Solutions Manual

Data Structures and Algorithm Analysis in Java™

Second Edition

Mark Allen Weiss
Florida International University

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Publisher Greg Tobin
Executive Editor Michael Hirsch
Editorial Assistant Lindsey Triebel
Marketing Manager Michelle Brown
Marketing Assistant Dana Lopreato
Digital Asset Manager Marianne Groth
Composition Windfall Software, using ZzTEX

Access the latest information about Addison-Wesley titles from our World Wide Web site: http://www.aw-bc.com/computing

1 2 3 4 5 6 7 8 9 10—PDF— 9 08 07 06

CONTENTS

PREFACE

Included in this manual are answers to many of the exercises in the textbook *Data Structures and Algorithm Analysis in Java,* second edition, published by Addison-Wesley. These answers reflect the state of the book in the first printing of the second edition.

Specifically omitted are general programming questions and any question whose solution is pointed to by a reference at the end of the chapter. Solutions vary in degree of completeness; generally, minor details are left to the reader. For clarity, the few code segments that are present are meant to be pseudo-Java rather than completely perfect code.

Errors can be reported to weiss@fiu.edu.

CHAPTER 1

Introduction

1.4 The general way to do this is to write a procedure with heading

```
void processFile( String fileName );
```

which opens *fileName*, does whatever processing is needed, and then closes it. If a line of the form

```
#include SomeFile
```

is detected, then the call

```
processFile( SomeFile );
```

is made recursively. Self-referential includes can be detected by keeping a list of files for which a call to *processFile* has not yet terminated, and checking this list before making a new call to *processFile*.

1.5

```
public static int ones( int n )
{
   if( n < 2 )
      return n;
   return n % 2 + ones( n / 2 );
}
```

1.7 **(a)** The proof is by induction. The theorem is clearly true for $0 < X \leq 1$, since it is true for $X = 1$, and for $X < 1$, $\log X$ is negative. It is also easy to see that the theorem holds for $1 < X \leq 2$, since it is true for $X = 2$, and for $X < 2$, $\log X$ is at most 1. Suppose the theorem is true for $p < X \leq 2p$ (where p is a positive integer), and consider any $2p < Y \leq 4p$ ($p \geq 1$). Then $\log Y = 1 + \log(Y/2) < 1 + Y/2 < Y/2 + Y/2 \leq Y$, where the first inequality follows by the inductive hypothesis.

(b) Let $2^X = A$. Then $A^B = (2^X)^B = 2^{XB}$. Thus $\log A^B = XB$. Since $X = \log A$, the theorem is proved.

1.8 **(a)** The sum is 4/3 and follows directly from the formula.

(b) $S = \frac{1}{4} + \frac{2}{4^2} + \frac{3}{4^3} + \cdots$. $4S = 1 + \frac{2}{4} + \frac{3}{4^2} + \cdots$. Subtracting the first equation from the second gives $3S = 1 + \frac{1}{4} + \frac{2}{4^2} + \cdots$. By part (a), $3S = 4/3$ so $S = 4/9$.

(c) $S = \frac{1}{4} + \frac{4}{4^2} + \frac{9}{4^3} + \cdots$. $4S = 1 + \frac{4}{4} + \frac{9}{4^2} + \frac{16}{4^3} + \cdots$. Subtracting the first equation from the second gives $3S = 1 + \frac{3}{4} + \frac{5}{4^2} + \frac{7}{4^3} + \cdots$. Rewriting, we get $3S = 2\sum_{i=0}^{\infty} \frac{i}{4^i} + \sum_{i=0}^{\infty} \frac{1}{4^i}$. Thus $3S = 2(4/9) + 4/3 = 20/9$. Thus $S = 20/27$.

(d) Let $S_N = \sum_{i=0}^{\infty} \frac{i^N}{4^i}$. Follow the same method as in parts (a)–(c) to obtain a formula for S_N in terms of $S_{N-1}, S_{N-2}, \ldots, S_0$ and solve the recurrence. Solving the recurrence is very difficult.

1.9 $\sum_{i=\lfloor N/2 \rfloor}^{N} \frac{1}{i} = \sum_{i=1}^{N} \frac{1}{i} - \sum_{i=1}^{\lfloor N/2-1 \rfloor} \frac{1}{i} \approx \ln N - \ln N/2 \approx \ln 2.$

1.10 $2^4 = 16 \equiv 1 \pmod 5$. $(2^4)^{25} \equiv 1^{25} \pmod 5$. Thus $2^{100} \equiv 1 \pmod 5$.

1.11 **(a)** Proof is by induction. The statement is clearly true for $N = 1$ and $N = 2$. Assume true for $N = 1, 2, \ldots, k$. Then $\sum_{i=1}^{k+1} F_i = \sum_{i=1}^{k} F_i + F_{k+1}$. By the induction hypothesis, the value of the sum on the right is $F_{k+2} - 2 + F_{k+1} = F_{k+3} - 2$, where the latter equality follows from the definition of the Fibonacci numbers. This proves the claim for $N = k + 1$, and hence for all N.

(b) As in the text, the proof is by induction. Observe that $\phi + 1 = \phi^2$. This implies that $\phi^{-1} + \phi^{-2} = 1$. For $N = 1$ and $N = 2$, the statement is true. Assume the claim is true for $N = 1, 2, \ldots, k$.

$$F_{k+1} = F_k + F_{k-1}$$

by the definition, and we can use the inductive hypothesis on the right-hand side, obtaining

$$\begin{aligned} F_{k+1} &< \phi^k + \phi^{k-1} \\ &< \phi^{-1}\phi^{k+1} + \phi^{-2}\phi^{k+1} \\ F_{k+1} &< (\phi^{-1} + \phi^{-2})\phi^{k+1} < \phi^{k+1} \end{aligned}$$

and proving the theorem.

(c) See any of the advanced math references at the end of the chapter. The derivation involves the use of generating functions.

1.12 **(a)** $\sum_{i=1}^{N}(2i - 1) = 2\sum_{i=1}^{N} i - \sum_{i=1}^{N} 1 = N(N + 1) - N = N^2$.

(b) The easiest way to prove this is by induction. The case $N = 1$ is trivial. Otherwise,

$$\begin{aligned} \sum_{i=1}^{N+1} i^3 &= (N + 1)^3 + \sum_{i=1}^{N} i^3 \\ &= (N + 1)^3 + \frac{N^2(N + 1)^2}{4} \\ &= (N + 1)^2\left[\frac{N^2}{4} + (N + 1)\right] \\ &= (N + 1)^2\left[\frac{N^2 + 4N + 4}{4}\right] \\ &= \frac{(N + 1)^2(N + 2)^2}{2^2} \\ &= \left[\frac{(N + 1)(N + 2)}{2}\right]^2 \\ &= \left[\sum_{i=1}^{N+1} i\right]^2 \end{aligned}$$

CHAPTER 2

Algorithm Analysis

2.1 $2/N$, 37, $\sqrt{N}$, N, $N \log\log N$, $N \log N$, $N \log(N^2)$, $N \log^2 N$, $N^{1.5}$, N^2, $N^2 \log N$, N^3, $2^{N/2}$, 2^N. $N \log N$ and $N \log(N^2)$ grow at the same rate.

2.2 **(a)** True.

(b) False. A counterexample is $T_1(N) = 2N$, $T_2(N) = N$, and $f(N) = N$.

(c) False. A counterexample is $T_1(N) = N^2$, $T_2(N) = N$, and $f(N) = N^2$.

(d) False. The same counterexample as in part (c) applies.

2.3 We claim that $N \log N$ is the slower growing function. To see this, suppose otherwise. Then, $N^{\epsilon/\sqrt{\log N}}$ would grow slower than $\log N$. Taking logs of both sides, we find that, under this assumption, $\epsilon/\sqrt{\log N} \log N$ grows slower than $\log\log N$. But the first expression simplifies to $\epsilon\sqrt{\log N}$. If $L = \log N$, then we are claiming that $\epsilon\sqrt{L}$ grows slower than $\log L$, or equivalently, that $\epsilon^2 L$ grows slower than $\log^2 L$. But we know that $\log^2 L = o(L)$, so the original assumption is false, proving the claim.

2.4 Clearly, $\log^{k_1} N = o(\log^{k_2} N)$ if $k_1 < k_2$, so we need to worry only about positive integers. The claim is clearly true for $k = 0$ and $k = 1$. Suppose it is true for $k < i$. Then, by L'Hôpital's rule,

$$\lim_{N\to\infty} \frac{\log^i N}{N} = \lim_{N\to\infty} i\frac{\log^{i-1} N}{N}$$

The second limit is zero by the inductive hypothesis, proving the claim.

2.5 Let $f(N) = 1$ when N is even, and N when N is odd. Likewise, let $g(N) = 1$ when N is odd, and N when N is even. Then the ratio $f(N)/g(N)$ oscillates between 0 and *inf*.

2.6 **(a)** 2^{2^N}

(b) $O(\log\log D)$

2.7 For all these programs, the following analysis will agree with a simulation:

(I) The running time is $O(N)$.

(II) The running time is $O(N^2)$.

(III) The running time is $O(N^3)$.

(IV) The running time is $O(N^2)$.

(V) j can be as large as i^2, which could be as large as N^2. k can be as large as j, which is N^2. The running time is thus proportional to $N \cdot N^2 \cdot N^2$, which is $O(N^5)$.

(VI) The *if* statement is executed at most N^3 times, by previous arguments, but it is true only $O(N^2)$ times (because it is true exactly i times for each i). Thus the innermost loop is only executed $O(N^2)$

times. Each time through, it takes $O(j^2) = O(N^2)$ time, for a total of $O(N^4)$. This is an example where multiplying loop sizes can occasionally give an overestimate.

2.8 **(a)** It should be clear that all algorithms generate only legal permutations. The first two algorithms have tests to guarantee no duplicates; the third algorithm works by shuffling an array that initially has no duplicates, so none can occur. It is also clear that the first two algorithms are completely random, and that each permutation is equally likely. The third algorithm, due to R. Floyd, is not as obvious; the correctness can be proved by induction. See J. Bentley, "Programming Pearls," *Communications of the ACM* 30 (1987), 754–757. Note that if the second line of algorithm 3 is replaced with the statement

```
swapReferences( a[i], a[ randInt( 0, n-1 ) ] );
```

then not all permutations are equally likely. To see this, notice that for $N = 3$, there are 27 equally likely ways of performing the three swaps, depending on the three random integers. Since there are only 6 permutations, and 6 does not evenly divide 27, each permutation cannot possibly be equally represented.

(b) For the first algorithm, the time to decide if a random number to be placed in $a[i]$ has not been used earlier is $O(i)$. The expected number of random numbers that need to be tried is $N/(N - i)$. This is obtained as follows: i of the N numbers would be duplicates. Thus the probability of success is $(N - i)/N$. Thus the expected number of independent trials is $N/(N - i)$. The time bound is thus

$$\sum_{i=0}^{N-1} \frac{Ni}{N-i} < \sum_{i=0}^{N-1} \frac{N^2}{N-i} < N^2 \sum_{i=0}^{N-1} \frac{1}{N-i} < N^2 \sum_{j=1}^{N} \frac{1}{j} = O(N^2 \log N)$$

The second algorithm saves a factor of i for each random number, and thus reduces the time bound to $O(N \log N)$ on average. The third algorithm is clearly linear.

(c,d) The running times should agree with the preceding analysis if the machine has enough memory. If not, the third algorithm will not seem linear because of a drastic increase for large N.

(e) The worst-case running time of algorithms I and II cannot be bounded because there is always a finite probability that the program will not terminate by some given time T. The algorithm does, however, terminate with probability 1. The worst-case running time of the third algorithm is linear—its running time does not depend on the sequence of random numbers.

2.9 Algorithm 1 at 10,000 is about 38 minutes and at 100,000 is about 26 days. Algorithms 1–4 at 1 million are approximately: 72 years, 4 hours, 0.7 seconds, and 0.03 seconds respectively. These calculations assume a machine with enough memory to hold the entire array.

2.10 **(a)** $O(N)$

(b) $O(N^2)$

(c) The answer depends on how many digits past the decimal point are computed. Each digit costs $O(N)$.

2.11 **(a)** Five times as long, or 2.5 ms.

(b) Slightly more than five times as long.

(c) 25 times as long, or 12.5 ms.

(d) 125 times as long, or 62.5 ms.

2.12 **(a)** 12000 times as large a problem, or input size 1,200,000.

(b) input size of approximately 425,000.

(c) $\sqrt{12000}$ times as large a problem, or input size 10,954.

(d) $12000^{1/3}$ times as large a problem, or input size 2,289.

2.13 **(a)** $O(N^2)$.

(b) $O(N \log N)$.

2.15 Use a variation of binary search to get an $O(\log N)$ solution (assuming the array is preread).

2.20 **(a)** Test to see if N is an odd number (or 2) and is not divisible by 3, 5, 7, . . . , $\sqrt{N}$.

(b) $O(\sqrt{N})$, assuming that all divisions count for one unit of time.

(c) $B = O(\log N)$.

(d) $O(2^{B/2})$.

(e) If a 20-bit number can be tested in time T, then a 40-bit number would require about T^2 time.

(f) B is the better measure because it more accurately represents the *size* of the input.

2.21 The running time is proportional to N times the sum of the reciprocals of the primes less than N. This is $O(N \log \log N)$. See Knuth, Volume 2.

2.22 Compute X^2, X^4, X^8, X^{10}, X^{20}, X^{40}, X^{60}, and X^{62}.

2.23 Maintain an array that can be filled in a for loop. The array will contain X, X^2, X^4, up to $X^{2^{\lfloor \log N \rfloor}}$. The binary representation of N (which can be obtained by testing even or odd and then dividing by 2, until all bits are examined) can be used to multiply the appropriate entries of the array.

2.24 For $N = 0$ or $N = 1$, the number of multiplies is zero. If $b(N)$ is the number of ones in the binary representation of N, then if $N > 1$, the number of multiplies used is

$$\lfloor \log N \rfloor + b(N) - 1$$

2.25 **(a)** A.

(b) B.

(c) The information given is not sufficient to determine an answer. We have only worst-case bounds.

(d) Yes.

2.26 **(a)** Recursion is unnecessary if there are two or fewer elements.

(b) One way to do this is to note that if the first $N - 1$ elements have a majority, then the last element cannot change this. Otherwise, the last element could be a majority. Thus if N is odd, ignore the last element. Run the algorithm as before. If no majority element emerges, then return the N^{th} element as a candidate.

(c) The running time is $O(N)$, and satisfies $T(N) = T(N/2) + O(N)$.

(d) One copy of the original needs to be saved. After this, the B array, and indeed the recursion, can be avoided by placing each B_i in the A array. The difference is that the original recursive strategy implies that $O(\log N)$ arrays are used; this guarantees only two copies.

2.27 Start from the top-right corner. With a comparison, either a match is found, we go left, or we go down. Therefore, the number of comparisons is linear.

2.28 **(a,c)** Find the two largest numbers in the array.

(b,d) Similar solutions; (b) is described here. The maximum difference is at least zero ($i \equiv j$), so that can be the initial value of the answer to beat. At any point in the algorithm, we have the current value j, and the current low point i. If $a[j] - a[i]$ is larger than the current best, update the best difference.

If $a[j]$ is less than $a[i]$, reset the current low point to i. Start with i at index 0, j at index 0. j just scans the array, so the running time is $O(N)$.

2.29 Otherwise, we could perform operations in parallel by cleverly encoding several integers into one. For instance, if A = 001, B = 101, C = 111, D = 100, we could add A and B at the same time as C and D by adding 00A00C + 00B00D. We could extend this to add N pairs of numbers at once in unit cost.

2.31 No. If $low = 1$, $high = 2$, then $mid = 1$, and the recursive call does not make progress.

2.33 No. As in Exercise 2.31, no progress is made.

2.34 See my textbook *Data Structures and Problem Solving using Java* for an explanation.

CHAPTER 3

Lists, Stacks, and Queues

3.1

```
public static <AnyType> void printLots(List<AnyType> L, List<Integer> P)
{
  Iterator<AnyType> iterL = L.iterator();
  Iterator<Integer> iterP = P.iterator();

  AnyType itemL=null;
  Integer itemP=0;
  int start = 0;

  while ( iterL.hasNext() && iterP.hasNext() )
  {
    itemP = iterP.next();

    System.out.println("Looking for position " + itemP);
    while ( start < itemP && iterL.hasNext() )
    {
      start++;
      itemL = iterL.next();
    }
    System.out.println( itemL );
  }
}
```

3.2 **(a)** For singly linked lists:

```
// beforeP is the cell before the two adjacent cells that are to be swapped.
// Error checks are omitted for clarity.

public static void swapWithNext( Node beforep )
{
  Node p, afterp;

  p = beforep.next;
  afterp = p.next;   // Both p and afterp assumed not null.

  p.next = afterp.next;
  beforep.next = afterp;
```

```
    afterp.next = p;
  }
```

(b) For doubly linked lists:

```
  // p and afterp are cells to be switched.  Error checks as before.

  public static void swapWithNext( Node p )
  {
    Node beforep, afterp;

    beforep = p.prev;
    afterp = p.next;

    p.next = afterp.next;
    beforep.next = afterp;
    afterp.next = p;
    p.next.prev = p;
    p.prev = afterp;
    afterp.prev = beforep;
  }
```

3.3

```
  public boolean contains( AnyType x )
  {
     Node<AnyType> p = beginMarker.next;
     while( p != endMarker && !(p.data.equals(x) ))
     {
           p = p.next;
     }

     return (p != endMarker);
  }
```

3.4

```
  public static <AnyType extends Comparable<? super AnyType>>
         void intersection(List<AnyType> L1, List<AnyType> L2,
                                             List<AnyType> Intersect)
  {

    ListIterator<AnyType> iterL1 = L1.listIterator();
    ListIterator<AnyType> iterL2 = L2.listIterator();

    AnyType itemL1=null, itemL2=null;

    // get first item in each list
    if ( iterL1.hasNext() && iterL2.hasNext() )
    {
      itemL1 = iterL1.next();
      itemL2 = iterL2.next();
    }
```

```
  while ( itemL1 != null && itemL2 != null )
  {
    int compareResult = itemL1.compareTo(itemL2);

    if ( compareResult == 0 )
    {
      Intersect.add(itemL1);
      itemL1 = iterL1.hasNext()?iterL1.next():null;
      itemL2 = iterL2.hasNext()?iterL2.next():null;
    }
    else if ( compareResult < 0 )
    {
      itemL1 = iterL1.hasNext()?iterL1.next():null;
    }
    else
    {
      itemL2 = iterL2.hasNext()?iterL2.next():null;
    }
  }
}
```

3.5

```
public static <AnyType extends Comparable<? super AnyType>>
       void union(List<AnyType> L1, List<AnyType> L2,
                                          List<AnyType> Result)
{

  ListIterator<AnyType> iterL1 = L1.listIterator();
  ListIterator<AnyType> iterL2 = L2.listIterator();

  AnyType itemL1=null, itemL2=null;

  // get first item in each list
  if ( iterL1.hasNext() && iterL2.hasNext() )
  {
    itemL1 = iterL1.next();
    itemL2 = iterL2.next();
  }

  while ( itemL1 != null && itemL2 != null )
  {
    int compareResult = itemL1.compareTo(itemL2);

    if ( compareResult == 0 )
    {
      Result.add(itemL1);
      itemL1 = iterL1.hasNext()?iterL1.next():null;
      itemL2 = iterL2.hasNext()?iterL2.next():null;
    }
```

```
      else if ( compareResult < 0 )
      {
        Result.add(itemL1);
        itemL1 = iterL1.hasNext()?iterL1.next():null;
      }
      else
      {
        Result.add(itemL2);
        itemL2 = iterL2.hasNext()?iterL2.next():null;
      }
    }
  }
```

3.6 A basic algorithm is to iterate through the list, removing every Mth item. This can be improved by two observations. The first is that an item M distance away is the same as an item that is only M mod N away. This is useful when M is large enough to cause iteration through the list multiple times. The second is that an item M distance away in the forward direction is the same as an item $(M - N)$ away in the backwards direction. This improvement is useful when M is more than $N/2$ (half the list). The solution shown below uses these two observations. Note that the list size changes as items are removed. The worst case running time is $O(N \min(M, N))$, though with the improvements given the algorithm might be significantly faster. If $M = 1$, the algorithm is linear.

```
public static void pass(int m, int n)
{
  int i, j, mPrime, numLeft;

  ArrayList<Integer> L = new ArrayList<Integer>();

  for (i=1; i<=n; i++)
    L.add(i);

  ListIterator<Integer> iter = L.listIterator();
  Integer item=0;

  numLeft = n;
  mPrime = m % n;

  for (i=0; i<n; i++)
  {
    mPrime = m % numLeft;
    if (mPrime <= numLeft/2)
    {
      if (iter.hasNext())
            item = iter.next();
      for (j=0; j<mPrime; j++)
      {
        if (!iter.hasNext())
          iter = L.listIterator();
```

```
        item = iter.next();
      }
    }
    else
    {
      for (j=0; j<numLeft-mPrime; j++)
      {
        if (!iter.hasPrevious())
          iter = L.listIterator(L.size());

        item = iter.previous();
      }
    }
    System.out.print("Removed " + item + " ");
    iter.remove();
    if (!iter.hasNext())
      iter = L.listIterator();

    System.out.println();
    for (Integer x:L)
       System.out.print(x + " ");
    System.out.println();
    numLeft--;
  }

  System.out.println();
}
```

3.7 $O(N^2)$. The trim method reduces the size of the array, requiring each add to resize it. The resize takes $O(N)$ time, and there are $O(N)$ calls.

3.8 **(a)** Because the remove call changes the size, which would affect the loop.

(b) $O(N^2)$. Each remove from the beginning requires moving all elements forward, which takes $O(N)$ time.

(c) $O(N)$. Each remove can be done in $O(1)$ time.

(d) No. Since it is always the first element being removed, finding the position does not take time, thus an iterator would not help.

3.9

```
public void addAll( Iterable<? extends AnyType> items )
{
  Iterator<? extends AnyType> iter = items.iterator();
  while ( iter.hasNext() )
  {
    add(iter.next());
  }
}
```

This runs in $O(N)$ time, where N is the size of the items collection.

3.10

```
public void removeAll( Iterable<? extends AnyType> items )
{
  AnyType item, element;
  Iterator<? extends AnyType> iterItems = items.iterator();

  while ( iterItems.hasNext ( ) )
  {
    item = iterItems.next();
    Iterator<? extends AnyType> iterList = iterator();
    while ( iterList.hasNext ( ) )
    {
      element = iterList.next();
      if ( element.equals(item) )
        iterList.remove();
    }
  }
 }
```

This runs in $O(MN)$, where M is the size of the items, and N is the size of the list.

3.11

```
import java.util.*;

public class SingleList
{
  SingleList()
  { init(); }

  boolean add( Object x )
  {
    if (contains(x))
      return false;
    else
    {
      Node<Object> p = new Node<Object>(x);
      p.next = head.next;
      head.next = p;
      theSize++;
    }
    return true;
  }

  boolean remove(Object x)
  {
    if (!contains(x))
      return false;
    else
    {
      Node<Object> p = head.next;
```

```
    Node<Object> trailer = head;
    while (!p.data.equals(x))
    {
      trailer = p;
      p = p.next;
    }
    trailer.next = p.next;
    theSize--;
  }
  return true;
}

int size()
{ return theSize; }

void print()
{
  Node<Object> p = head.next;
  while (p != null)
  {
    System.out.print(p.data + " ");
    p = p.next;
  }
  System.out.println();
}

boolean contains( Object x )
{
  Node<Object> p = head.next;
  while (p != null)
  {
    if (x.equals(p.data))
      return true;
    else
      p = p.next;
  }
  return false;
}

void init()
{
  theSize = 0;
  head = new Node<Object>();
  head.next = null;
}

private Node<Object> head;
private int theSize;
```

```
  private class Node<Object>
  {
    Node()
    {
      this(null, null);
    }
    Node(Object d)
    {
      this(d, null);
    }
    Node(Object d, Node n)
    {
      data = d;
      next = n;
    }
    Object data;
    Node next;
  }
}
```

3.12

```
import java.util.*;

public class SingleListSorted
{
  SingleListSorted()
  { init(); }

  boolean add( Comparable x )
  {
    if (contains(x))
      return false;
    else
    {
      Node<Comparable> p = head.next;
      Node<Comparable> trailer = head;
      while (p != null && p.data.compareTo(x) < 0)
      {
        trailer = p;
        p = p.next;
      }

      trailer.next = new Node<Comparable>(x);
      trailer.next.next = p;
      theSize++;
    }
    return true;
  }
```

```
boolean remove(Comparable x)
{
  if (!contains(x))
    return false;
  else
  {
    Node<Comparable> p = head.next;
    Node<Comparable> trailer = head;
    while (!p.data.equals(x))
    {
      trailer = p;
      p = p.next;
    }
    trailer.next = p.next;
    theSize--;
  }
  return true;
}

int size()
{ return theSize; }

void print()
{
  Node<Comparable> p = head.next;
  while (p != null)
  {
    System.out.print(p.data + " ");
    p = p.next;
  }
  System.out.println();
}

boolean contains( Comparable x )
{
  Node<Comparable> p = head.next;
  while (p != null && p.data.compareTo(x) <= 0)
  {
    if (x.equals(p.data))
      return true;
    else
      p = p.next;
  }
  return false;
}

void init()
{
```

```
    theSize = 0;
    head = new Node<Comparable>();
    head.next = null;
  }

  private Node<Comparable> head;
  private int theSize;

  private class Node<Comparable>
  {
    Node()
    {
      this(null, null);
    }
    Node(Comparable d)
    {
      this(d, null);
    }
    Node(Comparable d, Node n)
    {
      data = d;
      next = n;
    }
    Comparable data;
    Node next;
  }
}
```

3.13

```
public java.util.Iterator<AnyType> iterator()
{ return new ArrayListIterator( ); }

public java.util.ListIterator<AnyType> listIterator()
{ return new ArrayListIterator( ); }

private class ArrayListIterator implements java.util.ListIterator<AnyType>
{
   private int current = 0;
   boolean backwards = false;

   public boolean hasNext()
   { return current < size(); }

   public AnyType next()
   {
      if ( !hasNext() )
         throw new java.util.NoSuchElementException();
      backwards = false;
      return theItems[ current++ ];
   }
```

```
    public boolean hasPrevious()
    { return current > 0; }

    public AnyType previous()
    {
       if ( !hasPrevious() )
          throw new java.util.NoSuchElementException();
       backwards = true;
       return theItems[ --current ];
    }

    public void add( AnyType x )
    { MyArrayList.this.add( current++, x ); }

    public void remove()
    {
        if (backwards)
           MyArrayList.this.remove( current-- );
        else
           MyArrayList.this.remove( --current );
    }

    public void set( AnyType newVal )
    { MyArrayList.this.set( current, newVal ); }

    public int nextIndex()
    {
       throw new java.lang.UnsupportedOperationException();
    }

    public int previousIndex()
    {
       throw new java.lang.UnsupportedOperationException();
    }

}
```

3.14

```
public java.util.Iterator<AnyType> iterator()
{
  return new LinkedListIterator( );
}

public java.util.ListIterator<AnyType> listIterator()
{
  return new LinkedListIterator( );
}

private class LinkedListIterator implements java.util.ListIterator<AnyType>
{
```

```
private Node<AnyType> current = beginMarker.next;
private int expectedModCount = modCount;
private boolean okToRemove = false;

public boolean hasNext()
{  return current != endMarker; }

public AnyType next()
{
  if( modCount != expectedModCount )
      throw new java.util.ConcurrentModificationException( );
  if( !hasNext( ) )
      throw new java.util.NoSuchElementException( );

  AnyType nextItem = current.data;
  current = current.next;
  okToRemove = true;
  return nextItem;
}

public boolean hasPrevious()
{ return current.prev != beginMarker; }

public AnyType previous()
{
  if ( modCount != expectedModCount )
    throw new java.util.ConcurrentModificationException();
  if ( !hasPrevious() )
    throw new java.util.NoSuchElementException();

  current = current.prev;
  AnyType previousItem = current.data;
  okToRemove = true;
  return previousItem;
}

public void add( AnyType x )
{
  if ( modCount != expectedModCount )
    throw new java.util.ConcurrentModificationException();

  MyLinkedList.this.addBefore( current.next, x );
}

public void remove()
{
```

```
      if( modCount != expectedModCount )
        throw new java.util.ConcurrentModificationException( );
      if( !okToRemove )
        throw new IllegalStateException( );

      MyLinkedList.this.remove( current.prev );
      okToRemove = false;
    }

    public void set( AnyType newVal )
    {
      if ( modCount != expectedModCount )
         throw new java.util.ConcurrentModificationException();

      MyLinkedList.this.set( current.next, newVal );
    }

    public int nextIndex()
    {
      throw new java.lang.UnsupportedOperationException();
    }

    public int previousIndex()
    {
      throw new java.lang.UnsupportedOperationException();
    }

  }


// and change MyLinkedList as follows:

  public AnyType set( int idx, AnyType newVal )
  {   return set( getNode( idx ), newVal ); }

  private AnyType set( Node<AnyType> p, AnyType newVal )
  {
     AnyType oldVal = p.data;
     p.data = newVal;
     return oldVal;
  }
```

3.16

```
  Iterator<AnyType> reverseIterator()
  { return new ArrayListReverseIterator( ); }

  private class ArrayListReverseIterator implements java.util.Iterator<AnyType>
```

```
{
  private int current = size( ) - 1;

  public boolean hasNext()
  { return current >= 0; }

  public AnyType next()
  {
    if ( !hasNext() )
       throw new java.util.NoSuchElementException();
    return theItems[ current-- ];
  }

  public void remove()
  { MyArrayList.this.remove( --current ); }

}
```

3.18

```
public void addFirst( AnyType x )
{ addBefore( beginMarker.next, x ); }

public void addLast( AnyType x )
{ addBefore( endMarker, x ); }

public void removeFirst( )
{ remove( beginMarker.next ); }

public void removeLast( )
{ remove( endMarker.prev ); }

public AnyType getFirst( )
{ return get( 0 ); }

public AnyType getLast( )
{ return get( size( ) - 1 ); }
```

3.19 Without head or tail nodes the operations of inserting and deleting from the end becomes an $O(N)$ operation where the N is the number of elements in the list. The algorithm must walk down the list before inserting at the end. Without the head node insert needs a special case to account for when something is inserted before the first node.

3.20 **(a)** The advantages are that it is simpler to code, and there is a possible saving if deleted keys are subsequently reinserted (in the same place). The disadvantage is that it uses more space, because each cell needs an extra bit (which is typically a byte), and unused cells are not freed.

3.22 The following function evaluates a postfix expression, using +, −, ∗, /, and ^ ending in =. It requires spaces between all operators and =.

```
static double evalPostFix()
{
  Stack<Double> s = new Stack<Double>();
  String token;
  Double a, b, result=0.0;
  boolean isNumber;

  Scanner sc = new Scanner(System.in);
  token = sc.next();
  while (token.charAt(0) != '=')
  {
    try
    {
      isNumber = true;
      result = Double.parseDouble(token);
    }
    catch (Exception e)
    {
      isNumber = false;
    }
    if (isNumber)
      s.push(result);
    else
    {
      switch (token.charAt(0))
      {
        case '+': a = s.pop(); b = s.pop();
              s.push(a+b); break;
        case '-': a = s.pop(); b = s.pop();
              s.push(a-b); break;
        case '*': a = s.pop(); b = s.pop();
              s.push(a*b); break;
        case '/': a = s.pop(); b = s.pop();
              s.push(a/b); break;
        case '^': a = s.pop(); b = s.pop();
              s.push(Math.exp(a*Math.log(b)));
              break;
      }
    }

    token = sc.next();
  }

  return s.peek();
}
```

3.23 **(a, b)** This function will read in from standard input an infix expression of single lower case characters and the operators +, −, /, ∗, ^, and ( ), and outputs a postfix expression.

```
static void InFixToPostFix()
{
  Stack<Character> s = new Stack<Character>();
  String expression;
  Character token;
  int i=0;

  Scanner sc = new Scanner(System.in);
  expression = sc.next();

  while ((token = expression.charAt(i++)) != '=')
  {
    if (token >= 'a' && token <= 'z')
       System.out.print(token + " ");
    else
    {
       switch (token)
       {
        case ')': while ( !s.empty() && s.peek() != '(' )
                  { System.out.print(s.pop() + " "); }
                  s.pop();
                  break;
        case '(': s.push(token);
                  break;
        case '^': while ( !s.empty() && !(s.peek() == '^' ||
                                          s.peek() == '(') )
                  { System.out.print(s.pop()); }
                  s.push(token);
                  break;
        case '*':
        case '/': while ( !s.empty() && s.peek() != '+' &&
                         s.peek() != '-' && s.peek() != '(' )
                  { System.out.print(s.pop()); }
                  s.push(token);
                  break;
        case '+':
        case '-': while ( !s.empty() && s.peek() != '(' )
                  { System.out.print(s.pop() + " "); }
                  s.push(token);
                  break;
       }
    }
  }

  while (!s.empty())
```

```
    { System.out.print(s.pop()); }
    System.out.println();
  }
```

3.24 Two stacks can be implemented in an array by having one grow from the low end of the array up, and the other from the high end down.

3.25 **(a)** Let E be our extended stack. We will implement E with two stacks. One stack, which we'll call S, is used to keep track of the *push* and *pop* operations, and the other M, keeps track of the minimum. To implement *E.push* (x), we perform *S.push* (x). If x is smaller than or equal to the top element in stack M, then we also perform *M.push* (x). To implement *E.pop*( ) we perform *S.pop*( ). If x is equal to the top element in stack M, then we also *M.pop*( ). *E.findMin*( ) is performed by examining the top of M. All these operations are clearly $O(1)$.

(b) This result follows from a theorem in Chapter 7 that shows that sorting must take $\Omega(N \log N)$ time. $O(N)$ operations in the repertoire, including *deleteMin*, would be sufficient to sort.

3.26 Three stacks can be implemented by having one grow from the bottom up, another from the top down and a third somewhere in the middle growing in some (arbitrary) direction. If the third stack collides with either of the other two, it needs to be moved. A reasonable strategy is to move it so that its center (at the time of the move) is halfway between the tops of the other two stacks.

3.27 Stack space will not run out because only 49 calls will be stacked. However, the running time is exponential, as shown in Chapter 2, and thus the routine will not terminate in a reasonable amount of time.

3.28 Since the `LinkedList` class supports adding and removing from the first and end of the list, the `Deque` class shown below simply wraps these operations.

```
public class Deque<AnyType>
{
  Deque()
  { L = new LinkedList<AnyType>(); }

  void push(AnyType x)
  { L.addFirst(x); }

  AnyType pop()
  { return L.removeFirst(); }

  void inject(AnyType x)
  { L.addLast(x); }

  AnyType eject()
  { return L.removeLast(); }

  LinkedList<AnyType> L;
}
```

3.29 Reversal of a linked list can be done recursively using a stack, but this requires $O(N)$ extra space. The following solution is similar to strategies employed in garbage collection algorithms. At the top of the while loop, the list from the start to *previousPos* is already reversed, whereas the rest of the list,

from *currentPos* to the end is normal. This algorithm uses only constant extra space. This solution reverses the list which can then be printed in reverse order.

```
void reverseList()
{
  Node currentPos, nextPos, previousPos;

  previousPos = null;
  currentPos = head.next;  // skip header node
  nextPos = currentPos.next;

  while( nextPos != null)
  {
    currentPos.next = previousPos;
    previousPos = currentPos;
    currentPos = nextPos;
    nextPos = nextPos.next;
  }

  currentPos.next = previousPos;

  head.next = currentPos;
}
```

3.30 **(c)** This follows well-known statistical theorems. See Sleator and Tarjan's paper in Chapter 11 for references.

3.31

```
public class SingleStack<AnyType>
{

  SingleStack()
  {
    head = null;
  }

  void push(AnyType x)
  {
    Node<AnyType> p = new Node<AnyType>(x, head);
    head = p;
  }

  AnyType top()
  {
    return head.data;
  }

  void pop()
  {
    head = head.next;
  }
```

```
  private class Node<AnyType>
  {
    Node()
    { this(null, null); }
    Node(AnyType x)
    { this(x, null); }
    Node(AnyType x, Node p)
    {
      data = x;
      next = p;
    }
    AnyType data;
    Node next;
  }

  private Node<AnyType> head;

}
```

3.32

```
public class SingleQueue<AnyType>
{
  SingleQueue()
  {
    front = null;
    rear = null;
  }

  void enqueue(AnyType x)
  {
    Node<AnyType> p = new Node<AnyType>(x, null);
    if (rear != null)
       rear = rear.next = p;
    else
       front = rear = p;
  }

  AnyType dequeue()
  {
    AnyType temp = front.data;
    Node<AnyType> p = front;
    if (front.next == null)  // only 1 node
       front = rear = null;
    else
       front = front.next;

    return temp;
  }

  private class Node<AnyType>
```

```
  {
    Node()
    { this(null, null); }
    Node(AnyType x)
    { this(x, null); }
    Node(AnyType x, Node p)
    {
      data = x;
      next = p;
    }
    AnyType data;
    Node next;
  }

  private Node<AnyType> front, rear;

}
```

3.33

```
import java.util.*;

public class SingleQueueArray<AnyType>
{

  SingleQueueArray()
  { this(101); }  // note:  actually holds one less than given size

  SingleQueueArray(int s)
  {
    maxSize = s;
    front = 0;
    rear = 0;
    elements = new ArrayList<AnyType>(maxSize);
  }

  void enqueue(AnyType x)
  {
    if ( !full() )
    {
      if (elements.size() < maxSize)  // add elements until size is reached
        elements.add(x);
      else
        elements.set(rear, x);  // after size is reached, use set

      rear = (rear + 1) % maxSize;
    }
  }
```

```
  AnyType dequeue()
  {
    AnyType temp=null;

    if ( !empty() )
    {
      temp = elements.get(front);
      front = (front+1) % maxSize;
    }

    return temp;
  }

  boolean empty()
  { return front == rear; }

  boolean full()
  { return (rear + 1) % maxSize == front; }


  private int front, rear;
  private int maxSize;
  private ArrayList<AnyType> elements;

}
```

3.34 **(b)** Use two iterators p and q, both initially at the start of the list. Advance p one step at a time, and q two steps at a time. If q reaches the end there is no cycle; otherwise, p and q will eventually catch up to each other in the middle of the cycle.

3.35 **(a)** Does not work in constant time for insertions at the end.

(b) Because of the circularity, we can access the front item in constant time, so this works.

3.36 Copy the value of the item in the next node (that is, the node that follows the referenced node) into the current node (that is, the node being referenced). Then do a deletion of the next node.

3.37 **(a)** Add a copy of the node in position p after position p; then change the value stored in position p to x.

(b) Set `p.data = p.next.data` and set `p.next = p.next.next`. Note that the tail node guarantees that there is always a next node.

CHAPTER 4

Trees

4.1 **(a)** A.
(b) G, H, I, L, M, and K.

4.2 For node B:
(a) A.
(b) D and E.
(c) C.
(d) 1.
(e) 3.

4.3 4.

4.4 There are N nodes. Each node has two links, so there are $2N$ links. Each node but the root has one incoming link from its parent, which accounts for $N - 1$ links. The rest are *null*.

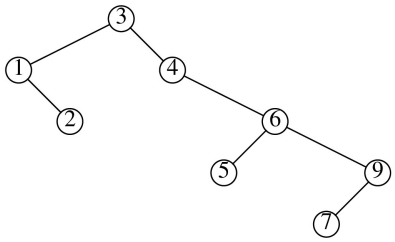
4.5 Proof is by induction. The theorem is trivially true for $h = 0$. Assume true for $h = 1, 2, \ldots, k$. A tree of height $k + 1$ can have two subtrees of height at most k. These can have at most $2^{k+1} - 1$ nodes each by the induction hypothesis. These $2^{k+2} - 2$ nodes plus the root prove the theorem for height $k + 1$ and hence for all heights.

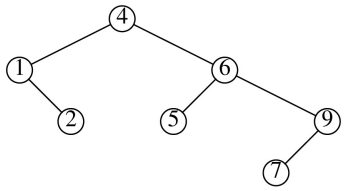
4.6 This can be shown by induction. Alternatively, let N = number of nodes, F = number of full nodes, L = number of leaves, and H = number of half nodes (nodes with one child). Clearly, $N = F + H + L$. Further, $2F + H = N - 1$ (see Exercise 4.4). Subtracting yields $L - F = 1$.

4.7 This can be shown by induction. In a tree with no nodes, the sum is zero, and in a one-node tree, the root is a leaf at depth zero, so the claim is true. Suppose the theorem is true for all trees with at most k nodes. Consider any tree with $k + 1$ nodes. Such a tree consists of an i node left subtree and a $k - i$ node right subtree. By the inductive hypothesis, the sum for the left subtree leaves is at most one with respect to the left tree root. Because all leaves are one deeper with respect to the original tree than with respect to the subtree, the sum is at most 1/2 with respect to the root. Similar logic implies that the sum for leaves in the right subtree is at most 1/2, proving the theorem. The equality is true if and only if there are no nodes with one child. If there is a node with one child, the equality cannot be true because adding the second child would increase the sum to higher than 1. If no nodes have one child, then we can find and remove two sibling leaves, creating a new tree. It is easy to see that this new tree has the same sum as the old. Applying this step repeatedly, we arrive at a single node, whose sum is 1. Thus the original tree had sum 1.

4.8 **(a)** - * * a b + c d e.
(b) ((a * b) * (c + d)) - e.
(c) a b * c d + * e -.

4.9





4.10

```
import java.io.*;

public class FileList
{
  public void list(File f)
  {
    list(f, 0);
  }

  public void list(File f, int depth)
  {
    printName(f, depth);
    if (f.isDirectory())
    {
        File [] files = f.listFiles();
        for (File i : files)
            list(i, depth+1);
    }
  }

  void printName(File f, int depth)
  {
    String name = f.getName();
    for (int i=0; i<depth; i++)
      System.out.print("   ");
    if (f.isDirectory())
      System.out.println("Dir: " + name);
    else
      System.out.println(f.getName() + "" + f.length());
  }

  public static void main(String args[])
  {
    FileList L = new FileList();
    File f = new File("C:\ ");
    L.list(f);
  }
}
```

4.11

```
import java.util.*;

class UnderflowException extends Exception { };


public class MyTreeSet<AnyType extends Comparable<? super AnyType>>
{
  private static class BinaryNode<AnyType>
  {
    BinaryNode( AnyType theElement )
    { this( theElement, null, null, null ); }

    BinaryNode( AnyType theElement,
                BinaryNode<AnyType> lt, BinaryNode<AnyType> rt,
                BinaryNode<AnyType> pt )
    { element = theElement; left = lt; right = rt; parent = pt; }

    AnyType element;
    BinaryNode<AnyType> left;
    BinaryNode<AnyType> right;
    BinaryNode<AnyType> parent;
  }

  public java.util.Iterator<AnyType> iterator()
  {
    return new MyTreeSetIterator( );
  }

  private class MyTreeSetIterator implements java.util.Iterator<AnyType>
  {
    private BinaryNode<AnyType> current = findMin(root);
    private BinaryNode<AnyType> previous;
    private int expectedModCount = modCount;
    private boolean okToRemove = false;
    private boolean atEnd = false;

    public boolean hasNext()
    {  return !atEnd; }

    public AnyType next()
    {
      if( modCount != expectedModCount )
          throw new java.util.ConcurrentModificationException( );
      if( !hasNext( ) )
          throw new java.util.NoSuchElementException( );

      AnyType nextItem = current.element;
```

```
        previous = current;

        //  if there is a right child, next node is min in right subtree
        if (current.right != null)
        {
          current = findMin(current.right);
        }
        else
        {
          // else, find ancestor that it is left of
            BinaryNode<AnyType> child = current;
            current = current.parent;
            while ( current != null && current.left != child)
            {
               child = current;
               current = current.parent;
            }
            if (current == null)
               atEnd = true;
        }

        okToRemove = true;

        return nextItem;
      }

      public void remove()
      {
        if( modCount != expectedModCount )
           throw new java.util.ConcurrentModificationException( );
        if( !okToRemove )
           throw new IllegalStateException( );

        MyTreeSet.this.remove( previous.element );
        okToRemove = false;
      }

    }

    public MyTreeSet()
    { root = null; }

    public void makeEmpty()
    {
      modCount++;
      root = null;
    }
```

```
public boolean isEmpty()
{ return root == null; }

public boolean contains( AnyType x )
{ return contains( x, root ); }

public AnyType findMin() throws UnderflowException
{
  if ( isEmpty() )
     throw new UnderflowException();
  else
     return findMin( root ).element;
}

public AnyType findMax() throws UnderflowException
{
  if ( isEmpty() )
     throw new UnderflowException();
  else
     return findMax( root ).element;
}

public void insert( AnyType x )
{ root = insert( x, root, null ); }

public void remove( AnyType x )
{ root = remove( x, root ); }

public void printTree()
{
   if ( isEmpty() )
      System.out.println( "Empty tree" );
   else
      printTree( root );
  }

  private void printTree( BinaryNode<AnyType> t )
  {
    if ( t != null )
    {
      printTree( t.left );
      System.out.println( t.element );
      printTree( t.right );
    }
  }

private boolean contains( AnyType x, BinaryNode<AnyType> t )
{
```

```
    if ( t == null )
        return false;

    int compareResult = x.compareTo( t.element );

    if ( compareResult < 0)
        return contains( x, t.left );
    else if ( compareResult > 0)
        return contains( x, t.right );
    else
        return true; // match

}

private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
    if ( t == null )
        return null;
    else if ( t.left == null )
        return t;
    return findMin( t.left );
}

private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
{
    if ( t == null )
        return null;
    else if ( t.right == null )
        return t;
    return findMax( t.right );
}

private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t,
                                    BinaryNode<AnyType> pt )
{
    if ( t == null )
    {
        modCount++;
        return new BinaryNode<AnyType>( x, null, null, pt);
    }

    int compareResult = x.compareTo( t.element );

    if ( compareResult < 0)
        t.left = insert( x, t.left, t );
    else if ( compareResult > 0)
        t.right = insert( x, t.right, t );
    else
```

```
      ; // duplicate

    return t;
  }

  private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
  {
    if ( t == null )
      return t; // not found

    int compareResult = x.compareTo( t.element );

    if ( compareResult < 0)
      t.left = remove( x, t.left );
    else if ( compareResult > 0)
      t.right = remove( x, t.right );
    else if ( t.left != null && t.right != null ) // two children
    {
      t.element = findMin( t.right ).element;
      t.right = remove( t.element, t.right );
    }
    else
    {
      modCount++;
      BinaryNode<AnyType> oneChild;
      oneChild = ( t.left != null ) ? t.left : t.right;
      oneChild.parent = t.parent;  // update parent link
      t = oneChild;
    }

    return t;
  }

  private BinaryNode<AnyType> root;
  int modCount = 0;
}
```

4.13 This solution does not use header and tail nodes.

```
import java.util.*;

class UnderflowException extends Exception { };


public class MyTreeSet2<AnyType extends Comparable<? super AnyType>>
{
  private static class BinaryNode<AnyType>
  {
    BinaryNode( AnyType theElement )
```

```
    { this( theElement, null, null, null, null ); }

    BinaryNode( AnyType theElement,
                BinaryNode<AnyType> lt, BinaryNode<AnyType> rt,
                BinaryNode<AnyType> nt, BinaryNode<AnyType> pv )
    { element = theElement; left = lt; right = rt; next = nt; prev = pv; }

    AnyType element;
    BinaryNode<AnyType> left;
    BinaryNode<AnyType> right;
    BinaryNode<AnyType> next;
    BinaryNode<AnyType> prev;
  }

  public java.util.Iterator<AnyType> iterator()
  {
    return new MyTreeSet2Iterator( );
  }

  private class MyTreeSet2Iterator implements java.util.Iterator<AnyType>
  {
    private BinaryNode<AnyType> current = findMin(root);
    private BinaryNode<AnyType> previous;
    private int expectedModCount = modCount;
    private boolean okToRemove = false;
    private boolean atEnd = false;

    public boolean hasNext()
    {  return !atEnd; }

    public AnyType next()
    {
      if( modCount != expectedModCount )
          throw new java.util.ConcurrentModificationException( );
      if( !hasNext( ) )
          throw new java.util.NoSuchElementException( );

      AnyType nextItem = current.element;
      previous = current;

      current = current.next;

      if (current == null)
          atEnd = true;

      okToRemove = true;

      return nextItem;
    }
```

```
  public void remove()
  {
    if( modCount != expectedModCount )
       throw new java.util.ConcurrentModificationException( );
    if( !okToRemove )
       throw new IllegalStateException( );

    MyTreeSet2.this.remove( previous.element );
    okToRemove = false;
  }

}

public MyTreeSet2()
{ root = null; }

public void makeEmpty()
{
  modCount++;
  root = null;
}

public boolean isEmpty()
{ return root == null; }

public boolean contains( AnyType x )
{ return contains( x, root ); }

public AnyType findMin() throws UnderflowException
{
  if ( isEmpty() )
     throw new UnderflowException();
  else
     return findMin( root ).element;
}

public AnyType findMax() throws UnderflowException
{
  if ( isEmpty() )
     throw new UnderflowException();
  else
     return findMax( root ).element;
}

public void insert( AnyType x )
{ root = insert( x, root, null, null ); }

public void remove( AnyType x )
{ root = remove( x, root ); }
```

```
public void printTree()
{
   if ( isEmpty() )
      System.out.println( "Empty tree" );
   else
      printTree( root );
  }

  private void printTree( BinaryNode<AnyType> t )
  {
    if ( t != null )
    {
      printTree( t.left );
      System.out.println( t.element );
      printTree( t.right );
    }
  }

private boolean contains( AnyType x, BinaryNode<AnyType> t )
{
  if ( t == null )
     return false;

  int compareResult = x.compareTo( t.element );

  if ( compareResult < 0)
     return contains( x, t.left );
  else if ( compareResult > 0)
     return contains( x, t.right );
  else
     return true; // match

}

private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
  if ( t == null )
     return null;
  else if ( t.left == null )
     return t;
  return findMin( t.left );
}

private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
{
  if ( t == null )
     return null;
  else if ( t.right == null )
```

```
    return t;
  return findMax( t.right );
}

private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t,
                 BinaryNode<AnyType> nt, BinaryNode<AnyType> pv )
{
  if ( t == null )
  {
      modCount++;
      BinaryNode<AnyType> newNode = new BinaryNode<AnyType>( x, null, null, nt, pv);
      if (nt != null)
      {
        nt.prev = newNode;
      }
      if (pv != null)
      {
        pv.next = newNode;
      }
    return newNode;
  }

  int compareResult = x.compareTo( t.element );

  if ( compareResult < 0)
    t.left = insert( x, t.left, t, pv );
  else if ( compareResult > 0)
  {
    t.right = insert( x, t.right, nt, t );
  }
  else
    ; // duplicate

  return t;
}

private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
{
  if ( t == null )
    return t; // not found

  int compareResult = x.compareTo( t.element );

  if ( compareResult < 0)
    t.left = remove( x, t.left );
  else if ( compareResult > 0)
    t.right = remove( x, t.right );
  else if ( t.left != null && t.right != null ) // two children
```

```
        {
            t.element = findMin( t.right ).element;
            t.right = remove( t.element, t.right );
        }
        else
        {
            modCount++;
            t.prev.next = t.next;  // update next and prev links
            t.next.prev = t.prev;
            t = ( t.left != null ) ? t.left : t.right;
        }

        return t;
    }

    private BinaryNode<AnyType> root;
    int modCount = 0;

}
```

4.14 **(a)** Keep a bit array B. If i is in the tree, then $B[i]$ is true; otherwise, it is false. Repeatedly generate random integers until an unused one is found. If there are N elements already in the tree, then $M - N$ are not, and the probability of finding one of these is $(M - N)/M$. Thus the expected number of trials is $M/(M - N) = \alpha/(\alpha - 1)$.

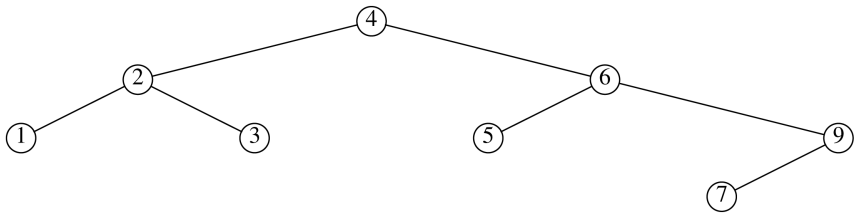
(b) To find an element that is in the tree, repeatedly generate random integers until an already-used integer is found. The probability of finding one is N/M, so the expected number of trials is $M/N = \alpha$.

(c) The total cost for one insert and one delete is $alpha/(\alpha - 1) + \alpha = 1 + \alpha + 1/(\alpha - 1)$. Setting $alpha = 2$ minimizes this cost.

4.18 **(a)** $N(0) = 1$, $N(1) = 2$, $N(h) = N(h - 1) + N(h - 2) + 1$.

(b) The heights are one less than the Fibonacci numbers.

4.19



4.20 It is easy to verify by hand that the claim is true for $1 \le k \le 3$. Suppose it is true for $k = 1, 2, 3, \ldots h$. Then after the first $2^h - 1$ insertions, 2^{h-1} is at the root, and the right subtree is a balanced tree containing $2^{h-1} + 1$ through $2^h - 1$. Each of the next 2^{h-1} insertions, namely, 2^h through $2^h + 2^{h-1} - 1$, insert a new maximum and get placed in the right subtree, eventually forming a perfectly balanced right subtree of height $h - 1$. This follows by the induction hypothesis because the right subtree may be viewed as being formed from the successive insertion of $2^{h-1} + 1$ through $2^h + 2^{h-1} - 1$. The next insertion forces an imbalance at the root, and thus a single rotation. It is easy

to check that this brings 2^h to the root and creates a perfectly balanced left subtree of height $h - 1$. The new key is attached to a perfectly balanced right subtree of height $h - 2$ as the last node in the right path. Thus the right subtree is exactly as if the nodes $2^h + 1$ through $2^h + 2^{h-1}$ were inserted in order. By the inductive hypothesis, the subsequent successive insertion of $2^h + 2^{h-1} + 1$ through $2^{h+1} - 1$ will create a perfectly balanced right subtree of height $h - 1$. Thus after the last insertion, both the left and the right subtrees are perfectly balanced, and of the same height, so the entire tree of $2^{h+1} - 1$ nodes is perfectly balanced (and has height h).

4.21 The two remaining functions are mirror images of the text procedures. Just switch *right* and *left* everywhere.

4.24 After applying the standard binary search tree deletion algorithm, nodes on the deletion path need to have their balance changed, and rotations may need to be performed. Unlike insertion, more than one node may need rotation.

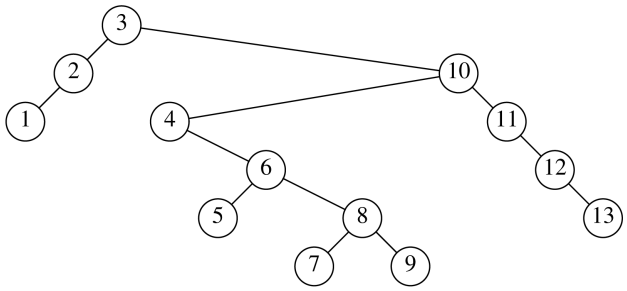
4.25 **(a)** $O(\log \log N)$.

(b) The minimum AVL tree of height 127 (8-byte ints range from -128 to 127). This is a huge tree.

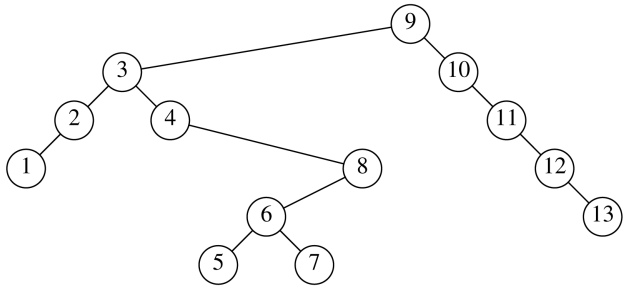
4.26

```
static AvlNode doubleRotateWithLeft( AvlNode k3 )
{
    AvlNode k1, k2;

    k1 = k3.left;
    k2 = k1.right;

    k1.right = k2.left;
    k3.left = k2.right;
    k2.left = k1;
    k2.right = k3;
    k1.height = max( height(k1.left), height(k1.right) ) + 1;
    k3.height = max( height(k3.left), height(k3.right) ) + 1;
    k2.height = max( k1. height, k3.height ) + 1;

    return k3;
}
```

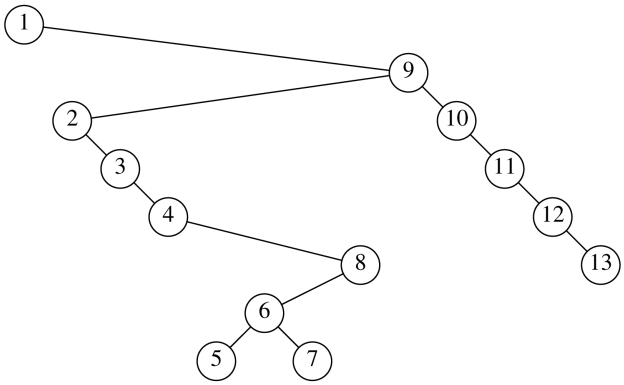
4.27 After accessing 3,



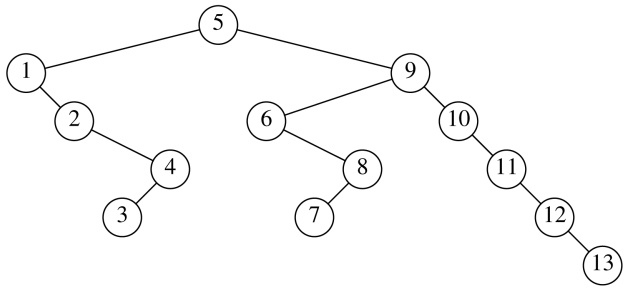
After accessing 9,



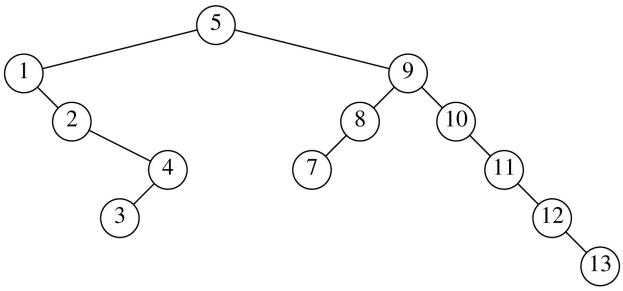
After accessing 1,



After accessing 5,



4.28



4.29 **(a)** An easy proof by induction.

4.31 **(a–c)** All of these routines take linear time.

```
static int countNodes( Node t )
{
  if (t == null)
     return 0;
  return 1 + countNodes(t.left) + countNodes(t.right);
}

static int countLeaves( Node t )
{
  if (t == null)
     return 0;
  else if ( t.left == null && t.right == null)
     return 1;
  return countLeaves(t.left) + countLeaves(t.right);
}

static int countFull( Node t )
{
  if (t == null)
     return 0;
  int tIsFull = ( t.left != null && t.right != null) ? 1 : 0;
  return tIsFull + countFull(t.left) + countFull(t.right);
}
```

4.32 Have the recursive routine return a triple that consists of a boolean (whether the tree is a BST) and the minimum and maximum items. Then a tree is a BST if it is empty or both subtrees are (recursively) BSTs, and the value in the node lies between the maximum of the left subtree and the minimum of the right subtrees. Coding details are omitted.

4.33

```
static Node removeLeaves( Node t )
{
  if (t == null || ( t.left == null && t.right == null ))
    return null;

  t.left = removeLeaves( t.left );
  t.right = removeLeaves( t.right );

  return t;
}
```

4.34 We assume the existence of a method `randInt(`*lower*`,` *upper*`)` that generates a uniform random integer in the appropriate closed interval.

```
static Node makeRandomTree( int lower, int upper )
{
    Node t = null;
    int randomValue;
```

```
        if ( lower <= upper )
            t = new Node( randomValue = randInt( lower, upper ),
                                        makeRandomTree( lower, randomValue - 1 ),
                                        makeRandomTree( randomValue + 1, upper ) );
        return t;
    }

    static Node makeRandomTree( int n )
    {
        return makeRandomTree( 1, n );
    }
```

4.35

```
    // LastNode[0] is the address containing the last value that was assigned to a node.
    // This is a standard trick to get a call by reference.
    static Node genTree( int height, int [] lastNode )
    {
        Node t = null;

        if ( height >= 0 )
        {
            t = new Node();
            t.left = genTree( height - 1, lastNode[0] );
            t.element = ++lastNode[0];
            t.right = genTree( height - 2, lastNode[0] );
        }
        return t;
    }

    static Node minAvlTree( int h)
    {
         int lastNodeAssigned[] = { 0 };
         return genTree( h, lastNodeAssigned );
    }
```

4.36 There are two obvious ways of solving this problem. One way mimics Exercise 4.34 by replacing *randInt(lower,upper)* with *(lower+upper) / 2*. This requires computing $2^{h+1} - 1$, which is not that difficult. The other mimics the previous exercise by noting that the heights of the subtrees are both $h - 1$.

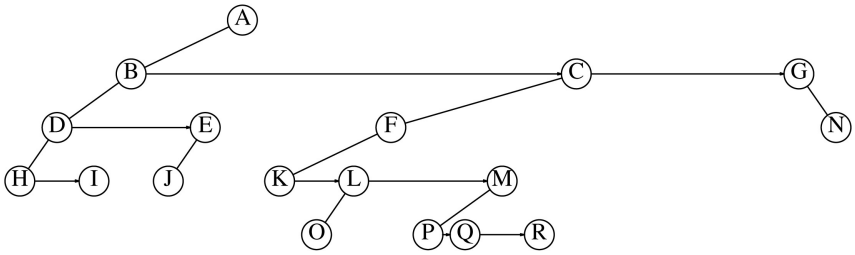
4.37 This is known as one-dimensional range searching. The time is $O(K)$ to perform the inorder traversal, if a significant number of nodes are found, and also proportional to the depth of the tree, if we get to some leaves (for instance, if no nodes are found). Since the average depth is $O(\log N)$, this gives an $O(K + \log N)$ average bound.

```
  static void printRange( Comparable lower, Comparable upper, BinaryNode t )
  {
    if( t != null )
    {
      if( lower.compareTo( t.element ) <= 0 )
        printRange( lower, upper, t.left );
      if( lower.compareTo( t.element ) <= 0 && t.element.compareTo( upper ) <= 0 )
```

```
            System.out.println( t.element );
          if( t.element.compareTo( upper ) <= 0 )
            printRange( lower, upper, t.right );
        }
    }
```

4.38 This exercise and Exercise 4.39 are likely programming assignments, so we do not provide code here.

4.44



4.46 The function shown here is clearly a linear time routine because in the worst case it does a traversal on both $t1$ and $t2$.

```
static boolean similar( Node t1, Node t2 )
{
    if( t1 == NULL || t2 == NULL )
        return t1 == NULL && t2 == NULL;
    return similar( t1.left, t2.left ) && similar( t1.right, t2.right );
}
```

4.48 The easiest solution is to compute, in linear time, the inorder numbers of the nodes in both trees. If the inorder number of the root of T_2 is x, then find x in T_1 and rotate it to the root. Recursively apply this strategy to the left and right subtrees of T_1 (by looking at the values in the root of T_2's left and right subtrees). If d_N is the depth of x, then the running time satisfies $T(N) = T(i) + T(N - i - 1) + d_N$, where i is the size of the left subtree. In the worst case, d_N is always $O(N)$, and i is always 0, so the worst-case running time is quadratic. Under the plausible assumption that all values of i are equally likely, then even if d_N is always $O(N)$, the average value of $T(N)$ is $O(N \log N)$. This is a common recurrence that was already formulated in the chapter and is solved in Chapter 7. Under the more reasonable assumption that d_N is typically logarithmic, then the running time is $O(N)$.

4.49 Add a data member to each node indicating the size of the tree it roots. This allows computation of its inorder traversal number.

4.50 **(a)** You need an extra bit for each thread.

(c) You can do tree traversals somewhat easier and without recursion. The disadvantage is that it reeks of old-style hacking.

4.51 **(a)** Both values are 0.

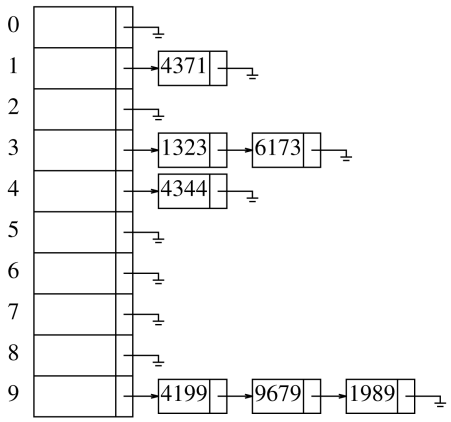
(b) The root contributes $(N - 2)/N$ full nodes on average, because the root is full as long as it does not contain the largest or smallest item. The remainder of the equation is the expected contribution of the subtrees.

(d) The average number of leaves is $(N + 1)/3$.

CHAPTER 5

Hashing

5.1 **(a)** On the assumption that we add collisions to the end of the list (which is the easier way if a hash table is being built by hand), the separate chaining hash table that results is shown here.



(b)

0	9679
1	4371
2	1989
3	1323
4	6173
5	4344
6	
7	
8	
9	4199

(c)

0	9679
1	4371
2	
3	1323
4	6173
5	4344
6	
7	
8	1989
9	4199

(d) 1989 cannot be inserted into the table because $hash_2(1989) = 6$, and the alternative locations 5, 1, 7, and 3 are already taken. The table at this point is as follows:

0	
1	4371
2	
3	1323
4	6173
5	9679
6	
7	4344
8	
9	4199

5.2 When rehashing, we choose a table size that is roughly twice as large and prime. In our case, the appropriate new table size is 19, with hash function $h(x) = x \pmod{19}$.

(a) Scanning down the separate chaining hash table, the new locations are 4371 in list 1, 1323 in list 12, 6173 in list 17, 4344 in list 12, 4199 in list 0, 9679 in list 8, and 1989 in list 13.

(b) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 14 (because both 12 and 13 are already occupied), and 4199 in bucket 0.

(c) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 16 (because both 12 and 13 are already occupied), and 4199 in bucket 0.

(d) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 15 because 12 is already occupied, and 4199 in bucket 0.

5.4 We must be careful not to rehash too often. Let p be the threshold (fraction of table size) at which we rehash to a smaller table. Then if the new table has size N, it contains $2pN$ elements. This table will require rehashing after either $2N - 2pN$ insertions or pN deletions. Balancing these costs suggests that a good choice is $p = 2/3$. For instance, suppose we have a table of size 300. If we rehash at 200 elements, then the new table size is $N = 150$, and we can do either 100 insertions or 100 deletions until a new rehash is required.

If we know that insertions are more frequent than deletions, then we might choose p to be somewhat larger. If p is too close to 1.0, however, then a sequence of a small number of deletions followed by insertions can cause frequent rehashing. In the worst case, if $p = 1.0$, then alternating deletions and insertions both require rehashing.

5.5 No; this does not take deletions into account.

5.6 **(b)** If the number of deleted cells is small, then we spend extra time looking for inactive cells that are not likely to be found. If the number of deleted cells is large, then we may get improvement.

5.7 In a good library implementation, the *length* method should be inlined.

5.8 Separate chaining hashing requires the use of links, which costs some memory, and the standard method of implementing calls on memory allocation routines, which typically are expensive. Linear probing is easily implemented, but performance degrades severely as the load factor increases because of primary clustering. Quadratic probing is only slightly more difficult to implement and gives good performance in practice. An insertion can fail if the table is half empty, but this is not likely. Even if it were, such an insertion would be so expensive that it wouldn't matter and would almost certainly point up a weakness in the hash function. Double hashing eliminates primary and secondary clustering, but the computation of a second hash function can be costly. Gonnet and Baeza-Yates [8] compare several hashing strategies; their results suggest that quadratic probing is the fastest method.

5.10 The old values would remain valid if the hashed values were less than the old table size.

5.11 Sorting the MN records and eliminating duplicates would require $O(MN \log MN)$ time using a standard sorting algorithm. If terms are merged by using a hash function, then the merging time is constant per term for a total of $O(MN)$. If the output polynomial is small and has only $O(M + N)$ terms, then it is easy to sort it in $O((M + N) \log(M + N))$ time, which is less than $O(MN)$. Thus the total is $O(MN)$. This bound is better because the model is less restrictive: Hashing is performing operations on the keys rather than just comparison between the keys. A similar bound can be obtained by using bucket sort instead of a standard sorting algorithm. Operations such as hashing are much more expensive than comparisons in practice, so this bound might not be an improvement. On the other hand, if the output polynomial is expected to have only $O(M + N)$ terms, then using a hash table saves a huge amount of space, since under these conditions, the hash table needs only $O(M + N)$ space.

Another method of implementing these operations is to use a search tree instead of a hash table; a balanced tree is required because elements are inserted in the tree with too much order. A splay tree might be particularly well suited for this type of a problem because it does well with sequential accesses. Comparing the different ways of solving the problem is a good programming assignment.

5.12 To each hash table slot, we can add an extra data member that we'll call *whereOnStack*, and we can keep an extra stack. When an insertion is first performed into a slot, we push the address (or number) of the slot onto the stack and set the *whereOnStack* data member to refer to the top of the stack. When we access a hash table slot, we check that *whereOnStack* refers to a valid part of the stack and that the entry in the (middle of the) stack that is referred to by the *whereOnStack* data member has that hash table slot as an address.

5.16 **(a)** The cost of an unsuccessful search equals the cost of an insertion.
(b)

$$I(\lambda) = \frac{1}{\lambda} \int_0^{\lambda} \left[\frac{1}{1-x} - x - \ln(1-x) \right] dx = 1 - \ln(1-\lambda) - \lambda/2$$

5.17

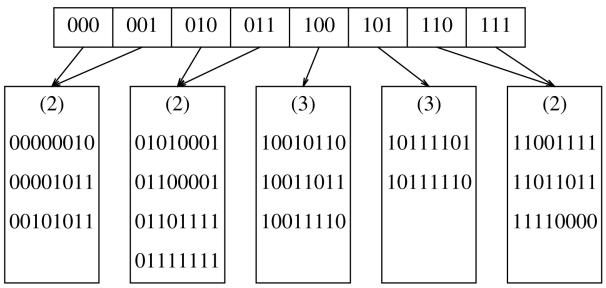
```
public class Map<KeyType, ValueType>
{
  public Map()
  {
    items = new QuadraticProbingHashTable<Entry<KeyType,ValueType>>();
  }

  public void put( KeyType key, ValueType val )
  {
    Entry<KeyType, ValueType> e = new Entry<KeyType, ValueType>(key, val);
    items.insert( e );
  }

  public ValueType get( KeyType key )
  {
    ValueType v = (ValueType) new Object();
    Entry<KeyType, ValueType> e = new Entry<KeyType, ValueType>(key, v);
    e = items.find(e);
    return e.val;
  }

  public boolean isEmpty()
  { return items.isEmpty(); }

  public void makeEmpty()
  { items.makeEmpty(); }

  private QuadraticProbingHashTable<Entry<KeyType,ValueType>> items;

  private static class Entry<KeyType, ValueType>
  {
    Entry( KeyType k, ValueType v )
    {
      key = k;
      val = v;
    }

    public int hashCode()
    {
      return key.hashCode();
    }

    //public boolean equals( Entry<KeyType, ValueType> rhs )
    public boolean equals( Object rhs )
    {
      return rhs instanceof Entry &&
                  key.equals( ((Entry<KeyType, ValueType>)rhs).key );
    }
```

```
    private KeyType key;
    private ValueType val;
  }
}
```

The QuadraticProbingHashTable.java file needs these additional methods (not given in the text) for the above solution to work:

```
public boolean isEmpty( )
{
  return currentSize == 0;
}

public AnyType find( AnyType x )
{
  int currentPos = findPos( x );
  return isActive( currentPos ) ? array[ currentPos ].element : null;
}
```

5.19

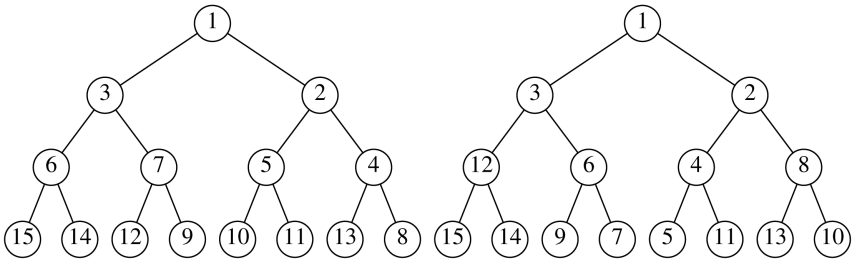


CHAPTER 6

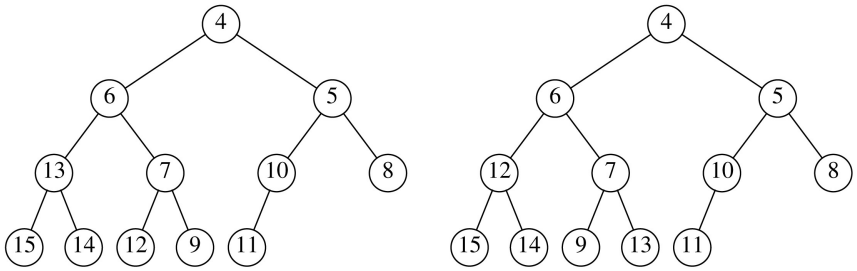
Priority Queues (Heaps)

6.1 Yes. When an element is inserted, we compare it to the current minimum and change the minimum if the new element is smaller. *deleteMin* operations are expensive in this scheme.

6.2



6.3 The result of three *deleteMins*, starting with both of the heaps in Exercise 6.2, is as follows:



6.4 **(a)** $4N$

(b) $O(N^2)$

(c) $O(N^{4.1})$

(d) $O(2^N)$

6.5

```
public void insert( AnyType x )
{
  if ( currentSize == array.length - 1 )
     enlargeArray( array.length * 2 + 1 );

  // Percolate up
  int hole = ++currentSize;
```

```
        for ( ; hole > 1 && x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2)
            array[ hole ] = array[ hole/2 ];
        array[ 0 ] = array[ hole ] = x;
    }
```

6.6 225. To see this, start with $i = 1$ and position at the root. Follow the path toward the last node, doubling i when taking a left child, and doubling i and adding one when taking a right child.

6.7 **(a)** We show that $H(N)$, which is the sum of the heights of nodes in a complete binary tree of N nodes, is $N - b(N)$, where $b(N)$ is the number of ones in the binary representation of N. Observe that for $N = 0$ and $N = 1$, the claim is true. Assume that it is true for values of k up to and including $N - 1$. Suppose the left and right subtrees have L and R nodes, respectively. Since the root has height $\lfloor \log N \rfloor$, we have

$$\begin{aligned} H(N) &= \lfloor \log N \rfloor + H(L) + H(R) \\ &= \lfloor \log N \rfloor + L - b(L) + R - b(R) \\ &= N - 1 + (\lfloor \log N \rfloor - b(L) - b(R)) \end{aligned}$$

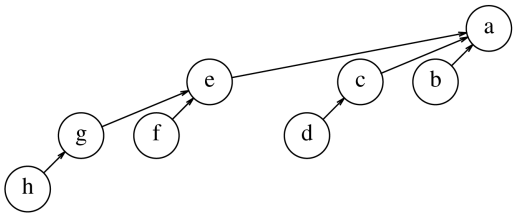
The second line follows from the inductive hypothesis, and the third follows because $L + R = N - 1$. Now the last node in the tree is in either the left subtree or the right subtree. If it is in the left subtree, then the right subtree is a perfect tree, and $b(R) = \lfloor \log N \rfloor - 1$. Further, the binary representation of N and L are identical, with the exception that the leading 10 in N becomes 1 in L. (For instance, if $N = 37 = \mathbf{10}0101$, $L = \mathbf{1}0101$.) It is clear that the second digit of N must be zero if the last node is in the left subtree. Thus in this case, $b(L) = b(N)$, and

$$H(N) = N - b(N)$$

If the last node is in the right subtree, then $b(L) = \lfloor \log N \rfloor$. The binary representation of R is identical to N, except that the leading 1 is not present. (For instance, if $N = 27 = \mathbf{1}01011$, $L = 01011$.) Thus $b(R) = b(N) - 1$, and again

$$H(N) = N - b(N)$$

(b) Run a single-elimination tournament among eight elements. This requires seven comparisons and generates ordering information indicated by the binomial tree shown here.



The eighth comparison is between b and c. If c is less than b, then b is made a child of c. Otherwise, both c and d are made children of b.

(c) A recursive strategy is used. Assume that $N = 2^k$. A binomial tree is built for the N elements as in part (b). The largest subtree of the root is then recursively converted into a binary heap of 2^{k-1} elements. The last element in the heap (which is the only one on an extra level) is then inserted into the binomial queue consisting of the remaining binomial trees, thus forming another binomial tree of 2^{k-1} elements. At that point, the root has a subtree that is a heap of $2^{k-1} - 1$ elements and another subtree that is a binomial tree of 2^{k-1} elements. Recursively convert that subtree into a heap; now

the whole structure is a binary heap. The running time for $N = 2^k$ satisfies $T(N) = 2T(N/2) + \log N$. The base case is $T(8) = 8$.

6.9 Let $D_1, D_2, \ldots, D_k$ be random variables representing the depth of the smallest, second smallest, and k^{th} smallest elements, respectively. We are interested in calculating $E(D_k)$. In what follows, we assume that the heap size N is one less than a power of two (that is, the bottom level is completely filled) but sufficiently large so that terms bounded by $O(1/N)$ are negligible. Without loss of generality, we may assume that the k^{th} smallest element is in the left subheap of the root. Let $p_{j,k}$ be the probability that this element is the j^{th} smallest element in the subheap.

Lemma.
For $k > 1$, $E(D_k) = \sum_{j=1}^{k-1} p_{j,k}(E(D_j) + 1)$.

Proof.
An element that is at depth d in the left subheap is at depth $d + 1$ in the entire subheap. Since $E(D_j + 1) = E(D_j) + 1$, the theorem follows.

Since by assumption, the bottom level of the heap is full, each of second, third, $\ldots$, $k - 1^{th}$ smallest elements are in the left subheap with probability of 0.5. (Technically, the probability should be $half - 1/(N - 1)$ of being in the right subheap and $half + 1/(N - 1)$ of being in the left, since we have already placed the k^{th} smallest in the right. Recall that we have assumed that terms of size $O(1/N)$ can be ignored.) Thus

$$p_{j,k} = p_{k-j,k} = \frac{1}{2^{k-2}} \binom{k-2}{j-1}$$

Theorem.
$E(D_k) \leq \log k$.

Proof.
The proof is by induction. The theorem clearly holds for $k = 1$ and $k = 2$. We then show that it holds for arbitrary $k > 2$ on the assumption that it holds for all smaller k. Now, by the inductive hypothesis, for any $1 \leq j \leq k - 1$,

$$E(D_j) + E(D_{k-j}) \leq \log j + \log k - j$$

Since $f(x) = \log x$ is convex for $x > 0$,

$$\log j + \log k - j \leq 2 \log(k/2)$$

Thus

$$E(D_j) + E(D_{k-j}) \leq \log(k/2) + \log(k/2)$$

Furthermore, since $p_{j,k} = p_{k-j,k}$,

$$p_{j,k}E(D_j) + p_{k-j,k}E(D_{k-j}) \leq p_{j,k} \log(k/2) + p_{k-j,k} \log(k/2)$$

From the lemma,

$$\begin{aligned} E(D_k) &= \sum_{j=1}^{k-1} p_{j,k}(E(D_j) + 1) \\ &= 1 + \sum_{j=1}^{k-1} p_{j,k}E(D_j) \end{aligned}$$

Thus

$$\begin{aligned} E(D_k) &\le 1 + \sum_{j=1}^{k-1} p_{j,k} \log(k/2) \\ &\le 1 + \log(k/2) \sum_{j=1}^{k-1} p_{j,k} \\ &\le 1 + \log(k/2) \\ &\le \log k \end{aligned}$$

completing the proof.

It can also be shown that asymptotically, $E(D_k) \approx \log(k-1) - 0.273548$.

6.10 **(a)** Perform a preorder traversal of the heap.

(b) Works for leftist and skew heaps. The running time is $O(Kd)$ for d-heaps.

6.12 Simulations show that the linear time algorithm is the faster, not only on worst-case inputs, but also on random data.

6.13 **(a)** If the heap is organized as a (min) heap, then starting at the hole at the root, find a path down to a leaf by taking the minimum child. The requires roughly $\log N$ comparisons. To find the correct place where to move the hole, perform a binary search on the $\log N$ elements. This takes $O(\log \log N)$ comparisons.

(b) Find a path of minimum children, stopping after $\log N - \log \log N$ levels. At this point, it is easy to determine if the hole should be placed above or below the stopping point. If it goes below, then continue finding the path, but perform the binary search on only the last $\log \log N$ elements on the path, for a total of $\log N + \log \log \log N$ comparisons. Otherwise, perform a binary search on the first $\log N - \log \log N$ elements. The binary search takes at most $\log \log N$ comparisons, and the path finding took only $\log N - \log \log N$, so the total in this case is $\log N$. So the worst case is the first case.

(c) The bound can be improved to $\log N + \log *N + O(1)$, where $\log *N$ is the inverse Ackerman function (see Chapter 8). This bound can be found in reference [17].

6.14 The parent is at position $\lfloor (i + d - 2)/d \rfloor$. The children are in positions $(i-1)d + 2, \ldots, id + 1$.

6.15 **(a)** $O((M + dN) \log_d N)$.

(b) $O((M + N) \log N)$.

(c) $O(M + N^2)$.

(d) $d = \max(2, M/N)$. (See the related discussion at the end of Section 11.4.)

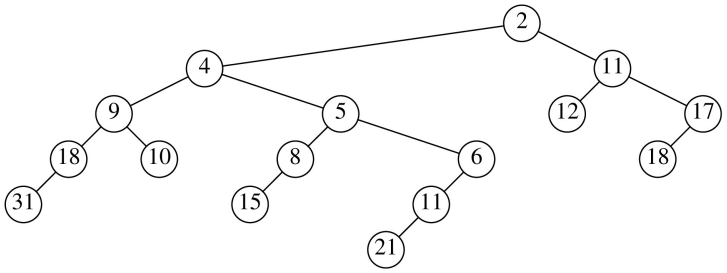
6.16 Starting from the second most signficant digit in i, and going toward the least significant digit, branch left for 0s, and right for 1s.

6.17 **(a)** Place negative infinity as a root with the two heaps as subtrees. Then do a *deleteMin*.

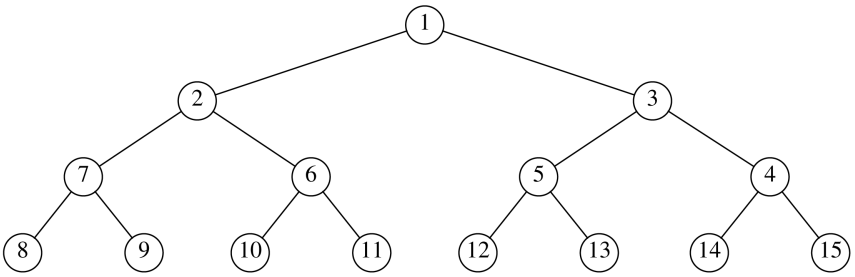
(b) Place negative infinity as a root with the larger heap as the left subheap, and the smaller heap as the right subheap. Then do a *deleteMin*.

(c) SKETCH: Split the larger subheap into smaller heaps as follows: on the left-most path, remove two subheaps of height $r - 1$, then one of height r, $r + 1$, and so one, until $l - 2$. Then merge the trees, going smaller to higher, using the results of parts (a) and (b), with the extra nodes on the left path substituting for the insertion of infinity, and subsequent *deleteMin*.

6.19



6.20



6.21 This theorem is true, and the proof is very much along the same lines as Exercise 4.20.

6.22 If elements are inserted in decreasing order, a leftist heap consisting of a chain of left children is formed. This is the best because the right path length is minimized.

6.23 **(a)** If a *decreaseKey* is performed on a node that is very deep (very left), the time to percolate up would be prohibitive. Thus the obvious solution doesn't work. However, we can still do the operation efficiently by a combination of *remove* and *insert*. To *remove* an arbitrary node x in the heap, replace x by the *merge* of its left and right subheaps. This might create an imbalance for nodes on the path from x's parent to the root that would need to be fixed by a child swap. However, it is easy to show that at most $\log N$ nodes can be affected, preserving the time bound.

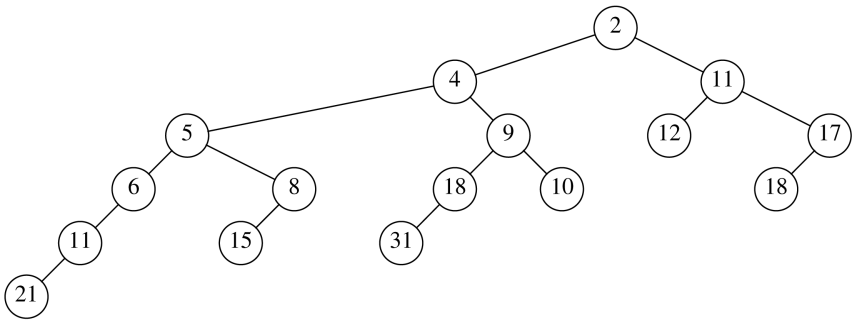
This is discussed in Chapter 11.

6.24 Lazy deletion in leftist heaps is discussed in the paper by Cheriton and Tarjan [10]. The general idea is that if the root is marked deleted, then a preorder traversal of the heap is formed, and the frontier of marked nodes is removed, leaving a collection of heaps. These can be merged two at a time by placing all the heaps on a queue, removing two, merging them, and placing the result at the end of the queue, terminating when only one heap remains.

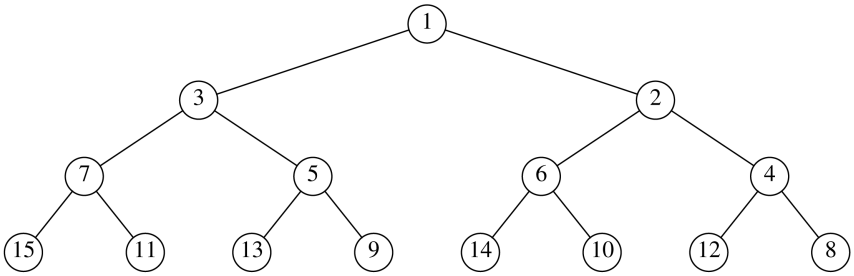
6.25 **(a)** The standard way to do this is to divide the work into passes. A new pass begins when the first element reappears in a heap that is dequeued. The first pass takes roughly $2 * 1 * (N/2)$ time units because there are $N/2$ merges of trees with one node each on the right path. The next pass takes $2 * 2 * (N/4)$ time units because of the roughly $N/4$ merges of trees with no more than two nodes on the right path. The third pass takes $2 * 3 * (N/8)$ time units, and so on. The sum converges to $4N$.

(b) It generates heaps that are more leftist.

6.26



6.27



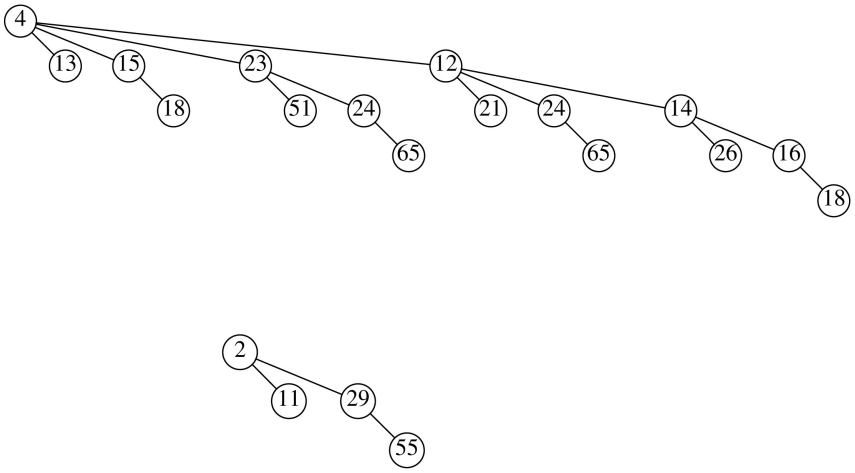
6.28 This claim is also true, and the proof is similar in spirit to Exercise 4.20 or 6.21.

6.29 Yes. All the single operation estimates in Exercise 6.25 become amortized instead of worst-case, but by the definition of amortized analysis, the sum of these estimates is a worst-case bound for the sequence.

6.30 Clearly the claim is true for $k = 1$. Suppose it is true for all values $i = 1, 2, \ldots, k$. A B_{k+1} tree is formed by attaching a B_k tree to the root of a B_k tree. Thus by induction, it contains a B_0 through B_{k-1} tree, as well as the newly attached B_k tree, proving the claim.

6.31 Proof is by induction. Clearly the claim is true for $k = 1$. Assume true for all values $i = 1, 2, \ldots, k$. A B_{k+1} tree is formed by attaching a B_k tree to the original B_k tree. The original thus had $\binom{k}{d}$ nodes at depth d. The attached tree had $\binom{k}{d-1}$ nodes at depth $d-1$, which are now at depth d. Adding these two terms and using a well-known formula establishes the theorem.

6.32



6.33 This is established in Chapter 11.

6.38 Don't keep the key values in the heap, but keep only the difference between the value of the key in a node and the value of the parent's key.

6.39 $O(N + k \log N)$ is a better bound than $O(N \log k)$. The first bound is $O(N)$ if $k = O(N/\log N)$. The second bound is more than this as soon as k grows faster than a constant. For the other values $\Omega(N/\log N) = k = o(N)$, the first bound is better. When $k = \Theta(N)$, the bounds are identical.

CHAPTER 7

Sorting

7.1

Original	3	1	4	1	5	9	2	6	5
after p=2	1	3	4	1	5	9	2	6	5
after p=3	1	3	4	1	5	9	2	6	5
after p=4	1	1	3	4	5	9	2	6	5
after p=5	1	1	3	4	5	9	2	6	5
after p=6	1	1	3	4	5	9	2	6	5
after p=7	1	1	2	3	4	5	9	6	5
after p=8	1	1	2	3	4	5	6	9	5
after p=9	1	1	2	3	4	5	5	6	9

7.2 $O(N)$, because the *while* loop terminates immediately. Of course, accidentally changing the test to include equalities raises the running time to quadratic for this type of input.

7.3 The inversion that existed between $a[i]$ and $a[i+k]$ is removed. This shows at least one inversion is removed. For each of the $k-1$ elements $a[i+1]$, $a[i+2]$, . . . , $a[i+k-1]$, at most two inversions can be removed by the exchange. This gives a maximum of $2(k-1)+1 = 2k-1$.

7.4

Original	9	8	7	6	5	4	3	2	1
after 7-sort	2	1	7	6	5	4	3	9	8
after 3-sort	2	1	4	3	5	7	6	9	8
after 1-sort	1	2	3	4	5	6	7	8	9

7.5 **(a)** $\Theta(N^2)$. The 2-sort removes at most only three inversions at a time; hence the algorithm is $\Omega(N^2)$. The 2-sort is two insertion sorts of size $N/2$, so the cost of that pass is $O(N^2)$. The 1-sort is also $O(N^2)$, so the total is $O(N^2)$.

7.6 Part (a) is an extension of the theorem proved in the text. Part (b) is fairly complicated; see reference [12].

7.7 See reference [12].

7.8 Use the input specified in the hint. If the number of inversions is shown to be $\Omega(N^2)$, then the bound follows, since no increments are removed until an $h_{t/2}$ sort. If we consider the pattern formed h_k through h_{2k-1}, where $k = t/2+1$, we find that it has length $N = h_k(h_k+1)-1$, and the number of inversions is roughly ${h_k}^4/24$, which is $\Omega(N^2)$.

7.9 **(a)** $O(N \log N)$. No exchanges, but each pass takes $O(N)$.

(b) $O(N \log N)$. It is easy to show that after an h_k sort, no element is farther than h_k from its rightful position. Thus if the increments satisfy $h_{k+1} \le ch_k$ for a constant c, which implies $O(\log N)$ increments, then the bound is $O(N \log N)$.

7.10 **(a)** No, because it is still possible for consecutive increments to share a common factor. An example is the sequence 1, 3, 9, 21, 45, $h_{t+1} = 2h_t + 3$.

(b) Yes, because consecutive increments are relatively prime. The running time becomes $O(N^{3/2})$.

7.11 The input is read in as

142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102

The result of the heapify is

879, 811, 572, 434, 543, 123, 142, 65, 111, 242, 453, 102

879 is removed from the heap and placed at the end. We'll place it in italics to signal that it is not part of the heap. 102 is placed in the hole and bubbled down, obtaining

811, 543, 572, 434, 453, 123, 142, 65, 111, 242, 102, *879*

Continuing the process, we obtain

572, 543, 142, 434, 453, 123, 102, 65, 111, 242, *811*, *879*
543, 453, 142, 434, 242, 123, 102, 65, 111, *572*, *811*, *879*
453, 434, 142, 111, 242, 123, 102, 65, *543*, *572*, *811*, *879*
434, 242, 142, 111, 65, 123, 102, *453*, *543*, *572*, *811*, *879*
242, 111, 142, 102, 65, 123, *434*, *453*, *543*, *572*, *811*, *879*
142, 111, 123, 102, 65, *242*, *434*, *453*, *543*, *572*, *811*, *879*
123, 111, 65, 102, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*
111, 102, 65, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*
102, 65, *111*, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*
65, *102*, *111*, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

7.13 Heapsort uses at least (roughly) $N \log N$ comparisons on any input, so there are no particularly good inputs. This bound is tight; see the paper by Schaeffer and Sedgewick [18]. This result applies for almost all variations of heapsort, which have different rearrangement strategies. See Y. Ding and M. A. Weiss, "Best Case Lower Bounds for Heapsort," *Computing* 49 (1992).

7.14 If the root is stored in position *low*, then the left child of node i is stored at position $2i + 1 - low$. This requires a small change to the heapsort code.

7.15 First the sequence {3, 1, 4, 1} is sorted. To do this, the sequence {3, 1} is sorted. This involves sorting {3} and {1}, which are base cases, and merging the result to obtain {1, 3}. The sequence {4, 1} is likewise sorted into {1, 4}. Then these two sequences are merged to obtain {1, 1, 3, 4}. The second half is sorted similarly, eventually obtaining {2, 5, 6, 9}. The merged result is then easily computed as {1, 1, 2, 3, 4, 5, 6, 9}.

7.16

```
private static <AnyType extends Comparable<? super AnyType>>
void mergeSort( AnyType [] a, AnyType [] tmpArray, int left, int right )
{
  int n = a.length;

  for ( int subListSize = 1; subListSize < n; subListSize *= 2 )
  {
    int part1Start = 0;
    while ( part1Start + subListSize < n )
    {
```

```
            int part2Start = part1Start + subListSize;
            int part2End = min( n - 1, part2Start + subListSize - 1 );

            merge( a, tmpArray, part1Start, part2Start, part2End );
            part1Start = part2End + 1;
          }
        }
      }
```

7.17 The merging step always takes $\Theta(N)$ time, so the sorting process takes $\Theta(N \log N)$ time on all inputs.

7.18 See reference [12] for the exact derivation of the worst case of mergesort.

7.19 The original input is

$$3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5$$

After sorting the first, middle, and last elements, we have

$$3, 1, 4, 1, 5, 5, 2, 6, 5, 3, 9$$

Thus the pivot is 5. Hiding it gives

$$3, 1, 4, 1, 5, 3, 2, 6, 5, 5, 9$$

The first swap is between two fives. The next swap has i and j crossing. Thus the pivot is swapped back with i:

$$3, 1, 4, 1, 5, 3, 2, 5, 5, 6, 9$$

We now recursively quicksort the first eight elements:

$$3, 1, 4, 1, 5, 3, 2, 5$$

Sorting the three appropriate elements gives

$$1, 1, 4, 3, 5, 3, 2, 5$$

Thus the pivot is 3, which gets hidden:

$$1, 1, 4, 2, 5, 3, 3, 5$$

The first swap is between 4 and 3:

$$1, 1, 3, 2, 5, 4, 3, 5$$

The next swap crosses pointers, so is undone; i points at 5, and so the pivot is swapped:

$$1, 1, 3, 2, 3, 4, 5, 5$$

A recursive call is now made to sort the first four elements. The pivot is 1, and the partition does not make any changes. The recursive calls are made, but the subfiles are below the cutoff, so nothing is done. Likewise, the last three elements constitute a base case, so nothing is done. We return to the original call, which now calls quicksort recursively on the right-hand side, but again, there are only three elements, so nothing is done. The result is

$$1, 1, 3, 2, 3, 4, 5, 5, 5, 6, 9$$

which is cleaned up by insertion sort.

7.20 **(a)** $O(N \log N)$ because the pivot will partition perfectly.

(b) Again, $O(N \log N)$ because the pivot will partition perfectly.

(c) $O(N \log N)$; the performance is slightly better than the analysis suggests because of the median-of-three partition and cutoff.

7.21 **(a)** If the first element is chosen as the pivot, the running time degenerates to quadratic in the first two cases. It is still $O(N \log N)$ for random input.

(b) The same results apply for this pivot choice.

(c) If a random element is chosen, then the running time is $O(N \log N)$ expected for all inputs, although there is an $O(N^2)$ worst case if very bad random numbers come up. There is, however, an essentially negligible chance of this occurring. Chapter 10 discusses the randomized philosophy.

(d) This is a dangerous road to go; it depends on the distribution of the keys. For many distributions, such as uniform, the performance is $O(N \log N)$ on average. For a skewed distribution, such as with the input $\{1, 2, 4, 8, 16, 32, 64, \ldots, \}$ the pivot will be consistently terrible, giving quadratic running time, independent of the ordering of the input.

7.22 **(a)** $O(N \log N)$ because the pivot will partition perfectly.

(b) Sentinels need to be used to guarantee that i and j don't run past the end. The running time will be $\Theta(N^2)$ since the partitioning step will put all but the pivot in S_1, because i won't stop until it hits the sentinel.

(c) Again a sentinel needs to be used to stop j. This is also $\Theta(N^2)$ because the partitioning is unbalanced.

7.23 Yes, but it doesn't reduce the average running time for random input. Using median-of-three partitioning reduces the average running time because it makes the partition more balanced on average.

7.24 The strategy used here is to force the worst possible pivot at each stage. This doesn't necessarily give the maximum amount of work (since there are few exchanges, just lots of comparisons), but it does give $\Omega(N^2)$ comparisons. By working backward, we can arrive at the following permutation:

$$20, 3, 5, 7, 9, 11, 13, 15, 17, 19, 4, 10, 2, 12, 6, 14, 1, 16, 8, 18$$

A method to extend this to larger numbers when N is even is as follows: The first element is N, the middle is $N - 1$, and the last is $N - 2$. Odd numbers (except 1) are written in decreasing order starting to the left of center. Even numbers are written in decreasing order by starting at the rightmost spot, always skipping one available empty slot, and wrapping around when the center is reached. This method takes $O(N \log N)$ time to generate the permutation, but is suitable for a hand calculation. By inverting the actions of quicksort, it is possible to generate the permutation in linear time.

7.26 Each recursive call is on a subarray that is less than half the size of the original, giving a logarithmic number of recursive calls.

7.27 See reference [14].

7.28 See reference [1].

7.30 This recurrence results from the analysis of the quick selection algorithm. $T(N) = O(N)$.

7.31 Insertion sort and mergesort are stable if coded correctly. Any of the sorts can be made stable by the addition of a second key, which indicates the original position.

7.32 **(d)** $f(N)$ can be $O(N/\log N)$. Sort the $f(N)$ elements using mergesort in $O(f(N)\log f(N))$ time. This is $O(N)$ if $f(N)$ is chosen using the criterion given. Then merge this sorted list with the already sorted list of N numbers in $O(N + f(N)) = O(N)$ time.

7.33 A decision tree would have N leaves, so $\lceil \log N! \rceil$ comparisons are required.

7.34 $\log N! \approx N \log N - N \log e$.

7.35 **(a)** $\binom{2N}{N}$

(b) The information-theoretic lower bound is $\log \binom{2N}{N}$. Applying Stirling's formula, we can estimate the bound as $2N - \mathit{half} \log N$. A better lower bound is known for this case: $2N - 1$ comparisons are necessary. Merging two lists of different sizes M and N likewise requires at least $\log \binom{M+N}{N}$ comparisons.

7.36 Algorithms A and B require at least three comparisons based on the information-theoretic lower bound. Algorithm C requires at least five comparisons, based on Exercise 7.35(b). So the algorithm requires 11 comparisons; but six elements can be sorted in 10 comparisons (for instance, use the results in Exercise 7.42, and use binary search to find the correct place to insert the sixth element).

7.37 Note that any two fractions that are not equal differ by at least $1/N^2$. Thus rewrite the fractions as $c/(N+1)^2 + d$, ignore the d, and sort on the basis of c using a two-pass radix sort.

7.38 Look at the middle elements of A and B. Let m_1 and m_2 be these two middle elements; assume m_1 is the larger (the mirror image is similar). Recursively use the elements in A that are larger than m_1 and the elements in B that are smaller than m_2, making sure that an equal number of elements from A and B are not included in the recursive problem (include m_1 or m_2 if needed to balance). Recursively solve the problem on the subarray consisting of elements larger than the larger of the two middles.

7.39 We add a dummy $N + 1^{th}$ element, which we'll call *maybe*. *maybe* satisfies *false* < *maybe* < *true*. Partition the array around *maybe*, using the standard quicksort techniques. Then swap *maybe* and the $N + 1^{th}$ element. The first N elements are then correctly arranged.

7.40 We add a dummy $N + 1^{th}$ element, which we'll call *probablyFalse*. *probablyFalse* satisfies *false* < *probablyFalse* < *maybe*. Partition the array around *probablyFalse* as in the previous exercise. Suppose that after the partition, *probablyFalse* winds up in position i. Then place the element that is in the $N + 1^{th}$ slot in position i, place *probablyTrue* (defined the obvious way) in position $N + 1$, and partition the subarray from position i onward. Finally, swap *probablyTrue* with the element in the $N + 1^{th}$ location. The first N elements are now correctly arranged. These two problems can be done without the assumption of an extra array slot; assuming so simplifies the presentation.

7.41 **(a)** $\lceil \log 4! \rceil = 5$.

(b) Compare and exchange (if necessary) a_1 and a_2 so that $a_1 \geq a_2$, and repeat with a_3 and a_4. Compare and exchange a_1 and a_3. Compare and exchange a_2 and a_4. Finally, compare and exchange a_2 and a_3.

7.42 **(a)** $\lceil \log 5! \rceil = 7$.

(b) Compare and exchange (if necessary) a_1 and a_2 so that $a_1 \geq a_2$, and repeat with a_3 and a_4 so that $a_3 \geq a_4$. Compare and exchange (if necessary) the two winners, a_1 and a_3. Assume without loss of generality that we now have $a_1 \geq a_3 \geq a_4$, and $a_1 \geq a_2$. (The other case is obviously identical.) Insert a_5 by binary search in the appropriate place among a_1 , a_3,a_4. This can be done in two comparisons.

Finally, insert a_2 among a_3 , a_4 , a_5. If it is the largest among those three, then it goes directly after a_1 since it is already known to be larger than a_1. This takes two more comparisons by a binary search. The total is thus seven comparisons.

7.46 **(a)** For the given input, the pivot is 2. It is swapped with the last element. i will point at the second element, and j will be stopped at the first element. Since the pointers have crossed, the pivot is swapped with the element in position 2. The input is now 1, 2, 4, 5, 6, . . . , $N-1$, N, 3. The recursive call on the right subarray is thus on an increasing sequence of numbers, except for the last number, which is the smallest. This is exactly the same form as the original. Thus each recursive call will have only two fewer elements than the previous. The running time will be quadratic.

(b) Although the first pivot generates equal partitions, both the left and right halves will have the same form as part (a). Thus the running time will be quadratic because after the first partition, the algorithm will grind slowly. This is one of the many interesting tidbits in reference [22].

7.47 We show that in a binary tree with L leaves, the average depth of a leaf is at least $\log L$. We can prove this by induction. Clearly, the claim is true if $L = 1$. Suppose it is true for trees with up to $L-1$ leaves. Consider a tree of L leaves with minimum average leaf depth. Clearly, the root of such a tree must have non-null left and right subtrees. Suppose that the left subtree has L_L leaves, and the right subtree has L_R leaves. By the inductive hypothesis, the total depth of the leaves (which is their average times their number) in the left subtree is $L_L(1 + \log L_L)$, and the total depth of the right subtree's leaves is $L_R(1 + \log L_R)$ (because the leaves in the subtrees are one deeper with respect to the root of the tree than with respect to the root of their subtree). Thus the total depth of all the leaves is $L + L_L \log L_L + L_R \log L_R$. Since $f(x) = x \log x$ is convex for $x \geq 1$, we know that $f(x) + f(y) \geq 2f((x+y)/2)$. Thus the total depth of all the leaves is at least $L + 2(L/2)\log(L/2) \geq L + L(\log L - 1) \geq L \log L$. Thus the average leaf depth is at least $\log L$.

7.48 **(a)** A brute-force search gives a quadratic solution.

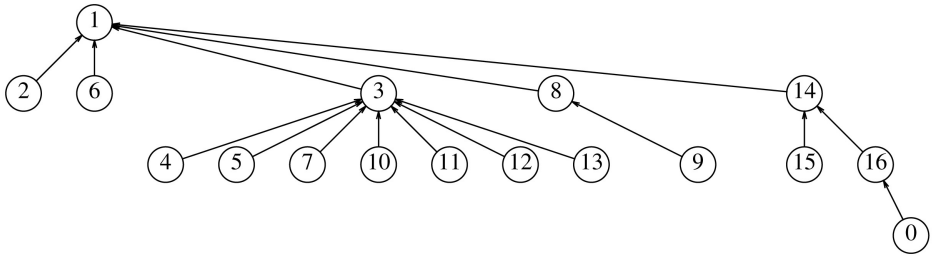
(b) Sort the items; then run i and j from opposite ends of the array toward each other, incrementing i if $a[i] + a[j] < K$, and decrementing j if $a[i] + a[j] > K$. The sort is $O(N \log N)$; the scan is $O(N)$.

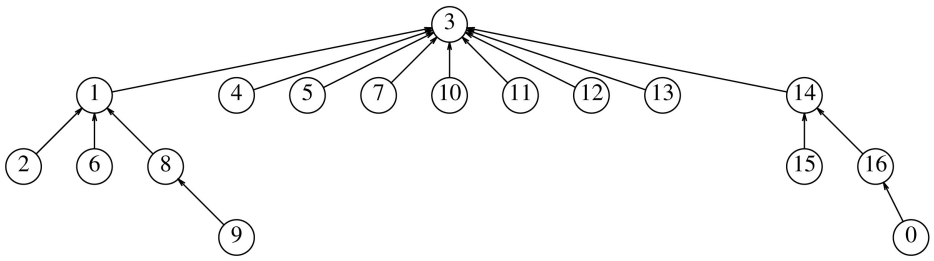
7.50 **(b)** After sorting the items do a scan using an outer loop that considers all positions p, and an inner loop that searches for two additional items that sum to $K - a[p]$. The two loops give a quadratic algorithm.

CHAPTER 8

The Disjoint Set Class

8.1 We assume that unions operated on the roots of the trees containing the arguments. Also, in case of ties, the second tree is made a child of the first. Arbitrary union and union by height give the same answer (shown as the first tree) for this problem. Union by size gives the second tree.





8.2 In both cases, have nodes 16 and 0 point directly to the root.

8.4 Claim: A tree of height h has at least 2^h nodes. The proof is by induction. A tree of height 0 clearly has at least 1 node, and a tree of height 1 clearly has at least 2. Let T be the tree of height h with fewest nodes. Thus at the time of T's last union, it must have been a tree of height $h - 1$, since otherwise T would have been smaller at that time than it is now and still would have been of height h, which is impossible by assumption of T's minimality. Since T's height was updated, it must have been as a result of a union with another tree of height $h - 1$. By the induction hypothesis, we know that at the time of the union, T had at least 2^{h-1} nodes, as did the tree attached to it, for a total of 2^h nodes, proving the claim. Thus an N-node tree has depth at most $\lfloor \log N \rfloor$.

8.5 All answers are $O(M)$ because in all cases $alpha(M, N) = 1$.

8.6 It is easy to see that this invariant is maintained in each disjoint set as the algorithm proceeds.

8.7 Run the normal algorithm; never remove the prespecified wall. Let s_1 and s_2 be squares on the opposite side of the wall. Do not remove any wall that results in s_1 being connected to the ending point or s_2 being connected to the starting point, and continue until there are only two disjoint sets remaining.

8.8 **(a)** When we perform a *union* we push onto a stack the two roots and the old values of their parents. To implement a *deunion*, we only have to pop the stack and restore the values. This strategy works fine in the absence of path compression.

(b) If path compression is implemented, the strategy described in part (a) does not work, because path compression moves elements out of subtrees. For instance, the sequence *union(1,2), union(3,4), union(1,3), find(4), deunion(1,3)* will leave 4 in set 1 if path compression is implemented.

8.9 We assume that the tree is implemented with links instead of a simple array. Thus *find* will return a reference instead of an actual set name. We will keep an array to map set numbers to their tree nodes. *union* and *find* are implemented in the standard manner. To perform *remove(x)*, first perform a *find(x)* with path compression. Then mark the node containing x as vacant. Create a new one-node tree with x and have it pointed to by the appropriate array entry. The time to perform a *remove* is the same as the time to perform a *find*, except that there potentially could be a large number of vacant nodes. To take care of this, after N *remove*s are performed, perform a *find* on every node, with path compression. If a *find*(x) returns a vacant root, then place x in the root node, and make the old node containing x vacant. The results of Exercise 8.11 guarantee that this will take linear time, which can be charged to the N *remove*s. At this point, all vacant nodes (indeed all nonroot nodes) are children of a root, and vacant nodes can be disposed (if an array of references to them has been kept). This also guarantees that there are never more than $2N$ nodes in the forest and preserves the $M\alpha(M, N)$ asymptotic time bound.

8.11 Suppose there are u *union* and f *find* operations. Each *union* costs constant time, for a total of u. A *find* costs one unit per vertex visited. We charge, as in the text, under the following slightly modified rules:

(A) the vertex is a root or child of the root

(B) otherwise

Essentially, all vertices are in one rank group. During any *find*, there can be at most two rule (A) charges, for a total of $2f$. Each vertex can be charged at most once under rule (B) because after path compression it will be a child of the root. The number of vertices that are not roots or children of roots is clearly bounded by u, independent of the unioning strategy, because each *union* changes exactly one vertex from root to nonroot status, and this bounds the number of type (B) nodes. Thus the total rule (B) charges are at most u. Adding all charges gives a bound of $2f + 2u$, which is linear in the number of operations.

8.13 For each vertex v, let the pseudorank R_v be defined as $\lfloor \log S_v \rfloor$, where S_v is the number of descendents (including itself) of v in the final tree, after all *union* operations are performed, ignoring path compression.

Although the pseudorank is not maintained by the algorithm, it is not hard to show that the pseudorank satisfies the same properties as the ranks do in union-by-rank. Clearly, a vertex with pseudorank R_v has at least 2^{R_v} descendents (by its definition), and the number of vertices of pseudorank R is at most $N/2^R$. The union-by-size rule ensures that the parent of a node has twice as many descendents as the node, so the pseudoranks monotonically increase on the path toward the root if there is no path compression. The argument in Lemma 8.3 tells us that path compression does not destroy this property.

If we partition the vertices by pseudoranks and assign the charges in the same manner as in the text proof for union-by-rank, the same steps follow, and the identical bound is obtained.

8.14 This is most conveniently implemented without recursion and is faster because, even if full path compression is implemented nonrecursively, it requires two passes up the tree. This requires only one.

We leave the coding to the reader since comparing the various *union* and *find* strategies is a reasonable programming project. The worst-case running time remains the same because the properties of the ranks are unchanged. Instead of charging one unit to each vertex on the path to the root, we can charge two units to alternating vertices (namely, the vertices whose parents are altered by path halving). These vertices get parents of higher rank, as before, and the same kind of analysis bounds the total charges.

CHAPTER 9

Graph Algorithms

9.1 The following ordering is arrived at by using a queue and assumes that vertices appear on an adjacency list alphabetically. The topological order that results is then

s, G, D, H, A, B, E, I, F, C, t

9.2 Assuming the same adjacency list, the topological order produced when a stack is used is

s, G, H, D, A, E, I, F, B, C, t

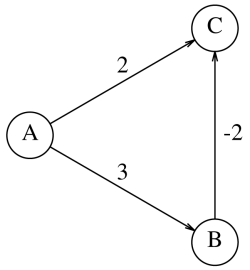
Because a topological sort processes vertices in the same manner as a breadth-first search, it tends to produce a more natural ordering.

9.4 The idea is the same as in Exercise 5.12.

9.5 **(a)** (Unweighted paths) A $\rightarrow$ B, A $\rightarrow$ C, A $\rightarrow$ B $\rightarrow$ G, A $\rightarrow$ B $\rightarrow$ E, A $\rightarrow$ C $\rightarrow$ D, A $\rightarrow$ B $\rightarrow$ E $\rightarrow$ F. **(b)** (Weighted paths) A $\rightarrow$ C, A $\rightarrow$ B, A $\rightarrow$ B $\rightarrow$ G, A $\rightarrow$ B $\rightarrow$ G $\rightarrow$ E, A $\rightarrow$ B $\rightarrow$ G $\rightarrow$ E $\rightarrow$ F, A $\rightarrow$ B $\rightarrow$ G $\rightarrow$ E $\rightarrow$ D.

9.6 We'll assume that Dijkstra's algorithm is implemented with a priority queue of vertices that uses the *decreaseKey* operation. Dijkstra's algorithm uses $|E|$ *decreaseKey* operations, which cost $O(\log_d |V|)$ each, and $|V|$ *deleteMin* operations, which cost $O(d \log_d |V|)$ each. The running time is thus $O(|E| \log_d |V| + |V| d \log_d |V|)$. The cost of the *decreaseKey* operations balances the *insert* operations when $d = |E|/|V|$. For a sparse graph, this might give a value of d that is less than 2; we can't allow this, so d is chosen to be $\max(2, \lfloor |E|/|V| \rfloor)$. This gives a running time of $O(|E| \log_{2+|E|/|V|} |V|)$, which is a slight theoretical improvement. Moret and Shapiro report (indirectly) that d-heaps do not improve the running time in practice.

9.7 **(a)** The graph shown here is an example. Dijkstra's algorithm gives a path from A to C of cost 2, when the path from A to B to C has cost 1.



(b) We define a pass of the algorithm as follows: Pass 0 consists of marking the start vertex as known and placing its adjacent vertices on the queue. For $j > 0$, pass j consists of marking as known all vertices on the queue at the end of pass $j - 1$. Each pass requires linear time, since during a pass, a vertex is placed on the queue at most once. It is easy to show by induction that if there is a shortest

path from s to v containing k edges, then d_v will equal the length of this path by the beginning of pass k. Thus there are at most $|V|$ passes, giving an $O(|E||V|)$ bound.

9.8 See the comments for Exercise 9.19.

9.10 **(a)** Use an array *count* such that for any vertex u, *count[u]* is the number of distinct paths from s to u known so far. When a vertex v is marked as known, its adjacency list is traversed. Let w be a vertex on the adjacency list.

If $d_v + c_{v,w} = d_w$, then increment *count[w]* by *count[v]* because all shortest paths from s to v with last edge (v, w) give a shortest path to w.

If $d_v + c_{v,w} < d_w$, then p_w and d_w get updated. All previously known shortest paths to w are now invalid, but all shortest paths to v now lead to shortest paths for w, so set *count[w]* to equal *count[v]*. Note: Zero-cost edges mess up this algorithm.

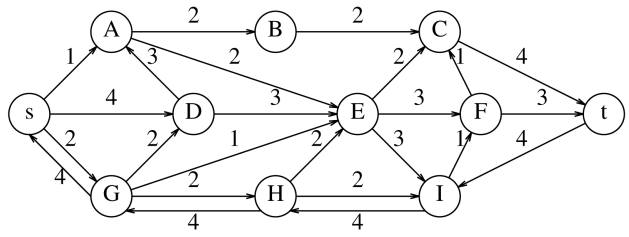
(b) Use an array *numEdges* such that for any vertex u, *numEdges[u]* is the shortest number of edges on a path of distance d_u from s to u known so far. Thus *numEdges* is used as a tiebreaker when selecting the vertex to mark. As before, v is the vertex marked known, and w is adjacent to v.

If $d_v + c_{v,w} = d_w$, then change p_w to v and *numEdges[w]* to *numEdges[v]+1* if *numEdges[v]+1* < *numEdges[w]*.

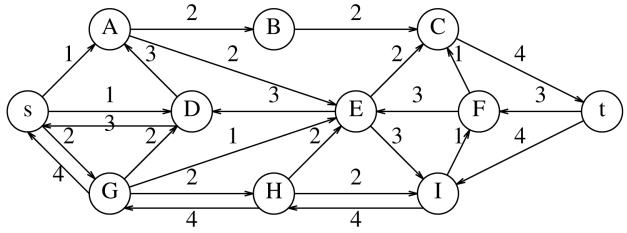
If $d_v + c_{v,w} < d_w$, then update p_w and d_w, and set *numEdges[w]* to *numEdges[v]+1*.

9.11 (This solution is not unique.)

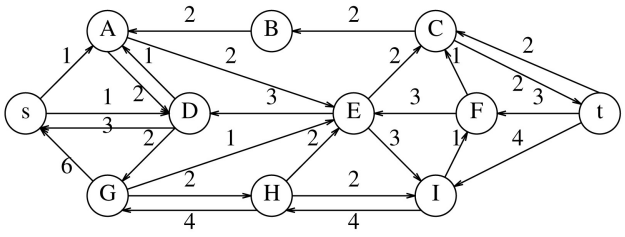
First, send four units of flow along the path s, G, H, I, t. This gives the following residual graph:



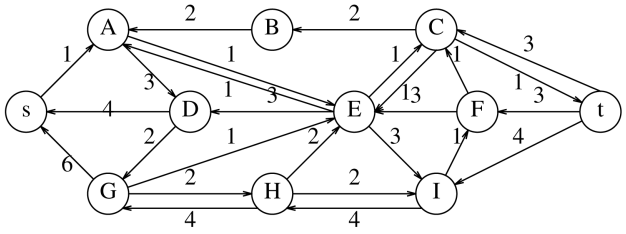
Next, send three units of flow along s, D, E, F, t. The residual graph that results is as follows:



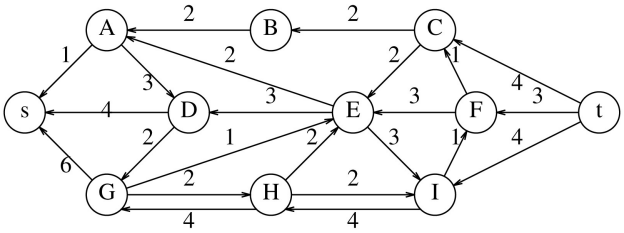
Now two units of flow are sent along the path s, G, D, A, B, C, t, yielding the following residual graph:



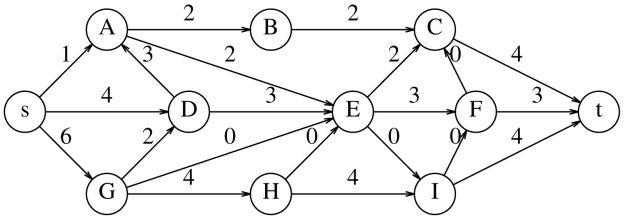
One unit of flow is then sent along s, D, A, E, C, t:



Finally, one unit of flow can go along the path s, A, E, C, t:



The preceding residual graph has no path from s to t. Thus the algorithm terminates. The final flow graph, which carries 11 units, is as follows:



This flow is not unique. For instance, two units of the flow that goes from G to D to A to E could go by G to H to E.

9.12 Let T be the tree with root r, and children $r_1, r_2, \ldots, r_k$, which are the roots of $T_1, T_2, \ldots, T_k$, which have maximum incoming flow of $c_1, c_2, \ldots, c_k$, respectively. By the problem statement, we may take the maximum incoming flow of r to be infinity. The recursive pseudo-method *findMaxFlow(T, incomingCap)* finds the value of the maximum flow in T (finding the actual flow is a matter of bookkeeping); the flow is guaranteed not to exceed *incomingCap*.

If T is a leaf, then *findMaxFlow* returns *incomingCap* since we have assumed a sink of infinite capacity. Otherwise, a standard postorder traversal can be used to compute the maximum flow in linear time.

```
// FlowType is an int or double
static FlowType findMaxFlow( Tree T, FlowType incomingCap )
{
    FlowType childFlow, totalFlow;

    if( T.hasNoSubtrees( ) )
        return incomingCap;
    else
    {
        totalFlow = 0;
        for( each subtree $T_i$ of T )
        {
            childFlow = findMaxFlow( $T_i$, min( incomingCap, $c_i$ ) );
            totalFlow += childFlow;
            incomingCap -= childFlow;
        }
        return totalFlow;
    }
}
```

9.13 **(a)** Assume that the graph is connected and undirected. If it is not connected, then apply the algorithm to the connected components. Initially, mark all vertices as unknown. Pick any vertex v, color it red, and perform a depth-first search. When a node is first encountered, color it blue if the DFS has just come from a red node, and red otherwise. If at any point, the depth-first search encounters an edge between two identical colors, then the graph is not bipartite; otherwise, it is. A breadth-first search (that is, using a queue) also works. This problem, which is essentially two-coloring a graph, is clearly solvable in linear time. This contrasts with three-coloring, which is NP-complete.

(b) Construct an undirected graph with a vertex for each instructor, a vertex for each course, and an edge between (v, w) if instructor v is qualified to teach course w. Such a graph is bipartite; a matching of M edges means that M courses can be covered simultaneously.

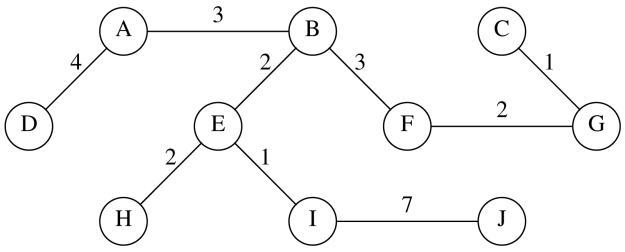
(c) Give each edge in the bipartite graph a weight of 1, and direct the edge from the instructor to the course. Add a vertex s with edges of weight 1 from s to all instructor vertices. Add a vertex t with edges of weight 1 from all course vertices to t. The maximum flow is equal to the maximum matching.

(d) The running time is $O(|E||V|^{half})$ because this is the special case of the network flow problem mentioned in the text. All edges have unit cost, and every vertex (except s and t) has either an indegree or outdegree of 1.

9.14 This is a slight modification of Dijkstra's algorithm. Let f_i be the best flow from s to i at any point in the algorithm. Initially, $f_i = 0$ for all vertices, except s: $f_s = inf$.

At each stage, we select v such that f_v is maximum among all unknown vertices. Then for each w adjacent to v, the cost of the flow to w using v as an intermediate is $\min(f_v, c_{v,w})$. If this value is higher than the current value of f_w, then f_w and p_w are updated.

9.15 One possible minimum spanning tree is shown here. This solution is not unique.



9.16 Both work correctly. The proof makes no use of the fact that an edge must be nonnegative.

9.17 The proof of this fact can be found in any good graph theory book. A more general theorem follows:

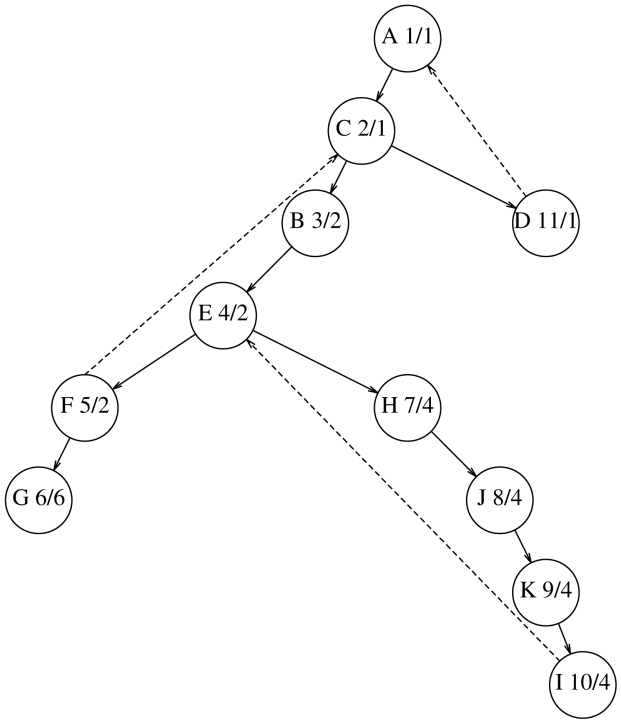
Theorem.
Let $G = (V, E)$ be an undirected, unweighted graph, and let A be the adjacency matrix for G (which contains either 1s or 0s). Let D be the matrix such that $D[v][v]$ is equal to the degree of v; all nondiagonal matrices are 0. Then the number of spanning trees of G is equal to the determinant of $A + D$.

9.19 The obvious solution using elementary methods is to bucket sort the edge weights in linear time. Then the running time of Kruskal's algorithm is dominated by the *union/find* operations and is $O(|E|\alpha(|E|, |V|))$. The Van-Emde Boas priority queues (see Chapter 6 references) give an immediate $O(|E| \log \log |V|)$ running time for Dijkstra's algorithm, but this isn't even as good as a Fibonacci heap implementation.

More sophisticated priority queue methods that combine these ideas have been proposed, including M. L. Fredman and D. E. Willard, "Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths," *Proceedings of the Thirty-first Annual IEEE Symposium on the Foundations of Computer Science* (1990), 719–725. The paper presents a linear-time minimum spanning tree algorithm and an $O(|E| + |V| \log |V| / \log \log |V|)$ implementation of Dijkstra's algorithm if the edge costs are suitably small. Various improvements have been discovered since.

9.20 Since the minimum spanning tree algorithm works for negative edge costs, an obvious solution is to replace all the edge costs by their negatives and use the minimum spanning tree algorithm. Alternatively, change the logic so that < is replaced by >, *min* by *max*, and vice versa.

9.21 We start the depth-first search at A and visit adjacent vertices alphabetically. The articulation points are C, E, and F. C is an articulation point because $low[B] \geq num[C]$; E is an articulation point because $low[H] \geq num[E]$; and F is an articulation point because $low[G] \geq num[F]$. The depth-first spanning tree is shown in the following Figure.



9.22 The only difficult part is showing that if some nonroot vertex a is an articulation point, then there is no back edge between any proper descendent of a and a proper ancestor of a in the depth-first spanning tree. We prove this by a contradiction.

Let u and v be two vertices such that every path from u to v goes through a. At least one of u and v is a proper descendent of a, since otherwise there is a path from u to v that avoids a. Assume without loss of generality that u is a proper descendent of a. Let c be the child of a that contains u as a descendent. If there is no back edge between a descendent of c and a proper ancestor of a, then the theorem is true immediately, so suppose for the sake of contradiction that there is a back edge (s, t). Then either v is a proper descendent of a or it isn't. In the second case, by taking a path from u to s to t to v, we can avoid a, which is a contradiction. In the first case, clearly v cannot be a descendent of c, so let c' be the child of a that contains v as a descendent. By a similar argument as before, the only possibility is that there is a back edge (s', t') between a descendent of c' and a proper ancestor of a. Then there is a path from u to s to t to t' to s' to v; this path avoids a, which is also a contradiction.

9.23 **(a)** Do a depth-first search and count the number of back edges.

(b) This is the feedback edge set problem. See reference [1] or [20].

9.24 Let (v, w) be a cross edge. Since at the time w is examined it is already marked, and w is not a descendent of v (else it would be a forward edge), processing for w is already complete when processing for v commences. Thus under the convention that trees (and subtrees) are placed left to right, the cross edge goes from right to left.

9.25 Suppose the vertices are numbered in preorder and postorder.

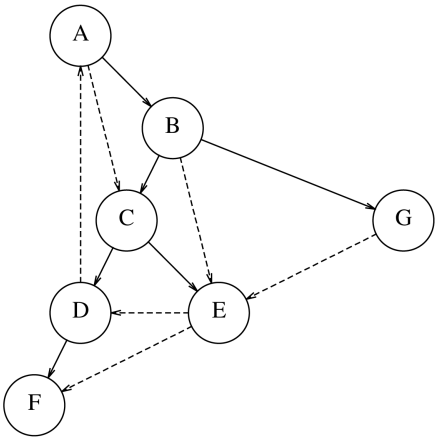
If (v, w) is a tree edge, then v must have a smaller preorder number than w. It is easy to see that the converse is true.

If (v, w) is a cross edge, then v must have both a larger preorder and postorder number than w. The converse is shown as follows: Because v has a larger preorder number, w cannot be a descendent

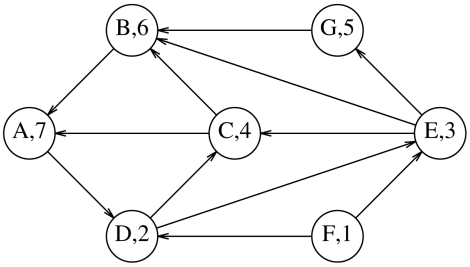
of v; because it has a larger postorder number, v cannot be a descendent of w; thus they must be in different trees.

Otherwise, v has a larger preorder number but is not a cross edge. To test if (v, w) is a back edge, keep a stack of vertices that are active in the depth-first search call (that is, a stack of vertices on the path from the current root). By keeping a bit array indicating presence on the stack, we can easily decide if (v, w) is a back edge or a forward edge.

9.26 The first depth-first spanning tree is



G_r, with the order in which to perform the second depth-first search, is shown next. The strongly connected components are F and all other vertices.



9.28 This is the algorithm mentioned in the references.

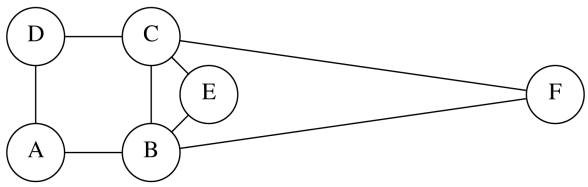
9.29 Because an edge (v, w) is implicitly processed, it is placed on a stack. If v is determined to be an articulation point because $low[w] \geq num[v]$, then the stack is popped until edge (v, w) is removed: The set of popped edges is a biconnected component. An edge (v, w) is not placed on the stack if the edge (w, v) was already processed as a back edge.

9.30 Let (u, v) be an edge of the breadth-first spanning tree. (u, v) are connected, thus they must be in the same tree. Let the root of the tree be r; if the shortest path from r to u is d_u, then u is at level d_u; likewise, v is at level d_v. If (u, v) were a back edge, then $d_u > d_v$, and v is visited before u. But if there were an edge between u and v, and v is visited first, then there would be a tree edge (v, u), and not a back edge (u, v). Likewise, if (u, v) were a forward edge, then there is some w, distinct from u and v, on the path from u to v; this contradicts the fact that $d_v = d_w + 1$. Thus only tree edges and cross edges are possible.

9.31 Perform a depth-first search. The return from each recursive call implies the edge traversal in the opposite direction. The time is clearly linear.

9.33 If there is an Euler circuit, then it consists of entering and exiting nodes; the number of entrances clearly must equal the number of exits. If the graph is not strongly connected, there cannot be a cycle connecting all the vertices. To prove the converse, an algorithm similar in spirit to the undirected version can be used.

9.34 Neither of the proposed algorithms works. For example, as shown, a depth-first search of a biconnected graph that follows A, B, C, D is forced back to A, where it is stranded.



9.35 These are classic graph theory results. Consult any graph theory text for a solution to this exercise.

9.36 All the algorithms work without modification for multigraphs.

9.37 Obviously, G must be connected. If each edge of G can be converted to a directed edge and produce a strongly connected graph G', then G is *convertible*.

Then, if the removal of a single edge disconnects G, G is not convertible, since this would also disconnect G'. This is easy to test by checking to see if there are any single-edge biconnected components.

Otherwise, perform a depth-first search on G and direct each tree edge away from the root and each back edge toward the root. The resulting graph is strongly connected because, for any vertex v, we can get to a higher level than v by taking some (possibly 0) tree edges and a back edge. We can apply this until we eventually get to the root, and then follow tree edges down to any other vertex.

9.38 **(b)** Define a graph where each stick is represented by a vertex. If stick S_i is above S_j and thus must be removed first, then place an edge from S_i to S_j. A legal pick-up ordering is given by a topological sort; if the graph has a cycle, then the sticks cannot be picked up.

9.39 Use a depth-first search, marking colors when a new vertex is visited, starting at the root, and returning false if a color clash is detected along a backedge.

9.40 Use a greedy algorithm: At each step, choose the vertex with the highest connectivity to vertices not already chosen. Proof that only 1/4 of the edges are not touched can be done with a straightforward calculation of edges that remain after each step. With E edges and V vertices, the total connectivity is $2E$ at the start, so at most $(V-2)/V$ of the edges remain after the first step. Then at most $(V-3)/(V-1)$ of those edges remain after the second step, and so on. Multiply it out, and you get about 1/4 of the edges remaining.

9.41 If no vertex has indegree 0, we can find a cycle by tracing backwards through vertices with positive indegree; since every vertex on the trace back has a positive indegree, we eventually reach a vertex twice, and the cycle has been found.

9.42 The basic idea is that we examine cell $A[s, t]$: If it is 0, then t is not a sink; otherwise s is not a sink. Each examination can eliminate one vertex; after $|V|-1$ queries, we have one candidate that can be checked in $2|V|-1$ additional tests. To do the first step, after the test $A[s, t]$ is performed, replace the eliminated vertex with the next vertex in the the sequence of vertices.

9.43 Perform a postorder traversal, computing node sizes; the first node that has a size greater than $N/2$ vertices is the node to remove.

9.44 This is essentially the algorithm described in Section 9.3.4 for critical paths.

9.45 These are all recursive depth first searches.

9.46 This is a single source unweighted shortest path problem.

9.47 This is a weighted shortest path problem: Adjacent squares have a cost of 1 if there is no wall, and $P + 1$ if there is a wall.

9.48 **(a)** Use the same ideas from the shortest path algorithm, but place unmarked adjacent squares at the front of a double-ended queue, and unmarked adjacent wall-separate squares at the end. Note that when *dequeue*s are performed from the front, a stale square (in other words, one that was marked after it entered the deque) may emerge and would be ignored.

(b) Use Exercise 9.47, with a penalty equal to the number of squares in the maze.

9.49 The following implementation does not use the map class. The use of ArrayList instead will speed up the algorithm since access now takes $O(1)$ instead of $O(\log N)$, if the list of words can fit in main memory. One could use maps to also solve the problem. Using maps with keys and data both as strings allows the maps to store adjacent words in the word ladder directly. The Chapter 4 code can be modified for this problem by just adding words that differ in length by one then checking to see if the cost should be 1 or p when Dijkstra's algorithm is applied.

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


public class WordLadder
{

  static final int INFINITY = 99999;

  private static class Vertex
  {
      Vertex()
      {
        adj = new ArrayList<Integer>();
        weight = new ArrayList<Integer>();
      }
      public ArrayList<Integer> adj;        // Adjacency list
      public ArrayList<Integer> weight;     // Weight list
      public boolean  known;
      public int    dist;
      public String   name;
      public int    path;
  };

  /**
   * Print shortest path to v after dijkstra has run.
```

```
 * Assume that the path exists.
 */
static void printPath( int vIndex, ArrayList<Vertex> V )
{
  if( vIndex >= 0 && V.get(vIndex).path > -1 )
  {
     printPath( V.get(vIndex).path, V );
     System.out.println(" to ");
  }
 System.out.println(V.get(vIndex).name);
}

static void dijkstra( int sIndex, int tIndex, ArrayList<Vertex> Vertices )
{
  int smallestDist;
  int smallestVertex;
  Vertex v, s, t;
  int n = Vertices.size();

  Vertices.get(sIndex).dist = 0;

  for ( ; ; )
    {
    smallestDist = INFINITY;
    smallestVertex = -1;

    for( int i=0; i<n; i++)
    {
      if (!Vertices.get(i).known && Vertices.get(i).dist < smallestDist )
      {
        smallestDist = Vertices.get(i).dist;
        smallestVertex = i;
      }
    }

    if( smallestVertex < 0 || smallestVertex == tIndex )
      break;

    Vertices.get(smallestVertex).known = true;
    v = Vertices.get(smallestVertex);

    for ( int j = 0; j < v.adj.size(); j++ )
    {
      if ( !(Vertices.get( v.adj.get(j) ).known) )
      {
        if ( v.dist + v.weight.get(j) < Vertices.get( v.adj.get(j) ).dist )
        {
          // Update w
```

```
          Vertices.get( v.adj.get(j) ).dist = v.dist + v.weight.get(j);
          Vertices.get( v.adj.get(j) ).path = smallestVertex;
        }
      }
    }
  }
}

static ArrayList<Vertex> readWords( Scanner in )
{
  String oneLine = new String();
  ArrayList<Vertex> v = new ArrayList<Vertex>();
  while ( in.hasNext() )
  {
    oneLine = in.next();
    Vertex w = new Vertex();
    w.name = oneLine;
    w.known = false;
    w.path = -1;
    w.dist = INFINITY;
    v.add( w ) ;
  }

  return v;
}

static int oneCharOff( String word1, String word2, int p)
{
  String big, small, shrink;
  int cost;

  if( Math.abs ((int)(word1.length() - word2.length())) > 1 )
    return 0;
  else
  if (word1.length() == word2.length())
  {
    int diffs = 0;

    for( int i = 0; i < word1.length( ); i++ )
      if( word1.charAt(i) != word2.charAt(i) )
        if( ++diffs > 1 )
          return 0;
    if (diffs == 1)
      return 1;
  }

  if (word2.length() > word1.length ())
  {
```

```
      big = word2;
      small = word1;
    }
    else
    {
      big = word1;
      small = word2;
    }

    for (int i = 0; i < big.length(); i++)
    {
      shrink = big.substring(0,i) + big.substring(i+1,big.length());
      if (shrink.compareTo(small) == 0)
        return p;
    }

    return 0;
  }

  // fills the adjacency lists for all words in the dictionary
  static void fillAdjacencies(ArrayList<Vertex> words, int p)
  {
    int cost;

    for (int i = 0 ; i < words.size(); i++)
      for (int j = i+1 ; j < words.size(); j++)
      {
        cost = oneCharOff(words.get(i).name, words.get(j).name,p);
        if (cost > 0)
        {
          words.get(i).adj.add(j);
          words.get(i).weight.add(cost);
          words.get(j).adj.add(i);
          words.get(j).weight.add(cost);
        }
      }
  }

  public static void main(String args[])
  {
    int p=0;
    int w1Index, w2Index;
    String w1 = new String();
    String w2 = new String();
    Scanner input = new Scanner(System.in);
    Scanner dict;
```

```
        try
        {
          dict = new Scanner(new File("dict.txt"));
        }
        catch (Exception e)
        {
          System.out.println("Error opening dictionary file");
          return;
        }

        System.out.println("What is the cost of single char deletions: ");
        p = input.nextInt();

        ArrayList<Vertex> words = readWords( dict );

        do
        {
          System.out.println("Enter two words in the dictionary: ");
          w1 = input.next();
          w2 = input.next();
          // Find their indices (here is where a map would be superior
          // However all other accesses are now in O(1) time
          for (w1Index = 0; w1Index < words.size()
              && (words.get(w1Index).name).compareTo(w1) != 0 ; w1Index++);
          for (w2Index = 0; w2Index < words.size()
              && (words.get(w2Index).name).compareTo(w2) != 0 ; w2Index++);
        } while (w1Index >= words.size() || w2Index >= words.size());

        fillAdjacencies(words, p);        // make the adjacency list
        dijkstra( w1Index, w2Index, words );  // use dijkstra's algorithm
        System.out.println();
        printPath(w2Index, words);        // print the result
        System.out.println();
      }

    }
```

9.50 Each team is a vertex; if X has defeated Y, draw an edge from X to Y.

9.51 Each currency is a vertex; draw an edge of cost $\log C$ between vertices to represent a currency exchange rate, C. A negative cycle represents an arbitrage play.

9.52 Each course is a vertex; if X is a prerequisite for Y, draw an edge from X to Y Find the longest path in the acyclic graph.

9.53 Each actor is a vertex; an edge connects two vertices if the actors have a shared movie role. The graph is not drawn explicitly, but edges can be deduced as needed by examining cast lists.

9.54 Given an instance of clique, form the graph G' that is the *complement graph* of G: (v, w) is an edge in G' if and only if it is not an edge in G. Then G' has a vertex cover of at most $|V| - K$ if G has a clique of size at least K. (The vertices that form the vertex cover are exactly those not in the clique.) The details of the proof are left to the reader.

9.55 A proof can be found in Garey and Johnson [20].

9.56 Clearly, the *baseball card collector problem (BCCP)* is in *NP*, because it is easy to check if K packets contain all the cards. To show it is NP-complete, we reduce vertex cover to it. Let $G = (V, E)$, and let K be an instance of vertex cover. For each vertex v, place all edges adjacent to v in packet P_v. The K packets will contain all edges (baseball cards) if G can be covered by K vertices.

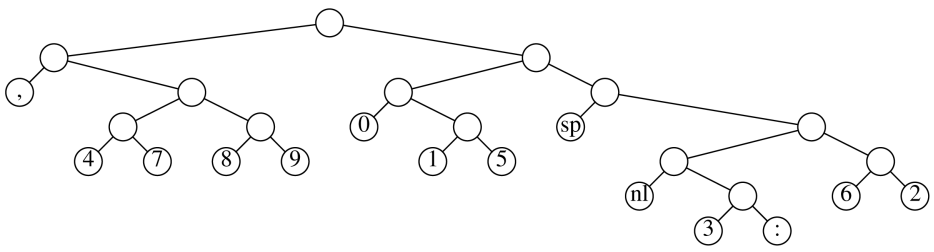
CHAPTER 10

Algorithm Design Techniques

10.1 First, we show that if N evenly divides P, then each of $j_{(i-1)P+1}$ through j_{iP} must be placed as the i^{th} job on some processor. Suppose otherwise. Then in the supposed optimal ordering, we must be able to find some jobs j_x and j_y such that j_x is the t^{th} job on some processor and j_y is the $t + 1^{th}$ job on some processor but $t_x > t_y$. Let j_z be the job immediately following j_x. If we swap j_y and j_z, it is easy to check that the mean processing time is unchanged and thus still optimal. But now j_y follows j_x, which is impossible because we know that the jobs on any processor must be in sorted order from the results of the one processor case.

Let $j_{e1}, j_{e2}, \ldots, j_{eM}$ be the extra jobs if N does not evenly divide P. It is easy to see that the processing time for these jobs depends only on how quickly they can be scheduled and that they must be the last scheduled job on some processor. It is easy to see that the first M processors must have jobs $j_{(i-1)P+1}$ through j_{iP+M}; we leave the details to the reader.

10.3



10.4 One method is to generate code that can be evaluated by a stack machine. The two operations are *push* (the one node tree corresponding to) a symbol onto a stack and *combine*, which pops two trees off the stack, merges them, and pushes the result back on. For the example in the text, the stack instructions are *push(s), push(nl), combine, push(t), combine, push(a), combine, push(e), combine, push(i), push (sp), combine, combine*.

By encoding a *combine* with a 0 and a *push* with a 1 followed by the symbol, the total extra space is $2N - 1$ bits if all the symbols are of equal length. Generating the stack machine code can be done with a simple recursive procedure and is left to the reader.

10.6 Maintain two queues, Q_1 and Q_2. Q_1 will store single-node trees in sorted order, and Q_2 will store multinode trees in sorted order. Place the initial single-node trees on Q_1, enqueueing the smallest weight tree first. Initially, Q_2 is empty. Examine the first two entries of each of Q_1 and Q_2, and dequeue the two smallest. (This requires an easily implemented extension to the ADT.) Merge the tree and place the result at the end of Q_2. Continue this step until Q_1 is empty and only one tree is left in Q_2.

10.9 To implement first fit, we keep track of bins b_i, which have more room than any of the lower numbered bins. A theoretically easy way to do this is to maintain a splay tree ordered by empty space. To insert w, we find the smallest of these bins, which has at least w empty space; after w is added to the bin, if the resulting amount of empty space is less than the inorder predecessor in the tree, the entry can be removed; otherwise, a *decreaseKey* is performed.

To implement best fit, we need to keep track of the amount of empty space in each bin. As before, a splay tree can keep track of this. To insert an item of size w, perform an insert of w. If there is a bin that can fit the item exactly, the insert will detect it and splay it to the root; the item can be added and the root deleted. Otherwise, the insert has placed w at the root (which eventually needs to be removed). We find the minimum element M in the right subtree, which brings M to the right subtree's root, attach the left subtree to M, and delete w. We then perform an easily implemented *decreaseKey* on M to reflect the fact that the bin is less empty.

10.10 *Next fit: 12 bins* (.42, .25, .27), (.07, .72), (.86, .09), (.44, .50), (.68), (.73), (.31), (.78, .17), (.79), (.37), (.73, .23), (.30).
First fit: 10 bins (.42, .25, .27), (.07, .72, .09), (.86), (.44, .50), (.68, .31), (.73, .17), (.78), (.79), (.37, .23, .30), (.73).
Best fit: 10 bins (.42, .25, .27), (.07, .72, .09), (.86), (.44, .50), (.68, .31), (.73, .23), (.78, .17), (.79), (.37, .30), (.73).
First fit decreasing: 10 bins (.86, .09), (.79, .17), (.78, .07), (.73, .27), (.73, .25), (.72, .23), (.68, .31), (.50, .44), (.42, .37), (.30).
Best fit decreasing: 10 bins (.86, .09), (.79, .17), (.78), (.73, .27), (.73, .25), (.72, .23), (.68, .31), (.50, .44), (.42, .37, .07), (.30).

Note that use of 10 bins is optimal.

10.12 We prove the second case, leaving the first and third (which give the same results as Theorem 10.6) to the reader. Observe that

$$\log^p N = \log^p (b^m) = m^p \log^p b$$

Working this through, Equation (10.9) becomes

$$T(N) = T(b^m) = a^m \sum_{i=0}^{m} \left\{ \frac{b^k}{a} \right\}^i i^p \log^p b$$

If $a = b^k$, then

$$\begin{aligned} T(N) &= a^m \log^p b \sum_{i=0}^{m} i^p \\ &= O(a^m m^{p+1} \log^p b) \end{aligned}$$

Since $m = \log N / \log b$, and $a^m = N^k$, and b is a constant, we obtain

$$T(N) = O(N^k \log^{p+1} N)$$

10.13 The easiest way to prove this is by an induction argument.

10.14 Divide the unit square into $N - 1$ square grids each with side $1/\sqrt{N} - 1$. Since there are N points, some grid must contain two points. Thus the shortest distance is conservatively given by at most $\sqrt{2/(N-1)}$.

10.15 The results of the previous exercise imply that the width of the strip is $O(1/\sqrt{N})$. Because the width of the strip is $O(1/\sqrt{N})$, and thus covers only $O(1/\sqrt{N})$ of the area of the square, we expect a similar fraction of the points to fall in the strip. Thus only $O(N/\sqrt{N})$ points are expected in the strip; this is $O(\sqrt{N})$.

10.17 The recurrence works out to

$$T(N) = T(2N/3) + T(N/3) + O(N)$$

This is not linear, because the sum is not less than one. The running time is $O(N \log N)$.

10.18 The recurrence for median-of-median-of-seven partitioning is

$$T(N) = T(5N/7) + T(N/7) + O(N)$$

If all we are concerned about is the linearity, then median-of-median-of-seven can be used.

10.21 We derive the values of s and δ, following the style in the original paper [17]. Let $R_{t,X}$ be the rank of element t in some sample X. If a sample S' of elements is chosen randomly from S, and $|S'| = s$, $|S| = N$, then we've already seen that

$$E(R_{t,S}) = \frac{N+1}{s+1} R_{t,S'}$$

where E means expected value. For instance, if t is the third largest in a sample of 5 elements, then in a group of 19 elements it is expected to be the tenth largest. We can also calculate the variance:

$$V(R_{t,S}) = \sqrt{\frac{(R_{t,S})(s - R_{t,S} + 1)(N+1)(N-s)}{(s+1)^2(s+2)}} = O(N/\sqrt{s})$$

We choose v_1 and v_2 so that

$$E(R_{v_1,S}) + 2dV(R_{v_1,S}) \approx k \approx E(R_{v_2,S}) - 2dV(R_{v_2,S})$$

where d indicates how many variances we allow. (The larger d is, the less likely the element we are looking for will not be in S'.)

The probability that k is not between v_1 and v_2 is

$$2\int_d^\infty erf(x)dx = O(e^{-d^2}/d)$$

If $d = \log^{1/2} N$, then this probability is $o(1/N)$, specifically $O(1/(N \log N))$. This means that the expected work in this case is $O(\log^{-1} N)$ because $O(N)$ work is performed with very small probability.

These mean and variance equations imply

$$R_{v_1,S'} \geq k\frac{(s+1)}{(N+1)} - d\sqrt{s}$$

and

$$R_{v_2,S'} \leq k\frac{(s+1)}{(N+1)} + d\sqrt{s}$$

This gives equation (A):

$$\delta = d\sqrt{s} = \sqrt{s}\log^{1/2} N \tag{A}$$

If we first pivot around v_2, the cost is N comparisons. If we now partition elements in S that are less than v_2 around v_1, the cost is $R_{v_2,S}$, which has expected value $k + \delta\frac{N+1}{s+1}$. Thus the total cost of partitioning is $N + k + \delta\frac{N+1}{s+1}$. The cost of the selections to find v_1 and v_2 in the sample S' is $O(s)$. Thus the total expected number of comparisons is

$$N + k + O(s) + O(N\delta/s)$$

The low order term is minimized when

$$s = N\delta/s \tag{B}$$

Combining Equations (A) and (B), we see that

$$s^2 = N\delta = \sqrt{s}N \log^{1/2} N \quad \text{(C)}$$

$$s^{3/2} = N \log^{1/2} N \quad \text{(D)}$$

$$s = N^{2/3} \log^{1/3} N \quad \text{(E)}$$

$$\delta = N^{1/3} \log^{2/3} N \quad \text{(F)}$$

10.22 First, we calculate 12*43. In this case, $X_L = 1$, $X_R = 2$, $Y_L = 4$, $Y_R = 3$, $D_1 = -1$, $D_2 = -1$, $X_L Y_L = 4$, $X_R Y_R = 6$, $D_1 D_2 = 1$, $D_3 = 11$, and the result is 516.

Next, we calculate 34*21. In this case, $X_L = 3$, $X_R = 4$, $Y_L = 2$, $Y_R = 1$, $D_1 = -1$, $D_2 = -1$, $X_L Y_L = 6$, $X_R Y_R = 4$, $D_1 D_2 = 1$, $D_3 = 11$, and the result is 714.

Third, we calculate 22*22. Here, $X_L = 2$, $X_R = 2$, $Y_L = 2$, $Y_R = 2$, $D_1 = 0$, $D_2 = 0$, $X_L Y_L = 4$, $X_R Y_R = 4$, $D_1 D_2 = 0$, $D_3 = 8$, and the result is 484.

Finally, we calculate 1234*4321. $X_L = 12$, $X_R = 34$, $Y_L = 43$, $Y_R = 21$, $D_1 = -22$, $D_2 = -2$. By previous calculations, $X_L Y_L = 516$, $X_R Y_R = 714$, and $D_1 D_2 = 484$. Thus $D_3 = 1714$, and the result is 714 + 171400 + 5160000 = 5332114.

10.23 The multiplication evaluates to $(ac - bd) + (bc + ad)i$. Compute ac, bd, and $(a - b)(d - c) + ac + bd$.

10.24 The algebra is easy to verify. The problem with this method is that if X and Y are positive N bit numbers, their sum might be an $N + 1$ bit number. This causes complications.

10.26 Matrix multiplication is not commutative, so the algorithm couldn't be used recursively on matrices if commutativity was used.

10.27 If the algorithm doesn't use commutativity (which turns out to be true), then a divide and conquer algorithm gives a running time of $O(N^{\log_{70} 143640}) = O(N^{2.795})$.

10.28 1150 scalar multiplications are used if the order of evaluation is

$$((A_1 A_2)(((A_3 A_4)A_5)A_6))$$

10.29 **(a)** Let the chain be a 1×1 matrix, a $1 \times A$ matrix, and an $A \times B$ matrix. Multiplication by using the first two matrices first makes the cost of the chain $A + AB$. The alternative method gives a cost of $AB + B$, so if $A > B$, then the algorithm fails. Thus a counterexample is multiplying a 1×1 matrix by a 1×3 matrix by a 3×2 matrix.

(b, c) A counterexample is multiplying a 1×1 matrix by a 1×2 matrix by a 2×3 matrix.

10.31 The optimal binary search tree is the same one that would be obtained by a greedy strategy: *I* is at the root and has children *and* and *it*; *a* and *or* are leaves; the total cost is 2.14.

10.33 This theorem is from F. Yao's paper, reference [60].

10.34 A recursive procedure is clearly called for; if there is an intermediate vertex, *stopOver* on the path from *s* to *t*, then we want to print out the path from *s* to *stopOver* and then *stopOver* to *t*. We don't want to print out *stopOver* twice, however, so the method does not print out the first or last vertex on the path and reserves that for the driver.

```
// Print the path between s and t, except do not print
// the first or last vertex.  Print a trailing " to " only.
static void printPath1( int [][] path, int s, int t )
{
  int stopOver = path[ s ][ t ];
  if( s != t && stopOver != -1 )
  {
```

```
    printPath1( path, s, stopOver );
    System.out.print( "" + stopOver + " to " );
    printPath1( path, stopOver, t );
  }
}

// Assume the existence of a path of length at least 1
static void printPath( int [][] path, int s, int t )
{
  System.out.print( "" + s + " to " );
  printPath1( path, s, t );
  System.out.println( t );
}
```

10.38 If the modulus is a power of two, then the least significant bit of the "random" number oscillates. Thus *flip* will always return heads and tails alternately, and the level chosen for a skip list insertion will always be one or two. Consequently, the performance of the skip list will be $\Theta(N)$ per operation.

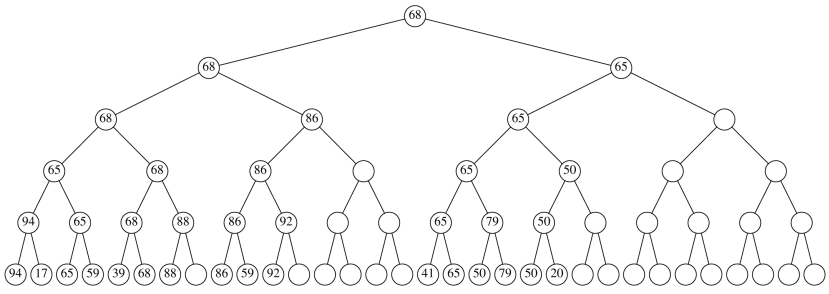
10.39 **(a)** $2^5 \equiv 32 \bmod 341$, $2^{10} \equiv 1 \bmod 341$. Since $32^2 \equiv 1 \bmod 341$, this proves that 341 is not prime. We can also immediately conclude that $2^{340} \equiv 1 \bmod 341$ by raising the last equation to the 34^{th} power. The exponentiation would continue as follows: $2^{20} \equiv 1 \bmod 341$, $2^{21} \equiv 2 \bmod 341$, $2^{42} \equiv 4 \bmod 341$, $2^{84} \equiv 16 \bmod 341$, $2^{85} \equiv 32 \bmod 341$, $2^{170} \equiv 1 \bmod 341$, and $2^{340} \equiv 1 \bmod 341$.

(b) If $A = 2$, then although $2^{560} \equiv 1 \bmod 561$, $2^{280} \equiv 1 \bmod 561$ proves that 561 is not prime. If $A = 3$, then $3^{561} \equiv 375 \bmod 561$, which proves that 561 is not prime. $A = 4$ obviously doesn't fool the algorithm, since $4^{140} \equiv 1 \bmod 561$. $A = 5$ fools the algorithm: $5^1 \equiv 5 \bmod 561$, $5^2 \equiv 25 \bmod 561$, $5^4 \equiv 64 \bmod 561$, $5^8 \equiv 169 \bmod 561$, $5^{16} \equiv 511 \bmod 561$, $5^{17} = 311 \bmod 561$, $5^{34} \equiv 229 \bmod 561$, $5^{35} \equiv 23 \bmod 561$, $5^{70} \equiv 529 \bmod 561$, $5^{140} \equiv 463 \bmod 561$, $5^{280} \equiv 67 \bmod 561$, $5^{560} \equiv 1 \bmod 561$.

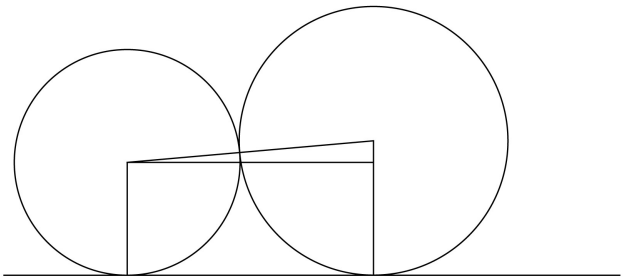
10.41 The two point sets are 0, 4, 6, 9, 16, 17 and 0, 5, 7, 13, 16, 17.

10.42 To find all point sets, we backtrack even if *found* == *true*, and print out information when line 2 is executed. In the case where there are some duplicate distances, it is possible that several symmetries will be printed.

10.43



10.46 We place circles in order. Suppose we are trying to place circle j of radius r_j. If some circle i of radius r_i is centered at x_i, then j is tangent to i if it is placed at $x_i + 2\sqrt{r_i r_j}$. To see this, notice that the line connecting the centers has length $r_i + r_j$, and the difference in y-coordinates of the centers is $|r_j - r_i|$. The difference in x-coordinates follows from the Pythagorean theorem.



To place circle j, we compute where it would be placed if it were tangent to each of the first $j-1$ circles, selecting the maximum value. If this value is less than r_j, then we place circle j at x_j. The running time is $O(N^2)$.

10.47 Construct a minimum spanning tree T of G, pick any vertex in the graph, and then find a path in T that goes through every edge exactly once in each direction. (This is done by a depth-first search; see Exercise 9.31.) This path has twice the cost of the minimum spanning tree, but it is not a simple cycle.

Make it a simple cycle, without increasing the total cost, by bypassing a vertex when it is seen a second time (except that if the start vertex is seen, close the cycle) and going to the next unseen vertex on the path, possibly bypassing several vertices in the process. The cost of this direct route cannot be larger than original because of the triangle inequality.

If there were a tour of cost K, then by removing one edge on the tour, we would have a minimum spanning tree of cost less than K (assuming that edge weights are positive). Thus the minimum spanning tree is a lower bound on the optimal traveling salesman tour. This implies that the algorithm is within a factor of 2 of optimal.

10.48 If there are two players, then the problem is easy, so assume $k > 1$. If the players are numbered 1 through N, then divide them into two groups: 1 through $N/2$ and $N/2+1$ though N. On the i^{th} day, for $1 \le i \le N/2$, player p in the second group plays players $((p+i) \bmod N/2)+1$ in the first group. Thus after $N/2$ days, everyone in group 1 has played everyone in group 2. In the last $N/2-1$ days, recursively conduct round-robin tournaments for the two groups of players.

10.49 Divide the players into two groups of size $\lfloor N/2 \rfloor$ and $\lceil N/2 \rceil$, respectively, and recursively arrange the players in any order. Then merge the two lists (declare that $p_x > p_y$ if x has defeated y, and $p_y > p_x$ if y has defeated x; exactly one is possible) in linear time as is done in mergesort.

10.51 Divide and conquer algorithms (among others) can be used for both problems, but neither is trivial to implement. See the computational geometry references for more information.

10.52 **(a)** Use dynamic programming. Let $S_k =$ the best setting of words w_k, w_{k+1}, . . . w_N, $U_k =$ the ugliness of this setting, and $l_k =$ for this setting, (a link to) the word that starts the second line.

To compute S_{k-1}, try putting w_{k-1}, w_k, . . . , w_M all on the first line for $k < M$ and $\sum_{i=k-1}^{M} w_i < L$. Compute the ugliness of each of these possibilities by, for each M, computing the ugliness of setting the first line and adding U_{m+1}. Let M' be the value of M that yields the minimum ugliness. Then $U_{k-1} =$ this value, and $l_{k-1} = M' + 1$. Compute values of U and l starting with the last word and working back to the first. The minimum ugliness of the paragraph is U_1; the actual setting can be found by starting at l_1 and following the links in l since this will yield the first word on each line.

(b) The running time is quadratic in the case where the number of words that can fit on a line is consistently $\Theta(N)$. The space is linear to keep the arrays U and l. If the line length is restricted to some constant, then the running time is linear because only $O(1)$ words can go on a line.

(c) Put as many words on a line as can fit. This clearly minimizes the number of lines, and hence the ugliness, as can be shown by a simple calculation.

10.53 An obvious $O(N^2)$ solution to construct a graph with vertices $1, 2, \ldots, N$ and place an edge (v, w) in G iff $a_v < a_w$. This graph must be acyclic, thus its longest path can be found in time linear in the number of edges; the whole computation thus takes $O(N^2)$ time.

Let $BEST(k)$ be the increasing subsequence of exactly k elements that has the minimum last element. Let t be the length of the maximum increasing subsequence. We show how to update $BEST(k)$ as we scan the input array. Let $LAST(k)$ be the last element in $BEST(k)$. It is easy to show that if $i < j$, $LAST(i) < LAST(j)$. When scanning a_M, find the largest k such that $LAST(k) < a_M$. This scan takes $O(\log t)$ time because it can be performed by a binary search. If $k = t$, then x_M extends the longest subsequence, so increase t, and set $BEST(t)$ and $LAST(t)$ (the new value of t is the argument). If k is 0 (that is, there is no k), then x_M is the smallest element in the list so far, so set $BEST(1)$ and $LAST(1)$, appropriately. Otherwise, we extend $BEST(k)$ with x_M, forming a new and improved sequence of length $k + 1$. Thus we adjust $BEST(k + 1)$ and $LAST(k + 1)$.

Processing each element takes logarithmic time, so the total is $O(N \log N)$.

10.54 This is a classic program using dynamic programming. Let the matrix $\mathbf{M}[i][j]$ be the integer matrix that contains the length of the longest common subsequence between the first i letters of A and the first j letters of B. Since a string with no letters forms no matches, we have that $M[0][j] = 0$ and $M[i][0] = 0$. Further, if i^{th} letter of A matches the j^{th} letter of B then we have that $\mathbf{M}[i1][j1] = 1 + \mathbf{M}[i - 1][j - 1]$. If there is not a match, then $\mathbf{M}[i][j] = \max(\mathbf{M}[i][j - 1], \mathbf{M}[i - 1][j])$. This then forms the basis for the dynamic programming algorithm for finding the length of the longest common subsequence. To find the actual sequence itself, we have to trace back along the matrix.

```
import java.util.*;

public class CommonLetters
{

  public static String lcs(String w1, String w2)
  {
    int i, j, n, m;
    String common = new String();
    n = w1.length();
    m = w2.length();

    Integer M[][] = new Integer[n+1][m+1];

    for (i=0; i<=n; i++)
       M[i][0] = 0;
    for (j=0; j<=m; j++)
       M[0][j] = 0;

    for (j=1; j<=m; j++)
       for (i = 1; i<=n; i++)
```

```
            {
               if ( w1.charAt(i-1) == w2.charAt(j-1) )
                  M[i][j] = 1 + M[i-1][j-1];
               else
                  M[i][j] = Math.max(M[i][j-1], M[i-1][j]);
            }

          // Now the trace back code
          i = n;
          j = m;
          while (i > 0 && j > 0)
          {
             if (M[i][j] == 1 + M[i-1][j-1]
                && w1.charAt(i-1) == w2.charAt(j-1) ) // there was a match
             {
               common = w1.charAt(i-1)+common;
               i--;
               j--;
             }
             else if (M[i-1][j] > M[i][j-1])
               i--;
             else
               j--;
          }
          return common;
       }

       public static void main(String args[])
       {
          String w1 = new String("dynamic");
          String w2 = new String("programming");

          String w3 = lcs(w1, w2);

          System.out.println(w3);
       }
    }
```

10.56 **(a)** A dynamic programming solution resolves part (a). Let $FITS(i, s)$ be 1 if a subset of the first i items sums to exactly s; $FITS(i, 0)$ is always 1. Then $FITS(x, t)$ is 1 if either $FITS(x - 1, t - a_x)$ or $FITS(x - 1, t)$ is 1, and 0 otherwise.

(b) This doesn't show that $P = NP$, because the *size* of the problem is a function of N and $\log K$. Only $\log K$ bits are needed to represent K; thus an $O(NK)$ solution is exponential in the input size.

10.57 **(a)** Let the minimum number of coins required to give x cents in change be $COIN(x)$; $COIN(0) = 0$. Then $COIN(x)$ is one more than the minimum value of $COIN(x - c_i)$, giving a dynamic programming solution.

(b) Let $WAYS(x, i)$ be the number of ways to make x cents in change without using the first i coin types. If there are N types of coins, then $WAYS(x, N) = 0$ if $x! = 0$, and $WAYS(0, i) = 1$. Then $WAYS(x, i - 1)$ is equal to the sum of $WAYS(x - pc_i, i)$, for integer values of p no larger than x/c_i (but including 0).

10.59 **(a)** Place eight queens randomly on the board, making sure that no two are on the same row or column. This is done by generating a random permutation of 1..8. There are only 5040 such permutations, and 92 of these give a solution.

10.60 **(a)** Since the knight leaves every square once, it makes B^2 moves. If the squares are alternately colored black and white like a checkerboard, then a knight always moves to a different colored square. If B is odd, then so is B^2, which means that at the end of the tour the knight will be on a different colored square than at the start of the tour. Thus the knight cannot be at the original square.

10.61 **(a)** If the graph has a cycle, then the recursion does not always make progress toward a base case, and thus an infinite loop will result.

(b) If the graph is acyclic, the recursive call makes progress, and the algorithm terminates. This could be proved formally by induction.

(c) This algorithm is exponential.

10.62 **(a)** A simple dynamic programming algorithm, in which we scan in row order suffices. Each square computes the maximum sized square for which it could be in the lower right corner. That information is easily obtained from the square that is above it and the square that is to its left.

(b) See the April 1998 issue of *Dr. Dobb's Journal* for an article by David Vandevoorde on the maximal rectangle problem.

10.67 The final score is 20-16; black wins.

CHAPTER 11

Amortized Analysis

11.1 When the number of trees after the insertions is more than the number before.

11.2 Although each insertion takes roughly $\log N$, and each *deleteMin* takes $2 \log N$ actual time, our accounting system is charging these particular operations as 2 for the insertion and $3 \log N - 2$ for the *deleteMin*. The total time is still the same; this is an accounting gimmick. If the number of insertions and *deleteMins* are roughly equivalent, then it really is just a gimmick and not very meaningful; the bound has more significance if, for instance, there are N insertions and $O(N/\log N)$ *deleteMins* (in which case, the total time is linear).

11.3 Insert the sequence $N, N+1, N-1, N+2, N-2, N+3, \ldots, 1, 2N$ into an initially empty skew heap. The right path has N nodes, so any operation could take $\Omega(N)$ time.

11.5 We implement *decreaseKey(x)* as follows: If lowering the value of x creates a heap order violation, then cut x from its parent, which creates a new skew heap H_1 with the new value of x as a root, and also makes the old skew heap (H) smaller. This operation might also increase the potential of H, but only by at most $\log N$. We now merge H and H_1. The total amortized time of the *merge* is $O(\log N)$, so the total time of the *decreaseKey* operation is $O(\log N)$.

11.8 For the *zig-zig* case, the actual cost is 2, and the potential change is $R_f(X) + R_f(P) + R_f(G) - R_i(X) - R_i(P) - R_i(G)$. This gives an amortized time bound of

$$AT_{\text{zig-zig}} = 2 + R_f(X) + R_f(P) + R_f(G) - R_i(X) - R_i(P) - R_i(G)$$

Since $R_f(X) = R_i(G)$, this reduces to

$$= 2 + R_f(P) + R_f(G) - R_i(X) - R_i(P)$$

Also, $R_f(X) > R_f(P)$ and $R_i(X) < R_i(P)$, so

$$AT_{\text{zig-zig}} < 2 + R_f(X) + R_f(G) - 2R_i(X)$$

Since $S_i(X) + S_f(G) < S_f(X)$, it follows that $R_i(X) + R_f(G) < 2R_f(X) - 2$. Thus

$$AT_{\text{zig-zig}} < 3R_f(X) - 3R_i(X)$$

11.9 **(a)** Choose $W(i) = 1/N$ for each item. Then for any access of node X, $R_f(X) = 0$, and $R_i(X) \geq -\log N$, so the amortized access for each item is at most $3 \log N + 1$, and the net potential drop over the sequence is at most $N \log N$, giving a bound of $O(M \log N + M + N \log N)$, as claimed.

(b) Assign a weight of q_i/M to items i. Then $R_f(X) = 0$, $R_i(X) \geq \log(q_i/M)$, so the amortized cost of accessing item i is at most $3 \log(M/q_i) + 1$, and the theorem follows immediately.

11.10 **(a)** To merge two splay trees T_1 and T_2, we access each node in the smaller tree and insert it into the larger tree. Each time a node is accessed, it joins a tree that is at least twice as large; thus a node

can be inserted $\log N$ times. This tells us that in any sequence of $N - 1$ merges, there are at most $N \log N$ inserts, giving a time bound of $O(N \log^2 N)$. This presumes that we keep track of the tree sizes. Philosophically, this is ugly since it defeats the purpose of self-adjustment.

(b) Port and Moffet [6] suggest the following algorithm: If T_2 is the smaller tree, insert its root into T_1. Then recursively merge the left subtrees of T_1 and T_2, and recursively merge their right subtrees. This algorithm is not analyzed; a variant in which the median of T_2 is splayed to the root first is with a claim of $O(N \log N)$ for the sequence of merges.

11.11 The potential function is c times the number of insertions since the last rehashing step, where c is a constant. For an insertion that doesn't require rehashing, the actual time is 1, and the potential increases by c, for a cost of $1 + c$.

If an insertion causes a table to be rehashed from size S to $2S$, then the actual cost is $1 + dS$, where dS represents the cost of initializing the new table and copying the old table back. A table that is rehashed when it reaches size S was last rehashed at size $S/2$, so $S/2$ insertions had taken place prior to the rehash, and the initial potential was $cS/2$. The new potential is 0, so the potential change is $-cS/2$, giving an amortized bound of $(d - c/2)S + 1$. We choose $c = 2d$, and obtain an $O(1)$ amortized bound in both cases.

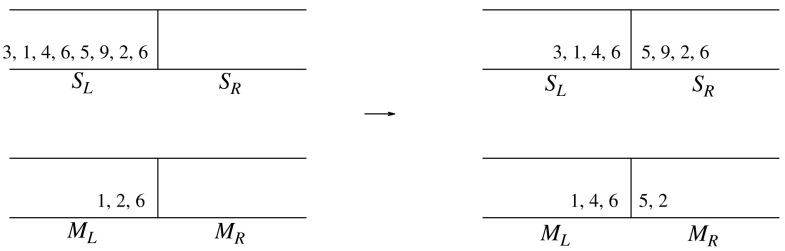
11.12 We can (inductively) take a Fibonacci heap consisting of a single degenerate tree that extends as deep as possible. Insert three very small items; do a *deleteMin* to force a merge. This leaves two of the newer small items, one as a root, the other as a child of the root. Then do a *decreaseKey* on that child, and then a *deleteMin*; now we have a longer degenerate tree. Thus the worst case is $O(N)$ depth.

11.13 **(a)** This problem is similar to Exercise 3.25. The first four operations are easy to implement by placing two stacks, S_L and S_R, next to each other (with bottoms touching). We can implement the fifth operation by using two more stacks, M_L and M_R (which hold minimums).

If both S_L and S_R never empty, then the operations can be implemented as follows:

push(x): push x onto S_L; if X is smaller than or equal to the top of M_L, push x onto M_L as well.
inject(x): same operation as *push*, except use S_R and M_R.
pop(): pop S_L; if the popped item is equal to the top of M_L, then pop M_L as well.
eject(): same operation as *pop*, except use S_R and M_R.
findMin(): return the minimum of the top of M_L and M_R.

These operations don't work if either S_L or S_R is empty. If a *pop* or *eject* is attempted on an empty stack, then we clear M_L and M_R. We then redistribute the elements so that half are in S_L and the rest in S_R, and adjust M_L and M_R to reflect what the state would be. We can then perform the *pop* or *eject* in the normal fashion. The following figure shows a transformation.



Define the potential function to be the absolute value of the number of elements in S_L minus the number of elements in S_R. Any operation that doesn't empty S_L or S_R can increase the potential by only 1; since the actual time for these operations is constant, so is the amortized time.

To complete the proof, we show that the cost of a reorganization is $O(1)$ amortized time. Without loss of generality, if S_R is empty, then the actual cost of the reorganization is $|S_L|$ units. The potential before the reorganization is $|S_L|$; afterward, it is at most 1. Thus the potential change is $1 - |S_L|$, and the amortized bound follows.

11.15 No; a counterexample is that we alternate between finding a node at the root (with splay) and a node that is very deep (without splay).

11.16 The maximum potential is $O(N \log N)$ for a splay tree of maximum depth. The minimum potential is $O(N)$ for a balanced splay tree (the logic is similar to that used in analyzing the *fixHeap* operation for binary heaps.) The potential can decrease by at most $O(N \log N)$, on that tree (half the nodes lose all their potential). The potential can only increase by $O(\log N)$, based on the splay tree amortized bound.

11.17 **(a)** $O(N \log N)$.

(b) $O(N \log \log N)$.

CHAPTER 12

Advanced Data Structures and Implementation

12.3 Incorporate an additional field for each node that indicates the size of its subtree. These fields are easy to update during a splay. This is difficult to do in a skip list.

12.6 If there are B black nodes on the path from the root to all leaves, it is easy to show by induction that there are at most 2^B leaves. Consequently, the number of black nodes on a path is at most $\log N$. Since there can't be two consecutive red nodes, the height is bounded by $2 \log N$.

12.7 Color nonroot nodes red if their height is even and their parents height is odd, and black otherwise. Not all red black trees are AVL trees (since the deepest red black tree is deeper than the deepest AVL tree).

12.19 See H. N. Gabow, J. L. Bentley, and R. E. Tarjan, "Scaling and Related Techniques for Computational Geometry," *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing* (1984), 135–143, or C. Levcopoulos and O. Petersson, "Heapsort Adapted for Presorted Files," *Journal of Algorithms* 14 (1993), 395–413.

12.29 A linked structure is unnecessary; we can store everything in an array. This is discussed in reference [12]. The bounds become $O(k \log N)$ for insertion, $O(k^2 \log N)$ for deletion of a minimum, and $O(k^2 N)$ for creation (an improvement over the bound in [12]).

12.35 Consider the pairing heap with 1 as the root and children 2, 3, . . . N. A *deleteMin* removes 1, and the resulting pairing heap is 2 as the root with children 3, 4, . . . N; the cost of this operation is N units. A subsequent *deleteMin* sequence of 2, 3, 4, . . . will take total time $\Omega(N^2)$.