see Exercise 17).

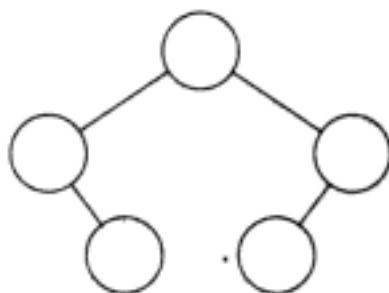
12.5 LEFTIST TREES

12.5.1 Height- and Weight-Biased Min and Max Leftist Trees

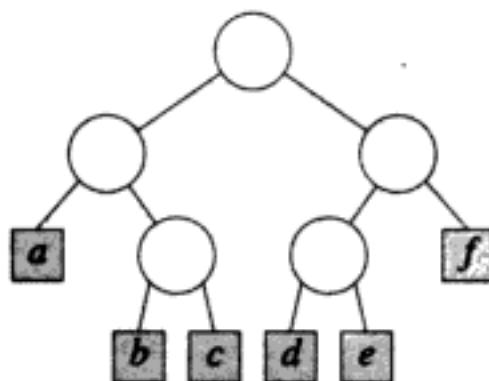
The heap structure of Section 12.4 is an example of an **implicit data structure**. The complete binary tree representing the heap is stored implicitly (i.e., there are

no explicit pointers or other explicit data from which the structure may be deduced) in an array. Since no explicit structural information is stored, the representation is very space efficient; in fact, there is no space overhead. Despite the heap structure being both space and time efficient, it is not suitable for all applications of priority queues. In particular, applications in which we wish to meld (i.e., combine or blend) pairs of priority queues, as well as those in which we have multiple queues of varying size, require a different data structure. Leftist tree structures are suitable for these applications.

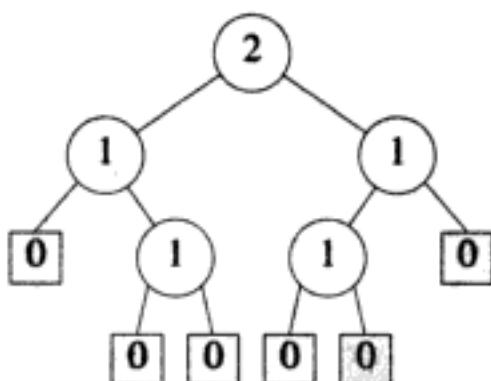
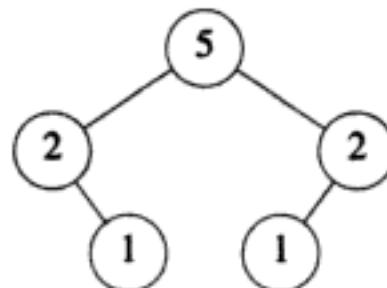
Consider a binary tree in which a special node called an **external node** replaces each empty subtree. The remaining nodes are called **internal nodes**. A binary tree with external nodes added is called an **extended binary tree**. Figure 12.6(a) shows a binary tree. Its corresponding extended binary tree is shown in Figure 12.6(b). The external nodes appear as shaded boxes. These nodes have been labeled *a* through *f* for convenience.



(a) A binary tree



(b) Extended binary tree

(c) *s* values(d) *w* values**Figure 12.6** *s* and *w* values

Let $s(x)$ be the length of a shortest path from node x to an external node in its

subtree. From the definition of $s(x)$, it follows that if x is an external node, its s value is 0. Furthermore, if x is an internal node, its s value is

$$\min\{s(L), s(R)\} + 1$$

where L and R are, respectively, the left and right children of x . The s values for the nodes of the extended binary tree of Figure 12.6(b) appear in Figure 12.6(c).

Definition 12.3 A binary tree is a height-biased leftist tree (HBLT) iff at every internal node, the s value of the left child is greater than or equal to the s value of the right child. ■

The binary tree of Figure 12.6(a) is not an HBLT. To see this, consider the parent of the external node a . The s value of its left child is 0, while that of its right is 1. All other internal nodes satisfy the requirements of the HBLT definition. By swapping the left and right subtrees of the parent of a , the binary tree of Figure 12.6(a) becomes an HBLT.

Theorem 12.1 Let x be any internal node of an HBLT.

- (a) The number of nodes in the subtree with root x is at least $2^{s(x)} - 1$.
- (b) If the subtree with root x has m nodes, $s(x)$ is at most $\log_2(m + 1)$.
- (c) The length of the right-most path from x to an external node (i.e., the path obtained by beginning at x and making a sequence of right-child moves) is $s(x)$.

Proof From the definition of $s(x)$, it follows that there are no external nodes on the $s(x) - 1$ levels immediately below node x (as otherwise the s value of x would be less). The subtree with root x has exactly one node on the level at which x is, two on the next level, four on the next, ..., and $2^{s(x)-1}$ nodes $s(x) - 1$ levels below x . The subtree may have additional nodes at levels more than $s(x) - 1$ below x . Hence the number of nodes in the subtree x is at least $\sum_{i=0}^{s(x)-1} 2^i = 2^{s(x)} - 1$. Part (b) follows from (a). Part (c) follows from the definition of s and the fact that, in an HBLT, the s value of the left child of a node is always greater than or equal to that of the right child. ■

Definition 12.4 A max HBLT is an HBLT that is also a max tree. A min HBLT is an HBLT that is also a min tree. ■

The max trees of Figure 12.1 as well as the min trees of Figure 12.2 are also HBLTs; therefore, the trees of Figure 12.1 are max HBLTs, and those of Figure 12.2 are min HBLTs. A max priority queue may be represented as a max HBLT, and a min priority queue may be represented as a min HBLT.

We arrive at another variety of leftist tree by considering the number of nodes in a subtree, rather than the length of a shortest root to external node path. Define the weight $w(x)$ of node x to be the number of internal nodes in the subtree with root x . Notice that if x is an external node, its weight is 0. If x is an internal node, its weight is 1 more than the sum of the weights of its children. The weights of the nodes of the binary tree of Figure 12.6(a) appear in Figure 12.6(d)

Definition 12.5 A binary tree is a weight-biased leftist tree (**WBLT**) iff at every internal node the w value of the left child is greater than or equal to the w value of the right child. A max (min) WBLT is a max (min) tree that is also a WBLT. ■

As was the case for HBLTs, the length of the right-most path in a WBLT that has m nodes is at most $\log_2(m+1)$. Using either WBLTs or HBLTs, we can perform the priority queue operations find, insert, and delete in the same asymptotic time as heaps take. Like heaps, WBLTs and HBLTs may be initialized in linear time. Two priority queues represented as WBLTs or HBLTs can be melded into one in logarithmic time. When priority queues are represented as heaps, they cannot be melded in logarithmic time.

The way in which finds, inserts, deletes, melds, and initializations are done in WBLTs and HBLTs is similar. Consequently, we describe these operations for HBLTs only and leave the adaptation of these methods to WBLTs as an exercise (Exercise 24).

12.5.2 Insertion into a Max HBLT

The insertion operation for max HBLTs may be performed by using the max HBLT meld operation. Suppose we are to insert an element x into the max HBLT H . If we create a max HBLT with the single element x and then meld this max HBLT and H , the resulting max HBLT will include all elements in H as well as the element x . Hence an insertion may be performed by creating a new max HBLT with just the element that is to be inserted and then melding this max HBLT and the original.

12.5.3 Deletion from a Max HBLT

The max element is in the root. If the root is deleted, two max HBLTs, the left and right subtrees of the root, remain. By melding together these two max HBLTs, we obtain a max HBLT that contains all elements in the original max HBLT other than the deleted max element. So the delete max operation may be performed by deleting the root and then melding its two subtrees.

12.5.4 Melding Two Max HBLTs

Since the length of the right-most path of an HBLT with n elements is $O(\log n)$, a meld algorithm that traverses only the right-most paths of the HBLTs being

melded, spending $O(1)$ time at each node on these two paths, will have complexity logarithmic in the number of elements in the resulting HBLT. With this observation in mind, we develop a meld algorithm that begins at the roots of the two HBLTs and makes right-child moves only.

The meld strategy is best described using recursion. Let A and B be the two max HBLTs that are to be melded. If one is empty, then we may use the other as the result. So assume that neither is empty. To perform the meld, we compare the elements in the two roots. The root with the larger element becomes the root of the melded HBLT. Ties may be broken arbitrarily. Suppose that A has the larger root and that its left subtree is L . Let C be the max HBLT that results from melding the right subtree of A and the max HBLT B . The result of melding A and B is the max HBLT that has A as its root and L and C as its subtrees. If the s value of L is smaller than that of C , then C is the left subtree. Otherwise, L is.

Example 12.3 Consider the two max HBLTs of Figure 12.7(a). The s value of a node is shown outside the node, while the element value is shown inside. When drawing two max HBLTs that are to be melded, we will always draw the one with larger root value on the left. Ties are broken arbitrarily. Because of this convention, the root of the left HBLT always becomes the root of the final HBLT. Also, we will shade the nodes of the HBLT on the right.

Since the right subtree of 9 is empty, the result of melding this subtree of 9 and the tree with root 7 is just the tree with root 7. We make the tree with root 7 the right subtree of 9 temporarily to get the max tree of Figure 12.7(b). Since the s value of the left subtree of 9 is 0 while that of its right subtree is 1, the left and right subtrees are swapped to get the max HBLT of Figure 12.7(c).

Next consider melding the two max HBLTs of Figure 12.7(d). The root of the left subtree becomes the root of the result. When the right subtree of 10 is melded with the HBLT with root 7, the result is just this latter HBLT. If this HBLT is made the right subtree of 10, we get the max tree of Figure 12.7(e). Comparing the s values of the left and right children of 10, we see that a swap is not necessary.

Now consider melding the two max HBLTs of Figure 12.7(f). The root of the left subtree is the root of the result. We proceed to meld the right subtree of 18 and the max HBLT with root 10. The two max HBLTs being melded are the same as those melded in Figure 12.7(d). The resultant max HBLT (Figure 12.7(e)) becomes the right subtree of 18, and the max tree of Figure 12.7(g) results. Comparing the s values of the left and right subtrees of 18, we see that these subtrees must be swapped. Swapping results in the max HBLT of Figure 12.7(h).

As a final example, consider melding the two max HBLTs of Figure 12.7(i). The root of the left max HBLT becomes the root of the result. We proceed to meld the right subtree of 40 and the max HBLT with root 18. These max HBLTs were melded in Figure 12.7(f). The resultant max HBLT (Figure 12.7(g)) becomes the right subtree of 40. Since the left subtree of 40 has a smaller s value than the right has, the two subtrees are swapped to get the max HBLT of Figure 12.7(k). Notice that when melding the max HBLTs of Figure 12.7(i), we first move to the right

Hidden page

child of 40, then to the right child of 18, and finally to the right child of 10. All moves follow the right-most paths of the initial max HBLTs. ■

12.5.5 Initialization

It takes $O(n \log n)$ time to initialize a max HBLT with n elements by inserting these elements into an initially empty max HBLT one at a time. To get a linear time initialization algorithm, we begin by creating n max HBLTs with each containing one of the n elements. These n max HBLTs are placed on a FIFO queue. Then max HBLTs are deleted from this queue in pairs, melded, and added to the end of the queue until only one max HBLT remains.

Example 12.4 We wish to create a max HBLT with the five elements 7, 1, 9, 11, and 2. Five single-element max HBLTs are created and placed in a FIFO queue. The first two, 7 and 1, are deleted from the queue and melded. The result (Figure 12.8(a)) is added to the queue. Next the max HBLTs 9 and 11 are deleted from the queue and melded. The result appears in Figure 12.8(b). This max HBLT is added to the queue. Now the max HBLT 2 and that of Figure 12.8(a) are deleted from the queue and melded. The resulting max HBLT (Figure 12.8(c)) is added to the queue. The next pair to be deleted from the queue consists of the max HBLTs of Figures 12.8(b) and (c). These HBLTs are melded to get the max HBLT of Figure 12.8(d). This max HBLT is added to the queue. The queue now has just one max HBLT, and we are done with the initialization. ■

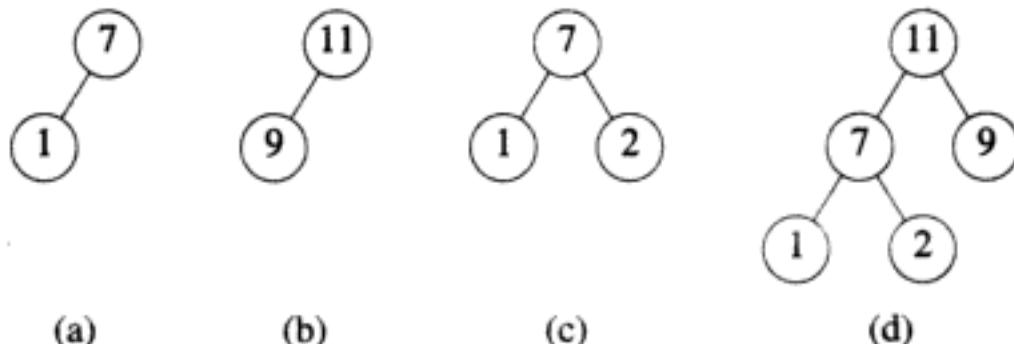


Figure 12.8 Initializing a max HBLT

12.5.6 The Class `maxHblt`

The data type of each node of a max HBLT is `binaryTreeNode<pair<int,T>>`, where `binaryTreeNode` is as in Program 11.1; the first component of the pair is the s value of the node; and the second component is the priority queue element. Our class

`maxHblt`, which implements a max HBLT, extends the class `linkedBinaryTree` (Program 11.9). The code for the elementary methods `empty`, `size`, and `top` is similar to that for the corresponding methods of `maxHeap`.

Since the `push`, `pop`, and `initialize` methods of `maxHblt` use the `meld` operation, let us examine the `meld` operation first. The public method `meld(maxHblt< T>& theMaxHblt)` melds the two max HBLTs `*this` and `theMaxHblt`. Upon completion, `*this` is the resulting melded max HBLT. This method accomplishes its task by invoking a private method `meld(x, y)` (Program 12.5) that recursively melds the max HBLTs whose roots are `x` and `y`. Upon termination, `x` is the root of the resulting max HBLT.

The private `meld` method given in Program 12.5 begins by handling the special case when at least one of the trees being melded is empty. When neither tree is empty, we make sure that node `x` has the larger element (i.e., the tree with larger root is on the left). If the element in `x` is not larger than that in `y`, `x` and `y` are swapped. Next the right subtree of `x` and the max HBLT with root `y` are melded recursively. Following this `meld`, `x` is the root of a max tree whose left and right subtrees may need to be swapped so as to ensure that the entire tree is, in fact, a max HBLT. This swapping is done, if necessary, and the `s` value of `x` is computed.

To insert `theElement` into a max HBLT, the code of Program 12.6 creates a max HBLT with the single element `theElement` and then uses the private method `meld` to meld this tree and the original one.

The code for `pop` (Program 12.6) throws a `queueEmpty` exception in case the max HBLT is empty. When the max HBLT is not empty, the root is deleted and the left and right subtrees of the root are melded to get the desired max HBLT.

The max HBLT initialization code is given in Program 12.7. An array FIFO queue holds the intermediate max HBLTs created by the initialization algorithm. In the first `for` loop, `size` single-element max HBLTs are created and added to an initially empty queue. In the next `for` loop, pairs of max HBLTs are popped from the queue, melded, and the result added to the queue. When this `for` loop terminates, the queue contains a single max HBLT (provided `size > 0`), which includes all `size` elements.

Complexity Analysis

The complexity of the `top` method is $\Theta(1)$. The complexity of `push`, `pop`, and the public method `meld` is the same as that of the private method `meld`. Since this private method moves only to right subtrees of the trees with roots `x` and `y` that are being melded, the complexity of this private method is $O(s(x) + s(y))$. $s(x)$ and $s(y)$ are at most $\log_2(m + 1)$ and $\log_2(n + 1)$ where m and n are, respectively, the number of elements in the max HBLTs with roots `x` and `y`. As a result, the complexity of the private method `meld` is $O(\log m + \log n) = O(\log(mn))$.

For the complexity analysis of `initialize`, assume, for simplicity, that $n = \text{size}$ is a power of 2. The first $n/2$ melds involve max HBLTs with one element each, the next $n/4$ melds involve max HBLTs with two elements each; the next $n/8$

```

template<class T>
void maxHblt<T>::meld(binaryTreeNode<pair<int, T>>*&x,
                        binaryTreeNode<pair<int, T>>*&y)
// Meld leftist trees with roots *x and *y.
// Return pointer to new root in x.
if (y == NULL) // y is empty
    return;
if (x == NULL) // x is empty
    {x = y; return;}

// neither is empty, swap x and y if necessary
if (x->element.second < y->element.second)
    swap(x, y);

// now x->element.second >= y->element.second

meld(x->rightChild,y);

// swap subtrees of x if necessary and set x->element.first
if (x->leftChild == NULL)
{// left subtree empty, swap the subtrees
    x->leftChild = x->rightChild;
    x->rightChild = NULL;
    x->element.first = 1;
}
else
{// swap only if left subtree has smaller s value
    if (x->leftChild->element.first < x->rightChild->element.first)
        swap(x->leftChild, x->rightChild);
    // update s value of x
    x->element.first = x->rightChild->element.first + 1;
}
}

```

Program 12.5 Melding two leftist trees

melds are with trees that have four elements each; and so on. The time needed to meld two trees with 2^i elements each is $O(i + 1)$, and so the total time taken by `initialize` is

$$O(n/2 + 2 * (n/4) + 3 * (n/8) + \dots) = O(n \sum \frac{i}{2^i}) = O(n)$$

Hidden page

```

template<class T>
void maxHblt<T>::initialize(T* theElements, int theSize)
{// Initialize hblt with theElements[1:theSize].
    arrayQueue<binaryTreeNode<pair<int,T>>> q(theSize);
    erase(); // make *this empty

    // initialize queue of trees
    for (int i = 1; i <= theSize; i++)
        // create trees with one node each
        q.push(new binaryTreeNode<pair<int,T>>
            (pair<int,T>(1, theElements[i])));

    // repeatedly meld from queue
    for (i = 1; i <= theSize - 1; i++)
    {// pop and meld two trees from queue
        binaryTreeNode<pair<int,T>> *b = q.front();
        q.pop();
        binaryTreeNode<pair<int,T>> *c = q.front();
        q.pop();
        meld(b,c);
        // put melded tree on queue
        q.push(b);
    }

    if (theSize > 0)
        root = q.front();
    treeSize = theSize;
}

```

Program 12.7 Initializing a max HBLT

EXERCISES

19. Consider the array `theElements` = [-, 3, 5, 6, 7, 20, 8, 2, 9, 12, 15, 30, 17].
- Draw the max leftist tree created by Program 12.7.
 - Now insert the elements 10, 18, 11, and 4 (in this order) using the `insert` method of Program 12.6. Show the max leftist tree following each insert.
 - Perform three remove max operations on the max leftist tree of part (c). Use the `remove` method of Program 12.6. Show the max leftist tree following each remove.

Hidden page

12.6 APPLICATIONS

12.6.1 Heap Sort

You might have already noticed that a heap can be used to sort n elements in $O(n \log n)$ time. We begin by initializing a max heap with the n elements to be sorted. Then we extract (i.e., delete) elements from the heap one at a time. The elements appear in nonincreasing order. The initialization takes $O(n)$ time, and each deletion takes $O(\log n)$ time. So the total time is $O(n \log n)$. This time is better than the $O(n^2)$ time taken by the sort methods of Chapter 2.

The sort method that results from the preceding strategy is called **heap sort**. Its implementation, which appears in Program 12.8, employs the max heap method `deactivateArray` that sets `maxHeap<T>::heap` to `NULL`. This is necessary because `maxHeap<T>::initialize` sets `maxHeap<T>::heap` to the element array `a` and when we exit the heap sort function, the max heap destructor deletes `maxHeap<T>::heap`. So to prevent the deletion of the element array `a`, it is necessary to invoke `deactivateArray`.

```
template <class T>
void heapSort(T a[], int n)
{// Sort a[1:n] using the heap sort method.
    // create a max heap of the elements
    maxHeap<T> heap(1);
    heap.initialize(a, n);

    // extract one by one from the max heap
    for (int i = n - 1; i >= 1; i--)
    {
        T x = heap.top();
        heap.pop();
        a[i+1] = x;
    }

    // save array a from heap destructor
    heap.deactivateArray();
}
```

Program 12.8 Sort $a[1:n]$ using heap sort

Figure 12.9 shows the progress of the `for` loop of Program 12.8 for the first few values of i . This loop begins with the max heap of Figure 12.5(d). Circles denote array positions that are part of the max heap, and squares denote array positions that have their sorted values.

Hidden page

Hidden page

Our task is to write a program that constructs a minimum-finish-time m -machine schedule for a given set of n jobs. Constructing such a schedule is very hard. In fact, no one has ever developed a polynomial time algorithm (i.e., an algorithm whose complexity is $O(n^k m^l)$ for any constants k and l) to construct a minimum-finish-time schedule.

The scheduling problem we have just defined is a member of the infamous class of NP-hard (NP stands for **nondeterministic polynomial**) problems. The NP-hard and NP-complete problem classes contain problems for which no one has developed a polynomial-time algorithm. The problems in the class NP-complete are decision problems. That is, for each problem instance the answer is either yes or no. Our machine-scheduling problem is not a decision problem, as the answer for each instance is an assignment of jobs to machines such that the finish time is minimum. We may formulate a related machine-scheduling problem in which, in addition to the tasks and machines, we are given a time $TMin$ and are asked to determine whether or not there is a schedule with finish time $TMin$ or less. For this related problem, the answer to each instance is either yes or no. This related problem is a decision problem that is NP-complete. NP-hard problems may or may not be decision problems.

Thousands of problems of practical interest are NP-hard or NP-complete. If anyone discovers a polynomial-time algorithm for an NP-hard or NP-complete problem, then he/she would have simultaneously discovered a way to solve all NP-complete problems in polynomial time. Although we are unable to prove that NP-complete problems cannot be solved in polynomial time, common wisdom very strongly suggests that this is the case. As a result, optimization problems that are NP-hard are often solved by **approximation algorithms**. Although approximation algorithms do not guarantee to obtain optimal solutions, they guarantee solutions “close” to optimal.

In the case of our scheduling problem, we can generate schedules whose lengths are at most $4/3 - 1/(3m)$ of optimal by employing a simple scheduling strategy called **longest processing time** (LPT). In LPT, jobs are assigned to machines in descending order of their processing time requirements t_i . When a job is being assigned to a machine, it is assigned to the machine that becomes idle first. Ties are broken arbitrarily.

For the job set example in Figure 12.10, we may construct an LPT schedule by first sorting the jobs into descending order of processing times. The job order is (4, 2, 5, 6, 3, 7, 1). First, job 4 is assigned to a machine. Since all three machines become available at time 0, job 4 may be assigned to any machine. Suppose we assign it to machine 1. Now machine 1 is unavailable until time 16. Job 2 is next assigned; we can assign it to either machine 2 or 3, as both become available at the same time (i.e., time 0). Assume that we assign job 2 to machine 2. Now machine 2 is unavailable until time 14. Next we assign job 5 to machine 3 from time 0 to time 6. Job 6 is to be assigned next. The first available machine is machine 3. It becomes available at time 6. Following the assignment of job 6 from time 6 to time

Hidden page

```
void makeSchedule(jobNode a[], int n, int m)
{// Output an m machine LPT schedule for the jobs a[1:n].
    if (n <= m)
    {
        cout << "Schedule each job on a different machine." << endl;
        return;
    }

    heapSort(a, n); // in ascending order

    // initialize m machines and the min heap
    minHeap<machineNode> machineHeap(m);
    for (int i = 1; i <= m; i++)
        machineHeap.push(machineNode(i, 0));

    // construct schedule
    for (int i = n; i >= 1; i--)
    {// schedule job i on first free machine
        machineNode x = machineHeap.top();
        machineHeap.pop();
        cout << "Schedule job " << a[i].id
            << " on machine " << x.id << " from " << x.avail
            << " to " << (x.avail + a[i].time) << endl;
        x.avail += a[i].time; // new available time for this machine
        machineHeap.push(x);
    }
}
```

Program 12.9 Output an m machine LPT schedule for a[1:n]

Complexity Analysis of makeSchedule

When $n \leq m$, `makeSchedule` takes $\Theta(1)$ time. When $n > m$, the heap sort takes $O(n \log n)$ time. The heap initialization takes $O(m)$ time, even though we are doing m inserts, because all elements have the same value; therefore, each insert actually takes only $\Theta(1)$ time. In the second `for` loop, n `top`, n `pop` and n `push` operations are performed. Each `top` operation takes $O(1)$ time and each `pop` and `push` takes $O(\log m)$ time. So the second `for` loop takes $O(n \log m)$ time. The total time is therefore $O(n \log n + n \log m) = O(n \log n)$ (as $n > m$).

12.6.3 Huffman Codes

In Section 10.6 we developed a text compressor based on the LZW method. This method relies on the recurrence of substrings in a text. Another approach to text compression, **Huffman codes**, relies on the relative frequency with which different symbols appear in a piece of text. Suppose our text is a string that comprises the characters *a*, *u*, *x*, and *z*. If the length of this string is 1000, then storing it as 1000 one-byte characters will take 1000 bytes (or 8000 bits) of space. If we encode the symbols in the string using 2 bits per symbol ($00 = a$, $01 = x$, $10 = u$, $11 = z$), then the 1000 symbols can be represented with 2000 bits of space. We also need space for the code table, which may be stored in following format:

number of table entries, code 1, symbol 1, code 2, symbol 2, ...

Eight bits are adequate for the number of entries and for each of the symbols. Each code is of size $\lceil \log_2 (\text{number of table entries}) \rceil$ bits. For our example the code table may be saved in $5 * 8 + 4 * 2 = 48$ bits. The compression ratio is $8000/2048 = 3.9$.

Using the above 2 bits per symbol encoding, the string *aaxuazz* is encoded as 00000110000111. The code for each symbol has the same number of bits (i.e., 2). By picking off pairs of bits from the coded string from left to right and using the code table, we can obtain the original string.

In the string *aaxuazz*, the symbol *a* occurs three times. The number of occurrences of a symbol is called its **frequency**. The frequencies of the symbols *a*, *x*, *u*, and *z* in the sample string are 3, 2, 1, and 1, respectively. When there is significant variation in the frequencies of different symbols, we can reduce the size of the coded string by using variable-length codes. If we use the codes ($0 = a$, $10 = x$, $110 = u$, $111 = z$), the encoded version of *aaxuazz* is 0010110010111. The length of this encoded version is 13 bits compared to 14 bits using the 2 bits per symbol code! The difference is more dramatic when the spread in frequencies is greater. If the frequencies of the four symbols are (996, 2, 1, 1), then the 2 bits per symbol code results in an encoding that is 2000 bits long, whereas the variable-length code results in an encoding that is 1006 bits.

But how do we decode the encoded string? When each code is 2 bits long, decoding is easy—just pick off every pair of bits and use the code table to determine what these 2 bits stand for. With variable-length codes, we do not know how many bits to pick off. The string *aaxuazz* was coded as 0010110010111. When decoding this code from left to right, we need to know whether the code for the first symbol is 0, 00, or 001. Since we have no codes that begin with 00, the first code must be 0. This code is decoded using the code table to get *a*. The next code is 0, 01, or 010. Again, because no codes begin with 01, the code must be 0. Continuing in this way, we are able to decode the encoded bit string.

What makes this decoding method work? If we examine the four codes in use (0, 10, 110, 111), we observe that no code is a prefix of another. Consequently,

when examining the coded bit string from left to right, we can get a match with exactly one code.

We may use extended binary trees (see Section 12.5.1 for a definition) to derive a special class of variable-length codes that satisfy this prefix property. This class of codes is called **Huffman codes**.

The root-to-external-node paths in an extended binary tree may be coded by using 0 to represent a move to a left subtree and 1 to represent a move to a right subtree. In Figure 12.6(b) the path from the root to the external node b gets the code 010. The codes for the paths to the nodes (a, b, c, d, e, f) are (00, 010, 011, 100, 101, 11). Notice that since no root-to-external-node path is a prefix of another such path, no path code is a prefix of another path code. Therefore, these codes may be used to encode the symbols a, b, \dots, f , respectively. Let S be a string made up of these symbols and let $F(x)$ be the frequency of the symbol $x \in \{a, b, c, d, e, f\}$. If S is encoded using these codes, the encoded string has a length

$$2 * F(a) + 3 * F(b) + 3 * F(c) + 3 * F(d) + 3 * F(e) + 2 * F(f)$$

For an extended binary tree with external nodes labeled $1, \dots, n$, the length of the encoded string is

$$WEP = \sum_{i=1}^n L(i) * F(i)$$

where $L(i)$ is the length of the path (i.e., number of edges on the path) from the root to the external node labeled i . WEP is called the **weighted external path length** of the binary tree. To minimize the length of the coded string, we must use codes from a binary tree whose external nodes correspond to the symbols in the string being encoded and whose WEP is minimum. A binary tree with minimum WEP for a given set of frequencies (weights) is called a **Huffman tree**.

To encode a string (or piece of text) using Huffman codes, we need to

1. Determine the different symbols in the string and their frequencies.
2. Construct a binary tree with minimum WEP (i.e., a Huffman tree). The external nodes of this tree are labeled by the symbols in the string, and the weight of each external node is the frequency of the symbol that is its label.
3. Traverse the root-to-external-node paths and obtain the codes.
4. Replace the symbols in the string by their codes.

To facilitate decoding, we need to save a table that contains the symbol to code mapping or a table that contains the frequency of each symbol. In the latter case the Huffman codes can be reconstructed using the method for (2). We will elaborate on step (2) only.

A Huffman tree can be constructed by beginning with a collection of binary trees, each having just an external node. Each external node represents a different string symbol and has a **weight** equal to the frequency of this symbol. Then we repeatedly select two binary trees of lowest weight (ties are broken arbitrarily) from the collection and combine them into one by making them subtrees of a new root node. The weight of the newly formed tree is the sum of the weights of the constituent subtrees. The process terminates when only one tree remains.

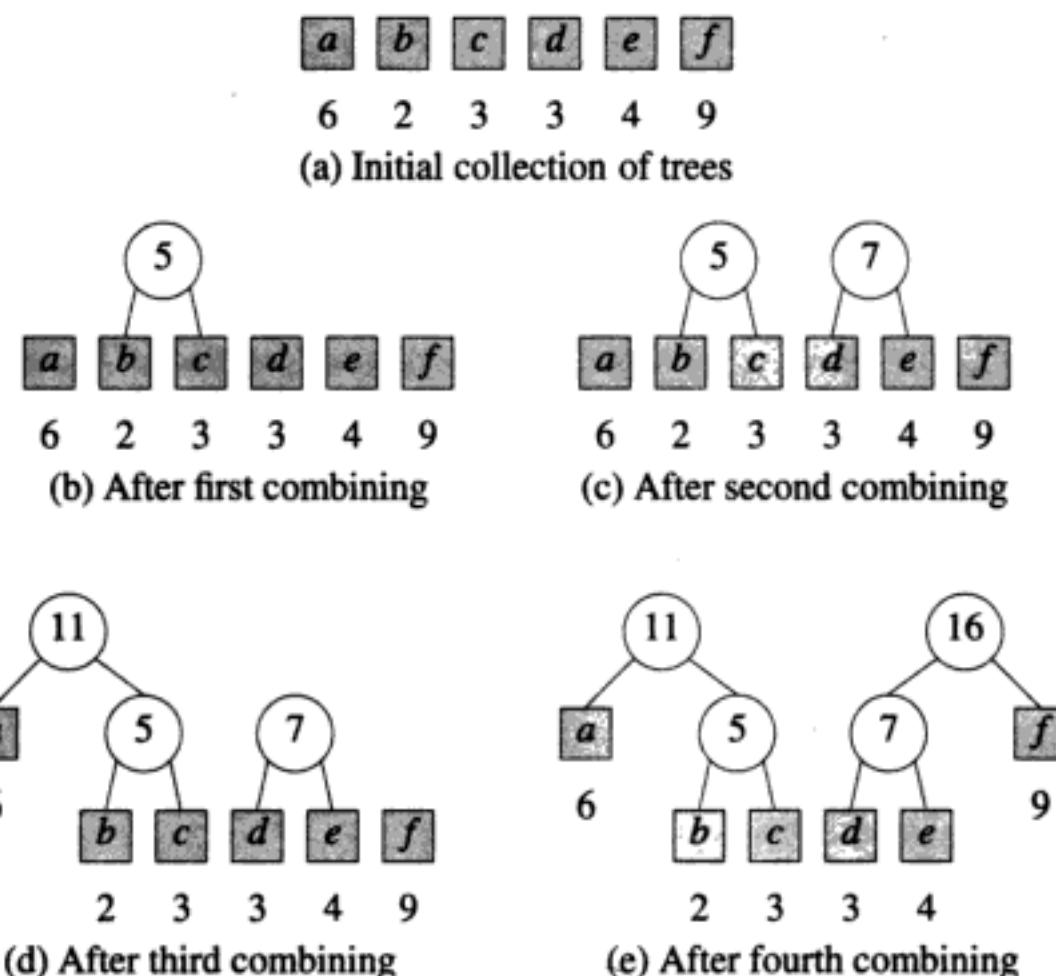


Figure 12.11 Constructing a Huffman tree

Let us try this construction method on a six-symbol (a, b, c, d, e, f) example with frequencies (6, 2, 3, 3, 4, 9). The initial binary trees are shown in Figure 12.11(a). The numbers outside the boxes are the tree weights. The tree with the lowest weight is b . We have a tie for the tree with the second-lowest weight. Suppose that we select tree c . Combining trees b and c yields the configuration of Figure 12.11(b). The root node is labeled with the weight 5. From the five trees of Figure 12.11(b), we select two of lowest weight. Trees d and e are selected and combined to get a tree of weight 7 (Figure 12.11(c)). From the four trees of Figure 12.11(c), we select

two of lowest weight to combine. Tree *a* and the one with weight 5 are selected. Combining these trees results in a tree of weight 11. From the remaining three trees (Figure 12.11(d)), the tree with weight 7 and the tree *f* are selected for combining. When these two trees are combined, the two trees of Figure 12.11(e) remain. These trees are next combined to get the single tree of Figure 12.6(b), whose weight is 27.

Theorem 12.3 *The procedure outlined above constructs binary trees with minimum WEP.*

Proof Left as an exercise (Exercise 41). ■

The Huffman tree construction procedure can be implemented using a min heap to store the collection of binary trees. Each element of the min heap consists of a binary tree and a value that is the weight of this binary tree. The binary tree itself is an instance of the class `linkedBinaryTree<int>` defined in Section 11.8. We assume, for convenience, that the symbols are of type `int` and are numbered 1 through *n*. For an external node the `element` field is set to the symbol it represents, and for an internal node this field is set to 0. The function `huffmanTree` (Program 12.10) assumes that the template struct `huffmanNode<T>` has the data members `tree`, which is of type `linkedBinaryTree<int>*`, and `weight`, which is of type `T`. We further assume that `huffmanNode<T>` defines a type conversion to the type `T` and that this conversion simply returns the `weight` field.

The method `huffmanTree` inputs a collection of *n* frequencies (or weights) in the array `w[1:n]` and returns a Huffman tree. The method begins by constructing *n* binary trees, each with just an external node. These binary trees are constructed using the `makeTree` method of `linkedBinaryTree`. This method constructs a binary tree given its root element and left and right subtrees. The constructed *n* binary trees are saved in the array `hNode`, which is later initialized to be a min heap. Each iteration of the second `for` loop removes two binary trees of minimum weight from the min heap and combines them into a single binary tree, which is then put into the min heap.

Complexity of `huffmanTree`

The time needed to create the array `hNode` is $O(n)$. The first `for` loop and the heap initialization also take $O(n)$ time. In the second `for` loop, a total of $2(n - 1)$ `top`, $2(n - 1)$ `pop`, and $n - 1$ `push` operations are performed, taking $O(n \log n)$ time. The remainder of `huffmanTree` takes $\Theta(n)$ time. So the overall time complexity is $O(n \log n)$.

EXERCISES

26. Show how the array [-, 5, 7, 2, 9, 3, 8, 6, 1] is sorted using heap sort. First draw the corresponding complete binary tree, then draw the heapified tree,

```
template <class T>
linkedBinaryTree<int>* huffmanTree(T weight[], int n)
{// Generate Huffman tree with weights weight[1:n], n >= 1.
    // create an array of single node trees
    huffmanNode<T> *hNode = new huffmanNode<T> [n + 1];
    linkedBinaryTree<int> emptyTree;
    for (int i = 1; i <= n; i++)
    {
        hNode[i].weight = weight[i];
        hNode[i].tree = new linkedBinaryTree<int>;
        hNode[i].tree->makeTree(i, emptyTree, emptyTree);
    }

    // make node array into a min heap
    minHeap<huffmanNode<T> > heap(1);
    heap.initialize(hNode, n);

    // repeatedly combine trees from min heap
    // until only one tree remains
    huffmanNode<T> w, x, y;
    linkedBinaryTree<int> *z;
    for (i = 1; i < n; i++)
    {
        // remove two lightest trees from the min heap
        x = heap.top(); heap.pop();
        y = heap.top(); heap.pop();

        // combine into a single tree
        z = new linkedBinaryTree<int>;
        z->makeTree(0, *x.tree, *y.tree);
        w.weight = x.weight + y.weight;
        w.tree = z;
        heap.push(w);
        delete x.tree;
        delete y.tree;
    }

    return heap.top().tree;
}
```

Program 12.10 Construct a Huffman tree

and then draw figures similar to those in Figure 12.9 to show the state of the tree following each removal from the max heap.

27. Do Exercise 26 using the array [-, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1].
28. Write a heap sort method that uses d -heaps (see Exercise 17). Compare the worst-case run times of heap sort using d -heaps for different d values. What value of d gives best performance?
29. Use the ideas of Exercises 15 and 16 to arrive at an implementation of heap sort that is faster than Program 12.8. Experiment with random data and compare the run times of the two implementations.
30. A sort method is said to be **stable** if the relative order of records with equal keys is the same after the sort as it was before the sort. Suppose that records 3 and 10 have the same key. In a stable sort record 3 will precede record 10 following the sort. Is heap sort a stable sort? How about insertion sort?
31. Draw the three-machine LPT schedule for the processing times [6, 5, 3, 2, 9, 7, 1, 4, 8]. What is the finish time for your schedule. Can you find a schedule with a smaller finish time or is your schedule optimal?
32. Do Exercise 31 using the processing times [20, 15, 10, 8, 8, 8].
33. Each iteration of the second `for` loop of Program 12.9 performs one `pop` and one `push`. The two operations together essentially increase the value of the minimum key by an amount equal to the processing time of the job just scheduled. We can speed up Program 12.9 by a constant factor by using an extended min priority queue. The extension includes the methods normally supported by a min priority queue plus the method `changeMin(x)` that changes the minimum element to `x`. This method moves down the heap (as in a remove min operation), moving elements up the heap until it finds an appropriate place for the changed element.
 - (a) Develop a new class `extendedMinHeap` that provides all the methods provided by the class `minHeap` plus the method `changeMin`. The class `extendedMinHeap` should be derived from the class `minHeap` that is available from the Web site for this book.
 - (b) Rewrite Program 12.9 using the method `changeMin`.
 - (c) Conduct experiments to determine the improvement in run time of your new code versus that of Program 12.9.
34. Construct a machine-scheduling instance for which the two-machine LPT schedule achieves the upper bound given in Theorem 12.2.
35. Do Exercise 34 for a three-machine LPT schedule.

36. n items are to be packed into containers. Item i uses s_i units of space, and each container has a capacity c . The packing is to be done using the **worst-fit** rule in which the items are assigned to containers one at a time. When an item is being assigned, we look for a nonempty container with maximum available capacity. If the item fits in this container, the assignment is made; otherwise, this item starts a new container.
- (a) Develop a program to input n , the s_i s, and c and to output the assignment of items to containers. Use a max heap to keep track of the available space in the containers.
 - (b) What is the time complexity of your program (as a function of n and the number m of containers used)?
37. Draw the Huffman tree for the weights (frequencies) [3, 7, 9, 12, 15, 20, 25].
38. Do Exercise 37 using the weights (frequencies) [2, 4, 5, 7, 9, 10, 14, 17, 18, 50].
39. A **run** is a sorted sequence of elements. Assume that two runs can be merged into a single run in time $O(r+s)$ where r and s are, respectively, the lengths of the two runs being merged. n runs of different lengths are to be merged into a single run by repeatedly merging pairs of runs until only one run remains. Explain how to use Huffman trees to determine a minimum-cost way to merge the n runs.
40. Suppose you are to code text that uses n symbols. A simple way to assign codes that satisfy the prefix property is to start with a right-skewed extended binary tree that has n external nodes, sort the n symbols in decreasing order of frequency $F()$, and assign symbols to external nodes so that an inorder listing of external nodes gives the symbols in decreasing order of their frequency. The sort step takes $O(n \log n)$ time, and the remaining steps take $O(n)$ time. So this method has the same asymptotic complexity as the optimal method described in Section 12.6.3.
- (a) Draw the Huffman tree and the right-skewed tree for the case when you have $n = 5$ symbols $a-e$ with frequencies [4, 6, 7, 9, 10]. Label each external node with the symbol it represents, list the code for each symbol, and give the WEP of each tree.
 - (b) Suppose that the n symbols have the same frequency. What is the ratio of the WEPs of the Huffman tree and the right-skewed tree? Assume that n is a power of 2.
 - (c) Write a method to construct the right-skewed extended binary tree as described above.
 - (d) Compare the actual run time of your method of part (c) and that of Program 12.10.

- (e) Generate random instances and measure the difference in the WEPs of the trees generated by the two methods.
 - (f) Based on your results for parts (b), (d), and (e), can you recommend the use of the method of this exercise over the method of Program 12.10? Why?
41. Prove Theorem 12.3 by using induction on the number of external nodes. The induction step should establish the existence of a binary tree with minimum WEP that has a subtree with one internal node and two external nodes corresponding to the two lowest frequencies.
 42. Write a method to take a Huffman tree as created by `huffmanTree` (Program 12.10) and to output the code table. What is the time complexity of your method?
 43. Develop a complete compression-decompression package based on Huffman codes. Test your code.
 44. A collection of n integers in the range 0 through 511 is to be stored. Develop a compression-decompression package for this application. Use Huffman codes.

12.7 REFERENCES AND SELECTED READINGS

A more detailed study of data structures for priority queues and priority-queue variants can be found in the text *Fundamentals of Data Structures in C++* by E. Horowitz, S. Sahni, and D. Mehta, W. H. Freeman, New York, NY, 1994.

Height-biased leftist trees are described in the monograph *Data Structures and Network Algorithms* by R. Tarjan, SIAM, Philadelphia, PA, 1983, while weight-biased leftist trees are developed in the paper “Weight Biased Leftist Trees and Modified Skip Lists” by S. Cho and S. Sahni, *ACM Jr. on Experimental Algorithms*, Article 2, 1998.

You can find out more about NP-hard problems from the books *Computers and Intractability: A Guide to the Theory of NP-Completeness* by M. Garey and D. Johnson, W. H. Freeman, New York, NY, 1979, and *Computer Algorithms* by E. Horowitz, S. Sahni, and S. Rajasekeran, Computer Science Press, New York, NY, 1998. Chapter 12 of *Computer Algorithms* proves Theorem 12.2.

CHAPTER 13

TOURNAMENT TREES

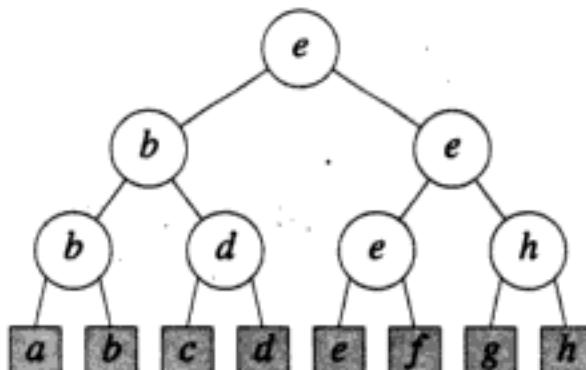
BIRD'S-EYE VIEW

We have reached the halfway point in our journey through the forest of trees. The new tree variety we encounter in this chapter is the tournament tree. Like the heap of Section 12.4, a tournament tree is a complete binary tree that is most efficiently stored by using the array binary tree representation of Section 11.4.1. The basic operation that a tournament tree supports is replacing the maximum (or minimum) element. If we have n elements, this operation takes $\Theta(\log n)$ time. Although this operation can be done with the same asymptotic complexity—in fact, $O(\log n)$ —using either a heap or a leftist tree, neither of these structures can implement a predictable tie breaker easily. The tournament tree becomes the data structure of choice when we need to break ties in a prescribed manner, such as to select the element that was inserted first or to select the element on the left (all elements are assumed to have a left-to-right ordering).

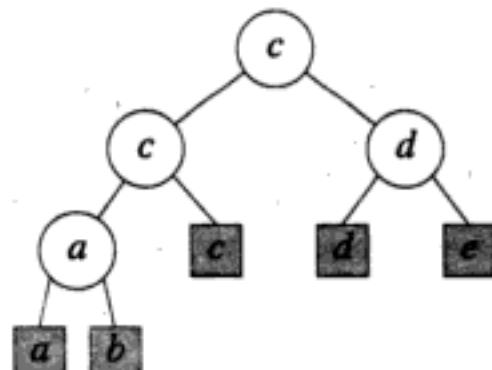
We study two varieties of tournament trees: winner and loser trees. Although winner trees are more intuitive and model real-world tournament trees, loser trees can be implemented more efficiently. The applications section at the end of the chapter considers another NP-hard problem, bin packing. Tournament trees are used to obtain efficient implementations of two approximation algorithms for the bin-packing problem. You will find it instructive to see whether you can implement these algorithms in the same time bounds with any of the other data structures developed so far in this text.

13.1 WINNER TREES AND APPLICATIONS

Suppose that n players enter a tennis tournament. The tournament is to be played in the *sudden-death* mode in which a player is eliminated upon losing a match. Pairs of players play matches until only one player remains undefeated. This surviving player is declared the tournament winner. Figure 13.1(a) shows a possible tennis tournament involving eight players a through h . The tournament is described by a binary tree in which each external node represents a player and each internal node represents a match played between players designated by the children of the node. Each level of internal nodes defines a *round* of matches that can be played in parallel. In the first round players a and b , c and d , e and f , and g and h play. The winner of each match is recorded at the internal node that represents the match. In the case of Figure 13.1(a), the four winners are b , d , e , and h . The remaining four players (i.e., the losers) are eliminated. In the next round of matches, b and d play against each other as do e and h . The winners are b and e , who play the final match. The overall winner is e . Figure 13.1(b) shows a possible tournament that involves five players a through e . The winner in this case is c .



(a) Eight players



(b) Five players

Figure 13.1 Tournament trees

Although both trees of Figure 13.1 are complete binary trees (actually, tree (a) is also a full binary tree), trees that correspond to real-world tournaments do not have to be complete binary trees. However, using complete binary trees minimizes the number of rounds of matches that have to be played. For an n -player tournament, this number is $\lceil \log_2 n \rceil$. The tournament tree depicted in Figure 13.1 is called a **winner tree** because at each internal node, we record the winner of the match played at that node. Section 13.4 considers another variety, called a **loser tree**, in which we record the loser at each internal node. Tournament trees are also known as **selection trees**.

Winner trees may be adapted for computer use. In this adaptation we restrict ourselves to complete binary trees.

Definition 13.1 A **winner tree** for n players is a complete binary tree with n external and $n - 1$ internal nodes. Each internal node records the winner of the match played there. ■

To determine the winner of a match, we assume that each player has a *value* and that there is a rule to determine the winner based on a comparison of the two players' values. In a **min winner tree**, the player with the smaller value wins, while in a **max winner tree**, the player with the larger value wins. In case of a tie, the player represented by the left child of the node wins. Figure 13.2(a) shows an eight-player min winner tree, while Figure 13.2(b) shows a five-player max winner tree. The number below each external node is the player's value.

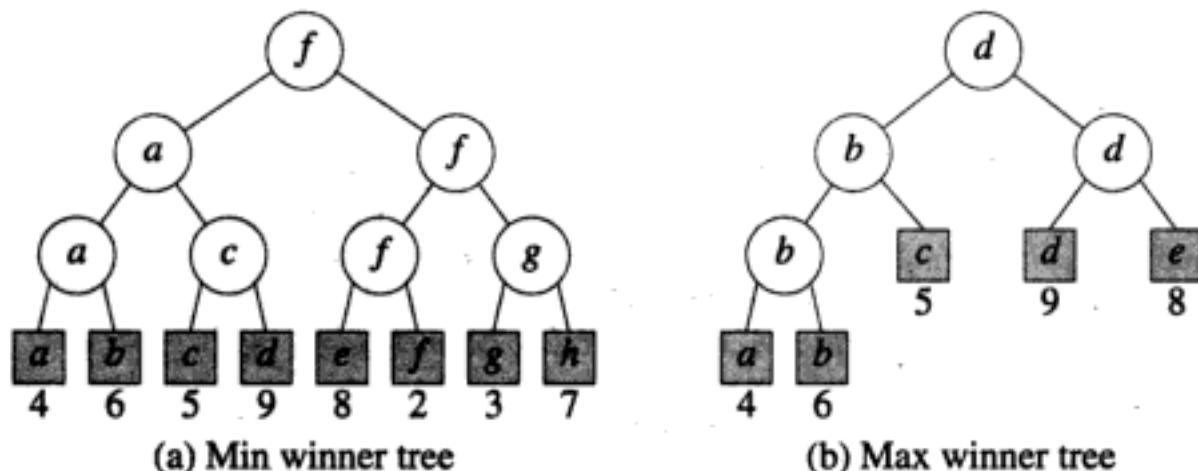


Figure 13.2 Winner trees

One of the nice things about a winner tree is that we can easily modify the tree to accommodate a change in one of the players. For example, if the value of player d changes from 9 to 1, then we need only replay matches on the path from d to the root. The change in this value does not affect the outcome of the remaining matches. In some situations we can avoid replaying some of the matches on the path to the root. For example, if in the min winner tree of Figure 13.2(a), the value of player b changes from 6 to 5, we play at its parent and b loses. There is no need to replay the matches at the grandparent and great-grandparent of b , as these matches will have the same outcome as they had before.

Since the number of matches needed to restructure an n -player winner tree following a change in one value ranges from a low of 1 to a high of $\lceil \log_2 n \rceil$, the time needed for restructuring is $O(\log n)$. Also, an n -player winner tree can be initialized in $\Theta(n)$ time by playing the $n - 1$ matches at the internal nodes by beginning with

the matches at the lowest level and working up to the root. Alternatively, the tree can be initialized by performing a postorder traversal. During the visit step, a match is played.

Example 13.1 [Sorting] We may use a min winner tree to sort n elements in $\Theta(n \log n)$ time. First, the winner tree is initialized with the n elements as the players. The sort key is used to decide the outcome of each match. The overall winner is the element with the smallest key. This player's key is now changed to a very large number (say ∞) so that it cannot win against any of the remaining players. We restructure the tree to reflect the change in this player's key. The new overall winner is the element that comes next in sorted order. Its key is changed to ∞ , and the tree is restructured. Now the overall winner is the element that is third in sorted order. Continuing in this way, we can sort the n elements. It takes $\Theta(n)$ time to initialize the winner tree. Each key change and restructure operation takes $\Theta(\log n)$ time because when the key of the tournament winner changes, we need to replay all matches on the path to the root. The restructuring needs to be done $n - 1$ times, so the overall restructuring time is $\Theta(n \log n)$. Adding in the time needed to initialize the winner tree, the complexity of the sort method becomes $\Theta(n + n \log n) = \Theta(n \log n)$. ■

Example 13.2 [Run Generation] The sorting methods (insertion sort, heap sort, etc.) we have discussed so far are all **internal sorting methods**. These methods require that the elements to be sorted fit in the memory of our computer. When the element collection does not fit in memory, internal sort methods do not work well because they require too many accesses to the external storage media (say a disk) on which all or part of the collection resides. In this case sorting is accomplished with an **external sorting method**. A popular approach to external sorting involves (1) generating sorted sequences called **runs** and (2) merging these runs together to create a single run.

Suppose we wish to sort a collection of 16,000 records and we are able to sort up to 1000 records at a time using an internal sort. Then in step 1, we do the following 16 times to create 16 runs:

- Input 1000 records.
- Sort these records using an internal sort.
- Output the sorted sequence (or run).

Following this run-generation step, we initiate the run-merging step, step 2. In this step we make several merge passes over the runs. In each merge pass we merge up to k runs, creating a single sorted run. So each merge pass reduces the number of runs by a factor of $1/k$. Merge passes are continued until the number of runs becomes 1.

In our example of 16 runs, we could perform two passes of four-way merges, as in Figure 13.3. The initial 16 runs are labeled $R_1 \dots R_{16}$. First, runs $R_1 \dots R_4$

merge to obtain the run S_1 , which is 4000 records long. Then $R_5 \dots R_8$ merge, and so on. At the second merge pass, $S_1 \dots S_4$ are merged to create the single run T_1 , which is the desired output from the external sort.

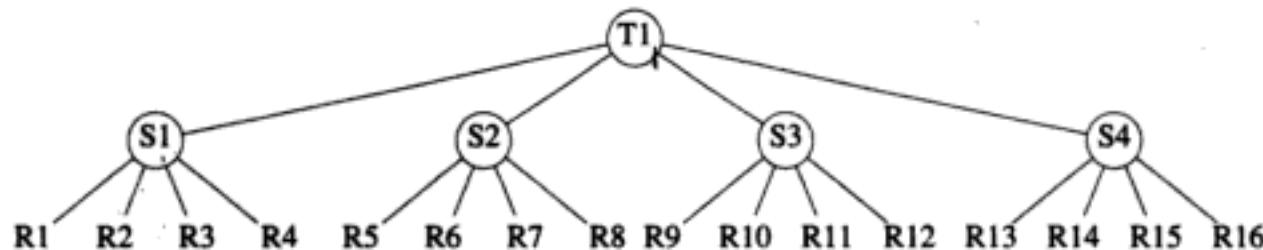


Figure 13.3 Four-way merging of 16 runs

A simple way to merge k runs is to repeatedly remove the element with smallest key from the front of these k runs. This element is moved to the output run being generated. The process is complete when all elements have been moved from the input k runs to the output run. Notice that to determine the next element in the output run all we need in memory is the key of the front element of each input run. We can merge k runs of arbitrary length as long as we have enough memory to hold k keys. In practice, we will want to input and output many elements at a time so as to reduce the input/output time.

In our 16,000-record example, each run is 1000 records long, and our memory capacity is also 1000 records. To merge the first four runs, we could partition memory into five buffers, each large enough to hold 200 records. Four of these buffers are designated input buffers, and the fifth is an output buffer. Two hundred records from each of the first four runs are input into the four input buffers. The output buffer is used to collect the merged records. Records are merged from the input buffers into the output buffer until one of the following conditions occurs:

- The output buffer becomes full.
- An input buffer becomes empty.

When the first condition occurs, we write the output buffer to disk and resume merging when this write has completed. When the second condition occurs, we read in the next buffer load (if any) for the run that corresponds to the empty input buffer and resume merging when this input has completed. The merge of the four runs is complete when all 4000 records from these runs have been written out as the single run S_1 . (A more sophisticated run-merging scheme is described in *Fundamentals of Data Structures in C++* by E. Horowitz, S. Sahni, and D. Mehta, Computer Science Press, New York, NY, 1995.)

One of the factors that determines the amount of time spent in the run-merging step is the number of runs generated in step 1. By using a winner tree, we can

often reduce the number of runs generated. We begin with a winner tree for p players where each player is an element of the input collection. Each player has an associated key and run number. The first p elements are assigned run number 1. When a match is played between two players, the element with the smaller run number wins. In case of a tie, the keys are compared, and the element with the smaller key wins. If a tie remains, it may be broken arbitrarily. To generate runs, we repeatedly move the overall winner W into the run corresponding to its run-number field and replace the moved element by the next input element N . If the key of N is \geq the key of W , then element N can be output as part of the same run. It is assigned a run number equal to that of W . If the key of N is less than that of W , outputting N after W in the same run violates the sorting constraint on a run. N is assigned a run number that is 1 more than that of W .

When using this method to generate runs, the average run length is $\approx 2p$. When $2p$ is larger than the memory capacity, we expect to get fewer runs than we do by using the simple scheme proposed earlier. In fact, if our input collection is already sorted (or nearly sorted), only one run is generated and we can skip the run-merging step, step 2. ■

Example 13.3 [k-Way Merging] In a k -way merge (see Example 13.2), k runs are merged to generate a single sorted run. The simple scheme, described in Example 13.2, to perform a k -way merge requires $O(k)$ time per element merged to the output run because in each iteration we need to find the smallest of k keys. The total time to generate a run of size n is, therefore, $O(kn)$. We can reduce this time to $\Theta(k + n \log k)$ using a winner tree. First we spend $\Theta(k)$ time to initialize a winner tree for k players. The k players are the first elements of the k runs to be merged. Then the winner is moved to the output run and replaced by the next element from the corresponding input run. If there is no next element in this run, it is replaced by an element with very large key (say ∞). We remove and replace the winner a total of n times at a cost of $\Theta(\log k)$ each time. The total time needed to perform the k -way merge is $\Theta(k + n \log k)$. ■

EXERCISES

1. Draw the max and min winner trees for players [3, 5, 6, 7, 20, 8, 2, 9]. Now draw the trees that result when 20 is changed to 1. Obtain the new trees by replaying only the matches on the path from 1 to the root.
2. Draw the max and min winner trees for players [20, 10, 12, 18, 30, 16, 35, 33, 45, 7, 15, 19, 33, 11, 17, 25]. Now draw the trees that result when 17 is changed to 42. Obtain the new trees by replaying only the matches on the path from 42 to the root.

Hidden page

AbstractDataType WinnerTree

{

instances

complete binary trees with each node pointing to the winner of the match played there; the external nodes represent the players

operations

initialize(a) : initialize a winner tree for the players in the array *a*

winner() : return the tournament winner

rePlay(i) : replay matches following a change in player *i*

}

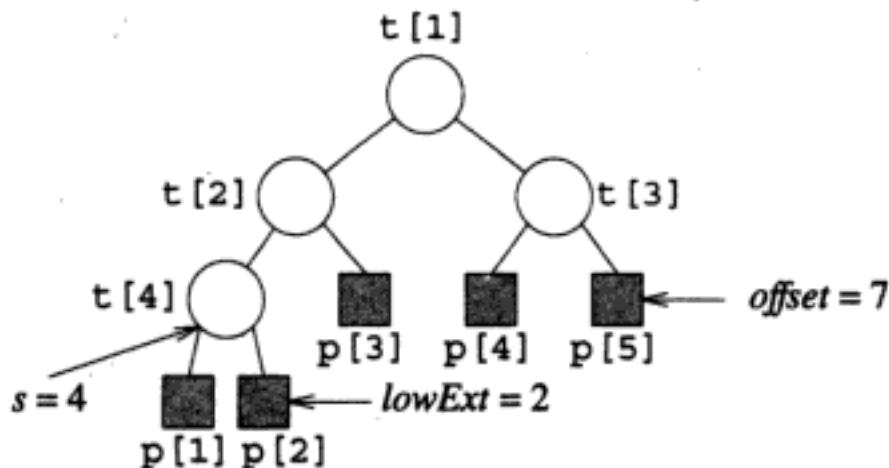
ADT 13.1 Abstract data type specification of a winner tree

```
template<class T>
class winnerTree
{
public:
    virtual ~winnerTree() {}
    virtual void initialize(T *thePlayer,
                           int theNumberOfPlayers) = 0;
    // create winner tree with thePlayer[1:numberOfPlayers]
    virtual int winner() const = 0;
    // return index of winner
    virtual void rePlay(int thePlayer) = 0;
    // replay matches following a change in thePlayer
};
```

Program 13.1 The abstract class winnerTree

correspondence between the nodes of a winner tree and the arrays *tree* and *player* for the case of a five-player tree.

To implement the interface methods, we must be able to determine the parent *tree[p]* of an external node *player[i]*. When the number of external nodes is *n*, the number of internal nodes is *n - 1*. The left-most internal node at the lowest level is numbered *s* where $s = 2^{\lfloor \log_2(n-1) \rfloor}$. Therefore, the number of internal nodes at the lowest level is *n - s*, and the number *lowExt* of external nodes at the lowest level is twice this number. For example, in the tree of Figure 13.4, *n* = 5 and *s* = 4. The left-most internal node at the lowest level is *tree[4]*, and the total number of



`t[] = tree[] and p[] = player[]`

Figure 13.4 Tree-to-array correspondence

internal nodes at this level is $n - 4 = 1$. The number of lowest-level external nodes is 2. The left-most external node at the second-lowest level is numbered $lowExt + 1$. Let $offset = 2 * s - 1$. Then we see that for any external node $player[i]$, its parent $tree[p]$ is given by

$$p = \begin{cases} (i + offset)/2 & i \leq lowExt \\ (i - lowExt + n - 1)/2 & i > lowExt \end{cases} \quad (13.1)$$

13.3.2 Initializing a Winner Tree

To initialize a winner tree, we play matches beginning at players that are right children and going up the tree whenever a match is played at a right child. For this purpose right-child players are considered from left to right. So in the tree of Figure 13.4, we first play matches beginning at player 2, then we begin at player 3, and finally, we begin at player 5. When we start at player 2, we play the match at $tree[4]$. The match at the next level, that is, at $tree[2]$, is not played because $tree[4]$ is a left child. We now start playing at $player[3]$. The match at $tree[2]$ is played, but the one at $tree[1]$ is not (because $tree[2]$ is a left child). Finally, we start at $player[5]$ and play the matches at $tree[3]$ and $tree[1]$. Notice that when a match is played at $tree[i]$, the players for this match have already been determined and recorded in the children of $tree[i]$.

Hidden page

Hidden page

is g . Next g plays at the root against a . Again a is the player who lost the match previously played at the root.

We can reduce the work needed to determine the players of each match on the path from the changed winner $\text{player}[i]$ to the root if we record in each internal node the loser of the match played at that node, rather than the winner. The overall winner may be recorded in $\text{tree}[0]$. Figure 13.5(a) shows the loser tree for the eight players of Figure 13.2(a). Now when the winner f is changed to have key 5, we move to its parent $\text{tree}[6]$. The match is to be played between $\text{player}[\text{tree}[6]]$ and $\text{player}[6]$. To determine the opponent for $f' = \text{player}[6]$, we simply look at $\text{tree}[6]$. In a winner tree we would need to determine the other child of $\text{tree}[6]$. After playing the match at $\text{tree}[6]$, the loser e is recorded here, and f' advances to play at $\text{tree}[3]$ with the previous loser of this match. This loser, g , is available from $\text{tree}[3]$. f' loses, and this fact is recorded in $\text{tree}[3]$. The winner g plays with the previous loser of the match at $\text{tree}[1]$. This loser, a , is available from $\text{tree}[1]$. The new loser tree appears in Figure 13.5(b).

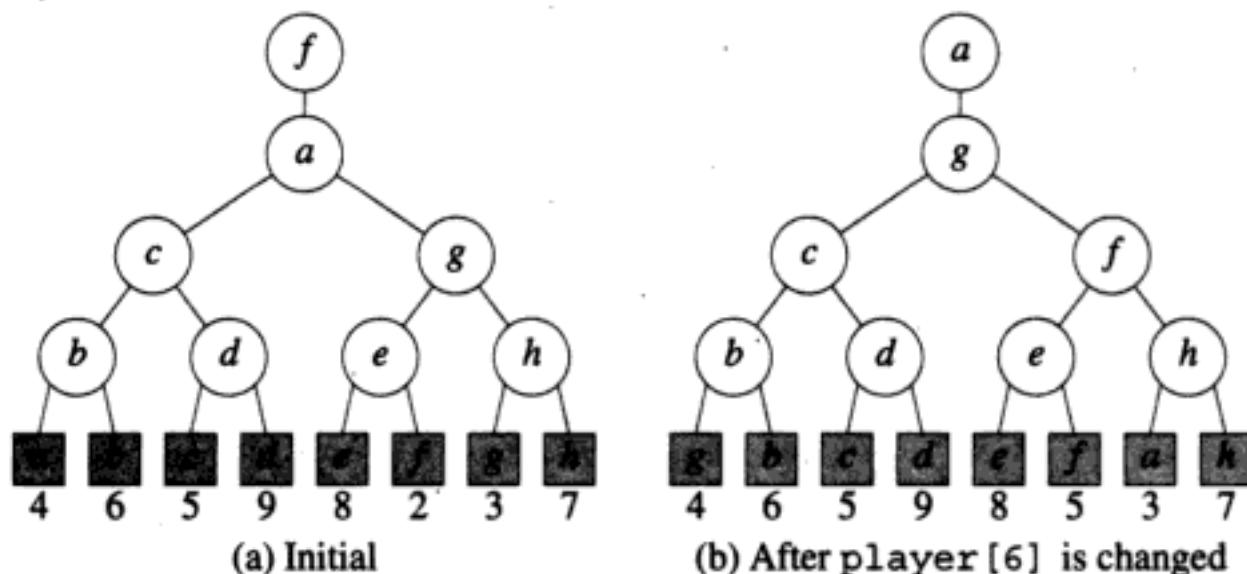


Figure 13.5 Eight-player min loser trees

Although a loser tree simplifies the replaying of matches following a change in the previous winner, it does not result in a similar simplification when a change is made in other players. For example, suppose that the key of player d is changed from 9 to 3. Matches are to be replayed at $\text{tree}[5]$, $\text{tree}[2]$, and $\text{tree}[1]$. At $\text{tree}[5]$ d has to play c , but c is not the previous loser of this match. At $\text{tree}[2]$ d has to play a , but a did not lose the match previously played here. At $\text{tree}[1]$ d has to play f , not the player who previously lost at $\text{tree}[1]$. To replay these matches easily, a winner tree is needed. Therefore, we expect a loser tree to give better performance than a winner tree only when the `rePlay(i)` method is restricted to

Hidden page

- (c) Write a recursive version of `initialize`. The complexity of your code should be $O(n)$.
 - (d) Write a version of `initialize` that uses the following strategy. Start playing matches at the left-most match node both of whose children are player nodes. Play matches from here to the root as far as you can, recording losers. When a match cannot be played (because one of the players is unknown), record the single player determined for that match. Repeat this process starting at a suitable match node. Continue until all matches have been played. Show that your code takes $O(n)$ time to initialize a loser tree with n players.
20. Develop the C++ class `fullLoserTree` that implements a loser tree as a full binary tree as described in Exercise 11. See Exercise 19 for initialization strategies. Test the correctness of your code.
21. Develop the C++ class `completeLoserTree2` that implements a loser tree using the strategy described in Exercise 12. See Exercise 19 for initialization strategies. Test the correctness of your code.
22. Write a sort program that uses a loser tree to repeatedly extract elements in sorted order. What is the complexity of your program?

13.5 APPLICATIONS

13.5.1 Bin Packing Using First Fit

Problem Description

In the bin-packing problem, we have an unlimited supply of bins that have a capacity `binCapacity` each and n objects that need to be packed into these bins. Object i requires `objectSize[i]`, $0 < \text{objectSize}[i] \leq \text{binCapacity}$, units of capacity. A **feasible** packing is an assignment of objects to bins so that no bin's capacity is exceeded. A feasible packing that uses the fewest number of bins is an **optimal packing**.

Example 13.4 [Truck Loading] A freight company needs to pack parcels into trucks. Each parcel has a weight, and each truck has a load limit (assumed to be the same for all trucks). In the truck-loading problem, we are to pack the parcels into trucks using the fewest number of trucks. This problem may be modeled as a bin-packing problem with each truck being a bin and each parcel an object that needs to be packed. ■

Example 13.5 [Chip Placement] A collection of electronic chips is to be placed in rows on a circuit board of a given width. The chips have the same height but

different widths. The height, and hence area, of the circuit board is minimized by minimizing the number of rows used. The chip-placement problem may also be modeled as a bin-packing problem with each row being a bin and each chip an object that needs to be packed. The board's width is the bin capacity, and the chip's length, the capacity needed by the corresponding object. ■

Approximation Algorithms

The bin-packing problem, like the machine-scheduling problem of Section 12.6.2, is an NP-hard problem. As a result, it is often solved using an approximation algorithm. In the case of bin packing, such an algorithm generates solutions that use a number of bins that is close to minimum. Four popular approximation algorithms for this problem are

1. First Fit (FF)

Objects are considered for packing in the order $1, 2, \dots, n$. We assume a large number of bins arranged from left to right. Object i is packed into the left-most bin into which it fits.

2. Best Fit (BF)

Let $\text{bin}[j].\text{unusedCapacity}$ denote the capacity available in bin j . Initially, the unused capacity is binCapacity for all bins. Object i is packed into the bin with the least unusedCapacity that is at least $\text{objectSize}[i]$.

3. First Fit Decreasing (FFD)

This method is the same as FF except that the objects are first reordered so that $\text{objectSize}[i] \geq \text{objectSize}[i+1]$, $1 \leq i < n$.

4. Best Fit Decreasing (BFD)

This method is the same as BF except that the objects are reordered as for FFD.

You should be able to show that none of these methods guarantee optimal packings. All four are intuitively appealing and can be expected to perform well in practice. A worst-fit packing strategy was considered in Exercise 36 of Chapter 12. We may also consider last fit, last-fit decreasing, and worst-fit decreasing strategies.

Theorem 13.1 Let I be any instance of the bin-packing problem. Let $b(I)$ be the number of bins used by an optimal packing. The number of bins used by FF and BF never exceeds $(17/10)b(I) + 2$, while that used by FFD and BFD does not exceed $(11/9)b(I) + 4$.

Example 13.6 Four objects with $\text{objectSize}[1:4] = [3, 5, 2, 4]$ are to be packed in bins of size 7. When FF is used, object 1 goes into bin 1 and object 2 into bin 2.

Hidden page

First Fit and Winner Trees

The FF and FFD methods can be implemented so as to run in $O(n \log n)$ time using a winner tree. Since the maximum number of bins ever needed is n , we can begin with n empty bins. Initially, `bin[j].unusedCapacity = binCapacity` for all n bins. Next a max winner tree with the `bin[j]`'s as players is created. Figure 13.6(a) shows the max winner tree for the case $n = 8$ and `binCapacity = 10`. The external nodes represent bins 1 through 8 from left to right. The number below an external node is the space available in that bin. Suppose that `objectSize[1] = 8`. To find the left-most bin for this object, we begin at the root `tree[1]`. By definition, `bin[tree[1]].unusedCapacity ≥ objectSize[1]`. This relationship simply means that there is at least one bin into which the object fits. To find the left-most bin, we determine whether there is enough space in one of the bins 1 through 4. One of these bins has enough space iff `bin[tree[2]].unusedCapacity ≥ objectSize[1]`. In our example this relationship holds, and so we can continue the search for a bin in the subtree with root 2. Now we determine whether there is adequate space in any bin covered by the left subtree of 2 (i.e., the subtree with root 4). If so, we are not interested in bins in the right subtree. In our example we move into the left subtree as `bin[tree[4]].unusedCapacity ≥ objectSize[1]`. Since the left subtree of 4 is an external node, we know that `objectSize[1]` is to be placed in one of node 4's two children. It goes in the left child provided this child has enough space. When object 1 is placed in bin 1, `bin[1].unusedCapacity` reduces to 2 and we must replay matches beginning at `bin[2]`. The new winner tree appears in Figure 13.6(b). Now suppose that `objectSize[2] = 6`. Since `bin[tree[2]].unusedCapacity ≥ 6`, we know that there is a bin with adequate space in the left subtree. So we move here. Then we move into the left subtree `tree[4]` and place object 2 in bin 2. The new configuration appears in Figure 13.6(c). When `objectSize[3] = 5`, the search for a bin leads us into the subtree with root 2. For its left subtree, `bin[tree[4]].unusedCapacity < objectSize[3]`, so no bin in the subtree with root 4 has enough space. As a result, we move into the right subtree 5 and place the object in bin 3. This placement results in the configuration of Figure 13.6(d). Next, suppose `objectSize[4] = 3`. Our search gets us to the subtree with root 4, as `bin[tree[4]].unusedCapacity ≥ objectSize[3]`, and we add object 3 to bin 2.

C++ Implementation of First Fit

First we add the following public method to the class `completeWinnerTree`.

```
int winner(int i) const
    {return (i < numberofPlayers) ? tree[i] : 0;}
```

This method returns the winner of the match played at internal node i .

The function `firstFitPack` (Program 13.2) implements the first-fit strategy. This function assumes that the number of objects is at least 2 and that the size

Hidden page

Hidden page

```
    cout << "Pack object " << i << " in bin "
        << binToUse << endl;
    bin[binToUse].unusedCapacity -= objectSize[i];
    winTree.rePlay(binToUse);
}
}
```

Program 13.2 The function `firstFitPack` (concluded)

defeats one of the objectives of using a class—information hiding. We wish to keep the implementation details of the class away from the user. When the user and class are so separated, we can change the implementation while keeping the public aspects of the class unchanged. Such changes do not affect the correctness of the applications. In the interests of information hiding, we may add methods to the class `completeWinnerTree` to enable the user to move to the left and right children of an internal node, and then we can use these methods in `firstFitPack`.

13.5.2 Bin Packing Using Next Fit

What Is Next Fit?

Next fit is a bin-packing strategy in which objects are packed into bins one at a time. We begin by packing object 1 in bin 1. For the remaining objects, we determine the next nonempty bin that can accommodate the object by polling the bins in a round-robin fashion, beginning at the bin next to the one last used. In such a polling process, if bins 1 through b are in use, then these bins are viewed as arranged in a circle. The bin next to (or after) bin i is $i+1$ except when $i = b$; in this case the next bin is bin 1. If the last object placed went into bin j , then the search for a bin for the current object begins at the bin next to bin j . We examine successive bins until we either encounter a bin with enough space or return to bin j . If no bin with sufficient capacity is found, a new bin is started and the object placed into it.

Example 13.7 Six objects with `objectSize[1:6] = [3, 5, 3, 4, 2, 1]` are to be packed in bins of size 7. When next fit is used, object 1 goes into bin 1. Object 2 doesn't fit into a nonempty bin. So a new bin, bin 2, is started with this object in it. For object 3, we begin by examining the nonempty bin next to the last bin used. The last bin used was bin 2, and the bin next to it is bin 1. Bin 1 has enough space, and object 3 goes in it. For object 4, the search begins at bin 2, as bin 1 was the last one used. This bin doesn't have enough space. The bin next to bin 2 (i.e., bin 1) doesn't have enough space either. So a new bin, bin 3, is started with object 4 in it. The search for a bin for object 5 begins at the bin next to bin 3. This bin is bin 1. From here the search moves to bin 2 where the object is actually packed.

For the last object we begin by examining bin 3. Since this bin has enough space, the object is placed here. ■

The next-fit strategy described above is modeled after a dynamic memory allocation strategy that has the same name. In the context of bin packing, there is another next-fit strategy in which objects are packed one at a time. If an object does not fit into the current bin, then the current bin is closed and a new bin is started. We do not consider this variant of the next-fit strategy in this section.

Next Fit and Winner Trees

A max winner tree may be used to obtain an efficient implementation of the next-fit strategy. As in the case of first fit, the external nodes represent the bins, and matches are played by comparing the available space in the bins. For an n -object problem, we begin with n bins/external nodes. Consider the max winner tree of Figure 13.7, in which six of eight bins are in use. The labeling convention is the same as that used in Figure 13.6. Although the situation shown in Figure 13.7 cannot arise when $n = 8$, it illustrates how to determine the bin in which to pack the next object. If the last object was placed in bin `lastBinUsed` and b bins are currently in use, the search for the next bin to use can be broken into two searches as follows:

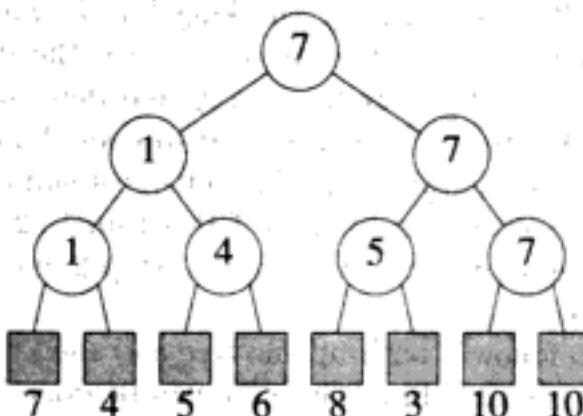


Figure 13.7 A next-fit max winner tree

Step 1: Find the first bin j , $j > \text{lastBinUsed}$ into which the object fits. Such a j always exists as the number of bins is n . If this bin is not empty (i.e., $j \leq b$), this bin is the bin to use.

Step 2: If step 1 does not find a nonempty bin, find the left-most bin into which the object fits. This bin is now the bin to use.

Consider the situation depicted in Figure 13.7 and suppose that the next object to be placed needs seven units of space. If `lastBinUsed` = 3, then in step 1 we determine that bin 5 has adequate space. Since bin 5 is not an empty bin, the object is placed into it. On the other hand, if `lastBinUsed` = 5, then in step 1 bin 7 is identified as the bin with enough space. Since bin 7 is empty, we move to step 1 and look for the left-most bin with enough capacity. This left-most bin is bin 1, and the object is placed into it.

To implement step 1, we begin at bin $j = \text{lastBinUsed} + 1$. Notice that if `lastBinUsed` = n , then all n objects must have been packed, as the only way to utilize n bins when the number of objects is n is to pack one object in each bin. So $j \leq n$. The pseudocode of Figure 13.8 describes the search process adopted from bin j . Basically, we traverse the path from bin j to the root, examining right subtrees until we find the first one that contains a bin with sufficient available capacity. When we find such a subtree, the bin we seek is the left-most bin in this subtree that has sufficient available capacity.

Consider the winner tree of Figure 13.7. Suppose that `lastBinUsed` = 1 and `objectSize[i]` = 7. We begin with $j = 2$. First we determine that bin 2 does not have enough capacity. Next we check bin $j+1 = 3$. It doesn't have enough capacity either. So we move to bin j 's parent and set p equal to 4. Since $p \neq n-1$, we reach the `while` loop and determine that the subtree with root 5 does not have a suitable bin. Next we move to node 2 and determine that the subtree with root 3 has a suitable bin. The required bin is the left-most bin in this subtree that has seven or more units of space available. This bin, bin 5, can be found following the strategy of Program 13.2. Suppose, instead, that `lastBinUsed` = 3 and `objectSize[i]` = 9. We begin by checking bin 4. Since neither bin 4 nor bin 5 has enough capacity, p is set to 5 and we reach the `while` loop. The first iteration checks `bin[tree[6]].unusedCapacity` and determines that the subtree with root 6 has no suitable bin. p is then moved to node 2, and we determine that the subtree with root 3 has a suitable bin. The left-most suitable bin in this subtree is identified, using a process similar to that of Program 13.2. This left-most bin is bin 7. Since this bin is an empty bin, we move to step 2 and determine that bin 7 is, in fact, the bin to use.

Step 1 requires us to follow a path up the tree and then make a downward pass to identify the left-most suitable bin. This step can be done in $O(\log n)$ time. Step 2 may be done in $O(\log n)$ by following the strategy of Program 13.2, so the overall complexity of the proposed next-fit implementation is $O(n \log n)$.

EXERCISES

23. Suppose that `binCapacity` = 10, n = 5, and `objectSize[0 : 4]` = [6, 1, 4, 4, 5].
 - (a) Determine an optimal packing for these objects.
 - (b) Give the assignment of objects to bins obtained by using each of these methods: FF, BF, FFD, and BFD.

CHAPTER 14

BINARY SEARCH TREES

BIRD'S-EYE VIEW

This chapter and the next develop tree structures suitable for the representation of a dictionary. Although we have already seen data structures such as skip lists and hash tables that can be used to represent a dictionary, the binary search tree data structures developed in this chapter and the balanced search tree structures developed in Chapter 15 provide additional flexibility and/or better worst-case performance guarantees.

This chapter examines binary search trees and indexed binary search trees. Binary search trees provide an asymptotic performance that is comparable to that of skip lists—the expected complexity of a search (or find), insert, or delete (or erase) operation is $\Theta(\log n)$, while the worst-case complexity is $\Theta(n)$; and elements may be output in ascending order in $\Theta(n)$ time. Although the worst-case complexities of the search, insert, and delete operations are the same for binary search trees and hash tables, hash tables have a superior expected complexity— $\Theta(1)$ —for these operations. Binary search trees, however, allow you to search efficiently for keys that are close to a specified key. For example, you can find the key nearest to the key k (i.e., either largest key $\leq k$ or smallest key $\geq k$) in expected time $\Theta(n)$. The expected time for this operation is $\Theta(n + D)$ for hash tables, where D is the hash table divisor, and $\Theta(\log n)$ for skip lists.

Indexed binary search trees allow you to perform dictionary operations both by

key value and by rank—get the element with the 10th smallest key and remove the element with the 100th smallest key. The expected time for by-rank operations is the same as for by-key operations. Indexed binary search trees may be used to represent linear lists (defined in Chapter 5); the elements have a rank (i.e., index) but no key. The result is a linear list representation in which the expected complexity of the *get*, *erase*, and *insert* operations is $O(\log n)$. Recall that the array and linked representations of linear lists developed in Chapters 5 and 6 perform these operations with expected complexity $\Theta(n)$. The *get* operation for array representations is an exception; it takes $\Theta(1)$ time. Hash tables cannot be extended to support by-rank operations efficiently. Skip lists may be extended to permit by-rank operations with expected time complexity $\Theta(\log n)$.

Although the asymptotic complexity (both expected and worst case) for all tasks stated above is the same for binary search trees and skip lists, we can impose a balancing constraint on binary search trees so that the worst-case complexity for each of the stated tasks is $\Theta(\log n)$. Balanced search trees are the subject of Chapter 15.

Three applications of binary search trees are developed in the applications section. The first is the computation of a histogram. The second application is the implementation of the best-fit approximation method for the NP-hard bin-packing problem of Section 13.5.1. The final application is the crossing-distribution problem that arises when we route wires in a routing channel. We can improve the expected performance of the histogram application by using hashing in place of the search tree. In the best-fit application, the searches are not done by an exact match (i.e., we do a nearest key search), and so hashing cannot be used. In the crossing-distribution problem, the operations are done by rank, and so hashing cannot be used here either. The worst-case performance of each of these applications can be improved by using any of the balanced binary search tree structures developed in Chapter 15 in place of the unbalanced binary search tree developed in this chapter.

14.1 DEFINITIONS

14.1.1 Binary Search Trees

The abstract data type *dictionary* was introduced in Section 10.1, and in Section 10.5 we saw that when a hash table represents a dictionary, the dictionary operations (find, insert, and erase) take $\Theta(1)$ expected time. However, the worst-case time for these operations is linear in the number n of dictionary entries. A hash table no longer provides good expected performance when we extend the ADT *dictionary* to include operations such as

1. Output in ascending order of keys.
2. Get the k th element in ascending order.
3. Delete the k th element.

To perform operation (1), we need to gather the elements from the table, sort them, and then output them. If a divisor D chained table is used, the elements can be gathered in $O(D + n)$ time, sorted in $O(n \log n)$ time, and output in $O(n)$ time. The total time for operation (1) is therefore $O(D + n \log n)$. If linear open addressing is used for the hash table, the gathering step takes $O(b)$ time where b is the number of buckets. The total time for operation (1) is then $O(b + n \log n)$. Operations (2) and (3) can be done in $O(D + n)$ time when a chained table is used and in $O(b)$ time when linear open addressing is used. To achieve these complexities for operations (2) and (3), we must use a linear-time algorithm to determine the k th element of a collection of n elements (explained in Section 18.2.4).

The basic dictionary operations (find, insert, erase) can be performed in $O(\log n)$ time when a balanced search tree is used. Operation (1) can then be performed in $\Theta(n)$ time. By using an indexed balanced search tree, we can also perform operations (2) and (3) in $O(\log n)$ time. Section 14.6 examines other applications where a balanced tree results in an efficient solution, while a hash table does not.

Rather than jump straight into the study of balanced trees, we first develop a simpler structure called a binary search tree.

Definition 14.1 A **binary search tree** is a binary tree that may be empty. A nonempty binary search tree satisfies the following properties:

1. Every element has a key (or value), and no two elements have the same key; therefore, all keys are distinct.
2. The keys (if any) in the left subtree of the root are smaller than the key in the root.
3. The keys (if any) in the right subtree of the root are larger than the key in the root.
4. The left and right subtrees of the root are also binary search trees. ■

There is some redundancy in this definition. Properties 2, 3, and 4 together imply that the keys must be distinct. Therefore, property 1 can be replaced by the following property: The root has a key. The preceding definition is, however, clearer than the nonredundant version.

Some binary trees in which the elements have distinct keys appear in Figure 14.1. The number inside a node is the element key. The tree of Figure 14.1(a) is not a binary search tree despite the fact that it satisfies properties 1, 2, and 3. The right subtree fails to satisfy property 4. This subtree is not a binary search tree, as its right subtree has a key value (22) that is smaller than the key value in the right subtrees' root (25). The binary trees of Figures 14.1(b) and (c) are binary search trees.

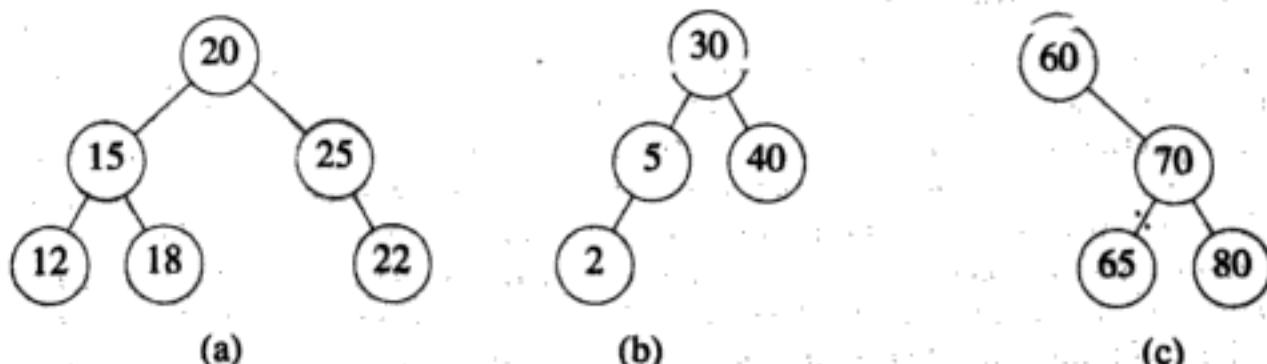


Figure 14.1 Binary trees

We can remove the requirement that all elements in a binary search tree need distinct keys. Now we replace smaller in property 2 by smaller or equal and larger in property 3 by larger or equal; the resulting tree is called a **binary search tree with duplicates**.

14.1.2 Indexed Binary Search Trees

An **indexed binary search tree** is derived from an ordinary binary search tree by adding the field `leftSize` to each tree node. This field gives the number of elements in the node's left subtree. Figure 14.2 shows two indexed binary search trees. The number inside a node is the element key, while that outside is the value of `leftSize`. Notice that `leftSize` also gives the index of an element with respect to the elements in its subtree (this condition is a consequence of the fact that all elements in the left subtree of x precede x). For example, in the tree of Figure 14.2 (a), the elements (in sorted order) in the subtree with root 20 are 12, 15, 18, 20, 25, and 30. Viewed as a linear list (e_0, e_1, \dots, e_4) , the index of the root is 3, which equals its `leftSize` value. In the subtree with root 25, the elements (in sorted order) are 25 and 30, so the index of the root element 25 is 0 and its `leftSize` value is also 0.

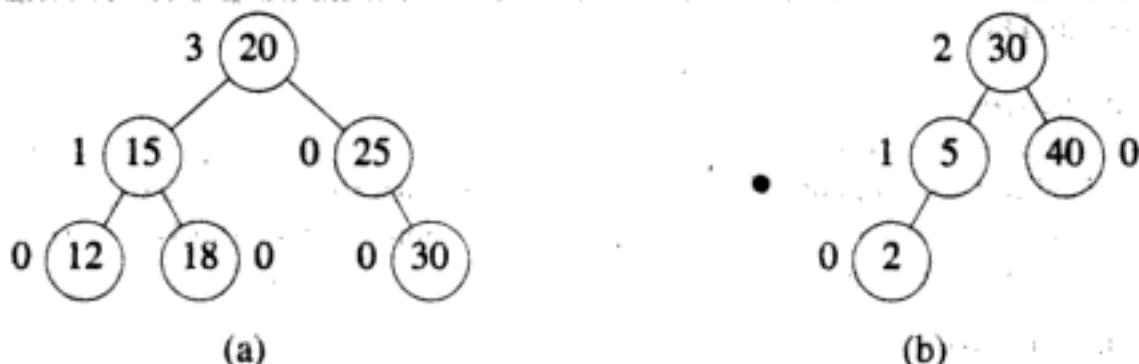


Figure 14.2 Indexed binary search trees

EXERCISES

- Start with a complete binary tree that has 10 nodes. Place the keys [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], one key per node, so that the result is a binary search tree. Label each node with its `leftSize` value.
- Start with a complete binary tree that has 13 nodes. Place the keys [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13], one key per node, so that the result is a binary search tree. Label each node with its `leftSize` value.

14.2 ABSTRACT DATA TYPES

ADT 14.1 gives the abstract data type specification for a binary search tree. An indexed binary search tree supports all the binary search tree operations. In addition, it supports search and deletion by rank. Its abstract data type specification is given in ADT 14.2. The abstract data types `dBSTree` (binary search trees with duplicates) and `dIndexedBSTree` may be specified in a similar way.

Programs 14.1 and 14.2 give C++ abstract classes for the ADTs `bsTree` and `indexedBSTree`, respectively.

EXERCISES

- How much (expected) time does it take to do the `bsTree` operations of ADT 14.1 using skip lists?
- Provide a specification for the abstract data type `dBSTree` (binary search tree with duplicates). Define a C++ abstract class that corresponds to this ADT.

Hidden page

Hidden page

method `inOrderOutput` as is done below.

```
void ascend() {inOrderOutput();}
```

The method `inOrderOutput` first outputs the elements in the left subtree (i.e., smaller elements), then the root element, and finally those in the right subtree (i.e., larger elements). The time complexity is $O(n)$ for an n -element tree.

```
void ascend() {inOrderOutput();}
```

Program 14.3 The method `binarySearchTree<K,E>::ascend`

14.3.2 Searching

Suppose we wish to search for a pair with key `theKey`. We begin at the root. If the root is `NULL`, the search tree contains no pairs and the search is unsuccessful. Otherwise, we compare `theKey` with the key in the root. If `theKey` is less than the key in the root, then no pair in the right subtree can have key value `theKey` and only the left subtree is to be searched. If `theKey` is larger than the key in the root, only the right subtree needs to be searched. If `theKey` equals the key in the root, then the search terminates successfully. The subtrees may be searched similarly. Program 14.4 gives the code. The time complexity is $O(h)$ where h is the height of the tree being searched.

14.3.3 Inserting an Element

To insert a new pair `thePair` into a binary search tree, we must first determine whether its key `thePair.first` is different from those of existing pairs by performing a search for `thePair.first`. If the search is successful, we must replace the old value associated with `thePair.first` with the value `thePair.second`. If the search is unsuccessful, then the new pair is inserted as a child of the last node examined during the search. For instance, to put a pair with key 80 into the tree of Figure 14.1(b), we first search for 80. This search terminates unsuccessfully, and the last node examined is the one with key 40. The new pair is inserted as the right child of this node. The resulting search tree appears in Figure 14.3(a). Figure 14.3(b) shows the result of inserting a pair with key 35 into the search tree of Figure 14.3(a). Program 14.5 implements this strategy to insert a pair into a binary search tree.

14.3.4 Deleting an Element

For deletion we consider the three possibilities for the node `p` that contains the pair that is to be removed: (1) `p` is a leaf, (2) `p` has exactly one nonempty subtree, and (3) `p` has exactly two nonempty subtrees.

Hidden page

```
template<class K, class E>
void binarySearchTree<K,E>::insert(const pair<const K, E>& thePair)
{// Insert thePair into the tree. Overwrite existing
// pair, if any, with same key.
// find place to insert
binaryTreeNode<pair<const K, E> > *p = root,
                                         *pp = NULL;
while (p != NULL)
{// examine p->element
    pp = p;
    // move p to a child
    if (thePair.first < p->element.first)
        p = p->leftChild;
    else
        if (thePair.first > p->element.first)
            p = p->rightChild;
        else
            // replace old value
            p->element.second = thePair.second;
            return;
    }
}
// get a node for thePair and attach to pp
binaryTreeNode<pair<const K, E> > *newNode
    = new binaryTreeNode<pair<const K, E> > (thePair);
if (root != NULL) // the tree is not empty
    if (thePair.first < pp->element.first)
        pp->leftChild = newNode;
    else
        pp->rightChild = newNode;
else
    root = newNode; // insertion into empty tree
treeSize++;
}
```

Program 14.5 Insert an element into a binary search tree

80 is discarded.

Next consider the case when the pair to be removed is in a node *p* that has only one nonempty subtree. If *p* has no parent (i.e., it is the root), the root of its single

subtree becomes the new search tree root. If p has a parent pp , then we change the pointer from pp so that it points to p 's only child. For instance, if we wish to delete the pair with key 5 from the tree of Figure 14.3(b), we change the left-child field of its parent (i.e., the node containing 30) to point to the node containing the 2.

Finally, to remove a pair in a node that has two nonempty subtrees, we replace this pair with either the largest pair in its left subtree or the smallest pair in its right subtree. Following this replacement, the replacing pair is removed from its original node. Suppose we wish to remove the pair with key 40 from the tree of Figure 14.4(a). Either the largest pair (35) from its left subtree or the smallest (60) from its right subtree can replace this pair. If we opt for the smallest pair in the right subtree, we move the pair with key 60 to the node from which the 40 was removed; in addition, the leaf from which the 60 is moved is removed. The resulting tree appears in Figure 14.4(b).

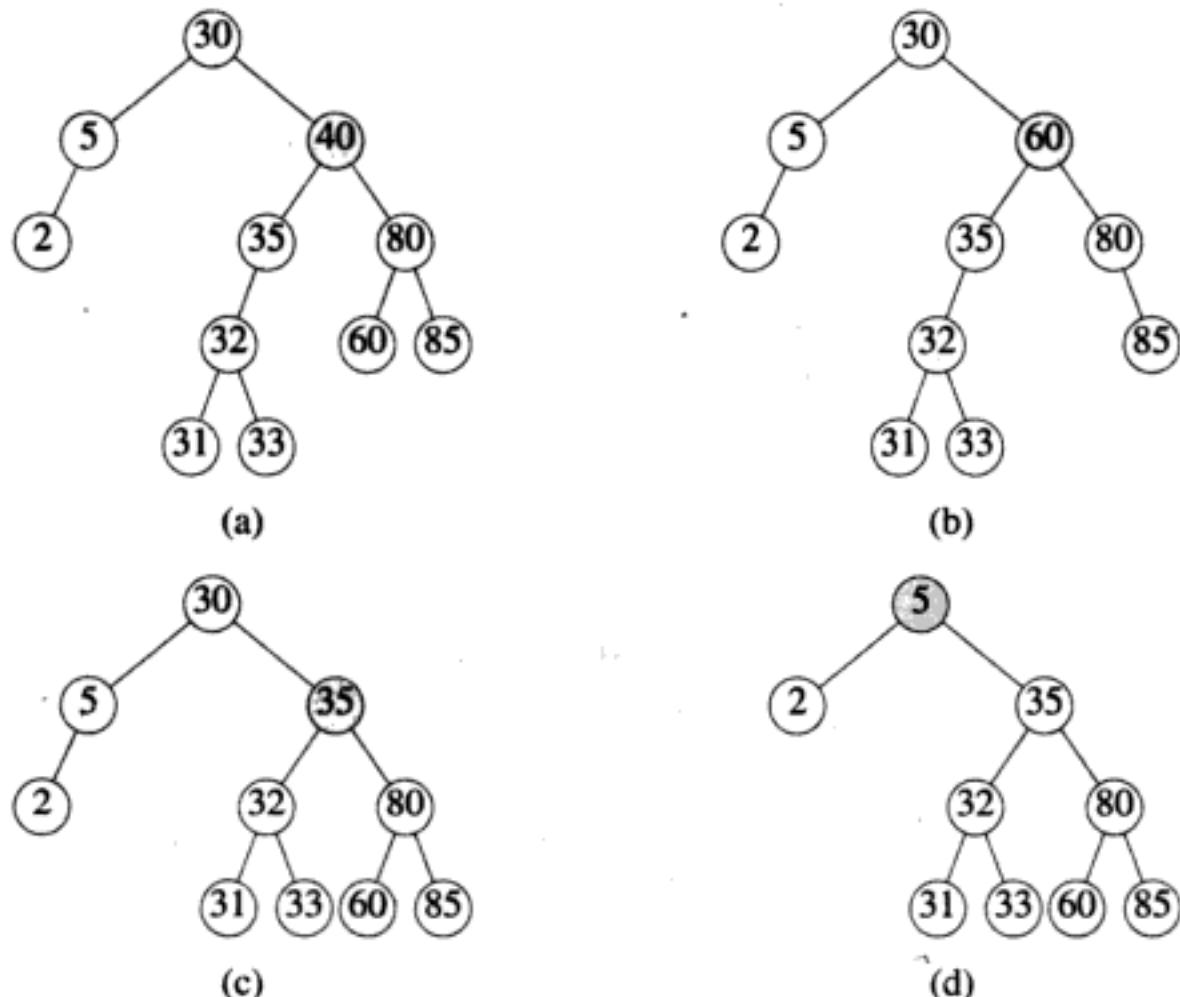


Figure 14.4 Deletion from a binary search tree

Suppose, instead, that when removing 40 from the search tree of Figure 14.4(a),

Hidden page

Hidden page

Hidden page

Hidden page

operations in the order specified by the random permutation. Measure the height of the resulting search tree. Repeat this experiment for several random permutations and compute the average of the measured heights. Compare this figure with $2\lceil \log_2(n+1) \rceil$. For n use the values 100; 500; 1000; 10,000; 20,000; and 50,000.

14. Extend the class `binarySearchTree` by including an iterator that allows you to examine the pairs in ascending order of the key. The time taken to enumerate all pairs of an n -element binary tree should be $O(n)$, the complexity of no method should exceed $O(h)$ where h is the tree height, and the space requirements should be $O(h)$. Show that this is the case. Test your code.
15. Write a method to delete the pair with largest key from a binary search tree. Your method must have time complexity $O(h)$ where h is the height of the binary search tree. Show that your code has this complexity.
 - (a) Use suitable test data to test the correctness of your delete max method.
 - (b) Create a random list of n pairs and a random sequence of insert and delete max operations of length m . Create the latter sequence so that the probability of an insert operation is approximately 0.5 (therefore, the probability of a delete max operation is also 0.5). Initialize a max heap and a binary search tree with duplicates to contain the n pairs in the first random list. Now measure the time to perform the m operations, using the max heap as well as the binary search tree. Divide this time by m to get the average time per operation. Do this experiment for $n = 100, 500, 1000, 2000, \dots$, and 5000. Let m be 5000. Tabulate your computing times.
 - (c) Based on your experiments, what can you say about the relative merits of the two priority queue schemes?

14.4 BINARY SEARCH TREES WITH DUPLICATES

The class for the case when the binary search tree is permitted to contain two or more pairs that have the same key is called `dBinarySearchTree`. We can implement this class by changing the `while` loop of `binarySearchTree<K,E>::insert` (Program 14.5) to that shown in Program 14.7 and by changing the second occurrence of the line

```
if (thePair.first < pp->element.first)
to
if (thePair.first <= pp->element.first)
```

```

while (p != NULL)
{// examine p->element
    pp = p;
    // move p to a child
    if (thePair.first <= p->element.first)
        p = p->leftChild;
    else
        p = p->rightChild;
}

```

Program 14.7 New while loop for Program 14.5 to permit duplicates

If we insert n pairs, all of which have the same key, into an initially empty binary search tree, the result is a left-skewed tree whose height is n . An alternative approach that uses random numbers and results in trees with better height properties is considered in Exercise 17.

EXERCISES

16. Start with an empty binary search tree with duplicates and insert the keys 2, 2, 2, and 2 using the `insert` method described in this section. Draw your tree following each insert. What is the tree height following the insertion of n 2s?
17. Develop a new implementation of the C++ class `dBinarySearchTree`. During an insert, instead of consistently moving to the left subtree of a node that contains a key equal to `thePair.first` (see Program 14.7), use a random number generator to move to the left or right subtree with equal probability. Test the correctness of your implementation.

14.5 INDEXED BINARY SEARCH TREES

The class `indexedBinarySearchTree` may also be defined as a derived class of `linkedBinaryTree` (see Exercise 19). For the class `indexedBinarySearchTree`, the `element` field of a node is a triple `leftSize`, `key`, and `value`.

We can perform an indexed search in a manner similar to that used to search for a pair (a pair is now defined by the fields `key` and `value`). Consider the tree of Figure 14.2(a). Suppose we are looking for the pair whose index is 2. The `leftSize` field (and hence index) of the root is 3. So the pair we desire is in the left subtree. Furthermore, the pair we desire has index 2 in the left subtree. The root, 15, of

the left subtree has `leftSize` = 1. So the pair we desire is in the right subtree of 15. However, the index, in the right subtree of 15, of the pair we desire is no longer 2 because all pairs in the right subtree of 15 follow the pairs in the left subtree of 15 as well as the pair 15. To determine the index of the desired pair in the right subtree, we subtract `leftSize` + 1. Since the `leftSize` value of 15 is 1, the index of the desired pair in its right subtree is $2 - (1 + 1) = 0$. Since `leftSize` = 0 (which equals the index of the pair we seek) for the root of the right subtree of 15, we have found the desired pair 18.

When inserting a pair into an indexed binary search tree, we use a procedure similar to Program 14.5. This time, though, we also need to update `leftSize` fields on the path from the root to the newly inserted node.

A delete by index is done by first performing an indexed search to locate the pair to be deleted. Next we delete the pair as outlined in Section 14.3.4 and update `leftSize` fields on the path from the root to the physically deleted node as necessary.

The time required to find, insert, and delete is $O(h)$ time where h is the height of the indexed search tree.

EXERCISES

18. Start with an empty indexed binary search tree.
 - (a) Insert the keys 4, 12, 8, 16, 6, 18, 24, 2, 14, 3 in this order. Draw the tree following each insert. Show `leftSize` values. Use the insert method described in this section.
 - (b) Use the method of this section to find the keys with index 3, 6, and 8. Describe the search process used in each case.
 - (c) Start with the tree of (a) and remove the keys whose index is 7, 5, and 0 in this order. Draw the search tree following each deletion. Use the deletion method of this section.
19. Develop the C++ class `indexedBinarySearchTree`. You should derive from the abstract class `indexedBSTree`. Test the correctness of your code. Express the time complexity of each method in terms of the number of elements and/or the height of the tree.
20. Do Exercise 19 for the class `dIndexedBinarySearchTree`, which represents an indexed binary search tree with duplicates.
21. You are to represent a linear list as an indexed binary tree which is like an indexed binary search tree except that no node has a key value. Each node of the indexed binary tree contains exactly one element of the linear list. When an indexed binary tree is traversed in inorder, we visit the elements in linear list order from left to right.

Hidden page

Hidden page

```
void main(void)
{// Histogram of nonnegative integer values.
    int n, // number of elements
        r; // values between 0 and r
    cout << "Enter number of elements and range"
        << endl;
    cin >> n >> r;

    // create histogram array h
    int *h = new int[r+1];

    // initialize array h to zero
    for (int i = 0; i <= r; i++)
        h[i] = 0;

    // input data and compute histogram
    for (i = 1; i <= n; i++)
    {// assume input values are between 0 and r
        int key; // input value
        cout << "Enter element " << i << endl;
        cin >> key;
        h[key]++;
    }

    // output histogram
    cout << "Distinct elements and frequencies are"
        << endl;
    for (i = 0; i <= r; i++)
        if (h[i] != 0)
            cout << i << " " << h[i] << endl;
}
```

Program 14.8 Simple histogramming program

Histogramming with a Binary Search Tree

Program 14.8 becomes infeasible when the key range is very large as well as when the key type is not integral (for example, when the keys are real numbers). Suppose we are determining the frequency with which different words occur in a text. The number of possible different words is very large compared to the number that might actually appear in the text. In such a situation we can use hashing to arrive at a solution whose expected complexity is $O(n)$ (Exercise 24). Alternatively, we may

sort the keys and then use a simple left-to-right scan to determine the number of keys for each distinct key value. The sort can be accomplished in $O(n \log n)$ time (using `heapSort` (Program 12.8), for example), and the ensuing left-to-right scan takes $\Theta(n)$ time; the overall complexity is $O(n \log n)$.

The histogramming-by-sorting solution can be improved when the number m of distinct keys is small compared to n . By using balanced search trees such as AVL and red-black trees (see Chapter 15), we can solve the histogramming problem in $O(n \log m)$ time. Furthermore, the balanced search tree solution requires only the distinct keys to be stored in memory. Therefore, this solution is appropriate even in situations when n is so large that we do not have enough memory to accommodate all keys (provided, of course, there is enough memory for the distinct keys).

The solution we describe in this section uses a binary search tree that may not be balanced. Therefore, this solution has expected complexity $O(n \log m)$.

The Class `binarySearchTreeWithVisit`

For our binary search tree solution, we first define the class `binarySearchTreeWithVisit` that extends the class `binarySearchTree` by adding the public method

```
void insert(const pair<const K, E>& thePair, void(*visit)(E&))
```

This method inserts `thePair` into the search tree provided no pair with key equal to `thePair.first` exists. If the tree already contains a pair `p` that has the key `thePair.first`, the method `visit(p.second)` is invoked.

Back to Histogramming with a Binary Search Tree

Program 14.9 gives the code to histogram using a binary search tree. This method inputs the data, enters it into an object of type `binarySearchTreeWithVisit`, and finally outputs the histogram by invoking the `ascend` method. The first component of each pair stored in the binary search tree is its key and the second if the frequency of the key. During a visit of a pair, the frequency count of the key is incremented by 1.

14.6.2 Best-Fit Bin Packing

Using a Binary Search Tree with Duplicates

The best-fit method to pack n objects into bins of capacity c was described in Section 13.5.1. By using a binary search tree with duplicates, we can implement the method to run in $O(n \log n)$ expected time. The worst-case complexity becomes $\Theta(n \log n)$ when a balanced search tree is used.

In our implementation of the best-fit method, the binary search tree (with duplicates) will contain one element for each bin that is currently in use and has nonzero unused capacity. Suppose that when object i is to be packed, there are nine bins (a

```
int main(void)
{// Histogram using a search tree.
    int n; // number of elements
    cout << "Enter number of elements" << endl;
    cin >> n;

    // input elements and enter into tree
    binarySearchTreeWithVisit<int, int> theTree;
    for (int i = 1; i <= n; i++)
    {
        pair<int, int> thePair; // input element
        cout << "Enter element " << i << endl;
        cin >> thePair.first; // key
        thePair.second = 1; // frequency
        // insert thePair in tree unless match already there
        // in latter case increase count by 1
        theTree.insert(thePair, add1);
    }

    // output distinct elements and their counts
    cout << "Distinct elements and frequencies are"
        << endl;
    theTree.ascend();
}
```

Program 14.9 Histogramming with a search tree

through *i*) in use that still have some space left. Let the unused capacity of these bins be 1, 3, 12, 6, 8, 1, 20, 6, and 5, respectively. Notice that it is possible for two or more bins to have the same unused capacity. The nine bins may be stored in a binary search tree with duplicates (i.e., an instance of `dBinarySearchTree`) using as key the unused capacity of a bin.

Figure 14.6 shows a possible binary search tree for the nine bins. For each bin the unused capacity is shown inside a node, and the bin name is shown outside. If the object *i* that is to be packed requires `objectSize[i] = 4` units, we can find the bin that provides the best fit by starting at the root of the tree of Figure 14.6. The root tells us bin *h* has an unused capacity of 6. Since object *i* fits into this bin, bin *h* becomes the candidate for the best bin. Also, since the capacity of all bins in the right subtree is at least 6, we need not look at the bins in this subtree in our quest for the best bin. The search proceeds to the left subtree. The capacity of bin *b* isn't adequate to accommodate our object, so the search for the best bin

moves into the right subtree of bin b . The bin, bin i , at the root of this subtree has enough capacity, and it becomes the new candidate for the best bin. From here the search moves into the left subtree of bin i . Since this subtree is empty, there are no better candidate bins and bin i is selected.

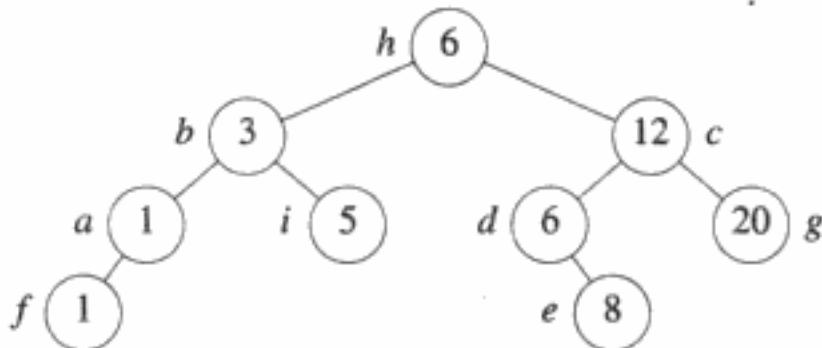


Figure 14.6 Binary search tree with duplicates

As another example of the search for the best bin, suppose `objectSize[i] = 7`. The search again starts at the root. The root bin, bin h , does not have enough capacity for this object, so our quest for a bin moves into the right subtree. Bin c has enough capacity and becomes the new candidate bin. From here we move into c 's left subtree and examine bin d . It does not have enough capacity to accommodate the object, so we continue with the right subtree of d . Bin e has enough capacity and becomes the new candidate bin. We then move into its left subtree, which is empty. The search terminates.

When we find the best bin, we can delete it from the search tree, reduce its capacity by `objectSize[i]`, and reinsert it (unless its remaining capacity is 0). If we do not find a bin with enough capacity, we can start a new bin.

C++ Implementation

To implement this scheme, we can use the class `dBinarySearchTree` to obtain $O(n \log n)$ expected performance or the class `davlTree` (Section 15.1) for $O(n \log n)$ performance in all instances. In either case we must first extend the class to include the public method `findGE(theKey)` that returns the smallest bin capacity that is $\geq \text{theKey}$. This method takes the form given in Program 14.10. Its complexity is $O(\text{height})$. The code for the method is unchanged if the class `davlTree` is extended instead of `dBinarySearchTree`.

The function `bestFitPack` (Program 14.11), which implements the best-fit packing strategy, uses pairs whose second component is a bin identifier (bin number) and whose first component is the unused capacity of the bin.

```

template<class K, class E>
pair<const K, E*>* dBinarySearchTreeWithGE<K,E>::
    findGE(const K& theKey) const
{ // Return pointer to pair with smallest key >= theKey.
// Return NULL if no element has key >= theKey.
    binaryTreeNode<pair<const K, E>> *currentNode = root;
    pair<const K, E> *bestElement = NULL; // element with smallest key
                                            // >= theKey found so far
    // search the tree
    while (currentNode != NULL)
        // is currentNode->element a candidate?
        if (currentNode->element.first >= theKey)
            {// yes, currentNode->element is
            // a better candidate than bestElement
            bestElement = &currentNode->element;
            // smaller keys in left subtree only
            currentNode = currentNode->leftChild;
        }
        else
            // no, currentNode->element.first is too small
            // try right subtree
            currentNode = currentNode->rightChild;

    return bestElement;
}

```

Program 14.10 Finding the smallest key \geq theKey

14.6.3 Crossing Distribution

Channel Routing and Crossings

In the crossing-distribution problem, we start with a routing channel with n pins on both the top and bottom of the channel. Figure 14.7 shows an instance with $n = 10$. The routing region is the shaded rectangular region. The pins are numbered 1 through n , left to right, on both the top and bottom of the channel. In addition, we have a permutation C of the numbers $[1, 2, 3, \dots, n]$. We must use a wire to connect pin i on the top side of the channel to pin C_i on the bottom side. The example in Figure 14.7 is for the case $C = [8, 7, 4, 2, 5, 1, 9, 3, 10, 6]$. The n wires needed to make these connections are numbered 1 through n . Wire i connects top pin i to bottom pin C_i . Wire i is to the left of wire j iff $i < j$.

No matter how we route wires 9 and 10 in the given routing region, these wires

```
void bestFitPack(int *objectSize, int numberOfObjects, int binCapacity)
{// Output best-fit packing into bins of size binCapacity.
// objectSize[1:numberOfObjects] are the object sizes.
    int n = numberOfObjects;
    int binsUsed = 0;
    dBinarySearchTreeWithGE<int,int> theTree; // tree of bin capacities
    pair<int, int> theBin;

    // pack objects one by one
    for (int i = 1; i <= n; i++)
    { // pack object i
        // find best bin
        pair<const int, int> *bestBin = theTree.findGE(objectSize[i]);
        if (bestBin == NULL)
            // no bin large enough, start a new bin
            theBin.first = binCapacity;
            theBin.second = ++binsUsed;
        }
        else
            // remove best bin from theTree
            theBin = *bestBin;
            theTree.erase(bestBin->first);
        }

        cout << "Pack object " << i << " in bin "
            << theBin.second << endl;

        // insert bin in tree unless bin is full
        theBin.first -= objectSize[i];
        if (theBin.first > 0)
            theTree.insert(theBin);
    }
}
```

Program 14.11 Bin packing using best fit

must cross at some point. Crossings are undesirable as special care must be taken at the crossing point to avoid a short circuit. This special care, for instance, might involve placing an insulator at the point of crossing or routing one wire onto another layer at this point and then bringing it back after the crossover point. Therefore, we seek to minimize the number of crossings. You may verify that the minimum number of crossings is made when the wires are run as straight lines as in Figure 14.7.

Each crossing is given by a pair (i, j) where i and j are the two wires that cross.

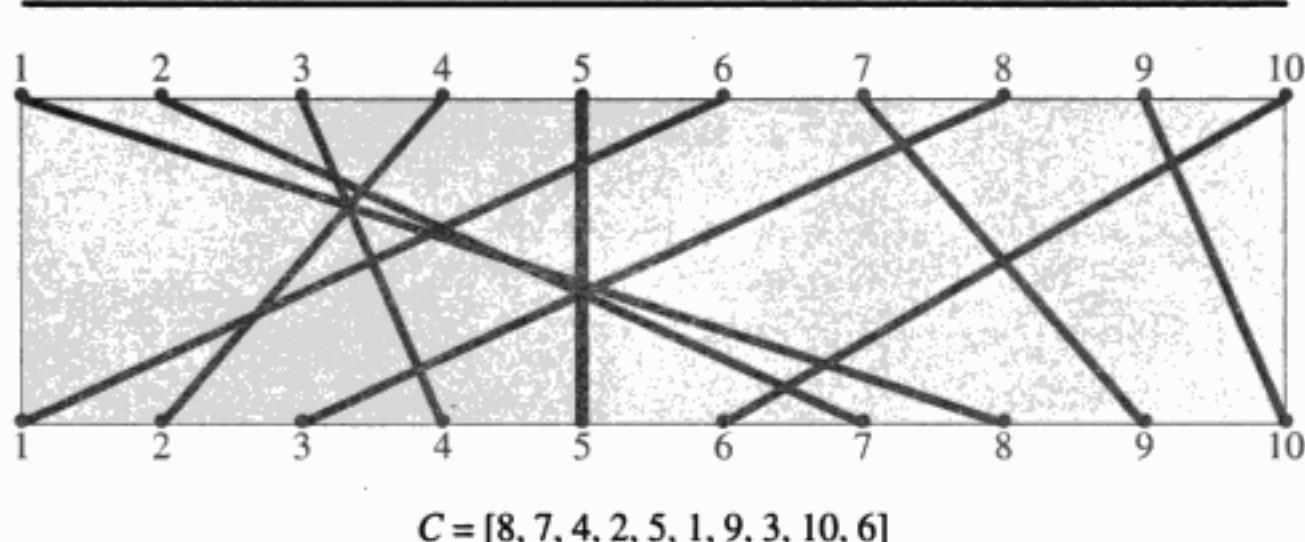


Figure 14.7 A wiring instance

To avoid listing the same pair of crossing wires twice, we require $i < j$ (note that the crossings $(9, 10)$ and $(10, 9)$ are the same). Note that wires i and j , $i < j$, cross iff $C_i > C_j$. Let k_i be the number of pairs (i, j) , $i < j$, such that wires i and j cross. For the example of Figure 14.7, $k_9 = 1$ and $k_{10} = 0$. In Figure 14.8 we list all crossings and k_i values for the example of Figure 14.7. Row i of this table first gives the value of k_i and then the values of j , $i < j$, such that wires i and j cross. The total number of crossings K may be determined by adding together all k_i 's. For our example $K = 22$. Since k_i counts the crossings of wire i only with wires to its right (i.e., $i < j$), k_i gives the number of right-side crossings of wire i .

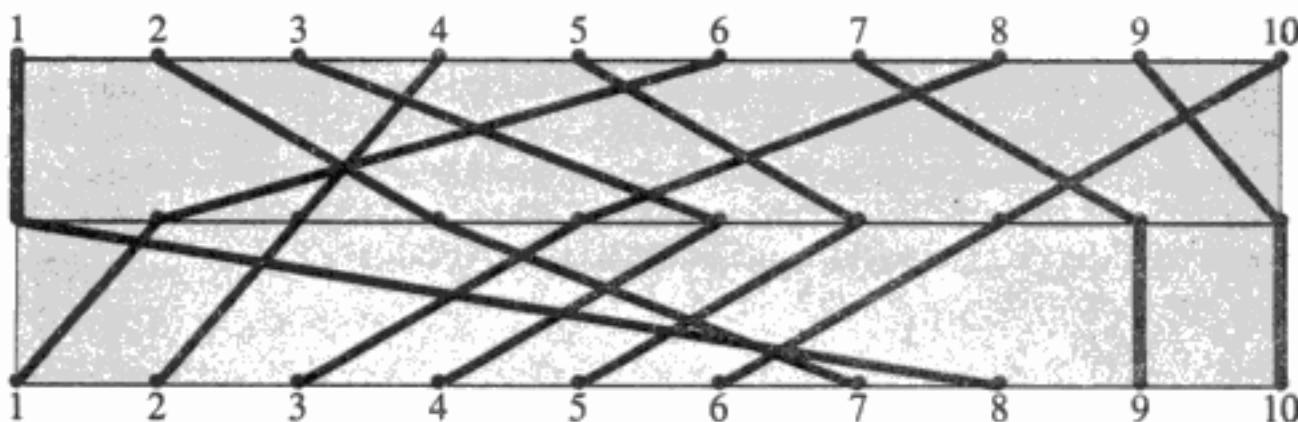
Distributing the Crossings

To balance the routing complexity in the top and lower halves of the channel, we require that each half have approximately the same number of crossings. (The top half should have $\lfloor K/2 \rfloor$ crossings, and the bottom should have $\lceil K/2 \rceil$ crossings.) Figure 14.9 shows a routing of Figure 14.7 in which we have exactly 11 crossings in each half of the channel.

The connections in the top half are given by the permutation $A = [1, 4, 6, 3, 7, 2, 9, 5, 10, 8]$. That is, top pin i is connected to center pin A_i . The connections in the bottom half are given by the permutation $B = [8, 1, 2, 7, 3, 4, 5, 6, 9, 10]$. That is, center pin i is connected to bottom pin B_i . Observe that $C_i = B_{A_i}$, $1 \leq i \leq n$. This equality is essential if we are to accomplish the connections given by C .

In this section we develop algorithms to compute the permutations A and B so that the top half of the channel has $\lfloor K/2 \rfloor$ crossings where K is the total number of crossings.

i	k_i	Crossings						
1	7	2	3	4	5	6	8	10
2	6	3	4	5	6	8	10	
3	3	4	6	8				
4	1	6						
5	2	6	7					
6	0							
7	2	8	10					
8	0							
9	1	10						
10	0							

Figure 14.8 Crossing table**Figure 14.9** Splitting the crossings

Crossing Distribution Using a Linear List

The crossing numbers k_i and the total number of crossings K can be computed in $\Theta(n^2)$ time by examining each wire pair (i, j) . The partitioning of C into A and B can then be computed by using an `arrayList` as in Program 14.12.

In the `while` loop, we scan the wires from right to left, determining their relative order at the center of the routing channel. The objective is to have a routing order at the center that requires exactly `crossingsNeeded = theK/2` crossings in the top half of the routing channel.

The linear list `theList` keeps track of the current ordering, at the center, of the

```
void main(void)
{
    // define the instance to be solved
    // connections at bottom of channel, theC[1:10]
    int theC[] = {0, 8, 7, 4, 2, 5, 1, 9, 3, 10, 6};
    // crossing numbers, k[1:10]
    int k[] = {0, 7, 6, 3, 1, 2, 0, 2, 0, 1, 0};
    int n = 10;           // number of pins on either side of channel
    int theK = 22;        // total number of crossings

    // create data structures
    arrayList<int> theList(n);
    int *theA = new int[n + 1],   // top-half permutation
        *theB = new int[n + 1],   // bottom-half permutation
        *theX = new int[n + 1];   // center connections

    int crossingsNeeded = theK / 2; // remaining number of crossings
                                   // needed in top half

    // scan wires right to left
    int currentWire = n;
    while (crossingsNeeded > 0)
    { // need more crossings in top half
        if (k[currentWire] < crossingsNeeded)
            { // use all crossings from currentWire
                theList.insert(k[currentWire], currentWire);
                crossingsNeeded -= k[currentWire];
            }
        else
            { // use only crossingsNeeded crossings from currentWire
                theList.insert(crossingsNeeded, currentWire);
                crossingsNeeded = 0;
            }
        currentWire--;
    }

    // determine wire permutation at center
    // first currentWire wires have same ordering
    for (int i = 1; i <= currentWire; i++)
        theX[i] = i;
```

Program 14.12 Crossing distribution using a linear list (continues)

Hidden page

Hidden page

$O(n \log n)$, we may use an indexed binary search tree rather than an indexed balanced search tree. The technique is the same in both cases. We will use an indexed binary search tree to illustrate this technique.

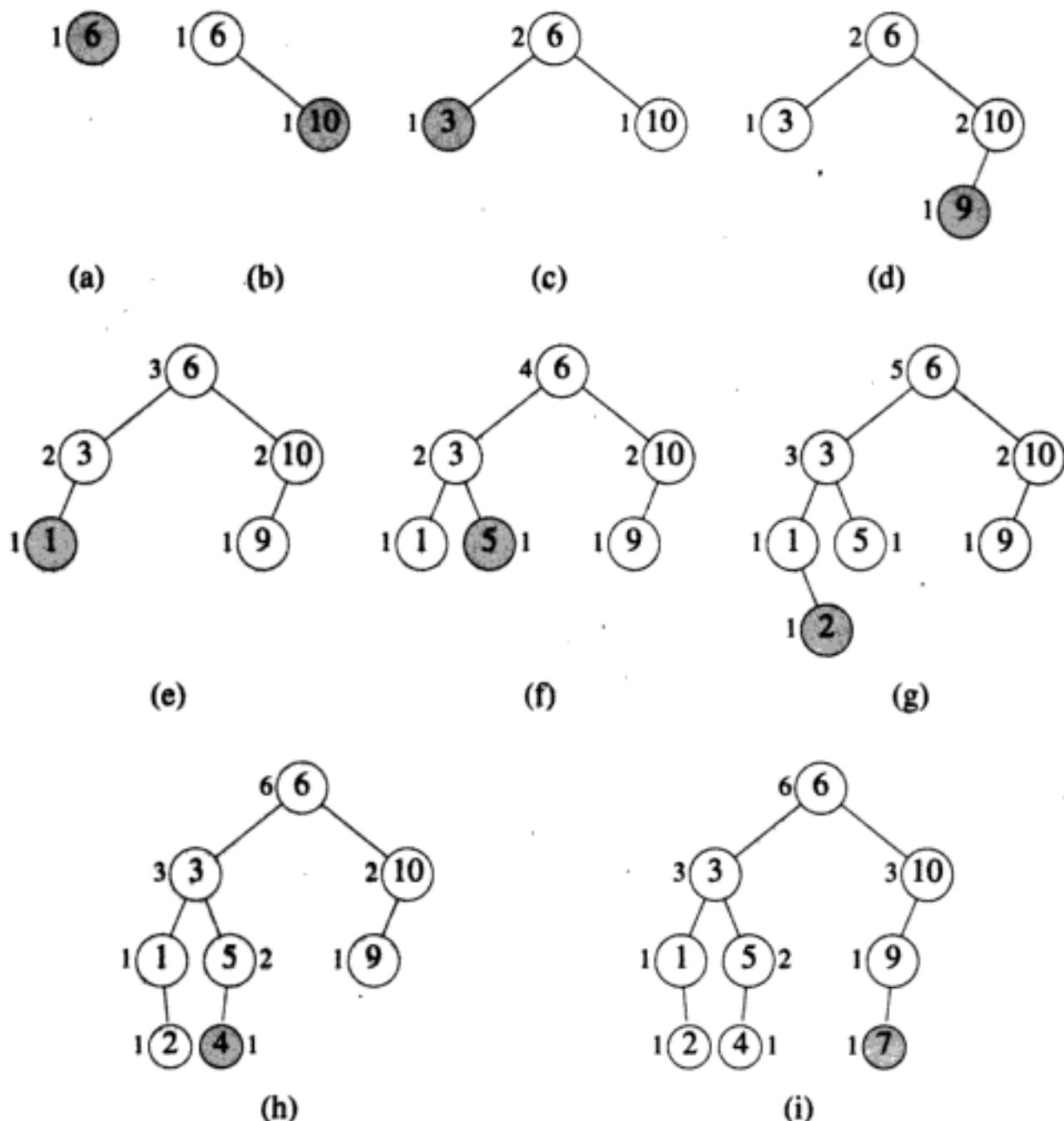
Using an Indexed Binary Search Tree

First let us see how to compute the crossing numbers k_i , $1 \leq i \leq n$. Suppose we examine the wires in the order $n, n - 1, \dots, 1$ and put C_i into an indexed search tree when wire i is examined. For the example of Figure 14.7, we start with an empty tree. We examine wire 10 and insert $C_{10} = 6$ into an empty tree to get the tree of Figure 14.10(a). The number outside the node is its `leftSize` value, and that inside the node is its key (or C value). Note that k_n is always 0, so we set $k_n = 0$. Next we examine wire 9 and insert $C_9 = 10$ into the tree to get the tree of Figure 14.10(b). To make the insertion, we pass over the root that has `leftSize` = 1. From this `leftSize` value, we know that wire 9's bottom endpoint is to the right of exactly one of the wires seen so far, so $k_9 = 1$. We examine wire 8 and insert $C_8 = 3$ to get the tree of Figure 14.10(c). Since C_8 is the smallest entry in the tree, no wires are crossed and $k_8 = 0$. For wire 7, $C_7 = 9$ is inserted to obtain tree (d). C_7 becomes the third-smallest entry in the tree. We can determine that its bottom endpoint is to the right of two others in the tree. As a result, $k_7 = 2$. Proceeding in this way, the trees of Figures 14.10(e) through 14.10(i) are generated when we examine wires 6 through 2. Finally, when we examine wire 1, we insert $C_1 = 8$ as the right child of the node with key 7. The sum of the `leftSize` values of the nodes whose right subtrees we enter is $6 + 1 = 7$. Wire 1 has a bottom endpoint that is to the right of seven of the wires in the tree, and so $k_1 = 7$.

The time needed to examine wire i and compute k_i is $O(h)$ where h is the current height of the indexed search tree. So all k_i 's can be computed in expected time $O(n \log n)$ by using an indexed binary search tree or in time $O(n \log n)$ by using an indexed balanced search tree.

To compute A , we can implement the code of Program 14.12, using an indexed binary tree representation of a linear list. To list the elements in order of rank, we can do an inorder traversal. The expected time needed by this implementation of Program 14.12 is $O(n \log n)$ when the linear list implementation of Exercise 22 is used; the worst-case time is $O(n \log n)$ when an indexed balanced tree implementation (see Exercise 20 in Chapter 15) is used.

Another way to obtain the permutation A is to first compute $r = \sum_{i=1}^n k_i / 2$ and $s = \text{smallest } i \text{ such that } \sum_{l=i}^n k_l \leq r$. For our example, $r = 11$ and $s = 3$. We see that Program 14.12 does all crossings (i.e., with wires whose top point is to the right) for wires $n, n - 1, \dots, s$ and $r - \sum_{l=s}^n k_l$ of the crossings for wire $s - 1$ in the top half. The remaining crossings are done in the bottom half. To get these top-half crossings, we examine the tree following the insertion of C_s . For our

**Figure 14.10** Computing the number of crossings

example we examine tree (h) of Figure 14.10. An inorder traversal of tree (h) yields the sequence $(1, 2, 3, 4, 5, 6, 9, 10)$. Replacing these bottom endpoints with the corresponding wire numbers, we get the sequence $(6, 4, 8, 3, 5, 10, 7, 9)$, which gives the wire permutation following the nine crossings represented in Figure 14.10(h). For the additional two crossings, we insert wire $s = 2$ after the second wire in the sequence to get the new wire sequence $(6, 4, 2, 8, 3, 5, 10, 7, 9)$. The remaining

wires 1 through $s - 1$ are added at the front to obtain $(1, 6, 4, 2, 8, 3, 5, 10, 7, 9)$, which is the `theX` permutation computed in Program 14.12. To obtain `theX` in this way, we need to rerun part of the code used to compute the k_i s, perform an inorder traversal, insert wire s , and add a few wires at the front of the sequence. The time needed for all of these steps is $O(n \log n)$. `theA` and `theB` may be obtained from `theX` in linear time using the last two `for` loops of Program 14.12.

EXERCISES

23. Write a histogramming program that first inputs the n keys into an array, then sorts this array, and finally makes a left-to-right scan of the array outputting the distinct key values and the number of times each occurs.
24. Write a histogramming program that uses a chained hash table, rather than a binary search tree as in Program 14.9, to store the distinct keys and their frequencies. Compare the run-time performance of your new program with that of Program 14.9.
25. (a) Extend the class `dBinarySearchTree` by adding the public method `eraseGE(theKey)`, which deletes the pair with smallest key $\geq \text{theKey}$. The deleted pair is returned.
(b) Use `eraseGE` (and not `findGE`) to obtain a new version of `bestFitPack`.
(c) Which will run faster? Why?
26. Write the crossing table (see Figure 14.8) for the permutation $C[1 : 10] = [6, 4, 5, 8, 3, 2, 10, 9, 1, 7]$. Compute the top-half and bottom-half permutations A and B that balance the number of crossings in the top and lower halves of the channel.
27. Do Exercise 26 with $C[1 : 10] = [10, 9, 8, 1, 2, 3, 7, 6, 5, 4]$.
28. (a) Use an indexed binary search tree to obtain an $O(n \log n)$ expected time solution for the crossing-distribution problem.
(b) Test the correctness of your code.
(c) Compare the actual run time of this solution to that of Program 14.12. Do this comparison using randomly generated permutations C and $n = 1000; 10,000; \text{ and } 50,000$.
29. Write a program to create a concordance for a piece of text (see Exercise 48 in Chapter 10). Use a binary search tree to construct the concordance entries and then perform an inorder traversal of this AVL tree to output the concordance entries in sorted order. Comment on the merits of using a binary search tree versus a hash table. In particular, what can you say about the expected performance of the two approaches when the input text has n words and only $m \leq n$ of these are distinct?

CHAPTER 15

BALANCED SEARCH TREES

BIRD'S-EYE VIEW

This last chapter on trees develops balanced tree structures—tree structures whose height is $O(\log n)$. We develop two balanced binary tree structures, AVL and red black, and one tree structure, B-tree, whose degree is more than 2. AVL and red-black trees are suitable for internal memory applications, and the B-tree is suitable for external memory (e.g., a large dictionary that resides on disk) applications. These balanced structures allow you to perform dictionary operations as well as by-rank operations in $O(\log n)$ time in the worst case. When a linear list is represented as an indexed balanced tree, the *get*, *insert*, and *erase* operations take $O(\log n)$ time.

The splay tree is another data structure that is developed in this chapter. Although the height of a splay tree is $O(n)$ and an individual dictionary operation performed on a splay tree takes $O(n)$ time, every sequence of u operations performed on a splay tree takes $O(u \log u)$ time. The asymptotic complexity of a sequence of u operations is the same regardless of whether you use splay trees, AVL trees, or red-black trees.

The following table summarizes the asymptotic performance of the various dictionary structures considered in this text. All complexities are theta of the given function.

Method	Worst Case			Expected		
	Search	Insert	Delete	Search	Insert	Delete
sorted array	$\log n$	n	n	$\log n$	n	n
sorted chain	n	n	n	n	n	n
skip lists	n	n	n	$\log n$	$\log n$	$\log n$
hash tables	n	n	n	1	1	1
binary search tree	n	n	n	$\log n$	$\log n$	$\log n$
AVL tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
red-black tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
splay tree	n	n	n	$\log n$	$\log n$	$\log n$
B-trees	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$

The STL classes `map` and `multimap` use red-black trees to guarantee logarithmic performance for the search, insert, and delete operations.

In practice, we expect hashing to outperform balanced search trees when the desired operations are search, insert, and delete (all by key value); therefore, hashing is the preferred method in these applications. When the dictionary operations are done solely by key value, balanced search trees are recommended only in time-critical applications in which we must guarantee that no dictionary operation ever takes more than a specified amount of time. Balanced search trees are also recommended when the search and delete operations are done by rank and for applications in which the dictionary operations are not done by exact key match. An example of the latter would be finding the smallest element with a key larger than k .

The actual run-time performance of both AVL and red-black trees is similar; in comparison, splay trees take less time to perform a sequence of u operations. In addition, splay trees are easier to implement.

Both AVL and red-black trees use “rotations” to maintain balance. AVL trees perform at most one rotation following an insert and $O(\log n)$ rotations following a delete. However, red-black trees perform a single rotation following either an insert or delete. This difference is not important in most applications where a rotation takes $\Theta(1)$ time. It does, however, become important in advanced applications where a rotation cannot be performed in constant time. One such application is the implementation of the balanced priority search trees of McCreight. These priority search trees are used to represent elements with two-dimensional keys. In this case each key is a pair (x, y) . A priority search tree is a tree that is simultaneously a min (or max) tree on y and a search tree on x . When rotations are performed in these trees, each has a cost of $O(\log n)$. Since red-black trees perform a single rotation following an insert or delete, the overall insert or delete time remains $O(\log n)$ if we use a red-black tree to represent a priority search tree. When we use an AVL tree, the time for the delete operation becomes $O(\log^2 n)$.

Although AVL trees, red-black trees, and splay trees provide good performance when the dictionary being represented is sufficiently small to fit in our computer’s memory, they are quite inadequate for larger dictionaries. When the dictionary resides on disk, we need to use search trees with a much higher degree and hence a

much smaller height. An example of such a search tree, the B-tree, is also considered in this chapter.

Although C++ codes are not given for any of the data structures developed in this chapter, several codes are available on the Web site as solutions to exercises. The Web site also has material on other search structures such as tries and suffix trees.

This chapter does not have an applications section because the applications of balanced search trees are the same as those of binary search trees (Chapter 14). By using a balanced search tree in these applications, we obtain code whose worst-case asymptotic complexity is the same as the expected complexity when unbalanced binary search trees are used.

15.1 AVL TREES

15.1.1 Definition

We can guarantee $O(\log n)$ performance for the search, insert and delete operations of a search tree by ensuring that the search tree height is always $O(\log n)$. Trees with a worst-case height of $O(\log n)$ are called **balanced trees**. One of the more popular balanced trees, known as an **AVL tree**, was introduced in 1962 by Adelson-Velskii and Landis.

Definition 15.1 *An empty binary tree is an AVL tree. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is an AVL tree iff (1) T_L and T_R are AVL trees and (2) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively.* ■

An **AVL search tree** is a binary search tree that is also an AVL tree. Trees (a) and (b) of Figure 14.1 are AVL trees, while tree (c) is not. Tree (a) is not an AVL search tree, as it is not a binary search tree. Tree (b) is an AVL search tree. The trees of Figure 14.3 are AVL search trees.

An **indexed AVL search tree** is an indexed binary search tree that is also an AVL tree. Both the search trees of Figure 14.2 are indexed AVL search trees. In the remainder of this section, we will not consider indexed AVL search trees explicitly. However, the techniques we develop carry over in a rather straightforward manner to such trees. We will use the terms *insert* and *put* as well as the terms *remove* and *delete* interchangeably.

If we are to use AVL search trees to represent a dictionary and perform each dictionary operation in logarithmic time, then we must establish the following properties:

1. The height of an AVL tree with n elements/nodes is $O(\log n)$.
2. For every value of n , $n \geq 0$, there exists an AVL tree. (Otherwise, some insertions cannot leave behind an AVL tree, as no such tree exists for the current number of elements.)
3. An n -element AVL search tree can be searched in $O(\text{height}) = O(\log n)$ time.
4. A new element can be inserted into an n -element AVL search tree so that the result is an $n + 1$ element AVL tree and such an insertion can be done in $O(\log n)$ time.
5. An element can be deleted from an n -element AVL search tree, $n > 0$, so that the result is an $n - 1$ element AVL tree and such a deletion can be done in $O(\log n)$ time.

Property 2 follows from property 4, so we will not show property 2 explicitly. Properties 1, 3, 4, and 5 are established in the following subsections.

15.1.2 Height of an AVL Tree

We will obtain a bound on the height of an AVL tree that has n nodes in it. Let N_h be the minimum number of nodes in an AVL tree of height h . In the worst case the height of one of the subtrees is $h - 1$, and the height of the other is $h - 2$. Both these subtrees are also AVL trees. Hence

$$N_h = N_{h-1} + N_{h-2} + 1, \quad N_0 = 0, \quad \text{and} \quad N_1 = 1$$

Notice the similarity between this definition for N_h and the definition of the Fibonacci numbers

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0, \quad \text{and} \quad F_1 = 1$$

It can be shown (see Exercise 9) that $N_h = F_{h+2} - 1$ for $h \geq 0$. From Fibonacci number theory we know that $F_h \approx \phi^h/\sqrt{5}$ where $\phi = (1 + \sqrt{5})/2$. Hence $N_h \approx \phi^{h+2}/\sqrt{5} - 1$. If there are n nodes in the tree, then its height h is at most $\log_\phi(\sqrt{5}(n+1)) - 2 \approx 1.44 \log_2(n+2) = O(\log n)$.

15.1.3 Representation of an AVL Tree

AVL trees are normally represented by using the linked representation scheme for binary trees. However, to facilitate insertion and deletion, a balance factor bf is associated with each node. The balance factor $bf(x)$ of a node x is defined as

$$\text{height of left subtree of } x - \text{height of right subtree of } x$$

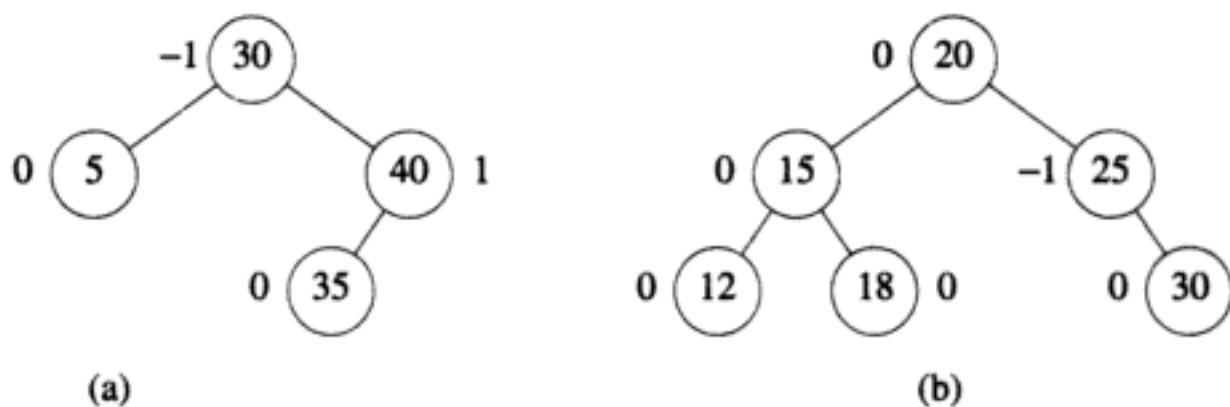
From the definition of an AVL tree, it follows that the permissible balance factors are -1 , 0 , and 1 . Figure 15.1 shows two AVL search trees and the balance factors for each node.

15.1.4 Searching an AVL Search Tree

To search an AVL search tree, we may use the code of Program 14.4 without change. Since the height of an AVL tree with n elements is $O(\log n)$, the search time is $O(\log n)$.

15.1.5 Inserting into an AVL Search Tree

If we use the strategy of Program 14.5 to insert an element into an AVL search tree, the tree following the insertion may no longer be AVL. For instance, when an element with key 32 is inserted into the AVL tree of Figure 15.1(a), the new search tree is the one shown in Figure 15.2(a). Since this tree contains nodes with balance



The number outside each node is its balance factor

Figure 15.1 AVL search trees

factors other than -1 , 0 , and 1 , it is not an AVL tree. When an insertion into an AVL tree using the strategy of Program 14.5 results in a search tree that has one or more nodes with balance factors other than -1 , 0 , and 1 , the resulting search tree is **unbalanced**. We can restore balance (i.e., make all balance factors -1 , 0 , and 1) by shifting some of the subtrees of the unbalanced tree as in Figure 15.2(b).

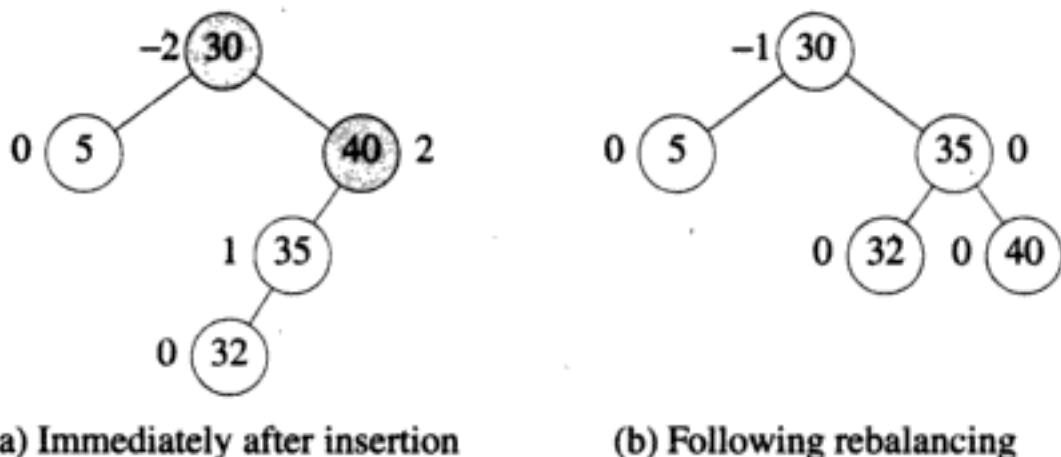


Figure 15.2 Sample insertion into an AVL search tree

Before examining the subtree movement needed to restore balance, let us make some observations about the unbalanced tree that results from an insertion. I1 denotes insertion observation 1.

- I1: In the unbalanced tree the balance factors are limited to -2 , -1 , 0 , 1 , and 2 .
- I2: A node with balance factor 2 had a balance factor 1 before the insertion.

Similarly, a node with balance factor -2 had a balance factor -1 before the insertion.

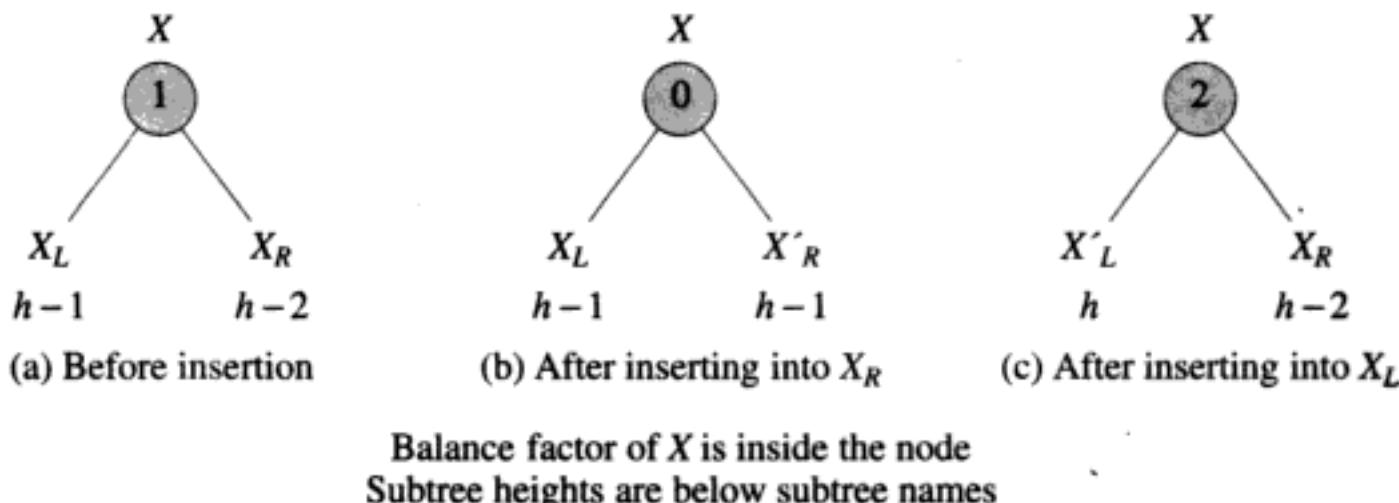
- I3: The balance factors of only those nodes on the path from the root to the newly inserted node can change as a result of the insertion.
- I4: Let A denote the nearest ancestor of the newly inserted node whose balance factor is either -2 or 2 . (In the case of Figure 15.2(a), the A node is the node with key 40.) The balance factor of all nodes on the path from A to the newly inserted node was 0 prior to the insertion.

Node A (see I4) may be identified while we are moving down from the root searching for the place to insert the new element. From I2 it follows that $bf(A)$ was either -1 or 1 prior to the insertion. Let X denote the last node encountered that has such a balance factor. When inserting 32 into the AVL tree of Figure 15.1(a), X is the node with key 40; when inserting 22, 28, or 50 into the AVL tree of Figure 15.1(b), X is the node with key 25; and when inserting 10, 14, 16, or 19 into the AVL tree of Figure 15.1(b), there is no node X .

When node X does not exist, all nodes on the path from the root to the newly inserted node have balance factor 0 prior to the insertion. The tree cannot be unbalanced following the insertion because an insertion changes balance factors by -1 , 0 , or 1 , and only balance factors on the path from the root may change. Therefore, if the tree is unbalanced following the insertion, X exists. If $bf(X) = 0$ after the insertion, then the height of the subtree with root X is the same before and after the insertion. For example, if this height was h before the insertion and if $bf(X) = 1$, the height of its left subtree X_L was $h - 1$ and that of its right subtree X_R was $h - 2$ before the insertion (see Figure 15.3(a)). For the balance factor to become 0 , the insertion must be made in X_R resulting in an X'_R of height $h - 1$ (see Figure 15.3(b)). The height of X'_R must increase to $h - 1$ as all balance factors on the path from X to the newly inserted node were 0 prior to the insertion. The height of X remains h , and the balance factors of the ancestors of X are the same before and after the insertion, so the tree is balanced.

The only way the tree can become unbalanced is if the insertion causes $bf(X)$ to change from -1 to -2 or from 1 to 2 . For the latter case to occur, the insertion must be made in the left subtree X_L of X (see Figure 15.3(c)). Now the height of X'_L must become h (as all balance factors on the path from X to the newly inserted node were 0 prior to the insertion). Therefore, the A node referred to in observation I4 is X .

When the A node has been identified, the imbalance at A can be classified as either an L (the newly inserted node is in the left subtree of A) or R type imbalance. This imbalance classification may be refined by determining which grandchild of A is on the path to the newly inserted node. Notice that such a grandchild exists, as the height of the subtree of A that contains the new node must be at least 2 for the balance factor of A to be -2 or 2 . With this refinement of the imbalance classification, the imbalance at A is of one of the types LL (new node is in the left

**Figure 15.3** Inserting into an AVL search tree

subtree of the left subtree of A), LR (new node is in the right subtree of the left subtree of A), RR, and RL.

A generic LL type imbalance appears in Figure 15.4. Figure 15.4(a) shows the conditions before the insertion, and Figure 15.4(b) shows the situation following the insertion of an element into the left subtree B_L of B . The subtree movement needed to restore balance at A appears in Figure 15.4(c). B becomes the root of the subtree that A was previously root of, B'_L remains the left subtree of B , A becomes the root of B 's right subtree, B_R becomes the left subtree of A , and the right subtree of A is unchanged. The balance factors of nodes in B'_L that are on the path from B to the newly inserted node change as does the balance factor of A . The remaining balance factors are the same as before the rotation. Since the heights of the subtrees of Figures 15.4(a) and (c) are the same, the balance factors of the ancestors (if any) of this subtree are the same as before the insertion. So no nodes with a balance factor other than -1 , 0 , or 1 remain. A single LL rotation has rebalanced the entire tree! You may verify that the rebalanced tree is indeed a binary search tree.

Figure 15.5 shows a generic LR type imbalance. Since the insertion took place in the right subtree of B , this subtree cannot be empty following the insertion; therefore, node C exists. However, its subtrees C_L and C_R may be empty. The rearrangement of subtrees needed to rebalance appears in Figure 15.5(c). The values of $bf(B)$ and $bf(A)$ following the rearrangement depend on the value, b , of $bf(C)$ just after the insertion but before the rearrangement. The figure gives these values as a function of b . The rearranged subtree is seen to be a binary search tree. Also, since the heights of the subtrees of Figures 15.5(a) and (c) are the same, the balance factors of their ancestors (if any) are the same before and after the insertion. So a single LR rotation at A rebalances the entire tree.

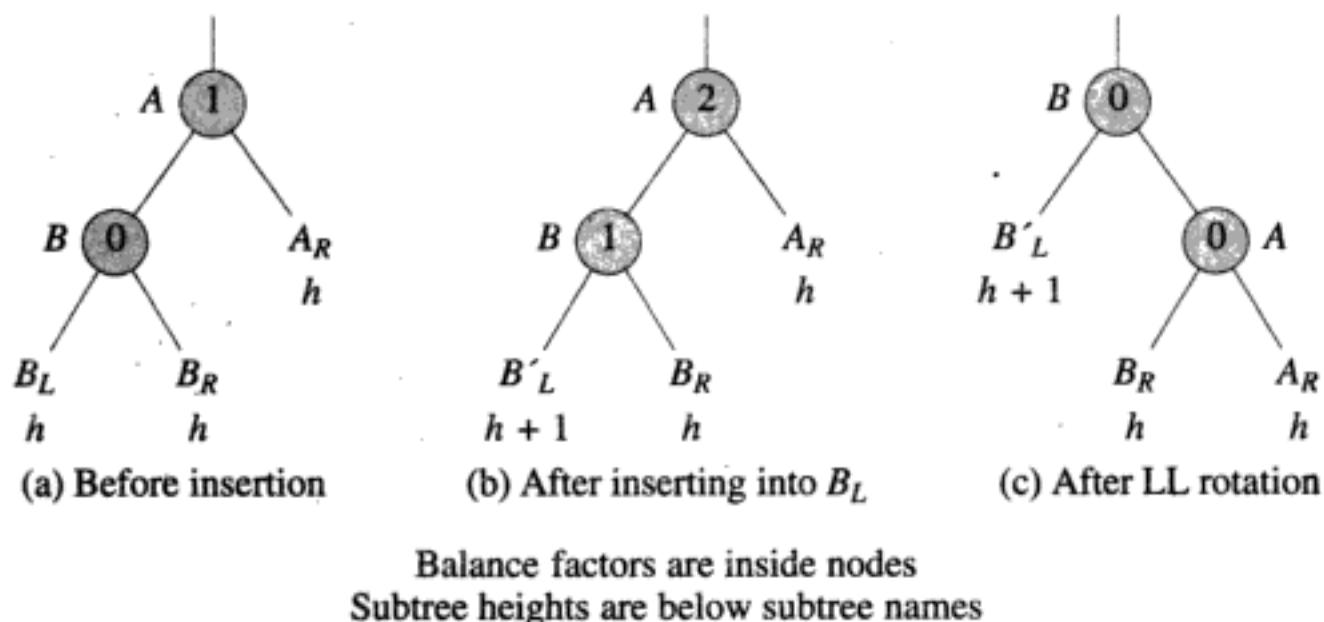
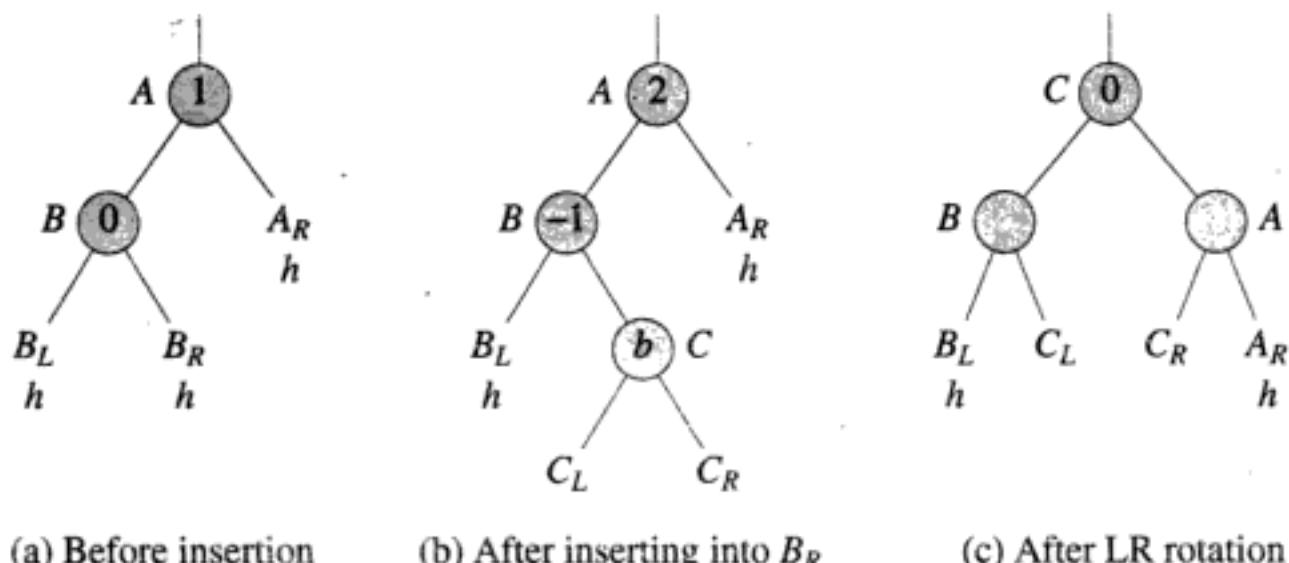


Figure 15.4 An LL rotation



$b = 0 \Rightarrow bf(B) = bf(A) = 0$ after rotation
 $b = 1 \Rightarrow bf(B) = 0$ and $bf(A) = -1$ after rotation
 $b = -1 \Rightarrow bf(B) = 1$ and $bf(A) = 0$ after rotation

Figure 15.5 An LR rotation

The cases RR and RL are symmetric to the ones we have just seen. The transformations done to remedy LL and RR imbalances are often called **single rotations**, while those done for LR and RL imbalances are called **double rotations**. The transformation for an LR imbalance can be viewed as an RR rotation followed by an LL rotation, while that for an RL imbalance can be viewed as an LL rotation followed by an RR rotation (see Exercise 13).

The steps in the AVL search-tree-insertion algorithm that results from our discussion appear in Figure 15.6. These steps can be refined into C++ code that has a complexity of $O(\text{height}) = O(\log n)$. Notice that a single rotation (LL, LR, RR, or RL) is sufficient to restore balance if the insertion causes imbalance.

- Step 1:** Find the place to insert the new element by following a path from the root as in a search for an element with the same key. During this process, keep track of the most recently seen node with balance factor -1 or 1 . Let this node be A . If an element with the same key is found, the insert fails and the remaining steps are not performed.
- Step 2:** If there is no node A , then make another pass from the root, updating balance factors. Terminate following this pass.
- Step 3:** If $bf(A) = 1$ and the new node was inserted in the right subtree of A or if $bf(A) = -1$ and the insertion took place in the left subtree, then the new balance factor of A is 0 . In this case update balance factors on the path from A to the new node and terminate.
- Step 4:** Classify the imbalance at A and perform the appropriate rotation. Change balance factors as required by the rotation as well as those of nodes on the path from the new subtree root to the newly inserted node.

Figure 15.6 Steps for AVL search tree insertion

15.1.6 Deletion from an AVL Search Tree

To delete an element from an AVL search tree, we proceed as in Program 14.6. Let q be the parent of the node that was physically deleted. For example, if the element with key 25 is deleted from the tree of Figure 15.1(b), the node containing this element is deleted and the right-child pointer from the root diverted to the only child of the deleted node. The parent of the deleted node is the root, so q is the root. If instead, the element with key 15 is deleted, its spot is used by the element with key 12 and the node previously containing this element is deleted. Now q is the node that originally contained 15 (i.e., the left child of the root). Since the balance factors of some (or all) of the nodes on the path from the root to q have changed

as a result of the deletion, we retrace this path backward from q toward the root.

If the deletion took place from the left subtree of q , $bf(q)$ decreases by 1, and if it took place from the right subtree, $bf(q)$ increases by 1. We may make the following observations (D1 denotes deletion observation 1):

- D1: If the new balance factor of q is 0, its height has decreased by 1; we need to change the balance factor of its parent (if any) and possibly those of its other ancestors.
- D2: If the new balance factor of q is either -1 or 1 , its height is the same as before the deletion and the balance factors of its ancestors are unchanged.
- D3: If the new balance factor of q is either -2 or 2 , the tree is unbalanced at q .

Since balance factor changes may propagate up the tree along the path from q to the root (see observation D1), it is possible for the balance factor of a node on this path to become -2 or 2 . Let A be the first such node on this path. To restore balance at node A , we classify the type of imbalance. The imbalance is of type L if the deletion took place from A 's left subtree. Otherwise, it is of type R. If $bf(A) = 2$ after the deletion, it must have been 1 before. So A has a left subtree with root B . A type R imbalance is subclassified into the types R0, R1, and R -1 , depending on $bf(B)$. The type R -1 , for instance, refers to the case when the deletion took place from the right subtree of A and $bf(B) = -1$. Similarly, type L imbalances are subclassified into the types L0, L1, and L -1 .

An R0 imbalance at A is rectified by performing the rotation shown in Figure 15.7. Notice that the height of the shown subtree was $h + 2$ before the deletion and is $h + 2$ after. So the balance factors of the remaining nodes on the path to the root are unchanged. As a result, the entire tree has been rebalanced.

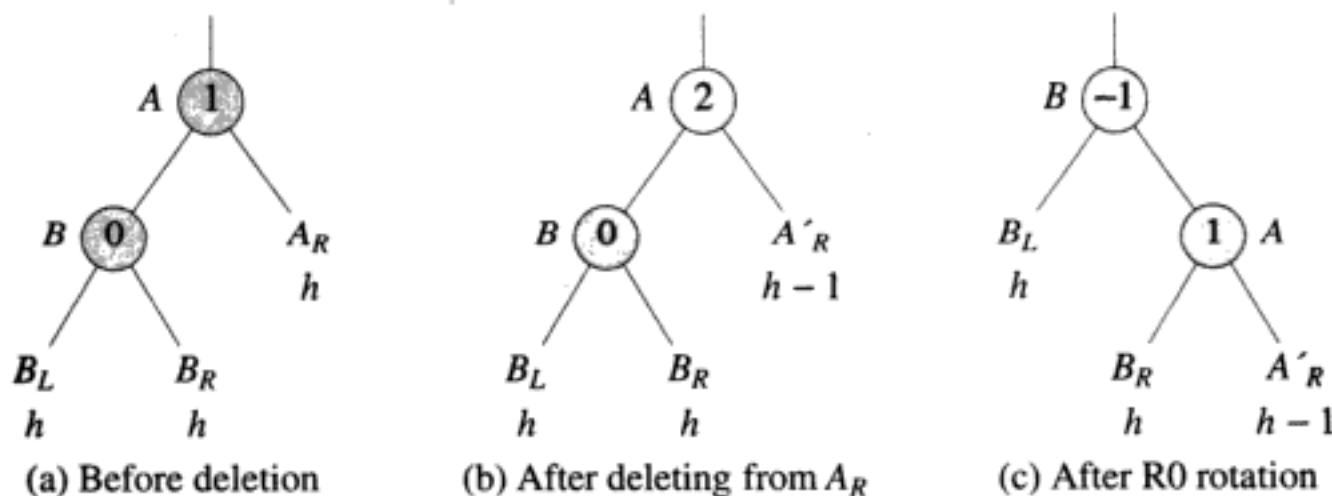


Figure 15.7 An R0 rotation (single rotation)

Hidden page

Hidden page

10. Prove observations I1 through I4 regarding an unbalanced tree resulting from an insertion using the strategy of Program 14.5.
11. Draw a figure analogous to Figure 15.3 for the case when $bf(X) = -1$ prior to the insertion.
12. Draw figures analogous to Figures 15.4 and 15.5 for the case of RR and RL imbalances.
13. Start with the LR imbalance shown in Figure 15.5(b) and draw the tree that results when we perform an RR rotation at node *B*. Observe that an LL rotation on this resulting tree results in the tree of Figure 15.5(b).
14. Draw figures analogous to Figures 15.7, 15.8, and 15.9 for the case of L0, L1, and L-1 imbalances.
15. Develop the concrete class `avlTree` that derives from the abstract class `indexed-BSTree` (Program 14.2). Fully code all your methods and test their correctness. Your implementations for `find`, `get`, `insert`, `erase` and `delete` must have complexity $O(\log n)$, and that for `ascend` should be $O(n)$. Show that this is the case.
16. Do Exercise 15 for the case when the search tree may contain several elements with the same key. Call the new class `davlTree`.
17. Develop a C++ class `indexedAVLtree` that includes the indexed binary search tree methods `find(theKey)`, `insert`, `delete(theKey)`, `get(theIndex)`, `erase(theIndex)`, and `ascend`. Fully code all your methods and test their correctness. Your implementations for the first five operations must have complexity $O(\log n)$ and that for the last operation should be $O(n)$. Show that this is the case.
18. Do Exercise 17 for the case when the search tree may contain several elements with the same key. Call the new class `dIndexedAVLtree`.
19. Explain how you could use an AVL tree to reduce the asymptotic complexity of our solution to the railroad car rearrangement problem of Section 8.5.3 to $O(n \log k)$.
20. Develop the class `linearListAsIndexedAVLtree` that derives from the abstract class `linearList` (Program 5.1). See Exercise 21 in Chapter 14 for some clues. Other than the operation `indexOf`, all operations should run in logarithmic or less time.
21. Replace the use of a binary search tree in Program 14.11 with an AVL search tree with duplicates. Measure the performance of the two versions of the best-fit bin-packing codes.

22. (a) Use an indexed AVL search tree to obtain an $O(n \log n)$ solution for the crossing-distribution problem (Section 14.6.3).
(b) Test the correctness of your code.
(c) Compare the actual run time of this solution to that of the $\Theta(n^2)$ solution described in Section 14.6.3 (see Program 14.12). Do this comparison using randomly generated permutations C and $n = 1000; 10,000;$ and $50,000.$

15.2 RED-BLACK TREES

15.2.1 Definition

A **red-black tree** is a binary search tree in which every node is colored either red or black. The remaining properties satisfied by a red-black tree are best stated in terms of the corresponding extended binary tree. Recall, from Section 12.5.1, that we obtain an extended binary tree from a regular binary tree by replacing every null pointer with an external node. The additional properties are

- RB1.** The root and all external nodes are colored black.
RB2. No root-to-external-node path has two consecutive red nodes.
RB3. All root-to-external-node paths have the same number of black nodes.

An equivalent definition arises from assigning colors to the pointers between a node and its children. The pointer from a parent to a black child is black and to a red child is red. Additionally,

- RB1'.** Pointers from an internal node to an external node are black.
RB2'. No root-to-external-node path has two consecutive red pointers.
RB3'. All root-to-external-node paths have the same number of black pointers.

Notice that if we know the pointer colors, we can deduce the node colors and vice versa. In the red-black tree of Figure 15.10, the external nodes are shaded squares, black nodes are shaded circles, red nodes are unshaded circles, black pointers are thick lines, and red pointers are thin lines. Notice that every path from the root to an external node has exactly two black pointers and three black nodes (including the root and the external node); no such path has two consecutive red nodes or pointers.

Let the **rank** of a node in a red-black tree be the number of black pointers (equivalently the number of black nodes minus 1) on any path from the node to any external node in its subtree. So the rank of an external node is 0. The rank of the root of Figure 15.10 is 2, that of its left child is 2, and of its right child is 1.

Hidden page

and 2 have $3 = 2^2 - 1$ internal nodes. There are additional internal nodes at levels 3 and 4.)

From (b) it follows that $r \leq \log_2(n + 1)$. This inequality together with (a) yields (c). ■

Since the height of a red-black tree is at most $2\log_2(n + 1)$, search, insert, and delete algorithms that work in $O(h)$ time have complexity $O(\log n)$.

Notice that the worst-case height of a red-black tree is more than the worst-case height (approximately $1.44\log_2(n + 2)$) of an AVL tree with the same number of (internal) nodes.

15.2.2 Representation of a Red-Black Tree

Although it is convenient to include external nodes when defining red-black trees, in an implementation null pointers, rather than physical nodes, represent external nodes. Further, since pointer and node colors are closely related, with each node we need to store only its color or the color of the two pointers to its children. Node colors require just one additional bit per node, while pointer colors require two. Since both schemes require almost the same amount of space, we may choose between them on the basis of actual run times of the resulting red-black tree algorithms.

In our discussion of the insert and delete operations, we will explicitly state the needed color changes only for the nodes. The corresponding pointer color changes may be inferred.

15.2.3 Searching a Red-Black Tree

We can search a red-black tree with the code we used to search an ordinary binary search tree (Program 14.4). This code has complexity $O(h)$, which is $O(\log n)$ for a red-black tree. Since we use the same code to search ordinary binary search trees, AVL trees, and red-black trees and since the worst-case height of an AVL tree is least, we expect AVL trees to show the best worst-case performance in applications where search is the dominant operation.

15.2.4 Inserting into a Red-Black Tree

Elements may be inserted using the strategy used for ordinary binary trees (Program 14.5). When the new node is attached to the red-black tree, we need to assign the node a color. If the tree was empty before the insertion, then the new node is the root and must be colored black (see property RB1). Suppose the tree was not empty prior to the insertion. If the new node is given the color black, then we will have an extra black node on paths from the root to the external nodes that are children of the new node. On the other hand, if the new node is assigned the color red, then we might have two consecutive red nodes. Making the new node black is guaranteed to cause a violation of property RB3, while making the new node red may or may not violate property RB2. We will make the new node red.

If making the new node red causes a violation of property RB2, we will say that the tree has become imbalanced. The nature of the imbalance is classified by examining the new node u , its parent pu , and the grandparent gu of u . Observe that since property RB2 has been violated, we have two consecutive red nodes. One of these red nodes is u , and the other must be its parent; therefore, pu exists. Since pu is red, it cannot be the root (as the root is black by property RB1); u must have a grandparent gu , which must be black (property RB2). When pu is the left child of gu , u is the left child of pu and the other child of gu is black (this case includes the case when the other child of gu is an external node); the imbalance is of type LLb. The other imbalance types are LLr (pu is the left child of gu , u is the left child of pu , the other child of gu is red), LRb (pu is the left child of gu , u is the right child of pu , the other child of gu is black), LRR, RRb, RRr, RLb, and RLr.

Imbalances of the type XYr (X and Y may be L or R) are handled by changing colors, while those of type XYb require a rotation. When we change a color, the RB2 violation may propagate two levels up the tree. In this case we will need to reclassify at the new level, with the new u being the former gu , and apply the transformations again. When a rotation is done, the RB2 violation is taken care of and no further work is needed.

Figure 15.11 shows the color changes performed for LLr and LRR imbalances; these color changes are identical. Black nodes are shaded, while red ones are not. In Figure 15.11(a), for example, gu is black, while pu and u are red; the pointers from gu to its left and right children are red; gu_R is the right subtree of gu ; and pu_R is the right subtree of pu . Both LLr and LRR color changes require us to change the color of pu and of the right child of gu from red to black. Additionally, we change the color of gu from black to red provided gu is not the root. Since this color change is not done when gu is the root, the number of black nodes on all root-to-external-node paths increases by 1 when gu is the root of the red-black tree.

If changing the color of gu to red causes an imbalance, gu becomes the new u node, its parent becomes the new pu , its grandparent becomes the new gu , and we continue to rebalance. If gu is the root or if the color change does not cause an RB2 violation at gu , we are done.

Figure 15.12 shows the rotations performed to handle LLb and LRb imbalances. In Figures 15.12(a) and (b), u is the root of pu_L . Notice the similarity between these rotations and the LL (refer to Figure 15.4) and LR (refer to Figure 15.5) rotations used to handle an imbalance following an insertion in an AVL tree. The pointer changes are the same. In the case of an LLb rotation, for example, in addition to pointer changes we need to change the color of gu from black to red and of pu from red to black.

In examining the node (or pointer) colors after the rotations of Figure 15.12, we see that the number of black nodes (or pointers) on all root-to-external-node paths is unchanged. Further, the root of the involved subtree (gu before the rotation and pu after) is black following the rotation; therefore, two consecutive red nodes cannot exist on the path from the tree root to the new pu . Consequently, no additional

Hidden page

Hidden page

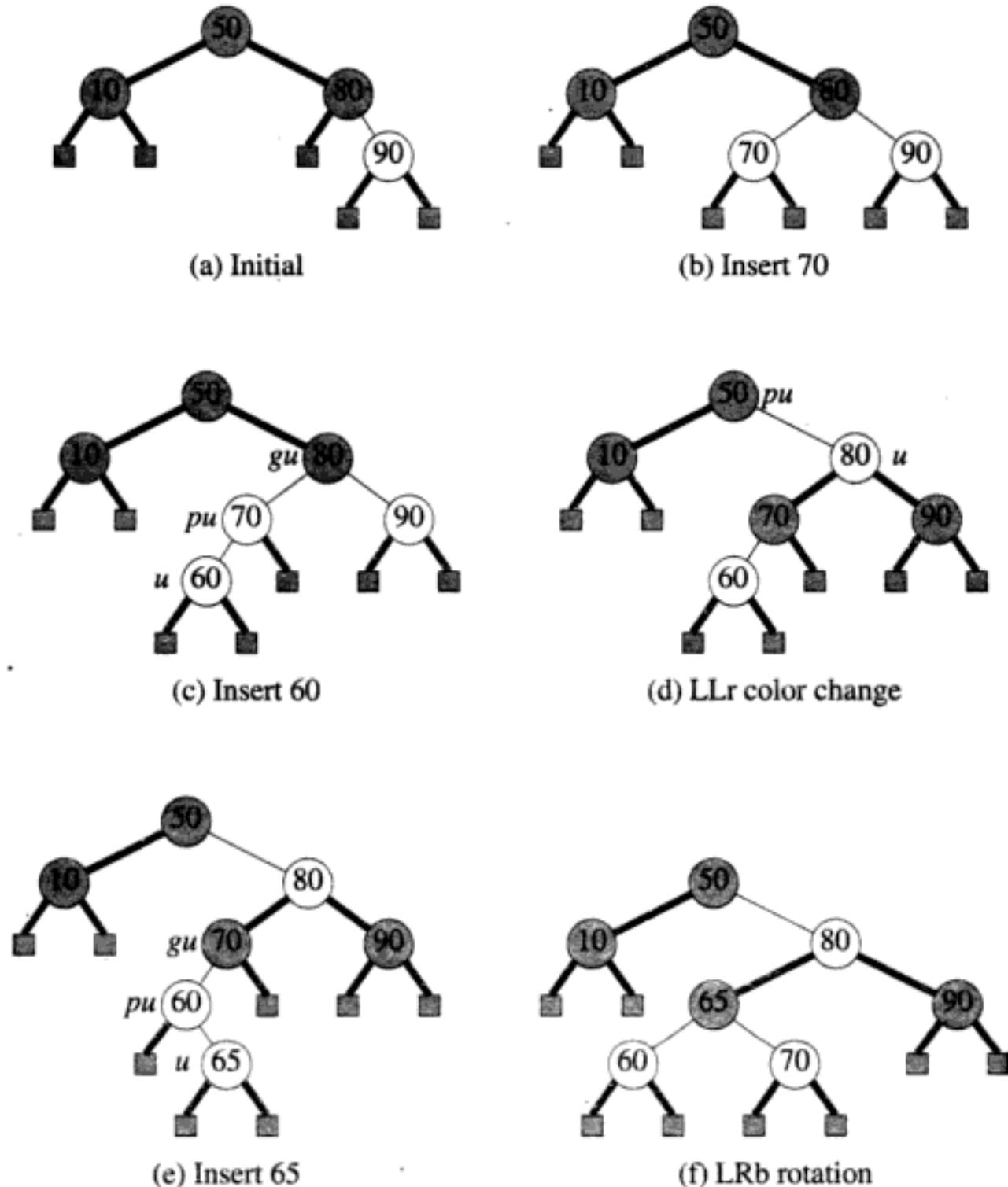


Figure 15.13 Insertion into a red-black tree (continues)

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

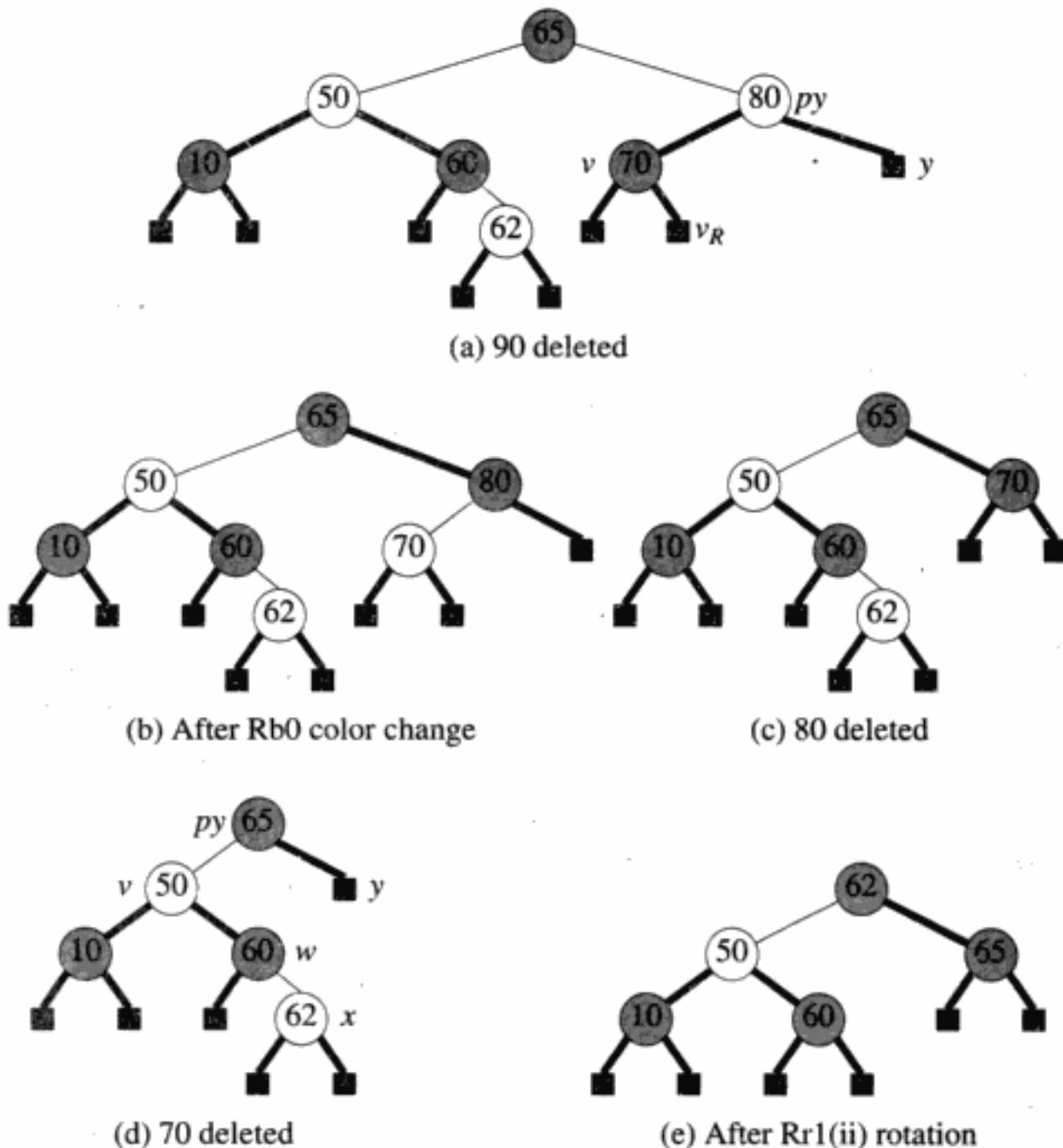


Figure 15.19 Deletion from a red-black tree

Now we may move back toward the root by performing deletes from the stack of saved pointers. For an n -element red-black tree, the addition of parent fields increases the space requirements by $\Theta(n)$, while the use of a stack increases the space requirements by $\Theta(\log n)$. Although the stack scheme is more efficient on space requirements, the parent-pointer scheme runs slightly faster.

Hidden page

34. Draw the Lb1 and Lb2 rotations that correspond to the Rb1 and Rb2 rotations of Figure 15.16.
35. Draw the Lr0, Lr1, and Lr2 rotations that correspond to the Rr0, Rr1, and Rr2 rotations of Figures 15.17 and 15.18.
36. Develop the concrete C++ class `redBlackTree` that derives from the abstract class `bsTree` (Program 14.1). Fully code all your methods and test their correctness. Your implementations for `find`, `insert`, and `delete` must have complexity $O(\log n)$, and that for the `ascend` should be $O(n)$. Show that this is the case. The implementations of `insert` and `delete` should follow the development in the text.
37. Develop the concrete C++ class `dRedBlackTree` that derives from `dBSTree` (see Exercise 4). Fully code all your methods and test their correctness. Your implementations for `find`, `insert`, and `delete` must have complexity $O(\log n)$, and that for the `ascend` should be $O(n)$. Show that this is the case.
38. Develop the C++ concrete class `indexedRedBlackTree` that derives from the abstract class `indexedBSTree` (Program 14.2). Fully code all your methods and test their correctness. Your implementations for `find`, `get`, `insert`, `erase`, and `delete` must have complexity $O(\log n)$, and that for the `ascend` should be $O(n)$. Show that this is the case.
39. Develop the C++ concrete class `dIndexedRedBlackTree` that implements the interface `dIndexedBSTree` (see Exercise 5). Fully code all your methods and test their correctness. Your implementations for `find`, `get`, `insert`, `delete`, and `erase` must have complexity $O(\log n)$, and that for the `ascend` should be $O(n)$. Show that this is the case.
40. Develop the C++ concrete class `linearListAsRedBlackTree` that derives from the abstract class `linearList`. Fully code all your methods and test their correctness.

15.3 SPLAY TREES

15.3.1 Introduction

When either AVL trees or red-black trees are used to implement a dictionary, the worst-case complexity of each dictionary operation is logarithmic in the dictionary size. No known data structures provide a better worst-case time complexity for these operations. However, in many applications of a dictionary, we are less interested in the time taken by an individual operation than we are in the time taken by a sequence of operations. This is the case, for example, for the applications considered at the end of Chapter 14. The complexity of each of these applications depends

Hidden page

Hidden page

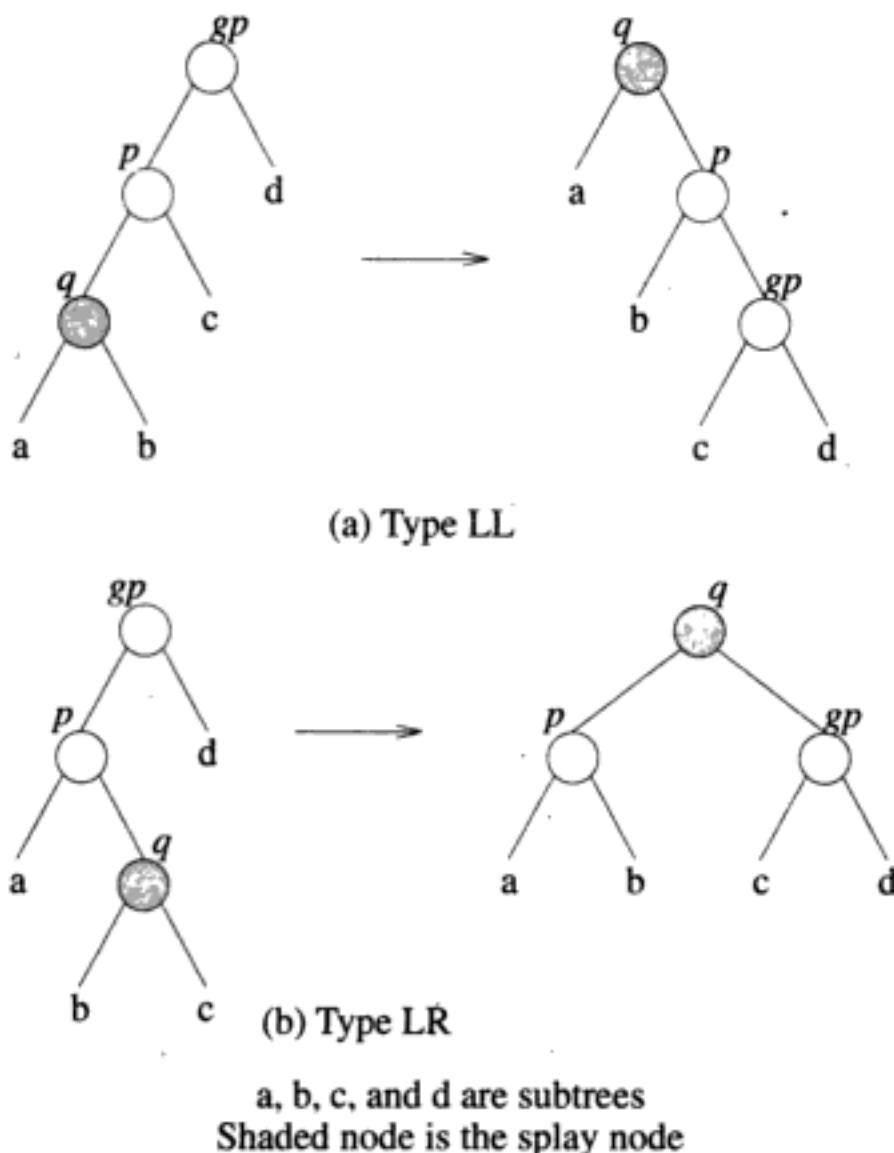


Figure 15.21 Types LL and LR splay steps

a splay operation to one-level splay steps does not ensure that every sequence of *f* *find*, *i* *insert*, and *d* *delete* operations is done in $O((f + i + d) \log i)$ time. To establish this time bound, it is necessary to use a sequence of two-level splay steps terminated by at most 1 one-level splay step.

15.3.3 Amortized Complexity

Unlike the actual and worst-case complexities of an operation, which are closely related to the step count for that operation, the **amortized complexity** of an operation is an accounting artifact that often bears no direct relationship to the actual complexity of that operation. The amortized complexity of an operation could be

Hidden page

Hidden page

Hidden page

Hidden page

with external nodes), each internal node has up to m children and between 1 and $m - 1$ elements. (External nodes contain no elements and have no children.)

2. Every node with p elements has exactly $p + 1$ children.
3. Consider any node with p elements. Let k_1, \dots, k_p be the keys of these elements. The elements are ordered so that $k_1 < k_2 < \dots < k_p$. Let c_0, c_1, \dots, c_p be the $p + 1$ children of the node. The elements in the subtree with root c_0 have keys smaller than k_1 , those in the subtree with root c_p have keys larger than k_p , and those in the subtree with root c_i have keys larger than k_i but smaller than k_{i+1} , $1 \leq i < p$. ■

Although it is useful to include external nodes when defining an m -way search tree, external nodes are not physically represented in actual implementations. Rather, a null pointer appears wherever there would otherwise be an external node.

Figure 15.23 shows a seven-way search tree. External nodes are shown as solid squares. All other nodes are internal nodes. The root has two elements (with keys 10 and 80) and three children. The middle child of the root has six elements and seven children; six of these children are external nodes.

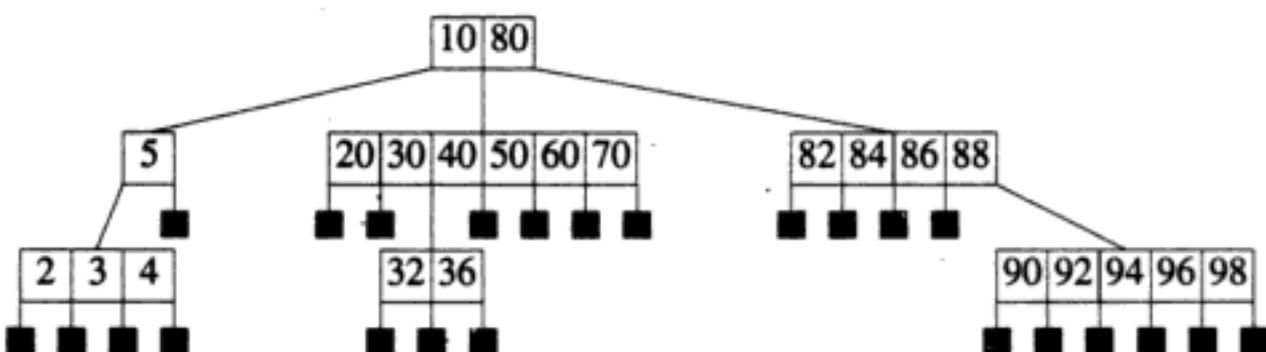


Figure 15.23 A seven-way search tree

Searching an m -Way Search Tree

To search the seven-way search tree in Figure 15.23 for an element with key 31, we begin at the root. Since 31 lies between 10 and 80, we follow the middle pointer. (By definition, all elements in the first subtree have key < 10 , and all in the third have key > 80 .) The root of the middle subtree is searched. Since $k_2 < 31 < k_3$, we move to the third subtree of this node. Now we determine that $31 < k_1$ and move into the first subtree. This move causes us to fall off the tree; that is, we reach an external node. We conclude that the search tree contains no element with key 31.

Inserting into an *m*-Way Search Tree

If we wish to insert an element with key 31, we search for 31 as above and fall off the tree at the node [32,36]. Since this node can hold up to six elements (each node of a seven-way search tree can have up to six elements), the new element may be inserted as the first element in the node.

To insert an element with key 65, we search for 65 and fall off the tree by moving to the sixth subtree of the node [20,30,40,50,60,70]. This node cannot accommodate additional elements, and a new node is obtained. The new element is put into this node, and the new node becomes the sixth child of [20,30,40,50,60,70].

Deleting from an *m*-Way Search Tree

To delete the element with key 20 from the search tree of Figure 15.23, we first perform a search. The element is the first element in the middle child of the root. Since $k_1 = 20$ and $c_0 = c_1 = 0$, we may simply delete the element from the node. The new middle child of the root is [30,40,50,60,70]. Similarly, to delete the element with key 84, we first locate the element. It is the second element in the third child of the root. Since $c_1 = c_2 = 0$, the element may be deleted from this node and the new node configuration is [82,86,88].

When deleting the element with key 5, we have to do more work. Since the element to be deleted is the first one in its node and since at least one of its neighboring children (these children are c_0 and c_1) is nonnull, we need to replace the deleted element with an element from a nonempty neighboring subtree. From the left neighboring subtree (c_0), we may move up the element with largest key (i.e., the element with key 4).

To delete the element with key 10 from the root of Figure 15.23, we may replace this element with either the largest element in c_0 or the smallest element in c_1 . If we opt to replace it with the largest in c_0 , then the element with key 5 moves up and we need to find a replacement for this element in its original node. The element with key 4 is moved up.

Height of an *m*-Way Search Tree

An *m*-way search tree of height h (excluding external nodes) may have as few as h elements (one node per level and one element per node) and as many as $m^h - 1$. The upper bound is achieved by an *m*-way search tree of height h in which each node at levels 1 through $h - 1$ has exactly m children and nodes at level h have no children. Such a tree has $\sum_{i=0}^{h-1} m^i = (m^h - 1)/(m - 1)$ nodes. Since each of these nodes has $m - 1$ elements, the number of elements is $m^h - 1$.

As the number of elements in an *m*-way search tree of height h ranges from a low of h to a high of $m^h - 1$, the height of an *m*-way search tree with n elements ranges from a low of $\log_m(n + 1)$ to a high of n .

A 200-way search tree of height 5, for example, can hold $32 * 10^{10} - 1$ elements but might hold as few as 5. Equivalently, a 200-way search tree with $32 * 10^{10} - 1$

elements has a height between 5 and $32 * 10^{10} - 1$. When the search tree resides on a disk, the search, insert, and delete times are dominated by the number of disk accesses made (under the assumption that each node is no larger than a disk block). Since the number of disk accesses needed for the search, insert, and delete operations are $O(h)$ where h is the tree height, we need to ensure that the height is close to $\log_m(n + 1)$. This assurance is provided by balanced m -way search trees.

15.4.3 B-Trees of Order m

Definition 15.3 A B-tree of order m is an m -way search tree. If the B-tree is not empty, the corresponding extended tree satisfies the following properties:

1. The root has at least two children.
2. All internal nodes other than the root have at least $\lceil m/2 \rceil$ children.
3. All external nodes are at the same level. ■

The seven-way search tree of Figure 15.23 is not a B-tree of order 7, as it contains external nodes at more than one level (levels 3 and 4). Even if all its external nodes were at the same level, it would fail to be a B-tree of order 7 because it contains nonroot internal nodes with two (node [5]) and three (node [32,36]) children. Nonroot internal nodes in a B-tree of order 7 must have at least $\lceil 7/2 \rceil = 4$ children. A B-tree of order 7 appears in Figure 15.24. All external nodes are at level 3, the root has three children, and all remaining internal nodes have at least four children. Additionally, it is a seven-way search tree.

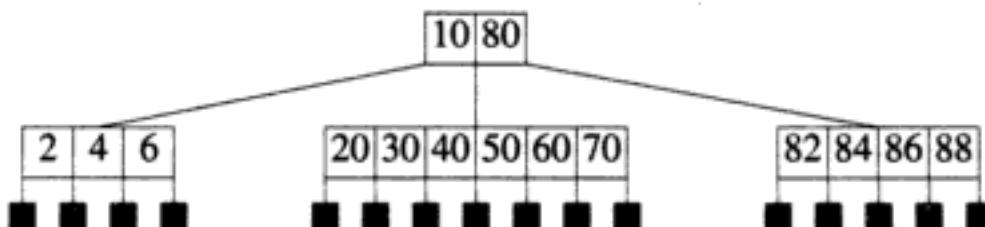


Figure 15.24 A B-tree of order 7

In a B-tree of order 2, no internal node has more than two children. Since an internal node must have at least two children, all internal nodes of a B-tree of order 2 have exactly two children. This observation, coupled with the requirement that all external nodes be on the same level, implies that B-trees of order 2 are full binary trees. As such, these trees exist only when the number of elements is $2^h - 1$ for some integer h .

In a B-tree of order 3, internal nodes have either two or three children. So a B-tree of order 3 is also known as a 2-3 tree. Since internal nodes in B-trees of

order 4 must have two, three, or four children, these B-trees are also referred to as 2-3-4 (or simply 2,4) trees. A 2-3 tree appears in Figure 15.25. Even though this tree has no internal node with four children, it is also an example of a 2-3-4 tree. To build a 2-3-4 tree in which at least one internal node has four children, simply add elements with keys 14 and 16 into the left child of 20.

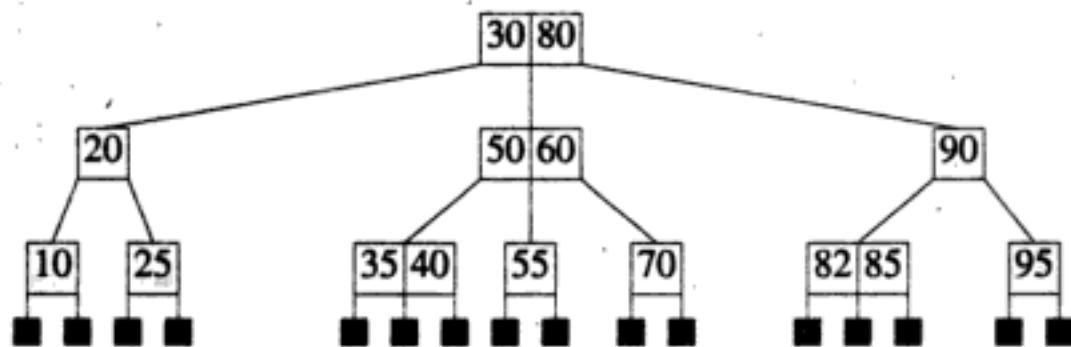


Figure 15.25 A 2-3 tree or B-tree of order 3

15.4.4 Height of a B-Tree

Lemma 15.3 Let T be a B-tree of order m and height h . Let $d = \lceil m/2 \rceil$ and let n be the number of elements in T .

- (a) $2d^{h-1} - 1 \leq n \leq m^h - 1$
- (b) $\log_m(n+1) \leq h \leq \log_d(\frac{n+1}{2}) + 1$

Proof The upper bound on n follows from the fact that T is an m -way search tree. For the lower bound, note that the external nodes of the corresponding extended B-tree are at level $h + 1$. The minimum number of nodes on levels 1, 2, 3, 4, ..., $h + 1$ is 1, 2, $2d$, $2d^2$, ..., $2d^{h-1}$, so the minimum number of external nodes in the B-tree is $2d^{h-1}$. Since the number of external nodes is 1 more than the number of elements

$$n \geq 2d^{h-1} - 1$$

Part (b) follows directly from (a). ■

From Lemma 15.3 it follows that a B-tree of order 200 and height 3 has at least 19,999 elements, and one of the same order and height 5 has at least $2 * 10^8 - 1$ elements. Consequently, if a B-tree of order 200 or more is used, the tree height is quite small even when the number of elements is rather large. In practice, the

B-tree order is determined by the disk block size and the size of individual elements. There is no advantage to using a node size smaller than the disk block size, as each disk access reads or writes one block. Using a larger node size involves multiple disk accesses, each accompanied by a seek and latency delay, so there is no advantage to making the node size larger than one block.

Although in actual applications the B-tree order is large, our examples use a small m because a two-level B-tree of order m has at least $2d - 1$ elements. When m is 200, d is 100 and a two-level B-tree of order 200 has at least 199 elements. Manipulating trees with this many elements is quite cumbersome. Our examples involve 2-3 trees and B-trees of order 7.

15.4.5 Searching a B-Tree

A B-tree is searched using the same algorithm as is used for an m -way search tree. Since all internal nodes on some root-to-external-node path may be retrieved during the search, the number of disk accesses is at most h (h is the height of the B-tree).

15.4.6 Inserting into a B-Tree

To insert an element into a B-tree, we first search for the presence of an element with the same key. If such an element is found, the insert fails because duplicates are not permitted. When the search is unsuccessful, we attempt to insert the new element into the last internal node encountered on the search path. For example, when inserting an element with key 3 into the B-tree of Figure 15.24, we examine the root and its left child. We fall off the tree at the second external node of the left child. Since the left child currently has three elements and can hold up to six, the new element may be inserted into this node. The result is the B-tree of Figure 15.26(a). Two disk accesses are made to read in the root and its left child. An additional access is necessary to write out the modified left child.

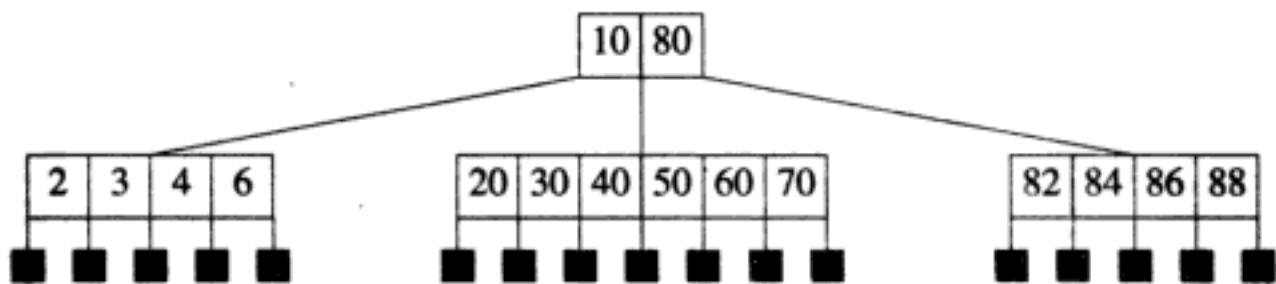
Next let us try to insert an element with key 25 into the B-tree of Figure 15.26(a). This element is to go into the node [20,30,40,50,60,70]. However, this node is full. *When the new element needs to go into a full node, the full node is split.* Let P be the full node. Insert the new element e together with a null pointer into P to get an overfull node with m elements and $m + 1$ children. Denote this overfull node as

$$m, c_0, (e_1, c_1), \dots, (e_m, c_m)$$

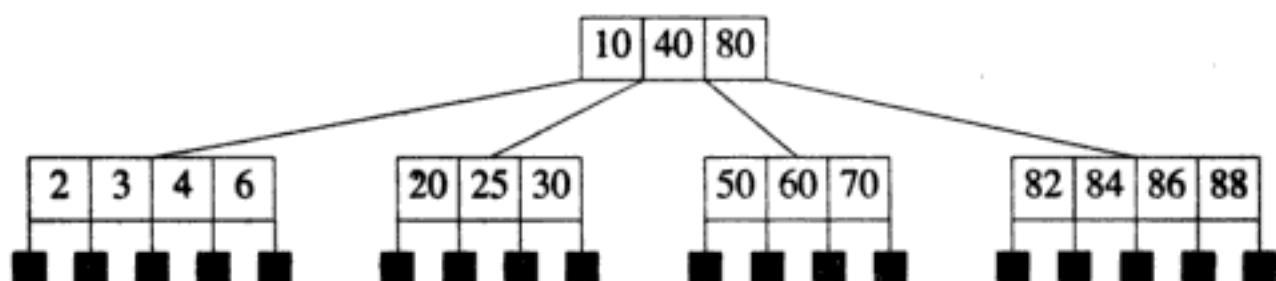
where the e_i s are the elements and the c_i s are the children pointers. The node is split around element e_d where $d = \lceil m/2 \rceil$. Elements to the left remain in P , and those to the right move into a new node Q . The pair (e_d, Q) is inserted into the parent of P . The format of the new P and Q is

$$P : d - 1, c_0, (e_1, c_1), \dots, (e_{d-1}, c_{d-1})$$

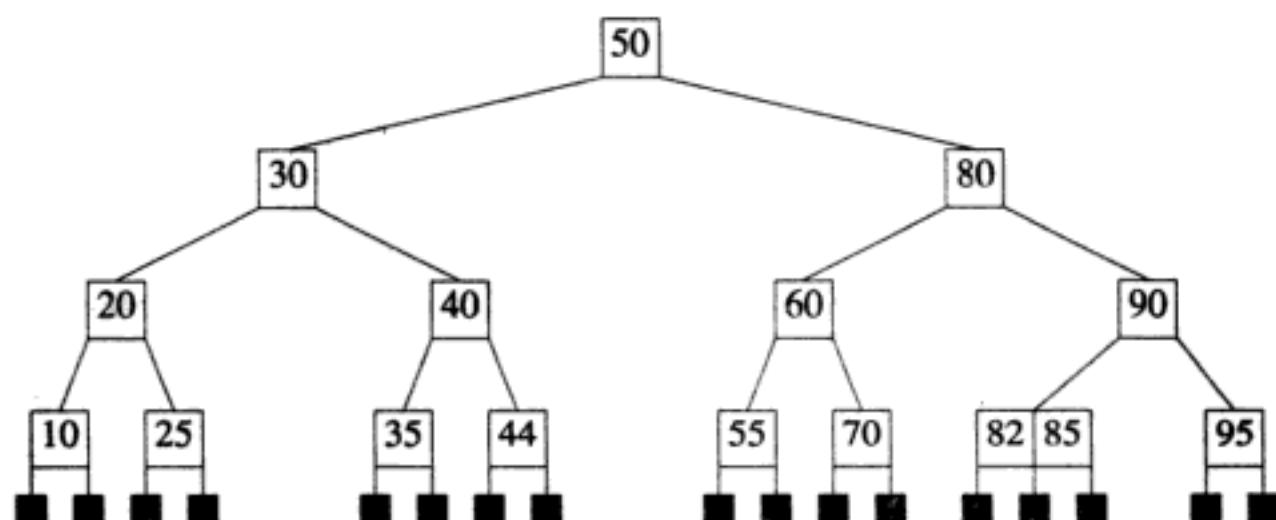
$$Q : m - d, c_d, (e_{d+1}, c_{d+1}), \dots, (e_m, c_m)$$



(a) Insert 3 into Figure 15.24



(b) Insert 25 into (a)



(c) Insert 44 into Figure 15.25

Figure 15.26 Inserting into a B-tree

Hidden page

Hidden page

Hidden page

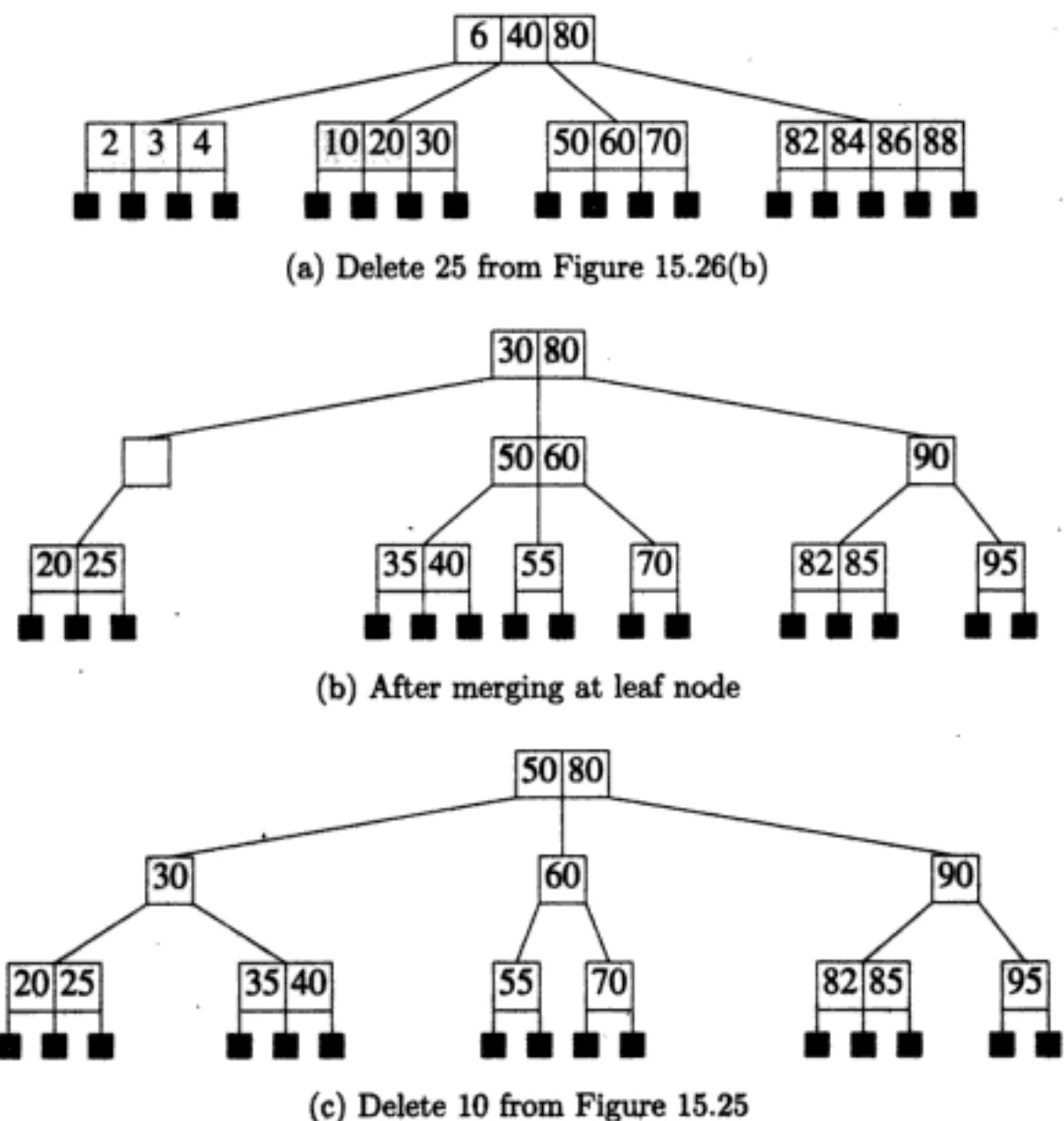


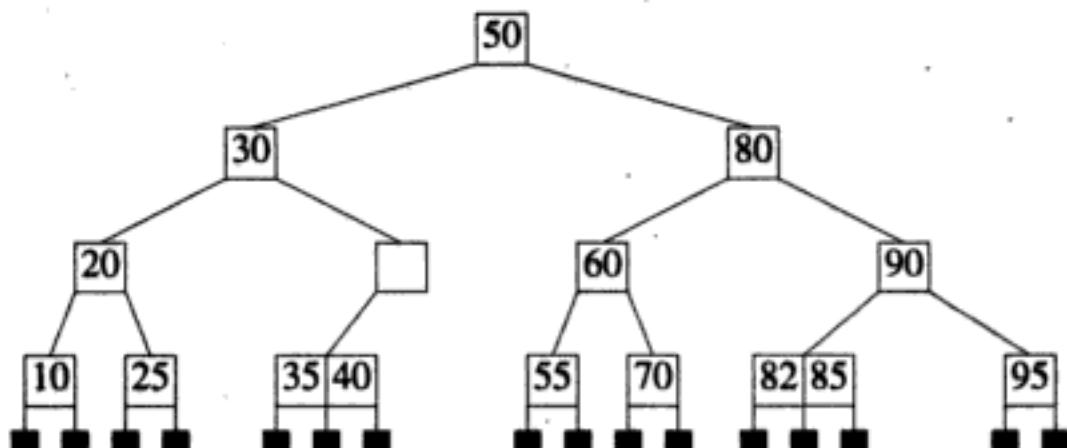
Figure 15.27 Deleting from a B-tree

merge, the grandparent may become one element short and the process will need to be applied at the grandparent. At worst the shortage will propagate back to the root. When the root is one element short, it is empty. The empty root is discarded, and the tree height decreases by 1.

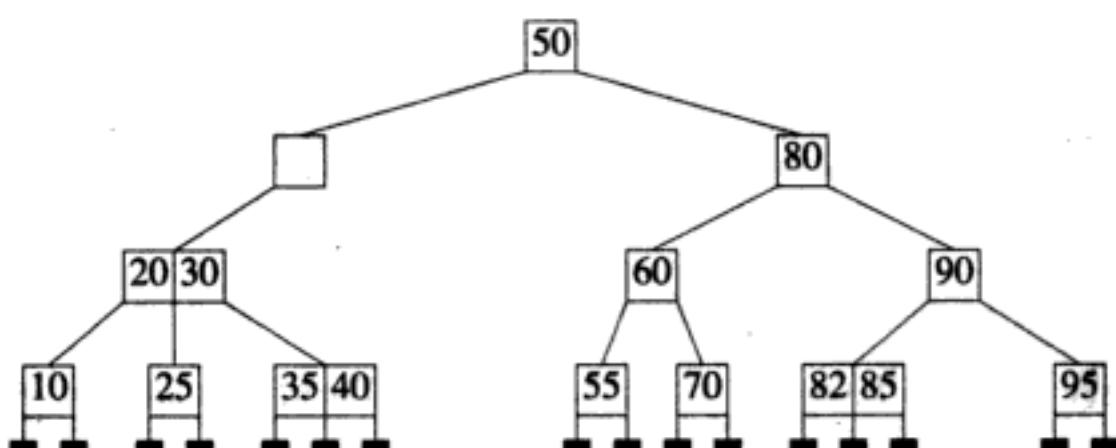
Suppose we wish to delete 10 from the 2-3 tree of Figure 15.25. This deletion leaves behind a leaf with zero elements. Its nearest-right sibling [25] does not have an extra element. Therefore, the two sibling leaves and the in-between element in the parent (10) are merged into a single node. The new tree structure appears

in Figure 15.27(b). We now have a node at level 2 that is an element short. Its nearest-right sibling has an extra element. The left-most element (i.e., the one with key 50) moves to the parent, and the element with key 30 moves down. The resulting 2-3 tree appears in Figure 15.27(c). Notice that the left subtree of the former [50,60] has moved also. This deletion took three read accesses to get to the leaf that contained the element that was to be deleted; two read accesses to get the nearest-right siblings of the level 3 and 2 nodes; and four write accesses to write out the four nodes at levels 1, 2, and 3 that were modified. The total number of disk accesses is 9.

As a final example, consider the deletion of 44 from the 2-3 tree of Figure 15.26(c). When the 44 is removed from the leaf it is in, this leaf becomes short one element. Its nearest-left sibling does not have an extra element, and so the two siblings to-



(a) After merging at leaf level



(b) After merging at level 3

Figure 15.28 Deleting 44 from the 2-3 tree of Figure 15.26(c) (continues)

Hidden page

We need four disk accesses to find the leaf that contains the element to be deleted, three nearest-sibling accesses, and three write accesses. The total number is 10.

The worst case for a deletion from a B-tree of height h is when merges take place at levels h , $h - 1$, ..., and 3 and we get an element from a nearest sibling at level 2. The worst-case disk access count is $3h$; (h reads to find the leaf with the element to be deleted) + ($h - 1$ reads to get nearest siblings at levels 2 through h) + ($h - 2$ writes of merged nodes at levels 3 through h) + (3 writes for the modified root and 2 level 2 nodes).

15.4.8 Node Structure

Our discussion has assumed a node structure of the form

$$s, c_0, (e_1, c_1), (e_2, c_2), \dots, (e_s, c_s)$$

where s is the number of elements in the node, the e_i s are the elements in ascending order of key, and the c_i s are children pointers. When the element size is large relative to the size of a key, we may use the node structure

$$s, c_0, (k_1, c_1, p_1), (k_2, c_2, p_2), \dots, (k_s, c_s, p_s)$$

where the k_i s are the element keys and the p_i s are the disk locations of the corresponding elements. By using this structure, we can use a B-tree of a higher order. An even higher-order B-tree, called a B'-tree, becomes possible if nonleaf nodes contain no p_i pointers and if, in the leaves, we replace the null children pointers with p_i pointers.

Another possibility is to use a balanced binary search tree to represent the contents of each node. Using a balanced binary search tree in this way reduces the permissible order of the B-tree, as with each element we need a left- and right-child pointer as well as a balance factor or color field. However, the CPU time spent inserting/deleting an element into/from a node decreases. Whether this approach results in improved overall performance depends on the application. In some cases a smaller m might increase the B-tree height, resulting in more disk accesses for each search/insert/delete operation.

EXERCISES

57. Start with an empty 2-3 tree and insert the keys 20, 40, 30, 10, 25, 28, 27, 32, 36, 34, 35, 8, 6, 2, and 3 in this order. Draw the 2-3 tree following each insert.
58. Start with an empty 2-3 tree and insert the keys 2, 1, 5, 6, 7, 4, 3, 8, 9, 10, and 11 in this order. Draw the 2-3 tree following each insert.

59. From the 2-3 tree of Figure 15.25, remove the keys 55, 40, 70, 35, 60, 95, 90, 82, 80 in this order. Draw a figure similar to Figure 15.28 showing the different steps in each remove.
60.
 - (a) Remove 10 from the 2-3 tree of Figure 15.26(c). Draw a figure similar to Figure 15.28 showing the different steps in the delete.
 - (b) Do part (a) but this time remove 70 from Figure 15.26(c).
 - (c) Do part (a) but this time remove 95 and 85 (in this order) from Figure 15.26(c).
61. What is the maximum number of disk accesses made during a search of a B-tree of order $2m$ if each node is two disk blocks and requires two disk accesses to retrieve? Compare this number with the corresponding number for a B-tree of order m that uses nodes that are one disk block in size. Based on this analysis, what can you say about the merits of using a node size larger than one block?
62. What is the maximum number of disk accesses needed to delete an element that is in a nonleaf node of a B-tree of order m ?
63. Suppose we modify the way an element is deleted from a B-tree as follows: If a node has both a nearest-left and nearest-right sibling, then both are checked before a merge is done. What is the maximum number of disk accesses that can be made when deleting from a B-tree of height h ?
64. A 2-3-4 tree may be represented as a binary tree in which each node is colored black or red. A 2-3-4 tree node that has just one element is represented as a black node; a node with two elements is represented as a black node with a red child (the red child may be either the left or right child of the black node); a node with three elements is represented as a black node with two red children.
 - (a) Draw a 2-3-4 tree that contains at least one node with two elements and one with three. Now draw it as a binary tree with colored nodes using the method just described.
 - (b) Verify that the binary tree is a red-black tree.
 - (c) Prove that when any 2-3-4 tree is represented as a colored binary tree as described here, the result is a red-black tree.
 - (d) Prove that every red-black tree can be represented as a 2-3-4 tree using the inverse mapping.
 - (e) Verify that the color changes and rotations given in Section 15.2.4 for an insertion into a red-black tree are obtainable from the B-tree insertion method using the mapping in (d).
 - (f) Do part (e) for the case of deletion from a red-black tree.

Hidden page

CHAPTER 16

GRAPHS

BIRD'S-EYE VIEW

Congratulations! You have successfully journeyed through the forest of trees. Awaiting you now is the study of the graph data structure. Surprisingly, graphs are used to model literally thousands of real-world problems. Not so surprisingly, we will see only a very small fraction of these problems in the remainder of this book. This chapter covers the following topics:

- Graph terminology including these terms: *vertex*, *edge*, *adjacent*, *incident*, *degree*, *cycle*, *path*, *connected component*, and *spanning tree*.
- Different types of graphs: undirected, directed, and weighted.
- Common graph representations: adjacency matrix, array adjacency lists, and linked adjacency lists.
- Standard graph search methods: breadth-first and depth-first search.
- Algorithms to find a path in a graph, to find the connected components of an undirected graph, and to find a spanning tree of a connected undirected graph.

Additional graph algorithms—topological sorting, bipartite covers, shortest paths, minimum-cost spanning trees, max clique, and traveling salesperson—are developed in the remaining chapters of this book.

16.1 DEFINITIONS

Informally, a graph is a collection of vertices or nodes, pairs of which are joined by lines or edges. More formally, a **graph** $G = (V, E)$ is an ordered pair of finite sets V and E . The elements of V are called **vertices** (vertices are also called **nodes** and **points**). The elements of E are called **edges** (edges are also called **arcs** and **lines**). Each edge in E joins two different vertices of V and is denoted by the tuple (i, j) , where i and j are the two vertices joined by E .

A graph is generally displayed as a figure in which the vertices are represented by circles and the edges by lines. Examples of graphs appear in Figure 16.1. Some of the edges in this figure are oriented (i.e., they have arrow heads), while others are not. An edge with an orientation is a **directed edge**, while an edge with no orientation is an **undirected edge**. The undirected edges (i, j) and (j, i) are the same; the directed edge (i, j) is different from the directed edge (j, i) , the former being oriented from i to j and the latter from j to i .¹

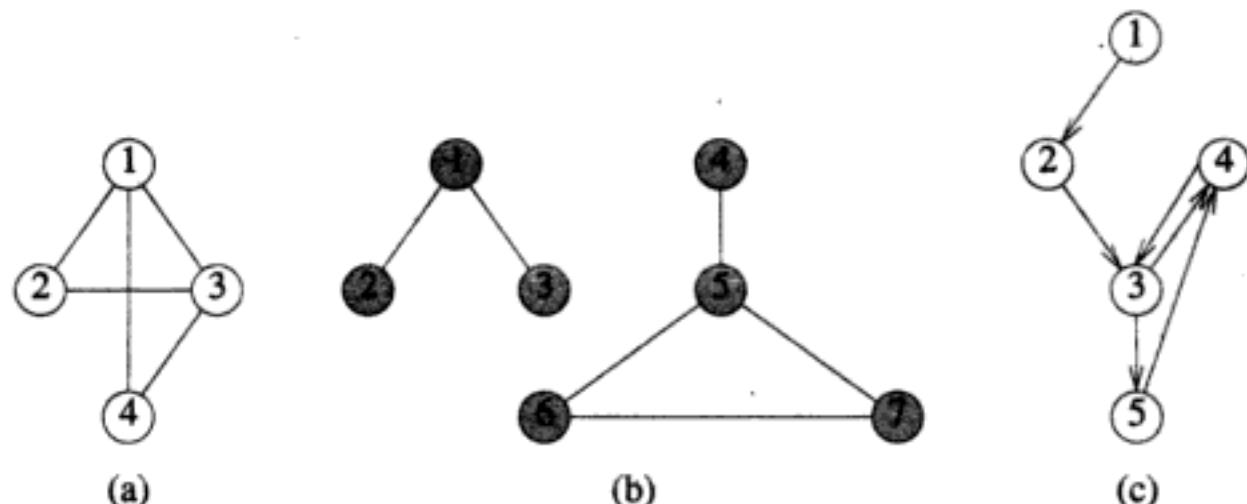


Figure 16.1 Graphs

Vertices i and j are **adjacent** vertices iff (i, j) is an edge in the graph. The edge (i, j) is **incident** on the vertices i and j . Vertices 1 and 2 of Figure 16.1(a) are adjacent, as are vertices 1 and 3; 1 and 4; 2 and 3; and 3 and 4. This graph has no other pairs of adjacent vertices. The edge $(1,2)$ is incident on the vertices 1 and 2 and the edge $(2,3)$ is incident on the vertices 2 and 3.

It is sometimes useful to have a slightly refined notion of adjacency and incidence for directed graphs. The directed edge (i, j) is **incident to** vertex j and **incident from** vertex i . Vertex i is **adjacent to** vertex j , and vertex j is **adjacent from**

¹Some books use the notation i, j for an undirected edge and (i, j) for a directed one. Others use (i, j) for an undirected edge and $\langle i, j \rangle$ for a directed one. This book uses the same notation, (i, j) , for both kinds of edges. Whether an edge is directed or not will be clear from the context.

vertex i . In the graph of Figure 16.1(c), vertex 2 is adjacent from 1, while 1 is adjacent to 2. Edge (1,2) is incident from 1 and incident to 2. Vertex 4 is both incident to and from 3. Edge (3,4) is incident from 3 and incident to 4. For an undirected edge, the refinements “to” and “from” are synonymous.

Using set notation, the graphs of Figure 16.1 may be specified as $G_1 = (V_1, E_1)$; $G_2 = (V_2, E_2)$; and $G_3 = (V_3, E_3)$ where

$$\begin{array}{ll} V_1 = \{1,2,3,4\}; & E_1 = \{(1,2), (1,3), (2,3), (1,4), (3,4)\} \\ V_2 = \{1,2,3,4,5,6,7\}; & E_2 = \{(1,2), (1,3), (4,5), (5,6), (5,7), (6,7)\} \\ V_3 = \{1,2,3,4,5\}; & E_3 = \{(1,2), (2,3), (3,4), (4,3), (3,5), (5,4)\} \end{array}$$

If all the edges in a graph are undirected, then the graph is an **undirected graph**. The graphs of Figures 16.1(a) and (b) are undirected graphs. If all the edges are directed, then the graph is a **directed graph**. The graph of Figure 16.1(c) is a directed graph.

By definition, a graph does not contain multiple copies of the same edge. Therefore, an undirected graph can have at most one edge between any pair of vertices, and a directed graph can have at most one edge from vertex i to vertex j and one from j to i . Also, a graph cannot contain any **self-edges**; that is, edges of the form (i, i) are not permitted. A self-edge is also called a **loop**.

A directed graph is also called a **digraph**. In some applications of graphs, we will assign a **weight** or **cost** to each edge. When weights have been assigned to edges, we use the terms **weighted undirected graph** and **weighted digraph** to refer to the resulting data object. The term **network** is often used to refer to a weighted undirected graph or digraph. Actually, all the graph variants defined here may be regarded as special cases of networks—an undirected (directed) graph may be viewed as an undirected (directed) network in which all edges have the same weight.

16.2 APPLICATIONS AND MORE DEFINITIONS

Graphs are used in the analysis of electrical networks; the study of the molecular structure of chemical compounds (particularly hydrocarbons); the representation of airline routes and communication networks; in planning projects, genetic studies, statistical mechanics, and social sciences; and in many other situations. In this section we formulate some real-world problems as problems on graphs.

Example 16.1 [Path Problems] In a city with many streets, we can say that each intersection is a vertex in a digraph. Each segment of a street that is between two adjacent intersections is represented by either one or two directed edges. Two directed edges, one in either direction, are used if the street segment is two way, and a single directed edge is used if it is a one-way segment. Figure 16.2 shows a hypothetical street map and the corresponding digraph. In this figure there are three

streets—1St, 2St, and 3St—and two avenues—1Ave and 2Ave. The intersections are labeled 1 through 6. The vertices of the corresponding digraph (Figure 16.2(b)) have the same labels as given to the intersection in Figure 16.2(a).

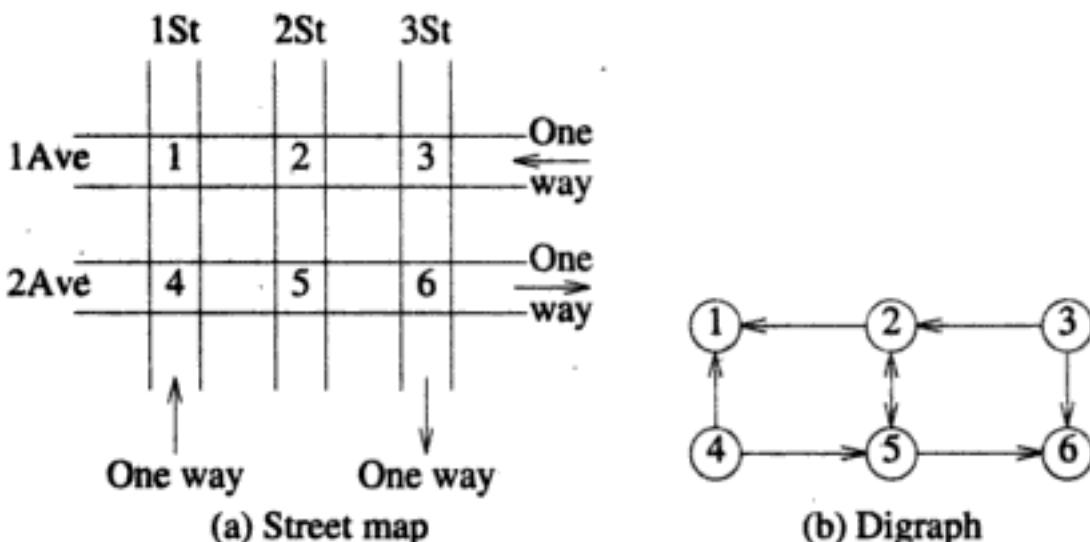


Figure 16.2 Street map and corresponding digraph

A sequence of vertices $P = i_1, i_2, \dots, i_k$ is an i_1 to i_k path in the graph $G = (V, E)$ iff the edge (i_j, i_{j+1}) is in E for every j , $1 \leq j < k$. There is a path from intersection i to intersection j iff there is a path from vertex i to vertex j in the corresponding digraph. In the digraph of Figure 16.2(b), 5, 2, 1 is a path from 5 to 1. There is no path from 5 to 4 in this digraph.

A **simple path** is a path in which all vertices, except possibly the first and last, are different. The path 5, 2, 1 is a simple path, whereas the path 2, 5, 2, 1 is not.

Each edge in a graph may have an associated length. The length of a path is the sum of the lengths of the edges on the path. The shortest way to get from intersection i to intersection j is obtained by finding a shortest path from vertex i to vertex j in the corresponding network (i.e., weighted digraph). ■

Example 16.2 [Spanning Trees] Let $G = (V, E)$ be an undirected graph. G is connected iff there is a path between every pair of vertices in G . The undirected graph of Figure 16.1(a) is connected, while that of Figure 16.1(b) is not. Suppose that G represents a possible communication network with V being the set of cities and E the set of communication links. It is possible to communicate between every pair of cities in V iff G is connected. In the communication network of Figure 16.1(a), cities 2 and 4 can communicate by using the communication path 2, 3, 4, while in the network of Figure 16.1(b), cities 2 and 4 cannot communicate.

Suppose that G is connected. Some of the edges of G may be unnecessary in that G remains connected even if these edges are removed. In the graph of Figure 16.1(a),

Hidden page

Hidden page

Hidden page

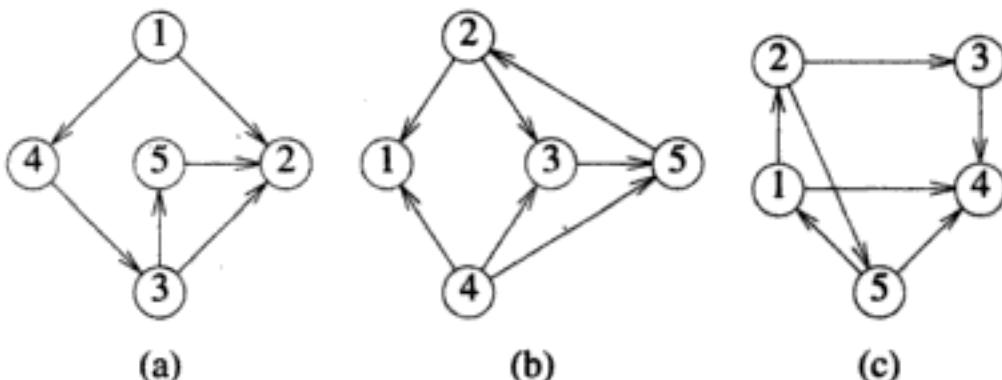


Figure 16.8 Digraphs

10. Prove Property 16.2.
11. Let G be any undirected graph. Show that the number of vertices with odd degrees is even.
12. Let $G = (V, E)$ be a connected graph with $|V| > 1$. Show that G contains either a vertex of degree 1 or a cycle (or both).
13. Let $G = (V, E)$ be a connected undirected graph that contains at least one cycle. Let $(i, j) \in E$ be an edge that is on at least one cycle of G . Show that the graph $H = (V, E - \{(i, j)\})$ is also connected.
14. Prove the following:
 - (a) For every n there exists a connected undirected graph containing exactly $n - 1$ edges, $n \geq 1$.
 - (b) Every n -vertex connected undirected graph contains at least $n - 1$ edges. You may use the results of the previous two exercises.
15. A digraph is **strongly connected** iff it contains a directed path from i to j and from j to i for every pair of distinct vertices i and j .
 - (a) Show that for every n , $n \geq 2$, there exists a strongly connected digraph that contains exactly n edges.
 - (b) Show that every n vertex strongly connected digraph contains at least n edges, $n \geq 2$.

16.4 THE ADT *graph*

The abstract data type *graph* refers to all varieties of graphs, whether directed, undirected, weighted, or unweighted. The abstract data type specification of ADT theChapter.1 lists only a few of the many operations commonly performed on a graph. As we progress through this text, we will add operations.

AbstractDataType *graph*

{

instances

a set *V* of vertices and a set *E* of edges

operations

numberOfVertices() : return the number of vertices in the graph

numberOfEdges() : return the number of edges in the graph

existsEdge(i, j) : return **true** if edge (i, j) exists; **false** otherwise

insertEdge(theEdge) : insert the edge *theEdge* into the graph

eraseEdge(i, j) : delete the edge (i, j)

degree(i) : return the degree of vertex *i*, defined only for undirected graphs

inDegree(i) : return the in-degree of vertex *i*

outDegree(i) : return the out-degree of vertex *i*

}

ADT 16.1 Abstract data type specification of a graph

Unlike the abstract classes for the data structures seen so far, the abstract class corresponding to the ADT *graph* will contain pure virtual methods as well as methods for which an implementation is provided in the abstract class. These latter methods, to be introduced later in this chapter and in succeeding chapters, will employ an iterator that will sequence through the vertices adjacent from a given vertex. Some of these methods also will need to tell whether the graph being worked on is directed and/or weighted. So we augment the ADT specification of *graph* (ADT 16.1) with methods to support these tasks. Program 16.1 is the resulting C++ abstract class. For weighted graphs, *T* is the data type of the edge weights; and for unweighted graphs, *T* is *bool*.

The template class *edge*, which gives the data type of the input to the *insertEdge* method is an abstract class that has the methods *vertex1*, *vertex2*, and *weight*

Hidden page

Hidden page

3. For an n -vertex undirected graph, $\sum_{j=1}^n A(i, j) = \sum_{j=1}^n A(j, i) = d_i$. (Recall that d_i is the degree of vertex i .)
4. For an n -vertex digraph, $\sum_{j=1}^n A(i, j) = d_i^{out}$ and $\sum_{j=1}^n A(j, i) = d_i^{in}$, $1 \leq i \leq n$.

Mapping the Adjacency Matrix into an Array

The $n \times n$ adjacency matrix A may be mapped into an $(n+1) \times (n+1)$ array a of type `bool` by using the mapping $A(i, j)$ equals 1 iff $a[i][j]$ is `true`, $1 \leq i \leq n$, $1 \leq j \leq n$. This representation requires $(n+1)^2 = n^2 + 2n + 1$ bytes. Alternatively, we may use an $n \times n$ array— $a[n][n]$ —and the mapping $A(i, j)$ equals 1 iff $a[i-1][j-1]$ is `true`, $1 \leq i \leq n$, $1 \leq j \leq n$. Since this alternative requires n^2 bytes, the storage requirement is reduced by $2n + 1$ bytes.

A further reduction by n bytes results if we use the fact that all diagonal entries are 0 and so need not be stored. When the diagonal is eliminated, an upper- and a lower-triangular matrix remain (see Section 7.3.4). These triangular matrices may be compacted into an $(n-1) \times n$ matrix as in Figure 16.10. The shaded entries represent the lower triangle of the original adjacency matrix.

	1	2	3	4	5	6	7
1	1	1	1	0	0	0	0
2	1	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	1	1	0	0
5	0	0	0	0	1	1	1
6	0	0	0	0	1	1	1

(a)

	1	2	3	4	5	6	7
1	1	1	1	0	0	0	0
2	1	1	1	0	0	0	0
3	1	0	1	1	0	0	0
4	0	0	0	1	1	0	0
5	0	0	0	0	1	1	1
6	0	0	0	0	1	1	1

(b)

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	1	0	0	0	0
3	0	0	1	1	0	0	0
4	0	0	0	1	0	0	0

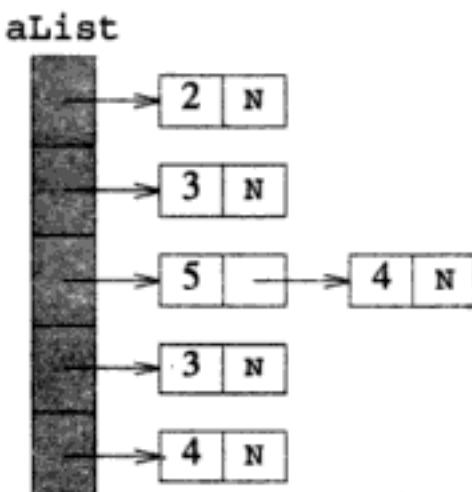
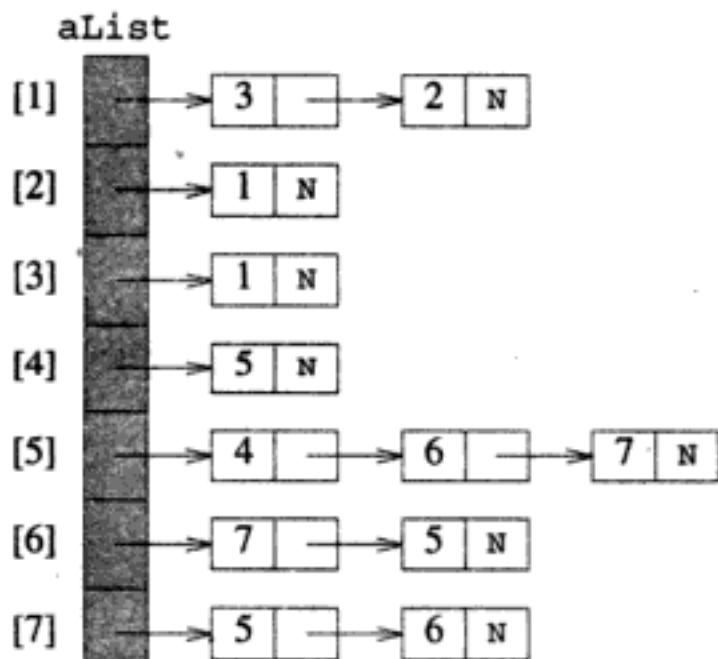
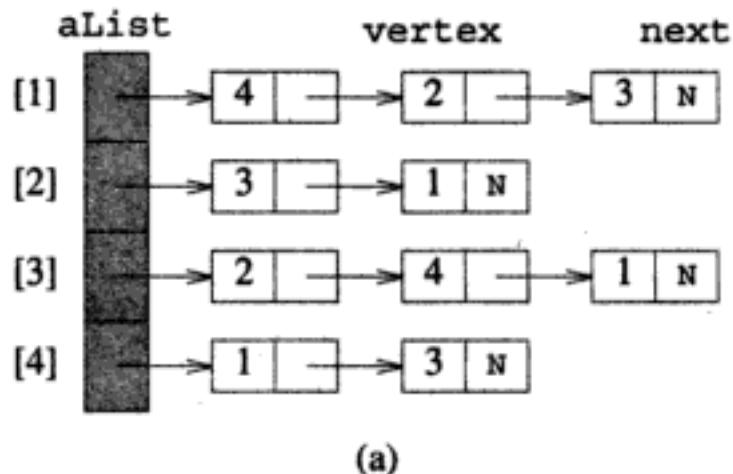
(c)

Figure 16.10 Adjacency matrices of Figure 16.9 with diagonals eliminated

The preceding methods to reduce the storage requirements yield a very small reduction and come at the cost of causing a mismatch between the vertex indexes used in the external and internal representation of a graph. This mismatch has the potential of causing errors in our codes. Further, the reduced storage methods require more time to access an edge. Therefore, we will use the $(n+1) \times (n+1)$ array mapping.

For undirected graphs the adjacency matrix is symmetric (see Section 7.3.5), so only the elements above (or below) the diagonal need to be stored explicitly. Hence we need only $(n^2 - n)/2$ bytes. By using the method of Section 7.3.5, we reduce

Hidden page



N denotes a NULL link

Figure 16.11 Linked adjacency lists for the graphs of Figure 16.1

The array-adjacency-list representation requires $4m$ bytes of space less than that required by linked adjacency lists, because, in the array-adjacency-list representation, we do not have the m next pointer fields that are present in the linked-adjacency-list representation. Most of the operations commonly performed on a graph can be done in the same asymptotic complexity by using either linked or array adjacency lists. Based on the experimental results of Sections 6.1.6 and 8.4.3, we expect array adjacency lists to outperform linked adjacency lists for most graph

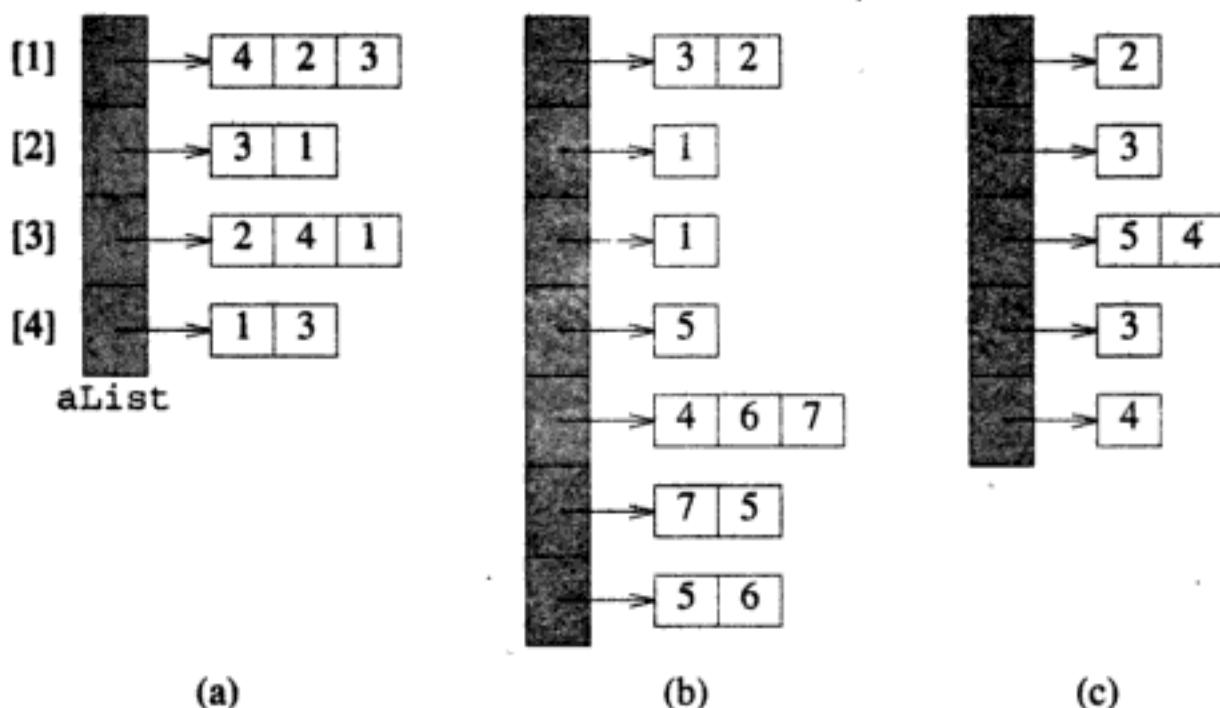


Figure 16.12 Array adjacency lists for graphs of Figure 16.1

applications.

A word of caution—the space analyses we have done for adjacency matrices and lists are approximate analyses. Actual implementations may take slightly more space because in an actual implementation we may store additional data such as the number of vertices and edges in a graph. It is expected, however, that the space unaccounted for in our analyses will not affect the relative comparisons made by us.

EXERCISES

16. Draw the following representations for the digraph of Figure 16.2(b).
 - (a) Adjacency matrix.
 - (b) Linked adjacency lists.
 - (c) Array adjacency lists.
17. Do Exercise 16 for the graph of Figure 16.4(a).
18. Do Exercise 16 for the graph of Figure 16.5.
19. Do Exercise 16 for the digraph of Figure 16.8(a).
20. Do Exercise 16 for the digraph of Figure 16.8(b).

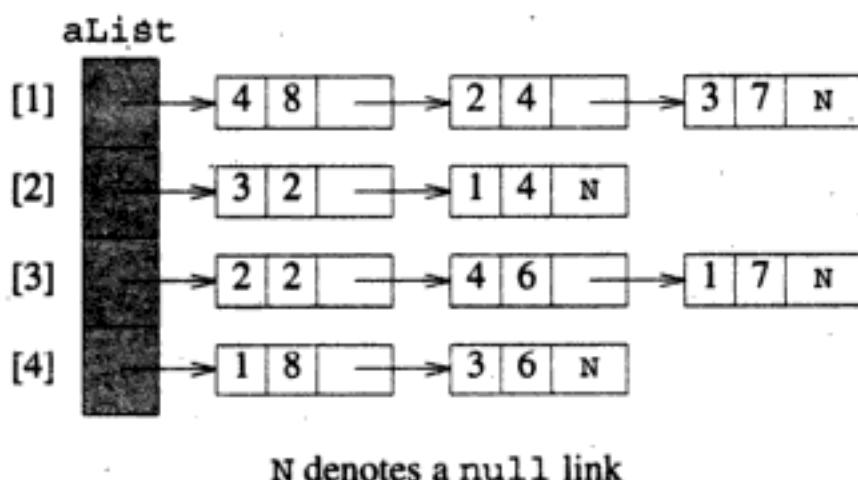
Hidden page

	1	2	3	4	5	6	7	
1	-	9	5	-	-	-	-	1
2	9	-	-	-	-	-	-	2
3	5	-	-	-	-	-	-	3
4	-	-	-	-	3	-	-	4
5	-	-	-	3	-	6	4	5
6	-	-	-	-	6	-	1	6
7	-	-	-	-	4	1	-	5

(a)	(b)	(c)
The small dash (-) denotes the value noEdge		

Figure 16.13 Possible cost-adjacency matrices for the graphs of Figure 16.1

We can obtain the linked-adjacency-list representation of a weighted graph from that of the corresponding graph by using chains whose elements have the two fields **vertex** and **weight**. Figure 16.14 shows the representation for the weighted graph that corresponds to the cost-adjacency matrix of Figure 16.13(a). The first component of each node in this figure is **vertex**, and the second is **weight**.

**Figure 16.14** Linked adjacency lists for weighted graph of Figure 16.13(a)

We can obtain the array-adjacency-list representation of a weighted graph from that of the corresponding unweighted graph by replacing each entry with a (**vertex**, **weight**) pair. The representation differs from that shown in Figure 16.14 only in

Hidden page

Hidden page

```
template <class T>
class adjacencyWDigraph : public graph<T>
{
protected:
    int n;           // number of vertices
    int e;           // number of edges
    T **a;          // adjacency array
    T noEdge;       // denotes absent edge

public:
    adjacencyWDigraph(int numberOfVertices = 0, T theNoEdge = 0)
    { // Constructor.
        // validate number of vertices
        if (numberOfVertices < 0)
            throw illegalParameterValue("number of vertices must be >= 0");
        n = numberOfVertices;
        e = 0;
        noEdge = theNoEdge;
        make2dArray(a, n + 1, n + 1);
        for (int i = 1; i <= n; i++)
            // initialize adjacency matrix
            fill(a[i], a[i] + n + 1, noEdge);
    }

    ~adjacencyWDigraph() {delete2dArray(a, n + 1);}

    int numberOfVertices() const {return n;}

    int numberOfEdges() const {return e;}

    bool directed() const {return true;}

    bool weighted() const {return true;}

    bool existsEdge(int i, int j) const
    { // Return true iff (i,j) is an edge of the graph.
        if (i < 1 || j < 1 || i > n || j > n || a[i][j] == noEdge)
            return false;
        else
            return true;
    }
}
```

Program 16.2 Cost-adjacency matrix for weighted directed graphs (continues)

```

void insertEdge(edge<T> *theEdge)
{// Insert edge theEdge into the digraph; if the edge is already
// there, update its weight to theEdge.weight().
    int v1 = theEdge->vertex1();
    int v2 = theEdge->vertex2();
    if (v1 < 1 || v2 < 1 || v1 > n || v2 > n || v1 == v2)
    {
        ostringstream s;
        s << "(" << v1 << "," << v2
            << ") is not a permissible edge";
        throw illegalParameterValue(s.str());
    }

    if (a[v1][v2] == noEdge) // new edge
        e++;
    a[v1][v2] = theEdge->weight();
}

void eraseEdge(int i, int j)
{// Delete the edge (i,j).
    if (i >= 1 && j >= 1 && i <= n && j <= n && a[i][j] != noEdge)
    {
        a[i][j] = noEdge;
        e--;
    }
}

void checkVertex(int theVertex) const
{// Verify that theVertex is a valid vertex.
    if (theVertex < 1 || theVertex > n)
    {
        ostringstream s;
        s << "no vertex " << theVertex;
        throw illegalParameterValue(s.str());
    }
}

int degree(int theVertex) const
{throw undefinedMethod("degree() undefined");}

```

Program 16.2 Cost-adjacency matrix for weighted directed graphs (continues)

```
int outDegree(int theVertex) const
{// Return out-degree of vertex theVertex.
    checkVertex(theVertex);

    // count out edges from vertex theVertex
    int sum = 0;
    for (int j = 1; j <= n; j++)
        if (a[theVertex][j] != noEdge)
            sum++;

    return sum;
}

int inDegree(int theVertex) const
{// Return in-degree of vertex theVertex.
    checkVertex(theVertex);

    // count in edges at vertex theVertex
    int sum = 0;
    for (int j = 1; j <= n; j++)
        if (a[j][theVertex] != noEdge)
            sum++;

    return sum;
}

class myIterator : public vertexIterator<T>
{
public:
    myIterator(T* theRow, T theNoEdge, int numberOfVertices)
    {
        row = theRow;
        noEdge = theNoEdge;
        n = numberOfVertices;
        currentVertex = 1;
    }

    ~myIterator() {}
}
```

Program 16.2 Cost-adjacency matrix for weighted directed graphs (continues)

```

int next(T& theWeight)
{ // Return next vertex if any. Return 0 if no next vertex.
    // Set theWeight = edge weight.
    // find next adjacent vertex
    for (int j = currentVertex; j <= n; j++)
        if (row[j] != noEdge)
    {
        currentVertex = j + 1;
        theWeight = row[j];
        return j;
    }
    // no next adjacent vertex
    currentVertex = n + 1;
    return 0;
}

// code for next() is similar to above

protected:
    T* row;           // row of adjacency matrix
    T noEdge;         // theRow[i] == noEdge iff no edge to i
    int n;            // number of vertices
    int currentVertex;
};

myIterator* iterator(int theVertex)
{ // Return iterator for vertex theVertex.
    checkVertex(theVertex);
    return new myIterator(a[theVertex], noEdge, n);
}
};

```

Program 16.2 Cost-adjacency matrix for weighted directed graphs (concluded)

The method `degree` simply throws an exception of type `undefinedMethod` because `degree` is defined only for undirected graphs (whether weighted or unweighted). The out-degree of vertex `theVertex` is computed by determining the number of entries `a[theVertex][*]` that are not equal to `noEdge`, and the in-degree of vertex `theVertex` is the number of entries `a[*][theVertex]` that are not equal to `noEdge`.

For the iterator, we define the class `myIterator`, which is a member class of `adjacencyWDigraph`. Recall from our discussion of the abstract class `graph` (Sec-

Hidden page

Hidden page

```
void eraseEdge(int i, int j)
{
    if (i >= 1 && j >= 1 && i <= n && j <= n)
    {
        int *v = aList[i].eraseElement(j);
        if (v != NULL) // edge (i,j) did exist
            e--;
    }
}

void checkVertex(int theVertex) const
{// Verify that theVertex is a valid vertex.
// code is same as for adjacencyWDigraph
}

int degree(int theVertex) const
{throw undefinedMethod("degree() undefined");}

int outDegree(int theVertex) const
{// Return out-degree of vertex theVertex.
checkVertex(theVertex);
return aList[theVertex].size();
}

int inDegree(int theVertex) const
{
    checkVertex(theVertex);

    // count in-edges at vertex theVertex
    int sum = 0;
    for (int j = 1; j <= n; j++)
        if (aList[j].indexOf(theVertex) != -1)
            sum++;

    return sum;
}

// code for iterator omitted
};
```

Program 16.3 The class linkedDigraph (concluded)

The method `degree` is undefined for a directed graph. The out-degree of vertex `theVertex` is just `aList[theVertex].listSize()`. Since the complexity of `chain::listSize()` is $\Theta(1)$, the out-degree of a vertex is determined in $\Theta(1)$ time. Computing the in-degree of a vertex is far more expensive. To compute the in-degree of vertex `theVertex`, we must search all the adjacency lists, keeping a count of the number of lists that contain `theVertex` (Program 16.3). The complexity of `inDegree` is $O(n + e)$, where n and e are, respectively, the number of vertices and the number of edges in the digraph; this complexity may be reduced to $\Theta(1)$ by keeping an array of in-degree values.

Other Linked Classes

The codes for the remaining linked classes may be obtained with modest effort and can be found at the Web site.

EXERCISES

32. Write the method `adjacencyWDigraph::input`, which inputs a weighted directed graph. Assume that the input consists of the number of vertices and edges in the graph together with a list of edges. Each edge is given as a pair of vertices plus the edge weight. Note that your input method is inherited by the remaining adjacency-matrix classes. Does your input method work correctly for these remaining adjacency-matrix classes? If not, override the input method by writing new ones as needed.
33. Write and test a copy constructor for `adjacencyWDigraph`.
34. Write the method `linkedDigraph::input`, which inputs a digraph. The complexity of your method should be linear in the number of vertices and edges in the input graph. Show that this is the case.
35. Write the method `linkedWDigraph::input`, which inputs a weighted digraph. The complexity of your method should be linear in the number of vertices and edges in the input graph. Show that this is the case.
36. We can speed `linkedWGraph` and `linkedWDigraph` if we include a new version of the `indexOf` method in `graphChain`. The new version updates the element in case it is already in the chain. Develop such an `indexOf` method and modify the `insertEdge` methods of the linked adjacency list classes to utilize this new method.
37. Develop the C++ class `arrayDigraph` in which digraphs are represented with array adjacency lists.
38. Develop the C++ class `arrayGraph` in which undirected graphs are represented with array adjacency lists.

Hidden page

queue. A pseudocode version of a possible implementation appears in Figure 16.17. Notice the similarity between BFS and level-order traversal of a binary tree.

```

breadthFirstSearch(v)
{
    Label vertex v as reached.
    Initialize Q to be a queue with only v in it.
    while (Q is not empty)
    {
        Delete a vertex w from the queue.
        Let u be a vertex (if any) adjacent from w.
        while (u != NULL)
        {
            if (u has not been labeled)
            {
                Add u to the queue.
                Label u as reached.
            }
            u = next vertex that is adjacent from w.
        }
    }
}

```

Figure 16.17 Pseudocode for BFS

If we use the pseudocode of Figure 16.17 on the graph of Figure 16.16(a) with $v = 1$, then vertices 2, 3, and 4 will get added to the queue (assume that they get added in this order) during the first iteration of the outer **while** loop. In the next iteration of this loop, 2 is removed from the queue, and vertex 5 added to it. Next 3 is deleted from the queue, and no new vertices are added. Then 4 is deleted and 6 and 7 added; 5 is deleted and 8 added; 6 is deleted and nothing added; and 7 is deleted and 9 added. Finally, 8 and 9 are deleted, and the queue becomes empty. The procedure terminates, and vertices 1 through 9 have been marked as reached. Figure 16.16(b) shows the subgraph formed by the edges used to reach the nodes that get visited.

Theorem 16.1 *Let G be an arbitrary graph and let v be any vertex of G . The pseudocode of Figure 16.17 labels all vertices that are reachable from v (including vertex v).*

Proof Exercise 43(a) asks you to prove this theorem. ■

Hidden page

```

virtual void bfs(int v, int reach[], int label)
{ // Breadth-first search. reach[i] is set to label for
// all vertices reachable from vertex v.
    arrayQueue<int> q(10);
    reach[v] = label;
    q.push(v);
    while (!q.empty())
    {
        // remove a labeled vertex from the queue
        int w = q.front();
        q.pop();

        // mark all unreached vertices adjacent from w
        vertexIterator<T> *iw = iterator(w);
        int u;
        while ((u = iw->next()) != 0)
            // visit an adjacent vertex of w
            if (reach[u] == 0)
                // u is an unreached vertex
                q.push(u);
                reach[u] = label; // mark reached
        }
        delete iw;
    }
}

```

Program 16.4 Implementation-independent BFS code

works with all representations, whereas the customized route requires several codes. Consequently, if we introduce new representations (for example array adjacency lists), we can use the implementation-independent code with no change. When developing code for a new graph application, we can first develop the implementation-independent code. This approach enables all graph implementations to make use of this application. Then as time and resources permit, we can develop the more efficient customized codes for individual representations.

16.8.4 Depth-First Search

Depth-first search (DFS) is an alternative to BFS. The DFS strategy has already been used in the rat-in-a-maze problem (Section 8.5.6) and is quite similar to pre-order traversal of a binary tree.

Figure 16.18 gives the pseudocode for DFS. Starting at a vertex v , a DFS pro-

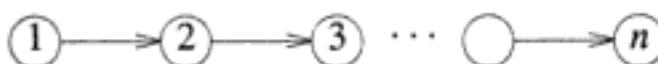
Hidden page

Hidden page

Hidden page

16.8.6 Complexity Analysis of graph::dfs

You can verify that the methods `dfs` and `bfs` have the same time and space complexities. However, the instances for which `dfs` takes maximum space (i.e., stack space for the recursion) are those on which `bfs` takes minimum space (i.e., queue space); and the instances for which `bfs` takes maximum space are those on which `dfs` takes minimum space. Figure 16.19 gives the best-case and worst-case instances for `dfs` and `bfs`.



(a) Worst case for `dfs` (1); best case for `bfs` (1)



(b) Best case for `dfs` (1); worst case for `bfs` (1)

Figure 16.19 Worst-case and best-case space complexity graphs

EXERCISES

41. Consider the graph of Figure 16.4(a).
 - (a) Draw an adjacency-list representation (either linked or array) for this graph.
 - (b) Label the vertices in the order in which they are visited in a BFS that starts at vertex 4. Use your representation of part (a) and the code of Program 16.4.
 - (c) Show the subgraph formed by the edges used to reach new vertices during your search of part (b).
 - (d) Redo parts (b) and (c), but this time do a DFS using the code of Program 16.7.
42. Do Exercise 41 using vertex 7 as the start vertex for the search.
43. (a) Prove Theorem 16.1.
 (b) Prove Theorem 16.2.

Hidden page

Hidden page

Hidden page

Hidden page

```
int labelComponents(int c[])
{// Label the components of an undirected graph.
 // Return the number of components.
// Set c[i] to be the component number of vertex i.
 // make sure this is an undirected graph
if (directed())
    throw undefinedMethod
("graph::labelComponents() not defined for directed graphs");

int n = numberOfVertices();

// assign all vertices to no component
for (int i = 1; i <= n; i++)
    c[i] = 0;

label = 0; // ID of last component
// identify components
for (int i = 1; i <= n; i++)
    if (c[i] == 0) // vertex i is unreached
        {// vertex i is in a new component
            label++;
            bfs(i, c, label); // mark new component
        }

return label;
}
```

Program 16.11 Component labeling

(a); the start vertex for each BFS is the shaded node.

When a DFS is performed in a connected undirected graph or network, exactly $n-1$ edges are used to reach new vertices. The subgraph formed by these edges is also a spanning tree. Spanning trees obtained in this manner from a DFS are called **depth-first spanning trees**. Figure 16.21 shows some of the depth-first spanning trees of Figure 16.20(a). In each case the start vertex is vertex 1.

EXERCISES

45. Write a version of Program 16.8 that uses a BFS rather than a DFS. Show that this version finds a shortest path from `theSource` to `theDestination`.
46. For the graph of Figure 16.20(a), do the following:

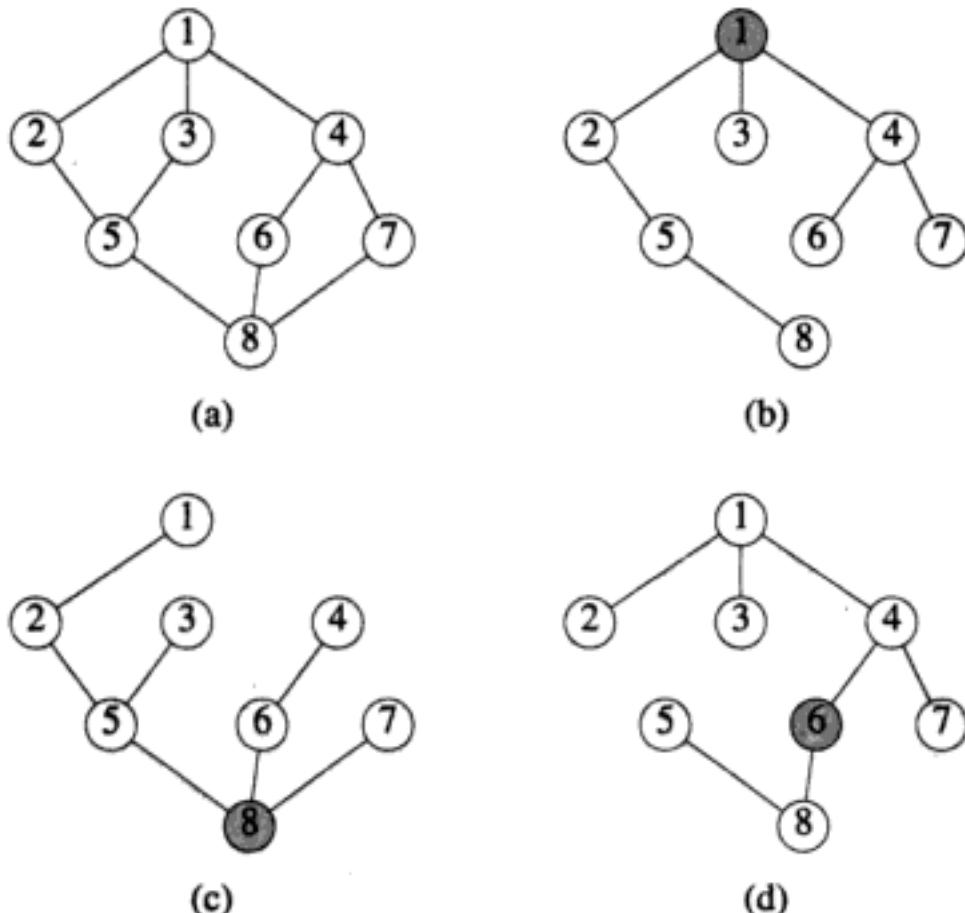
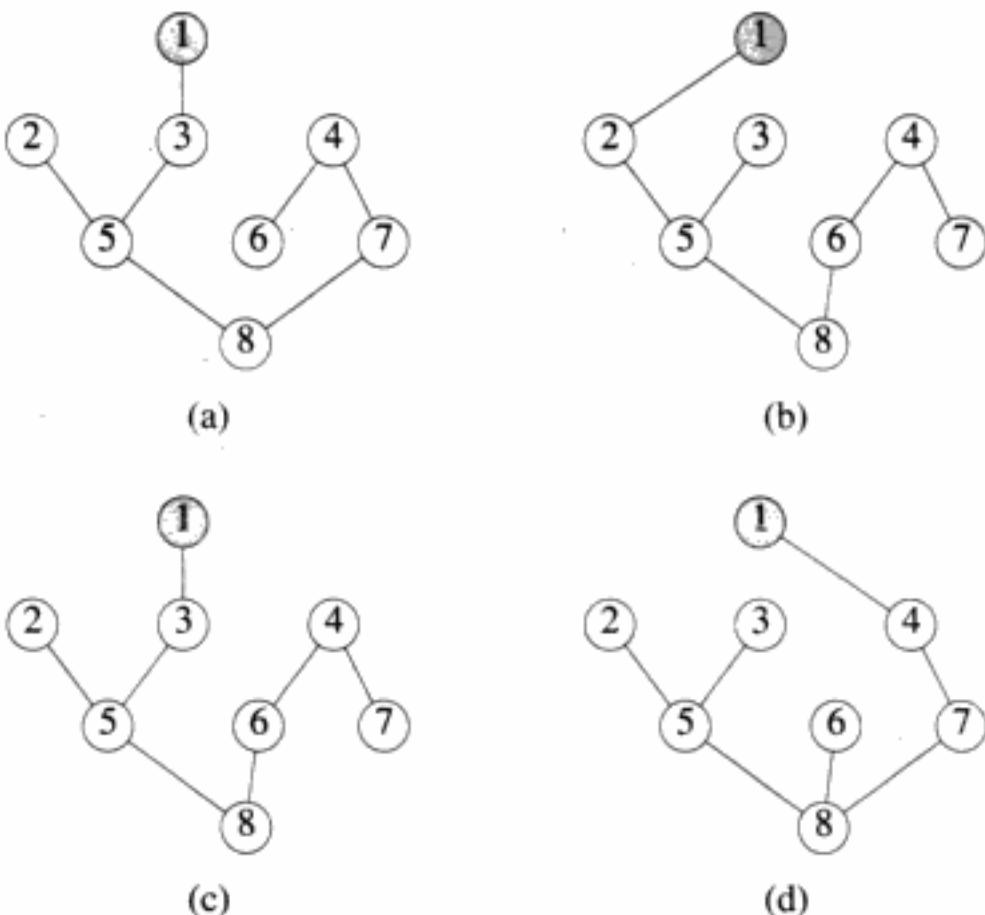


Figure 16.20 A graph and some of its breadth-first spanning trees

- (a) Draw a breadth-first spanning tree starting at vertex 3.
 - (b) Draw a breadth-first spanning tree starting at vertex 7.
 - (c) Draw a depth-first spanning tree starting at vertex 3.
 - (d) Draw a depth-first spanning tree starting at vertex 7.
47. Do Exercise 46 using the graph of Figure 16.4(a).
48. Write code for the method `graph::bfSpanningTree(theVertex)`, which finds a breadth-first spanning tree of a connected undirected graph by initiating a BFS at `theVertex`. Your code should return `NULL` if it fails because there is no spanning tree (i.e., the graph is not connected). When a spanning tree is found, your method should return an array of edges that make up the spanning tree. The data type of the edge array that is returned is `pair<int, int>`.
49. Do Exercise 48 for the case of `graph::dfSpanningTree(theVertex)`, which finds a depth-first spanning tree beginning at `theVertex`.

**Figure 16.21** Some depth-first spanning trees of Figure 16.20(a)

50. Write code for the public method `graph::cycle()`, which determines whether an undirected graph has a cycle. Base your code on either a DFS or a BFS.
- Prove the correctness of your code.
 - Determine the time and space complexities of your code.
51. Let G be an undirected graph. Write code for the method `graph::bipartite()`, which returns `NULL` if G is not a bipartite graph (see Example 16.3) and an integer array `label` if it is. When G is bipartite, the array `label` is a labeling of the vertices such that `label[i] = 1` for vertices in one subset and `label[i] = 2` for vertices in the other subset. The complexity of your code should be $O(n^2)$ if G has n vertices and is represented as a matrix and $O(n+e)$ if a linked list representation is used. Show that this is the case. (*Hint:* Perform several BFSs, each time beginning at an as-yet-unreached vertex. Assign this unreached vertex to set 1; vertices adjacent from this vertex are assigned to set 2, those adjacent to these set 2 vertices are assigned to set 1,

and so on. Also check for conflicting assignments, i.e., the set assignment of a vertex is changed.)

52. Let G be an undirected graph. Its **transitive closure** is a 0/1 valued array `tc` such that `tc[i][j] = 1` iff G has a path with one or more edges from vertex i to vertex j . Write a method `graph::undirectedTC()` to compute and return the transitive closure of G . The time complexity of your method should be $O(n^2)$ where n is the number of vertices in G . (*Hint:* Use component labeling.)
53. Do Exercise 52 for `graph::directedTC()`, which is to be used when G is directed. What is the complexity of your algorithm?

CHAPTER 17

THE GREEDY METHOD

BIRD'S-EYE VIEW

Exit the world of data structures. Enter the world of algorithm-design methods. In the remainder of this book, we study methods for the design of good algorithms. Although the design of a good algorithm for any given problem is more an art than a science, some design methods are effective in solving many problems. You can apply these methods to computer problems and see how the resulting algorithm works. Generally, you must fine-tune the resulting algorithm to achieve acceptable performance. In some cases, however, fine-tuning is not possible, and you will have to think of some other way to solve the problem.

Chapters 17 through 21 present five basic algorithm-design methods: the greedy method, divide and conquer, dynamic programming, backtracking, and branch and bound. This list excludes several more advanced methods, such as linear programming, integer programming, neural networks, genetic algorithms, and simulated annealing, that are also widely used. These methods are typically covered in courses and texts dedicated to them.

This chapter begins by introducing the notion of an optimization problem. Next the greedy method, which is a very intuitive method, is described. Although it is usually quite easy to formulate a greedy algorithm for a problem, the formulated algorithm may not always produce an optimal answer. Therefore, *it is essential that we prove that our greedy algorithms actually work*. Even when greedy algo-

rithms do not guarantee optimal answers, they remain useful, as they often get us answers that are close to optimal. We use the greedy method to obtain algorithms for the container-loading, knapsack, topological-ordering, bipartite-cover, shortest-path, and minimum-cost spanning-tree problems.

17.1 OPTIMIZATION PROBLEMS

Many of the examples used in this chapter and in the remaining chapters are **optimization problems**. In an optimization problem we are given a set of **constraints** and an **optimization function**. Solutions that satisfy the constraints are called **feasible solutions**. A feasible solution for which the optimization function has the best possible value is called an **optimal solution**.

Example 17.1 [Thirsty Baby] A very thirsty, and intelligent, baby wants to quench her thirst. She has access to a glass of water, a carton of milk, cans of various juices, and bottles and cans of various sodas. In all the baby has access to n different liquids. From past experience with these n liquids, the baby knows that some are more satisfying than others. In fact, the baby has assigned satisfaction values to each liquid. s_i units of satisfaction are obtained by drinking 1 ounce of the i th liquid.

Ordinarily, the baby would just drink enough of the liquid that gives her greatest satisfaction per ounce and thereby quench her thirst in the most satisfying way. Unfortunately, there isn't enough of this most satisfying liquid available. Let a_i be the amount in ounces of liquid i that is available. The baby needs to drink a total of t ounces to quench her thirst. How much of each available liquid should she drink?

We may assume that satisfaction is additive. Let x_i denote the amount of liquid i that the baby should drink. The solution to her problem is obtained by finding real numbers x_i , $1 \leq i \leq n$ that maximize $\sum_{i=1}^n s_i x_i$ subject to the constraints $\sum_{i=1}^n x_i = t$ and $0 \leq x_i \leq a_i$, $1 \leq i \leq n$.

Note that if $\sum_{i=1}^n a_i < t$, then there is no solution to the baby's problem. Even if she drinks all the liquids available, she will be unable to quench her thirst.

This precise mathematical formulation of the problem provides an unambiguous specification of what the program is to do. Having obtained this formulation, we can provide the input/output specification, which takes the form:

[Input] n , t , s_i , a_i , $1 \leq i \leq n$. n is an integer, and the remaining numbers are positive reals.

[Output] Real numbers x_i , $1 \leq i \leq n$, such that $\sum_{i=1}^n s_i x_i$ is maximum, $\sum_{i=1}^n x_i = t$, and $0 \leq x_i \leq a_i$, $1 \leq i \leq n$. Output a suitable message if $\sum_{i=1}^n a_i < t$.

The constraints are $\sum_{i=1}^n x_i = t$ and $0 \leq x_i \leq a_i$, and the optimization function is $\sum_{i=1}^n s_i x_i$. Every set of x_i 's that satisfies the constraints is a feasible solution. Every feasible solution that maximizes $\sum_{i=1}^n s_i x_i$ is an optimal solution. ■

Example 17.2 [Loading Problem] A large ship is to be loaded with cargo. The cargo is containerized, and all containers are the same size. Different containers

may have different weights. Let w_i be the weight of the i th container, $1 \leq i \leq n$. The cargo capacity of the ship is c . We wish to load the ship with the maximum number of containers.

This problem can be formulated as an optimization problem in the following way: Let x_i be a variable whose value can be either 0 or 1. If we set x_i to 0, then container i is not to be loaded. If x_i is 1, then the container is to be loaded. We wish to assign values to the x_i s that satisfy the constraints $\sum_{i=1}^n w_i x_i \leq c$ and $x_i \in \{0, 1\}$, $1 \leq i \leq n$. The optimization function is $\sum_{i=1}^n x_i$.

Every set of x_i s that satisfies the constraints is a feasible solution. Every feasible solution that maximizes $\sum_{i=1}^n x_i$ is an optimal solution. ■

Example 17.3 [Minimum-Cost Communication Network] We introduced this problem in Example 16.2. The set of cities and possible communication links can be represented as an undirected graph. Each edge has a cost (or weight) assigned to it. This cost is the cost of constructing the link represented by the edge. Every connected subgraph that includes all the vertices represents a feasible solution. Under the assumption that all weights are nonnegative, the set of feasible solutions can be narrowed to the set of spanning trees of the graph. An optimal solution is a spanning tree with minimum cost.

In this problem we need to select a subset of the edges. This subset must satisfy the following constraint: *The set of selected edges forms a spanning tree.* The optimization function is the sum of the weights of the selected edges. ■

17.2 THE GREEDY METHOD

In the **greedy method** we attempt to construct an optimal solution in stages. At each stage we make a decision that appears to be the best (under some criterion) at the time. A decision made in one stage is not changed in a later stage, so each decision should assure feasibility. The criterion used to make the greedy decision at each stage is called the **greedy criterion**.

Example 17.4 [Change Making] A child buys candy valued at less than \$1 and gives a \$1 bill to the cashier. The cashier wishes to return change using the fewest number of coins. Assume that an unlimited supply of quarters, dimes, nickels, and pennies is available. The cashier constructs the change in stages. In each stage a coin is added to the change. This coin is selected using the greedy criterion: *At each stage increase the total amount of change constructed by as much as possible.* To assure feasibility (i.e., the change given exactly equals the desired amount) of the solution, the selected coin should not cause the total amount of change given so far to exceed the final desired amount.

Suppose that 67 cents in change is due the child. The first two coins selected are quarters. A quarter cannot be selected for the third coin because such a selection results in an infeasible selection of coins (change exceeds 67 cents). The third coin

selected is a dime, then a nickel is selected, and finally two pennies are added to the change.

The greedy method has intuitive appeal in that we construct the change by using a strategy that our intuition tells us should result in the fewest (or at least close to the fewest) number of coins being given out. We can actually prove that the greedy algorithm just described does indeed generate change with the fewest number of coins (see Exercise 1). ■

Example 17.5 [Machine Scheduling] You are given n tasks and an infinite supply of machines on which these tasks can be performed. Each task has a start time s_i and a finish time f_i , $s_i < f_i$. $[s_i, f_i]$ is the processing interval for task i . Two tasks i and j overlap iff their processing intervals overlap at a point other than the interval start or end. For example, the interval $[1, 4]$ overlaps with $[2, 4]$, but not with $[4, 7]$. A **feasible** task-to-machine assignment is an assignment in which no machine is assigned two overlapping tasks. Therefore, in a feasible assignment each machine works on at most one task at any time. An **optimal assignment** is a feasible assignment that utilizes the fewest number of machines.

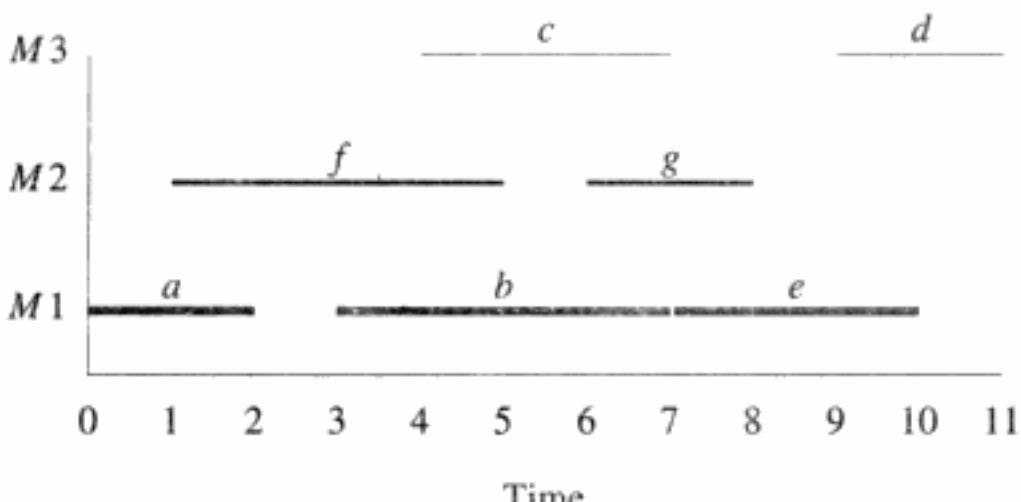
Suppose we have $n = 7$ tasks labeled a through g and that their start and finish times are as shown in Figure 17.1(a). The following task-to-machine assignment is a feasible assignment that utilizes seven machines: Assign task a to machine $M1$, task b to machine $M2$, ..., task g to machine $M7$. This assignment is not an optimal assignment because other assignments use fewer machines. For example, we can assign tasks a , b , and d to the same machine, reducing the number of utilized machines to five.

A greedy way to obtain an optimal task assignment is to assign the tasks in stages, one task per stage and in nondecreasing order of task start times. Call a machine **old** if at least one task has been assigned to it. If a machine is not old, it is **new**. For machine selection, use the greedy criterion: *If an old machine becomes available by the start time of the task to be assigned, assign the task to this machine; if not, assign it to a new machine.*

For the data of Figure 17.1(a), the tasks in nondecreasing order of task start time are a, f, b, c, g, e, d . The greedy algorithm assigns tasks to machines in this order. The algorithm has $n = 7$ stages, and in each stage one task is assigned to a machine. Stage 1 has no old machines, so a is assigned to a new machine (say, $M1$). This machine is now busy from time 0 to time 2 (see Figure 17.1(b)). In stage 2 task f is considered. Since the only old machine is busy when task f is to start, it is assigned to a new machine (say, $M2$). When task b is considered in stage 3, we find that the old machine $M1$ is free at time $s_b = 3$, so b is assigned to $M1$. The availability time for $M1$ becomes $f_b = 7$, and that for $M2$ is $f_f = 5$. In stage 4 task c is considered. Since neither of the old machines is available at time $s_c = 4$, task c is assigned to a new machine (say, $M3$). The availability time of this machine becomes $f_c = 7$. Task g is considered in stage 5 and assigned to machine $M2$, which is the first to become available. Task e is assigned in stage 6 to machine

task	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>
start	0	3	4	9	7	1	6
finish	2	7	7	11	10	5	8

(a) Seven tasks



(b) Schedule

Figure 17.1 Tasks and a three-machine schedule

*M*1 or *M*3 (assume it is assigned to *M*1), and finally, in stage 7, task *d* is assigned to machine *M*2 or *M*3 (Figure 17.1(b) assumes task *d* is assigned to *M*3).

The proof that the described greedy algorithm generates optimal assignments is left as an exercise (Exercise 7). The algorithm may be implemented to have complexity $O(n \log n)$ by sorting the tasks in nondecreasing order of s_i , using an $O(n \log n)$ sort (such as heap sort) and then using a min heap of availability times for the old machines. ■

Example 17.6 [Shortest Path] You are given a directed network as in Figure 17.2. The length of a path is defined to be the sum of the costs of the edges on the path. You are to find a shortest path from a start vertex *s* to a destination vertex *d*.

A greedy way to construct such a path is to do so in stages. In each stage a vertex is added to the path. Suppose that the path built so far ends at vertex *q* and *q* is not the destination vertex *d*. The vertex to add in the next stage is obtained using the greedy criterion: *Select a nearest vertex adjacent from q that is not already on the path.*

This greedy method does not necessarily obtain a shortest path. For example, suppose we wish to construct a shortest path from vertex 1 to vertex 5 in Figure 17.2.

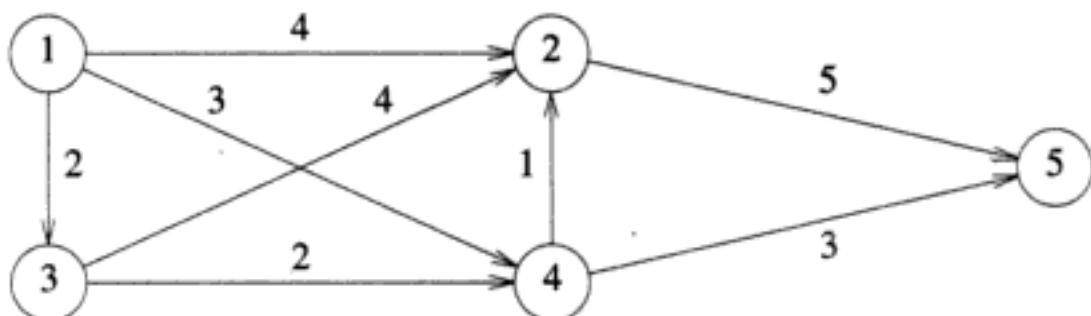


Figure 17.2 Sample digraph

Using the greedy method just outlined, we begin at vertex 1 and move to the nearest vertex not already on the path. We move to vertex 3, a distance of only two units. From vertex 3 the nearest vertex we can move to is vertex 4. From vertex 4 we move to vertex 2 and then to the destination vertex 5. The constructed path is 1, 3, 4, 2, 5, and its length is 10. This path is not the shortest 1-to-5 path in the digraph. In fact, several shorter paths exist. For instance, the path 1, 4, 5 has length 6. ■

Now that you have seen three examples of a greedy algorithm, you should be able to go over the applications considered in earlier chapters and identify several of the solutions developed as greedy ones. For example, the Huffman tree algorithm of Section 12.6.3 constructs a binary tree with minimum weighted external path length in $n - 1$ stages. In each stage two binary trees are combined to create a new binary tree. This algorithm uses the greedy criterion: Of the available binary trees, combine two with the least weight.

The LPT-scheduling (longest processing time first) rule of Section 12.6.2 is a greedy one. It schedules the n jobs in n stages. First the jobs are ordered by length. Then in each stage a machine is selected for the next job. The machine is selected using the greedy criterion: *Minimize the length of the schedule constructed so far*. This criterion translates into scheduling the job on the machine on which it finishes first. This machine is also the machine that becomes idle first.

Notice that in the case of the machine-scheduling problem of Section 12.6.2, the greedy algorithm does not guarantee optimal solutions. However, it is intuitively appealing and generally produces solutions that are very close in value to the optimal. It uses a rule of thumb that we might expect a human machine scheduler to use in a real scheduling environment. Algorithms that do not guarantee optimal solutions but generally produce solutions that are close to optimal are called **heuristics**. So the LPT method is a heuristic for machine scheduling. Theorem 9.2 states a bound between the finish time of LPT schedules and optimal schedules, so the LPT heuristic has **bounded performance**. A heuristic with bounded performance is an **approximation algorithm**.

Section 13.5.1 stated several bounded performance heuristics (i.e., approxima-

tion algorithms) for the bin-packing problem. Each of these heuristics is a greedy heuristic. The LPT method of Section 12.6.2 is also a greedy heuristic. All these heuristics have intuitive appeal and, in practice, yield solutions much closer to optimal than suggested by the bounds given in that section.

The rest of this chapter presents several applications of the greedy method. In some applications the result is an algorithm that always obtains an optimal solution. In others the algorithm is just a heuristic that may or may not be an approximation algorithm.

EXERCISES

1. Show that the greedy algorithm for the change-making problem (Example 17.4) generates change with the fewest number of coins when the cashier has an unlimited supply of quarters, dimes, nickels, and pennies.
2. Consider the change-making problem of Example 17.4. Suppose that the cashier has only a limited number of quarters, dimes, nickels, and pennies. Formulate a greedy solution to the change-making problem. Does your algorithm always use the fewest number of coins? Prove your result.
3. Extend the algorithm of Example 17.4 to the case when the cashier has \$100, \$50, \$20, \$10, \$5, and \$1 bills in addition to coins and a customer gives u dollars and v cents as payment toward a purchase of x dollars and y cents. Does your algorithm always generate change with the fewest total number of bills and coins? Prove this.
4. Write a C++ program implementing the change-making solution of Example 17.4. Assume that the cashier has bills in the denominations \$100, \$50, \$20, \$10, \$5, and \$1 in addition to coins. Your program should include a method to input the purchase amount and the amount given by the customer as well as a method to output the amount of change and a breakdown by denomination.
5. Suppose that some country has coins in the denominations 14, 12, 5, and 1 cents. Will the greedy method of Example 17.4 always generate change with the fewest number of coins? Prove your answer.
6. (a) Show that the greedy algorithm of Example 17.5 always finds an optimal task assignment.
(b) Program your algorithm so that its complexity is $O(n \log n)$; n is the number of tasks.
7. Consider the machine-scheduling problem of Example 17.5. Assume that only one machine is available and that we are to select the largest number of tasks that can be scheduled on this machine. For the example, the largest task

selection is $\{a, b, e\}$. A greedy algorithm for this task-selection problem would select the tasks in stages. In each stage one task is selected using the following criterion: *From the remaining tasks, select the one that has the least finish time and does not overlap with any of the already selected tasks.*

- (a) Show that this greedy algorithm obtains optimal selections.
- (b) Develop an $O(n \log n)$ implementation of this algorithm. (*Hint:* Use a min heap of finish times.)

17.3 APPLICATIONS

17.3.1 Container Loading

A Greedy Solution

The terminology is from Example 17.2. The ship may be loaded in stages; one container per stage. At each stage we need to decide which container to load. For this decision we may use the greedy criterion: *From the remaining containers, select the one with least weight.* This order of selection will keep the total weight of the selected containers minimum and hence leave maximum capacity for loading more containers. Using the greedy algorithm just outlined, we first select the container that has least weight, then the one with the next smallest weight, and so on until either all containers have been loaded or there isn't enough capacity for the next one.

Example 17.7 Suppose that $n = 8$, $[w_1, \dots, w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$, and $c = 400$. When the greedy algorithm is used, the containers are considered for loading in the order 7, 3, 6, 8, 4, 1, 5, 2. Containers 7, 3, 6, 8, 4, and 1 together weigh 390 units and are loaded. The available capacity is now 10 units, which is inadequate for any of the remaining containers. In the greedy solution we have $[x_1, \dots, x_8] = [1, 0, 1, 1, 0, 1, 1, 1]$ and $\sum x_i = 6$. ■

Correctness of Greedy Algorithm

Theorem 17.1 *The greedy algorithm generates optimal loadings.*

Proof Let $x = [x_1, \dots, x_n]$ be the solution produced by the greedy algorithm and let $y = [y_1, \dots, y_n]$ be any feasible solution. We will show that $\sum_{i=1}^n x_i \geq \sum_{i=1}^n y_i$. Without loss of generality we may assume that the containers have been ordered so that $w_i \leq w_{i+1}$, $1 \leq i < n$. From the way the greedy algorithm works, we know that there is a k , $0 \leq k \leq n$ such that $x_i = 1$, $i \leq k$, and $x_i = 0$, $i > k$.

We use induction on the number p of positions i such that $x_i \neq y_i$. For the induction base, we see that when $p = 0$, x and y are the same. So $\sum_{i=1}^n x_i \geq$

Hidden page

```
void containerLoading(container* c, int capacity,
                      int numberofContainers, int* x)
{// Greedy algorithm for container loading.
// Set x[i] = 1 iff container i, i >= 1 is loaded.
// sort into increasing order of weight
heapSort(c, numberofContainers);

int n = numberofContainers;

// initialize x
for (int i = 1; i <= n; i++)
    x[i] = 0;

// select containers in order of weight
for (int i = 1; i <= n && c[i].weight <= capacity; i++)
{// enough capacity for container c[i].id
    x[c[i].id] = 1;
    capacity -= c[i].weight; // remaining capacity
}
}
```

Program 17.1 Loading containers

subject to the constraints

$$\sum_{i=1}^n w_i x_i \leq c \text{ and } x_i \in \{0, 1\}, 1 \leq i \leq n$$

In this formulation we are to find the values of x_i . $x_i = 1$ means that object i is packed into the knapsack, and $x_i = 0$ means that object i is not packed. The 0/1 knapsack problem is really a generalization of the container-loading problem to the case where the profit earned from each container is different. In the context of the knapsack problem, the ship is the *knapsack*, and the containers are *objects* that may be packed into the knapsack.

Example 17.8 You are the first-prize winner in a grocery-store contest, and the prize is a free cart load of groceries. There are n different items available in the store, and the contest rules stipulate that you can pick at most one of each. The cart has a capacity of c , and item i takes up w_i amount of cart space. The cost of item i is p_i . Your objective is to fill the cart with groceries that have the maximum value. Of course, you cannot exceed the cart capacity, and you cannot take two of any item. The problem may be modeled by using the 0/1 knapsack formulation.

Hidden page

no. To see this, consider the instance $n = 2$, $w = [1, y]$, $p = [10, 9y]$, and $c = y$. The greedy solution is $x = [1, 0]$. This solution has value 10. For $y \geq 10/9$, the optimal solution has value $9y$. Therefore, the value of the greedy solution is $(9y - 10)/(9y) * 100$ percent away from the optimal value. For large y this value approaches 100 percent.

We can modify the greedy heuristic to provide solutions within x percent of optimal for $x < 100$. First we place a subset of at most k objects into the knapsack. If this subset has weight greater than c , we discard it. Otherwise, the remaining capacity is filled by considering the remaining objects in decreasing order of p_i/w_i . The best solution obtained considering all possible subsets with at most k objects is the solution generated by the heuristic.

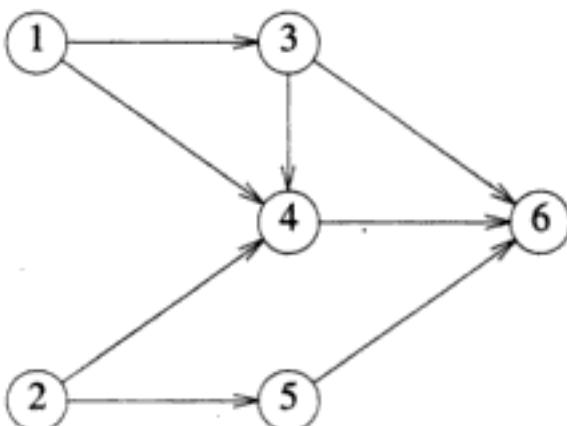
Example 17.9 Consider the knapsack instance $n = 4$, $w = [2, 4, 6, 7]$, $p = [6, 10, 12, 13]$, and $c = 11$. When $k = 0$, the knapsack is filled in nonincreasing order of profit density. First we place object 1 into the knapsack, then object 2. The capacity that remains at this time is five units. None of the remaining objects fits, and the solution $x = [1, 1, 0, 0]$ is produced. The profit earned from this solution is 16.

Let us now try the greedy heuristic with $k = 1$. The subsets to begin with are $\{1\}$, $\{2\}$, $\{3\}$, and $\{4\}$. The subsets $\{1\}$ and $\{2\}$ yield the same solution as obtained with $k = 0$. When the subset $\{3\}$ is considered, x_3 is set to 1. Five units of capacity remain, and we attempt to use this capacity by considering the remaining objects in nonincreasing order of profit density. Object 1 is considered first. It fits, and x_1 is set to 1. At this time only three units of capacity remain, and none of the remaining objects can be added to the knapsack. The solution obtained when we begin with the subset $\{3\}$ in the knapsack is $x = [1, 0, 1, 0]$. The profit earned from this solution is 18. When we begin with the subset $\{4\}$, we produce the solution $x = [1, 0, 0, 1]$ that has a profit value of 19. The best solution obtained considering subsets of size 0 and 1 is $[1, 0, 0, 1]$. This solution is produced by the greedy heuristic when $k = 1$.

If $k = 2$, then in addition to the subsets considered for $k < 2$, we need to consider the subsets $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, and $\{3, 4\}$. The last of these subsets represents an infeasible starting point and is discarded. For the remaining, the solutions obtained are $[1, 1, 0, 0]$, $[1, 0, 1, 0]$, $[1, 0, 0, 1]$, $[0, 1, 1, 0]$, and $[0, 1, 0, 1]$. The last of these solutions has the profit value 23, which is higher than that obtained from the subsets of size 0 and 1. This solution is therefore the solution produced by the heuristic. ■

The solution produced by the modified greedy heuristic is k -optimal. That is, if we remove up to k objects from the solution and put back up to k , the new solution is no better than the original. Further, the value of a solution obtained in this manner comes within $100/(k + 1)$ percent of optimal. Therefore, we refer to this heuristic as a *bounded performance* heuristic. When $k = 1$, the solutions are guaranteed to have value within 50 percent of optimal; when $k = 2$, they are

Hidden page

**Figure 17.4** A task digraph

the process of constructing a topological order from a task digraph is **topological sorting**.

The task digraph of Figure 17.4 has several topological orders. Three of these orders are 123456, 132456, and 215346. The sequence 142356 is not a topological order, as (for example) task 4 precedes task 3 in this sequence, whereas the task digraph contains the edge (3,4). This sequence violates the precedence dictated by this edge (and others).

A Greedy Solution

We may formulate a greedy algorithm to construct a topological order or sequence. This algorithm constructs the sequence from left to right in stages. In each stage we add a vertex to the sequence. We select the new vertex using the greedy criterion: *From the remaining vertices, select a vertex w that has no incoming edge (v, w) with the property that v hasn't already been placed into the sequence.* Notice that if we add a vertex w that violates this criterion (i.e., the digraph has an edge (v, w) and vertex v is not part of the constructed sequence), then we cannot complete the sequence in a topological order, as vertex v will necessarily come after vertex w . A high-level statement of the greedy algorithm appears in Figure 17.5. Each iteration of the `while` loop represents a stage of the greedy algorithm.

Let us try out this algorithm on the digraph of Figure 17.4. We start with an empty sequence `theOrder`. In the first stage we select the first vertex for `theOrder`. The digraph has two candidate vertices, 1 and 2, for the first position in the sequence. If we select vertex 2, the sequence becomes `theOrder = 2` and stage 1 is complete. In stage 2 we select the second vertex for `theOrder`. Applying the greedy criterion with `theOrder = 2`, we see that the candidate vertices are 1 and 5. If we select vertex 5, then `theOrder = 25`. For the next stage vertex 1 is the only candidate for w . Following stage 3 `theOrder = 251`. In stage 4 vertex 3 is the only

Let n be the number of vertices in the digraph.

Let theOrder be an empty sequence.

while (**true**)

{

 Let w be any vertex that has no incoming edge (v, w) such that v is not in theOrder .

if there is no such w , **break**.

 Add w to the end of theOrder .

}

if (theOrder has fewer than n vertices)

 the algorithm fails.

else

theOrder is a topological sequence.

Figure 17.5 Topological sorting

candidate for w . Thus we add vertex 3 to theOrder to get $\text{theOrder} = 2513$. In the next two stages, we add vertices 4 and 6 to get $\text{theOrder} = 251346$.

Correctness of the Greedy Algorithm

To establish the correctness of the greedy algorithm, we need to show (1) that when the algorithm fails, the digraph has no topological sequence and (2) that when the algorithm doesn't fail, theOrder is, in fact, a topological sequence. Item (2) is a direct consequence of the greedy criterion used to select the next vertex. For (1) we show in Lemma 17.1 that when the algorithm fails, the digraph has a cycle. When the digraph has a cycle, $q_j q_{j+1} \cdots q_k q_j$, there can be no topological order, as the sequence of precedences defined by the cycle implies that q_j must finish before q_j can start.

Lemma 17.1 *If the algorithm of Figure 17.5 fails, the digraph has a cycle.*

Proof Note that upon failure $|\text{theOrder}| < n$ and there are no candidates for inclusion in theOrder . So there is at least one vertex, q_1 , that is not in theOrder . The digraph must contain an edge (q_2, q_1) where q_2 is not in theOrder ; otherwise, q_1 is a candidate for inclusion in theOrder . Similarly, there must be an edge (q_3, q_2) such that q_3 is not in theOrder . If $q_3 = q_1$, then $q_1 q_2 q_3$ is a cycle in the digraph. If $q_3 \neq q_1$, there must be a q_4 such that (q_4, q_3) is an edge and q_4 is not in theOrder ; otherwise, q_3 is a candidate for inclusion in theOrder . If q_4 is one of q_1, q_2 , or q_3 , then again the digraph has a cycle. Since the digraph has a finite number n of vertices, continued application of this argument will eventually detect a cycle. ■

Selection of Data Structures

To refine the algorithm of Figure 17.5 into C++ code, we must decide on a representation for the sequence *theOrder*, as well as how to detect candidates for inclusion into *theOrder*. An efficient implementation results if we represent *theOrder* as a one-dimensional array; use a stack to keep track of all vertices that are candidates for inclusion into *theOrder*; and use a one-dimensional array *inDegree* such that *inDegree*[*j*] is the number of vertices *i* for which (*i*, *j*) is an edge of the digraph and *i* is not a member of *theOrder*. A vertex *j* becomes a candidate for inclusion in *theOrder* when *inDegree*[*j*] becomes 0. *theOrder* is initially the empty sequence, and *inDegree*[*j*] is simply the in-degree of vertex *j*. Each time we add a vertex to *x*, *inDegree*[*j*] decreases by 1 for all *j* that are adjacent from the added vertex.

For the digraph of Figure 17.4, *inDegree*[1:6] = [0, 0, 1, 3, 1, 3] in the beginning. Vertices 1 and 2 are candidates for inclusion in *theOrder*, as their *inDegree* value is 0. Therefore, we start with 1 and 2 on the stack. In each stage we remove a vertex from the stack and add that vertex to *theOrder*. We also reduce the *inDegree* values of the vertices that are adjacent from the vertex just added to *theOrder*. If vertex 2 is removed from the stack and added to *theOrder* in stage 1, we get *theOrder*[0] = 2 and *inDegree*[1:6] = [0, 0, 1, 2, 0, 3]. Since *inDegree*[5] has just become 0, it is added to the stack.

C++ Implementation

Program 17.2 gives the resulting C++ code. This code is defined as a method of the abstract class *graph* (Program 16.1). Consequently, it can be used for digraphs with and without edge weights. The method *topologicalOrder* returns *true* if a topological order is found and *false* if the input digraph does not have a topological order. When a topological order is found, the order is returned in *theOrder*[0:*n*-1], where *n* is the number of vertices in the digraph.

Complexity Analysis

The total time spent in the first **for** loop is $O(n^2)$ if we use an adjacency-matrix representation and $O(n+e)$ if we use a linked-adjacency-list representation. Here *n* is the number of vertices and *e* is the number of edges. The second **for** loop takes $O(n)$ time. Of the two nested **while** loops, the outer one is iterated at most *n* times. Each iteration adds a vertex *nextVertex* to *theOrder* and initiates the inner **while** loop. When adjacency matrices are used, this inner **while** loop takes $O(n)$ time for each *nextVertex*. When we use linked adjacency lists, this loop takes $d_{\text{nextVertex}}^{\text{out}}$ time. Therefore, the time spent on the inner **while** loop is either $O(n^2)$ or $O(n+e)$. Hence the complexity of Program 17.2 is $O(n^2)$ when we use adjacency matrices and $O(n+e)$ when we use linked adjacency lists.

```
bool topologicalOrder(int *theOrder)
{// Return false iff the digraph has no topological order.
 // theOrder[0:n-1] is set to a topological order when
 // such an order exists.

 // code to verify that *this is a digraph comes here

 int n = numberVertices();
 // compute in-degrees
 int *inDegree = new int [n + 1];
 fill(inDegree + 1, inDegree + n + 1, 0);
 for (int i = 1; i <= n; i++)
 { // edges out of vertex i
     vertexIterator<T> *ii = iterator(i);
     int u;
     while ((u = ii->next()) != 0)
         // visit an adjacent vertex of i
         inDegree[u]++;
 }
 // stack vertices with zero in-degree
 arrayStack<int> stack;
 for (int i = 1; i <= n; i++)
     if (inDegree[i] == 0)
         stack.push(i);
 // generate topological order
 int j = 0; // cursor for array theOrder
 while (!stack.empty())
 { // select from stack
     int nextVertex = stack.top();
     stack.pop();
     theOrder[j++] = nextVertex;
     // update in-degrees
     vertexIterator<T> *iNextVertex = iterator(nextVertex);
     int u;
     while ((u = iNextVertex->next()) != 0)
     { // visit an adjacent vertex of nextVertex
         inDegree[u]--;
         if (inDegree[u] == 0)
             stack.push(u);
     }
 }
 return (j == n);
}
```

Program 17.2 Topological sorting

Hidden page

elements of the universe U . An edge exists between a vertex in A and one in B iff the corresponding element of U is in the corresponding set of S .

Example 17.11 Let $S = \{S_1, \dots, S_5\}$, $U = \{4, 5, \dots, 15\}$, $S_1 = \{4, 6, 7, 8, 9, 13\}$, $S_2 = \{4, 5, 6, 8\}$, $S_3 = \{8, 10, 12, 14, 15\}$, $S_4 = \{5, 6, 8, 12, 14, 15\}$, and $S_5 = \{4, 9, 10, 11\}$. $S' = \{S_1, S_4, S_5\}$ is a cover of size 3. No smaller cover exists, so it is a minimum cover. The set-cover instance may be mapped into the bipartite graph of Figure 17.6 using vertices 1, 2, 3, 16, and 17 to represent sets S_1 , S_2 , S_3 , S_4 , and S_5 , respectively. Vertex j represents universe element j , $4 \leq j \leq 15$. ■

A Greedy Heuristic

The set-cover problem is known to be NP-hard. Since the set-cover and bipartite-cover problems are identical, the bipartite-cover problem is also NP-hard. As a result we probably will not be able to develop a fast algorithm to solve it. We can, however, use the greedy method to develop a fast heuristic. One possibility is to construct the cover A' in stages. In each stage we select a vertex of A for inclusion into the cover. This vertex is selected by using the greedy criterion: *Select a vertex of A that covers the largest number of uncovered vertices of B .*

Example 17.12 Consider the bipartite graph of Figure 17.6. Initially $A' = \emptyset$, and no vertex of B is covered. Vertices 1 and 16 each covers six uncovered vertices of B ; vertex 3 covers five; and vertices 2 and 17 each covers four. Therefore, in the first stage, we add either vertex 1 or vertex 16 to A' . If we add vertex 16, it covers the vertices $\{5, 6, 8, 12, 14, 15\}$. The uncovered vertices are $\{4, 7, 9, 10, 11, 13\}$. Vertex 1 of A covers four of these uncovered vertices ($\{4, 7, 9, 13\}$), vertex 2 covers one ($\{4\}$), vertex 3 covers one ($\{10\}$), vertex 16 covers zero, and vertex 17 covers four. In the next stage, either 1 or 17 is selected for inclusion into A' . If we choose vertex 1, vertices $\{10, 11\}$ remain uncovered. Vertices 1, 2, and 16 cover none of these uncovered vertices; vertex 3 covers one; and vertex 17 covers two. Therefore, we select vertex 17. Now no uncovered vertices remain, and we are done. $A' = \{1, 16, 17\}$. ■

A high-level statement of the greedy covering heuristic appears in Figure 17.7. You should be able to show (1) that the algorithm fails to find a cover iff the initial bipartite graph does not have a cover, and (2) that bipartite graphs exist on which the heuristic will fail to find a minimum cover.

Selection of Data Structure and Complexity

To implement the algorithm of Figure 17.7, we need to select a representation for A' and to decide how to keep track of the number of uncovered vertices of B that each vertex of A covers. Since only additions are made to the set A' , we can represent A' as a one-dimensional integer array `theCover` and use `coverSize`,

Hidden page

Hidden page

Hidden page

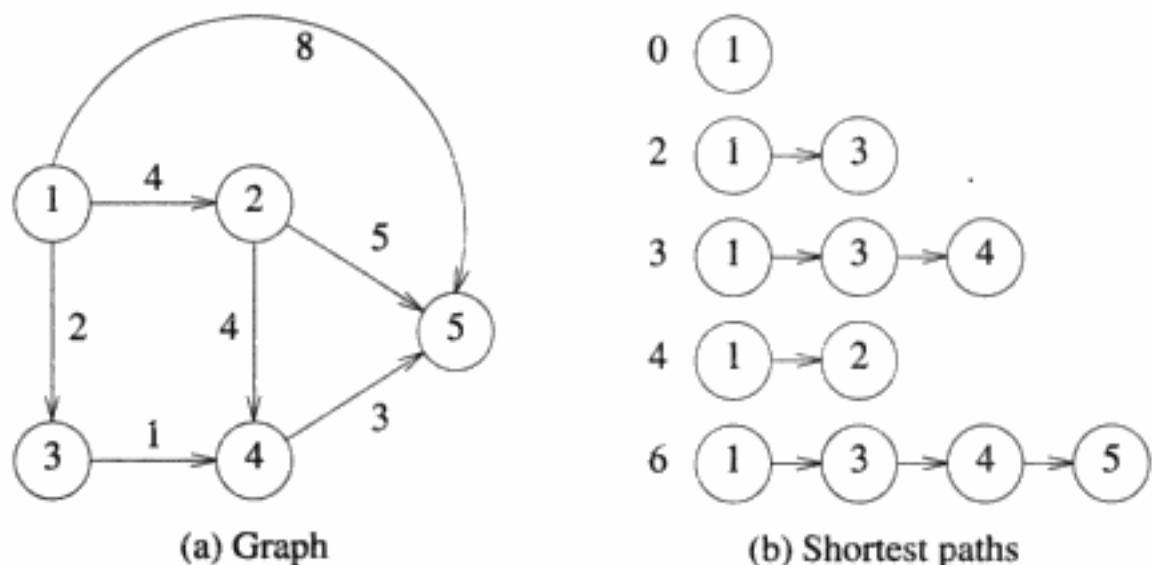


Figure 17.9 Shortest-paths example

path to a new destination vertex is generated. The destination vertex for the next shortest path is selected using the greedy criterion: *From the vertices to which a shortest path has not been generated, select one that results in the least path length.* In other words, Dijkstra's method generates the shortest paths in increasing order of length.

We begin with the trivial path from the source vertex to itself. This path has no edges and has a length of 0. In each stage of the greedy algorithm, the next shortest path is generated. This next shortest path is the shortest possible one-edge extension of an already generated shortest path (Exercise 31). For the example of Figure 17.9, the second path of (b) is a one-edge extension of the first; the third is a one-edge extension of the second; the fourth is a one-edge extension of the first; and the fifth is a one-edge extension of the third.

This observation results in a convenient way to store the shortest paths. We can use an array `predecessor` such that `predecessor[i]` gives the vertex that immediately precedes vertex i on the shortest `sourceVertex` to i path. For our example `predecessor[1:5] = [0, 1, 1, 3, 4]`. The path from `sourceVertex` to any vertex i may be constructed backward from i and following the sequence `predecessor[i], predecessor[predecessor[i]], predecessor[predecessor[predecessor[i]]], ...` until we reach `sourceVertex`. If we begin our example with $i = 5$, we get the vertex sequence `predecessor[i] = 4, predecessor[4] = 3, predecessor[3] = 1 = sourceVertex`. Therefore, the path is 1, 3, 4, 5.

To facilitate the generation of shortest paths in increasing order of length, we define `distanceFromSource[i]` to be the length of the shortest one-edge extension of a path already generated so that the extended path ends at vertex i . When

Hidden page

Hidden page

- Step 1:** Initialize `distanceFromSource[i] = a[sourceVertex][i]`, $1 \leq i \leq n$.
Set `predecessor[i] = sourceVertex` for all i adjacent from `sourceVertex`.
Set `predecessor[sourceVertex] = 0` and `predecessor[i] = -1` for all other vertices.
Create a list `newReachableVertices` of all vertices for which `predecessor[i] > 0` (i.e., `newReachableVertices` now contains all vertices that are adjacent from `sourceVertex`).
- Step 2:** If `newReachableVertices` is empty, terminate. Otherwise, go to step 3.
- Step 3:** Delete from `newReachableVertices` the vertex i with least value of `distanceFromSource` (ties are broken arbitrarily).
- Step 4:** Update `distanceFromSource[j]` to $\min\{\text{distanceFromSource}[j], \text{distanceFromSource}[i] + a[i][j]\}$ for all vertices j adjacent from i . If `distanceFromSource[j]` changes, set `predecessor[j] = i` and add j to `newReachableVertices` in case it isn't already there. Go to step 2.

Figure 17.10 High-level description of Dijkstra's shortest-path algorithm

C++ Implementation

The refinement of Figure 17.10 into the C++ method `adjacencyWDigraph::shortestPaths` appears in Program 17.3. This code uses the class `graphChain` (Section 16.7.3).

Comments on Complexity

The complexity of Program 17.3 is $O(n^2)$. Any shortest-path algorithm must examine each edge in the graph at least once, since any of the edges could be in a shortest path. Hence the minimum possible time for such an algorithm would be $O(e)$. Since cost-adjacency matrices were used to represent the graph, it takes $O(n^2)$ time just to determine which edges are in the digraph. Therefore, any shortest-path algorithm that uses this representation must take $O(n^2)$. For this representation, then, Program 17.3 is optimal to within a constant factor. Even if a change to adjacency lists is made, only the overall time for the last `for` loop can be brought down to $O(e)$ (because `distanceFromSource` can change only for vertices adjacent from i). The total time spent selecting and deleting the minimum-distance vertex from `newReachableVertices` remains $O(n^2)$.

```

// next shortest path is to vertex v, delete v from
// newReachableVertices and update distanceFromSource
newReachableVertices.eraseElement(v);
for (int j = 1; j <= n; j++)
{
    if (a[v][j] != noEdge && (predecessor[j] == -1 ||
        distanceFromSource[j] > distanceFromSource[v] + a[v][j]))
    {
        // distanceFromSource[j] decreases
        distanceFromSource[j] = distanceFromSource[v] + a[v][j];
        // add j to newReachableVertices
        if (predecessor[j] == -1)
            // not reached before
            newReachableVertices.insert(0, j);
        predecessor[j] = v;
    }
}
}
}

```

Program 17.3 Shortest-path program (concluded)

17.3.6 Minimum-Cost Spanning Trees

This problem was considered in Examples 16.2 and 17.3. Since every spanning tree of an n -vertex undirected network G has exactly $n - 1$ edges, the problem is to select $n - 1$ edges in such a way that the selected edges form a least-cost spanning tree of G . We can formulate at least three different greedy strategies to select these $n - 1$ edges. These strategies result in three greedy algorithms for the minimum-cost spanning-tree problem: Kruskal's algorithm, Prim's algorithm, and Sollin's algorithm.

Kruskal's Algorithm

The Method

Kruskal's algorithm selects the $n - 1$ edges one at a time using the greedy criterion: *From the remaining edges, select a least-cost edge that does not result in a cycle when added to the set of already selected edges.* Note that a collection of edges that contains a cycle cannot be completed into a spanning tree. Kruskal's algorithm has up to e stages where e is the number of edges in the network. The e edges are considered in order of increasing cost, one edge per stage. When an edge is considered, it is rejected if it forms a cycle when added to the set of already selected

Hidden page

Hidden page

Hidden page

Clearly, the cost of V is the cost of U plus the cost of edge e minus the cost of edge f . If the cost of e is less than the cost of f , then the spanning tree V has a smaller cost than the tree U , which is impossible (U is a minimum-cost spanning tree).

If e has a higher cost than f , then f is considered before e by Kruskal's algorithm. Since f is not in T , Kruskal's algorithm must have discarded f when f was considered for inclusion in T . Hence f together with edges in T having a cost less than or equal to the cost of f must form a cycle. Since e is a least-cost edge of T that is not in U , all edges of T whose cost is less than that of e (and hence all edges of T whose cost is \leq that of f) are also in U . Hence U must also contain a cycle, but it does not because it is a spanning tree. The assumption that e is of higher cost than f therefore is invalid.

The only possibility left is that e and f have the same cost. Hence V has the same cost as U , and so V is a minimum-cost spanning tree.

Choice of Data Structures and Complexity

To select edges in nondecreasing order of cost, we can set up a min heap and extract edges one by one as needed. When there are e edges in the graph, it takes $O(e)$ time to initialize the heap and $O(\log e)$ time to extract each edge.

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by the set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need merely check whether u and v are in the same vertex set (i.e., in the same component). If so, then a cycle is created. If not, then no cycle is created. Hence two **finds** on the vertex sets suffice. When an edge is included in T , two components are combined into one and a **unite** is to be performed on the two sets. The set operations **find** and **unite** can be carried out efficiently using the tree scheme (together with the weighting rule and path compaction) of Section 11.9.2. The number of **finds** is at most $2e$, and the number of **unites** at most $n - 1$ (exactly $n - 1$ if the weighted undirected graph is connected). Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than $O(n + e)$.

The only operation performed on the set T is that of adding a new edge to it. T may be implemented as an array **spanningTreeEdges** with additions being performed at one end. We can add at most $n - 1$ edges to T . So the total time for operations on T is $O(n)$.

Summing up the various components of the computing time, we get $O(n + e \log e)$ as the asymptotic complexity of Figure 17.12.

C++ Implementation

Using the data structures just described, Figure 17.12 may be refined into C++ code. Program 17.4 gives the resulting code. The method of Program 17.4 is a

Hidden page

member of `graph` and has been written so as to work with all representations of a weighted undirected graph. The class `weightedEdge<T>` defines a type conversion to the data type `T`. This type conversion returns the weight of the edge. Consequently, when edges are extracted from the min heap of Program 17.4 these edges are in ascending order of weight.

The code of Program 17.4 returns `false` if there is no spanning tree and `true` otherwise. Note that when the code returns `true`, a minimum-cost spanning tree is returned in the array `spanningTreeEdges`.

Prim's Algorithm

Prim's algorithm, like Kruskal's, constructs the minimum-cost spanning tree by selecting edges one at a time. The greedy criterion used to determine the next edge to select is *From the remaining edges, select a least-cost edge whose addition to the set of selected edges forms a tree*. Consequently, at each stage the set of selected edges forms a tree. By contrast, the set of selected edges in Kruskal's algorithm forms a forest at each stage.

Prim's algorithm begins with a tree T that contains a single vertex. This vertex can be any of the vertices in the original graph. Then we add a least-cost edge (u, v) to T such that $T \cup \{(u, v)\}$ is also a tree. This edge-addition step is repeated until T contains $n - 1$ edges. Notice that edge (u, v) is always such that exactly one of u and v is in T . A high-level description of Prim's algorithm appears in Figure 17.13. This description also provides for the possibility that the input graph may not be connected. In this case there is no spanning tree. Figure 17.14 shows the progress of Prim's algorithm on the graph of Figure 17.11(a). The refinement of Figure 17.13 into a C++ program and its correctness proof are left as Exercise 37.

Prim's algorithm can be implemented to have a time complexity $O(n^2)$ if we associate with each vertex v not in TV a vertex $near(v)$ such that $near(v) \in TV$ and $cost(v, near(v))$ is minimum over all such choices for $near(v)$. The next edge to add to T is such that $cost(v, near(v))$ is minimum and $v \notin TV$.

Sollin's Algorithm

Sollin's algorithm selects several edges at each stage. At the start of a stage, the selected edges together with the n vertices of the graph form a spanning forest. During a stage we select one edge for each tree in this forest. This edge is a minimum-cost edge that has exactly one vertex in the tree. The selected edges are added to the spanning tree being constructed. Note that two trees in the forest can select the same edge, so we must eliminate duplicates. When several edges have the same cost, two trees can select different edges that connect them. In this case also, we must discard one of the selected edges. At the start of the first stage, the set of selected edges is empty. The algorithm terminates when only one tree remains at the end of a stage or when no edges remain to be selected.

```

// Assume that the network has at least one vertex.
Let  $T$  be the set of selected edges. Initialize  $T = \emptyset$ .
Let  $TV$  be the set of vertices already in the tree. Set  $TV = \{1\}$ .
Let  $E$  be the set of network edges.
while ( $E \neq \emptyset$ ) && ( $|T| \neq n - 1$ )
{
    Let  $(u, v)$  be a least-cost edge such that  $u \in TV$  and  $v \notin TV$ .
    if (there is no such edge)
        break.
     $E = E - \{(u, v)\}$ . // delete edge from  $E$ 
    Add edge  $(u, v)$  to  $T$ .
    Add vertex  $v$  to  $TV$ .
}
if ( $|T| == n - 1$ )
     $T$  is a minimum-cost spanning tree.
else
    The network is not connected and has no spanning tree.

```

Figure 17.13 Prim's minimum-spanning-tree algorithm

Figure 17.15 shows the stages in Sollin's algorithm when it begins with the graph of Figure 17.11(a). The initial configuration of zero selected edges is the same as that shown in Figure 17.11(b). Each tree in this spanning forest is a single vertex. The edges selected by vertices $1, 2, \dots, 7$ are, respectively, $(1, 6), (2, 7), (3, 4), (4, 3), (5, 4), (6, 1), (7, 2)$. The distinct edges in this selection are $(1, 6), (2, 7), (3, 4)$, and $(5, 4)$. Adding these edges to the set of selected edges results in the configuration of Figure 17.15(a). In the next stage the tree with vertex set $1, 6$ selects edge $(6, 5)$, and the remaining two trees select the edge $(2, 3)$. Following the addition of these two edges to the set of selected edges the spanning-tree construction is complete. The constructed spanning tree appears in Figure 17.15(b). The development of Sollin's algorithm into a C++ program and its correctness proof are left as Exercise 38.

Which Algorithm Should You Use?

We have seen three greedy algorithms for the minimum-cost spanning-tree problem. Since all three always construct a minimum-cost spanning tree, you should select one based on performance criteria. Because the space requirements of all three are approximately the same, the decision is made based on their relative time complexities. The asymptotic complexity of Kruskal's method is $O(n + e \log e)$ and that of Prim's method is $O(n^2)$ (though an $O(e + n \log n)$ implementation is also possible; see the solution to Exercise 37 on the Web site); Sollin's method was not analyzed in this text. Experimental results indicate that Prim's method is generally the fastest. Therefore, this method should be used.

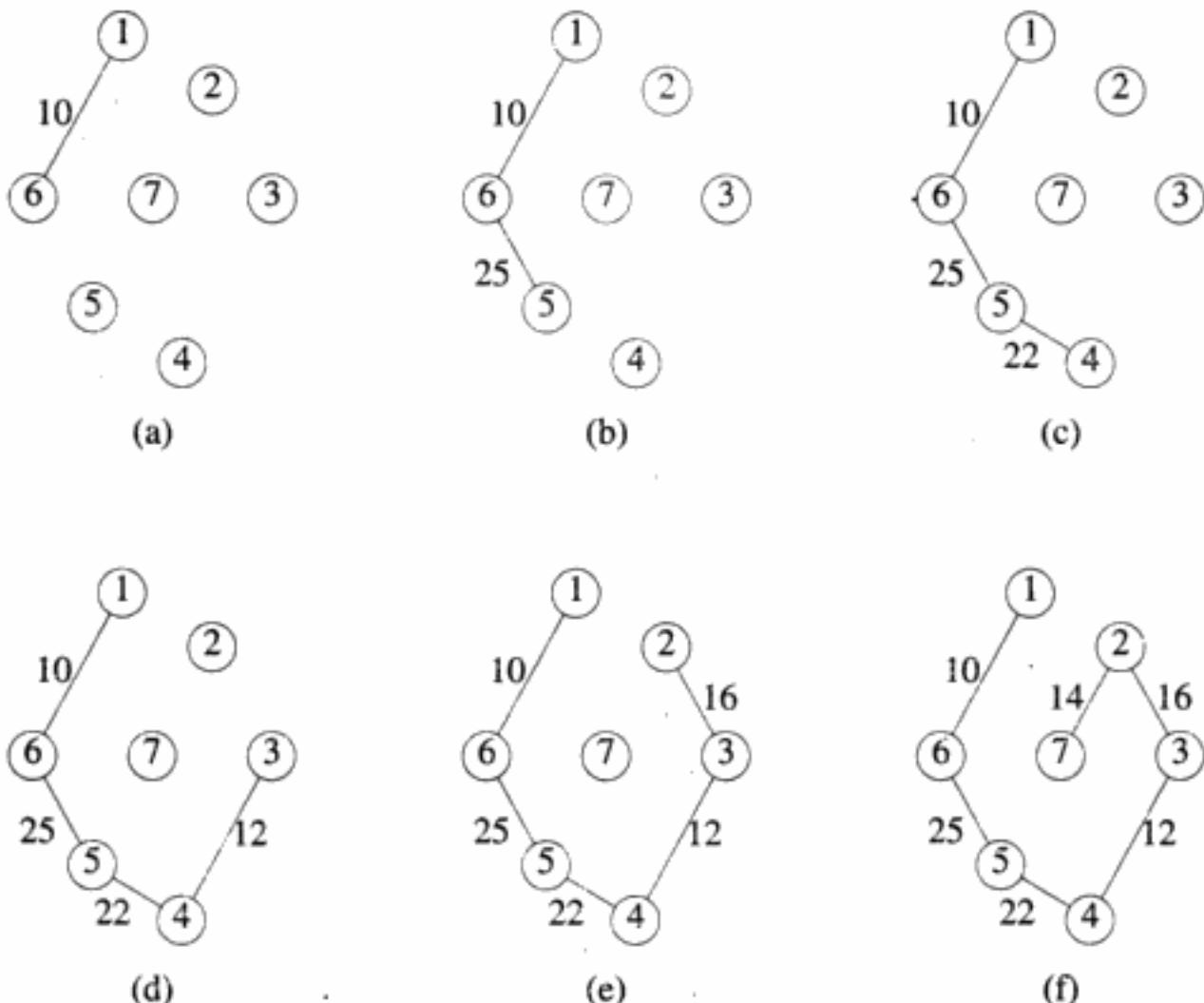


Figure 17.14 Stages in Prim's algorithm

EXERCISES

8. Extend the greedy solution for the loading problem to the case when there are two ships. Does the algorithm always generate optimal solutions?
9. We are given n tasks to perform in sequence. Suppose that task i needs t_i units of time. If the tasks are done in the order $1, 2, \dots, n$, then task i completes at time $c_i = \sum_{j=1}^i t_j$. The average completion time (ACT) is $\frac{1}{n} \sum_{i=1}^n c_i$.
 - (a) Consider the case of four tasks with task times $(4, 2, 8, 1)$. What is the ACT when the task order is $1, 2, 3, 4$?
 - (b) What is the ACT when the task order is $2, 1, 4, 3$?

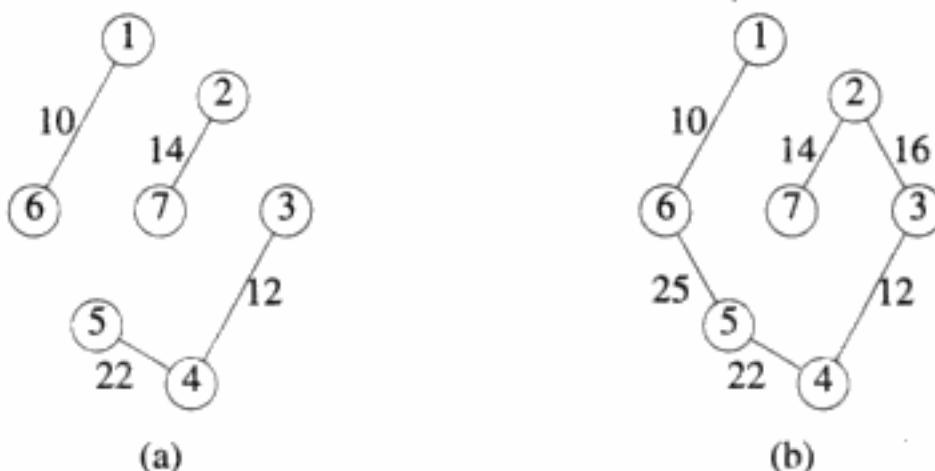


Figure 17.15 Stages in Sollin's algorithm

- (c) The following method constructs a task order that tries to minimize the ACT: *Construct the order in n stages; in each stage select from the remaining tasks one with least task time.* For the example of part (a), this strategy results in the task order 4, 2, 1, 3. What is the ACT for this greedy order?
- (d) Write a C++ program that implements the greedy strategy of (c). The complexity of your program should be $O(n \log n)$. Show that this is so.
- (e) Show that the greedy strategy of (c) results in task orders that have minimum ACT.
10. If two people perform the n tasks of Exercise 9, we need an assignment of tasks to each and an order in which each person is to perform his/her assigned tasks. The task completion times and ACT are defined as in Exercise 9. A possible greedy method that aims to minimize the ACT is as follows: *The two workers select tasks alternately and one at a time; from the remaining tasks, one with least task time is selected; each person does his/her tasks in the order selected.* For the example of Exercise 9(a), if person 1 begins the selection process, he/she selects task 4, person 2 selects task 2, person 1 selects task 1, and finally person 2 selects task 3.
- (a) Write a C++ program to implement this strategy. What is its time complexity?
- (b) Does the outlined greedy strategy always minimize the ACT? Prove your answer.
11. (a) Extend the greedy algorithm of Exercise 10 to the case when m persons are available to do the tasks.

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

36. Write a version of Program 17.3 that works for weighted undirected and directed graphs and is independent of the representation that is used. Your method will be a member of `graph`.
37.
 - (a) Provide a correctness proof for Prim's method (Figure 17.13).
 - (b) Refine Figure 17.13 into a C++ method `graph::prim` with complexity $O(n^2)$. (*Hint:* Use a strategy similar to that used in the shortest paths code of Program 17.3.)
 - (c) Show that the complexity of your method is indeed $O(n^2)$.
38.
 - (a) Prove that Sollin's algorithm finds a minimum-cost spanning tree for every connected undirected graph.
 - (b) What is the maximum number of stages in Sollin's algorithm? Give this as a function of the number of vertices n in the graph.
 - (c) Write a C++ method, `graph::sollin`, that finds a minimum-cost spanning tree using Sollin's algorithm.
 - (d) What is the complexity of your method?
39. Let T be a tree (not necessarily binary) in which a length is associated with each edge. Let S be a subset of the vertices of T and let T/S denote the forest that results when the vertices of S are deleted from T . We wish to find a minimum-cardinality subset S such that no forest in T/S has a root-to-leaf path whose length exceeds d .
 - (a) Develop a greedy algorithm to find a minimum-cardinality S . (*Hint:* Start at the leaves and move toward the root.)
 - (b) Prove the correctness of your algorithm.
 - (c) What is the complexity of your algorithm? In case it is not linear in the number of vertices in T , redesign your algorithm so that its complexity is linear.
40. Do Exercise 39 for the case when T/S denotes the forest that results from making two copies of each vertex in S . The pointer from the parent goes to one copy, and pointers to the children go from the other copy.
41. A convex polygon (Section 6.5.3) with $n > 2$ vertices is to be triangulated (i.e., partitioned or cut into triangles; each corner of a triangle is a vertex of the polygon). A cut starts at a polygon vertex u and ends at a nonadjacent vertex v . The cost of the cut (u, v) is $c(u, v)$. A total of $n - 2$ cuts are required to triangulate the polygon.
 - (a) Formulate a greedy strategy to find a triangulation with minimum cost.
 - (b) Does your strategy always find a minimum-cost triangulation? Prove your answer.

- (c) Write a program that implements your strategy. Test your code.
- (d) What is the complexity of your code?

17.4 REFERENCES AND SELECTED READINGS

Greedy approximation algorithms for several problems appear in the paper “A Survey of Approximately Optimal Solutions to Some Covering and Packing Problems” by V. Paschos, *ACM Computing Surveys*, 29, 2, 1997, 171–209. An experimental evaluation of greedy algorithms for the minimum-cost spanning-tree problem appears in the paper “An Empirical Assessment of Algorithms for Constructing a Minimum Spanning Tree” by B. Moret and H. Shapiro, *DIMACS Series in Discrete Mathematics*, 15, 1994, 99–117.

CHAPTER 18

DIVIDE AND CONQUER

BIRD'S-EYE VIEW

The divide-and-conquer strategy so successfully used by monarchs and colonizers may also be applied to the development of efficient computer algorithms. We begin this chapter by showing how to adapt this ancient strategy to the algorithm-development arena. Then we use the strategy to obtain good algorithms for the minmax problem; matrix multiplication; a problem from recreational mathematics—the defective-chessboard problem; sorting; selection; and a computational geometry problem—find the closest pair of points in two-dimensional space.

Since divide-and-conquer algorithms decompose a problem instance into several smaller independent instances, divide-and-conquer algorithms may be effectively run on a parallel computer; the independent smaller instances can be worked on by different processors of the parallel computer.

This chapter develops the mathematics needed to analyze the complexity of frequently occurring divide-and-conquer algorithms and proves that the divide-and-conquer algorithms for the minmax and sorting problems are optimal by deriving lower bounds on the complexity of these problems. The derived lower bounds agree with the complexity of the divide-and-conquer algorithms for these problems.

18.1 THE METHOD

The divide-and-conquer methodology is very similar to the modularization approach to software design. Small instances of a problem are solved using some direct approach. To solve a large instance, we (1) divide it into two or more smaller instances, (2) solve each of these smaller problems, and (3) combine the solutions of these smaller problems to obtain the solution to the original instance. The smaller instances are often instances of the original problem and may be solved by using the divide-and-conquer strategy recursively.

Example 18.1 [Detecting a Counterfeit Coin] You are given a bag with 16 coins and told that one of these coins may be counterfeit. Further, you are told that counterfeit coins are lighter than genuine ones. Your task is to determine whether the bag contains a counterfeit coin. To aid you in this task, you have a machine that compares the weights of two sets of coins and tells you which set is lighter or whether both sets have the same weight.

We can compare the weights of coins 1 and 2. If coin 1 is lighter than coin 2, then coin 1 is counterfeit and we are done with our task. If coin 2 is lighter than coin 1, then coin 2 is counterfeit. If both coins have the same weight, we compare coins 3 and 4. Again, if one coin is lighter, a counterfeit coin has been detected and we are done. If not, we compare coins 5 and 6. Proceeding in this way, we can determine whether the bag contains a counterfeit coin by making at most eight weight comparisons. This process also identifies the counterfeit coin.

Another approach is to use the divide-and-conquer methodology. Suppose that our 16-coin instance is considered a large instance. In step 1, we divide the original instance into two or more smaller instances. Let us divide our 16-coin instance into two 8-coin instances by arbitrarily selecting 8 coins for the first instance (say A) and the remaining 8 coins for the second instance B . In step 2, we need to determine whether A or B has a counterfeit coin. For this step we use our machine to compare the weights of the coin sets A and B . If both sets have the same weight, a counterfeit coin is not present in the 16-coin set. If A and B have different weights, a counterfeit coin is present and it is in the lighter set. Finally, in step 3 we take the results from step 2 and generate the answer for the original 16-coin instance. For the counterfeit-coin problem, step 3 is easy. The 16-coin instance has a counterfeit coin iff either A or B has one. So with just one weight comparison, we can complete the task of determining the presence of a counterfeit coin.

Now suppose we need to identify the counterfeit coin. We will define a “small” instance to be one with two or three coins. Note that if there is only one coin, we cannot tell whether it is counterfeit. All other instances are “large” instances. If we have a small instance, we may identify the counterfeit coin by comparing one of the coins with up to two other coins, performing at most two weight comparisons.

The 16-coin instance is a large instance. So it is divided into two 8-coin instances A and B as above. By comparing the weights of these two instances, we determine whether or not a counterfeit coin is present. If not, the algorithm terminates. Otherwise, we continue with the subinstance known to have the counterfeit coin.

Suppose B is the lighter set. It is divided into two sets of four coins each. Call these sets $B1$ and $B2$. The two sets are compared. One set of coins must be lighter. If $B1$ is lighter, the counterfeit coin is in $B1$ and $B1$ is divided into two sets of two coins each. Call these sets $B1a$ and $B1b$. The two sets are compared, and we continue with the lighter set. Since the lighter set has only two coins, it is a small instance. Comparing the weights of the two coins in the lighter set, we can determine which is lighter. The lighter one is the counterfeit coin. ■

Example 18.2 [Gold Nuggets] Your boss has a bag of gold nuggets. Each month two employees are given one nugget each for exemplary performance. By tradition, the first-ranked employee gets the heaviest nugget in the bag, and the second-ranked employee gets the lightest. This way, unless new nuggets are added to the bag, first-ranked employees get heavier nuggets than second-ranked employees get. Since new nuggets are added periodically, it is necessary to determine the heaviest and lightest nugget each month. You have a machine that can compare the weights of two nuggets and report which is lighter or whether both have the same weight. As this comparison is time-consuming, we wish to determine the heaviest and lightest nuggets, using the fewest number of comparisons.

Suppose the bag has n nuggets. We can use the strategy used in method `max` (Program 1.37) to find the heaviest nugget by making $n - 1$ comparisons. After we identify the heaviest nugget, we can find the lightest from the remaining $n - 1$ nuggets using a similar strategy and performing an additional $n - 2$ comparisons. The total number of weight comparisons is $2n - 3$. Two alternative strategies appear in Programs 2.24 and 2.25. The first strategy performs $2n - 2$ comparisons, and the second performs at most $2n - 2$ comparisons.

Let us try to formulate a solution that uses the divide-and-conquer method. When n is small, say, $n \leq 2$, one comparison is sufficient to identify the heaviest and lightest nuggets. When n is large (in this case $n > 2$), in step 1 we divide the instance into two or more smaller instances. Suppose we divide the bag of nuggets into two smaller bags A and B , each containing half the nuggets. In step 2 we determine the heaviest and lightest nuggets in A and B . Let these nuggets be H_A (heaviest in A), L_A , H_B , and L_B . Step 3 determines the heaviest overall nugget by comparing H_A and H_B and the lightest by comparing L_A and L_B . We can use the outlined divide-and-conquer scheme recursively to perform step 2.

Suppose $n = 8$. The bag is divided into two bags A and B with four nuggets each (Figure 18.1(a)). To find the heaviest and lightest nuggets in A , the four nuggets in A are divided into two groups $A1$ and $A2$. Each group contains two nuggets. We can identify the heavier nugget H_{A1} and the lighter one L_{A1} in $A1$ with one comparison (Figure 18.1(b)). With another comparison, we can identify H_{A2} and L_{A2} . Now by comparing H_{A1} and H_{A2} , we can identify H_A . A comparison between L_{A1} and L_{A2} identifies L_A . So with four comparisons we have found H_A and L_A . We need another four comparisons to determine H_B and L_B . By comparing H_A and H_B (L_A and L_B), we determine the overall heaviest (lightest) nugget. Therefore, the divide-and-conquer approach requires 10 comparisons when $n = 8$. In contrast,

Program 1.37 requires 13 comparisons, and both Programs 2.24 and 2.25 require up to 14 comparisons.

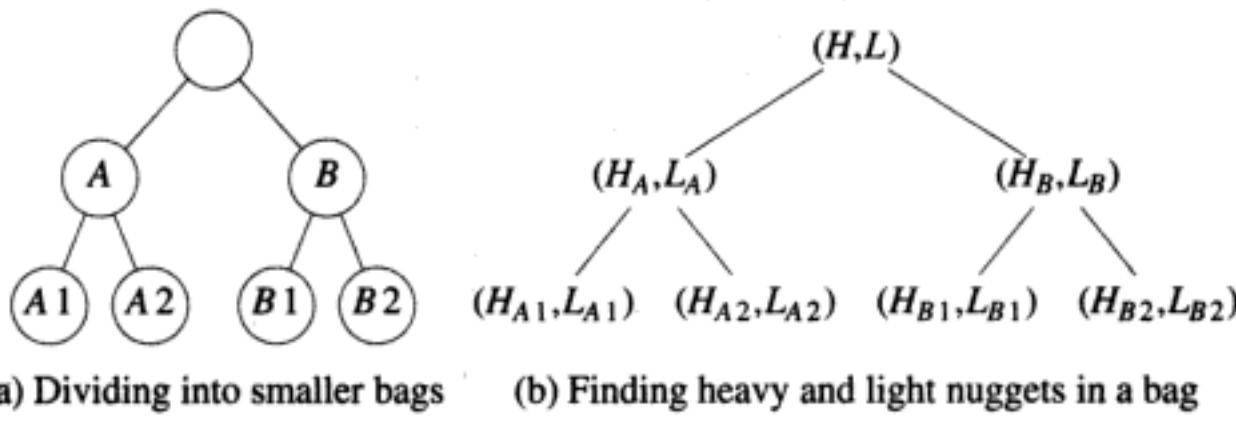


Figure 18.1 Finding the heaviest and lightest of eight nuggets

Let $c(n)$ be the number of comparisons used by the divide-and-conquer approach. For simplicity, assume that n is a power of 2. When $n = 2$, $c(n) = 1$. For larger n , $c(n) = 2c(n/2) + 2$. Using the substitution method (see Example 2.20), this recurrence can be solved to obtain $c(n) = 3n/2 - 2$ when n is a power of 2. The divide-and-conquer approach uses almost 25 percent fewer comparisons than the alternative schemes suggested in this example use. ■

Example 18.3 [Matrix Multiplication] The product of two $n \times n$ matrices A and B is a third $n \times n$ matrix C where $C(i, j)$ is given by

$$C(i, j) = \sum_{k=1}^n A(i, k) * B(k, j), \quad 1 \leq i \leq n, \quad 1 \leq j \leq n \quad (18.1)$$

If each $C(i, j)$ is computed by using this equation, then the computation of each requires n multiplications and $n - 1$ additions. The total operation count for the computation of all terms of C is therefore $n^3m + n^2(n - 1)a$ where m denotes a multiplication and a an addition or subtraction.

To formulate a divide-and-conquer algorithm to multiply the two matrices, we need to define a “small” instance, specify how small instances are multiplied, determine how a large instance may be subdivided into smaller instances, state how these smaller instances are to be multiplied, and finally describe how the solutions of these smaller instances may be combined to obtain the solution for the larger instance. To keep the discussion simple, let’s assume that n is a power of 2 (i.e., n is one of the numbers 1, 2, 4, 8, 16, …).

To begin with, let us assume that $n = 1$ is a small instance and that $n > 1$ is a large instance. We will modify this assumption later if we need to. Since a small

matrix is a 1×1 matrix, we can multiply two such matrices by multiplying together the single element in each.

Consider a large instance, that is, one with $n > 1$. We can divide such a matrix A into four $n/2 \times n/2$ matrices A_1, A_2, A_3 , and A_4 as shown in Figure 18.2(a). When n is greater than 1 and a power of 2, $n/2$ is also a power of 2. So the smaller matrices satisfy our assumption on the matrix size also. The matrices B_i and C_i , $1 \leq i \leq 4$, are defined in a similar way. The matrix product we are to perform may be represented as in Figure 18.2(b). We may use Equation 18.1 to verify that the following equations are valid:

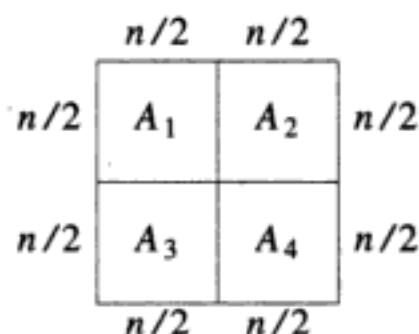
$$C_1 = A_1B_1 + A_2B_3 \quad (18.2)$$

$$C_2 = A_1B_2 + A_2B_4 \quad (18.3)$$

$$C_3 = A_3B_1 + A_4B_3 \quad (18.4)$$

$$C_4 = A_3B_2 + A_4B_4 \quad (18.5)$$

These equations allow us to compute the product of A and B by performing eight multiplications and four additions of $n/2 \times n/2$ matrices. We can use these equations to complete our divide-and-conquer algorithm. In step 2 of the algorithm, the eight multiplications involving the smaller matrices are done using the divide-and-conquer algorithm recursively. In step 3 the eight products are combined using a direct matrix addition algorithm (see Program 2.21). The complexity of the resulting algorithm is $\Theta(n^3)$, the same as the complexity of Program 2.22, which uses Equation 18.1 directly. The divide-and-conquer algorithm will actually run slower than Program 2.22 because of the overheads introduced by the instance-dividing and -recombining steps.

(a) Dividing A into four

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} * \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix}$$

(b) $A * B = C$

Figure 18.2 Dividing a matrix into smaller matrices

To get a faster algorithm, we need to be more clever about the instance-dividing and -recombining steps. A scheme, known as Strassen's method, involves the computation of seven smaller matrix products (versus eight in the preceding scheme).

The results of these seven smaller products are matrices D, E, \dots, J , which are defined as

$$\begin{aligned} D &= A_1(B_2 - B_4) \\ E &= A_4(B_3 - B_1) \\ F &= (A_3 + A_4)B_1 \\ G &= (A_1 + A_2)B_4 \\ H &= (A_3 - A_1)(B_1 + B_2) \\ I &= (A_2 - A_4)(B_3 + B_4) \\ J &= (A_1 + A_4)(B_1 + B_4) \end{aligned}$$

The matrices D through J may be computed by performing seven matrix multiplications, six matrix additions, and four matrix subtractions. The components of the answer may be computed by using another six matrix additions and two matrix subtractions as below:

$$\begin{aligned} C_1 &= E + I + J - G \\ C_2 &= D + G \\ C_3 &= E + F \\ C_4 &= D + H + J - F \end{aligned}$$

Let us try this scheme on a multiplication instance with $n = 2$. Sample A and B matrices together with their product C are given below:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Since $n > 1$, the matrix multiplication instance is divided into four smaller matrices as in Figure 18.2(a); each smaller matrix is a 1×1 matrix and has a single element. The 1×1 multiplication instances are small instances, and they are solved directly. Using the equations for D through J , we obtain the values that follow.

$$\begin{aligned} D &= 1(6 - 8) = -2 \\ E &= 4(7 - 5) = 8 \end{aligned}$$

$$\begin{aligned}
 F &= (3+4)5 = 35 \\
 G &= (1+2)8 = 24 \\
 H &= (3-1)(5+6) = 22 \\
 I &= (2-4)(7+8) = -30 \\
 J &= (1+4)(5+8) = 65
 \end{aligned}$$

From these values the components of the answer are computed as follows:

$$\begin{aligned}
 C_1 &= 8 - 30 + 65 - 24 = 19 \\
 C_2 &= -2 + 24 = 22 \\
 C_3 &= 8 + 35 = 43 \\
 C_4 &= -2 + 22 + 65 - 35 = 50
 \end{aligned}$$

For our 2×2 instance, the divide-and-conquer algorithm has done seven multiplications and 18 add/subtracts. We could have computed C by performing eight multiplications and four add/subtracts by using Equation 18.1 directly. For the divide-and-conquer scheme to be faster, the time cost of a multiplication must be more than the time cost of 14 add/subtracts.

If Strassen's instance-dividing scheme is used only when $n \geq 8$ and smaller instances are solved using Equation 18.1, then the case $n = 8$ requires seven multiplications of 4×4 matrices and 18 add/subtracts of matrices of this size. The multiplications take $64m + 48a$ operations each, and each matrix addition or subtraction takes $16a$ operations. The total operation count is $7(64m + 48a) + 18(16a) = 448m + 624$. The direct method has an operation count of $512m + 448a$. A minimum requirement for Strassen's method to be faster is that the cost of $512 - 448$ multiplications be more than that of $624 - 448$ add/subtracts. Or one multiplication should cost more than approximately 2.75 add/subtracts.

If we consider a "small" instance to be one with $n < 16$, the Strassen's decomposition scheme is used only for matrices with $n \geq 16$; smaller matrices are multiplied using Equation 18.1. The operation count for the divide-and-conquer algorithm becomes $7(512m + 448a) + 18(64a) = 3584m + 4288a$ when $n = 16$. The direct method has an operation count of $4096m + 3840a$. If the cost of a multiplication is the same as that of an add/subtract, then Strassen's method needs time for 7872 operations plus overhead time to do the problem division. The direct method needs time for 7936 operations plus time for the `for` loops and other overhead items in the program. Even though the operation count is less for Strassen's method, it is not expected to run faster because of its larger overhead.

For larger values of n , the difference between the operation counts of Strassen's method and the direct method becomes larger and larger. So for suitably large n ,

Strassen's method will be faster. Let $t(n)$ denote the time required by Strassen's divide-and-conquer method. Since large instances are recursively divided into smaller ones until each instance becomes of size k or less (k is at least eight and may be larger depending on the implementation and computer characteristics), the recurrence for t is

$$t(n) = \begin{cases} d & n \leq k \\ 7t(n/2) + cn^2 & n > k \end{cases} \quad (18.6)$$

where cn^2 represents the time to perform 18 add/subtracts of $n/2 \times n/2$ matrices as well as the time to divide the instance of size n into the smaller instances. Using the substitution method, this recurrence may be solved to obtain $t(n) = \Theta(n^{\log_2 7})$. Since $\log_2 7 \approx 2.81$, the divide-and-conquer matrix multiplication algorithm is asymptotically faster than Program 2.22. ■

Implementation Note

The divide-and-conquer methodology naturally leads to recursive algorithms. In many instances, these recursive algorithms are best implemented as recursive programs. In fact, in many cases all attempts to obtain nonrecursive programs result in the use of a stack to simulate the recursion stack. However, in some instances we can implement the divide-and-conquer algorithm as a nonrecursive program without the use of such a stack and in such a way that the resulting program is faster, by a constant factor, than the natural recursive implementation. The divide-and-conquer algorithms for both the gold nuggets problem (Example 18.2) and the merge sort method (Section 18.2.2) can be implemented without the use of recursion so as to obtain fast programs that do not directly simulate the recursion.

Example 18.4 [Gold Nuggets] The work done by the algorithm of Example 18.2 to find the lightest and heaviest of eight nuggets is described by the binary tree of Figure 18.3. The leaves of this tree denote the eight nuggets (a, b, ..., h). Each shaded node represents an instance containing all the leaves in its subtree. Therefore, root A represents the problem of finding the lightest and heaviest of all eight nuggets, while node B represents the problem of finding the lightest and heaviest of the nuggets a, b, c, and d. The algorithm begins at the root. The eight-nugget instance represented by the root is divided into 2 four-nugget instances represented by the nodes B and C. At B the four-nugget instance is divided into the two-nugget instances D and E. We solve the two-nugget instance at node D by comparing the nuggets a and b to determine which is heavier. After we solve the problems at D and E, we solve the problem at B by comparing the lighter nuggets from D and E, as well as the heavier nuggets at D and E. We repeat this process at F, G, and C and then at A.

We can classify the work of the recursive divide-and-conquer algorithm as follows:

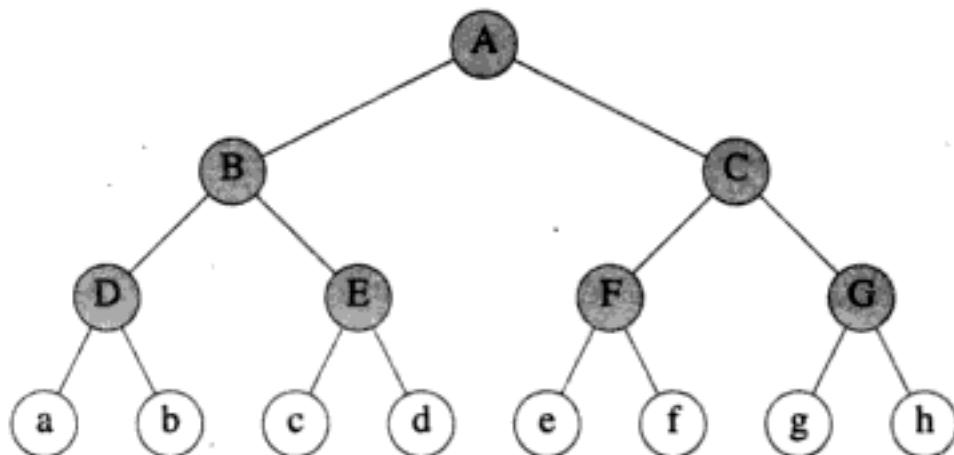


Figure 18.3 Finding the lightest and heaviest of eight nuggets

1. Divide a large instance into many smaller ones, each of size 1 or 2, during a downward root-to-leaf pass of the binary tree of Figure 18.3.
2. Compare the nuggets in each smaller size 2 instance to determine which nugget is heavier and which is lighter. This comparison is done at nodes D, E, F, and G. For size 1 instances the single nugget is both the smaller and lighter nugget.
3. Compare the lighter nuggets to determine which is lightest. Compare the heavier nuggets to determine which is heaviest. These comparisons are done at nodes A through C.

This classification of the work leads to the nonrecursive code of Program 18.1 to find the locations of the minimum and maximum of the n weights $a[0:n-1]$. These locations are returned in the variables `indexOfMin` and `indexOfMax`.

The cases $n < 1$ and $n = 1$ are handled first. If $n > 1$ and odd, the first weight $a[0]$ becomes the candidate for minimum and maximum, and we are left with an even number of weights $a[1:n-1]$ to account for in the `for` loop. When n is even, the first two weights are compared outside the `for` loop and `indexOfMin` and `indexOfMax` are set to the location of the smaller and larger weight, respectively. Again, we are left with an even number of weights $a[2:n-1]$ to account for in the `for` loop.

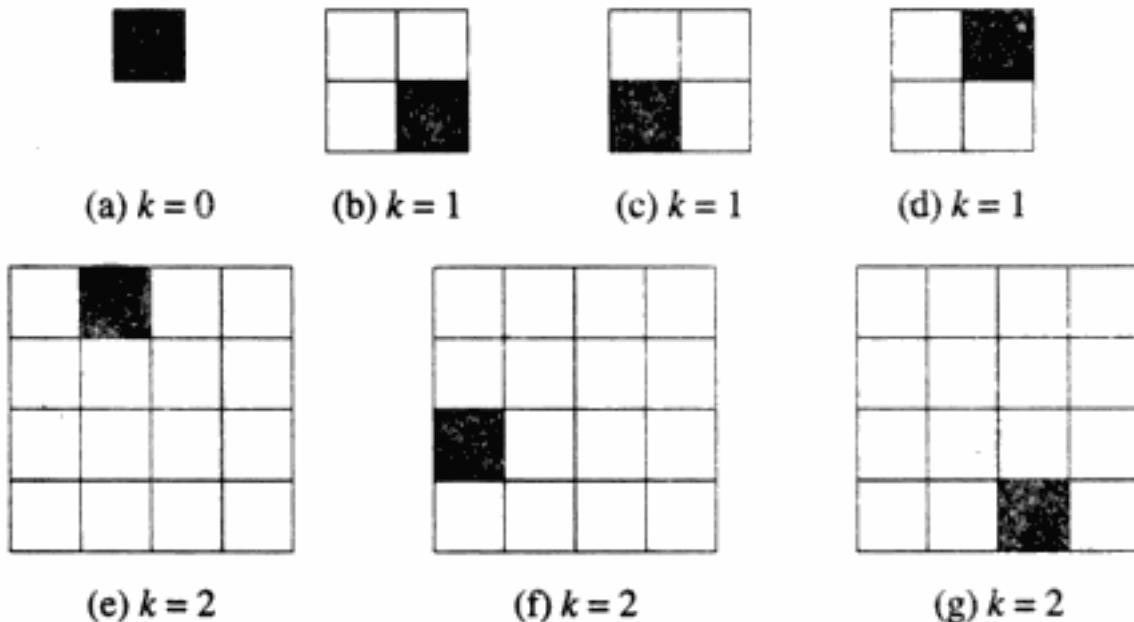
In the `for` loop the outer `if` finds the larger and smaller of the pair $(a[i], a[i+1])$ being compared. This work corresponds to the category 2 work in our classification of the work of our divide-and-conquer algorithm. The embedded `ifs` find the smallest of the smaller weights and the largest of the larger weights. This work is the category 3 work.

The `for` loop compares the smaller of each pair to the current minimum weight

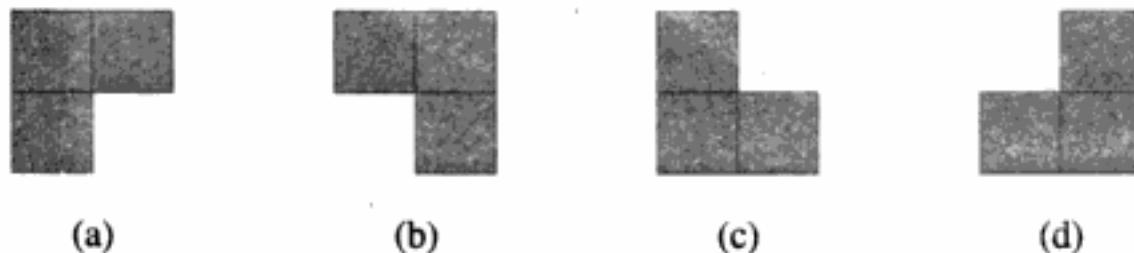
Hidden page

Hidden page

Hidden page

**Figure 18.4** Defective chessboards

squares and these squares are covered using a triomino in one of the orientations of Figure 18.5.

**Figure 18.5** Triominoes with different orientations

Solution Strategy

The divide-and-conquer method leads to an elegant solution to the defective-chessboard problem. The method suggests reducing the problem of tiling a $2^k \times 2^k$ defective chessboard to that of tiling smaller defective chessboards. A natural partitioning of a $2^k \times 2^k$ chessboard would be into four $2^{k-1} \times 2^{k-1}$ chessboards as in Figure 18.6(a). Notice that when such a partitioning is done, only one of the four smaller boards has a defect (as the original $2^k \times 2^k$ board had exactly one defective square). Tiling one of the four smaller boards corresponds to tiling a defective

Hidden page

Hidden page

$$t(k) = \begin{cases} d & k = 0 \\ 4t(k-1) + c & k > 0 \end{cases} \quad (18.7)$$

We may solve this recurrence using the substitution method (see Example 2.20) to obtain $t(k) = \Theta(4^k) = \Theta(\text{number of tiles needed})$. Since we must spend at least $\Theta(1)$ time placing each tile, we cannot obtain an asymptotically faster algorithm than divide and conquer.

18.2.2 Merge Sort

The Sort Method

We can apply the divide-and-conquer method to the sorting problem. In this problem we must sort n elements into nondecreasing order. The divide-and-conquer method suggests sorting algorithms with the following general structure: If n is 1, terminate; otherwise, partition the collection of elements into two or more subcollections; sort each; combine the sorted subcollections into a single sorted collection.

Suppose we limit ourselves to partitioning the n elements into two subcollections. Now we need to decide how to perform this partitioning. One possibility is to put the first $n - 1$ elements into the first subcollection (say, A) and the last element into the second subcollection (say, B). A is sorted using this partitioning scheme recursively. Since B has only one element, it is already sorted. Following the sort of A , we need to combine A and B , using the method `insert` of Program 2.10. Comparing the resulting sort algorithm with `insertionSort` (Program 2.15), we see that we have really discovered the recursive version of insertion sort. The complexity of this sort algorithm is $O(n^2)$.

Another possibility for the two-way partitioning of n elements is to put the element with largest key in B and the remaining elements in A . Then A is sorted recursively. To combine the sorted A and B , we need merely append B to the sorted A . If we find the element with largest key using the function `Max` of Program 1.37, the resulting sort algorithm is a recursive formulation of `selectionSort` (Program 2.7). If we use a bubbling process (Program 2.8) to locate and move the element with largest key to the right-most position, the resulting algorithm is a recursive version of `bubbleSort` (Program 2.9). In either case the sort algorithm has complexity $\Theta(n^2)$. This complexity can be made $O(n^2)$ by terminating the recursive partitioning of A as soon as A is known to be in sorted order (see Examples 2.16 and 2.17).

The partitioning schemes used to arrive at the three preceding sort algorithms partitioned the n elements into two very unbalanced collections A and B . A has $n - 1$ elements, while B has only 1 element. Let us see what happens when this partitioning is done in a more balanced way, that is, when A gets a fraction n/k of the elements and B gets the rest. Now both A and B are to be sorted by recursive

Hidden page

where c and d are constants. From Equation 18.8 it follows that $t(n)$ is minimum when $t(n/k) + t(n - n/k)$ is minimum.

Theorem 18.1 *Let $f(x)$ satisfy*

$$f(y+d) - f(y) \geq f(z+d) - f(z) \quad (18.9)$$

for all $y \geq z$ and all positive d . For every real number w , $f(w/k) + f(w - w/k)$ is minimum when $k = 2$.

Proof. Substituting $z = y - d$ into Equation 18.9, we get

$$f(y+d) - f(y) \geq f(y) - f(y-d)$$

or

$$2f(y) \leq f(y+d) + f(y-d) \quad (18.10)$$

Substituting $d = w/2 - w/k$ when $k \geq 2$, $d = w/k - w/2$ when $k < 2$, and $y = w/2$ into Equation 18.10, we get

$$2f(w/2) \leq f(w/k) + f(w - w/k) \quad \blacksquare$$

Every sort algorithm has complexity $\Omega(n \log n)$ (see Section 18.4.2). Therefore, $f(n) = t(n)$ satisfies Equation 18.9, and the complexity of Figure 18.7 is minimum when $k = 2$, that is, when the two smaller instances are of approximately the same size. (Actually, a complexity of $\Omega(n)$ is sufficient to satisfy Equation 18.9.) *Divide-and-conquer algorithms usually have optimal performance when the smaller instances created are of approximately the same size.*

Setting $k = 2$ in the recurrence for $t(n)$, we get the following recurrence:

$$t(n) = \begin{cases} d & n \leq 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + cn & n > 1 \end{cases}$$

The presence of the floor and ceiling operators makes this recurrence difficult to solve. We can overcome this difficulty by solving the recurrence only for values of n that are a power of 2. In this case the recurrence takes the simpler form

$$t(n) = \begin{cases} d & n \leq 1 \\ 2t(n/2) + cn & n > 1 \end{cases}$$

Hidden page

Hidden page

Hidden page

```

template <class T>
void merge(T c[], T d[], int startOfFirst, int endOfFirst,
           int endOfSecond)
{ // Merge two adjacent segments from c to d.
    int first = startOfFirst,           // cursor for first segment
        second = endOfFirst + 1,         // cursor for second
        result = startOfFirst;          // cursor for result

    // merge until one segment is done
    while ((first <= endOfFirst) && (second <= endOfSecond))
        if (c[first] <= c[second])
            d[result++] = c[first++];
        else
            d[result++] = c[second++];

    // take care of leftovers
    if (first > endOfFirst)
        for (int q = second; q <= endOfSecond; q++)
            d[result++] = c[q];
    else
        for (int q = first; q <= endOfFirst; q++)
            d[result++] = c[q];
}

```

Program 18.5 Merge two adjacent segments from c to d

would begin with segments of size 1 and make three merge passes.

The best case for natural merge sort is when the input element list is already sorted. Natural merge sort would identify exactly one sorted segment and make no merge passes, while Program 18.3 would make $\lceil \log_2 n \rceil$ merge passes. So natural merge sort would complete in $\Theta(n)$ time, while Program 18.3 would take $\Theta(n \log n)$ time.

The worst case for natural merge sort is when the input elements are in decreasing order of their keys. With this input, n initial segments are identified; both merge sort and natural merge sort make the same number of passes, but natural merge sort has a higher overhead in keeping track of segment boundaries. The worst-case performance of natural merge sort is worse than that of straight merge sort.

On average, we expect a list of n elements to have $n/2$ segments because, on average, the i th key is larger than the $i + 1$ st key with probability 0.5. Starting with half as many segments, natural merge sort is expected to make one fewer merge pass than is made by straight merge sort. However, the time saved is offset

Hidden page

C++ Implementation

The `quickSort` function of Program 18.6 moves the largest element of the element array `a` to the right-most position of the array; and invokes the recursive function `quickSort` of Program 18.7, which does the actual sorting. We move the largest element to the right-most position because our element partitioning scheme (Program 18.7) requires that each segment either has its largest element on the right or is followed by an element that is \geq all elements in the segment; in case this condition is not satisfied, the first `do` loop of Program 18.7 results in a left cursor value larger than $n - 1$ when the pivot is the largest element, for example.

```
template <class T>
void quickSort(T a[], int n)
{// Sort a[0 : n - 1] using the quick sort method.
    if (n <= 1) return;
    // move largest element to right end
    int max = indexOfMax(a,n);
    swap(a[n - 1], a[max]);
    quickSort(a, 0, n - 2);
}
```

Program 18.6 Driver for recursive quick sort function

The partitioning of the element list into *left*, *middle*, and *right* is done in place (Program 18.7). In this implementation the pivot is always the element at the left end of the segment that is to be sorted. Other choices that result in improved performance are possible. One such choice is discussed later in this section.

Program 18.7 remains correct when we change the `<` and `>` in the conditionals of the `do-while` statements to `\leq` and `\geq` , respectively (provided the right-most element of a segment is larger than the pivot). Experiments suggest that the average performance of quick sort is better when it is coded as in Program 18.6. All attempts to eliminate the recursion from this procedure result in the introduction of a stack. The last recursive call, however, can be eliminated without the introduction of a stack. We leave the elimination of this recursive call as Exercise 21.

Complexity Analysis

Program 18.6 requires $O(n)$ recursion stack space. The space requirements can be reduced to $O(\log n)$ by simulating the recursion using a stack. In this simulation the smaller of the two segments *left* and *right* is sorted first. The boundaries of the other segment are put on the stack.

The worst-case computing time for quick sort is $\Theta(n^2)$, and it is achieved, for instance, when *left* is always empty. However, if we are lucky and *left* and *right*

Hidden page

number of elements to be sorted.

Proof Let $t(n)$ denote the average time needed to sort an n -element array. When $n \leq 1$, $t(n) \leq d$ for some constant d . Suppose that $n > 1$. Let s be the size of the left segment following the partitioning of the elements. Because the pivot element is in the middle segment, the size of the right segment is $n - s - 1$. The average times to sort the left and right segments are $t(s)$ and $t(n - s - 1)$, respectively. The time needed to partition the elements is bounded by cn where c is a constant. Since s can have any of the n values 0 through $n - 1$ with equal probability, we obtain the following recurrence:

$$t(n) \leq cn + \frac{1}{n} \sum_{s=0}^{n-1} [t(s) + t(n - s - 1)]$$

We can simplify this recurrence as follows:

$$t(n) \leq cn + \frac{2}{n} \sum_{s=0}^{n-1} t(s) \leq cn + \frac{4d}{n} + \frac{2}{n} \sum_{s=2}^{n-1} t(s) \quad (18.11)$$

Now using induction on n we show that $t(n) \leq kn \log_e n$ for $n > 1$ and $k = 2(c + d)$. Here $e \approx 2.718$ is the base of natural logarithms. The induction base covers the case $n = 2$. From Equation 18.11 we obtain $t(2) \leq 2c + 2d \leq kn \log_e 2$. For the induction hypothesis we assume $t(n) \leq kn \log_e n$ for $2 \leq n < m$ where m is an arbitrary integer that is greater than 2. In the induction step we need to prove $t(m) \leq km \log_e m$. From Equation 18.11 and the induction hypothesis, we obtain

$$t(m) \leq cm + \frac{4d}{m} + \frac{2}{m} \sum_{s=2}^{m-1} t(s) \leq cm + \frac{4d}{m} + \frac{2k}{m} \sum_{s=2}^{m-1} s \log_e s \quad (18.12)$$

To proceed further we use the following facts:

- $s \log_e s$ is an increasing function of s .
- $\int_2^m s \log_e s ds < \frac{m^2 \log_e m}{2} - \frac{m^2}{4}$.

Using these facts and Equation 18.12, we obtain

$$\begin{aligned} t(m) &< cm + \frac{4d}{m} + \frac{2k}{m} \int_2^m s \log_e s ds \\ &< cm + \frac{4d}{m} + \frac{2k}{m} \left[\frac{m^2 \log_e m}{2} - \frac{m^2}{4} \right] \\ &= cm + \frac{4d}{m} + km \log_e m - \frac{km}{2} \\ &< km \log_e m \end{aligned}$$

So the average complexity of `quickSort` is $O(n \log n)$. In Section 18.4.2 we show that the complexity of every comparison sort method (including `quickSort`) is $\Omega(n \log n)$. Therefore, the average complexity of `quickSort` is $\Theta(n \log n)$. ■

The table of Figure 18.11 compares the average and worst-case complexities of the sort methods developed in this book.

Method	Worst	Average
bubble sort	n^2	n^2
count sort	n^2	n^2
insertion sort	n^2	n^2
selection sort	n^2	n^2
heap sort	$n \log n$	$n \log n$
merge sort	$n \log n$	$n \log n$
quick sort	n^2	$n \log n$

Figure 18.11 Comparison of sort methods

Median-of-Three Quick Sort

Our implementation of quick sort exhibits its worst-case performance when presented with a sorted list. It is distressing to have a sort program that takes more time on a sorted list than on an unsorted one. We can remedy this problem and, at the same time, improve the average performance of quick sort by selecting the pivot element using the **median-of-three rule**.

In a median-of-three quick sort, the pivot is chosen to be the median of the three elements $\{a[\text{leftEnd}], a[(\text{leftEnd}+\text{rightEnd})/2], a[\text{rightEnd}]\}$. For example, if these elements have keys $\{5, 9, 7\}$, then $a[\text{rightEnd}]$ is used as the value of `pivot`. To implement the median-of-three rule, it is easiest to swap the element in the median position with that at $a[\text{leftEnd}]$ and then proceed as in Program 18.6. If $a[\text{rightEnd}]$ is the median element, then we swap $a[\text{leftEnd}]$ and $a[\text{rightEnd}]$ just before `pivot` is set to $a[\text{leftEnd}]$ in Program 18.7 and proceed as in the remainder of the code.

When the median-of-three rule is used, quick sort takes $O(n \log n)$ time when the input list is in sorted order. Moreover, the case when one of the two partitions is empty is eliminated (except when we have duplicates). In other words, the median-of-three rule ensures a better balance between the two partitions. Is this improvement in balance enough to pay for the added cost of computing the pivot? Only an experiment can answer this question.

Performance Measurement

The observed average times for `quickSort` appear in Figure 18.12. These times are for Program 18.6 with the pivot being the first element of the segment. This figure includes the average times for merge, heap, and insertion sort. For each n at least 100 randomly generated integer instances were run. These random instances were constructed by making repeated calls to the C++ function `rand`. If the time taken to sort these instances was less than 1 second (see Section 4.4), then additional random instances were sorted until the total time taken was at least this much. The times reported in Figure 18.12 include the time taken to set up the random data. For each n the time taken to set up the data and the time for the remaining overheads included in the reported numbers is the same for all sort methods. As a result, the data of Figure 18.12 is useful for comparative purposes. The data of this figure is plotted in Figure 18.13.

n	Insert	Heap	Merge	Quick
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.059	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

Times are in milliseconds

Figure 18.12 Average times for sort methods

As Figure 18.13 shows, quick sort outperforms the other sort methods for suitably large n . We see that the break-even point between insertion and quick sort is between 50 and 100. The exact break-even point can be found experimentally by obtaining run-time data for n between 50 and 100. Let the exact break-even point be `nBreak`. For average performance, insertion sort is the best sort method (of those tested) to use when $n \leq nBreak$, and quick sort is the best when $n > nBreak$. We can improve on the performance of quick sort for $n > nBreak$ by combining insertion and quick sort into a single sort function by replacing the following statement in Program 18.6

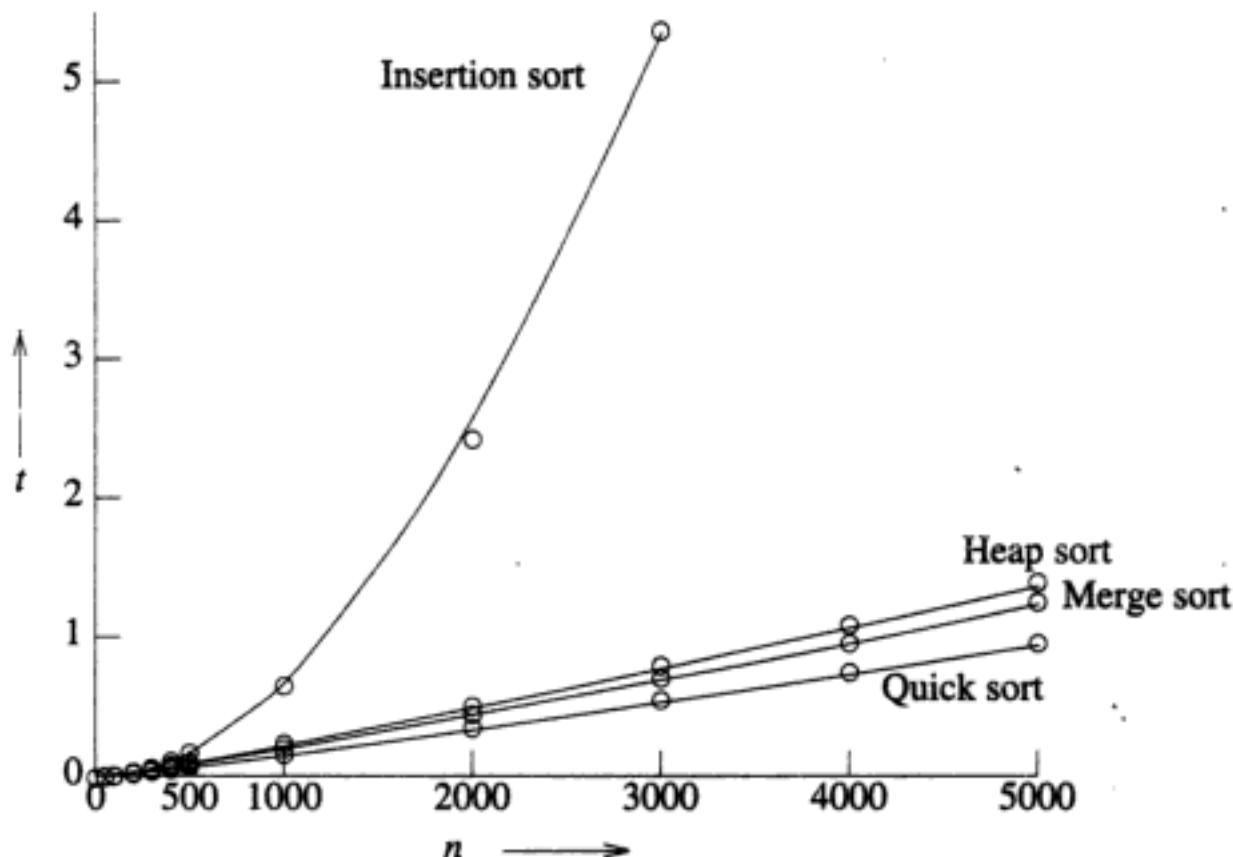


Figure 18.13 Plot of average times (milliseconds)

```
if (leftEnd >= rightEnd) return;
```

with the code

```
if (rightEnd - leftEnd < nBreak)
{
    insertionSort(a, leftEnd, rightEnd);
    return;
}
```

Here `insertionSort(a, leftEnd, rightEnd)` is a function that sorts `a[leftEnd:rightEnd]`, using the insertion sort method. The performance measurement of the modified quick sort code is left as Exercise 28. Further improvement in performance may be possible by replacing `nBreak` with a smaller value (see Exercise 28).

For worst-case behavior most implementations will show merge sort to be best for $n > c$ where c is some constant. For $n \leq c$ insertion sort has the best worst-case behavior. The performance of merge sort can be improved by combining insertion sort and merge sort (see Exercise 29).

Hidden page

position $a[k-1]$. We can obtain better average performance by using quick sort (see Figure 18.12), even though this method has an inferior asymptotic complexity of $O(n^2)$.

We can adapt the code of Program 18.6 to the selection problem so as to obtain an even faster solution. If the pivot $a[leftEnd]$ is to be placed in $a[j]$ following the execution of the two `while` loops, then $a[leftEnd]$ is known to be the $j-leftEnd+1$ th element of $a[leftEnd:rightEnd]$. If we are looking for the k th element in $a[leftEnd:rightEnd]$ and $j-leftEnd+1$ equals k , then the answer is $a[leftEnd]$; if $j-leftEnd+1 < k$, then the element we are looking for is the $k-j+leftEnd-1$ th element of $right$; otherwise, it is the k th element of $left$. Therefore, we need to make either zero or one recursive call. The code for the new selection program appears in Programs 18.8 and 18.9. A `for` or `while` loop can replace the recursive calls made by `select` (see Exercise 35).

```
template <class T>
T select(T a[], int n, int k)
{// Return k'th smallest element in a[0 : n - 1].
if (k < 1 || k > n)
    throw illegalParameterValue("k must be between 1 and n");

// move largest element to right end
int max = indexOfMax(a, n);
swap(a[n-1], a[max]);
return select(a, 0, n - 1, k);
}
```

Program 18.8 Preprocessor to find the k th element

Complexity Analysis

The worst-case complexity of Program 18.8 is $\Theta(n^2)$. This worst case is achieved, for example, when $left$ is always empty and the k th element is in $right$. However, if $left$ and $right$ are always of the same size or differ in size by at most 1, then we get the following recurrence for the time needed by Program 18.8:

$$t(n) \leq \begin{cases} d & n \leq 1 \\ t(\lfloor n/2 \rfloor) + cn & n > 1 \end{cases} \quad (18.13)$$

If we assume that n is a power of 2, the floor operator may be dropped and the recurrence solved, using the substitution method, to obtain $t(n) = \Theta(n)$. By selecting the partitioning element more carefully, the worst-case time also becomes $\Theta(n)$. The more careful way to select the partitioning element is to use the **median-of-medians** rule in which the n elements of a are divided into $\lfloor n/r \rfloor$ groups for some

```

template <class T>
T select(T a[], int leftEnd, int rightEnd, int k)
{// Return k'th element in a[leftEnd:rightEnd].
    if (leftEnd >= rightEnd)
        return a[leftEnd];
    int leftCursor = leftEnd,           // left-to-right cursor
        rightCursor = rightEnd + 1;   // right-to-left cursor
    T pivot = a[leftEnd];

    // swap elements >= pivot on left side
    // with elements <= pivot on right side
    while (true)
    {
        do
        {// find >= element on left side
            leftCursor++;
        } while (a[leftCursor] < pivot);

        do
        {// find <= element on right side
            rightCursor--;
        } while (a[rightCursor] > pivot);

        if (leftCursor >= rightCursor) break; // swap pair not found
        swap(a[leftCursor], a[rightCursor]);
    }

    if (rightCursor - leftEnd + 1 == k)
        return pivot;

    // place pivot
    a[leftEnd] = a[rightCursor];
    a[rightCursor] = pivot;

    // recursive call on one segment
    if (rightCursor - leftEnd + 1 < k)
        return select(a, rightCursor + 1, rightEnd,
                      k - rightCursor + leftEnd - 1);
    else return select(a, leftEnd, rightCursor - 1, k);
}

```

Program 18.9 Recursive method to find the k th element

integer constant r . Each of these groups contains exactly r elements. The remaining $n \bmod r$ elements are not used in the selection of the pivot. Next we find the median of each group by sorting the r elements in each group and then selecting the one in the middle position (i.e., in position $\lceil r/2 \rceil$). The median of these $\lfloor n/r \rfloor$ medians is computed, using the selection algorithm recursively, and is used as the partitioning element.

Example 18.6 [Median of Medians] Consider the case $r = 5$, $n = 27$, and $a = [2, 6, 8, 1, 4, 9, 20, 6, 22, 11, 9, 8, 4, 3, 7, 8, 16, 11, 10, 8, 2, 14, 15, 1, 12, 5, 4]$. These 27 elements may be divided into the five groups $[2, 6, 8, 1, 4]$, $[9, 20, 6, 22, 11]$, $[9, 8, 4, 3, 7]$, $[8, 16, 11, 10, 8]$, and $[2, 14, 15, 1, 12]$. The remaining elements, 5 and 4, are not used when selecting the pivot. The medians of these five groups are 4, 11, 7, 10, and 12, respectively. The median of the elements $[4, 11, 7, 10, 12]$ is 10. This median is used as the partitioning element. With this choice of the pivot, we get $left = [2, 6, 8, 1, 4, 9, 6, 9, 8, 4, 3, 7, 8, 8, 2, 1, 5, 4]$, $middle = [10]$, and $right = [20, 22, 11, 16, 11, 14, 15, 12]$. If we are to find the k th element for $k < 19$, only $left$ needs to be examined; if $k = 19$, the element is the pivot; and if $k > 19$, the eight elements of $right$ need to be examined. In this last case we need to find the $k - 19$ th element of $right$. ■

Theorem 18.3 *When the partitioning element is chosen using the median-of-medians rule, the following statements are true:*

- (a) *If $r = 9$, then $\max\{|left|, |right|\} \leq 7n/8$ for $n \geq 90$.*
- (b) *If $r = 5$ and all elements of a are distinct, then $\max\{|left|, |right|\} \leq 3n/4$ for $n \geq 24$.*

Proof Exercise 33 asks you to prove this theorem. ■

From Theorem 18.3 and Program 18.8, it follows that if the median-of-medians rule with $r = 9$ is used, the time $t(n)$ needed to select the k th element is given by the following recurrence:

$$t(n) = \begin{cases} cn \log n & n < 90 \\ t(\lfloor n/9 \rfloor) + t(\lfloor 7n/8 \rfloor) + cn & n \geq 90 \end{cases} \quad (18.14)$$

where c is a constant. This recurrence assumes that an $n \log n$ method is used when $n < 90$ and that larger instances are solved using divide and conquer with the median-of-medians rule. Using induction, you can show (Exercise 34) that $t(n) \leq 72cn$ for $n \geq 1$. When the elements are distinct, we may use $r = 5$ to get linear-time performance.

18.2.5 Closest Pair of Points

Problem Description

In this problem you are given n points (x_i, y_i) , $1 \leq i \leq n$, and are to find two that are closest. The distance between two points i and j is given by the following formula:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Example 18.7 Suppose that n equal-size holes are to be drilled into a sheet of metal. If any two holes are too close, metal failure may occur during the drilling process. By determining the minimum distance between any two holes, we can assess the probability of such a failure. This minimum distance corresponds to the distance between a closest pair of points. ■

Solution Strategy

We can solve the closest-pair-of-points problem in $O(n^2)$ time by examining all $n(n - 1)/2$ pairs of points, computing the distance between the points in each pair, and determining the pair for which this distance is minimum. We will call this method the *direct approach*. The divide-and-conquer method suggests the high-level algorithm of Figure 18.14.

```
if (n is small)
{
    Find the closest pair using the direct approach.
    return;
}
```

// n is large

Divide the point set into two roughly equal parts A and B .

Determine the closest pairs of points in A and B .

Determine the closest point pair such that one point is in A and the other in B .

From the three closest pairs computed, select the one with least distance.

Figure 18.14 Finding a closest pair of points

This algorithm uses the direct approach to solve small instances and solves large instances by dividing them into two smaller instances. One instance (say, A) will be of size $\lceil n/2 \rceil$, and the other (say, B) of size $\lfloor n/2 \rfloor$. The closest pair of points

in the original instance falls into one of the three categories: (1) both points are in A (i.e., it is a closest pair of A); (2) both points are in B ; and (3) one point is in A , and the other in B . Suppose we determine the closest pair in each of these categories. The pair with the least distance is the overall closest pair. The closest pair in category 1 can be determined by using the closest-pair algorithm recursively on the smaller point set A . The closest pair in B can be similarly determined.

To determine the closest pair in category 3, we need a different method. The method depends on how the points are divided into A and B . A reasonable way to do this division is to cut the plane by a vertical line that goes through the median x_i value. All points to the left of this line are in A ; all to the right are in B ; and those on the line are distributed between A and B so as to meet the size requirements on A and B .

Example 18.8 Consider the 14 points a through n of Figure 18.15(a). These points are plotted in Figure 18.15(b). The median $x_i = 1$, and the vertical line $x = 1$ is shown as a broken line in Figure 18.15(b). The points to the left of this line (i.e., points b, c, h , and n) are in A , and those to the right of the line (i.e., points a, e, f, j, k , and l) are in B . Of the points d, g , and m that are on the line, two are added to A and one to B so that A and B have seven points each. Suppose that d and m are assigned to A , and g is assigned to B . ■

Let δ be the smaller of the distances between the points in the closest pairs of A and B . For a pair in category 3 to be closer than δ , each point of the pair must be less than distance δ from the dividing line. Therefore, we can eliminate from consideration all points that are a distance $\geq \delta$ from this line. The broken line of Figure 18.16 is the dividing line. The shaded box has width 2δ and is centered at the dividing line. Points on or outside the boundary of this box are eliminated. Only the points inside the shaded region need be retained when determining whether there is a category 3 pair with distance less than δ .

Let R_A and R_B , respectively, denote the points of A and B that remain. If there is a point pair (p, q) such that $p \in A$, $q \in B$, and p and q are less than δ apart, then $p \in R_A$ and $q \in R_B$. We can find this point pair by considering the points in R_A one at a time. Suppose that we are considering point p of R_A and that p 's y -coordinate is $p.y$. We need to look only at points q in R_B with y -coordinate $q.y$ such that $p.y - \delta < q.y < p.y + \delta$ and see whether any point is less than distance δ from p . The region of R_B that contains these points q appears in Figure 18.17(a). Only the points of R_B within the shaded $\delta \times 2\delta$ box need be paired with p to see whether p is part of a category 3 pair with distance less than δ . This $\delta \times 2\delta$ region is p 's comparing region.

Example 18.9 Consider the 14 points of Example 18.8. The closest point pair in A (see Example 18.8) is (b, h) with a distance of approximately 0.316. The closest point pair in B is (f, j) with a distance of 0.3. Therefore, $\delta = 0.3$. When determining whether there is a category 3 pair with distance less than δ , all points

Hidden page

Hidden page

data members *x* and *y* to store the *x*- and *y*-coordinates of a point, and both *x* and *y* are of type *double*. The struct *point1* derives from *point* and adds the additional data member *id*, which is of type *int*. The struct *point1* defines a type conversion to *double* that returns the value of *x*. This type conversion allows us to use merge sort (Program 18.3) to sort the points into ascending order of *x*-coordinate. The third class for points is *point2*. This class also derives from *point*, and adds the integer data member *p* whose significance is described below. The struct *point2* defines a type conversion to *double* that returns the *y*-coordinate of the point. This type conversion allows us to use merge sort (Program 18.3) to sort the points into ascending order of *y*-coordinate. The fourth and last point struct is *pointPair*. This struct has the data members *a* (first point in the pair), *b* (second point in the pair), and *dist* (distance between the points *a* and *b*). *a* and *b* are of type *point1*, and *dist* is of type *double*.

The input points may be represented in an array *x* of type *point1*. Suppose that the points in *x* have been sorted by their *x*-coordinate. If at any stage of the division process the points under consideration are *x[l : r]*, then we may obtain *A* and *B* by first computing $m = (l + r)/2$. The points *X[l : m]* are in *A*, and the remaining points are in *B*.

After we compute the closest pairs in *A* and *B*, we need to compute R_A and R_B and then determine whether there is a closer pair with one point in R_A and the other in R_B . The test of Figure 18.17 may be implemented in a simple way if the points are sorted by their *y*-coordinate. A list of the points sorted by *y*-coordinate is maintained in another array using the struct *point2*. Notice that for this struct, the type conversion to *double* has been implemented so as to facilitate sorting by *y*-coordinate. The field *p* is used to index back to the same point in the array *x*.

C++ Implementation

With the necessary data structures determined, let us examine the resulting code. First we define a function *dist* (Program 18.10) that computes the distance between two points *a* and *b*. Notice that Program 18.10 may also be used to compute the distance between points of type *point1* and *point2* because the class *point* is the base class of *point1* and *point2*.

```
double dist(const point& u, const point& v)
{// return distance between points u and v.
    double dx = u.x - v.x;
    double dy = u.y - v.y;
    return sqrt(dx * dx + dy * dy);
}
```

Program 18.10 Computing the distance between two points

The function *closestPair* of Program 18.11 throws an exception if the number of points is fewer than 2. When the number of points exceeds 1, the function

Hidden page

Hidden page

Hidden page

that have y -coordinate $\leq p.y$. These two parts may be implemented by pairing each point $z[i]$, $1 \leq i < k$ (regardless of whether it is in R_A or R_B) with a point $z[j]$, $i < j$, for which $z[j].y - z[i].y < \delta$. For each $z[i]$ the points that get examined lie inside the $2\delta \times \delta$ region shown in Figure 18.18. Since the points in each $\delta \times \delta$ subregion are at least δ apart, the number in each subregion cannot exceed four. So the number of points $z[j]$ that each $z[i]$ is paired with is at most seven.



Figure 18.18 Region of points paired with $z[i]$

Complexity Analysis

Let $t(n)$ denote the time taken by the recursive function `closestPair` on a set of n points. When $n < 4$, $t(n)$ equals some constant d . When $n \geq 4$, it takes $\Theta(n)$ time to divide the instance into two parts, reconstruct y after the two recursive calls, eliminate points that are too far from the dividing line, and search for a better category 3 pair. The recursive calls take $t(\lceil n/2 \rceil)$ and $t(\lfloor n/2 \rfloor)$ time, respectively. Therefore, we obtain the recurrence

$$t(n) = \begin{cases} d & n < 4 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & n \geq 4 \end{cases}$$

which is the same as the recurrence for merge sort. Its solution is $t(n) = \Theta(n \log n)$. The additional work done by the driver function `closestPair` of Program 18.11 consists of sorting x , creating y and z , and sorting y . The total time for this additional work is also $\Theta(n \log n)$. The overall time complexity of the divide-and-conquer closest-pair code is $\Theta(n \log n)$ under the assumption that no exception is thrown.

EXERCISES

8. Write a complete program for the defective-chessboard problem. Include modules to welcome the user to the program; input the chessboard size and location

of the defect; and output the tiled chessboard. The output is to be provided on a color monitor using colored tiles. No two tiles that share a common boundary should be colored the same. Since the chessboard is a planar figure, it is possible to color the tiles in this way using at most four colors. However, for this exercise, it is sufficient to devise a greedy coloring heuristic that attempts to use as few colors as possible.

9. Solve the recurrence of Equation 18.7 using the substitution method.
10. Does $f(x) = \sqrt{x}$ satisfy Equation 18.9?
11. Show that $f(x) = x^a \log^b x$ satisfies Equation 18.9 for all $a \geq 1$, all integer $b \geq 0$, and all $x \geq 1$.
12. Start with the array [11, 2, 8, 3, 6, 15, 12, 0, 7, 4, 1, 13, 5, 9, 14, 10] and draw a figure similar to Figure 18.9 that shows the steps involved in a merge sort.
13. Do Exercise 12 using the array [11, 3, 8, 7, 5, 10, 0, 9, 4, 2, 6, 1].
14. Write a merge sort code that works on chains of elements. The output should be a sorted chain. Make your sort method a member of the class `chain` or of a class that extends `chain` (Section 6.1).
15. Start with the array [2, 3, 6, 8, 11, 15, 0, 7, 12, 1, 4, 13, 5, 9, 10, 14] and draw a figure similar to Figure 18.9 that shows the steps involved in a natural merge sort.
16. Do Exercise 15 using the array [11, 3, 8, 5, 7, 10, 0, 9, 2, 4, 6, 1].
17. Write the function `naturalMergeSort` that implements a natural merge sort. The input and output configurations are the same as for Program 18.3.
18. Write a natural merge sort code that sorts a chain of elements. Your function should be a member of the class `chain` or of a class that extends `chain` (Section 6.1).
19. Start with the segment [5, 3, 8, 4, 7, 1, 0, 9, 2, 10, 6, 11] and draw a figure to show the status of this segment following each swap that is done in the `while` loop of Program 18.7. Also show the segment after the pivot is properly placed. Use 5 as the pivot.
20. Do Exercise 19 using the array [7, 3, 6, 8, 11, 14, 0, 2, 12, 1, 4, 13, 5, 9, 10, 15]. Use 7 as the pivot.
21. Replace the last recursive call to `quickSort` in Program 18.7 with a `while` loop. Compare the average run time of the resulting sort method with that of Program 18.7.

22. Rewrite the quick sort code (Programs 18.6 and 18.7) using a stack to simulate the recursion. The new code should stack the boundaries of only the larger of the segments *left* and *right*.
- Show that the stack space needed is $O(\log n)$.
 - Compare the average run time of your nonrecursive code with that of the recursive code given in the text.
23. Show that the worst-case time complexity of `quickSort` is $\Theta(n^2)$.
24. Suppose that the partitioning into *left*, *middle*, and *right* is always such that *left* and *right* have the same size when n (n is the number of elements in the segment being sorted) is odd and *left* has one element more than *right* when n is even. Show that under this assumption, the time complexity of Program 18.6 is $O(n \log n)$.
25. Show that $\int s \log_e s ds = \frac{s^2 \log_e s}{2} - \frac{s^2}{4}$. Use this result to show that $\int_2^m s \log_e s ds < \frac{m^2 \log_e m}{2} - \frac{m^2}{4}$.
26. Compare the worst-case and average times of Program 18.7 when the median-of-three rule is used to the times when it is not used. Do this comparison experimentally; use suitable test data and $n = 10, 20, \dots, 100, 200, 300, 400, 500, 1000$.
27. Do Exercise 26 using a random number generator rather than the median-of-three rule to select the pivot element.
28. At the end of the quick sort section, we suggested combining two sort methods: quick sort and insertion sort. The combined algorithm is essentially a quick sort that reverts to an insertion sort when the size of a segment is less than or equal to `changeOver = nBreak`. Can we obtain a faster algorithm by using a different value for `changeOver`? Why? Modify Program 18.7 to use the median-of-three rule; experiment with different values of `changeOver` and see what happens. Determine the best value of `changeOver` for the case of average performance. After you have optimized your quick sort code, compare the average performance of your code and that of the C++ STL function `sort`.
29. In this exercise we will develop a sort procedure with best worst-case performance.
- Compare the worst-case run times of insertion, bubble, selection, heap, merge, and quick sort. The worst-case input data for insertion, bubble, selection, and quick sort are easy to generate. For merge sort, write a program to generate the worst-case data. This program will essentially unmerge a sorted sequence of n elements. For heap sort, estimate the worst-case time using random permutations.

- (b) Use the results of part (a) to obtain a composite sort function that has the best worst-case performance. More likely than not, your composite procedure will include only merge and insertion sort.
 - (c) Run an experiment to determine the worst-case run time of your composite function. Compare the performance with that of the original sort functions and of the STL function `stable_sort`.
 - (d) Plot the worst-case times of the eight sort methods on a single graph sheet.
30. Start with the array [4, 3, 8, 5, 7, 10, 0, 9, 2, 11, 6, 1] and draw a figure to show the progress of Programs 18.8 and 18.9 when started with $k = 7$. You should show the segment to be searched following each partitioning pass and also give the new k value.
31. Do Exercise 30 using the array [7, 3, 6, 8, 11, 15, 0, 2, 12, 1, 4, 13, 5, 9, 10, 14] and $k = 5$.
32. Use the substitution method to solve Equation 18.13 for the case when n is a power of 2.
33. Prove Theorem 18.3.
34. Use induction to show that Equation 18.14 implies $t(n) \leq 72cn$ for $n \geq 1$.
35. Programs 18.8 and 18.9 need $O(n)$, where n is the number of elements, space for the recursion stack. This space can be entirely eliminated by replacing the recursive calls with a `while` or `for` loop. Rewrite the code in this way. Compare the run time of the two versions of the selection code.
36. (a) Recode Program 18.9 using a random number generator to select the partitioning element. Conduct experiments to compare the average performance of the two codes.
(b) Recode Program 18.9 using the median-of-medians rule with $r = 9$.
37. In an attempt to speed Program 18.12, we might eliminate the square root operator from the computation of the distance between two points and instead work with the square of the distance. Finding the closest pair is the same as finding a pair with minimum squared distance. What changes need to be made to Program 18.12? Experiment with the two versions and measure the performance improvement you can achieve.
38. Devise a faster algorithm to find the closest pair of points when all points are known to lie on a straight line. For example, suppose the points are on a horizontal line. If the points are sorted by x -coordinate, then the nearest pair contains two adjacent points. Although this strategy results in an $O(n \log n)$ algorithm if we use `mergeSort` (Program 18.3), the algorithm has considerably less overhead than Program 18.11 has and so runs faster.

Hidden page

$h(n)$	$f(n)$
$O(n^r), r < 0$	$O(1)$
$\Theta((\log n)^i), i \geq 0$	$\Theta(((\log n)^{i+1})/(i+1))$
$\Omega(n^r), r > 0$	$\Theta(h(n))$

Figure 18.19 $f(n)$ values for various $h(n)$ values

For the merge sort recurrence, we obtain $a = 2$, $b = 2$, and $g(n) = cn$. So $\log_b a = 1$, and $h(n) = g(n)/n = c = \Theta((\log n)^0)$. Hence $f(n) = \Theta(\log n)$ and $t(n) = n(t(1) + \Theta(\log n)) = \Theta(n \log n)$.

As another example, consider the recurrence

$$t(n) = 7t(n/2) + 18n^2, \quad n \geq 2 \text{ and a power of 2}$$

that corresponds to the recurrence for Strassen's matrix-multiplication method (Equation 18.6) with $k = 1$ and $c = 18$. We obtain $a = 7$, $b = 2$, and $g(n) = 18n^2$. Therefore, $\log_b a = \log_2 7 \approx 2.81$, and $h(n) = 18n^2/n^{\log_2 7} = 18n^{2-\log_2 7} = O(n^r)$ where $r = 2 - \log_2 7 < 0$. Therefore, $f(n) = O(1)$. The expression for $t(n)$ is

$$t(n) = n^{\log_2 7}(t(1) + O(1)) = \Theta(n^{\log_2 7})$$

as $t(1)$ is assumed to be a constant.

As a final example, consider the following recurrence:

$$t(n) = 9t(n/3) + 4n^6, \quad n \geq 3 \text{ and a power of 3}$$

Comparing this recurrence with Equation 18.14, we obtain $a = 9$, $b = 3$, and $g(n) = 4n^6$. Therefore, $\log_b a = 2$ and $h(n) = 4n^6/n^2 = 4n^4 = \Omega(n^4)$. From Figure 18.13 we see that $f(n) = \Theta(h(n)) = \Theta(n^4)$. Therefore,

$$t(n) = n^2(t(1) + \Theta(n^4)) = \Theta(n^6)$$

as $t(1)$ may be assumed constant.

EXERCISES

40. Use the substitution method to show that Equation 18.16 is the solution to the recurrence 18.15.
41. Use the table of Figure 18.19 to solve the following recurrences. In each case assume $t(1) = 1$.
- $t(n) = 10t(n/3) + 11n$, $n \geq 3$ and a power of 3.
 - $t(n) = 10t(n/3) + 11n^5$, $n \geq 3$ and a power of 3.
 - $t(n) = 27t(n/3) + 11n^3$, $n \geq 3$ and a power of 3.
 - $t(n) = 64t(n/4) + 10n^3 \log^2 n$, $n \geq 4$ and a power of 4.
 - $t(n) = 9t(n/2) + n^2 2^n$, $n \geq 2$ and a power of 2.
 - $t(n) = 3t(n/8) + n^2 2^n \log n$, $n \geq 8$ and a power of 8.
 - $t(n) = 128t(n/2) + 6n$, $n \geq 2$ and a power of 2.
 - $t(n) = 128t(n/2) + 6n^8$, $n \geq 2$ and a power of 2.
 - $t(n) = 128t(n/2) + 2^n/n$, $n \geq 2$ and a power of 2.
 - $t(n) = 128t(n/2) + \log^3 n$, $n \geq 2$ and a power of 2.

18.4 LOWER BOUNDS ON COMPLEXITY

$f(n)$ is an **upper bound** on the complexity of a problem iff at least one algorithm solves this problem in $O(f(n))$ time. One way to establish an upper bound of $f(n)$ on the complexity of a problem is to develop an algorithm whose complexity is $O(f(n))$. Each algorithm developed in this book established an upper bound on the complexity of the problem it solved. For example, until the discovery of Strassen's matrix-multiplication algorithm (Example 18.3), the upper bound on the complexity of matrix multiplication was n^3 , as the algorithm of Program 2.22 was already known and this algorithm runs in $\Theta(n^3)$ time. The discovery of Strassen's algorithm reduced the upper bound on the complexity of matrix multiplication to $n^{2.81}$.

$f(n)$ is a **lower bound** on the complexity of a problem iff every algorithm for this problem has complexity $\Omega(f(n))$. To establish a lower bound of $g(n)$ on the complexity of a problem, we must show that *every* algorithm for this problem has complexity $\Omega(g(n))$. Making such a statement is usually quite difficult as we are making a claim about all possible ways to solve a problem, rather than about a single way to solve it.

For many problems we can establish a trivial lower bound based on the number of inputs and/or outputs. For example, every algorithm that sorts n elements must have complexity $\Omega(n)$, as every sorting algorithm must examine each element at

least once or run the risk that the unexamined elements are in the wrong place. Similarly, every algorithm to multiply two $n \times n$ matrices must have complexity $\Omega(n^2)$, as the result contains n^2 elements and it takes $\Omega(1)$ time to produce each of these elements, and so on. Nontrivial lower bounds are known for a very limited number of problems.

In this section we establish nontrivial lower bounds on two of the divide-and-conquer problems studied in this chapter—finding the minimum and maximum of n elements and sorting. For both of these problems, we limit ourselves to **comparison algorithms**. These algorithms perform their task by making comparisons between pairs of elements and possibly moving elements around; they do not perform other operations on elements. The minmax algorithms of Chapter 2, as well as those proposed in this chapter, satisfy this property, as do all the sort methods studied in this book except for bin sort and radix sort (Sections 6.5.1 and 6.5.2).

18.4.1 Lower Bound for the Minmax Problem

Program 18.1 gave a divide-and-conquer function to find the minimum and maximum of n elements. This function makes $\lceil 3n/2 \rceil - 2$ comparisons between pairs of elements. We will show that every comparison algorithm for the minmax problem must make at least $\lceil 3n/2 \rceil - 2$ comparisons between the elements. For purposes of the proof, we assume that the n elements are distinct. This assumption does not affect the generality of the proof, as distinct element inputs form a subset of the input space. In addition, every minmax algorithm must work correctly on these inputs as well as on those that have duplicates.

The proof uses the **state space method**. In this method we describe the start, intermediate, and finish states of every algorithm for the problem as well as how a comparison algorithm can go from one state to another. Then we determine the minimum number of transitions needed to go from the start state to the finish state. This minimum number of transitions is a lower bound on the complexity of the problem. The start, intermediate, and finish states of an algorithm are abstract entities, and there is no requirement that an algorithm keep track of its state explicitly.

For the minmax problem the algorithm state can be described by a tuple (a, b, c, d) where a is the number of elements that the minmax algorithm still considers candidates for the maximum and minimum elements; b is the number of elements that are no longer candidates for the minimum, but are still candidates for the maximum; c is the number of elements that are no longer candidates for the maximum, but are still candidates for the minimum; and d is the number of elements that the minmax algorithm has determined to be neither the minimum nor the maximum. Let A , B , C , and D denote the elements in each of the preceding categories.

When the minmax algorithm starts, all n elements are candidates for the min and max. So the start state is $(n, 0, 0, 0)$. When the algorithm finishes, there are no elements in A , one in B , one in C , and $n - 2$ in D . Therefore, the finish state is $(0, 1, 1, n - 2)$. Transitions from one state to another are made on the basis of

Hidden page

state is to make $\lceil n/2 \rceil$ comparisons between elements in A , $\lceil n/2 \rceil - 1$ comparisons between elements in B , $\lceil n/2 \rceil - 1$ comparisons between elements in C , and up to two more comparisons involving the remaining element of A . The total count is $\lceil 3n/2 \rceil - 2$.

Since no comparison algorithm for the minmax problem can go from the start state to the finish state making fewer than $\lceil 3n/2 \rceil - 2$ comparisons between pairs of elements, this number is a lower bound on the number of comparisons every minmax comparison algorithm must make. Hence Program 18.1 is an optimal comparison algorithm for the minmax problem.

18.4.2 Lower Bound for Sorting

A lower bound of $n \log n$ on the worst-case complexity of every comparison algorithm that sorts n elements can be established by using the state space method. This time the algorithm state is given by the number of permutations of the n elements that are still candidates for the output permutation. When the sort algorithm starts, all $n!$ permutations of the n elements are candidates for the sorted output and when the algorithm terminates, only one candidate permutation remains. (As for the minmax problem, we assume the n elements to be sorted are distinct.)

When two elements a_i and a_j are compared, the current set of candidate permutations is divided into two groups—one group retains permutations that are consistent with the outcome $a_i < a_j$, and the other set retains those that are consistent with the outcome $a_i > a_j$. Since we have assumed that the elements are distinct, the outcome $a_i = a_j$ is not possible. For example, suppose that $n = 3$ and that the first comparison is between a_1 and a_3 . Prior to this comparison, as far as the algorithm is concerned, all six permutations of the elements are candidates for the sorted permutation. If $a_1 < a_3$, then the best the algorithm can do is eliminate the permutations (a_3, a_1, a_2) , (a_3, a_2, a_1) , and (a_2, a_3, a_1) . The remaining three permutations must be retained as candidates for the output.

If the current candidate set has m permutations, then a comparison produces two groups, one of which must have at least $\lceil m/2 \rceil$ permutations. A worst-case execution of the sort algorithm begins with a candidate set of size $n!$, reduces this set to one of size at least $n!/2$, then reduces the candidate set further to one of size at least $n!/4$, and so on until the size of the candidate set becomes 1. The number of reduction steps (and hence comparisons needed) is at least $\lceil \log n! \rceil$.

Since $n! \geq \lceil n/2 \rceil^{\lceil n/2 \rceil - 1}$, $\log n! \geq (n/2 - 1) \log(n/2) = \Omega(n \log n)$. Hence every sort algorithm that is a comparison algorithm must make $\Omega(n \log n)$ comparisons in the worst case.

We can arrive at this same lower bound from a **decision-tree** proof. In such a proof we model the progress of an algorithm by using a tree. At each internal node of this tree, the algorithm makes a comparison and moves to one of the children based on the outcome of this comparison. External nodes are nodes at which the algorithm terminates. Figure 18.20 shows the decision tree for `insertionSort` (Program 2.15) while sorting the three-element sequence `a[0:2]`. Each internal node has a label of

the type $i:j$. This label denotes a comparison between $a[i]$ and $a[j]$. If $a[i] < a[j]$, the algorithm moves to the left child. A right child move occurs when $a[i] > a[j]$. Since the elements are distinct, the case $a[i] = a[j]$ is not possible. The external nodes are labeled with the sorted permutation that is generated. The left-most path in the decision tree of Figure 18.20 is followed when $a[1] < a[0]$, $a[2] < a[0]$, and $a[2] < a[1]$; the permutation at the left-most external node is $(a[2], a[1], a[0])$.

Notice that each leaf of a decision tree for a comparison sort algorithm defines a unique output permutation. Since every correct sorting algorithm must be able to produce all $n!$ permutations of n inputs, the decision tree for every correct comparison sort algorithm must have at least $n!$ external nodes. Because a tree whose height is h has at most 2^h external nodes, the decision tree for a correct comparison sort algorithm must have a height that is at least $\lceil \log_2 n! \rceil = \Omega(n \log n)$. Therefore, every comparison sort algorithm must perform $\Omega(n \log n)$ comparisons in the worst case. Further, since the average height of every binary tree that has $n!$ external nodes is also $\Omega(n \log n)$ (Exercise 47), the average complexity of every comparison sort algorithm is also $\Omega(n \log n)$.

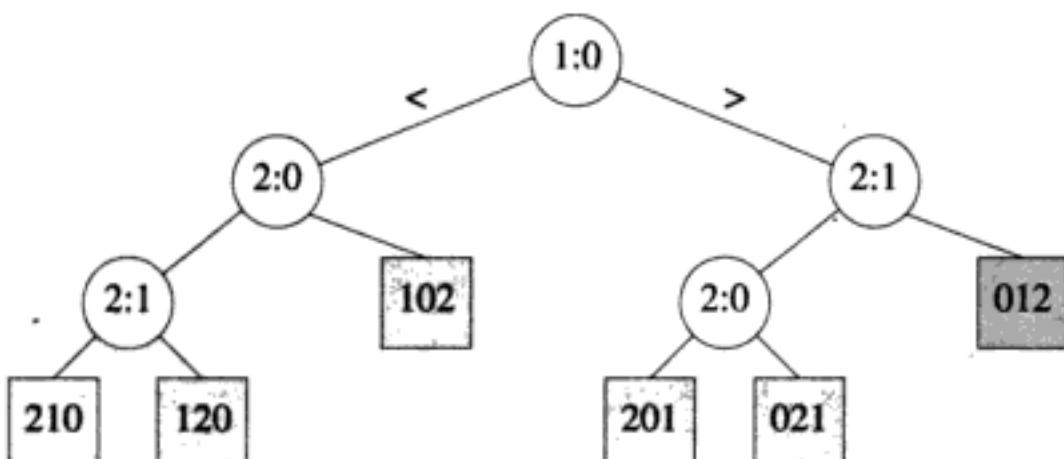


Figure 18.20 Decision tree for `insertionSort` when $n = 3$

The preceding lower-bound proof shows that heap sort and merge sort are optimal worst-case sort methods (as far as asymptotic complexity is concerned) and that heap sort, merge sort, and quick sort are optimal average-case methods.

EXERCISES

42. Use the state space method to show that every comparison algorithm that finds the maximum of n elements makes at least $n - 1$ comparisons between pairs of elements.
43. Show that $n! \geq [n/2]^{\lceil n/2 \rceil - 1}$.

Hidden page

CHAPTER 19

DYNAMIC PROGRAMMING

BIRD'S-EYE VIEW

Dynamic programming is arguably the most difficult of the five design methods we are studying. It has its foundations in the principle of optimality. We can use this method to obtain elegant and efficient solutions to many problems that cannot be so solved with either the greedy or divide-and-conquer methods. After describing the method, we consider its application to the solution of the knapsack, matrix multiplication chains, shortest-path, and noncrossing subset of nets problems. Additional applications (e.g., image compression and component folding) are developed on the Web site. You should study these as well as the exercise solutions to gain mastery of dynamic programming.

19.1 THE METHOD

In dynamic programming, as in the greedy method, we view the solution to a problem as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequences. Some examples that illustrate this point are given below.

Example 19.1 [Shortest Path] Consider the digraph of Figure 17.2. We wish to find a shortest path from the source vertex $s = 1$ to the destination vertex $d = 5$. We need to make decisions on the intermediate vertices. The choices for the first decision are 2, 3, and 4. That is, from vertex 1 we may move to any one of these vertices. Suppose we decide to move to vertex 3. Now we need to decide on how to get from 3 to 5. If we go from 3 to 5 in a suboptimal way, then the 1-to-5 path constructed cannot be optimal, even under the restriction that from vertex 1 we must go to vertex 3. For example, if we use the suboptimal path 3, 2, 5 with length 9, the constructed 1-to-5 path 1, 3, 2, 5 has length 11. Replacing the suboptimal path 3, 2, 5 with an optimal one 3, 4, 5 results in the path 1, 3, 4, 5 of length 9.

For this shortest-path problem, suppose that our first decision gets us to some vertex v . Although we do not know how to make this first decision we do know that the remaining decisions must be optimal for the problem of going from v to d . ■

Example 19.2 [0/1 Knapsack Problem] Consider the 0/1 knapsack problem of Section 17.3.2. We need to make decisions on the values of x_1, \dots, x_n . Suppose we are deciding the values of the x_i 's in the order $i = 1, 2, \dots, n$. If we set $x_1 = 0$, then the available knapsack capacity for the remaining objects (i.e., objects 2, 3, ..., n) is c . If we set $x_1 = 1$, the available knapsack capacity is $c - w_1$. Let $r \in \{c, c - w_1\}$ denote the remaining knapsack capacity.

Following the first decision, we are left with the problem of filling a knapsack with capacity r . The available objects (i.e., 2 through n) and the available capacity r define the *problem state* following the first decision. Regardless of whether x_1 is 0 or 1, $[x_2, \dots, x_n]$ must be an optimal solution for the problem state following the first decision. If not, there is a solution $[y_2, \dots, y_n]$ that provides greater profit for the problem state following the first decision. So $[x_1, y_2, \dots, y_n]$ is a better solution for the initial problem.

Suppose that $n = 3$, $w = [100, 14, 10]$, $p = [20, 18, 15]$, and $c = 116$. If we set $x_1 = 1$, then following this decision, the available knapsack capacity is 16. $[x_2, x_3] = [0, 1]$ is a feasible solution to the two-object problem that remains. It returns a profit of 15. However, it is not an optimal solution to the remaining two-object problem, as $[x_2, x_3] = [1, 0]$ is feasible and returns a greater profit of 18. So $x = [1, 0, 1]$ can be improved to $[1, 1, 0]$. If we set $x_1 = 0$, the available capacity for the two-object instance that remains is 116. If the subsequence $[x_2, x_3]$ is not an

optimal solution for this remaining instance, then $[x_1, x_2, x_3]$ cannot be optimal for the initial instance. ■

Example 19.3 [Airfares] A certain airline has the following airfare structure: From Atlanta to New York or Chicago, or from Los Angeles to Atlanta, the fare is \$100; from Chicago to New York, it is \$20; and for passengers connecting through Atlanta, the Atlanta to Chicago segment is only \$20. A routing from Los Angeles to New York involves decisions on the intermediate airports. If problem states are encoded as (origin, destination) pairs, then following a decision to go from Los Angeles to Atlanta, the problem state is *We are at Atlanta and need to get to New York*. The cheapest way to go from Atlanta to New York is a direct flight with cost \$100. Using this direct flight results in a total Los Angeles-to-New York cost of \$200. However, the cheapest routing is Los Angeles-Atlanta-Chicago-New York with a cost of \$140, which involves using a suboptimal decision subsequence for the Atlanta-to-New York problem (Atlanta-Chicago-New York).

If instead we encode the problem state as a triple $(tag, origin, destination)$ where *tag* is 0 for connecting flights and 1 for all others, then once we reach Atlanta, the state becomes $(0, \text{Atlanta}, \text{New York})$ for which the optimal routing is through Chicago. ■

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called **dynamic-programming recurrence equations**, that enable us to solve the problem in an efficient way.

Example 19.4 [0/1 Knapsack] In Example 19.2 we saw that for the 0/1 knapsack problem, optimal decision sequences were composed of optimal subsequences. Let $f(i, y)$ denote the value of an optimal solution to the knapsack instance with remaining capacity y and remaining objects $i, i+1, \dots, n$. From Example 19.2 it follows that

$$f(n, y) = \begin{cases} p_n & y \geq w_n \\ 0 & 0 \leq y < w_n \end{cases} \quad (19.1)$$

and

$$f(i, y) = \begin{cases} \max\{f(i+1, y), f(i+1, y - w_i) + p_i\} & y \geq w_i \\ f(i+1, y) & 0 \leq y < w_i \end{cases} \quad (19.2)$$

By making use of the observation that optimal decision sequences are made up of optimal subsequences, we have obtained a recurrence for f . $f(1, c)$ is the value of the optimal solution to the knapsack problem we started with. Equation 19.2 may be used to determine $f(1, c)$ either recursively or iteratively. In the iterative approach, we start with $f(n, *)$, as given by Equation 19.1, and then obtain $f(i, *)$.

in the order $i = n - 1, n - 2, \dots, 2$, using Equation 19.2. Finally, $f(1, c)$ is computed by using Equation 19.2.

For the instance of Example 19.2, we see that $f(3, y) = 0$ if $0 \leq y < 10$, and 15 if $y \geq 10$. Using Equation 19.2, we obtain $f(2, y) = 0$ if $0 \leq y < 10$, 15 if $10 \leq y < 14$, 18 if $14 \leq y < 24$, and 33 if $y \geq 24$. The optimal solution has value $f(1, 116) = \max\{f(2, 116), f(2, 116 - w_1) + p_1\} = \max\{f(2, 116), f(2, 16) + 20\} = \max\{33, 38\} = 38$.

To obtain the values of the x_i s, we proceed as follows: If $f(1, c) = f(2, c)$, then we may set $x_1 = 0$ because we can utilize the c units of capacity getting a return of $f(1, c)$ from objects $2, \dots, n$. In case $f(1, c) \neq f(2, c)$, then we must set $x_1 = 1$. Next we need to find an optimal solution that uses the remaining capacity $c - w_1$. This solution has value $f(2, c - w_1)$. Proceeding in this way, we may determine the value of all the x_i s.

For our sample instance we see that $f(2, 116) = 33 \neq f(1, 116)$. Therefore, $x_1 = 1$, and we need to find x_2 and x_3 so as to obtain a return of $38 - p_1 = 18$ and use a capacity of at most $116 - w_1 = 16$. Note that $f(2, 16) = 18$. Since $f(3, 16) = 15 \neq f(2, 16)$, $x_2 = 1$; the remaining capacity is $16 - w_2 = 2$. Since $f(3, 2) = 0$, we set $x_3 = 0$. ■

The principle of optimality states that no matter what the first decision, the remaining decisions must be optimal with respect to the state that results from this first decision. This principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. *Dynamic programming cannot be applied when this principle does not hold.*

The steps in a dynamic-programming solution are

- Verify that the principle of optimality holds.
- Set up the dynamic-programming recurrence equations.
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a **traceback** step in which the solution itself is constructed.

It is very tempting to write a simple recursive program to solve the dynamic-programming recurrence. *However, as we will see in subsequent sections, unless care is taken to avoid recomputing previously computed values, the recursive program will have prohibitive complexity.* When the recursive program is designed to avoid this recomputation, the complexity is drastically reduced. The dynamic-programming recurrence may also be solved by iterative code that naturally avoids recomputation of already computed values. Although this iterative code has the same time complexity as the “careful” recursive code, the former has the advantage of not requiring additional space for the recursion stack. As a result, the iterative code generally runs faster than the careful recursive code.

Hidden page

Hidden page

Hidden page

```
void knapsack(int *profit, int *weight, int numberOfObjects,
               int knapsackCapacity, int **f)
{// Iterative method to solve dynamic programming recurrence.
// Computes f[1][knapsackCapacity] and f[i][y],
// 2 <= i <= numberOfObjects, 0 <= y <= knapsackCapacity.
// profit[1:numberOfObjects] gives object profits.
// weight[1:numberOfObjects] gives object weights.

// initialize f[numberOfObjects][0]
int yMax = min(weight[numberOfObjects] - 1, knapsackCapacity);
for (int y = 0; y <= yMax; y++)
    f[numberOfObjects][y] = 0;
for (int y = weight[numberOfObjects]; y <= knapsackCapacity; y++)
    f[numberOfObjects][y] = profit[numberOfObjects];

// compute f[i][y], 1 < i < numberOfObjects
for (int i = numberOfObjects - 1; i > 1; i--)
{
    yMax = min(weight[i] - 1, knapsackCapacity);
    for (int y = 0; y <= yMax; y++)
        f[i][y] = f[i + 1][y];
    for (int y = weight[i]; y <= knapsackCapacity; y++)
        f[i][y] = max(f[i + 1][y], f[i + 1][y - weight[i]] + profit[i]);
}

// compute f[1][knapsackCapacity]
f[1][knapsackCapacity] = f[2][knapsackCapacity];
if (knapsackCapacity >= weight[1])
    f[1][knapsackCapacity] = max(f[1][knapsackCapacity],
                                 f[2][knapsackCapacity - weight[1]] + profit[1]);
}
```

Program 19.3 Iterative computation of f

Example 19.6 Figure 19.2 shows the f array computed by Program 19.3 using the data of Example 19.5. The data are computed by rows from top to bottom and within a row from left to right. The value $f[1][10] = 15$ is not shown.

To determine the x_i values, we begin with x_1 . Since $f(1, 10) \neq f(2, 10)$, $f(1, 10)$ must equal $f(2, 10 - w_1) + p_1 = f(2, 8) + 6$. Therefore, $x_1 = 1$. Since $f(2, 8) \neq f(3, 8)$, $f(2, 8)$ must equal $f(3, 8 - w_2) + p_2 = f(3, 6) + 3$ and $x_2 = 1$. $x_3 = x_4 = 0$ because $f(3, 6) = f(4, 6) = f(5, 6)$. Finally, since $f(5, 4) \neq 0$, $x_5 = 1$. ■

```

void traceback(int **f, int *weight, int numberObjects,
              int knapsackCapacity, int *x)
{// Compute solution vector x.
    for (int i = 1; i < numberObjects; i++)
        if (f[i][knapsackCapacity] == f[i+1][knapsackCapacity])
            // do not include object i
            x[i] = 0;
        else
            // include object i
            x[i] = 1;
            knapsackCapacity -= weight[i];
    }
    x[numberObjects] = (f[numberObjects][knapsackCapacity] > 0)
                    ? 1 : 0;
}

```

Program 19.4 Iterative computation of x

i	y										
	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11

Figure 19.2 f function/array for Example 19.6

The complexity of the iterative knapsack code is $\Theta(nc)$ and that of `traceback` is $\Theta(n)$.

Tuple Method (Optional)

The code of Program 19.3 has two drawbacks. First, it requires that the weights be integer. Second, it is slower than Program 19.1 when the knapsack capacity is large. In particular, if $c > 2^n$, its complexity is $\Omega(n2^n)$. We can overcome both of these shortcomings by using a tuple approach in which for each i , $f(i, y)$ is stored as an ordered list $P(i)$ of pairs $(y, f(i, y))$ that correspond to the y values at which the function f changes. The pairs in each $P(i)$ are in increasing order of y . In addition, since $f(i, y)$ is a nondecreasing function of y , the pairs are also in increasing order of $f(i, y)$.

Hidden page

the two matrices. Suppose we are to multiply three matrices A , B , and C . There are two ways in which we can accomplish this task. In the first, we multiply A and B to get the product matrix D and then multiply D and C to get the desired result. This multiplication order can be written as $(A * B) * C$. The second way is $A * (B * C)$. Although both multiplication orders obtain the same result, one may take a lot more computing time than the other.

Example 19.9 Suppose that A is a 100×1 matrix, B is a 1×100 matrix, and C is a 100×1 matrix. Then the time needed to compute $A * B$ is 10,000. Since the result is a 100×100 matrix, the time needed to perform the multiplication with C is 1,000,000. The overall time needed to compute $(A * B) * C$ is therefore 1,010,000. $B * C$ can be computed in 10,000 units of time. Since the result is a 1×1 matrix, the time needed for the multiplication with A is 100. The total time needed to compute $A * (B * C)$ is therefore 10,100! Furthermore, when computing $(A * B) * C$, we need 10,000 units of space to store $A * B$; however, when $A * (B * C)$ is computed, only one unit of space is needed for $B * C$.

As an example of a real problem that can benefit from computing the matrix product $A * B * C$ in the proper order, consider the registration of 2 three-dimensional images. In the registration problem, we are to determine the amount by which one image needs to be rotated, translated, and shrunk (or expanded) so that it approximates the second. One way to perform this registration involves doing about 100 iterations of computation. Each iteration computes the following 12×1 vector T :

$$T = \sum A(x, y, z) * B(x, y, z) * C(x, y, z)$$

Here A , B , and C are, respectively, 12×3 , 3×3 , and 3×1 matrices. (x, y, z) gives the coordinates of a voxel, and the sum is done over all voxels. Let t be the number of computations needed to compute $A(x, y, z) * B(x, y, z) * C(x, y, z)$ for a single voxel. Assume that the image is of size $256 \times 256 \times 256$ voxels. In this case the total number of computations needed for the 100 iterations is approximately $100 * 256^3 * t \approx 1.7 * 10^9 t$. When the three matrices are multiplied from left to right, $t = 12 * 3 * 3 + 12 * 3 * 1 = 144$. When we multiply from right to left, $t = 3 * 3 * 1 + 12 * 3 * 1 = 45$. The left-to-right computation requires approximately $2.4 * 10^{11}$ operations, while the right-to-left computation requires about $7.5 * 10^{10}$ operations. On a computer that can do 100 million operations per second, the first scheme would take 40 minutes and the second would take 12.5 minutes. ■

When we are to compute the matrix product $A * B * C$, only two multiplication orders are possible (left to right and right to left). We can determine the number of operations each order requires and go with the cheaper one. In a more general situation, we are to compute the matrix product $M_1 \times M_2 \times \cdots \times M_q$ where M_i is an $r_i \times r_{i+1}$ matrix, $1 \leq i \leq q$. Consider the case $q = 4$. The matrix product $A * B * C * D$ may be computed in any of the following five ways:

$$\begin{array}{ccc}
 A * ((B * C) * D) & A * (B * (C * D)) \\
 (A * B) * (C * D) & ((A * B) * C) * D & (A * (B * C)) * D
 \end{array}$$

The number of different ways in which the product of q matrices may be computed increases exponentially with q . As a result, for large q it is not practical to evaluate each multiplication scheme and select the cheapest.

Dynamic-Programming Formulation

We can use dynamic programming to determine an optimal sequence of pairwise matrix multiplications to use. The resulting algorithm runs in $O(q^3)$ time. Let M_{ij} denote the result of the product chain $M_i \times \dots \times M_j$, $i \leq j$, and let $c(i, j)$ be the cost of the optimal way to compute M_{ij} . Let $kay(i, j)$ be such that the optimal computation of M_{ij} computes $M_{ik} \times M_{k+1,j}$. An optimal computation of M_{ij} therefore comprises the product $M_{ik} \times M_{k+1,j}$ preceded by optimal computations of M_{ik} and $M_{k+1,j}$. The principle of optimality holds, and we obtain the dynamic-programming recurrence that follows.

$$\begin{aligned}
 c(i, i) &= 0, 1 \leq i \leq q \\
 c(i, i+1) &= r_i r_{i+1} r_{i+2} \text{ and } kay(i, i+1) = i, 1 \leq i < q \\
 c(i, i+s) &= \min_{i \leq k < i+s} \{c(i, k) + c(k+1, i+s) + r_i r_{k+1} r_{i+s+1}\}, \\
 &\quad 1 \leq i \leq q-s, 1 < s < q \\
 kay(i, i+s) &= \text{value of } k \text{ that obtains the above minimum}
 \end{aligned}$$

The above recurrence for c may be solved recursively or iteratively. $c(1, q)$ is the cost of the optimal way to compute the matrix product chain, and $kay(1, q)$ defines the last product to be done. The remaining products can be determined by using the kay values.

Recursive Solution

As in the case of the 0/1 knapsack problem, any recursive solution must be implemented so as to avoid computing the same $c(i, j)$ and $kay(i, j)$ values more than once otherwise, the complexity of the algorithm is too high.

Example 19.10 Consider the case $q = 5$ and $r = (10, 5, 1, 10, 2, 10)$. The dynamic-programming recurrence yields

$$\begin{aligned}
 c(1, 5) &= \min\{c(1, 1) + c(2, 5) + 500, c(1, 2) + c(3, 5) + 100, \\
 &\quad c(1, 3) + c(4, 5) + 1000, c(1, 4) + c(5, 5) + 200\}
 \end{aligned} \tag{19.3}$$

Hidden page

is $M_{12} \times M_{35}$. Since both M_{12} and M_{35} are to be computed optimally, the *kay* values may be used to figure out how. $\text{kay}(1, 2) = 1$, so M_{12} is computed as $M_{11} \times M_{22}$. Also, since $\text{kay}(3, 5) = 4$, M_{35} is optimally computed as $M_{34} \times M_{55}$. M_{34} in turn is computed as $M_{33} \times M_{44}$, so the optimal multiplication sequence is

Multiply M_{11} and M_{22} to get M_{12}
 Multiply M_{33} and M_{44} to get M_{34}
 Multiply M_{34} and M_{55} to get M_{35}
 Multiply M_{12} and M_{35} to get M_{15}

■

The recursive code to determine $c(i, j)$ and $\text{kay}(i, j)$ appears in Program 19.5. This code assumes that the one-dimensional integer array **r** and the two-dimensional integer array **kay** are global. The code computes the value of $c(i, j)$ and also sets **kay[a][b] = kay(a, b)** for all a and b for which $c(a, b)$ is computed during the computation of $c(i, j)$. The function **traceback** uses the **kay** values computed by the function **c** to determine the optimal multiplication sequence.

Let $t(q)$ be the complexity of function **c** when $j - i + 1 = q$ (i.e., when M_{ij} is composed of q matrices). We see that when q is 1 or 2, $t(q) = d$ where d is a constant. When $q > 2$, $t(q) = 2 \sum_{k=1}^{q-1} t(k) + eq$ where e is a constant. For $q > 2$, $t(q) > 2t(q-1) + e$. So $t(q) = \Omega(2^q)$. The complexity of function **traceback** is $O(q)$.

Recursive Solution without Recomputations

By avoiding the recomputation of **c** (and hence **kay**) values previously computed, the complexity can be reduced to $O(q^3)$. To avoid the recomputation, we need to save the values of the $c(i, j)$ s in a global two-dimensional array **theC[][]** whose initial values are 0. Program 19.6 gives the new recursive code for function **c**.

To analyze the complexity of the new function **c** (Program 19.6), we will again use the cost amortization method. Observe that the invocation **c(1, q)** causes each $c(i, j)$, $1 \leq i \leq j \leq q$ to be computed exactly once. For $s = j - i > 1$, the computation of each requires s amount of work in addition to the work done computing the needed $c(a, b)$ s that haven't as yet been computed. This additional work is charged to the $c(a, b)$ s that are being computed for the first time. These $c(a, b)$ s, in turn, offload some of this charge to the first-time c s that need to be computed during the computation of $c(a, b)$. As a result, each $c(i, j)$ is charged s amount of work. For each s , $q - s + 1$ $c(i, j)$ s are computed. The total cost is therefore $\sum_{s=1}^{q-1} s(q - s + 1) = O(q^3)$.

Iterative Solution

The dynamic-programming recurrence for **c** may be solved iteratively, computing each **c** and **kay** exactly once, by computing the $c(i, i+s)$ s in the order $s = 2, 3, \dots, q-1$.

Hidden page

Hidden page

values are 1, 2, 3, and 4.

When $s = 2$, we get

$$\begin{aligned} c(1,3) &= \min\{c(1,1) + c(2,3) + r_1r_2r_4, c(1,2) + c(3,3) + r_1r_3r_4\} \\ &= \min\{0 + 50 + 500, 50 + 0 + 100\} = 150 \end{aligned}$$

and $\text{kay}(1,3) = 2$. $c(2,4)$ and $c(3,5)$ are computed in a similar way. Their values are 30 and 40. The corresponding kay values are 2 and 3.

When $s = 3$, we compute $c(1,4)$ and $c(2,5)$. All the values needed to compute $c(2,5)$ (see Equation 19.4) are known. Substituting these values, we get $c(2,5) = 90$ and $\text{kay}(2,5) = 2$. $c(1,4)$ is computed from a similar equation. Finally, when $s = 4$, only $c(1,5)$ is to be computed. The equation is 19.3. All quantities on the right side are known. ■

The iterative code to compute the c and kay values is function `matrixChain` (Program 19.7). Its complexity is $O(q^3)$. We can use a traceback to determine the optimal multiplication sequence following the computation of kay .

19.2.3 All-Pairs Shortest Paths

Problem Description

In the **all-pairs shortest-path problem**, we are to find a shortest path between every pair of vertices in a directed graph G . That is, for every pair of vertices (i, j) , we are to find a shortest path from i to j as well as one from j to i . These two paths are the same when G is undirected.

When no edge has a negative length, the all-pairs shortest-path problem may be solved by using Dijkstra's greedy single-source algorithm (Section 17.3.5) n times, once with each of the n vertices as the source vertex. This process results in an $O(n^3)$ solution to the all-pairs problem. The dynamic-programming solution we develop in this section, called Floyd's algorithm, runs in $\Theta(n^3)$ time. Floyd's algorithm works even when the graph has negative-length edges (provided there are no negative-length cycles). Also, the constant factor associated with Floyd's algorithm is smaller than that associated with Dijkstra's algorithm. Therefore, Floyd's algorithm takes less time than the worst-case time for n applications of Dijkstra's algorithm.

Dynamic-Programming Formulation

We permit negative-length edges but require that G not contain any negative-length cycles. Under this assumption, every pair of vertices (i, j) always has a shortest path that contains no cycle. When the graph has a cycle whose length is less than 0, some shortest paths have length $-\infty$, as they involve going around the negative-length cycle indefinitely.

Hidden page

∞ otherwise. $c(i, j, n)$ is the length of a shortest path from i to j .

Example 19.12 For the digraph of Figure 17.17, $c(1, 3, k) = \infty$ for $k = 0, 1, 2, 3$; $c(1, 3, 4) = 28$; $c(1, 3, k) = 10$ for $k = 5, 6, 7$; and $c(1, 3, k) = 9$ for $k = 8, 9, 10$. Hence the shortest 1-to-3 path has length 9. ■

How can we determine $c(i, j, k)$ for any $k, k > 0$? There are two possibilities for a shortest i -to- j path that has no intermediate vertex larger than k . This path may or may not have k as an intermediate vertex. If it does not, then its length is $c(i, j, k - 1)$. If it does, then its length is $c(i, k, k - 1) + c(k, j, k - 1)$. $c(i, j, k)$ is the smaller of these two quantities. So we obtain the recurrence

$$c(i, j, k) = \min\{c(i, j, k - 1), c(i, k, k - 1) + c(k, j, k - 1)\}, \quad k > 0$$

The above recurrence formulates the solution for one k in terms of the solutions for $k - 1$. Obtaining solutions for $k - 1$ should be easier than obtaining those for k directly.

Recursive Solution

If the above recurrence is solved recursively, the complexity of the resulting procedure is excessive. Let $t(k)$ be the time needed to solve the recurrence recursively for any i, j, k combination. From the recurrence, we see that $t(k) = 3t(k - 1) + c$. Using the substitution method, we obtain $t(n) = \Theta(3^n)$. So the time needed to obtain all the $c(i, j, n)$ values is $\Theta(n^2 3^n)$.

Iterative Solution

The values $c(i, j, n)$ may be obtained far more efficiently by noticing that some $c(i, j, k - 1)$ values get used several times. By avoiding the recomputation of $c(i, j, k)$ s that were computed earlier, all c values may be determined in $\Theta(n^3)$ time. This strategy may be implemented recursively as we did for the matrix chain problem (see Program 19.6) or iteratively. We will develop only the iterative code. Our first attempt at developing this iterative code results in the pseudocode of Figure 19.3.

Observe that $c(i, k, k) = c(i, k, k - 1)$ and that $c(k, i, k) = c(k, i, k - 1)$ for all i . As a result, if $c(i, j)$ replaces $c(i, j, *)$ throughout Figure 19.3, the final value of $c(i, j)$ will be the same as $c(i, j, n)$.

C++ Implementation

With this observation Figure 19.3 may be refined into the C++ code of Program 19.8. This refinement uses the class `adjacencyWDigraph` defined in Program 16.2. Method `allPairs` returns, in `c`, the lengths of the shortest paths. In case there is no path from i to j , `c[i][j]` is set to `noEdge`, which is the weight used to designate an edge that is absent from the digraph. This method also computes

```
// Find the lengths of the shortest paths.  
// initialize c(i,j,0)  
for (int i = 1; i <= n; i++)  
    for (int j = 1; j <= n; j++)  
        c(i, j, 0) = a(i, j); // a is the cost adjacency matrix  
  
// compute c(i,j,k) for 0 < k <= n  
for (int k = 1; k <= n; k++)  
    for (int i = 1; i <= n; i++)  
        for (int j = 1; j <= n; j++)  
            if (c(i, k, k - 1) + c(k, j, k - 1) < c(i, j, k - 1))  
                c(i, j, k) = c(i, k, k - 1) + c(k, j, k - 1);  
            else  
                c(i, j, k) = c(i, j, k - 1);
```

Figure 19.3 Initial shortest-paths algorithm

`kay[i][j]` such that `kay[i][j]` is the largest k that is on a shortest i -to- j path. The `kay` values may be used to construct a shortest path from one vertex to another (see method `outputPath` of Program 19.9).

The time complexity of Program 19.8 is readily seen to be $\Theta(n^3)$. Program 19.9 takes $O(n)$ time to output a shortest path.

Example 19.13 A sample cost array `a` appears in Figure 19.4(a). Figure 19.4(b) gives the `c` array computed by Program 19.8, and Figure 19.4(c) gives the `kay` values. From these `kay` values we see that the shortest path from 1 to 5 is the shortest path from 1 to `kay[1][5] = 4` followed by the shortest path from 4 to 5. The shortest path from 4 to 5 has no intermediate vertex on it, as `kay[4][5] = 0`. The shortest path from 1 to 4 goes through `kay[1][4] = 3`. Repeating this process, we determine that the shortest 1-to-5 path is 1, 2, 3, 4, 5. ■

19.2.4 Single-Source Shortest Paths with Negative Costs Introduction

Do graphs with negative edge costs actually arise in practice? They must; otherwise, we would not study such graphs. Consider a graph in which vertices represent cities, and an edge cost gives the cost of renting a car in one city and dropping it off in another. Most edge costs are positive. However, if there is a net migration into Florida, then Florida will have a surplus of cars and cities that are losing population will have no cars. To fix this imbalance in rental cars, the rental company will

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

```
void bellmanFord(int s, T *d, int *p)
{// Bellman and Ford algorithm to find the shortest paths from vertex s.
// Graph may have edges with negative cost but should not have a
// cycle with negative cost.
// Shortest distances from s are returned in d.
// Predecessor info in returned p.
if (!weighted())
    throw undefinedMethod
    ("graph::bellmanFord() not defined for unweighted graphs");

int n = numberVertices();
if (s < 1 || s > n)
    throw illegalParameterValue("illegal source vertex");

// define two lists for vertices whose d has changed
arrayList<int> *list1 = new arrayList<int>;
arrayList<int> *list2 = new arrayList<int>;

// define an array to record vertices that are in list2
bool *inList2 = new bool [n+1];

// initialize p[1:n] = 0 and inList2[1:n] = false
fill(p + 1, p + n + 1, 0);
fill(inList2 + 1, inList2 + n + 1, false);

// set d[s] = d^0(s) = 0
d[s] = 0;
p[s] = s; // p[i] != 0 means vertex i has been reached
           // will later reset p[s] = 0

// initialize list1
list1->insert(0, s);

// do n - 1 rounds of updating d
for (int k = 1; k < n; k++)
{
    if (list1->empty())
        break; // no more changes possible
```

Program 19.10 Bellman-Ford algorithm (continues)

Hidden page

Hidden page

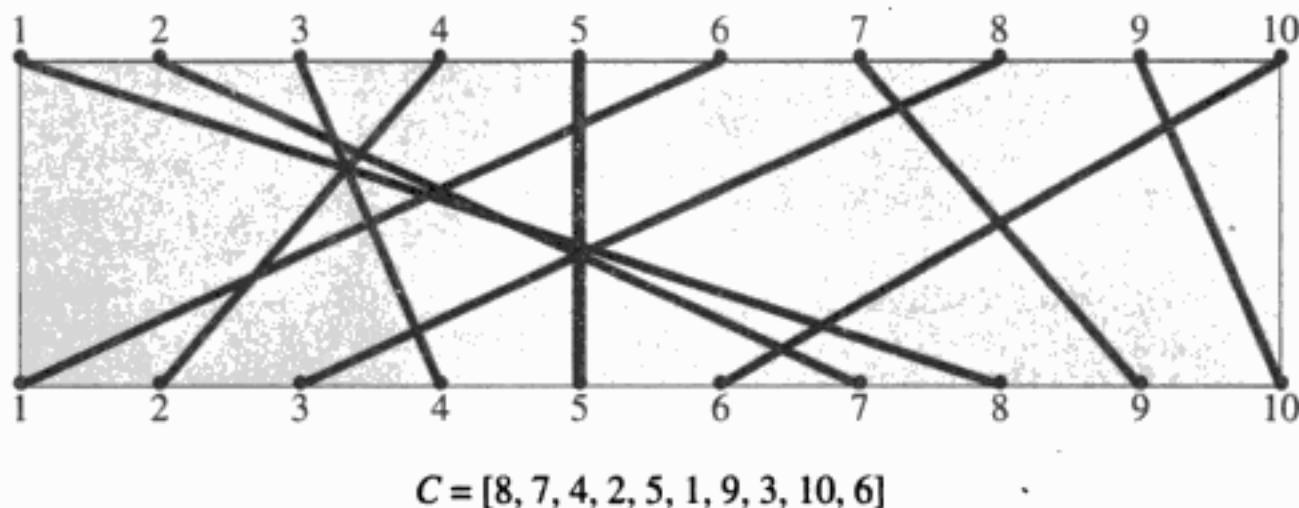
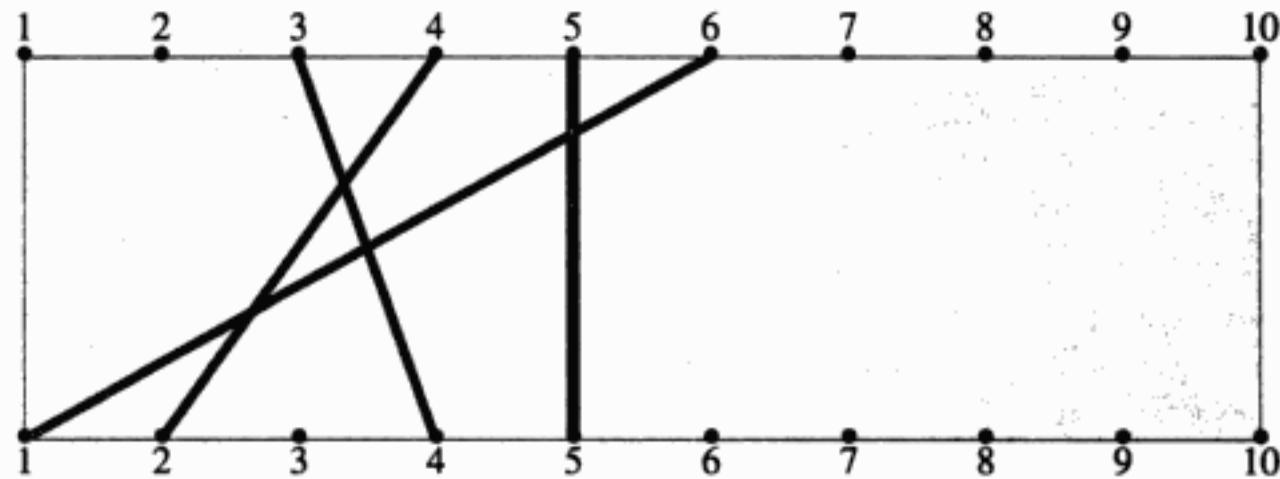


Figure 19.8 A wiring instance

pin number is greater than 7 or their bottom pin number is greater than 6. So we are left with four nets that are eligible for membership in $MNS(7,6)$. These nets appear in Figure 19.9. The subset $(3,4), (5,5)$ is a noncrossing subset of size 2. There is no noncrossing subset of size 3. So $\text{size}(7,6) = 2$. ■

Figure 19.9 Nets of Figure 19.8 that may be in $MNS(7,6)$

When $i = 1$, the net $(1, C_1)$ is the only candidate for membership in $MNS(1,j)$. This net can be a member only when $j \geq C_1$. So we obtain

$$\text{size}(1,j) = \begin{cases} 0 & \text{if } j < C_1 \\ 1 & j \geq C_1 \end{cases} \quad (19.5)$$

Next consider the case $i > 1$. If $j < C_i$, then the net (i, C_i) cannot be part of $MNS(i, j)$. In this case all nets (u, C_u) in $MNS(i, j)$ have $u < i$ and $C_u < j$. Therefore

$$\text{size}(i, j) = \text{size}(i - 1, j), \quad j < C_i \quad (19.6)$$

If $j \geq C_i$, then the net (i, C_i) may or may not be in $MNS(i, j)$. If it is, then no nets (u, C_u) such that $u < i$ and $C_u > C_i$ can be members of $MNS(i, j)$, as nets of this form cross (i, C_i) . All other nets in $MNS(i, j)$ must have $u < i$ and $C_u < C_i$. The number of such nets in $MNS(i, j)$ must be M_{i-1, C_i-1} ; otherwise, $MNS(i, j)$ doesn't have the maximum number of nets possible. If (i, C_i) is not in $MNS(i, j)$, then all nets in $MNS(i, j)$ have $u < i$; therefore, $\text{size}(i, j) = \text{size}(i - 1, j)$. Although we do not know whether the net (i, C_i) is in $MNS(i, j)$, of the two possibilities, the one that gives the larger MNS must hold. Therefore

$$\text{size}(i, j) = \max\{\text{size}(i - 1, j), \text{size}(i - 1, C_i - 1) + 1\}, \quad j \geq C_i \quad (19.7)$$

Iterative Solution

Although Equations 19.5 through 19.7 may be solved recursively, our earlier examples have shown that the recursive solution of dynamic-programming recurrences is inefficient even when we avoid recomputation of previously computed values. So we consider only an iterative solution. For this iterative solution, we use Equation 19.5 to first compute $\text{size}(1, j)$. Next we compute $\text{size}(i, j)$ for $i = 2, 3, \dots, n$ in this order of i using Equations 19.6 and 19.7. Finally, we use a traceback to determine the nets in $MNS(n, n)$.

Example 19.16 Figure 19.10 shows the $\text{size}(i, j)$ values obtained for the example of Figure 19.8. Since $\text{size}(10, 10) = 4$, we know that the MNS for this instance has four nets. To find these four nets, we begin at $\text{size}(10, 10)$. $\text{size}(10, 10)$ was computed using Equation 19.7. Since $\text{size}(10, 10) = \text{size}(9, 10)$, it follows from the reasoning used to obtain Equation 19.7 that there is an MNS of size 4 that does not include net 10. We must now find $MNS(9, 10)$. Since $\text{size}(9, 10) \neq \text{size}(8, 10)$, $MNS(9, 10)$ must include net 9. The remainder of the nets in $MNS(9, 10)$ also constitute $MNS(8, C_9 - 1) = MNS(8, 9)$. Since $\text{size}(8, 9) = \text{size}(7, 9)$, net 8 may be excluded from the MNS. So we proceed to determine $MNS(7, 9)$, which must include net 7 as $\text{size}(7, 9) \neq \text{size}(6, 9)$. The remainder of the MNS is $MNS(6, C_7 - 1) = MNS(6, 8)$. Net 6 is excluded as $\text{size}(6, 8) = \text{size}(5, 8)$. Net 5 is added to the MNS, and we proceed to determine $MNS(4, C_5 - 1) = MNS(4, 4)$. Net 4 is

excluded, and then net 3 is added to the MNS. No other nets are added. The traceback yields the size 4 MNS {3, 5, 7, 9}.

Notice that the traceback does not require $\text{size}(10, j)$ for values of j other than 10. We need not compute the values that are not required. ■

i	j									
	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	1	1	1
3	0	0	0	1	1	1	1	1	1	1
4	0	1	1	1	1	1	1	1	1	1
5	0	1	1	1	2	2	2	2	2	2
6	1	1	1	1	2	2	2	2	2	2
7	1	1	1	1	2	2	2	2	3	3
8	1	1	2	2	2	2	2	2	3	3
9	1	1	2	2	2	2	2	2	3	4
10	1	1	2	2	2	3	3	3	3	4

Figure 19.10 $\text{size}(i, j)$ s for the instance of Figure 19.8

C++ Implementation

Program 19.11 gives the iterative codes to compute the $\text{size}(i, j)$ s and then the MNS. Function `mns` computes the $\text{size}(i, j)$ values in a two-dimensional array `size`. The mapping is $\text{size}(i, j) = \text{size}[i][j]$. $\text{size}[i][j] = \text{size}(i, j)$ is computed for $1 \leq i < n$, $0 \leq j \leq n$ as well as for $i = j = n$. The time taken by this computation is $\Theta(n^2)$. Function `traceback` (Program 19.12) returns, in `net[0:sizeOfMNS]`, the nets that form an MNS. This function takes $\Theta(n)$ time, so the total time taken by the dynamic-programming algorithm for the MNS problem is $\Theta(n^2)$.

EXERCISES

- Let $n = 5$, $p = [7, 3, 5, 2, 4]$, $w = [3, 1, 2, 1, 2]$, and $c = 6$. Draw a figure similar to Figure 19.1 that shows the recursive class made by Program 19.1. Label each node by the y value and shade nodes that represent the recomputation of a previously computed value.
- Use the data of Exercise 1 and arrive at a table similar to that of Figure 19.2. Use the iterative function of Program 19.3. Now trace back from $f(1, c)$ and determine the x_i values.

```
void mns(int *c, int numberOfNets, int **size)
{// Compute size[i][j] for all i and j.
    // initialize size[1][*]
    for (int j = 0; j < c[1]; j++)
        size[1][j] = 0;
    for (int j = c[1]; j <= numberOfNets; j++)
        size[1][j] = 1;

    // compute size[i][*], 1 < i < numberOfNets
    for (int i = 2; i < numberOfNets; i++)
    {
        for (int j = 0; j < c[i]; j++)
            size[i][j] = size[i - 1][j];
        for (int j = c[i]; j <= numberOfNets; j++)
            size[i][j] = max(size[i - 1][j], size[i - 1][c[i] - 1] + 1);
    }

    size[numberOfNets][numberOfNets] =
        max(size[numberOfNets - 1][numberOfNets],
            size[numberOfNets - 1][c[numberOfNets] - 1] + 1);
}
```

Program 19.11 Compute $\text{size}(i, j)$ for all i and j

3. Modify Program 19.1 so that it also computes the values of the x_i s that result in an optimal packing.
4. Write C++ code implementing the tuple method. Your code should determine the x_i values that define an optimal packing.
5. [0/1/2 Knapsack] Define the 0/1/2 knapsack problem to be

$$\text{maximize } \sum_{i=1}^n p_i x_i$$

subject to the constraints

$$\sum_{i=1}^n w_i x_i \leq c \text{ and } x_i \in \{0, 1, 2\}, \quad 1 \leq i \leq n$$

Let f be as defined for the 0/1 knapsack problem.

Hidden page

Hidden page

15. [Reflexive Transitive Closure] Modify Program 19.8 so that it starts with an adjacency matrix of a directed graph and computes its reflexive transitive closure matrix rtc . $\text{rtc}[i][j] = 1$ if there is a directed path from vertex i to vertex j that uses zero or more edges. $\text{rtc}[i][j] = 0$ otherwise. The complexity of your code should be $O(n^3)$ where n is the number of vertices in the graph.
16. Use Equations 19.5 through 19.7 to compute the *size* values for the case when $C = [4, 2, 6, 8, 9, 3, 1, 10, 7, 5]$. Present your results as is shown in Figure 19.10. Use these results to determine an MNS.
17. In Section 17.3.3 we saw that a project may be decomposed into several tasks and that these tasks may be performed in topological order. Let the tasks be numbered 1 through n so that task 1 is done first, then task 2 is done, and so on. Suppose that we have two ways to perform each task. Let $C_{i,1}$ be the cost of doing task i the first way, and let $C_{i,2}$ be the cost of doing it the second way. Let $T_{i,1}$ be the time it takes to do task i the first way, and let $T_{i,2}$ be the time when the task is done the second way. Assume that the T 's are integers. Develop a dynamic-programming algorithm to determine the least-cost way to complete the entire project in no more than t time. Assume that the cost of the project is the sum of the task costs and the total time is the sum of the task times. (*Hint:* Let $c(i,j)$ be the least cost with which tasks i through n can be completed in time j .) What is the complexity of your algorithm?
18. A machine has n components. For each component, there are three suppliers. The weight of component i from supplier j is $W_{i,j}$, and its cost is $C_{i,j}$, $1 \leq j \leq 3$. The cost of the machine is the sum of the component costs, and its weight is the sum of the component weights. Assume that all costs are positive integers. Write a dynamic-programming algorithm to determine from which supplier to buy each component so as to have the lightest machine with cost no more than c . Assume that the costs are integer. (*Hint:* Let $w(i,j)$ be the weight of the lightest machine composed of components i through n that costs no more than j .) What is the complexity of your algorithm?
19. Do Exercise 18 but this time define $w(i,j)$ to be the weight of the lightest machine, with components 1 through i , that costs no more than j .
20. [Longest Common Subsequence] String s is a subsequence of string a if s can be obtained from a by deleting some of the characters in a . The string “onion” is a subsequence of “recognition.” s is a common subsequence of a and b iff it is a subsequence of both a and b . The length of s is its number of characters. Develop a dynamic-programming algorithm to find a longest common subsequence of the strings a and b . (*Hint:* Let $a = a_1a_2 \dots a_n$ and $b = b_1b_2 \dots b_m$. Define $l(i,j)$ to be the length of a longest common subsequence of the strings $a_i \dots a_n$ and $b_j \dots b_m$.) What is the complexity of your algorithm?

21. Do Exercise 20 but this time define $l(i, j)$ to be the length of a longest common subsequence of the strings $a_1 \cdots a_i$ and $b_1 \cdots b_j$.
22. [Longest Sorted Subsequence] Write an algorithm to find a longest sorted subsequence (see Exercise 20) of a sequence of integers. What is the complexity of your algorithm?
23. Let x_1, x_2, \dots , be a sequence of integers. Let $\text{sum}(i, j) = \sum_i^j x_k$, $i \leq j$. Write an algorithm to find i and j for which $\text{sum}(i, j)$ is maximum. What is the complexity of your algorithm?
24. [String Editing] In the **string-editing problem**, you are given two strings $a = a_1 a_2 \cdots a_n$ and $b = b_1 b_2 \cdots b_m$ and three cost functions C , D , and I . $C(i, j)$ is the cost of changing a_i to b_j , $D(i)$ is the cost of deleting a_i from a , and $I(i)$ is the cost of inserting b_i into a . String a may be changed to string b by performing a sequence of change, delete, and insert operations. Such a sequence is called an **edit sequence**. For example, we could delete all the a_i s and then insert the b_i s, or when $n \geq m$, we could change a_i to b_i , $1 \leq i \leq n$, and then delete the remaining a_i s. The cost of a sequence of operations is the sum of the individual operation costs. Write a dynamic-programming algorithm to determine a least-cost edit sequence. (*Hint:* Define $c(i, j)$ to be the cost of a least-cost edit sequence that transforms $a_1 \cdots a_i$ into $b_1 \cdots b_j$.) What is the complexity of your algorithm?
25. Do Exercise 41 in Chapter 17 using dynamic programming.

19.3 REFERENCES AND SELECTED READINGS

An $O(n \log n)$ algorithm for the matrix multiplication chains problem may be found in the papers "Computation of Matrix Chain Products" parts I & II by T. Hu and M. Shing, *SIAM Journal on Computing*, 11, 1982, 362–372 and 13, 1984, 228–251.

The dynamic-programming algorithm for the noncrossing subset of nets problem is based on the work reported in the paper "Finding a Maximum Planar Subset of a Set of Nets in a Channel" by K. Supowit, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6, 1, 1987, 93–94.

Chapters 20 and 21 are available from the Web site for this book. The URL for this site is

<http://www.cise.ufl.edu/~sahni/dsaac>

Hidden page

INDEX

- abstract data type, 141
 - as an abstract class, 142
- array, 223
 - binary search tree, 533
 - binary tree, 439
 - dictionary, 366
 - graph, 625
 - linear list, 141
 - priority queue, 467
 - queue, 320
 - stack, 274
 - winner tree, 511
- Ackermann's function, 41, 460
- activity network, 673
- adjacency list
 - array, 629–631
 - linked, 629
- adjacency matrix, 627–629
 - cost, 632
- adjacencyDigraph, 634
- adjacencyGraph, 634
- adjacencyWDigraph, 634, 636
- adjacencyWGraph, 634
- algorithm
 - approximation, 494, 519–520, 666
 - nondeterministic, 494
 - polynomial time, 494
- algorithm design method
 - backtracking, 793–828
 - branch and bound, 829–858
 - divide and conquer, 704–756
 - dynamic programming, 757–792
 - greedy method, 660–703
- array
 - abstract data type, 223
 - array of arrays, 227
 - changing length, 145–146
 - indexing, 224
 - irregular, 228
 - subscript, 224
- array, 223
 - arrayList, 146
 - arrayQueue, 325
 - arrayStack, 278
- asymptotic notation, 95–120
 - big oh, 99–101, 104–107
 - little oh, 111
 - omega, 102–103, 107–109
 - properties, 111–112
 - theta, 102–103, 109–111
- AVL tree, 566–577
 - delete, 572–574
 - height, 567
 - indexed search tree, 566
 - insert, 567–572
 - representation, 567
 - search, 567
 - search tree, 566
- B'-tree, 614
- B*-tree, 614
- B-tree, 598–614
 - delete, 607–612
 - height, 603
 - insert, 604–607
 - node structure, 612
 - of order m , 602
 - search, 604
 - symmetric balanced, 614

- backtracking, 793–828
 - applications, [802–828](#)
 - bounding function, [800](#)
 - E-node, [796](#)
 - live node, [796](#)
 - the method, [795–802](#)
 - solution space, [795](#)
 - subset tree, [809](#)
- Bayer, R., [614](#)
- Beizer, B., [54](#)
- Bellman and Ford algorithm, [779](#)
- best fit, [519](#)
 - best fit decreasing, [519](#)
 - big oh notation, [99–101](#), [104–107](#)
 - bin packing, [518–528](#)
 - best fit, [519](#), [550–552](#)
 - best fit decreasing, [519](#)
 - first fit, [519](#)
 - first fit decreasing, [519](#)
 - next fit, [524](#)
 - bin sort, [197–202](#)
 - binary search, [116](#)
 - binary search tree, [531](#)
 - abstract data type, [533](#)
 - balanced, [566](#)
 - delete, [536–540](#)
 - with duplicates, [544–545](#)
 - height, [540](#)
 - indexed, [532](#), [545–547](#)
 - insert, [536](#)
 - search, [536](#)
 - binary tree, [425–443](#)
 - abstract data type, [439](#)
 - as array, [429–430](#)
 - AVL, [566](#)
 - balanced, [566](#)
 - complete, [427](#)
 - extended, [480](#)
 - external node, [480](#)
 - full, [427](#)
 - internal node, [480](#)
 - linked, [430–432](#)
 - operations, [432](#)
 - properties, [426–428](#)
 - red black, [577](#)
 - representation of a tree, [447–448](#)
 - search tree, [531](#)
 - skewed, [429](#)
 - splay tree, [592](#)
 - traversal, [432–438](#)
 - weighted external path length, [498](#)
 - `binarySearchTree`, [535](#)
 - `binarySearchTreeWithVisit`, [550](#)
 - BinaryTree*, [439](#)
 - `binaryTree`, [440](#)
 - `binaryTreeNode`, [431](#)
 - bipartite graph, [620](#)
 - cover, [678–682](#)
 - board-permutation problem
 - backtracking, [821–823](#)
 - branch and bound, [853–856](#)
 - bounding function, [800](#)
 - branch and bound, [829–858](#)
 - applications, [834–858](#)
 - FIFO, [830](#)
 - least cost, [830](#)
 - LIFO, [834](#)
 - max profit, [830](#)
 - the method, [830–834](#)
 - breadth-first search, [644–647](#)
 - BSTree*, [534](#)
 - `bsTree`, [535](#)
 - bubble sort, [71](#), [78](#)
 - C++
 - exceptions, [9–11](#)
 - function, [3–8](#)
 - review, [1–54](#)
 - STL, [43](#)
 - cache, [131–136](#)
 - L1, [131](#)
 - L2, [131](#)
 - chain, [172](#)
 - sorted, [367](#)
 - chain, [175](#), [176](#)
 - `chainIterator`, [183](#)
 - `chainNode`, [174](#)

- change making, 663–664
- characteristic roots, 749, Web
- Cho, S., 504
- circular list
 - doubly linked, 195
 - singly linked, 192–195
- CircularWithHeader**, 194
- class
 - equivalence, 210
- clause coverage, 51
- closest pair of points, 737–745
- Cohoon, J., 54
- column-major mapping, 225
- completeWinnerTree**, 514
- complexity
 - amortized, 595–597, Web
 - asymptotic, 95–120
 - lower bound, 751
 - operation count, 67–79
 - practical, 117–119
 - space, 57–65
 - step counts, 80–89
 - time, 57, 66–94
 - upper bound, 751
- compression
 - Huffman code, 497–500
 - LZW method, 402–416
- connected component, 652–655
- connected graph, 652–655
- convex hull, 205–209
- Cormen, T., 120
- count sort, 69
- counterfeit coin, 705
- critical activity, 699
- crossing distribution, 553–562
- Dale, N., 169
- data
 - atomic, 139
 - object, 139
 - primitive, 139
 - structure, 139
- data space, 58, 59
- data structure, 139
- array, 223
- AVL tree, 566
- B-tree, 598
- binary search tree, 531
- binary tree, 425
- dictionary, 364
- double ended priority queue, Web
- hash table, 381
- implicit, 479
- interval heap, Web
- leftist tree, 481
- linear list, 140
- loser tree, 515
- matrix, 230
- max heap, 469
- max tree, 469
- min heap, 469
- min tree, 469
- priority queue, 466
- queue, 319
- red-black tree, 577
- skip list, 372
- splay tree, 592
- stack, 271
- suffix tree, Web
- tournament tree, 506
- tree, 423
- tries, Web
- winner tree, 507
- Davidson, J., 54
- dBinarySearchTree**, 544
- debugging, 53–54
- decision coverage, 51
- decision problem, 494
- decision tree, 754
- defective chessboard, 715–719
- Demers, A., 528
- depth-first search, 647–651
- derivedArrayStack**, 276
- derivedLinkedStack**, 282
- Dictionary**, 366
- dictionary, 364
 - abstract data type, 366

- as B-tree, 598–614
- with duplicates, 364
- as hash table, 381–402
- as linear list, 367–368
- as red-black tree, 577–592
- as skip list, 368–380
- as splay tree, 592–598
- as AVL tree, 566–577
- dictionary**, 367
- Dietel, H., 54
- Dietel, P., 54
- Dijkstra, E., 682
- Dijkstra's algorithm, 682–686
- divide and conquer, 704–756, Web
 - applications, 715–749
 - lower bounds, 751–756
 - the method, 705–715
 - nonrecursive, 711
 - recurrence equation, 749–751
- double ended priority queue
 - generic methods, Web
 - interval heap, Web
- doubly linked list, 195–197
- doublyLinkedList**, 195
- dynamic programming, 757–792, Web
 - applications, 761–792
 - decision sequence, 758
 - decision subsequence, 758
 - the method, 758–760
 - principle of optimality, 760
 - problem state, 758
 - recurrence equation, 759
 - traceback, 760
- E-node, 796
- edge
 - directed, 616
 - incident from vertex, 616
 - incident to vertex, 616
 - loop, 617
 - self, 617
 - undirected, 616
- environment stack, 59, 61
- equivalence
 - class, 210
 - offline, 210, 297–300
 - online, 210
 - relation, 209
- event list, 466
- exception
 - handling, 10–11
 - throwing, 9–10
- execution path coverage, 52
- extendedChain**, 184
- extendedLinearList**, 184
- factorial, 35, 37, 41, 65
- Fibonacci number, 36, 41
- find kth smallest, 733–736
- first fit, 519
- first fit decreasing, 519
- Floyd's algorithm, 773
- function
 - overloading, 8
 - signature, 7
- Garey, M., 504, 528
- generating functions, 749, Web
- Goodrich, M., 169
- Graham, R., 495, 528
- Graph**, 625
- graph, 615–659
 - abstract data type, 625
 - adjacency list, 629–631
 - adjacency matrix, 627–629
 - arc, 616
 - bipartite, 620
 - breadth-first spanning tree, 655
 - clique, 700
 - coloring, 700
 - compatibility, 816
 - complement, 815
 - complete, 622
 - component labeling, 654
 - connected, 618, 652–655
 - connected component, 652–655
 - cover, 703
 - cycle, 619

- degree, 622
- in degree, 623
- out degree, 623
- depth-first spanning tree, 656
- digraph, 617
- directed, 617
- edge, 616
- independent set, 700, 814
- line, 616
- longest path, 790
- max clique, 700, 812–816, 844–848
- network, 617
- node, 616
- path, 618
- path finding, 652
- path length, 618
- point, 616
- properties, 622–624
- reflexive transitive closure, 791
- search methods, 644–652
- shortest path, 665–666, 682–686, 758, 773–784
- simple max cut, 827
- simple path, 618
- spanning tree, 618–619, 687–694
- strongly connected, 624
- transitive closure, 659
- traveling salesperson, 798–800, 817–821, 832–833, 848–853
- undirected, 617
- vertex, 616
- vertex cover, 827
- weighted, 617
- graph**, 626
- graphChain**, 640
- Gray code, 42
- greatest common divisor, 42
- greedy method, 660–703
 - applications, 668–703
 - greedy criterion, 663
 - the method, 663–668
- Guibas, L., 614
- Hamming distance, 42
- hash function
 - digit analysis, Web
 - division, 383
 - folding, Web
 - good, 385
 - middle of square, Web
 - multiplication, Web
 - uniform, 384
- hash table
 - bucket, 382
 - chained, 394–398
 - home bucket, 382
 - load factor, 390
- hashing, 381–402
 - with chains, 394–398
 - collision, 383
 - double hashing, Web
 - functions, 382–386
 - hash functions, Web
 - ideal, 381–382
 - linear open addressing, 387
 - linear probing, 387–393
 - load factor, 390
 - overflow, 383
 - performance, 390–393, 395–397
 - quadratic probing, Web
 - random probing, 391, Web
- HashTable**, 389
- header node, 192
- heap
 - delete, 471–472
 - initialize, 472–474
 - insert, 470–471
 - max, 469
 - min, 469
- heap sort, 491
 - performance, 731–733
- Hennessey, J., 136
- heuristic, 666
 - bounded performance, 666
- Hirschberg, D., 417
- histogramming, 548–550

- using an array, 548
- using a binary search tree, 549–550
- Horner's rule, 68
- Horowitz, E., 120, 169, 504, 509, 614
- Hsu, C., 316
- Hu, T., 792
- Huffman code, 497–500
- Huffman tree, 498
- image component labeling, 341–342
- image compression, Web
- independent set, 814
 - greedy solution, 700
- `indexedBinarySearchTree`, 545
- `IndexedBSTree`, 534
- `indexedBSTree`, 535
- induction, 36–37, 749, Web
 - base, 36
 - hypothesis, 36
 - step, 36
- inorder traversal, 432
- insertion sort, 78, 117, 123, 125, 126
 - performance, 731–733
- instruction space, 58, 59
- interval heap, Web
- inversion number, 756
- ISAM, 598
- iterator, 153
- Johnson, D., 504, 528
- knapsack problem
 - 0/1/2, 788
 - approximation algorithms, 671–673
 - backtracking, 797, 809–812
 - branch and bound, 831–832, 842–844
 - continuous, 697
 - decision sequence, 758
 - dynamic programming, 761–766
 - greedy solution, 669–673
 - heuristics, 671–673
- recurrence equation, 759
- solution space, 795
- two dimensional, 789
- Knuth, D., 416, 463, 528, 614
- Kruskal's algorithm, 687–693
- Lee's algorithm, 317
- Leeuwen, J., 461, 463
- leftist tree
 - delete, 482
 - height biased, 481
 - initialize, 485
 - insert, 482
 - meld, 482–485
 - weight biased, 481
- Leiserson, C., 120
- Lelewer, D., 417
- level order traversal, 432
- linear list, 140
 - abstract class, 142
 - abstract data type, 141
 - as array, 143–161
 - as chain, 172–192
 - as circular list, 192–195
 - as doubly linked list, 195–197
 - empty, 140
 - front, 140
 - as indexed AVL tree, 576
 - as indexed binary tree, 546
 - as indexed red-black tree, 592
 - iterator, 153
 - length, 140
 - as linked list, 172–192
 - multiple, 163–166
 - performance, 166–169, 183–187
 - size, 140
 - as vector, 161–162
- `LinearList`, 141
- `linearList`, 142
- link, 172
- linked list
 - doubly, 195
 - singly, 172
- `linkedBinaryTree`, 441

- linkedDigraph**, 634
linkedGraph, 634
linkedMatrix, 261
linkedQueue, 330
LinkedStack, 283
linkedWDigraph, 634
linkedWGraph, 634
little oh notation, 111
live node, 796
loading problem, 662
 - backtracking, 802–809
 - branch and bound, 834–842
 - greedy solution, 668–669
longest common subsequence, 791
longest sorted subsequence, 792
loser tree, 515–518
lower bound proof, 751–756
LZW compression, 402–416

machine design, 827
machine scheduling, 493–496, 664–665
machine shop simulation, 344–358, 466
matrix, 230
 - add, 90, 232
 - antidiagonal, 250
 - dense, 252
 - diagonal, 239, 242
 - lower triangular, 239
 - multiplication chain, 766–773
 - multiply, 90, 133–134, 232, 707–711
 - product, 232
 - sparse, 252–268
 - square, 239
 - square band, 250
 - symmetric, 240, 246
 - Toeplitz, 250
 - transpose, 84, 115, 232
 - triangular, 245
 - tridiagonal, 239, 244
 - upper triangular, 239**matrix**, 234
max clique
 - backtracking, 812–816
 - branch and bound, 844–848
 - greedy solution, 700
max element, 51, 67
max tree, 469
maxHblt, 486
maxHeap, 474
maxPriorityQueue, 467
maxPriorityQueue, 468
McCreight, E., 614
median of medians, 736
median-of-three rule, 730
Mehta, D., 169, 504, 509, 614
merge
 - k*-way, 510
 - two way, 723
merge sort, 719–726
 - natural, 723–726
 - performance, 731–733
 - straight, 723
 - two way, 722
method
 - recursive, 37–40
min and max, 91
 - divide and conquer, 706–707, 711–714
 - lower bound, 752–754
min tree, 469
Moret, B., 703
Mount, D., 169
Myers, G., 54

n-queens problem, 828
natural merge sort, 723–726
net, 212
network, 617
 - activity on edge (AOE), 697
 - activity on vertex (AOV), 673, 697
 - design, 828
next fit, 524
noncrossing nets, 784–787
NP-complete problem, 494
NP-hard problem, 494, 671, 679, 700, 802, 809, 815, 822

- omega notation, 102–103, 107–109
- operation count, 67–79
 - average, 73
 - best, 73
 - worst, 73
- optimization
 - constraint, 662
 - function, 662
 - problem, 662–663
- ordered list, 140
- Paik, D., 463
- pairing heap, Web
- parameter
 - actual, 4
 - formal, 3
- parenthesis matching, 284–285
- partition problem, 802
- Paschos, V., 703
- path compression
 - compaction, 457
 - halving, 458
 - splitting, 458
- path in a graph, 618
 - all pairs, 773–776
 - finding, 652
 - length, 618
 - negative edge cost, 776–784
 - shortest, 665–666, 758
 - simple, 618
 - single source, 682–686
- Patterson, D., 136
- performance, 57
 - analysis, 55–94
 - measurement, 57, 121–136
- permutation tree, 800
- permutations, 38, 65, 115
- Pinter, R., 316
- pointer, 172
- polygon, 205, 209
- polynomial
 - evaluation, 67
 - Horner's rule, 68
 - as linked list, 218
- postorder traversal, 432
- prefix sums, 85, 115
- preorder traversal, 432
- Prim's algorithm, 693
- principle of optimality, 760
- priority queue, 464–504
 - abstract data type, 467
 - double ended, 842, Web
 - as heap, 469–479
 - as leftist tree, 479–490
 - as linear list, 468–469
 - pairing heap, Web
- profiling, 83
- program step, 81
- Pugh, W., 416
- quadratic roots, 48
- Queue*, 321
- queue, 319
 - abstract data type, 320
 - as array, 322–329
 - as chain, 330–333
 - linked, 330–333
- queue*, 321
- quick sort, 726–733
 - median-of-three rule, 730
 - performance, 731–733
- radix sort, 202–205
- railroad cars
 - using queues, 333–336
 - using stacks, 289–294
- Rajasekeran, S., 120, 504
- rank sort, 69, 75
- rat in a maze, 300–313, 795, 796
 - branch and bound, 830
- Rawlins, G., 120
- recurrence equation, 83, 749–751, Web
 - characteristic roots, 749, Web
 - dynamic programming, 759
 - generating functions, 749, Web
 - induction, 749, Web
 - substitution method, 749
 - table look-up, 749–751

- recursion, 33–43
 stack, 272
 stack space, 62
- red-black tree, 577–592
 delete, 584–588
 height, 578
 insert, 579–584
 representation, 579
 search, 579
- Reddy, S., 463
- reflexive transitive closure, 791
- relation
 equivalence, 209
 reflexive, 209
 symmetric, 209
 transitive, 209
- Rivest, R., 120
- rotation
 double, 572
 single, 572
- row-major mapping, 225
- run, 508
- Sahni, S., 120, 169, 463, 504, 509, 614
- schedule, 211, 493
 finish time, 493
 length, 493
 longest processing time (LPT), 494
- search
 binary, 116
 sequential, 63, 73, 86, 91, 115
 successful, 73
 unsuccessful, 73
- Sedgewick, R., 614
- selection sort, 70, 76
- selection tree, 506
- shaker sort, Web
- Shapiro, H., 703
- Shell sort, Web
- Sherwani, N., 361
- Shing, M., 792
- signal boosters, 443–448, 828
- simple max cut, 827
- skip list, 368–380
- Sollin's algorithm, 693–694
- sortedArrayList, 367
- sortedChain, 368
- sorting
 bin sort, 197–202
 bubble sort, 71, 78
 count sort, 69
 external sort, 508
 heap sort, 491
 insertion sort, 78, 117, 123, 125, 126
 internal sort, 508
 lower bound, 754–755
 merge sort, 719–726
 natural merge sort, 723–726
 performance, 731–733
 quick sort, 726–733
 radix sort, 202–205
 rank sort, 69, 75
 selection sort, 70, 76
 shaker sort, Web
 Shell sort, Web
 stable, 202
 topological sort, 673–676
- spanning tree, 618–619
 breadth first, 655
 depth first, 656
 Kruskal's algorithm, 687–693
 minimum cost, 687–694
 Prim's algorithm, 693, 702
 Sollin's algorithm, 693–694, 702
- sparse matrix
 as many linear lists, 258–262
 as single linear list, 252–258
- sparseMatrix, 254
- splay tree, 592–598
 splay operation, 593–595
- stable sort, 202
- stack, 271
 abstract data type, 274
 as array, 274–281
 as chain, 282–284
 folding, 240, Web

- linked, 282–284
- performance, 277–280, 282
- stack**, 274
- stack**, 275
- statement coverage, 51
- step**, 80
- step count**, 80–89
 - profiling, 83
 - program step, 81
 - step**, 80
 - steps per execution, 83
- STL**, 43
- Strassen's matrix multiply, 708
- string-editing problem, 792
- subscript, 224
- subset tree, 800
- subsets, 42
- substitution method, 83, 749
- suffix tree, Web
- sum-of-subset problem, 802
- Supowit, K., 792
- switch box routing, 294–297
- Tamassia, R., 169
- Tarjan, R., 461, 463, 504
- testing, 47–53, Web
 - black box, 50
 - clause coverage, 51
 - decision coverage, 51
 - execution path coverage, 52
 - incremental, 54
 - statement coverage, 51
 - white box, 51
- theta notation, 102–103, 109–111
- thirsty baby problem, 662
- topological
 - order, 673
 - sequence, 673
 - sorting, 673–676
- tournament tree
 - loser tree, 515
 - winner tree, 507
- Tower of Brahma, 285
- Towers of HaHa, 313
- Towers of Hanoi, 285–288
- transitive closure, 659
- traveling salesperson
 - backtracking, 798–800, 817–821
 - branch and bound, 832–833, 848–853
- tree**, 420–425
 - ancestor, 423
 - AVL, 566
 - B-tree, 598
 - balanced, 566
 - binary, 425–443
 - binary search tree, 531
 - as binary tree, 447–448
 - child, 423
 - decision, 754
 - degree, 424
 - depth, 424
 - descendent, 423
 - grandchild, 423
 - grandparent, 423
 - height, 424
 - Huffman, 498
 - leaf, 423
 - loser, 515–518
 - m-way search tree, 599
 - max, 469
 - min, 469
 - parent, 423
 - permutation, 800
 - red black, 577
 - root, 423
 - selection, 506
 - sibling, 423
 - splay tree, 592
 - subset, 800, 809
 - subtree, 423
 - tournament, 506
 - winner, 507
- tries, Web
- triomino, 715
- Ullman, J., 528
- union-find problem, 210

using arrays, 213
using a chain, 214–217
height rule, 454
path compression, 456
using trees, 450–461
weight rule, 454

vector, 161
vectorList, 162
vertex
 adjacent, 616
 adjacent from vertex, 616
 adjacent to vertex, 616
 cover, 827
 degree, 622
 in degree, 623
 out degree, 623

Weiss, M., 169
Welch, T., 417
winner tree, 507
 abstract data type, 511
 initialize, 513
 replay match, 514
WinnerTree, 512
winnerTree, 512
wire routing, 336–338

Yao, A., 528

Hidden page

04/8/11/09.2020

Data Structures, Algorithms, and Applications in C++ (2/E) is the new version of the very popular first edition. It provides a comprehensive coverage of fundamental data structures, making it ideal for use in computer science courses. The author has made the book very user friendly by starting with a gentle introduction, providing intuitive discussions, and including real-world applications. This new edition makes significant use of the Standard Templates Library (STL) and relates the data structures and algorithms developed in the text to corresponding implementations in the STL. Many new examples and exercises also have been included.

Real-world applications are a unique feature of this text. The author provides several applications for each data structure and algorithm design method discussed, taking examples from topics such as sorting, compression and coding, and image processing. These applications motivate and interest students by connecting concepts with their use. Dr Sahni does an excellent job of balancing theoretical and practical information, resulting in learned concepts and interested students.

The market-developed pedagogy in this book reinforces concepts and gives students plenty of practice. There are almost 1,000 exercises, including comprehension and simple programming problems, and projects. Additionally, the book has an associated web site that contains all the programs in the book, sample data, generated output, solutions to selected exercises, and sample tests with answers.

Sartaj Sahni is a Distinguished Professor and Chair of Computer and Information Sciences and Engineering at the University of Florida. He is a member of the European Academy of Sciences, a Fellow of IEEE, ACM, AAAS, and Minnesota Supercomputer Institute, and a Distinguished Alumnus of the IIT, Kanpur. Dr Sahni is the recipient of the 1997 IEEE Computer Society Taylor L Booth Education Award, the 2003 IEEE Computer Society W.Wallace McDowell Award and the 2003 ACM Karl Karlstrom Outstanding Educator Award.

Dr Sahni received his B.Tech. (EE) degree from the IIT, Kanpur, and MS and PhD degrees in Computer Science from Cornell University. He has published over 250 research papers and written 15 texts. His research publications are on the design and analysis of efficient algorithms, parallel computing, interconnection networks, design automation, and medical algorithms.

For sale in India, Pakistan, Nepal, Myanmar, Bhutan, Bangladesh and Sri Lanka only.
Not for export to other countries.

Orient Longman

 Rs350



Universities Press

Sartaj Sahni: *Data Structures, Algorithms, and Applications in C++ (2/E)*

ISBN 81 7371 522 X



9 788173 715228