

PART V

DATA STRUCTURES AND COLLECTIONS FRAMEWORK

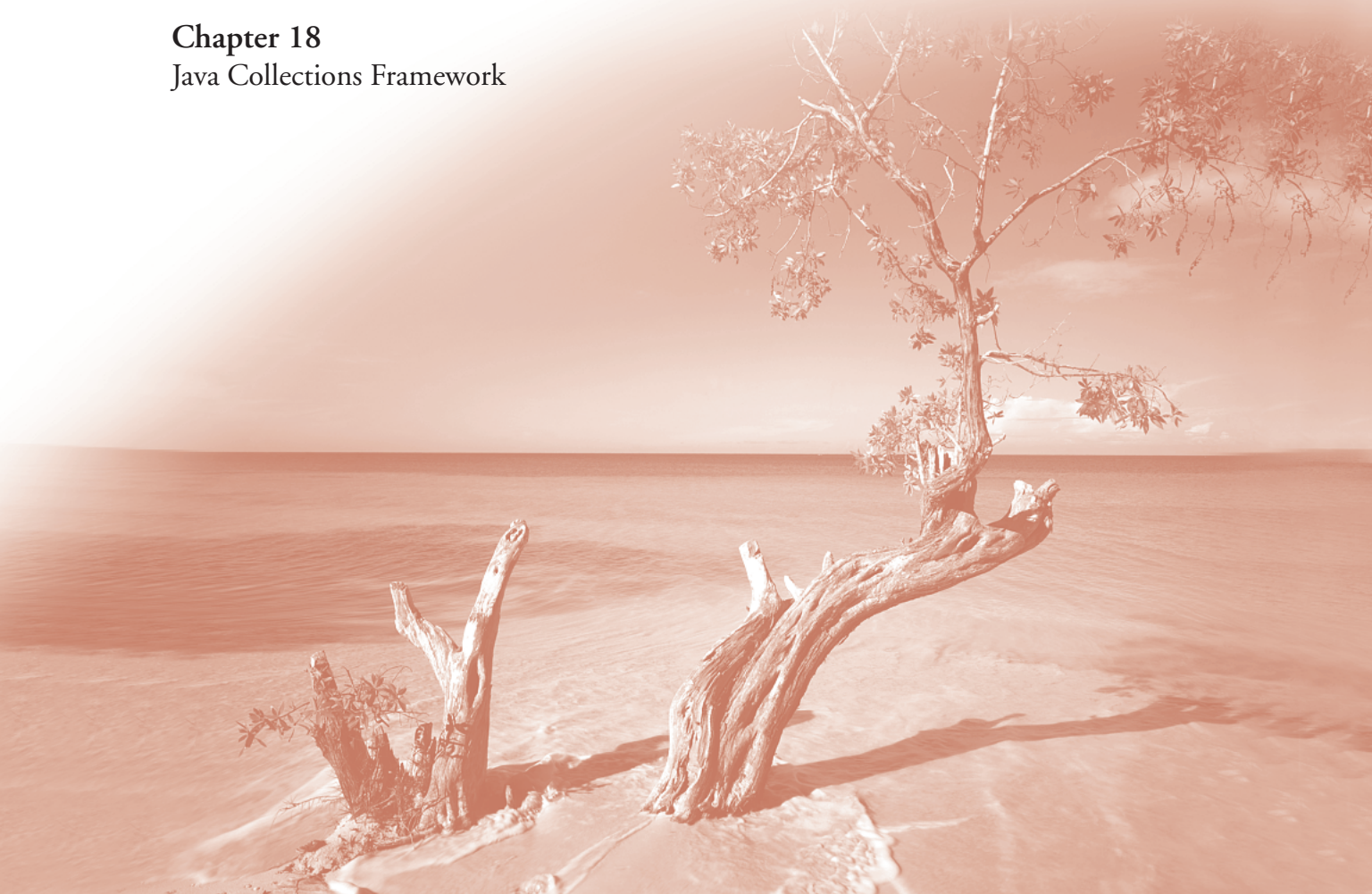
The design and implementation of efficient data structures is an important subject in computer science. Data structures such as lists, stacks, queues, sets, maps, and binary trees have many applications in compiler construction, computer operating systems, and file management. Java provides the data structures for lists, stacks, sets, and maps in the collections framework. This part of the book introduces data structure concepts and implementation techniques in Chapter 17, “Object-Oriented Data Structures,” and introduces how to use the classes and interfaces in the Java collections framework in Chapter 18, “Java Collections Framework.” These two chapters are designed to be independent. You can skip Chapter 17 if you are only interested in learning how to use the predefined data structures in the Java collections framework.

Chapter 17

Object-Oriented Data Structures

Chapter 18

Java Collections Framework

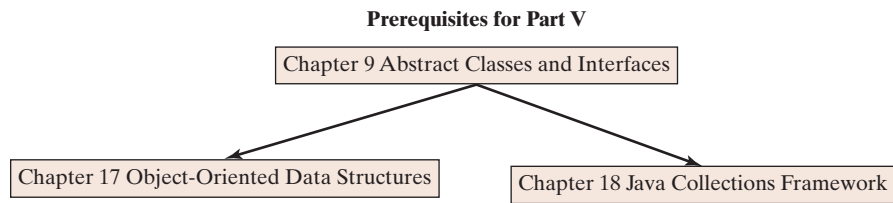


Prerequisites for Part V

Chapter 9 Abstract Classes and Interfaces

Chapter 17 Object-Oriented Data Structures

Chapter 18 Java Collections Framework



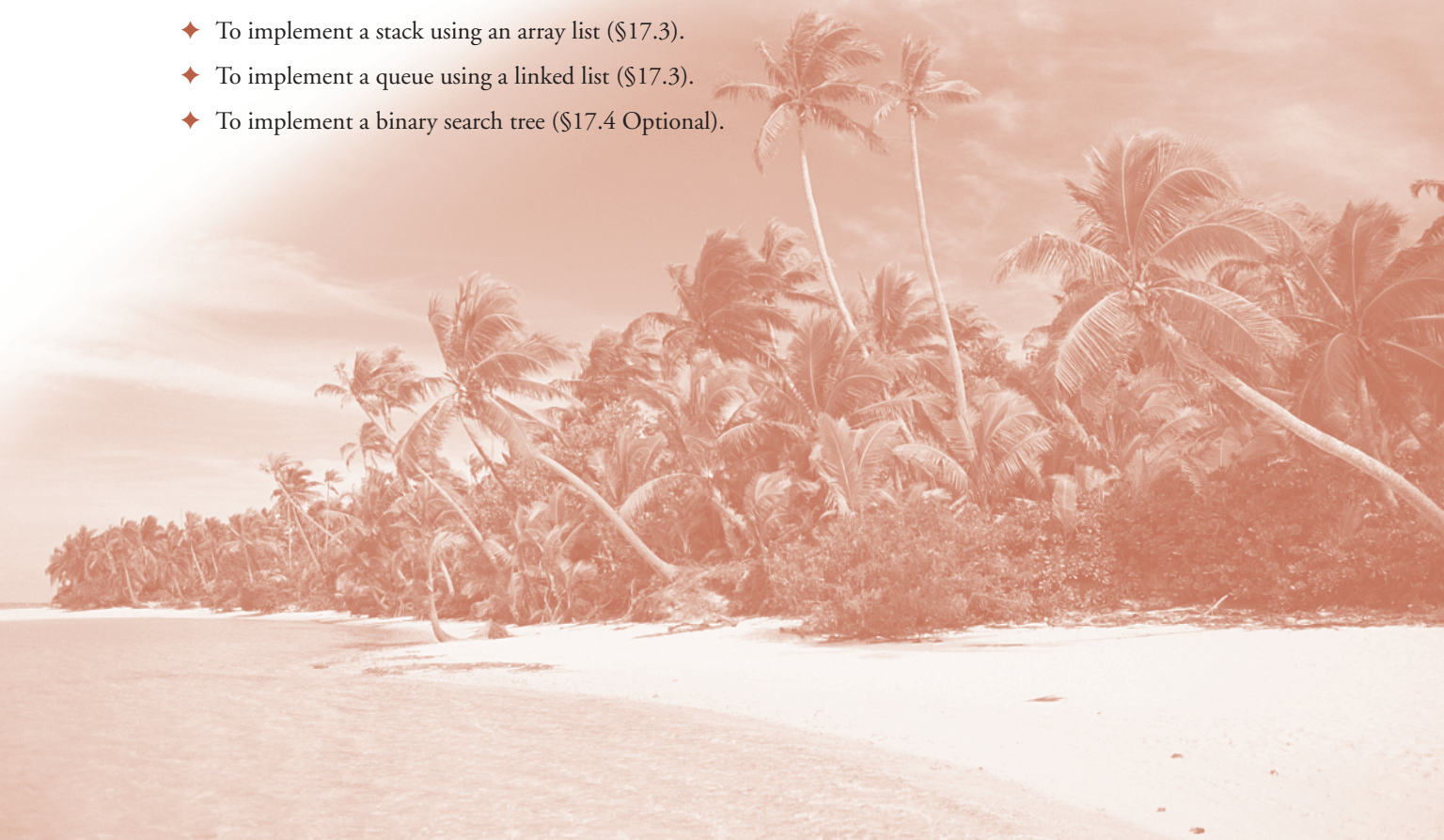
chapter

17

OBJECT-ORIENTED DATA STRUCTURES

Objectives

- ◆ To describe what a data structure is (§17.1).
- ◆ To explain the limitations of arrays (§17.1).
- ◆ To implement a dynamic list using an array (§17.2.1).
- ◆ To implement a dynamic list using a linked structure (§17.2.2 Optional).
- ◆ To implement a stack using an array list (§17.3).
- ◆ To implement a queue using a linked list (§17.3).
- ◆ To implement a binary search tree (§17.4 Optional).



17.1 Introduction

A data structure is a collection of data organized in some fashion. A data structure not only stores data, but also supports the operations for manipulating data in the structure. For example, an array is a data structure that holds a collection of data in sequential order. You can find the size of the array, and store, retrieve, and modify data in the array. Arrays are simple and easy to use, but they have two limitations: (1) once an array is created, its size cannot be altered; (2) an array does not provide adequate support for insertion and deletion operations. This chapter introduces dynamic data structures that grow and shrink at runtime.

Four classic dynamic data structures are introduced in this chapter: lists, stacks, queues, and binary trees. A *list* is a collection of data stored sequentially. It supports insertion and deletion anywhere in the list. A *stack* can be perceived as a special type of list where insertions and deletions take place only at one end, referred to as the *top* of the stack. A *queue* represents a waiting list, where insertions take place at the back (also referred to as the tail) of the queue, and deletions take place from the front (also referred to as the head) of the queue. A *binary tree* is a data structure that supports searching, sorting, inserting, and deleting data efficiently.

In object-oriented thinking, a data structure is an object that stores other objects, referred to as data or elements. Some people refer to data structures as *container objects* or *collection objects*. To define a data structure is essentially to declare a class. The class for a data structure should use data fields to store data and provide methods to support such operations as insertion and deletion. To create a data structure is therefore to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into the data structure or deleting an element from the data structure.

This chapter introduces the design and implementation of the classes for data structures: lists, stacks, queues, and binary trees. *The whole chapter is optional and can be skipped.*

17.2 Lists

A list is a popular data structure for storing data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books can all be stored using lists. The operations listed below are typical of most lists:

- ◆ Retrieve an element from a list.
- ◆ Insert a new element to a list.
- ◆ Delete an element from a list.
- ◆ Find how many elements are in a list.
- ◆ Find whether an element is in a list.
- ◆ Find whether a list is empty.

There are two ways to implement a list. One is to use an *array* to store the elements. Arrays are dynamically created. If the capacity of the array is exceeded, create a new, larger array and copy all the elements from the current array to the new array. The other approach is to use a *linked structure*. A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list. Thus you can declare two classes for lists. For convenience, let's name these two classes `MyArrayList` and `MyLinkedList`. These two classes have common operations but different data fields. The common operations can be generalized in an interface or an abstract class. As discussed in Section 10.6.5, "Using Interfaces or Abstract Classes," a good strategy is to combine the virtues of interfaces and abstract classes by providing both an interface and an abstract class in the design so that the user can use either of them, whichever is convenient. Such an abstract class is known as a *convenience class*.

Let us name the interface `MyList` and the convenience class `MyAbstractList`. Figure 17.1 shows the relationship of `MyList`, `MyAbstractList`, `MyArrayList`, and `MyLinkedList`. The methods in

array limitation

list
stack
queue

binary tree

collection object

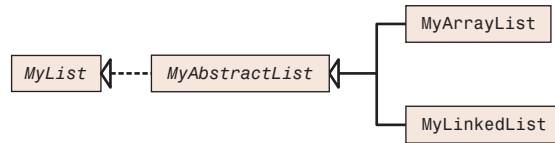


FIGURE 17.1 *MyList defines a common interface for MyAbstractList, MyArrayList, and MyLinkedList.*

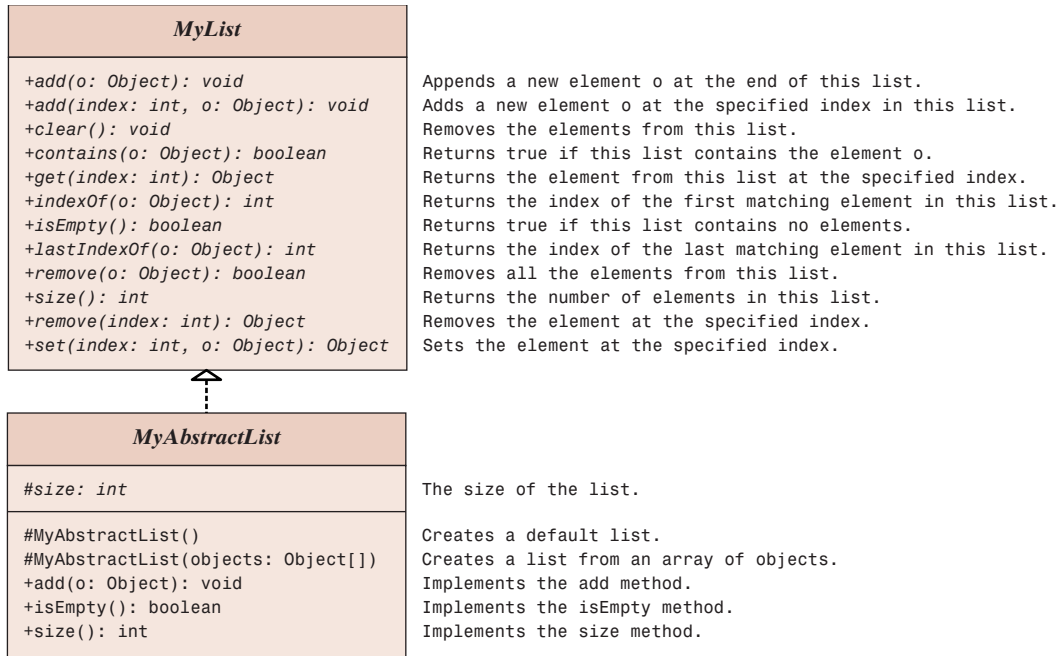


FIGURE 17.2 *List supports many methods for manipulating a list. MyAbstractList provides a partial implementation of the List interface.*

MyList and the methods implemented in MyAbstractList are shown in Figure 17.2. Listing 17.1 gives the source code for MyList.

LISTING 17.1 MyList.java (The Interface for Lists)

```

1 public interface MyList {
2     /** Add a new element o at the end of this list */
3     public void add(Object o);
4
5     /** Add a new element o at the specified index in this list */
6     public void add(int index, Object o);
7
8     /** Clear the list */
9     public void clear();
10
11     /** Return true if this list contains the element o */
12     public boolean contains(Object o);
13
14     /** Return the element from this list at the specified index */
15     public Object get(int index);
16
17     /** Return the index of the first matching element in this list.
18      * Return -1 if no match. */
19     public int indexOf(Object o);
20
21     /** Return true if this list contains no elements */
22     public boolean isEmpty();
  
```

```

23
24 /** Return the index of the last matching element in this list
25  * Return -1 if no match. */
26 public int lastIndexOf(Object o);
27
28 /** Remove the first occurrence of the element o from this list.
29  * Shift any subsequent elements to the left.
30  * Return true if the element is removed. */
31 public boolean remove(Object o);
32
33 /** Remove the element at the specified position in this list
34  * Shift any subsequent elements to the left.
35  * Return the element that was removed from the list. */
36 public Object remove(int index);
37
38 /** Replace the element at the specified position in this list
39  * with the specified element and return the new set. */
40 public Object set(int index, Object o);
41
42 /** Return the number of elements in this list */
43 public int size();
44 }

```

MyAbstractList declares variable size to indicate the number of elements in the list. The methods isEmpty and size can be implemented in the class in Listing 17.2.

LISTING 17.2 MyAbstractList.java (Partially Implements MyList)

```

size
2 public abstract class MyAbstractList implements MyList {
3     protected int size; // The size of the list
4
5     /** Create a default list */
6     protected MyAbstractList() {
7     }
8
9     /** Create a list from an array of objects */
10    protected MyAbstractList(Object[] objects) {
11        for (int i = 0; i < objects.length; i++)
12            this.add(objects[i]);
13    }
14
15    /** Add a new element o at the end of this list */
16    public void add(Object o) {
17        add(size, o);
18    }
19
20    /** Return true if this list contains no elements */
21    public boolean isEmpty() {
22        return size == 0;
23    }
24
25    /** Return the number of elements in this list */
26    public int size() {
27        return size;
28    }
29 }

```

no-arg constructor

CONSTRUCTOR

The following sections give the implementation for MyArrayList and MyLinkedList, respectively.

17.2.1 Array Lists

An array is a fixed-size data structure. Once an array is created, its size cannot be changed. Nevertheless, you can still use arrays to implement dynamic data structures. The trick is to create a larger new array to replace the current array if the current array cannot hold new elements in the list. This section shows how to use arrays to implement MyArrayList.

Initially, an array, say data of Object[] type, is created with a default size. When inserting a new element into the array, first make sure that there is enough room in the array. If not, create a new array twice as large as the current one. Copy the elements from the current array to the new array. The new

array now becomes the current array. Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by 1, as shown in Figure 17.3.

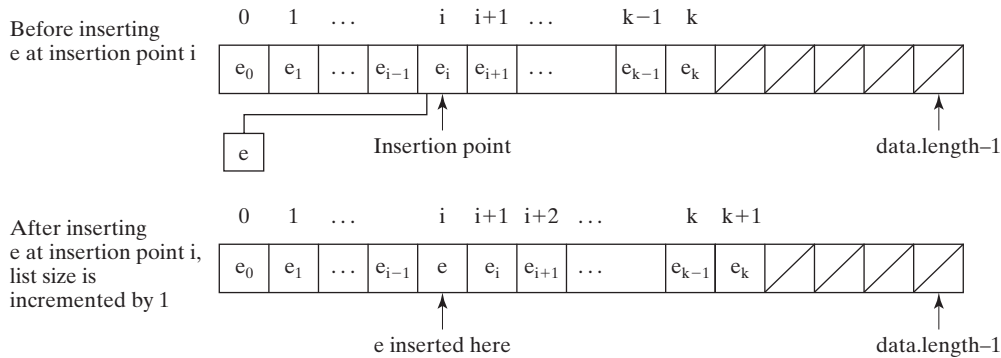


FIGURE 17.3 Inserting a new element to the array requires that all the elements after the insertion point be shifted one position to the right so that the new element can be inserted at the insertion point.



NOTE

The data array is of type `Object[]`. Each cell in the array actually stores the reference of an object.

To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by 1, as shown in Figure 17.4.

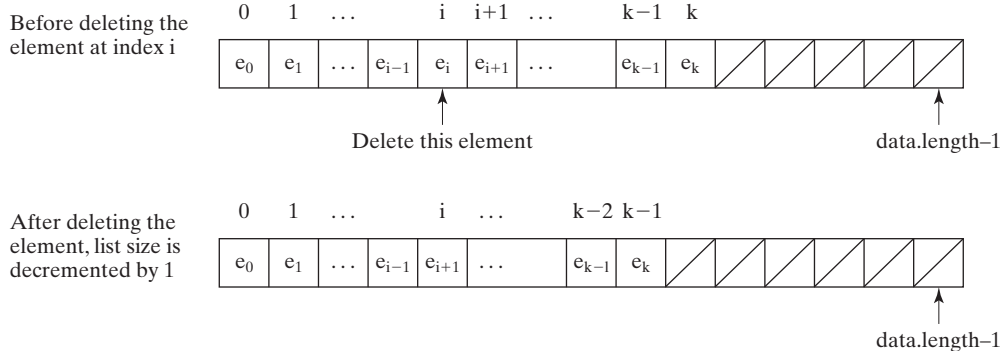


FIGURE 17.4 Deleting an element from the array requires that all the elements after the deletion point be shifted one position to the left.

`MyArrayList` uses an array to implement `MyAbstractList`, as shown in Figure 17.5. Its implementation is given in Listing 17.3.

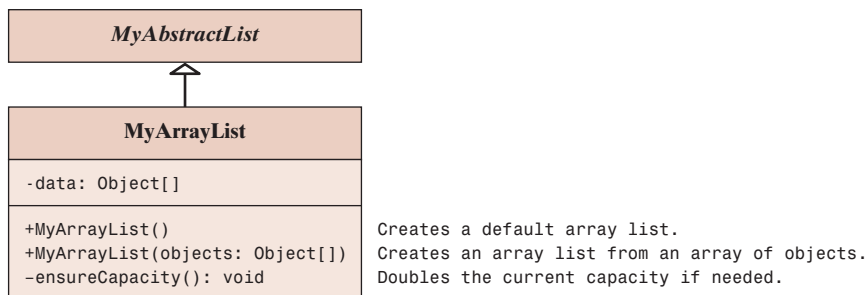


FIGURE 17.5 `MyArrayList` implements a list using an array.

LISTING 17.3 MyArrayList.java (Implementing List Using Array)

initial capacity
array

```

1 public class MyArrayList extends MyAbstractList {
2     public static final int INITIAL_CAPACITY = 16;
3     private Object[] data = new Object[INITIAL_CAPACITY];
4
5     /** Create a default list */
6     public MyArrayList() {
7     }
8
9     /** Create a list from an array of objects */
10    public MyArrayList(Object[] objects) {
11        data = objects;
12        size = objects.length;
13    }
14
15    /** Add a new element o at the specified index in this list */
16    public void add(int index, Object o) {
17        ensureCapacity();
18
19        // Move the elements to the right after the specified index
20        for (int i = size - 1; i >= index; i--)
21            data[i + 1] = data[i];
22
23        // Insert new element to data[index]
24        data[index] = o;
25
26        // Increase size by 1
27        size++;
28    }
29
30    /** Create a new larger array, double the current size */
31    private void ensureCapacity() {
32        if (size >= data.length) {
33            Object[] newData = new Object[data.length * 2];
34            System.arraycopy(data, 0, newData, 0, data.length);
35            data = newData;
36        }
37    }
38
39    /** Clear the list */
40    public void clear() {
41        data = new Object[INITIAL_CAPACITY];
42    }
43
44    /** Return true if this list contains the element o */
45    public boolean contains(Object o) {
46        for (int i = 0; i < size; i++)
47            if (o.equals(data[i])) return true;
48
49        return false;
50    }
51
52    /** Return the element from this list at the specified index */
53    public Object get(int index) {
54        return data[index];
55    }
56
57    /** Return the index of the first matching element in this list.
58     * Return -1 if no match. */
59    public int indexOf(Object o) {
60        for (int i = 0; i < size; i++)
61            if (o.equals(data[i])) return i;
62
63        return -1;
64    }

```

double capacity


```

65
66 /** Return the index of the last matching element in this list
67  * Return -1 if no match. */
68 public int lastIndexOf(Object o) {
69     for (int i = size - 1; i >= 0; i--)
70         if (o.equals(data[i])) return i;
71
72     return -1;
73 }
74
75 /** Remove the first occurrence of the element o from this list.
76  * Shift any subsequent elements to the left.
77  * Return true if the element is removed. */
78 public boolean remove(Object o) {
79     for (int i = 0; i < size; i++)
80         if (o.equals(data[i])) {
81             remove(i);
82             return true;
83         }
84
85     return false;
86 }
87
88 /** Remove the element at the specified position in this list
89  * Shift any subsequent elements to the left.
90  * Return the element that was removed from the list. */
91 public Object remove(int index) {
92     Object o = data[index];
93
94     // Shift data to the left
95     for (int j = index; j < size - 1; j++)
96         data[j] = data[j + 1];
97
98     // Decrement size
99     size--;
100
101     return o;
102 }
103
104 /** Replace the element at the specified position in this list
105  * with the specified element. */
106 public Object set(int index, Object o) {
107     data[index] = o;
108     return o;
109 }
110
111 /** Override toString() to return elements in the list */
112 public String toString() {
113     StringBuffer result = new StringBuffer("");
114
115     for (int i = 0; i < size; i++) {
116         result.append(data[i]);
117         if (i < size - 1) result.append(", ");
118     }
119
120     return result.toString() + "];";
121 }
122 }

```

The constant `INITIAL_CAPACITY` (Line 2) is used to create an initial array `data` of type `Object` (Line 3).

The `add(int index, Object o)` method (Lines 16–28) adds element `o` at the specified index in the array. This method first invokes `ensureCapacity()` (Line 17), which ensures that there is a space in the array for the new element. It then shifts all the elements after the index one position

to the right before inserting the element (Lines 20–21). After the element is added, `size` is incremented by 1 (Line 27). Note that variable `size` is defined as `protected` in `MyAbstractList`, so it can be accessed in `MyArrayList`.

The `ensureCapacity()` method (Lines 31–37) checks whether the array is full. If so, create a new array that doubles the current array size, copy the current array to the new array using the `System.arraycopy` method, and set the new array as the current array.

The `clear()` method (Lines 40–42) creates a brand-new array with initial capacity.

The `contains(Object o)` method (Lines 45–50) checks whether element `o` is contained in the array by comparing `o` with each element in the array using the `equals` method.

The `get(int index)` method (Lines 53–55) simply returns `data[index]`. The implementation of this method is simple and efficient.

The `indexOf(Object o)` method (Lines 59–64) compares element `o` with the elements in the array starting from the first one. If a match is found, the index of the element is returned; otherwise, it returns `-1`.

The `lastIndexOf(Object o)` method (Lines 68–73) compares element `o` with the elements in the array starting from the last one. If a match is found, the index of the element is returned; otherwise, it returns `-1`.

The `remove(Object o)` method (Lines 78–86) compares element `o` with the elements in the array. If a match is found, the first matching element in the list is deleted.

The `remove(int index)` method (Lines 91–102) shifts all the elements before the index one position to the left and decrements `size` by 1.

The `set(int index, Object o)` method (Lines 106–109) simply assigns `o` to `data[index]` to replace the element at the specified index with element `o`.

The `toString()` method (Lines 112–121) overrides the `toString` method in the `Object` class to return a string representing all the elements in the list.

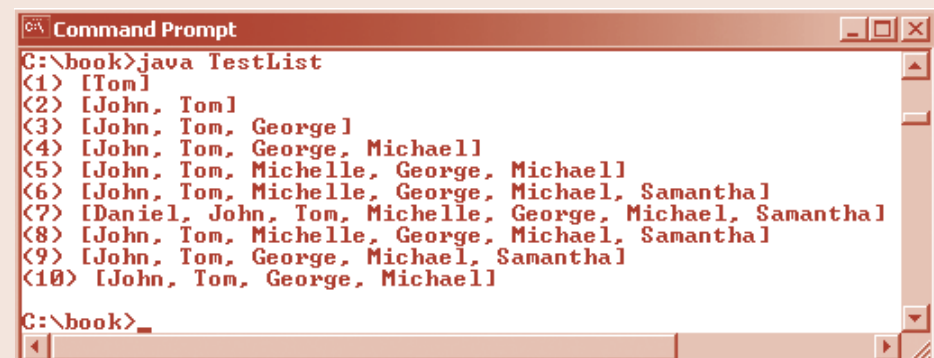
EXAMPLE 17.1 USING ARRAY LISTS

Problem

Write a program that creates a list using `MyArrayList`. It then uses the `add` method to add strings to the list and the `remove` method to remove strings from the list.

Solution

Listing 17.4 gives the solution to the problem. A sample run of the program is shown in Figure 17.6.



```

C:\book>java TestList
<1> [Tom]
<2> [John, Tom]
<3> [John, Tom, George]
<4> [John, Tom, George, Michael]
<5> [John, Tom, Michelle, George, Michael]
<6> [John, Tom, Michelle, George, Michael, Samantha]
<7> [Daniel, John, Tom, Michelle, George, Michael, Samantha]
<8> [John, Tom, Michelle, George, Michael, Samantha]
<9> [John, Tom, George, Michael, Samantha]
<10> [John, Tom, George, Michael]
C:\book>

```

FIGURE 17.6 The program uses a list to store and process strings.

EXAMPLE 17.1 (CONTINUED)**LISTING 17.4** TestList.java (Using Lists)

```

1 public class TestList {
2     public static void main(String[] args) {
3         // Create a list
4         MyList list = new MyArrayList();
5
6         // Add elements to the list
7         list.add("Tom"); // Add it to the list
8         System.out.println("(1) " + list);
9
10        list.add(0, "John"); // Add it to the beginning of the list
11        System.out.println("(2) " + list);
12
13        list.add("George"); // Add it to the end of the list
14        System.out.println("(3) " + list);
15
16        list.add("Michael"); // Add it to the end of the list
17        System.out.println("(4) " + list);
18
19        list.add(2, "Michelle"); // Add it to the list at index 2
20        System.out.println("(5) " + list);
21
22        list.add(5, "Samantha"); // Add it to the list at index 5
23        System.out.println("(6) " + list);
24
25        list.add(0, "Daniel"); // Same as list.addFirst("Daniel")
26        System.out.println("(7) " + list);
27
28        // Remove elements from the list
29        list.remove("Daniel"); // Same as list.remove(0) in this case
30        System.out.println("(8) " + list);
31
32        list.remove(2); // Remove the element at index 2
33        System.out.println("(9) " + list);
34
35        list.remove(list.size() - 1); // Remove the last element
36        System.out.println("(10) " + list);
37    }
38 }

```

Review

`MyArrayList` is implemented using arrays. Although an array is a fixed-size data structure, `MyArrayList` is a dynamic data structure. The user can create an instance of `MyArrayList` to store any number of elements. The internal array in `MyArrayList` is encapsulated. The user manipulates the list through the public methods in `MyArrayList`.

The list can hold any objects. A primitive data type value cannot be directly stored in a list. However, you can create an object for a primitive data type value using the corresponding wrapper class. For example, to store number 10, use the following method:

```
list.add(new Integer(10));
```

17.2.2 Linked Lists (Optional)

Since `MyArrayList` is implemented using an array, the methods `get(int index)` and `set(int index, Object o)` for accessing and modifying an element through an index and the `add(Object o)` for adding an element at the end of the list are efficient. However, the methods `add(int index,`

Object `o`) and `remove(int index)` are inefficient because they require shifting a potentially large number of elements. You can use a linked structure to implement a list to improve efficiency for adding and removing an element anywhere in a list.

A linked list consists of nodes, as shown in Figure 17.7. Each node contains an element, and each node is linked to its next neighbor. Thus a node can be defined as a class, as follows:

```
class Node {
    Object element;
    Node next;

    public Node(Object o) {
        element = o;
    }
}
```

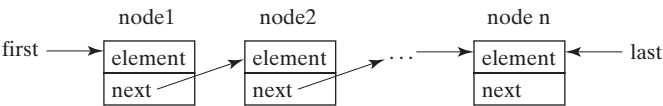


FIGURE 17.7 A linked list consists of any number of nodes chained together.

The variable `first` refers to the first node in the list, and the variable `last` refers to the last node in the list. If the list is empty, both are `null`. For example, you can create three nodes to store three circle objects (radius 1, 2, and 3) in a list:

```
Node first, last;

// Create a node to store the first circle object
first = new Node(new Circle(1));
last = first;

// Create a node to store the second circle object
last.next = new Node(new Circle(2));
last = last.next;

// Create a node to store the third circle object
last.next = new Node(new Circle(3));
last = last.next;
```

The process of creating a new linked list and adding three nodes is shown in Figure 17.8.

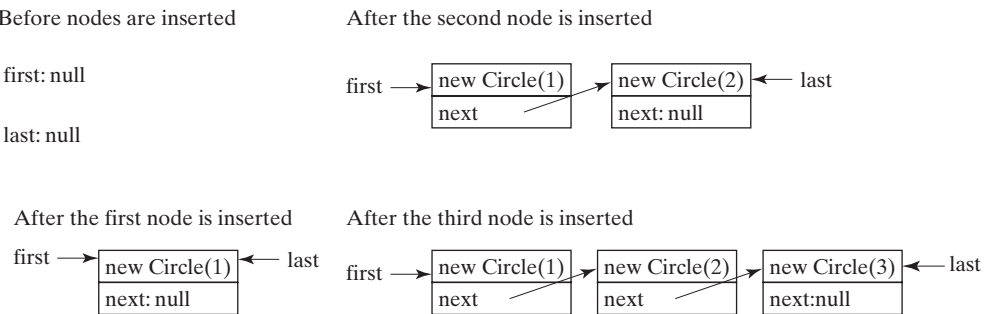


FIGURE 17.8 Three nodes are added to a new linked list.

`MyLinkedList` uses a linked structure to implement a dynamic list. It extends `MyAbstractList`. In addition, it provides the methods `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst`, and `getLast`, as shown in Figure 17.9. Its implementation is given in Listing 17.5.

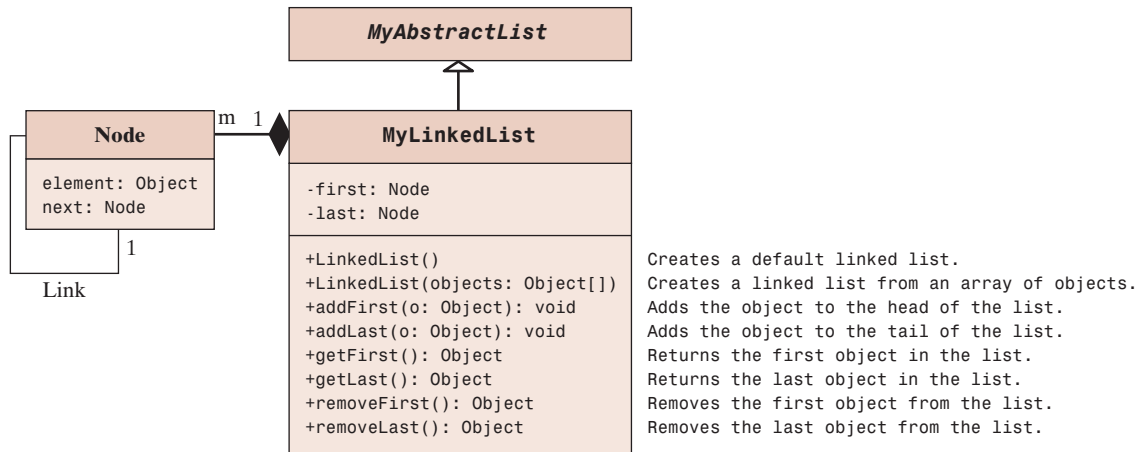


FIGURE 17.9 *MyLinkedList implements a list using a linked list of nodes.*

LISTING 17.5 MyLinkedList.java (Implementing Lists Using Linked Structures)

```

1 public class MyLinkedList extends MyAbstractList {
2     private Node first, last;
3
4     /** Create a default list */
5     public MyLinkedList() {
6     }
7
8     /** Create a list from an array of objects */
9     public MyLinkedList(Object[] objects) {
10         super(objects);
11     }
12
13     /** Return the first element in the list */
14     public Object getFirst() {
15         if (size == 0) return null;
16         else return first.element;
17     }
18
19     /** Return the last element in the list */
20     public Object getLast() {
21         if (size == 0) return null;
22         else return last.element;
23     }
24
25     /** Add an element to the beginning of the list */
26     public void addFirst(Object o) {
27         Node newNode = new Node(o);
28         newNode.next = first;
29         first = newNode;
30         size++;
31
32         if (last == null)
33             last = first;
34     }
35
36     /** Add an element to the end of the list */
37     public void addLast(Object o) {
38         if (last == null) {
39             first = last = new Node(o);
40         }
41         else {
42             last.next = new Node(o);
43             last = last.next;
44         }
45     }
46 }

```

addFirst
create node

addLast

add

```

45
46     size++;
47 }
48
49 /** Adds a new element o at the specified index in this list
50  * The index of the first element is 0 */
51 public void add(int index, Object o) {
52     if (index == 0) addFirst(o);
53     else if (index >= size) addLast(o);
54     else {
55         Node current = first;
56         for (int i = 1; i < index; i++)
57             current = current.next;
58         Node temp = current.next;
59         current.next = new Node(o);
60         (current.next).next = temp;
61         size++;
62     }
63 }
64
65 /** Remove the first node and
66  * return the object that is contained in the removed node. */
67 public Object removeFirst() {
68     if (size == 0) return null;
69     else {
70         Node temp = first;
71         first = first.next;
72         size--;
73         return temp.element;
74     }
75 }
76
77 /** Remove the last node and
78  * return the object that is contained in the removed node. */
79 public Object removeLast() {
80     // Implementation left as an exercise
81     return null;
82 }
83
84 /** Removes the element at the specified position in this list
85  * Returns the element that was removed from the list. */
86 public Object remove(int index) {
87     if ((index < 0) || (index >= size)) return null;
88     else if (index == 0) return removeFirst();
89     else if (index == size - 1) return removeLast();
90     else {
91         Node previous = first;
92
93         for (int i = 1; i < index; i++) {
94             previous = previous.next;
95         }
96
97         Node current = previous.next;
98         previous.next = current.next;
99         size--;
100        return current.element;
101    }
102 }
103
104 /** Override toString() to return elements in the list */
105 public String toString() {
106     StringBuffer result = new StringBuffer("[");
107
108     Node current = first;
109     for (int i = 0; i < size; i++) {
110         result.append(current.element);

```

```

111     current = current.next;
112     if (current != null)
113         result.append(", "); // Seperate two elements with a comma
114     else
115         result.append("]"); // Insert the closing ] in the string
116 }
117
118 return result.toString();
119 }
120
121 /** Clear the list */
122 public void clear() {
123     first = last = null;
124 }
125
126 /** Return true if this list contains the element o */
127 public boolean contains(Object o) {
128     // Implementation left as an exercise
129     return true;
130 }
131
132 /** Return the element from this list at the specified index */
133 public Object get(int index) {
134     // Implementation left as an exercise
135     return null;
136 }
137
138 /** Returns the index of the first matching element in this list.
139  * Returns -1 if no match. */
140 public int indexOf(Object o) {
141     // Implementation left as an exercise
142     return 0;
143 }
144
145 /** Returns the index of the last matching element in this list
146  * Returns -1 if no match. */
147 public int lastIndexOf(Object o) {
148     // Implementation left as an exercise
149     return 0;
150 }
151
152 /** Remove the first node that contains the specified element
153  * Return true if the element is removed
154  * Return false if no element is removed
155  */
156 public boolean remove(Object o) {
157     // Implementation left as an exercise
158     // return true;
159 }
160
161 /** Replace the element at the specified position in this list
162  * with the specified element. */
163 public Object set(int index, Object o) {
164     // Implementation left as an exercise
165     return null;
166 }
167
168 private static class Node {
169     Object element;
170     Node next;
171
172     public Node(Object o) {
173         element = o;
174     }
175 }
176 }

```

The variables `first` and `last` (Line 2) refer to the first and last nodes in the list, respectively. The `getFirst()` and `getLast()` methods (Lines 14–23) return the first and last elements in the list, respectively.

Since variable `size` is defined as `protected` in `MyAbstractList`, it can be accessed in `MyLinkedList`. When a new element is added to the list, `size` is incremented by 1, and when an element is removed from the list, `size` is decremented by 1. The `addFirst(Object o)` method (Lines 26–34) creates a new node to store the element and insert the node to the beginning of the list. After the insertion, `first` should refer to this new element node (Line 29), as shown in Figure 17.10.

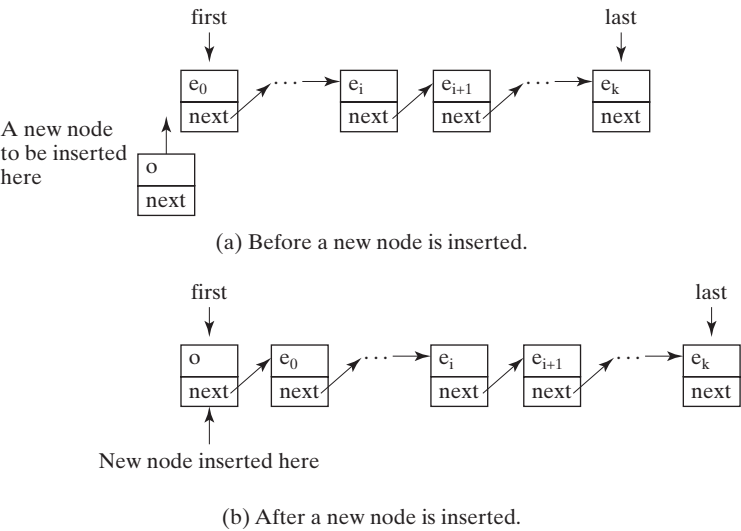


FIGURE 17.10 A new element *o* is added to the beginning of the list.

The `addLast(Object o)` method (Lines 37–47) creates a node to hold element *o* and inserts the node at the end of the list. After the insertion, `last` should refer to this new element node (Line 43), as shown in Figure 17.11.

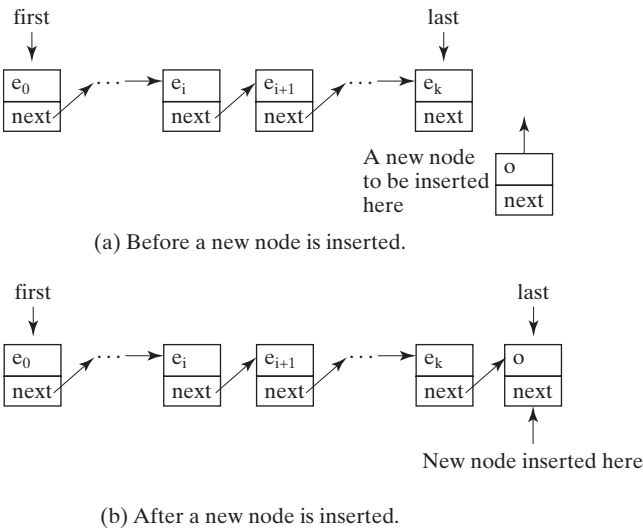


FIGURE 17.11 A new element *o* is added at the end of the list.

The `add(int index, Object o)` method (Lines 51–63) adds an element `o` to the list at the specified index. Consider three cases: (1) if `index` is 0, invoke `addFirst(o)` to insert the element at the beginning of the list; (2) if `index` is greater than or equal to `size`, invoke `addLast(o)` to insert the element at the end of the list; (3) create a new node to store the new element and locate where to insert it. As shown in Figure 17.12, the new node is to be inserted between the nodes `current` and `temp`. The method assigns the new node to `current.next` and assigns `temp` to the new node's `next`.

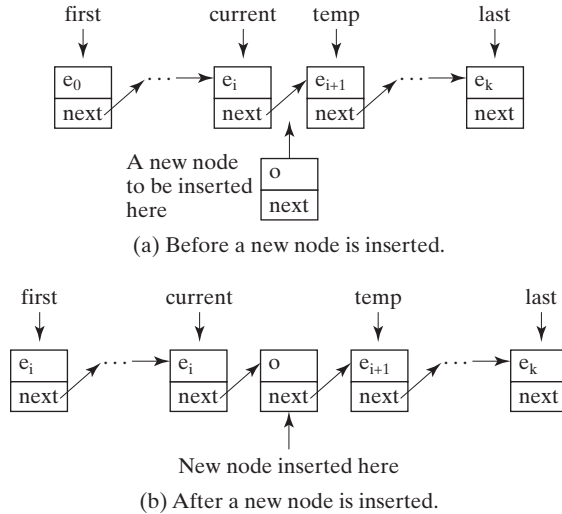


FIGURE 17.12 A new element is inserted in the middle of the list.

The `removeFirst()` method (Lines 67–75) removes the first node from the list by pointing `first` to the second node, as shown in Figure 17.13. The `removeLast()` method (Lines 79–82) removes the last node from the list. Afterwards, `last` should refer to the former second-last node.

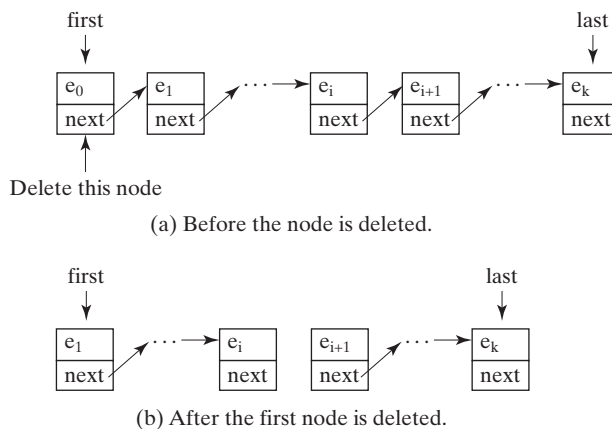


FIGURE 17.13 The first node is deleted from the list.

The `remove(int index)` method (Lines 86–102) finds the node at the specified index and then removes it. Consider four cases: (1) if `index` is beyond the range of the list (i.e., `index < 0 || index >= size`), return `null`; (2) if `index` is 0, invoke `removeFirst()` to remove the first node; (3) if `index` is `size - 1`, invoke `removeLast()` to remove the last node; (4) locate the node at the specified index. Let `current` denote this node and `previous` denote the node before this node, as shown in Figure 17.14. Assign `current.next` to `previous.next` to eliminate the current node.

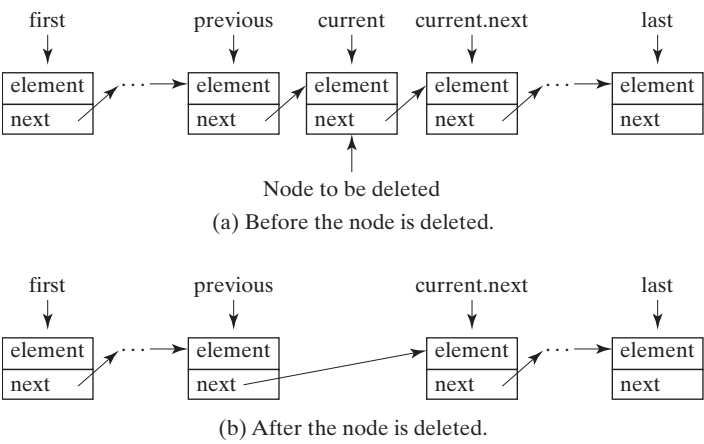


FIGURE 17.14 A node is deleted from the list.

The implementation of methods `removeLast()`, `clear()`, `contains(Object o)`, `get(int index)`, `indexOf(Object o)`, `lastIndexOf(Object o)`, `remove(Object o)`, and `set(int index, Object o)` is omitted and left as an exercise.

To test `MyLinkedList`, simply replace `MyArrayList` on Line 7 by `MyLinkedList` in `TestList` in Example 17.1. The output should be the same as shown in Figure 17.6.

17.3 Stacks and Queues

A stack can be viewed as a special type of list whose elements are accessed, inserted, and deleted only from the end (top), of the stack. A queue represents a waiting list. A queue can be viewed as a special type of list whose elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue.

Since the insertion and deletion operations on a stack are made only at the end of the stack, it is more efficient to implement a stack with an array list than with a linked list. Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list. This section implements a stack class using an array list and a queue using a linked list.

There are two ways to design stack and queue classes:

inheritance

- ◆ Using *inheritance*: You can declare a stack class by extending the array list class, and a queue class by extending the linked list class.

composition

- ◆ Using *composition*: You can declare an array list as a data field in the stack class, and a linked list as a data field in the queue class.

Both designs are fine, but using composition is better because it enables you to declare a completely new stack class and queue class without inheriting unnecessary and inappropriate methods from the array list and linked list. The stack class named `MyStack` is shown in Figure 17.15. Its implementation is given in Listing 17.6.

MyStack	
-list: <code>MyArrayList</code>	
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): Object	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the first element in the stack from the top that matches the specified element.

FIGURE 17.15 *MyStack uses an array list to provide a last-in/first-out data structure.*

LISTING 17.6 `MyStack.java` (Implementing Stack)

```

1 public class MyStack {
2     private MyArrayList list = new MyArrayList();
3
4     public boolean isEmpty() {
5         return list.isEmpty();
6     }
7
8     public int getSize() {
9         return list.size();
10    }
11
12    public Object peek() {
13        return list.get(getSize() - 1);
14    }
15
16    public Object pop() {
17        Object o = list.get(getSize() - 1);
18        list.remove(getSize() - 1);
19        return o;
20    }
21
22    public Object push(Object o) {
23        list.add(o);
24        return o;
25    }
26
27    public int search(Object o) {
28        return list.lastIndexOf(o);
29    }
30
31    public String toString() {
32        return "stack: " + list.toString();
33    }
34 }
```

array list

An array list is created to store the elements in a stack (Line 2). The `isEmpty()` method (Lines 4–6) is the same as `list.isEmpty()`. The `getSize()` method (Lines 8–10) is the same as `list.size()`. The `peek()` method (Lines 12–14) looks at the element at the top of the stack without removing it. The `pop()` method (Lines 16–20) removes the top element from the stack and returns it.

The `push(Object element)` method (Lines 22–24) adds the specified element to the stack. The `search(Object element)` method checks whether the specified element is in the stack.

The queue class named `MyQueue` is shown in Figure 17.16. Its implementation is given in Listing 17.7.

MyQueue	
-list: MyLinkedList	
+enqueue(element: Object): void	Adds an element to this queue.
+dequeue(): Object	Removes an element from this queue.
+getSize(): int	Returns the number of elements from this queue.

FIGURE 17.16 *MyQueue uses a linked list to provide a first-in/first-out data structure.*

LISTING 17.7 `MyQueue.java` (Implementing Queue)

linked list

```
1 public class MyQueue {
2     private MyLinkedList list = new MyLinkedList();
3
4     public void enqueue(Object o) {
5         list.addLast(o);
6     }
7
8     public Object dequeue() {
9         return list.removeFirst();
10    }
11
12    public int getSize() {
13        return list.size();
14    }
15
16    public String toString() {
17        return "Queue: " + list.toString();
18    }
19 }
```

A linked list is created to store the elements in a queue (Line 2). The `enqueue(Object o)` method (Lines 4–6) adds element `o` into the tail of the queue. The `dequeue()` method (Lines 8–10) removes an element from the head of the queue and returns the removed element. The `getSize()` method (Lines 12–14) returns the number of elements in the queue.

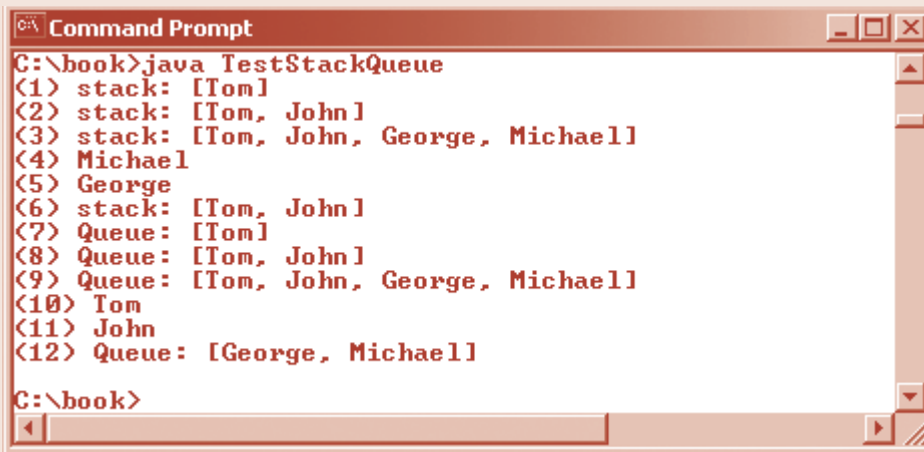
EXAMPLE 17.2 USING STACKS AND QUEUES

Problem

Write a program that creates a stack using `MyStack` and a queue using `MyQueue`. It then uses the `push` (`enqueue`) method to add strings to the stack (queue) and the `pop` (`dequeue`) method to remove strings from the stack (queue).

Solution

Listing 17.8 gives the solution to the problem. A sample run of the program is shown in Figure 17.17.

EXAMPLE 17.2 (CONTINUED)


```

C:\book>java TestStackQueue
(1) stack: [Tom]
(2) stack: [Tom, John]
(3) stack: [Tom, John, George, Michael]
(4) Michael
(5) George
(6) stack: [Tom, John]
(7) Queue: [Tom]
(8) Queue: [Tom, John]
(9) Queue: [Tom, John, George, Michael]
(10) Tom
(11) John
(12) Queue: [George, Michael]
C:\book>

```

FIGURE 17.17 *The program uses a stack and a queue to store and process strings.***LISTING 17.8** TestStackQueue.java (Using Stack and Queue)

```

1 public class TestStackQueue {
2     public static void main(String[] args) {
3         // Create a stack
4         MyStack stack = new MyStack();
5
6         // Add elements to the stack
7         stack.push("Tom"); // Push it to the stack
8         System.out.println("(1) " + stack);
9
10        stack.push("John"); // Push it to the stack
11        System.out.println("(2) " + stack);
12
13        stack.push("George"); // Push it to the stack
14        stack.push("Michael"); // Push it to the stack
15        System.out.println("(3) " + stack);
16
17        // Remove elements from the stack
18        System.out.println("(4) " + stack.pop());
19        System.out.println("(5) " + stack.pop());
20        System.out.println("(6) " + stack);
21
22        // Create a queue
23        MyQueue queue = new MyQueue();
24
25        // Add elements to the queue
26        queue.enqueue("Tom"); // Add it to the queue
27        System.out.println("(7) " + queue);
28
29        queue.enqueue("John"); // Add it to the queue
30        System.out.println("(8) " + queue);
31
32        queue.enqueue("George"); // Add it to the queue
33        queue.enqueue("Michael"); // Add it to the queue
34        System.out.println("(9) " + queue);
35
36        // Remove elements from the queue
37        System.out.println("(10) " + queue.dequeue());
38        System.out.println("(11) " + queue.dequeue());
39        System.out.println("(12) " + queue);
40    }
41 }

```

EXAMPLE 17.2 (CONTINUED)

Review

For a stack, the `push(o)` method adds an element to the top of the stack, and the `pop()` method removes the top element from the stack and returns the removed element.

For a queue, the `enqueue(o)` method adds an element to the tail of the queue, and the `dequeue()` method removes the element from the head of the queue.

17.4 Binary Trees (Optional)

A list, stack, or queue is a linear structure that consists of a sequence of elements. A binary tree is a hierarchical structure. It is either empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*. Examples of binary trees are shown in Figure 17.18.

root
left subtree
right subtree

binary search tree

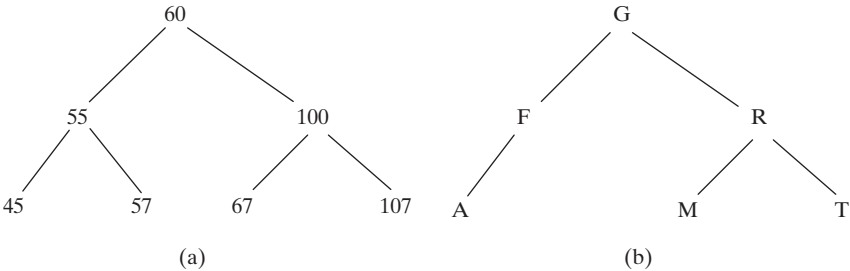


FIGURE 17.18 Each node in a binary tree has zero, one, or two branches.

The root of a left (right) subtree of a node is called a *left (right) child* of the node. A node without children is called a *leaf*. A special type of binary tree called a *binary search tree* is often useful. A binary search tree (with no duplicate elements) has the property that for every node in the tree, the value of any node in its left subtree is less than the value of the node, and the value of any node in its right subtree is greater than the value of the node. The binary trees in Figure 17.18 are all binary search trees. This section is concerned with binary search trees.

leaf
binary search tree

17.4.1 Representing Binary Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown in Figure 17.19.

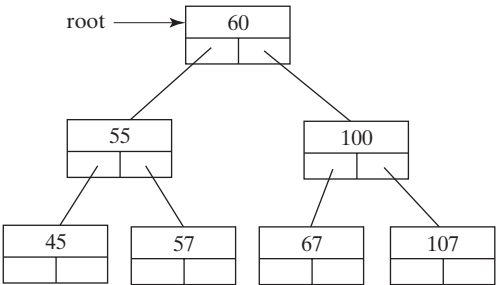


FIGURE 17.19 A binary tree can be represented using a set of linked nodes.

A node can be defined as a class, as follows:

```
class TreeNode {
    Object element;
    TreeNode left;
    TreeNode right;

    public TreeNode(Object o) {
        element = o;
    }
}
```

The variable `root` refers to the root node of the tree. If the tree is empty, `root` is `null`. The following code creates the first three nodes of the tree in Figure 17.19:

```
// Create the root node
TreeNode root = new TreeNode(new Integer(60));

// Create the left child node
root.left = new TreeNode(new Integer(55));

// Create the right child node
root.right = new TreeNode(new Integer(100));
```

17.4.2 Inserting an Element into a Binary Search Tree

If a binary tree is empty, create a root node with the new element. Otherwise, locate the parent node for the new element node. If the new element is less than the parent element, the node for the new element becomes the left child of the parent. If the new element is greater than the parent element, the node for the new element becomes the right child of the parent. Here is the algorithm:

```
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element value > the value in current.element) {
            parent = current;
            current = current.right;
        }
        else
            return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

For example, to insert 101 into the tree in Figure 17.19, the parent is the node for 107. The new node for 101 becomes the left child of the parent. To insert 59 into the tree, the parent is the node for 57. The new node for 59 becomes the right child of the parent. Both of these insertions are shown in Figure 17.20.

17.4.3 Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. This section presents *inorder*, *preorder*, *postorder*, *depth-first*, and *breadth-first* traversals.

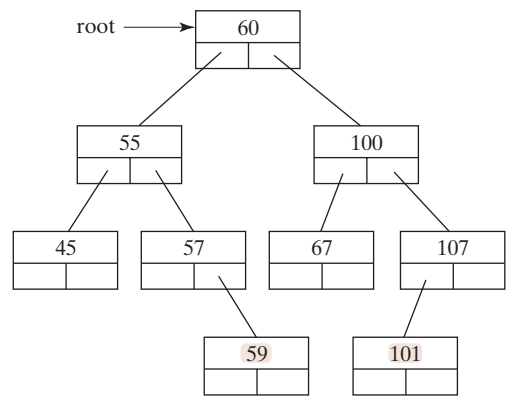


FIGURE 17.20 Two new elements are inserted into the tree.

inorder

With *inorder traversal*, the left subtree of the current node is visited first, then the current node, and finally the right subtree of the current node.

postorder

With *postorder traversal*, the left subtree of the current node is visited first, then the right subtree of the current node, and finally the current node itself.

preorder
depth-first

With *preorder traversal*, the current node is visited first, then the left subtree of the current node, and finally the right subtree of the current node. *Depth-first traversal* is the same as preorder traversal.

breadth-first

With *breadth-first traversal*, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on.

For example, in the tree in Figure 17.20, the inorder is 45 55 57 59 60 67 100 101 107. The postorder is 45 59 57 55 67 101 107 100 60. The preorder is 60 55 45 57 59 100 67 107 101. The breadth-first traversal is 60 55 100 45 57 67 107 59 101.

17.4.4 The Binary Tree Class

Let us define a binary tree class named `BinaryTree` with `insert`, `inorder traversal`, `postorder traversal`, and `preorder traversal`, as shown in Figure 17.21. Its implementation is given in Listing 17.9.

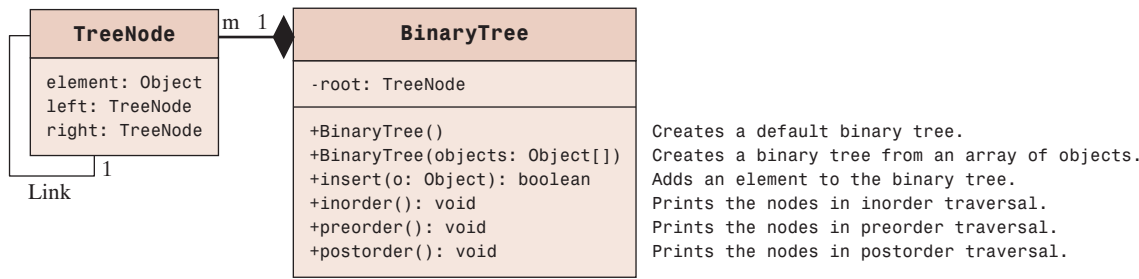


FIGURE 17.21 `BinaryTree` implements a binary tree with operations `insert`, `inorder`, `preorder`, and `postorder`.

LISTING 17.9 `BinaryTree.java` (Implementing Binary Tree)

no-arg constructor

```
1 public class BinaryTree {
2     private TreeNode root;
3
4     /** Create a default binary tree */
5     public BinaryTree() {
6     }
7 }
```

```

8  /** Create a binary tree from an array of objects */
9  public BinaryTree(Object[] objects) {
10     for (int i = 0; i < objects.length; i++)
11         insert(objects[i]);
12 }
13
14 /** Insert element o into the binary tree
15  * Return true if the element is inserted successfully */
16 public boolean insert(Object o) {
17     if (root == null)
18         root = new TreeNode(o); // Create a new root
19     else {
20         // Locate the parent node
21         TreeNode parent = null;
22         TreeNode current = root;
23         while (current != null)
24             if (((Comparable)o).compareTo(current.element) < 0) {
25                 parent = current;
26                 current = current.left;
27             }
28             else if (((Comparable)o).compareTo(current.element) > 0) {
29                 parent = current;
30                 current = current.right;
31             }
32             else
33                 return false; // Duplicate node not inserted
34
35         // Create the new node and attach it to the parent node
36         if (((Comparable)o).compareTo(parent.element) < 0)
37             parent.left = new TreeNode(o);
38         else
39             parent.right = new TreeNode(o);
40     }
41
42     return true; // Element inserted
43 }
44
45 /** Inorder traversal */
46 public void inorder() {
47     inorder(root);
48 }
49
50 /** Inorder traversal from a subtree */
51 private void inorder(TreeNode root) {
52     if (root == null) return;
53     inorder(root.left);
54     System.out.print(root.element + " ");
55     inorder(root.right);
56 }
57
58 /** Postorder traversal */
59 public void postorder() {
60     postorder(root);
61 }
62
63 /** Postorder traversal from a subtree */
64 private void postorder(TreeNode root) {
65     if (root == null) return;
66     postorder(root.left);
67     postorder(root.right);
68     System.out.print(root.element + " ");
69 }
70
71 /** Preorder traversal */
72 public void preorder() {
73     preorder(root);
74 }
75
76 /** Preorder traversal from a subtree */
77 private void preorder(TreeNode root) {
78     if (root == null) return;

```

constructor

insert

inorder

postorder

preorder

tree node

```

79     System.out.print(root.element + " ");
80     preorder(root.left);
81     preorder(root.right);
82 }
83
84 /** Inner class tree node */
85 private static class TreeNode {
86     Object element;
87     TreeNode left;
88     TreeNode right;
89
90     public TreeNode(Object o) {
91         element = o;
92     }
93 }
94 }

```

The `insert(Object o)` method (Lines 16–43) creates a node for element `o` and inserts it into the tree. If the tree is empty, the node becomes the root. Otherwise, the method finds an appropriate parent for the node to maintain the order of the tree. If the element is already in the tree, the method returns `false`; otherwise it returns `true`.

The `inorder()` method (Lines 46–48) invokes `inorder(root)` to traverse the entire tree. The method `inorder(TreeNode root)` traverses the tree with the specified root. This is a recursive method. It recursively traverses the left subtree, then the root, and finally the right subtree. The traversal ends when the tree is empty.

The `postorder()` method (Lines 59–61) and the `preorder()` method (Lines 72–74) are implemented similarly using recursion.

EXAMPLE 17.3 USING BINARY TREES

Problem

Write a program that creates a binary tree using `BinaryTree`. Add strings into the binary tree and traverse the tree in `inorder`, `postorder`, and `preorder`.

Solution

Listing 17.10 gives the solution to the problem. A sample run of the program is shown in Figure 17.22.



FIGURE 17.22 The program creates a tree, inserts elements into it, and displays them in `inorder`, `postorder`, and `preorder`.

LISTING 17.10 TestBinaryTree.java (Using `BinaryTree`)

```

1 public class TestBinaryTree {
2     public static void main(String[] args) {
3         BinaryTree tree = new BinaryTree();
4         tree.insert("George");
5         tree.insert("Michael");
6         tree.insert("Tom");
7         tree.insert("Adam");
8         tree.insert("Jones");
9         tree.insert("Peter");

```

create tree
insert

EXAMPLE 17.3 (CONTINUED)

```

10  tree.insert("Daniel");
11  System.out.print("Inorder: ");
12  tree.inorder();
13  System.out.print("\nPostorder: ");
14  tree.postorder();
15  System.out.print("\nPreorder: ");
16  tree.preorder();
17  }
18  }

```

inorder
postorder
preorder

Review

After all the elements are inserted, the tree should appear as shown in Figure 17.23.

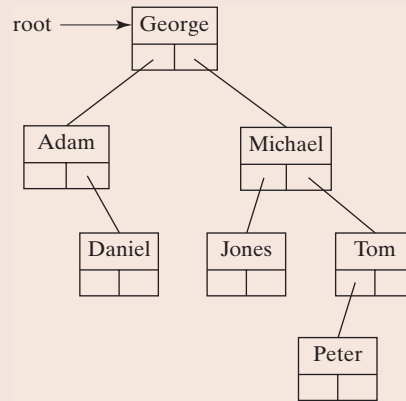


FIGURE 17.23 A binary search tree is pictured here after Line 10 is executed.

If the elements are inserted in a different order (e.g., Daniel, Adam, Jones, Peter, Tom, Michael, George), the tree will look different. However, the inorder is the same as long as the set of elements is the same.

KEY TERMS

binary search tree 638

binary tree 638

data structure 620

dynamic data structure 620

inorder traversal 640

list 620

object-oriented data structure 620

postorder traversal 640

preorder traversal 640

queue 620

stack 620

tree traversal 639

CHAPTER SUMMARY

- ◆ You have learned the concept of object-oriented data structures and how to create lists, stacks, queues, and binary trees using object-oriented approach.
- ◆ To define a data structure is essentially to declare a class. The class for a data structure should use data fields to store data and provide methods to support such operations as insertion and deletion.
- ◆ To create a data structure is to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure, such as inserting an element to the data structure or deleting an element from the data structure.

REVIEW QUESTIONS

Sections 17.1–17.2

- 17.1** What is a data structure? What is an object-oriented data structure?
- 17.2** What are the limitations of the array data type?
- 17.3** `MyArrayList` is implemented using an array, and an array is a fixed-size data structure. Why is `MyArrayList` considered as a dynamic data structure?
- 17.4** What are the benefits of defining both the `MyList` interface and the `MyAbstractList` class? What is a convenience class?
- 17.5** What is wrong in the following code?
- ```
MyArrayList list = new MyArrayList();
list.add(100);
```
- 17.6** What is wrong if Lines 11–12 in `MyArrayList.java`
- ```
data = objects;
size = objects.length;
```
- are replaced by
- ```
super(objects);
```
- 17.7** If the number of elements in the program is fixed, what data structure should you use? If the number of elements in the program changes, what data structure should you use?
- 17.8** If you have to add or delete the elements anywhere in a list, should you use `ArrayList` or `LinkedList`?

### Section 17.3 Stacks and Queues

- 17.9** You can use inheritance or composition to design the data structures for stacks and queues. Discuss the pros and cons of these two approaches.
- 17.10** Which lines of the following code are wrong?
- ```
MyList list = new MyArrayList();
list.add("Tom");
list = new MyLinkedList();
list.add("Tom");
list = new MyStack();
list.add("Tom");
```

Section 17.4 Binary Trees

- 17.11** If a set of the same elements is inserted into a binary tree in two different orders, will the two corresponding binary trees look the same? Will the inorder traversal be the same? Will the postorder traversal be the same? Will the preorder traversal be the same?

PROGRAMMING EXERCISES

Section 17.2 Lists

- 17.1** (*Adding set operations in `MyAbstractList`*) Add and implement the following methods in `MyAbstractList`:
- ```
/** Add the elements in otherList to this list.
 * Returns true if this list changed as a result of the call */
public boolean addAll(MyList otherList)
```



```

/** Remove all the elements in otherList from this list
 * Returns true if this list changed as a result of the call */
public boolean removeAll(MyList otherList)

/** Retain the elements in this list if they are also in otherList
 * Returns true if this list changed as a result of the call */
public boolean retainAll(MyList otherList)

```

Write a test program that creates two `MyArrayList`s, `list1` and `list2`, with the initial values {"Tom", "George", "Peter", "Jean", "Jane"} and {"Tom", "George", "Michael", "Michelle", "Daniel"}, then invokes `list1.addAll(list2)`, `list1.removeAll(list2)`, and `list1.retainAll(list2)`, and displays the resulting new `list1`.

**17.2\*** (*Completing the implementation of `MyLinkedList`*) The implementations of methods `removeLast()`, `contains(Object o)`, `get(int index)`, `indexOf(Object o)`, `lastIndexOf(Object o)`, `remove(Object o)`, and `set(int index, Object o)` are omitted in the text. Implement these methods.

**17.3\*** (*Creating a two-way linked list*) The `MyLinkedList` class in Listing 17.5 is a one-way directional linked list that enables one-way traversal of the list. Modify the `Node` class to add the new field `name previous` to refer to the previous node in the list, as follows:

```

public class TreeNode {
 Object element;
 TreeNode next;
 TreeNode previous;

 public TreeNode(Object o) {
 element = o;
 }
}

```

Simplify the implementation of the `add(Object element, int index)` method and the `remove(int index)` and `remove(Object element)` to take advantage of the bi-directional linked list.

### Section 17.3 Stacks and Queues

**17.4** (*Implementing `MyStack` using inheritance*) In Section 17.3, “Stacks and Queues,” `MyStack` is implemented using composition. Create a new stack class that extends `MyArrayList`.

**17.5** (*Implementing `MyQueue` using inheritance*) In Section 17.3, “Stacks and Queues,” `MyQueue` is implemented using composition. Create a new queue class that extends `MyLinkedList`.

### Section 17.4 Binary Trees

**17.6\*** (*Adding new methods in `BinaryTree`*) Add the following new methods in `BinaryTree`.

```

/** Search element o in this binary tree */
public boolean search(Object o)

/** Return the number of nodes in this binary tree */
public int size()

/** Return the depth of this binary tree. Depth is the
 * number of the nodes in the longest path of the tree */
public int depth()

```

**17.7\*\*** (*Implementing inorder traversal using a stack*) Implement the `inorder` method in `BinaryTree` using a stack instead of recursion.