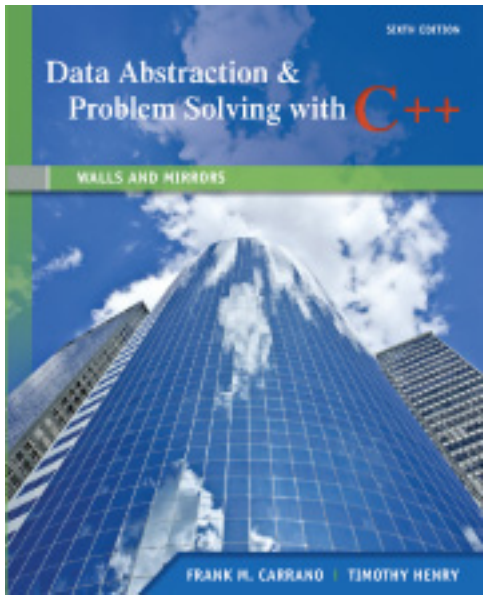
Answers to Checkpoint Questions

Data Abstraction & Problem Solving with C++

WALLS AND MIRRORS

SIXTH EDITION



Frank M. Carrano

University of Rhode Island

Timothy Henry

University of Rhode Island

Chapter 1

Question 1 Write specifications using UML notation for a function that computes the sum of the first five positive integers in an array of *n* arbitrary integers.

The specifications include type definitions, arguments, and preconditions and postconditions.

```
// Computes the sum of the first 5 positive integers
// in an array anArray.
// Precondition: The array anArray contains n integers, n >= 5;
// at least 5 integers in anArray are positive.
// Postcondition: Returns the sum of the first 5 positive
// integers in anArray; anArray and n are unchanged.
sumOf(in anArray: arrayType, in n: integer): integer
```

Another solution:

```
// Computes the sum of the first 5 positive integers
// in an array anArray.
// Precondition: The array anArray contains n integers.
// Postcondition: If anArray contains at least 5
// positive integers, returns their sum; otherwise, returns 0.
// anArray and n are unchanged.
sumOf(in anArray: arrayType, in n:integer): integer
```

Question 2 What is an abstract data type?

An abstract data type is a specification of data values and the operations on that data. This specification does not indicate how to store the data or how to implement the operations, and it is independent of any programming language.

Question 3 What steps should you take when designing an ADT?

- Describe the data.
- Identify the ADT's behaviors.
- Write the behaviors on a CRC card.
- Specify each operation by writing comments and UML; clearly identify assumptions.
- Refine the specifications by considering their behavior under unusual conditions.
- Write an abstract base class.

C++ Interlude 1

Question 1 Revise the parameterized constructor to call the base-class's constructor instead of `MagicBox`'s constructor.

```
template<class ItemType>
MagicBox<ItemType>::MagicBox(const ItemType& theItem)
{
  PlainBox<ItemType>::setItem(theItem);
  firstItemStored = true;
}  // end constructor
```

Chapter 2

Question 1 The following function computes the sum of the first $n \geq 1$ integers. Show how this function satisfies the properties of a recursive function.

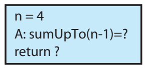
```
/** Computes the sum of the integers from 1 through n.
 @pre  n > 0.
 @post  None.
 @param n  A positive integer
 @return  The sum 1 + 2 + . . . + n. */
int sumUpTo(int n)
{
   int sum = 0;
   if (n == 1)
      sum = 1;
   else // n > 1
      sum = n + sumUpTo(n - 1);

   return sum;
}  // end sumUpTo
```

The product of n numbers is defined in terms of the product of $n - 1$ numbers, which is a smaller problem of the same type. When n is 1, the product is `anArray[0]`; this occurrence is the base case. Because $n \geq 1$ initially and n decreases by 1 at each recursive call, the base case will be reached.

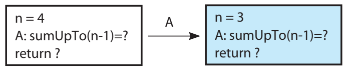
Question 2 Write a box trace of the function given in Checkpoint Question 1.

We trace the function with 4 as its argument (see next page).

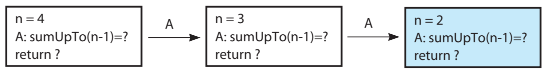
The initial call `sumUpTo(4)` is made, and method `sumUpTo` begins execution:



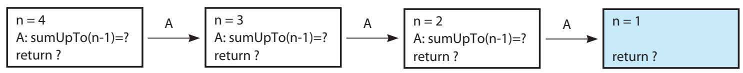
At point A a recursive call is made, and the new invocation of the method `sumUpTo` begins execution:



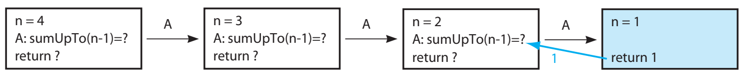
At point A a recursive call is made, and the new invocation of the method `sumUpTo` begins execution:



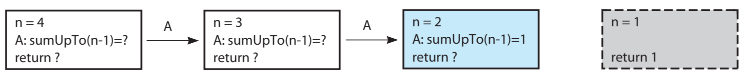
At point A a recursive call is made, and the new invocation of the method `sumUpTo` begins execution:



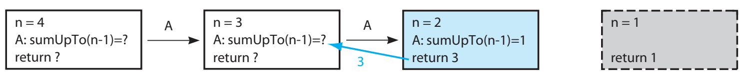
This is the base case, so this invocation of `sumUpTo` completes and returns a value to the caller:



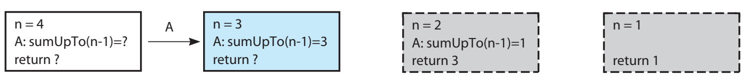
The method value is returned to the calling box, which continues execution:



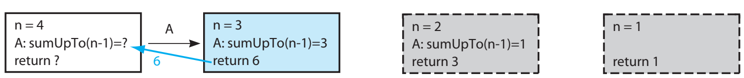
The current invocation of `sumUpTo` completes and returns a value to the caller:



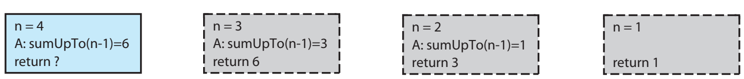
The method value is returned to the calling box, which continues execution:



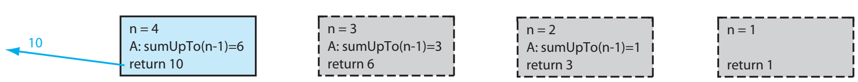
The current invocation of `sumUpTo` completes and returns a value to the caller:



The method value is returned to the calling box, which continues execution:



The current invocation of `sumUpTo` completes and returns a value to the caller:



The value 10 is returned to the initial call.

Question 3 Given an integer $n > 0$, write a recursive function `countDown` that writes the integers n, $n - 1, \ldots, 1$. *Hint:* What task can you do and what task can you ask a friend to do for you?

```
// Precondition: n > 0.
// Postcondition: Writes n, n - 1, ... , 1.
void countDown(int n)
{
   if (n > 0)
   {
      cout << n << endl;
      countDown(n-1);
   }  // end if
}  // end countDown
```

Question 4 In the previous definition of `writeArrayBackward`, why does the base case occur when the value of `first` exceeds the value of `last`?

When `first` > `last`, the array is empty. That is the base case. Since the body of the `if` statement is skipped in this case, no action takes place.

Question 5 Write a recursive function that computes and returns the product of the first $n \geq 1$ real numbers in an array.

```
// Precondition: anArray is an array of n real numbers, n ≥ 1.
// Postcondition: Returns the product of the n numbers in
// anArray.
double computeProduct(const double anArray[], int n),
{
   if (n == 1)
      return anArray[0];
   else
      return anArray[n - 1] * computeProduct(anArray, n - 1);
}  // end computeProduct
```

Question 6 Show how the function that you wrote for the previous question satisfies the properties of a recursive function.

1. `computeProduct` calls itself.
2. An array of `n` numbers is passed to the method. The recursive call is given a smaller array of `n - 1` numbers.
3. `anArray[0]` is the base case.
4. Since `n ≥ 1` and the number of entries considered in `anArray` decreases by 1 at each recursive call, eventually the recursive call is `computeProduct(anArray, 1)`. That is, `n` is 1, and the base case is reached.

Question 7 Write a recursive function that computes and returns the product of the integers in the array `anArray[first..last]`.

```
// Precondition: anArray[first..last] is an array of integers,
// where first <= last.
// Postcondition: Returns the product of the integers in
// anArray[first..last].
double computeProduct(const int anArray[], int first, int last)
{
   if (first == last)
      return anArray[first];
   else
      return anArray[last] * computeProduct(anArray, first,
                                            last - 1);
}  // end computeProduct
```

Question 8 Define the recursive C++ function `maxArray` that returns the largest value in an array and adheres to the pseudocode just given.

```
// Precondition: anArray[first..last] is an array of integers,
// where first <= last.
// Postcondition: Returns the largest integer in
// anArray[first..last].
double maxArray(const int anArray[], int first, int last)
{
   if (first == last)
      return anArray[first];
   else
   {
      int mid = first + (last - first) / 2;
      return max(maxArray(anArray, first, mid),
                 maxArray(anArray, mid + 1, last))
   }  // end if
}  // end maxArray
```

Question 9 Trace the execution of the function `solveTowers` to solve the Towers of Hanoi problem for two disks.

The three recursive calls result in the following moves: Move a disk from *A* to *C*, from *A* to *B*, and then from *C* to *B*.

Question 10 Compute $g(4, 2)$.

6.

Question 11 Of the following recursive functions that you saw in this chapter, identify those that exhibit tail recursion: `fact`, `writeBackward`, `writeBackward2`, `rabbit`, *P* in the parade problem, `getNumberOfGroups`, `maxArray`, `binarySearch`, and `kSmall`.

`writeBackward`, `binarySearch`, and `kSmall`.

Chapter 3

Question 1 What happens to the array `items` when the method `add` cannot add another entry to it, because it is already full?

Nothing.

Question 2 If a client of `ArrayBag` creates a bag `aBag` and a vector `v` containing five items, what happens to those items after the statement `v = aBag.toVector()` executes?

The memory originally allocated for `v` is now inaccessible. The five items in the original vector still exist but are no longer in `v`.

Question 3 What is an advantage and a disadvantage of calling the method `getFrequencyOf` from `contains`?

Advantage: Once `getFrequencyOf` works correctly, writing a correct version of `contains` is simple. Disadvantage: `contains` does more work than needed, because `getFrequencyOf` must check every entry in the bag. A more involved definition of `contains` could stop looking for the desired entry as soon as it is found.

Question 4 Revise the definition of the method `contains` so that it calls the method `getIndexOf`.

```
template<class ItemType>
bool ArrayBag<ItemType>::contains(const ItemType& anEntry) const
{
  return getIndexOf(anEntry) > -1;
}  // end getCurrentSize
```

Question 5 Should we revise the specification of the method `contains` so that if it locates a given entry within the bag, it returns the index of that entry?

No. As a public method, `contains` should not provide a client with such implementation details. The client should have no expectation that a bag's entries are in an array, since they are in no particular order.

Question 6 Revise the definition of the method `getIndexOf` so that it does not use a boolean variable.

```
template<class ItemType>
int ArrayBag<ItemType>::getIndexOf(const ItemType& target) const
{
   int result = -1;
   int searchIndex = 0;

   // If the bag is empty, itemCount is zero, so loop is skipped
   while ((result == -1) && (searchIndex < itemCount))
   {
      if (items[searchIndex] == target)
      {
         result = searchIndex;
      }
      else
      {
         searchIndex++;
      }  // end if
   }  // end while
   return result;
}  // end getIndexOf
```

Chapter 4

Question 1 Consider a linked chain of three nodes, such that each node contains a string. The first node contains "A", the second node contains "B", and the third node contains "C".

a. Write C++ statements that create the described linked chain. Beginning with a head pointer `headPtr` that contains `nullptr`, create and attach a node for "C", then create and attach a node for "B", and finally create and attach a node for "A".

b. Repeat part *a*, but instead create and attach nodes in the order "A", "B", "C".

```
// Part a: Create nodes for C, B, and A; insert nodes at beginning of
chain.
headPtr = new Node<ItemType>("C");
Node<ItemType>* newNodePtr = new Node<ItemType>("B");
newNodePtr->setNext(headPtr);
headPtr = newNodePtr;
newNodePtr = new Node<ItemType>("A");
newNodePtr->setNext(headPtr);
headPtr = newNodePtr;

// Part b: Create nodes for A, B, and C; insert nodes at end of chain.
headPtr = new Node<ItemType>("A");
Node<ItemType>* newNodePtr = new Node<ItemType>("B");
headPtr->setNext(newNodePtr);
Node<ItemType>* lastNodePtr = newNodePtr;
newNodePtr = new Node<ItemType>("C");
lastNodePtr->setNext(newNodePtr);
```

Question 2 Why are only a few changes necessary to reuse the code in Listing 3-2? How would you implement the changes using the “find and replace” functionality of a text editor or IDE?

Because both `ArrayBag` and `LinkedBag` are derived from `BagInterface`, they each implement the same methods. Moreover, `BagInterface` is a template that represents the data type of the entries in the bag generically with the identifier `ItemType`. The program in Listing 3-2 was written in this context. Thus, by replacing `ArrayBag` with `LinkedBag` in Listing 3-2, you will get a program to test the core methods of `LinkedBag`.

Question 3 Why is a `LinkedBag` object not concerned about becoming full?

Because you are able to create a new node and attach it to the linked chain each time you want to add a new entry to a bag.

Question 4 Suppose that the ADT bag had an operation that displayed its contents. Write a C++ definition for such a method for the class `LinkedBag`.

```
template<class ItemType>
void LinkedBag<ItemType>::displayBag() const
{
   cout << "The bag contains " << itemCount << " items:" << endl;
   Node<ItemType>* curPtr = headPtr;
   int counter = 0;
   while ((curPtr != nullptr) && (counter < itemCount))
   {
      cout << curPtr->getItem()) << " ";
      curPtr = curPtr->getNext();
      counter++;
   }  // end while
   cout << endl;
}  // end displayBag
```

Question 5 How many assignment operations does the method that you wrote for the previous question require?

If we count `counter++` as 1 assignment, `displayBag` requires 2 x `itemCount` + 2 assignments during its execution.

Question 6 If the pointer variable `curPtr` becomes `nullptr` in the method `getPointerTo`, what value does the method `contains` return when the bag is not empty?

False.

Question 7 Trace the execution of the method `contains` when the bag is empty.

- `contains` calls `getPointerTo`.
- In `getPointerTo`,
  - found = `false`
  - `curPtr` = `headPtr` = `nullptr`
  - The `while` loop exits immediately.
  - The method returns `nullptr`.
- `contains` returns `false`.

Question 8 Revise the definition of the method `getPointerTo` so that the loop is controlled by a counter and the value of `itemCount`.

```
template<class ItemType>
Node<ItemType>* LinkedBag<ItemType>::
                getPointerTo(const ItemType& target) const
{
   bool found = false;
   Node<ItemType>* curPtr = headPtr;
   int counter = 0;
   while (!found && (counter < itemCount))
   {
      if (target == curPtr->getItem())
         found = true;
      else
      {
         counter++;
         curPtr = curPtr->getNext();
      }  // end if
   }  // end while

   return curPtr;
}  // end getPointerTo
```

Question 9 What is a disadvantage of the definition of the method `getPointerTo`, as described in the previous question, when compared to its original definition?

If you make a mistake in the loop so that it counts incorrectly, `curPtr` can be set to `nullptr` within the body of the loop. As a result, the reference `curPtr->getItem()` will cause an exception.

Question 10 Why should the method `getPointerTo` not be made public?

Because it returns a pointer, which is an implementation detail that should be hidden from the client.

Question 11 Given the previous definition of the method `remove`, which entry in a bag can be deleted in the least time? Why?

The first entry. It is in the first node of the linked chain, so `getPointerTo` can locate it sooner than any other entry.

Question 12 Given the previous definition of the method `remove`, which entry in a bag takes the most time to delete? Why?

The last entry. It is in the last node of the linked chain, so `getPointerTo` must traverse the entire chain before locating it.

Question 13 Revise the destructor in the class `LinkedBag` so that it does not call `clear` but instead directly deletes each node of the underlying linked chain.

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

      // Return node to the system
      nodeToDeletePtr->setNext(nullptr);
      delete nodeToDeletePtr;
   }  // end while
   // headPtr is nullptr

   nodeToDeletePtr = nullptr;
   itemCount = 0;
}  // end destructor
```

Question 14 Revise the method `clear` so that it calls a recursive method to deallocate the nodes in the chain.

```
template<class ItemType>
void LinkedBag<ItemType>::clear()
{
   deallocate(headPtr);
   headPtr = nullptr;
   itemCount = 0;
}  // end clear

// Private method: Deallocates all nodes assigned to the bag
template<class ItemType>
void LinkedBag<ItemType>::deallocate(Node<ItemType>* nextNodePtr)
{
   if (nextNodePtr != nullptr)
   {
      Node<ItemType>* nodeToDeletePtr = nextNodePtr;
      nextNodePtr = nextNodePtr->getNext();
      delete nodeToDeletePtr;
      deallocate(nextNodePtr);
   }  // end if
}  // end deallocate
```

Question 15 Revise the program in Listing 4-4 so that it tests first the array-based implementation and then the link-based implementation. Ensure that the program does not have a memory leak.

```
int main()
{
   BagInterface<string>* bagPtr = new ArrayBag<string>();
   cout << "Testing the Array-Based Bag:" << endl;
   cout << "The initial bag is empty." << endl;
   bagTester(bagPtr);
   delete bagPtr;
   cout << endl;

   bagPtr = new LinkedBag<string>();
   cout << "Testing the Link-Based Bag:" << endl;
   cout << "The initial bag is empty." << endl;
   bagTester(bagPtr);
   delete bagPtr;
   bagPtr = nullptr;
   cout << "All done!" << endl;

   return 0;
}  // end main
```

With the definition of `BagInterface` given in Listing 1-1 of Chapter 1, both the program in Listing 4-4 and this revision will cause a warning message regarding the `delete` operator, because `BagInterface` is abstract and has a nonvirtual destructor. Despite the warning, the program will produce the desired output. To avoid the warning, you should add the following statement to the definition of `BagInterface`:

```
virtual ~BagInterface() {}
```

Note that this constructor has an empty method; it is not a pure virtual method.

Chapter 5

Question 1 Consider the language of these character strings: $, cc$d, cccc$dd, cccccc$ddd, and so on. Write a recursive grammar for this language.

<*aString*> = $ | cc<*aString*>d

Question 2 Write the prefix expression that represents the following infix expression:

$(a / b) * c - (d + e) * f$

$- * / a\ b\ c * + d\ e\ f$

Question 3 Write the infix expression that represents the following prefix expression:

$- - a / b + c * d\ e\ f$

$(a - b / (c + d * e)) - f$

Question 4 Is the following string a prefix expression? $+ - / a\ b\ c * + d\ e\ f * g\ h$

No.

Question 5 Write the postfix expression that represents the following infix expression:

$(a * b - c) / d + (e - f)$

$a\ b * c - d / e\ f - +$

Question 6 Trace the method `isPath` with the map in Figure 5-6 for the following requests. Show the recursive calls and the returns from each.

- Fly from *A* to *B*.
- Fly from *A* to *D*.
- Fly from *C* to *G*.

Fly from *A* to *B*: Stack contains A, then A B.
Fly from *A* to *D:* Stack contains A, then A B, then A B D.
Fly from *C* to *G:* Stack contains C, then C D, then C D H, then C D H G.

Question 7 Consider a Four Queens problem, which has the same rules as the Eight Queens problem but uses a 4 x 4 board. Find all solutions to this new problem by applying backtracking by hand.

The queens are in the squares indicated by the following (row, column) pairs:
Solution 1: (2, 1), (4, 2), (1, 3), (3, 4)
Solution 2: (3, 1), (1, 2), (4, 3), (2, 4)

Chapter 6

Question 1 If you push the letters *A*, *B*, *C*, and *D* in order onto a stack of characters and then pop them, in what order will they be deleted from the stack?

D, *C*, *B*, *A*.

Question 2 What do the initially empty stacks `stack1` and `stack2` "look like" after the following sequence of operations?

```
stack1.push(1)               stack1: 1      (stack entries are listed bottom to top)
stack1.push(2)               stack1: 1 2
stack2.push(3)               stack2: 3
stack2.push(4)               stack2: 3 4
stack1.pop()                 stack1: 1
stackTop = stack2.peek()     stackTop : 4
stack1.push(stackTop)        stack1: 1 4
stack1.push(5)               stack1: 1 4 5
stack2.pop()                 stack2: 3
stack2.push(6)               stack2: 3 6
```

Question 3 For each of the following strings, trace the execution of the balanced-braces algorithm and show the contents of the stack at each step.

a. x{{yz}}}

b. {x{y{{z}}}

c. {{{x}}}

a. The stack is empty when the last close brace is encountered. When the loop ends, `balancedSoFar` is false.

b. When the loop ends, the stack contains one open brace and `balancedSoFar` is true.

c. When the loop ends, the stack is empty and `balancedSoFar` is true.

Question 4 Trace the execution of the language-recognition algorithm described in the previous section for each of the following strings, and show the contents of the stack at each step.

a. a$a

b. ab$ab

c. ab$a

d. ab$ba

(*Stack contents are listed from bottom to top.*)

a.
`aStack`: a
`inLanguage`: true
`stackTop`: a
`aStack`:
`while` loop exits because we are at the end of the string.
`inLanguage` is true and `aStack` is empty, so a$a is in the language.
b.
`aStack`: a
`aStack`: a b
`inLanguage`: true
`stackTop`: b
`aStack`: a
`ch`: $
`ch`: a
`stackTop` != `ch`, so `inLanguage` is set to false.
`while` loop exits because `inLanguage` is false.
`inLanguage` is false, so ab$ab is not in the language.
c.
`aStack`: a
`aStack`: a b
`inLanguage`: true
`stackTop`: b
`aStack`: a
`ch`: $
`ch`: a
`stackTop` != `ch`, so `inLanguage` is set to false.
`while` loop exits because `inLanguage` is false.
`inLanguage` is false, so ab$a is not in the language.
d.
`aStack`: a
`aStack`: a b
`inLanguage`: true
`stackTop`: b
`aStack`: a
`ch`: $
`ch`: b
`stackTop` == `ch`
`stackTop`: a
`aStack`:
`ch`: a
`stackTop` == `ch`
`while` loop exits because we are at the end of the string.
`inLanguage` is true and `aStack` is empty, so ab$ba is in the language.

Question 5 Evaluate the postfix expression $a\ b - c\ +$. Assume the following values for the identifiers: $a = 7$, $b = 3$, and $c = -2$. Show the status of the stack after each step.

(*Stack contents are listed from bottom to top.*)
Let `aStack` be the stack.
`ch`: a
`aStack`: 7
`ch`: b
`aStack`: 7 3
`ch`: –
`operand2`: 3
`aStack`: 7
`operand1`: 7
`aStack`:
`result`: 7 – 3 = 4
`aStack`: 4
`ch`: c
`aStack`: 4 –2
`ch`: +
`operand2`: –2
`aStack`: 4
`operand1`: 4
`aStack`:
`result`: 4 + (–2) = 2
`aStack`: 2
The value of the expression is 2.

Question 6 Convert the infix expression $a / b * c$ to postfix form. Be sure to account for left-to-right association. Show the status of the stack after each step.

`ch`: a
`postfix`: a
`ch`: /
`aStack`: /
`ch`: b
`postfix`: a b
`ch`: *
`postfix`: a b /
`aStack`:
`aStack`: *
`ch`: c
`postfix`: a b / c
`postfix`: a b / c *
`aStack`:

Question 7 Explain the significance of the precedence tests in the infix-to-postfix conversion algorithm. Why is a ≥ test used rather than a > test?

The precedence tests control association. The ≥ test enables left-to-right association when operators have the same precedence.

Question 8 Trace the method `isPath` with the map in Figure 6-10 for the following requests. Show the state of the stack after each step.

a. Fly from A to B.

b. Fly from A to D.

c. Fly from C to G.

a. (*Stack contents are listed from bottom to top.*)
Push origin A onto stack and mark A as visited.
`aStack`: A　　`topCity`: A
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: B　　Push B onto stack and mark B as visited.
`aStack`: A B　　`topCity`: B
`aStack` is not empty and `topCity` is the destination, so the `while` loop exits.
`aStack` is not empty, so `isPath` returns true.

b.
Push origin A onto stack and mark A as visited.
`aStack`: A　　`topCity`: A
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: B　　Push B onto stack and mark B as visited.
`aStack`: A B　　`topCity`: B
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: D　　Push D onto stack and mark D as visited.
`aStack`: A B D　　`topCity`: D
`aStack` is not empty and `topCity` is the destination, so the `while` loop exits.
`aStack` is not empty, so `isPath` returns true.

c.
Push origin C onto stack and mark C as visited.
`aStack`: C　　`topCity`: C
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: D　　Push D onto stack and mark D as visited.
`aStack`: C D　　`topCity`: D
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: E　　Push E onto stack and mark E as visited.
`aStack`: C D E　　`topCity`: E
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: I　　Push I onto stack and mark I as visited.
`aStack`: C D E I　　`topCity`: I
`aStack` is not empty and `topCity` is not the destination, so the `while` loop exits.

`nextCity`: `NO_CITY`
Backtrack by popping the stack.
`aStack`: C D E   `topCity`: E
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: `NO_CITY`
Backtrack by popping the stack.
`aStack`: C D   `topCity`: D
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: F   Push F onto stack and mark F as visited.
`aStack`: C D F   `topCity`: F
`aStack` is not empty and `topCity` is not the destination, so the `while` loop continues.
`nextCity`: G   Push G onto stack and mark G as visited.
`aStack`: C D F G   `topCity`: G
`aStack` is not empty and `topCity` is the destination, so the `while` loop exits.
`aStack` is not empty, so `isPath` returns true.

Chapter 7

Question 1 In Chapter 6, the algorithms that appear in Section 6.2 involve strings. Under what conditions would you choose an array-based implementation for the stack in these algorithms?

You would use an array-based implementation if you know the maximum string length in advance and you know that the average string length is not much shorter than the maximum length.

Question 2 Describe the changes to the previous stack implementation that are necessary to replace the fixed-size array with a resizable array.

In the header file for `ArrayStack`, change the declaration of `items` and declare a new data member `maxItems`:

```
ItemType* items = new ItemType[DEFAULT_CAPACITY]; // Array of stack items
int maxItems;                                     // Max capacity of the stack
```

Also, change the name of `MAX_STACK` to `DEFAULT_CAPACITY` to reflect its change in meaning. In the implementation file, add an initializer to the definition of the constructor that initializes `maxItems` to `DEFAULT_CAPACITY`. Thus, the constructor appears as follows:

```
template<class ItemType>
ArrayStack<ItemType>::ArrayStack() : top(-1), maxItems(DEFAULT_CAPACITY)
{
}  // end default constructor
```

Then change the definition of the `push` method so that when the array—and hence the stack—becomes full, `push` doubles the capacity of the array instead of failing to add the item and returning false. Since the stack is never full, the following version of the `push` method always returns true:

```
template<class ItemType>
bool ArrayBag<ItemType>::add(const ItemType& newEntry)
{
   bool hasRoomToAdd = (itemCount < maxItems);
   if (!hasRoomToAdd)
   {
      ItemType* oldArray = items;
      items = new ItemType[2 * maxItems];
      for (int index = 0; index < maxItems; index++)
         items[index] = oldArray[index];
      delete [ ] oldArray;
      maxItems = 2 * maxItems;
   }  // end if
   // We can always add the item
   items[itemCount] = newEntry;
```

```
      itemCount++;
      return true;
   }  // end push
```

Question 3 In Chapter 6, the algorithms that appear in Section 6.2 involve strings. Under what conditions would you choose a link-based implementation?

You would use a link-based implementation if you could not predict the maximum string length in advance or if the maximum string length is much larger than the average string length. For example, if the maximum string length is 300 but the average string length is 30, a link-based implementation would use less storage on average than an array-based implementation.

Question 4 Define the exception class `MemoryAllocationException` and then revise the definition of the method `push` in the class `LinkedStack` so that it throws this exception if it cannot allocate a new node.

```
#include <stdexcept>
#include <string>
using namespace std;
class MemoryAllocationException: public exception
{
public:
   MemoryAllocationException(const string& message = "") :
                exception("The operation push cannot allocate memory: " +
                          message.c_str())
   {
   }  // end constructor
}; // end MemoryAllocationException

template<class ItemType>
bool LinkedStack<ItemType>::push(const ItemType& newItem)
                             throw(MemoryAllocationException )
{
   try
   {
      Node<ItemType>* newTopNode = new Node<ItemType>(newItem, topPtr);
      topPtr = newTopNode;
   }
   catch (bad_alloc e)
   {
      throw MemoryAllocationException("MemoryAllocationException : " +
                                      "push() cannot allocate memory.");
   }  // end try/catch
   return true;
}  // end push
```

Chapter 8

Question 1 The specifications of the ADT list do not mention the case in which two or more items have the same value. Are these specifications sufficient to cover this case, or must they be revised?

Duplicate values in an ADT list are permissible and do not affect its specifications, because its operations are by position.

Question 2 Write specifications for a list whose operations `insert`, `remove`, `getEntry`, and `setEntry` always act at the end of the list.

Specify `isEmpty` and `getLength` as you would for the ADT list given in this chapter.

`+insert(newEntry: ItemType): boolean`
Task: Inserts an entry at the end of this list.
Input: `newEntry` is the new entry.
Output: True if the insertion is successful; otherwise false.

`+remove(): boolean`
Task: Removes the entry at the end of a list.
Output: True if the removal is successful; otherwise false.

`+getEntry(): ItemType`
Task: Retrieves the item at the end of a list.
Output: The desired entry.

`+setEntry(newEntry: ItemType): void`
Task: Replaces the entry at the end of a list with the given entry.
Input: `newEntry` is the replacement entry.
Output: None.

Question 3 Write a pseudocode function `swap(aList, i, j)` that interchanges the items currently in positions *i* and *j* of a list. Define the function in terms of the ADT list operations, so that it is independent of any particular implementation of the list. Assume that the list, in fact, has items at positions *i* and *j*. What impact does this assumption have on your solution?

```
// Swaps the ith and jth items in the list aList.
swap(aList: List, i: integer, j: integer): void

   // Copy the ith and jth items.
   ithItem = aList.getEntry(i)
   jthItem = aList.retrieve(j)

   // Replace the ith item with the jth item.
   aList.remove(i)
   aList.insert(i, jthItem)

   // Replace the jth item with the ith item.
   aList.remove(j)
   aList.insert(j, ithItem)
```

Notice that the order of operations is important because when you remove an entry, the remaining entries are renumbered. If you did not assume that the entries at positions *i* and *j* existed, you would have to check that each operation was successful.

Question 4 What grocery list results from the following sequence of ADT list operations?

`aList` = *a new empty list*

```
aList.insert(1, "butter")
aList.insert(1, "eggs")
aList.insert(1, "milk")
```

milk
eggs
butter

Question 5 Suppose that `myList` is a list that contains the five objects `a b c d e`.

a. What does `myList` contain after executing `myList.insert(5, w)`?

b. Starting with the original five entries, what does `myList` contain after executing `myList.insert(6, w)`?

c. Which of the operations in parts *a* and *b* of this question require entries in the array to shift?

a. `a b c d w e`

b. `a b c d e w`

c. The insertion in part *a*.

Chapter 9

Question 1 Given a nonempty list that is an instance of `ArrayList`, at what position does an insertion of a new entry require the fewest operations? Explain.

For a list of n entries, an insertion at position $n + 1$ does not require any other entry to move. Therefore, as long as the array `items` can accommodate an additional entry, an insertion at position $n + 1$ requires the fewest operations.

Question 2 Describe an implementation of the method `insert` for `ArrayList` that places the entry in position 1 in the last element of the array, the entry in position 2 in the next-to-last element, and so on.

If the array `items` has `maxItems` elements, `items[maxItems – 1]` contains the entry at position 1, `items[maxItems – 2]` contains the entry at position 2, and `items[maxItems – n]` contains the entry at position n. Suppose that this entry at position n is the last one in the list. That is, the list contains n items. To insert a new entry at position k, where $k < n$, you create room for the new entry in the array `items` by shifting the entries at lower-numbered indices (higher-numbered positions) toward the beginning of the array. That is, prior to adding a new entry at position k, you copy the entry in `items[i]` to `items[i – 1]` for values of `i` ranging from `maxItems – n` up to `maxItems – k` .

Question 3 How does the original version of `insert` given previously compare with the one described in Question 2 with respect to the number of operations required?

Except for more involved index computations, there is no difference in the effort required for these two approaches.

Question 4 Although the method `remove` cannot remove an entry from an empty list, it does not explicitly check for one. How does this method avoid an attempted deletion from an empty list?

The boolean expression `(position >= 1) && (position <= itemCount)` will be false when `itemCount` is zero, regardless of the value of `position`. Therefore, the boolean variable `ableToRemove` is false, and so the method does not do anything more than return false.

Question 5 Given a nonempty list that is an instance of `LinkedList`, at what position does an insertion of a new entry require the fewest operations? Explain.

An insertion at position 1 of the list is implemented as an insertion at the beginning of the linked chain. This insertion can be accomplished without a search of the chain for the desired position. Therefore, it requires the fewest operations.

Question 6 In the previous method `insert`, the second `if` statement tests the value of `newPosition`. Should the boolean expression it tests be `isEmpty() || (newPosition == 1)`? Explain.

Although you could test for an empty list, such a test is not necessary. When a list is empty, `itemCount` is zero. The only acceptable position for an insertion is 1, because the boolean expression `(newPosition >= 1) && (newPosition <= itemCount + 1)` will be true. Therefore, the body of the first `if` statement will execute, and the insertion into the empty list will occur correctly.

Question 7 How does the `insert` method enforce the precondition of `getNodeAt`?

`getNodeAt`'s precondition is `(position >= 1) && (position <= itemCount)`. Before `insert` invokes `getNodeAt`, the boolean expression `(newPosition >= 1) && (newPosition <= itemCount + 1)` must be true. The call `getNodeAt(newPosition - 1)` occurs only if `newPosition` > 1 and sets `position` to `newPosition` – 1. Therefore, `position >= 1` is true. Further, since `newPosition <= itemCount + 1`, `newPosition - 1 <= itemCount`, which implies that `position <= itemCount`.

Question 8 The link-based implementation of the method `clear` contains the following loop:

```
while (!isEmpty())
  remove(1);
```

a. Can you correctly replace the loop with

```
for (int position = getLength(); position >= 1; position--)
    remove(1);
```

b. Does your answer to part *a* differ if you replace `remove(1)` with `remove(position)`?

c. Do your answers to parts *a* and *b* differ if you replace the `for` statement with

```
for (int position = 1; position <= getLength(); position++)
```

a. Yes.

b. This replacement is still correct. We remove the last entry from the list without renumbering the remaining entries.

c. Whether this `for` statement is correct depends on how the compiler makes the comparison. If `position` is compared to the original value of `getLength()`, the loop will cycle the correct number of times. Repeatedly executing `remove(1)` will work. However, repeatedly executing `remove(position)` will not work because entries are renumbered after each iteration. For example, after the first entry is removed, the second entry becomes the first entry, but `position` is set to 2. This new first entry is never removed. If `position` is compared to the current value of `getLength()` at each iteration, the loop will exit before the entire list is emptied.

Question 9 Revise the destructor in the class `LinkedList` so that it directly deletes each node of the underlying linked chain without calling either `clear` or `remove`.

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

      // Return node to the system
      nodeToDeletePtr->setNext(nullptr);
      delete nodeToDeletePtr;
   }  // end while
   // headPtr is nullptr

   nodeToDeletePtr = nullptr;
   itemCount = 0;
}  // end destructor
```

Question 10 Using recursion, revise the destructor in the class `LinkedList` so that it deletes each node of the underlying linked chain.

```
template<class ItemType>
LinkedList<ItemType>::~LinkedList()
{
   deallocate(headPtr);
   headPtr = nullptr;
   itemCount = 0;
}  // end destructor

// Private method: Deallocates all nodes assigned to the bag
template<class ItemType>
void LinkedList<ItemType>::deallocate(Node<ItemType>* nextNodePtr)
{
   if (nextNodePtr != nullptr)
   {
      Node<ItemType>* nodeToDeletePtr = nextNodePtr;
      nextNodePtr = nextNodePtr->getNext();
      delete nodeToDeletePtr;
      deallocate(nextNodePtr);
   }  // end if
}  // end deallocate
```

Chapter 10

Question 1 How many comparisons of array items do the following loops contain?

```
for (j = 1; j <= n - 1; j++)
{
   i = j + 1;
   do
   {
      if (theArray[i] < theArray[j])
         swap(theArray[i], theArray[j]);
      i++;
   } while (i <= n);
}  // end for
```

$(n - 1) + (n - 2) + . . . + 1 = n \times (n - 1) / 2$

Question 2 Repeat Question 1, replacing the statement `i = j + 1` with `i = j`.

$n + (n - 1) + . . . + 2 = n \times (n + 1) / 2 - 1$

Question 3 What order is an algorithm that has as a growth-rate function

a. $8 \times n^3 - 9 \times n$ **b.** $7 \times \log_2 n + 20$ **c.** $7 \times \log_2 n + n$

a. $O(n^3)$; **b.** $O(\log n)$; **c.** $O(n)$

Question 4 Consider a sequential search of n data items.

a. If the data items are sorted into ascending order, how can you determine that your desired item is not in the data collection without always making n comparisons?

b. What is the order of the sequential search algorithm when the desired item is not in the data collection? Do this for both sorted and unsorted data, and consider the best, average, and worst cases.

c. Show that if the sequential search algorithm finds the desired item in the data collection, the algorithm's order does not depend upon whether or not the data items are sorted.

a. You can stop searching as soon as the target is less than an entry, because you will have passed the point where the target would have occurred if it was in the data collection.

b. Sorted data using the scheme just described in the answer to part *a*: Best case: $O(1)$; average case: $O(n)$; worst case: $O(n)$.

Unsorted data: $O(n)$ in all cases.

Chapter 11

Question 1 Trace the selection sort as it sorts the following array into ascending order:

20 80 40 25 60 30

At each pass, the selected element is bold; the sorted elements are blue.

20	**80**	40	25	60	30
20	30	40	25	**60**	**80**
20	30	**40**	25	**60**	**80**
20	**30**	25	**40**	**60**	**80**
20	**25**	**30**	**40**	**60**	**80**
20	**25**	**30**	**40**	**60**	**80**
20	**25**	**30**	**40**	**60**	**80**

Question 2 Repeat the previous question, but instead sort the array into descending order.

Find the smallest instead of the largest entry at each pass.

20	80	40	25	60	30
30	80	40	**25**	60	**20**
30	80	40	60	**25**	**20**
60	80	**40**	**30**	**25**	**20**
60	80	**40**	**30**	**25**	**20**
80	**60**	**40**	**30**	**25**	**20**
80	**60**	**40**	**30**	**25**	**20**

Question 3 Trace the bubble sort as it sorts the following array into ascending order:

25 30 20 80 40 60.

Pass 1:

25	**30**	20	80	40	60
25	**30**	**20**	80	40	60
25	20	**30**	**80**	40	60
25	20	30	**80**	**40**	60
25	20	30	40	**80**	**60**
25	20	30	40	60	**80**

Pass 2:

25	**20**	30	40	60	**80**
20	**25**	**30**	40	60	**80**
20	25	**30**	**40**	60	**80**
20	25	30	**40**	**60**	**80**
20	25	30	40	**60**	**80**

Pass 3: There are no exchanges, so the algorithm will terminate.

Question 4 Repeat the previous question, but instead sort the array into descending order.

We can move either the smallest entries to the end of the array or the largest entries to the beginning of the array. The following trace uses the former strategy:

Pass 1:

25	**30**	20	80	40	60
30	**25**	**20**	80	40	60
30	25	**20**	**80**	40	60
30	25	80	**20**	**40**	60
30	25	80	40	**20**	**60**
30	25	80	40	60	**20**

Pass 2:

30	**25**	80	40	60	**20**
30	**25**	**80**	40	60	**20**
30	80	**25**	**40**	60	**20**
30	80	40	**25**	**60**	**20**
30	80	40	60	**25**	**20**

Pass 3:

30	**80**	40	60	**25**	**20**
80	**30**	**40**	60	**25**	**20**
80	40	**30**	**60**	**25**	**20**
80	40	60	**30**	**25**	**20**

Pass 4:

80	**40**	60	**30**	**25**	**20**
80	**40**	**60**	**30**	**25**	**20**
80	60	**40**	**30**	**25**	**20**

Pass 5: There are no exchanges, so the algorithm will terminate.

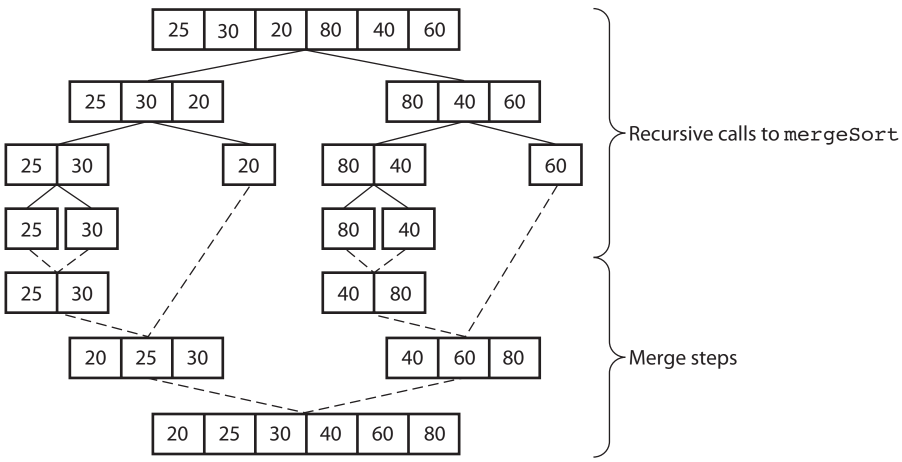
Question 5 Trace the insertion sort as it sorts the array in Checkpoint Question 3 into ascending order.

25	**30**	20	80	40	60
25	**30**	**20**	80	40	60
25	**20**	**30**	**80**	40	60
25	**20**	**30**	**80**	**40**	60
25	**20**	**30**	**40**	**80**	**60**
25	**20**	**30**	**40**	**60**	**80**

Question 6 Repeat the previous question, but instead sort the array into descending order.

25	**30**	20	80	40	60
30	**25**	**20**	80	40	60
30	**25**	**20**	**80**	40	60
80	**30**	**25**	**20**	**40**	60
80	**40**	**30**	**25**	**20**	**60**
80	**60**	**40**	**30**	**25**	**20**

Question 7 By drawing a diagram like the one shown in Figure 11-6, trace the merge sort as it sorts the following array into ascending order: 25 30 20 80 40 60.



Question 8 Show that the merge sort algorithm satisfies the four criteria of recursion that Chapter 2 describes.

1. Merge sort sorts an array by using a merge sort to sort each half of the array.
2. Sorting half of an array is a smaller problem than sorting the entire array.
3. An array of one element is the base case.
4. By halving an array and repeatedly halving the halves, you must reach array segments of one element each—that is, the base case.

Question 9 Trace the quick sort's partitioning algorithm as it partitions the following array:

38 16 40 39 12 27

38	16	**40**	39	12	**27**	Identify first, middle, and last entries
27	16	**38**	39	12	**40**	Sort first, middle, and last entries; choose 38 as pivot
27	16	12	39	**38**	40	Position pivot
27	16	12	39	**38**	40	`indexFromLeft` = 3, `indexFromRight` = 2
27	16	12	**38**	39	40	Interchange the pivot and `theArray[3]`

Question 10 Suppose that you sort a large array of integers by using a merge sort. Next you use a binary search to determine whether a given integer occurs in the array. Finally, you display all of the integers in the sorted array.

a. Which algorithm is faster, in general: the merge sort or the binary search? Explain in terms of Big O notation.

b. Which algorithm is faster, in general: the binary search or displaying the integers? Explain in terms of Big O notation.

a. A binary search is O(log *n*), so it is faster than a merge sort, which is O(*n* log *n*).

b. A binary search is O(log *n*), so it is faster than displaying the array, which is O(*n*).

Question 11 Trace the radix sort as it sorts the following array into ascending order:

3812 1600 4012 3934 1234 2724 3333 5432

3812 1600 4012 3934 1234 2724 3333 5432	The original array
(160**0**) (381**2**, 401**2**, 543**2**) (333**3**) (393**4**, 123**4**, 272**4**)	Group by rightmost digit
1600 3812 4012 5432 3333 3934 1234 2724	Combine
(16**0**0) (38**1**2, 40**1**2) (27**2**4) (54**3**2, 33**3**3, 39**3**4, 12**3**4)	Group by third digit
1600 3812 4012 2724 5432 3333 3934 1234	Combine
(4**0**12) (1**2**34) (3**3**33) (5**4**32) (1**6**00) (2**7**24) (3**8**12) (3**9**34)	Group by second digit
4012 1234 3333 5432 1600 2724 3812 3934	Combine
(**1**234, **1**600) (**2**724) (**3**333, **3**812, **3**934) (**4**012) (**5**432)	Group by first digit
1234 1600 2724 3333 3812 3934 4012 5432	Combine; the array is now sorted

Chapter 12

Question 1 The specifications of the ADT sorted list do not mention the case in which two or more items have the same value. Are these specifications sufficient to cover this case, or must they be revised?

Duplicate values in an ADT sorted list are permissible and do not affect its specifications. Both `removeSorted` and `getPosition` deal with the first or only occurrences of the specified value; `remove` and `getEntry` are by position. Although `insertSorted` does not specify where the insertion will occur, duplicate values will be at adjacent positions.

Question 2 Write specifications for the operation `insertSorted` when the sorted list must not contain duplicate entries.

Task: Inserts an entry into this sorted list in its proper order so that the list remains sorted.
Input: `newEntry` is the new entry and is not in the sorted list already.
Output: Throws an exception (`PrecondViolatedExcep`) if `newEntry` already is in the sorted list.

Question 3 Suppose that `wordListPtr` points to an unsorted list of words. Using the operations of the ADT list and the ADT sorted list, create a sorted list of these words.

```
SortedListInterface<string>* sortedWordListPtr =
                                   new SortedList<string>();
int numberOfWords = wordListPtr->getLength();
for (int position = 1; position <= numberOfWords; position++)
   sortedWordListPtr->insertSorted(wordListPtr->getEntry(position));
```

Question 4 Assuming that the sorted list you created in the previous question is not empty, write C++ statements that

a. Display the last entry in the sorted list.

b. Add the sorted list's first entry to the sorted list again.

a. `cout << sortedWordListPtr->getEntry(numberOfWords) << endl;`

b. `sortedWordListPtr->insertSorted(sortedWordListPtr->getEntry(1));`

Question 5 In the `while` statement of the method `getNodeBefore`, how important is the order of the two boolean expressions that the operator `&&` joins? Explain.

The order of the boolean expressions is important. If the given entry—`anEntry`—is larger than any of the current entries in the sorted list, `curPtr` will become `nullptr` after it is compared to the list's last entry. The `while` statement will then evaluate the expression
`(curPtr != nullptr) && (anEntry > curPtr->getItem())`
Because `curPtr` is `nullptr`, `curPtr != nullptr` will be false and `curPtr->getItem()` will not execute due to short-circuit evaluation. If the boolean expression had been
`(anEntry > curPtr->getItem()) && (curPtr != nullptr)`
`curPtr->getItem()` would execute and since `curPtr` is `nullptr`, an exception would occur.

Question 6 What does `getNodeBefore` return if the sorted list is empty? How can you use this fact to simplify the implementation of the method `insertSorted` given previously?

If the sorted list is empty, `headPtr is nullptr`. The `while` loop exits immediately, and the method returns `nullptr`—the current value of `prevPtr`. Because of this fact, you can replace the `if` statement in the method `insertSorted` with

```
if (prevPtr == nullptr)
```

Question 7 Suppose that you use the previous method `insertSorted` to add an entry to a sorted list. If the entry is already in the list, where in the list will the method insert it? Before the first occurrence of the entry, after the first occurrence of the entry, after the last occurrence of the entry, or somewhere else?

Before the first occurrence of the entry.

Question 8 What would be the answer to the previous question if you changed `>` to `>=` in the `while` statement of the method `getNodeBefore`?

After the last occurrence of the entry.

Question 9 Define the method `getPosition` for the class `SortedListHasA`.

```
template<class ItemType>
int SortedListHasA<ItemType>::getPosition(const ItemType& anEntry) const
{
   int position = 1;
   int length = listPtr->getLength();
   while ( (position <= length) &&
           (anEntry > listPtr->getEntry(position)) )
      position++;
   if ( (position > length) || (anEntry != listPtr->getEntry(position)) )
      position = -position;
   return position;
}  // end getPosition
```

Question 10 Repeat Checkpoint Question 7 using the method `insertSorted` of the class `SortedListHasA`.

Before the first occurrence of the entry, assuming that `getPosition` is defined as in Checkpoint Question 9.

Question 11 Can a client of `SortedListHasA` invoke the method `insert` of the ADT list? Explain.

No. `SortedListHasA` is derived from `SortedListInterface`, which does not declare `insert`. Moreover, the underlying list in `SortedListHasA` is private. A client of `SortedListHasA` has no access to this list.

Question 12 Define the method `removeSorted` for the class `SortedListHasA`.

```
template<class ItemType>
bool SortedListHasA<ItemType>::removeSorted(const ItemType& anEntry)
{
   bool ableToDelete = false;
   if (!isEmpty())
   {
      int position = getPosition(anEntry);
      ableToDelete = position > 0;
      if (ableToDelete)
      {
         ableToDelete = listPtr->remove(position);
      }  // end if
   }  // end if

   return ableToDelete;
}  // end removeSorted
```

Question 13 Suppose that instead of using `LinkedList` in the implementation of `SortedListHasA`, you used `ArrayList`. What Big O would describe the performance of the method `getPosition`?

The method `getEntry` is always an O(1) operation. The loop in `getPosition` is therefore O(n) in the worst case, and so `getPosition` is O(n) when the list has an array-based implementation.

Question 14 Give an advantage and a disadvantage of using containment in the implementation of the class `SortedListHasA`.

Advantage: It is easier to write because the underlying list does most of the work and has been debugged.
Disadvantage: It is less efficient than an implementation that does not call and is not restricted to the methods of the ADT list.

Question 15 What would have happened if you preceded the call to `insert` in the method `insertSorted` by `this->` instead of `LinkedList<ItemType>::`? Explain.

The call `this->insert(newPosition, newEntry)` would call the overriding version of `insert`, which `SortedListIsA` defines, instead of `LinkedList`'s method `insert`.

Chapter 13

Question 1 If you add the letters *A*, *B*, *C*, and *D* in sequence to a queue of characters and then remove them, in what order will they leave the queue?

A, B, C, D.

Question 2 What do the initially empty queues `queue1` and `queue2` "look like" after the following sequence of operations? Compare these results with Checkpoint Question 2 in Chapter 6.

(Queue entries are listed front to back)

```
queue1.enqueue(1)                queue1: 1
queue1.enqueue(2)                queue1: 1  2
queue2.enqueue(3)                queue2: 3
queue2.enqueue(4)                queue2: 3  4
queue1.dequeue()                 queue1: 2
queueFront = queue2.peekFront()  queueFront = 3
queue1.enqueue(queueFront)       queue1: 2  3
queue1.enqueue(5)                queue1: 2  3  5
queue2.dequeue()                 queue2: 4
queue2.enqueue(6)                queue2: 4  6
```

Question 3 Trace the palindrome-recognition algorithm described in this section for each of the following strings of characters:

a. abcda

b. radar

a. When the `for` loop ends, the stack and queue are as follows:

Stack: a b c d a ← top

Queue: a b c d a ← back

The *a* at the top of the stack matches the *a* at the front of the queue. After removing the *a* from both containers, the *d* at the top of the stack does not match the *b* at the front of the queue, so the string is not a palindrome.

b. When the `for` loop ends, the stack and queue are as follows:

Stack: r a d a r ← top

Queue: r a d a r ← back

The letters that you remove from the stack and the queue are the same, so the string is a palindrome.

Question 4 Improve the palindrome-recognition algorithm described in this section by adding the first `length / 2` characters to the queue and then pushing the remaining characters onto the stack.

```
isPalindrome(someString: string): boolean
   // Create an empty queue and an empty stack
   aQueue = a new empty queue
   aStack = a new empty stack

   // Add the first half of the string to the queue
   length = length of someString
   halfLength = length / 2
   for (i = 1 through halfLength)
   {
      nextChar = ith character of someString
      aQueue.enqueue(nextChar)
   }
   // Add the rest of the string to the stack
   for (i = halfLength + 1 through length)
   {
      nextChar = ith character of someString
      aStack.push(nextChar)
   }

   // Compare the queue characters with the stack characters
   charactersAreEqual = true
   while (aQueue is not empty and charactersAreEqual)
   {
      queueFront = aQueue.peekFront()
      stackTop = aStack.peek()
      if (queueFront equals stackTop)
      {
         aQueue.dequeue()
         aStack.pop()
      }
      else
         charactersAreEqual = false
   }
   return charactersAreEqual
```

Question 5 In the bank simulation problem, why is it impractical to read the entire input file and create a list of all the arrival and departure events before the simulation begins?

You cannot generate a departure event for a given arrival event independently of other events. So to read the file of arrival events and generate departure events, you would need to perform the same computations that the simulation performs.

Question 6 Complete the hand trace of the bank-line simulation that Figure 13-8 began. Show the state of the queue and the event list at each step.

The only missing portion is to indicate that at time 35 the last customer leaves the bank, the bank queue is empty, and the event list is empty.

Question 7 For each of the following situations, which of these ADTs (1 through 6) would be most appropriate? (1) a queue; (2) a stack; (3) a list; (4) a sorted list; (5) a priority queue; (6) none of these

a. The customers at a deli counter who take numbers to mark their turn **1**

b. An alphabetic list of names **4**

c. Integers that need to be sorted **3**

d. The boxes in a box trace of a recursive function **2**

e. A grocery list ordered by the occurrence of the items in the store **3**

f. The items on a cash register tape **1 or 3**

g. A word processor that allows you to correct typing errors by using the Backspace key **2**

h. A program that uses backtracking **2**

i. A list of ideas in chronological order **1, 3, or 5**

j. Airplanes that stack above a busy airport, waiting to land **1 or 5**

k. People who are put on hold when they call for customer service **1**

l. An employer who fires the most recently hired person **2**

Chapter 14

Question 1 Why is a tail pointer desirable when you use a chain of linked nodes to implement a queue?

Because additions to a queue occur at its end, you would want to add nodes to the end of the linked chain. The tail pointer enables you to make such addtions efficiently. With only a head pointer, you would have to traverse the entire chain each time you wanted to add to the end of the chain.

Question 2 If you use a circular chain that has only a tail pointer, as Figure 14-3 illustrates, how do you access the data in the first node?

First, you get a pointer to the first node by executing the following statement:

```
Node<ItemType>* frontPtr = backPtr->getNext();
```

Then the data in the first node is `frontPtr->getItem()`.

Question 3 If the ADT queue had a method `clear` that removed all entries from a queue, what would its definition be in the previous link-based implementation?

A method `clear` could repeatedly call `dequeue` until the queue is empty. One such definition follows:

```
template<class ItemType>
void LinkedQueue<ItemType>::clear()
{
   while (!isEmpty())
      dequeue();
}  // end clear
```

However, since `dequeue` is a boolean-valued method, you could replace the previous `while` loop with the following one:

```
while (dequeue())
{}
```

Question 4 Suppose that we change the naive array-based implementation of a queue pictured in Figure 14-7 so that the back of the queue is in `items[0]`. Although repeated removals from the front would no longer cause rightward drift, what other problem would this implementation cause?

Additions to the queue, which will be at its back, will require that all entries curently in the queue shift one array location toward the array's end.

Question 5 If the ADT queue had a method `clear` that removed all entries from a queue, what would its definition be in the previous array-based implementation?

```
template<class ItemType>
void ArrayQueue<ItemType>::clear()
{
   front = 0;
   back = MAX_QUQUE - 1;
   count = 0;
}  // end clear
```

Question 6 Define the method `peek` for the sorted list implementation of the ADT priority queue.

```
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
```

Chapter 15

Question 1 What kind of tree is the tree in Figure 15-1a?

A general tree.

Question 2 Repeat the previous question, but use the tree in Figure 15-3c instead.

A binary tree.

Question 3 Given the tree in Figure 15-3c, what node or nodes are

a. Ancestors of *b*?

b. Descendants of `x`?

c. Leaves?

a. – and `x`.

b. – , c, a, b.

c. a, b, c.

Question 4 Given the tree in Figure 15-4, what node or nodes are

a. The tree's root?

b. Parents?

c. Children of the parents in part *b* of this question?

d. Siblings?

a. Jane.

b. Jane, Bob, Tom.

c. Bob and Tom are children of Jane. Alan and Ellen are children of Bob. Nancy and Wendy are children of Tom.

d. Bob and Tom are siblings, Alan and Ellen are siblings, Nancy and Wendy are siblings.

Question 5 What are the levels of all nodes in the trees in parts *b*, *c*, and *d* of Figure 15-5?

Figure15-5b: Level 1: A; Level 2: B, C; Level 3: D, E; Level 4: F; Level 5: G.
Figure15-5c: Level 1: A; Level 2: B; Level 3: C; Level 4: D; Level 5: E; Level 6: F; Level 7: G.
Figure15-5d: Level 1: A; Level 2: B; Level 3: C; Level 4: D; Level 5: E; Level 6: F; Level 7: G.

Question 6 What is the height of the tree in Figure 15-4?

3.

Question 7 Consider the binary trees in Figure 15-8.

a. Which are complete?

b. Which are full?

c. Which are balanced?

d. Which have minimum height?

e. Which have maximum height?

a. b, c, d, e.

b. e.

c. b, c, d, e.

d. b, c, d, e.

e. a.

Question 8 What are the preorder, inorder, and postorder traversals of the binary trees in parts *a*, *b*, and *c* of Figure 15-5?

Figure 15-5a:	Preorder:	A, B, D, E, C, F, G;
	Inorder:	D, B, E, A, F, C, G;
	Postorder:	D, E, B, F, G, C, A.
Figure 15-5b:	Preorder:	A, B, D, F, G, E, C;
	Inorder:	G, F, D, B, E, A, C;
	Postorder:	G, F, D, E, B, C, A.
Figure 15-5c:	Preorder:	A, B, C, D, E, F, G;
	Inorder:	A, C, E, G, F, D, B;
	Postorder:	G, F, E, D, C, B, A.

Question 9 Show that each tree in Figures 15-13 and 15-14 is a binary search tree.

Figure 15-13:	Jane is larger than all of the data in its left subtree and smaller than all of the data in its right subtree. Bob is larger than Allan in its left subtree and smaller than Elisa in its right subtree. Tom is larger than Nancy in its left subtree and smaller than Wendy in its right subtree.
Figure 15-14a:	Jane is larger than all of the data in its left subtree and smaller than all of the data in its right subtree. Bob is larger than Allan in its left subtree and smaller than Elisa in its right subtree. Nancy is smaller than all of the data in its right subtree. Tom is smaller than Wendy in its right subtree.

Figure 15-14b:	Alan is smaller than all of the data in its right subtree. Bob is smaller than all of the data in its right subtree. Elisa is smaller than all of the data in its right subtree. Jane is smaller than all of the data in its right subtree. Nancy is smaller than all of the data in its right subtree. Tom is smaller than Wendy in its right subtree.
Figure 15-14c:	Tom is larger than all of the data in its left subtree and smaller than Wendy in its right subtree. Jane is larger than all of the data in its left subtree and smaller than Nancy in its right subtree. Bob is larger than Allan in its left subtree and smaller than Elisa in its right subtree.

Question 10 Show that the inorder traversals of each binary search tree in Figures 15-13 and 15-14 are the same.

Figure 15-13: Beginning at Jane, the root, the traversal takes the following steps:
inorder(Jane), inorder(Bob), inorder(Alan), inorder(<empty>), return, **Visit Alan**, inorder(<empty>), return, **Visit Bob**, inorder(Elisa), inorder(<empty>), return, **Visit Elisa**, inorder(<empty>), return, return, return, **Visit Jane**, inorder(Tom), inorder(Nancy), inorder(<empty>), return, **Visit Nancy**, inorder(<empty>), return, return, **Visit Tom**, inorder(Wendy), inorder(<empty>), return, **Visit Wendy**, inorder(<empty>), return, return, return, return.

Figure 15-14a: Beginning at Jane, the root, the traversal takes the following steps:
inorder(Jane), inorder(Bob), inorder(Alan), inorder(<empty>), return, **Visit Alan**, inorder(<empty>), return, **Visit Bob**, inorder(Elisa), inorder(<empty>), return, **Visit Elisa**, inorder(<empty>), return, return, return, **Visit Jane**, inorder(Nancy), inorder(<empty>), return, **Visit Nancy**, inorder(Tom), inorder(<empty>), return, **Visit Tom**, inorder(Wendy), inorder(<empty>), return, **Visit Wendy**, inorder(<empty>), return, return, return, return, return.
The inorder traversals of Figures 15-14b and 15-14c are like the previous ones.

Question 11 What are the preorder and postorder traversals of each binary search tree in Figures 15-13 and 15-14? Are the preorder traversals the same? Are the postorder traversals the same?

Figure 15-13: Preorder: Jane, Bob, Alan, Elisa, Tom, Nancy, Wendy;
Postorder: Alan, Elisa, Bob, Nancy, Wendy, Tom, Jane.

Figure 15-14a: Preorder: Jane, Bob, Alan, Elisa, Nancy, Tom, Wendy;
Postorder: Alan, Elisa, Bob, Wendy, Tom, Nancy, Jane.

Figure 15-14b: Preorder: Alan, Bob, Elisa, Jane, Nancy, Tom, Wendy;
Postorder: Wendy, Tom, Nancy, Jane, Elisa, Bob, Alan.

Figure 15-14c: Preorder: Tom, Jane, Bob, Alan, Elisa, Nancy, Wendy;
Postorder: Alan, Elisa, Bob, Nancy, Jane, Wendy, Tom.

No two traversals are the same.

Question 12 Using the tree in Figure 15-14c, trace the algorithm that searches a binary search tree for

a. Elisa

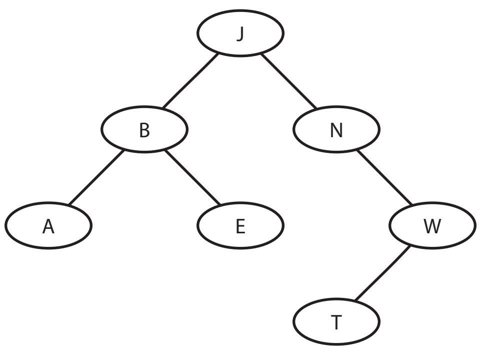
b. Kyle

In each case, list the nodes in the order in which the search visits them.

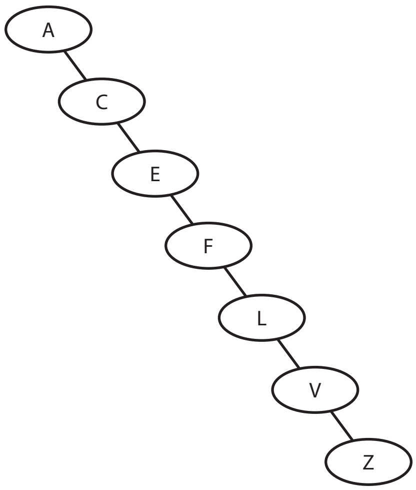
a. Tom, Jane, Bob, Elisa.

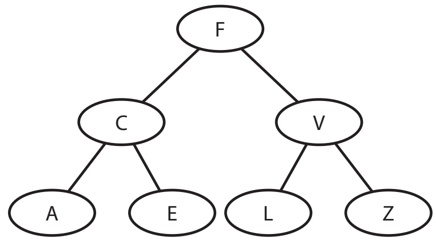
b. Tom, Jane, Nancy.

Question 13 Beginning with an empty binary search tree, what binary search tree is formed when you insert the following letters in the order given? J, N, B, A, W, E, T



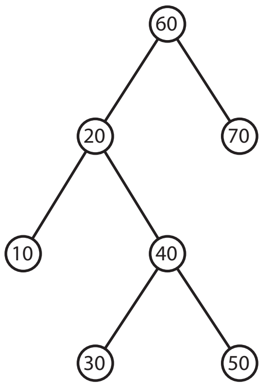
Question 14 Arrange nodes that contain the letters A, C, E, F, L, V, and Z into two binary search trees: one that has maximum height and one that has minimum height.





Chapter 16

Question 1 Represent the binary tree in Figure 15-18 of Chapter 15 with an array.



tree

	item	leftChild	rightChild
0	60	1	2
1	20	3	4
2	70	–1	–1
3	10	–1	–1
4	40	5	6
5	30	–1	–1
6	50	–1	–1
7	?	–1	8
8	?	–1	9
. . .	. . .	. . .	. . .

Free list

root

0

free

7

Question 2 What are the definitions of the public method `getNumberOfNodes` and the protected helper method `getNumberOfNodesHelper`?

```
template<class ItemType>
int BinaryNodeTree<ItemType>::getNumberOfNodes() const
{
   return getNumberOfNodesHelper(rootPtr);
}  // end getNumberOfNodes

template<class ItemType>
int BinaryNodeTree<ItemType>::
    getNumberOfNodesHelper(BinaryNode<ItemType>* subTreePtr) const
{
   if (subTreePtr == nullptr)
      return 0;
   else
      return 1 + getNumberOfNodesHelper(subTreePtr->getLeftChildPtr())
               + getNumberOfNodesHelper(subTreePtr->getRightChildPtr());
}  // end getNumberOfNodesHelper
```

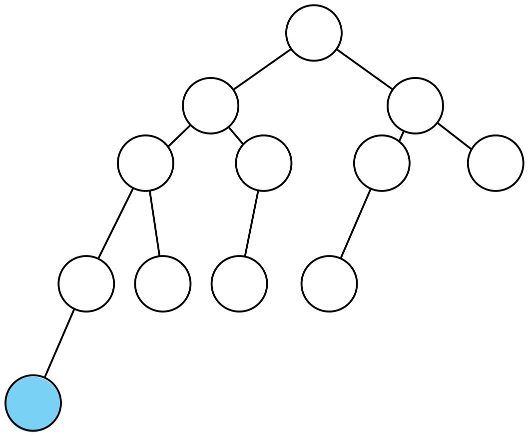
Question 3 What is the definition of the public method `setRootData`?

```
template<class ItemType>
void BinaryNodeTree<ItemType>::setRootData(const ItemType& newItem)
{
   if (isEmpty())
      rootPtr = new BinaryNode<ItemType>(newItem, nullptr, nullptr);
   else
      rootPtr->setItem(newItem);
}  // end setRootData
```

Question 4 What is the definition of the public method `getRootData`? Recall that this method has a precondition.

```
template<class ItemType>
ItemType BinaryNodeTree<ItemType>::getRootData() const
         throw(PrecondViolatedExcep)
{
   if (isEmpty())
      throw PrecondViolatedExcep(
                                "getRootData() called with empty tree.");

   return rootPtr->getItem();
}  // end getRootData
```

Question 5 Where would a new node be placed next in the binary tree shown in Figure 16-3?



Question 6 Define the protected method `postorder`.

```
template<class ItemType>
void BinaryNodeTree<ItemType>::
postorder(void visit(ItemType&), BinaryNode<ItemType>* treePtr) const
{
   if (treePtr != nullptr)
   {
      postorder(visit, treePtr->getLeftChildPtr());
      postorder(visit, treePtr->getRightChildPtr());
      ItemType theItem = treePtr->getItem();
      visit(theItem);
   }  // end if
}  // end postorder
```

Question 7 Starting with an empty binary search tree, in what order should you insert items to get the binary search tree in Figure 15-18 of Chapter 15?

60, 20, 10, 40, 30, 50, 70 is one of several possible orders. (This order results from a preorder traversal of the tree.)

Question 8 Given the binary search tree in Figure 15-18 of Chapter 15, trace the removal algorithms when removing each of the following values from the tree. Begin with the original tree each time.

a. 70

b. 20

c. 60

a. The call to `removeValue` begins at the root and locates 70 in the root's right child. This child is passed to `removeNode`, which removes the child from the tree because it is a leaf.

b. The call to `removeValue` begins at the root and locates 20 in the root's left child. This child is passed to `removeNode`, which determines that it has two children. The inorder successor of 20 is 30, and occurs in a leaf. 30 replaces 20 in the left child of the root, and the leaf that contained 30 originally is removed from the tree.

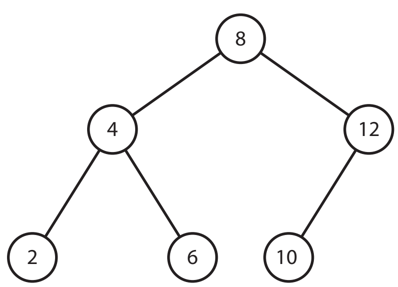
c. The call to `removeValue` begins at the root and locates 60 in the root itself. The root is passed to `removeNode`, which determines that it has two children. The inorder successor of 60 is 70, and occurs in a leaf. 70 replaces 60 in the root, and the leaf that contained 70 originally is removed from the tree.

Question 9 Consider the pseudocode operation `readTree`.

a. What binary search tree results when you execute `readTree` with a file of the six integers 2, 4, 6, 8, 10, 12?

b. Is the resulting tree's height a minimum? Is the tree complete? Is it full?

a.

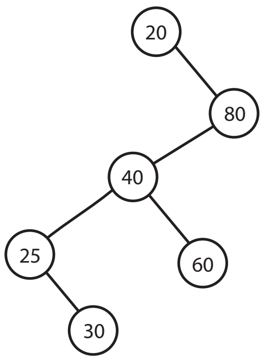


b. The tree has minimum height and is complete but not full.

Question 10 Trace the tree sort algorithm as it sorts the following array into ascending order:

20 80 40 25 60 30.

Inserting the array entries into a binary search tree produces the following tree:



An inorder traversal of this tree results in the sorted array.

Chapter 17

Question 1 Is the full binary tree in Figure 16-16 of Chapter 16 a heap? Why?

No, the tree is not a maxheap or a minheap. The root 30 is neither the largest nor the smallest value in the tree. Each of the root's children is neither the largest nor the smallest value in its subtree.

Question 2 What array represents the maxheap shown in Figure 17-1a?

10	9	6	3	2	5
0	1	2	3	4	5

Question 3 What array represents the minheap shown in Figure 17-1b?

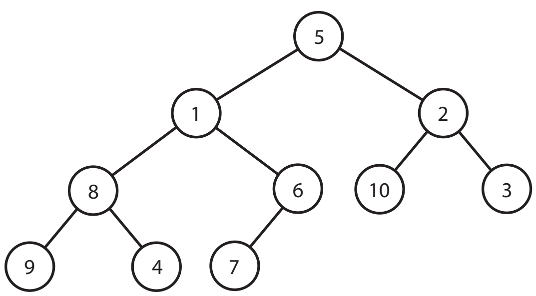
2	3	5	9	6	10
0	1	2	3	4	5

Question 4 What criterion can you use to tell whether the node in `items[i]` is a leaf?

To see whether the node in `items[i]` is a leaf, we need to test whether its children either exist or contain node values. These children are in `items[2 * i + 1]` and `items[2 * i + 2]`. Suppose that `size` is the array's length. The node in `items[i]` is a leaf if `2 * i + 1` is greater than `size`. If `2 * i + 1` is less than or equal to `size`, we must assume that sentinel values are within the unused array locations—that is, those locations after the last one used by the heap. Then if a sentinel value is in `items[2 * i + 1]`, `items[i]` is a leaf.

Question 5 What complete binary tree does the following array represent?

5	1	2	8	6	10	3	9	4	7
0	1	2	3	4	5	6	7	8	9



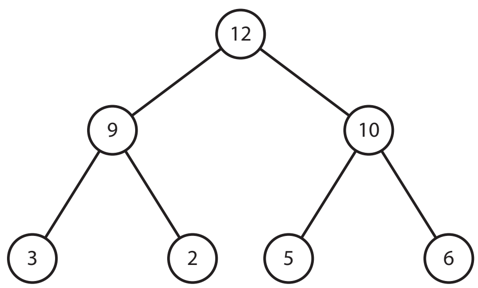
Question 6 Does the array in the previous question represent a heap?

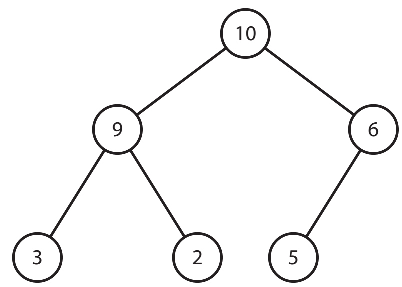
No.

Question 7 Is the full binary tree in Figure 16-16 of Chapter 16 a semiheap?

No.

Question 8 Consider the maxheap in Figure 17-1a. Draw the heap after you insert 12 and then remove 12.

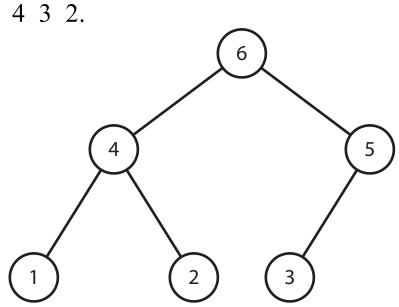




Question 9 What does the initially empty heap `myHeap` contain after the following sequence of pseudocode operations?

```
myHeap.add(2)
myHeap.add(3)
myHeap.add(4)
myHeap.add(1)
myHeap.add(9)
myHeap.remove()
myHeap.add(7)
myHeap.add(6)
myHeap.remove()
myHeap.add(5)
```

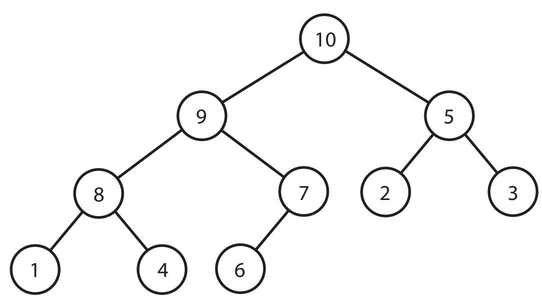
The array that represents the heap is 7 5 6 4 3 2.



Question 10 Execute the following pseudocode statements on the array shown in Checkpoint Question 5.

```
for (index = n - 1 down to 0)
   heapRebuild(index)
```

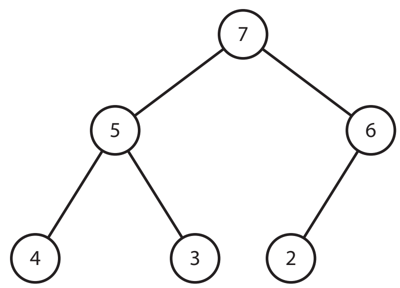
10	9	5	8	7	2	3	1	4	6
0	1	2	3	4	5	6	7	8	9



Question 11 Consider a heap-based implementation of the ADT priority queue. What does the underlying heap contain after the following sequence of pseudocode operations, assuming that `pQueue` is an initially empty priority queue?

```
pQueue.add(5)
pQueue.add(9)
pQueue.add(6)
pQueue.add(7)
pQueue.add(3)
pQueue.add(4)
pQueue.remove()
pQueue.add(9)
pQueue.add(2)
pQueue.remove()
```

The array that represents the heap is 7 5 6 4 3 2.



Question 12 Trace the heap sort as it sorts the following array into ascending order:

25 30 20 80 40 60.

25 30 20 80 40 60	Original array
80 40 60 30 25 20	Initial rebuild to form a heap
20 40 60 30 25 80	After swapping 20 and 80
60 40 20 30 25 80	`rebuild(0, anArray, 5)`
25 40 20 30 60 80	After swapping 25 and 60
40 30 20 25 60 80	`rebuild(0, anArray, 4)`
25 30 20 40 60 80	After swapping 25 and 40
30 25 20 40 60 80	`rebuild(0, anArray, 3)`
20 25 30 40 60 80	After swapping 20 and 30
25 20 30 40 60 80	`rebuild(0, anArray, 2)`
20 25 30 40 60 80	After swapping 20 and 25
20 25 30 40 60 80	Sorted array

Chapter 18

Question 1 Using the ADT dictionary operations, write pseudocode for a `replace` function at the client level that replaces the dictionary item whose search key is *x* with another item whose search key is also *x*.

```
// Replaces in the dictionary aDictionary the item whose search key matches searchKey
// with replacementItem. Returns either true if the operation is successful or false if not.
replace(aDictionary: Dictionary, searchKey: KeyType,
        replacementItem: ItemType): boolean

   result = aDictionary.remove(searchKey)
   if (result)
      result = aDictionary.add(searchKey, replacementItem)
   return result
```

Question 2 Explain how the `while` loop in the previous definition of the method `add` locates the insertion point for the new entry in the array `items`.

The loop compares the given search key with the search key of the entries in the array `items`. The search begins at the last entry in the array and progresses toward the beginning of the array. If the search key of the current entry is greater than the given search key, the entry is moved to the next higher location in the array. As soon as an entry is considered whose search key either matches or is less than the given search key, the new entry is inserted into the array at the array location just beyond that of the current entry. This location is available for use, because it either is vacant or its former entry was moved to the next higher location by this loop.

Question 3 Why is short-circuit evaluation important in the `while` loop of the previous definition of the method `add`?

Short-circuit evaluation ensures that the value of `index` is greater than zero when it is used in the expression `items[index - 1]`. If the value of `index` is zero or smaller, short-circuit evaluation prevents the execution of the second part of the boolean expression—that is, `searchKey < items[index-1].getKey()`—thereby avoiding an illegal index for the array `items`.

Question 4 We mentioned that the `remove` method calls the private method `findEntryIndex` to locate the entry to remove. Assuming that the entry is located, what does `remove` need to do after it gets the index of this entry?

If `itemIndex` is the index of the entry to remove, the `remove` method must shift the entries after `items[itemIndex]` by one position toward the beginning of the array. That is, it must execute the assigment `items[i] = items[i + 1]` for values of `i` that range from `itemIndex` to `itemCount` – 2. After these shifts, `remove` must decrease the value of `itemCount` by 1.

Question 5 What is the definition of the method `traverse` for `ArrayDictionary`?

```
template<class KeyType, class ItemType>
void ArrayDictionary<KeyType, ItemType>::traverse(void visit(ItemType&))
const
{
   // The array items is sorted; simply traverse the array.
   for (int index = 0; index < itemCount; index++)
   {
      ItemType currentItem = items[index].getItem();
      visit(currentItem);
   }  // end for
}  // end traverse
```

Question 6 Write the pseudocode for the `remove` operation when linear probing is used to implement the hash table.

```
// Removes a specific entry from the dictionary, given its search key.
// Returns true if the operation was successful, or false otherwise.
remove(searchKey: KeyType): boolean

   index = getHashIndex(searchKey)
   Search the probe sequence that begins at hashTable[index] for searchKey
   if (searchKey is found)
   {
      Flag entry as removed  // No need to actually remove the entry
      itemCount--
      return true
   }
   else
      return false
```

Question 7 What is the probe sequence that double hashing uses when

$h_1(key) = key \bmod 11$, $h_2(key) = 7 - (key \bmod 7)$, and $key = 19$?

$h_1(19) = 8$ and $h_2(19) = 2$. Thus, the probe sequence is
8, 10, 1, 3, 5, 7, 9, 0, 2, 4, 6.

Question 8 If $h(x) = x \bmod 7$ and separate chaining resolves collisions, what does the hash table look like after the following insertions occur: 8, 10, 24, 15, 32, 17? Assume that each item contains only a search key.

$h(8) = 1$, $h(10) = 3$, $h(24) = 3$, $h(15) = 1$, $h(32) = 4$, $h(17) = 3$.

`hashTable[1]`⟶15⟶8

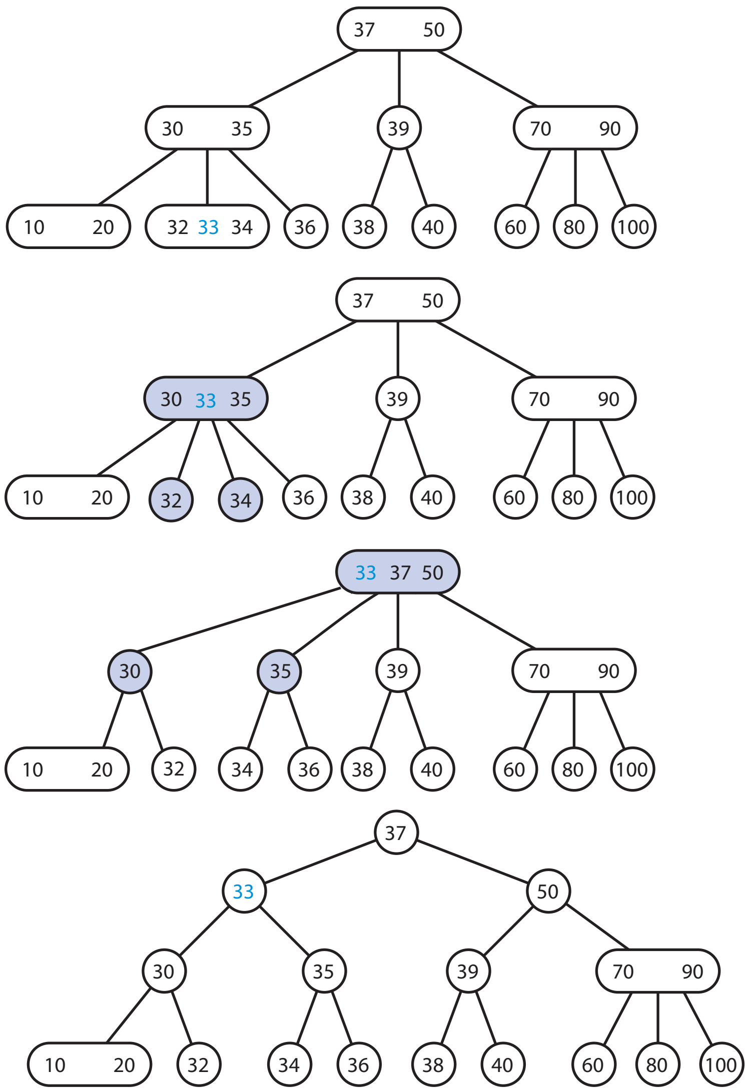
`hashTable[2`⟶`nullptr`

`hashTable[3]`⟶17⟶24⟶10

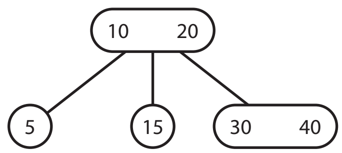
`hashTable[4]`⟶32

Chapter 19

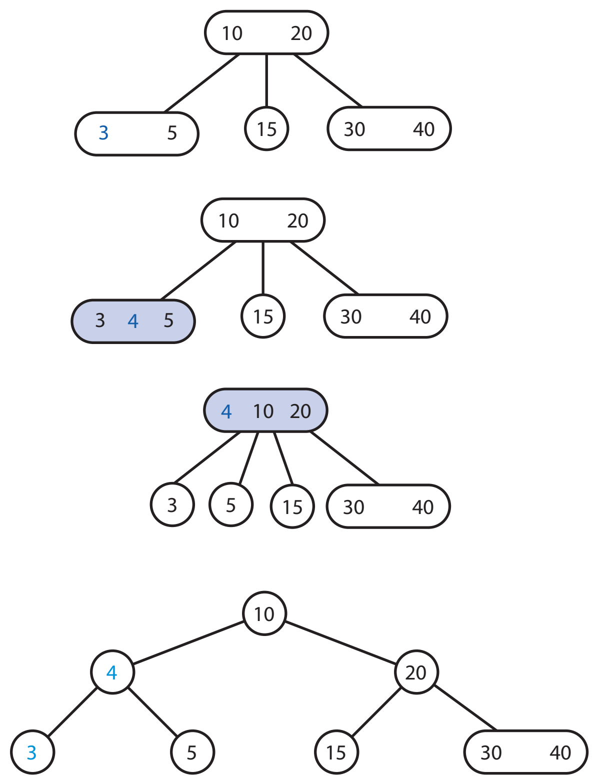
Question 1 To be sure that you fully understand the insertion algorithm, insert 32 into the 2-3 tree in Figure 19-11. The result should be the tree shown in Figure 19-6b. Once again, compare this tree with the binary search tree in Figure 19-6a and notice the dramatic advantage of the 2-3 tree's insertion strategy.



Question 2 What is the result of inserting 5, 40, 10, 20, 15, and 30—in the order given—into an initially empty 2-3 tree? Note that insertion of one item into an empty 2-3 tree will create a single node that contains the inserted item.

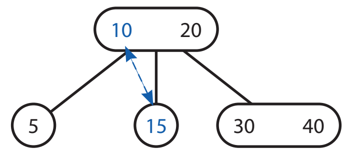


Question 3 What is the result of inserting 3 and 4 into the 2-3 tree that you created in the previous question?

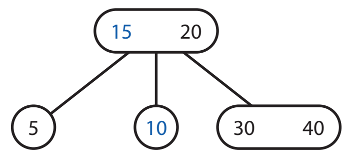


Question 4 What is the result of removing the 10 from the 2-3 tree that you created in Checkpoint Question 2?

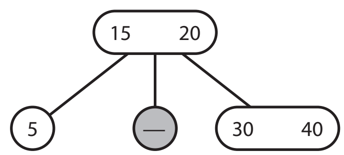
Locate 10 and its inorder successor



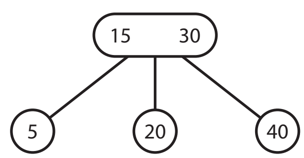
After swap with inorder successor



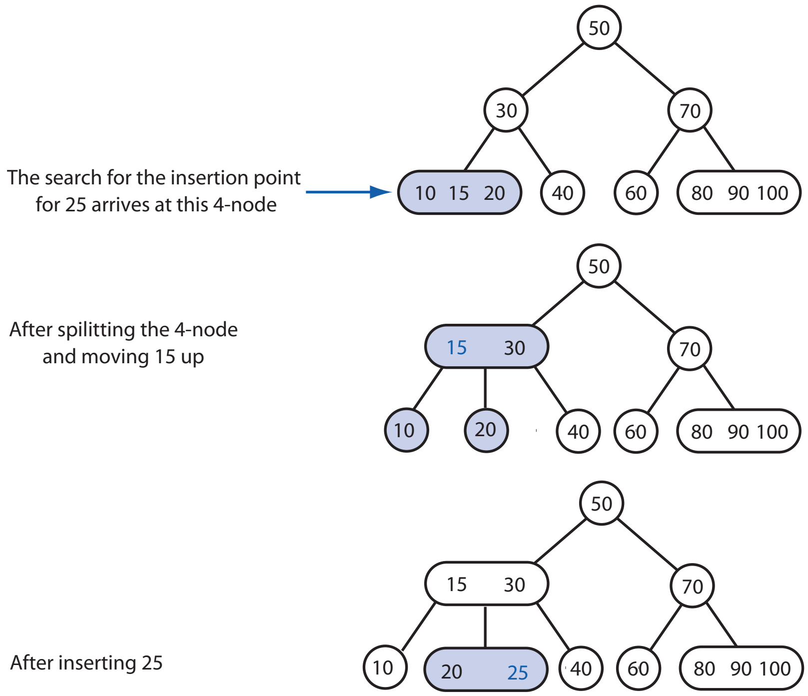
Remove 10 from leaf



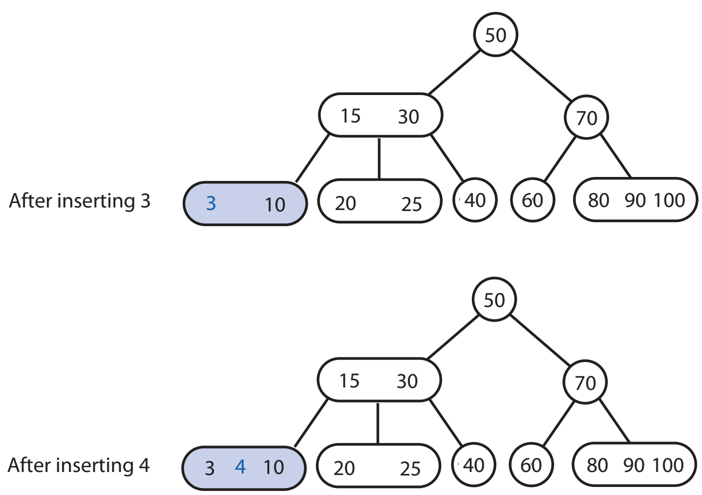
Redistribute by moving 20 down and 30 up



Question 5 Insert 25 into the 2-3-4 tree in Figure 19-27b.



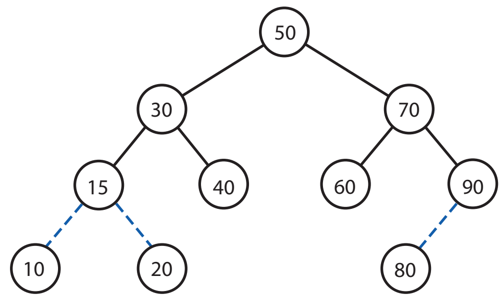
Question 6 Insert 3 and 4 into the 2-3-4 tree that you created in the previous question.



Question 7 Why does a node in a red-black tree require less memory than a node in a 2-3-4 tree?

Each node in a red-black tree requires memory for two pointers and two pointer colors. These pointers and pointer colors require no more memory than the four pointers in a node in a 2-3-4 tree. In addition, a node in a red-black tree requires memory for only one data item, whereas a node in a 2-3-4 tree requires memory for three data items.

Question 8 What red-black tree represents the 2-3-4 tree in Figure 19-27a?



Chapter 20

Question 1 Describe the graphs in Figure 20-32. For example, are they directed? Connected? Complete? Weighted?

Figure 20-32a is directed and connected. Figure 20-32b is undirected and connected. Neither graph is weighted.

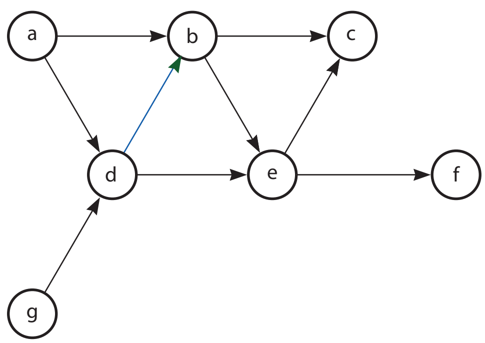
Question 2 Use the depth-first strategy and the breadth-first strategy to traverse the graph in Figure 20-32a, beginning with vertex 0. List the vertices in the order in which each traversal visits them.

DFS: 0, 1, 2, 4, 3; BFS: 0, 1, 2, 3, 4.

Question 3 Write the adjacency matrix for the graph in Figure 20-32a.

	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	1	0
2	0	0	0	0	1
3	0	1	0	0	0
4	1	0	0	0	0

Question 4 Add an edge to the directed graph in Figure 20-14 that runs from vertex *d* to vertex *b*. Write all possible topological orders for the vertices in this new graph.

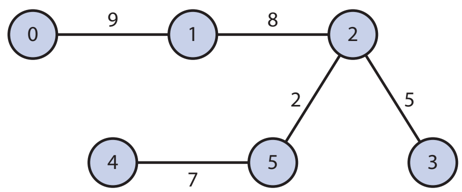


The topological orders are
a g d b e c f
g a d b e c f
a g d b e f c
g a d b e f c

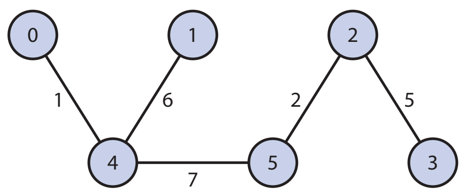
Question 5 Is it possible for a connected undirected graph with five vertices and four edges to contain a simple cycle? Explain.

No. See Observation 2 in Section 20.4.2.

Question 6 Draw the DFS spanning tree whose root is vertex 0 for the graph in Figure 20-33.



Question 7 Draw the minimum spanning tree whose root is vertex 0 for the graph in Figure 20-33.



Question 8 What are the shortest paths from vertex 0 to each vertex of the graph in Figure 20- 24a? (Note the weights of these paths in Figure 20-25.)

Path from 0 to 1 [0, 4, 2, 1] has weight 7.
Path from 0 to 2 [0, 4, 2] has weight 5.
Path from 0 to 3 [0, 4, 2, 3] has weight 8.
Path from 0 to 4 [0, 4] has weight 4.

Chapter 21

Question 1 Consider two files of 1,600 employee records each. The records in each file are organized into sixteen 100-record blocks. One file is sequential access and the other is direct access. Describe how you would append one record to the end of each file.

Sequential access file: Copy the original file `file1` into another file `file2`. Write on `file2` a new block containing the desired record and 99 blank records. Copy `file2` to the original file `file1`.
Direct access file: Create a new block containing the desired record and 99 blank records. Write the new block to the file as the 17th block.

Question 2 Trace `externalMergesort` with an external file of 16 blocks. Assume that the arrays `in1`, `in2`, and `out` are each one block long. List the calls to the various functions in the order in which they occur.

```
externalMergesort(unsortedFileName, sortedFileName)
// This call to externalMergesort results in the following actions:
  Associate unsortedFileName with the file variable inFile
  Associate sortedFileName with the file variable outFile
  blocksort(inFile, tempFile1, numBlocks)
  // Records in each block are now sorted; numBlocks is 16
  mergeFile(tempFile1, tempFile2, 1, 16)
  // This call to mergeFile makes the following calls to mergeRuns:
     mergeRuns(tempFile1, tempFile2, 1, 1)
     mergeRuns(tempFile1, tempFile2, 3, 1)
     mergeRuns(tempFile1, tempFile2, 5, 1)
     mergeRuns(tempFile1, tempFile2, 7, 1)
     mergeRuns(tempFile1, tempFile2, 9, 1)
     mergeRuns(tempFile1, tempFile2, 11, 1)
     mergeRuns(tempFile1, tempFile2, 13, 1)
     mergeRuns(tempFile1, tempFile2, 15, 1)

  mergeFile(tempFile2, tempFile1, 2, 16)
  // This call to mergeFile makes the following calls to mergeRuns:
     mergeRuns(tempFile2, tempFile1, 1, 2)
     mergeRuns(tempFile2, tempFile1, 5, 2)
     mergeRuns(tempFile2, tempFile1, 9, 2)
```

```
      mergeRuns(tempFile2, tempFile1, 13, 2)

   mergeFile(tempFile1, tempFile2, 4, 16)
   // This call to mergeFile makes the following calls to mergeRuns:
      mergeRuns(tempFile1, tempFile2, 1, 4)
      mergeRuns(tempFile1, tempFile2, 9, 4)

   mergeFile(tempFile2, tempFile1, 8, 16)
   // This call to mergeFile makes the following calls to mergeRuns:
      mergeRuns(tempFile2, tempFile1, 1, 8)

   copyFile(tempFile1, outFile)
```

Question 3 Trace the retrieval algorithm for an indexed external file when the search key is less than all keys in the index. Assume that the index file stores the index records sequentially, sorted by their search keys, and contains 20 blocks of 50 records each. Also assume that the data file contains 100 blocks, and that each block contains 10 employee records. List the calls to the various functions in the order in which they occur.

```
getItem(indexFile[1..20], dataFile, searchKey)
  buf.readBlock(indexFile[1..20], 10)
  getItem(indexFile[1..9], dataFile, searchKey)
     buf.readBlock(indexFile[1..9], 5)
     getItem(indexFile[1..4], dataFile, searchKey)
        buf.readBlock(indexFile[1..4], 2)
        getItem(indexFile[1..1], dataFile, searchKey)
           buf.readBlock(indexFile[1..1], 1)
           throw NotFoundException
```

Question 4 Repeat Checkpoint Question 3, but this time assume that the search key equals the key in record 26 of block 12 of the index. Also assume that record 26 of the index points to block 98 of the data file.

```
getItem(indexFile[1..20], dataFile, searchKey)
  buf.readBlock(indexFile[1..20], 10)
  getItem(tIndex[11..20], dataFile, searchKey)
     buf.readBlock(indexFile[11..20], 15)
     getItem(tIndex[11..14], dataFile, searchKey)
        buf.readBlock(indexFile[11..14], 12)
        j = 26
        blockNum = 98
        data.readBlock(dataFile, 98)
        Find data record data.getRecord(k) whose search key equals searchKey
        return data.getRecord(k)
```