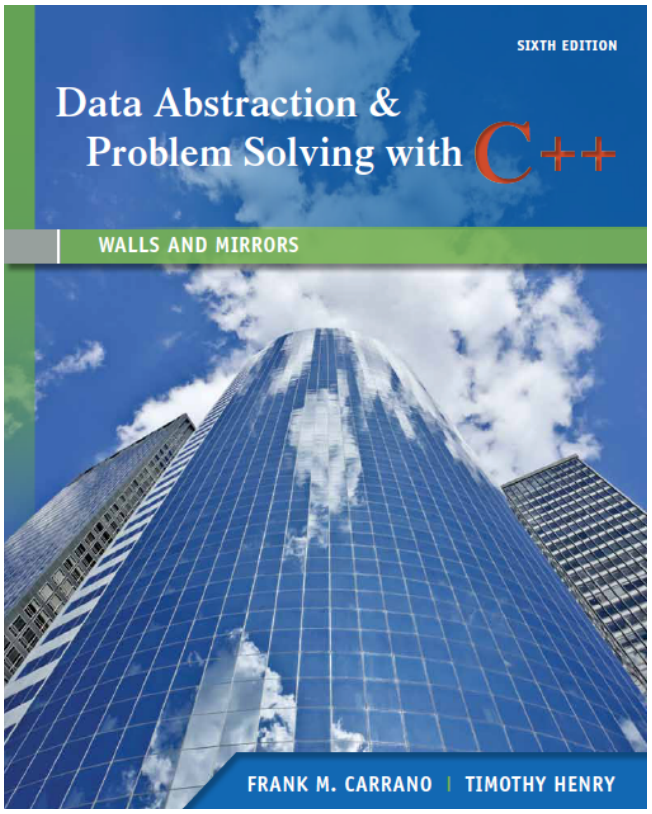
Solutions for Selected Exercises


SIXTH EDITION
Data Abstraction & Problem Solving with C++
WALLS AND MIRRORS
FRANK M. CARRANO | TIMOTHY HENRY

Frank M. Carrano

University of Rhode Island

Timothy Henry

University of Rhode Island

Chapter 1 Data Abstraction: The Walls

1

```
const CENTS_PER_DOLLAR = 100;

/** Computes the change remaining from purchasing an item costing
    dollarCost dollars and centsCost cents with d dollars and c cents.
    Precondition: dollarCost, centsCost, d and c are all nonnegative
    integers and centsCost and c are both less than CENTS_PER_DOLLAR.
    Postcondition: d and c contain the computed remainder values in
    dollars and cents respectively. If input value d < dollarCost, the
    proper negative values for the amount owed in d dollars and/or c
    cents is returned. */
 void computeChange(int dollarCost, int centsCost, int& d, int& c);
```

2a

```
const MONTHS_PER_YEAR = 12;
const DAYS_PER_MONTH[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

/** Increments the input Date values (month, day, year) by one day.
    Precondition: 1 <= month <= MONTHS_PER_YEAR,
                  1 <= day <= DAYS_PER_MONTH[month - 1], except
                  when month == 2, day == 29 and isLeapYear(year) is true.
    Postcondition: The valid numeric values for the succeeding month, day,
                   and year are returned. */
void incrementDate(int& month, int& day, int& year);

/** Determines if the input year is a leap year.
    Precondition: year > 0.
    Postcondition: Returns true if year is a leap year; false otherwise. */
bool isLeapYear(int year);
```

3a Change the purpose of an appointment:

```
changeAppointmentPurpose(apptDate: Date, apptTime: Time,
                         purpose: string): boolean
   if (isAppointment(apptDate, apptTime))
      cancelAppointment(apptDate, apptTime)

   return makeAppointment(apptDate, apptTime, purpose)
```

3b Display all the appointments for a given date:

```
displayAllAppointments(apptDate: Date)
   time = startOfDay

   while (time < endOfDay)
      if (isAppointment(apptDate, time))
         displayAppointment(apptDate, time)

      time = time + halfHour
```

This implementation requires the definition of a new operation, `displayAppointment`(), as well as definitions for the constants `startOfDay`, `endOfDay` and `halfHour`.

4
```
   Bag<string> fragileBag;

   while (storeBag.contains("eggs"))
   {
      storeBag.remove("eggs");
      fragileBag.add("eggs");
   }  // end while

   while (storeBag.contains("bread"))
   {
      storeBag.remove("bread");
      fragileBag.add("bread");
   }  // end while

   // Transfer remaining items from storeBag to groceryBag;
   Bag<string> groceryBag;
   v = storeBag.toVector();
   for (int i = 0; i < v.size(); i++)
      groceryBag.add(v.at(i));
```

5
```
/** Removes and counts all occurrences, if any, of a given string
 from a given bag of strings.
 @param bag  A given bag of strings.
 @param givenString  A string.
 @return  The number of occurrences of givenString that occurred
 and were removed from the given bag. */
int removeAndCount(ArrayBag<string>& bag, string givenString)
{
   int counter = 0;
   while (bag.contains(givenString))
   {
      counter++;
      bag.remove(givenString);
   }  // end while

   return counter;
}  // end removeAndCount
```

6
```
/** Creates a new bag that combines the contents of this bag and a
 second given bag without affecting the original two bags.
 @param anotherBag  The given bag.
 @return  A bag that is the union of the two bags. */
public BagInterface<ItemType> union(BagInterface<ItemType> anotherBag);
```

7
```
/** Creates a new bag that contains those objects that occur in both this
 bag and a second given bag without affecting the original two bags.
 @param anotherBag  The given bag.
 @return  A bag that is the intersection of the two bags. */
public BagInterface<ItemType> intersection(BagInterface<ItemType> anotherBag);
```

8
```
/** Creates a new bag of objects that would be left in this bag
 after removing those that also occur in a second given bag
 without affecting the original two bags.
 @param anotherBag  The given bag.
 @return  A bag that is the difference of the two bags. */
public BagInterface<T> difference(BagInterface<T> anotherBag);
```

9a `display(p.coefficient(p.degree()))`

9b `p.changeCoefficient(p.coefficient(3) + 8, 3)`

9c
```
for (power = 0; power < p.degree() || power < q.degree(); power++)
     // R is the sum of polynomials P and Q to degree power.
     r.changeCoefficient(p.coefficient(power) + q.coefficient(power), power)
```

Chapter 2 Recursion: The Mirrors

1 The problem is defined in terms of a smaller problem of the same type:
Here, the last value in the array is checked and then the remaining part of the array is passed to the function.

Each recursive call diminishes the size of the problem:
The recursive call to `getNumberEqual` subtracts 1 from the current value for `n`, passing this as the parameter `n` in the next call, effectively reducing the size of the unsearched remainder of the array by 1.

An instance of the problem serves as the base case:
Here, the case where the size of the array is 0 (i.e.: $n \leq 0$) results in the return of the value 0: an array of size 0 can have no instances of the `desiredValue`. This terminates the recursion.

As the problem size diminishes, the base case is reached:
`n` is an integer and is decremented by 1 with each recursive call. After `n` recursive calls, the parameter `n` in the *n*th call will have the value 0 and the base case will be reached.

2a The call `rabbit(5)` produces the following box trace:

```
n = 5
rabbit(4) = ?
rabbit(3) = ?      Follow the rabbit(4) call
return ?

n = 5              n = 4
rabbit(4) = ?      rabbit(3) = ?
rabbit(3) = ?      rabbit(2) = ?      Follow the rabbit(3) call
return ?           return ?

n = 5              n = 4              n = 3
rabbit(4) = ?      rabbit(3) = ?      rabbit(2) = ?
rabbit(3) = ?      rabbit(2) = ?      rabbit(1) = ?      Follow the rabbit(2) call
return ?           return ?           return ?

n = 5              n = 4              n = 3              n = 2
rabbit(4) = ?      rabbit(3) = ?      rabbit(2) = ?
rabbit(3) = ?      rabbit(2) = ?      rabbit(1) = ?                         Base case: n = 2
return ?           return ?           return ?           return 1

n = 5              n = 4              n = 3              n = 2
rabbit(4) = ?      rabbit(3) = ?      rabbit(2) = ?                         The rabbit(2) call
rabbit(3) = ?      rabbit(2) = ?      rabbit(1) = ?                          completes
return ?           return ?           return ?           return 1
```

```
n = 5              n = 4              n = 3              Follow the rabbit(1) call
rabbit(4) = ?      rabbit(3) = ?      rabbit(2) = 1
rabbit(3) = ?      rabbit(2) = ?      rabbit(1) = ?
return ?           return ?           return ?

n = 5              n = 4              n = 3              n = 1
rabbit(4) = ?      rabbit(3) = ?      rabbit(2) = 1                         Base case: n = 1
rabbit(3) = ?      rabbit(2) = ?      rabbit(1) = ?
return ?           return ?           return ?           return 1

n = 5              n = 4              n = 3              n = 1
rabbit(4) = ?      rabbit(3) = ?      rabbit(2) = 1                         The rabbit(1) call
rabbit(3) = ?      rabbit(2) = ?      rabbit(1) = 1                         completes
return ?           return ?           return 2           return 1

n = 5              n = 4              n = 3              n = 1
rabbit(4) = ?      rabbit(3) = 2      rabbit(2) = 1                         The rabbit(3) call
rabbit(3) = ?      rabbit(2) = ?      rabbit(1) = 1                         completes
return ?           return ?           return 2           return 1

n = 5              n = 4
rabbit(4) = ?      rabbit(3) = 2
rabbit(3) = ?      rabbit(2) = ?      Follow the rabbit(2) call
return ?           return ?

n = 5              n = 4              n = 2
rabbit(4) = ?      rabbit(3) = 2                         Base case:  n = 2
rabbit(3) = ?      rabbit(2) = ?
return ?           return ?           return 1

n = 5              n = 4              n = 2
rabbit(4) = ?      rabbit(3) = 2                         The rabbit(2) call completes
rabbit(3) = ?      rabbit(2) = 1
return ?           return 3           return 1

n = 5              n = 4              n = 2
rabbit(4) = 3      rabbit(3) = 2                         The rabbit(4) call completes
rabbit(3) = ?      rabbit(2) = 1
return ?           return 3           return 1

n = 5
rabbit(4) = 3
rabbit(3) = ?       Follow the rabbit(3) call
return ?

n = 5              n = 3
rabbit(4) = 3      rabbit(2) = ?
rabbit(3) = ?      rabbit(1) = ?       Follow the rabbit(2) call
return ?           return ?

n = 5              n = 3              n = 2
rabbit(4) = 3      rabbit(2) = ?
rabbit(3) = ?      rabbit(1) = ?                          Base case:  n = 2
return ?           return ?           return 1

n = 5              n = 3              n = 2
rabbit(4) = 3      rabbit(2) = 1
rabbit(3) = ?      rabbit(1) = ?                          The rabbit(2) call completes
return ?           return ?           return 1
```

```
n = 5             n = 3
rabbit(4) = 3     rabbit(2) = 1
rabbit(3) = ?     rabbit(1) = ?      Follow the rabbit(1) call
return ?          return ?


n = 5             n = 3             n = 1
rabbit(4) = 3     rabbit(2) = 1
rabbit(3) = ?     rabbit(1) = ?                       Base case:  n = 1
return ?          return ?          return 1


n = 5             n = 3             n = 1
rabbit(4) = 3     rabbit(2) = 1                       The rabbit(1) call completes
rabbit(3) = ?     rabbit(1) = 1
return ?          return 2          return 1


n = 5             n = 3             n = 1
rabbit(4) = 3     rabbit(2) = 1                       The rabbit(3) call completes
rabbit(3) = 2     rabbit(1) = 1
return 5          return 2          return 1


n = 5             n = 3             n = 1
rabbit(4) = 3     rabbit(2) = 1                       The rabbit(5) call completes and
rabbit(3) = 2     rabbit(1) = 1                       the value 5 is returned to the
return 5          return 2          return 1          calling function
```

2b The call `countDown(5)` produces the following box trace:

```
n = 5           The value 5 is printed.
cout << "5 ";   Follow the call to countDown(4)


n = 5           n = 4
cout << "5 ";   cout << "4 ";   The value 4 is printed.
                                Follow the call to countDown(3)

n = 5           n = 4           n = 3
cout << "5 ";   cout << "4 ";   cout << "3 ";   The value 3 is printed.
                                                Follow the call to countDown(2)

n = 5           n = 4           n = 3           n = 2
cout << "5 ";   cout << "4 ";   cout << "3 ";   cout << "2 ";   The value 2 is printed.
                                                                Follow the call to countDown(1)

n = 5           n = 4           n = 3           n = 2           n = 1           The value 1 is
cout << "5 ";   cout << "4 ";   cout << "3 ";   cout << "2 ";   cout << "1 ";   printed.  Follow
                                                                                the call to
                                                                                countDown(0)

n = 5           n = 4           n = 3           n = 2           n = 1           n = 0
cout << "5 ";   cout << "4 ";   cout << "3 ";   cout << "2 ";   cout << "1 ";   cout << endl;
                                                                                return

                                                                The end of line is printed and
                                                                the countDown(0) call completes.

n = 5           n = 4           n = 3           n = 2           n = 1           n = 0
cout << "5 ";   cout << "4 ";   cout << "3 ";   cout << "2 ";   cout << "1 ";   cout << endl;
                                                                                return

                                                                The countDown(1) call completes.
```

```
n = 5          n = 4          n = 3          n = 2          n = 1          n = 0
cout << "5 ";  cout << "4 ";  cout << "3 ";  cout << "2 ";  cout << "1 ";  cout << endl;
                                                                           return

                                             The countDown(2) call completes.

n = 5          n = 4          n = 3          n = 2          n = 1          n = 0
cout << "5 ";  cout << "4 ";  cout << "3 ";  cout << "2 ";  cout << "1 ";  cout << endl;
                                                                           return

                              The countDown(3) call completes.

n = 5          n = 4          n = 3          n = 2          n = 1          n = 0
cout << "5 ";  cout << "4 ";  cout << "3 ";  cout << "2 ";  cout << "1 ";  cout << endl;
                                                                           return

               The countDown(4) call completes.

n = 5          n = 4          n = 3          n = 2          n = 1          n = 0
cout << "5 ";  cout << "4 ";  cout << "3 ";  cout << "2 ";  cout << "1 ";  cout << endl;
                                                                           return

The countDown(5) call completes and returns to the calling function.
```

3

```
/** Returns the sum of the first n integers in the array anArray.
    Precondition:  0 <= n <= size of anArray.
    Postcondition:  The Sum of the first n integers in the
                    array anArray are returned.  The contents of
                    anArray and the value of n are unchanged. */
int computeSum(const  int anArray[], int n)
{  // base case
   if (n <= 0)
      return 0;
   else  // reduce the problem size
      return anArray[n - 1] + computeSum(anArray, n - 1);
}  // end computeSum
```

4

```
int sum (int start, int end )
{
   if (start == end)
      return end;
   else
      return start + sum(start + 1, end);
}
```

5

```
#include <string>
using namespace std;

// --------------------------------------------------
// Writes a character string backward.
// Precondition:  The string str contains size characters,
//                where size >= 1.
// Postcondition:  str is written backward, but remains
//                 unchanged.
// --------------------------------------------------
void writeBackward(string str, int size)
{  // base case
   if (size == 1)
      cout << str[0];

   // else, write rest of string
   else if (size > 1)
   {
      cout << str[size - 1];
      writeBackward(str, size - 1);
   }
   // size <= 0  do nothing;
}  // end writeBackward
```

---

6 The recursive method does not have a base case. As such, it will never terminate.

7

```
// ---------------------------------------------
// Prints out the integers from 1 through n as a
// comma separated list followed by a newline.
// Precondition:  n >= 0 and limit == n.
// Postcondition:  The integers from 1 through n
//                 are printed out followed by a
//                 newline.
// ---------------------------------------------
void printIntegers(int n, int limit)
{
   if (n > 0)
   {  // print out the rest of the integers
      printIntegers(n - 1, limit);

      // now print out this integer
      cout << n;

      // test for end of string
      if (n != limit)
        cout << ", ";
      else
        cout << "." << endl;   // end of string
   } // end if

   // n <= 0 do nothing
}  // end printIntegers
```

8

```
int getSum(int n)
{
   int result;
   if (n == 1)
      result = 1;
   else
      result = n + sum (n-1);
   return result;
}  // end getSum
```

9

```
const int NUMBER_BASE = 10;

/** Displays the decimal digits of number in reverse order.
    Precondition:  number >= 0.
    Postcondition:  The decimal digits of number are printed in reverse order.
                    This function does not output a newline character at the
                    end of a string. */
void reverseDigits(int number)
{  // check for input bounds
   if (number >= 0)
   {  // base case
      if (number < NUMBER_BASE)
         cout << number;
      else
      {  // print out rightmost digit
         cout << number % NUMBER_BASE;

         // pass remainder of digits to next call
         reverseDigits(number / NUMBER_BASE);
      } // end if
   }  // end if
}  // end reverseDigits
```

10a

```
/** Displays a line of n characters, where ch is the character.
    Precondition:   n >= 0.
    Postcondition:  A line of n characters ch is output
                    followed by a newline. */
void writeLine(char ch, int n)
{  // base case
   if (n <= 0)
      cout << endl;

   // write rest of line
   else
   {
      cout << ch;

      writeLine(ch, n - 1);
   }  // end if
}  // end writeLine
```

10b

```
/** Displays a block of m rows by n columns of character ch.
    Precondition:  m >= 0 and n >= 0.
    Postcondition:  A block of m rows by n columns of
                    character ch is printed. */
void writeBlock(char ch, int m, int n)
{  if (m > 0)
   {
      writeLine(ch, n);              // write first line
      writeBlock(ch, m - 1, n);      // write rest of block
   }

   // base case:  m <= 0 do nothing.
}  // end writeBlock
```

---

11 Running the given program produces the following output:

```
Enter: a = 1 b = 7
Enter: a = 1 b = 3
Leave: a = 1 b = 3
Leave: a = 1 b = 7
2
```

---

12 Running the given program produces the following output:

```
Enter: first = 1 last = 30
Enter: first = 1 last = 14
Enter: first = 1 last = 6
Enter: first = 4 last = 6
Leave: first = 4 last = 6
Leave: first = 1 last = 6
Leave: first = 1 last = 14
Leave: first = 1 last = 30
5
```

---

13 The algorithm first checks to see if *n* is a positive number: if not it immediately terminates. Otherwise, an integer division of *n* by 8 is taken and if the result is greater than 0 (i.e.: if $n > 8$), the function is called again with $n/8$ as an argument. This call processes that portion of the number composed of higher powers of 8. After this call, the residue for the current power, *n* % 8, is printed.

The given function computes the number of times 8^0, 8^1, 8^2, ... will divide *n*. These values are stacked recursively and are printed out in the reverse of the order of computation. The following is the hand execution with $n = 100$:

```
n = 100
displayOctal(12)
      n = 12
      displayOctal(1)
            n = 1
            cout << 1
      cout << 4
cout << 4

Output:  144
```

---

14 Even though the precondition states that *n* is nonnegative, there is no actual code to keep a negative value for *n* from being used as the argument in the function.

A call to the function `f` will produce a further call to `f` with a negative argument when *n* = 3. Because 3 is not within the subrange of 0 to 2, the default case will execute, and the function will attempt to evaluate `f(1)` and `f(-1)`. Because the value for `f(n)` is based on the values for `f(n-2)` and `f(n-4)`, if *n* is even, its addends will be the next two smaller even integers; likewise, if *n* is odd, `f(n)`'s addends will be *n*'s next two smaller odd integers. Thus any odd nonnegative integer will eventually cause `f` to evaluate `f(3)`.

Theoretically, calling `f` with an odd integer will cause an infinite sequence of function calls. On the practical level, the computer's run-time stack will overflow, or an integer underflow will happen.

The following is the exact output of the program:

```
Function entered with n = 8
Function entered with n = 6
Function entered with n = 4
Function entered with n = 2
Function entered with n = 0
Function entered with n = 2
Function entered with n = 4
Function entered with n = 2
Function entered with n = 0
The value of f(8) is 27
```

15 The following output is produced when `x` is a value argument:

```
6 2
7 1
8 0
8 0
7 1
6 2
```

Changing `x` to a reference argument produces:

```
6 2
7 1
8 0
8 0
8 1
8 2
```

16a The call `binSearch(5)` produces the following box trace:

```
value = 5                value = 5
first = 1                first = 1
last = 8                ►last = 3
mid = 4                  mid = 2
value < anArray[4]       value = anArray[2]
                         return 2
```

16b The call `binSearch(13)` produces the following box trace:

```
value = 13            value = 13            value = 13            value = 13
first = 1             first = 5             first = 5             first = 5
last = 8             ▶last = 8             ▶last = 5             ▶last = 4
mid = 4               mid = 6               mid = 5               first > last
value > anArray[4]    value < anArray[6]    value < anArray[5]    return 0
```

16c The call `binSearch(16)` produces the following box trace:

```
value = 16            value = 16            value = 16            value = 16
first = 1             first = 5             first = 5             first = 6
last = 8             ▶last = 8             ▶last = 5             ▶last = 5
mid = 4               mid = 6               mid = 5               first > last
value > anArray[4]    value < anArray[6]    value > anArray[5]    return 0
```

---

17 **a** For a binary search to work, the array must first be sorted in either ascending or descending order.

b The index is (0 + 101)/2 = 50.

c Number of comparisons = $\lfloor \lg 101 \rfloor$ = 6.

18a

```
/** Returns the value of x raised to the nth power.
    Precondition:  n >= 0
    Postcondition:  The computed value is returned. */
double power1(double x, int n)
{  double result = 1;         // value of x^0

   while (n > 0)              // iterate until n == 0
   {  result *= x;
      n--;
   }
   return result;
}  // end power1
```

18b

```
/** Returns the value of x raised to the nth power.
    Precondition:  n >= 0
    Postcondition:  The computed value is returned. */
double power2(double x, int n)
{  // base case
   if (n == 0)
      return 1;

   // else, multiply x by rest of computation
   else
      return x * power2(x, n-1);
}  // end power2
```

18c

```
/** Returns the value of x raised to the xth power.
    Precondition:  n >= 0
    Postcondition:  The computed value is returned. */
double power3(double x, int n)
{
   if (n == 0)
      return 1;
   else
   {  // do this computation only once!!
      double halfPower = power3(x, n/2);

      // if n is even...
      if (n % 2 == 0)
        return halfPower * halfPower;

      // if n is odd...
      else
        return x * halfPower * halfPower;
   }
}  // end power3
```

18d The following table lists the number of multiplications performed by each of the algorithms for computing the values on the top line:

	3^{32}	3^{19}
power1	32	19
power2	32	19
power3	7	8

18e The following table lists the number of recursive calls made by each of the algorithms indicated in order to perform the computation on the inputs given on the top line:

	3^{32}	3^{19}
power2	32	19
power3	6	5

19 Maintain a count of the recursive depth of each call by passing this count as an additional parameter to the function call and print that many spaces or tabs in front of each message:

```
/** Computes a term in the Fibonacci sequence.
    Precondition:  n is a positive integer and tab = 0.
    Postcondition:  The progress of the recursive function call is displayed
                    as a sequence of increasingly nested blocks.  The function
                    returns the nth Fibonacci number. */
int rabbit(int n, int tab)
{
   int value;

   // Indent the proper distance for this block
   for (int i = 0; i < tab; i++)
      cout << '\t';

   // Display status of call
   cout << "Enter: n = " << n << endl;

   if (n <= 2)
      value = 1;

   else  // n > 2, so n-1 > 0 and n-2 > 0
      // indent by one for next call
      value = rabbit(n-1, tab+1) + rabbit(n-2, tab+1);

   // Indent the proper distance for this block
   for (i = 0; i < tab; i)
      cout << '\t';

   // Display status of call
   cout << "Leave: n = " << n << " value = " << value << endl;

   return value;
}
```

20a `f(6)` is 8; `f(7)` is 11; `f(12)` is 95; `f(15)` is 320.

20b Since we only need the five most recently computed values, we will maintain a "circular" five-element array indexed modulus 5.

```
// Pre: n > 0.
int fOfN(int n)
{
   int last5[5] = {1, 1, 1, 3, 5};

   for (int i = 5; i < n; i++)
   {
      int fi = last5[(i - 1) % 5] + 3 * last5[(i - 5) % 5];

      // Replace entry in last5
      last5[i % 5] = fi;  // f(i) = f(i - 1) + 3 x f(i - 5)
   }  // end for

   return last5[(n - 1) % 5];
}  // end fOfN
```

21a A function to compute n! iteratively:

```
long fact(int n)
{  int i;
   long result;

   if (n < 1) // base case
      result = 0;
   else
   {
      result = 1;
      for (i = 2; i <= n; i++)
         result *= i;
   }  // end if

   return result;
}  // end fact
```

21b A simple iterative solution to writing a string backwards:

```
#include <string>

void writeBackward(string  str)
{
   for (int i = str.size() - 1; i >= 0; i--)
      cout << str[i];

   cout << endl;
}  // end writeBackward
```

21c A function to perform an iterative binary search:

```
/** Searches a sorted array and returns the index in the
    array corresponding to the value key if key is in the
    array, -1 otherwise.
    Precondition:  high is sorted in ascending order.
                   low = 0 and high = the size of high - 1.
    Postcondition:  If key is found, its location index
                    in high is returned, else -1 is returned. */
int binarySearch(int  anArray[], int key, int low, int high)
{
   int mid, result;
   while (low < high)
   {  mid = (low + high)/2;

      if (anArray[mid] == key)
      {  low = mid;
         high = mid;
      }
      else if (anArray[mid] < key)
        low = mid + 1; // search the upper half
      else
         high = mid - 1; // search the lower half
   }  // end while

   if (low > high)
      result = -1;       // if not found, return -1
   else if (anArray[low] != key)
      result = -1;
   else
      result = low;

   return result;
}  // end binarySearch
```

21d We implement a function to find the kth smallest entry in an array using an integer array and a selection sort up to k times. We assume a standard integer swap function.

```
int kSmall(int k, int anArray[], int size)
{
   for (int i=0; i<k; i++)
      for (int j = i+1; j < size; j++)
         if (anArray[j] < anArray[i])
            swap(anArray[i], anArray[j]);

   return anArray[k-1];
}  // end kSmall
```

---

22 The for loop invariant is:

$3 \leq i \leq n$

So the sum = $\sum_{i=3}^{n} rabbit(i-1) + rabbit(i-2)$.

23a We must verify that the equation holds for both the base case and the recursive case.

For the base case, let *gcd*(*a*, *b*) = *b*. Then, *a* mod *b* = 0 and, since 0/*n* = 0 for all *n*, then *gcd*(*b*, 0) = *b*. Hence, *gcd*(*b*, *a* mod *b*) = *b*.

For the recursive case, let *gcd*(*a*, *b*) = *d*, i.e.: *a* = *dj* and *b* = *dk* for integers *d*, *j* and *k*. Now there exists integer *n* = *a* mod *b* such that (*n* - *a*)/*b* = *q*, where *q* is an integer. Then, *n*-*a* = *bq* and, so *n* - *dj* = *dkq* i.e. *n* = *d*(*kq* + *j*). Then, (*n*/*d*) = *kq* + *j*, where (*kq* + *j*) is an integer. So, *d* divides *n* i.e.: *d* divides (*a* mod *b*).

To show that *d* is the greatest common divisor of *b* and *a* mod *b*, suppose for contradiction there exists integer *g* > *d* such that *b* = *gr* and (*a* mod *b*) = *gs* for integers *r* and *s*. Then, (*gs* - *a*)/*gr* = *q'* where *q'* is an integer. So *gs* - *a* = *grq'* i.e.: *a* = *g*(*s* - *rq'*). Thus, *g* divides *a* and *g* divides *b*. But *gcd*(*a*, *b*) = *d* by hypothesis. Therefore, *gcd*(*b*, *a* mod *b*) = *d*.

The proof is symmetrical where *gcd*(*b*, *a* mod *b*) = *d* is taken for the hypothesis.

23b If *b* > *a* in the call to *gcd*, *a* mod *b* = *a* and so the recursive call effectively reverses the arguments.

23c When *a* > *b*, the argument associated with the parameter *a* decreases in the next recursive call. If *b* > *a*, the next recursive call will swap the arguments so that *a* > *b*. Thus, the first argument will eventually equal the second and so eventually *a* mod *b* will be 0.

24a $$c(n) = \begin{cases} 0 & \textit{if} \quad n = 1 \\ 1 & \textit{if} \quad n = 2 \\ \sum_{i=1}^{n-1} (c(n-i)+1) & \textit{if} \quad n > 2 \end{cases}$$

24b $$c(n) = \begin{cases} 0 & \textit{if} \quad n = 1 \\ 1 & \textit{if} \quad n = 2 \\ c(n-1)+c(n-2) & \textit{if} \quad n > 2 \end{cases}$$

---

25 *Acker*(1, 2) = 4

```
int acker(int m, int n)
{
   int result;

   if (m == 0)
      result = n+1;

   else if (n == 0)
      result = acker(m-1, 1);

   else
      result = acker(m-1, acker(m, n-1));

   return result;
}  // end acker
```

Chapter 3 Array-Based Implementations

1

```
/** Computes the sum of the integers in the bag aBag.
 @param aBag  A bag of integers.
 @return  The sum of the integers in aBag. */
int sumOfBag(ArrayBag<int>& aBag)
{
   int sum = 0;
   int size = aBag.getCurrentSize();
   vector<int> bagContents = aBag.toVector();

   for (int i = 0; i < size; i++)
      sum += bagContents.at(i);

   return sum;
}  // end sumOfBag
```

2

```
/** Replaces one occurrence of a given item in a bag with another one.
 @param aBag  A bag.
 @param itemToReplace  The item to replace.
 @param replacement  The item that replaces itemToReplace.
 @return  True if the replacement is successful; otherwise returns false. */
bool replace(ArrayBag<string>& aBag, string itemToReplace, string replacement )
{
   bool success = aBag.remove(itemToReplace);
   if (success)
      success = aBag.add(replacement);

   return success;
} // end replace
```

3

a An advantage to defining such operations externally to the ADT is that they are independent of the ADT's implementation. The client simply uses ADT operations.

The disadvantage is that the client must use existing ADT operations. Certain operations might be impossible to define—either at all or efficiently— at the client level if the ADT does not provide sufficient access to the ADT's data. For `replace`, the client has no control over which occurrence of the item is replaced.

b Defining `replace` within the ADT obviously alleviates the disadvantage cited in part a. Replacing an item can be done more efficiently than first removing it from the bag and then adding another item to the bag.

However, the client cannot control how `replace` behaves. Its specification is at the discretion of the ADT designer.

4

Some ADT rectangle operations:

```
// Creates a rectangle and initializes its length and width to default values.
+Rectangle()

// Creates a rectangle and initilializes its length and width to the
// values received as parameters.
+Rectangle(length: double, width: double)

// Sets or modifies the length of this rectangle.
// Checks to make sure that the new length is greater than 0.
+setLength(length: double)

// Sets or modifies the width of this rectangle.
// Checks to make sure that the new width is greater than 0.
+setWidth(width:double)

// Returns this rectangle's length.
+getLength(): double

// Returns this rectangle's width.
+getWidth(): double

// Returns this rectangle's area.
+getArea(): double

// Returns this  rectangle's perimeter.
+getPerimeter(): double
```

```
/** Header file for the class Rectangle. */
class Rectangle
{
private:
   double length;
   double width;

public:
   /** Creates a rectangle and initializes its length and width to
       default values. */
   Rectangle();

   /** Creates a rectangle and initializes its length and width to given
       values. */
   Rectangle(double initialLength, double initialWidth);

   /** Sets or modifies the length of this rectangle.
       Checks to make sure that the new length is greater than 0. */
   void setLength(double newLength);

   /** Sets or modifies the width of this rectangle.
       Checks to make sure that the new width is greater than 0. */
   void setWidth(double newWidth);

   /** @return  This rectangle's length. */
   double getLength();

   /** @return  This rectangle's width. */
   double getWidth();

   /** @return  This rectangle's area. */
   double getArea();
   /** @return  This rectangle's perimeter. */
   double getPerimeter();
};


/** Implementation file for the class Rectangle. */

#include "Rectangle.h"

Rectangle::Rectangle()
{
   setLength(1.0);
   setWidth(1.0);
}

Rectangle::Rectangle(double  initialLength,  double initialWidth)
{
   setLength(initialLength);
   setWidth(initialWidth);
}

void Rectangle::setLength(double newLength)
{
   if (newLength > 0.0)
      length = newLength;
}

void Rectangle::setWidth(double newWidth)
{
   if (newWidth > 0.0)
      width = newWidth;
}

double Rectangle::getLength()
{
   return length;
}

double Rectangle::getWidth()
{
   return width;
}

double Rectangle::getArea()
{
   double area = length * width;
   return area;
}

double Rectangle::getPerimeter()
{
   double perimeter = 2 * (length + width);
   return perimeter;
}
```

5

```
/** Header file for the class Triangle. */
#include <vector>

using namespace std;

class Triangle
{
private:
   double side1, side2, side3;

public:
   /** Creates a triangle and initializes its sides to default values. */
   Triangle();

   /** Creates a triangle and initializes its sides to given values. */
   Triangle(double initialSide1, double initialSide2, double initialSide3);

   /** Sets or modifies the sides of this triangle.
       Ensures that the new sides form a triangle. */
   void setSides(double newSide1, double newSide2, double newSide3);

   /** Gets the three sides of this triangle.
    @return  A vector containing the values of the sides. */
   vector<double> getSides() const;

   /** Computes the area of this triangle.
    @return  This triangle's area. */
   double getArea();

   /** Computes the perimeter of this triangle.
    @return  This triangle's perimeter. */
   double getPerimeter();

   /** @return  True if this triangle is a right triangle. */
   bool isRightTriangle();

   /** @return  True if this triangle is an equilateral triangle. */
   bool isEquilateral();

   /** @return  True if this triangle is an isosceles triangle. */
   bool isIsosceles();
};

/** Implementation file for the class Triangle. */

#include "Triangle.h"
#include <cmath>

Triangle::Triangle()
{
   setSides(3.0, 4.0, 5.0);
}  // end default constructor

Triangle::Triangle(double initialSide1, double initialSide2, double initialSide3)
{
   setSides(initialSide1, initialSide2, initialSide3);
}  // end constructor
```

```
void Triangle::setSides(double newSide1, double newSide2, double newSide3)
{
   if ((newSide1 + newSide2 > newSide3) &&
       (newSide1 + newSide3 > newSide2) &&
       (newSide2 + newSide3 > newSide1))
   {
      side1 = newSide1;
      side2 = newSide2;
      side3 = newSide3;
   }
   else
      cout << "The sides "
           << newSide1 << " " << newSide2  << " " << newSide3
           << " do not form a triangle" << endl;
}  // end setSides

vector<double> Triangle::getSides() const
{
   vector<double> sides;
   sides.push_back(side1);
   sides.push_back(side2);
   sides.push_back(side3);

   return sides;
}  // end getSides

double Triangle::getArea()
{
   double z = (side1 + side2 + side3) / 2;
   return sqrt(z * (z - side1) * (z - side2) * (z - side3));
}  // end getArea

double Triangle::getPerimeter()
{
   return side1 + side2 + side3;
}  // end getPerimeter

bool Triangle::isRightTriangle()
{
   double side1Squared = side1 * side1;
   double side2Squared = side2 * side2;
   double side3Squared = side3 * side3;

   bool rightTriangle = false;
   if (side1Squared + side2Squared == side3Squared)
      rightTriangle = true;
   else if (side1Squared + side3Squared == side2Squared)
      rightTriangle = true;
   else if (side2Squared + side3Squared == side1Squared)
      rightTriangle = true;

   return rightTriangle;
}  // end isRightTriangle

bool Triangle::isEquilateral()
{
   return (side1 == side2) && (side1 == side3) && (side2 == side3);
}  // end isEquilateral

bool Triangle::isIsosceles()
{
   return (side1 == side2) || (side1 == side3) || (side2 == side3);
}  // end isIsosceles
```

---

6 We add a private recursive method, `addToVector`, to the class `ArrayBag`.

```
template<class ItemType>
void ArrayBag<ItemType>::addToVector(vector<ItemType>& v, int i) const
{
   if (i < itemCount)
   {
      v.push_back(items[i]);
      addToVector(v, i + 1);
   }  // end if
}  // end addToVector

template<class ItemType>
vector<ItemType> ArrayBag<ItemType>::toVector() const
{
   vector<ItemType> bagContents;
   addToVector(bagContents, 0);

   return bagContents;
}  // end toVector
```

7

```
/** Merges the contents of two given bags into a third one without
    destroying the original bags.
 @param oneBag  A bag.
 @param anotherBag  A bag.
 @return  A bag that contains the entries in the two given bags. */
ArrayBag<string> merge(const ArrayBag<string>& oneBag,
                       const ArrayBag<string>& anotherBag)
{
   ArrayBag<string> newBag;

   // Add entries from first bag to the new bag
   vector<string> v = oneBag.toVector();

   for (int index = 0; index < oneBag.getCurrentSize(); index++)
      newBag.add(v.at(index));

   // Add entries from the second bag to the new bag
   v = anotherBag.toVector();

   for (int index = 0; index < anotherBag.getCurrentSize(); index++)
      newBag.add(v.at(index));

   return newBag;
}  // end merge
```

8
Add the following statements to `BagInterface`:

```
/** Removes one occurrence of a random entry from this bag.
 @pre  The bag is not empty.
 @post  If successful, anEntry has been removed from the bag
     and the count of items in the bag has decreased by 1.
 @return  The entry that was removed. */
virtual ItemType remove() = 0;
```

Add the following method to `ArrayBag.cpp`:

```
template<class ItemType>
ItemType ArrayBag<ItemType>::remove()
{
   int randomIndex = abs(rand() % itemCount);
   ItemType removedEntry = items[randomIndex];
       itemCount--;
       items[randomIndex] = items[itemCount];

       return removedEntry;
}  // end remove
```

9

```
template<class ItemType>
ArrayBag<ItemType>::ArrayBag(ItemType entries[], int entryCount):
                    itemCount(entryCount), maxItems(entryCount + DEFAULT_CAPACITY)
{
   for (int index = 0; index < entryCount; index++)
      items[index] = entries[index];
}  // end constructor
```

Chapter 4 Link-Based Implementations

1

```
Node<string>* secondNode = headPtr->getNext();
headPtr->setNext(secondNode->getNext());
secondNode->setNext(nullptr);
delete secondNode;
```

2

```
template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
   // Create new node
   Node<ItemType>* newNodePtr = new Node<ItemType>();
   newNodePtr->setItem(newEntry);
   newNodePtr->setNext(nullptr);  // New node will end the chain

   if (isEmpty())
      headPtr = newNodePtr;
   else  // Add to end of chain
   {
      // Locate last node:
      Node<ItemType>* lastNodePtr = headPtr;
      for (int counter = 1; counter < itemCount; counter++)
         lastNodePtr = lastNodePtr->getNext();

      // Link new node to end of chain
      lastNodePtr->setNext(newNodePtr);
   }  // end if

   itemCount++;  // Count new node

   return true;
}  // end add
```

3a

```
template<class ItemType>
int LinkedBag<ItemType>::getCurrentSize() const
{
   int count = 0;
   Node<ItemType>* curPtr = headPtr;
   while (curPtr != nullptr)
   {
      count++;
      curPtr = curPtr->getNext();
   }  // end while

   return count;
}  // end getCurrentSize
```

3b

```
template<class ItemType>
int LinkedBag<ItemType>::getCurrentSize() const
{
   return recursiveCounter(headPtr);
}  // end getCurrentSize

template<class ItemType>
int LinkedBag<ItemType>::recursiveCounter(Node<ItemType>* curPtr) const
{
   if (curPtr == nullptr)
      return 0;
   else
      return 1 + recursiveCounter(curPtr->getNext());
}  // end recursiveCounter
```

4

```
template<class ItemType>
int LinkedBag<ItemType>::getFrequencyOf(const ItemType& anEntry) const
{
   return frequencyCounter(headPtr, anEntry);
}  // end getFrequencyOf

template<class ItemType>
int LinkedBag<ItemType>::frequencyCounter(Node<ItemType>* curPtr,
                                          const ItemType& anEntry) const
{
   if (curPtr == nullptr)
      return 0;
   else if (anEntry == curPtr->getItem())
      return 1 + frequencyCounter(curPtr->getNext(), anEntry);
   else
      return frequencyCounter(curPtr->getNext(), anEntry);
}  // end frequencyCounter
```

5

```
template<class ItemType>
LinkedBag<ItemType>::LinkedBag(ItemType entries[], int entryCount)
{
   for (int index = 0; index < entryCount; index++)
      add(entries[index]);
}  // end constructor
```

6

Add the following statements to `BagInterface`:

```
   /** Removes one occurrence of a random entry from this bag.
     @pre  The bag is not empty.
     @post  If successful, anEntry has been removed from the bag
         and the count of items in the bag has decreased by 1.
     @return  The entry that was removed. */
    virtual ItemType remove() = 0;
```

Add the following method to `LinkedBag.cpp`:

```
template<class ItemType>
ItemType LinkedBag<ItemType>::remove()
{
   int randomIndex = abs(rand() % itemCount);
   Node<ItemType>* nodePtr = headPtr;  // Precondition implies not nullptr
   for (int counter = 0; counter < randomIndex; counter++)
      nodePtr = nodePtr->getNext();

   ItemType removedEntry = nodePtr->getItem();
   remove(removedEntry);
// NOTE: The previous call to the method remove is the easiest
// way to define this method, but remember that the remove method
// traverses the chain to locate removedEntry. Since we have already
// located that entry, this traversal wastes time. It is more
// efficient to execute the following statements (between /* and */)
// instead of calling remove(removedEntry).
  /*
   // Copy data from first node to located node
   nodePtr->setItem(headPtr->getItem());

   // Delete first node
   Node<ItemType>* nodeToDeletePtr = headPtr;
   headPtr = headPtr->getNext();

   // Return node to the system
   nodeToDeletePtr->setNext(nullptr);
   delete nodeToDeletePtr;
   nodeToDeletePtr = nullptr;

   itemCount--;
*/
   return removedEntry;
}  // end remove
```

7 Traversing an array of size *n* and traversing a linked chain of *n* nodes both require about the same number of operations. This number is proportional to *n*. Chapter 10 will discuss how to make such comparisons.

8 To display the data in the *n*th node of a linked chain requires the same traversal of the chain as when you display the data in its first *n* nodes. Thus, the number of operations necessary for both problems is proportional to *n*. To display the *n*th entry in an array, no traversal is needed. Thus, the number of operations required is independent of *n* and much smaller than the number required for the linked chain.

9 In an array-based implementation, we remove an entry by overwriting it with the last entry in the array. By decrementing `itemCount`, the last entry is ignored subsequently. This part of the operation is independent of where `anEntry` occurs in the array. However, the location of `anEntry` within the array affects the effort required by `getIndexOf`. If it is first in the array, `getIndexOf` finds it at the first comparison. If `anEntry` is last in the array, *n* comparisons are needed to locate it. In the average case, `getIndexOf` makes *n* / 2 comparisons.

In the link-based implementation, we remove an entry by overwriting it with the entry in the last node of the chain. Then we disconnect the last node from the chain and return the node to the system. Again, this part of the operation is independent of where `anEntry` occurs in the chain. However, this part of the operation takes more effort than the first part of the array-based method. The location of `anEntry` within the chain affects the effort required by `getPointerTo`. This effort is comparable to that required by `getIndexOf` in the array-based case.

If, for example, more removals occur from the second half of the array or chain than from the first half, an array-based implementation will incur fewer operational steps than the link-based one. If no predictable pattern of removals is known in advance, however, the merits of an array-based versus a link-based implementation of the bag will have less impact on the overall efficiency.

10

```
template<class ItemType>
class DoubleNode
{
private:
   ItemType              item; // A data item
   DoubleNode<ItemType>* next; // Pointer to next node
   DoubleNode<ItemType>* prev; // Pointer to previous node

public:
   DoubleNode();
   DoubleNode(const ItemType& anItem);
   DoubleNode(const ItemType& anItem, DoubleNode<ItemType>* nextNodePtr);

   void setItem(const ItemType& anItem);
   ItemType getItem() const;

   void setNext(DoubleNode<ItemType>* nextNodePtr);
   DoubleNode<ItemType>* getNext() const;

   void setPrevious(DoubleNode<ItemType>* previousNodePtr);
   DoubleNode<ItemType>* getPrevious() const;
}; // end DoubleNode
```

11 Assume the class `DoubleNode` given in the answer to Question 10.

```
DoubleNode<string>* newNodePtr = new DoubleNode(newEntry);  // Create new node
newNodePtr->setNext(headPtr);      // Make new node point to existing chain
headPtr->setPrevious(newNodePtr); // Make first node in existing chain point to new node
headPtr = newNodePtr;             // Set head pointer to new first node
```

12 Assume the class `DoubleNode` given in the answer to Question 10.

```
DoubleNode<string>* nodeToRemovePtr = headPtr;  // Point to first node (to be removed)
headPtr = headPtr->getNext();      // Advance head pointer to point to new first node
headPtr->setPrevious(nullptr);     // Set previous pointer of new first node to nullptr
nodeToRemovePtr->setNext(nullptr); // Set next pointer of old first node to nullptr
delete nodeToRemovePtr;            // Return removed node to system
```

Chapter 5 Recursion as a Problem-Solving Technique

1a The call `isPalindrome("abcdeba")` produces the following box trace:

```
The initial call is made and the function begins execution:

  w = "abcdeba"

At the else if clause, the call isPalindrome("bcdeb") is made:

  w = "abcdeba"     ▶   w = "bcdeb"

At the else if clause, the call isPalindrome("cde") is made:

  w = "abcdeba"     ▶   w = "bcdeb"      ▶    w = "cde"

Base case:  isPalindrome("cde") returns false:

                                              w = "cde"
  w = "abcdeba"     ▶   w = "bcdeb"      ▶
                                            return false;

The call isPalindrome("bcdeb") returns false:

                        w = "bcdeb"
  w = "abcdeba"     ▶
                      return false;            w = "cde"

The call isPalindrome("abcdeba") returns false a return false;  call completes.

  w = "abcdeba"         w = "bcdeb"           w = "cde"

  return false;       return false;         return false;
```

1b The call `isAnBn("AABB")` produces the following box trace:

```
The initial call is made and the function begins execution:

    w = "AABB"

The call to isAnBn("AB") is made:

    w = "AABB"       ►     w = "AB"

The call to isAnBn("") is made:

    w = "AABB"       ►     w = "AB"        ►     w = ""

Base case:  w = empty string, returns true:

                                                 w = ""
    w = "AABB"       ►     w = "AB"        ►
                                              return true;
The call isAnBn("AB") returns true:

                           w = "AB"              w = ""
    w = "AABB"       ►
                        return true;          return true;
The call isAnBn("AABB") returns true and the initial call completes.

    w = "AABB"             w = "AB"              w = ""

  return true;          return true;          return true;
```

1c The call `endPre("-*/abcd")` produces the following box trace:

```
s = "-*/abcd";

first = 0
last = 6


first = 0          first = 1
last = 6           last = 6
                  ►
X:endPre(1)

first = 0          first = 1          first = 2
last = 6           last = 6           last = 6
                  ►                  ►
X:endPre(1)        X:endPre(2)

first = 0          first = 1          first = 2          first = 3
last = 6           last = 6           last = 6           last = 6
                  ►                  ►                  ►
X:endPre(1)        X:endPre(2)        X:endPre(3)

first = 0          first = 1          first = 2          first = 3
last = 6           last = 6           last = 6           last = 6
                  ►                  ►firstEnd = 3       return 3
X:endPre(1)        X:endPre(2)

first = 0          first = 1          first = 2          first = 4
last = 6           last = 6           last = 6           last = 6
                  ►                  ►firstEnd = 3      ►
X:endPre(1)        X:endPre(2)        Y:endPre(4)

first = 0          first = 1          first = 2          first = 4
last = 6           last = 6           last = 6           last = 6
                  ►                  ►firstEnd = 3       return 4
X:endPre(1)        X:endPre(2)        return 4

first = 0          first = 1          first = 2
last = 6           last = 6           last = 6
                  ►firstEnd = 4       firstEnd = 3
X:endPre(1)                           return 4


first = 0          first = 1          first = 5
last = 6           last = 6           last = 6
                  ►firstEnd = 4      ►
X:endPre(1)        Y:endPre(5)

first = 0          first = 1          first = 5
last = 6           last = 6           last = 6
                  ►firstEnd = 4       return 5
X:endPre(1)        return 5

first = 0          first = 1
last = 6           last = 6
firstEnd = 5       firstEnd = 4
                   return 5

first = 0          first 6
last = 6           last = 6
firstEnd = 5      ►
Y:endPre(6)
```

```
first = 0          first = 6
last = 6           last = 6
firstEnd = 5       return 6
return 6

first = 0
last = 6
firstEnd = 5
return 6
```

---

2 The following lists all of the strings of length 7 or less that may be constructed from the given grammar:

$	abb	aabbbb
$$	$abb	$aabbbb
$$$	$$abb	
$$$$	$$$abb	
$$$$$	$$$$abb	
$$$$$$		
$$$$$$$		

---

3 <*S*> = <*U*> | <*U*> <*L*>
<*U*> = A | B | … | Z
<*L*> = a | b | … | z | a <*L*> | b <*L*> | …| z <*L*>

---

4 <*S*> = **- -** <*C*> **.** | **. .** <*C*> **-**
<*C*> = **.** | **-** | **.** <*C*> | **-** <*C*>

5a <*string*> = <*X*> <*U*> | <*X*> <*string*> | <*X*> Y
<*U*> = X | Z
<*X*> = X

5b XX
XZ
XY

6a **- - .**
- . .
. . .

6b The string **. . . . - -** is not in the language because it is not possible for a legal string to possess dashes anywhere except at the beginning.

6c *word0* = **- - - - - - .**

word0 is legal because it is of the form <*dash*><*word*>. We may recursively remove the initial dash from each subword thus discovered until we get to the terminal subword "**.**" which is of the legal form <*dot*>.

6d

```
isIn(str)
// Returns true if str is in the indicated language, false otherwise
   size = length of string str

   if (size == 1 and str[0] == dot)
      return true
   else if (size > 1 and str[0] == dash)
      return isIn(str minus first character)
   else if (size > 1 and str[size - 1] == dot)
      return isIn(str minus last character)
   else
      return false
```

7

```
isIn (str)
// Returns true if str is in the indicated language, false otherwise
  if (str consists only of the character R)
     return true
  else if (str consists only of the character P or the character Q)
     return true
  else if (str begins with the character P or the character Q and ends
           with the number 1)
     return isIn(str minus its first and last characters)
  else
     return false
```

8a

```
isIn (w)

  if (the length of w is zero)
    return true

  if (w consists of only the character & or the character #)
    return true
  else if (w starts with the character W or the character A
           and ends with the character & or the character #)
    return true
  else if (w starts with the character & or the character #
           and ends with the character W or the character A)
    return isIn (w minus its first and last characters)
  else
    return false
```

8b Yes

9a <*S*> = **ABB | A** <*S*> **BB**

9b

```
#include <string>
using namespace std;

// Determines if the string str is in the language L.
// Precondition:  none.
// Postcondition:  returns true if str is in L, false otherwise.
bool isInL(string strExp)
{
   int size = strExp.size();

   if (size < 3)   // No strings shorter than 3 are in L.
      return false;
   else if ((strExp == "ABB"))  // Base case
      return true;
   else
   {
       if (strExp[1] == 'A' && strExp.substr(size - 2, 2) == "BB")
       {  // Trim off first and last two characters of strExp
         return isInL(strExp.substr(1, size - 3));
       }
       else
         return false;
   }  // end if
}  // end isInL
```

---

10 Let S = `"+*a-b/c++de-fg"`.
S is not a prefix expression since it is defined neither by the form <*identifier*> nor by the form <*operator*><*prefix*><*prefix*>. The first case is clear by inspection. To show the second case, we derive component prefix substrings as follows:

Let $S = +E_1E_2$. We continue recursively by deriving the expressions E_i:

$E_1 = *E_3E_4$
$E_3 = a$
$E_4 = -E_5E_6$
$E_5 = b$
$E_6 = /E_7E_8$
$E_7 = c$
$E_8 = +E_9E_{10}$
$E_9 = +E_{11}E_{12}$
$E_{11} = d$
$E_{12} = e$
$E_{10} = -E_{13}E_{14}$
$E_{13} = f$
$E_{14} = g$

This exhausts the string S before the prefix expression E_2 is defined. Thus S is not a valid prefix expression.

---

11 The string S = "`ab/c*efg*h/+d-+`" is a valid postfix string. To show this we first observe that S is not a simple identifier and so it must be an expression of the form

$E_x E_y$+

where the terminal symbol "+" is preceded by two postfix expressions. We attempt to build these expressions by proceeding from the front of the string as follows:

E_1 = `a`
E_2 = `b`
$E_3 = E_1E_2$/
E_4 = `c`
$E_5 = E_3E_4$*
E_6 = `e`
E_7 = `f`
E_8 = `g`
$E_9 = E_7E_8$*
E_{10} = `h`
$E_{11} = E_9E_{10}$/
$E_{12} = E_6E_{11}$+
E_{13} = `d`
$E_{14} = E_{12}E_{13}$-
$E_{15} = E_5E_{14}$+

Thus, $E_5 = E_x$ and $E_{14} = E_y$ and so S is a valid postfix expression.

12a

1AA	2AA
1AB	2AB
1BA	2BA
1BB	2BB

12b One such string is 12AAB.

We note that this is of the form <*D*><*S*><*S*> where <*D*> = 1, <*S*> = 2AA and <*S*> = B. The middle <*S*> is also of the form <*D*><*S*><*S*> where <*D*> = 2, <*S*> = A and the other <*S*> also equals A.

13a <*S*> = A | B | C <*S*> | D <*S*>

13b CAB is not in the language since both A and B are terminals and there is no expression permitting a sequence of terminals.

13c

```
#include <string>
using namespace std;

// Determines if str is in L
// Precondition:  none.
// Postcondition:  returns true if str is in L, false otherwise.
bool isInL(string str)
{
   int size = str.size();
   char firstChar = str[0];

   // Base case
   if ((size == 1) &&
       ((firstChar == 'A') || (firstChar == 'B')))
      return true;
   else if ((size > 0) &&
            ((firstChar == 'C') || (firstChar == 'D')))
           // Trim off initial character from s
      return isInL(str.substr(1, size - 1));
   else
      return false;
} // end isInL
```

---

14 The recognition algorithm uses recursion to backtrack, reading in the character indexed by `first` (if any), calling the function recursively on the substring indexed by `first` + 1 and `last - 1`, and, if returning true, compares the character at index `last` with that of `first`. A C++ implementation is as follows:

```
#include <string>
using namespace std;

bool isPal(string str, int first, int last)
{
   if (last < first)
      return false;
   else if ((first == last) &&  // Base case
            (str[first] == '$'))
      return true;
   else if (first < last)
      return (isPal(str, first+1, last-1) &&
              (str[first] == str[last]));
   else
      return false;
} // end isPal
```

15a

$$m(x) = \begin{cases} 0 & x < 3 \\ m(x-1) + m(x-3) + 1 & x \geq 3 \end{cases}$$

15b **Base case**: By definition, $m(x)$ returns 0 for the case $x < 3$.
Inductive hypothesis: Let $m(x) = m(x\text{-}1) + m(x\text{-}3) + 1$, $x \geq 3$.

Inductive conclusion: Consider $m(x+1)$: This function returns the number of multiplications performed by a call to $p(x+1)$. We know $x+1 > 3$ since $x \geq 3$ by inductive hypothesis. Therefore the function $p()$ will make a recursive call to itself with the arguments $p((x+1)\text{-}1)$ and $p((x+1)\text{-}3)$ and then perform an additional single multiplication of the return values of these function calls. We also know from inductive hypothesis $p((x+1)\text{-}1)$ or $p(x)$ performs $m(x)$ multiplications and $p((x+1)\text{-}3)$ or $p(x\text{-}2)$ performs $m(x\text{-}2)$ multiplications. Thus the total number of multiplications for $m(x+1) = m(x) + m(x\text{-}2) + 1$, i.e.: $m(x+1) = m((x+1)\text{-}1) + m((x+1)\text{-}3) + 1$.

16a

$$c(n) = \begin{cases} 1 & n = 0 \quad \text{(the null string is a palindrome)} \\ 26 & n = 1 \\ 26 \cdot c(n-2) & n > 1 \end{cases}$$

16b **Base case(s):** The null string is the same backwards and forwards so it counts as a palindrome. Since the number of lowercase letters is 26, this is the number of possible palindromes of length 1.
Inductive hypothesis: $c(n) = 26*c(n\text{-}2)$, $n > 1$

Inductive conclusion: Consider $c(n+1)$: Since the string is a palindrome we know that its starting and finishing characters are identical and that there are 26 possible letters for this repeated character. Since this starting/ending character may be chosen independently from the characters in the substring embraced by this set, the count of all possible palindromes of length $n+1$ is then 26 times the count of all possible palindromes of the length of this substring, $(n+1)\text{-}2$ or $n\text{-}1$. By hypothesis we know that the equation holds for all integers i, $1 < i \leq n$. Thus, $c(n+1) = 26*c((n+1)\text{-}2)$ or $c(n+1) = 26*c(n\text{-}1)$.

---

17 If E is a prefix expression, then EY is not a prefix expression for any nonempty Y (i.e., a prefix expression cannot be the initial substring of another prefix expression.) We will give a proof by induction on $|E|$: the number of characters in E.

Basis $|E| = 1$: Definition of prefix implies E is a single letter and definition of prefix implies EY must begin with an operator.
Inductive hypothesis: For all E with $1 < |E| < n$ and for all nonempty Y, if E is prefix, then EY is NOT prefix.

Inductive conclusion: Say $|E| = n$, then $E = \text{op } E_1\, E_2$ for some E_1 and E_2, which implies that E_1 and E_2 are prefix expressions and that $|E_1|, |E_2| < \text{n}$. If EY is prefix for some Y, then $EY = \text{op } W_1\, W_2$ for some W_1 and W_2, and W_1 and W_2 are prefix expressions.

We claim that $E_1 = W_1$; if not, then one is a substring of the other, so both cannot be a prefix by inductive hypothesis ($|E_1| < \text{n}$).

Summarizing what we have so far:

$E = \text{op } E_1\, E_2$

and

$E = \text{op } W_1\, W_2$
$E = \text{op } E_1\, W_2$
$E = \text{op } E_1\, E_2\, Y$

But this implies that $W_2 = E_2Y$, which cannot be because E_2 and W_2 are both prefixes and E_2 is an initial substring of W_2.

---

18 Proof by induction on *n* for all *n* and *k* such that $0 \le k \le n$.

Basis: Show true for $n = 0$. That is, show

$$g(0,k) = \frac{0!}{(0-k)!k!}$$

for all *k* such that $0 \le k \le 0$.

If $n = 0$, then *k* must equal 0 and we have $g(0, 0) = 1$ by definition. And, $0!/(0!0!) = 1$, so the basis is established.

Inductive hypothesis: Assume true for $n = m$-1. That is, assume

$$g(m-1,k) = \frac{(m-1)!}{(m-1-k)!k!}$$

for all *k* such that $0 \le k \le m - 1$.

Inductive conclusion: We must show the claim to be true for $n = m$. That is, show

$$g(m,k) = \frac{m!}{(m-k)!k!}$$

for all *k* such that $0 \le k \le m$.

There are three cases to consider:

Case 1: If $k = 0$, then we have $g(m, 0) = 1$ by definition and $m!/((m\text{-}0)!0!) = 1$.

Case 2: If $k = m$, then we have $g(m, m) = 1$ by definition and $m!/((m\text{-}m)!\ m!) = 1$.

Case 3: If $0 < k < m$, then we have $g(m, k) = g(m -1, k - 1) + g(m - 1, k)$ by definition, and so by the inductive hypothesis (note $k = m - 1$), we have

$$g(m,k) = \frac{(m-1)!}{((m-1)-(k-1))!(k-1)!} + \frac{(m-1)!}{(m-1-k)!k!} = \frac{m!}{(m-k)!k!}$$

This establishes that the inductive hypothesis implies the inductive conclusion and completes the proof.

Chapter 6 Stacks

1

```
aStack.push("Jill")
aStack.push("Jane")
aStack.push("Jamie")
```

2 Jamie, Jane, Jill.

3a

```
// Put items into tempStack in reverse order
while (!aStack.isEmpty())
{
   stackItem = aStack.peek();
   tempStack.push(stackItem);
   aStack.pop();
}  // end while

// Restore the original stack and display the items
while (!tempStack.isEmpty())
{
   stackItem = tempStack.peek();
   cout << stackItem << endl;
   aStack.push(stackItem);
   tempStack.pop();
}  // end while
```

3b

```
// Counts the number of items in the stack aStack.
int counter = 0;

// Count items as we place them into tempStack in reverse order
while (!aStack.isEmpty())
{
   stackItem = aStack.peek();
   tempStack.push(stackItem);
   aStack.pop();
   counter++;
}  // end while

// Restore the original stack
while (!tempStack.isEmpty())
{
   stackItem = tempStack.peek();
   aStack.push(stackItem);
   tempStack.pop();
}  // end while
```

3c

```
// Examine items as we place them into tempStack in reverse order,
// but remove all occurrences of givenItem
while (!aStack.isEmpty())
{
   stackItem = aStack.peek();
   if (stackItem != givenItem)
      tempStack.push(stackItem);
   aStack.pop();
} // end while

// restore the original order of the remaining stack items
while (!tempStack.isEmpty())
{
   stackItem = tempStack.peek();
   aStack.push(stackItem);
   tempStack.pop();
} // end while
```

---

4 There are three stacks: the upper-left, the upper-right, and the lower-middle section of the track. Pops from the upper-left and the upper-right stacks are pushed onto the lower-middle stack. Pops from the lower-middle stack can be pushed onto either the upper-left or the upper-right stack. Any possible permutation of cars can be constructed.

5

```
/** Removes the topmost n entries from this stack. If the stack has fewer
    than n entries, it will be empty after this operation ends.
    @param n  The number of entries to remove; n >= 0 */
void remove(int n);
```

6a Contents of the stack during input:

```
                         e                   g         h
          c         d    d    d         f    f    f    f
     b    b    b    b    b    b    b    b    b    b    b
a    a    a    a    a    a    a    a    a    a    a    a
```

Contents of the stack during output:

```
h
f    f
b    b    b
a    a    a    a
```

The final output is `hfba`.

6b See the answer to Exercise 3a in this chapter.

6c

```
/** Reads a string of characters and recognizes the character
    '<' as a backspace that erases the previous character read.
    Assumes the character '#' ends the string as a sentinel.
 @return  The corrected string. */
string readAndCorrect()
{
   ArrayStack<char> inputStack;
   ArrayStack<char> tempStack;
   string correctedInput = "";
   char ch;

   cin >> ch;
   while (ch != '#')
   {
      // If not a backspace, add character to stack
      if (ch != '<')
         inputStack.push(ch);

      // Else it is a backspace:
      // pop a character if possible
      else if (!inputStack.isEmpty())
         inputStack.pop();

      cin >> ch;  // Read next character
   }  // end while

   // Reverse the order of the stack
   while (!inputStack.isEmpty())
   {
      ch = inputStack.peek();
      inputStack.pop();
      tempStack.push(ch);
   }  // end while

   // Construct the string to return
   while (!tempStack.isEmpty())
   {
      ch = tempStack.peek();
      tempStack.pop();
      correctedInput = correctedInput + ch;
   }  // end for

   return correctedInput;
}  // end readAndCorrect
```

7

```
aStack = a new empty stack
continue = true

for (each character ch in the string && continue)
{  // Ignore all characters but (, {, [, ], } and )
   if ( (ch != ')') && (ch != '(') &&
        (ch != '}') && (ch != '{') &&
        (ch != ']') && (ch != '[') )
  {
      // Process open delimiter
      else if ( (ch == '(') or (ch == '{') or (ch == '[') )
         aStack.push(ch)

      // Process close delimiter
      else if (!aStack.isEmpty())
      {
         stackItem = aStack.peek()

         // Try to match open and close delimiters of the same kind
         if ( ((str[i] == ')') && (stackItem == '(')) ||
              ((str[i] == '}') && (stackItem == '{')) ||
              ((str[i] == ']') && (stackItem == '[')) )
            aStack.pop()
         else // Open and close parenthesis are not of the same type
            continue = false
      }
      else // Unmatched open and close parenthesis
         continue = false
} // end for

if ( (all of str was examined) && aStack.isEmpty() )
   Report the string has matching parenthesis
else
   Report the string does not have matching parenthesis
```

8a Stack contents are

```
           y
     x     x
```

String rejected: unmatched characters. The x after the $ does not match the y at the top of the stack.

8b Stack contents are

```
           y
     x     x
```

String rejected: unmatched characters. The x after the $ does not match the y at the top of the stack.

8c Stack contents are

```
     y
```

String rejected: stack empty but input not entirely scanned.

8d Stack contents are

```
      y
x     x     x
```

String rejected: stack not empty but all input scanned

8e Stack contents are

```
      x
x     x     x
```

String recognized

9a **SOLUTION 1: Uses two stacks.**

```
 // Tests whether the string word is in the language L.
 // Returns true if the number of A's in word equals the number of B's.
isInL(word: string): boolean

   aStack = a new empty stack
   bStack = a new empty stack
   i = 0
   size = length of word

   while (i < size)
   {
      ch = character a position i of word
      if (ch == 'A')
         aStack.push(ch)
      else if (ch == 'B')
         bStack. push(ch)
      // Else ignore character
      i++
   }  // end while

   while (!aStack.isEmpty() && !bStack.isEmpty())
   {
      aStack.pop()
      bStack.pop()
   }  // end while

   // If word contains equal numbers of A's and B's,
   // both stacks should be empty
   return aStack.isEmpty() && bStack.isEmpty()
```
. . . .

9a SOLUTION 2: Uses one stack.

```
 // Tests whether the string word is in the language L.
 // Returns true if the number of A's in word equals the number of B's.
isInL(word: string): boolean


   aStack = a new empty stack
   i = 0
   size = length of word

   // Count A's and B's; push them into a stack
   countAandB = 0
   while (i < size)
   {
      ch = character a position i of word
      if (ch == 'A' or ch == 'B')
      {
         aStack.push(ch)
         countAandB++
      }  // end if
      // Else ignore character
      i++
   }  // end while

   result = false
   // countAandB must be even, if A and B occur in equal numbers
   if (countAandB % 2 == 0)
   {
      AorBcount = countAandB / 2
      // word is in L if and only if the number of A's and the number of B's
      // each equals AorBcount
      while (!aStack.isEmpty())
      {
         top = aStack.peek()
         aStack.pop()
         if (top == 'A')
            countAandB--  // Subtract A's from count, ignore B's
      }  // end while
      // countAandB is the number of B's

      if (countAandB == AorBcount)
         result = true
   }
   // Else number of A's does not equal number of B's and result is false

   return result
```
. . . .

9a SOLUTION 3: Uses one stack.

```
 // Tests whether the string word is in the language L.
 // Returns true if the number of A's in word equals the number of B's.
isInL(word: string): boolean

   aStack = a new empty stack
   i = 0
   size = length of word
   bCount = 0  // Number of B's
   // Push A’s into stack, count B’s, ignore others
   while (i < size)
   {
      ch = character a position i of word
      if (ch == 'A')
         aStack.push('A')
      else if (ch == 'B')
         bCount++
      // Else ignore character
      i++
   }  // end while

   // Pop stack to match number of A’s with bCount
   while (!aStack.isEmpty())
   {
      aStack.pop()
      bCount--
   }  // end while

   // if bCount == 0, number of A’s == number of B’s;
   // if bCount  > 0, number of A’s  < number of B’s;
   // if bCount  < 0, number of A’s  > number of B’s
   return bCount == 0
```

9b

```
 // Tests whether the string word is in the language L.
 // Returns true if word consists of a number of A's followed by the
 // same number of B's.
isInL(word: string): boolean

   aStack = a new empty stack
   i = 0
   size = length of word

   // While there are A's in the string, push onto stack
   ch = character a position i of word
   while (i < size and ch == 'A')
   {
      aStack.push(str[i])
      i++
      ch = character a position i of word
   }  // end while

   // Should only be B's in rest of string
   while (i < size and !aStack.isEmpty and ch == 'B')
   {  aStack.pop()
      i++
      ch = character a position i of word
   }  // end while

   // If word is in L, stack should be empty after we look at entire string
   return (aStack.isEmpty() and i == size)
```

10

```
/** Determines whether a given string is in the language L.
 @return  True if word is in L, else returns false. */
bool isInL(string word)
{
   ArrayStack<char> stack;

   int numberOfCharacters = (int)word.length();
   if ((numberOfCharacters > 0) && (numberOfCharacters % 2 == 0))
   {
      // word contains an even number of characters.
      // Place first half of the word into a stack
      int halfNumber = numberOfCharacters / 2;
      for (int i = 0; i < halfNumber; i++)
         stack.push(word[i]);

      // Compare second half of word with reverse of first half
      int i = halfNumber;
      bool inLanguage = true;
      while (inLanguage && !stack.isEmpty() && (i < numberOfCharacters))
      {
         char stackTop = stack.peek();
         stack.pop();

         if (stackTop != word[i])
            inLanguage = false;
         else
            i++;
      }  // end while

      if (inLanguage && stack.isEmpty())
         return true;
      else
         return false;
   }
   else  // word is empty or contains an odd number of characters
      return false;
}  // end isInL
```

11a For the following: $a = 7$; $b = 3$; $c = 12$; $d = -5$; $e = 1$.

abc+-

```
Character read            Action              Stack
-----------------------------------------------------------------
a                   push a                    7
b                   push b                    7 3
c                   push c                    7 3 12
+                   op2 = top of stack (12)   7 3 12
                    pop                       7 3
                    op1 = top of stack (3)    7 3
                    pop                       7
                    result = op1 + op2 (15)   7
                    push result               7 15
-                   op2 = top of stack (15)   7 15
                    pop                       7
                    op1 = top of stack (7)    7
                    pop
                    result = op1 - op2 (-8)
                    push result               -8    // **DONE**
```

11b *abc*-*d**-

```
Character read          Action                  Stack
-----------------------------------------------------------------
a                   push a                      7
b                   push b                      7 3
c                   push c                      7 3 12
-                   op2 = top of stack (12)     7 3 12
                    pop                         7 3
                    op1 = top of stack (3)      7 3
                    pop                         7
                    result = op1 - op2 (-9)     7
                    push result                 7 -9
d                   push d                      7 -9 -5
*                   op2 = top of stack (-5)     7 -9 -5
                    pop                         7 -9
                    op1 = top of stack (-9)     7 -9
                    pop                         7
                    result = op1 * op2 (45)     7
                    push result                 7 45
-                   op2 = top of stack (45)     7 45
                    pop                         7
                    op1 = top of stack (7)      7
                    pop
                    result = op1 - op2 (-38)
                    push result                 -38    // **DONE**
```

11c *ab*+*c*-*de**+

```
Character read          Action                  Stack
-----------------------------------------------------------------
a                   push a                      7
b                   push b                      7 3
+                   op2 = top of stack (3)      7 3
                    pop                         7
                    op1 = top of stack (7)      7
                    pop
                    result = op1 + op2 (10)
                    push result                 10
c                   push c                      10 12
-                   op2 = top of stack (12)     10 12
                    pop                         10
                    op1 = top of stack (10)     10
                    pop
                    result = op1 - op2 (-2)
                    push result                 -2
d                   push d                      -2 -5
e                   push e                      -2 -5 1
*                   op2 = top of stack (1)      -2 -5 1
                    pop                         -2 -5
                    op1 = top of stack (-5)     -2 -5
                    pop                         -2
                    result = op1 * op2 (-5)     -2
                    push result                 -2 -5
+                   op2 = top of stack (-5)     -2 -5
                    pop                         -2
                    op1 = top of stack (-2)     -2
                    pop
                    result = op1 + op2 (-7)
                    push result                 -7    // **DONE**
```

---

12 **a** a-b+c ==> ab-c+

Stack	Output
empty	a
-	a
-	ab
+	ab-
+	ab-c
empty	ab-c+

b a/(b*c) ==> abc*/

Stack	Output
empty	a
/	a
/ (	a
/ (	ab
/ (*	ab
/ (*	abc
/ (	abc*
empty	abc*/

c (a+b)*c ==> ab+c*

Stack	Output
(	nothing
(	a
(+	a
(+	ab
(	ab+
*	ab+
*	ab+c
empty	ab+c*

d a-(b+c) ==> abc+-

Stack	Output
empty	a
-	a
- (	a
- (	ab
- (+	ab
- (+	abc
-	abc+
empty	abc+-

e a-(b/c*d) ==> abc/d*-

Stack	Output
empty	a
-	a
- (	a
- (	ab
- (/	ab
- (/	abc
- (	abc/
- (*	abc/
- (*	abc/d
-	abc/d*
empty	abc/d*-

f a/b/c-(d+e)*f ==> ab/c/de +f

Stack	Output
empty	a
/	a
/	ab
/	ab/
/	ab/c
-	ab/c/
- (	ab/c/
- (+	ab/c/d
- (+	ab/c/d
- (+	ab/c/de
-	ab/c/de+
- *	ab/c/de+
- *	ab/c/de+f
-	ab/c/de+f*
empty	ab/c/de+f*-

g a*(b/c/d)+e ==> abc/d/*e+

Stack	**Output**
empty	a
*	a
* (	a
* (	ab
* (/	ab
* (/	abc
* (	abc/
* (/	abc/
* (/	abc/d
*	abc/d/
+	abc/d/*
+	abc/d/*e
empty	abc/d/*e+

h a-(b+c*d)/e ==> abcd*+ e/-

Stack	**Output**
empty	a
-	a
- (	a
- (	ab
- (+	ab
- (+	abc
- (+ *	abc
- (+ *	abcd
- (+	abcd*
-	abcd*+
- /	abcd*+
- /	abcd*+e
-	abcd*+e/
empty	abcd*+e/-

13a Fly from A to F:

```
Action       Reason                      Stack
------------------------------------------------------
Push A       Initialize                 A
Push B       Next adjacent city         A B
Push D       Next adjacent city         A B D
Push E       Next adjacent city         A B D E
Push I       Next adjacent city         A B D E I
Push C       Next adjacent city         A B D E I C
Pop C        No unvisited neighbor      A B D E I
Pop I        No unvisited neighbor      A B D E
Pop E        No unvisited neighbor      A B D
Push F       Next adjacent city         A B D F    **DONE**
```

13b Fly from A to G:

```
Action       Reason                      Stack
------------------------------------------------------
Push A       Initialize                 A
Push B       Next adjacent city         A B
Push D       Next adjacent city         A B D
Push E       Next adjacent city         A B D E
Push I       Next adjacent city         A B D E I
Push C       Next adjacent city         A B D E I C
Pop C        No unvisited neighbor      A B D E I
Pop I        No unvisited neighbor      A B D E
Pop E        No unvisited neighbor      A B D
Push F       Next adjacent city         A B D F
Push G       Next adjacent city         A B D F G    **DONE**
```

13c Fly from F to H:

```
Action       Reason                      Stack
---------------------------------------------------------
Push F       Initialize                  F
Push G       Next adjacent city          F G
Push C       Next adjacent city          F G C
Push B       Next adjacent city          F G C B
Push D       Next adjacent city          F G C B D
Push E       Next adjacent city          F G C B D E
Push I       Next adjacent city          F G C B D E I
Pop I        No unvisited neighbor       F G C B D E
Pop E        No unvisited neighbor       F G C B D
Push H       Next adjacent city          F G C B D H   **DONE**
```

13d Fly from D to A:

```
Action       Reason                      Stack
---------------------------------------------------------
Push D       Initialize                  D
Push E       Next adjacent city          D E
Push I       Next adjacent city          D E I
Push C       Next adjacent city          D E I C
Pop C        No unvisited neighbor       D E I
Pop I        No unvisited neighbor       D E
Pop E        No unvisited neighbor       D
Push F       Next adjacent city          D F
Push G       Next adjacent city          D F G
Pop G        No unvisited neighbor       D F
Pop F        No unvisited neighbor       D
Push H       Next adjacent city          D H
Pop H        No unvisited neighbor       D
Pop D        No unvisited neighbor       -      **FAIL**
```

13e Fly from I to G:

```
Action       Reason                      Stack
---------------------------------------------------------
Push I       Initialize                  I
Push C       Next adjacent city          I C
Push B       Next adjacent city          I C B
Push D       Next adjacent city          I C B D
Push E       Next adjacent city          I C B D E
Pop E        No unvisited neighbor       I C B D
Push F       Next adjacent city          I C B D F
Push G       Next adjacent city          I C B D F G   **DONE**
```

---

14 Here is a C++ function instead of a pseudocode solution:

```
/** Determines whether a given string is a palindrome.
 @return  True if word is a palindrome, else returns false. */
bool isPalindrome(string word)
{
   ArrayStack<char> stack;
   bool isPal = true;

   int numberOfCharacters = (int)word.length();
   if (numberOfCharacters > 0)
   {
      // Place characters in word into a stack
      for (int i = 0; i < numberOfCharacters; i++)
         stack.push(word[i]);

      // Compare word with reverse of itself
      int i = 0;
      while (isPal && !stack.isEmpty() && (i < numberOfCharacters))
      {
         char stackTop = stack.peek();
         stack.pop();

         if (stackTop != word[i])
            isPal = false;
         else
            i++;
      }  // end while
   }
   else  // word is empty
      isPal = false;

   return isPal;
}  // end isPalindrome
```

15a By the axioms *(s.push(item)).pop() = true* and *(s.push(item)).peek() = item*, we can conclude that each *push* immediately followed by a *pop* restores the stack to the state prior to the *push*. We can represent this fact by the notation *(s.push(item)).pop() = s*.

Now let *m = s.push(x)*, *n = m.push(y)*, and *o = n.push(z)*. Then:

```
((o.pop()).pop()).pop()
          = (((n.push(z)).pop()).pop()).pop()     by definition of o
          = (n.pop()).pop()                       by the fact quoted above
          = ((m.push(x)).pop()).pop()             by definition of n
          = m.pop()                               by the fact quoted above
          = (s.push(x)).pop()                     by definition of m
          = s                                     by the fact quoted above
```

Thus the contents of the stack *s* at any point depends on the number of calls to *push* that are not balanced by subsequent calls to *pop*. That is, the contents of any stack can be replicated by a sequence of *push* operations without any *pop* operations, which is the canonical form.

15b By the argument in part ***a*****,** a sequence of paired or nested *push* and *pop* operations restores the stack *s* to the state it had prior to the sequence. In addition, the order of items in *s* are dependent on the order of their insertion. A change in the order of the *push* operations would change the order of the items in the stack. Since the ordering is unique, the canonical form is unique.

15c By repeated application of the fact `(s.push(item)).pop() = s`, as given in the answer to part *a*, we have the following:

```
(((((((((Stack()).push(6)).push(9)).pop()).pop()).push(2)).pop()).push(3)).
   push(1)).pop()).peek()

 = (((((((Stack()).push(6)).pop()).push(2)).pop()).push(3)).push(1)).pop()).
        peek()

 = (((((Stack()).push(2)).pop()).push(3)).push(1)).pop()).peek()

 =  (((Stack()).push(3)).push(1)).pop()).peek()
 =  ((Stack()).push(3)).peek()
```

= *3* by the axiom: `(s.push(item)).peek() = item`

Chapter 7 Implementations of the ADT Stack

1

Advantages of array-based versus link-based:
Compiler supplied copy constructor and destructor are sufficient, making the implementation easier to write.
Best for applications that require a fixed-size stack.
For stacks that might be large, a large array is necessary, even if all of its memory is not used.

Disadvantage of array-based versus link-based:
Using a stack that must grow as needed requires resizing the array. Doing takes time that the link-based version does not require.

Advantage of link-based versus array-based:
The stack grows a node at a time as needed.

Disadvantage of link-based versus array-based:
A copy constructor and destructor must be defined.

2a No. A bag cannot retain the order of a stack's entries.

2b Yes. A bag is indifferent to the order of its entries, so you could store its entries in a stack. However, if you wanted to remove a bag's entry at random, doing so using a stack would be awkward.

3a The methods in each of the three parts of Question 3 display the stack items from top to bottom.

We have arbitrarily added the following method to `ArrayStack`. Notice that this method cannot be a `const` method, because it changes the stack temporarily. The methods in parts *b* and *c*, however, can be `const` methods.

```
template<class ItemType>
void ArrayStack<ItemType>::displayA()
{
   ArrayStack<ItemType> tempStack;

   // Put items into reversed order and
   // display the stack items top to bottom
   while (!isEmpty())
   {
      ItemType stackItem = peek();
      pop();
      tempStack.push(stackItem);
      cout << stackItem << " ";
   }  // end while
   cout << endl;

   // Restore the original stack
   while (!tempStack.isEmpty())
   {
      push(tempStack.peek());
      tempStack.pop();
   }  // end while
}  // end displayA
```

3b The following method is added to `LinkedStack`.

```
template<class ItemType>
void LinkedStack<ItemType>::displayB() const
{
   // Display items top to bottom (top is in the first node of the chain)
   Node<ItemType>* curPtr = topPtr;
   while (curPtr != nullptr)
   {
      cout << curPtr->getItem() << " ";
      curPtr = curPtr->getNext();
   }  // end while
   cout << endl;
}  // end displayB
```

3c The following method is added to `ArrayStack`.

```
template<class ItemType>
void ArrayStack<ItemType>::displayC() const
{
   // Display items top to bottom
   for (int i = top; i >= 0; i--)
      cout << items[i] << " ";
   cout << endl;
}  // end displayC
```

4a The methods in each of the three parts of Question 4 place the stack items from top to bottom into a vector.

We have arbitrarily added the following method to `ArrayStack`. Notice that this method cannot be a `const` method, because it changes the stack temporarily. The methods in parts *b* and *c*, however, can be `const` methods.

```
template<class ItemType>
vector<ItemType> ArrayStack<ItemType>::toVectorA()
{
   vector<ItemType> returnVector;  // Top to bottom

   // Put stack items into a vector in top to bottom order
   while (!isEmpty())
   {
      ItemType stackItem = peek();
      pop();
      returnVector.push_back(stackItem);
   }  // end while

   // Restore the original stack
   for (int i = (int)returnVector.size() - 1; i >= 0 ; i--)
      push(returnVector.at(i));

   return returnVector;
}  // end toVectorA
```

4b The following method is added to `LinkedStack`.

```
template<class ItemType>
vector<ItemType> LinkedStack<ItemType>::toVectorB() const
{
   vector<ItemType> returnVector;  // Top to bottom

   // Put stack items into a vector in top to bottom order
   Node<ItemType>* curPtr = topPtr;
   while (curPtr != nullptr)
   {
      returnVector.push_back(curPtr->getItem());
      curPtr = curPtr->getNext();
   }  // end while

   return returnVector;
}  // end toVectorB
```

4c The following method is added to `ArrayStack`.

```
template<class ItemType>
vector<ItemType> ArrayStack<ItemType>::toVectorC() const
{
   vector<ItemType> returnVector;  // Top to bottom

   // Put stack items into a vector in top to bottom order
   for (int i = top; i >= 0; i--)
      returnVector.push_back(items[i]);

   return returnVector;
}  // end toVectorC
```

---

5 Our first solution uses the method `pop` to remove and delete the nodes from the chain. Although this approach uses an ADT operation, and so is implementation independent, the link-based `pop` operation is efficient.

```
template<class ItemType>
void LinkedStack<ItemType>::remove(int n)
{
   int counter = 0;
   while (!isEmpty() && (counter < n))
   {
      pop();
      counter++;
   }  // end while
}  // end remove
```

Our second solution removes the first *n* nodes as a subchain of the original chain and then deletes the nodes in the subchain, returning them to the system using code like that in the method `pop`. This approach is more susceptible to programming errors than our first solution.

```
template<class ItemType>
void LinkedStack<ItemType>::remove2(int n)
{
   // Locate (n + 1)-st node
   int counter = 0;
   Node<ItemType>* curPtr = topPtr;


   while ((curPtr != nullptr) && (counter < n))
   {
      curPtr = curPtr->getNext();
      counter++;
   }  // end while

   // Extract the subchain consisting of the first n nodes
   Node<ItemType>* subchainPtr = topPtr;
   topPtr = curPtr;

   // Delete the subchain
   for (int counter = 0; counter < n; counter++)
   {
      Node<ItemType>* nodeToDeletePtr = subchainPtr;
      subchainPtr = subchainPtr->getNext();

      // Return deleted node to system
      nodeToDeletePtr->setNext(nullptr);
      delete nodeToDeletePtr;
      nodeToDeletePtr = nullptr;
   }  // end for
}  // end remove2
```

---

6 As for the link-based implementation given in Question 5, we can use the same definition of `remove` that calls the method `pop`. Despite an efficient implementation of `pop`, we can do much better in the array-based implementation by simply changing the value of the index `top`. The method `remove2` shows this improved approach.

```
template<class ItemType>
void ArrayStack<ItemType>::remove(int n)
{
   int counter = 0;
   while (!isEmpty() && (counter < n))
   {
      pop();
      counter++;
   }
}  // end remove

template<class ItemType>
void ArrayStack<ItemType>::remove2(int n)
{
   // Decrease the value of the index top to
   // reference the new top of the stack.
   top = top - n;
   // Note that if n was > top, top will now be negative;
   // top < 0 signals an empty stack as desired.
}  // end remove2
```

7

```
template<class ItemType>
LinkedStack<ItemType>::~LinkedStack()
{
   while (topPtr != nullptr)
   {
      Node<ItemType>* nodeToDeletePtr = topPtr;
      topPtr = topPtr->getNext();

      // Return node to system
      nodeToDeletePtr->setNext(nullptr);
      delete nodeToDeletePtr;
   }  // end while
   // topPtr == nullptr, so stack is empty
}  // end destructor
```

---

8 See the answer to Question 11. Question 8 asks for the same copy constructor as Question 11.

---

9 Let the bottom of the stack be in the last element of the array. Thus, the index `top`—which is a data member of the class—is initialized to the size of the array, `MAX_STACK`, thus indicating an empty stack. To push an item onto the stack, you decrement `top` and use it as an index to the array. To pop an item, you simply increment `top`.

The header file for the class `ArrayStack` is the same as the one given in Listing 7-1. The implementation file follows:

```
/** @file ArrayStack.cpp */
#include <cassert>       // For assert
#include "ArrayStack.h"  // Header file

template<class ItemType>
ArrayStack<ItemType>::ArrayStack() : top(MAX_STACK)
{
}  // end default constructor

// Copy constructor and destructor are supplied by the compiler

template<class ItemType>
bool ArrayStack<ItemType>::isEmpty() const
{
   return top >= MAX_STACK;
}  // end isEmpty

template<class ItemType>
bool ArrayStack<ItemType>::push(const ItemType& newEntry)
{
   bool result = false;
   if (top > 0)  // Does stack have room for newEntry?
   {
      top--;
      items[top] = newEntry;
      result = true;
   }  // end if

   return result;
}  // end push
```

```
template<class ItemType>
bool ArrayStack<ItemType>::pop()
{
   bool result = false;
   if (!isEmpty())
   {
      top++;
      result = true;
   }  // end if

   return result;
}  // end pop

template<class ItemType>
ItemType ArrayStack<ItemType>::peek() const
{
   assert(!isEmpty());  // Enforce precondition

   // Stack is not empty; return top
   return items[top];
}  // end peek
// End of implementation file.
```

---

10 Maintain an external pointer to the last node in the chain. Also, have each node keep a pointer to the previous node. The methods `push`, `pop`, and `peek` are implemented as follows:
• To push a node, use the external pointer to find the end of the chain and then link in the new node. Afterwards, update the external pointer.
• To peek, use the external pointer to find the last node. Return that node's data value, since it is the stack's top.
• To pop, use the external pointer to find the last node. Using the previous pointer in that node, update the external pointer.

```
/** @file DLNode.h */
#ifndef _DLNODE
#define _DLNODE

template<class ItemType>
class DLNode
{
private:
   ItemType           item; // A data item
   DLNode<ItemType>* next; // Pointer to next node
   DLNode<ItemType>* prev; // Pointer to previous node

public:
   DLNode();
   DLNode(const ItemType& anItem);
   void setItem(const ItemType& anItem);
   void setNext(DLNode<ItemType>* nextNodePtr);
   void setPrevious(DLNode<ItemType>* previousNodePtr);
   ItemType getItem() const;
   DLNode<ItemType>* getNext() const;
   DLNode<ItemType>* getPrevious() const;
}; // end DLNode

#include "DLNode.cpp"
#endif
```

```
/** @file DLNode.cpp */
#include "DLNode.h"
#include <cstddef>

template<class ItemType>
DLNode<ItemType>::DLNode() : next(nullptr), prev(nullptr)
{
} // end default constructor

template<class ItemType>
DLNode<ItemType>::DLNode(const ItemType& anItem) : item(anItem), next(nullptr),
                                                   prev(nullptr)
{
}  // end constructor

template<class ItemType>
void DLNode<ItemType>::setItem(const ItemType& anItem)
{
   item = anItem;
}  // end setItem

template<class ItemType>
void DLNode<ItemType>::setNext(DLNode<ItemType>* nextNodePtr)
{
   next = nextNodePtr;
}  // end setNext

template<class ItemType>
void DLNode<ItemType>::setPrevious(DLNode<ItemType>* previousNodePtr)
{
   prev = previousNodePtr;
}  // end setPrevious

template<class ItemType>
ItemType DLNode<ItemType>::getItem() const
{
   return item;
}  // end getItem

template<class ItemType>
DLNode<ItemType>* DLNode<ItemType>::getNext() const
{
   return next;
}  // end getNext

template<class ItemType>
DLNode<ItemType>* DLNode<ItemType>::getPrevious() const
{
   return prev;
}  // end getPrevious
```

```
/** ADT stack: Link-based implementation.
    @file LinkedStack.h */

#ifndef _LINKED_STACK
#define _LINKED_STACK

#include "StackInterface.h"
#include "DLNode.h"
#include <vector>

using namespace std;
```

```
template<class ItemType>
class LinkedStack : public StackInterface<ItemType>
{
private:
   DLNode<ItemType>* headPtr;  // Pointer to first node in the chain
   DLNode<ItemType>* topPtr;   // Pointer to (last) node in the chain
                               //   that contains the stack's top

public:
   // Constructors and destructor:
   LinkedStack();                                    // Default constructor
   LinkedStack(const LinkedStack<ItemType>& aStack); // Copy constructor
   virtual ~LinkedStack();                           // Destructor

   // Stack operations:
   bool isEmpty() const;
   bool push(const ItemType& newItem);
   bool pop();
   ItemType peek() const;
   vector<ItemType> toVector() const;
}; // end LinkedStack

#include "LinkedStack.cpp"
#endif
```

```
/** @file LinkedStack.cpp */

#include <cassert>        // For assert
#include "LinkedStack.h"  // Header file

template<class ItemType>
LinkedStack<ItemType>::LinkedStack() : headPtr(nullptr), topPtr(nullptr)
{
}  // end default constructor

template<class ItemType>
LinkedStack<ItemType>::LinkedStack(const LinkedStack<ItemType>& aStack)
{
   // Point to nodes in original chain
   DLNode<ItemType>* origChainPtr = aStack->headPtr;

   if (origChainPtr == nullptr)
      topPtr = nullptr;  // Original stack is empty
   else
   {
      // Copy first node
      headPtr = new DLNode<ItemType>();
      headPtr->setItem(origChainPtr->getItem());
      topPtr = headPtr;

      // Point to last node in new chain
      DLNode<ItemType>* newChainPtr = topPtr;

      // Copy remaining nodes
      while (origChainPtr != nullptr)
      {
         // Advance original-chain pointer
         origChainPtr = origChainPtr->getNext();

         // Get next item from original chain
         ItemType nextItem = origChainPtr->getItem();
```

```
            // Create a new node containing the next item
            DLNode<ItemType>* newDLNodePtr = new DLNode<ItemType>(nextItem);

            // Link new node to end of new chain
            newChainPtr->setNext(newDLNodePtr);

            // Advance pointer to new last node
            newChainPtr = newChainPtr->getNext();
         }  // end while

         newChainPtr->setNext(nullptr);            // Flag end of chain
      }  // end if
   }  // end copy constructor

   template<class ItemType>
   LinkedStack<ItemType>::~LinkedStack()
   {
      // Pop until stack is empty
      while (!isEmpty())
         pop();
   }  // end destructor

   template<class ItemType>
   bool LinkedStack<ItemType>::isEmpty() const
   {
      return headPtr == nullptr;
   }  // end isEmpty

   template<class ItemType>
   bool LinkedStack<ItemType>::push(const ItemType& newItem)
   {
      DLNode<ItemType>* newNodePtr = new DLNode<ItemType>(newItem);
      if (isEmpty())
         headPtr = newNodePtr;
      else
      {
         // Add node to end of chain
         newNodePtr->setPrevious(topPtr);  // Make new node point to old last node
         topPtr->setNext(newNodePtr);      // Make old last node point to new node
      }  // end if

      topPtr = newNodePtr;
      newNodePtr = nullptr;

      return true;
   }  // end push

   template<class ItemType>
   bool LinkedStack<ItemType>::pop()
   {
      bool result = false;
      if (!isEmpty())
      {
         // Stack is not empty; delete top
         DLNode<ItemType>* nodeToDeletePtr = topPtr;
         topPtr = topPtr->getPrevious();
         if (topPtr == nullptr)
            headPtr = nullptr;  // Stack is now empty

         // Return deleted node to system
         nodeToDeletePtr->setNext(nullptr);
         delete nodeToDeletePtr;
         nodeToDeletePtr = nullptr;

         result = true;
      }  // end if

      return result;
   }  // end pop

   template<class ItemType>
   ItemType LinkedStack<ItemType>::peek() const
   {
      assert(!isEmpty());  // Enforce precondition

      // Stack is not empty; return top
      return topPtr->getItem();
   }  // end peek

   template<class ItemType>
   vector<ItemType> LinkedStack<ItemType>::toVector() const
   {
      vector<ItemType> returnVector;  // Top to bottom

      // Put stack items into a vector in top to bottom order
      DLNode<ItemType>* curPtr = topPtr;
      while (curPtr != nullptr)
      {
         returnVector.push_back(curPtr->getItem());
         curPtr = curPtr->getPrevious();
      }  // end while

      return returnVector;
   } // end toVector

   // End of implementation file.
```

11

```
   template<class ItemType>
   LinkedStack<ItemType>::LinkedStack(const LinkedStack<ItemType>& aStack)
   {
      // Point to nodes in original chain
      Node<ItemType>* origChainPtr = aStack.topPtr;

      if (origChainPtr == nullptr)
         topPtr = nullptr;  // Original bag is empty
      else
      {
         // Set a pointer to traverse the original chain.
         // Must save origChainPtr in case something goes wrong.
         Node<ItemType>* curPtr = origChainPtr;

         try
         {
            // Copy first node
            topPtr = new Node<ItemType>();
            topPtr->setItem(curPtr->getItem());

            // Point to last node in new chain
            Node<ItemType>* newChainPtr = topPtr;

            // Copy remaining nodes
            while (curPtr != nullptr)
            {
```

```
            // Advance original-chain pointer
            curPtr = curPtr->getNext();

            // Get next item from original chain
            ItemType nextItem = curPtr->getItem();

            // Create a new node containing the next item
            Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);

            // Link new node to end of new chain
            newChainPtr->setNext(newNodePtr);

            // Advance pointer to new last node
            newChainPtr = newChainPtr->getNext();
         }  // end while

         newChainPtr->setNext(nullptr);           // Flag end of chain
      }
      catch (bad_alloc allocationFailure)
      {
         // Memory allocation failed

         // Save pointer to partial new chain
         curPtr = topPtr;

         // Restore topPtr to original value
         topPtr = origChainPtr;

         // Delete any allocated nodes
         while (curPtr != nullptr)
         {
            Node<ItemType>* nodeToDeletePtr = curPtr;
            curPtr = curPtr->getNext();

            // Return node to system
            nodeToDeletePtr->setNext(nullptr);
            delete nodeToDeletePtr;
         }  // end while
         throw MemoryAllocationException(
              "Memory allocation failure in copy constructor.");
      }  // end try/catch
   }  // end if
}  // end copy constructor
```

Chapter 8 Lists

1

```
// Computes the sum of the integers in the list aList.
// Precondition: aList is a list of integers.
// Postcondition: The sum of the integers in aList is returned.
addList(aList: List) : integer

   sum = 0
   position = 1
   while (position <= aList.getLength())
   {
      // sum is the total of the integers in the first position elements of aList.

      next = aList.getEntry(position)
      sum = sum + next
      position ++
   }
   return sum
```

2 Since we are writing a client function, we need to know the data type of the entries in the list. We assume that they are strings. We also assume that `List` is a class of lists.

```
/** Interchanges two entries at given positions within a given list.
 @param aList  A given list of strings.
 @param i  The integer position of the first of two entries to swap.
 @param j  The integer position of the second of two entries to swap.
 @return  True if the interchange is successful, otherwise false. */
bool swap(List<string>& aList, int i, int j)
{
   bool result = false;
   int size = aList.getLength();
   if ( (i > 0) && (i <= size ) && (j > 0) && (j <= size ) )
   {
      result = true;

      // Copy the ith and jth entries.
      string ithEntry = aList.getEntry(i);
      string jthEntry = aList.getEntry(j);

      // Replace the ith entry with the jth entry.
      aList.remove(i);
      aList.insert(i, jthEntry);

      // Replace the jth entry with the ith entry.
      aList.remove(j);
      aList.insert(j, ithEntry);
   }  // end if

   return result;
}  // end swap
```

3

```
/** Reverses the order of the entries in a given list.
 @param aList  A given list of strings. */
void reverse(ArrayList<string>& aList)
{
   int size = aList.getLength();
   int halfSize = size / 2;
   for (int i = 1; i <= halfSize; i++)
      swap(aList, i, size - i + 1);
}  // end reverse
```

4a Advantage: The functions are easy to write, because they use operations of the ADT list. And because `displayList` is a client function, its output can be designed to suit the client.
Disadvantage: The time that each function requires to execute depends on how the ADT list is defined. You will see how this is true in the next chapter.

4b Advantage: As ADT operations, `displayList` and `replace` can take advantage of how the list's data is stored and, therefore, operate as quickly as possible.
Disadvantage: As an ADT operation, `displayList` will produce output to suit the designer of the class but not the client. For this reason, we typically do not define ADT operations that produce displayed output. In general, ADT designers minimize the number of operations. Such designers might consider `replace` to be an unnecessary operation, since it has a simple implementation at the client level. On the other hand, `replace` as an operation of the ADT list is a convenience for the client.

5

```
   /** Locates a given object in this list.
    @param anObject  A given object.
    @return  The position of anObject in the list.
       If the object does not exist in the list, returns 0. */
   int getPosition(ItemType anObject)
```

---

6 We assume a list of strings.

```
// Gets the position of a given string that exists in a given list.
// Returns the integer position of the string in the list beginning with 1.
// If the object does not exist in the list, returns 0.
getPosition(aList: List, givenString: string) : integer

   position = 0
   found = false
   while (!found)
   {
      if (givenString == aList.getEntry(position + 1)))
      found = true
      position++
   }
   return position
```

7

```
/** Sees whether a given list contains a given entry.
 @param list  The given list.
 @param anEntry  The object that is the desired entry.
 @return  True if the list contains anEntry, or false if not. */
bool contains(ItemType anEntry)
```

---

8 We assume a list of strings.

```
// Sees whether a given list contains a given entry.
// Returns true if the list contains anEntry, or false if not.
contains(aList: List, anEntry: string) : boolean

    foundInList = false
    pos = 1
    while (pos <= nameList.getLength() && !foundInList)
    {
       listItem = aList.getEntry(pos)
       if (listItem == anEntry)
          foundInList = true
       else
          pos++
    }
    return foundInList
```

9

```
/** Removes the first occurrence of a given entry from this list.
 @return  True if the removal is successful, or false if not. */
bool remove(ItemType anEntry)
```

---

10 We assume that we have the client function getPosition, as given previously in the answer to Exercise 6.

```
// Removes the first occurrence of a given entry from a given list.
// Returns true if the removal is successful, or false if not.
remove(aList: List, anEntry: string) : boolean

    position = getPosition(aList, anEntry)
    return remove(position)
```

---

11 The first sequence of insertions is written as follows:

`((aList.insert(2, A)).insert(2, B)).insert(2, C)`

It is sufficient to show the characters *A*, *B*, and *C* are at positions 4, 3, and 2 in this list.

Let *m* represent `aList.insert(2, A)` and *n* represent `m.insert(2, B)`
Now retrieve the second, third, and fourth items in the list `n.insert(2, C)` as follows:

Show the second item, *C*:
`(n.insert(2, c)).getEntry(2) = c` by axiom 9

Show the third item, *B*:
`(n.insert(2, C)).getEntry(3) = n.getEntry(2)` by axiom 10
= `(m.insert(2, B)).getEntry(2)` by definition of *n*
= *B* by axiom 9

Show the fourth item, *A*:
`(m.insert(2, B)).getEntry(3) = m.getEntry(2)` by axiom 10
= `(aList.insert(2, A)).getEntry(2)` by definition of *m*
= *A* by axiom 9

12 We assume that the class `ArrayList` implements the ADT list.

Our first solution is a naïve approach that uses a list of *n* entries to compute *f*(*n*):

```
/** Computes the function f(n) such that
    f(1) = 1, f(2)= 1, f(3) = 1, f(4) = 3, f(5) = 5, and
    f(n) = f(n - 1) + 3 x f(n - 5) for any n > 5.
 @param n  An integer greater than 0. */
int f(int n)
{
   ArrayList<int> fList;
   fList.insert(1, 1);  // f(1) = 1
   fList.insert(2, 1);  // f(2) = 1
   fList.insert(3, 1);  // f(3) = 1
   fList.insert(4, 3);  // f(4) = 3
   fList.insert(5, 5);  // f(5) = 5
   for (int i = 6; i <= n; i++)
   {
      int fi = fList.getEntry(i - 1) + 3 * fList.getEntry(i - 5);
      fList.insert(i, fi);  // f(i) = f(i - 1) + 3 x f(i - 5)
   }  // end for

   return fList.getEntry(n);
}  // end f
```

Our second solution retains only the last five values of *f*(*n*), as described in Exercise 22 of Chapter 2, in a list of five entries:

```
/** Computes the function f(n) such that
 f(1) = 1, f(2)= 1, f(3) = 1, f(4) = 3, f(5) = 5, and
 f(n) = f(n - 1) + 3 x f(n - 5) for any n > 5.
 @param n  An integer greater than 0. */
int f(int n)
{
   ArrayList<int> last5;
   last5.insert(1, 1);  // f(1) = 1
   last5.insert(2, 1);  // f(2) = 1
   last5.insert(3, 1);  // f(3) = 1
   last5.insert(4, 3);  // f(4) = 3
   last5.insert(5, 5);  // f(5) = 5
   for (int i = 5; i < n; i++)
   {
      int fi = last5.getEntry((i - 1) % 5 + 1) +
               3 * last5.getEntry((i - 5) % 5 + 1);

      // Replace entry in list by using a remove/insert sequence
      last5.remove(i % 5 + 1);
      last5.insert(i % 5 + 1, fi);  // f(i) = f(i - 1) + 3 x f(i - 5)
   }  // end for

   return last5.getEntry((n - 1) % 5 + 1);
}  // end f
```

We tolerate the small extra effort required to compute `f(n)` when `n` < 5 to simplify the logic. We justify this approach because we assume that a client rarely will call the function when `n` < 5.

Chapter 9 List Implementations

1

```
template<class ItemType>
ArrayList<ItemType>::ArrayList(ItemType entries[], int size) :
                                                    maxItems(DEFAULT_CAPACITY)
{
   itemCount = min(size, maxItems);
   for (int i = 0; i < itemCount; i++)
      items[i] = entries[i];
}  // end constructor

template<class ItemType>
LinkedList<ItemType>::LinkedList(ItemType entries[], int size) : headPtr(nullptr),
                                                                 itemCount(0)
{
   for (int index = size - 1; index >= 0; index--)
      insert(1, entries[index]); // Add entries in reverse order at the list's front
}  // end constructor
```

2

```
template<class ItemType>
void LinkedList<ItemType>::setEntry(int position, const ItemType& newEntry)
                           throw(PrecondViolatedExcep)
{
   // Enforce precondition
   bool ableToSet = (position >= 1) && (position <= itemCount);
   if (ableToSet)
   {
      Node<ItemType>* nodePtr = getNodeAt(position);
      nodePtr->setItem(newEntry);
   }
   else
   {
      string message = "setEntry() called with an invalid position.";
      throw(PrecondViolatedExcep(message));
   }  // end if
}  // end setEntry
```

3

```
template<class ItemType>
LinkedList<ItemType>::LinkedList(const LinkedList<ItemType>& aList) :
                                                        itemCount(aList.itemCount)
{
   Node<ItemType>* origChainPtr = aList.headPtr;  // Points to nodes in original chain

   if (origChainPtr == nullptr)
      headPtr = nullptr;  // Original list is empty
   else
   {
      // Copy first node
      headPtr = new Node<ItemType>();
      headPtr->setItem(origChainPtr->getItem());
```

```
         // Copy remaining nodes
         Node<ItemType>* newChainPtr = headPtr;     // Points to last node in new chain
         while (origChainPtr != nullptr)
         {
            // Get next item from original chain
            ItemType nextItem = origChainPtr->getItem();

            // Create a new node containing the next item
            Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);

            // Link new node to end of new chain
            newChainPtr->setNext(newNodePtr);

            // Advance pointer to new last node
            newChainPtr = newChainPtr->getNext();
            origChainPtr = origChainPtr->getNext();
         }  // end while

         newChainPtr->setNext(nullptr);           // Flag end of chain
      }  // end if
   }  // end copy constructor
```

4

```
   template<class ItemType>
   LinkedList<ItemType>::LinkedList(const LinkedList<ItemType>& aList) :
                                                               itemCount(aList.itemCount)
   {
      headPtr = copyChain(aList.headPtr);
   }  // end copy constructor

   template<class ItemType>
   Node<ItemType>* LinkedList<ItemType>::copyChain(Node<ItemType>* chainPtr)
   {
      Node<ItemType>* origChainPtr = chainPtr;  // Points to nodes in original chain
      Node<ItemType>* newChainPtr = nullptr;

      if (origChainPtr != nullptr)
      {
         // Original list is not empty
         // Copy first node
         newChainPtr = new Node<ItemType>();
         newChainPtr->setItem(origChainPtr->getItem());

         // Link new node to a copy of the rest of the chain
         newChainPtr->setNext(copyChain(origChainPtr->getNext()));
      }  // end if

      return newChainPtr;
   }  // end copyChain
```

5

```
    /** Locates a given object in this list.
    @param anObject  A given object.
       @return  The position of anObject in the list.
       If the object does not exist in the list, returns 0. */
   int getPosition(ItemType anObject);
```

Array-based:

```
template<class ItemType>
int ArrayList<ItemType>::getPosition(const ItemType& anEntry) const
{
   int position = 0;
   if (!isEmpty())
   {
      int index = 0;
      while ((position == 0) && (index < itemCount)) // while not found
      {
         if (anEntry == items[index])
            position = index + 1;                    // anObject is located
         else
            index++;
      }  // end while
   }  // end if

   return position;
}  // end getPosition
```

Link-Based:

```
template<class ItemType>
int LinkedList<ItemType>::getPosition(const ItemType& anEntry) const
{
   int position = 0;
   if (!isEmpty())
   {
      int index = 0;
      Node<ItemType>* curPtr = headPtr;
      while ((position == 0) && (index < itemCount)) // While not found
      {
         if (anEntry == curPtr->getItem())
            position = index + 1;                    // anEntry is located
         else
         {
            index++;
            curPtr = curPtr->getNext();
         }  // end if
      }  // end while
   }  // end if

   return position;
}  // end getPosition
```

6
Array-based:

```
template<class ItemType>
int ArrayList<ItemType>::getPosition(const ItemType& anEntry) const
{
   return search(0, anEntry);
}  // end getPosition

template<class ItemType>
int ArrayList<ItemType>::search(int index, const ItemType& anEntry) const
{
   int position = 0;

   if ((index >= 0) && (index < itemCount))
   {
      if (anEntry == items[index])
         position = index + 1;             // anEntry is located
      else
         position = search(index + 1, anEntry);
   }  // end if

   return position;
}  // end search
```

Link-Based:

```
template<class ItemType>
int LinkedList<ItemType>::getPosition(const ItemType& anEntry) const
{
    return search(1, headPtr, anEntry);
}  // end getPosition

template<class ItemType>
int LinkedList<ItemType>::search(int position, Node<ItemType>* startPtr,
                                 const ItemType& anEntry) const
{
   int result = 0;

   if (startPtr != nullptr)
   {
      if (anEntry == startPtr->getItem())
         result = position;             // anEntry is located
      else
         result = search(position + 1, startPtr->getNext(), anEntry);
   }  // end if

   return result;
}  // end search
```

7
Array-based:

```
template<class ItemType>
bool ArrayList<ItemType>::contains(ItemType anEntry) const
{
   bool found = false;
   for (int index = 0; !found && (index < itemCount); index++)
   {
      if (anEntry == items[index])
         found = true;
   }  // end for

   return found;
}  // end contains
```

Link-Based:

```
template<class ItemType>
bool LinkedList<ItemType>::contains(ItemType anEntry) const
{
   bool found = false;
   Node<ItemType>* curPtr = headPtr;

   while (!found && (curPtr != nullptr))
   {
      if (anEntry == curPtr->getItem())
         found = true;
      else
         curPtr = curPtr->getNext();
   }  // end while

   return found;
}  // end contains
```

8
Array-based:

```
template<class ItemType>
bool ArrayList<ItemType>::contains(ItemType anEntry) const
{
   return holds(anEntry, 0);
} // end contains

template<class ItemType>
bool ArrayList<ItemType>::holds(ItemType anEntry, int startIndex) const
{
   bool found = false;
   if ((startIndex >= 0) && (startIndex < itemCount))
   {
      if (anEntry == items[startIndex])
         found = true;
      else
         found = holds(anEntry, startIndex + 1);
   } // end if

   return found;
} // end holds
```

Link-Based:

```
template<class ItemType>
bool LinkedList<ItemType>::contains(ItemType anEntry) const
{
   return holds(anEntry, headPtr);
} // end contains

template<class ItemType>
bool LinkedList<ItemType>::holds(ItemType anEntry, Node<ItemType>* startPtr) const
{
   bool found = false;
   if (startPtr != nullptr)
   {
      if (anEntry == startPtr->getItem())
         found = true;
      else
         found = holds(anEntry, startPtr->getNext());
   } // end if

   return found;
} // end holds
```

9

```
/** Removes the first occurrence of a given entry from this list.
 @return  True if the removal is successful, or false if not. */
bool remove(ItemType anObject)
```

We assume the existence of and use the method `getPosition`, which we implemented in Exercise 5 of this chapter.

Array-based:

```
template<class ItemType>
bool ArrayList<ItemType>::remove(ItemType anEntry)
{
   int position = getPosition(anEntry);
   return remove(position);
}  // end remove
```

Link-Based:

```
template<class ItemType>
bool LinkedList<ItemType>::remove(ItemType anEntry)
{
   int position = getPosition(anEntry);
   return remove(position);
}  // end remove
```

---

10 Since `remove` calls `getPosition`, we can solve this exercise by implementing the method `getPosition` recursively. We have already done that in the solution to Exercise 6 in this chapter.

11

```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNodeAt(int position) const
{
   return getNode(position, headPtr);
}  // end getNodeAt

template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNode(int position,
                                              Node<ItemType>* startPtr) const
{
   if (position == 1)
      return startPtr;
   else
      return getNode(position - 1, startPtr->getNext());
}  // end getNode
```

Because `getNodeAt` is a private method, we could have altered its parameter list to make it recursive instead of defining the second private method `getNode`. Doing so, however, would have required us to change all of the calls to `getNodeAt` within the class. Adding `getNode` to the class in the header file is simpler to do!

12

```
/** Interface for the ADT double-ended list */

template<class ItemType>
class DEListInterface
{
public:
   virtual bool insertFirst(const ItemType& newEntry) = 0;
   virtual bool insertLast(const ItemType& newEntry) = 0;
   virtual bool removeFirst() = 0;
   virtual bool removeLast() = 0;
   virtual void setFirst(const ItemType& newEntry) = 0;
   virtual void setLast(const ItemType& newEntry) = 0;
   virtual ItemType getFirst() const = 0;
   virtual ItemType getLast() const = 0;

// Operations of the ADT list:
   virtual bool isEmpty() const = 0;
   virtual int getLength() const = 0;
   virtual bool insert(int newPosition, const ItemType& newEntry) = 0;
   virtual bool remove(int position) = 0;
   virtual void clear() = 0;
   virtual ItemType getEntry(int position) const = 0;
   virtual void setEntry(int position, const ItemType& newEntry) = 0;
   virtual int getPosition(const ItemType& anEntry) const = 0;
}; // end DEListInterface
```

Chapter 10 Algorithm Efficiency

1a O(n), where n is the number of people at the party under the assumptions that there is a fixed maximum time between hand shakes, and there is a fixed maximum time for a handshake.
1b O(n2), where n is the number of people at the party under the assumptions that there is a fixed maximum time between hand shakes, and there is a fixed maximum time for a handshake.
1c O(n),where n is the number of steps under the assumptions that you never take a backward step, and there is a fixed maximum time between steps.
1d O(h), where h is the height of the banister under the assumptions that you never slow down and you reach a limiting speed.
1e O(1), under the assumption that you reach a decision and press the button within a fixed amount of time.
1f O(n), where n is the number of the floor under the assumptions that the elevator only stops at floors, there is a fixed maximum time for each stop, and there is a fixed maximum time that an elevator requires to travel between two adjacent floors.
1g O(n), where n is the number of pages in the book and under the assumptions that you never read a word more than twice, you read at least one word in a session, there is a fixed maximum time between sessions, and there is a fixed maximum number of words on a page.

2

```
Repeat (until you reach the top)
{
   Go up n / 2 steps, then down n / 2 - 1 steps
}
```

3a O(n).
3b O(n).
3c O(n).
3d O(n), if the total number of names is n. If the array contains n linked chains and each chain has up to m nodes, the time requirement is O(n) when n does not depend on m or O(n^2) when m is proportional to n.
3e O(1).
3f O(n). This is the worst case, since the entire list must be traversed first.
3g O(1) in the best case; O(n) in the average case or the worst case.
3h O(log n).
3i O(n). The worst case occurs for an array-based stack.
3j O(1).

---

4 O(n^2). The innermost loop on `count` is O(1), since it is independent of n. Thus, the loop on `index` is O(n), making the outmost loop O(n^2).

---

5 The comparisons that affect the time requirement of this function are those made by the `if` statement. The computations involved in controlling a loop are typically insignificant and so we will ignore them.

The outer `for` loop cycles n times. For each of the n passes through this loop, the `while` loop executes the `if` statement j+1 times, thus making j+1 comparisons. Since j increases from 0 to n – 1, the total number of comparisons is

$1 + 2 + 3 + ... + n = n * (n + 1) / 2$

Therefore, the time requirement is O(n^2).

---

6 No. A binary search is O(log n), while a sequential search is O(n), in the worst case.

---

7 By the properties of growth-rate functions given in Section 10.2.3, O($f(x)$) = O($c_n x^n$) = O(x^n).

---

8 We must show that $O(\log_a n) = O(\log_b n)$. We have $\log_a n = \log b\ n / \log b\ a$. Let $c = 1 / (\log_b a)$, which is a constant. Thus, $O(\log_a n) = c * O(\log_b n) = O(\log_b n)$.

---

9 By the properties of growth-rate functions given in Section 10.2.3, O($7n^2 + 5n$) = O($7n^2$) = O(n^2). Since O(n^2) > O(n), O($7n^2 + 5n$) is not O(n).

10a
```
// An O(n^2) method.
void methodA(int theArray[], int n)
{
   int negIndex = 0;
   int posIndex = n - 1;
   bool done = false; // False when swaps occur
   int pass = 1;
   while (!done && (pass < n))
   {
      done = true; // Assume done
      for (int index = negIndex; index <= posIndex; index++)
      {
         if (theArray[index] > 0)
         {
            // Exchange entries
            swap(theArray[index], theArray[posIndex]);
            posIndex--;
            done = false; // Signal exchange
         }
         else // theArray[index] < 0
         {
            // Exchange entries
            swap(theArray[index], theArray[negIndex]);
            negIndex++;
            done = false; // Signal exchange
         } // end if
      }  // end for

      pass++;
   }  // end while
}  // end methodA
```

10b
```
// An O(n) method.
void methodB(int theArray[], int n)
{
   // Place integers from given array into proper positions within a spare array
   int spareArray[n];
   int negIndex = 0;
   int posIndex = n - 1;
   for (int index = 0; index < n; index++)
   {
      if (theArray[index] > 0)
      {
         spareArray[posIndex] = theArray[index];
         posIndex--;
      }
      else
      {
         spareArray[negIndex] = theArray[index];
         negIndex++;
      }  // end if
   }  // end for

   // Copy integers from spare array into given array
   for (int index = 0; index < n; index++)
      theArray[index] = spareArray[index];
}  // end methodB
```

Chapter 11 Sorting Algorithms and Their Efficiency

1

- The `for` loop in the method `selectionSort` requires an assignment to last, n comparisons between `last` and 1, $n - 1$ subtractions from `last`, and $n - 1$ more assignments to `last`. Thus, this loop control requires n assignments, n comparisons, and $n - 1$ subtractions. Thus, this loop control is $O(n)$.
- By similar reasoning, the `for` loop in the method `findIndexOfLargest` is controlled by $O(n)$ operations.
- The $O(n^2)$ operations in the other statements of this sorting algorithm dominate the $O(n)$ operations of the loop control. Thus, the selection sort is $O(n^2)$.

2 Insertion sort

Array	Action
20 80 40 25 60 40	
20 80 40 25 60 40	Copy 20 on top of itself
20 80 80 25 60 40	Copy 40 and shift 80
20 40 80 25 60 40	Insert 40
20 40 40 80 60 40	Copy 25 and shift 40, 80
20 25 40 80 60 40	Insert 25
20 25 40 80 80 40	Copy 60 and shift 80
20 25 40 60 80 40	Insert 60
20 25 40 60 60 80	Copy 40 and shift 60, 80
20 25 40 40 60 80	Insert 40; array is sorted

3 Selection sort

Array	Action
7 12 24 4 19 32	Select largest entry 32
7 12 24 4 19 32	Position 32 by swapping it with itself; select next largest entry 24
7 12 19 4 24 32	Position 24 by swapping it with 19; select next largest entry 19
7 12 4 19 24 32	Position 19 by swapping it with 4; select next largest entry 12
7 4 12 19 24 32	Position 12 by swapping it with 4; select next largest entry 7
4 7 12 19 24 32	Position 7 by swapping it with 4; array is sorted

4 Bubble sort

Array	Action
Pass 1	
12 23 5 10 34	Compare 12 and 23
23 12 5 10 34	Swap 12 and 23; compare 12 and 5
23 12 5 10 34	Compare 5 and 10
23 12 10 5 34	Swap 5 and 10; compare 5 and 34
23 12 10 34 5	Swap 5 and 34
Pass 2	
23 12 10 34 5	Compare 23 and 12
23 12 10 34 5	Compare 12 and 10
23 12 10 34 5	Compare 10 and 34
23 12 34 10 5	Swap 10 and 34; compare 10 and 5
23 12 34 10 5	

Pass 3	
23 12 34 10 5	Compare 23 and 12
23 12 34 10 5	Compare 12 and 34
23 34 12 10 5	Swap 12 and 34; compare 12 and 10
23 34 12 10 5	Compare 10 and 5
23 34 12 10 5	
Pass 4	
23 34 12 10 5	Compare 23 and 34
34 23 12 10 5	Compare 23 and 12
34 23 12 10 5	Compare 12 and 10
34 23 12 10 5	Compare 10 and 5
34 23 12 10 5	Array is sorted

5a Initial array sorted into descending order: 8 6 4 2. Sort it into ascending order.

i. Selection sort

Array	Action
8 6 4 2	Select largest entry 8
2 6 4 8	Position 8 by swapping it with 2; select next largest entry 6
2 4 6 8	Position 6 by swapping it with 4; select next largest entry 4
2 4 6 8	Position 4 by swapping it with itself

ii. Bubble sort

Array	Action
Pass 1	
8 6 4 2	Compare 8 and 6
6 8 4 2	Swap 8 and 6; compare 8 and 4
6 4 8 2	Swap 8 and 4; compare 8 and 2
6 4 2 8	Swap 8 and 2
Pass 2	
4 6 2 8	Compare 4 and 6
4 6 2 8	Compare 6 and 2
4 2 6 8	Swap 6 and 2
Pass 3	
4 2 6 8	Compare 4 and 2
2 4 6 8	Swap 4 and 2

iii. Insertion sort

Array	Action
8 6 4 2	
8 8 4 2	Copy 6 and shift 8
6 8 4 2	Insert 6
6 6 8 2	Copy 4 and shift 8, 6
4 6 8 2	Insert 4
4 4 6 8	Copy 2 and shift 8, 6, 4
2 4 6 8	Insert 2

5b Initial array sorted into ascending order: 2 4 6 8. Sort it into ascending order.

i. Selection sort

Array	Action
2 4 6 8	Select largest entry 8
2 4 6 8	Position 8 by swapping it with itself; select next largest entry 6
2 4 6 8	Position 6 by swapping it with itself; select next largest entry 4
2 4 6 8	Position 4 by swapping it with itself

ii. Bubble sort

Array	Action
Pass 1	
2 4 6 8	Compare 2 and 4
2 4 6 8	Compare 4 and 6
2 4 6 8	Compare 6 and 8

iii. Insertion sort

Array	Action
2 4 6 8	Copy 2; insert it onto itself
2 4 6 8	Copy 4; insert it onto itself
2 4 6 8	Copy 6; insert it onto itself
2 4 6 8	Copy 8; insert it onto itself

6a 300
6b 24

---

7 Any array that is in the reverse order of the sorted order. So if our bubble sort arranges an array into ascending order, an initial array in descending order will cause the bubble sort to have worst case behavior.

---

8 We use the following class `Thing` and write the selection sort in terms of it:

```
class Thing
{
private:
   int sortKey;
   string name;

public:
   Thing();
   void setSortKey(int newKey);
   int getSortKey() const;
   void setName(string newName);
   string getName() const;
}; // end Thing
```

```
int findIndexOfLargest(const Thing theArray[], int size)
{
   int indexSoFar = 0; // Index of largest entry found so far
   for (int currentIndex = 1; currentIndex < size; currentIndex++)
   {
      // At this point, theArray[indexSoFar] >= all entries in
      // theArray[0..currentIndex - 1]
      if (theArray[currentIndex].getSortKey() > theArray[indexSoFar].getSortKey())
         indexSoFar = currentIndex;
   }  // end for

   return indexSoFar; // Index of largest entry
}  // end findIndexOfLargest

void selectionSort(Thing theArray[], int n)
{
   // last = index of the last item in the subarray of items yet
   //        to be sorted;
   // largest = index of the largest item found
   for (int last = n - 1; last >= 1; last--)
   {
      // At this point, theArray[last+1..n-1] is sorted, and its
      // entries are greater than those in theArray[0..last].
      // Select the largest entry in theArray[0..last]
      int largest = findIndexOfLargest(theArray, last+1);

      // Swap the largest entry, theArray[largest], with
      // theArray[last]
      std::swap(theArray[largest], theArray[last]);
   }  // end for
}  // end selectionSort
```

9

The following recursive version of `selectionSort` uses the function `findIndexOfLargest` from the iterative version.

```
template<class ItemType>
void selectionSort(ItemType theArray[], int n)
{
   if (n > 1)
   {
      int largest = findIndexOfLargest(theArray, n);

      // Swap the largest entry, theArray[largest], with theArray[n - 1]
      std::swap(theArray[largest], theArray[n - 1]);
      selectionSort(theArray, n - 1);
   }  // end if
}  // end selectionSort

template<class ItemType>
void bubbleSort(ItemType theArray[], int n)
{
   if (n > 1)
   {
      for (int index = 0; index < n - 1; index++)
      {
         if (theArray[index] > theArray[index + 1])
            std::swap(theArray[index], theArray[index + 1]);
      }  // end for
      bubbleSort(theArray, n - 1);
   }  // end if
}  // end bubbleSort
```

```
 // Inserts the given entry into the sorted array a[begin] through a[end].
template<class ItemType>
void  insertInOrder(ItemType entry, ItemType a[], int begin, int end)
{
   if (entry >= a[end])
      a[end + 1] = entry;
   else if (begin < end)
   {
      a[end + 1] = a[end];
      insertInOrder(entry, a, begin, end - 1);
   }
   else
   {
      a[end + 1] = a[end];
      a[end] = entry;
   } // end if
} // end insertInOrder

template<class ItemType>
void insertionSortHelper(ItemType theArray[], int first, int last)
{
   if (first < last)
   {
      // sort all but the last entry
      insertionSortHelper(theArray, first, last - 1);

      // insert the last entry in sorted order
      insertInOrder(theArray[last], theArray, first, last - 1);
   } // end if
}  // end insertionSortHelper

template<class ItemType>
void insertionSort(ItemType theArray[], int n)
{
   insertionSortHelper(theArray, 0, n - 1);
}  // end insertionSort
```

10

```
mergeSort([20 80 40 25 60 30], 0 , 5)
mergeSort([20 80 40], 0 , 2)
mergeSort([20 80], 0 , 1)
mergeSort([20], 0 , 0)
mergeSort([80], 1 , 1)
merge([20 80], 0 , 0 , 1)
mergeSort([40], 2 , 2)
merge([20 80 40], 0 , 1 , 2)
mergeSort([25 60 30], 3 , 5)
mergeSort([25 60], 3 , 4)
mergeSort([25], 3 , 3)
mergeSort([60], 4 , 4)
merge([25 60], 3 , 3 , 4)
mergeSort([30], 5 , 5)
merge([25 60 30], 3 , 4 , 5)
merge([20 40 80 25 30 60], 0 , 2 , 5)
```

```
        20 80 40 25 60 30

     20 80 40         25 60

   20 80    40        25 60    30

 20    80    40     25    60    30

  20 80     40       25 60     30

   20 40 80            25 30 60

        20 25 30 40 60 80
```

11a The recursive calls to `mergeSort` simply subdivide the array *a* until it can be divided no further, that is, when the size of the subdivided array is 1. At that point, the simpler sub-arrays are merged into a larger array. Only the dimension of the array, the relative values of its first and last indices, govern whether another pair of recursive calls to `mergeSort` takes place. The values within the array have no impact on this mechanism.

11b The items in the array are only swapped during the call to `merge` when the two sorted sub-arrays resulting from the previous two calls to `mergeSort` are merged together. More specifically, this occurs within the first loop of `merge` where two index values, `first1` and `first2`, maintain the locations of the largest as yet unmerged items in the first and second sub-arrays respectively. The smaller of the two items indicated by these indices is merged into the new array, the selected index is incremented, and a new comparison is made. This process continues until one or the other index exceeds the length of the sub-array, whereupon the remaining elements of the other sub-array are copied.

---

12 Since the given array is small, we trace `quickSort` after giving `MIN_SIZE` a value of 3 instead of the suggested 10.

```
quickSort([20 80 40 25 60 30], 0 , 5)
partition([20 80 40 25 60 30], 0 , 5)
quickSort([20 25], 0 , 1)
quickSort([80 60 40], 3 , 5)
partition([80 60 40], 3 , 5)
quickSort([40], 3 , 3)
quickSort([80], 5 , 5)
```

---

13 Because the actual sorting is done in the merge steps, the array will not be sorted. Thus, the new algorithm has no effect on the array.

---

14 If the actual median of the entries in the array is selected as the pivot at each step, `quickSort` will make no more than $\log_2 n$ recursive calls. However, since median-of-three pivot selection might not choose the actual median, the worst case of *n* recursive calls can occur.

---

15 An iterative version of `mergeSort` would begin by considering the *n* entries in the array as *n* sorted segments of size 1 that are already sorted. The first pass would form sorted segments of size 2 by merging adjacent segments of size 1. The second pass would form sorted segments of size 4 by merging adjacent segments of size 2, and so on. Eventually, we will reach one of the following situations:

- Two full segments
- One full segment and a partial second segment
- One full segment only
- One partial segment

In the first case, the full segments are merged into one sorted segment and the algorithm finishes. In the other cases, the merge is handled specially to produce the sorted array.

---

16 The insertion sort and the merge sort can have stable implementations.

17a We will trace the radix sort for the following cards: S2, HT, D6, S4, C9, CJ, DQ, ST, HQ, DK.

S2, HT, D6, S4, C9, CJ, DQ, ST, HQ, DK	Original cards
(S**2**) (S**4**) (D**6**) (C**9**) (H**T**, S**T**) (C**J**) (D**Q**, H**Q**) (D**K**)	Sorted by rank
S2, S4, D6, C9, HT, ST, CJ, DQ, HQ, DK	Combined
(**S**2, **S**4, **S**T) (**H**T, **H**Q) (**C**9, **C**J) (**D**6, **D**Q, **D**K)	Sorted by suit
S2, S4, ST, HT, HQ, C9, CJ, D6, DQ, DK	Combined (Sorted)

17b All 10's would become sorted before all 2's within the same rank. For example, we sort the same cards given in part *a*:

S2*b*, H10, D6*b*, S4*b*, C9*b*, CJ*b*, DQ*b*, S10, HQ*b*, DK*b*	Original cards
(H10, S10) (S2*b*, D6*b*, S4*b*, C9*b*, CJ*b*, DQ*b*, HQ*b*, DK*b*)	Sorted by rightmost character
H10, S10, S2*b*, D6*b*, S4*b*, C9*b*, CJ*b*, DQ*b*, HQ*b*, DK*b*	Combined
(H10, S10) (S2*b*) (S4*b*) (D6*b*) (C9*b*) (CJ*b*) (DQ*b*, HQ*b*) (DK*b*)	Sorted by middle character
H10, S10, S2*b*, S4*b*, D6*b*, C9*b*, CJ*b*, DQ*b*, HQ*b*, DK*b*	Combined
(S10, S2*b*, S4*b*) (H10, HQ*b*) (C9*b*, CJ*b*) (D6*b*, DQ*b*, DK*b*)	Sorted by suit (leftmost character)
S10, S2*b*, S4*b*, H10, HQ*b*, C9*b*, CJ*b*, D6*b*, DQ*b*, DK*b*	Combined (Sorted)

Chapter 12 Sorted Lists and Their Implementations

1a Each `displayStatistics` method invokes the `getArea` method in its own class. Within the `Ball` class, you can make the `displayStatistics` method invoke the inherited `Sphere` class `getArea` method by writing `Sphere::getArea()`.

1b
```
spherePtr->displayStatistics();  ==> Calls the Sphere version of getArea
spherePtr = ballPtr;
spherePtr ->displayStatistics(); ==> Still calls the Sphere version of getArea
ballPtr->displayStatistics();    ==> Calls the Ball version of getArea
```

`spherePtr` is defined as a pointer to `Sphere`, and `ballPtr` is a pointer to `Ball`, which is derived from `Sphere`. Since `displayStatistics` is not declared as a virtual method in `Sphere`, `spherePtr` accesses the functionality inherited from `Sphere` rather than the redefined functionality in `Ball`.

2
```
/** @file pen.h */
#ifndef _PEN
#define _PEN

#include <iostream>
#include "Ball.h"

using namespace std;

const double INK_CAPACITY = 10.0;
const double LINE_THICKNESS = 0.01;
const int POINT_RADIUS = 3;
enum colorType {BLACK, RED, BLUE, GREEN};

class Pen
{
private:
   Ball      point;
   colorType color;
   bool      clickedOn;
   double    inkAmount;

   double getInkPerLineLength(int lineLength) const;

public:
   Pen();
   Pen(colorType newColor, double newInkAmount);

   bool isEmpty() const;
   bool isWritable() const;
   bool clickPen();
   bool write(int lineLength);
   void replaceInkCartridge(colorType newColor, double newInkAmount);
   colorType getColor() const;
   double getInkAmount() const;
};  // end Pen
#endif
```

```
/** Pen.cpp */

#include "Pen.h"

Pen::Pen() : Pen(BLACK, INK_CAPACITY)
{
}  // end default constructor

Pen::Pen(colorType newColor, double newInkAmount) : point(POINT_RADIUS, ""),
         color(newColor), clickedOn(false), inkAmount(newInkAmount)
{
}  // end constructor

bool Pen::isEmpty() const
{
   return (inkAmount  <= 0.0);
}  // end isEmpty

bool Pen::isWritable() const
{
   return clickedOn;
}  // end isWritable

bool Pen::clickPen()
{
   clickedOn = !clickedOn;

   return clickedOn;
}  // end clickPen

bool Pen::write(int lineLength)
{
   double amount = inkAmount - getInkPerLineLength(lineLength);

   if (amount >= 0.0)
   {
      inkAmount = amount;
      return true;
   }
   else
      return false;
}  // end write

void Pen::replaceInkCartridge(colorType newColor, double newInkAmount)
{
   color = newColor;
   inkAmount = newInkAmount;
}  // end replaceInkCartridge

colorType Pen::getColor() const
{
   return color;
}  // end getColor

double Pen::getInkAmount() const
{
   return inkAmount;
}  // end getInkAmount
```

```
double Pen::getInkPerLineLength(int lineLength) const
{
   // Amount of ink used is the surface area of Ball times the
   // LineLength divided by the circumference of the Ball times the
   // thickness factor

   return (point.getArea() * lineLength / point.getCircumference() * LINE_THICKNESS);
}  // end getInkPerLineLength
```

3a `wheelCount`.

3b `wheelCount`, `speed`, and `fuelType`.

4a `landVeh` and `motorVeh`.

4b `motorVeh`.

5a `isBlank` should be protected. Its primary use is as a utility operation used by classes derived from `AlgExpr`. At the same time, it is unnecessary for a client using an instance of `AlgExpr` to have access to the `isBlank` method.

5b No. The client declaring `infixExpr` cannot invoke `endExpression`, because it is a protected method and is inaccessible to the client.

5c Add the keyword `virtual` before the prototype for `isExpression` in the `AlgExpr` definition.

6a `getLength` is a public method declared within the class `Expr`. This method is inherited and not redefined by any of the descendants of `Expr`. Thus, all three objects may invoke the method correctly.

6b `aExp` can call the `isExpression` method declared within `AlgExp`; `inExp` can call the `isExpression` method declared within `InfixExpr`.

6c `inExpr`. Only objects of class `InfixExpr` can invoke the method `valueOf`.

6d

```
void f(AlgExpr& aExp);
. . .

int main()
{
   InfixExpr inExpr;  // Declare object of derived type
      .
      .
   f(inExpr);         // Legal call
}  // end main
```

8

```
/** Interface for the ADT front list
    @file FrontListInterface.h */
#ifndef _FRONT_LIST_INTERFACE
#define _FRONT_LIST_INTERFACE

template<class ItemType>
class FrontListInterface
{
public:
   /** Sees whether this list is empty.
    @return True if the list is empty; otherwise returns false. */
   virtual bool isEmpty() const = 0;

   /** Gets the current number of entries in this list.
    @return The integer number of entries currently in the list. */
   virtual int getLength() const = 0;

   /** Inserts an entry into the first position of this list.
    @param newEntry  The entry to insert into the list.
    @return  True if insertion is successful, or false if not. */
   virtual bool insert(const ItemType& newEntry) = 0;

   /** Removes the entry at the first position from this list.
    @return  True if removal is successful, or false if not. */
   virtual bool remove() = 0;

   /** Removes all entries from this list. */
   virtual void clear() = 0;

   /** Gets the entry at the front of this list.
    @pre  The list is not empty.
    @return  The entry at the front of the list. */
   virtual ItemType getEntry() const = 0;

   /** Replaces the entry at the front of this list.
    @pre  The list is not empty.
    @param newEntry  The replacement entry. */
   virtual void setEntry(const ItemType& newEntry) = 0;
}; // end FrontListInterface
#endif

/** ADT stack.
 @file FrontListStack.h */
#ifndef _FRONT_LIST_STACK
#define _FRONT_LIST_STACK
#include "StackInterface.h"
#include "FrontList.h"

template<class ItemType>
class FrontListStack : public FrontList<ItemType>, public StackInterface<ItemType>
{
public:
  FrontListStack();
  bool isEmpty() const;
  bool push(const ItemType& newEntry);
  bool pop();
  ItemType peek() const;
}; // end FrontListStack
#include "FrontListStack.cpp"
#endif
```

```
/** @file FrontListStack.cpp */
#include <cassert>           // For assert
#include "FrontListStack.h"  // Header file

template<class ItemType>
FrontListStack<ItemType>::FrontListStack() : FrontList<ItemType>()
{
}  // end default constructor

template<class ItemType>
bool FrontListStack<ItemType>::isEmpty() const
{
   return FrontList<ItemType>::isEmpty();
}  // end isEmpty

template<class ItemType>
bool FrontListStack<ItemType>::push(const ItemType& newEntry)
{
   return FrontList<ItemType>::insert(newEntry);
}  // end push

template<class ItemType>
bool FrontListStack<ItemType>::pop()
{
   return FrontList<ItemType>::remove();
}  // end pop

template<class ItemType>
ItemType FrontListStack<ItemType>::peek() const
{
   assert(!isEmpty());  // Enforce precondition

   // Stack is not empty; return top
   return FrontList<ItemType>::getEntry();
}  // end peek
// End of implementation file.
```

---

9 The `Person` class has basic information like name, age, gender. The `toString` method returns a string that describes the person. We make this method a pure virtual method, thereby making `Person` an abstract class.

```
/** @file Person.h */
#include <string>
using namespace std;

class Person
{
private:
   string name;
   int    age;
   char   gender;
public:
   Person();
   virtual string getName() const;
   virtual void   setName(string newName);
   virtual int    getAge() const;
   virtual void   setAge(int newAge);
   virtual char   getGender() const;
   virtual void   setGender(char  newGender);
   virtual string toString() const = 0;
}; // end Person
```

The `Student` class has the basic data of the `Person` class, and adds year of graduation, student ID, major, and a list of courses.

```
/** @file Student.h */

typedef FrontList List;     // Uses FrontList from Exercise 8
                            // for the list of a student's courses
class Student: public Person
{
private:
   string id, major;
   int    yearOfGraduation;
   List   courseList;

public:
   Student();
   virtual string getStudentID() const;
   virtual void   setStudentID(string newId);

   virtual string getMajor() const;
   virtual void   setMajor(string newMajor);

   virtual int    getYearOfGraduation() const;
   virtual void   setYearOfGraduation(int year);

   virtual void   addCourse(string courseCode);
   virtual void   dropCourse(string courseCode);

   virtual double getGPA() const;
}; // end Student
```

The `GradStudent` class has the basic `Student` class data, and adds an enumerated type to indicate whether the graduate student is pursing a master or doctorate degree.

```
/** GradStudent.h */

enum ProgramType {MASTERS_NON_THESIS,  MASTERS_THESIS,  DOCTORATE};

class GradStudent: public Student
{
private:
   ProgramType program;

public:
   GradStudent();
   virtual ProgramType getProgram() const;
   virtual void setProgram(ProgramType newProgram);
}; // end GradStudent
```

10a We have used the class `Person`, as defined in Exercise 9, as the base class for `Employee`.

```
/** @file Employee.h */

#ifndef _EMPLOYEE
#define _EMPLOYEE

#include "Person.h"
#include <string>
using namespace std;

class Employee : public Person
{
private:
   string address;
   string employeeNumber;
   string department;

public:
   Employee(string eName, int eAge, char eGender, string eAddress, string eNum,
            string eDept);
   string getAddress() const;
   string getEmployeeNumber() const;
   string getDepartment() const;
   virtual string toString() const;
   virtual double getPay() const = 0;
}; // end Employee
#endif

/** Implementation file Employee.cpp */

#include "Employee.h"

Employee::Employee(string eName, int eAge, char eGender, string eAddress,
                   string eNum, string eDept)
{
   setName(eName);
   setAge(eAge);
   setGender(eGender);
   address = eAddress;
   employeeNumber = eNum;
   department = eDept;
}  // end constructor

string Employee::getAddress() const
{
   return address;
} // end getAddress

string Employee::getEmployeeNumber() const
{
   return employeeNumber;
} // end getEmployeeNumber

string Employee::getDepartment() const
{
   return department;
} // end getDepartment

string Employee::toString() const
{
   string result = "Name: " + getName() + "; Age: " + to_string(getAge()) +
                   "; Gender: " + getGender() + "\n";
```

```
      result = result + "Employee Number: " + employeeNumber + "; Department: " +
                        department + "\n";
      result = result + address + "\n";

      return result;
   }  // end toString
```

10b

```
   /** HourlyEmployee.h */
   #ifndef _HOURLY_EMPLOYEE
   #define _HOURLY_EMPLOYEE

   #include "Employee.h"

   class HourlyEmployee : public Employee
   {
   private:
      double payRate;  // Hourly wage
      double hours;    // Hours worked in a week

   public:
      HourlyEmployee(string eName, int eAge, char eGender, string eAddress, string eNum,
                     string eDept, double hourlyRate, double hoursWorked);
      virtual double getHourlyPayRate() const;
      virtual double getHoursWorked() const;
      virtual double getPay() const;
      virtual string toString() const;
   };  // end HourlyEmployee
   #endif

   /** Implementation file HourlyEmployee.cpp */

   #include "HourlyEmployee.h"

   HourlyEmployee::HourlyEmployee(string eName, int eAge, char eGender,
                                  string eAddress, string eNum, string eDept,
                                  double hourlyRate, double hoursWorked) :
                                  Employee(eName, eAge, eGender, eAddress, eNum, eDept)
   {
      payRate = hourlyRate;
      hours = hoursWorked;
   }  // end constructor

   double HourlyEmployee::getHourlyPayRate() const
   {
      return payRate;
   }  // end getHourlyPayRate

   double HourlyEmployee::getHoursWorked() const
   {
      return hours;
   }  // end getHoursWorked

   double HourlyEmployee::getPay() const
   {
      return payRate * hours;
   }  // end getPay

   string HourlyEmployee::toString() const
   {
      string result = Employee::toString() + "\n";
      result = result + "Hourly Pay: " + to_string(payRate) + "; Hours Worked: " +
       to_string(hours) + "\n";
      result = result + "Pay: " + to_string(getPay()) + "\n";

      return result;
   }  // end toString
```

10c

```
   /** @file NonHourlyEmployee.h */

   #ifndef _NONHOURLY_EMPLOYEE
   #define _NONHOURLY_EMPLOYEE

   #include "Employee.h"

   class NonHourlyEmployee : public Employee
   {
   private:
      double salary;

   public:
      NonHourlyEmployee(string eName, int eAge, char eGender,
                        string eAddress, string eNum, string eDept, double eSalary);
      virtual double getPay() const;
      virtual string toString() const;
   };  // end NonHourlyEmployee
   #endif

   /** Implementation file NonHourlyEmployee.cpp */

   #include "NonHourlyEmployee.h"

   NonHourlyEmployee::NonHourlyEmployee(string eName, int eAge, char eGender,
                                        string eAddress, string eNum, string eDept,
                                        double eSalary) :
                                   Employee(eName, eAge, eGender, eAddress, eNum, eDept)
   {
      salary = eSalary;
   }  // end constructor

   double NonHourlyEmployee::getPay() const
   {
      return salary;
   }  // end getPay

   string NonHourlyEmployee::toString() const
   {
      string result = Employee::toString() + "\n";
      result = result +  "Pay = " + to_string(salary) + "\n";
      return result;
   }  // end toString
```

11

Figure 12-5: The ADT list.

`insert`: The array-based insertion must shift array entries to position the new entry correctly. The link-based insertion must traverse the chain to find the correct position for a new node. Both of these operations are O(*n*).

`remove`: The array-based removal must shift array entries to remove the correct entry without leaving a gap in the array. The link-based removal must traverse the chain to find the correct node to delete. Both of these operations are O(*n*).

`getEntry`: The array-based retrieval can access the correct array element directly without a search, so it is O(1). The link-based retrieval must traverse the chain to find the correct node to access, so it is O(*n*).

`setEntry`: This operation has the same time requirements as `getEntry`.

`clear`: The array-based version simply sets `itemCount` to zero, so it is O(1). The link-based version deallocates each node in the chain, so it is O(*n*).

`getLength` and `isEmpty`: Both implementations of these methods involve one operation that is independent of *n*. They are O(1) operations.

Figure 12-6: The ADT sorted list.

`getPosition`: This method calls the list's `getEntry` method up to *n* times. Since `getEntry` is O(1) for the array-based list and is O(*n*) for the link-based list, `getPosition` is, respectively, O(*n*) and O(n^2).

`insertSorted`: This method first calls `getPosition` to position the new entry and then calls the list's `insert` method. For the array-based list, both `getPosition` and `insert` are O(*n*) operations. Since these operations occur sequentially, `insertSorted` is O(*n*). For the link-based list, `getPosition` is O(n^2), and `insert` is O(*n*). Therefore, `insertSorted` is O(n^2).

`removeSorted`: This method first calls `getPosition` to locate the entry and then calls the list's `remove` method. For the array-based list, both `getPosition` and `remove` are O(*n*) operations. Since these operations occur sequentially, `removeSorted` is O(*n*). For the link-based list, `getPosition` is O(n^2), and `remove` is O(*n*). Therefore, `insertSorted` is O(n^2).

The remaining operations—`getEntry`, `remove`, `clear`, `getLength`, and `isEmpty`—each call the corresponding list method, and so they have the same time requirements as they do for the list.

12

```
 // Insert array entries into sorted list
SortedListInterface<int>* listPtr = new LinkedSortedList<int>();
for (int i = 0; i < numberOfItems; i++)
   listPtr->insertSorted(data[i]);

// Move sorted data from sorted list into the array
for (int i = 0; i < numberOfItems; i++)
   data[i] = listPtr->getEntry(i + 1);
```

13

```
+merge(list1: SortedList, list2: SortedList): SortedList

   mergedLists = a new empty sorted list
   count1 = length of list1
   for (pos = 1; pos <= count1; pos++)
      mergedLists.insertSorted(list1.getEntry(pos))

   count2 = length of list2
   for (pos = 1; pos <= count2; pos++)
      mergedLists.insertSorted(list2.getEntry(pos))

   return mergedLists;
```

Here is a C++ function that merges two sorted lists of integers:

```
SortedListInterface<int>* merge(SortedListInterface<int>* list1Ptr,
                                SortedListInterface<int>* list2Ptr)
{
   SortedListInterface<int>* mergedListsPtr = new LinkedSortedList<int>();
   int count1 = list1Ptr->getLength();
   for (int pos = 1; pos <= count1; pos++)
      mergedListsPtr->insertSorted(list1Ptr->getEntry(pos));

   int count2 = list2Ptr->getLength();
   for (int pos = 1; pos <= count2; pos++)
      mergedListsPtr->insertSorted(list2Ptr->getEntry(pos));

   return mergedListsPtr;
}  // end merge
```

Chapter 13 Queues and Priority Queues

1 The `while` loop needs only to examine one half of the contents of the stack and the queue. The first half of the string is removed from the queue one character at a time and compared with the second half of the string as it is removed from the stack. If these string segments match, there is no need to continue the loop.

2
```
inLanguage(inputString: string): boolean

   aQueue = a new empty queue
   aStack = a new empty stack
   inputString = a string
   index = 0

   if (the length of inputString == 0)
      return false                      // Empty string not in L

   // Save the first half of the string
   while ((index < length of inputString) and (inputString[index] != '$'))
   {
      aQueue.enqueue(inputString[index])
      index++
   }
   // inputString[index] == '$' or index > length of inputString

   if (index > length of inputString)
      return false                      // inputString does not contain $

   // Save the second half of the string
   index++  // Advance ahead of $
   while (index < the length of inputString)
   {
      aStack.push(inputString[index])
      index++
   }

   // Match the first half of the string with the second half of the string
   while (!aQueue.isEmpty() and !aStack.isEmpty())
   {
      qItem = aQueue.peekFront()
      aQueue.dequeue()
      sItem = aStack.peek()
      aStack.pop()
      if (qItem != sItem)
         return false
   }

   if (aQueue.isEmpty() and aStack.isEmpty())
      return true
   else
      return false
```

3 4 4 4

4

```
PE = a new empty queue       // Will hold the postfix expression
aStack = a new empty stack
for (each character ch in the infix expression)
{
   switch (ch)
   {
      case operand:
         PE.enqueue(ch)
         break
      case '(':
         aStack.push(ch)
         break
      case operator:
         while ((aStack is not empty) and (aStack.peek() != '(')
                and (precedence(ch()) <= aStack.peek() ))
         {
            PE.enqueue(aStack.peek())
            aStack.pop()
         }
         aStack.push(ch)
         break
      case ')':
         // Pop down to the matching open parenthesis
         while (aStack.peek() != '(')
         {
            PE.enqueue(aStack.peek())
            aStack.pop()
         }
         aStack.pop()  // Remove open parenthesis
         break
   }
}
// Enqueue the operators remaining on the stack
while (!aStack.isEmpty())
{
   PE.enqueue(aStack.peek())
   aStack.pop()
}
```

5

```
string getLast(QueueInterface<string>* queuePtr)
{
   QueueInterface<string>* tempQueuePtr = new ArrayQueue<string>();
   string lastElement;

   // Locate last entry in queue while saving entries before it
   while (!queuePtr->isEmpty())
   {
      lastElement = queuePtr->peekFront();
      queuePtr->dequeue();
      tempQueuePtr->enqueue(lastElement);
   }  // end while

   // Restore original queue
   while (!tempQueuePtr->isEmpty())
   {
      queuePtr->enqueue(tempQueuePtr->peekFront());
      tempQueuePtr->dequeue();
   }  // end while
   return lastElement;
}  // end getLast
```

6 See Exercise 7 in Chapter 14.

7 We add the declaration
`virtual void display() = 0;`
to `QueueInterface`, add the declaration
`void display();`
to the class declaration in the header file, and add the following method definition to the implementation file. Since we are required to use only ADT queue operations, you can add this method to any class of queues after you change `ArrayQueue` to the name of the desired class of queues. Although declaring display as a const method would be ideal, this implementation requires us to remove items from the queue.

```
template<class ItemType>
void ArrayQueue<ItemType>::display()
{
   ItemType item;
   ArrayQueue<ItemType>* tempQueuePtr = new ArrayQueue<ItemType>();

   // Display contents of queue and save them in another queue
   while (!this->isEmpty())
   {
      item = this->peekFront();
      this->dequeue();
      tempQueuePtr->enqueue(item);
      cout << item << " ";    // Display items on same line
   }  // end while
   cout << endl;              // Terminate output with an end of line

   // Restore original queue
   while (!tempQueuePtr->isEmpty())
   {
      this->enqueue(tempQueuePtr->peekFront());
      tempQueuePtr->dequeue();
   }  // end while
}  // end display
```

8

```
/** @file PriorityQueueInterface.h */

#ifndef _PRIORITY_QUEUE_INTERFACE
#define _PRIORITY_QUEUE_INTERFACE

template<class ItemType>
class PriorityQueueInterface
{
public:
   /** Sees whether this priority queue is empty.
    @return  True if the priority queue is empty, or false if not. */
   virtual bool isEmpty() const = 0;

   /** Adds a new entry to this priority queue.
    @post  If the operation was successful, newEntry is in the
       priority queue.
    @param newEntry  The object to be added as a new entry.
    @return  True if the addition is successful or false if not. */
   virtual bool add(const ItemType& newEntry) = 0;
   /** Removes from this priority queue the entry having the
       highest priority.
    @post  If the operation was successful, the highest priority
       entry has been removed.
    @return  True if the removal is successful or false if not. */
   virtual bool remove() = 0;

   /** Returns the highest-priority entry in this priority queue.
    @pre  The priority queue is not empty.
    @post  The highest-priority entry has been returned, and the
       priority queue is unchanged.
    @return  The highest-priority entry. */
   virtual ItemType peek() const = 0;
}; // end PriorityQueueInterface
#endif
```

9

```
/** An interface for the ADT deque.
 @file DequeInterface.h */

#ifndef _DEQUE_INTERFACE
#define _DEQUE_INTERFACE

template<class ItemType>
class DequeInterface
{
   /** Sees whether this deque is empty.
    @pre  None.
    @return  True if the deque is empty, or false if not. */
   virtual bool isEmpty() const = 0;

   /** Adds a new entry to the front of this deque.
    @post If the operation was successful, newEntry is at the
       front of the deque.
    @param newEntry  The object to be added as a new entry.
    @return  True if the addition is successful or false if not. */
   virtual bool addToFront(const ItemType& newEntry) = 0;

   /** Adds a new entry to the back of this deque.
    @post If the operation was successful, newEntry is at the
       back of the deque.
    @param newEntry  The object to be added as a new entry.
    @return  True if the addition is successful or false if not. */
   virtual bool addToBack(const ItemType& newEntry) = 0;

   /** Removes the front of this deque.
    @post  If the operation was successful, the front of the deque
       has been removed.
    @return  True if the removal is successful or false if not. */
   virtual bool removeFront() = 0;

   /** Removes the back of this deque.
    @post  If the operation was successful, the front of the deque
       has been removed.
    @return  True if the removal is successful or false if not. */
   virtual bool removeBack() = 0;
/** Returns the front of this deque.
     @pre  The deque is not empty.
     @post The front of the deque has been returned, and the
        deque is unchanged.
     @return  The front of the deque. */
    virtual ItemType peekFront() const = 0;

/** Returns the back of this deque.
     @pre  The deque is not empty.
     @post The back of the deque has been returned, and the
        deque is unchanged.
     @return  The back of the deque. */
    virtual ItemType peekBack() const = 0;
 } // end DequeInterface
 #endif
```

10

```
// Reads a string of characters, assuming that the Backspace key is used
// to correct typing mistakes. Returns the corrected string.
readAndCorrect(): string

   result = a new empty string
   aDeque = a new empty deque

   // Read the line, correcting mistakes along the way
   while (not end of line)
   {
      Read a new character ch
      if (ch is not a '←')
         aDeque.addToBack(ch)
      else if (!aDeque.isEmpty())
         aDeque.removeBack()  // Remove mistake
      else
         Ignore the '←'       // Deque is empty; nothing to correct,
                              // so ignore backspace
   }

   // Form the string from the deque
   while (!aDeque.isEmpty)
   {
      ch = aDeque.peekFront()
      aDeque.removeFront()
      Concatenate ch to end of the string result
   }

   return result
```

---

11 Let E represent the event list (the priority queue) and Q represent the bank line (queue). For the book's pseudocode, E corresponds to `eventListPQueue`, and Q corresponds to `bankQueue`.

```
E: A|5|9              Simulation begins
Q:

E: A|7|5 D|14         Processing arrival at time 5
Q: 5|9

E: A|14|5 D|14        Processing arrival at time 7;  if there is a tie, arrivals are processed first
Q: 5|9 7|5

E: D|14 A|30|5        Processing arrival at time 14
Q: 5|9 7|5 14|5

E: D|19 A|30|5        Processing departure at time 14
Q: 7|5 14|5

E: D|24 A|30|5        Processing departure at time 19
Q: 14|5

E: A|30|5             Processing departure at time 24
Q: empty

E: A|32|5 D|35        Processing arrival at time 30
Q: 30|5

E: A|34|5 D|35        Processing arrival at time 32
Q: 30|5 32|5

E: D|35               Processing arrival at time 34
Q: 30|5 32|5 34|5

E: D|40               Processing departure at time 35
Q: 32|5 34|5

E: D|45               Processing departure at time 40
Q: 34|5

E: empty              Processing departure at time 45
Q: empty
```

12 The event list cannot be a queue since the `processArrival` operation places a departure event in the event list before it reads the input file for the next arrival event. Since this arrival time might precede the departure time of an event already in the event list, a queue would not position the arrival event correctly.

The event list could be an ADT list, although all of the functionality for sorting the events would have to be externally defined. The event list could also be an ADT sorted list, except that any policy regarding the ordering of arrival and. departure events that occur at the same time might be difficult to achieve by only the ADT operations.

13 We use the same notation as in the original algorithm. Thus, `topCity` is the front of the queue here.

Action	**Reason**	**Queue Contents (front to rear)**	**`topCity`**	**`nextCity`**
Add P	Add origin	P	P	R
Add R	Next unvisited adjacent city	PR	P	W
Add W	Next unvisited adjacent city	PRW	P	
Remove P	No unvisited adjacent city	RW	R	X
Add X	Next unvisited adjacent city	RWX	R	
Remove R	No unvisited adjacent city	WX	W	S
Add S	Next unvisited adjacent city	WXS	W	Y
Add Y	Next unvisited adjacent city	WXSY	W	
Remove W	No unvisited adjacent city	XSY	X	
Remove X	No unvisited adjacent city	SY	S	T
Add T	Next unvisited adjacent city	SYT	S	
Remove S	No unvisited adjacent city	YT	Y	Z
Add Z	Next unvisited adjacent city	YTZ	Y	
Remove Y	No unvisited adjacent city	TZ	T	
Remove T	No unvisited adjacent city	Z	Z	
Return true	Destination reached			

14a Base case: *`(Queue()).enqueue(item)).peekFront() = item`*

Recursive step:

```
if (!aQueue.isEmpty())
   (aQueue.enqueue(item)).peekFront() = aQueue.peekFront()
```

The `isEmpty` test prevents the invocation of `peekFront` by an empty queue.

For a stack, `peek` returns the last item that was added to a stack, that is the item added by the last `push`. In contrast, `peekFront` returns the first item that was added to a queue, that is the item added by the first `enqueue`. Thus, `peekFront` must be defined to "skip over" applications of `enqueue` until it finds the first one.

14b Yes, any queue can be represented as a sequence of only `enqueue` operations, as indicated by the following canonical form: `(...(((Queue()).enqueue()).enqueue()).enqueue())...)enqueue()).`

Chapter 14 Queue and Priority Queue Implementations

1

```
template<class ItemType>
LinkedQueue<ItemType>::LinkedQueue(const LinkedQueue& aQueue)
{
   Node<ItemType>* origChainPtr = aQueue->frontPtr;

   if (origChainPtr == nullptr)
   {
      frontPtr = nullptr;  // Original queue is empty
      backPtr = nullptr;
   }
   else
   {
      // Copy first node
      frontPtr = new Node<ItemType>();
      frontPtr->setItem(origChainPtr->getItem());

      // Advance original-chain pointer
      origChainPtr = origChainPtr->getNext();

      // Copy remaining nodes
      Node<ItemType>* newChainPtr = frontPtr;     // Points to last node in new chain
      while (origChainPtr != nullptr)
      {
         // Get next item from original chain
         ItemType nextItem = origChainPtr->getItem();

         // Create a new node containing the next item
         Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);

         // Link new node to end of new chain
         newChainPtr->setNext(newNodePtr);

         // Advance pointers
         newChainPtr = newChainPtr->getNext();
         origChainPtr = origChainPtr->getNext();
      }  // end while

      newChainPtr->setNext(nullptr);              // Flag end of chain
      backPtr = newChainPtr;
   }  // end if
}  // end copy constructor
```

2 This exercise is like Exercise 11 in Chapter 7.

3

```
template<class ItemType>
LinkedQueue<ItemType>::~LinkedQueue()
{
   if (frontPtr != nullptr)
   {
      Node<ItemType>* curPtr = frontPtr;  // Start with front end of queue
      while (curPtr != backPtr)
      {
         Node<ItemType>* tempPtr = curPtr->getNext();
         delete curPtr;
         curPtr = tempPtr;
         tempPtr = nullptr;
      }  // end while

      delete curPtr;                      // Delete last node
      curPtr = nullptr;
      frontPtr = nullptr;
      backPtr = nullptr;
   }  // end if
}  // end destructor
```

---

4 We add the declaration

```
virtual void display() const = 0;
```

to `QueueInterface`, add the declaration

```
void display() const;
```

to the class declaration in the header file, and add the following method definition to the implementation file:

```
template<class ItemType>
void LinkedQueue<ItemType>::display() const
{
   Node<ItemType>* curPtr = frontPtr;

   while (curPtr != backPtr)
   {
      cout << curPtr->getItem() << " "; // Display items on same line
      curPtr = curPtr->getNext();
   }  // end while
   cout << backPtr->getItem() << " "; // Display last item

   cout << endl;                      // Terminate output with an end of line
}  // end display
```

5a We add the declaration

```
virtual int getNumberOfElements() const = 0;
```

to `QueueInterface`, add the declaration

```
int getNumberOfElements() const;
```

to the class declaration in the header file, and add the following method definition to the implementation file:

```
template<class ItemType>
int ArrayQueue<ItemType>::getNumberOfElements() const
{
   return count;
}  // end getNumberOfElements
```

5b We make the same additions to the interface and header file as we did for part *a*. We will use a recursion in our definition, so we also declare the following private method in the class:

```
int count(Node<ItemType>* curPtr) const;
```

The method definitions are

```
template<class ItemType>
int LinkedQueue<ItemType>::count(Node<ItemType>* curPtr) const
{
   int result;
   if (curPtr == nullptr)
      result = 0;
   else if (curPtr == backPtr)
      result = 1;
   else result = 1 + count(curPtr->getNext());

   return result;
}  // end count

template<class ItemType>
int LinkedQueue<ItemType>::getNumberOfElements() const
{
   return count(frontPtr);
}  // end getNumberOfElements
```

---

6 If you examine the implementation that Programming Problem 4 asks you to write, you will find that the class methods require fewer statements due to the omission of the counter. The logic is a bit trickier than simply counting the queue entries. The difference in computing time required by the two implementations is likely insignificant.

7a The `enqueue` operation adds a new item to the end of the underlying list. For an array-based list, this addition is at the end of its underlying array. Thus, the operation is O(1) and has optimal efficiency comparable to that of the `ArrayQueue`. The `dequeue` operation, however, removes the first item in the underlying list, and thus removes the first entry in the array. This action forces each remaining entry of the array to shift one position toward the beginning of the array, taking O(*n*) time. The `dequeue` operation in `ArrayQueue`, in comparison, is O(1).

7b Since the `enqueue` operation adds a new item to the end of the underlying list, the list's `insert` operation must traverse the entire chain of nodes before it can append a new node to the chain. This takes O(*n*) time. In comparison, `LinkedQueue`'s `enqueue` operation is O(1). The `dequeue` operation removes the first item in the underlying list, and thus removes the first node from the list's chain of nodes. To do so, `LinkList`'s `remove` operation simply resets the head pointer in O(1) time. Note that `LinkedQueue`'s `dequeue` operation also is O(1).

---

8 Three methods are affected by this change: `enqueue`, `dequeue`, and `peekFront`. Their definitions follow:

```
template<class ItemType>
bool ListQueue<ItemType>::enqueue(const ItemType& newEntry)
{
// Back of queue is first in the list
   return listPtr->insert(1, newEntry);
}  // end enqueue

template<class ItemType>
bool ListQueue<ItemType>::dequeue()
{
// Front of queue is last in the list
  return listPtr->remove(listPtr->getLength());
}  // end dequeue

template<class ItemType>
ItemType ListQueue<ItemType>::peekFront() const throw(PrecondViolatedExcep)
{
   if (isEmpty())
      throw PrecondViolatedExcep("peekFront() called with empty queue.");

   // Queue is not empty; return front (last in the list)
   return listPtr->getEntry(listPtr->getLength());
}  // end peekFront
```

By adding a new entry at the beginning of the list, `enqueue` is less efficient than when it added the entry at the end of the list. This extra time is the result of having to shift existing entries in the list to make room for the new entry. The opposite is true for `dequeue`. Since it removes the last entry in the list, no shifting is necessary. Thus, `dequeue` is more efficient than when it removed the first entry in the list.

9

```
/** ADT priority queue: ADT sorted list implementation.
 @file SL_PriorityQueue.cpp */
#include "SL_PriorityQueue.h"  // Header file

template<class ItemType>
SL_PriorityQueue<ItemType>::SL_PriorityQueue()
{
   slistPtr = new LinkedSortedList<ItemType>();
}  // end default constructor

template<class ItemType>
SL_PriorityQueue<ItemType>::SL_PriorityQueue(const SL_PriorityQueue& pq) :
slistPtr(pq.slistPtr)
{
}  // end copy constructor

template<class ItemType>
SL_PriorityQueue<ItemType>::~SL_PriorityQueue()
{
}  // end destructor
```

```
template<class ItemType>
bool SL_PriorityQueue<ItemType>::isEmpty() const
{
   return slistPtr->isEmpty();
}  // end isEmpty

template<class ItemType>
bool SL_PriorityQueue<ItemType>::add(const ItemType& newEntry)
{
   slistPtr->insertSorted(newEntry);
   return true;
}  // end add

template<class ItemType>
bool SL_PriorityQueue<ItemType>::remove()
{
   // The highest priority item is at the end of the sorted list
   return slistPtr->remove(slistPtr->getLength());
}  // end remove

template<class ItemType>
ItemType SL_PriorityQueue<ItemType>::peek() const throw(PrecondViolatedExcep)
{
   if (isEmpty())
      throw PrecondViolatedExcep("peekFront() called with empty queue.");

   // Priority queue is not empty; return highest priority item;
   // it is at the end of the sorted list
   return slistPtr->getEntry(slistPtr->getLength());
}  // end peek
```

---

10 The only difference between this implementation and the one given in Exercise 9 is the definitions of the methods `remove` and `peek`. The highest priority item is at the beginning of the underlying sorted list. As a result, `remove` and `peek` do not require that the underlying sorted chain of nodes be traversed. The first node will contain the highest priority item. Thus, we have the following definitions:

```
template<class ItemType>
bool SL_PriorityQueue<ItemType>::remove()
{
   // The highest priority item is at the beginning of the sorted list
   return slistPtr->remove(1);
}  // end remove

template<class ItemType>
ItemType SL_PriorityQueue<ItemType>::peek() const throw(PrecondViolatedExcep)
{
   if (isEmpty())
      throw PrecondViolatedExcep("peekFront() called with empty queue.");

   // Priority queue is not empty; return highest priority item;
   // it is at the beginning of the sorted list
   return slistPtr->getEntry(1);
}  // end peek
```

The increase in the efficiency of remove and peek is because the

---

Chapter 15 Trees

---

1 **a.** 60; **b.** 60, 20, 40; **c.** 20, 70; 10, 40; 30, 50; **d.** 20 and 70, 10 and 40, 30 and 50; **e.** 40, 20, 60; **f.** 10, 40, 30, 50; **g.** 70, 10, 30, 50.

---

2 4.

3

`+isEmpty(): boolean`
Precondition: None.
Postcondition: Returns either true if the tree is empty or false if not. The tree is unchanged.

`+getHeight(): integer`
Precondition: None.
Postcondition: Returns the height of the tree. The tree is unchanged.

`+getNumberOfNodes(): integer`
Precondition: None.
Postcondition: Returns the number of nodes in the tree. The tree is unchanged.

`+getRootData(): ItemType`
Precondition: None.
Postcondition: Returns the data in the root. The tree is unchanged.

`+add(newEntry: ItemType): boolean`
Precondition: The current entries in the tree are distinct and differ from `newEntry`.
Postcondition: If the operation is successful, `newEntry` is inserted into the tree such that the properties of a binary search tree are maintained. Returns either true if the addition was successful or false if not.

`+remove(anEntry: ItemType): boolean`
Precondition: None.
Postcondition: If the operation is successful, `anEntry` is not in the tree. The properties of a binary search tree are maintained. Returns either true if the removal was successful or false if not.

`+clear(): void`
Precondition: None.
Postcondition: The tree is empty.

`+getEntry(anEntry: ItemType): ItemType`
Precondition: None.
Postcondition: If the tree contains an entry that matches `anEntry`, the entry in the tree is returned. Otherwise, a `NotFoundException` is thrown. The tree is unchanged.

`+contains(anEntry: ItemType): boolean`
Precondition: None.
Postcondition: Returns either true if the tree contains `anEntry` or false if it does not. The tree is unchanged.

`+preorderTraverse(visit(anEntry: ItemType): void): void`
Precondition: `visit` is a client function that performs an operation on an entry in the tree without changing the structure of the tree.
Postcondition: The nodes in the tree have been traversed in preorder, and the function `visit` has operated the entry in each node.

`+inorderTraverse(visit(anEntry: ItemType): void): void`
Precondition: `visit` is a client function that performs an operation on an entry in the tree without changing the structure of the tree.
Postcondition: The nodes in the tree have been traversed in inorder, and the function `visit` has operated the entry in each node.

`+postorderTraverse(visit(anEntry: ItemType): void): void`
Precondition: `visit` is a client function that performs an operation on an entry in the tree without changing the structure of the tree.
Postcondition: The nodes in the tree have been traversed in postorder, and the function `visit` has operated the entry in each node.

---

4 Preorder: MGDAHKLTRVUW
Inorder: ADHGKLMRUVTW
Postorder: AHDLKGUVRWTM

5a
```
+isLeaf(): boolean
// Returns true if the binary tree contains only one node. Otherwise, return false.
// The tree is unchanged.
```

5b Yes. The client could have `isLeaf` call `getNumberOfNodes`. If the tree contains one node, `isLeaf` would return true. Otherwise, it would return false.

---

6 60, 20, 10, 40, 30, 50, 70 is one of several possible orders. This particular order results from a preorder traversal of the tree.

7a The algorithm compares 30 with the following entries in the tree: 60, 20, 40, 30.

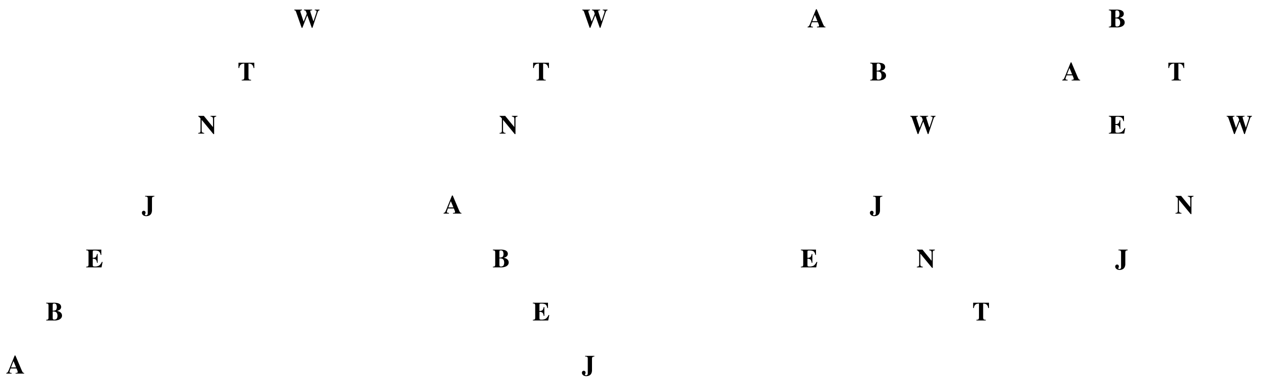
7b The algorithm compares 15 with the following entries in the tree: 60, 20, 10.

---

8 No. H should be in G's right subtree. U and V should be in T's right subtree.

---

9 **Insert Order:** **a.** W T N J E B A **b.** W T N A B E J **c.** A B W J N T E **d.** B T E A N W J



10a Node 6 must contain the inorder successor of the root's value, because it is the leftmost descendant of the root's right child.

10b 4, 2, 5, 8, 1, 6, 9, 3, 7.

11 **Insert order:** 80 65 75 45 35 25.

```
                              60

            20                                 70

  10                40               65                 80

              30            50                    75

         25       35      45
```

12 Without a convention for placing duplicates in the left or right subtree, searching for duplicates would require that both subtrees be searched.

13a The value of the whole tree is 23.

13b We could derive a class of minimax trees from the class `BinaryNodeTree` to give us the functionality we need for this problem. The class would need at least the following additional methods:

```
+getLeftSubtree(): MinimaxTree
+getRightSubtree(): MinimaxTree
+isRootMaxNode(): boolean
+isRootMinNode(): boolean
```

We could then define the following client function:

```
int getValue(MinimaxTree mTree)
{
   if (mTree.isEmpty())
      return 0;
   else if (mTree.getLeftSubtree().isEmpty() && mTree.getRightSubtree().isEmpty())
      return mTree.getRootData();
   else if (mTree.isRootMaxNode())
      return max(getValue(mTree.getLeftSubtree()),
                 getValue(mTree.getRightSubtree()));
   else // Root is min node
      return min(getValue(mTree.getLeftSubtree()),
                 getValue(mTree.getRightSubtree()));
}  // getValue
```

14 You cannot always create a binary search tree from a given a list of data items such that the order of the items in your list matches the preorder traversal of the tree. For example, you cannot form a binary search tree that gives a preorder traversal of 2 3 1. Moreover, a given preorder traversal uniquely specifies a binary search tree.

15 Let $t(n)$ denote the number of differently shaped, n-node binary trees. Then

$$t(0) = 0$$

$$t(n) = \sum_{i=0}^{n-1} (t(i) \times t(n-i-1))$$

The result is the same for binary search trees.

16

```
// Visits all nodes in a binary search tree whose values are in the range low to high.
rangeTraverse(aTree: BinarySearchTree, low: ItemType, high: ItemType,
              visit(anEntry: ItemType): void): void

   if (!aTree.isEmpty())
   {
      if (low < aTree.getRootData())
         rangeTraverse(Left subtree of aTree's root, low, high, visit)
      else if ((low <= aTree.getRootData()) &&(aTree.getRootData() <= high))
         visit(aTree.getRootData())
      else if (high > aTree.getRootData())
         rangeTraverse(Right subtree of aTree's root, low, high, visit)
   }
```

---

17 The proof given here uses Axiom E-2, as stated in Appendix E, but with a slight change in notation:
The inductive step of Axiom E-2 is
If P(0), P(1), . . . , P(k) are true for any $k \geq 0$, then P($k + 1$) is true.
However, you can also write this step as
If P(0), P(1), . . . , P($k - 1$) are true for any $k \geq 0$, then P(k) is true.
or after changing k to h as
If P(0), P(1), . . . , P($h - 1$) are true for any $h \geq 0$, then P(h) is true.
The latter form is equivalent to
If P(k) is true for all k such the $0 \leq k < h$, then P(h) is true.
We use this last form of the inductive step in this proof.

PROVE: A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes.
PROOF: The proof is by induction on h.
Basis. When $h = 0$, the full binary tree is empty, and it contains $0 = 2^0 - 1$ nodes.
Inductive hypothesis. Assume that a full binary tree of height k has $2^k - 1$ nodes when $0 \leq k < h$.
Inductive conclusion. We must show that a full binary tree of height h has $2^h - 1$ nodes.
If T is a full binary tree of height $h > 0$, its form is:

r

T_L T_R

where T_L and T_R are full binary trees of height $h - 1$. By the inductive hypothesis, T_L and T_R each have $2^{h-1} - 1$ nodes.
Thus, the number of nodes in T is

1 (for the root) + (number of nodes in T_L) + (number of nodes in T_R) =
$1 + (2^{h-1} - 1) + (2^{h-1} - 1) =$
$1 + 2 \times (2^{h-1} - 1) =$
$2^h - 1$
which is what we needed to show. *(End of proof.)*

---

18 **PROVE:** The maximum number of nodes in a binary tree of height h can have is $2^h - 1$.
PROOF: The proof is by induction on h.
Basis. When $h = 0$, the binary tree is empty, and it contains $0 = 2^0 - 1$ nodes.
Inductive hypothesis. Assume that a binary tree of height k has at most $2^k - 1$ nodes, when $0 \le k < h$.
Inductive conclusion. We must show that a binary tree of height h has at most $2^h - 1$ nodes.
If T is a binary tree of height $h > 0$, its form is:

r

T_L T_R

where T_L and T_R are binary trees each of height at most $h - 1$. By the inductive hypothesis, T_L and T_R each have at most $2^{h-1} - 1$ nodes each. Thus, the maximum number of nodes in T is

1 (for the root) + (maximum number of nodes in T_L) + (maximum number of nodes in T_R) =
$1 + (2^{h-1} - 1) + (2^{h-1} - 1) =$
$1 + 2 \times (2^{h-1} - 1) =$
$2^h - 1$
which is what we needed to show. *(End of proof.)*

---

19 The maximum number of nodes that can exist at level n of a binary tree is 2^{n-1}. We can prove this statement by an induction on n.
Basis. When $n = 1$, only one node—the root—can be at that level. Thus, the statement is true since $2^{1-1} = 2^0 = 1$.
Inductive hypothesis. Assume that the maximum number of nodes on level k is 2^{k-1}, when $1 \le k < n$.
Inductive conclusion. We must show that the maximum number of nodes at level n is 2^{n-1}.
By the inductive hypothesis, the maximum number of nodes on level $n - 1$ is 2^{n-2}. Each node on level $n - 1$ can have at most two children. Therefore, the maximum number of nodes at level n is $2 \times 2^{n-2} = 2^{n-1}$, which is what we want to show. *(End of proof.)*

19a Part 1 of the formal definition of a complete binary tree that appears on page 432, can be rewritten as
1. All levels k, such that $1 \le k < h - 1$, have 2^{k-1} nodes

---

19b The closed form of the given formula is $2^h - 1$. We can prove this closed form by an induction on h.
Basis. When $h = 1$, the value of the summation is 1 and the value of the closed form is $2^1 - 1 = 1$.
Inductive hypothesis. Assume that the value of the summation for $i = 1, \ldots, h$ is $2^h - 1$.
Inductive conclusion. We must show that the value of the summation for $i = 1, \ldots, h + 1$ is $2^{h+1} - 1$.
$\sum_{i=1}^{h+1} 2^{i-1} = \sum_{i=1}^{h} 2^{i-1} + 2^h$
$= (2^h - 1) + 2^h$ by the inductive hypothesis
$= 2^{h+1} - 1$
which is what we want to show. *(End of proof.)*
This number is the number of nodes in a full binary tree. Recall that a binary tree is full if all of its levels have the maximum number of nodes.

---

20 **PROVE:** A binary tree with n nodes has exactly $n + 1$ empty subtrees.
PROOF: The proof is by induction on n.
Basis. A binary tree with one node has two empty subtrees, so the statement is true for $n = 1$.
Inductive hypothesis. Assume that a binary tree with $n - 1$ nodes has exactly n empty subtrees.
Inductive conclusion. We must show that a binary tree with n nodes has exactly $n + 1$ empty subtrees.
If we add a node to the binary tree with $n - 1$ nodes, we will replace one of its n empty subtrees with a node, but that node will have two empty subtrees. Therefore, the resulting tree will have n nodes and $n - 1 + 2 = n + 1$ empty subtrees, which is what we want to show. *(End of proof.)*

---

21 **PROVE:** A strictly binary tree with n leaves has $2n - 1$ nodes.
PROOF: The proof is by induction on n.
Basis. A strictly binary tree with one leaf has one node. Thus, the statement is true since $2 \times 1 - 1 = 1$.
Inductive hypothesis. Assume that a strictly binary tree with $n - 1$ leaves has $2(n - 1) - 1 = 2n - 3$ nodes.
Inductive conclusion. We must show that a strictly binary tree with n leaves has $2n - 1$ nodes.
When we add a node to a binary tree, we add it as a child of an existing leaf, thus changing that leaf to an interior node with one child. Therefore, we cannot add only one node to a strictly binary tree with $n - 1$ leaves and preserve the strict binary tree property. If instead we add another node, it will be a leaf and its parent will still be an interior node, but it will have two children. Therefore, we have added n leaves and $(2n - 3) + 2 = 2n - 1$ nodes, which is what we want to show. *(End of proof.)*

22a Algorithm 1: Jane, Tom, Wendy, Nancy, Bob, Ellen, Alan
Algorithm 2: Jane, Bob, Tom, Alan, Ellen, Nancy, Wendy

22b Use a stack for algorithm 1 and a queue for algorithm 2. The data item for the stack or queue should be a pointer to the node being added. Each pointer, when it is retrieved from the stack or queue, will indicate the correct tree node. Storing the entire node in the container is not necessary, since the traversal does not change the tree.

---

23 You should traverse the tree in preorder traversal. As a result, the contents of the file would be

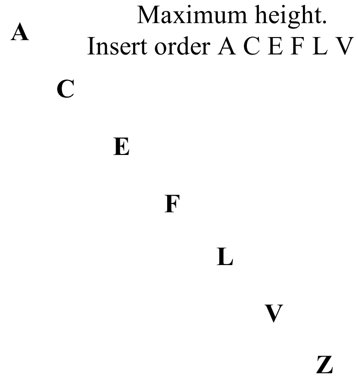
60 20 10 40 30 50 70

Chapter 16 Tree Implementations

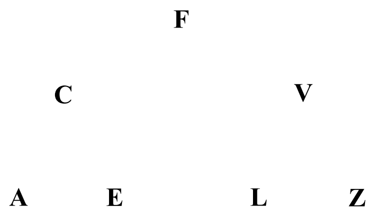
1a Node 6 must contain the inorder successor of the root's value, because it is the leftmost descendant of the root's right child.

1b Node 8 must contain the inorder predecessor of the root's value, because it is the rightmost descendant of the root's left child.

2

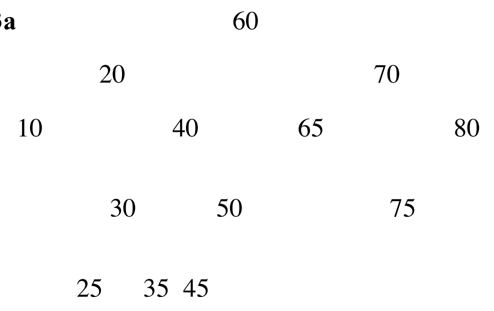


Maximum height.
Insert order A C E F L V Z

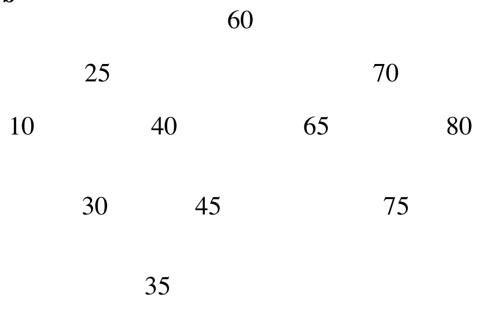


Minimum height.
Insert order F C V A E L Z

3a

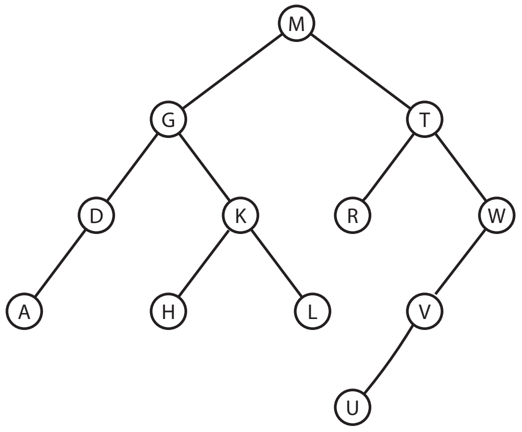


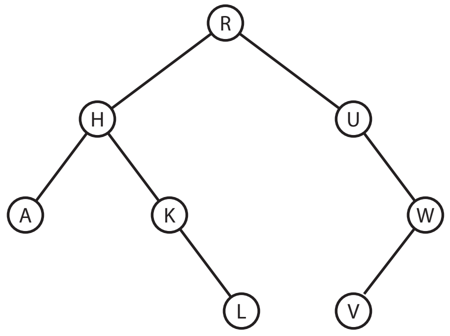
3b



---

4 The tree in Figure 15-19 is not a binary search tree, so we have modified it. The new binary search tree before any removals is on the left, and the tree after removing M, D, G, and T is on the right:





---

5 Removing an item from a leaf of a binary search tree and then inserting it back into the tree does not alter the shape of the tree. Removing and inserting an item from a node that has one or two children will alter the shape of a binary search tree. For example, consider the binary search tree in Exercise 3a. After removing 50 and 20, we get the tree shown in the answer to Exercise 3b. If we insert 20 and 50 back into the binary search tree, we get the following tree. Its shape differs from the one before the values were removed from the tree.

```
                  60

           25                      70

       10        40         65          80


         20   30     45          75


                 35      50
```

7a Array-based implementation of the binary search tree in Figure 15-14a of the text.

Index	0	1	2	3	4	5	6	7	8
`item`	Jane	Bob	Nancy	Alan	Elisa	Tom	Wendy		
`leftChild`	1	3	-1	-1	-1	-1	-1		
`rightChild`	2	4	5	-1	-1	6	-1	8	-1

`root` contains 0 and `free` contains 7.

7b **`bst.add("Doug");`**

Index	0	1	2	3	4	5	6	7	8
`item`	Jane	Bob	Nancy	Alan	Elisa	Tom	Wendy	Doug	
`leftChild`	1	3	-1	-1	7	-1	-1	-1	
`rightChild`	2	4	5	-1	-1	6	-1	-1	-1

`root` contains 0 and `free` contains 8.

`bst.remove("Nancy");`

Index	0	1	2	3	4	5	6	7	8
`item`	Jane	Bob		Alan	Elisa	Tom	Wendy	Doug	
`leftChild`	1	3		-1	7	-1	-1	-1	
`rightChild`	5	4	8	-1	-1	6	-1	-1	-1

`root` contains 0 and `free` contains 2.

`bst.remove("Bob");`

Index	0	1	2	3	4	5	6	7	8
`item`	Jane	Doug		Alan	Elisa	Tom	Wendy		
`leftChild`	7	3		-1	-1	-1	-1		
`rightChild`	5	4	8	-1	-1	6	-1	2	-1

`root` contains 0 and `free` contains 7.

`bst.add("Sarah");`

Index	0	1	2	3	4	5	6	7	8
`item`	Jane	Doug		Alan	Elisa	Tom	Wendy	Sarah	
`leftChild`	7	3		-1	-1	7	-1	-1	
`rightChild`	5	4	8	-1	-1	6	-1	-1	-1

`root` contains 0 and `free` contains 2.

7c

Index	0	1	2	3	4	5	6	7	8
`item`	Tom	Jane	Wendy	Bob	Nancy	Alan	Elisa		
`leftChild`	1	3	-1	5	-1	-1	-1		
`rightChild`	2	4	-1	6	-1	-1	-1	8	-1

`root` contains 0 and `free` contains 7.

`bst.add("Doug");`

Index	0	1	2	3	4	5	6	7	8
`item`	Tom	Jane	Wendy	Bob	Nancy	Alan	Elisa	Doug	
`leftChild`	1	3	-1	5	-1	-1	7	-1	-1
`rightChild`	2	4	-1	6	-1	-1	-1	-1	-1

`root` contains 0 and `free` contains 8.

`bst.remove("Nancy");`

Index	0	1	2	3	4	5	6	7	8
`item`	Tom	Jane	Wendy	Bob		Alan	Elisa	Doug	
`leftChild`	1	3	-1	5		-1	7	-1	
`rightChild`	2	-1	-1	6	8	-1	-1	-1	-1

`root` contains 0 and `free` contains 4.

`bst.remove("Bob");`

Index	0	1	2	3	4	5	6	7	8
`item`	Tom	Jane	Wendy	Doug		Alan	Elisa		
`leftChild`	1	3	-1	5		-1	-1		
`rightChild`	2	-1	-1	6	8	-1	-1	4	-1

`root` contains 0 and `free` contains 7.

`bst.add("Sarah");`

Index	0	1	2	3	4	5	6	7	8
`item`	Tom	Jane	Wendy	Doug		Alan	Elisa	Sarah	
`leftChild`	1	3	-1	5		-1	-1	-1	
`rightChild`	2	7	-1	6	8	-1	-1	-1	-1

`root` contains 0 and `free` contains 4.

7d
```
// Performs an inorder traversal a an array-based binary tree.
// Assumes that that array is named tree.
inOrderTraversal(visit(anEntry: ItemType): void): void

   if (tree[root].leftChild != -1)
   {
      oldRoot = root
      root = tree[root].leftChild
      inOrderTraversal(visit)
      root = oldRoot
   }

   visit(tree[root])

   if (tree[root].rightChild != -1)
   {  oldRoot = root
      root = tree[root].rightChild
      inOrderTraversal(visit)
      root = oldRoot
   }
```

8a Without a convention for placing duplicates in the left or right subtree, searching for duplicates would require that both subtrees be searched.

8b If duplicates are placed in the right subtree, for example, a value removed from an internal node must be replaced by its inorder successor. If duplicates are placed in the left subtree, however, a value removed from an internal node must be replaced by its inorder predecessor.

---

9 The succession of stack contents for each step of the inorder traversal of the tree in Figure 16-4 follows:

```
                              nullptr                       nullptr
                    >10       >10       >10                 >10       >10                 >40
          >20       >20       >20       >20                 >20       >20       >20       >20
>60       >60       >60       >60       >60       Visit 10  >60       >60       >60       Visit 20  >60
```

```
          nullptr                       nullptr                                           nullptr
>30       >30       >30                 >30       >30                           >50       >50       >50
>40       >40       >40                 >40       >40       >40                 >40       >40       >40
>20       >20       >20                 >20       >20       >20                 >20       >20       >20
>60       >60       >60       Visit 30  >60       >60       >60       Visit 40  >60       >60       >60
```

```
          nullptr
          >50       >50
          >40       >40       >40                                               nullptr
          >20       >20       >20       >20                           >70       >70       >70
Visit 50  >60       >60       >60       >60       >60       Visit 60  >60       >60       >60       Visit 70
```

```
nullptr
>70       >70
>60       >60       >60
```

10
```
template<class ItemType>
void BinaryNodeTree<ItemType>::iterativeInorderTraverse(void visit(ItemType&)) const
{
   // Initialize
   LinkedStack<BinaryNode<ItemType>*>* stackPtr =
                                       new LinkedStack<BinaryNode<ItemType>*>();
   BinaryNode<ItemType>* curPtr = rootPtr;
   bool done = false;

   while (!done)
   {
      if (curPtr != nullptr)
      {
         // Place pointer to node on stack before
         // traversing node's left subtree
         stackPtr->push(curPtr);

         // Traverse the left subtree
         curPtr = curPtr->getLeftChildPtr();
      }
      else if (!stackPtr->isEmpty())
      {  // Backtrack from the empty subtree and
         // visit the node at the top of the stack

         curPtr = stackPtr->peek();
         stackPtr->pop();
         ItemType theItem = curPtr->getItem();
         visit(theItem);
```

```
            // Traverse the right subtree
            // of the node just visited
            curPtr = curPtr->getRightChildPtr();
         }
         else // If the stack is empty, you are done
            done = true;
      } // end while
   } // end iteratively InorderTraverse
```

13

```
   template<class ItemType>
   bool BinarySearchTree<ItemType>::addIterative(const ItemType& newEntry)
   {
      BinaryNode<ItemType>* parent = nullptr;
      BinaryNode<ItemType>* curPtr = rootPtr;

      while (curPtr != nullptr) // Loop until a leaf is found
      {
         parent = curPtr;
         if (newEntry < curPtr->getItem())
            curPtr = curPtr->getLeftChildPtr();
         else
            curPtr = curPtr->getRightChildPtr();
      }  // end while

      curPtr = new BinaryNode<ItemType>(newEntry);

      // Parent points to the node that will be the parent of new node
      if (parent != nullptr)
      {
         if (newEntry < parent->getItem())
            parent->setLeftChildPtr(curPtr);
         else
            parent->setRightChildPtr(curPtr);
      }
      else  // insert into an empty tree
         rootPtr = curPtr;

      return true;
   }  // end addIterative
```

14

```
template<class ItemType>
void BinaryNodeTree<ItemType>::levelOrderTraverse(void visit(ItemType&)) const
{
   QueueInterface<BinaryNode<ItemType>*>* queuePtr =
                                        new ArrayQueue<BinaryNode<ItemType>*>();
   queuePtr->enqueue(rootPtr);

   while (!queuePtr->isEmpty())
   {
      // Remove and visit a node from queue front
      BinaryNode<ItemType>* nodePtr = queuePtr->peekFront();
      queuePtr->dequeue();
      ItemType item = nodePtr->getItem();
      visit(item);

      // Enqueue the left child, if any
      BinaryNode<ItemType>* childPtr = nodePtr->getLeftChildPtr();
      if (childPtr != nullptr)
         queuePtr->enqueue(childPtr);

      // Enqueue the right child, if any
      childPtr = nodePtr->getRightChildPtr();
      if (childPtr != nullptr)
         queuePtr->enqueue(childPtr);
   }  // end while
}  // end levelOrderTraverse
```

17a Assume that the nodes in the general tree are instances of the class `GeneralNode`. This class has the following data members:

`item`—A data item
`children`—An array of node pointers that point to the children of this node
`numberOfChildren`—An integer that indicates the number of children belonging to this node

The following pseudocode describes a recursive preorder traversal:

```
preorderTraverse(rootPtr: GeneralNode*, visit(entry: ItemType): void): void

   if (rootPtr != nullptr)
   {
      visit(rootPtr->getItem();
      for (i = 0; i < rootPtr->getNumberOfChildren(); i++)
      {
         cArray = rootPtr->getChildren();;
         preorderTraverse(cArray[i], visit);
      }
   }
```

Advantages: Access to the children is easy; direct access to each child is available.
Disadvantages: Limits the number of children to a fixed maximum. If this maximum number of children is much higher than the average number of children, memory is wasted. If the children have an order, additions to and removals from could require shifting the pointers in the array.

---

19 The following solutions use the functions `max` and `min` which return the maximum and minimum between two integers.

19a Recursive computation of the number of nodes in the tree.

```
int numNodes(BinaryTree aTree)
{  if (aTree.isEmpty())
      return 0;
   else
      return 1 + numNodes(aTree.leftSubtree())
               + numNodes(aTree.rightSubtree());
}  // end numNodes
```

19b Recursive computation of the height of a binary tree:

```
int height(BinaryTree aTree)
{
   if (aTree.isEmpty())
      return 0;
   else

   {
      int heightLeft=height(aTree.leftSubtree());
      int heightRight=height(aTree.rightSubtree());
      if (heightLeft  >= heightRight)
         return heightLeft+1;
      else
         return heightRight+1;
   }  // end if
}  // end height
```

19c Recursive computation of the maximum element of a binary tree:

```
#include <limits.h>
int maxElement(BinaryTree aTree)
{  int leftSubtreeMax, rightSubtreeMax;

   if (aTree.isEmpty())
      return INT_MIN;
   else
   { leftSubtreeMax  = maxElement(aTree.leftSubtree());
     rightSubtreeMax = maxElement(aTree.rightSubtree());

      if (leftSubtreeMax > aTree.rootData())
         return max(leftSubtreeMax, rightSubtreeMax);
      else
         return max(aTree.rootData(),rightSubtreeMax);
   } // end if
} // end maxElement
```

19d Recursive computation of the sum of all elements of a binary tree:

```
int sum(BinaryTree aTree)
{  if (aTree.isEmpty())
      return 0;
   else
      return sum(aTree.leftSubtree()) + sum(aTree.rightSubtree())
                 + aTree.rootData();
} // end sum
```

19e Recursive computation of the average of all elements of a binary tree. (The `count` routine is used to count the number of nodes in the tree.)

```
int count(BinaryTree aTree)
{  if (aTree.isEmpty())
      return 0;
   else
      return count(aTree.leftSubtree()) +
             count(aTree.rightSubtree()) + 1;
}  // end count

float average(BinaryTree aTree)
{  int numNodes;

   numNodes = count (aTree);
   if (numNodes == 0)
      return 0.0;
   else
      return sum(aTree)/(float) numNodes;
}  // end Average
```

19f Recursive item search in a (unordered) binary tree, returns a tree rooted at the node where the item is found:

```
BinaryTree find (BinaryTree aTree, int findItem)
{  BinaryTree tempTree;

   if (aTree.isEmpty())
   {  tempTree = aTree;
      return tempTree;
   }
   else if (aTree.rootData() == findItem)
      return aTree;
   else
   {  tempTree = find (aTree.leftSubtree(), findItem);
      if (tempTree.isEmpty())
         return (find (aTree.rightSubtree(), findItem));
      else
         return tempTree;
   } // end if
} // end find
```

19g Recursive ancestor relationship computation in a binary tree:

```
bool ancestor(BinaryTree aTree, int item1, int item2)
{  BinaryTree leftTemp, rightTemp;

   if (aTree.isEmpty())
      return false;
   else if (aTree.rootData() == item1)
   {  leftTemp  = find(aTree.leftSubtree(), item2);
      rightTemp = find(aTree.rightSubtree(), item2);
      return(leftTemp.isEmpty() || rightTemp.isEmpty());
   }
   else
      return (ancestor(aTree.leftSubtree(), item1, item2) ||
               ancestor (aTree.rightSubtree(), item1, item2));
}  // end ancestor
```

19h Recursive computation of highest full level in a binary tree:

```
int fullLevel (BinaryTree aTree)
{  if (aTree.isEmpty())
      return 0;
   else
      return min(fullLevel(aTree.leftSubtree()),
                 fullLevel(aTree.rightSubtree()))+1;
}  // end fullLevel
```

Chapter 17 Heaps

1a For exercises 1 and 2, we assume that the three operations have a cumulative effect on the heaps.

```
          4
        /   \
      7       9
    /   \    / \
   8    12  13  11
  /
10
```

1b

```
          4
        /   \
      5       9
    /   \    / \
   7    12  13  11
  / \
10   8
```

1c

```
          5
        /   \
      7       9
    /   \    / \
   8    12  13  11
  /
10
```

2a

```
         16
        /   \
      10     13
     /  \    / \
    3    7  8   9
```

2b

```
         16
        /   \
      14     13
     /  \    / \
   10    7  8   9
   /
  3
```

2c

```
         14
        /   \
      10     13
     /  \    / \
    3    7  8   9
```

3

```
          15
         /  \
       12    11
      / \   / \
     9   3 4   6
    /
   8
```

---

4 **PROVE:** The root of a maxheap contains the largest value in the tree.
PROOF: The proof is by induction on *n*.
Basis. If the heap contains only a single item, the assertion is obviously true.
Inductive hypothesis. Suppose that the assertion is true for all heaps with $n - 1$ or fewer items.
Inductive conclusion. Consider a heap, *H*, containing *n* items. By definition, the root of *H* is larger than either of its children, and each of its children are roots of a heap of size $n - 1$. Thus, the children of the root of *H* are the largest values in their subtrees by the inductive hypothesis. Since the root of *H* is larger than either of its children, the assertion is true for a heap of size *n*. *(End of proof.)*

---

5 The order in which you insert items into a heap does not affect its shape, but it does affect where the values are placed into the individual nodes. The following diagram provides an example:

```
        Insertion order: 10  9  3  5  2  6          Insertion order: 6  3  5  10  9  2
                   10                                          10


        9                   6                          9                   5


    5          2          3                        3         6           2
```

---

7 The order in which two items with the same priority value are inserted into a priority queue can affect the order in which they are removed, but this effect depends on the implementation of the priority queue. To remove duplicate priorities in first in, first out (FIFO) order, the order in which the items were inserted could be maintained as a secondary value that affects the comparison of priority values. Alternatively, each entry in the priority queue could be a queue of items having the same priority.

---

8 Changing the implementation of the maxheap to one for a minheap affects only the `add` and `heapRebuild` methods. In the `add` method, the item should trickle up if it is smaller than its parent. In `heapRebuild`, the item should trickle down if it is larger than one of its children. Only the comparison operators need be changed.

---

9 You can maintain an index to the minimum value in a maxheap by comparing the value of a newly inserted item with the current minimum value. If the value is smaller than the current minimum, you update the minimum value and store its index after the trickle up process has terminated. If the value is not smaller than the current minimum, you check to see whether the insertion changes the location of the current minimum due to a swap. During a removal, you check to see whether the index of the current minimum changes. For example, its position in the array will change when it currently is the last entry in the array.

---

10 If the value increases, make it trickle up. If the value decreases, make it trickle down.

---

11 `heapCreate` calls `heapRebuild(index)`, as `index` ranges from `itemCount` – 1 down to 0. When `itemCount / 2` < `index` < `itemCount`, `2 * index` > `itemCount`. This fact implies that the value at `index` is in a leaf and, therefore, the call to `heapRebuild` does nothing. Thus, you can shorten the loop by avoiding values of `index` that are greater than `itemCount / 2`.

12

The original order	5 1 2 8 6 10 3 9 4 7
After the heap is created	10 9 8 6 7 2 3 1 4 5 |
After swap 1 and `heapRebuild`:	9 7 8 6 5 2 3 1 4 | 10
After swap 2 and `heapRebuild`:	8 7 4 6 5 2 3 1 | 9 10
After swap 3 and `heapRebuild`:	7 6 4 1 5 2 3 | 8 9 10
After swap 4 and `heapRebuild`:	6 5 4 1 3 2 | 7 8 9 10
After swap 5 and `heapRebuild`:	5 3 4 1 2 | 6 7 8 9 10
After swap 6 and `heapRebuild`:	4 3 2 1 | 5 6 7 8 9 10
After swap 7 and `heapRebuild`:	3 1 2 | 4 5 6 7 8 9 10
After swap 8 and `heapRebuild`:	2 1 | 3 4 5 6 7 8 9 10
After swap 9 and `heapRebuild`:	1 | 2 3 4 5 6 7 8 9 10
The array is sorted.	1 2 3 4 5 6 7 8 9 10

13
```
typedef type-of-array-entry ItemType;

int getLeftChildIndex(int nodeIndex)
{
   return (2 * nodeIndex) + 1;
}  // end getLeftChildIndex

int getRightChildIndex(int nodeIndex)
{
   return (2 * nodeIndex) + 2;
}  // end getRightChildIndex

int getParentIndex(int nodeIndex)
{
   return (nodeIndex - 1) / 2;
}  // end getParentIndex

bool isLeaf(const int nodeIndex, int itemCount)
{
   return (getLeftChildIndex(nodeIndex) >= itemCount);
}  // end isLeaf

void heapRebuild(const int subTreeNodeIndex, ItemType items[], int itemCount)
{
   if (!isLeaf(subTreeNodeIndex, itemCount))
   {
      // Find larger child; a left child must exist; right child might not
      int leftChildIndex = getLeftChildIndex(subTreeNodeIndex);
      int largerChildIndex = leftChildIndex; // Assume left child is larger
      int rightChildIndex = getRightChildIndex(subTreeNodeIndex);

      // Check to see whether a right child exists
      if (rightChildIndex < itemCount)
      {
         // A right child exists; check whether it is larger
         if (items[rightChildIndex] > items[largerChildIndex])
            largerChildIndex = rightChildIndex; // Assumption was wrong
      }  // end if
      // If root value is smaller that the value in the larger child, swap values
      if (items[subTreeNodeIndex] < items[largerChildIndex])
      {
         swap(items[largerChildIndex], items[subTreeNodeIndex]);

         // Continue with the recursion at that child
         heapRebuild(largerChildIndex, items, itemCount);
      }  // end if
   }  // end if
}  // end heapRebuild

/** Sorts the items in an array into ascending order.
 @pre  None.
 @post  anArray is sorted into ascending order; n is unchanged.
 @param anArray  The given array.
 @param n  The size of theArray. */
void heapSort(ItemType anArray[], int n)
{
   // Build initial heap
   for (int index = n / 2; index >= 0; index--)
      heapRebuild(index, anArray, n);

   // The heap is anArray[0..n-1].
   // Assertion: anArray[0] is the largest entry in the array

   swap(anArray[0], anArray[n - 1]);

   int heapSize = n - 1;  // Heap region size decreases by 1
   while (heapSize > 1)
   {
      heapRebuild(0, anArray, heapSize);
      heapSize--;
      swap(anArray[0], anArray[heapSize]);
   }  // end while
}  // end heapSort
```

---

14 Simply change the first comparison of array values within `heapRebuild` from > to < and the second comparison from < to >. The result, after changing the names of variables, is

```
void heapRebuild(const int subTreeNodeIndex, ItemType items[], int itemCount)
{
   if (!isLeaf(subTreeNodeIndex, itemCount))
   {
      // Find smaller child; a left child must exist; right child might not
      int leftChildIndex = getLeftChildIndex(subTreeNodeIndex);
      int smallerChildIndex = leftChildIndex;    // Assume left child is smaller
      int rightChildIndex = getRightChildIndex(subTreeNodeIndex);

      // Check to see whether a right child exists
      if (rightChildIndex < itemCount)
      {
         // A right child exists; check whether it is smaller
         if (items[rightChildIndex] < items[smallerChildIndex])
            smallerChildIndex = rightChildIndex; // Assumption was wrong
      }  // end if

      // If root value is larger than the value in the smaller child, swap values
      if (items[subTreeNodeIndex] > items[smallerChildIndex])
      {
         swap(items[smallerChildIndex], items[subTreeNodeIndex]);
```

```
            // Continue with the recursion at that child
            heapRebuild(smallerChildIndex, items, itemCount);
         }  // end if
      }  // end if
   }  // end heapRebuild
```

Chapter 18 Dictionaries and Their Implementations

1
```
#include "Entry.h"

template <class KeyType, class ItemType>
Entry<KeyType, ItemType>::Entry()
{ }

template <class KeyType, class ItemType>
Entry<KeyType, ItemType>::Entry(ItemType newEntry, KeyType itemKey):
                          item(newEntry), searchKey(itemKey)
{ }

template <class KeyType, class ItemType>
ItemType Entry<KeyType, ItemType>::getItem() const
{
   return item;
}

template <class KeyType, class ItemType>
KeyType Entry<KeyType, ItemType>::getKey() const
{
   return searchKey;
}

template <class KeyType, class ItemType>
void Entry<KeyType, ItemType>::setItem(const ItemType& newEntry)
{
   item = newEntry;
}

template <class KeyType, class ItemType>
void Entry<KeyType, ItemType>::setKey(const KeyType& itemKey)
{
   searchKey = itemKey;
}

template <class KeyType, class ItemType>
bool Entry<KeyType, ItemType>::operator==(const Entry<KeyType,
                                          ItemType>& rightHandItem) const
{
   return (searchKey == rightHandItem.getKey());
}  // end operator==

template <class KeyType, class ItemType>
bool Entry<KeyType, ItemType>::operator>(const Entry<KeyType, ItemType>& rightHandItem)
const
{
   return (searchKey > rightHandItem.getKey());
}
```

2a

```
template <class KeyType, class ItemType>
bool ArrayDictionary<KeyType, ItemType>::replace(const KeyType& searchKey,
                                                 const ItemType& replacementItem)
{
   int index = findEntryIndex(0, itemCount - 1, searchKey);
   bool result = false;
   if (index >= 0)
   {
      items[index].setItem(replacementItem);
      result = true;
   }  // end if

   return result;
}  // end replace
```

2b The `replace` method will preserve the existing structure of the underlying binary search tree if the replaced item is in a leaf node.

---

3 A sorted array-based implementation that uses a binary search for its retrieval operation will take O(log *n*) time.
A sorted linked-based implementation will require a sequential search, so will take O(*n*) time.
An implementation that uses a binary search tree will take O(log *n*) time.
An implementation that uses hashing with a sufficiently large hash table and separate chaining will be the most time efficient. Since we would know the number of words to be stored, we can make the hash table as large as needed.

---

4 The answer to the previous exercise considers the retrievals. For insertions, we have the following.
A sorted array-based implementation performs an insertion in O(*n*) time.
A sorted linked-based implementation performs an insertion in in O(*n*) time.
An implementation that uses a binary search tree performs an insertion in O(log *n*) time.
An implementation that uses hashing with a sufficiently large hash table and separate chaining will be the most time efficient. Since we would know the initial number of words to be stored, we can make the hash table as large as needed. We should be able to accommodate the infrequent insertions.

---

5 Unless the symbol table grows unexpectedly large, a hashed dictionary is the best choice.

---

6 The intersection operation looks for entries in each dictionary that have the same search key. If none are present, the intersection is empty. So we only need to be concerned with the union operation. When this operation encounters two entries that have the same search key, it can take the following action:

- If the items in each entry are the same, the union should contain one of the two entries.
- If the items in each entry are different, the union could contain one entry that combines the two data items into one item having the common search key.

---

7 The dictionary should test for and disallow duplicate search keys in case the underlying data structure does not do that for you. Only the dictionary's `add` method is involved. In `ArrayDictionary`, you can modify `add` in one of two ways. The easiest way is to change its first statement to

```
bool ableToInsert = (itemCount < maxItems) && !contains(searchKey);
```

The disadvantage to this solution is that the method `contains` must search the dictionary. If the search is unsuccessful, the rest of the `add` method searches the dictionary again. A more time-efficient solution uses the original definition of `ableToInsert` and modifies the code after the `while` loop in `add`, as follows:

```
if (searchKey == items[index - 1].getKey())
   ableToInsert = false;
else
{
   // Insert new entry
   items[index] = Entry<KeyType, ItemType>(newItem, searchKey);
   itemCount++; // Increase count of entries
}  // end if
```

Other implementations would make similar tests. The unsorted implementations will require more time to disallow duplicates than the sorted implementations, because they would need to perform a sequential search.

8a If an entry has the same search key and item as an entry already in the dictionary—that is, the two entries are identical—the best action is to prevent the insertion of the duplicate entry.

8b The insertion into a dictionary of an entry that has the same search key, but a different data item, as an entry already in the dictionary must be done in a way that facilitates the retrieval and removal of these entries. When a dictionary contains more than one entry having the same search key, retrieving or removing the correct entry becomes more involved. You could given the entries a secondary search key that is tested when the primary search keys match. For example, two people that have the same name could be distinguished by their addresses or telephone numbers. In the absence of secondary search keys, the retrieval and removal operations should involve all entries having the given key. The retrieval operation could return a structure such as an array or a list.

---

9 Let's consider a database of people organized by their names and identification numbers. One approach uses two dictionaries. One dictionary uses the name as the search key and the identification number as the data item; the other dictionary uses the identification number as the search key and the name as the data item. The data base then defines two insertion operations, two retrieval operations, and two removal operations. One operation in each pair uses the name as the search key, and the other operation uses the identification number. When an entry is removed, `removeByName` must remove the correct entry from each of the dictionaries. To do so, it finds the entry by name and then uses the corresponding identification number as a search key in the second dictionary. Analogous comments apply to `removeById`.

A second approach uses one modified dictionary. Each entry has two fields, each of which can serve as a search key, in addition to a data item. This data item could contain additional information, such as an address, or could be empty. The methods `removeByName` and `removeById` are each like the original remove method, but use the appropriate search key to locate the entry to be removed.

---

11 You could use a heap in an implementation of a dictionary, but it would be a poor choice. Searching a heap is not as efficient as searching a binary search tree, because you do not know which subtree will contain the desired entry. For example, consider the following heap. To search for 4, you see that it is less than the root, 15, and so it could be in either of the root's subtrees. If we check the left subtree first, we find that 4 is less than 5, implying that it could be in either of 5's subtrees. We check both of these subtrees, and learn that 4 is not present. Thus, we need to search the right subtree of the heap's root. Searching this heap for 4 is as inefficient as sequential search.

```
      15
     /  \
    5     9
   / \   / \
  3   2 6   4
```

Likewise, an inorder traversal of a heap would be quite inefficient.

12

```
add(searchKey: KeyType, newItem: ItemType): boolean

   i = getHashIndex(searchKey)

   if (hashTable[i] is not empty)
   {
      do
         i = (i + 1) mod hashTableSize
      while (hashTable[i] is not empty && (i != getHashIndex(searchKey)))
   }

   table[i] = a new entry containing searchKey and newItem
   return true

remove(searchKey: KeyType): boolean

   removed = true
   i = getHashIndex(searchKey)

   if (hashTable[i].getKey() != searchKey)
   {
      do
        i = (i + 1) mod hashTableSize
      while ((hashTable[i].getKey() != searchKey) &&
             (i != getHashIndex(searchKey)))
   }

   if (table[i].getKey() == searchKey)
      delete table[i]
   else
      removed = false

   return removed
```

```
getItem(searchKey: KeyType): ItemType

   i = getHashIndex(searchKey)

   if (hashTable[i].getKey() != searchKey)
   {
      do
        i = (i + 1) mod hashTableSize
      while ((hashTable[i].getKey() != searchKey) &&
             (i != getHashIndex(searchKey)))
   }

   if (table[i].getKey() == searchKey)
      return table[i].getItem()
   else
      throw NotFoundException
```

13

```
remove(searchKey: KeyType): boolean

   itemFound = false

   // Compute the hashed index into the array
   itemHashIndex = getHashIndex(searchKey)
   if (hashTable[itemHashIndex] != nullptr)
   {
      // Special case - first node has target
      if (searchKey == hashTable[itemHashIndex]->getKey())
      {
         entryToRemovePtr = hashTable[itemHashIndex]
         hashTable[itemHashIndex] = hashTable[itemHashIndex]->getNext()
         delete entryToRemovePtr
         entryToRemovePtr = nullptr
         itemFound = true
      }
      else // Search the rest of the chain
      {
         prevPtr = hashTable[itemHashIndex]
         curPtr = prevPtr->getNext()
         while ((curPtr != nullptr) && !itemFound)
         {
            // Found item in chain so remove that node
            if (searchKey == curPtr->getKey())
            {
               prevPtr->setNext(curPtr->getNext())
               delete curPtr
               curPtr = nullptr
               itemFound = true
            }
            else // Look at next entry in chain
            {
               prevPtr = curPtr
               curPtr = curPtr->getNext()
            }
         }
      }
   }
   return itemFound
```

14a This hash function will probably not distribute the keys uniformly, since the table size is not prime. Collisions will certainly occur for any keys whose sums are a power of 2.

14b This hash function uses only 26 of a possible 2,048 table locations and does not distribute the keys uniformly among the 26 accessible positions.

14c Because the function produces random results, you cannot use it to locate a previously added entry.

14d This function requires too many computations, so is too inefficient to use.

Chapter 19 Balanced Search Trees

1a Binary search tree:

```
Insert 10    Insert 100    Insert 30    Insert 80     Insert 50
10           10            10           10            10
                  100          100          100           100
                           30           30            30
                                            80            80
                                                      50

Remove 10    Insert 60     Insert 70    Insert 40     Remove 80
    100           100          100          100          100
30           30            30           30            30
    80            80           80           80            50
50           50            50           50            40      60
                  60           60     40    60                    70
                                  70           70

Insert 90    Insert 20     Remove 30    Remove 70
    100           100          100          100
30           30            40           40
    50     20    50     20    50      20    50
 40    60      40   60            60            60
          70           70            70            90
             90           90            90
```

1b 2-3 tree:

```
Insert 10           Insert 100          Insert 30            Insert 80
    10                10|100                30                   30
                                        10      100          10     80|100

Insert 50           Remove 10           Insert 60            Insert 70
  30|80                 80                 50|80                50|80
10  50  100         30|50   100         30  60   100         30 60|70 100
```

```
        Insert 40                Remove 80               Insert 90               Insert 20

          50|80                    50|70                   50|70                     50

   30|40  60|70  100        30|40   60   100       30|40  60  90|100           30       70

                                                                           20    40 60  90|100


        Remove 30                Remove 70

           50|70                   50|90

   30|40   60  90|100       20|40  60   100
```

1c 2-3-4 tree:

```
        Insert 10                Insert 100              Insert 30               Insert 80

           10                      10|100                10|30|100                   30

                                                                                10      80|100

        Insert 50                Remove 10               Insert 60               Insert 70

           30                        50                      50                     50|80

        10   50|80|100          30     80|100          30   60|80|100          30   60|70   100

        Insert 40                Remove 80               Insert 90               Insert 20

          50|80                    50|70                   50|70                     50|70

    30|40 60|70  100        30|40 60  100          30|40 60 90|100         20|30|40 60 90|100

        Remove 30                Remove 70

          50|70                    50|90

     20|40 60 90|100        20|40 60  100
```

1d A red-black tree:

```
   Insert 10           Insert 100          Insert 30           Insert 80
      10                  10                  30                  30
                              100          10     100          10     100
                                                                    80

   Insert 50           Remove 10           Insert 60           Insert 70
      30                  50                  50                  50
   10     80           30     80           30     80           30     80
       50   100                 100            60   100            60   100
                                                                      70

   Insert 40           Remove 80           Insert 90           Insert 20
      50                  50                  50                  50
   30     80           30     70           30     70           30     70
    40  60  100         40  60  100         40  60  100     20  40  60  100
          70                                        90                 90

   Remove 30           Remove 70
      50                  50
   40     70           40     90
20      60   100    20      60  100
            90
```

1e AVL tree:

```
   Insert 10           Insert 100          Insert 30           Insert 80
      10                  10                  30                  30
                              100          10     100          10     100
                                                                    80

   Insert 50           Remove 10           Insert 60           Insert 70
      30                  80                  80                  60
   10     80           30     100          50     100          50     80
       50   100          50             30   60             30     70   100
       Insert 40              Remove 80              Insert 90              Insert 20

          60                     60                     60                     60

       40     80              40     70              40     90              40     90

   30   50  70  100       30     50      100     30    50 70   100      30    50 70    100

                                                                  20

       Remove 30              Remove 70

          60                     60

       40     90              40     90

   20    50 70   100      20     50      100
```

---

2 The height of a binary search tree depends on the order in which the items are inserted. A tree containing *n* items can have a height as great as *n*—in which case it behaves like a linked chain—and as small as $\log_2 (n+1)$ when it is completely balanced. Maintaining a balanced binary search tree may become very expensive in the face of frequent insertions and removals, as the entire tree must be rebuilt in the worst case. A 2-3 tree is always balanced and thus has height proportional to log *n*. Since the time required by most dictionary operations is proportional to the height of the tree, a 2-3 tree is a more efficient dictionary implementation than a binary search tree.

3

```
// Visits all items in a nonempty 2-3 tree within the range low...high.
rangeQuery(23Tree: TwoThreeTree, low: ItemType, high: ItemType,
           visit(item: ItemType): void): void

   r = 23Tree's root node
   if (r.isLeaf())
      Visit r's data item(s) that are within the range
   else if (r.isThreeNode()) // r has two data items
   {
      rangeQuery(left subtree of r, low, high, visit)
      item = r.getSmallItem()
      if ((low <= item) && (item <= high))
         visit(item)
      rangeQuery(middle subtree of r, low, high, visit)
      item = r.getLargeItem()
      if ((low <= item) && (item <= high))
         visit(item)
      rangeQuery(right subtree of r, low, high, visit)
   }
   else                      // r is a 2-node and has one data item
   {
      rangeQuery(left subtree of r, low, high, visit)
      item = r.getSmallItem()
      if ((low <= item) && (item <= high))
         visit(item)
      rangeQuery(right subtree of r, low, high, visit)
   }
```

4

```
              h|r
               |
        c|e         o          u|w
       / | \       / \        /  |  \
      b  d  f     k   p     s|t  v   y
```

5

```
              o
             /  \
           m      q
          / \    / \
         h   n  p   r|v
```

6

```
              h |  n
             /  |   \
         a|d|g  j   o|s
```

7

```
                                    37|50

                       30|35             39             70|90

        10|20     32|33|34    36   38   40    60     80     100
```

9

```
                                    50

                       37                            70

                35             39              60              90

          30          36     38      40                      80     100

     20       33

  10       32    34
```

10

```
                                    37 | 50

                       30|35              39                   70|85|90

    10|20          32|33|34    36   38    40|45   60|65   80   87|89    100
```

Chapter 20 Graphs

1a The adjacency matrix for the weighted graph in Figure 20-33:

	0	1	2	3	4	5
0	0	9	∞	∞	1	∞
1	9	0	8	∞	6	∞
2	∞	8	0	5	∞	2
3	∞	∞	5	0	∞	∞
4	1	6	∞	∞	0	7
5	∞	∞	2	∞	7	0

The adjacency list:

$\boxed{0}$ –> $\boxed{1 | 9}$ –> $\boxed{4 | 1}$

$\boxed{1}$ –> $\boxed{2 | 8}$ –> $\boxed{4 | 6}$

$\boxed{2}$ –> $\boxed{3 | 5}$ –> $\boxed{5 | 2}$

$\boxed{3}$ –> $\boxed{2 | 5}$

$\boxed{4}$ –> $\boxed{1 | 6}$ –> $\boxed{5 | 7}$

$\boxed{5}$ –> $\boxed{2 | 2}$ –> $\boxed{3 | 5}$

1b The adjacency matrix for the directed graph in Figure 20-34 (a 1 indicates the existence of an edge, and a 0 indicates the absence of an edge):

	a	b	c	d	e	f	g	h	i
a	0	1	1	0	0	0	0	0	0
b	0	0	0	1	0	0	0	1	0
c	0	0	0	1	1	0	0	0	0
d	0	0	0	0	0	0	0	1	0
e	0	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	1	0	1
g	0	0	1	0	0	0	0	0	0
h	0	0	0	0	0	0	1	0	0
I	0	0	1	0	0	0	0	0	0

The adjacency list:

$\boxed{a}$ –> $\boxed{b}$ –> $\boxed{c}$

$\boxed{b}$ –> $\boxed{d}$ –> $\boxed{h}$

$\boxed{c}$ –> $\boxed{d}$ –> $\boxed{e}$

$\boxed{d}$ –> $\boxed{h}$

$\boxed{e}$ –> $\boxed{g}$

$\boxed{f}$ –> $\boxed{g}$ –> $\boxed{i}$

$\boxed{g}$ –> $\boxed{c}$

$\boxed{h}$ –> $\boxed{g}$

$\boxed{i}$ –> $\boxed{c}$

2 Let each integer require *m* bytes and assume that addresses are stored as integers. The size of the adjacency matrix is then (9 * 9) *m* = 81*m* bytes. The adjacency list requires an array of 9 addresses of size *m* plus ten nodes each composed of an integer for storing the vertex name and another integer for the address of the next node (if any). Thus the total is $9m + (10 * 2)\ m = 29m < 81m$.

3a Yes

3b

	a	b	c	d	e	f
a	0	1	1	0	0	0
b	1	0	1	0	0	0
c	1	1	0	1	1	1
d	0	0	1	0	0	1
e	0	0	1	0	0	0
f	0	0	1	1	0	0

3c

$\boxed{a}$ –> $\boxed{b}$ –> $\boxed{c}$

$\boxed{b}$ –> $\boxed{a}$ –> $\boxed{c}$

$\boxed{c}$ –> $\boxed{a}$ –> $\boxed{b}$ –> $\boxed{d}$ –> $\boxed{e}$ –> $\boxed{f}$

$\boxed{d}$ –> $\boxed{c}$ –> $\boxed{f}$

$\boxed{e}$ –> $\boxed{c}$

$\boxed{f}$ –> $\boxed{c}$ –> $\boxed{d}$

4 The possible vertices at each step are visited in sorted order.

Graph in Figure 20-33:
Depth-first search: 0, 1, 2, 3, 5, 4.
Breadth-first search: 0, 1, 4, 2, 5, 3.

Graph in Figure 20-36:
Depth-first search: a, b, c, d, f, e, g, i, h.
Breadth-first search: a, b, c, d, e, f, h, g, i.

5 During an iterative DFS, the vertices on the stack lie along a path. Thus, if the vertex at the top of the stack is adjacent to a vertex currently on the stack, a cycle exists. To detect a cycle, we need to do more than simply designate a vertex as visited. We must indicate that it is either on the stack or removed from the stack because we are finished with it. Thus, each vertex is in one of three states: unvisited, on the stack, or finished. You can add a field to the vertex to indicate its state, as well as methods to report or change the state. We will use the states unvisited, onStack, and finished. Some people use colors—such as white, gray, and black—to indicate the states.

```
// Detects whether a cycle exists beginning at vertex v.
hasCycle(v: Vertex): boolean

   s= a new empty stack
   Mark all vertices as unvisited

   // Push v onto the stack and mark it
   s.push(v)
   Mark v as onStack

   // Loop invariant: there is a path from vertex v at the
   // bottom of the stack s to the vertex at the top of s
   while (!s.isEmpty())
   {
      top = s.peek()
      if (top has a neighbor marked as onStack)
         return true  // A cycle exists
      else if (top has an unvisited neighbor u)
      {
         Mark u as onStack
         s.push(u)
      }
      else
      {
         Mark top as finished
         s.pop() // Backtrack
      }
   }
   return false
```

---

6 Using `topSort1`:

Graph (a):	a, d, e, b, c
Graph (b):	a, d, b, c
Graph (c):	a, e, d, b, c

---

7 Using `topSort1`:
(a)

Action	Stack (bottom -> top)	Ordered List
push a	a	Empty
push b	a b	Empty
push c	a b c	Empty
pop c	a b	C
pop b	a	b c
push d	a d	b c
push e	a d e	b c
pop e	a d	e b c
pop d	a	d e b c
pop a	empty	a d e b c

(b)

Action	Stack (bottom -> top)	Ordered List
push a	a	Empty
push b	a b	Empty
push c	a b c	Empty
pop c	a b	C
pop b	a	b c
push d	a d	b c
pop d	a	d b c
pop a	empty	a d b c

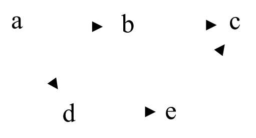
(c)

Action	Stack (bottom -> top)	Ordered List
push a	a	Empty
push b	a b	Empty
push c	a b c	Empty
pop c	a b	C
pop b	a	b c
push d	a d	b c
pop d	a	d b c
push e	a e	d b c
pop e	a	e d b c
pop a	empty	a e d b c

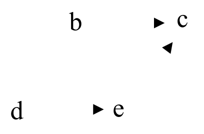
8

```
// Arranges the vertices of the graph g into a
// topological order and returns them in a list.
topSort1(g: Graph): List

   aList = a new empty list
   n = number of vertices in g

   for (step = 1 through n)
   {
      Select a vertex v that has no predecessors
      aList.insert(step, v)
      Delete vertex v and its edges from g
   }
   return aList
```

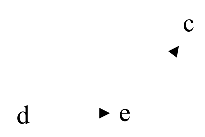
The topological order for the graph in Figure 20-37a is a, b, d, e, c, as seen from the following trace of the new version of `topSort1`:



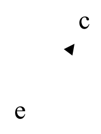
Remove `a` from the graph
Add `a` to a list



Remove `b` from the graph
Add `b` to a list



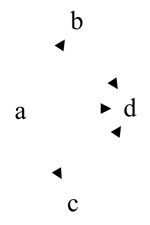
Remove `d` from the graph
Add `d` to a list



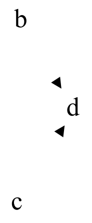
Remove `e` from the graph
Add `e` to a list

Remove `c` from the graph
Add `c` to a list

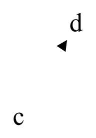
The topological order for the graph in Figure 20-37b is a, b, d, c, as seen from the following trace of the new version of `topSort1`:



Remove `a` from the graph
Add `a` to a list



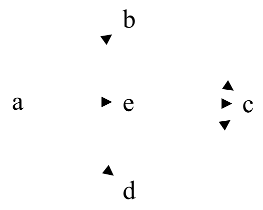
Remove `b` from the graph
Add `b` to a list



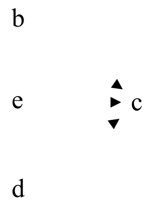
Remove `c` from the graph
Add `c` to a list

Remove `d` from the graph
Add `d` to a list

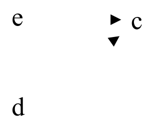
The topological order for the graph in Figure 20-37c is a, b, d, e, c, as seen from the following trace of the new version of `topSort1`:



Remove `a` from the graph
Add `a` to a list



Remove `b` from the graph
Add `b` to a list



Remove `d` from the graph
Add `d` to a list

Remove `e` from the graph
Add `e` to a list

Remove `c` from the graph
Add `c` to a list

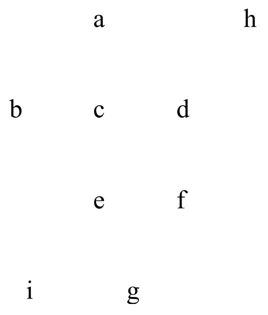
---

9 The trace of the DFS algorithm is in Figure 20-12. As each new vertex is visited, the edge between it and the vertex at the top of the stack is marked and becomes part of the spanning tree. The new vertex is then added to the stack. As a result, we get the tree in Figure 20-20.

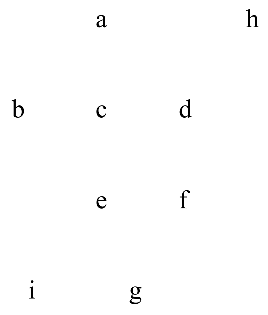
The trace of the BFS algorithm is in Figure 20-13. When a vertex is removed from the queue, the edges between it and the next group of visited vertices are marked and become a part of the spanning tree. For example, *a* has edges to *b*, *f* and *i*; *b* has edges to *c* and *e*, and so on. Thus the tree in Figure 20-21 is the result.

---

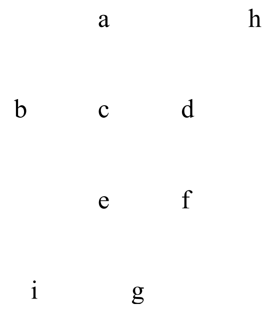
10 A DFS spanning tree



A BFS spanning tree



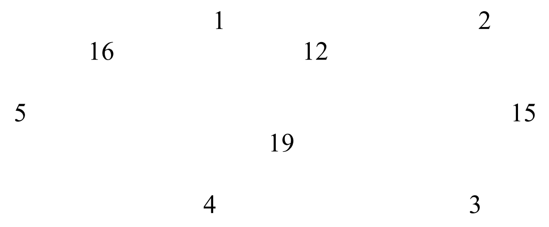
A minimum spanning tree rooted at vertex *a*.



11a

```
                  1                 2
          16            12
     5                                 15
           22
                  4                 3


                  1                 2
          16            12
     5                                 15
                     19
                  4                 3


                  1                 2
          16            12
     5
                       19
          22
                  4                 3


                  1                 2
                        12
     5                                 15
                     19
          22
                  4                 3


                  1                 2
          16
     5                                 15
                       19
          22
                  4                 3
```

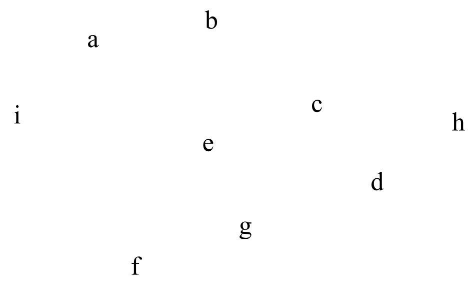
11b



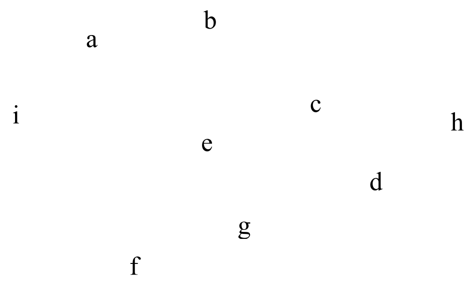
12

```
// Forms a spanning tree for a connected, undirected graph
// beginning at vertex v by using a depth-first search.
DFSTree(v: Vertex): void

    aStack = a new empty stack

    // Push v onto the stack and mark it
    aStack.push(v)
    Mark v as visited

    // Loop invariant:  there is a path from the vertex v at the
    // bottom of the stack s to the vertex at the top of s
    while (!aStack.isEmpty())
    {
       if (no unvisited vertices are adjacent to
           the vertex at the top of the stack)
          aStack.pop()  // Backtrack
       else
       {
          Select an unvisited vertex u adjacent to
            the vertex at the top of the stack
          Mark the edge from u to the vertex at the
            the top of the stack.
          aStack.push(u)
          Mark u as visited
       }
    }
```

13 (a) Minimal spanning tree rooted at *g*



(b) Minimal spanning tree rooted at *c*



14 A trace of the shortest-path algorithm for the graph given in Figure 20-39 of the text.
Let w[i] stand for `weight[i]` in the table below:

Step	*v*	vertexSet	w[0]	w[1]	w[2]	w[3]	w[4]	w[5]	w[6]
1	-	[0]	0	2	4	6	∞	∞	∞
2	1	[01]	0	2	4	6	5	∞	∞
3	2	[012]	0	2	4	6	5	∞	∞
4	4	[0124]	0	2	4	6	5	10	6
5	3	[01243]	0	2	4	6	5	9	6
6	6	[012436]	1	2	4	6	5	8	6
7	5	[0124365]	1	2	4	6	5	8	6

---

16 One such circuit is: *a b e f c g f b c d g e a.* We know the graph has an Euler circuit because the graph is connected.

---

17 Consider any connected undirected graph, *G*, not containing loops or multiple edges. If *G* has *n* vertices and *n* – 1 edges, *G* must form a tree. Since *G* is a tree, it does not have a cycle. Moreover, *G* has two vertices, *i* and *j*, that are not connected by an edge. However, there is a path from vertex *i* to vertex *j*. Adding an edge between *i* and *j* will form a cycle.

---

18 Suppose that the traversal starts at vertex *v* and does not visit vertex *w*. Then there is no path from *v* to *w* and so the graph is not connected, contrary to our assumption. On the other hand, if the traversal starts at vertex *v* and does visit every other vertex in the graph. Then we assert that there is a path between every pair of vertices *u* and *w*. To see this conclusion, we note that a) there is a path from *v* to *u*, and b) there is a path from *v* to *w*. Since the graph is undirected, there is also a path from *u* to *v*, and so a path from *u* to *w* proceeds from *u* to *v* and then from *v* to *w*. Hence, the graph is connected.

19a The recursive BFS algorithm is not simple because the underlying data structure is a queue. We use a queue to ensure that vertices close to the current vertex are visited before those further away. DFS uses a stack and so can naturally be implemented recursively.

19b

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Recursive version.
bfs(v: Vertex)

   aQueue = a new empty queue
   Mark v as visited

   for (each unvisited vertex u adjacent to v)
      aQueue.enqueue(u)

   // Loop invariant:  there is a path from vertex v
   // to every vertex in the queue
   while (!aQueue.isEmpty())
   {
      aQueue.dequeue(w)
      bfs(w)
   }
```

---

20 Proof of the loop invariant of Dijkstra's shortest-path algorithm:

The loop invariant: For v not in *vertexSet*, *weight*[v] is the smallest weight of all paths from 1 to v that pass through only vertices in *vertexSet* before reaching v. For v in *vertexSet*, *weight*[[v] is the smallest weight of all paths from 1 to v (including paths outside *vertexSet*) and the shortest path from 0 to v lies entirely in *vertexSet*.

Proof by induction on the step of the algorithm.
Base case: Step = 1. The initialization step creates the set *vertexSet* containing vertex 0 and sets *weight* to the first row of the adjacency matrix. The paths that the invariant describes are simply single edges from vertex 0 to any vertex, and the weights for these paths are in the array *weight*. The invariant is true for `step` = 1.

Inductive hypothesis: Assume the invariant is true during steps 2 through $k < n$.

Inductive conclusion: At the beginning of step $k + 1$, there are k vertices in *vertexSet*. Find the smallest *weight*[v] such that v is not in *vertexSet*, and let P designate the path from 0 to v. By the inductive hypothesis (specifically, the first part of the invariant), *weight*[v] is the smallest weight of all paths from 0 to v that pass through only vertices in *vertexSet* before reaching v. Let v' be the last vertex on the path P that is in *vertexSet*. By the inductive hypothesis (the second part of the invariant), *weight[v']* is the smallest weight of all paths from 0 to v', including paths outside *vertexSet*, and the shortest path from 0 to v' —call it P'—lies entirely in *vertexSet*. Path P is thus obtained by adding to P' the edge from v' to v. Thus, *weight*[v] is obtained by adding to *weight[v']* the weight of the edge from v' to v. Therefore, when the algorithm adds v to *vertexSet* and adjusts W such that

$$weight[\ u] \leq weight[v] + matrix[v, u]$$

for each vertex u not in *vertexSet*, *weight*[u] must be the smallest of the weights of all paths from 0 to u that pass through only vertices in *vertexSet* before reaching u.

We must now show that *weight[v]* is the smallest weight of all paths from 0 to v (including paths outside *vertexSet*). If this statement were untrue, there would be a path from 0 to v whose weight was less than *weight[v]*. If this shorter path passed only through vertices in *vertexSet*, its existence would contradict the choice of v: v was chosen earlier so that *weight[v]* would be a minimum of all paths that passed through vertices in *vertexSet* before reaching v. On the other hand, if the shorter path leaves *vertexSet* before reaching v, it must pass through a vertex v' outside of *vertexSet*. Clearly, the path to v' is shorter than the path to v, so *weight[v']* < *weight[v]*. This result also contradicts the earlier choice of v. Therefore, the statement that began this paragraph must be true.

Finally, we must show that the shortest path from 0 to v lies entirely in *vertexSet*. If the path left *vertexSet* before reaching v, it would pass through a vertex v' outside of *vertexSet*. This implication is a contradiction. (*End of proof.*)

Chapter 21 Processing Data in External Storage

1

```
// Makes a gap at record i in block blockNum of the sorted file dataFile.
// Assumptions: recordsPerBlock is the max number of records in a block,
// lastBlock is the block number of the last block in the file;
// this last block is not full.
// lastRec is the record number of the last record in block lastBlock.
shiftData(dataFile: File, i: integer, blockNum: integer): void

   if (blockNum == lastBlock - 1) // Special case
   {
      buf.readBlock(dataFile, blockNum)

      // Assuming lastRecord + 1 < recordsPerBlock
      for (j = lastRecord - 1; j > i; j--)
         buf.setRecord(j, buf.getRecord(j - 1))
   }
   else
   {
      // First, process block blockNum; get the block
      buf.readBlock(dataFile, blockNum)

      // Save the last record
      temp1 = buf.getRecord(recordsPerBlock)

      // Make a gap at location i
      for (j = recordsPerBlock - 1; j > i; j--)
         buf.setRecord(j, buf.getRecord(j - 1))

      // Write the block
      buf.writeBlock(dataFile, blockNum)

      // Process all blocks except lastBlock
      for (k = blockNum + 1; k < lastBlock; k++)
      {
         buf.readBlock (dataFile, k)

         // Save the last record
         temp2 = getRecord(recordsPerBlock - 1)

         // Shift all records down one position within its block,
         // leaving a gap at location 1
         for (j = recordsPerBlock - 1; j > 0; j--)
            buf.setRecord(j, buf.getRecord(j - 1))

         // Insert record saved from last location in previous block
         buf.setRecord(0, temp1)

         // Get ready for next block
         temp1 = temp2
         buf.writeBlock(dataFile, k)
      }
      // Process lastBlock
      buf.readBlock(dataFile, lastBlock)
      for (j = recordsPerBlock - 1; j > 0; j--)
         buf.setRecord(j, buf.getRecord(j - 1))

      buf.setRecord(0, temp1)
      buf.writeBlock(dataFile, lastBlock)
   }
```

2 We need to keep track of the available, or free, blocks in the file. Rather than maintaining a separate list of the free blocks, we can allocate some space in each block for the numbers of its preceding and succeeding blocks. The free blocks are then "doubly linked" together within the file. When either a record is deleted from a full block or a new block is allocated, we add that block to the front of the free-block list. When a slot is needed for a record, use the block at the beginning of the list; if the block becomes full, remove it from the list. When a record is deleted from a block that contains only one record—making that block empty—return the block's memory to the system and delete it from the free-block list.

```
// Assume that freeList is the number of the first block with available space.
getSlot(dataFile: File)

   if (freeList != 0)
   {
      // Free list is not empty; there is room in existing blocks
      blockNum = freeList

      // Read first block on free list into internal buffer buf
      buf.readBlock(dataFile, blockNum)
      recNum = buf.emptySlot() // Find an empty slot
      buf.markUsed(recNum)

      // Has the block become full?
      if (buf.numFreeSlots() == 0)
         freeList = buf.nextBlock()
      buf.writeBlock(dataFile, blockNum)
   }
   else
   {  // Free list is empty; get a block from the system
      allocateBlock(dataFile, blockNum)
      buf.readBlock(dataFile, blockNum)
      recNum = 1
      buf.markUsed(recNum)

      // Start a free list
      buf.nextBlockUpdate(NULL)
      buf.prevBlockUpdate(NULL)
      freeList = blockNum
      buf.writeBlock(dataFile, blockNum)
   }
   return blockNum and recNum



freeSlot(dataFile: File, blockNum: integer, recNum: integer) : void

   buf.readBlock(dataFile, blockNum)
   if (buf.blockFull())
   {  // Add to freeList
      buf.markUnused(recNum)
      buf.nextBlockUpdate(freeList)
      buf.prevBlockUpdate(NULL)
      buf.writeBlock(dataFile, blockNum)
      buf.readBlock(dataFile, freeList)
      buf.prevBlockUpdate(blockNum)
      buf.writeBlock(dataFile, freeList)
      freeList = blockNum
   }
```

```
      else
      {
         buf.markUnused(recNum)
         if (buf.blockEmpty())
            Detach block from freeList and return it to system
      }
```

3

```
// Inserts newItem into tData and updates the index.
insert(tIndex: File, tData: File, newItem: ItemType): void

   // Find an empty slot in tData; if no empty slot exists, get a new block.
   p = number of blocks with an empty slot
   data.readBlock(tData, p)
   data.setRecord(emptySlotNumber, newItem)
   data.writeBlock(tData, p)

   // Apply hash function to newItem's key
   i = hash(newItem.getKey())

   // Get first block on chain of index blocks
   q = table[i]
   if (q == 0)
   { // No items hashed there so far
      allocateBlock(tIndex, q)
      buf.readBlock (tIndex, q)

      // p is tData block that newItem is in
      buf.setRecord(1, <newItem.getKey(), p>)
      table[i] = q  // Attach new block to hash table
      buf.writeBlock(tIndex, q)
   }
   else
   {
      buf.readBlock(tIndex, q)
      while ((q!=0) && (there is no EmptySlot in q))
      {
         q = block number of next block on chain
         if (p != 0)
            buf.readBlock(tIndex, q)
      }
      if (q != 0)
      { // Block with empty slot
         buf.setRecord(j, <newItem.getKey(), p>)
         buf.writeBlock(tIndex, q)
      }
      else
      {  // Allocate a new block
         allocateBlock(tIndex, q)
         buf.readBlock(tIndex, q)

         // Attach Q to beginning of chain
         buf.setRecord(ptr, table[i])

         // The block must contain an indicator of the next block
         table[i] = q
         buf.setRecord(1, <newItem.getKey(),p>)
         buf.writeBlock(tIndex, q)
      }
   }
```

```
// Deletes item with key searchKey from the file tData and updates the tIndex file.
// Finds the item as described in the retrieve pseudocode in the text.
delete(tIndex: File, tData: File, searchKey: KeyType): boolean

   if (p != 0)
   {  // The item is present
      blockNum = block number of data file pointed to by buf.getRecord(j)
      Mark buf.getRecord(j) deleted and do whatever memory management is appropriate

      // At this point, we would call freeSlot. See Exercises 2 and 8.
      buf.writeBlock(tIndex, p)

      // Get the appropriate block from the data file
      data.readBlock(tData, blockNum)
      Find data record data.getRecord(k) with search key searchKey
      Mark data.getRecord(k) deleted and do appropriate memory management
      data.writeBlock(tData, blockNum)
      return true
   }
   else
      return false
```

---

4 B-tree of order 5 (starting with the empty tree):

```
add(10)             10

add(100)            10 | 100

add(30)             10 | 30 | 100

add(80)             10 | 30 | 80 | 100

add(50)                          50


                    10 | 30             80 | 100

remove(10)          30 | 50 | 80 | 100

add(60)                          60


                    30 | 50             80 | 100

add(70)                          60


                    30 | 50             70 | 80 | 100

add(40)                          60


                    30 | 40 | 50        70 | 80 | 100

remove(80)                       60


                    30 | 40 | 50        70 | 100

add(90)                            60
```

```
                    30 | 40 | 50        70 | 90 | 100

add(20)                            60

                    20 | 30 | 40 | 50   70 | 80 | 90 | 100

remove(30)                       60

                    20 | 40 | 50        70 | 80 | 90 | 100

remove(70)                       60

                    20 | 40 | 50        80 | 90 | 100
```

---

5 **a.** 31 nodes
b. 124 records

6

```
                  [ g k p ]
                 /  /    \  \
               /  /        \  \
        [a b c d e] [h i j] [m n o] [r s v w x y]
```

7

```
                 [q]
                /   \
              /       \
        [b f j l n o]  [r u x]
```

---

8 The following pseudocode assumes that the block containing the item has been read already. If you do not want to make this assumption, you can pass `rootNum` instead of `block` to the function. However, to swap `searchKey` with its inorder successor, having both blocks in memory is useful.

```
// Returns number of the block containing the search key of the inorder successor
// of searchKey. block contains searchKey.
inorderSuccessor(tIndex: File, searchKey: KeyType, block: BlockType): integer

   k = the key position of searchKey // 1..m-1 for an order m tree
   succBlockNum = block number containing the root of the subtree immediately following k
   if (succBlockNum != 0) // If a subtree exists
   {
      succBlock.readBlock(tIndex, succBlockNum)
      succBlockNum = The block number of the root of the 0th subtree
      while (succBlockNum != 0)
      {
         succBlock.readBlock (tIndex, succBlockNum)
         succBlockNum = the block number of the root of the 0th subtree
      // When this loop exits, we're at the leftmost leaf of the subtree
      // immediately to the right of searchKey. The leftmost key value
      // will be the inorder successor of searchKey.
      succ = the first key in succBlock
   }
   else
   {
      succBlockNum = 0
      if (searchKey is not the m-1th key)
         succ = key immediately to right of searchKey
      // else there is no inorder successor
   }
   return succBlockNum
```

---

9 An actual implementation would be complicated by memory considerations, such as how many blocks can be in memory at one time.

```
   // Inserts newItem into the dictionary stored in tData and updates the index
   // stored in tIndex.
   insert(tIndex: File, tData :File, newItem: ItemType): void

      x = search key of newItem
      Find a free slot in tData // Use memory management routines
     // At this point, call getSlot (see Exercises 2 and 8)

     // blockNum is the block number of the block in tData that has a free slot
      data.readBlock (tData , blockNum)
      data.setRecord(j, newItem) // j is the free slot in block blockNum
      data.writeBlock(tData , blockNum)

     // Update index
      Locate the leaf block leafNum in tIndex
      buf.readBlock(tData , leafNum)
      Add newItem to block leafNum

      if (block leafNum now has m items)
         split(leafNum)


   // Splits the node in block blockNum. This block contains m items. If it is
   // an internal node, it has m + 1 children.
   split(blockNum: integer):void

      p = parent of block blockNum
     // If block blockNum is the root, get a new block for p
     // Replace block blockNum by two blocks (blockNum1, blockNum2)
     // Note: we may have to be careful here, depending on how many blocks
     // can be in memory at one time.
     // Give blockNum1 the m div 2 items in blockNum with smallest search key values
     // Give blockNum2 the m div 2 items in blockNum with largest search key values
      if blockNum (contains an internal node)
      {
        // blockNum1 becomes the parent of blockNum's (m+1) div 2 leftmost children
        // blockNum2 becomes the parent of blockNum's (m+1) div 2 rightmost children
      }
      Send to p(key in blockNum with the middle search key value)

      if (p now has m items)
         split(p)
```

10

```
// Visits the search keys between low and high that are contained in tIndex,
// which is organized as a B-tree of order m.
// rootnum is the block number in tIndex of the root of the B-tree.
rangeQuery(tIndex: File, rootNum: integer, low: KeyType, high: KeyType,
           visit(key: KeyType): void): void

   buf.readBlock(tIndex, rootNum)
   if (rootNum is a leaf)                  // Key
   {
      for (each search key k in the leaf)
         if ((k >= low) and (k <= high))
            visit(k)
   }
   else                                   // Internal node
   {
      if (low < key_0 of the root)
         rangeQuery(tIndex, 0, low, high, visit)
      if (key_1 >= low) && (key_1 <= high)
         visit(key_1)
      for (i = 1; i < number of keys in root - 1; i++)
      {
         if (high >= key_i) || (low <= key_i+1)
            // If any values in the i-th subtree are in the range
            rangeQuery(tIndex, i, low, high, visit)
         if (key_i+1 >= low) && key_i+1 <= high)
            visit(key_i+1])
      }
      if (high > last key in root)
         rangeQuery(tIndex, last subtree in root, low, high, visit)
   }
```

---

11 See Exercises 3 and 9. The locations for the memory management calls are indicated.

12

```
// Traverses the index file of a B-tree in sorted order.
// blockNum is the block number of the root of the B-tree in the index file.
// buf is a buffer that is not local to this function; if it were local, it would
// be part of the recursive stack, but we're assuming there is insufficient
// memory to accommodate a recursive stack containing h blocks.
traverse(blockNum: integer, visit(key: KeyType): void

   // Read the root
   if (blockNum != 0)
   {
      buf.readBlock(tIndex, blockNum)
      // Traverse the children
      p = block number of the 0th child of buf
      traverse(p, visit)

      // We read the block again only if a subtree at the i-1st position exists
      for (i = 1; i < m; i++)
      {
         if (p != 0)
            buf.readBlock(tIndex, blockNum)
         visit(key k_i of buf)
         p = block number of the ith child of B
         traverse(p, visit)
      }
   }
```

13a For a range query, replace each call to `visit` in Exercise 10 with the following statements:

```
pData = pointer in the index record of buf for the search key
theData.readBlock(tData, pData)
Extract from theData the record that has the appropriate search key
visit(data record) // Where visit's parameter is ItemType instead of KeyType
```

13b If the records in the data file are sorted, you do not need to use the index file to traverse the records in sorted order. For a range query, you must get the block number of the first block containing the search keys in the range. In this case, you do need the index file.

```
traverse(tIndex: File, tData: File, rootNum: integer, visit(data: ItemType): void): void

   // Find the smallest key in the index
   buf.readBlock(tIndex, rootNum)
   p = block number of the root of the 0th subtree
   while (p != 0)
      buf.readBlock(tIndex, rootNum)

   // Traverse the data file
   pData = pointer in the first index record of buf
   for (i = pData; i <= lastBlock; i++)
   // May want special case for last block which may not be full
   {
      data.readBlock(tData, i)
      for (j = 0; j < recordsPerBlock; j++)
         visit(data.getRecord(j))
   }


rangeQuery(tIndex: File, tData: File, low: KeyType, high: KeyType, rootNum: integer
           visit(data: ItemType): void): void

   buf.readBlock(tIndex, rootNum)
   Locate the leaf where search key low would be
   Find the first search key higher than low
   pData = pointer in the appropriate index record of buf
   done = false
   while(!done)
   {
     data.readBlock(tData, pData)
     j = location of the first record in data whose search key is not smaller than low
     while ((j <= recordsPerBlock) && ((data.getRecord(j)).getkey() <= high))
     {
        visit(data.getRecord(j))
        j++
     }
     pData++
     done = ((j != recordsPerBlock) && (pData > lastBlock))
   }
```

13c Since `tData` is sorted, it is no longer possible to use any free slot to insert a new item. You can insert the new search key into the index, find the inorder predecessor's data block number, and insert the new data item into that block (or successor blocks), allocating a new block if necessary. You can use the shift data algorithm developed in Exercise 1, but you shift only until the end of the chain, rather than the end of the file. Also, you cannot assume that the last block in the chain will have room—it is possible that we must allocate another block.

Deletions from the data file involves moving records to fill the empty slot. This could involve moving records across block boundaries (similar to the shifting data necessary for insertion), and could result in returning a block to the system if the last block in the chain becomes empty.

Retrieving an item might need to search a chain of blocks to find the correct item.

Traversal and range-query operations with sorted chains of blocks require fewer block accesses than if the data file were unsorted. For a traverse, use the B-tree to find the location of first record. Visit each record in the chain (one or more block reads, depending on the length of the chain), and note the search key of the last record. Using the B-tree, find the inorder successor of that key value, and traverse the chain starting with that value. Repeat the process until all records have been visited. The same process can be used for range query, except that we start at low and continue until we reach high.

---

14 We sort `size` entries in array `anArray` by using an iterative merging technique. We start by merging runs of size 1 (already sorted) into sorted runs of size 2, then of size 4, and so on.

```
/** Merges two runs in a given array and places the result in another array.
  @param source  The array containing the original two runs.
  @param r1first  The index of the first entry in run 1.
  @param r1last  The index of the last entry in run 1.
  @param r2first  The index of the first entry in run 2.
  @param r2last  The index of the last entry in run 2.
  @param dest  The array containing the merged runs.
  @param first  The index of the first entry of the merged run in dest.
  @param size  The number of entries in each run. */
void merge(ItemType source[], int r1first, int r1last, int r2first, int r2last,
           ItemType dest[], int first, int size)
{
   if (r2last > size)
      r2last = size;
   while ((r1first <= r1last) && (r2first <= r2last))
   {
      if (source[r1first] <= source[r2first])
      {
         dest[first] = source[r1first];
         first++;
         r1first++;
      }
      else
      {
         dest[first] = source[r2first];
         first++;
         r2first++;
      }  // end if
   }  // end while

   while (r1first <= r1last)
   {
      dest[first] = source[r1first  ];
      first++;
      r1first  ++;
   }  // end while
```

```
      while (r2first <= r2last)
      {
         dest[first] = source[r2first];
         first++;
         r2first++;
      }  // end while
   }  // end merge


   void mergesort(ItemType anArray[], int size)
   {
      int runSize;
      ItemType tempArray[]; // Mergesort needs extra storage
      bool which = true; // Flag to distinguish the source and
                         // destination arrays.
                         // True: anArray is source, tempArray is destination.
                         // False: tempArray is source, anArray is destination.
      runSize = 1;
      while (runSize < size)
      {
         int r1 = 1;             // Index to beginning of first run
         int r2 = r1 + runSize;  // Index to beginning of second run
         int r = r1;             // Index to merged run in second array
         while (r2 <= size)
         {
            if (which)
               merge(anArray, r1, (r1 + runSize - 1), r2, r2 + runSize - 1,
                     tempArray, r, runSize);
            else
               merge(anArray, r1, (r1 + runSize - 1), r2, r2 + runSize - 1,
                     tempArray, r, runSize);
            r1 = r1 + (runSize * 2);
            r2 = r1 + runSize;
            r = r1;
         }
         runSize = runSize * 2;
         which = !which;
      }  // end while (runSize < size)

      if (!which) // Sorted array is in B
         for (int i = 0; i < size; i++)
            anArray[i] = tempArray[i];
   }  // end mergeSort
```