Data Structures (Into Java)
(Fourth Edition)

Paul N. Hilfinger
University of California, Berkeley

Contents

Chapter 1

Algorithmic Complexity

The obvious way to answer to the question "How fast does such-and-such a program run?" is to use something like the UNIX `time` command to find out directly. There are various possible objections to this easy answer. The time required by a program is a function of the input, so presumably we have to time several instances of the command and extrapolate the result. Some programs, however, behave fine for *most* inputs, but sometimes take a very long time; how do we report (indeed, how can we be sure to notice) such anomalies? What do we do about all the inputs for which we have no measurements? How do we validly apply results gathered on one machine to another machine?

The trouble with measuring raw time is that the information is precise, but limited: the time for *this* input on *this* configuration of *this* machine. On a different machine whose instructions take different absolute or relative times, the numbers don't necessarily apply. Indeed, suppose we compare two different programs for doing the same thing on the same inputs and the same machine. Program A may turn out faster than program B. This does *not* imply, however, that program A will be faster than B when they are run on some other input, or on the same input, but some other machine.

In mathematese, we might say that a raw time is the value of a function $C_r(I, P, M)$ for some particular input I, some program P, and some "platform" M (*platform* here is a catchall term for a combination of machine, operating system, compiler, and runtime library support). I've invented the function C_r here to mean "the raw cost of...." We can make the figure a little more informative by summarizing over *all* inputs of a particular size

$$C_w(N, P, M) = \max_{|I|=N} C_r(I, P, M),$$

where $|I|$ denotes the "size" of input I. How one defines the size depends on the problem: if I is an array to be sorted, for example, $|I|$ might denote I.`length`. We say that C_w measures *worst-case time* of a program. Of course, since the number of inputs of a given size could be very large (the number of arrays of 5 `int`s, for example, is $2^{160} > 10^{48}$), we can't directly measure C_w, but we can perhaps estimate it with the help of some analysis of P. By knowing worst-case times, we can make

conservative statements about the running time of a program: if the worst-case time for input of size N is T, then we are guaranteed that P will consume no more than time T for *any* input of size N.

But of course, it always possible that our program will work fine on most inputs, but take a really long time on one or two (unlikely) inputs. In such cases, we might claim that C_w is too harsh a summary measure, and we should really look at an *average* time. Assuming all values of the input, I, are equally likely, the average time is

$$C_a(N, P, M) = \frac{\sum_{|I|=N} C_r(I, P, M)}{N}$$

Fair this may be, but it is usually very hard to compute. In this course, therefore, I will say very little about average cases, leaving that to your next course on algorithms.

We've summarized over inputs by considering worst-case times; now let's consider how we can summarize over machines. Just as summarizing over inputs required that we give up some information—namely, performance on particular inputs—so summarizing over machines requires that we give up information on precise performance on particular machines. Suppose that two different models of computer are running (different translations of) the same program, performing the same steps in the same order. Although they run at different speeds, and possibly execute different numbers of instructions, the speeds at which they perform any particular step tend to differ by some constant factor. By taking the largest and smallest of these constant factors, we can put bounds around the difference in their overall execution times. (The argument is not really this simple, but for our purposes here, it will suffice.) That is, the timings of the same program on any two platforms will tend to differ by no more than some constant factor over all possible inputs. If we can nail down the timing of a program on one platform, we can use it for all others, and our results will "only be off by a constant factor."

But of course, 1000 is a constant factor, and you would not normally be insensitive to the fact that Brand X program is 1000 times slower than Brand Y. There is, however, an important case in which this sort of characterization is useful: namely, when we are trying to determine or compare the performance of *algorithms*—idealized procedures for performing some task. The distinction between algorithm and program (a concrete, executable procedure) is somewhat vague. Most higher-level programming languages allow one to write programs that look very much like the algorithms they are supposed to implement. The distinction lies in the level of detail. A procedure that is cast in terms of operations on "sets," with no specific implementation given for these sets, probably qualifies as an algorithm. When talking about idealized procedures, it doesn't make a great deal of sense to talk about the number of seconds they take to execute. Rather, we are interested in what I might call the *shape* of an algorithm's behavior: such questions as "If we double the size of the input, what happens to the execution time?" Given that kind of question, the particular *units* of time (or space) used to measure the performance of an algorithm are unimportant—constant factors don't matter.

If we only care about characterizing the speed of an algorithm to within a constant factor, other simplifications are possible. We need no longer worry about the timing of each little statement in the algorithm, but can measure time using any convenient "marker step." For example, to do decimal multiplication in the standard way, you multiply each digit of the multiplicand by each digit of the multiplier, and perform roughly one one-digit addition with carry for each of these one-digit multiplications. Counting just the one-digit multiplications, therefore, will give you the time within a constant factor, and these multiplications are very easy to count (the product of the numbers of digits in the operands).

Another characteristic assumption in the study of *algorithmic complexity* (i.e., the time or memory consumption of an algorithm) is that we are interested in *typical* behavior of an idealized program over the entire set of possible inputs. Idealized programs, of course, being ideal, can operate on inputs of any possible size, and most "possible sizes" in the ideal world of mathematics are extremely large. Therefore, in this kind of analysis, it is traditional not to be interested in the fact that a particular algorithm does very well for small inputs, but rather to consider its behavior "in the limit" as input gets very large. For example, suppose that one wanted to analyze algorithms for computing π to any given number of decimal places. I can make *any* algorithm look good for inputs up to, say, 1,000,000 by simply storing the first 1,000,000 digits of π in an array and using that to supply the answer when 1,000,000 or fewer digits are requested. If you paid any attention to how my program performed for inputs up to 1,000,000, you could be seriously misled as to the cleverness of my algorithm. Therefore, when studying algorithms, we look at their *asymptotic behavior*—how they behave as they input size goes to infinity.

The result of all these considerations is that in considering the time complexity of algorithms, we may choose any particular machine and count any convenient marker step, and we try to find characterizations that are true asymptotically—out to infinity. This implies that our typical complexity measure for algorithms will have the form $C_w(N, A)$—meaning "the worst-case time over all inputs of size N of algorithm A (in some units)." Since the algorithm will be understood in any particular discussion, we will usually just write $C_w(N)$ or something similar. So the first thing we need to describe algorithmic complexity is a way to characterize the asymptotic behavior of functions.

1.1 Asymptotic complexity analysis and order notation

As it happens, there is a convenient notational tool—known collectively as *order notation* for "order of growth"—for describing the asymptotic behavior of functions. It may be (and is) used for any kind of integer- or real-valued function—not just complexity functions. You've probably seen it used in calculus courses, for example.

We write

$$f(n) \in O(g(n))$$

(aloud, this is "$f(n)$ is in big-Oh of $g(n)$") to mean that the function f is eventually
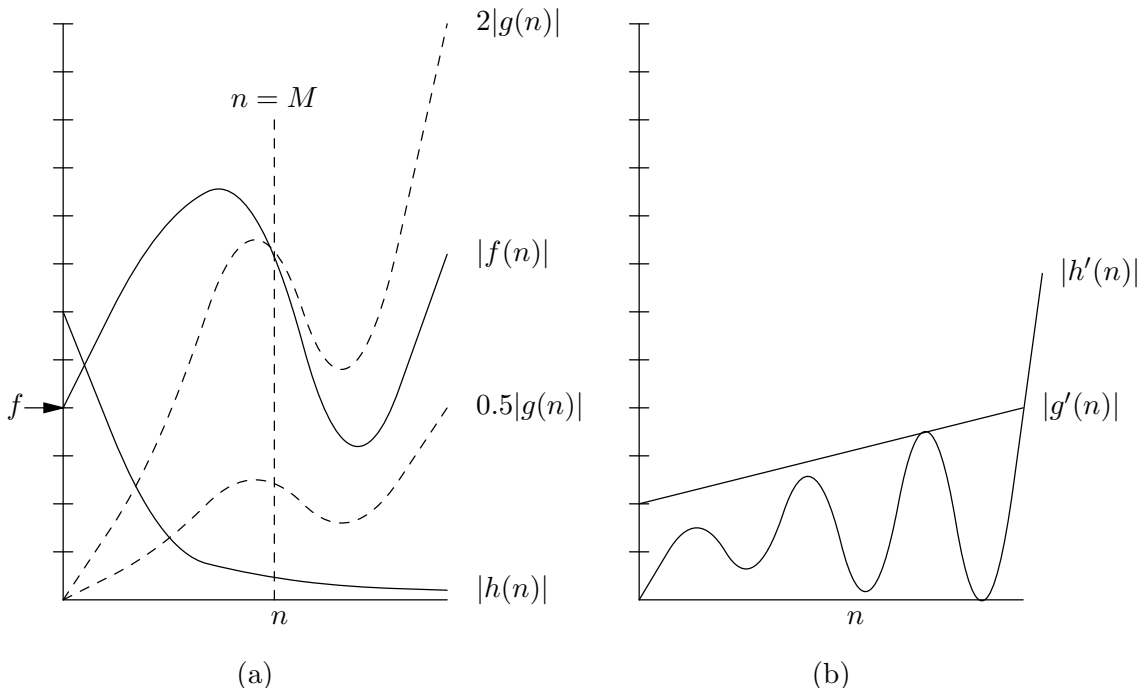
Figure 1.1: Illustration of big-Oh notation. In graph (a), we see that $|f(n)| \leq 2|g(n)|$ for $n > M$, so that $f(n) \in O(g(n))$ (with $K = 2$). Likewise, $h(n) \in O(g(n))$, illustrating the g can be a very over-cautious bound. The function f is also bounded *below* by both g (with, for example, $K = 0.5$ and M any value larger than 0) and by h. That is, $f(n) \in \Omega(g(n))$ and $f(n) \in \Omega(h(n))$. Because f is bounded above and below by multiples of g, we say $f(n) \in \Theta(g(n))$. On the other hand, $h(n) \notin \Omega(g(n))$. In fact, assuming that g continues to grow as shown and h to shrink, $h(n) \in o(g(n))$. Graph (b) shows that $o(\cdot)$ is not simply the set complement of $\Omega(\cdot)$; $h'(n) \notin \Omega(g'(n))$, but $h'(n) \notin o(g'(n))$, either.

bounded by some multiple of $|g(n)|$. More precisely, $f(n) \in O(g(n))$ iff

$$|f(n)| \leq K \cdot |g(n)|, \text{ for all } n > M,$$

for some constants $K > 0$ and M. That is, $O(g(n))$ is the *set* of functions that "grow no more quickly than" $|g(n)|$ does as n gets sufficiently large. Somewhat confusingly, $f(n)$ here does not mean "the result of applying f to n," as it usually does. Rather, it is to be interpreted as the *body of a function* whose parameter is n. Thus, we often write things like $O(n^2)$ to mean "the set of all functions that grow no more quickly than the square of their argument[1]." Figure 1.1a gives an intuitive idea of what it means to be in $O(g(n))$.

Saying that $f(n) \in O(g(n))$ gives us only an *upper bound* on the behavior of f. For example, the function h in Figure 1.1a—and for that matter, the function that

---

[1]If we wanted to be formally correct, we'd use lambda notation to represent functions (such as Scheme uses) and write instead $O(\lambda n. \, n^2)$, but I'm sure you can see how such a degree of rigor would become tedious very soon.

is 0 everywhere—are both in $O(g(n))$, but certainly don't grow like g. Accordingly, we define $f(n) \in \Omega(g(n))$ iff for all $n > M$, $|f(n)| \geq K|g(n)|$ for $n > M$, for some constants $K > 0$ and M. That is, $\Omega(g(n))$ is the set of all functions that "grow *at least* as fast as" g beyond some point. A little algebra suffices to show the relationship between $O(\cdot)$ and $\Omega(\cdot)$:

$$|f(n)| \geq K|g(n)| \equiv |g(n)| \leq (1/K) \cdot |f(n)|$$

so

$$f(n) \in \Omega(g(n)) \Longleftrightarrow g(n) \in O(f(n))$$

Because of our cavalier treatment of constant factors, it is possible for a function $f(n)$ to be bounded both above and below by another function $g(n)$: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. For brevity, we write $f(n) \in \Theta(g(n))$, so that $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Just because we know that $f(n) \in O(g(n))$, we don't necessarily know that $f(n)$ gets much smaller than $g(n)$, or even (as illustrated in Figure 1.1a) that it is ever smaller than $g(n)$. We occasionally do want to say something like "$h(n)$ *becomes negligible* compared to $g(n)$." You sometimes see the notation $h(n) \ll g(n)$, meaning "$h(n)$ is much smaller than $g(n)$," but this could apply to a situation where $h(n) = 0.001g(n)$. Not being interested in mere constant factors like this, we need something stronger. A traditional notation is "little-oh," defined as follows.

$$h(n) \in o(g(n)) \Longleftrightarrow \lim_{n \to \infty} h(n)/g(n) = 0.$$

It's easy to see that if $h(n) \in o(g(n))$, then $h(n) \notin \Omega(g(n))$; no constant K can work in the definition of $\Omega(\cdot)$. It is not the case, however, that all functions that are *outside* of $\Omega(g(n))$ must be in $o(g(n))$, as illustrated in Figure 1.1b.

1.2 Examples

You may have seen the big-Oh notation already in calculus courses. For example, Taylor's theorem tells us[2] that (under appropriate conditions)

$$f(x) = \underbrace{\frac{x^n}{n!} f^{[n]}(y)}_{\text{error term}} + \underbrace{\sum_{0 \leq k < n} f^{[k]}(0) \frac{x^k}{k!}}_{\text{approximation}}$$

for some y between 0 and x, where $f^{[k]}$ represents the kth derivative of f. Therefore, if $g(x)$ represents the maximum absolute value of $f^{[n]}$ between 0 and x, then we could also write the error term as

$$f(x) - \sum_{0 \leq k < n} f^{[k]}(0) \frac{x^k}{k!}$$
$$\in \quad O(\frac{x^n}{n!} g(x)) = O(x^n g(x))$$

---

[2] Yes, I know it's a Maclaurin series here, but it's still Taylor's theorem.

$f(n)$	*Is contained in*	*Is* not *contained in*
1, $1 + 1/n$	$O(10000)$, $O(\sqrt{n})$, $O(n)$, $O(n^2)$, $O(\lg n)$, $O(1 - 1/n)$ $\Omega(1)$, $\Omega(1/n)$, $\Omega(1 - 1/n)$ $\Theta(1)$, $\Theta(1 - 1/n)$ $o(n)$, $o(\sqrt{n})$, $o(n^2)$	$O(1/n)$, $O(e^{-n})$ $\Omega(n)$, $\Omega(\sqrt{n})$, $\Omega(\lg n)$, $\Omega(n^2)$ $\Theta(n)$, $\Theta(n^2)$, $\Theta(\lg n)$, $\Theta(\sqrt{n})$ $o(100 + e^{-n})$, $o(1)$
$\log_k n$, $\lfloor \log_k n \rfloor$, $\lceil \log_k n \rceil$	$O(n)$, $O(n^\epsilon)$, $O(\sqrt{n})$, $O(\log_{k'} n)$ $O(\lfloor \log_{k'} n \rfloor)$, $O(n/\log_{k'} n)$ $\Omega(1), \Omega(\log_{k'} n), \Omega(\lfloor \log_{k'} n \rfloor)$ $\Theta(\log_{k'} n)$, $\Theta(\lfloor \log_{k'} n \rfloor)$, $\Theta(\log_{k'} n + 1000)$ $o(n)$, $o(n^\epsilon)$	$O(1)$ $\Omega(n^\epsilon), \Omega(\sqrt{n})$ $\Theta(\log_{k'}^2 n), \Theta(\log_{k'} n + n)$
n, $100n + 15$	$O(.0005n - 1000)$, $O(n^2)$, $O(n \lg n)$ $\Omega(50n + 1000)$, $\Omega(\sqrt{n})$, $\Omega(n + \lg n)$, $\Omega(1/n)$ $\Theta(50n + 100)$, $\Theta(n + \lg n)$ $o(n^3)$, $o(n \lg n)$	$O(10000)$, $O(\lg n)$, $O(n - n^2/10000)$, $O(\sqrt{n})$ $\Omega(n^2)$, $\Omega(n \lg n)$ $\Theta(n^2)$, $\Theta(1)$ $o(1000n)$, $o(n^2 \sin n)$
n^2, $10n^2 + n$	$O(n^2 + 2n + 12)$, $O(n^3)$, $O(n^2 + \sqrt{n})$ $\Omega(n^2 + 2n + 12)$, $\Omega(n)$, $\Omega(1)$, $\Omega(n \lg n)$ $\Theta(n^2 + 2n + 12)$, $\Theta(n^2 + \lg n)$	$O(n)$, $O(n \lg n)$, $O(1)$ $o(50n^2 + 1000)$ $\Omega(n^3)$, $\Omega(n^2 \lg n)$ $\Theta(n)$, $\Theta(n \cdot \sin n)$
n^p	$O(p^n)$, $O(n^p + 1000n^{p-1})$ $\Omega(n^{p-\epsilon})$, $\Theta(n^p + n^{p-\epsilon})$ $o(p^n)$, $o(n!)$, $o(n^{p+\epsilon})$	$O(n^{p-1})$, $O(1)$ $\Omega(n^{p+\epsilon})$, $\Omega(p^n)$ $\Theta(n^{p+\epsilon})$, $\Theta(1)$ $o(p^n + n^p)$
2^n, $2^n + n^p$	$O(n!)$, $O(2^n - n^p)$, $O(3^n)$, $O(2^{n+p})$ $\Omega(n^p)$, $\Omega((2 - \delta)^n)$, $\Theta(2^n + n^p)$ $o(n2^n)$, $o(n!)$, $o(2^{n+\epsilon})$, $o((2 + \epsilon)^n)$	$O(n^p)$, $O((2 - \delta)^n)$ $\Omega((2 + \epsilon)^n)$, $\Omega(n!)$ $\Theta(2^{2n})$

Table 1.1: Some examples of order relations. In the above, $\epsilon > 0$, $0 \le \delta \le 1$, $p > 1$, and $k, k' > 1$.

for fixed n. This is, of course, a much weaker statement than the original (it allows the error to be much bigger than it really is).

You'll often seen statements like this written with a little algebraic manipulation:

$$f(x) \in \sum_{0 \le k < n} f^{[k]}(0)\frac{x^k}{k!} + O(x^n g(x)).$$

To make sense of this sort of statement, we define addition (and so on) between functions (a, b, etc.) and sets of functions (A, B, etc.):

$$
\begin{aligned}
a + b &= \lambda x.a(x) + b(x) \\
A + B &= \{a + b \mid a \in A,\ b \in B\} \\
A + b &= \{a + b \mid a \in A\} \\
a + B &= \{a + b \mid b \in B\}
\end{aligned}
$$

Similar definitions apply for multiplication, subtraction, and division. So if a is $\sqrt{x}$ and b is $\lg x$, then $a + b$ is a function whose value is $\sqrt{x} + \lg x$ for every (postive) x. $O(a(x)) + O(b(x))$ (or just $O(a) + O(b)$) is then the set of functions you can get by adding a member of $O(\sqrt{x})$ to a member of $O(\lg x)$. For example, $O(a)$ contains $5\sqrt{x}+3$ and $O(b)$ contains $0.01 \lg x - 16$, so $O(a)+O(b)$ contains $5\sqrt{x}+0.01 \lg k - 13$, among many others.

1.2.1 Demonstrating "Big-Ohness"

Suppose we want to show that $5n^2 + 10\sqrt{n} \in O(n^2)$. That is, we need to find K and M so that

$$|5n^2 + 10\sqrt{n}| \le |Kn^2|, \quad \text{for } n > M.$$

We realize that n^2 grows faster than $\sqrt{n}$, so it eventually gets bigger than $10\sqrt{n}$ as well. So perhaps we can take $K = 6$ and find $M > 0$ such that

$$5n^2 + 10\sqrt{n} \le 5n^2 + n^2 = 6n^2$$

To get $10\sqrt{n} < n^2$, we need $10 < n^{3/2}$, or $n > 10^{2/3} \approx 4.7$. So choosing $M > 5$ certainly works.

1.3 Applications to Algorithm Analysis

In this course, we will be usually deal with integer-valued functions arising from measuring the complexity of algorithms. Table 1.1 gives a few common examples of orders that we deal with and their containment relations, and the sections below give examples of simple algorithmic analyses that use them.

1.3.1 Linear search

Let's apply all of this to a particular program. Here's a tail-recursive linear search for seeing if a particular value is in a sorted array:

```
/** True iff X is one of A[k]...A[A.length-1].
 *  Assumes A is increasing, k>= 0. */
static boolean isIn (int[] A, int k, int X) {
    if (k >= A.length)
        return false;
    else if (A[k] > X)
        return false;
    else if (A[k] == X)
        return true;
    else
        return isIn (A, k+1, X);
}
```

This is essentially a loop. As a measure of its complexity, let's define $C_{\mathtt{isIn}}(N)$ as the maximum number of instructions it executes for a call with $k = 0$ and `A.length`$= N$. By inspection, you can see that such a call will execute the first `if` test up to $N + 1$ times, the second and third up to N times, and the tail-recursive call on `isIn` up to N times. With one compiler[3], each recursive call of `isIn` executes at most 14 instructions before returning or tail-recursively calling `isIn`. The initial call executes 18. That gives a total of at most $14N + 18$ instructions. If instead we count the number of comparisons `k>=A.length`, we get at most $N + 1$. If we count the number of comparisons against `X` or the number of fetches of `A[0]`, we get at most $2N$. We could therefore say that the function giving the largest amount of time required to process an input of size N is either in $O(14N + 18)$, $O(N + 1)$, or $O(2N)$. However, these are all the same set, and in fact all are equal to $O(N)$. Therefore, we may throw away all those messy integers and describe $C_{\mathtt{isIn}}(N)$ as being in $O(N)$, thus illustrating the simplifying power of ignoring constant factors.

This bound is a worst-case time. For all arguments in which `X<=A[0]`, the `isIn` function runs in constant time. That time bound—the *best-case* bound—is seldom very useful, especially when it applies to so atypical an input.

Giving an $O(\cdot)$ bound to $C_{\mathtt{isIn}}(N)$ doesn't tell us that `isIn` *must* take time proportional to N even in the worst case, only that it takes no more. In this particular case, however, the argument used above shows that the worst case is, in fact, at least proportional to N, so that we may also say that $C_{\mathtt{isIn}}(N) \in \Omega(N)$. Putting the two results together, $C_{\mathtt{isIn}}(N) \in \Theta(N)$.

In general, then, asymptotic analysis of the space or time required for a given algorithm involves the following.

- Deciding on an appropriate measure for the *size* of an input (e.g., length of an array or a list).

---

[3]a version of gcc with the -O option, generating SPARC code for a Sun Sparcstation IPC workstation.

- Choosing a representative quantity to measure—one that is proportional to the "real" space or time required.

- Coming up with one or more functions that bound the quantity we've decided to measure, usually in the worst case.

- Possibly summarizing these functions by giving $O(\cdot)$, $\Omega(\cdot)$, or $\Theta(\cdot)$ characterizations of them.

1.3.2 Quadratic example

Here is a bit of code for sorting integers:

```
static void sort (int[] A) {
   for (int i = 1; i < A.length; i += 1) {
      int x = A[i];
      int j;
      for (j = i; j > 0 && x < A[j-1]; j -= 1)
        A[j] = A[j-1];
      A[j] = x;
   }
}
```

If we define $C_{\texttt{sort}}(N)$ as the worst-case number of times the comparison `x < A[j-1]` is executed for $N = $ `A.length`, we see that for each value of `i` from 1 to `A.length-1`, the program executes the comparison in the inner loop (on `j`) at most `i` times. Therefore,

$$
\begin{aligned}
C_{\texttt{sort}}(N) &= 1 + 2 + \ldots + N - 1 \\
&= N(N-1)/2 \\
&\in \Theta(N^2)
\end{aligned}
$$

This is a common pattern for nested loops.

1.3.3 Explosive example

Consider a function with the following form.

```
static int boom (int M, int X) {
   if (M == 0)
       return H (X);
   return boom (M-1, Q(X))
        + boom (M-1, R(X));
}
```

and suppose we want to compute $C_{\texttt{boom}}(M)$—the number of times `Q` is called for a given M in the worst case. If $M = 0$, this is 0. If $M > 0$, then `Q` gets executed once in computing the argument of the first recursive call, and then it gets executed however many times the two inner calls of `boom` with arguments of $M - 1$ execute

it. In other words,

$$
\begin{aligned}
C_{\mathtt{boom}}(0) &= 0 \\
C_{\mathtt{boom}}(i) &= 2C_{\mathtt{boom}}(i-1)+1
\end{aligned}
$$

A little mathematical massage:

$$
\begin{aligned}
C_{\mathtt{boom}}(M) &= 2C_{\mathtt{boom}}(M-1)+1, \\
&\qquad \text{for } M \geq 1 \\
&= 2(2C_{\mathtt{boom}}(M-2)+1)+1, \\
&\qquad \text{for } M \geq 2 \\
&\vdots \\
&= \underbrace{2(\cdots(2\cdot 0 + \underbrace{1)+1)\cdots+1}_{M}}_{M} \\
&= \sum_{0 \leq j \leq M-1} 2^j \\
&= 2^M - 1
\end{aligned}
$$

and so $C_{\mathtt{boom}}(M) \in \Theta(2^M)$.

1.3.4   Divide and conquer

Things become more interesting when the recursive calls decrease the size of parameters by a multiplicative rather than an additive factor. Consider, for example, binary search.

```
/** Returns true iff X is one of
 * A[L]...A[U]. Assumes A increasing,
 * L>=0, U-L < A.length. */
static boolean isInB (int[] A, int L, int U, int X) {
  if (L > U)
    return false;
  else {
    int m = (L+U)/2;
    if (A[m] == X)
      return true;
    else if (A[m] > X)
      return isInB (A, L, m-1, X);
    else
      return isInB (A, m+1, U, X);
  }
}
```

The worst-case time here depends on the number of elements of A under consideration, $U - L + 1$, which we'll call N. Let's use the number of times the first line is executed as the cost, since if the rest of the body is executed, the first line also had to have been executed[4]. If $N > 1$, the cost of executing isInB is 1 comparison

---

[4]For those of you seeking arcane knowledge, we say that the test L>U *dominates* all other statements.

of L and U followed by the cost of executing isInB either with $\lfloor(N-1)/2\rfloor$ or with $\lceil(N-1)/2\rceil$ as the new value of N[5]. Either quantity is no more than $\lceil(N-1)/2\rceil$. If $N \le 1$, there are two comparisons against N in the worst case.

Therefore, the following recurrence describes the cost, $C_{\texttt{isInB}}(i)$, of executing this function when $U - L + 1 = i$.

$$
\begin{aligned}
C_{\texttt{isInB}}(1) &= 2 \\
C_{\texttt{isInB}}(i) &= 1 + C_{\texttt{isInB}}(\lceil(i-1)/2\rceil), \quad i > 1.
\end{aligned}
$$

This is a bit hard to deal with, so let's again make the reasonable assumption that the value of the cost function, whatever it is, must increase as N increases. Then we can compute a cost function, $C'_{\texttt{isInB}}$ that is slightly larger than $C_{\texttt{isInB}}$, but easier to compute.

$$
\begin{aligned}
C'_{\texttt{isInB}}(1) &= 2 \\
C'_{\texttt{isInB}}(i) &= 1 + C'_{\texttt{isInB}}(i/2), \quad i > 1 \text{ a power of 2.}
\end{aligned}
$$

This is a slight over-estimate of $C_{\texttt{isInB}}$, but that still allows us to compute upper bounds. Furthermore, $C'_{\texttt{isInB}}$ is defined only on powers of two, but since isInB's cost increases as N increases, we can still bound $C_{\texttt{isInB}}(N)$ conservatively by computing $C'_{\texttt{isInB}}$ of the next higher power of 2. Again with the massage:

$$
\begin{aligned}
C'_{\texttt{isInB}}(i) &= 1 + C'_{\texttt{isInB}}(i/2), \quad i > 1 \text{ a power of 2.} \\
&= 1 + 1 + C'_{\texttt{isInB}}(i/4), \quad i > 2 \text{ a power of 2.} \\
&\vdots \\
&= \underbrace{1 + \cdots + 1}_{\lg N} + 2
\end{aligned}
$$

The quantity $\lg N$ is the logarithm of N base 2, or roughly "the number of times one can divide N by 2 before reaching 1." In summary, we can say $C_{\texttt{isIn}}(N) \in O(\lg N)$. Similarly, one can in fact derive that $C_{\texttt{isIn}}(N) \in \Theta(\lg N)$.

1.3.5 Divide and fight to a standstill

Consider now a subprogram that contains *two* recursive calls.

```
static void mung (int[] A, L, U) {
    if (L < U) {
        int m = (L+U)/2;
        mung (A, L, m);
        mung (A, m+1, U);
    }
}
```

---

[5]The notation $\lfloor x \rfloor$ means the result of rounding x down (toward $-\infty$) to an integer, and $\lceil x \rceil$ means the result of rounding x up to an integer.

We can approximate the arguments of both of the internal calls by $N/2$ as before, ending up with the following approximation, $C_{\texttt{mung}}(N)$, to the cost of calling $\texttt{mung}$ with argument $N = U - L + 1$ (we are counting the number of times the test in the first line executes).

$$
\begin{aligned}
C_{\texttt{mung}}(1) &= 3 \\
C_{\texttt{mung}}(i) &= 1 + 2C_{\texttt{mung}}(i/2), \quad i > 1 \text{ a power of 2.}
\end{aligned}
$$

So,

$$
\begin{aligned}
C_{\texttt{mung}}(N) &= 1 + 2(1 + 2C_{\texttt{mung}}(N/4)), \quad N > 2 \text{ a power of 2.} \\
&\vdots \\
&= 1 + 2 + 4 + \ldots + N/2 + N \cdot 3
\end{aligned}
$$

This is a sum of a geometric series $(1 + r + r^2 + \cdots + r^m)$, with a little extra added on. The general rule for geometric series is

$$
\sum_{0 \le k \le m} r^k = (r^{m+1} - 1)/(r - 1)
$$

so, taking $r = 2$,

$$
C_{\texttt{mung}}(N) = 4N - 1
$$

or $C_{\texttt{mung}}(N) \in \Theta(N)$.

1.4   Amortization

So far, we have considered the time spent by individual operations, or individual calls on a certain function of interest. Sometimes, however, it is fruitful to consider the cost of whole sequence of calls, especially when each call affects the cost of later calls.

Consider, for example, a simple binary counter. Incrementing this counter causes it to go through a sequence like this:

```
0 0 0 0 0
0 0 0 0 1
0 0 0 1 0
0 0 0 1 1
0 0 1 0 0
 ...
0 1 1 1 1
1 0 0 0 0
 ...
```

Each step consists of *flipping* a certain number of bits, converting bit b to $1 - b$. More precisely, the algorithm for going from one step to another is

Increment: Flip the bits of the counter from right to left, up to and including the first 0-bit encountered (if any).

Clearly, if we are asked to give a worst-case bound on the cost of the increment operation for an N-bit counter (in number of flips), we'd have to say that it is $\Theta(N)$: all the bits can be flipped. Using just that bound, we'd then have to say that the cost of performing M increment operations is $\Theta(M \cdot N)$.

But the costs of consecutive increment operations are related. For example, if one increment flips more than one bit, the next increment will always flip exactly one (why?). In fact, if you consider the pattern of bit changes, you'll see that the units (rightmost) bit flips on every increment, the 2's bit on every second increment, the 4's bit on every fourth increment, and in general, then 2^k's bit on every $(2^k)^{\text{th}}$ increment. Therefore, over any sequence of M consecutive increments, starting at 0, there will be

$$\underbrace{M}_{\text{unit's flips}} + \underbrace{\lfloor M/2 \rfloor}_{\text{2's flips}} + \underbrace{\lfloor M/4 \rfloor}_{\text{4's flips}} + \ldots + \underbrace{\lfloor M/2^n \rfloor}_{2^n\text{'s flips}}, \text{ where } n = \lfloor \lg M \rfloor$$

$$= \underbrace{2^n + 2^{n-1} + 2^{n-2} + \ldots + 1}_{=2^{n+1}-1} + (M - 2^n)$$

$$= 2^n - 1 + M$$

$$< 2M \text{ flips}$$

In other words, this is the same result we would get if we performed M increments each of which had a worst-case cost of 2 flips, rather than N. We call 2 flips the *amortized cost* of an increment. To *amortize* in the context of algorithms is to treat the cost of each individual operation in a sequence as if it were spread out among all the operations in the sequence[6]. Any particular increment might take up to N flips, but we treat that as N/M flips credited to each increment operation in the sequence (and likewise count each increment that takes only one flip as $1/M$ flip for each increment operation). The result is that we get a more realistic idea of how much time the entire program will take; simply multiplying the ordinary worst-case time by M gives us a very loose and pessimistic estimate. Nor is amortized cost the same as average cost; it is a stronger measure. If a certain operation has a given average cost, that leaves open the possibility that there is some unlikely sequence of inputs that will make it look bad. A bound on amortized worst-case cost, on the other hand, is *guaranteed* to hold regardless of input.

1.5 Complexity of Problems

So far, I have discussed only the analysis of an algorithm's complexity. An algorithm, however, is just a particular way of solving some problem. We might therefore

---

[6]The word *amortize* comes from an Old French word meaning "to death." The original meaning from which the computer-science usage comes, is "to gradually write off the initial cost of something."

consider asking for complexity bounds on the *problem's* complexity. That is, can we bound the complexity of the *best possible* algorithm? Obviously, if we have a particular algorithm and its time complexity is $O(f(n))$, where n is the size of the input, then the complexity of the best possible algorithm must also be $O(f(n))$. We call $f(n)$, therefore, an *upper bound* on the (unknown) complexity of the best-possible algorithm. But this tells us nothing about whether the best-possible algorithm is any *faster* than this—it puts no *lower bound* on the time required for the best algorithm. For example, the worst-case time for `isIn` is $\Theta(N)$. However, `isInB` is much faster. Indeed, one can show that if the only knowledge the algorithm can have is the result of comparisons between `X` and elements of the array, then `isInB` has the best possible bound (it is *optimal*), so that the entire *problem* of finding an element in an ordered array has worst-case time $\Theta(\lg N)$.

Putting an upper bound on the time required to perform some problem simply involves finding an algorithm for the problem. By contrast, putting a good lower bound on the required time is much harder. We essentially have to prove that no algorithm can have a better execution time than our bound, regardless of how much smarter the algorithm designer is than we are. Trivial lower bounds, of course, are easy: every problem's worst-case time is $\Omega(1)$, and the worst-case time of any problem whose answer depends on all the data is $\Omega(N)$, assuming that one's idealized machine is at all realistic. Better lower bounds than those, however, require quite a bit of work. All the better to keep our theoretical computer scientists employed.

1.6   A Note on Notation

Other authors use notation such as $f(n) = O(n^2)$ rather than $f(n) \in O(n^2)$. I don't because I consider it nonsensical. To justify the use of '=', one either has to think of $f(n)$ as a set of functions (which it isn't), or think of $O(n^2)$ as a single function that differs with each separate appearance of $O(n^2)$ (which is bizarre). I can see no disadvantages to using '$\in$', which makes perfect sense, so that's what I use.

Exercises

1.1.   Demonstrate the following, or give counter-examples where indicated. Showing that a certain $O(\cdot)$ formula is true means producing suitable K and M for the definition at the beginning of §1.1. Hint: sometimes it is useful to take the logarithms of two functions you are comparing.

   a. $O(\max(|f_0(n)|, |f_1(n)|)) = O(f_0(n)) + O(f_1(n))$.

   b. If $f(n)$ is a polynomial in n, then $\lg f(n) \in O(\lg n)$.

   c. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$. This is a bit of trick question, really, to make you look at the definitions carefully. Under what conditions is the equation true?

   d. There is a function $f(x) > 0$ such that $f(x) \notin O(x)$ and $f(x) \notin \Omega(x)$.

e. There is a function $f(x)$ such that $f(0) = 0$, $f(1) = 100$, $f(2) = 10000$, $f(3) = 10^6$, but $f(n) \in O(n)$.

f. $n^3 \lg n \in O(n^{3.0001})$.

g. There is no constant k such that $n^3 \lg n \in \Theta(n^k)$.

1.2. Show each of the following *false* by exhibiting a counterexample. Assume that f and g are any real-valued functions.

a. $O(f(x) \cdot s(x)) = o(f(x))$, assuming $\lim_{x \to \infty} s(x) = 0$.

b. If $f(x) \in O(x^3)$ and $g(x) \in O(x)$ then $f(x)/g(x) \in O(x^2)$.

c. If $f(x) \in \Omega(x)$ and $g(x) \in \Omega(x)$ then $f(x) + g(x) \in \Omega(x)$.

d. If $f(100) = 1000$ and $f(1000) = 1000000$ then f cannot be $O(1)$.

e. If $f_1(x), f_2(x), \ldots$ are a bunch of functions that are all in $\Omega(1)$, then

$$F(x) = \sum_{1 \leq i \leq N} f_i(x) \in \Omega(N).$$

Chapter 2

Data Types in the Abstract

Most of the "classical" data structures covered in courses like this represent some sort of *collection* of data. That is, they contain some set or multiset[1] of values, possibly with some ordering on them. Some of these collections of data are *associatively indexed;* they are search structures that act like functions mapping certain indexing values (*keys*) into other data (such as names into street addresses).

We can characterize the situation in the abstract by describing sets of operations that are supported by different data structures—that is by describing possible *abstract data types.* From the point of view of a program that needs to represent some kind of collection of data, this set of operations is all that one needs to know.

For each different abstract data type, there are typically several possible implementations. Which you choose depends on how much data your program has to process, how fast it has to process the data, and what constraints it has on such things as memory space. It is a dirty little secret of the trade that for quite a few programs, it hardly matters what implementation you choose. Nevertheless, the well-equipped programmer should be familiar with the available tools.

I expect that many of you will find this chapter frustrating, because it will talk mostly about *interfaces* to data types without talking very much at all about the implementations behind them. Get used to it. After all, the standard library behind any widely used programming language is presented to you, the programmer, as a set of interfaces—directions for what parameters to pass to each function and some commentary, generally in English, about what it does. As a working programmer, you will in turn spend much of your time producing modules that present the same features to your clients.

2.1 Iterators

If we are to develop some general notion of a collection of data, there is at least one generic question we'll have to answer: how are we going to get items *out* of such a collection? You are familiar with one kind of collection already—an array. Getting

---

[1]A *multiset* or *bag* is like a set except that it may contain multiple copies of a particular data value. That is, each member of a multiset has a *multiplicity:* a number of times that it appears.

items out of an array is easy; for example, to print the contents of an array, you
might write

```
for (int i = 0; i < A.length; i += 1)
   System.out.print (A[i] + ", ");
```

Arrays have a natural notion of an nth element, so such loops are easy. But what
about other collections? Which is the "first penney" in a jar of penneys? Even if
we do arbitrarily choose to give every item in a collection a number, we will find
that the operation "fetch the nth item" may be expensive (consider lists of things
such as in Scheme).

   The problem with attempting to impose a numbering on every collection of items
as way to extract them is that it forces the implementor of the collection to provide
a more specific tool than our problem may require. It's a classic engineering trade-
off: satisfying one constraint (that one be able to fetch the nth item) may have
other costs (fetching all items one by one may become expensive).

   So the problem is to provide the items in a collection without relying on indices,
or possibly without relying on order at all. Java provides two conventions, realized as
interfaces. The interface `java.util.Iterator` provides a way to access all the items
in a collection in *some* order. The interface `java.util.ListIterator` provides a
way to access items in a collection in some specific order, but without assigning an
index to each item[2].

2.1.1   The Iterator Interface

The Java library defines an interface, `java.util.Iterator`, shown in Figure 2.1,
that captures the general notion of "something that sequences through all items
in a collection" without any commitment to order. This is only a Java interface;
there is no implementation behind it. In the Java library, the standard way for a
class that represents a collection of data items to provide a way to sequence through
those items is to define a method such as

```
Iterator<SomeType> iterator () { ... }
```

that allocates and returns an Iterator (Figure 3.3 includes an example). Often the
actual type of this iterator will be hidden (even private); all the user of the class
needs to know is that the object returned by `iterator` provides the operations
`hasNext` and `next` (and sometimes `remove`). For example, a general way to print
all elements of a collection of `String`s (analogous to the previous array printer)
might be

```
for (Iterator<String> i = C.iterator (); i.hasNext (); )
   System.out.println (i.next () + " ");
```

---

[2]The library also defines the interface `java.util.Enumeration`, which is essentially an older
version of the same idea. We won't talk about that interface here, since the official position is that
`Iterator` is preferred for new programs.

```
package java.util;
/** An object that delivers each item in some collection of items
 *  each of which is a T. */
public interface Iterator <T> {
  /** True iff there are more items to deliver. */
  boolean hasNext ();
  /** Advance THIS to the next item and return it. */
  T next ();
  /** Remove the last item delivered by next() from the collection
   *  being iterated over.  Optional operation: may throw
   *  UnsupportedOperationException if removal is not possible. */
  void remove ();
}
```

Figure 2.1: The `java.util.Iterator` interface.

The programmer who writes this loop needn't know what gyrations the object `i` has to go through to produce the requested elements; even a major change in how `C` represents its collection requires no modification to the loop.

This particular kind of **for** loop is so common and useful that in Java 2, version 1.5, it has its own "syntactic sugar," known as an *enhanced* **for** *loop.* You can write

```
for (String i : C)
    System.out.println (i + " ");
```

to get the same effect as the previous **for** loop. Java will insert the missing pieces, turning this into

```
for (Iterator<String> ρ = C.iterator (); ρ.hasNext; ) {
    String i = ρ.next ();
    System.out.println (i + " ");
}
```

where ρ is some new variable introduced by the compiler and unused elsewhere in the program, and whose type is taken from that of `C.iterator()`. This enhanced **for** loop will work for any object `C` whose type implements the interface `java.lang.Iterable`, defined simply

```
public interface Iterable<T> {
    Iterator<T> iterator ();
}
```

Thanks to the enhanced **for** loop, simply by defining an `iterator` method on a type you define, you provide a very convenient way to sequence through any subparts that objects of that type might contain.

Well, needless to say, having introduced this convenient shorthand for `Iterator`s, Java's designers were suddenly in the position that iterating through the elements

of an array was much clumsier than iterating through those of a library class. So
they extended the enhanced **for** statement to encompass arrays. So, for example,
these two methods are equivalent:

```
/** The sum of the              /** The sum of the elements
 *   elements of A */            *  of A */
int sum (int[] A) {             int sum (int[] A) {
  int S;                          int S;
  S = 0;                          S = 0;
  for (int x : A)        ⟹       for (int κ = 0; κ < A.length; κ++)
    S += x;                       {
}                                    int x = A[κ];
                                     S += x;
                                 }
                                }
```

where κ is a new variable introduced by the compiler.

2.1.2   The ListIterator Interface

Some collections do have a natural notion of ordering, but it may still be expensive
to extract an arbitrary item from the collection by index. For example, you may
have seen linked lists in the Scheme language: given an item in the list, it requires
n operations to find the nth succeeding item (in contrast to a Java array, which
requires only one Java operation or a few machine operations to retrieve any item).
The standard Java library contains the interface `java.util.ListIterator`, which
captures the idea of sequencing through an ordered sequence without fetching each
explicitly by number. It is summarized in Figure 2.2. In addition to the "nav-
igational" methods and the `remove` method of `Iterator` (which it extends), the
`ListIterator` class provides operations for inserting new items or replacing items
in a collection.

2.2   The Java Collection Abstractions

The Java library (beginning with JDK 1.2) provides a hierarchy of interfaces rep-
resenting various kinds of collection, plus a hierarchy of abstract classes to help
programmers provide implementations of these interfaces, as well as a few actual
("concrete") implementations. These classes are all found in the package `java.util`.
Figure 2.4 illustrates the hierarchy of classes and interfaces devoted to collections.

2.2.1   The Collection Interface

The Java library interface `java.util.Collection`, whose methods are summarized
in Figures 2.5 and 2.6, is supposed to describe data structures that contain collec-
tions of values, where each value is a reference to some Object (or null). The
term "collection" as opposed to "set" is appropriate here, because `Collection` is
supposed to be able describe multisets (bags) as well as ordinary mathematical sets.

```
package java.util;
/** Abstraction of a position in an ordered collection.  At any
 *  given time, THIS represents a position (called its cursor)
 *  that is just after some number of items of type T (0 or more) of
 *  a particular collection, called the underlying collection.  */
public interface ListIterator<T> extends Iterator<T> {
  /* Exceptions: Methods that return items from the collection throw
   * NoSuchElementException if there is no appropriate item.  Optional
   * methods throw UnsupportedOperationException if the method is not
   * supported.  */

  /* Required methods:  */

  /** True unless THIS is past the last item of the collection */
  boolean hasNext ();

  /** True unless THIS is before the first item of the collection */
  boolean hasPrevious ();

  /** Returns the item immediately after the cursor, and
   *  moves the current position to just after that item.
   *  Throws NoSuchElementException if there is no such item. */
  T next ();

  /** Returns the item immediately before the cursor, and
   *  moves the current position to just before that item.
   *  Throws NoSuchElementException if there is no such item. */
  T previous ();

  /** The number of items before the cursor */
  int nextIndex ();

  /* nextIndex () - 1 */
  int previousIndex ();
```

Figure 2.2: The `java.util.ListIterator` interface.

```
/* Optional methods:  */

/** Insert item X into the underlying collection immediately before
 *  the cursor (X will be returned by previous()). */
void add (T x);

/** Remove the item returned by the most recent call to .next ()
 *  or .previous (). There must not have been a more recent
 *  call to .add(). */
void remove ();

/** Replace the item returned by the most recent call to .next ()
 *  or .previous () with X in the underlying collection.
 *  There must not have been a more recent call to .add() or .remove. */
void set (T x);
}
```

Figure 2.2, continued: Optional methods in the `ListIterator` class.
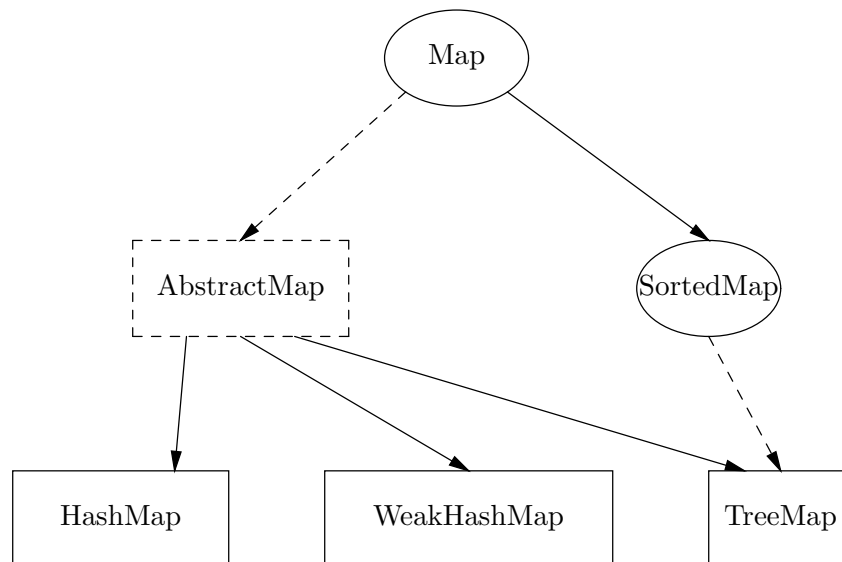
Figure 2.3: The Java library's Map-related types (from `java.util`). Ellipses represent interfaces; dashed boxes are abstract classes, and solid boxes are concrete (non-abstract) classes. Solid arrows indicate **extends** relationships, and dashed arrows indicate **implements** relationships. The abstract classes are for use by implementors wishing to add new collection classes; they provide default implementations of some methods. Clients apply **new** to the concrete classes to get instances, and (at least ideally), use the interfaces as formal parameter types so as to make their methods as widely applicable as possible.
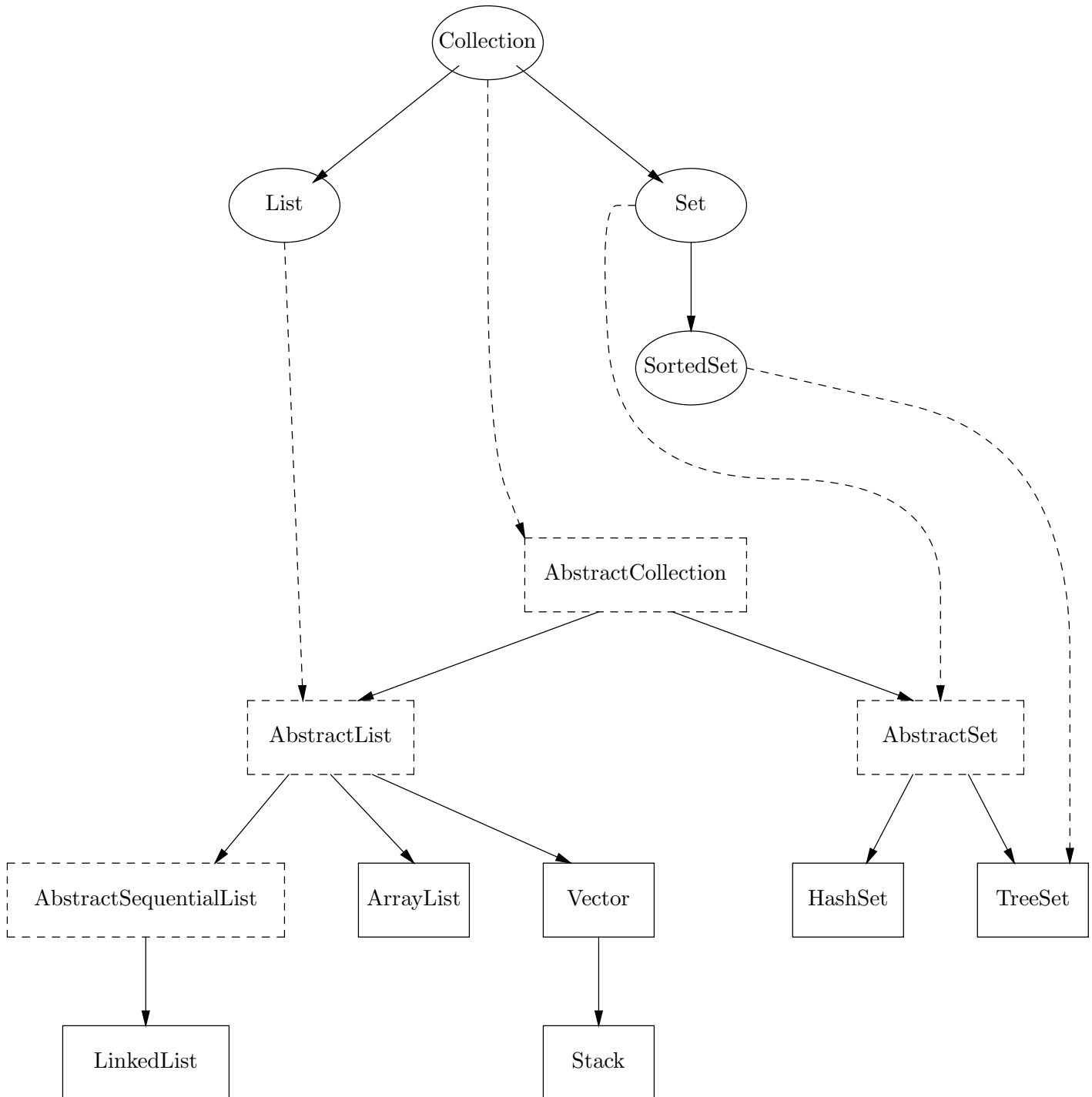
Figure 2.4: The Java library's Collection-related types (from `java.util`). See Figure 2.3 for the notation.

Since this is an interface, the documentation comments describing the operations need not be accurate; an inept or mischievous programmer can write a class that implements `Collection` in which the `add` method *removes* values. Nevertheless, any decent implementor will honor the comments, so that any method that accepts a `Collection`, C, as an argument can expect that, after executing C.`add(x)`, the value `x` will be in C.

Not every kind of `Collection` needs to implement every method—specifically, not the optional methods in Figure 2.6—but may instead choose to raise the standard exception `UnsupportedOperationException`. See §2.5 for a further discussion of this particular design choice. Classes that implement only the required methods are essentially *read-only* collections; they can't be modified once they are created.

The comment concerning constructors in Figure 2.5 is, of course, merely a comment. Java interfaces do not have constructors, since they do not represent specific types of concrete object. Nevertheless, you ultimately need some constructor to create a `Collection` in the first place, and the purpose of the comment is to suggest some useful uniformity.

At this point, you may well be wondering of what possible use the `Collection` class might be, inasmuch as it is impossible to create one directly (it is an interface), and you are missing details about what its members do (for example, can a given `Collection` have two equal elements?). The point is that any function that you *can* write using just the information provided in the `Collection` interface will work for *all* implementations of `Collection`.

For example, here is simple method to determine if the elements of one `Collection` are a subset of another:

```
/** True iff C0 is a subset of C1, ignoring repetitions. */
public static boolean subsetOf (Collection<?> C0, Collection<?> C1) {
   for (Object i : C0)
      if (! C1.contains (i))
         return false;
   // Note: equivalent to
   // for (Iterator<?> iter = C0.iterator(); iter.hasNext (); ) {
   //    Object i = iter.next ();
   //    ...
   return true;
}
```

We have no idea what kinds of objects `C0` and `C1` are (they might be completely different implementations of `Collection`), in what order their iterators deliver elements, or whether they allow repetitions. This method relies solely on the properties described in the interface and its comments, and therefore always works (assuming, as always, that the programmers who write classes that implement `Collection` do their jobs). We don't have to rewrite it for each new kind of `Collection` we implement.

```
package java.util;
/** A collection of values, each an Object reference. */
public interface Collection<T> extends Iterable<T> {
    /* Constructors. Classes that implement Collection should
     * have at least two constructors:
     *   CLASS (): Constructs an empty CLASS
     *   CLASS (C): Where C is any Collection, constructs a CLASS that
     *       contains the same elements as C. */

    /* Required methods:  */

    /** The number of values in THIS. */
    int size ();

    /** True iff size () == 0. */
    boolean isEmpty ();

    /** True iff THIS contains X: that is, if for some z in
     *  THIS, either z and X are null, or z.equals (X). */
    boolean contains (Object x);

    /** True iff contains(x) for all elements x in C. */
    boolean containsAll (Collection<?> c);

    /** An iterator that yields all the elements of THIS, in some
     *  order. */
    Iterator<T> iterator ();

    /** A new array containing all elements of THIS. */
    Object[] toArray ();

    /** Assuming ANARRAY has dynamic type T[] (where T is some
     *  reference type), the result is an array of type T[] containing
     *  all elements of THIS.  The result is ANARRAY itself, if all of
     *  these elements fit (leftover elements of ANARRAY are set to null).
     *  Otherwise, the result is a new array.  It is an error if not
     *  all items in THIS are assignable to T. */
    <T> T[] toArray (T[] anArray);
```

Figure 2.5: The interface `java.util.Collection`, required members.

```
// Interface java.util.Collection, continued.
  /* Optional methods.  Any of these may do nothing except to
   * throw UnsupportedOperationException. */

  /** Cause X to be contained in THIS.  Returns true if the Collection */
   *  changes as a result. */
  boolean add (T x);

  /** Cause all members of C to be contained in THIS.  Returns true
   *  if the object THIS changes as a result. */
  boolean addAll (Collection<? extends T> c);

  /** Remove all members of THIS. */
  void clear ();

  /** Remove a Object .equal to X from THIS, if one exists,
   *  returning true iff the object THIS changes as a result. */
  boolean remove (Object X);

  /** Remove all elements, x, such that C.contains(x) (if any
   *  are present), returning true iff there were any
   *  objects removed. */
  boolean removeAll (Collection<?> c);

  /** Intersection: Remove all elements, x, such that C.contains(x)
   *  is false, returning true iff any items were removed. */
  boolean retainAll (Collection<?> c);
}
```

Figure 2.6: Optional members of the interface `java.util.Collection`

2.2.2 The Set Interface

In mathematics, a set is a collection of values in which there are no duplicates. This is the idea also for the interface `java.util.Set`. Unfortunately, this provision is not directly expressible in the form of a Java interface. In fact, as far as the Java compiler is concerned, the following serves as a perfectly good definition:

```
package java.util;
public interface Set<T> extends Collection<T> { }
```

The methods, that is, are all the same. The differences are all in the comments. The one-copy-of-each-element rule is reflected in more specific comments on several methods. The result is shown in Figure 2.7. In this definition, we also include the methods `equals` and `hashCode`. These methods are automatically part of any interface, because they are defined in the Java class `java.lang.Object`, but I included them here because their semantic specification (the comment) is more stringent than for the general Object. The idea, of course, is for `equals` to denote set equality. We'll return to `hashCode` in Chapter 7.

2.2.3 The List Interface

As the term is used in the Java libraries, a list is a sequence of items, possibly with repetitions. That is, it is a specialized kind of `Collection`, one in which there is a sequence to the elements—a first item, a last item, an nth item—and items may be repeated (it can't be considered a `Set`). As a result, it makes sense to extend the interface (relative to `Collection`) to include additional methods that make sense for well-ordered sequences. Figure 2.8 displays the interface.

A great deal of functionality here is wrapped up in the `listIterator` method and the object it returns. As you can see from the interface descriptions, you can insert, add, remove, or sequence through items in a `List` either by using methods in the `List` interface itself, or by using `listIterator` to create a list iterator with which you can do the same. The idea is that using the `listIterator` to process an entire list (or some part of it) will generally be faster than using `get` and other methods of `List` that use numeric indices to denote items of interest.

Views

The `subList` method is particularly interesting. A call such as `L.subList(i,j)` is supposed to produce another `List` (which will generally *not* be of the same type as L) consisting of the `i`$^{\text{th}}$ through the `(j-1)`$^{\text{th}}$ items of L. Furthermore, it is to do this by providing a *view* of this part of L—that is, an alternative way of accessing the same data containers. The idea is that modifying the sublist (using methods such as `add`, `remove`, and `set`) is supposed to modify the corresponding portion of L as well. For example, to remove all but the first `k` items in list L, you might write

```
L.subList (k, L.size ()).clear ();
```

```
package java.util;
/** A Collection that contains at most one null item and in which no
 *  two distinct non-null items are .equal. The effects of modifying
 *  an item contained in a Set so as to change the value of .equal
 *  on it are undefined. */
public interface Set<T> extends Collection<T> {
   /* Constructors. Classes that implement Set should
    * have at least two constructors:
    *    CLASS (): Constructs an empty CLASS
    *    CLASS (C): Where C is any Collection, constructs a CLASS that
    *        contains the same elements as C, with duplicates removed. */

   /** Cause X to be contained in THIS.  Returns true iff X was */
    *  not previously a member. */
   boolean add (T x);

   /** True iff S is a Set (instanceof Set) and is equal to THIS as a
    *  set (size()==S.size() each of item in S is contained in THIS). */
   boolean equals (Object S);

   /** The sum of the values of x.hashCode () for all x in THIS, with
    *  the hashCode of null taken to be 0. */
   int hashCode ();

   /* Other methods inherited from Collection:
    *    size, isEmpty, contains, containsAll, iterator, toArray,
    *    addAll, clear, remove, removeAll, retainAll */
}
```

Figure 2.7: The interface `java.util.Set`. Only methods with comments that are more specific than those of `Collection` are shown.

```
package java.util;
/** An ordered sequence of items, indexed by numbers 0 .. N-1,
 *  where N is the size() of the List.  */
public interface List<T> extends Collection<T> {

  /* Required methods:  */

  /** The Kth element of THIS, where 0 <= K < size().  Throws
   *  IndexOutOfBoundsException if K is out of range. */
  T get (int k);

  /** The first value k such that get(k) is null if X==null,
   *  X.equals (get(k)), otherwise, or -1 if there is no such k. */
  int indexOf (Object x);

  /** The largest value k such that get(k) is null if X==null,
   *  X.equals (get(k)), otherwise, or -1 if there is no such k. */
  int lastIndexOf (Object x);

  /* NOTE: The methods iterator, listIterator, and subList produce
   * views that become invalid if THIS is structurally modified by
   * any other means (see text). */

  /** An iterator that yields all the elements of THIS, in proper
   *  index order.   (NOTE: it is always valid for iterator() to
   *  return the same value as would listIterator, below.) */
  Iterator<T> iterator ();

  /** A ListIterator that yields the elements K, K+1, ..., size()-1
   *  of THIS, in that order, where 0 <= K <= size().   Throws
   *  IndexOutOfBoundsException if K is out of range. */
  ListIterator<T> listIterator (int k);

  /** Same as listIterator (0) */
  ListIterator<T> listIterator ();

  /** A view of THIS consisting of the elements L, L+1, ..., U-1,
   *  in that order.  Throws IndexOutOfBoundsException unless
   *  0 <= L <= U <= size(). */
  List<T> subList (int L, int U);

  /* Other methods inherited from Collection:
   *    add, addAll, size, isEmpty, contains, containsAll, remove, toArray */
```

Figure 2.8: Required methods of interface `java.util.List`, beyond those inherited from `Collection`.

```
/* Optional methods:  */

/** Cause item K of THIS to be X, and items K+1, K+2, ... to contain
 *  the previous values of get(K), get(K+1), .... Throws
 *  IndexOutOfBoundsException unless 0<=K<=size(). */
void add (int k, T x);

/** Same effect as add (size (), x); always returns true. */
boolean add (T x);

/** If the elements returned by C.iterator () are x0, x1,..., in
 *  that order, then perform the equivalent of add(K,x0),
 *  add(K+1,x1), ..., returning true iff there was anything to
 *  insert.  IndexOutOfBoundsException unless 0<=K<=size(). */
boolean addAll (int k, Collection<T> c);

/** Same as addAll(size(), c). */
boolean addAll (Collection<T> c);

/** Remove item K, moving items K+1, ... down one index position,
 *  and returning the removed item.  Throws
 *  IndexOutOfBoundsException if there is no item K. */
Object remove (int k);

/** Remove the first item equal to X, if any, moving subsequent
 *  elements  one index position lower.  Return true iff anything
 *  was removed. */
boolean remove (Object x);

/** Replace get(K) with X, returning the initial (replaced) value of
 *  get(K).  Throws IndexOutOfBoundsException if there is no item K. */
Object set (int k, T x);

/* Other methods inherited from Collection: removeAll, retainAll */
}
```

Figure 2.8, continued: Optional methods of interface `java.util.List`, beyond
from those inherited from `Collection`.

As a result, there are a lot of possible operations on `List` that don't have to be defined, because they fall out as a natural consequence of operations on sublists. There is no need for a version of `remove` that deletes items i through j of a list, or for a version of `indexOf` that starts searching at item k.

Iterators (including ListIterators) provide another example of a view of Collections. Again, you can access or (sometimes) modify the current contents of a Collection through an iterator that its methods supply. For that matter, any Collection is itself a view—the "identity view" if you want.

Whenever there are two possible views of the same entity, there is a possibility that using one of them to modify the entity will interfere with the other view. It's not just that changes in one view are supposed to be seen in other views (as in the example of clearing a sublist, above), but straightforward and fast implementations of some views may malfunction when the entity being viewed is changed by other means. What is supposed to happen when you call `remove` on an iterator, but the item that is supposed to be removed (according to the specification of `Iterator`) has already been removed directly (by calling `remove` on the full Collection)? Or suppose you have a sublist containing items 2 through 4 of some full list. If the full list is `cleared`, and then 3 items are added to it, what is in the sublist view?

Because of these quandries, the full specification of many view-producing methods (in the `List` interface, these are `iterator`, `listIterator`, and `subList`) have a provision that the view becomes invalid if the underlying `List` is *structurally modified* (that is, if items are added or removed) through some means other than that view. Thus, the result of `L.iterator()` becomes invalid if you perform `L.add(...)`, or if you perform `remove` on some other `Iterator` or sublist produced from `L`. By contrast, we will also encounter views, such as those produced by the `values` method on `Map` (see Figure 2.12), that are supposed to remain valid even when the underlying object is structurally modified; it is an obligation on the implementors of new kinds of `Map` that they see that this is so.

2.2.4 Ordered Sets

The `List` interface describes data types that describe sequences in which the programmer explicitly determines the order of items in the sequence by the order or place in which they are added to the sequence. By contrast, the `SortedSet` interface is intended to describe sequences in which the *data* determine the ordering according to some selected relation. Of course, this immediately raises a question: in Java, how do we represent this "selected relation" so that we can specify it? How do we make an ordering relation a parameter?

Orderings: the `Comparable` and `Comparator` Interfaces

There are various ways for functions to define an ordering over some set of objects. One way is to define boolean operations `equals`, `less`, `greater`, etc., with the obvious meanings. Libraries in the C family of languages (which includes Java) tend to combine all of these into a single function that returns an integer whose sign denotes the relation. For example, on the type `String`, `x.compareTo("cat")`

```
package java.lang;
/** Describes types that have a natural ordering. */
public interface Comparable<T> {
   /** Returns
    *    * a negative value iff THIS < Y under the natural ordering
    *    * a positive value iff THIS > Y;
    *    * 0 iff X and Y are "equivalent".
    *  Throws ClassCastException if X and Y are incomparable. */
   int compareTo (T y);
}
```

Figure 2.9: The interface `java.lang.Comparable`, which marks classes that define a natural ordering.

returns an integer that is zero, negative, or positive, depending on whether `x` equals `"cat"`, comes before it in lexicographic order, or comes after it. Thus, the ordering $x \leq y$ on Strings corresponds to the condition `x.compareTo(y)<=0`.

For the purposes of the `SortedSet` interface, this $\leq$ (or $\geq$) ordering represented by `compareTo` (or `compare`, described below) is intended to be a *total ordering*. That is, it is supposed to be transitive ($x \leq y$ and $y \leq z$ implies $x \leq z$), reflexive ($x \leq x$), and antisymmetric ($x \leq y$ and $y \leq x$ implies that x equals y). Also, for all x and y in the function's domain, either $x \leq y$ or $y \leq x$.

Some classes (such as `String`) define their own standard comparison operation. The standard way to do so is to implement the `Comparable` interface, shown in Figure 2.9. However, not all classes have such an ordering, nor is *the* natural ordering necessarily what you want in any given case. For example, one can sort Strings in dictionary order, reverse dictionary order, or case-insensitive order.

In the Scheme language, there is no particular problem: an ordering relation is just a function, and functions are perfectly good values in Scheme. To a certain extent, the same is true in languages like C and Fortran, where functions can be used as arguments to subprograms, but unlike Scheme, have access only to global variables (what are called static fields or class variables in Java). Java does not directly support functions as values, but it turns out that this is not a limitation. The Java standard library defines the `Comparator` interface (Figure 2.10) to represent things that may be used as ordering relations.

The methods provided by both of these interfaces are supposed to be proper total orderings. However, as usual, none of the conditions can actually be enforced by the Java language; they are just conventions imposed by comment. The programmer who violates these assumptions may cause all kinds of unexpected behavior. Likewise, nothing can keep you from defining a `compare` operation that is inconsistent with the `.equals` function. We say that `compare` (or `compareTo`) is *consistent with equals* if `x.equals(y)` iff `C.compare(x,y)==0`. It's generally good practice to maintain this consistency in the absence of a good reason to the contrary.

```
package java.util;
/** An ordering relation on certain pairs of objects. If */
public interface Comparator<T> {
   /** Returns
    *    * a negative value iff X < Y according to THIS ordering;
    *    * a positive value iff X > Y;
    *    * 0 iff X and Y are "equivalent" under the order;
    *  Throws ClassCastException if X and Y are incomparable.
    */
   int compare (T x, T y);

   /** True if ORD is "same" ordering as THIS.  It is legal to return
    *  false (conservatively) even if ORD does define the same ordering,
    *  but should return true only if ORD.compare (X, Y) and
    *  THIS.compare(X, Y) always have the same value. */
   boolean equals (Object ord);
}
```

Figure 2.10: The interface `java.util.Comparator`, which represents ordering relations between Objects.

The SortedSet Interface

The `SortedSet` interface shown in Figure 2.11 extends the `Set` interface so that its `iterator` method delivers an `Iterator` that sequences through its contents "in order." It also provides additional methods that make sense only when there is such an order. There are intended to be two ways to define this ordering: either the programmer supplies a `Comparator` when constructing a `SortedSet` that defines the order, or else the contents of the set must `Comparable`, and their natural order is used.

2.3   The Java Map Abstractions

The term *map* or *mapping* is used in computer science and elsewhere as a synonym for *function* in the mathematical sense—a correspondence between items in some set (the *domain*) and another set (the *codomain*) in which each item of the domain corresponds to (*is mapped to by*) a single item of the codomain[3].

It is typical among programmers to take a rather operational view, and say that a map-like data structure "looks up" a given *key* (domain value) to find the associated *value* (codomain value). However, from a mathematical point of view, a perfectly good interpretation is that a mapping is a set of pairs, (d, c), where d is a

---

[3]Any number of members of the domain, including zero, may correspond to a given member of the codomain. The subset of the codomain that is mapped to by some member of the domain is called the *range* of the mapping, or the *image* of the domain under the mapping.

```
package java.util;
public interface SortedSet<T> extends Set<T> {
  /* Constructors. Classes that implement SortedSet should define
   * at least the constructors
   *    CLASS ():    An empty set ordered by natural order (compareTo).
   *    CLASS (CMP): An empty set ordered by the Comparator CMP.
   *    CLASS (C):   A set containing the items in Collection C, in
   *                 natural order.
   *    CLASS (S):   A set containing a copy of SortedSet S, with the
   *                 same order.
   */

  /** The comparator used by THIS, or null if natural ordering used. */
  Comparator<? super T> comparator ();

  /** The first (smallest) item in THIS according to its ordering */
  T first ();

  /** The last (largest) item in THIS according to its ordering */
  T last ();

  /* NOTE: The methods headSet, tailSet, and subSet produce
   * views that become invalid if THIS is structurally modified by
   * any other means. */

  /** A view of all items in THIS that are strictly less than X. */
  SortedSet<T> headSet (T x);

  /** A view of all items in THIS that are strictly >= X. */
  SortedSet<T> tailSet (T x);

  /** A view of all items, y, in THIS such that X0 <= y < X1. */
  SortedSet<T> subSet (T X0, T X1);
}
```

Figure 2.11: The interface java.util.SortedSet.

member of the domain, and c of the codomain.

2.3.1 The Map Interface

The standard Java library uses the `java.util.Map` interface, displayed in Figures 2.12 and 2.13, to capture these notions of "mapping." This interface provides both the view of a map as a look-up operation (with the method `get`), but also the view of a map as a set of ordered pairs (with the method `entrySet`). This in turn requires some representation for "ordered pair," provided here by the nested interface `Map.Entry`. A programmer who wishes to introduce a new kind of map therefore defines not only a concrete class to implement the `Map` interface, but another one to implement `Map.Entry`.

2.3.2 The SortedMap Interface

An object that implements `java.util.SortedMap` is supposed to be a `Map` in which the set of keys is ordered. As you might expect, the operations are analogous to those of the interface `SortedSet`, as shown in Figure 2.15.

2.4 An Example

Consider the problem of reading in a sequence of pairs of names, (n_i, m_i). We wish to create a list of all the first members, n_i, in alphabetical order, and, for each of them, a list of all names m_i that are paired with them, with each m_i appearing once, and listed in the order of first appearance. Thus, the input

```
John Mary   George Jeff   Tom Bert   George Paul   John Peter
Tom Jim     George Paul   Ann Cyril  John Mary     George Eric
```

might produce the output

```
Ann: Cyril
George: Jeff Paul Eric
John: Mary Peter
Tom: Bert Jim
```

We can use some kind of `SortedMap` to handle the n_i and for each, a `List` to handle the m_i. A possible method (taking a `Reader` as a source of input and a `PrintWriter` as a destination for output) is shown in Figure 2.16.

```
package java.util;
public interface Map<Key, Val> {
  /* Constructors:  Classes that implement Map should
    * have at least two constructors:
    *    CLASS (): Constructs an empty CLASS
    *    CLASS (M): Where M is any Map, constructs a CLASS that
    *        denotes the same abstract mapping as C. */

  /* Required methods:  */

  /** The number of keys in the domain of THIS map. */
  int size ();
  /** True iff size () == 0 */
  boolean isEmpty ();

  /* NOTE: The methods keySet, values, and entrySet produce views
   * that remain valid even if THIS is structurally modified. */

  /** The domain of THIS. */
  Set<Key> keySet ();
  /** The range of THIS. */
  Collection<Val> values ();
  /** A view of THIS as the set of all its (key,value) pairs. */
  Set<Map.Entry<Key, Val>> entrySet ();
  /** The value mapped to by KEY, or null if KEY is not
   *  in the domain of THIS. */
  /** True iff keySet().contains (KEY) */
  boolean containsKey (Object key);
  /** True iff values().contains (VAL). */
  boolean containsValue (Object val);
  Object get (Object key);
  /** True iff M is a Map and THIS and M represent the same mapping. */
  boolean equals (Object M);
  /** The sum of the hashCode values of all members of entrySet(). */
  int hashCode ();

  static interface Entry { ... // See Figure 2.14 }
```

Figure 2.12: Required methods of the interface `java.util.Map`.

```
// Interface java.util.Map, continued

  /* Optional methods:  */

  /** Set the domain of THIS to the empty set. */
  void clear();
  /** Cause get(KEY) to yield VAL, without disturbing other values. */
  Object put(Key key, Val val);
  /** Add all members of M.entrySet() to the entrySet() of THIS. */
  void putAll(Map<? extends Key, ? extends Val> M);
  /** Remove KEY from the domain of THIS. */
  Object remove(Object key);
}
```

Figure 2.13: Optional methods of the interface `java.util.Map`.

```
/** Represents a (key,value) pair from some Map.  In general, an Entry
 *  is associated with a particular underlying Map value.  Operations that
 *  change the Entry (specifically setValue) are reflected in that
 *  Map.  Once an entry has been removed from a Map as a result of
 *  remove or clear, further operations on it may fail. */
static interface Entry<Key,Val> {
  /** The key part of THIS. */
  Key getKey();
  /** The value part of THIS. */
  Val getValue();
  /** Cause getValue() to become VAL, returning the previous value.  */
  Val setValue(Val val);

  /** True iff E is a Map.Entry and both represent the same (key,value)
   *  pair (i.e., keys are both null, or are .equal, and likewise for
   *  values).
  boolean equals(Object e);
  /** An integer hash value that depends only on the hashCode values
   *  of getKey() and getValue() according to the formula:
   *      (getKey() == null ? 0 : getKey().hashCode ())
   *    ^ (getValue() == null ? 0 : getValue.hashCode ()) */
  int hashCode();
}
```

Figure 2.14: The nested interface `java.util.Map.Entry`, which is nested within the `java.util.Map` interface.

```
package java.util;
public interface SortedMap<Key,Val> extends Map<Key,Val> {
  /* Constructors:  Classes that implement SortedMap should
   * have at least four constructors:
   *    CLASS ():     An empty map whose keys are ordered by natural order.
   *    CLASS (CMP): An empty map whose keys are ordered by the Comparator CMP.
   *    CLASS (M):    A map that is a copy of Map M, with keys ordered
   *                  in natural order.
   *    CLASS (S):    A map  containing a copy of SortedMap S, with
   *                  keys obeying the same ordering.
   */

  /** The comparator used by THIS, or null if natural ordering used. */
  Comparator<? super Key> comparator ();

  /** The first (smallest) key in the domain of THIS according to
   *  its ordering */
  Key firstKey ();

  /** The last (largest) item in the domain of THIS according to
   *  its ordering */
  Key lastKey ();

  /* NOTE: The methods headMap, tailMap, and subMap produce views
   * that remain valid even if THIS is structurally modified. */

  /** A view of THIS consisting of the restriction to all keys in the
   *  domain that are strictly less than KEY. */
  SortedMap<Key,Val> headMap (Key key);

  /** A view of THIS consisting of the restriction to all keys in the
   *  domain that are greater than or equal to KEY. */
  SortedMap<Key,Val> tailMap (Key key);

  /** A view of THIS restricted to the domain of all keys, y,
   *  such that KEY0 <= y < KEY1. */
  SortedMap<Key,Val> subMap (Key key0, Key key1);
}
```

Figure 2.15: The interface `java.util.SortedMap`, showing methods not included
in `Map`.

```
import java.util.*;
import java.io.*;

class Example {

  /** Read (nᵢ, mᵢ) pairs from INP, and summarize all
   *  pairings for each $n_i$ in order on OUT. */
  static void correlate (Reader inp, PrintWriter out)
     throws IOException
  {
    Scanner scn = new Scanner (inp);
    SortedMap<String, List<String>> associatesMap
       = new TreeMap<String,List<String>> ();
    while (scn.hasNext ()) {
      String n = scn.next ();
      String m = scn.next ();
      if (m == null || n == null)
        throw new IOException ("bad input format");
      List<String> associates = associatesMap.get (n);
      if (associates == null) {
        associates = new ArrayList<String> ();
        associatesMap.put (n, associates);
      }
      if (! associates.contains (m))
        associates.add (m);
    }

    for (Map.Entry<String, List<String>> e : associatesMap.entrySet ()) {
      System.out.format ("%s:", e.getKey ());
      for (String s : e.getValue ())
        System.out.format (" %s", s);
      System.out.println ();
    }
  }
}
```

Figure 2.16: An example using `SortedMaps` and `Lists`.

2.5   Managing Partial Implementations: Design Options

Throughout the `Collection` interfaces, you saw (in comments) that certain opera-
tions were "optional." Their specifications gave the implementor leave to use

```
throw new UnsupportedOperationException ();
```

as the body of the operation. This provides an elegant enough way not to implement
something, but it raises an important design issue. Throwing an exception is a
dynamic action. In general, the compiler will have no comment about the fact that
you have written a program that must inevitably throw such an exception; you will
discover only upon testing the program that the implementation you have chosen
for some data structure is not sufficient.

An alternative design would split the interfaces into smaller pieces, like this:

```
public interface ConstantIterator<T> {
    Required methods of Iterator
}

public interface Iterator<T> extends ConstantIterator<T> {
    void remove ();
}

public interface ConstantCollection<T> {
    Required methods of Collection
}

public interface Collection<T> extends ConstantCollection<T> {
    Optional methods of Collection
}

public interface ConstantSet<T> extends ConstantCollection<T> {
}

public interface Set<T> extends ConstantSet<T>, Collection<T> {
}

public interface ConstantList<T> extends ConstantCollection<T> {
    Required methods of List
}

public interface List<T> extends Collection<T>, ConstantList<T> {
    Optional methods of List
}

etc....
```

With such a design the compiler could catch attempts to call unsupported methods, so that you wouldn't need testing to discover a gap in your implementation.

However, such a redesign would have its own costs. It's not quite as simple as the listing above makes it appear. Consider, for example, the `subList` method in `ConstantList`. Presumably, this would most sensibly return a `ConstantList`, since if you are not allowed to alter a list, you cannot be allowed to alter one of its views. That means, however, that the type `List` would need two `subList` methods (with differing names), the one inherited from `ConstantList`, and a new one that produces a `List` as its result, which would allow modification. Similar considerations apply to the results of the `iterator` method; there would have to be two—one to return a `ConstantIterator`, and the other to return `Iterator`. Furthermore, this proposed redesign would not deal with an implementation of `List` that allowed one to add items, or clear all items, but not remove individual items. For that, you would either still need the `UnsupportedOperationException` or an even more complicated nest of classes.

Evidently, the Java designers decided to accept the cost of leaving some problems to be discovered by testing in order to simplify the design of their library. By contrast, the designers of the corresponding standard libraries in C++ opted to distinguish operations that work on any collections from those that work only on "mutable" collections. However, they did not design their library out of interfaces; it is awkward at best to introduce new kinds of collection or map in the C++ library.

Chapter 3

Meeting a Specification

In Chapter 2, we saw and exercised a number of abstract interfaces—abstract in the sense that they describe the common features, the method signatures, of whole families of types without saying anything about the internals of those types and without providing a way to create any concrete objects that implement those interfaces.

In this chapter, we get a little closer to concrete representations, by showing one way to fill in the blanks. In one sense, these won't be serious implementations; they will use "naive," rather slow data structures. Our purpose, rather, will be one of exercising the machinery of object-oriented programming to illustrate ideas that you can apply elsewhere.

To help implementors who wish to introduce new implementations of the abstract interfaces we've covered, the Java standard library provides a parallel collection of abstract classes with some methods filled in. Once you've supplied a few key methods that remain unimplemented, you get all the rest "for free". These partial implementation classes are not intended to be used directly in most ordinary programs, but only as implementation aids for library writers. Here is a list of these classes and the interfaces they partially implement (all from the package `java.util`):

Abstract Class	Interfaces
AbstractCollection	Collection
AbstractSet	Collection, Set
AbstractList	Collection, List
AbstractSequentialList	Collection, List
AbstractMap	Map

The idea of using partial implementations in this way is an instance of a design pattern called Template Method. The term *design pattern* in the context of object-oriented programming has come to mean "the core of a solution to a particular commonly occurring problem in program design[1]." The `Abstract...` classes are

---

[1] The seminal work on the topic is the excellent book by E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995. This group and their book are often referred to as "The Gang of Four."

```java
import java.util.*;
import java.lang.reflect.Array;
public class ArrayCollection<T> implements Collection<T> {
    private T[] data;

    /** An empty Collection */
    public ArrayCollection () { data = (T[]) new Object[0]; }

    /** A Collection consisting of the elements of C */
    public ArrayCollection (Collection<? extends T> C) {
        data = C.toArray((T[]) new Object[C.size ()]);
    }

    /** A Collection consisting of a view of the elements of A. */
    public ArrayCollection (T[] A) { data = T; }

    public int size () { return data.length; }

    public Iterator<T> iterator () {
        return new Iterator<T> () {
            private int k = 0;
            public boolean hasNext () { return k < size (); }
            public T next () {
                if (! hasNext ()) throw new NoSuchElementException ();
                k += 1;
                return data[k-1];
            }
            public void remove () {
                throw new UnsupportedOperationException ();
            }
        };
    }

    public boolean isEmpty () { return size () == 0; }

    public boolean contains (Object x) {
        for (T y : this) {
            if (x == null && y == null
                    || x != null && x.equals (y))
                return true;
        }
        return false;
    }
```

Figure 3.1: Implementation of a new kind of read-only `Collection` "from scratch."

```
public boolean containsAll (Collection<?> c) {
  for (Object x : c)
    if (! contains (x))
      return false;
  return true;
}

public Object[] toArray () { return toArray (new Object[size ()]); }

public <E> E[] toArray (E[] anArray) {
  if (anArray.length < size ()) {
    Class<?> typeOfElement = anArray.getClass ().getComponentType ();
    anArray = (E[]) Array.newInstance (typeOfElement, size ());
  }
  System.arraycopy (anArray, 0, data, 0, size ());
  return anArray;
}

private boolean UNSUPPORTED () {
  throw new UnsupportedOperationException ();
}

public boolean add (T x) { return UNSUPPORTED (); }
public boolean addAll (Collection<? extends T> c) { return UNSUPPORTED (); }
public void clear () { UNSUPPORTED (); }
public boolean remove (Object x) { return UNSUPPORTED (); }
public boolean removeAll (Collection<?> c) { return UNSUPPORTED (); }
public boolean retainAll (Collection<?> c) { return UNSUPPORTED (); }
}
```

Figure 3.1, continued: Since this is a read-only collection, the methods for modifying the collection all throw `UnsupportedOperationException`, the standard way to signal unsupported features.

used as templates for real implementations. Using method overriding, the implementor fills in a few methods; everything else in the template uses those methods[2].

In the sections to follow, we'll look at how these classes are used and we'll look at some of their internals for ideas about how to use some of the features of Java classes. But first, let's have a quick look at the alternative.

3.1   Doing it from Scratch

For comparison, let's suppose we wanted to introduce a simple implementation that simply allowed us to treat an ordinary array of Objects as a read-only `Collection`. The direct way to do so is shown in Figure 3.1. Following the specification of `Collection`, the first two constructors for `ArrayCollection` provide ways of forming an empty collection (not terribly useful, of course, since you can't add to it) and a copy of an existing collection. The third constructor is specific to the new class, and provides a view of an array as a `Collection`—that is, the items in the `Collection` are the elements of the array, and the operations are those of the `Collection` interface. Next come the required methods. The `Iterator` that is returned by `iterator` has an anonymous type; no user of `ArrayCollection` can create an object of this type directly. Since this is a read-only collection, the optional methods (which modify collections) are all unsupported.

A Side Excursion on Reflection.   The implementation of the second `toArray` method is rather interesting, in that it uses a fairly exotic feature of the Java language: *reflection.* This term refers to language features that allow one to manipulate constructs of a programming language within the language itself. In English, we employ reflection when we say something like "The word 'hit' is a verb." The specification of `toArray` calls for us to produce an array of the same dynamic type as the argument. To do so, we first use the method `getClass`, which is defined on all `Objects`, to get a value of the built-in type `java.lang.Class` that stands for (*reflects*) the dynamic type of the `anArray` argument. One of the operations on type `Class` is `getComponentType`, which, for an array type, fetches the `Class` that reflects the type of its elements. Finally, the `newInstance` method (defined in the class `java.lang.reflect.Array`) creates a new array object, given its size and the `Class` for its component type.

3.2   The AbstractCollection Class

The implementation of `ArrayCollection` has an interesting feature: the methods starting with `isEmpty` make no mention of the private data of `ArrayCollection`,

---

[2]While the name *Template Method* may be appropriate for this design pattern, I must admit that it has some unfortunate clashes with other uses of the terminology. First, the library defines whole *classes,* while the name of the pattern focuses on individual methods within that class. Second, the term *template* has another meaning within object-oriented programming; in C++ (and apparently in upcoming revisions of Java), it refers to a particular language construct.

but instead rely entirely on the other (public) methods. As a result, they could be employed verbatim in the implementation of *any* `Collection` class. The standard Java library class `AbstractCollection` exploits this observation (see Figure 3.2). It is a partially implemented abstract class that new kinds of `Collection` can extend. At a bare minimum, an implementor can override just the definitions of `iterator` and `size` to get a read-only collection class. For example, Figure 3.3 shows an easier re-write of `ArrayCollection`. If, in addition, the programmer overrides the `add` method, then `AbstractCollection` will automatically provide `addAll` as well. Finally, if the `iterator` method returns an `Iterator` that supports the `remove` method, then `AbstractCollection` will automatically provide `clear`, `remove`, `removeAll`, and `retainAll`.

In programs, the idea is to use `AbstractCollection` *only* in an **extends** clause. That is, it is simply a utility class for the benefit of implementors creating new kinds of `Collection`, and should not generally be used to specify the type of a formal parameter, local variable, or field. This, by the way, is the explanation for declaring the constructor for `AbstractCollection` to be **protected**; that keyword emphasizes the fact that only extensions of `AbstractClass` will call it.

You've already seen five examples of how `AbstractCollection` might work in Figure 3.1: methods `isEmpty`, `contains`, `containsAll`, and the two `toArray` methods. Once you get the general idea, it is fairly easy to produce such method bodies The exercises ask you to produce a few more.

3.3 Implementing the List Interface

The abstract classes `AbstractList` and `AbstractSequentialList` are specialized extensions of the class `AbstractCollection` provided by the Java standard library to help define classes that implement the `List` interface. Which you choose depends on the nature of the representation used for the concrete list type being implemented.

3.3.1 The AbstractList Class

The abstract implementation of `List`, `AbstractList`, sketched in Figure 3.4 is intended for representations that provide fast (generally constant time) *random access* to their elements—that is, representations with a fast implementation of `get` and (if supplied) `remove` Figure 3.5 shows how `listIterator` works, as a partial illustration. There are a number of interesting techniques illustrated by this class.

Protected methods. The method `removeRange` is not part of the public interface. Since it is declared **protected**, it may only be called within other classes in the package `java.util`, and within the bodies of extensions of `AbstractList`. Such methods are *implementation utilities* for use in the class and its extensions. In the standard implementation of `AbstractList`, `removeRange` is used to implement `clear` (which might not sound too important until you remember that `L.subList(k0,k1).clear()` is how one removes an arbitrary section of a `List`).

```
package java.util;
public abstract class AbstractCollection<T> implements Collection<T> {
  /** The empty Collection. */
  protected AbstractCollection<T> () { }

  /** Unimplemented methods that must be overridden in any
   *  non-abstract class that extends AbstractCollection */

  /** The number of values in THIS. */
  public abstract int size ();

  /** An iterator that yields all the elements of THIS, in some
   *  order.    If the remove operation is supported on this iterator,
   *  then remove, removeAll, clear, and retainAll on THIS will work. */
  public abstract Iterator<T> iterator ();

  /** Override this default implementation to support adding */
  public boolean add (T x) {
     throw new UnsupportedOperationException ();
  }

  Default, general-purpose implementations of
     contains (Object x), containsAll (Collection c), isEmpty (),
     toArray (), toArray (Object[] A),
     addAll (Collection c), clear (), remove (Object x),
     removeAll (Collection c), and retainAll (Collection c)

  /** A String representing THIS, consisting of a comma-separated
   *  list of the values in THIS, as returned by its iterator,
   *  surrounded by square brackets ([]).   The elements are
   *  converted to Strings by String.valueOf (which returns "null"
   *  for the null pointer and otherwise calls the .toString() method).  */
  public String toString () { ... }
}
```

Figure 3.2: The abstract class `java.util.AbstractCollection`, which may be used to help implement new kinds of `Collection`. All the methods behave as specified in the specification of `Collection`. Implementors must fill in definitions of `iterator` and `size`, and may either override the other methods, or simply use their default implementations (not shown here).

```java
import java.util.*;
/** A read-only Collection whose elements are those of an array. */
public class ArrayCollection<T> extends AbstractCollection<T> {
  private T[] data;

  /** An empty Collection */
  public ArrayCollection () {
     data = (T[]) new Object[0];
  }

  /** A Collection consisting of the elements of C */
  public ArrayCollection (Collection<? extends T> C) {
     data = C.toArray(new Object[C.size ()]);
  }

  /** A Collection consisting of a view of the elements of A. */
  public ArrayCollection (Object[] A) {
     data = A;
  }

  public int size () { return data.length; }

  public Iterator<T> iterator () {
    return new Iterator<T> () {
      private int k = 0;
      public boolean hasNext () { return k < size (); }
      public T next () {
         if (! hasNext ()) throw new NoSuchElementException ();
         k += 1;
         return data[k-1];
      }
      public void remove () {
         throw new UnsupportedOperationException ();
      }
    };
  }
}
```

Figure 3.3: Re-implementation of `ArrayCollection`, using the default implementations from `java.util.AbstractCollection`.

The default implementation of `removeRange` simply calls `remove(k)` repeatedly and so is not particularly fast. But if a particular `List` representation allows some better strategy, then the programmer can override `removeRange`, getting better performance for `clear` (that's why the default implementaion of the method is not declared **final**, even though it is written to work for any representation of `List`).

Checking for Invalidity.   As we discussed in §2.2.3, the `iterator`, `listIterator`, and `subList` methods of the `List` interface produce views of a list that "become invalid" if the list is structurally changed. Implementors of `List` are under no particular obligation to do anything sensible for the programmer who ignores this provision; using an invalidated view may produce unpredictable results or throw an unexpected exception, as convenient. Nevertheless, the `AbstractList` class goes to some trouble to provide a way to explicitly check for this error, and immediately throw a specific exception, `ConcurrentModificationException`, if it happens. The field `modCount` (declared **protected** to indicate it is intended for List implementors, not users) keeps track of the number of structural modifications to an `AbstractList`. Every call to `add` or `remove` (either on the List directly or through a view) is supposed to increment it. Individual views can then keep track of the last value they "saw" for the `modCount` field of their underlying List and throw an exception if it seems to have changed in the interim. We'll see an example in Figure 3.5.

Helper Classes.   The `subList` method of `AbstractList` (at least in Sun's implementation) uses a non-public utility type `java.util.SubList` to produce its result. Because it is not public, `java.util.SubList` is in effect private to the `java.util` package, and is not an official part of the services provided by that package. However, being in the same package, it is allowed to access the non-public fields (`modCount`) and utility methods (`removeRange`) of `AbstractList`. This is an example of Java's mechanism for allowing "trusted" classes (those in the same package) access to the internals of a class while excluding access from other "untrusted" classes.

3.3.2   The AbstractSequentialList Class

The second abstract implementation of `List`, `AbstractSequentialList` (Figure 3.6), is intended for use with representations where random access is relatively slow, but the `next` operation of the list iterator is still fast.

The reason for having a distinct class for this case becomes clear when you consider the implementations of `get` and the `next` methods of the iterators. If we assume a fast `get` method, then it is easy to implement the iterators to have fast `next` methods, as was shown in Figure 3.5. If `get` is slow—specifically, if the only way to retrieve item k of the list is to sequence through the preceding k items—then implementing `next` as in that figure would be disasterous; it would require $\Theta(N^2)$ operations to iterate through an N-element list. So using `get` to implement the iterators is not always a good idea.

```
package java.util;
public abstract class AbstractList<T>
    extends AbstractCollection<T> implements List<T> {

  /** Construct an empty list. */
  protected AbstractList () { modCount = 0; }

  abstract T get (int index);
  abstract int size ();

  T set (int k, T x)  { return UNSUPPORTED (); }

  void add (int k, T x) { UNSUPPORTED (); }

  T remove (int k) { return UNSUPPORTED (); }
```

Default, general-purpose implementations of
     `add (x), addAll, clear, equals, hashCode, indexOf, iterator,`
     `lastIndexOf, listIterator, set, and subList`

```
  /** The number of times THIS has had elements added or removed. */
  protected int modCount;

  /** Remove from THIS all elements with indices in the
      range K0 .. K1-1. */
  protected void removeRange (int k0, int k1) {
     ListIterator<T> i = listIterator (k0);
     for (int k = k0; k < k1 && i.hasNext (); k += 1) {
        i.next (); i.remove ();
     }
  }

  private Object UNSUPPORTED ()
    { throw new UnsupportedOperationException (); }
}
```

Figure 3.4: The abstract class `AbstractList`, used as an implementation aid in writing implementations of `List` that are intended for random access. See Figure 3.5 for the inner class `ListIteratorImpl`.

```
public ListIterator<T> listIterator (int k0) {
  return new ListIteratorImpl (k0);
}

private class ListIteratorImpl<T> implements ListIterator<T> {
  ListIteratorImpl (int k0)
    { lastMod = modCount; k = k0; lastIndex = -1; }

  public boolean hasNext () { return k < size (); }
  public hasPrevious () { return k > 0; }

  public T next () {
    check (0, size ());
    lastIndex = k; k += 1; return get (lastIndex);
  }
  public T previous () {
    check (1, size ()+1);
    k -= 1; lastIndex = k; return get (k);
  }
  public int nextIndex () { return k; }
  public int previousIndex () { return k-1; }

  public void add (T x) {
    check (); lastIndex = -1;
    k += 1; AbstractList.this.add (k-1, x);
    lastMod = modCount;
  }

  public void remove () {
    checkLast (); AbstractList.this.remove (lastIndex);
    lastIndex = -1; lastMod = modCount;
  }

  public void set (T x) {
    checkLast (); AbstractList.this.remove (lastIndex, x);
    lastIndex = -1; lastMod = modCount;
  }
```

Figure 3.5: Part of a possible implementation of `AbstractList`, showing the inner class providing the value of `listIterator`.

```
// Class AbstractList.ListIteratorImpl, continued.

/* Private definitions */

/** modCount value expected for underlying list. */
private int lastMod;
/** Current position. */
private int k;
/** Index of last result returned by next or previous. */
private int lastIndex;

/** Check that there has been no concurrent modification. Throws
 *  appropriate exception if there has. */
private void check () {
  if (modCount != lastMod) throw new ConcurrentModificationException();
}

/** Check that there has been no concurrent modification and that
 *  the current position, k, is in the range K0 <= k < K1.  Throws
 *  appropriate exception if either test fails. */
private void check (int k0, int k1) {
  check ();
  if (k < k0 || k >= k1)
     throw new NoSuchElementException ();
}

/** Check that there has been no concurrent modification and that
 *  there is a valid ``last element returned by next or previous''.
 *  Throws appropriate exception if either test fails. */
private checkLast () {
  check ();
  if (lastIndex == -1) throw new IllegalStateException ();
}
```

Figure 3.5, continued: Private representation of the `ListIterator`.

```
public abstract class AbstractSequentialList<T> extends AbstractList<T> {
  /** An empty list */
  protected AbstractSequentialList () { }

  abstract int size ();
  abstract ListIterator<T> listIterator (int k);

  Default implementations of
      add(k,x), addAll(k,c), get, iterator, remove(k), set

  From AbstractList, inherited implementations of
      add(x), clear, equals, hashCode, indexOf, lastIndexOf,
      listIterator(), removeRange, subList

  From AbstractCollection, inherited implementations of
      addAll(), contains, containsAll, isEmpty, remove(), removeAll,
      retainAll, toArray, toString
}
```

Figure 3.6: The class `AbstractSequentialList`.

On the other hand, if we were always to implement `get(k)` by iterating over the preceding k items (that is, use the `Iterator`'s methods to implement `get` rather than the reverse), we would obviously lose out on representations where `get` *is* fast.

3.4   The AbstractMap Class

The `AbstractMap` class shown in Figure 3.7 provides a template implementation for the `Map` interface. Overriding just the `entrySet` to provide a read-only `Set` gives a read-only `Map`. Additionally overriding the `put` method gives an extendable `Map`, and implementing the `remove` method for `entrySet().iterator()` gives a fully modifiable `Map`.

3.5   Performance Predictions

At the beginning of Chapter 2, I said that there are typically several implementations for a given interface. There are several possible reasons one might need more than one. First, special kinds of stored items, keys, or values might need special handling, either for speed, or because there are extra operations that make sense only for these special kinds of things. Second, some particular `Collections` or `Maps` may need a special implementation because they are part of something else, such as the `subList` or `entrySet` views. Third, one implementation may perform

```
package java.util;
public abstract class AbstractMap<Key,Val> implements Map<Key,Val> {
   /** An empty map. */
  protected AbstractMap () { }

  /** A view of THIS as the set of all its (key,value) pairs.
   *  If the resulting Set's iterator supports remove, then THIS
   *  map will support the remove and clear operations.  */
  public abstract Set<Entry<Key,Val>> entrySet ();

  /** Cause get(KEY) to yield VAL, without disturbing other values. */
  public Val put (Key key, Val val) {
     throw new UnsupportedOperationException ();
  }

  Default implementations of
      clear, containsKey, containsValue, equals, get, hashCode,
      isEmpty, keySet, putAll, remove, size, values

  /** Print a String representation of THIS, in the form
   *     {KEY0=VALUE0, KEY1=VALUE1, ...}
   *  where keys and values are converted using String.valueOf(...). */
  public String toString () { ... }
}
```

Figure 3.7: The class `AbstractMap`.

better than another in some circumstances, but not in others. Finally, there may be time-vs.-space tradeoffs between different implementations, and some applications may have particular need for a compact (space-efficient) representation.

We can't make specific claims about the performance of the `Abstract...` family of classes described here because they are templates rather than complete implementations. However, we can characterize their performance as a function of the methods that the programmer fills in. Here, let's consider two examples: the implementation templates for the `List` interface.

AbstractList. The strategy behind `AbstractList` is to use the methods `size`, `get(k)`, `add(k,x)`, `set(k,x)`, and `remove(k)` supplied by the extending type to implement everything else. The `listIterator` method returns a `ListIterator` that uses `get` to implement `next` and `previous`, `add` (on `AbstractList`) to implement the iterator's `add`, and `remove` (on `AbstractList`) to implement the iterator's `remove`. The cost of the additional bookkeeping done by the iterator consists of incrementing or decrementing an integer variable, and is therefore a small constant. Thus, we can easily relate the costs of the iterator functions directly to those of the supplied methods, as shown in the following table. To simplify matters, we take the time costs of the `size` operation and the `equals` operation on individual items to be constant. The values of the "plugged-in" methods are given names of the form C_α; the size of **this** (the `List`) is N, and the size of the other Collection argument (denoted `c`, which we'll assume is the same kind of `List`, just to be able to say more) is M.

Costs of `AbstractList` Implementations

List		ListIterator	
Method	Time as $\Theta(\cdot)$	Method	Time as $\Theta(\cdot)$
add(k,X)	C_a	add	C_a
get(k)	C_g	remove	C_r
remove(k)	C_r	next	C_g
set	C_s	previous	C_g
remove(X)	$C_r + N \cdot C_g$	set	C_s
indexOf	$N \cdot C_g$	hasNext	1
lastIndexOf	$N \cdot C_g$		
listIterator(k)	1		
iterator()	1		
subList	1		
size	1		
isEmpty	1		
contains	$N \cdot C_g$		
containsAll(c)	$N \cdot M \cdot C_g$		
addAll(c)	$M \cdot C_g + (N + M) \cdot C_a$		
toArray	$N \cdot C_g$		

AbstractSequentialList. Let's now compare the `AbstractList` implementation with `AbstractSequentialList`, which is intended to be used with representations that don't have cheap `get` operations, but still do have cheap iterators. In this case, the `get(k)` operation is implemented by creating a `ListIterator` and performing a `next` operation on it k times. We get the following table:

Costs of `AbstractList` Implementations

List			ListIterator	
Method	Time as $\Theta(\cdot)$		Method	Time as $\Theta(\cdot)$
add(k,X)	$C_a + k \cdot C_n$		add	C_a
get(k)	$k \cdot C_n$		remove	C_r
remove(k)	$C_r + k \cdot C_n$		next	C_n
set(k,X)	$C_s + k \cdot C_n$		previous	C_p
remove(X)	$C_r + N \cdot C_g$		set	C_s
indexOf	$N \cdot C_n$		hasNext	1
lastIndexOf	$N \cdot C_p$			
listIterator(k)	$k \cdot C_n$			
iterator()	1			
subList	1			
size	1			
isEmpty	1			
contains	$N \cdot C_n$			
containsAll(c)	$N \cdot M \cdot C_n$			
addAll(c)	$M \cdot C_n + N \cdot C_a$			
toArray	$N \cdot C_n$			

Exercises

3.1. Provide a body for the `addAll` method of `AbstractCollection`. It can assume that `add` will either throw `UnsupportedOperationException` if adding to the Collection is not supported, or will add an element.

3.2. Provide a body for the `removeAll` method of `AbstractCollection`. You may assume that, if removal is supported, the `remove` operation of the result of `iterator` works.

3.3. Provide a possible implementation of the `java.util.SubList` class. This utility class implements `List` and has one constructor:

```
/** A view of items K0 throught K1-1 of THELIST.  Subsequent
 *  modifications to THIS also modify THELIST.  Any structural
 *  modification to THELIST other than through THIS and any
 *  iterators or sublists derived from it renders THIS invalid.
 *  Operations on an invalid SubList throw
 *  ConcurrentModificationException */
SubList (AbstractList theList, int k0, int k1) { ... }
```

3.4. For class `AbstractSequentialList`, provide possible implementations of `add(k,x)` and `get`. Arrange the implementation so that performing a `get` of an element at or near *either end* of the list is fast.

3.5. Extend the `AbstractMap` class to produce a full implementation of `Map`. Try to leave as much as possible up to `AbstractMap`, implementing just what you need. For a representation, provide an implementation of `Map.Entry` and then use the existing implementation of `Set` provided by the Java library, `HashSet`. Call the resulting class `SimpleMap`.

3.6. In §3.5, we did not talk about the performance of operations on the `List`s returned by the `subList` method. Provide these estimates for both `AbstractList` and `AbstractSequentialList`. For `AbstractSequentialList`, the time requirement for the `get` method on a sublist *must* depend on the the first argument to `subList` (the starting point). Why is this? What change to the definition of `ListIterator` could make the performance of `get` (and other operations) on sublists independent on where in the original list the sublist comes from?

Chapter 4

Sequences and Their Implementations

In Chapters 2 and 3, we saw quite a bit of the `List` interface and some skeleton implementations. Here, we review the standard representations (concrete implementations) of this interface, and also look at interfaces and implementations of some specialized versions, the *queue* data structures.

4.1   Array Representation of the List Interface

Most "production" programming languages have some built-in data structure like the Java array—a random-access sequence of variables, indexed by integers. The array data structure has two main performance advantages. First, it is a compact (space-efficient) representation for a sequence of variables, typically taking little more space than the constituent variables themselves. Second, random access to any given variable in the sequence is a fast, constant-time operation. The chief disadvantage is that changing the size of the sequence represented is slow (in the worst case). Nevertheless, with a little care, we will see that the *amortized cost* of operations on array-backed lists is constant.

One of the built-in Java types is `java.util.ArrayList`, which has, in part, the implementation shown in Figure 4.1[1]. So, you can create a new `ArrayList` with its constructors, optionally choosing how much space it initially has. Then you can add items (with `add`), and the array holding these items will be expanded as needed.

What can we say about the cost of the operations on `ArrayList`? Obviously, `get` and `size` are $\Theta(1)$; the interesting one is `add`. As you can see, the capacity of an `ArrayList` is always positive. The implementation of `add` uses `ensureCapacity` whenever the array pointed to by `data` needs to expand, and it requests that the

---

[1]The Java standard library type `java.util.Vector` provides essentially the same representation. It predates `ArrayList` and the introduction of Java's standard `Collection` classes, and was "retrofitted" to meet the `List` interface. As a result, many existing Java programs tend to use `Vector`, and tend to use its (now redundant) pre-`List` operations, such as `elementAt` and `removeAllElements` (same as `get` and `clear`). The `Vector` class has another difference: it is *synchronized,* whereas `ArrayList` is not. See §10.1 for further discussion.

capacity of the `ArrayList`—the size of `data`—should *double* whenever it needs to expand. Let's look into the reason behind this design choice. We'll consider just the call `A.add(x)`, which calls `A.add(A.size(), x)`.

Suppose that we replace the lines

```
if (count + 1 > data.length)
  ensureCapacity (data.length * 2);
```

with the alternative minimal expansion:

```
ensureCapacity (count+1);
```

In this case, once the initial capacity is exhausted, each `add` operation will expand the array `data`. Let's measure the cost of `add` in number of assignments to array elements [why is that reasonable?]. In Java, we can take the cost of the expression `new Object[K]` to be $\Theta(K)$. This does not change when we add in the cost of copying elements from the previous array into it (using `System.arraycopy`). Therefore, the worst-case cost, C_i, of executing `A.add(x)` using our simple increment-size-by-1 scheme is

$$C_i(K, M) = \begin{cases} \alpha_1, & \text{if } M > K; \\ \alpha_2(K+1), & \text{if } M = K \end{cases}$$

where K is `A.size()`, $M \geq K$ is `A`'s current capacity (that is, `A.data.length`), and the α_i are some constants. So, conservatively speaking, we can just say that $C(K, M, I) \in \Theta(K)$.

Now let's consider the cost, C_d, of the implementation as shown in Figure 4.1, where we always double the capacity when it must be increased. This time, we get

$$C_d(K, M) = \begin{cases} \alpha_1, & \text{if } M > K; \\ \alpha_3(2K+1), & \text{if } M = K \end{cases}$$

the worst-case cost looks identical; the factor of two increase in size simply changes the constant factor, and we can still use the same formula as before: $C_d(K, M) \in O(K)$.

So from this naïve worst-case asymptotic point of view, it would appear the two alternative strategies have identical costs. Yet we ought to be suspicious. Consider an entire *series* of `add` operations together, rather than just one. With the increment-size-by-1 strategy, we expand every time. By constrast, with the size-doubling strategy, we expand less and less often as the array grows, so that most calls to `add` complete in constant time. So is it really accurate to characterize them as taking time proportional to K?

Consider a series of N calls to `A.add(x)`, starting with `A` an empty `ArrayList` with initial capacity of $M_0 < N$. With the increment-by-1 strategy, call number M_0, (numbering from 0), M_0+1, M_0+2, etc. will cost time proportional to M_0+1, $M_0 + 2$, ..., respectively. Let's use

$$C_i^*(N, M_0)$$

```
package java.util;
/** A List with a constant-time get operation. At any given time,
 *   an ArrayList has a capacity, which indicates the maximum
 *   size() for which the one-argument add operation (which adds to
 *   the end) will execute in constant time.   The capacity expands
 *   automatically as needed so as to provide constant amortized
 *   time for the one-argument add operation. */
public class ArrayList extends AbstractList implements Cloneable {

  /** An empty ArrayList whose capacity is initially at least
   *  CAPACITY.  */
  public ArrayList(int capacity) {
    data = new Object[Math.max (capacity, 2)]; count = 0;
  }

  public ArrayList () { this (8); }

  public ArrayList (Collection c) {
    this (c.size ()); addAll (c);
  }

  public int size() { return count; }

  public Object get (int k) {
    check (k, count); return data[k];
  }

  public Object remove (int k) {
    Object old = data[k];
    removeRange (k, k+1);
    return old;
  }

  public Object set (int k, Object x) {
    check (k, count);
    Object old = data[k];
    data[k] = x;
    return old;
  }
```

Figure 4.1: Implementation of the class `java.util.ArrayList`.

```java
public void add (int k, Object obj) {
  check (k, count+1);
  if (count + 1 > data.length)
    ensureCapacity (data.length * 2);
  System.arraycopy (data, k, data, k+1, count - k);
  data[k] = obj; count += 1;
}

/* Cause the capacity of this ArrayList to be at least N. */
public void ensureCapacity (int N) {
  if (N <= data.length)
    return;
  Object[] newData = new Object[N];
  System.arraycopy (data, 0, newData, 0, count);
  data = newData;
}

/** A copy of THIS (overrides method in Object). */
public Object clone () { return new ArrayList (this); }

protected void removeRange (int k0, int k1) {
  if (k0 >= k1)
    return;
  check (k0, count); check (k1, count+1);
  System.arraycopy (data, k1, data, k0, count - k1);
  count -= k1-k0;
}

private void check (int k, int limit) {
  if (k < 0 || k >= limit)
    throw new IndexOutOfBoundsException ();
}

private int count;            /* Current size */
private Object[] data;        /* Current contents */
}
```

Figure 4.1, continued.

to stand for the cost of a sequence of N `add(x)` operations on an `ArrayList` with initial capacity M_0, starting from an empty list, using the increment-by-1 strategy. This is clearly

$$
\begin{aligned}
C_i^*(N, M_0) &= C_i(0, M_0) + \ldots + C_i(M_0 - 1, M_0) \\
&\quad + C_i(M_0, M_0) + C_i(M_0 + 1, M_0 + 1) + \ldots + C_i(N - 1, N - 1) \\
&= \alpha_1 \cdot M_0 \\
&\quad + \alpha_2 \cdot (M_0 + 1 + M_0 + 2 + \ldots + N) \\
&= M_0 \cdot (\alpha_1 + \alpha_2(N - M_0)) + \alpha_2 \cdot (1 + 2 + \ldots + N - M_0) \\
&= M_0 \cdot (\alpha_1 + \alpha_2(N - M_0)) + \alpha_2 \cdot (N - M_0)(N - M_0 + 1)/2 \\
&\in \Theta(N^2), \text{ for fixed } M_0.
\end{aligned}
$$

The cost, in other words, is quadratic in the number of items added.

Now consider the doubling strategy. In this case, calls number M_0, $2M_0$, $4M_0$, etc., will incur costs proportional to $2M_0$, $4M_0$, $8M_0$, etc. (cost of doubling the array and copying the original elements to it). All other calls will have constant cost α_1. For simplicity, consider just sequences that end on a call that expands the arrays. That is, $N = 2^k M_0 + 1$ for some k. For fixed M_0:

$$
\begin{aligned}
C_d^*(N, M_0) &= C_d^*(2^k M_0 + 1, M_0) \\
&= \underbrace{\alpha_1 M_0}_{\text{calls \#0 to } M_0 - 1} + \underbrace{\alpha_3 2M_0}_{\text{call \#} M_0} + \underbrace{\alpha_1 (M_0 - 1)}_{\text{calls } M_0 + 1 \text{ to } 2M_0 - 1} \\
&\quad + \underbrace{\alpha_3 4M_0}_{\text{call \#} 2M_0} + \underbrace{\alpha_1 (2M_0 - 1)}_{\text{calls \#} 2M_0 + 1 \text{ to } 4M_0 - 1} + \underbrace{\alpha_3 8M_0}_{\text{call \#} 4M_0} \\
&\quad + \ldots + \underbrace{\alpha_1 (2M_0 - 1)}_{\text{calls \#} 2^{k-1} M_0 + 1 \text{ to } 2^k M_0 - 1} + \underbrace{\alpha_3 2^{k+1} M_0}_{\text{call \#} 2^k M_0} \\
&= \alpha_1 (M_0 + M_0 + 2M_0 + 4M_0 + \ldots + 2^{k-1} M_0 - k + 1) \\
&\quad + \alpha_3 (2M_0 + 4M_0 + \ldots + 2^{k+1} M_0) \\
&= M_0 (\alpha_1 2^k + \alpha_3 (2^{k+2} - 3)) - k + 1 \\
&\in \Theta(2^{k+2}) = \Theta(2^k) = \Theta(2^k M_0 + 1) = \Theta(N)
\end{aligned}
$$

Surprise! The total time is linear in the total number of `add`s. If we credit the total cost of N `add`s equally to all N operations, each gets a *constant* time. For the doubling strategy, `add` has *constant-time amortized cost.*

4.2 Linking in Sequential Structures

The term *linked structure* refers generally to composite, dynamically growable data structures comprising small objects for the individual members connected together by means of pointers (*links*).

4.2.1 Singly Linked Lists

The Scheme language has one pervasive compound data structure, the *pair* or *cons cell,* which can serve to represent just about any data structure one can imagine. Perhaps its most common use is in representing lists of things, as illustrated in Figure 4.2a. Each pair consists of two containers, one of which is used to store a (pointer to) the data item, and the second a pointer to the next pair in the list, or a null pointer at the end. In Java, a rough equivalent to the pair is a class such as the following:

```
class Entry {
  Entry (Object head, Entry next) {
      this.head = head; this.next = next;
  }
  Object head;
  Entry next;
}
```

We call lists formed from such pairs *singly linked,* because each pair carries one pointer (*link*) to another pair.

Changing the structure of a linked list (its set of containers) involves what is colloquially known as "pointer swinging." Figure 4.2 reviews the basic operations for insertion and deletion on pairs used as lists.

4.2.2 Sentinels

As Figure 4.2 illustrates, the procedure for inserting or deleting at the beginning of a linked list differs from the procedure for middle items, because it is the variable L, rather than a `next` field that gets changed:

```
L = L.next;   // Remove first item of linked list pointed to by L
L = new Entry ("aardvark", L);  // Add item to front of list.
```

We can avoid this recourse to special cases for the beginning of a list by employing a clever trick known as a *sentinel node.*

The idea behind a sentinel is to use an extra object, one that does not carry one of the items of the collection being stored, to avoid having any special cases. Figure 4.3 illustrates the resulting representation.

Use of sentinels changes some tests. For example, testing to see if linked list L list is empty without a sentinel is simply a matter of comparing L to null, whereas the test for a list with a sentinel compares `L.next` to null.

4.2.3 Doubly Linked Lists

Singly linked lists are simple to create and manipulate, but they are at a disadvantage for fully implementing the Java `List` interface. One obvious problem is that the `previous` operation on list iterators has no fast implementation on singly linked structures. One is pretty much forced to return to the start of the list and
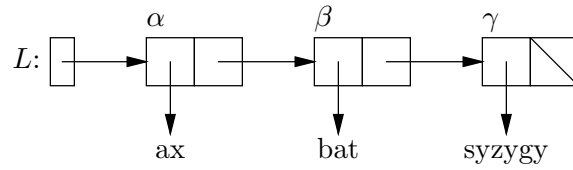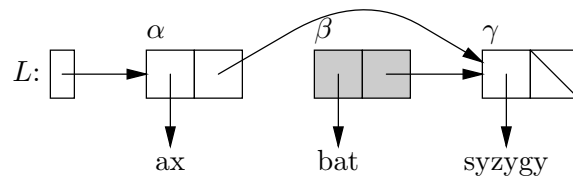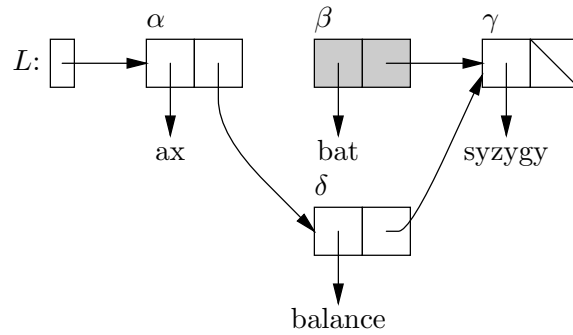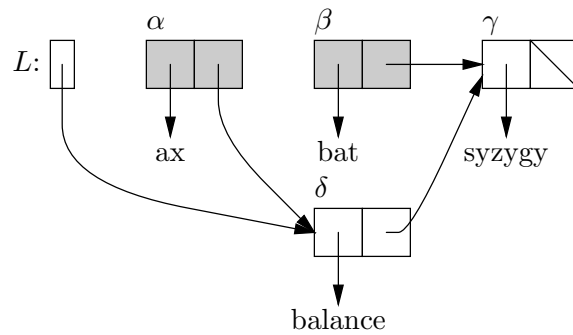
(a) Original list

(b) After removing `bat` with `L.next = L.next.next`

(c) After adding `balance` with `L.next = new Entry("balance", L.next)`

(d) After deleting `ax` with `L = L.next`

Figure 4.2: Common operations on the singly linked list representation. Starting from an initial list, we remove object β, and then insert in its place a new one. Next we remove the first item in the list. The objects removed become "garbage," and are no longer reachable via L.
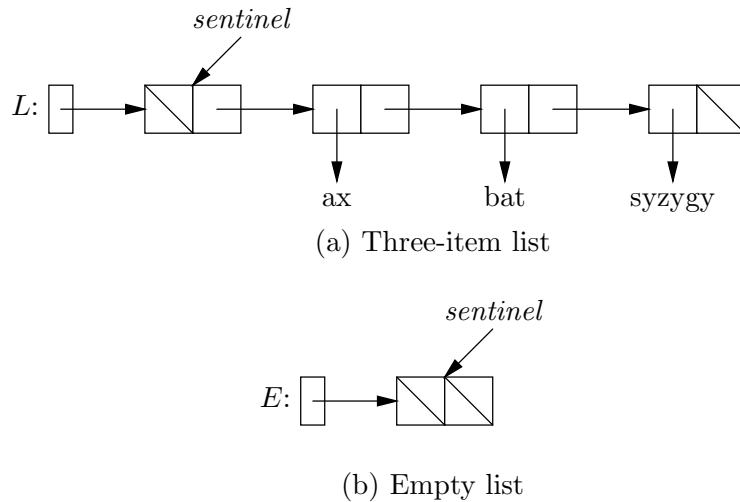
(a) Three-item list

(b) Empty list

Figure 4.3: Singly linked lists employing sentinel nodes. The sentinels contain no useful data. They allow all items in the list to treated identically, with no special case for the first node. The sentinel node is typically never removed or replaced while the list is in use.

follow an appropriate number of `next` fields to almost but not quite return to the current position, requiring time proportional to the size of the list. A somewhat more subtle annoyance comes in the implementation of the `remove` operation on the list iterator. To remove an item `p` from a singly linked list, you need a pointer to the item *before* `p`, because it is the `next` field of that object that must be modified.

Both problems are easily solved by adding a *predecessor* link to the objects in our list structure, making both the items before and after a given item in the list equally accessible. As with singly linked structures, the use of front and end sentinels further simplifies operations by removing the special cases of adding to or removing from the beginning or end of a list. A further device in the case of doubly linked structures is to make the entire list *circular,* that is, to use one sentinel as both the front and back of the list. This cute trick saves the small amount of space otherwise wasted by the `prev` link of the front sentinel and the `next` link of the last. Figure 4.4 illustrates the resulting representations and the principal operations upon it.

4.3   Linked Implementation of the List Interface

The doubly linked structure supports everything we need to do to implement the Java `List` interface. The type of the links (`LinkedList.Entry`) is private to the implementation. A `LinkedList` object itself contains just a pointer to the list's sentinel (which never changes, once created) and an integer variable containing the number of items in the list. Technically, of course, the latter is redundant, since one can always count the number of items in the list, but keeping this variable allows `size` to be a constant-time operation. Figure 4.5 illustrates the three
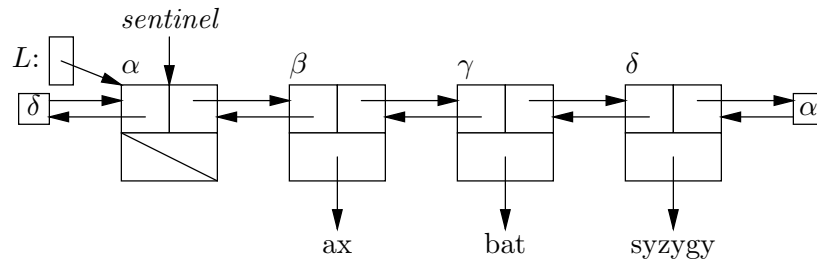
(a) Initial list
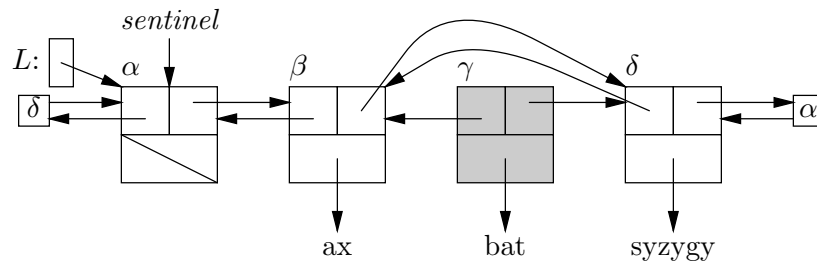
(b) After deleting item γ (`bat`)
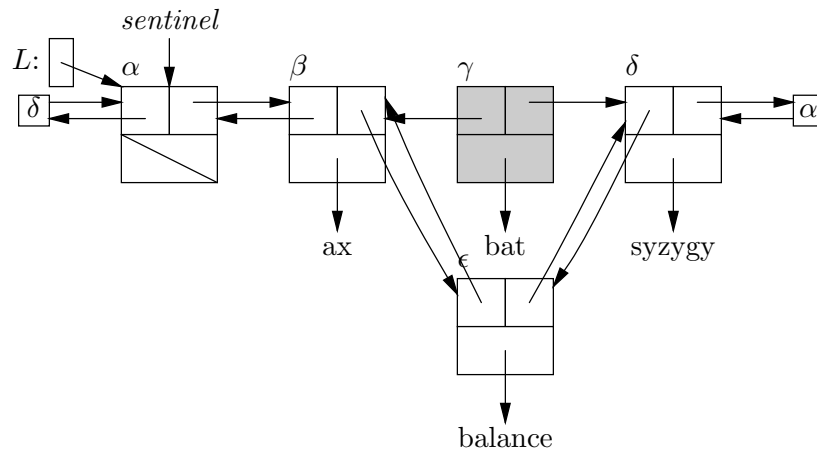
(c) After adding item ϵ (`balance`)
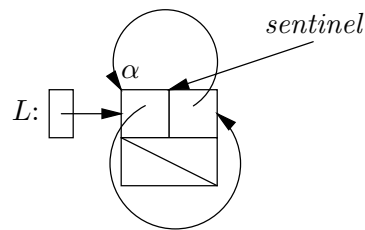
(d) After removing all items, and removing garbage.

Figure 4.4: Doubly linked lists employing a single sentinel node to mark both front and back. Shaded item is garbage.

Data structure after executing:

```
L = new LinkedList<String>();
L.add("axolotl");
L.add("kludge");
L.add("xerophyte");
I = L.listIterator();
I.next();
```
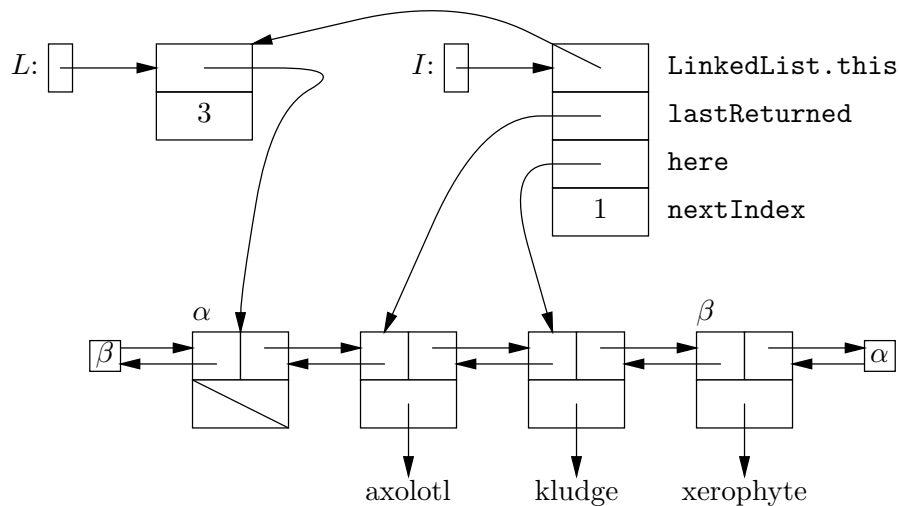

Figure 4.5: A typical `LinkedList` (pointed to by `L` and a list iterator (pointed to by `I`). Since the iterator belongs to an inner class of `LinkedList`, it contains an implicit private pointer (`LinkedList.this`) that points back to the `LinkedList` object from which it was created.

main data structures involved: `LinkedList`, `LinkedList.Entry`, and the iterator `LinkedList.LinkedIter`.

4.4 Specialized Lists

A common use for lists is in representing sequences of items that are manipulated and examined only at one or both ends. Of these, the most familiar are

- The *stack* (or *LIFO queue* for "Last-In First Out"), which supports only adding and deleting items at one end;

- The *queue* (or *FIFO queue*, for "First-In First Out"), which supports adding at one end and deletion from the other; and

- The *deque* or *double-ended queue*, which supports addition and deletion from either end.

whose operations are illustrated in Figure 4.8.

```
package java.util;
public class LinkedList<T> extends AbstractSequentialList<T>
       implements Cloneable {

  public LinkedList () {
    sentinel = new Entry ();
    size = 0;
  }

  public LinkedList (Collection<? extends T> c) {
    this();
    addAll (c);
  }

  public ListIterator<T> listIterator (int k) {
    if (k < 0 || k > size)
        throw new IndexOutOfBoundsException ();
    return new LinkedIter (k);
  }

  public Object clone () {
    return new LinkedList (this);
  }

  public int size () { return size; }

  private static class Entry<E> {
    E data;
    Entry prev, next;
    Entry (E data, Entry<E> prev, Entry<E> next) {
        this.data = data; this.prev = prev; this.next = next;
    }
    Entry () { data = null; prev = next = this; }
  }

  private class LinkedIter implements ListIterator {
    See Figure 4.7.
  }

  private final Entry<T> sentinel;
  private int size;
}
```

Figure 4.6: The class LinkedList.

```
package java.util;
public class LinkedList<T> extends AbstractSequentialList<T>
  implements Cloneable {
  ⋮
  private class LinkedIter<E> implements ListIterator<E> {
    Entry<E> here, lastReturned;
    int nextIndex;

    /** An iterator whose initial next element is item
     *  K of the containing LinkedList. */
    LinkedIter (int k) {
       if (k > size - k) {  // Closer to the end
          here = sentinel; nextIndex = size;
          while (k < nextIndex) previous ();
       } else {
          here = sentinel.next; nextIndex = 0;
          while (k > nextIndex) next ();
       }
       lastReturned = null;
    }

    public boolean hasNext () { return here != sentinel; }
    public boolean hasPrevious () { return here.prev != sentinel; }

    public E next () {
      check (here);
      lastReturned = here;
      here = here.next; nextIndex += 1;
      return lastReturned.data;
    }
    public E previous () {
      check (here.prev);
      lastReturned = here = here.prev;
      nextIndex -= 1;
      return lastReturned.data;
    }
```

Figure 4.7: The inner class `LinkedList.LinkedIter`. This version does not check for concurrent modification of the underlying List.

```
    public void add (T x) {
      lastReturned = null;
      Entry<T> ent = new Entry<T> (x, here.prev, here);
      nextIndex += 1;
      here.prev.next = here.prev = ent;
      size += 1;
    }

    public void set (T x) {
      checkReturned ();
      lastReturned.data = x;
    }

    public void remove () {
      checkReturned ();
      lastReturned.prev.next = lastReturned.next;
      lastReturned.next.prev = lastReturned.prev;
      if (lastReturned == here)
        here = lastReturned.next;
      else
        nextIndex -= 1;
      lastReturned = null;
      size -= 1;
    }

    public int nextIndex () { return nextIndex; }
    public int previousIndex () { return nextIndex-1; }

    void check (Object p) {
      if (p == sentinel) throw new NoSuchElementException ();
    }

    void checkReturned () {
       if (lastReturned == null) throw new IllegalStateException ();
    }
  }
```
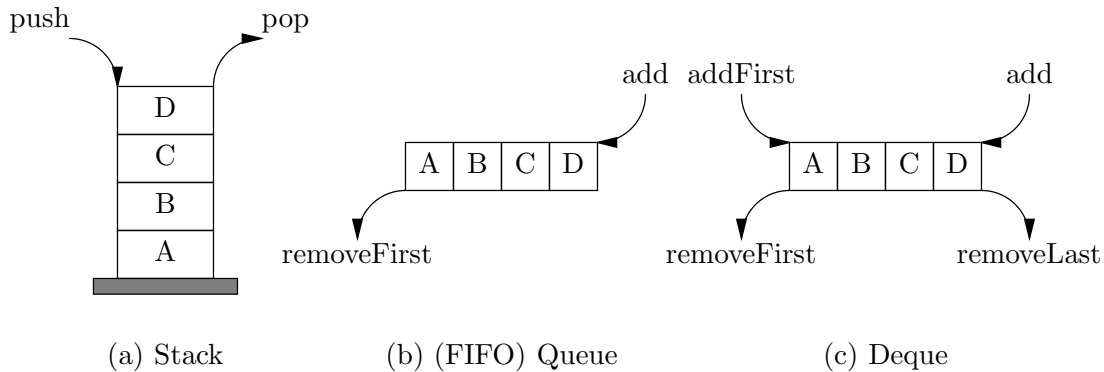
Figure 4.7, continued.

(a) Stack   (b) (FIFO) Queue   (c) Deque

Figure 4.8: Three varieties of queues—sequential data structures manipulated only at their ends.

4.4.1   Stacks

Java provides a type `java.util.Stack` as an extension of the type `java.util.Vector` (itself an older variation of `ArrayList`):

```
package java.util;
public class Stack<T> extends Vector<T> {
   /** An empty Stack. */
   public Stack () { }
   public boolean empty () { return isEmpty (); }
   public T peek () { check (); return get (size () - 1); }
   public T pop () { check (); return remove (size () - 1); }
   public T push (T x) { add (x); return x; }
   public int search (Object x) {
      int r = lastIndexOf (x);
      return r == -1 ? -1 : size () - r;
   }
   private void check () {
      if (empty ()) throw new EmptyStackException ();
   }
}
```

However, because it is one of the older types in the library, `java.util.Stack` does not fit in as well as it might. In particular, there is no separate interface describing "stackness." Instead there is just the `Stack` class, inextricably combining an interface with an implementation. Figure 4.9 shows how a `Stack` interface (in the Java sense) might be designed.

Stacks have numerous uses, in part because of their close relationship to *recursion* and *backtracking search.* Consider, for example, a simple-minded strategy for finding an exit to a maze. We assume some `Maze` class, and a `Position` class that represents a position in the maze. From any position in the maze, you may be able to move in up to four different directions (represented by numbers 0–4, standing perhaps for the compass points north, east, south, and west). The idea is that we

```
package ucb.util;
import java.util.*;
/** A LIFO queue of T's. */
public interface Stack<T> {
   /** True iff THIS is empty. */
   boolean isEmpty ();
   /** Number of items in the stack. */
   int size ();
   /** The last item inserted in the stack and not yet removed. */
   T top ();
   /** Remove and return the top item. */
   T  pop ();
   /** Add X as the last item of THIS. */
   void push (T x);
   /** The index of the most-recently inserted item that is .equal to
    *  X, or -1 if it is not present.  Item 0 is the least recently
    *  pushed. */
   int lastIndexOf (Object x);
}
```

Figure 4.9: A possible definition of the abstract type `Stack` as a Java interface. This is *not* part of the Java library, but its method names are more traditional than those of Java's official `java.util.Stack` type. It is designed, furthermore, to fit in with implementations of the `List` interface.

leave bread crumbs to mark each position we've already visited. From each position
we visit, we try stepping in each of the possible directions and continuing from that
point. If we find that we have already visited a position, or we run out of directions
to go from some position, we *backtrack* to the last position we visited before that
and continue with the directions we haven't tried yet from that previous position,
stopping when we get to an exit (see Figure 4.10). As a program (using method
names that I hope are suggestive), we can write this in two equivalent ways. First,
recursively:

```
/** Find an exit from M starting from PLACE. */
void findExit(Maze M, Position place) {
  if (M.isAnExit (place))
    M.exitAt (place);
  if (! M.isMarkedAsVisited (place)) {
    M.markAsVisited (place);
    for (dir = 0; dir < 4; dir += 1)
      if (M.isLegalToMove (place, dir))
        findExit (M, place.move (dir));
  }
}
```

Second, an iterative version:

```
import ucb.util.Stack;
import ucb.util.ArrayStack;
/** Find an exit from M starting from PLACE. */
void findExit(Maze M, Position place0) {
  Stack<Position> toDo = new ArrayStack<Position> ();
  toDo.push (place0);
  while (! toDo.isEmpty ()) {
    Position place = toDo.pop ();
    if (M.isAnExit (place))
      M.exitAt (place);
    if (! M.isMarkedAsVisited (place)) {
      M.markAsVisited (place);
      for (dir = 3; dir >= 0; dir -= 1)
        if (M.isLegalToMove (place, dir))
          toDo.push (place.move (dir));
    }
  }
}
```

where `ArrayStack` is an implementation of `ucb.util.Stack` (see §4.5).

 The idea behind the iterative version of `findExit` is that the `toDo` stack keeps
track of the values of `place` that appear as arguments to `findExit` in the recursive
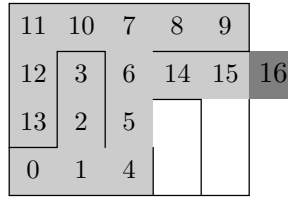version. Both versions visit the same positions in the same order (which is why

Figure 4.10: Example of searching a maze using backtracking search (the `findExit` procedure from the text). We start in the lower-left corner. The exit is the dark square on the right. The lightly shaded squares are those visited by the algorithm, assuming that direction 0 is up, 1 is right, 2 is down, and 3 is left. The numbers in the squares show the order in which the algorithm first visits them.

the loop runs backwards in the iterative version). In effect, the `toDo` plays the role of the *call stack* in the recursive version. Indeed, typical implementations of recursive procedures also use a stack for this purpose, although it is invisible to the programmer.

4.4.2 FIFO and Double-Ended Queues

A first-in, first-out queue is what we usually mean by *queue* in informal English (or *line* in American English): people or things join a queue at one end, and leave it at the other, so that the first to arrive (or *enqueue*) are the first to leave (or *dequeue*). Queues appear extensively in programs, where they can represent such things as sequences of requests that need servicing. The Java library (as of Java 2, version 1.5) provides a standard FIFO queue interface, but it is intended specifically for uses in which a program might have to wait for an element to get added to the queue. Figure 4.11 shows a more "classic" possible interface.

The *deque,* which is the most general, double-ended queue, probably sees rather little explicit use in programs. It uses even more of the `List` interface than does the FIFO queue, and so the need to specialize is not particularly acute. Nevertheless, for completeness, I have included a possible interface in Figure 4.12.

4.5 Stack, Queue, and Deque Implementation

We could implement a concrete stack class for our `ucb.util.Stack` interface as in Figure 4.13: as an extension of `ArrayList` just as `java.util.Stack` is an extension of `java.util.Vector`. As you can see, the names of the `Stack` interface methods are such that we can simply inherit implementations of `size`, `isEmpty`, and `lastIndexOf` from `ArrayList`.

But let's instead spice up our implementation of `ArrayStack` with a little generalization. Figure 4.14 illustrates an interesting kind of class known as an *adapter* or *wrapper* (another of the *design patterns* introduced at the beginning of Chapter 3). The class `StackAdapter` shown there will make any `List` object look like a stack. The figure also shows an example of using it to make a concrete stack representation out of the `ArrayList` class.

```
package ucb.util;

/** A FIFO queue */
public interface Queue<T> {
   /** True iff THIS is empty. */
   boolean isEmpty ();
   /** Number of items in the queue. */
   int size ();
   /** The first item inserted in the stack and not yet removed.
    *  Requires !isEmpty (). */
   T first ();
   /** Remove and return the first item. Requires !isEmpty (). */
   T removeFirst ();
   /** Add X as the last item of THIS. */
   void add (T x);
   /** The index of the first (least-recently inserted) item that is
    *  .equal to X, or -1 if it is not present.  Item 0 is first. */
   int indexOf (Object x);
   /** The index of the last (most-recently inserted) item that is
    *  .equal to X, or -1 if it is not present.  Item 0 is first. */
   int lastIndexOf (Object x);
}
```

Figure 4.11: A possible FIFO (First In, First Out) queue interface.

```
package ucb.util;

/** A double-ended queue */
public interface Deque<T> extends Queue<T> {
  /** The last inserted item in the sequence. Assumes !isEmpty(). */
  T last ();

  /** Insert X at the beginning of the sequence. */
  void addFirst (T x);

  /** Remove the last item from the sequence. Assumes !isEmpty(). */
  T removeLast ();

  /* Plus inherited definitions of isEmpty, size, first, add,
   * removeFirst, indexOf, and lastIndexOf */
}
```

Figure 4.12: A possible Deque (double-ended queue) interface

```
public class ArrayStack<T>
    extends java.util.ArrayList<T> implements Stack<T>
{
   /** An empty Stack. */
   public ArrayStack () { }
   public T top () { check (); return get (size () - 1); }
   public T pop () { check (); return remove (size () - 1); }
   public void push (T x) { add (x);  }
   private void check () {
      if (empty ()) throw new EmptyStackException ();
   }
}
```

Figure 4.13: An implementation of `ArrayStack` as an extension of `ArrayList`.

```
package ucb.util;
import java.util.*;

public class StackAdapter<T> implements Stack<T> {
   public StackAdapter (List<T> rep) { this.rep = rep; }

   public boolean isEmpty () { return rep.isEmpty (); }
   public int size () { return rep.size (); }
   public T top () { return rep.get (rep.size () - 1); }
   public T pop () { return rep.remove (rep.size () - 1); }
   public void push (T x) { rep.add (x); }
   public int lastIndexOf (Object x) { return rep.lastIndexOf (); }
}

public class ArrayStack extends StackAdapter {
   public ArrayStack () { this (new ArrayList ()); }
}
```

Figure 4.14: An adapter class that makes any List look like a `Stack`, and an example of using it to create an array-based implementation of the `ucb.util.Stack` interface.

Likewise, given any implementation of the `List` interface, we can easily provide implementations of `Queue` or `Deque`, but there is a catch. Both array-based and linked-list-based implementations of `List` will support our `Stack` interface equally well, giving `push` and `pop` methods that operate in constant amortized time. However, using an `ArrayList` in the same naïve fashion to implement either of the `Queue` or `Deque` interface gives very poor performance. The problem is obvious: as we've seen, we can add or remove from the end (high index) of an array quickly, but removing from the other (index 0) end requires moving over all the elements of the array, which takes time $\Theta(N)$, where N is the size of the queue. Of course, we can simply stick to `LinkedList`s, which don't have this problem, but there is also a clever trick that makes it possible to represent general queues efficiently with an array.

Instead of shifting over the items of a queue when we remove the first, let's instead just change our idea of where in the array the queue *starts.* We keep two indices into the array, one pointing to the first enqueued item, and one to the last. These two indices "chase each other" around the array, circling back to index 0 when they pass the high-index end, and vice-versa. Such an arrangement is known as a *circular buffer.* Figure 4.15 illustrates the representation. Figure 4.16 shows part of a possible implementation.
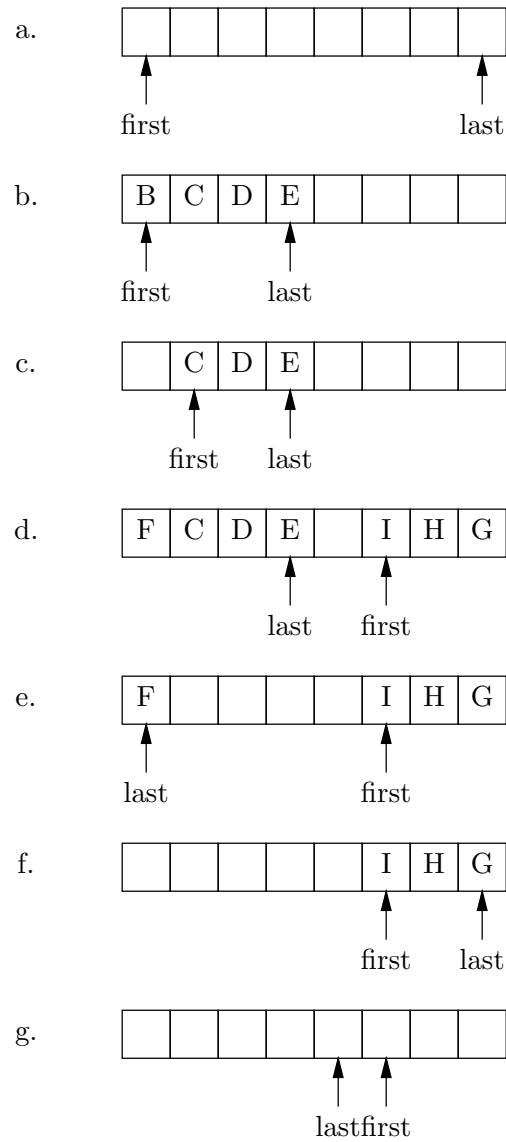
Figure 4.15: Circular-buffer representation of a deque with N==7. Part (a) shows an initial empty deque. In part (b), we've inserted four items at the end. Part (c) shows the result of removing the first item. Part (d) shows the full deque resulting from adding four items to the front. Removing the last three items gives (e), and after removing one more we have (f). Finally, removing the rest of the items from the end gives the empty deque shown in (g).

```
class ArrayDeque<T> implements Deque<T> {
   /** An empty Deque. */
   public ArrayDeque(int N) {
       first = 0; last = N; size = 0;
   }

   public int size () {
     return size;
   }

   public boolean isEmpty() {
    return size == 0;
   }

   public T first() {
     return data.get (first);
   }
   public T last() {
     return data.get (last);
   }
```

Figure 4.16: Implementation of `Deque` interface using a circular buffer.

```
public void add (T x) {
    size += 1;
    resize ();
    last = (last+1 == data.size ()) ? 0 : last+1;
    data.put (last, x);
}

public void addFirst (T x) {
    size += 1;
    resize ();
    first = (first == 0) ? data.size () - 1 : first-1;
    data.put (first,x) ;
}

public T removeLast () {
    T val = last ();
    last = (last == 0) ? data.size ()-1 : last-1;
    return val;
}

public T removeFirst () {
    T val = first ();
    first = (first+1 == data.size ()) ? 0 : first+1;
    return val;
}

private int first, last;
private final ArrayList<T> data = new ArrayList<T> ();
private int size;

/** Insure that DATA has at least size elements. */
private void resize () { left to the reader }

etc.
}
```

Figure 4.16, continued.

Exercises

4.1.   Implement a type `Deque`, as an extension of `java.util.Vector`. To the operations required by `java.util.AbstractList`, add `first`, `last`, `insertFirst`, `insertLast`, `removeFirst`, `removeLast`, and do this in such a way that all the operations on `Vector` continue to work (e.g., `get(0)` continues to get the same element as `first()`), and such that the amortized cost for all these operations remains constant.

4.2.   Implement a type of `List` with the constructor:

```
public ConcatList (List<T> L0, List<T> L1) { ... }
```

that does not support the optional operations for adding and removing objects, but gives a *view* of the concatenation of L0 and L1. That is, `get(i)` on such a list gives element `i` in the concatenation of L0 and L1 at the time of the `get` operation (that is, changes to the lists referenced by L0 and L1 are reflected in the concatenated list). Be sure also to make `iterator` and `listIterator` work.

4.3.   A singly linked list structure can be circular. That is, some element in the list can have a tail (next) field that points to an item *earlier* in the list (not necessarily to the first element in the list). Come up with a way to detect whether there is such a circularity somewhere in a list. Do *not,* however, use any destructive operations on any data structure. That is, you can't use additional arrays, lists, `Vectors`, hash tables, or anything like them to keep track of items in the list. Use just simple list pointers without changing any fields of any list. See `CList.java` in the `hw5` directory.

4.4.   The implementations of `LinkedList` in Figure 4.6 and `LinkedList.LinkedIter` in Figure 4.7 do not provide checking for concurrent modification of the underlying list. As a result, a code fragment such as

```
for (ListIterator<Object> i = L.listIterator (); i.hasNext (); ) {
   if (bad (i.next ()))
      L.remove (i.previousIndex ());
}
```

can have unexpected effects. What is supposed to happen, according to the specification for `LinkedList`, is that i becomes invalid as soon as you call `L.remove`, and subsequent calls on methods of i will throw `ConcurrentModificationExceptions`.

   a. For the `LinkedList` class, what goes wrong with the loop above, and why?

   b. Modify our `LinkedList` implementation to perform the check for concurrent modification (so that the loop above throws `ConcurrentModificationException`).

4.5.   Devise a `DequeAdapter` class analogous to `StackAdapter`, that allows one to create deques (or queues) from arbitrary `List` objects.

4.6. Provide an implementation for the `resize` method of `ArrayDeque` (Figure 4.16). Your method should double the size of the `ArrayList` being used to represent the circular buffer if expansion is needed. Be careful! You have to do more than simply increase the size of the array, or the representation will break.

Chapter 5

Trees

In this chapter, we'll take a break from the definition of interfaces to libraries and look at one of the basic data-structuring tools used in representing searchable collections of objects, expressions, and other hierarchical structures, the *tree*. The term *tree* refers to several different variants of what we will later call a connected, acyclic, undirected graph. For now, though, let's *not* call it that and instead concentrate on two varieties of *rooted tree*. First,

> **Definition:** an *ordered tree* consists of
>
> A. A *node*[1], which may contain a piece of data known as a *label*. Depending on the application, a node may stand for any number of things and the data labeling it may be arbitrarily elaborate. The node part of a tree is known as its *root node* or *root*.
>
> B. A sequence of 0 or more trees, whose root nodes are known as the *children* of the root. Each node in a tree is the child of at most one node—its *parent*. The children of any node are *siblings* of each other[2].
>
> The number of children of a node is known as the *degree* of that node. A node with no children is called a *leaf (node), external node,* or *terminal node;* all other nodes are called *internal* or *non-terminal* nodes.

We usually think of there being connections called *edges* between each node and its children, and often speak of *traversing* or *following* an edge from parent to child or back. Starting at any node, r, there is a unique, non-repeating *path* or sequence of edges leading from r to any other node, n, in the tree with r as root. All nodes along that path, including r and n, are called *descendents* of r, and *ancestors* of n. A descendent of r is a *proper descendent* if it is not r itself; *proper ancestors* are defined analogously. Any node in a tree is the root of a *subtree* of that tree. Again, a *proper subtree* of a tree is one that not equal to (and is therefore smaller than)

---

[1] The term *vertex* may also be used, as it is with other graphs, but *node* is traditional for trees.

[2] The word *father* has been used in the past for *parent* and *son* for *child.* Of course, this is no longer considered quite proper, although I don't believe that the freedom to live one's life as a tree was ever an official goal of the women's movement.
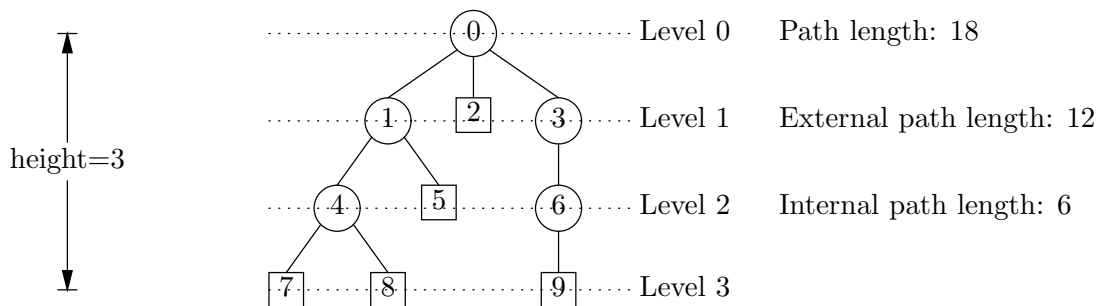
Figure 5.1: An illustrative ordered tree. The leaf nodes are squares. As is tradtional, the tree "grows" downward.

the tree. Any set of disjoint trees (such as the trees rooted at all the children of a node) is called a *forest*.

The distance from a node, n, to the root, r, of a tree—the number of edges that must be followed to get from n to r—is the *level* (or *depth*) of that node in the tree. The maximum level of all nodes in a tree is called the *height* of the tree. The sum of the levels of all nodes in a tree is the *path length* of the tree. We also define of the *internal (external) path length* as the sum of the levels of all internal (external) nodes. Figure 5.1 illustrates these definitions. All the levels shown in that figure are relative to node 0. It also makes sense to talk about "the level of node 7 in the tree rooted at node 1," which would be 2.

If you look closely at the definition of ordered tree, you'll see that it has to have at least one node, so that there is no such thing as an empty ordered tree. Thus, child number j of a node with $k > j$ children is always non-empty. It is easy enough to change the definition to allow for empty trees:

> **Definition:** A *positional tree* is either
>
> A. Empty (*missing*), or
>
> B. A node (generally labeled) and, for every non-negative integer, j, a positional tree—the jth child.
>
> The degree of a node is the number of non-empty children. If all nodes in a tree have children only in positions $< k$, we say it is a *k-ary tree*. Leaf nodes are those with no non-empty children; all others are internal nodes.

Perhaps the most important kind of positional tree is the *binary tree*, in which $k = 2$. For binary trees, we generally refer to child 0 and child 1 as the *left* and *right* children, respectively.

A *full* k-*ary tree* is one in which all internal nodes except possibly the rightmost bottom one have degree k. A tree is *complete* if it is full and all its leaf nodes occur last when read top to bottom, left to right, as in Figure 5.2c. Complete binary trees are of interest because they are, in some sense, maximally "bushy"; for any given number of internal nodes, they minimize the internal path length of the tree, which
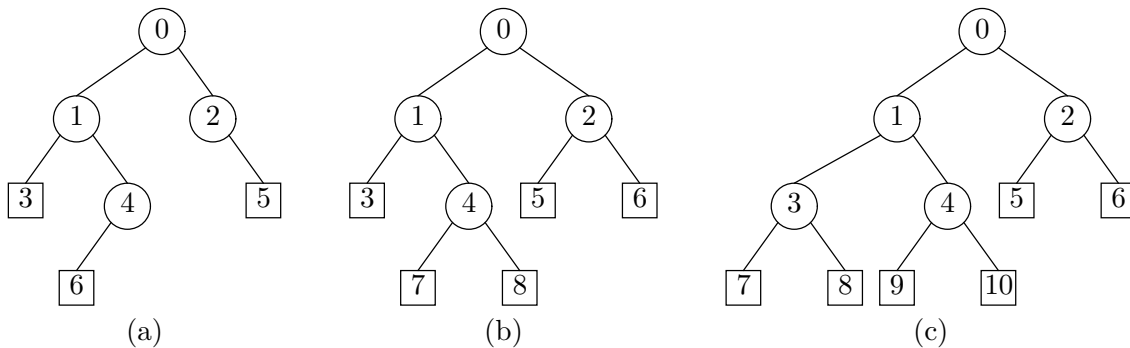
Figure 5.2: A forest of binary trees: (a) is not full; (b) is full, but not complete; (c) is complete. Tree (c) would still be complete if node 10 were missing, but not if node 9 were missing.

is interesting because it is proportional to the total time required to perform the operation of moving from the root to an internal node once for each internal node in the tree.

5.1 Expression trees

Trees are generally interesting when one is trying to represent a recursively-defined type. A familiar example is the *expression tree,* which represents an expression recursively defined as

- An identifier or constant, or

- An operator (which stands for some function of k arguments) and k expressions (which are its operands).

Given this definition, expressions are conveniently represented by trees whose internal nodes contain operators and whose external nodes contain identifiers or constants. Figure 5.3 shows a representation of the expression `x*(y + 3) - z`.
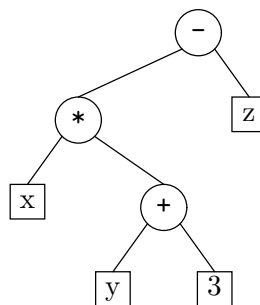

Figure 5.3: An expression tree for `x*(y+3)-z`.

As an illustration of how one deals with trees, consider the evaluation of expressions. As often happens, the definition of the value denoted by an expression corresponds closely to the structure of an expression:

- The value of a constant is the value it denotes as a numeral. The value of a variable is its currently-defined value.

- The value of an expression consisting of an operator and operand expressions is the result of applying the operator to the values of the operand expressions.

This definition immediately suggests a program.

```
/** The value currently denoted by the expression E (given
 *  current values of any variables).  Assumes E represents
 *  a valid expression tree, and that all variables
 *  contained in it have values. */
static int eval(Tree E)
{
    if (E.isConstant ())
        return E.valueOf ();
    else if (E.isVar ())
        return currentValueOf (E.variableName ());
    else
        return perform (E.operator (),
                        eval (E.left ()), eval (E.right ()));
}
```

Here, we posit the existence of a definition of `Tree` that provides operators for detecting whether `E` is a leaf representing a constant or variable, for extracting the data—values, variable names, or operator names—stored at `E`, and for finding the left and right children of `E` (for internal nodes). We assume also that `perform` takes an operator name (e.g., `"+"`) and two integer values, and performs the indicated computation on those integers. The correctness of this program follows immediately by using induction on the structure of trees (the trees rooted at a node's children are always subtrees of the node's tree), and by observing the match between the definition of the value of an expression and the program.

5.2   Basic tree primitives

There are a number of possible sets of operations one might define on trees, just as there were for sequences. Figure 5.4 shows one possible class (assuming integer labels). Typically, only some of the operations shown would actually be provided in a given application. For binary trees, we can be a bit more specialized, as shown in Figure 5.5. In practice, we don't often define `BinaryTree` as an extension of `Tree`, but I've done so here just as an illustration.

```
/** A positional tree with labels of type T.  The empty tree is null. */
class Tree<T> {
    /** A leaf node with given LABEL */
    public Tree(T label)  ...

    /** An internal node with given label, and K empty children */
    public Tree(T label, int k) ...

    /** The label of this node. */
    public T label() ...

    /** The number of non-empty children of this node. */
    public int degree() ...

    /** Number of children (argument K to the constructor) */
    public int numChildren() ...

    /** Child number K of this. */
    public Tree<T> child(int k) ...

    /** Set child number K of this to C, 0<=K<numChildren().
     *  C must not already be in this tree, or vice-versa. */
    public void setChild(int k, Tree<T> C) ...
}
```

Figure 5.4: A class representing positional tree nodes.

```
class BinaryTree<T> extends Tree<T> {
    public BinaryTree(T label,
                      BinaryTree<T> left, BinaryTree<T> right) {
        super(label, 2);
        child(0, left); child(1, right);
    }

    public BinaryTree<T> left() { return (BinaryTree) child(0); }
    public void setLeft(BinaryTree<T> C) { setChild(0, C);  }
    public BinaryTree<T> right() { return (BinaryTree) child(1); }
    public void setRight(BinaryTree<T> C) { setChild(1, C);  }
}
```

Figure 5.5: A possible class representing binary trees.

The operations so far all assume "root-down" processing of the tree, in which it is necessary to proceed from parent to child. When it is more appropriate to go the other way, the following operations are useful as an addition to (or substitute for) for the constructors and `child` methods of `Tree`.

```
/** The parent of T, if any (otherwise null). */
public Tree<T> parent() ...

/** Sets parent() to P. */
public void setParent(Tree<T> P);

/** A leaf node with label L and parent P */
public Tree(T L, Tree<T> P);
```

5.3   Representing trees

As usual, the representation one uses for a tree depends in large part upon the uses one has for it.

5.3.1   Root-down pointer-based binary trees

For doing the traversals on binary trees described below (§5.4), a straightforward transcription of the recursive definition is often appropriate, so that the fields are

```
T L;                /* Data stored at node. */
BinaryTree<T> left, right;  /* Left and right children */
```

As I said about the sample definition of `BinaryTree`, this specialized representation is in practice more common than simply re-using the implementation of `Tree`. If the `parent` operation is to be supported, of course, we can add an additional pointer:

```
BinaryTree<T> parent;   // or Tree, as appopriate
```

5.3.2   Root-down pointer-based ordered trees

The fields used for `BinaryTree` are also useful for certain non-binary trees, thanks to the *leftmost-child, right-sibling* representation. Assume that we are representing an ordered tree in which each internal node may have any number of children. We can have `left` for any node point to child #0 of the node and have `right` point to the next sibling of the node (if any), illustrated in Figure 5.6.

A small example might be in order. Consider the problem of computing the sum of all the node values in a tree whose nodes contain integers (for which we'll use the library class `Integer`, since our labels have to be `Object`s). The sum of all nodes in a tree is the sum of the value in the root plus the sum of the values in all children. We can write this as follows:
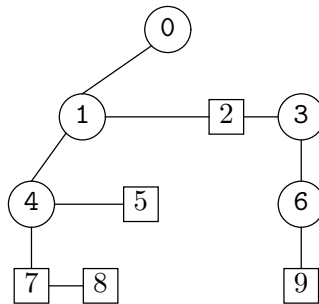
Figure 5.6: Using a binary tree representation to represent an ordered tree of arbitrary degree. The tree represented is the one from Figure 5.1. Left (down) links point to the first child of a node, and right links to the next sibling.

```
/** The sum of the values of all nodes of T, assuming T is an
    ordered tree with no missing children. */
static int treeSum(Tree<Integer> T)
{
    int S;
    S = T.label();
    for (int i = 0; i < T.degree(); i += 1)
        S += treeSum(T.child(i));
    return S;
}
```

(Java's unboxing operations are silently at work here turning `Integer` labels into **int**s.)

An interesting side light is that the inductive proof of this program contains no obvious base case. The program above is nearly a direct transcription of "the sum of the value in the root plus the sum of the values in all children."

5.3.3 Leaf-up representation

For applications where `parent` is the important operation, and `child` is not, a different representation is useful.

```
T label;
Tree<T> parent;      /* Parent of current node */
```

Here, `child` is an impossible operation.

The representation has a rather interesting advantage: it uses less space; there is one fewer pointers in each node. If you think about it, this might at first seem odd, since each representation requires one pointer per edge. The difference is that the "parental" representation does not need null pointers in all the external nodes, only in the root node. We will see applications of this representation in later chapters.

5.3.4   Array representations of complete trees

When a tree is complete, there is a particularly compact representation using an array. Consider the complete tree in Figure 5.2c. The parent of any node numbered $k > 0$ in that figure is node number $\lfloor (k-1)/2 \rfloor$ (or in Java, `(k-1)/2` or `(k-1)>>1`); the left child of node k is $2k+1$ and the right is $2k+2$. Had we numbered the nodes from 1 instead of 0, these formulae would have been even simpler: $\lfloor k/2 \rfloor$ for the parent, $2k$ for the left child, and $2k+1$ for the right. As a result, we can represent such complete trees as arrays containing just the `label` information, using indices into the array as pointers. Both the parent and child operations become simple. Of course, one must be careful to maintain the completeness property, or gaps will develop in the array (indeed, for certain incomplete trees, it can require an array with $2^h - 1$ elements to represent a tree with h nodes).

Unfortunately, the headers needed to effect this representation differ slightly from the ones above, since accessing an element of a tree represented by an array in this way requires three pieces of information—an array, an upper bound, and an index—rather than just a single pointer. In addition, we presumably want routines for allocating space for a new tree, specifying in advance its size. Here is an example, with some of the bodies supplied as well.

```
/** A BinaryTree2<T> is an entire binary tree with labels of type T.
    The nodes in it are denoted by their depth-first number in
    a complete tree. */
class BinaryTree2<T> {
   protected T[] label;
   protected int size;

   /** A new BinaryTree2 with room for N labels. */
   public BinaryTree2(int N) {
      label = (T[]) new Object[N]; size = 0;
   }

   public int currentSize() { return size; }
   public int maxSize() { return label.length; }

   /** The label of node K in breadth-first order.
    *  Assumes 0 <= k < size. */
   public T label(int k) { return label[k]; }
   /** Cause label(K) to be VAL. */
   public void setLabel(int k, T val) { label[k] = val; }

   public int left(int k) { return 2*k+1; }
   public int right(int k) { return 2*k+2; }
   public int parent(int k) { return (k-1)/2; }
   Continues...
```

0	1	2	3	4	5	6	7	8	9	10				

Figure 5.7: The binary tree in Figure 5.2c, represented with an array. The labels of the nodes happen to equal their breadth-first positions. In that figure, nodes 7 and 8 are the left and right children of node 3. Therefore, their labels appear at positions 7 $(3 \cdot 2 + 1)$ and 8 $(3 \cdot 2 + 2)$.

```
Continuation of  BinaryTree2<T>:
/** Add one more node to the tree, the next in breadth-first
 *  order. Assumes currentSize() < maxSize(). */
public void extend(T label) {
   this.label[size] = label; size += 1;
}
}
```

We will see this array representation later, when we deal with heap data structures. The representation of the binary tree in Figure 5.2c is illustrated in Figure 5.7.

5.3.5 Alternative representations of empty trees

In our representations, the empty tree tends to have a special status. For example, we can formulate a method to access the left node of a tree, `T`, with the syntax `T.left()`, but we can't write a method such that `T.isEmpty()` is true iff `T` references the empty tree. Instead, we must write `T == null`. The reason, of course, is that we represent the empty tree with the null pointer, and no instance methods are defined on the null pointer's dynamic type (more concretely, we get a `NullPointerException` if we try). If the null tree were represented by an ordinary object pointer, it wouldn't need a special status.

For example, we could extend our definition of `Tree` from the beginning of §5.2 as follows:

```
class Tree<T> {
  ...
  public final Tree<T> EMPTY = new EmptyTree<T> ();

  /** True iff THIS is the empty tree. */
  public boolean isEmpty () { return false; }

  private static class EmptyTree<T> extends Tree<T> {
    /** The empty tree */
    private EmptyTree () { }
    public boolean isEmpty () { return true; }
    public int degree() { return 0; }
    public int numChildren() { return 0; }
    /** The kth child (always an error). */
    public Tree<T> child(int k) {
```

```
        throw new IndexOutOfBoundsException ();
    }
    /** The label of THIS (always an error). */
    public T label () {
        throw new IllegalStateException ();
    }
  }
}
```

There is only one empty tree (guaranteed because the `EmptyTree` class is private to the `Tree` class, an example of the Singleton design pattern), but this tree is a full-fledged object, and we will have less need to make special tests for null to avoid exceptions. We'll be extending this representation further in the discussions of tree traversals (see §5.4.2).

5.4   Tree traversals.

The function `eval` in §5.1 *traverses* (or *walks*) its argument—that is, it processes each node in the tree. Traversals are classified by the order in which they process the nodes of a tree. In the program `eval`, we first traverse (i.e., evaluate in this case) any children of a node, and then perform some processing on the results of these traversals and other data in the node. The latter processing is known generically as *visiting* the node. Thus, the pattern for `eval` is "traverse the children of the node, then visit the node," an order known as *postorder*. One could also use postorder traversal for printing out the expression tree in reverse Polish form, where visiting a node means printing its contents (the tree in Figure 5.3 would come out as "x y 3 + * z -"). If the primary processing for each node (the "visitation") occurs *before* that of the children, giving the pattern "visit the node, then traverse its children", we get what is known as *preorder traversal*. Finally, the nodes in Figures 5.1 and 5.2 are all numbered in *level order* or *breadth-first order*, in which a nodes at a given level of the tree are visited before any nodes at the next.

All of the traversal orders so far make sense for any kind of tree we've considered. There is one other standard traversal ordering that applies exclusively to binary trees: the *inorder* or *symmetric-order* traversal. Here, the pattern is "traverse the left child of the node, visit the node, and then traverse the right child." In the case of expression trees, for example, such an order would reproduce the represented expression in infix order. Actually, that's not quite accurate, because to get the expression properly parenthesized, the precise operation would have to be something like "write a left parenthesis, then traverse the left child, then write the operator, then traverse the right child, then write a right parenthesis," in which the node seems to be visited several times. However, although such examples have led to at least one attempt to introduce a more general notation for traversals[3], we usually

---

[3]For example, Wulf, Shaw, Hilfinger, and Flon used such a classification scheme in *Fundamental Structures of Computer Science* (Addison-Wesley, 1980). Under that system, a preorder traversal is NLR (for visit Node, traverse Left, traverse Right), postorder is LRN, inorder is LNR, and the
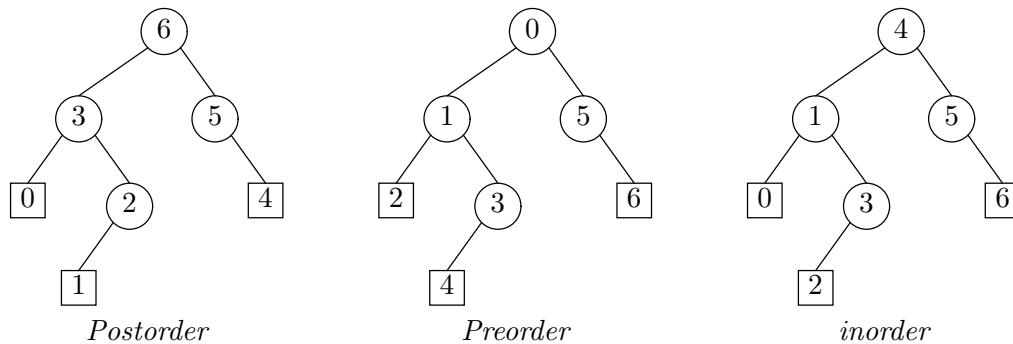
Figure 5.8: Order of node visitation in postorder, preorder, and inorder traversals.

just classify them approximately in one of the categories described above and leave it at that.

Figure 5.8 illustrates the nodes of several binary trees numbered in the order they would be visited by preorder, inorder, and postorder tree traversals.

5.4.1 Generalized visitation

I've been deliberately vague about what "visiting" means, since tree traversal is a general concept that is not specific to any particular action at the tree nodes. In fact, it is possible to write a general definition of traversal that takes as a parameter the action to be "visited upon" each node of the tree. In languages that, like Scheme, have function closures, we simply make the visitation parameter be a function parameter, as in

```
;; Visit the nodes of TREE, applying VISIT to each in inorder
(define inorder-walk (tree visit)
  (if (not (null? tree))
      (begin (inorder-walk (left tree) visit)
             (visit tree)
             (inorder-walk (right tree) visit))))
```

so that printing all the nodes of a tree, for example is just

```
(inorder-walk myTree (lambda (x) (display (label x)) (newline)))
```

Rather than using functions, in Java we use objects (as for the `java.util.Comparator` interface in §2.2.4). For example, we can define interfaces such as

```
public interface TreeVisitor<T> {
    void visit (Tree<T> node);
}
public interface BinaryTreeVisitor<T> {
    void visit (BinaryTree<T> node);
}
```

---

true order for writing parenthesized expressions is NLNRN. This nomenclature seems not to have caught on.

The `inorder-walk` procedure above becomes

```
static <T> BinaryTreeVisitor<T>
    inorderWalk (BinaryTree<T> tree,
                 BinaryTreeVisitor<T> visitor)
{
  if (tree != null) {
    inorderWalk (tree.left (), visitor);
    visitor.visit (tree);
    inorderWalk (tree.right (), visitor);
  }
  return visitor;
}
```

and our sample call is

```
inorderWalk (myTree, new PrintNode ());
```

where `myTree` is, let's say, a `BinaryTree<String>` and we have defined

```
class PrintNode implements BinaryTreeVisitor<String> {
  public void visit (BinaryTree<String> node) {
    System.out.println (node.label ());
  }
}
```

Clearly, the `PrintNode` class could also be used with other kinds of traverals. Alternatively, we can leave the visitor anonymous, as it was in the original Scheme program:

```
inorderWalk (myTree,
             new BinaryTreeVisitor<String> () {
               public void visit (BinaryTree<String> node) {
                 System.out.println (node.label ());
               }
             });
```

   The general idea of encapsulating an operation as we've done here and then carrying it to each item in a collection is another design pattern known simply as *Visitor.*

By adding state to a visitor, we can use it to *accumulate* results:

```
/** A TreeVisitor that concatenates the labels of all nodes it
 *  visits. */
public class ConcatNode implements BinaryTreeVisitor<String> {
  private StringBuffer result = new StringBuffer ();
  public void visit (BinaryTree<String> node) {
    if (result.length () > 0)
      result.append (", ");
    result.append (node.label ());
  }
  public String toString () { return result.toString (); }
}
```

With this definition, we can print a comma-separated list of the items in `myTree` in inorder:

```
System.out.println (inorderWalk (myTree, new ConcatNode ()));
```

(This example illustrates why I had `inorderWalk` return its visitor argument. I suggest that you go through all the details of why this example works.)

5.4.2   Visiting empty trees

I defined the `inorderWalk` method of §5.4.1 to be a static (class) method rather than an instance method in part to make the handling of null trees clean. If we use the alternative empty-tree representation of §5.3.5, on the other hand, we can avoid special-casing the null tree and make traversal methods be part of the `Tree` class. For example, here is a possible preorder-walk method:

```
class Tree<T> {
  ...
  public TreeVisitor<T> preorderWalk (TreeVisitor<T> visitor) {
    visitor.visit (this);
    for (int i = 0; i < numChildren (); i += 1)
      child(i).preorderWalk (visitor);
    return visitor;
  }
  ...
  private static class EmptyTree<T> extends Tree<T> {
    ...
    public TreeVisitor<T> preorderWalk (TreeVisitor<T> visitor) {
      return visitor;
    }
  }
}
```

Here you see that there are no explicit tests for the empty tree at all; everything is implicit in which of the two versions of `preorderWalk` get called.

```
import java.util.Stack;
public class PreorderIterator<T> implements Iterator<T> {
   private Stack<BinaryTree<T>> toDo = new Stack<BinaryTree<T>> ();
   /** An Iterator that returns the labels of TREE in
    *  preorder. */
   public PreorderIterator (BinaryTree<T> tree) {
     if (tree != null)
       toDo.push (tree);
   }
   public boolean hasNext () {
     return ! toDo.empty ();
   }
   public T next () {
     if (toDo.empty ())
       throw new NoSuchElementException ();
     BinaryTree<T> node = toDo.pop ();
     if (node.right () != null)
       toDo.push (node.right ());
     if (node.left () != null)
       toDo.push (node.left ());
     return node.label ();
   }
   public void remove () {
     throw new UnsupportedOperationException ();
   }
}
```

Figure 5.9: An iterator for preorder binary-tree traversal using a stack to keep track of the recursive structure.

5.4.3   Iterators on trees

Recursion fits tree data structures perfectly, since they are themselves recursively defined data structures. The task of providing a non-recursive traversal of a tree using an iterator, on the other hand, is rather more troublesome than was the case for sequences.

One possible approach is to use a stack and simply transform the recursive structure of a traversal in the same manner we showed for the `findExit` procedure in §4.4.1. We might get an iterator like that for `BinaryTrees` shown in Figure 5.9.

Another alternative is to use a tree data structure with parent links, as shown for binary trees in Figure 5.10. As you can see, this implementation keeps track of the next node to be visited (in postorder) in the field `next`. It finds the node to visit after `next` by looking at the parent, and deciding what to do based on whether `next` is the left or right child of its parent. Since this iterator does postorder traversal, the node after `next` is `next`'s parent if `next` is a right child, and otherwise it is the

deepest, leftmost descendent of the right child of the parent.

Exercises

5.1. Implement an Iterator that enumerates the labels of a tree's nodes in inorder, using a stack as in Figure 5.9.

5.2. Implement an Iterator that enumerates the labels of a tree's nodes in inorder, using parent links as in Figure 5.10.

5.3. Implement a preorder Iterator that operates on the general type `Tree` (rather than `BinaryTree`).

```java
import java.util.Stack;
public class PostorderIterator<T> implements Iterator<T> {
   private BinaryTree<T> next;
   /** An Iterator that returns the labels of TREE in
    *  postorder. */
   public PostorderIterator (BinaryTree<T> tree) {
     next = tree;
     while (next != null && next.left () != null)
       next = next.left ();
   }
   public boolean hasNext () {
     return next != null;
   }
   public T next () {
     if (next == null)
       throw new NoSuchElementException ();
     T result = next.label ();
     BinaryTree<T> p = next.parent ();
     if (p.right == next)
        // Have just finished with the right child of p.
        next = p;
     else {
        next = p.right ();
        while (next != null && next.left () != null)
          next = next.left ();
     }
     return result;
   }
   public void remove () {
     throw new UnsupportedOperationException ();
   }
}
```

Figure 5.10: An iterator for postorder binary-tree traversal using a parent links in the tree to keep track of the recursive structure.

Chapter 6

Search Trees

A rather important use of trees is in searching. The task is to find out whether some target value is present in a data structure that represents a set of data, and possibly to return some auxiliary information associated with that value. In all these searches, we perform a number of steps until we either find the value we're looking for, or exhaust the possibilities. At each step, we eliminate some part of the remaining set from further consideration. In the case of linear searches (see §1.3.1), we eliminate one item at each step. In the case of binary searches (see §1.3.4), we eliminate half the remaining data at each step.

The problem with binary search is that the set of search items is difficult to change; adding a new item, unless it is larger than all existing data, requires that we move some portion of the array over to make room for the new item. The worst-case cost of this operation rises proportionately with the size of the set. Changing the array to a list solves the insertion problem, but the crucial operation of a binary search—finding the middle of a section of the array, becomes expensive.

Enter the tree. Let's suppose that we have a set of data values, that we can extract from each data value a *key,* and that the set of possible keys is *totally ordered*—that is, we can always say that one key is either less than, greater than, or equal to another. What these mean exactly depends on the kind of data, but the terms are supposed to be suggestive. We can approximate binary search by having these data values serve as the labels of a *binary search tree* (or *BST*), which is defined to be binary tree having the following property:

> **Binary-Search-Tree Property.** For every node, x, of the tree, all nodes in the left subtree of x have keys that are less than or equal to the key of x and all nodes in the right subtree of x have keys that are greater than or equal to the key of x. □

Figure 6.1a is an example of a typical BST. In that example, the labels are integers, the keys are the same as the labels, and the terms "less than," "greater than," and "equal to" have their usual meanings.

The keys don't have to be integers. In general, we can organize a set of values into a BST using any *total ordering* on the keys. A total ordering, let's call it '$\preceq$', has the following properties:

- *Completeness:* For any values x and y, either $x \preceq y$ or $y \preceq x$, or both;

- *Transitivity:* If $x \preceq y$ and $y \preceq z$, then $x \preceq z$, and

- *Anti-symmetry:* If $x \preceq y$ and $y \preceq x$, then $x = y$.

For example, the keys can be integers, and greater than, etc., can have their usual meanings. Or the data and keys can be strings, with the ordering being dictionary order. Or the data can be pairs, (a, b), and the keys can be the first items of the pairs. A dictionary is like that—it is ordered by the words being defined, regardless of their meanings. This last order is an example where one might expect to have several distinct items in the search tree with equal keys.

(a)

(b)
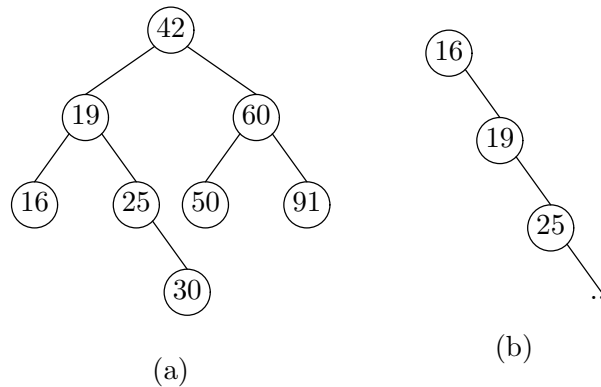
Figure 6.1: Two binary search trees. Tree (b) is right-leaning linear tree.

An important property of BSTs, which follows immediately from their definition, is that traversing a BST in inorder visits its nodes in ascending order of their labels. This leads to a simple algorithm for sorting known as "treesort."

```
/** Permute the elements of A into non-decreasing order. Assumes
 *  the elements of A have an order on them.  */
static void sort(SomeType[] A) {
    int i;
    BST T;

    T = null;
    for (i = 0; i < A.length; i += 1) {
        insert A[i] into search tree T.
    }

    i = 0;
    traverse T in inorder, where visiting a node, Q, means
        A[i] = Q.label(); i += 1;
}
```

The array contains elements of type `SomeType`, by which I intend to denote a type that has a less-than and equals operators on it, as required by the definition of a BST.

6.1 Operations on a BST

A BST is simply a binary tree, and therefore we can use the representation from §5.2, giving the class in Figure 6.2. For now, I will use the type `int` for labels, and we'll assume that labels are the same as keys.

Since it is possible to have more than one instance of a label in this particular version of binary search tree, I have to specify carefully what it means to remove that label or to find a node that contains it. I have chosen here to choose the "highest" node containing the label—the one nearest the root. [Why will this always be unique? That is, why can't there be two highest nodes containing a label, equally near the root?]

One problematic feature of this particular BST definition is that the data structure is relatively unprotected. As the comment on `insert` indicates, it is possible to "break" a BST by inserting something injudicious into one of its children, as with `BST.insert(T.left(),42)`, when `T.label()` is 20. When we incorporate the representation into a full-fledged implementation of `SortedSet` (see §6.2), we'll protect it against such abuse.

6.1.1 Searching a BST

Searching a BST is very similar to binary search in an array, with the root of the tree corresponding to the middle of the array.

```
/** The highest node in this tree that contains the
 *  label L, or null if there is none. */
public static BST find(BST T, int L)
{
    if (T == null || L == T.label)
       return T;
    else if (L < T.label)
       return find(T.left, L);
    else return find(T.right, L);
}
```

6.1.2 Inserting into a BST

As promised, the advantage of using a tree is that it is relatively cheap to add things to it, as in the following routine.

```
/** A binary search tree. */
class BST {
   protected int label;
   protected BST left, right;

   /** A leaf node with given LABEL */
   public BST(int label) { this(label, null, null); }

   /** Fetch the label of this node. */
   public int label();

   /** Fetch the left (right) child of this. */
   public BST left() ...
   public BST right() ...

   /** The highest node in T that contains the
    *  label L, or null if there is none. */
   public static BST find(BST T, int L) ...

   /** True iff label L is in T. */
   public static boolean isIn(BST T, int L)
     { return find (T, L) != null; }

   /** Insert the label L into T, returning the modified tree.
    *  The nodes of the original tree may be modified.  If
    *  T is a subtree of a larger BST, T', then insertion into
    *  T will render T' invalid due to violation of the binary-
    *  search-tree property if L > T'.label() and T is in
    *  T'.left() or L < T'.label() and T is in T'.right(). */
   public static BST insert(BST T, int L) ...

   /** Delete the instance of label L from T that is closest to
    *  to the root and return the modified tree.  The nodes of
    *  the original tree may be modified. */
   public static BST remove(BST T, int L) ...

   /* This constructor is private to force all BST creation
    * to be done by the insert method. */
   private BST(int label, BST left, BST right) {
      this.label = label; this.left = left; this.right = right;
   }
}
```

Figure 6.2: A BST representation.

```
/** Insert the label L into T, returning the modified tree.
 *  The nodes of the original tree may be modified.... */
static BST insert(BST T, int L)
{
   if (T == null)
      return new BST (L, null, null);
   if (L < T.label)
      T.left = insert(T.left, L);
   else
      T.right = insert(T.right, L);
   return this;
}
```

Because of the particular way that I have written this, when I insert multiple copies of a value into the tree, they always go "to the right" of all existing copies. I will preserve this property in the delete operation.

6.1.3 Deleting items from a BST.

Deletion is quite a bit more complex, since when one removes an internal node, one can't just let its children fall off, but must re-attach them somewhere in the tree. Obviously, deletion of an external node is easy; just replace it with the null tree (see Figure 6.3(a)). It's also easy to remove an internal node that is missing one child—just have the other child commit patricide and move up (Figure 6.3(b)).

When neither child is empty, we can find the *successor* of the node we want to remove—the first node in the right tree, when it is traversed in inorder. Now that node will contain the smallest key in the right subtree. Furthermore, because it is the first node in inorder, its left child will be null [why?]. Therefore, we can replace that node with its right child and move its key to the node we are removing, as shown in Figure 6.3(c).

remove 30   remove 25   remove 42
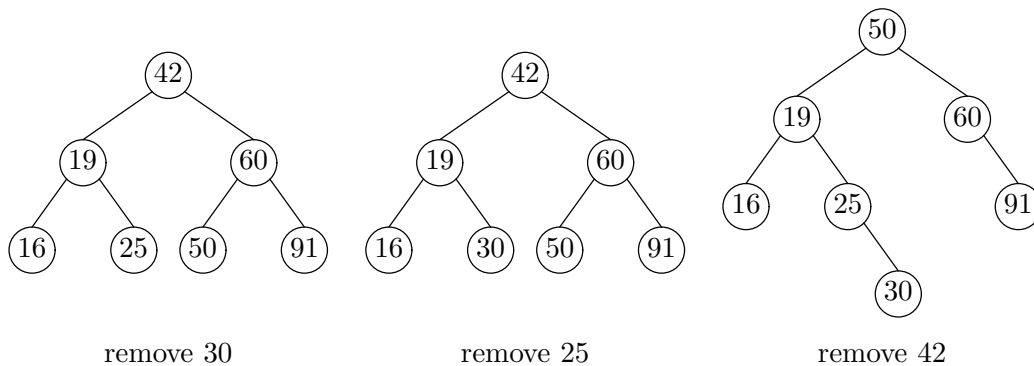
Figure 6.3: Three possible deletions, each starting from the tree in Figure 6.1.

A possible set of subprograms for deletion from a BST appears in Figure 6.4. The auxiliary routine `swapSmallest` is an additional method private to `BST`, and defined as follows.

```
/** Delete the instance of label L from T that is closest to
 *  to the root and return the modified tree.  The nodes of
 *  the original tree may be modified. */
public static BST remove(BST T, int L) {
   if (T == null)
      return null;
   if (L < T.label)
      T.left = remove(T.left, L);
   else if (L > T.label)
      T.right = remove(T.right, L);
   // Otherwise, we've found L
   else if (T.left == null)
      return T.right;
   else if (T.right == null)
      return T.left;
   else
      T.right = swapSmallest(T.right, T);
   return T;
}


/** Move the label from the first node in T (in an inorder
 *  traversal) to node R (over-writing the current label of R),
 *  remove the first node of T from T, and return the resulting tree.
 */
private static BST swapSmallest(BST T, BST R) {
   if (T.left == null) {
      R.label = T.label;
      return T.right;
   } else {
      T.left = swapSmallest(T.left, R);
      return T;
   }
}
```

Figure 6.4: Removing items from a BST without parent pointers.

```
static BST insert(BST T, int L) {
   BST newNode;
   if (T == null)
      return new BST (L, null, null);
   if (L < T.label)
      T.left = newNode = insert(T.left, L);
   else
      T.right = newNode = insert(T.right, L);
   newNode.parent = T;
   return this;
}
```

Figure 6.5: Insertion into a BST that has parent pointers.

6.1.4 Operations with parent pointers

If we revise the BST class to provide a `parent` operation, and add a corresponding `parent` field to the representation, the operations become more complex, but provide a bit more flexibility. It is probably wise *not* to provide a `setParent` operation for BST, since it is particularly easy to destroy the binary-search-tree property with this operation, and a client of BST would be unlikely to need it in any case, given the existence of `insert` and `remove` operations.

The operation `find` operation is unaffected, since it ignores parent nodes. When inserting in a BST, on the other hand, life is complicated by the fact that `insert` must set the parent of any node inserted. Figure 6.5 shows one way. Finally, removal from a BST with parent pointers—shown in Figure 6.6—is trickiest of all, as usual.

6.1.5 Degeneracy strikes

Unfortunately, all is not roses. The tree in Figure 6.1(b) is the result of inserting nodes into a tree in ascending order (obviously, the same tree can result from appropriate deletions from a larger tree as well). You should be able to see that doing a search or insertion on this tree is just like doing a search or insertion on a linked list; it *is* a linked list, but with extra pointers in each element that are always null. This tree is not *balanced*: it contains subtrees in which left and right children have much different heights. We will return to this question in Chapter 9, after developing a bit more machinery.

6.2 Implementing the SortedSet interface

The standard Java library interface `SortedSet` (see §2.2.4) provides a kind of `Collection` that supports *range queries.* That is, a program can use the interface to find all items in a collection that are within a certain range of values, according to some ordering relation. Searching for a single specific value is simply a special case in which the range contains just one value. It is fairly easy to implement this

```
/** Delete the instance of label L from T that is closest to
 *  to the root and return the modified tree.  The nodes of
 *  the original tree may be modified. */
public static BST remove(BST T, int L) {
   if (T == null)
      return null;
   BST newChild;
   newChild = null; result = T;
   if (L < T.label)
      T.left = newChild = remove(T.left, L);
   else if (L > T.label)
      T.right = newChild = remove(T.right, L);
   // Otherwise, we've found L
   else if (T.left == null)
      return T.right;
   else if (T.right == null)
      return T.left;
   else
      T.right = newChild = swapSmallest(T.right, T);
   if (newChild != null)
      newChild.parent = T;
   return T;
}

private static BST swapSmallest(BST T, BST R) {
   if (T.left == null) {
      R.label = T.label;
      return T.right;
   } else {
      T.left = swapSmallest(T.left, R);
      if (T.left != null)
          T.left.parent = T;
      return T;
   }
}
```

Figure 6.6: Removing items from a BST with parent pointers.

interface using a binary search tree as the representation; we'll call the result a
`BSTSet`.

Let's plan ahead a little. Among the operations we'll have to support are
`headSet`, `tailSet`, and `subSet`, which return views of some underlying set that con-
sist of a subrange of that set. The values returned will be full-fledged `SortedSet`s
in their own right, modifications to which are supposed to modify the underlying
set as well (and vice-versa). Since a full-fledged set can also be thought of as a view
of a range in which the bounds are "infinitely small" to "infinitely large," we might
look for a representation that supports *both* sets created "fresh" from a constructor,
and those that are views of other sets. This suggests a representation for our set
that contains a pointer to the root of a `BST`, and two bounds indicating the largest
and smallest members of the set, with null indicating a missing bound.

We make the root of the `BST` a (permanent) sentinel node for an important
reason. We will use the same tree for all views of the set. If our representation
simply pointed at a root of the tree that contained data, then this pointer would
have to change whenever that node of the tree was removed. But then, we would
have to make sure to update the root pointer in all other views of the set as well,
since they are also supposed to reflect changes in the set. By introducing the sentinel
node, shared by all views and never deleted, we make the problem of keeping them
all up to date trivial. This is a typical example of the old computer-science maxim:
Most technical problems can be solved by introducing another level of indirection.

Assuming we use parent pointers, an iterator through a set can consist of a
pointer to the next node whose label is to be returned, a pointer to the last node
whose label was returned (for implementing `remove`) and a pointer to the `BSTSet`
being iterated over (conveniently provided in Java by making the iterator an inner
class). The iterator will proceed in inorder, skipping over portions of the tree that
are outside the bounds on the set. See also Exercise 5.2 concerning iterating using
a `parent` pointer.

Figure 6.8 illustrates a `BSTSet`, showing the major elements of the representa-
tion: the original set, the `BST` that contains its data, a view of the same set, and
an iterator over this view. The sets all contain space for a `Comparator` (see §2.2.4)
to allow the user of the set to specify an ordering; in Figure 6.8, we use the natu-
ral ordering, which on strings gives us lexicographical order. Figure 6.7 contains a
sketch of the corresponding Java declarations for the representation.

6.3 Orthogonal Range Queries

Binary search trees divide data (ideally) into halves, using a linear ordering on the
data. The divide-and-conquer idea, however, does not require that the factor be two.
Suppose we are dealing with keys that have more structure. For example, consider a
collection of items that have locations on, say, some two-dimensional area. In some
cases, we may wish to find items in this collection based on their location; their
keys are their locations. While it is *possible* to impose a linear ordering on such
keys, it is not terribly useful. For example, we could use a lexicographic ordering,
and define $(x_0, y_0) > (x_1, y_1)$ iff $x_0 > x_1$ or $x_0 = x_1$ and $y_0 > y_1$. However, with

```
public class BSTSet<T> extends AbstractSet<T> {
  /** The empty set, using COMP as the ordering. */
  public BSTSet (Comparator<T> comp) {
    comparator = comp;
    low = high = null;
    sent = new BST ();
  }

  /** The empty set, using natural ordering. */
  public BSTSet () { this (null); }

  /** The set initialized to the contents of C, with natural order. */
  public BSTSet (Collection<? extends T> c) { addAll (c); }

  /** The set initialized to the contents of S, same ordering. */
  public BSTSet (SortedSet<? extends T> s) {
      this (s.comparator()); addAll (c);
  }

  ...

  /** Value of comparator(); null if naturally ordered. */
  private Comparator<T> comp;
  /** Bounds on elements in this class, null if no bounds. */
  private T low, high;
  /** Sentinel of BST containing data. */
  private final BST<T> sent;
```

Figure 6.7: Java representation for BSTSet class, showing only constructors and
instance variables.

```
  /** Used internally to form views. */
  private BSTSet (BSTSet<T> set, T low, T high) {
    comparator = set.comparator ();
    this.low = low; this.high = high;
    this.sent = set.sent;
  }

  /** An iterator over BSTSet. */
  private class BSTIter<T> implements Iterator<T> {
    /** Next node in iteration to yield. Equals the sentinel node
     *  when done. */
    BST<T> next;
    /** Node last returned by next(), or null if none, or if remove()
     *  has intervened. */
    BST<T> last;
    BSTIter () {
        last = null;
        next = first node that is in bounds, or sent if none;
    }
    ...
  }

  /** A node in the BST */
  private static class BST<T> {
    T label;
    BST<T> left, right, parent;

    /** A sentinel node */
    BST () { label = null; parent = null; }
    BST (T label, BST<T> left, BST<T> right) {
      this.label = label; this.left = left; this.right = right;
    }
  }
}
```

Figure 6.7, continued: Private nested classes used in implementation

Figure 6.8: A `BSTSet`, `fauna`, a view, `subset`, formed from `fauna.subSet("dog",` `"gnu")`, and an iterator, `I`, over `subset`. The `BST` part of the representation is shared between `fauna` and `subset`. Triangles represent whole subtrees, and rounded rectangles represent individual nodes. Each set contains a pointer to the root of the `BST` (a sentinel node, whose label is considered larger than any value in the tree), plus lower and upper bounds on the values (null means unbounded), and a `Comparator` (in this case, null, indicating natural order). The iterator contains a pointer to `subset`, which it is iterating over, a pointer (`next`) to the node containing the next label in sequence ("duck") and another pointer (`last`) to the node containing the label in the sequence that was last delivered by `I.next()`. The dashed regions of the `BST` are skipped entirely by the iterator. The "hartebeest" node is not returned by the iterator, but the iterator does have to pass through it to get down to the nodes it does return.

that definition, the set of all objects between points A and B consists of all those objects whose horizontal position lies between those of A and B, but whose vertical position is arbitrary (a long vertical strip). Half the information is unused.

The term *quadtree* (or *quad tree*) refers to a class of search tree structure that better exploits two-dimensional location data. Each step of a search divides the remaining data into four groups, one for each of four quadrants of a rectangle about some interior point. This interior dividing point can be the center (so that the quadrants are equal) giving a *PR quadtree* (also called a *point-region quadtree* or just *region quadtree*), or it can be one of the points that is stored in the tree, giving a *point quadtree.*

Figure 6.9 illustrates the idea behind the two types of quadtree. Each node of the tree corresponds to a rectangular region (possibly infinite in the case of point quadtrees). Any region may be subdivided into four rectangular subregions to the northwest, northeast, southeast, and southwest of some interior dividing point. These subregions are represented by children of the tree node that corresponds to the dividing point. For PR quadtrees, these dividing points are the centers of rectangles, while for point quadtrees, they are selected from the data points, just as the dividing values in a binary search tree are selected from the data stored in the tree.

6.4 Priority queues and heaps

Suppose that we are faced with a different problem. Instead of being able to search quickly for the presence of *any* element in a set, let us restrict ourselves to searching for the *largest* (by flipping everything in the following discussion around in the obvious way, we can search for smallest elements instead). Finding the largest in a BST is reasonably easy [how?], but we still have to deal with the imbalance problem described above. By restricting ourselves to the operations of inserting an element, and finding and deleting the largest element, we can avoid the balancing problem easily. A data structure supporting just those operations is called a *priority queue*, because we remove items from it in the order of their values, regardless of arrival order.

In Java, we could simply make a class that implements `SortedSet` and that was particularly fast at the operations `first` and `remove(x)`, when `x` happens to be the first element of the set. But of course, the user of such a class might be surprised to find how slow it is to iterate through an entire set. Therefore, we might specialize a bit, as shown in Figure 6.10.

A convenient data structure for representing priority queues is the *heap* (not to be confused with the large area of storage from which **new** allocates memory, an unfortunate but traditional clash of nomenclature). A heap is simply a positional tree (usually binary) satisfying the following property.

> **Heap Property.** The label at any node in the tree is greater than or equal to the label of any descendant of that node.
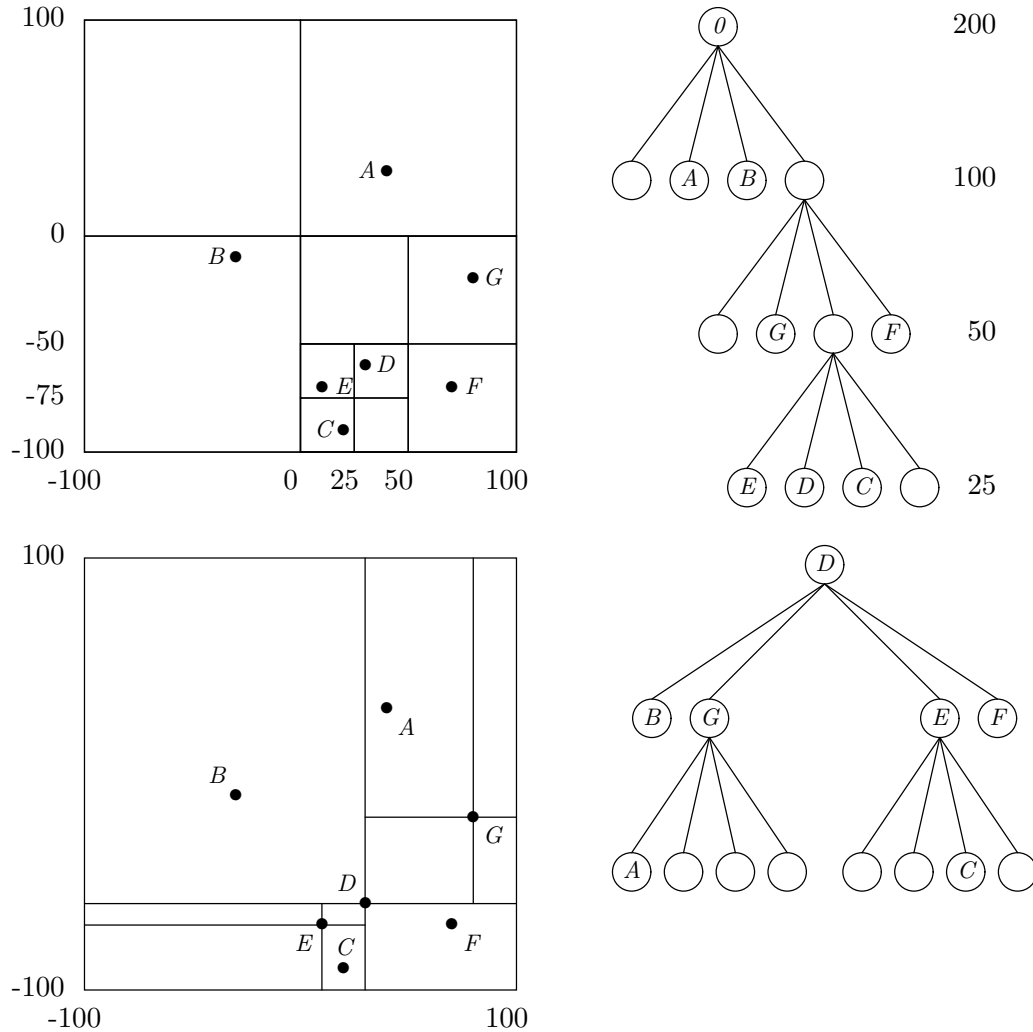
Figure 6.9: Illustration of two kinds of quadtree for the same set of data. On top is a four-level PR quadtree, using square regions for simplicity. Below is a corresponding point quadtree (there are many, depending on which points are used to divide the data). In each, the left diagram shows the geometry; the dots represent the positions—the keys—of seven data items at $(40, 30)$, $(-30, -10)$, $(20, -90)$, $(30, -60)$, $(10, -70)$, $(70, -70)$, and $(80, -20)$. On the right, we see the corresponding tree data structures. For the PR quadtree, each level of the tree contains nodes that represent squares with the same size of edge (shown at the right). For the point quadtree, each point is the root of a subtree that divides a rectangular region into four, generally unequal, regions. The four children of each node represent the upper-left, upper-right, lower-left, and lower-right quadrants of their common parent node, respectively. To simplify the drawing, we have not shown the children of a node when they are all empty.

```
interface PriorityQueue<T extends Comparable<T>> {
   /** Insert item L into this queue. */
   public void insert(T L);

   /** True iff this queue is empty. */
   public boolean isEmpty();

   /** The largest element in this queue.  Assumes !isEmpty(). */
   public T first();

   /** Remove and return an instance of the largest element (there may
    *  be more than one; removes only one). Assumes !isEmpty(). */
   public T removeFirst();
}
```

Figure 6.10: A possible interface to priority queues.

Since the order of the children is immaterial, there is more freedom in how to arrange the values in the heap, making it easy to keep a heap bushy. Accordingly, when we use the unqualified term "heap" in this context, we will mean a *complete* tree with the heap property. This speeds up all the manipulations of heaps, since the time required to do insertions and deletions is proportional to the height of the heap. Figure 6.11 illustrates a typical heap.

Implementing the operation of finding the largest value is obviously easy. To delete the largest element, while keeping both the heap property and the bushiness of the tree, we first move the "last" item on the bottom level of the heap to the root of the tree, replacing and deleting the largest element, and then "reheapify" to re-establish the heap property. Figure 6.11b–d illustrates the process. It is typical to do this with a binary tree represented as an array, as in the class `BinaryTree2` of §5.3. Figure 6.12 gives a possible implementation.

By repeatedly finding the largest element, of course, we can sort an arbitrary set of objects:

```
/** Sort the elements of A in ascending order. */
static void heapSort(Integer[] A) {
   if (A.length <= 1)
      return;
   Heap<Integer> H = new Heap<Integer>(A.length);
   H.setHeap(A, 0, A.length);
   for (int i = A.length-1; i >= 0; i -= 1)
      A[i] = H.removeFirst();
}
```

The process is illustrated in Figure 6.13.
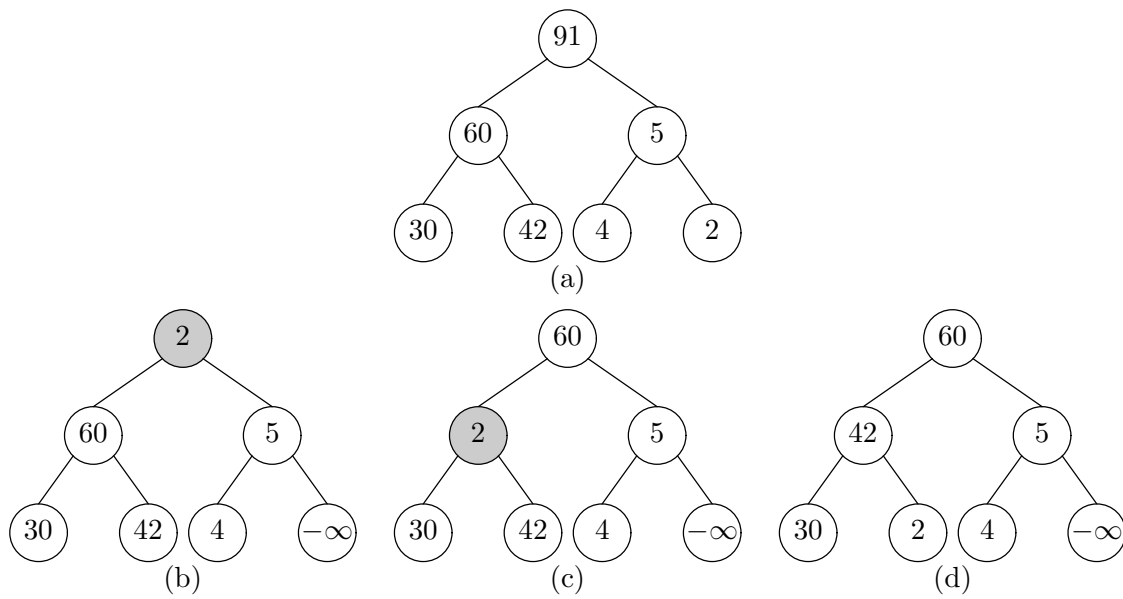
Figure 6.11: Illustrative heap (a). The sequence (b)–(d) shows steps in the deletion of the largest item. The last (bottommost, rightmost) label is first moved up to overwrite that of the root. It is then "sifted down" until the heap property is restored. The shaded nodes show where the heap property is violated during the process.

```
class Heap<T extends Comparable<T>>
      extends BinaryTree2<T> implements PriorityQueue<T> {

  /** A heap containing up to N > 0 elements. */
  public Heap (int N) { super(N);  }

  /** The minimum label value (written −∞). */
  static final int MIN = Integer.MIN_VALUE;

  /** Insert item L into this queue. */
  public void insert(T L) {
     extend(L);
     reHeapifyUp(currentSize()-1);
  }

  /** True iff this queue is empty. */
  public boolean isEmpty() { return currentSize() == 0; }

  /** The largest element in this queue. Assumes !isEmpty(). */
  public int first() { return label(0); }

  /** Remove and return an instance of the largest element (there may
   *  be more than one; removes only one). Assumes !isEmpty(). */
  public T removeFirst() {
     int result = label(0);
     setLabel(0, label(currentSize()-1));
     size -= 1;
     reHeapifyDown(0);
     return result;
  }
```

Figure 6.12: Implementation of a common kind of priority queue: the heap.

```java
/** Restore the heap property in this tree, assuming that only
 *  NODE may have a label larger than that of its parent. */
protected void reHeapifyUp(int node) {
   if (node <= 0)
      return;
   T x = label(node);
   while (node != 0 && label(parent(node)).compareTo (x) < 0) {
      setLabel(node, label(parent(node)));
      node = parent(node);
   }
   setLabel(node, x);
}

/** Restore the heap property in this tree, assuming that only
 *  NODE may have a label smaller than those of its children. */
protected void reHeapifyDown(int node) {
   T x = label(node);
   while (true) {
      if (left(node) >= currentSize())
         break;

      int largerChild =
         (right(node) >= currentSize()
          || label(right(node)).compareTo (label(left(node))) <= 0)
         ? left(node) : right(node);

      if (x >= label(largerChild))
         break;
      setLabel(node, label(largerChild));
      node = largerChild;
   }
   setLabel(node, x);
}

/** Set the labels in this Heap to A[off], A[off+1], ...
 *  A[off+len-1].  Assumes that LEN <= maxSize(). */
public void setHeap(T[] A, int off, int len) {
   for (int i = 0; i < len; i += 1)
      setLabel(i, A[off+i]);
   size = len;
   heapify();
}

/** Turn label(0)..label(size-1) into a proper heap. */
protected void heapify() { ... }
}
```

Figure 6.12, continued.

(a)	19	0	-1	7	23	2	42

(b)	42	23	19	7	0	2	-1

(c)	23	7	19	-1	0	2		42

(d)	19	7	2	-1	0		23	42

(e)	7	0	2	-1		19	23	42

(f)	2	0	-1		7	19	23	42

(g)	0	-1		2	7	19	23	42

(h)	-1		0	2	7	19	23	42

Figure 6.13: An example of heapsort. The original array is in (a); (b) is the result of
`setHeap`; (c)–(h) are the results of successive iterations. Each shows the active part
of the heap array and the portion of the output array that has been set, separated
by a gap.

We could simply implement `heapify` like this:

```
protected void heapify()
{
    for (int i = 1; i < size; i += 1)
        reHeapifyUp(i);
}
```

Interestingly enough, however, this implementation is not quite as fast as it could be, and it is faster to perform the operation by a different method, in which we work from the leaves back up. That is, in reverse level order, we swap each node with its parent, if it is larger, and then, as for `reHeapifyDown`, continue moving the parent's value down the tree until heapness is restored. It might seem that this is no different from repeated insertion, but we will see later that it is.

```
protected void heapify()
{
    for (int i = size/2-1; i >= 0; i -= 1)
        reHeapifyDown(i);
}
```

6.5   Heapify Time

If we measure the time requirements for sorting N items with `heapSort`, we see that it is the cost of "heapifying" N elements plus the time required to extract N items. The worst-case cost of extracting N items from the heap, $C_e(N)$ is dominated by the cost of `reHeapifyDown`, starting at the top node of the heap. If we count comparisons of parent labels against child labels, you can see that the worst-case cost here is proportional to the current height of the heap. Suppose the initial height of the heap is k (and that $N = 2^{k+1} - 1$). It stays that way until 2^k items have been extracted (removing the whole bottom row), and then becomes $k - 1$. It stays at $k - 1$ for the next 2^{k-1} items, and so forth. Thus, the total time spent extracting items is

$$C_e(N) = C_e(2^{k+1} - 1) = 2^k \cdot k + 2^{k-1} \cdot (k - 1) + \ldots + 2^0 \cdot 0$$

If we write $2^k \cdot k$ as $\underbrace{2^k + \ldots + 2^k}_{k}$ and re-arrange the terms, we get

$$
\begin{aligned}
C_e(2^{k+1} - 1) &= 2^k \cdot k + 2^{k-1} \cdot (k - 1) + \ldots + 2^0 \cdot 0 \\
&= \quad 2^1 \quad + \quad 2^2 \quad + \quad \cdots \quad + \quad 2^{k-1} \quad + \quad 2^k \\
&+ \quad 2^2 \quad + \quad \cdots \quad + \quad 2^{k-1} \quad + \quad 2^k \\
&+ \quad \vdots \\
&+ \quad 2^{k-1} \quad + \quad 2^k \\
&+ \quad 2^k \\
&= (2^{k+1} - 2) + (2^{k+1} - 4) + \cdots + (2^{k+1} - 2^{k-1}) + (2^{k+1} - 2^k) \\
&= k2^{k+1} - (2^{k+1} - 2) \\
&\in \Theta(k2^{k+1}) = \Theta(N \lg N)
\end{aligned}
$$

Now let's consider the cost of heapifying N elements. If we do it by inserting the N elements one by one and performing `reHeapifyUp`, then we get a cost like that of the extracting N elements: For the first insertion, we do 0 label comparisons; for the next 2, we do 1; for the next 4, we do 2; etc, or

$$C_h^u(2^{k+1} - 1) = 2^0 \cdot 0 + 2^1 \cdot 1 + \ldots + 2^k \cdot k$$

where $C_h^u(N)$ is the worst-case cost of heapifying N elements by repeated `reHeapifyUp`s. This is the same as the one we just did, giving

$$C_h^u(N) \in \Theta(N \lg N)$$

But suppose we heapify by performing the second algorithm at the end of §6.4, performing a `reHeapifyDown` on all the items of the array starting at item $\lfloor N/2 \rfloor - 1$ and going toward item 0. The cost of `reHeapifyDown` depends on the distance to the deepest level. For the last 2^k items in the heap, this cost is 0 (which is why we skip them). For the preceding 2^{k-1}, the cost is 1, etc. This gives

$$C_h^d(N) = C_h^d(2^{k+1} - 1) = 2^{k-1} \cdot 1 + 2^{k-2} \cdot 2 + \cdots + 2^0 \cdot k$$

Using the same trick as before,

$$
\begin{aligned}
C_h^d(2^{k+1} - 1) &= 2^{k-1} \cdot 1 + 2^{k-2} \cdot 2 + \cdots + 2^0 \cdot k \\
&= 2^0 + 2^1 + \cdots + 2^{k-2} + 2^{k-1} \\
&+ 2^0 + 2^1 + \cdots + 2^{k-2} \\
&+ \vdots \\
&+ 2^0 \\
&= (2^k - 1) + (2^{k-1} - 1) + \cdots + (2^1 - 1) \\
&= 2^{k+1} - 2 - k \\
&\in \Theta(N)
\end{aligned}
$$

So this second heapification method runs considerably faster (asymptotically) than the obvious repeated-insertion method. Of course, since the cost of extracting N elements is still $\Theta(N \lg N)$ in the worst case, the overall worst-case cost of heapsort is still $\Theta(N \lg N)$. However, this does lead you to expect that for big enough N, there will be some constant-factor advantage to using the second form of heapification, and that's an important practical consideration.

Exercises

6.1. Fill in a concrete implementation for the type `QuadTree` that has the following constructor:

```
/** An initially empty quadtree that is restricted to contain points
 *  within the  W x H rectangle whose center is at (X0,Y0).  */
public QuadTree (double x0, double y0, double w, double h) ...
```

and no other constructors.

6.2. Fill in a concrete implementation for the type `QuadTree` that has the following constructor:

```
/** An initially empty quadtree. */
public QuadTree () ...
```

and no other constructors. This problem is more difficult than the preceding exercise, because there is no *a priori* limit on the boundaries of the entire region. While you *could* simply use the maximum and minimum floating-point numbers for these bounds, the result would in general be a wasteful tree structure with many useless levels. Therefore, it makes sense to grow the region covered, as necessary, starting from some arbitrary initial size.

6.3. Suppose that we introduce a new kind of removal operation for BSTs that have parent pointers (see §6.1.4):

```
/** Delete the label T.label() from the BST containing T, assuming
 *  that the parent of T is not null. The nodes of the original tree
 *  will be modified. */
public static BST remove(BST T) { ⋯ }
```

Give an implementation of this operation.

6.4. The implementation of `BSTSet` in §6.2 left out one detail: the implementation of the `size` method. Indeed, the representation given in Figure 6.7 provides no way to compute it other than to count the nodes in the underlying BST each time. Show how to augment the representation of `BSTSet` or its nested classes as necessary so as to allow a constant-time implementation of `size`. Remember that the size of any view of a `BSTSet` might change when you change add or remove elements from any other view of the same `BSTSet`.

6.5. Assume that we have a heap that is stored with the largest element at the root. To print all elements of this heap that are greater than or equal to some key X, we *could* perform the `removeFirst` operation repeatedly until we get something less than X, but this would presumably take worst-case time $\Theta(k \lg N)$, where N is the number of items in the heap and k is the number of items greater than or equal to X. Furthermore, of course, it changes the heap. Show how to perform this operation in $\Theta(k)$ time *without* modifying the heap.

Chapter 7

Hashing

Sorted arrays and binary search trees all allow fast queries of the form "is there something larger (smaller) than X in here?" Heaps allow the query "what is the largest item in here?" Sometimes, however, we are interested in knowing only whether some item is present—in other words, only in equality.

Consider again the `isIn` procedure from §1.3.1—a linear search in a sorted array. This algorithm requires an amount of time at least proportional to N, the number of items stored in the array being searched. If we could reduce N, we would speed up the algorithm. One way to reduce N is to divide the set of keys being searched into some number, say M, of disjoint subsets and to then find some fast way of choosing the right subset. By dividing the keys more-or-less evenly among subsets, we can reduce the time required to find something to be proportional, on the average, to N/M. This is what binary search does recursively (`isInB` from §1.3.4), with $M = 2$. If we could go even further and choose a value for M that is comparable to N, then the time required to find a key becomes almost constant.

The problem is to find a way—preferably fast—of picking subsets (*bins*) in which to put keys to be searched for. This method must be consistent, since whenever we are asked to search for something, we must go to the subset we originally selected for it. That is, there must be a function—known as a *hashing function*—that maps keys to be searched for into the range of values 0 to $M - 1$.

7.1 Chaining

Once we have this hashing function, we must also have a representation of the set of bins. Perhaps the simplest scheme is to use linked lists to represent the bins, a practice known as *chaining* in the hash-table literature. The standard Java library class `HashSet` uses just such a strategy, illustrated in Figure 7.1. More usually, hash tables appear as mappings, such as implementaions of the standard Java interface `java.util.Map`. The representation is the same, except that the entries in the bins carry not only keys, but also the additional information that is supposed to be indexed by those keys. Figure 7.2 shows part of a possible implementation of the standard Java class `java.util.HashMap`, which is itself an implementation of the

`Map` interface.

The `HashMap` class shown in Figure 7.2 uses the `hashCode` method defined for all Java Objects to select a bin number for any key. If this hash function is a good one the bins will receive roughly equal numbers of items (see §7.3 for more discussion).

We can decide on an *a priori* limit on the average number of items per bin, and then grow the table whenever that limit is exceeded. This is the purpose of the `loadFactor` field and constructor argument. It's natural to ask whether we might use a "faster" data structure (such as a binary search tree) for the bins. However, if we really do choose reasonable values for the size of the tree, so that each bin contains only a few items, that clearly won't gain us much. Growing the `bins` array when it exceeds our chosen limit is like growing an `ArrayList` (§4.1). For good asymptotic time performance, we roughly double its size each time it becomes necessary to grow the table. We have to remember, in addition, that the bin numbers of most items will change, so that we'll have to move them.

7.2   Open-address hashing

In the Good Old Days, the overhead of "all those link fields" and the expense of "all those **new** operations" led people to consider ways of avoiding linked lists for representing the contents of bins. The *open-addressing* schemes put the entries directly into the bins (one per bin). If a bin is already full, then subsequent entries that have the same hash value overflow into other, unused entries according to some systematic scheme. As a result, the `put` operation from Figure 7.2 would look something like this:

```
public Val put (Key key, Val value) {
  int h = hash (key);
  while (bins.get (h) != null && ! bins.get (h).key.equals (key))
    h = nextProbe (h);
  if (bins.get (h) == null) {
    bins.add (new entry);
    size += 1;
    if ((float) size/bins.size () > loadFactor)
        resize  bins;
    return null;
  } else
    return bins.get (h).setValue (value);
}
```

and `get` would be similarly modified.

The function `nextProbe` provides another value in the index range of `bins` for use when it turns out that the table is already occupied at position `h` (a situation known as a *collision*).

In the simplest case `nextProbe(L)` simply returns `(h+1) % bins.size ()`, an instance of what is called known *linear probing*. More generally, linear probing
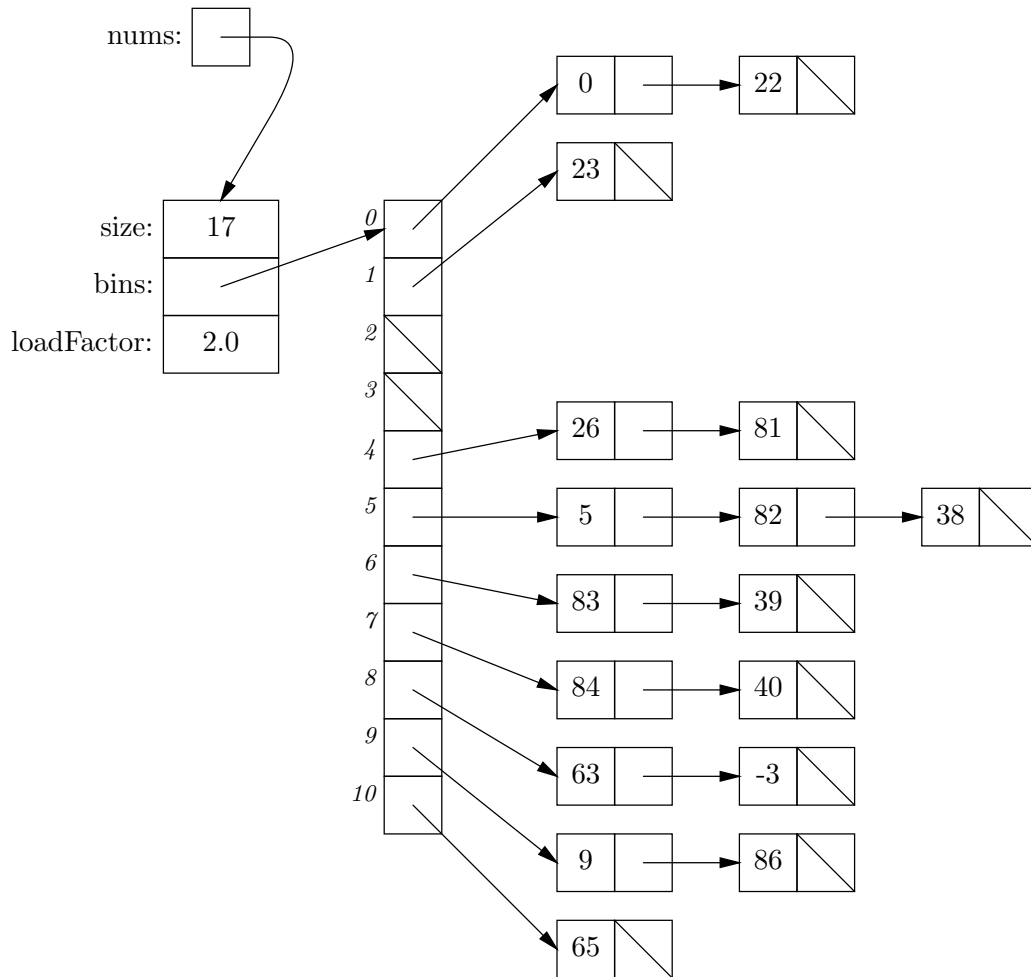
Figure 7.1: Illustration of a simple hash table with chaining, pointed to by the variable `nums`. The table contains 11 bins, each containing a pointer to a linked list of the items (if any) in that bin. This particular table represents the set

$$\{81, 22, 38, 26, 86, 82, 0, 23, 39, 65, 83, 40, 9, -3, 84, 63, 5\}.$$

The hash function is simply $h(x) = x \bmod 11$ on integer keys. (The mathematical operation $a \bmod b$ is defined to yield $a - b\lfloor a/b \rfloor$ when $b \neq 0$. Therefore, it is always non-negative if $b > 0$.) The current load factor in this set is $17/11 \approx 1.5$, against a maximum of 2.0 (the `loadFactor` field), although as you can see, the bin sizes vary from 0 to 3.

```java
package java.util;
public class HashMap<Key,Val> extends AbstractMap<Key,Val> {
  /** A new, empty mapping using a hash table that initially has
   *  INITIALBINS bins, and maintains a load factor <= LOADFACTOR.  */
  public HashMap (int initialBins, float loadFactor) {
    if (initialBuckets < 1 || loadFactor <= 0.0)
      throw new IllegalArgumentException ();
    bins = new ArrayList<Entry<Key,Val>>(initialBuckets);
    bins.addAll (Arrays.ncopies (initialBins, null));
    size = 0; this.loadFactor = loadFactor;
  }

  /** An empty map with INITIALBINS initial bins and load factor 0.75. */
  public HashMap (int initialBins) { this (initialBins, 0.75); }

  /** An empty map with default initial bins and load factor 0.75. */
  public HashMap () { this (127, 0.75); }

  /** A mapping that is a copy of M. */
  public HashMap (Map<Key,Val> M) { this (M.size (), 0.75); putAll (M); }

  public T get (Object key) {
    Entry e = find (key, bins.get (hash (key)));
    return (e == null) ? null : e.value;
  }

  /** Cause get(KEY) == VALUE.  Returns the previous get(KEY). */
  public Val put (Key key, Val value) {
    int h = hash (key);
    Entry<Key,Val> e = find (key, bins.get (h));
    if (e == null) {
      bins.set (h, new Entry<Key,Val> (key, value, bins.get (h)));
      size += 1;
      if (size > bins.size () * loadFactor) grow ();
      return null;
    } else
      return e.setValue (value);
  }
  ...
```

Figure 7.2: Part of an implementation of class `java.util.HashMap`, a hash-table-based implementation of the `java.util.Map` interface.

```
private static class Entry<K,V> implements Map.Entry<K,V> {
  K key; V value;
  Entry<K,V> next;
  Entry (K key, V value, Entry<K,V> next)
    { this.key = key; this.value = value; this.next = next; }
  public K getKey () { return key; }
  public V getValue () { return value; }
  public V setValue (V x)
    { V old = value; value = x; return old; }
  public int hashCode () { see Figure 2.14 }
  public boolean equals () { see Figure 2.14 }
}

private ArrayList<Entry<Key,Val>> bins;
private int size;      /** Number of items currently stored */
private float loadFactor;

/** Increase number of bins. */
private void grow () {
  HashMap<Key,Val> newMap
      = new HashMap (primeAbove (bins.size ()*2), loadFactor);
  newMap.putAll (this); copyFrom (newMap);
}

/** Return a value in the range 0 .. bins.size ()-1, based on
 *  the hash code of KEY. */
private int hash (Object key) {
  return (key == null) ? 0
    : (0x7fffffff & key.hashCode ()) % bins.size ();
}

/** Set THIS to the contents of S, destroying the previous
 *  contents of THIS, and invalidating S. */
private void copyFrom (HashMap<Key,Val> S)
  { size = S.size; bins = S.bins; loadFactor = S.loadFactor; }

/** The Entry in the list BIN whose key is KEY, or null if none. */
private Entry<Key,Val> find (Object key, Entry<Key,Val> bin) {
  for (Entry<Key,Val> e = bin; e != null; e = e.next)
    if (key == null && e.key == null || key.equals (e.key))
      return e;
  return null;
}

private int primeAbove (int N) { return a prime number ≥ N; }
}
```

Figure 7.2, continued: Private declarations for HashMap.

adds a positive constant that is relatively prime to the table size `bins.size ()` [why relatively prime?]. If we take the 17 keys of Figure 7.1:

$$\{81, 22, 38, 26, 86, 82, 0, 23, 39, 65, 83, 40, 9, -3, 84, 63, 5\}.$$

and insert them in this order into an array of size 23 using linear probing with increment 1 and $x \bmod 23$ as the hash function, the array of bins will contain the following keys:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0	23	63	26		5				9			81	82	83	38	39	86	40	65	-3	84	22

As you can see, several keys are displaced from their natural positions. For example, $84 \bmod 23 = 15$ and $63 \bmod 23 = 17$.

There is a *clustering* phenomenon associated with linear probing. The problem is simple to see with reference to the chaining method. If the sequence of entries examined in searching for some key is, say, $b_0, b_1, \ldots, b_n$, and if any other key should hash to one of these b_i, then the sequence of entries examined in searching for it will be part of the same sequence, $b_i, b_{i+1}, \ldots b_n$, even when the two keys have different hash values. In effect, what would be two distinct lists under chaining are merged together under linear probing, as much as doubling the effective average size of the bins for those keys. The longest chain for our set of integers (see Figure 7.1) was only 3 long. In the open-address example above, the longest chain is 9 items long (look at 63), even though only one other key (40) has the same hash value.

By having `nextProbe` increment the value by different amounts, depending on the original key—a technique known as *double hashing*—we can ameliorate this effect.

Deletion from an open-addressed hash table is non-trivial. Simply marking an entry as "unoccupied" can break the chain of colliding entries, and delete more than the desired item from the table [why?]. If deletion is necessary (often, it is not), we have to be more subtle. The interested reader is referred to volume 3 of Knuth, *The Art of Computer Programming.*

The problem with open-address schemes in general is that keys that would be in separate bins under the chaining scheme can compete with each other. Under the chaining scheme, if all entries are full and we search for an key that is not in the table, the search requires only as many probes (i.e., tests for equality) as there are keys in the table that have the same hashed value. Under any open-addressing scheme, it would require N probes to find that the key is not in the table. In my experience, the cost of the extra link nodes required for chaining is relatively unimportant, and for most purposes, I recommend using chaining rather than open-address schemes.

7.3   The hash function

This leaves the question of what to use for the function `hash`, used to choose the bin in which to place a key. In order for the map or set we are implementing to work properly, it is first important that our hash function satisfy two constraints:

1. For any key value, K, the value of `hash(`K`)` must remain constant while K is in the table (or the table must be reconstructed if `hash(`K`)` changes). during the execution of the program.

2. If two keys are equal (according to the `equals` method, or whatever equality test the hash table is using), then their `hash` values must be equal.

If either condition is violated, a key can effectively disappear from the table. On the other hand, it is *not* generally necessary for the value of `hash` to be constant from one execution of a program to the next, nor is it necessary that unequal keys have unequal hash values (although performance will clearly suffer if too many keys have the same hash value).

If the keys are simply non-negative integers, a simple and effective function is to use the remainder modulo the table size:

```
hash(X) == X % bins.size ();
```

For integers that might be negative, we have to make some provision. For example

```
hash(X) = (X & 0x7fffffff) % bins.size ();
```

has the effect of adding 2^{31} to any negative value of `X` first [why?]. Alternatively, if `bins.size ()` is odd, then

```
hash(X) = X % ((bins.size ()+1) / 2) + bins.size ()/2;
```

will also work [why?].

Handling non-numeric key values requires a bit more work. All Java objects have defined on them a `hashCode` method that we have used to convert `Objects` into integers (whence we can apply the procedure on integers above). The default implementation of `x.equals(y)` on `Object` is `x==y`—that is, that `x` and `y` are references to the same object. Correspondingly, the default implementation of `x.hashCode()` supplied by `Object` simply returns an integer value that is derived from the address of the object pointed to by `x`—that is, by the pointer value `x` treated as an integer (which is all it really is, behind the scenes). This default implementation is not suitable for cases where we want to consider two different objects to be the same. For example, the two `String`s computed by

```
String s1 = "Hello, world!", s2 = "Hello," + " " + "world!";
```

will have the property that `s1.equals (s2)`, but `s1 != s2` (that is, they are two different `String` objects that happen to contain the same sequence of characters). Hence, the default `hashCode` operation is not suitable for `String`, and therefore the `String` class overrides the default definition with its own.

For converting to an index into `bins`, we used the remainder operation. This obviously produces a number in range; what is not so obvious is why we chose the table sizes we did (primes not close to a power of 2). Suffice it to say that other choices of size tend to produce unfortunate results. For example, using a power of 2 means that the high-order bits of `X.hashCode()` get ignored.

If keys are not simple integers (strings, for example), a workable strategy is to first mash them into integers and then apply the remaindering method above. Here is a representative string-hashing function that does well empirically, taken from a C compiler by P. J. Weinberger[1]. It assumes 8-bit characters and 32-bit `ints`.

```
static int hash(String S)
{
    int h;
    h = 0;
    for (int p = 0; p < S.length (); p += 1) {
        h = (h << 4) + S.charAt(p);
        h = (h ^ ((h & 0xf0000000) >> 24)) & 0x0fffffff;
    }
    return h;
}
```

The Java String type has a different function for `hashCode`, which computes

$$\sum_{0 \leq i < n} c_i 31^{n-i-1}$$

using modular `int` arithmetic to get a result in the range -2^{31} to $2^{31} - 1$. Here, c_i denotes the ith character in the clsString.

7.4   Performance

Assuming the keys *are* evenly distributed, a hash table will do retrieval in constant time, regardless of N, the number of items contained. As indicated in the analysis we did in §4.1 about growing `ArrayLists`, insertion also has constant amortized cost (i.e., cost averaged over all insertions). Of course, if the keys are not evenly distributed, then we can see $\Theta(N)$ cost.

If there is a possibility that one hash function will sometimes have bad clustering problems, a technique known as *universal hashing* can help. Here, you choose a hash function at random from some carefully chosen set. On average over all runs of your program, your hash function will then perform well.

Exercises

7.1. Give an implementation for the `iterator` function over the `HashMap` representation given in §7.1, and the `Iterator` class it needs. Since we have chosen rather simple linked lists, you will have to use care in getting the `remove` operation right.

---

[1]The version here is adapted from Aho, Sethi, and Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, p. 436.

Chapter 8

Sorting and Selecting

At least at one time, most CPU time and I/O bandwidth was spent sorting (these days, I suspect more may be spent rendering MPEG files). As a result, sorting has been the subject of extensive study and writing. We will hardly scratch the surface here.

8.1 Basic concepts

The purpose of any sort is to permute some set of items that we'll call *records* so that they are sorted according to some ordering relation. In general, the ordering relation looks at only part of each record, the *key*. The records may be sorted according to more than one key, in which case we refer to the *primary key* and to *secondary keys*. This distinction is actually realized in the ordering function: record A comes before B iff either A's primary key comes before B's, or their primary keys are equal and A's secondary key comes before B's. One can extend this definition in an obvious way to hierarchies of multiple keys. For the purposes of this book, I'll usually assume that records are of some type `Record` and that there is an ordering relation on the records we are sorting. I'll write `before(A, B)` to mean that the key of A comes before that of B in whatever order we are using.

Although conceptually we move around the records we are sorting so as to put them in order, in fact these records may be rather large. Therefore, it is often preferable to keep around pointers to the records and exchange those instead. If necessary, the real data can be physically re-arranged as a last step. In Java, this is very easy of course, since "large" data items are always referred to by pointers.

Stability. A sort is called *stable* if it preserves the original order of records that have equal keys. Any sort can be made stable by (in effect) adding the original record position as a final secondary key, so that the list of keys (Bob, Mary, Bob, Robert) becomes something like (Bob.1, Mary.2, Bob.3, Robert.4).

Inversions. For some analyses, we need to have an idea of *how out-of-order* a given sequence of keys is. One useful measure is the number of *inversions* in the

sequence—in a sequence of keys $k_0, \ldots, k_{N-1}$, this is the number of pairs of integers, (i, j), such that $i < j$ and $k_i > k_j$. For example, there are two inversions in the sequence of words

        Charlie, Alpha, Bravo

and three inversions in

        Charlie, Bravo, Alpha.

When the keys are already in order, the number of inversions is 0, and when they are in reverse order, so that *every* pair of keys is in the wrong order, the number of inversions is $N(N-1)/2$, which is the number of pairs of keys. When all keys are originally within some distance D of their correct positions in the sorted permutation, we can establish a pessimistic upper bound of DN inversions in the original permutation.

Internal vs. external sorting.   A sort that is carried out entirely in primary memory is known as an *internal* sort. Those that involve auxiliary disks (or, in the old days especially, tapes) to hold intermediate results are called *external* sorts. The sources of input and output are irrelevant to this classification (one can have internal sorts on data that comes from an external file; it's just the intermediate files that matter).

8.2   A Little Notation

Many of the algorithms in these notes deal with (or can be thought of as dealing with) arrays. In describing or commenting them, we sometimes need to make assertions about the contents of these arrays. For this purpose, I am going to use a notation used by David Gries to make descriptive comments about my arrays. The notation
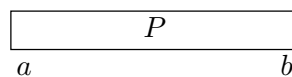
denotes a section of an array whose elements are indexed from a to b and that satisfies property P. It also asserts that $a \leq b + 1$; if $a > b$, then the segment is empty. I can also write
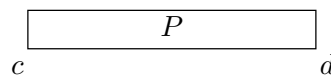
to describe an array segment in which items $c+1$ to $d-1$ satisfy P, and that $c < d$. By putting these segments together, I can describe an entire array. For example,
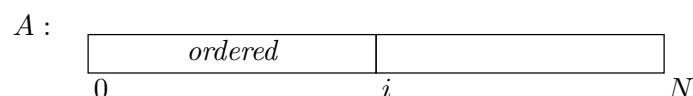
is true if the array A has N elements, elements 0 through $i-1$ are ordered, and $0 \le i \le N$. A notation such as

$$\boxed{P} \atop j$$

denotes a 1-element array segment whose index is j and whose (single) value satisfies P. Finally, I'll occasionally need to have simultaneous conditions on nested pieces of an array. For example,

refers to an array segment in which items 0 to $N-1$ satisfy P, items 0 to $i-1$ satisfy Q, $0 \le N$, and $0 \le i \le N$.

8.3 Insertion sorting

One very simple sort—and quite adequate for small applications, really—is the *straight insertion sort.* The name comes from the fact that at each stage, we insert an as-yet-unprocessed record into a (sorted) list of the records processed so far, as illustrated in Figure 8.2. The algorithm is shown in Figure 8.1.
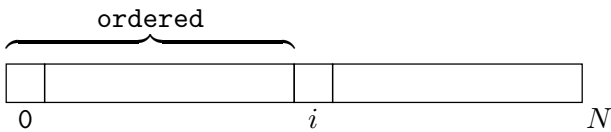
A common way to measure the time required to do a sort is to count the comparisons of keys (for Figure 8.1, the calls to `before`). The total (worst-case) time required by `insertionSort` is $\sum_{0<i<N} C_{IL}(i)$, where $C_{IL}(m)$ is the cost of the inner (`j`) loop when `i=` m, and N is the size of `A`. Examination of the inner loop shows that the number of comparisons required is equal to the number of records numbered 0 to `i-1` whose keys larger than that of `x`, plus one if there is at least one smaller key. Since `A[0..i-1]` is sorted, it contains no inversions, and therefore, the number of elements after `X` in the sorted part of `A` happens to be equal to the number of inversions in the sequence `A[0],..., A[i]` (since `X` is `A[i]`). When `X` is inserted correctly, there will be no inversions in the resulting sequence. It is fairly easy to work out from that point that the running time of `insertionSort`, measured in key comparisons, is bounded by $I + N - 1$, where I is the total number of inversions in the original argument to `insertionSort`. Thus, the more sorted an array is to begin with, the faster `insertionSort` runs.
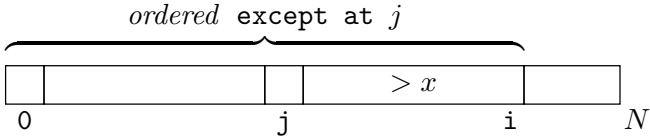
8.4 Shell's sort
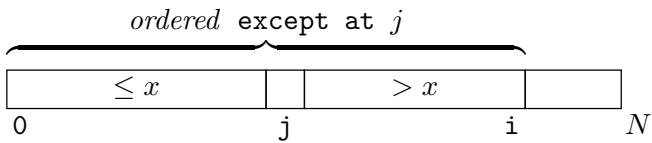
The problem with insertion sort can be seen by examining the worst case—where the array is initially in reverse order. The keys are a great distance from their final resting places, and must be moved one slot at a time until they get there. If keys could be moved great distances in little time, it might speed things up a bit.

```
/** Permute the elements of A to be in ascending order. */
static void insertionSort(Record[] A) {
   int N = A.length;
   for (int i = 1; i < N; i += 1) {
```



```
      Record x = A[i];
      int j;
      for (j = i; j > 0 && before(x, A[j-1]); j -= 1) {
```



```
         A[j] = A[j-1];
      }
```



```
      A[j] = x;
   }
}
```

Figure 8.1: Program for performing insertion sort on an array. The `before` function is assumed to embody the desired ordering relation.

| 13 | | 9 | 10 | 0 | 22 | 12 | 4 |

| 9 | 13 | | 10 | 0 | 22 | 12 | 4 |

| 9 | 10 | 13 | | 0 | 22 | 12 | 4 |

| 0 | 9 | 10 | 13 | | 22 | 12 | 4 |

| 0 | 9 | 10 | 13 | 22 | | 12 | 4 |

| 0 | 9 | 10 | 12 | 13 | 22 | | 4 |

| 0 | 4 | 9 | 10 | 12 | 13 | 22 |
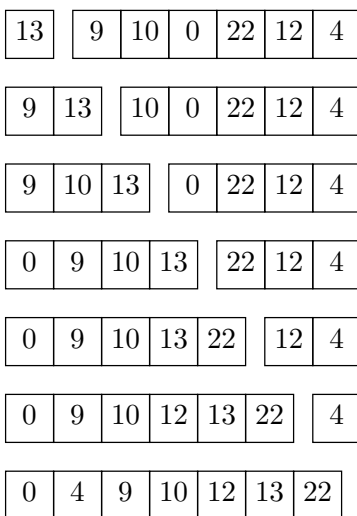
Figure 8.2: Example of insertion sort, showing the array before each call of `insertElement`. The gap at each point separates the portion of the array known to be sorted from the unprocessed portion.

```
/** Permute the elements of KEYS, which must be distinct,
 *   into ascending order. */
static void distributionSort1(int[] keys) {
    int N = keys.length;
    int L = min(keys), U = max(keys);

    java.util.BitSet b = new java.util.BitSet();
    for (int i = 0; i < N; i += 1)
        b.set(keys[i] - L);
    for (int i = L, k = 0; i <= U; i += 1)
        if (b.get(i-L)) {
            keys[k] = i; k += 1;
        }
}
```

Figure 8.3: Sorting distinct keys from a reasonably small and dense set. Here, assume that the functions min and max return the minimum and maximum values in an array. Their values are arbitrary if the arrays are empty.

This is the idea behind Shell's sort[1]. We choose a diminishing sequence of strides, $s_0 > s_1 > \ldots > s_{m-1}$, typically choosing $s_{m-1} = 1$. Then, for each j, we divide the N records into the s_j interleaved sequences

$$R_0, R_{s_j}, R_{2s_j}, \ldots,$$
$$R_1, R_{s_j+1}, R_{2s_j+1}, \ldots$$
$$\ldots$$
$$R_{s_j-1}, R_{2s_j-1}, \ldots$$

and sort each of these using insertion sort. Figure 8.4 illustrates the process with a vector in reverse order (requiring a total of 49 comparisons as compared with 120 comparisons for straight insertion sort).

A good sequence of s_j turns out to be $s_j = \lfloor 2^{m-j} - 1 \rfloor$, where $m = \lfloor \lg N \rfloor$. With this sequence, it can be shown that the number of comparisons required is $O(N^{1.5})$, which is considerably better than $O(N^2)$. Intuitively, the advantages of such a sequence—in which the successive s_j are relatively prime—is that on each pass, each position of the vector participates in a sort with a new set of other positions. The sorts get "jumbled" and get more of a chance to improve the number of inversions for later passes.

---

[1]Also known as "shellsort." Knuth's reference: Donald L. Shell, in the *Communications of the ACM* **2** (July, 1959), pp. 30–32.

	#I	#C
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	120	-
0 14 13 12 11 10 9 8 7 6 5 4 3 2 1 15	91	1
0 7 6 5 4 3 2 1 14 13 12 11 10 9 8 15	42	9
0 1 3 2 4 6 5 7 8 10 9 11 13 12 14 15	4	20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0	19

Figure 8.4: An illustration of Shell's sort, starting with a vector in reverse order. The increments are 15, 7, 3, and 1. The column marked *#I* gives the number of inversions remaining in the array, and the column marked *#C* gives the number of key comparisons required to obtain each line from its predecessor. The arcs underneath the arrays indicate which subsequences of elements are processed at each stage.

8.5   Distribution counting

When the range of keys is restricted, there are a number of optimizations possible. In Column #1 of his book *Programming Pearls*[2], Jon Bentley gives a simple algorithm for the problem of sorting N distinct keys, all of which are in a range of integers so limited that the programmer can build a vector of bits indexed by those integers. In the program shown in Figure 8.3, I use a Java `BitSet`, which is abstractly a set of non-negative integers (implemented as a packed array of 1-bit quantities).

Let's consider a more general technique we can apply even when there are multiple records with the same key. Assume that the keys of the records to be sorted are in some reasonably small range of integers. Then the function `distributionSort2` shown in Figure 8.6 sorts N records stably, moving them from an input array (`A`) to a different output array (`B`). It computes the correct final position in `B` for each record in `A`. To do so, it uses the fact that the position of any record in `B` is supposed to be the number the records that either have smaller keys than it has, or that have the same key, but appear before it in `A`. Figure 8.5 contains an example of the program in operation.

8.6   Selection sort

In insertion sort, we determine an item's final position piecemeal. Another way to proceed is to place each record in its final position in one move by selecting the smallest (or largest) key at each step. The simplest implementation of this idea is *straight selection sorting,* as follows.

```
static void selectionSort(Record[] A)
{
   int N = A.length;
   for (int i = 0; i < N-1; i += 1) {

      /*  A:                                  */
      int m, j;
      for (j = i+1, m = i; j < N; j += 1)
         if (before(A[j], A[m]) m = j;
      /*  Now A[m] is the smallest element in A[i..N-1] */
      swap(A, i, m);
   }
}
```

Here, `swap(A,i,m)` is assumed to swap elements `i` and `m` of `A`. This sort is not stable; the swapping of records prevents stability. On the other hand, the program can be

---

[2]Addison-Wesley, 1986. By the way, that column makes very nice "consciousness-raising" column on the subject of appropriately-engineered solutions. I highly recommend both this book and his *More Programming Pearls,* Addison-Wesley, 1988.

A:

3/A	2/B	2/C	1/D	4/E	2/F	3/G

count_1:

0	1	3	2	1

count_2:

0	1	4	6	7

B_0:

				3/A		

count:

0	1	5	6	7

B_1:

	2/B			3/A		

count:

0	2	5	6	7

B_2:

	2/B	2/C		3/A		

count:

0	3	5	6	7

B_3:

1/D	2/B	2/C		3/A		

count:

1	3	5	6	7

B_4:

1/D	2/B	2/C		3/A		4/E

count:

1	3	5	7	7

B_5:

1/D	2/B	2/C	2/F	3/A		4/E

count:

1	4	5	7	7

B_6:

1/D	2/B	2/C	2/F	3/A	3/G	4/E

count:

1	4	6	7	7

Figure 8.5: Illustration of the `distributionSort2` program. The values to be sorted are shown in the array marked `A`. The keys are the numbers to the left of the slashes. The data are sorted into the array `B`, shown at various points in the algorithm. The labels at the left refer to points in the program in Figure 8.6. Each point B_k indicates the situation at the end of the last loop where $i = k$. The role of array `count` changes. First (at count_1) `count[k-1]` contains the number of instances of key `(k-1)-1`. Next (at count_2), it contains the number of instances of keys less than `k-1`. In the B_i lines, `count`$[k-1]$ indicates the position (in `B`) at which to put the next instance of key k. (It's `k-1` in these places, rather than `k`, because 1 is the smallest key.

```
/** Assuming that A and B are not the same array and are of
 *  the same size, sort the elements of A stably into B.
 */
void distributionSort2(Record[] A, Record[] B)
{
   int N = A.length;

   int L = min(A), U = max(A);

   /* count[i-L] will contain the number of items <i */
   // NOTE: count[U-L+1] is not terribly useful, but is
   // included to avoid having to test for for i == U in
   // the first i loop below.
   int[] count = new int[U-L+2];

   // Clear count: Not really needed in Java, but a good habit
   // to get into for other languages (e.g., C, C++).
   for (int j = L; j <= U+1; j += 1)
      count[j-L] = 0;

   for (int i = 0; i < N; i += 1)
      count[key(A[i]) - L + 1] += 1;
   /* Now count[i-L] == # of records whose key is equal to i-1 */
   // See Figure 8.5, point count1.

   for (int j = L+1; j <= U; j += 1)
       count[j-L] += count[j-L-1];
   /* Now count[k-L] == # of records whose key is less than k,
    * for all k, L <= k <= U. */
   // See Figure 8.5, point count2.

   for (i = 0; i < N; i += 1) {
      /* Now count[k-L] == # of records whose key is less than k,
       * or whose key is k and have already been moved to B. */
      B[count[key(A[i])-L]] = A[i];
      count[key(A[i])-L] += 1;
      // See Figure 8.5, points B0–B6.
   }
}
```

Figure 8.6: Distribution Sorting. This program assumes that key(R) is an integer.

modified to produce its output in a separate output array, and then it is relatively easy to maintain stability [how?].

It should be clear that the algorithm above is insensitive to the data. Unlike insertion sort, it *always* takes the same number of key comparisons—$N(N-1)/2$. Thus, in this naive form, although it is very simple, it suffers in comparison to insertion sort (at least on a sequential machine).

On the other hand, we have seen another kind of selection sort before—heapsort (from §6.4) is a form of selection sort that (in effect) keeps around information about the results of comparisons from each previous pass, thus speeding up the minimum selection considerably.

8.7   Exchange sorting: Quicksort

One of the most popular methods for internal sorting was developed by C. A. R. Hoare[3]. Evidently much taken with the technique, he named it "quicksort." The name is actually quite appropriate. The basic algorithm is as follows.

```
static final int K = ...;

void quickSort(Record A[])
{
    quickSort(A,0,A.length-1);
    insertionSort(A);
}

/* Permute A[L..U] so that all records are < K away from their */
/* correct positions in sorted order.  Assumes K > 0. */
void quickSort(Record[] A, int L, int U)
{
   if (U−L+1 > K) {
       Choose Record T = A[p], where L ≤ p ≤ U;
 P:   Set i and permute A[L..U] to establish the partitioning
          condition:
```

key $\leq$ key(T)	T	key $\geq$ key(T)
0	i	N

                                                                        ;

```
       quickSort(A, L, i-1); quickSort(A, i+1, U);
   }
}
```

Here, K is a constant value that can be adjusted to tune the speed of the sort. Once the approximate sort gets all records within a distance K-1 of their final locations, the final insertion sort proceeds in $O(KN)$ time. If T can be chosen so that its

[3]Knuth's reference: *Computing Journal* **5** (1962), pp. 10–15.

key is near the median key for the records in `A`, then we can compute roughly that the time in key comparisons required for performing `quicksort` on N records is approximated by $C(N)$, defined as follows.

$$
\begin{aligned}
C(K) &= 0 \\
C(N) &= N - 1 + 2C(\lfloor N/2 \rfloor)
\end{aligned}
$$

This assumes that we can partition an N-element array in $N-1$ comparisons, which we'll see to be possible. We can get a sense for the solution by considering the case $N = 2^m K$:

$$
\begin{aligned}
C(N) &= 2^m K + 2C(2^{m-1}K) \\
&= 2^m K - 1 + 2^m K - 2 + 4C(2^{m-2}K) \\
&= \underbrace{2^m K + \ldots + 2^m K}_{m} - 1 - 2 - 4 - \ldots - 2^{m-1} + C(K) \\
&= m2^m K - 2^m + 1 \\
&\in \Theta(m2^m K) = \Theta(N \lg N)
\end{aligned}
$$

(since $\lg(2^m K) = m \lg K$).

Unfortunately, in the worst case—where the partition `T` has the largest or smallest key, quicksort is essentially a straight selection sort, with running time $\Theta(N^2)$. Thus, we must be careful in the choice of the partitioning element. One technique is to choose a random record's key for `T`. This is certainly likely to avoid the bad cases. A common choice for `T` is the *median* of `A[L]`, `A[(L+U)/2]`, and `A[U]`, which is also unlikely to fail.

Partitioning. This leaves the small loose end of how to partition the array at each stage (step P in the program above). There are many ways to do this. Here is one due to Nico Lomuto—not the fastest, but simple.

```
P:
swap(A, L, p);
i = L;
for (int j = L+1; j <= U; j += 1) {

   /*  A[L..U]:                                      */
                  +---+------+------+-----+-----+
                  | T |  <T  |  ≥T  |     |     |
                  +---+------+------+-----+-----+
                  L      i       j      U
   if (before(A[j],T)) {
      i += 1;
      swap(A, j, i);
   }
}

   /*  A[L..U]:                          */
               +---+-------+------+
               | T |   <T  |  ≥T  |
               +---+-------+------+
               L       i      U
```

```
swap(A, L, i);
/*  A[L..U]:                        */
```



```
            <T        ≥T
        L         i        U
```

Some authors go to the trouble of developing non-recursive versions of quick-sort, evidently under the impression that they are thereby vastly improving its performance. This view of the cost of recursion is widely held, so I suppose I can't be surprised. However, a quick test using a C version indicated about a 3% improvement using his iterative version. This is hardly worth obscuring one's code to obtain.

8.8 Merge sorting

Quicksort was a kind of divide-and-conquer algorithm[4] that we might call "try to divide-and-conquer," since it is not guaranteed to succeed in dividing the data evenly. An older technique, known as merge sorting, is a form of divide-and-conquer that does guarantee that the data are divided evenly.

At a high level, it goes as follows.

```
/** Sort items A[L..U]. */
static void mergeSort(Record[] A, int L, int U)
{
    if (L >= U)
        return;
    mergeSort(A, L, (L+U)/2);
    mergeSort(A, (L+U)/2+1, U);
    merge(A, L, (L+U)/2, A, (L+U)/2+1, U, A, L);
}
```

The `merge` program has the following specification

```
/** Assuming V0[L0..U0] and V1[L1..U1] are each sorted in */
/*  ascending order by keys, set V2[L2..U2] to the sorted contents */
/*  of V0[L0..U0], V1[L1..U1].  (U2 = L2+U0+U1-L0-L1+1). */
void merge(Record[] V0, int L0, int U0, Record[] V1, int L1, int U1,
           Record[] V2, int L2)
```

Since `V0` and `V1` are in ascending order already, it is easy to do this in $\Theta(N)$ time, where $N = U2 - L2 + 1$, the combined size of the two arrays. Merging progresses through the arrays from left to right. That makes it well-suited for computers with small memories and lots to sort. The arrays can be on secondary storage devices

---

[4]The term *divide-and-conquer* is used to describe algorithms that divide a problem into some number of smaller problems, and then combine the answers to those into a single result.

that are restricted to *sequential access*—i.e., that require reading or writing the arrays in increasing (or decreasing) order of index[5].

The real work is done by the merging process, of course. The pattern of these merges is rather interesting. For simplicity, consider the case where N is a power of two. If you trace the execution of `mergeSort`, you'll see the following pattern of calls on `merge`.

Call #	V0	V1
0.	`A[0]`	`A[1]`
1.	`A[2]`	`A[3]`
2.	`A[0..1]`	`A[2..3]`
3.	`A[4]`	`A[5]`
4.	`A[6]`	`A[7]`
5.	`A[4..5]`	`A[6..7]`
6.	`A[0..3]`	`A[4..7]`
7.	`A[8]`	`A[9]`

etc.

We can exploit this pattern to good advantage when trying to do merge sorting on linked lists of elements, where the process of dividing the list in half is not as easy as it is for arrays. Assume that records are linked together into `List`s. The program below shows how to perform a merge sort on these lists; Figure 8.7 illustrates the process. The program maintains a *binomial comb* of sorted sublists, `comb[0 .. M-1]`, such that the list in `comb[i]` is either null or has length 2^i.

```
/** Permute the Records in List A so as to be sorted by key. */
static void mergeSort(List<Record> A)
{
    int M = a number such that 2^{M-1} ≥ length of A;
    List<Record>[] comb = new List<Record>[M];

    for (int i = 0; i < M; i += 1)
        comb[i] = new LinkedList<Record> ();
    for (Record R : A)
        addToComb(comb, R);
    A.clear ();
    for (List<Record> L : comb)
        mergeInto(A, L);
}
```

---

[5]A familiar movie cliché of decades past was spinning tape units to indicate that some piece of machinery was a computer (also operators flipping console switches—something one almost *never* really did during normal operation). When those images came from footage of real computers, the computer was most likely sorting.

At each point, the comb contains sorted lists that are to be merged. We first build up the comb one new item at a time, and then take a final pass through it, merging all its lists. To add one element to the comb, we have

```
/** Assuming that each C[i] is a sorted list whose length is either 0
 *  or 2^i elements, adds P to the items in C so as to
 *  maintain this same condition. */
static void addToComb(List<Record> C[], Record p)
{
   if (C[0].size () == 0) {
      C[0].add (p);
      return;
   } else if (before(C[0].get (0), p))
      C[0].add (p);
   else
      C[0].add (p, 0);
   // Now C[0] contains 2 items

   int i;
   for (i = 1; C[i].size () != 0; i += 1)
      mergeLists(C[i], C[i-1]);
   C[i] = C[i-1]; C[i-1] = new LinkedList ();
}
```

I leave to you the `mergeLists` procedure:

```
/** Merge L1 into L0, producing a sorted list containing all the
 *  elements originally in L0 and L1.  Assumes that L0 and L1 are
 *  each sorted initially (according to the before ordering).
 *  The result ends up in L0; L1 becomes empty. */
static void mergeLists (List<Record> L0, List<Record> L1) ...
```

8.8.1   Complexity

The optimistic time estimate for quicksort applies in the worst case to merge sorting, because merge sorts really do divide the data in half with each step (and merging of two lists or arrays takes linear time). Thus, merge sorting is a $\Theta(N \lg N)$ algorithm, with N the number of records. Unlike quicksort or insertion sort, merge sorting as I have described it is generally insensitive to the ordering of the data. This changes somewhat when we consider external sorting, but $O(N \lg N)$ comparisons remains an upper bound.

8.9   Speed of comparison-based sorting

I've presented a number of algorithms and have claimed that the best of them require $\Theta(N \lg N)$ comparisons in the worst case. There are several obvious questions to

L: (9, 15, 5, 3, 0, 6, 10, −1, 2, 20, 8)

0: 0
1: 0
2: 0
3: 0

0 elements processed

L: (15, 5, 3, 0, 6, 10, −1, 2, 20, 8)   L: (5, 3, 0, 6, 10, −1, 2, 20, 8)

0: 1 •⟶ (9)   0: 0
1: 0   1: 1 •⟶ (9, 15)
2: 0   2: 0
3: 0   3: 0

1 element processed   2 elements processed

L: (3, 0, 6, 10, −1, 2, 20, 8)   L: (0, 6, 10, −1, 2, 20, 8)

0: 1 •⟶ (5)   0: 0
1: 1 •⟶ (9, 15)   1: 0
2: 0   2: 1 •⟶ (3, 5, 9, 15)
3: 0   3: 0

3 elements processed   4 elements processed

L: (10, −1, 2, 20, 8)   L:

0: 0   0: 1 •⟶ (8)
1: 1 •⟶ (0, 6)   1: 1 •⟶ (2, 20)
2: 1 •⟶ (3, 5, 9, 15)   2: 0
3: 0   3: 1 •⟶ (−1, 0, 3, 5, 6, 9, 10, 15)

6 elements processed   11 elements processed
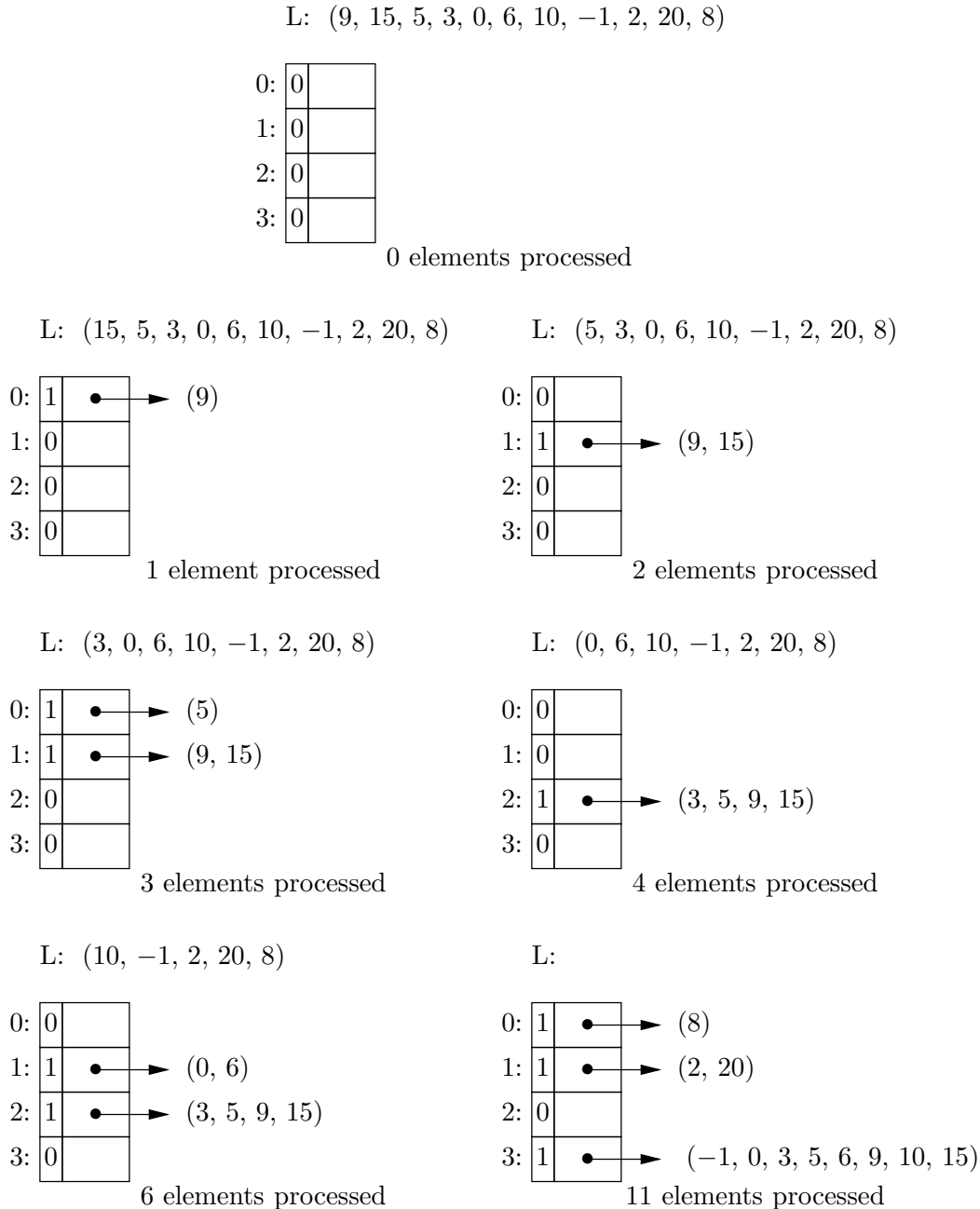
Figure 8.7: Merge sorting of lists, showing the state of the "comb" after various numbers of items from the list L have been processed. The final step is to merge the lists remaining in the comb after all 11 elements from the original list have been added to it. The 0s and 1s in the small boxes are decorations to illustrate the pattern of merges that occurs. Each empty box has a 0 and each non-empty box has a 1. If you read the contents of the four boxes as a single binary number, units bit on top, it equals the number of elements processed.

ask about this bound. First, how do "comparisons" translate into "instructions"? Second, can we do better than $N \lg N$?

The point of the first question is that I have been a bit dishonest to suggest that a comparison is a constant-time operation. For example, when comparing strings, the size of the strings matters in the time required for comparison in the worst case. Of course, on the average, one expects not to have to look too far into a string to determine a difference. Still, this means that to correctly translate comparisons into instructions, we should throw in another factor of the length of the key. Suppose that the N records in our set all have distinct keys. This means that the keys themselves have to be $\Omega(\lg N)$ long. Assuming keys are no longer than necessary, and assuming that comparison time goes up proportionally to the size of a key (in the worst case), this means that sorting *really* takes $\Theta(N(\lg N)^2)$ time (assuming that the time required to move one of these records is at worst proportional to the size of the key).

As to the question about whether it is possible to do better than $\Theta(N \lg N)$, the answer is that *if* the only information we can obtain about keys is how they compare to each other, then we cannot do better than $\Theta(N \lg N)$. That is, $\Theta(N \lg N)$ comparisons is a lower bound on the worst case of all possible sorting algorithms that use comparisons.

The proof of this assertion is instructive. A sorting program can be thought of as first performing a sequence of comparisons, and then deciding how to permute its inputs, based *only* on the information garnered by the comparisons. The two operations actually get mixed, of course, but we can ignore that fact here. In order for the program to "know" enough to permute two different inputs differently, these inputs must cause different sequences of comparison results. Thus, we can represent this idealized sorting process as a tree in which the leaf nodes are permutations and the internal nodes are comparisons, with each left child containing the comparisons and permutations that are performed when the comparison turns out true and the right child containing those that are performed when the comparison turns out false. Figure 8.8 illustrates this for the case $N = 3$. The height of this tree corresponds to the number of comparisons performed. Since the number of possible permutations (and thus leaves) is $N!$, and the minimal height of a binary tree with M leaves is $\lceil \lg M \rceil$, the minimal height of the comparison tree for N records is roughly $\lg(N!)$. Now

$$\begin{aligned}
\lg N! &= \lg N + \lg(N-1) + \ldots + 1 \\
&\leq \lg N + \lg N + \ldots + \lg N \\
&= N \lg N \\
&\in O(N \lg N)
\end{aligned}$$

and also (taking N to be even)

$$\begin{aligned}
\lg N! &\geq \lg N + \lg(N-1) + \ldots + \lg(N/2) \\
&\geq (N/2 + 1) \lg(N/2) \\
&\in \Omega(N \lg N)
\end{aligned}$$
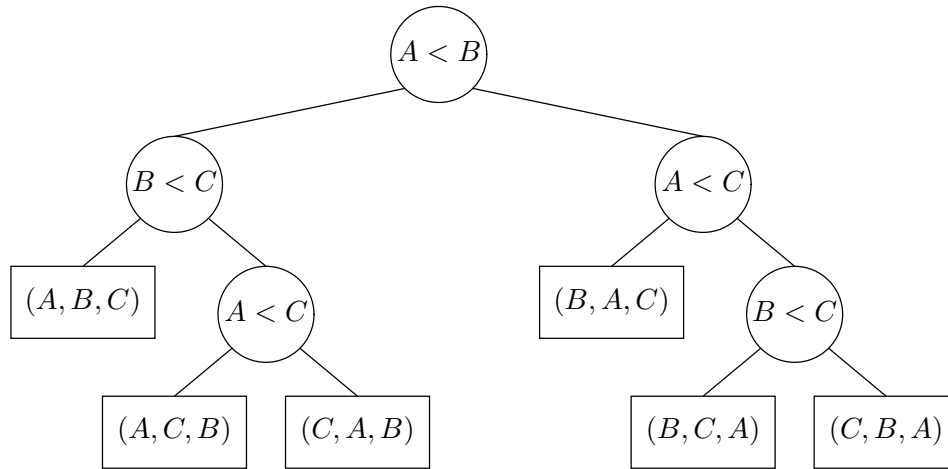
Figure 8.8: A comparison tree for $N = 3$. The three values being sorted are A, B, and C. Each internal node indicates a test. The left children indicate what happens when the test is successful (true), and the right children indicate what happens if it is unsuccessful. The leaf nodes (rectangular) indicate the ordering of the three values that is uniquely determined by the comparison results that lead down to them. We assume here that A, B, and C are distinct. This tree is optimal, demonstrating that three comparisons are needed in the worst case to sort three items.

so that

$$\lg N! \in \Theta(N \lg N).$$

Thus *any* sorting algorithm that uses only (true/false) key comparisons to get information about the order of its input's keys requires $\Theta(N \lg N)$ comparisons in the worst case to sort N keys.

8.10   Radix sorting

To get the result in §8.9, we assumed that the only examination of keys available was comparing them for order. Suppose that we are *not* restricted to simply comparing keys. Can we improve on our $O(N \lg N)$ bounds? Interestingly enough, we can, sort of. This is possible by means of a technique known as *radix sort*.

Most keys are actually sequences of fixed-size pieces (characters or bytes, in particular) with a lexicographic ordering relation—that is, the key $k_0 k_1 \cdots k_{n-1}$ is less than $k'_0 k'_1 \cdots k'_{n-1}$ if $k_0 < k'_0$ or $k_0 = k'_0$ and $k_1 \cdots k_{n-1}$ is less than $k'_1 \cdots k'_{n-1}$ (we can always treat the keys as having equal length by choosing a suitable padding character for the shorter string). Just as in a search trie we used successive characters in a set of keys to distribute the strings amongst subtrees, we can use successive characters of keys to sort them. There are basically two varieties of algorithm— one that works from least significant to most significant digit (LSD-first) and one that works from most significant to least significant digit (MSD-first). I use "digit"

here as a generic term encompassing not only decimal digits, but also alphabetic characters, or whatever is appropriate to the data one is sorting.

8.10.1 LSD-first radix sorting

The idea of the LSD-first algorithm is to first use the least significant character to order all records, then the second-least significant, and so forth. At each stage, we perform a *stable* sort, so that if the k most significant characters of two records are identical, they will remain sorted by the remaining, least significant, characters. Because characters have a limited range of values, it is easy to sort them in linear time (using, for example, `distributionSort2`, or, if the records are kept in a linked list, by keeping an array of list headers, one for each possible character value). Figure 8.9 illustrates the process.

LSD-first radix sort is precisely the algorithm used by card sorters. These machines had a series of bins and could be programmed (using plugboards) to drop cards from a feeder into bins depending on what was punched in a particular column. By repeating the process for each column, one ended up with a sorted deck of cards.

Each distribution of a record to a bin takes (about) constant time (assuming we use pointers to avoid moving large amounts of data around). Thus, the total time is proportional to the total amount of key data—which is the total number of bytes in all keys. In other words, radix sorting is $O(B)$ where B is the total number of bytes of key data. If keys are K bytes long, then $B = NK$, where N is the number of records. Since merge sorting, heap sorting, etc., require $O(N \lg N)$ comparisons, each requiring in the worst case K time, we get a total time of $O(NK \lg N) = O(B \lg N)$ time for these sorts. Even if we assume constant comparison time, if keys are no longer than they have to be (in order to provide N different keys we must have $K \geq \log_C N$, where C is the number of possible characters), then radix sorting is also $O(N \lg N)$.

Thus, relaxing the constraint on what we can do to keys yields a fast sorting procedure, at least in principle. As usual, the Devil is in the details. If the keys are considerably longer than $\log_C N$, as they very often are, the passes made on the last characters will typically be largely wasted. One possible improvement, which Knuth credits to M. D. Maclaren, is to use LSD-first radix sort on the first two characters, and then finish with an insertion sort (on the theory that things will almost be in order after the radix sort). We must fudge the definition of "character" for this purpose, allowing characters to grow slightly with N. For example, when $N = 100000$, Maclaren's optimal procedure is to sort on the first and second 10-bit segments of the key (on an 8-bit machine, this is the first 2.25 characters). Of course, this technique can, in principle, make no guarantees of $O(B)$ performance.

8.10.2 MSD-first radix sorting

Performing radix sort starting at the most significant digit probably seems more natural to most of us. We sort the input by the first (most-significant) character into C (or fewer) subsequences, one for each starting character (that is, the first

Initial: set, cat, cad, con, bat, can, be, let, bet

```
                                        bet
                                        let
                                        bat
                            can         cat
          be        cad     con         set
         ‾‾‾‾‾     ‾‾‾‾‾    ‾‾‾‾‾       ‾‾‾‾‾

          ‘␣’       ‘d’     ‘n’         ‘t’
```

After first pass: be, cad, con, can, set, cat, bat, let, bet

```
          bat       bet
          cat       let
          can       set
          cad       be      con
         ‾‾‾‾‾     ‾‾‾‾‾    ‾‾‾‾‾

          ‘a’       ‘e’     ‘o’
```

After second pass: cad, can, cat, bat, be, set, let, bet, con

```
                    con
          bet       cat
          be        can
          bat       cad     let      set
         ‾‾‾‾‾     ‾‾‾‾‾    ‾‾‾‾‾    ‾‾‾‾‾

          ‘b’       ‘c’     ‘l’      ‘s’
```

After final pass: bat, be, bet, cad, can, cat, con, let, set

Figure 8.9: An example of a LSD-first radix sort. Each pass sorts by one character, starting with the last. Sorting consists of distributing the records to bins indexed by characters, and then concatenating the bins' contents together. Only non-empty bins are shown.

A	posn
⋆ set, cat, cad, con, bat, can, be, let, bet	0
⋆ bat, be, bet / cat, cad, con, can / let / set	1
bat / ⋆ be, bet / cat, cad, con, can / let / set	2
bat / be / bet / ⋆ cat, cad, con, can / let / set	1
bat / be / bet / ⋆ cat, cad, can / con / let / set	2
bat / be / bet / cad / can / cat / con / let / set	

Figure 8.10: An example of an MSD radix sort on the same data as in Figure 8.9. The first line shows the initial contents of A and the last shows the final contents. Partially-sorted segments that agree in their initial characters are separated by single slash (/) characters. The ⋆ character indicates the segment of the array that is about to be sorted and the `posn` column shows which character position is about to be used for the sort.

character of all the keys in any given subsequence is the same). Next, we sort each of the subsequences that has more than one key individually by its second character, yielding another group of subsequences in which all keys in any given subsequence agree in their first two characters. This process continues until all subsequences are of length 1. At each stage, we order the subsequences, so that one subsequence precedes another if all its strings precede all those in the other. When we are done, we simply write out all the subsequences in the proper order.

The tricky part is keeping track of all the subsequences so that they can be output in the proper order at the end and so that we can quickly find the next subsequence of length greater than one. Here is a sketch of one technique for sorting an array; it is illustrated in Figure 8.10.

```
static final int ALPHA = size of alphabet of digits;

/** Sort A[L..U] stably, ignoring the first k characters in each key.  */
static void MSDradixSort(Record[] A, int L, int U, int k) {
    int[] countLess = new int[ALPHA+1];

    Sort A[L..U] stably by the kth character of each key, and for each
        digit, c, set countLess[c] to the number of records in A
        whose kth character comes before c in alphabetical order.

    for (int i = 0; i <= ALPHA; i += 1)
        if (countLess[i+1] - countLess[i] > 1)
            MSDradixSort(A, L + countLess[i],
                         L + countLess[i+1] - 1, k+1);
}
```

8.11   Using the library

Notwithstanding all the trouble we've taken in this chapter to look at sorting algorithms, in most programs you shouldn't even think about writing your own sorting subprogram! Good libraries provide them for you. The Java standard library has a class called `java.util.Collections`, which contains only static definitions of useful utilities related to Collections. For sorting, we have

```
/** Sort L stably into ascending order, as defined by C.  L must
 *  be modifiable, but need not be expandable.  */
public static <T> void sort (List<T> L, Comparator<? super T> c) { ⋯ }
/** Sort L into ascending order, as defined by the natural ordering
 *  of the elements. L must be modifiable, but need not be expandable. */
public static <T extends Comparable<T>> void sort (List<T> L) { ⋯ }
```

These two methods use a form of mergesort, guaranteeing $O(N \lg N)$ worst-case performance. Given these definitions, you should not generally need to write your own sorting routine unless the sequence to be sorted is extremely large (in particular, if it requires external sorting), if the items to be sorted have primitive types (like **int**), or you have an application where it is necessary to squeeze every single microsecond out of the algorithm (a rare occurrence).

8.12   Selection

Consider the problem of finding the *median* value in an array—a value in the array with as many array elements less than it as greater than it. A brute-force method of finding such an element is to sort the array and choose the middle element (or *a* middle element, if the array has an even number of elements). However, we can do substantially better.

The general problem is *selection*—given a (generally unsorted) sequence of elements and a number k, find the kth value in the sorted sequence of elements. Finding a median, maximum, or minimum value is a special case of this general problem. Perhaps the easiest efficient method is the following simple adaptation of Hoare's quicksort algorithm.

```
/** Assuming 0<=k<N, return a record of A whose key is kth smallest
 *  (k=0 gives the smallest, k=1, the next smallest, etc.). A may
 *  be permuted by the algorithm. */
Record select(Record[] A, int L, int U, int k) {
    Record T = some member of A[L..U];
    Permute A[L..U] and find p to establish the partitioning
      condition:
```

key ≤ key(T)	T	key ≥ key(T)
L	p	U

;
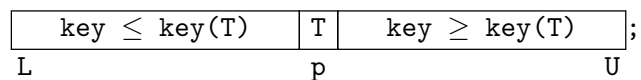
```
        if (p-L == k)
            return T;
        if (p-L < k)
            return select(A, p+1, U, k - p + L - 1);
        else
            return select(A, L, p-1, k);
    }
```

The key observation here is that when the array is partitioned as for quicksort, the value T is the $(p - L)$st smallest element; the $p - L$ smallest record keys will be in A[L..p-1]; and the larger record keys will be in A[p+1..U]. Hence, if $k < p - L$, the kth smallest key is in the left part of A and if $k > p$, it must be the $(k - p + L - 1)$st largest key in the right half.

Optimistically, assuming that each partition divides the array in half, the recurrence governing cost here (measured in number of comparisons) is

$$
\begin{aligned}
C(1) &= 0 \\
C(N) &= N + C(\lceil N/2 \rceil).
\end{aligned}
$$

where $N = U - L + 1$. The N comes from the cost of partitioning, and the $C(\lceil N/2 \rceil)$ from the recursive call. This differs from the quicksort and mergesort recurrences by the fact that the multiplier of $C(\cdots)$ is 1 rather than 2. For $N = 2^m$ we get

$$
\begin{aligned}
C(N) &= 2^m + C(2^{m-1}) \\
&= 2^m + 2^{m-1} + C(2^{m-2}) \\
&= 2^{m+1} - 1 = 2N - 1 \\
&\in \Theta(N)
\end{aligned}
$$

This algorithm is only probabilistically good, just as was quicksort. There are selection algorithms that *guarantee* linear bounds, but we'll leave them to a course on algorithms.

Exercises

8.1. You are given two sets of keys (i.e., so that neither contains duplicate keys), S_0 and S_1, both represented as arrays. Assuming that you can compare keys for "greater than or equal to," how would you compute the intersection of the S_0 and S_1, and how long would it take?

8.2. Given a large list of words, how would you quickly find all anagrams in the list? (An *anagram* here is a word in the list that can be formed from another word on the list by rearranging its letters, as in "dearth" and "thread").

8.3. Suppose that we have an array, D, of N records. Without modifying this array, I would like to compute an N-element array, P, containing a permutation of the integers 0 to $N-1$ such that the sequence $D[P[0]], \ D[P[1]], \ldots, D[P[N-1]]$ is sorted *stably*. Give a general method that works with any sorting algorithm (stable or not) and doesn't require any additional storage (other than that normally used by the sorting algorithm).

8.4. A very simple spelling checker simply removes all ending punctuation from its words and looks up each in a dictionary. Compare ways of doing this from the classes in the Java library: using an `ArrayList` to store the words in sorted order, a `TreeSet`, and a `HashSet`. There's little programming involved, aside from learning to use the Java library.

8.5. I am given a list of ranges of numbers, $[x_i, x_i']$, each with $0 \le x_i < x_i' \le 1$. I want to know all the ranges of values between 0 and 1 that are *not* covered by one of these ranges of numbers. So, if the only input is $[0.25, 0.5]$, then the output would be $[0.0, 0.25]$ and $[0.5, 1.0]$ (never mind the end points). Show how to do this quickly.

Chapter 9

Balanced Searching

We've seen that binary search trees have a weakness: a tendency to be become *unbalanced,* so that they are ineffective in dividing the set of data they represent into two substantially smaller parts. Let's consider what we can do about this.

Of course, we could always rebalance an unbalanced tree by simply laying all the keys out in order and then re-inserting them in such a way as to keep the tree balanced. That operation, however, requires time linear in the number of keys in the tree, and it is difficult to see how to avoid having a $\Theta(N^2)$ factor creep in to the time required to insert N keys. By contrast, only $O(N \lg N)$ time is required to make N insertions if the data happen to be presented in an order that keeps the tree bushy. So let's first look at operations to re-balance a tree (or keep it balanced) without taking it apart and reconstructing it.

9.1   Balance by Construction: B-Trees

Another way to keep a search tree balanced is to be careful always to "insert new keys in a good place" so that the tree remains bushy by construction. The database community has long used a data structure that does exactly this: the *B-tree*[1]. We will describe the data structure and operations abstractly here, rather than give code, since in practice there is a whole raft of devices one uses to gain speed.

A *B-tree of order m* is a positional tree with the following properties:

1. All nodes have m or fewer children.

2. All nodes other than the root have at least $m/2$ children (we can also say that each node other than the root contains at least $\lceil m/2 \rceil$ children[2]).

3. A node with children $C_0, C_1, \ldots, C_{n-1}$ is labeled with keys $K_1, \ldots, K_{n-1}$ (think of key K_i as resting "between" C_{i-1} and C_i), with $K_1 < K_2 < \cdots < K_{n-1}$.

---

[1]D. Knuth's reference: R. Bayer and E. McCreight, *Acta Informatica* (1972), 173–189, and also unpublished independent work by M. Kaufman.

[2]The notation $\lceil x \rceil$ means "the smallest integer that is $\geq x$."

4. A B-tree is a search tree: For any node, all keys in the subtree rooted at C_i are strictly less than K_{i+1}, and (for $i > 0$), strictly greater than K_i.

5. All the empty children occur at the same level of the tree.

Figure 9.1 contains an example of an order-4 tree. In real implementations, B-trees tend to be kept on secondary storage (disks and the like), with their nodes being read in as needed. We choose m so as to make the transfer of data from secondary storage as fast as possible. Disks in particular tend to have minimum transfer times for each read operation, so that for a wide range of values of m, there is little difference in the time required to read in a node. Making m too small in that case is an obviously bad idea.

We'll represent the nodes of a B-tree with a structure we'll call a `BTreeNode`, for which we'll use the following terminology:

B.child(i) Child number i of B-tree node B, where $0 \leq i < m$.

B.key(i) Key number i of B-tree node B, where $1 \leq i < m$.

B.parent () The parent node of B.

B.index () The integer, i, such that

$$B == B.parent ().child (i)$$

B.arity () The number of children in B.

An entire B-tree, then, would consist of a pointer to the root, with perhaps some extra useful information, such as the current size of the B-tree.

Because of properties (2) and (5), a B-tree containing N keys must have $O(\log_{m/2} N)$ levels. Because of property (1), searching a single node's keys takes $O(1)$ time (we assume m is fixed). Therefore, searching a B-tree by the following obvious recursive algorithm is an $O(\log_m N) = O(\lg N)$ operation:

```
boolean search (BTreeNode B, Key X) {
  if (B is the empty tree)
    return false;
  else {
    Find largest c such that B.key(i) ≤ X, for all 1 ≤ i ≤ c.
    if (c > 0 && X.equals (B.key (c)))
      return true;
    else
      return search (B.child (c), K);
  }
```
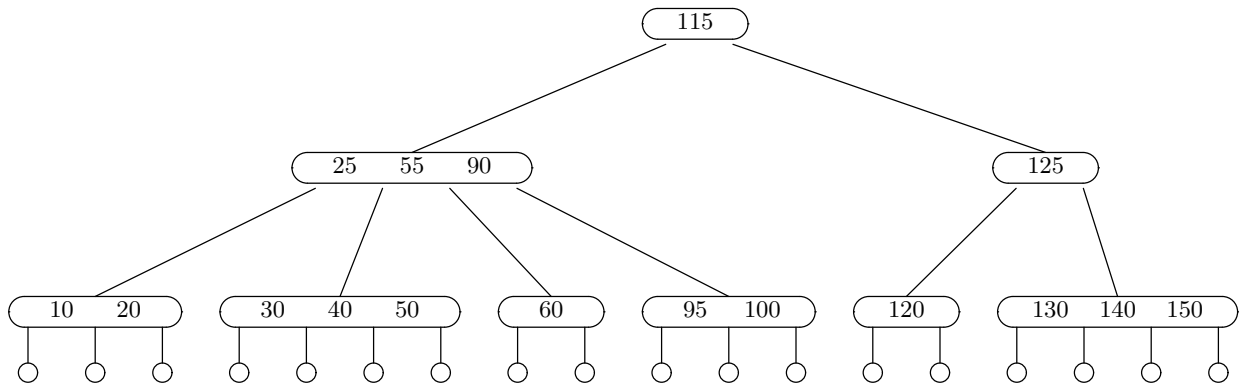
Figure 9.1: Example of a B-tree of order 4 with integer keys. Circles represent empty nodes, which appear all at the same level. Each node has two to four children, and one to three keys. Each key is greater than all keys in the children to its left and less than all keys in the children to its right.

9.1.1 B-tree Insertion

Initially, we insert into the bottom of a B-tree, just as for binary search trees. However, we avoid "scrawny" trees by filling our nodes up and splitting them, rather than extending them down. The idea is simple: we find an appropriate place at the bottom of the tree to insert a given key, and perform the insertion (also adding an additional empty child). If this makes the node too big (so that it has m keys and $m + 1$ (empty) children), we *split* the node, as in the code in Figure 9.2. Figure 9.3 illustrates the process.

9.1.2 B-tree deletion

Deleting from a B-tree is generally more complicated than insertion, but not too bad. As usual, real, production implementations introduce numerous intricacies for speed. To keep things simple, I'll just describe a straightforward, idealized method. Taking our cue from the way that insertion works, we will first move the key to be deleted down to the bottom of the tree (where deletion is straightforward). Then, if deletion has made the original node too small, we merge it with a sibling, pulling down the key that used to separate the two from the parent. The pseudocode in Figure 9.4 describes the process, which is also illustrated in Figure 9.5.

9.2 Tries

Loosely speaking, balanced (maximally bushy) binary search trees containing N keys require $\Theta(\lg N)$ time to find a key. This is not entirely accurate, of course, because it neglects the possibility that the time required to *compare* against a key depends on the key. For example, the time required to compare two strings depends on the length of the shorter string. Therefore, in all the places I've said "$\Theta(\lg N)$" before, I *really* meant "$\Theta(L \lg N)$" for L a bound on the number of bytes in the

```
/** Split B-tree node B, which has from m + 1 to 2m + 1
 *  children. */
void split (BTreeNode B) {
  int k = B.arity () / 2;
  Key X = B.key (k);
  BTreeNode B2 = a new BTree node;
  move B.child (k) through B.child(m) and
      B.key (k+1) through B.key (m) out of B and into B2;
  remove B.key (k) from B;
  if (B was the root) {
    create a new root with children  B  and  B2
        and with key  X;
  } else {
    BtreeNode P = B.parent ();
    int c = B.index ();
    insert child B2 at position c+1 in P, and key
        X at position c+1 in P, moving subsequent
        children and keys of P over as needed;
    if (P.arity () > m)
      split (P);
  }
}
```

Figure 9.2: Splitting a B-tree node. Figure 9.3 contains illustrations.

(a) Insert 15:

```
   10   20                          10   15   20
  ┌──┬──┐         ──────▶         ┌──┬──┬──┐
  ○  ○  ○                        ○  ○  ○  ○
```

(b) Insert 145:

```
        125                                      125   140
       /   \                                    /   |   \
    120    130 140 150        ──────▶         120   130   145 150
    ○ ○    ○ ○ ○ ○                            ○ ○   ○ ○   ○ ○ ○
```

(c) Insert 35:

```
                    115
                   /    \
              25   55   90                         . . .
             /   |   |    \
   10 15 20   30 40 50   60   95 100
   ○ ○ ○ ○   ○ ○ ○ ○   ○ ○   ○ ○ ○
```

```
                    35   115
                   /    |    \
            25         55   90               . . .
           /  \       /   |   \
   10 15 20   30   40 50  60   95 100
   ○ ○ ○ ○   ○ ○   ○ ○ ○  ○ ○   ○ ○ ○
```
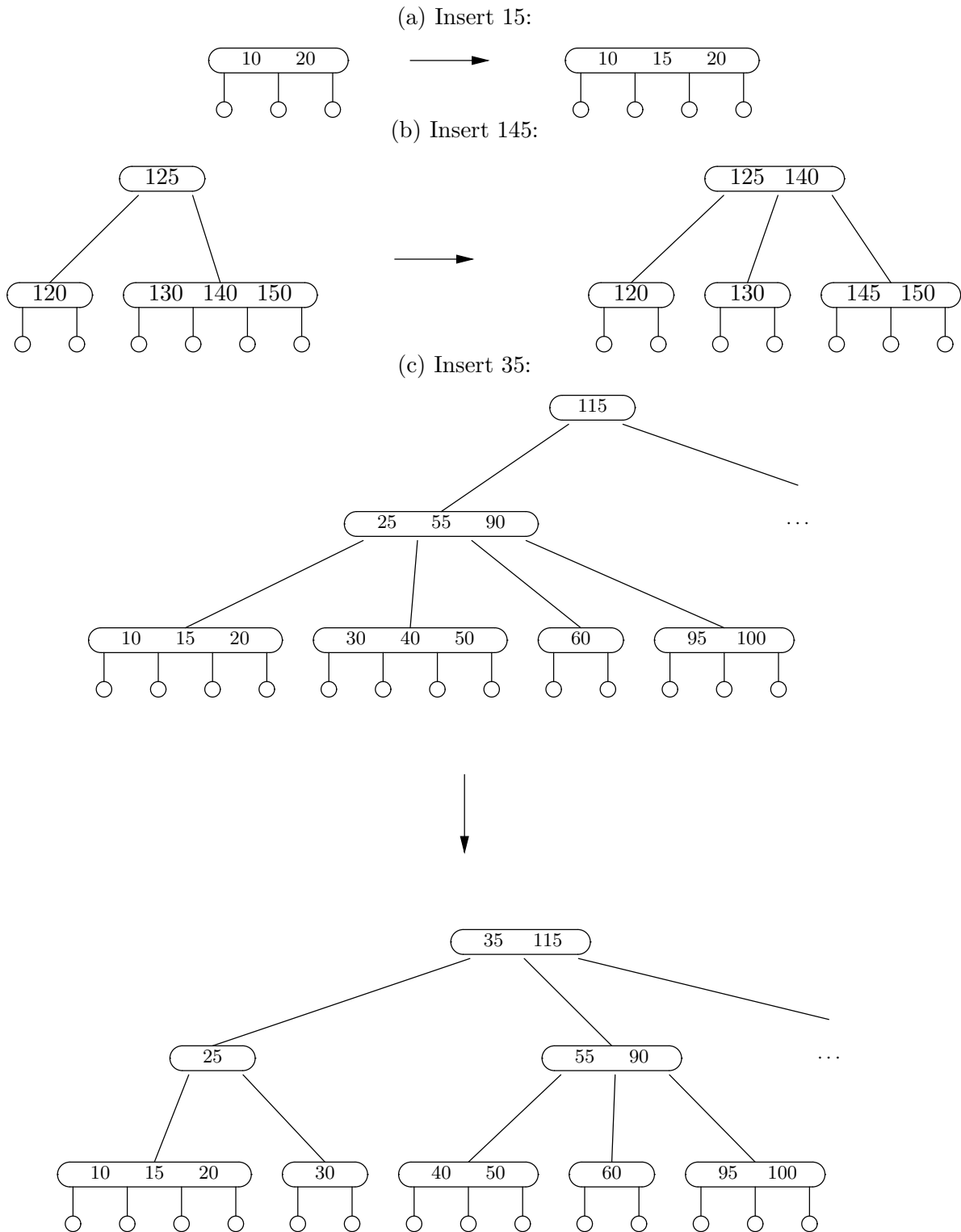
Figure 9.3: Inserting into a B-tree. The examples modify the tree in 9.1 by inserting 15, 145, and then 35.

```
/** Delete B.key(i) from the BTree containing B. */
void deleteKey (BTreeNode B, int i) {
  if (B's children are all empty)
    remove B.key(i), moving over remaining keys;
  else {
    int n = B.child(i-1).arity();
    merge (B, i);
    // The key we want to delete is now #n of child #i-1.
    deleteKey (B.child(i-1), n);
  }
  if (B.arity () > m) // Happens only on recursive calls
    split (B);
  regroup (B);
}


/** Move B.key(i) and the contents of B.child(i) into B.child(i-1),
 *  after its existing keys and children.  Remove B.key(i) and
 *  B.child(i) from B, moving over the remaining contents.
 *  (Temporarily makes B.child(i-1) too large. The former
 *  B.child(i) becomes garbage). */
void merge (BTreeNode B, int i) { implementation straightforward }


/** If B has too few children, regroup the B-tree to re-establish
 *  the B-tree conditions. */
void regroup (BTreeNode B) {
  if (B is the root && B.arity () == 1)
    make B.child(0) the new root;
  else if (B is not the root && B.arity () < m/2) {
    if (B.index () == 0)
      merge (B.parent (), 1);
    else
      merge (B.parent (), B.index ());
    regroup (B.parent ());
  }
}
```

Figure 9.4: Deleting from a B-tree node. See Figure 9.5 for an illustration.

(a) Steps in removing 25:

(b) Removing 10:
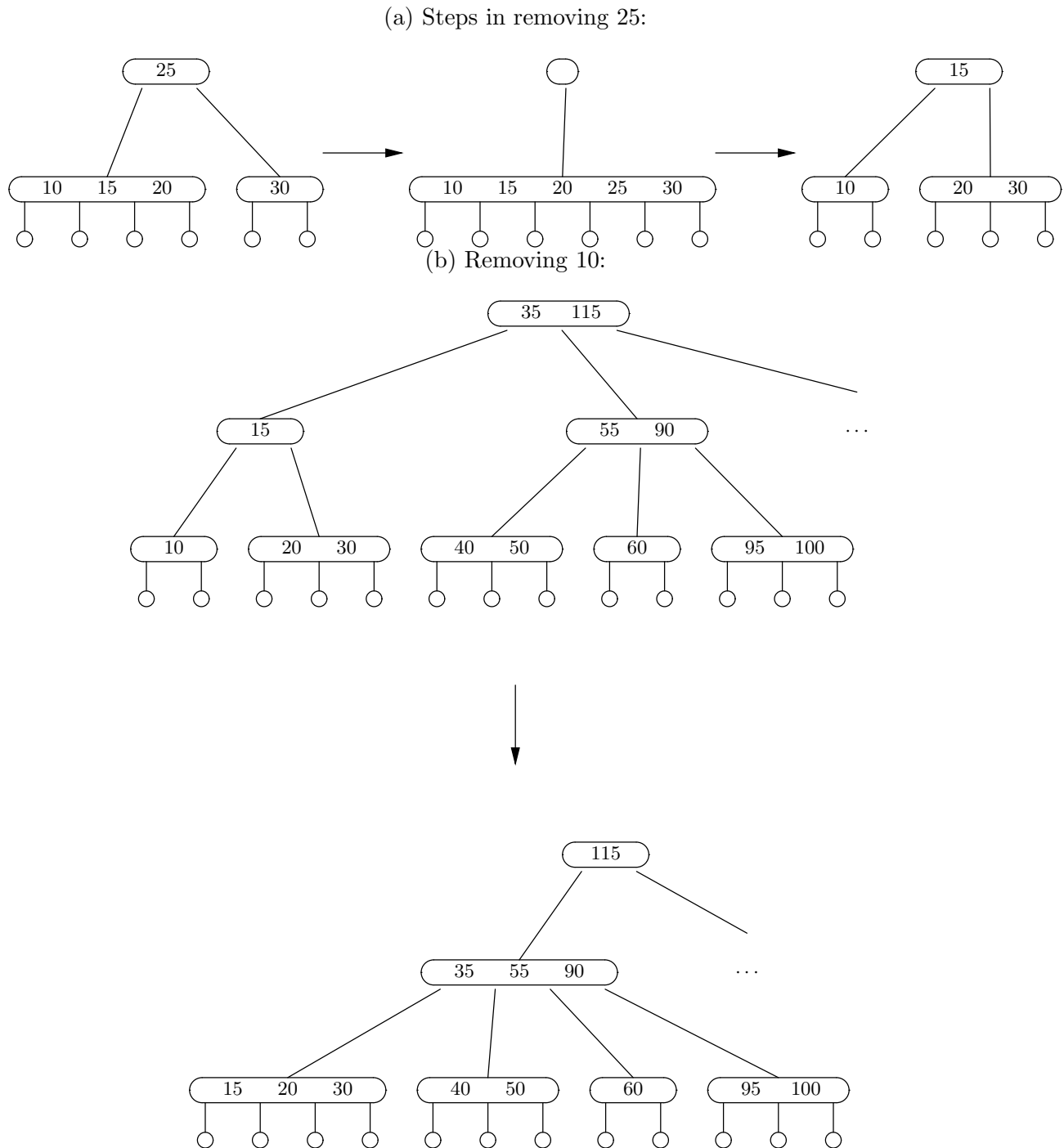
Figure 9.5: Deletion from a B-tree. The examples start from the final tree in Figure 9.3c. In (a), we remove 25. First, merge to move 25 to the bottom. Then remove it and split resulting node (which is too big), at 15. Next, (b) shows deletion of 10 from the tree produced by (a). Deleting 10 from its node at the bottom makes that node too small, so we merge it, moving 15 down from the parent. That in turn makes the parent too small, so we merge it, moving 35 down from the root, giving the final tree.

key. In most applications, this doesn't much matter, since L tends to increase very slowly, if at all, as N increases. Nevertheless, we come to an interesting question: we evidently can't get rid of the factor of L too easily (after all, you have to look at the key you're searching for), but can we get rid of the factor of $\lg N$?

9.2.1   Tries: basic properties and algorithms

It turns out that we can avoid the $\lg N$ factor, using a data structure known as a *trie*[3]. A pure trie is a kind of tree that represents a set of strings from some alphabet of fixed size, say $A = \{a_0, \ldots, a_{M-1}\}$. One of the characters is a special delimiter that appears only at the ends of words, '□'. For example, A might be the set of printable ASCII characters, with □ represented by an unprintable character, such as '\000' (NUL). A trie, T, may be abstractly defined by the following recursive definition[4]: A trie, T, is either

- empty, or

- a leaf node containing a string, or

- an internal node containing M children that are also tries. The edges leading to these children are labeled by the characters of the alphabet, a_i, like this: $C_{a_0}, C_{a_1}, \ldots C_{a_{M-1}}$.

We can think of a trie as a tree whose leaf nodes are strings. We impose one other condition:

- If by starting at the root of a trie and following edges labeled $s_0, s_1, \ldots, s_{h-1}$, we reach a string, then that string begins $s_0 s_1 \cdots s_{h-1}$.

Therefore, you can think of every internal node of a trie as standing for some *prefix* of all the strings in the leaves below it: specifically, an internal node at level k stands for the first k characters of each string below it.

A string $S = s_0 s_1 \cdots s_{m-1}$ is in T if by starting at the root of T and following 0 or more edges with labeled $s_0 \cdots s_j$, we arrive at the string S. We will pretend that all strings in T end in □, which appears only as the last character of a string.

---

[3]How is it pronounced? I have no idea. The word was suggested by E. Fredkin in 1960, who derived it from the word "re*trie*val. Despite this etymology, I usually pronounce it like "try" to avoid verbal confusion with "tree."

[4]This version of the trie data structure is described in D. E. Knuth, *The Art of Programming*, vol. 3, which is *the* standard reference on sorting and searching. The original data structure, proposed in 1959 by de la Briandais, was slightly different.
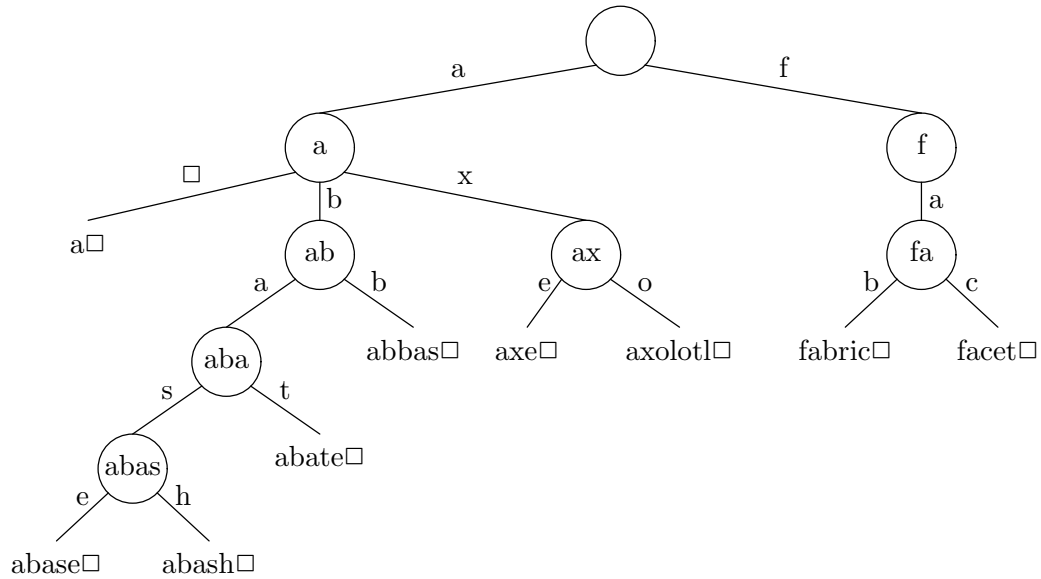
Figure 9.6: A trie containing the set of strings {a, abase, abash, abate, abbas, axe, axolotl, fabric, facet}. The internal nodes are labeled to show the string prefixes to which they correspond.
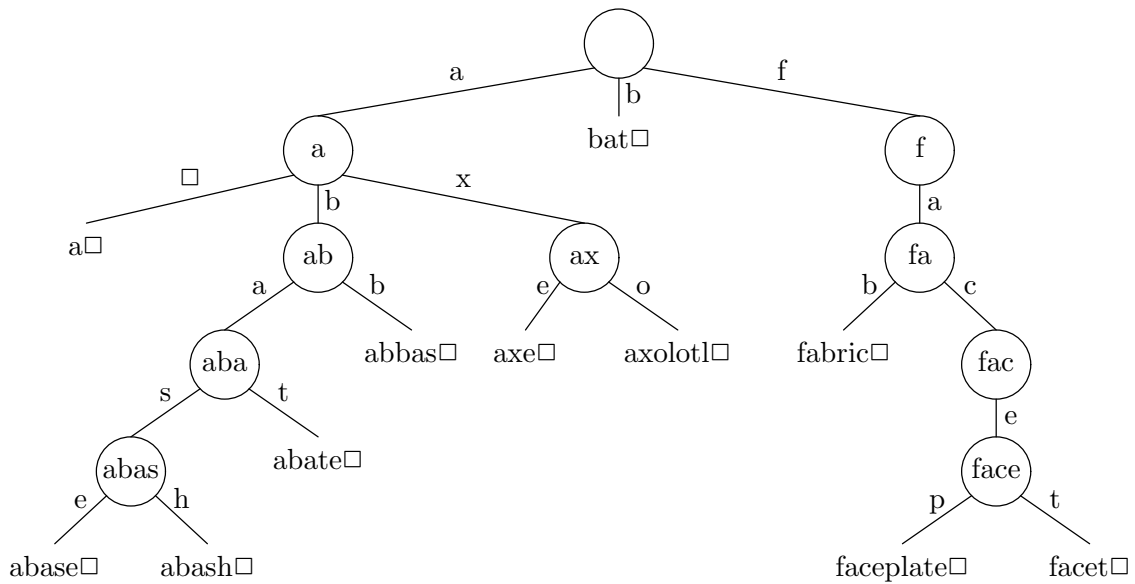
Figure 9.7: Result of inserting the strings "bat" and "faceplate" into the trie in Figure 9.6.

Figure 9.6 shows a trie that represents a small set of strings. To see if a string is in the set, we start at the root of the trie and follow the edges (links to children) marked with the successive characters in the string we are looking for (including the imaginary □ at the end). If we succeed in finding a string somewhere along this path and it equals the string we are searching for, then the string we are searching for is in the trie. If we don't, it is not in the trie. For each word, we need internal nodes only as far down as there are multiple words stored that start with the characters traversed to that point. The convention of ending everything with a special character allows us to distinguish between a situation in which the trie contains two words, one of which is a prefix of the other (like "a" and "abate"), from the situation where the trie contains only one long word.

From a trie user's point of view, it looks like a kind of tree with String labels:

```java
public abstract class Trie {
   /** The empty Trie. */
   public static final Trie EMPTY = new EmptyTrie();

   /** The label at this node.  Defined only on leaves. */
   abstract public String label();

   /** True is X is in this Trie. */
   public boolean isIn(String x) ...

   /** The result of inserting X into this Trie, if it is not
    *  already there, and returning this.  This trie is
    *  unchanged if X is in it already. */
   public Trie insert(String x) ...

   /** The result of removing X from this Trie, if it is present.
    *  The trie is unchanged if X is not present. */
   public Trie remove(String x) ...

   /** True if this Trie is a leaf (containing a single String). */
   abstract public boolean isLeaf();

   /** True if this Trie is empty */
   abstract public boolean isEmpty();

   /** The child numbered with character K. Requires that this node
    *  not be empty.  Child 0 corresponds to □. */
   abstract public Trie child(int k);

   /** Set the child numbered with character K to C.  Requires that
    *  this node not be empty.  (Intended only for internal use. */
   abstract protected void setChild(int k, Trie C);
}
```

The following algorithm describes a search through a trie.

```
/** True is X is in this Trie. */
public boolean isIn(String x) {
   Trie P = longestPrefix(x, 0);
   return P.isLeaf() && x.equals(P.label());
}

/** The node representing the longest prefix of X.substring(K) that
 *  matches a String in this trie. */
private Trie longestPrefix(String x, int k) {
   if (isEmpty() || isLeaf())
      return this;
   int c = nth(x, k);
   if (child(c).isEmpty())
      return this;
   else
      return child(c).longestPrefix(x, k+1);
}

/** Character K of X, or □ if K is off the end of X. */
static char nth(String x, int k) {
   if (k >= x.length())
      return (char) 0;
   else
      return x.charAt(k);
}
```

It should be clear from following this procedure that the time required to find a key is proportional to the length of the key. In fact, the number of levels of the trie that need to be traversed can be considerably less than the length of the key, especially when there are few keys stored. However, if a string is in the trie, you will have to look at all its characters, so `isIn` has a worst-case time of $\Theta(x.\texttt{length})$.

To insert a key X in a trie, we again find the longest prefix of X in the trie, which corresponds to some node P. Then, if P is a leaf node, we insert enough internal nodes to distinguish X from $P.\texttt{label()}$. Otherwise, we can insert a leaf for X in the appropriate child of P. Figure 9.7 illustrates the results of adding "bat" and "faceplate" to the trie in Figure 9.6. Adding "bat" simply requires adding a leaf to an existing node. Adding "faceplate" requires inserting two new nodes first.

The method `insert` below performs the trie insertion.

```
/** The result of inserting X into this Trie, if it is not
 *  already there, and returning this.  This trie is
 *  unchanged if X is in it already. */
public Trie insert(String X)
```

```
    {
        return insert(X, 0);
    }

    /** Assumes this is a level L node in some Trie.  Returns the  */
     *  result of inserting X into this Trie.  Has no effect (returns
     *  this) if X is already in this Trie. */
    private Trie insert(String X, int L)
    {
        if (isEmpty())
            return new LeafTrie(X);
        int c = nth(X, L);
        if (isLeaf()) {
            if (X.equals(label()))
                return this;
            else if (c == label().charAt(L))
                return new InnerTrie(c, insert(X, L+1));
            else {
                Trie newNode = new InnerTrie(c, new LeafTrie(X));
                newNode.child(label().charAt(L), this);
                return newNode;
            }
        } else {
            child(c, child(c).insert(X, L+1));
            return this;
        }
    }
```

Here, the constructor for InnerTrie(c, T), described later, gives us a Trie for which
child(c) is T and all other children are empty.

Deleting from a trie just reverses this process. Whenever a trie node is reduced
to containing a single leaf, it may be replaced by that leaf. The following program
indicates the process.

```
    public Trie remove(String x)
    {
        return remove(x, 0);
    }

    /** Remove x from this Trie, which is assumed to be level L, and
     *  return the result. */
    private Trie remove(String x, int L)
    {
        if (isEmpty())
            return this;
        if (isLeaf(T)) {
```

```
          if (x.equals(label()))
              return EMPTY;
          else
              return this;
      }
      int c = nth(x, L);
      child(c, child(c).remove(x, L+1));
      int d = onlyMember();
      if (d >= 0)
          return child(d);
      return this;
  }


  /** If this Trie contains a single string, which is in
   *  child(K), return K.  Otherwise returns -1.
  private int onlyMember() {  /* Left to the reader. */ }
```

9.2.2   Tries: Representation

We are left with the question of how to represent these tries. The main problem of course is that the nodes contain a variable number of children. If the number of children in each node is small, a linked tree representation like those described in §5.2 will work. However, for fast access, it is traditional to use an array to hold the children of a node, indexed by the characters that label the edges.

This leads to something like the following:

```
class EmptyTrie extends Trie {
   public boolean isEmpty() { return true; }
   public boolean isLeaf() { return false; }
   public String label() { throw new Error(...); }
   public Trie child(int c) { throw new Error(...); }
   protected void child(int c, Trie T) { throw new Error(...); }
}

class LeafTrie extends Trie {
   private String L;

   /** A Trie containing just the string S. */
   LeafTrie(String s) { L = s; }

   public boolean isEmpty() { return false; }
   public boolean isLeaf() { return true; }
   public String label() { return L; }
   public Trie child(int c) { return EMPTY; }
   protected void child(int c, Trie T) { throw new Error(...); }
}
```

```
class InnerTrie extends Trie {
    // ALPHABETSIZE has to be defined somewhere */
    private Trie[] kids = new kids[ALPHABETSIZE];

    /** A Trie with child(K) == T and all other children empty. */
    InnerTrie(int k, Trie T) {
        for (int i = 0; i < kids.length; i += 1)
            kids[i] = EMPTY;
        child(k, T);
    }

    public boolean isEmpty() { return false; }
    public boolean isLeaf() { return false; }
    public String label() { throw new Error(...); }
    public Trie child(int c) { return kids[c]; }
    protected void child(int c, Trie T) { kids[c] = T; }
}
```

9.2.3   Table compression

Actually, our alphabet is likely to have "holes" in it—stretches of encodings that
don't correspond to any character that will appear in the Strings we insert. We
could cut down on the size of the inner nodes (the `kids` arrays) by performing a
preliminary mapping of `chars` into a compressed encoding. For example, if the
only characters in our strings are the digits 0–9, then we could re-do `InnerTrie` as
follows:

```
class InnerTrie extends Trie {
    private static char[] charMap = new char['9'+1];

    static {
        charMap[0] = 0;
        charMap['0'] = 1; charMap['1'] = 1; ...
    }

    public Trie child(int c) { return kids[charMap[c]]; }
    protected void child(int c, Trie T) { kids[charMap[c]] = T; }
}
```

This helps, but even so, arrays that may be indexed by all characters valid in a
key are likely to be relatively large (for a tree node)—say on the order of $M = 60$
bytes even for nodes that can contain only digits (assuming 4 bytes per pointer, 4
bytes overhead for every object, 4 bytes for a length field in the array). If there is a
total of N characters in all keys, then the space needed is bounded by about $NM/2$.
The bound is reached only in a highly pathological case (where the trie contains only

two very long strings that are identical except in their last characters). Nevertheless, the arrays that arise in tries can be quite *sparse*.

One approach to solving this is to *compress* the tables. This is especially applicable when there are few insertions once some initial set of strings is accommodated. By the way, the techniques described below are generally applicable to any such sparse array, not just tries.

The basic idea is that sparse arrays (i.e., those that mostly contain empty or "null" entries) can be *overlaid* on top of each other by making sure that the non-null entries in one fall on top of null entries in the others. We allocate all the arrays in a single large one, and store extra information with each entry so that we can tell which of the overlaid arrays that entry belongs to. Figure 9.8 shows an appropriate alternative data structure.

The idea is that when we store everybody's array of kids in one place, and store an edge label that tells us what character is supposed to correspond to each kid. That allows us to distinguish between a slot that contains somebody else's child (which means that I have no child for that character), and a slot that contains one of my children. We arrange that the `me` field for every node is unique by making sure that the 0th child (corresponding to □) is always full.

As an example, Figure 9.9 shows the ten internal nodes of the trie in Figure 9.7 overlaid on top of each other. As the figure illustrates, this representation can be very compact. The number of extra empty entries that are needed on the right (to prevent indexing off the end of the array) is limited to $M - 1$, so that it becomes negligible when the array is large enough. (Aside: When dealing with a set of arrays that one wishes to compress in this way, it is best to allocate the fullest (least sparse) first.)

Such close packing comes at a price: insertions are expensive. When one adds a new child to an existing node, the necessary slot may already be used by some other array, making it necessary to move the node to a new location by (in effect) first erasing its non-null entries from the packed storage area, finding another spot for it and moving its entries there, and finally updating the pointer to the node being moved in its parent. There are ways to mitigate this, but we won't go into them here.

9.3 Restoring Balance by Rotation

Another approach is to find an operation that changes the balance of a BST—choosing a new root that moves keys from a deep side to a shallow side—while preserving the binary search tree property. The simplest such operations are the *rotations* of a tree. Figure 9.10 shows two BSTs holding identical sets of keys. Consider the rightRotation first (the left is a mirror image). First, the rotation preserves the binary search tree property. In the unrotated tree, the nodes in A are

```
abstract class Trie {
   ...
   static protected Trie[] allKids;
   static protected char[] edgeLabels;
   static final char NOEDGE = /* Some char that isn't used. */
   static {
      allKids = new Trie[INITIAL_SPACE];
      edgeLabels = new char[INITIAL_SPACE];
      for (int i = 0; i < INITIAL_SPACE; i += 1) {
         allKids[i] = EMPTY; edgeLabels[i] = NOEDGE;
      }
   }
   ...
}

class InnerTrie extends Trie {
   /* Position of my child 0 in allKids.  My kth child, if
    * non-empty, is at allKids[me + k].  If my kth child is
    * not empty, then edgeLabels[me+k] == k.  edgeLabels[me]
    * is always 0 (□). */
   private int me;


   /** A Trie with child(K) == T and all other children empty. */
   InnerTrie(int k, Trie T) {
      // Set me such that edgeLabels[me + k].isEmpty(). */
      child(0, EMPTY);
      child(k, T);
   }

   public Trie child(int c) {
      if (edgeLabels[me + c] == c)
         return allKids[me + c];
      else
         return EMPTY;
   }

   protected void child(int c, Trie T) {
      if (edgeLabels[me + c] != NOEDGE &&
          edgeLabels[me + c] != c) {
        // Move my kids to a new location, and point me at it.
      }
      allKids[me + c] = T;
      edgeLabels[me + c] = c;
   }
}
```

Figure 9.8: Data structures used with compressed Tries.
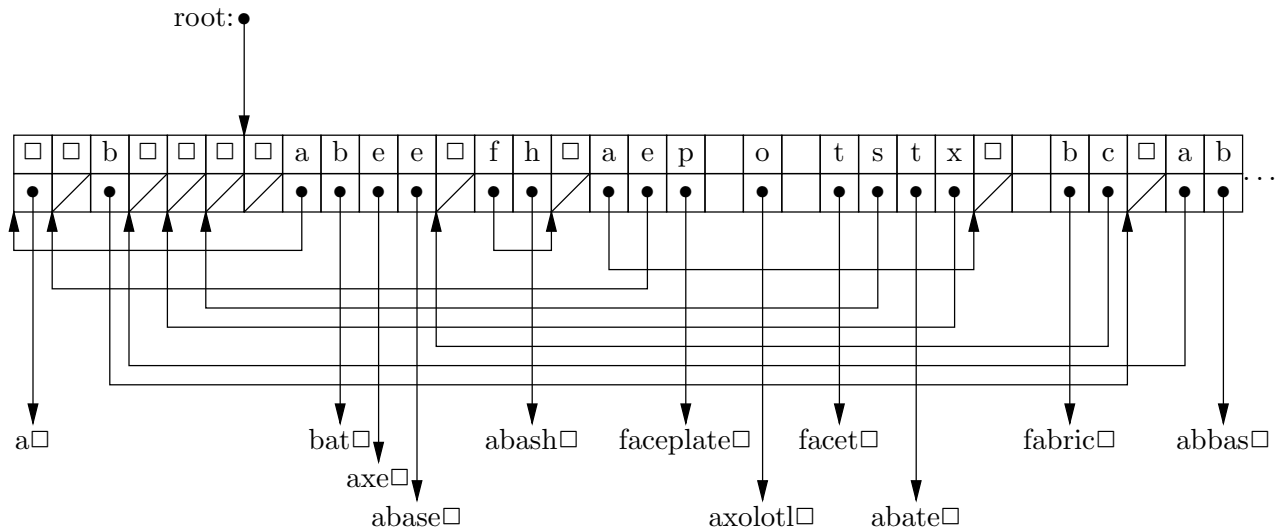
Figure 9.9: A packed version of the trie from Figure 9.7. Each of the trie nodes from that figure is represented as an array of children indexed by character, the character that is the index of a child is stored in the upper row (which corresponds to the array `edgeLabels`). The pointer to the child itself is in the lower row (which corresponds to the `allKids` array). Empty boxes on top indicate unused locations (the `NOEDGE` value). To compress the diagram, I've changed the character set encoding so that □ is 0, 'a' is 1, 'b' is 2, etc. The crossed boxes in the lower row indicate empty nodes. There must also be an additional 24 empty entries on the right (not shown) to account for the c–z entries of the rightmost trie node stored. The search algorithm uses `edgeLabels` to determine when an entry actually belongs to the node it is currently examining. For example, the root node is supposed to contain entries for 'a', 'b', and 'f'. And indeed, if you count 1, 2, and 6 over from the "root" box above, you'll find entries whose edge labels are 'a', 'b', and 'f'. If, on the other hand, you count over 3 from the root box, looking for the non-existent 'c' edge, you find instead an edge label of 'e', telling you that the root node has no 'c' edge.

the exactly the ones less than B, as they are on the right; D is greater, as on the right; and subtree C is greater, as on the right. You can also assure yourself that the nodes under D in the rotated tree bear the proper relation to it.

Turning to height, let's use the notation H_A, H_C, H_E, H_B, and H_D to denote the heights of subtrees A, C, and E and of the subtrees whose roots are nodes B and D. Any of A, C, or E can be empty; we'll take their heights in that case to be -1. The height of the tree on the left is $1 + \max(H_E, 1 + H_A, 1 + H_C)$. The height of the tree on the right is $1 + \max(H_A, 1 + H_C, 1 + H_E)$. Therefore, as long as $H_A > \max(H_C + 1, H_E)$ (as would happen in a left-leaning tree, for example), the height of the right-hand tree will be less than that of the left-hand tree. One gets a similar situation in the other direction.

In fact, it is possible to convert any BST into any other that contains the same keys by means of rotations. This amounts to showing that by rotation, we can move any node of a BST to the root of the tree while preserving the binary search tree
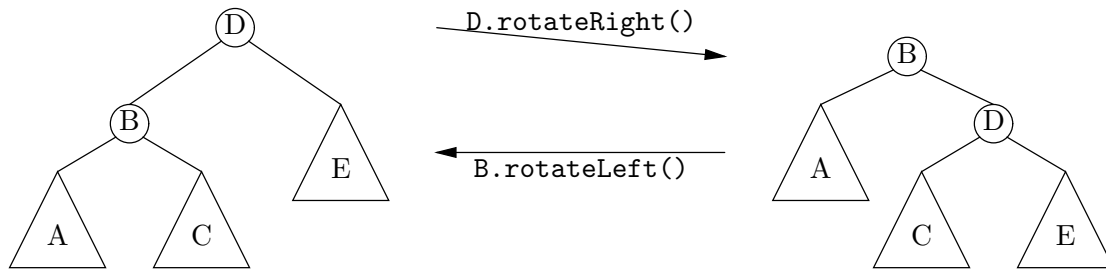
Figure 9.10: Rotations in a binary search tree. Triangles represent subtrees and circles represent individual nodes. The binary search tree relation is maintained by both operations, but the levels of various nodes are affected.

property [why is this sufficient?]. The argument is an induction on the structure of trees.

- It is clearly possible for empty or one-element trees.

- Suppose we want to show it for a larger tree, assuming (inductively) that all smaller trees can be rotated to bring any of their nodes their root. We proceed as follows:

  - If the node we want to make the root is already there, we're done.

  - If the node we want to make the root is in the left child, rotate the left child to make it the root of the left child (inductive hypothesis). The perform a right rotation on the whole tree.

  - Similarly if the node we want is in the right child.

9.3.1   AVL Trees

Of course, knowing that it is possible to re-arrange a BST by means of rotation doesn't tell us which rotations to perform. The *AVL tree* is an example of a technique for keeping track of the heights of subtrees and performing rotations when they get too far out of line. An AVL tree[5] is simply a BST that satisfies the

> **AVL Property:** the heights of the left and right subtrees of every node differ by at most one.

Adding or deleting a node at the bottom of such a tree (as happens with the simple BST insertion and deletion algorithms from §6.1) may invalidate the AVL property, but it may be restored by working up toward the root from the point of the insertion or deletion and performing certain selected rotations, depending on the nature of the imbalance that needs to be corrected. In the following diagrams, the expressions in the subtrees indicate their heights. An unbalanced subtree having the form

---

[5]The name is taken from the names of the two discoverers, Adel'son-Vel'skiĭ and Landis.
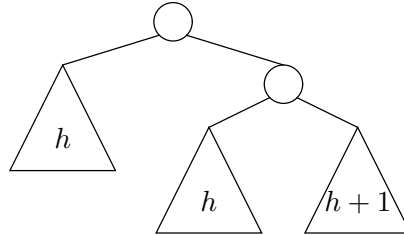
can be rebalanced with a single left rotation, giving an AVL tree of the form:
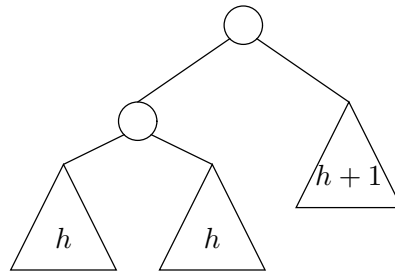
Finally, consider an unbalanced tree of the form
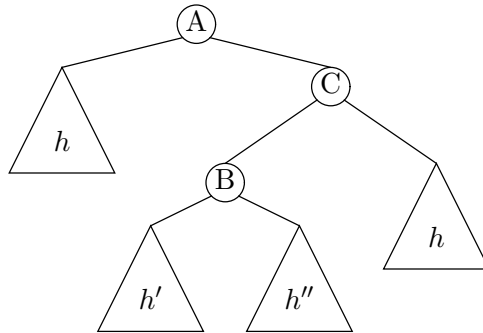
where at least one of h' and h'' is h and the other is either h or $h-1$. Here, we can rebalance by performing two rotations, first a right rotation on C, and then a left rotation on A, giving the correct AVL tree
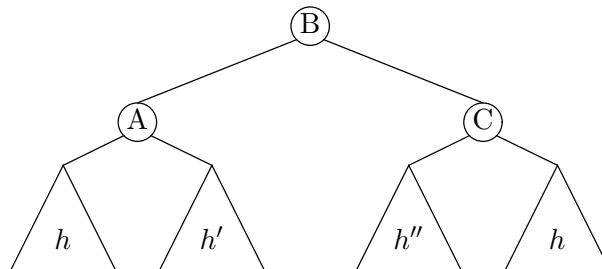
The other possible cases of imbalance that result from adding or removing a single node are mirror images of these.

Thus, if we keep track of the heights of all subtrees, we can always restore the AVL property, starting from the point of insertion or deletion at the base of the tree and proceeding upwards. In fact, it turns out that it isn't necessary to know the precise heights of all subtrees, but merely to keep track of the three cases at each node: the two subtrees have the same height, the height of the left subtree is

greater by 1, and the height of the right subtree is greater by 1.

9.4   Skip Lists

The B-tree was an example of a search tree in which nodes had variable numbers
of children, with each child representing some ordered set of keys. It speeds up
searches as does a vanilla binary search tree by subdividing the keys at each node
into disjoint ranges of keys, and contrives to keep these sequences of comparable
length, balancing the tree. Here we look at another structure that does much the
same thing, except that it uses rotation as needed to approximately balance the tree
and it merely achieves this balance with high probability, rather than with certainty.
Consider the same set of integer keys from Figure 9.1, arranged into a search tree
where each node has one key and any number of children, and the children of any
node all have keys that are at least as large as that of their parent. Figure 9.11
shows a possible arrangement. The maximum heights at which the keys appear are
chosen independently according to a rule that gives a probability of $(1 - p)p^k$ of a
key appearing being at height k (0 being the bottom). That is, $0 < p < 1$ is an
arbitrary constant that represents the approximate proportion of all nodes at height
$\geq e$ that have height $> e$. We add a minimal $(-\infty)$ key at the left with sufficient
height to serve as a root for the whole tree.

Figure 9.11 shows an example, created using $p = 0.5$. To look for a key, we
can scan this tree from left to right starting at any level and working downwards.
Starting at the bottom (level 0) just gives us a simple linear search. At higher levels,
we search a forest of trees, choosing which forest to examine more closely on the
basis of the value of its root node. To see if 127 is a member, for example, we can
look at

- the first 15 entries of level 0 (not including $-\infty$) [15 entries]; or

- the first 7 level-1 entries, and then the 2 level-0 items below the key 120 [9
  entries]; or

- the first 3 level-2 entries, then the level-1 entry 140, and then the 2 level-0
  items below 120 [6 entries]; or

- the level-3 entry 90, then the level-2 entry 120, then the level-1 entry 140, and
  then the 2 level-0 items below 120 [5 entries].

We can represent this tree as a kind of linear list of nodes in in-order (see
Figure 9.12) in which the nodes have random numbers of `next` links, and the ith
`next` link in each (numbering from 0 as usual) is connected to the next node that
has at least $i + 1$ links. This list-like representation, with some links "skipping"
arbitrary numbers of list elements, explains the name given to this data structure:
the *skip list*[6].

---

[6]William Pugh, Skip lists: A probabilistic alternative to balanced trees, "*Comm. of the ACM,*
33, 6 (June, 1990) pp. 668–676.
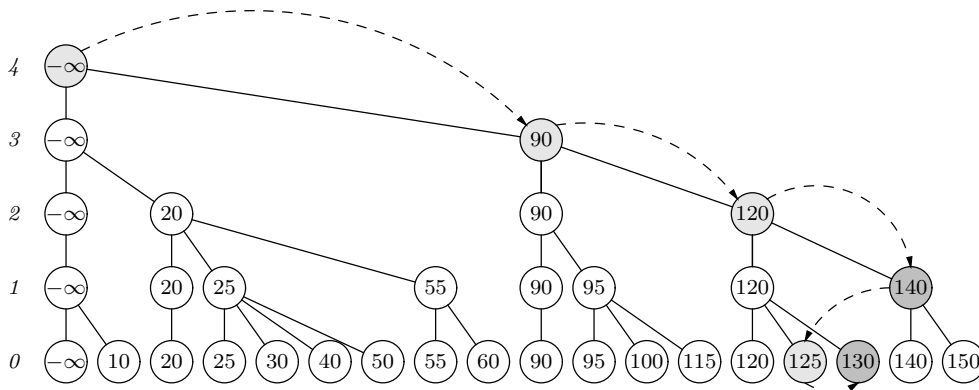
Figure 9.11: An abstract view of a skip list, showing its relationship to a (non-binary) search tree. Each key other than $-\infty$ is duplicated to a random height. We can search this structure beginning at any level. In the best case, to search (unsuccessfully) for the target value 127, we need only look at the keys in the shaded nodes. Darker shaded nodes indicate keys larger than 127 that bound the search.
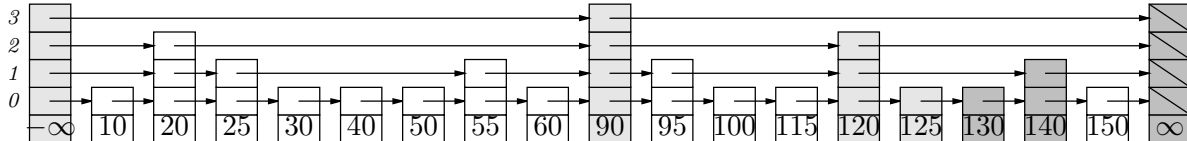
Figure 9.12: The skip list from Figure 9.11, showing a possible representation. The data structure is an ordered list whose nodes contain random numbers of pointers to later nodes (which allow intervening items in the list to be skipped during a search; hence the name). If a node has at least k pointers, then it contains a pointer to the next node that has at least k pointers. A node for ∞ at the right allows us to avoid tests for null. Again, the nodes looked at during a search for 127 are shaded; the darker shading indicates nodes that limit the search.
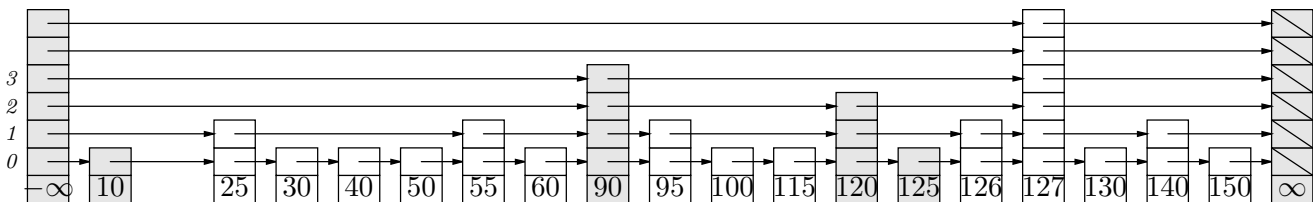
Figure 9.13: The skip list from Figure 9.12 after inserting 127 and 126 (in either order), and deleting 20. Here, the 127 node is randomly given a height of 5, and the 126 node a height of 1. The shaded nodes show which previously existing nodes need to change. For the two insertions, the nodes needing change are the same as the light-shaded nodes that were examined to search for 127 (or 126), plus the $\pm\infty$ nodes at the ends (if they need to be heightened).

Searching is very simple. If we denote the value at one of these nodes as `L.value` (here, we'll use integer keys) and the next pointer at height k as `L.next[k]`, then:

```
/** True iff X is in the skip list beginning at node L at
 *  a height <= K, where K>=0. */
static boolean contains (SkipListNode L, int k, int x) {
  if (x == L.next[k].value)
    return true;
  else if (x > L.next[k].value)
    return contains (L.next[k], k, x);
  else if (k > 0)
    return contains (L, k-1, x);
  else
    return false;
}
```

We can start the at any level $k \geq 0$ up to the height of the tree. It turns out that a reasonable place to start for a list containing N nodes is at level $\log_{1/p} N$, as explained below.

To insert or delete into the list, we find the position of the node to be added or deleted by the process above, keeping track of the nodes we traverse to do so. When the item is added or deleted, these are the nodes whose pointers may need to be updated. When we insert nodes, we choose a height for them randomly in such a way that the number of nodes at height $k + 1$ is roughly pk, where p is some probability (typical values for which might be 0.5 or 0.25). That is, if we are shooting for a roughly n-ary search tree, we let $p = 1/n$. A suitable procedure might look like this:

```
/** A random integer, h, in the range 0 .. MAX such that
 *    Pr(h >= k) = P^k,  0 <= k <= MAX. */
static int randomHeight (double p, int max, Random r) {
  int h;
  h = 0;
  while (h < max && r.nextDouble () < p)
    h += 1;
  return h;
}
```

In general, it is pointless to accommodate arbitrarily large heights, so we impose some maximum, generally the logarithm (base $1/p$) of the maximum number of keys one expects to need.

Intuitively, any sequence of M inserted nodes each of whose heights is at least k will be randomly broken about every $1/p$ nodes by one whose height is strictly greater than k. Likewise, for nodes of height at least $k + 1$, and so forth. So, if our list contains N items, and we start looking at level $\log_{1/p} N$, we'd expect to look at most at roughly $(1/p)\log_{1/p} N$ keys (that is, $1/p$ keys at each of $\log_{1/p} N$ levels). In other words, $\Theta(\lg N)$ on average, which is what we want. Admittedly,

this analysis is a bit handwavy, but the true bound is not significantly larger. Since inserting and deleting consists of finding the node, plus some insertion or deletion time proportional to the node's height, we actually have $\Theta(\lg N)$ expected bounds on search, insertion, and deletion.

Exercises

9.1. Fill in the following to agree with its comments:

```
/** Return a modified version of T containing the same nodes
 *  with the same inorder traversal, but with the node containing
 *  label X at the root.  Does not create any new Tree nodes. */
static Tree rotateUp (Tree T, Object X) {
  // FILL IN
}
```

9.2. What is the maximum height of an order-5 B-tree containing N nodes? What is the minimum height? What sequences of keys give the maximum height (that is, give a general characterization of such sequences). What sequences of keys give the minimum height?

9.3. Write a non-recursive version of the `contains` function for skip lists (§9.4).

9.4. Define an implementation of the `SortedSet` interface that uses a skip list representation.

Chapter 10

Concurrency and Synchronization

An implicit assumption in everything we've done so far is that a single program is modifying our data structures. In Java, one *can* have the effect of multiple programs modifying an object, due to the existence of *threads.*

Although the language used to describe threads suggests that their purpose is to allow several things to happen simultaneously, this is a somewhat misleading impression. Even the smallest Java application running on Sun's JDK platform, for example, has five threads, and that's only if the application has not created any itself, and even if the machine on which the program runs consists of a single processor (which can only execute one instruction at a time). The four additional "system threads" perform a number of tasks (such as "finalizing" objects that are no longer reachable by the program) that are *logically independent* of the rest of the program. Their actions could usefully occur at any time relative to the rest of the program. Sun's Java runtime system, in other words, is using threads as a *organizational tool* for its system. Threads abound in Java programs that use *graphical user interfaces (GUIs).* One thread draws or redraws the screen. Another responds to *events* such as the clicking of a mouse button at some point on the screen. These are related, but largely independent activities: objects must be redrawn, for example, whenever a window becomes invisible and uncovers them, which happens independently of any calculations the program is doing.

Threads violate our implicit assumption that a single program operates on our data, so that even an otherwise perfectly implemented data structure, with all of its instance variables private, can become corrupted in rather bizarre ways. The existence of multiple threads operating on the same data objects also raises the general problem of how these threads are to communicate with each other in an orderly fashion.

10.1   Synchronized Data Structures

Consider the `ArrayList` implementation from §4.1. In the method `ensureCapacity`, we find

```
public void ensureCapacity (int N) {  public Object set (int k, Object x) {
  if (N <= data.length)                 check (k, count);
    return;                             Object old = data[k];
  Object[] newData = new Object[N];     data[k] = x;
  System.arraycopy (data, 0,            return old;
                  newData, 0, count);};
  data = newData;
}
```

Suppose one program executes `ensureCapacity` while another is executing `set` on the same `ArrayList` object. We could see the following interleaving of their actions:

```
/* Program 1 executes: */ newData = new Object[N];
/* Program 1 executes: */ System.arraycopy (data, 0,
                                             newData, 0, count);
/* Program 2 executes: */ data[k] = x;
/* Program 1 executes: */ data = newData;
```

Thus, we lose the value that Program 2 set, because it puts this value into the old value of `data` after `data`'s contents have been copied to the new, expanded array.

To solve the simple problem presented by `ArrayList`, threads can arrange to access any particular `ArrayList` in *mutual exclusion*—that is, in such a way that only one thread at a time operates on the object. Java's **synchronized** statement provide mutual exclusion, allowing us to produce *synchronized* (or *thread-safe*) data structures. Here is part of an example, showing both the use of the **synchronized** method modifier and equivalent use of the **synchronized** statement:

```
public class SyncArrayList<T> extends ArrayList<T> {
  ...
  public void ensureCapacity (int n) {
    synchronized (this) {
      super.ensureCapacity (n);
    }
  }

  public synchronized T set (int k, T x) {
    return super.set (k, x);
  }
```

The process of providing such wrapper functions for all methods of a List is sufficiently tedious that the standard Java library class `java.util.Collections` provides the following method:

```
/** A synchronized (thread-safe) view of the list L, in which only
 *  one thread at a time executes any method.  To be effective,
 *  (a) there should be no subsequent direct use of L,
 *  and (b) the returned List must be synchronized upon
 *  during any iteration, as in
 *
 *       List aList = Collections.synchronizedList(new ArrayList());
 *          ...
 *       synchronized(aList) {
 *         for (Iterator i = aList.iterator(); i.hasNext(); )
 *            foo(i.next());
 *       }
 */
 public static List<T> synchronizedList (List L<T>) { ... }
```

Unfortunately, there is a time cost associated with synchronizing on every operation, which is why the Java library designers decided that `Collection` and most of its subtypes would not be synchronized. On the other hand, `StringBuffers` and `Vectors` are synchronized, and cannot be corrupted by simultaneous use.

10.2 Monitors and Orderly Communication

The objects returned by the `synchronizedList` method are examples of the simplest kind of *monitor.* This term refers to an object (or type of object) that controls ("monitors") concurrent access to some data structure so as to make it work correctly. One function of a monitor is to provide mutually exclusive access to the operations of the data structure, where needed. Another is to arrange for *synchronization* between threads—so that one thread can wait until an object is "ready" to provide it with some service.

Monitors are exemplified by one of the classic examples: the *shared buffer* or *mailbox.* A simple version of its public specification looks like this:

```
/** A container for a single message (an arbitrary Object).  At any
 *  time, a SmallMailbox is either empty (containing no message) or
 *  full (containing one message). */
public class SmallMailbox {
   /** When THIS is empty, set its current message to MESSAGE, making
    *  it full. */
   public synchronized void deposit (Object message)
      throws InterruptedException { ... }
   /** When THIS is full, empty it and return its current message. */
   public synchronized Object receive ()
      throws InterruptedException { ... }
}
```

Since the specifications suggest that either method might have to wait for a new message to be deposited or an old one to be received, we specify both as possibly

throwing an `InterruptedException`, which is the standard Java way to indicate that while we were waiting, some other thread interrupted us.

The `SmallMailbox` specification illustrates the features of a typical monitor:

- None of the modifiable state variables (i.e., fields) are exposed.

- Accesses from separate threads that make any reference to modifiable state are mutually excluded; only one thread at a time *holds a lock* on a `SmallMailbox` object.

- A thread may relinquish a lock temporarily and await notification of some change. But changes in the ownership of a lock occur only at well-defined points in the program.

The internal representation is simple:

```
private Object message;
private boolean amFull;
```

The implementations make use of the primitive Java features for "waiting until notified:"

```
public synchronized void deposit (Object message)
  throws InterruptedException
{
  while (amFull)
    wait ();  // Same as this.wait ();
  this.message = message; this.amFull = true;
  notifyAll (); // Same as this.notifyAll ()
}

public synchronized Object receive ()
  throws InterruptedException
{
  while (! amFull)
    wait ();
  amFull = false;
  notifyAll ();
  return message;
}
```

The methods of `SmallMailbox` allow other threads in only at carefully controlled points: the calls to `wait`. For example, the loop in `deposit` means "If there is still old unreceived mail, wait until some other thread to receives it and wakes me up again (with `notifyAll`) *and* I have managed to lock this mailbox again." From the point of view of a thread that is executing `deposit` or `receive`, each call to `wait` has the effect of causing some change to the instance variables of **this**—some change, that is, that could be effected by other calls `deposit` or `receive`.

As long as the threads of a program are careful to protect all their data in monitors in this fashion, they will avoid the sorts of bizarre interaction described at the beginning of §10.1. Of course, there is no such thing as a free lunch; the use of locking can lead to the situation known as *deadlock* in which two or more threads wait for each other indefinitely, as in this artificial example:

```
class Communicate {
  static SimpleMailbox
    box1 = new SimpleMailbox (),
    box2 = new SimpleMailbox ();
}
```

```
// Thread #1:                    |   // Thread #2:
m1 = Communicate.box1.receive ();|  m2 = Communicate.box2.receive ();
Communicate.box2.deposit (msg1); |  Communicate.box1.deposit (msg2);
```

Since neither thread sends anything before trying to receive a message from its box, both threads wait for each other (the problem could be solved by having one of the two threads reverse the order in which it receives and deposits).

10.3 Message Passing

Monitors provide a disciplined way for multiple threads to access data without stumbling over each other. Lurking behind the concept of monitor is a simple idea:

> Thinking about multiple programs executing simultaneously is hard, so don't do it! Instead, write a bunch of *one-thread* programs, and have them exchange data with each other.

In the case of general monitors, "exchanging data" means setting variables that each can see. If we take the idea further, we can instead define "exchanging data" as "reading input and writing output." We get a concurrent programming discipline called *message passing.*

In the message-passing world, threads are independent sequential programs than send each other *messages.* They read and write messages using methods that correspond to `read` on Java `Readers`, or `print` on Java `PrintStreams`. As a result, one thread is affected by another only when it bothers to "read its messages."

We can get the effect of message passing by writing our threads to perform all interaction with each other by means of mailboxes. That is, the threads share some set of mailboxes, but share no other modifiable objects or variables (unmodifiable objects, like `Strings`, are fine to share).

Exercises

10.1. Give a possible implementation for the `Collections.synchronizedList` static method in §10.1.

Chapter 11

Pseudo-Random Sequences

Random sequences of numbers have a number of uses in simulation, game playing, cryptography, and efficient algorithm development. The term "random" is rather difficult to define. For most of our purposes, we really don't need to answer the deep philosophical questions, since our needs are generally served by sequences that display certain statistical properties. This is a good thing, because truly "random" sequences in the sense of "unpredictable" are difficult to obtain quickly, and programmers generally resort, therefore, to *pseudo-random* sequences. These are generated by some formula, and are therefore predictable in principle. Nevertheless, for many purposes, such sequences are acceptable, if they have the desired statistics.

We commonly use sequences of integers or floating-point numbers that are *uniformly* distributed throughout some interval—that is, if one picks a number (truly) at random out of the sequence, the probability that it is in any set of numbers from the interval is proportional to the size of that set. It is relatively easy to arrange that a sequence of integers in some interval has this particular property: simply enumerate a permutation of the integers in that interval over and over. Each integer is enumerated once per repetition, and so the sequence is uniformly distributed. Of course, having described it like this, it becomes even more apparent that the sequence is anything but "random" in the informal sense of this term. Nevertheless, when the interval of integers is large enough, and the permutation "jumbled" enough, it is hard to tell the difference. The rest of this Chapter will deal with generating sequences of this sort.

11.1   Linear congruential generators

Perhaps the most common pseudo-random-number generators use the following recurrence.

$$X_n = (aX_{n-1} + c) \bmod m, \tag{11.1}$$

where $X_n \geq 0$ is the nth integer in the sequence, and $a, m > 0$ and $c \geq 0$ are integers. The *seed* value, X_0, may be any value such that $0 \leq X_0 < m$. When m is

a power of two, the X_n are particularly easy to compute, as in the following Java class.

```java
/** A generator of pseudo-random numbers in the range 0 .. 2^31−1. */
class Random1 {

    private int randomState;

    static final int
        a = ...,
        c = ...;

    Random1(int seed) { randomState = seed; }

    int nextInt() {
        randomState = (a * randomState + c) & 0x7fffffff;
        return randomState;
    }

}
```

Here, m is 2^{31}. The '&' operation computes $\mathrm{mod}2^{31}$ [why?]. The result can be any non-negative integer. If we change the calculation of `randomState` to

```java
randomState = a * randomState + c;
```

then the computation is implicitly done modulo 2^{32}, and the results are integers in the range -2^{31} to $2^{31} - 1$.

The question to ask now is how to choose a and c appropriately. Considerable analysis has been devoted to this question[1]. Here, I'll just summarize. I will restrict the discussion to the common case of $m = 2^w$, where $w > 2$ is typically the word size of the machine (as in the Java code above). The following criteria for a and c are desirable.

1. In order to get a sequence that has maximum *period*—that is, which cycles through all integers between 0 and $m-1$ (or in our case, $-m/2$ to $m/2-1$)—it is necessary and sufficient that c and m be relatively prime (have no common factors other than 1), and that a have the form $4k + 1$ for some integer k.

2. A very low value of a is easily seen to be undesirable (the resulting sequence will show a sort of sawtooth behavior). It is desirable for a to be reasonably large relative to m (Knuth, for example, suggests a value between $0.01m$ and $0.99m$) and have no "obvious pattern" to its binary digits.

3. It turns out that values of a that display low *potency* (defined as the minimal value of s such that $(a - 1)^s$ is divisible by m) are not good. Since $a - 1$ must

---

[1] For details, see D. E. Knuth, *Seminumerical Algorithms* (*The Art of Computer Programming*, volume 2), second edition, Addison-Wesley, 1981.

be divisible by 4, (see item 1 above), the best we can do is to insure that $(a-1)/4$ is not even—that is, $a \bmod 8 = 5$.

4. Under the conditions above, $c = 1$ is a suitable value.

5. Finally, although most arbitrarily-chosen values of a satisfying the above conditions work reasonably well, it is generally preferable to apply various statistical tests (see Knuth) just to make sure.

For example, when $m = 2^{32}$, some good choices for a are 1566083941 (which Knuth credits to Waterman) and 1664525 (credited to Lavaux and Janssens).

There are also bad choices of parameters, of which the most famous is one that was part of the IBM FORTRAN library for some time—`RANDU`, which had $m = 2^{31}$, X_0 odd, $c = 0$, and $a = 65539$. This does not have maximum period, of course (it skips all even numbers). Moreover, if you take the numbers three at a time and consider them as points in space, the set of points is restricted to a relatively few widely-spaced planes—strikingly bad behavior.

The Java library has a class `java.util.Random` similar to `Random1`. It takes $m = 2^{48}$, $a = 25214903917$, and $c = 11$ to generate `long` quantities in the range 0 to $2^{48}-1$, which doesn't quite satisfy Knuth's criterion 2. I haven't checked to see how good it is. There are two ways to initialize a `Random`: either with a specific "seed" value, or with the current value of the system timer (which on UNIX systems gives a number of milliseconds since some time in 1970)—a fairly common way to get an unpredictable starting value. It's important to have both: for games or encryption, unpredictability is useful. The first constructor, however, is also important because it makes it possible to reproduce results.

11.2   Additive Generators

One can get very long periods, and avoid multiplications (which can be a little expensive for Java `long` quantities) by computing each successive output, X_n, as a sum of selected of previous outputs: X_{n-k} for several fixed values of k. Here's an instance of this scheme that apparently works quite well[2]:

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \text{ for } n \geq 55 \tag{11.2}$$

where $m = 2^e$ for some e. We initially choose some "random" seed values for X_0 to X_{54}. This has a large period of $2^f(2^{55} - 1)$ for some $0 \leq f < e$. That is, although numbers it produces must repeat before then (since there are only 2^e of them, and e is typically something like 32), they won't repeat in the same pattern.

Implementing this scheme gives us another nice opportunity to illustrate the *circular buffer* (see §4.5). Keep your eye on the array `state` in the following:

---

[2]Knuth credits this to unpublished work of G. J. Mitchell and D. P. Moore in 1958.

```
class Random2 {
   /** state[k] will hold Xₖ, Xₖ₊₅₅, Xₖ₊₁₁₀,... */
   private int[] state = new int[55];
   /** nm will hold n mod 55 after each call to nextInt.
    * Initially n = 55. */
   private int nm;

   public Random2(...) {
       initialize state[0..54] to values for X₀ to X₅₄;
       nm = -1;
   }

   public int nextInt() {
      nm = mod55(nm + 1);
      int k24 = mod55(nm - 24);
      // Now state[nm] is Xₙ₋₅₅ and state[k24] is Xₙ₋₂₄.
      return state[nm] += state[k24];
      // Now state[nm] (just returned) represents Xₙ.
   }

   private int mod55 (int x) {
       return (x >= 55) ? x - 55 : (x < 0) ? x + 55 : x;
   }
}
```

Other values than 24 and 55 will also produce pseudo-random streams with good characteristics. See Knuth.

11.3   Other distributions

11.3.1   Changing the range

The linear congruential generators above give us pseudo-random numbers in some fixed range. Typically, we are really interested in some other, smaller, range of numbers instead. Let's first consider the case where we want a sequence, Y_i, of integers uniformly distributed in a range 0 to $m' - 1$ and are given pseudo-random integers, X_i, in the range 0 to $m - 1$, with $m > m'$. A possible transformation is

$$Y_i = \lfloor \frac{m'}{m} X_i \rfloor,$$

which results in numbers that are reasonably evenly distributed as long as $m \gg m'$. From this, it is easy to get a sequence of pseudo-random integers evenly distributed in the range $L \leq Y_i' < U$:

$$Y_i' = \lfloor \frac{U - L}{m} X_i \rfloor.$$

It might seem that

$$Y_i = X_i \bmod m' \tag{11.3}$$

is a more obvious formula for Y_i. However, it has problems when m' is a small power of two and we are using a linear congruential generator as in Equation 11.1, with m a power of 2. For such a generator, the last k bits of X_i have a period of 2^k [why?], and thus so will Y_i. Equation 11.3 works much better if m' is not a power of 2.

The `nextInt` method in the class `java.util.Random` produces its 32-bit result from a 48-bit state by dividing by 2^{16} (shifting right by 16 binary places), which gets converted to an `int` in the range -2^{31} to $2^{31} - 1$. The `nextLong` method produces a 64-bit result by calling `nextInt` twice:

```
(nextInt() << 32L) + nextInt();
```

11.3.2   Non-uniform distributions

So far, we have discussed only uniform distributions. Sometimes that isn't what we want. In general, assume that we want to pick a number Y in some range u_l to u_h so that[3]

$$\Pr[Y \le y] = P(y),$$

where P is the desired *distribution function*—that is, it is a non-decreasing function with $P(y) = 0$ for $y < u_l$ and $P(y) = 1$ for $y \ge u_h$. The idea of what we must do is illustrated in Figure 11.1, which shows a graph of a distribution P. The key observation is that the desired probability of Y being no greater than y_0, $P(y_0)$, is the same as the probability that a uniformly distributed random number X on the interval 0 to 1, is less than $P(y_0)$. Suppose, therefore, that we had an inverse function P^{-1} so that $P(P^{-1}(x)) = x$. Then,

$$\Pr[P^{-1}(X) \le y] = \Pr[X \le P(y)] = P(y)$$

In other words, we can define

$$Y = P^{-1}(X)$$

as the desired random variable.

All of this is straightforward when P is strictly increasing. However, we have to exercise care when P is not invertible, which happens when P does not strictly increase (i.e., it has "plateaus" where its value does not change). If $P(y)$ has a constant value between y_0 and y_1, this means that the probability that Y falls between these two values is 0. Therefore, we can uniquely define $P^{-1}(x)$ as the *smallest y* such that $P(y) \le x$.

Unfortunately, inverting a continuous distribution (that is, in which Y ranges—at least ideally—over some interval of real numbers) is not always easy to do. There are various tricks; as usual, the interested reader is referred to Knuth for details. In particular, Java uses one of his algorithms (the *polar method* of Box, Muller, and

---

[3]The notation $Pr[E]$ means "the probability that situation E (called an *event*) is true."
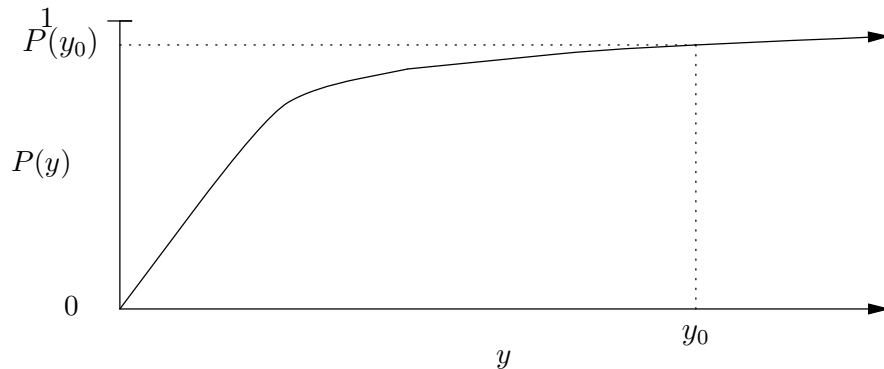
Figure 11.1: A typical non-uniform distribution, illustrating how to convert a uniformly distributed random variable into one governed by an arbitrary distribution, $P(y)$. The probability that y is less than y_0 is the same as the probability that a uniformly distributed random variable on the interval 0 to 1 is less than or equal to $P(y_0)$.

Marsaglia) to implement the `nextGaussian` method in `java.util.Random`, which returns normally distributed values (i.e., the "bell curve" density) with a mean value of 0 and standard deviation of 1.

11.3.3   Finite distributions

There is a simpler common case: that in which Y is to range over a finite set— say the integers from 0 to u, inclusive. We are typically given the probabilities $p_i = \Pr[Y = i]$. In the interesting case, the distribution is not uniform, and hence the p_i are not necessarily all $1/(u+1)$. The relationship between these p_i and $P(i)$ is

$$P(i) = \Pr[Y \leq i] = \sum_{0 \leq k \leq i} p_k.$$

The obvious technique for computing the inverse P^{-1} is to perform a lookup on a table representing the distribution P. To compute a random i satisfying the desired conditions, we choose a random X in the range 0–1, and return the first i such that $X \leq P(i)$. This works because we return i iff $P(i-1) < X \leq P(i)$ (taking $P(-1) = 0$). The distance between $P(i-1)$ and $P(i)$ is p_i, and since X is uniformly distributed across 0 to 1, the probability of getting a point in this interval is equal to the size of the interval, p_i.

For example, if $1/12$ of the time we want to return 0, $1/2$ the time we want to return 1, $1/3$ of the time we want to return 2, and $1/12$ of the time we want to return 3, we return the index of the first element of table PT that does not exceed a random X chosen uniformly on the interval 0 to 1, where PT is defined to have $PT[0] = 1/12$, $PT[1] = 7/12$, $PT[2] = 11/12$, and $PT[3] = 1$.

Oddly enough, there is a faster way of doing this computation for large u, discovered by A. J. Walker[4]. Imagine the numbers between 0 and u as labels on $u+1$

---

[4]Knuth's citations are *Electronics Letters* **10**, 8 (1974), 127–128 and *ACM Transactions on*

beakers, each of which can contain $1/(u + 1)$ units of liquid. Imagine further that we have $u + 1$ vials of colored liquids, also numbered 0 to u, each of a different color and all immiscible in each other; we'll use the integer i as the name of the color in vial number i. The total amount of liquid in all the vials is 1 unit, but the vials may contain different amounts. These amounts correspond to the desired probabilities of picking the numbers 0 through $u + 1$.

Suppose that we can distribute the liquid from the vials to the beakers so that

- Beaker number i contains two colors of liquid (the quantity of one of the colors, however, may be 0), and

- One of the colors of liquid in beaker i is color number i.

Then we can pick a number from 0 to u with the desired probabilities by the following procedure.

- Pick a random floating-point number, X, uniformly in the range $0 \leq X < u+1$. Let K be the integer part of this number and F the fractional part, so that $K + F = X$, $F < 1$, and $K, F \geq 0$.

- If the amount of liquid of color K in beaker K is greater than or equal to F, then return K. Otherwise return the number of the other color in beaker K.

A little thought should convince you that the probability of picking color i under this scheme is proportional to the amount of liquid of color i. The number K represents a randomly-chosen beaker, and F represents a randomly-chosen point along the side of that beaker. We choose the color we find at this randomly chosen point. We can represent this selection process with two tables indexed by K: Y_K is the color of the other liquid in beaker K (i.e., besides color K itself), and H_K is the height of the liquid with color K in beaker K (as a fraction of the distance to the top gradation of the beaker).

For example, consider the probabilities given previously; an appropriate distribution of liquid is illustrated in Figure 11.2. The tables corresponding to this figure are $Y = [1, 2, -, 1]$ (Y_2 doesn't matter in this case), and $H = [0.3333, 0.6667, 1.0, 0.3333]$.

The only remaining problem is perform the distribution of liquids to beakers, for which the following procedure suffices (in outline):

Mathematical Software, **3** (1976), 253–256.
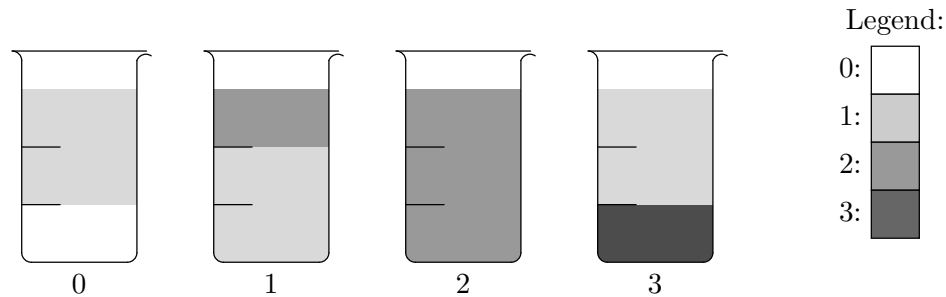
Figure 11.2: An example dividing probabilities (colored liquids) into beakers. Each beaker holds 1/4 unit of liquid. There is 1/12 unit of 0-colored liquid, 1/2 unit of 1-colored liquid, 1/3 unit of 2-colored liquid, and 1/12 unit of 3-colored liquid.

```
/** S is a set of integers that are the names of beakers and
 *  vial colors.  Assumes that all the beakers named in S are
 *  empty and have equal capacity, and the total contents of the vials
 *  named in S is equal to the total capacity of the beakers in
 *  S.  Fills the beakers in S from the vials in V so that
 *  each beaker contains liquid from no more than two vials and the
 *  beaker named s contains liquid of color s. */
void fillBeakers(SetOfIntegers S)
{
    if (S is empty)
        return;

    v0 = the color of a vial in S with the least liquid;
    Pour the contents of vial v0 into beaker v0;
    /* The contents must fit in the beaker, because since v0
     * contains the least fluid, it must have no more than the
     * capacity of a single beaker.  Vial v0 is now empty. */

    v1 = the color of a vial in S with the most liquid;
    Fill beaker v0 the rest of the way from vial v1;
    /* If |S| = 1 so that v0 = v1, this is the null operation.
     * Otherwise, v0 ≠ v1 and vial v1 must contain at
     * least as much liquid as each beaker can contain.  Thus, beaker
     * v0 is filled by this step. (NOTE: |S| is the
     * cardinality of S.) */

    fillBeakers(S − {v0});
}
```

The action of "pouring the contents of vial v_0 into beaker v_0" corresponds to setting H_{v_0} to the ratio between the amount of liquid in vial v_0 and the capacity of beaker v_0. The action of "filling beaker v_0 the rest of the way from vial v_1" corresponds to setting Y_{v_0} to v_1.

11.4 Random permutations and combinations

Given a set of N values, consider the problem of selecting a *random sequence without replacement* of length M from the set. That is, we want a random sequence of M values from among these N, where each value occurs in the sequence no more than once. By "random sequence" we mean that all possible sequences are equally likely[5]. If we assume that the original values are stored in an array, then the following is a very simple way of obtaining such a sequence.

```
/** Permute A so as to randomly select M of its elements,
 *  placing them in A[0] .. A[M-1], using R as a source of
 *  random numbers. */
static void selectRandomSequence(SomeType[] A, int M, Random1 R)
{
    int N = A.length;
    for (int i = 0; i < M; i += 1)
        swap(A, i, R.randInt(i,N-1));
}
```

Here, we assume `swap(V,j,k)` exchanges the values of `V[j]` and `V[k]`.

For example, if `DECK[0]` is A♣, `DECK[1]` is 2♣,..., and `DECK[51]` is K♠, then

```
selectRandomSequence(DECK, 52, new Random());
```

shuffles the deck of cards.

This technique works, but if $M \ll N$, it is not a terribly efficient use of space, at least when the contents of the array `A` is something simple, like the integers between 0 and $N-1$. For that case, we can better use some algorithms due to Floyd (names of types and functions are meant to make them self-explanatory).

---

[5]Here, I'll assume that the original set contains no duplicate values. If it does, then we have to treat the duplicates as if they were all different. In particular, if there are k duplicates of a value in the original set, it may appear up to k times in the selected sequence.

```
/** Returns a random sequence of M distinct integers from 0..N-1,
 *  with all possible sequences equally likely.  Assumes 0<=M<=N. */
static SequenceOfIntegers selectRandomIntegers(int N, int M, Random1 R)
{
    SequenceOfIntegers S = new SequenceOfIntegers();

    for (int i = N-M; i < N; i += 1) {
        int s = R.randInt(0, i);
        if (s ∈ S)
            insert i into S after s;
        else
            prefix s to the front of S;
    }
    return S;
}
```

This procedure produces all possible sequences with equal probability because every possible sequence of values for s generates a distinct value of S, and all such sequences are equally probable.

Sanity check: the number of ways to select a sequence of M objects from a set of N objects is

$$\frac{N!}{(N-M)!}$$

and the number of possible sequences of values for s is equal to the number of possible values of `R.randInt(0,N-M)` times the number of possible values of `R.randInt(0,N-M-1)`, etc., which is

$$(N-M+1)(N-M+2)\cdots N = \frac{N!}{(N-M)!}.$$

By replacing the `SequenceOfIntegers` with a set of integers, and replacing "prefix" and "insert" with simply adding to a set, we get an algorithm for selecting *combinations* of M numbers from the first N integers (i.e., where order doesn't matter).

The Java standard library provides two static methods in the class `java.util.Collections` for randomly permuting an arbitrary `List`:

```
/** Permute L, using R as a source of randomness.  As a result,
 *  calling shuffle twice with values of R that produce identical
 *  sequences will give identical permutations. */
public static void shuffle (List<?> L, Random r) { ··· }
/** Same as shuffle (L, D), where D is a default Random value.  */
public static void shuffle (List L<?>) { ··· }
```

This takes linear time if the list supports fast random access.

Chapter 12

Graphs

When the term is used in computer science, a *graph* is a data structure that represents a mathematical relation. It consists of a set of *vertices* (or *nodes*) and a set of *edges,* which are pairs of vertices[1]. These edge pairs may be unordered, in which case we have an *undirected graph,* or they may be ordered, in which case we have a *directed graph* (or *digraph*) in which each edge *leaves*, *exits*, or is *out of* one vertex and *enters* or is *into* the other. For vertices v and w we denote a general edge between v and w as (v, w), or $\{v, w\}$ if we specifically want to indicate an undirected edge, or $[v, w]$ if we specifically want to indicate a directed edge that leaves v and enters w. An edge (v, w) is said to be *incident* on its two *ends*, v and w; if (v, w) is undirected, we say that v and w are *adjacent* vertices. The *degree* of a vertex is the number of edges incident on it. For a directed graph, the *in-degree* is the number of edges that enter it and the *out-degree* is the number that leave. Usually, the ends of an edge will be distinct; that is, there will be no *reflexive* edges from a vertex to itself.

A *subgraph* of a graph G is simply a graph whose vertices and edges are subsets of the vertices and edges of G.

A *path* of length $k \geq 0$ in a graph from vertex v to vertex v' is a sequence of vertices $v_0, v_1, \ldots, v_{k-1}$ with $v = v_0$, $v' = v_{k-1}$ with all the (v_i, v_{i+1}) edges being in the graph. This definition applies both to directed and undirected graphs; in the case of directed graphs, the path has a direction. The path is *simple* if there are no repetitions of vertices in it. It is a *cycle* if $k > 1$ and $v = v'$, and a *simple cycle* if $v_0, \ldots, v_{k-2}$ are distinct; in an undirected graph, a cycle must additionally not follow the same edge twice. A graph with no cycles is called *acyclic.*

If there is a path from v to v', then v' is said to be *reachable* from v. In an undirected graph, a *connected component* is a set of vertices from the graph and all edges incident on those vertices such that each vertex is reachable from any given vertex, and no other vertex from the graph is reachable from any vertex in the set. An undirected graph is *connected* if it contains exactly one connected component (containing all vertices in the graph).

---

[1]Definitions in this section are taken from Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.
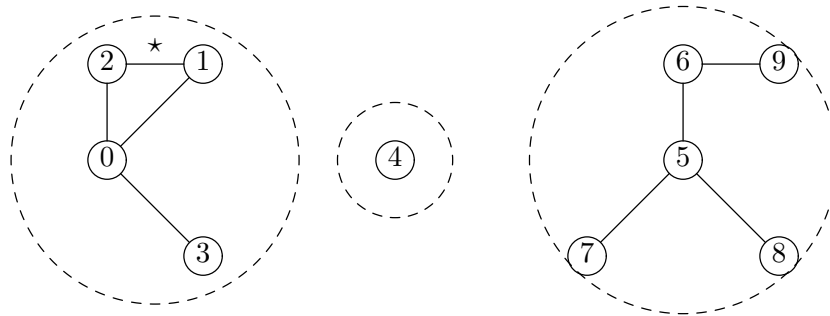
Figure 12.1: An undirected graph. The starred edge is incident on vertices 1 and 2. Vertex 4 has degree 0; 3, 7, 8, and 9 have degree 1; 1, 2 and 6 have degree 2; and 0 and 5 have degree 3. The dashed lines surround the connected components; since there is more than one, the graph is unconnected. The sequence [2,1,0,3] is a path from vertex 2 to vertex 3. The path [2,1,0,2] is a cycle. The only path involving vertex 4 is the 0-length path [4]. The rightmost connected component is acyclic, and is therefore a free tree.

In a directed graph, the connected components contain the same sets of vertices that you would get by replacing all directed edges by undirected ones. A subgraph of a directed graph in which every vertex can be reached from every other is called a *strongly connected component.* Figures 12.1 and 12.2 illustrate these definitions.

A *free tree* is a connected, undirected, acyclic graph (which implies that there is exactly one simple path from any node to any other). An undirected graph is *biconnected* if there are at least two simple paths between any two nodes.

For some applications, we associate information with the edges of a graph. For example, if vertices represent cities and edges represent roads, we might wish to associate distances with the edges. Or if vertices represent pumping stations and edges represent pipelines, we might wish to associate capacities with the edges. We'll call numeric information of this sort *weights.*

12.1 A Programmer's Specification

There isn't an obvious single class specification that one might give for programs dealing with graphs, because variations in what various algorithms need can have a profound effect on the appropriate representations and what operations those representations conveniently support. For instructional use, however, Figure 12.3 gives a sample "one-size-fits-all" abstraction for general directed graphs, and Figure 12.4 does the same for undirected graphs. The idea is that vertices and edges are identified by non-negative integers. Any additional data that one wants to associate with a vertex or edge—such as a more informative label or a weight—can be added "on the side" in the form of additional arrays indexed by vertex or edge number.
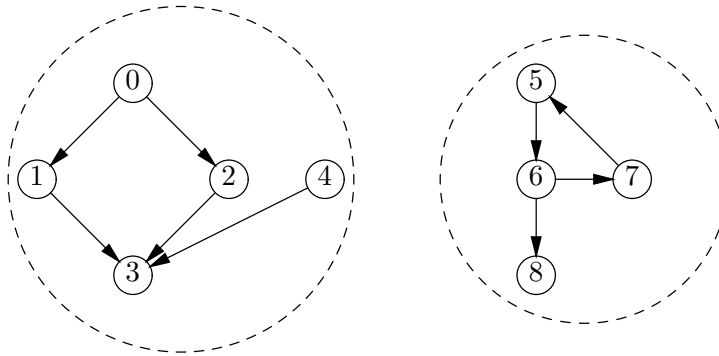
Figure 12.2: A directed graph. The dashed circles show connected components. Nodes 5, 6 and 7 form a strongly connected component. The other strongly connected components are the remaining individual nodes. The left component is acyclic. Nodes 0 and 4 have an in-degree of 0; nodes 1, 2, and 5–8 have an in-degree of 1; and node 3 has an in-degree of 3. Nodes 3 and 8 have out-degrees of 0; 1, 2, 4, 5, and 7 have out-degrees of 1; and 0 and 6 have out-degrees of 2.

12.2   Representing graphs

Graphs have numerous representations, all tailored to the operations that are critical to some application.

12.2.1   Adjacency Lists

If the operations `succ`, `pred`, `leaving`, and `entering` for directed graphs are important to one's problem (or `incident` and `adjacent` for undirected graphs), then it may be convenient to associate a list of predecessors, successors, or neighbors with each vertex—an adjacency list. There are many ways to represent such things—as a linked list, for example. Figure 12.5 shows a method that uses arrays in such a way that to allow a programmer both to sequence easily through the neighbors of a directed graph, and to sequence through the set of all edges. I've included only a couple of indicative operations to show how the data structure works. It is essentially a set of linked list structures implemented with arrays and integers instead of objects containing pointers. Figure 12.6 shows an example of a particular directed graph and the data structures that would represent it.

Another variation on essentially the same structure is to introduce separate types for vertices and edges. Vertices and Edges would then contain fields such as

```java
/** A general directed graph.   For any given concrete extension of this
 *  class, a different subset of the operations listed will work.    For
 *  uniformity, we take all vertices to be numbered with integers
 *  between 0 and N-1.  */
public interface Digraph {
   /** Number of vertices.  Vertices are labeled 0 .. numVertices()-1. */
   int numVertices();

   /** Number of edges.  Edges are numbered 0 .. numEdges()-1. */
   int numEdges();

   /** The vertices that edge E leaves and enters. */
   int leaves(int e);
   int enters(int e);

   /** True iff [v0,v1] is an edge in this graph. */
   boolean isEdge(int v0, int v1);

   /** The out-degree and in-degree of vertex #V. */
   int outDegree(int v);
   int inDegree(int v);

   /** The number of the Kth edge leaving vertex V, 0<=K<outDegree(V). */
   int leaving(int v, int k);
   /** The number of the Kth edge entering vertex V, 0<=K<inDegree(V). */
   int entering(int v, int k);
   /** The Kth successor of vertex V, 0<=K<outDegree(V). It is intended
    *  that succ(v,k) = enters(leaving(v,k)).   */
   int succ(int v, int k);
   /** The Kth predecessor of vertex V, 0<=K<inDegree(V). It is intended
    *  that pred(v,k) = leaves(entering(v,k)).   */
   int pred(int v, int k);

   /** Add M initially unconnected vertices to this graph. */
   void addVertices(int M);

   /** Add an edge from V0 to V1. */
   void addEdge(int v0, int v1);

   /** Remove all edges incident on vertex V from this graph. */
   void removeEdges(int v);
   /** Remove edge (v0, v1) from this graph */
   void removeEdge(int v0, int v1);
}
```

Figure 12.3: A sample abstract directed-graph interface in Java.

```
/** A general undirected graph.   For any given concrete extension of
 *  this class, a different subset of the operations listed will work.
 *  For uniformity, we take all vertices to be numbered with integers
 *  between 0 and N-1.  */
public interface Graph {
    /** Number of vertices.  Vertices are labeled 0 .. numVertices()-1. */
    int numVertices();

    /** Number of edges.  Edges are numbered 0 .. numEdges()-1. */
    int numEdges();

    /** The vertices on which edge E is incident. node0 is the
     *  smaller-numbered vertex. */
    int node0(int e);
    int node1(int e);

    /** True iff vertices V0 and V1 are adjacent. */
    boolean isEdge(int v0, int v1);

    /** The number of edges incident on vertex #V. */
    int degree(int v);

    /** The number of the Kth edge incident on V, 0<=k<degree(V). */
    int incident(int v, int k);
    /** The Kth node adjacent to V, 0<=K<outDegree(V). It is
     *  intended that adjacent(v,k) = either node0(incident(v,k))
     *  or node1(incident(v,k)). */
    int adjacent(int v, int k);

    /** Add M initially unconnected vertices to this graph. */
    void addVertices(int M);

    /** Add an (undirected) edge between V0 and V1. */
    void addEdge(int v0, int v1);
    /** Remove all edges involving vertex V from this graph. */
    void removeEdges(int v);
    /** Remove the (undirected) edge (v0, v1) from this graph. */
    void removeEdge(int v0, int v1);
}
```

Figure 12.4: A sample abstract undirected-graph class.

```
/** A digraph */
public class AdjGraph implements Digraph {

   /** A new Digraph with N unconnected vertices */
   public AdjGraph(int N) {
      numVertices = N; numEdges = 0;
      enters = new int[N*N]; leaves = new int[N*N];
      nextOutEdge = new int[N*N]; nextInEdge = new int[N*N];
      edgeOut0 = new int[N]; edgeIn0 = new int[N];
   }

   /** The vertices that edge E leaves and enters. */
   public int leaves(int e) { return leaves[e]; }
   public int enters(int e) { return enters[e]; }

   /** Add an edge from V0 to V1. */
   public void addEdge(int v0, int v1) {
      if (numEdges >= enters.length)
         expandEdges(); // Expand all edge-indexed arrays
      enters[numEdges] = v1; leaves[numEdges] = v0;
      nextInEdge[numEdges] = edgeIn0[v1];
      edgeIn0[v1] = numEdges;
      nextOutEdge[numEdges] = edgeOut0[v0];
      edgeOut0[v0] = numEdges;
      numEdges += 1;
   }
}
```

Figure 12.5: Adjacency-list implementation for a directed graph. Only a few representative operations are shown.

```
/** The number of the Kth edge leaving vertex V, 0<=K<outDegree(V). */
public int leaving(int v, int k) {
    int e;
    for (e = edgeOut0[v]; k > 0; k -= 1)
        e = nextOutEdge[e];
    return e;
}

...


/* Private section */

private int numVertices, numEdges;
/* The following are indexed by edge number */
private int[]
    enters, leaves,
    nextOutEdge,      /* The # of sibling outgoing edge, or -1 */
    nextInEdge;       /* The # of sibling incoming edge, or -1 */

/* edgeOut0[v] is # of first edge leaving v, or -1.  */
private int[] edgeOut0;
/* edgeIn0[v] is # of first edge entering v, or -1.  */
private int[] edgeIn0;
}
```

Figure 12.5, continued.

edgeOut0	1	3	5	11	−1	8	9	12
edgeIn0	−1	12	10	9	7	3	5	11
	A	B	C	D	E	F	G	H

nextOutEdge	−1	0	−1	2	−1	−1	−1	6	−1	−1	4	7	10
nextInEdge	−1	−1	1	−1	−1	4	−1	−1	2	8	6	−1	0
enters	1	3	3	5	6	6	2	4	3	3	2	7	1
leaves	0	0	1	1	7	2	3	3	5	6	7	3	7
	0	1	2	3	4	5	6	7	8	9	10	11	12
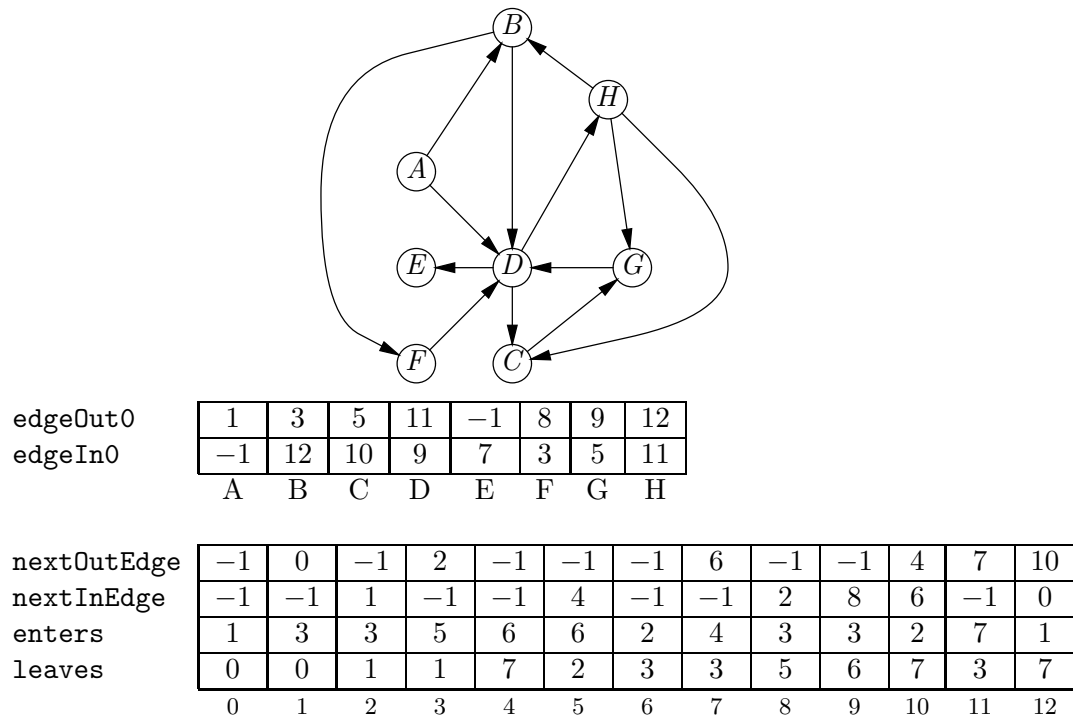
Figure 12.6: A graph and one form of adjacency list representation. The lists in this case are arrays. The lower four arrays are indexed by edge number, and the first two by vertex number. The array `nextOutEdge` forms linked lists of out-going edges for each vertex, with roots in `edgeOut0`. Likewise, `nextInEdge` and `edgeIn0` form linked lists of incoming edges for each vertex. The `enters` and `leaves` arrays give the incident vertices for each edge.

```
class Vertex {
   private int num;  /* Number of this vertex */
   private Edge edgeOut0, edgeIn0; /* First outgoing & incoming edges. */
   ...
}

class Edge {
   private int num;  /* Number of this edge */
   private Vertex enters, leaves;
   private Edge nextOutEdge, nextInEdge;
}
```

12.2.2   Edge sets

If all we need to do is enumerate the edges and tell what nodes they are incident on, we can simplify the representation in §12.2.1 quite a bit by throwing out fields `edgeOut0`, `edgeIn0`, `nextOutEdge`, and `nextInEdge`. We will see one algorithm where this is useful.

$$M \;=\; \begin{array}{c|cccccccc}
 & A & B & C & D & E & F & G & H \\
\hline
A & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
B & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
C & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
D & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
E & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
F & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
G & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
H & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0
\end{array}$$

$$M' \;=\; \begin{array}{c|cccccccc}
 & A & B & C & D & E & F & G & H \\
\hline
A & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
B & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
C & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
D & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
E & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
F & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
G & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
H & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0
\end{array}$$
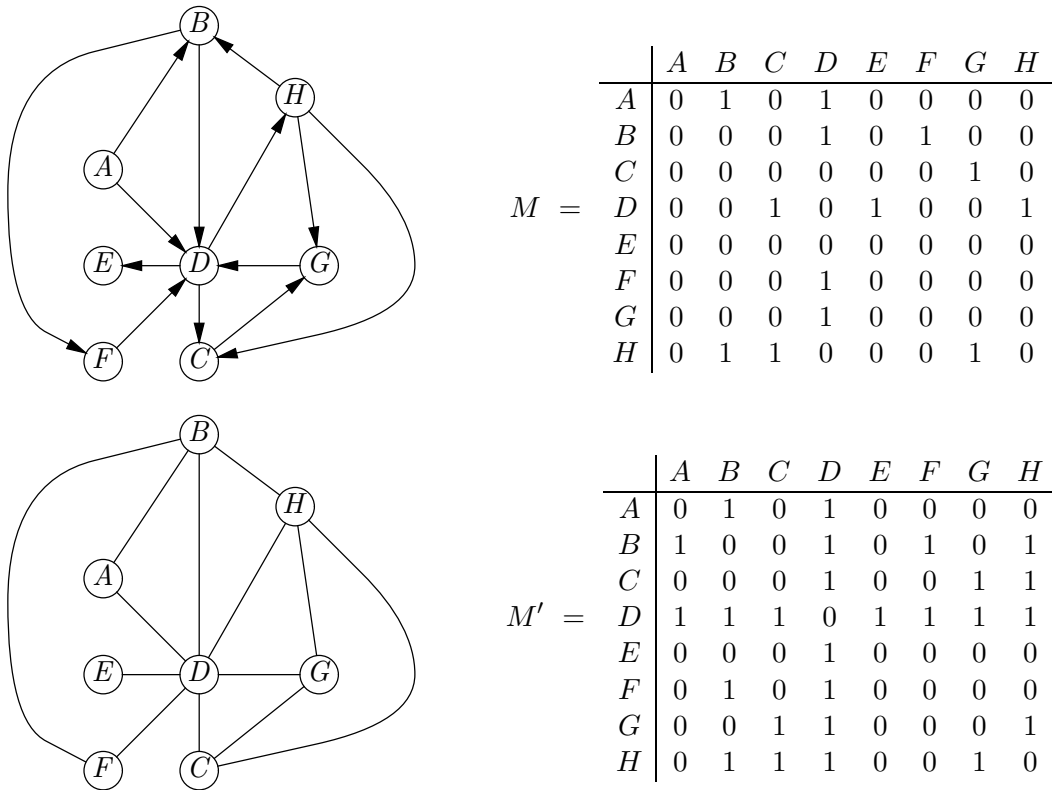
Figure 12.7: Top: a directed graph and corresponding adjacency matrix. Bottom: an undirected variant of the graph and adjacency matrix.

12.2.3   Adjacency matrices

If one's graphs are *dense* (many of the possibly edges exist) and if the important operations include "Is there an edge from v to w?" or "The weight of the edge between v and w," then we can use an *adjacency matrix*. We number the vertices 0 to $|V|-1$ (where $|V|$ is the size of the set V of vertices), and then set up a $|V| \times |V|$ matrix with entry (i,j) equal to 1 if there is an edge from the vertex numbers i to the one numbered j and 0 otherwise. For weighted edges, we can let entry (i,j) be the weight of the edge between i and j, or some special value if there is no edge (this would be an extension of the specifications of Figure 12.3). When a graph is undirected, the matrix will be symmetric. Figure 12.7 illustrates two unweighted graphs—directed and undirected—and their corresponding adjacency matrices.

Adjacency matrices for unweighted graphs have a rather interesting property. Take, for example, the top matrix in Figure 12.7, and consider the result of *multiplying* this matrix by itself. We define the product of any matrix X with itself as

$$(X \cdot X)_{ij} = \sum_{0 \le k < |V|} X_{ik} \cdot X_{kj}.$$

For the example in question, we get

$$M^2 =$$

	A	B	C	D	E	F	G	H
A	0	0	1	1	1	1	0	1
B	0	0	1	1	1	0	0	1
C	0	0	0	1	0	0	0	0
D	0	1	1	0	0	0	2	0
E	0	0	0	0	0	0	0	0
F	0	0	1	0	1	0	0	1
G	0	0	1	0	1	0	0	1
H	0	0	0	2	0	1	1	0

$$M^3 =$$

	A	B	C	D	E	F	G	H
A	0	1	2	1	1	0	2	1
B	0	1	2	0	1	0	2	1
C	0	0	1	0	1	0	0	1
D	0	0	0	3	0	1	1	0
E	0	0	0	0	0	0	0	0
F	0	1	1	0	0	0	2	0
G	0	1	1	0	0	0	2	0
H	0	0	2	2	2	0	0	2

Translating this, we see that $(M \cdot M)_{ij}$ is equal to the number of vertices, k, such that there is an edge from vertex i to vertex k ($M_{ik} = 1$) and there is also an edge from vertex k to vertex j ($M_{kj} = 1$). For any other vertex, one of M_{ik} or M_{kj} will be 0. It should be easy to see, therefore, that M_{ij}^2 is the number of paths following exactly *two* edges from i to j. Likewise, M_{ij}^3 represents the number of paths that are exactly three edges long between i and j. If we use boolean arithmetic instead (where $0 + 1 = 1 + 1 = 1$), we instead get 1's in all positions where there is at least one path of length exactly two between two vertices.

Adjacency matrices are not good for sparse graphs (those where the number of edges is much smaller than V^2). It should be obvious also that they present problems when one wants to add and subtract vertices dynamically.

12.3   Graph Algorithms

Many interesting graph algorithms involve some sort of traversal of the vertices or edges of a graph. Exactly as for trees, one can traverse a graph in either depth-first or breadth-first fashion (intuitively, walking away from the starting vertex as quickly or as slowly as possible).

12.3.1   Marking.

However, in graphs, unlike trees, one can get back to a vertex by following edges away from it, making it necessary to keep track of what vertices have already been visited, an operation I'll call *marking* the vertices. There are several ways to accomplish this.

Mark bits. If vertices are represented by objects, as in the class `Vertex` illustrated in §12.2.1, we can keep a bit in each vertex that indicates whether the vertex has been visited. These bits must initially all be on (or off) and are then flipped when a vertex is first visited. Similarly, we could do this for edges instead.

Mark counts. A problem with mark bits is that one must be sure they are all set the same way at the beginning of a traversal. If traversals may get cut short,

causing mark bits to have arbitrary settings after a traversal, one may be able to use a larger mark instead. Give each traversal a number in increasing sequence (the first traversal is number 1, the second is 2, etc.). To visit a node, set its mark count to the current traversal number. Each new traversal is guaranteed to have a number contained in none of the mark fields (assuming the mark fields are initialized appropriately, say to 0).

Bit vectors. If, as in our abstractions, vertices have numbers, one can keep a bit vector, M, on the side, where $M[i]$ is 1 iff vertex number i has been visited. Bit vectors are easy to reset at the beginning of a traversal.

Ad hoc. Sometimes, the particular traversal being performed provides a way of recognizing a visited vertex. One can't say anything general about this, of course.

12.3.2   A general traversal schema.

Many graph algorithms have the following general form. Italicized capital-letter names must be replaced according to the application.

```
        /* GENERAL GRAPH-TRAVERSAL SCHEMA */

COLLECTION_OF_VERTICES fringe;

fringe = INITIAL_COLLECTION;
while (! fringe.isEmpty()) {
    Vertex v = fringe.REMOVE_HIGHEST_PRIORITY_ITEM();

    if (! MARKED(v)) {
        MARK(v);
        VISIT(v);
        For each edge (v,w) {
            if (NEEDS_PROCESSING(w))
                Add w to fringe;
        }
    }
}
```

In the following sections, we look at various algorithms that fit this schema[2].

---

[2]In this context, a *schema* (plural *schemas* or *schemata*) is a template, containing some pieces that must be replaced. Logical systems, for example, often contain *axiom schemata* such as

$$(\forall x \mathcal{P}(x)) \supset \mathcal{P}(y),$$

where $\mathcal{P}$ may be replaced by any logical formula with a distinguished free variable (well, roughly).

12.3.3   Generic depth-first and breadth-first traversal

Depth-first traversal in graphs is essentially the same as in trees, with the exception of the check for "already visited." To implement

```
/** Perform the operation VISIT on each vertex reachable from V
 * in depth-first order. */
void depthFirstVisit(Vertex v)
```

we use the general graph-traversal schema with the following replacements.

COLLECTION_OF_VERTICES is a stack type.

INITIAL_COLLECTION is the set $\{v\}$.

REMOVE_HIGHEST_PRIORITY_ITEM pops and returns the top.

MARK **and** *MARKED* set and check a mark bit (see discussion above).

NEEDS_PROCESSING means "not *MARKED*."

   Here, as is often the case, we could dispense with *NEEDS_PROCESSING* (make it always TRUE). The only effect would be to increase the size of the stack somewhat.

   Breadth-first search is nearly identical. The only differences are as follows.

COLLECTION_OF_VERTICES is a (FIFO) queue type.

REMOVE_HIGHEST_PRIORITY_ITEM is to remove and return the first (least-recently-added) item in the queue.

12.3.4   Topological sorting.

A *topological sort* of a directed graph is a listing of its vertices in such an order that if vertex w is reachable from vertex v, then w is listed after v. Thus, if we think of a graph as representing an ordering relation on the vertices, a topological sort is a linear ordering of the vertices that is consistent with that ordering relation. A cyclic directed graph has no topological sort. For example, topological sort is the operation that the UNIX `make` utility implicitly performs to find an order for executing commands that brings every file up to date before it is needed in a subsequent command.

   To perform a topological sort, we associate a count with each vertex of the number of incoming edges from as-yet unprocessed vertices. For the version below, I use an array to keep these counts. The algorithm for topological sort now looks like this.

```
/** An array of the vertices in G in topologically sorted order.
 * Assumes G is acyclic. */
static int[] topologicalSort(Digraph G)
{
   int[] count = new int[G.numVertices()];
   int[] result = new int[G.numVertices()];
   int k;

   for (int v = 0; v < G.numVertices(); v += 1)
      count[v] = G.inDegree(v);

   Graph-traversal schema replacement for topological sorting;

    return result;
}
```

The schema replacement for topological sorting is as follows.

COLLECTION_OF_VERTICES can be any set, multiset, list, or sequence type for vertices (stacks, queues, etc., etc.).

INITIAL_COLLECTION is the set of all `v` with `count[v]=0`.

REMOVE_HIGHEST_PRIORITY_ITEM can remove any item.

MARKED **and** *MARK* can be trivial (i.e., always return FALSE and do nothing, respectively).

VISIT(v) makes `v` the next non-null element of `result` and decrements `count[w]` for each edge `(v,w)` in `G`.

NEEDS_PROCESSING is true if `count[w]==0`.

Figure 12.8 illustrates the algorithm.

12.3.5 Minimum spanning trees

Consider a connected undirected graph with edge weights. A *minimum(-weight) spanning tree* (or *MST* for short) is a tree that is a subgraph of the given graph, contains all the vertices of the given graph, and minimizes the sum of its edge weights. For example, we might have a bunch of cities that we wish to connect up with telephone lines so as to provide a path between any two, all at minimal cost. The cities correspond to vertices and the possible connections between cities correspond to edges[3]. Finding a minimal set of possible connections is the same as finding a minimum spanning tree (there can be more than one). To do this, we make use of a useful fact about MSTs.

---

[3]It turns out that to get *really* lowest costs, you want to introduce strategically placed extra "cities" to serve as connecting points. We'll ignore that here.
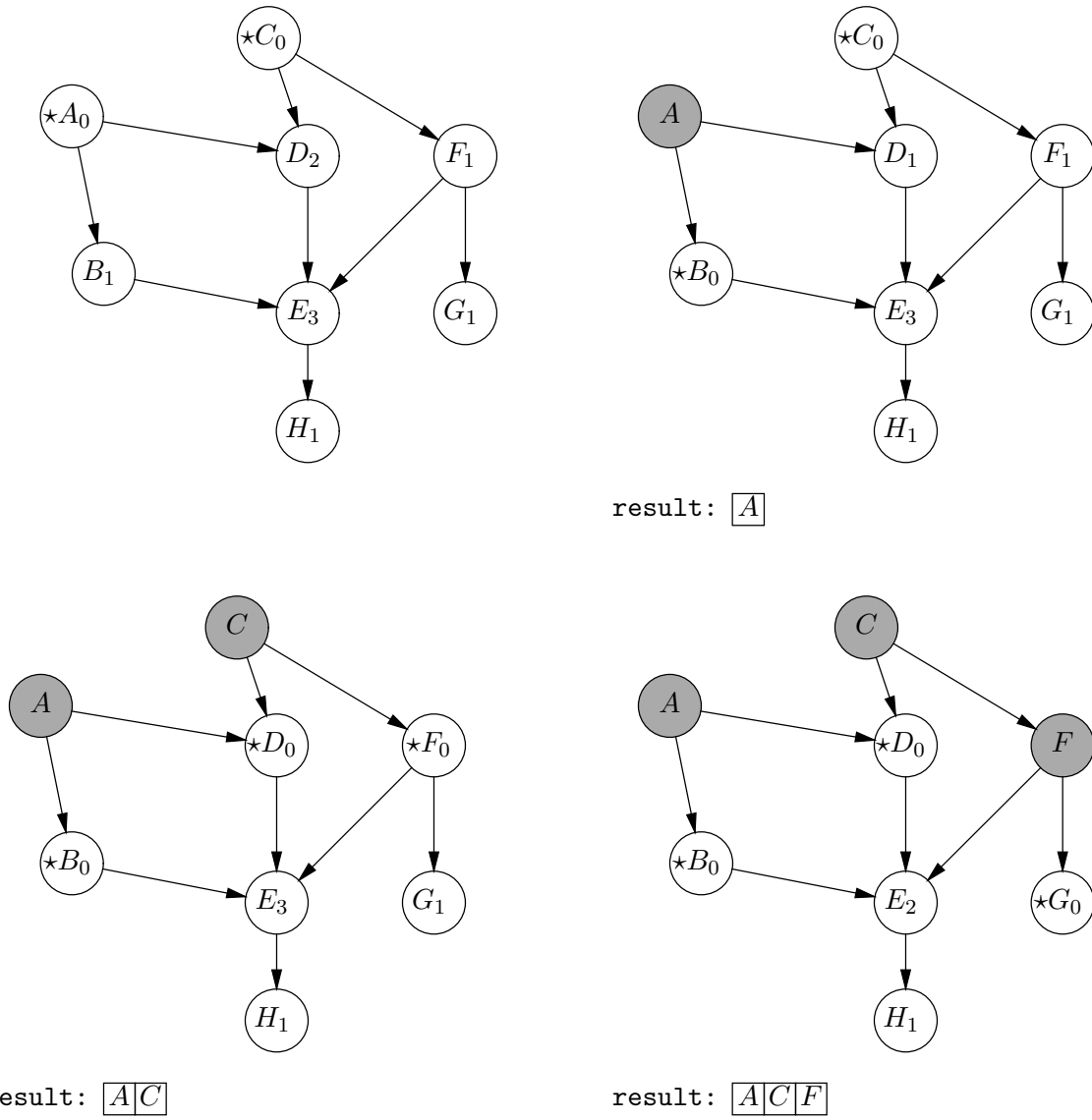
Figure 12.8: The input to a topological sort (upper left) and three stages in its computation. The shaded nodes are those that have been processed and moved to the `result`. The starred nodes are the ones in the fringe. Subscripts indicate `count` fields. A possible final sequence of nodes, given this start, is *A, C, F, D, B, E, G, H*.

FACT: If the vertices of a connected graph G are divided into two disjoint non-empty sets, V_0 and V_1, then any MST for G will contain one of the edges running between a vertex in V_0 and a vertex in V_1 that has minimal weight.

Proof. It's convenient to use a proof by contradiction. Suppose that some MST, T, doesn't contain any of the edges between V_0 and V_1 with minimal weight. Consider the effect of adding to T an edge from V_0 to V_1, e, that does have minimal weight, thus giving T' (there must be such an edge, since otherwise T would be unconnected). Since T was a tree, the result of adding this new edge must have a cycle involving e (since it adds a new path between two nodes that already had a path between them in T). This is only possible if the cycle contains another edge from T, e', that also runs between V_0 and V_1. By hypothesis, e has weight less than e'. If we remove e' from T', we get a tree once again, but since we have substituted e for e', the sum of the edge weights for this new tree is less than that for T, a contradiction of T's minimality. Therefore, it was wrong to assume that T contained no minimal-weight edges from V_0 to V_1. (End of Proof)

We use this fact by taking V_0 to be a set of processed (marked) vertices for which we have selected edges that form a tree, and taking V_1 to be the set of all other vertices. By the Fact above, we may safely add to the tree any minimal-weight edge from the marked vertices to an unmarked vertex.

This gives what is known as Prim's algorithm. This time, we introduce two extra pieces of information for each node, `dist[v]` (a weight value), and `parent[v]` (a Vertex). At each point in the algorithm, the `dist` value for an unprocessed vertex (still in the fringe) is the minimal distance (weight) between it and a processed vertex, and the `parent` value is the processed vertex that achieves this minimal distance.

```
/** For all vertices v in G, set PARENT[v] to be the parent of v in
 *  a MST of G. For each v in G, DIST[v] may be altered arbitrarily.
 *  Assumes that G is connected. WEIGHT[e] is the weight of edge e. */
static void MST(Graph G, int[] weight, int[] parent, int[] dist)
{
    for (int v = 0; v < G.numVertices(); v += 1) {
        dist[v] = ∞;
        parent[v] = -1;
    }

    Let r be an arbitrary vertex in G;
    dist[r] = 0;

    Graph-traversal schema replacement for MST;
}
```

The appropriate "settings" for the graph-traversal schema are as follows.

COLLECTION_OF_VERTICES is a priority queue of vertices ordered by `dist` values, with smaller `dist`s having higher priorities.

INITIAL_COLLECTION contains all the vertices of G.

REMOVE_HIGHEST_PRIORITY_ITEM removes the first item in the priority queue.

VISIT(v): for each edge (v, w) with `weight` n, if w is unmarked, and $dist[w] > n$, set $dist[w]$ to n and set $parent[w]$ to v.

NEEDS_PROCESSING(v) is always false.

Figure 12.9 illustrates this algorithm in action.

12.3.6   Shortest paths

Suppose that we are given a weighted graph (directed or otherwise) and we want to find the shortest paths from some starting node to each other reachable node. A succinct presentation of the results of this algorithm is known as a *shortest-path tree*. This is a (not necessarily minimum) spanning tree for the graph with the desired starting node as the root such that the path from the root to each other node in the tree is also a path of minimal total weight in the full graph.

A common algorithm for doing this, known as Dijkstra's algorithm, looks almost identical to Prim's algorithm for MSTs. We have the same `PARENT` and `DIST` data as before. However, whereas in Prim's algorithm, `DIST` gives the shortest distance from an unmarked vertex to the marked vertices, in Dijkstra's algorithm it gives the length of the shortest path known so far that leads to it from the starting node.

```
/** For all vertices v in G reachable from START, set PARENT[v]
 *  to be the parent of v in a shortest-path tree from START in G. For
 *  all vertices in this tree, DIST[v] is set to the distance from START
 *  WEIGHT[e] are edge weights.  Assumes that vertex START is in G. */
static void shortestPaths(Graph G, int start, int[] weight,
                          int[] parent, int[] dist)
{
    for (int v = 0; v < G.numVertices(); v += 1) {
        dist[v] = ∞;
        parent[v] = -1;
    }
    dist[start] = 0;

    Graph-traversal schema replacement for shortest-path tree;
}
```

where we substitute into the schema as follows:

COLLECTION_OF_VERTICES is a priority queue of vertices ordered by `dist` values, with smaller `dist`s having higher priorities.

INITIAL_COLLECTION contains all the vertices of G.

REMOVE_HIGHEST_PRIORITY_ITEM removes the first item in the priority queue.
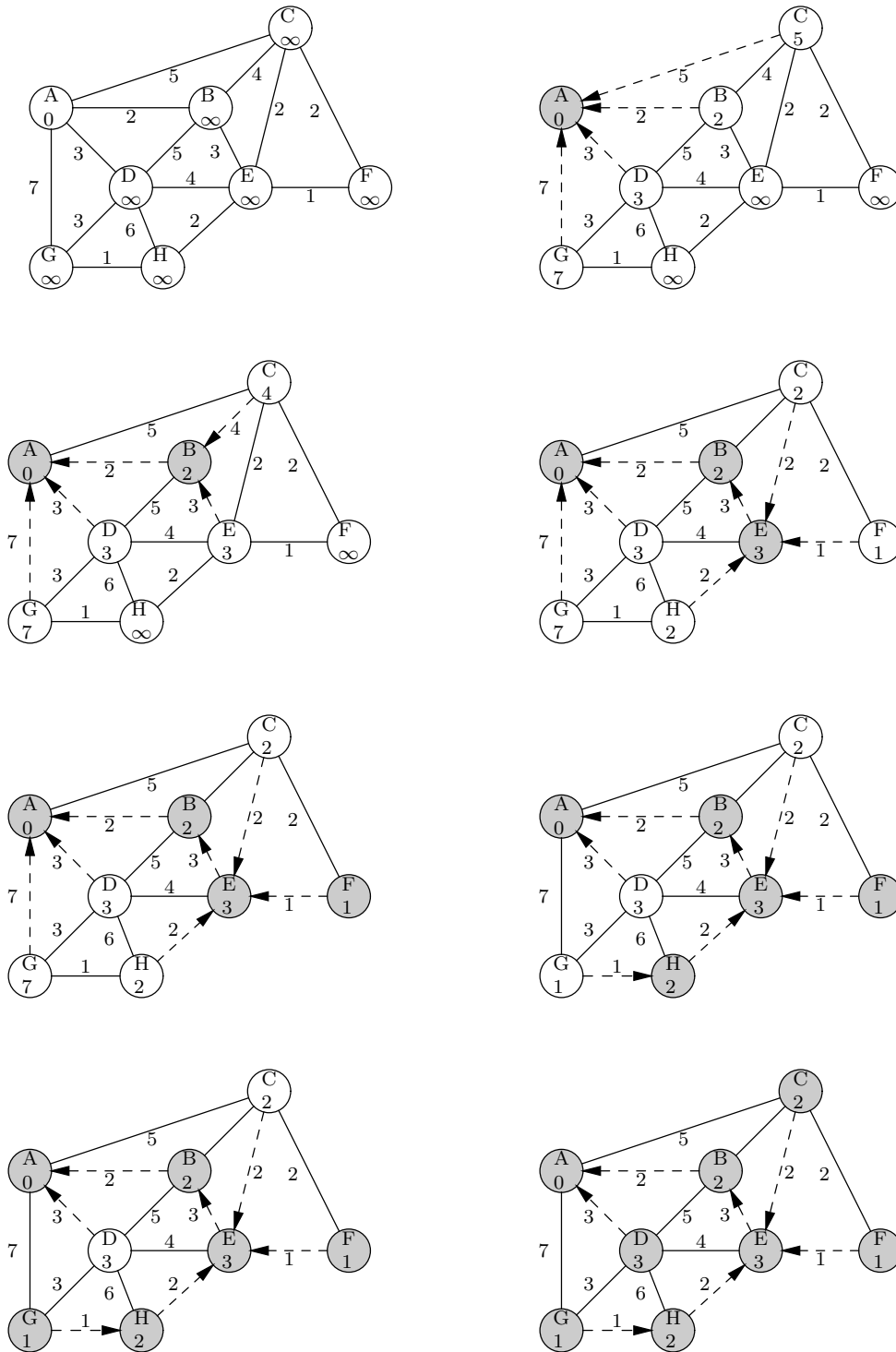
Figure 12.9: Prim's algorithm for minimum spanning tree. Vertex **r** is *A*. The numbers in the nodes denote `dist` values. Dashed edges denote `parent` values; they form a MST after the last step. Unshaded nodes are in the fringe. The last two steps (which don't change `parent` pointers) have been collapsed into one.

MARK and MARKED can be trivial (return false and do nothing, respectively).

VISIT(v): for each edge (v, w) with weight n, if w is unmarked, and $dist[w] > n + dist[v]$, set $dist[w]$ to $n + dist[v]$ and set $parent[w]$ to v. Reorder `fringe` as needed.

NEEDS_PROCESSING(v) is always false.

Figure 12.10 illustrates Dijkstra's algorithm in action.

Because of their very similar structure, the asymptotic running times of Dijkstra's and Prim's algorithms are similar. We visit each vertex once (removing an item from the priority queue), and reorder the priority queue at most once for each edge. Hence, if V is the number of vertices of G and E is the number of edges, we get an upper bound on the time required by these algorithms of $O((V + E) \cdot \lg V)$.

12.3.7   Kruskal's algorithm for MST

Just so you don't get the idea that our graph traversal schema is the only possible way to go, we'll consider a "classical" method for forming a minimum spanning tree, known as Kruskal's algorithm. This algorithm relies on a *union-find* structure. At any time, this structure contains a *partition* of the vertices: a collection of disjoint sets of vertices that includes all of vertices. Initially, each vertex is alone in its own set. The idea is that we build up an MST one edge at a time. We repeatedly choose an edge of minimum weight that joins vertices in two different sets, add that edge to the MST we are building, and then combine (union) the two sets of vertices into one set. This process continues until all the sets have been combined into one (which must contain all the vertices). At any point, each set is a bunch of vertices that are all reachable from each other via the edges so far added to the MST. When there is only one set, it means that all of the vertices are reachable, and so we have a set of edges that spans the tree. It follows from the Fact in §12.3.5 that if we always add the minimally weighted edge that connects two of the disjoint sets of vertices, that edge can always be part of a MST, so the final result must also be a MST. Figure 12.11 illustrates the idea.

For the program, I'll assume we have a type—`UnionFind`—representing sets of sets of vertices. We need two operations on this type: an inquiry S.`sameSet(`v`, `w`)` that tells us whether vertices v and w are in the same set in S, and an operation S.`union(`v`, `w`)` that combines the sets containing vertices v and w into one. I'll also assume a "set of edges" set to contain the result.
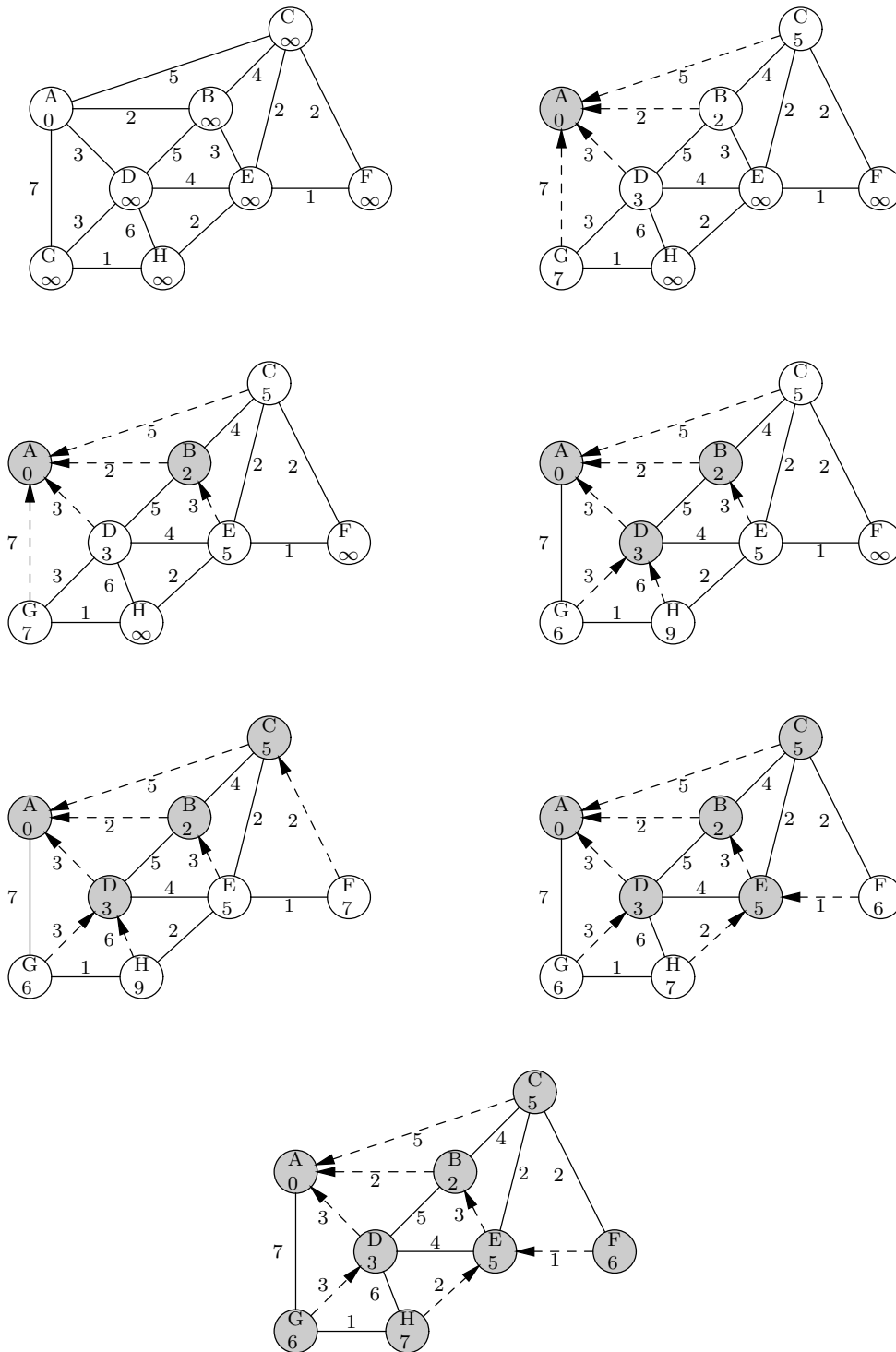
Figure 12.10: Dijkstra's algorithm for shortest paths. The starting node is *A*. Numbers in nodes represent minimum distance to node *A* so far found (`dist`). Dashed arrows represent `parent` pointers; their final values show the shortest-path tree. The last three steps have been collapsed to one.
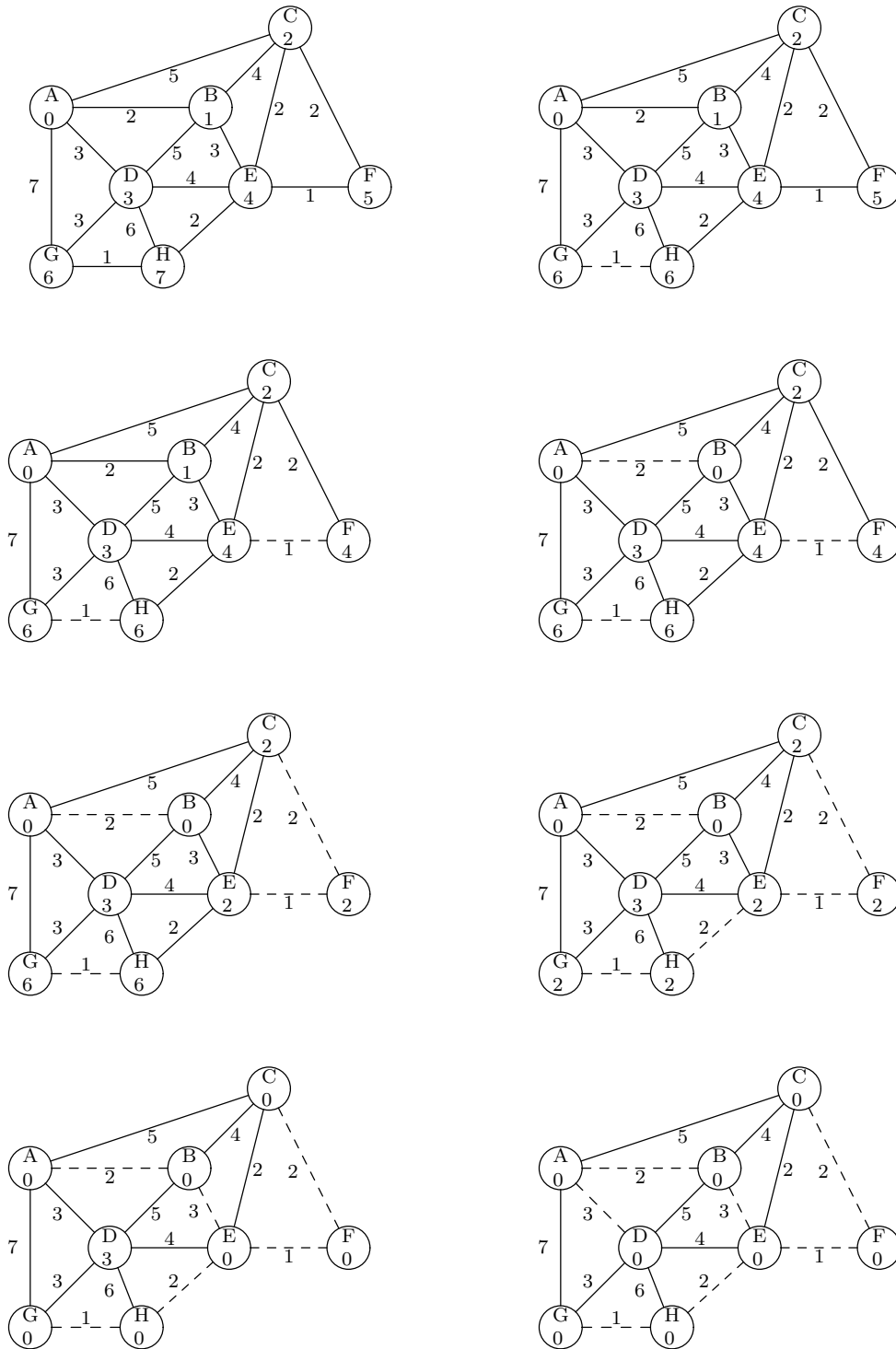
Figure 12.11: Kruskal's algorithm. The numbers in the vertices denote sets: vertices with the same number are in the same set. Dashed edges have been added to the MST. This is different from the MST found in Figure 12.9.

```
/** Return a subset of edges of G forming a minimum spanning tree for G.
 *  G must be a connected undirected graph. WEIGHT gives edge weights. */
EdgeSet MST(Graph G, int[] weight)
{
    UnionFind S;
    EdgeSet E;

    // Initialize S to { {v} | v is a vertex of G };
    S = new UnionFind(G.numVertices());
    E = new EdgeSet();

    For each edge (v,w) in G in order of increasing weight {
        if (! S.sameSet(v, w)) {
            Add (v,w) to E;
            S.union(v, w);
        }
    }

    return E;
}
```

The tricky part is this union-find bit. From what you know, you might well guess that each **sameSet** operation will require time $\Theta(N \lg N)$ in the worst case (look in each of up to N sets each of size up to N). Interestingly enough, there is a better way. Let's assume (as in this problem) that the sets contain integers from 0 to $N-1$. At any given time, there will be up to N disjoint sets; we'll give them names (well, numbers really) by selecting a single *representative member* of each set and using that member (a number between 0 and $N-1$) to identify the set. Then, if we can find the current representative member of the set containing any vertex, we can tell if two vertices are in the same set by seeing if their representative members are the same. One way to do this is to represent each disjoint set as a *tree* of vertices, but with children pointing at parents (you may recall that I said such a structure would eventually be useful). The root of each tree is the representative member, which we may find by following parent links. For example, we can represent the set of sets

$$\{\{1, 2, 4, 6, 7\}, \{0, 3, 5\}, \{8, 9, 10\}\}$$

with the forest of trees
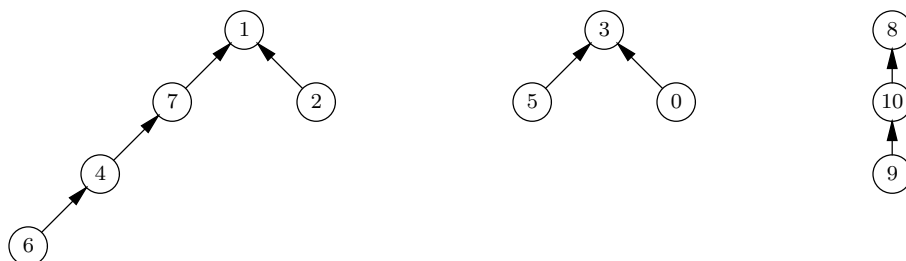
We represent all this with a single integer array, parent, with parent[v] containing the number of parent node of v, or -1 if v has no parent (i.e., is a representative member). The union operation is quite simple: to compute $S.\text{union}(v, w)$, we find the roots of the trees containing v and w (by following the parent chain) and then make one of the two roots the child of the other. So, for example, we could compute $S.\text{union(6, 0)}$ by finding the representative member for 6 (which is 1), and for 0 (which is 3) and then making 3 point to 1:
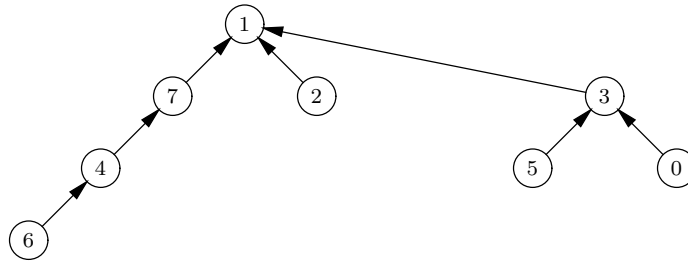
For best results, we should make the tree of lesser "rank" (roughly, height) point to the one of larger rank[4].

However, while we're at it, let's throw in a twist. After we traverse the paths from 6 up to 1 and from 0 to 3, we'll re-organize the tree by having every node in those paths point directly at node 1 (in effect "memoizing" the result of the operation of finding the representative member). Thus, after finding the representative member for 6 and 0 and unioning, we will have the following, much flatter tree:
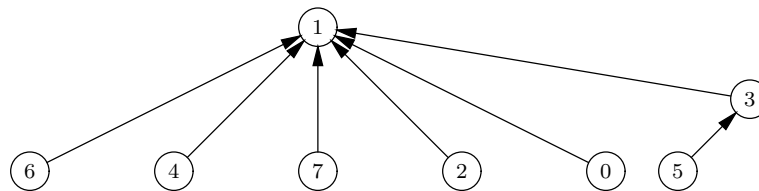
This re-arrangement, which is called *path compression,* causes subsequent inquiries about vertices 6, 4, and 0 to be considerably faster than before. It turns out that with this trick (and the heuristic of making the shallower tree point at the deeper in a union), any sequence of M union and sameSet operations on a set of sets containing a total of N elements can be performed in time $O(\alpha(M, N)M)$. Here, $\alpha(M, N)$ is an inverse of *Ackerman's function.* Specifically, $\alpha(M, N)$ is defined as the minimum i such that $A(i, \lfloor M/N \rfloor) > \lg N$, where

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

---

[4]We're cheating a bit in this section to make the effects of the optimizations we describe a bit clearer. We could not have constructed the "stringy" trees in these examples had we always made the lesser-rank tree point to the greater. So in effect, our examples start from union-find trees that were constructed in haphazard fashion, and we show what happens if we start doing things right from then on.

Well, this is all rather complicated, but suffice it to say that A grows monumentally fast, so that α grows with subglacial slowness, and is for all mortal purposes ≤ 4. In short, the *amortized cost* of M operations (`union` and `sameSets` in any combination) is roughly constant per operation. Thus, the time required for Kruskal's algorithm is dominated by the sorting time for the edges, and is asymptotically $O(E \lg E)$, for E the number of edges. This in turn equals $O(E \lg V)$ for a connected graph, where V is the number of vertices.

Exercises

12.1. A borogove and a snark find themselves in a maze of twisty little passages that connect numerous rooms, one of which is the maze exit. The snark, being a boojum, finds borogoves especially tasty after a long day of causing people to softly and silently vanish away. Unfortunately for the snark (and contrariwise for his prospective snack), borogoves can run twice as fast as snarks and have an uncanny ability of finding the shortest route to the exit. Fortunately for the snark, his preternatural senses tell him precisely where the borogove is at any time, and he knows the maze like the back of his, er, talons. If he can arrive at the exit or in any of the rooms in the borogove's path before the borogove does (strictly before, not at the same time), he can catch it. The borogove is not particularly intelligent, and will always take the shortest path, even if the snark is waiting on it.

Thus, for example, in the following maze, the snark (starting at 'S') will dine in the shaded room, which he reaches in 6 time units, and the borogove (starting at 'B') in 7. The numbers on the connecting passages indicate distances (the numbers inside rooms are just labels). The snark travels at 0.5 units/hour, and the borogove at 1 unit/hour.
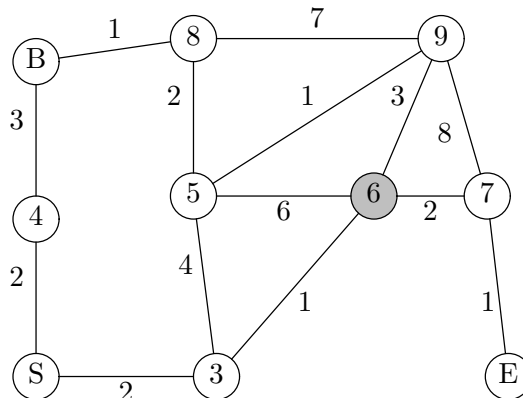
Write a program to read in a maze such as the above, and print one of two messages: `Snark eats`, or `Borogove escapes`, as appropriate. Place your answer in a class `Chase` (see the templates in ˜cs61b/hw/hw7).

The input is as follows.

- A positive integer $N \geq 3$ indicating the number of rooms. You may assume that $N < 1024$. The rooms are assumed to be numbered from 0 to $N - 1$.

Room 0 is always the exit. Initially room 1 contains the borogove and room 2 contains the snark.

- A sequence of edges, each consisting of two room numbers (the order of the room numbers is immaterial) followed by an integer distance.

Assume that whenever the borogove has a choice between passages to take (i.e., all lead to a shortest path), he chooses the one to the room with the lowest number.

For the maze shown, a possible input is as follows.

```
10
2 3 2   2 4 2            3 5 4   3 6 1        4 1 3
5 6 6   5 8 2  5 9 1     6 7 2   6 9 3
7 0 1   7 9 8            1 8 1
8 9 7
```

Index