CHAPTER

24

Implementing Lists, Stacks, Queues, and Priority Queues

Objectives

- To design common operations of lists in an interface and make the interface a subtype of `Collection` (§24.2).
- To design and implement an array list using an array (§24.3).
- To design and implement a linked list using a linked structure (§24.4).
- To design and implement a stack class using an array list and a queue class using a linked list (§24.5).
- To design and implement a priority queue using a heap (§24.6).

24.1 Introduction

Key Point

This chapter focuses on implementing data structures.

Lists, stacks, queues, and priority queues are classic data structures typically covered in a data structures course. They are supported in the Java API, and their uses were presented in Chapter 20, Lists, Stacks, Queues, and Priority Queues. This chapter will examine how these data structures are implemented under the hood. Implementation of sets and maps will be covered in Chapter 27. Through these implementations, you will gain valuable insight on data structures and learn how to design and implement custom data structures.

24.2 Common Operations for Lists

Key Point

Common operations of lists are defined in the `List` interface.

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. You can perform the following operations on a list:

- Retrieve an element from the list.
- Insert a new element into the list.
- Delete an element from the list.
- Find out how many elements are in the list.
- Determine whether an element is in the list.
- Check whether the list is empty.

There are two ways to implement a list. One is to use an *array* to store the elements. Array size is fixed. If the capacity of the array is exceeded, you need to 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 define two classes for lists. For convenience, let's name these two classes `MyArrayList` and `MyLinkedList`. These two classes have common operations but different implementations.

For an interactive demo on how array lists and linked lists work, see https://liveexample.pearsoncmg.com/dsanimation/ArrayListeBook.html and https://liveexample.pearsoncmg.com/dsanimation/LinkedListeBook.html, as shown in Figure 24.1.

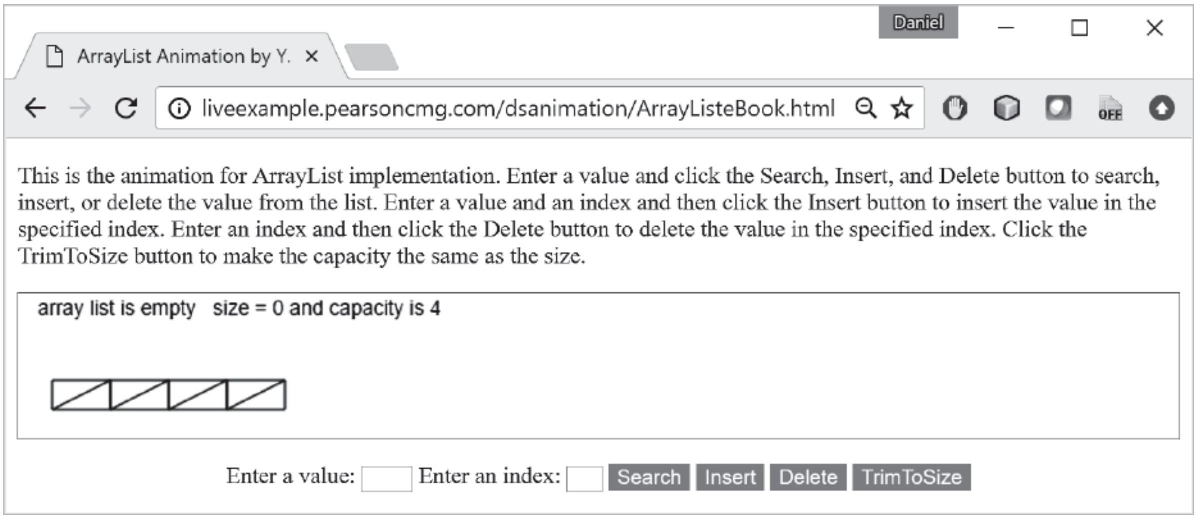
Design Guide

default methods in interfaces

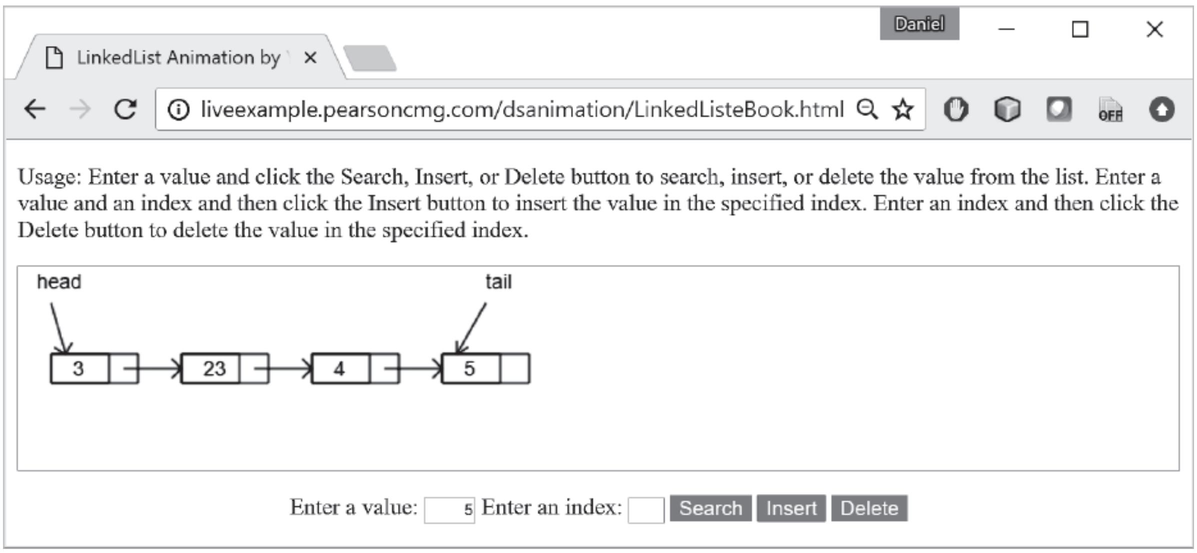
Prior to Java 8, a popular design strategy for Java data structures is to define common operations in interfaces and provide convenient abstract classes for partially implementing the interfaces. So, the concrete classes can simply extend the convenient abstract classes without implementing the full interfaces. Java 8 enables you to define default methods. You can provide default implementation for some of the methods in the interfaces rather than in convenient abstract classes. Using default methods eliminate the need for convenient abstract classes.

list animation on Companion Website

Let us name the interface `MyList` and define it as a subtype of `Collection` so the common operations in the `Collection` interface are also available in `MyList`. Figure 24.2 shows the relationship of `Collection`, `MyList`, `MyArrayList`, and `MyLinkedList`. The methods in `MyList` are shown in Figure 24.3. Listing 24.1 gives the source code for `MyList`.



(a) ArrayList animation. *Source*



(b) LinkedList animation

FIGURE 24.1 The animation tool enables you to see how array lists and linked lists work.

FIGURE 24.2 `MyList` defines a common interface for `MyArrayList` and `MyLinkedList`.

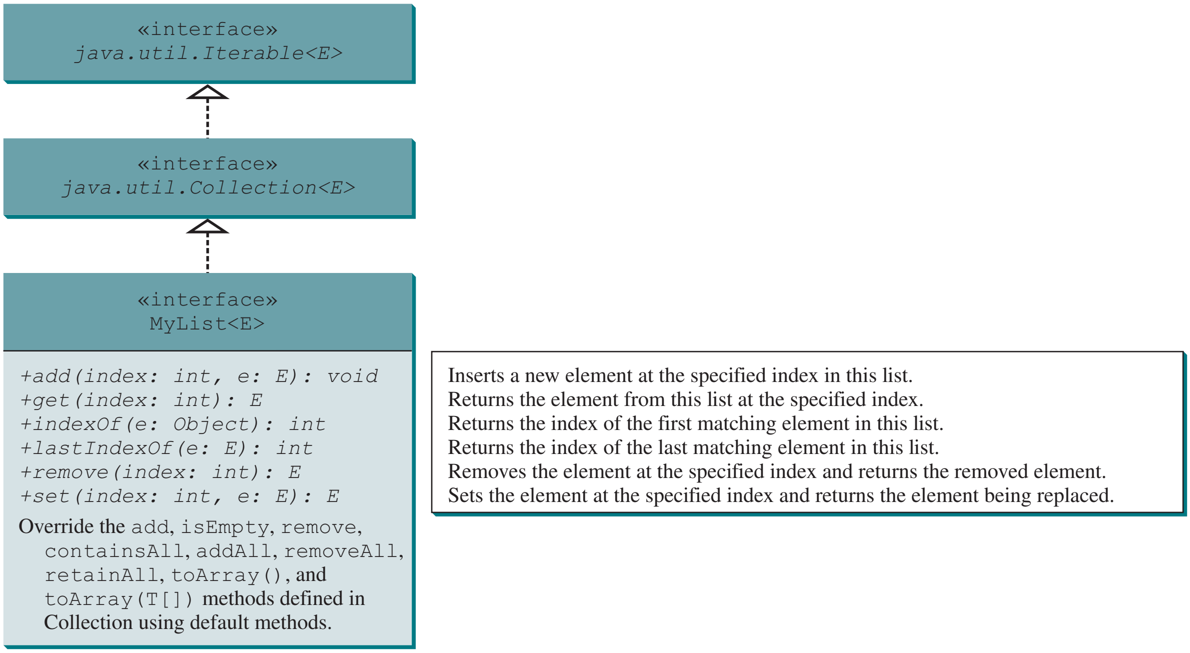


FIGURE 24.3 `MyList` defines the methods for manipulating a list and partially implements some of the methods defined in the `Collection` interface.

LISTING 24.1 MyList.java

add(index, e) · get(index) · indexOf(e) · lastIndexOf(e) · remove(e) · set(index, e) · default add(e)

```java
import java.util.Collection;

public interface MyList<E> extends Collection<E> {
  /** Add a new element at the specified index in this list */
  public void add(int index, E e);

  /** Return the element from this list at the specified index */
  public E get(int index);

  /** Return the index of the first matching element in this list.
   *  Return -1 if no match. */
  public int indexOf(Object e);

  /** Return the index of the last matching element in this list
   *  Return -1 if no match. */
  public int lastIndexOf(E e);

  /** Remove the element at the specified position in this list
   *  Shift any subsequent elements to the left.
   *  Return the element that was removed from the list. */
  public E remove(int index);

  /** Replace the element at the specified position in this list
   *  with the specified element and returns the new set. */
  public E set(int index, E e);

  @Override /** Add a new element at the end of this list */
  public default boolean add(E e) {
    add(size(), e);
    return true;
```

```java
  }

  @Override /** Return true if this list contains no elements */
  public default boolean isEmpty() {
    return size() == 0;
  }

  @Override /** Remove the first occurrence of the element e
   * from this list. Shift any subsequent elements to the left.
   * Return true if the element is removed. */
  public default boolean remove(Object e) {
    if (indexOf(e) >= 0) {
      remove(indexOf(e));
      return true;
    }
    else
      return false;
  }

  @Override
  public default boolean containsAll(Collection<?> c) {
    // Left as an exercise
    return true;
  }

  @Override
  public default boolean addAll(Collection<?  extends E> c) {
    // Left as an exercise
    return true;
  }

  @Override
  public default boolean removeAll(Collection<?> c) {
    // Left as an exercise
    return true;
  }

  @Override
  public default boolean retainAll(Collection<?> c) {
    // Left as an exercise
    return true;
  }

  @Override
  public default Object[] toArray() {
    // Left as an exercise
    return null;
  }

  @Override
  public default <T> T[] toArray(T[] array) {
    // Left as an exercise
    return null;
  }
}
```

default `isEmpty()`

implement `remove(E e)`

implement `containsAll`

implement `addAll`

implement `removeAll`

implement `retainAll`

implement `toArray()`

implement `toArray(T[])`

The methods **`isEmpty()`**, **`add(E)`**, **`remove(E)`**, **`containsAll`**, **`addAll`**, **`removeAll`**, **`retainAll`**, **`toArray()`**, and **`toArray(T[])`** are defined in the **`Collection`** interface. Since these methods are implementable in **`MyList`**, they are overridden in the **`MyList`** interface as default methods. The implementation for **`isEmpty()`**, **`add(E)`**, and **`remove(E)`** are provided and the implementation for other default methods are left as exercises in Programming Exercise 24.1.

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

Check Point

24.2.1 Suppose `list` is an instance of `MyList`, can you get an iterator for list using `list.iterator()`?

24.2.2 Can you create a list using `new MyList()`?

24.2.3 What methods in `Collection` are overridden as default methods in `MyList`?

24.2.4 What are the benefits of overriding the methods in `Collection` as default methods in `MyList`?

24.3 Array Lists

Key Point

An array list is implemented using an array.

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.

Initially, an array, say `data` of `E[]` type, is created with a default size. When inserting a new element into the array, first make sure 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 24.4.

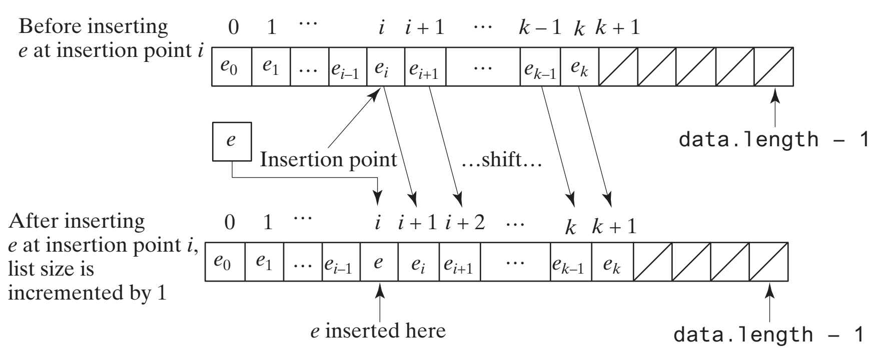


FIGURE 24.4 Inserting a new element into the array requires that all the elements after the insertion point be shifted one position to the right, so the new element can be inserted at the insertion point.

> **Note**
> The data array is of type `E[]`. 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 24.5.

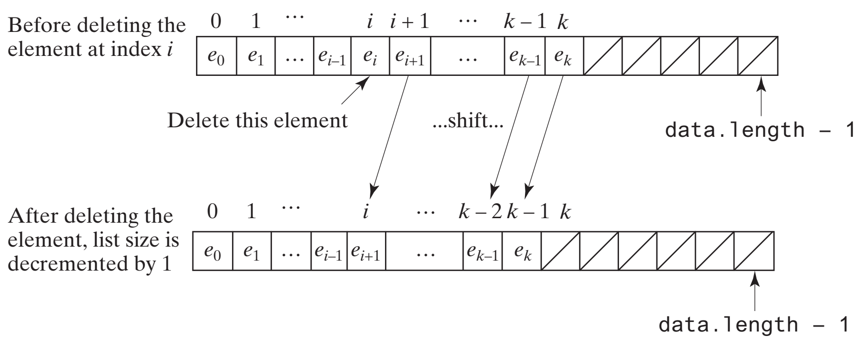


FIGURE 24.5 Deleting an element from the array requires that all the elements after the deletion point be shifted one position to the left.

`MyArrayList` uses an array to implement `MyList`, as shown in Figure 24.6. Its implementation is given in Listing 24.2.

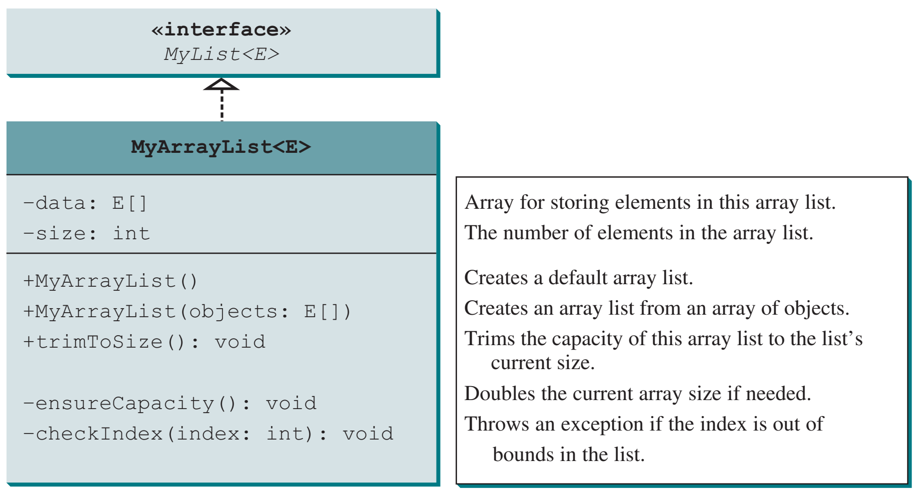


FIGURE 24.6 `MyArrayList` implements a list using an array.

LISTING 24.2 MyArrayList.java

```java
public class MyArrayList<E> implements MyList<E> {
  public static final int INITIAL_CAPACITY = 16;
  private E[] data = (E[])new Object[INITIAL_CAPACITY];
  private int size = 0; // Number of elements in the list

  /** Create an empty list */
  public MyArrayList() {
  }

  /** Create a list from an array of objects */
```

initial capacity
create an array
number of elements
no-arg constructor

constructor

```
  public MyArrayList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]); // Warning: don't use super(objects)!
  }

  @Override /** Add a new element at the specified index */
```

add

```
  public void add(int index, E e) {
    // Ensure the index is in the right range
    if (index < 0 || index > size)
      throw new IndexOutOfBoundsException
        ("Index: " + index + ", Size: " + size);

    ensureCapacity();

    // Move the elements to the right after the specified index
    for (int i = size - 1; i >= index; i--)
      data[i + 1] = data[i];

    // Insert new element to data[index]
    data[index] = e;

    // Increase size by 1
    size++;
  }

  /** Create a new larger array, double the current size + 1 */
```

ensureCapacity

```
  private void ensureCapacity() {
    if (size >= data.length) {
```

double capacity + 1

```
      E[] newData = (E[])(new Object[size * 2 + 1]);
      System.arraycopy(data, 0, newData, 0, size);
      data = newData;
    }
  }

  @Override /** Clear the list */
```

clear

```
  public void clear() {
    data = (E[])new Object[INITIAL_CAPACITY];
    size = 0;
  }

  @Override /** Return true if this list contains the element */
```

contains

```
  public boolean contains(Object e) {
    for (int i = 0; i < size; i++)
      if (e.equals(data[i])) return true;

    return false;
  }

  @Override /** Return the element at the specified index */
```

get

```
  public E get(int index) {
    checkIndex(index);
    return data[index];
  }
```

checkIndex

```
  private void checkIndex(int index) {
    if (index < 0 || index >= size)
      throw new IndexOutOfBoundsException
        ("Index: " + index + ", Size: " + size);
  }
```

```java
@Override /** Return the index of the first matching element
 * in this list. Return -1 if no match. */
public int indexOf(Object e) {
  for (int i = 0; i < size; i++)
    if (e.equals(data[i]))  return i;

  return -1;
}

@Override /** Return the index of the last matching element
 * in this list. Return -1 if no match. */
public int lastIndexOf(E e) {
  for (int i = size - 1; i >= 0; i--)
    if (e.equals(data[i])) return i;

  return -1;
}

@Override /** Remove the element at the specified position
 * in this list. Shift any subsequent elements to the left.
 * Return the element that was removed from the list. */
public E remove(int index) {
  checkIndex(index);

  E e = data[index];

  // Shift data to the left
  for (int j = index; j < size - 1; j++)
    data[j] = data[j + 1];

  data[size - 1] = null; // This element is now null

  // Decrement size
  size--;

  return e;
}

@Override /** Replace the element at the specified position
 * in this list with the specified element. */
public E set(int index, E e) {
  checkIndex(index);
  E old = data[index];
  data[index] = e;
  return old;
}

@Override
public String toString() {
  StringBuilder result = new StringBuilder("[");

  for (int i = 0; i < size; i++) {
    result.append(data[i]);
    if (i < size - 1) result.append(", ");
  }

  return result.toString() + "]";
}

/** Trims the capacity to current size */
```

indexOf

lastIndexOf

remove

set

toString

trimToSize

```
131    public void trimToSize() {
132      if (size != data.length) {
133        E[] newData = (E[])(new Object[size]);
134        System.arraycopy(data, 0, newData, 0, size);
135        data = newData;
136      } // If size == capacity, no need to trim
137    }
138
139    @Override /** Override iterator() defined in Iterable */
```

iterator

```
140    public java.util.Iterator<E> iterator() {
141      return new ArrayListIterator();
142    }
143
144    private class ArrayListIterator
145        implements java.util.Iterator<E> {
146      private int current = 0; // Current index
147
148      @Override
149      public boolean hasNext() {
150        return current < size;
151      }
152
153      @Override
154      public E next() {
155        return data[current++];
156      }
157
158      @Override // Remove the element returned by the last next()
159      public void remove() {
160        if (current == 0)  // next() has not been called yet
161          throw new IllegalStateException();
162        MyArrayList.this.remove(--current);
163      }
164    }
165
166    @Override /** Return the number of elements in this list */
```

size

```
167    public int size() {
168      return size;
169    }
170  }
```

The constant **INITIAL_CAPACITY** (line 2) is used to create an initial array **data** (line 3). Owing to generics type erasure (see Restriction 2 in Section 19.8), you cannot create a generic array using the syntax **new e[INITIAL_CAPACITY]**. To circumvent this limitation, an array of the **Object** type is created in line 3 and cast into **E[]**. The **size** data field tracks the number of elements in the list (line 4).

add

The **add(int index, E e)** method (lines 17–34) inserts the element **e** at the specified **index** in the array. This method first invokes **ensureCapacity()** (line 23), 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 26 and 27). After the element is added, **size** is incremented by **1** (line 33).

ensureCapacity

The **ensureCapacity()** method (lines 37–43) checks whether the array is full. If so, the program creates a new array that doubles the current array size + **1**, copies the current array to the new array using the **System.arraycopy** method, and sets the new array as the current array. Note the current size might be **0** after invoking the **trimToSize()** method. **new Object[2 * size + 1]** (line 39) ensures that the new size is not **0**.

clear

The **clear()** method (lines 46–49) creates a new array using the size as **INITIAL_CAPACITY** and resets the variable **size** to **0**. The class will work if line 47 is deleted. However, the class will have a memory leak because the elements are still in the array, although they are no longer needed. By creating a new array and assigning it to **data**, the old array and the elements stored in the old array become garbage, which will be automatically collected by the JVM.

contains

The **contains(Object e)** method (lines 52–57) checks whether element **e** is contained in the array by comparing **e** with each element in the array using the **equals** method.

The **get(int index)** method (lines 60–63) checks if **index** is within the range and returns **data[index]** if **index** is in the range.

checkIndex

The **checkIndex(int index)** method (lines 65–69) checks if **index** is within the range. If not, the method throws an **IndexOutOfBoundsException** (line 67).

indexOf

The **indexOf(Object e)** method (lines 73–78) compares element **e** with the elements in the array, starting from the first one. If a match is found, the index of the element is returned; otherwise, **–1** is returned.

lastIndexOf

The **lastIndexOf(Object e)** method (lines 82–87) compares element **e** with the elements in the array, starting from the last one. If a match is found, the index of the element is returned; otherwise, **–1** is returned.

remove

The **remove(int index)** method (lines 92–107) shifts all the elements after the index one position to the left (lines 98 and 99) and decrements **size** by **1** (line 104). The last element is not used anymore and is set to **null** (line 101).

set

The **set(int index, E e)** method (lines 111–116) simply assigns **e** to **data[index]** to replace the element at the specified index with element **e**.

toString

The **toString()** method (lines 119–128) overrides the **toString** method in the **Object** class to return a string representing all the elements in the list.

trimToSize

The **trimToSize()** method (lines 131–137) creates a new array whose size matches the current array-list size (line 133), copies the current array to the new array using the **System.arraycopy** method (line 134), and sets the new array as the current array (line 135). Note if **size == capacity**, there is no need to trim the size of the array.

iterator

The **iterator()** method defined in the **java.lang.Iterable** interface is implemented to return an instance on **java.util.Iterator** (lines 140–142). The **ArrayListIterator** class implements **Iterator** with concrete methods for **hasNext**, **next**, and **remove** (lines 144–164). It uses **current** to denote the current position of the element being traversed (line 146).

size()

The **size()** method simply returns the number of elements in the array list (lines 167–169).

Listing 24.3 gives an example that creates a list using **MyArrayList**. It uses the **add** method to add strings to the list, and the **remove** method to remove strings. Since **MyArrayList** implements **Iterable**, the elements can be traversed using a foreach loop (lines 35 and 36).

LISTING 24.3 TestMyArrayList.java

```
1  public class TestMyArrayList {
2    public static void main(String[] args) {
3      // Create a list
4      MyList<String> list = new MyArrayList<>();                create a list
5
6      // Add elements to the list
7      list.add("America"); // Add it to the list
8      System.out.println("(1) " + list);                       add to list
9
10     list.add(0, "Canada"); // Add it to the beginning of the list
11     System.out.println("(2) " + list);
12
13     list.add("Russia"); // Add it to the end of the list
14     System.out.println("(3) " + list);
15
16     list.add("France"); // Add it to the end of the list
17     System.out.println("(4) " + list);
```

remove from list

use iterator

```

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);

    // Remove elements from the list
    list.remove("Canada"); // Same as list.remove(0) in this case
    System.out.println("(7) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(8) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.print("(9) " + list + "\n(10) ");

    for (String s: list)
      System.out.print(s.toUpperCase() + " ");
  }
}
```

```
(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [America, Germany, Russia, France, Norway]
(8) [America, Germany, France, Norway]
(9) [America, Germany, France]
(10) AMERICA GERMANY FRANCE
```

Check Point

24.3.1 What are the limitations of the array data type?

24.3.2 `MyArrayList` is implemented using an array, and an array is a fixed-size data structure. Why is `MyArrayList` considered a dynamic data structure?

24.3.3 Show the length of the array in `MyArrayList` after each of the following statements is executed:

```
MyArrayList<Double> list = new MyArrayList<>();
list.add(1.5);
list.trimToSize();
list.add(3.4);
list.add(7.4);
list.add(17.4);
```

24.3.4 What is wrong if lines 11 and 12 in Listing 24.2, MyArrayList.java,

```
for (int i = 0; i < objects.length; i++)
  add(objects[i]);
```

are replaced by

```
data = objects;
size = objects.length;
```

24.3.5 If you change the code in line 33 in Listing 24.2, MyArrayList.java, from

```
E[] newData = (E[])(new Object[size * 2 + 1]);
```

to

```
E[] newData = (E[])(new Object[size * 2]);
```

the program is incorrect. Can you find the reason?

24.3.6 Will the `MyArrayList` class have memory leak if the following code in line 41 is deleted?

```
data = (E[])new Object[INITIAL_CAPACITY];
```

24.3.7 The **`get(index)`** method invokes the **`checkIndex(index)`** method (lines 59–63 in Listing 24.2) to throw an **`IndexOutOfBoundsException`** if the index is out of bounds. Suppose the **`add(index, e)`** method is implemented as follows:

```
public void add(int index, E e) {
  checkIndex(index);

  // Same as lines 23-33 in Listing 24.2 MyArrayList.java
}
```

What will happen if you run the following code?

```
MyArrayList<String> list = new MyArrayList<>();
list.add("New York");
```

24.4 Linked Lists

A linked list is implemented using a linked structure.

Key Point

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

24.4.1 Nodes

In a linked list, each element is contained in an object, called the *node*. When a new element is added to the list, a node is created to contain it. Each node is linked to its next neighbor, as shown in Figure 24.7.

A node can be created from a class defined as follows:

```
class Node<E> {
  E element;
  Node<E> next;

  public Node(E e) {
    element = e;
  }
}
```

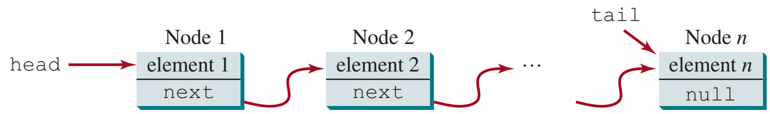


Figure 24.7 A linked list consists of any number of nodes chained together.

We use the variable **head** to refer to the first node in the list and the variable **tail** to the last node. If the list is empty, both **head** and **tail** are **null**. Here is an example that creates a linked list to hold three nodes. Each node stores a string element.

Step 1: Declare **head** and **tail**.

```
Node<String> head = null;     The list is empty now
Node<String> tail = null;
```

head and **tail** are both **null**. The list is empty.

Step 2: Create the first node and append it to the list, as shown in Figure 24.8. After the first node is inserted in the list, **head** and **tail** point to this node.

```
head = new Node<>("Chicago");
tail = head;
```

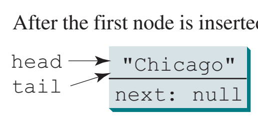
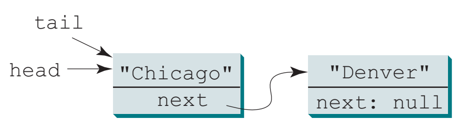


Figure 24.8 Append the first node to the list.

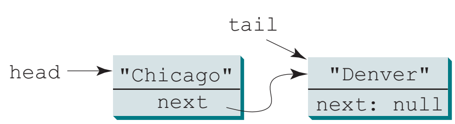
Step 3: Create the second node and append it into the list, as shown in Figure 24.9a. To append the second node to the list, link the first node with the new node. The new node is now the tail node, so you should move **tail** to point to this new node, as shown in Figure 24.9b.

```
tail.next = new Node<>("Denver");
```



(a)

```
tail = tail.next;
```



(b)

Figure 24.9 Append the second node to the list.

Step 4: Create the third node and append it to the list, as shown in Figure 24.10a. To append the new node to the list, link the last node in the list with the new node. The new node is now the tail node, so you should move **tail** to point to this new node, as shown in Figure 24.10b.

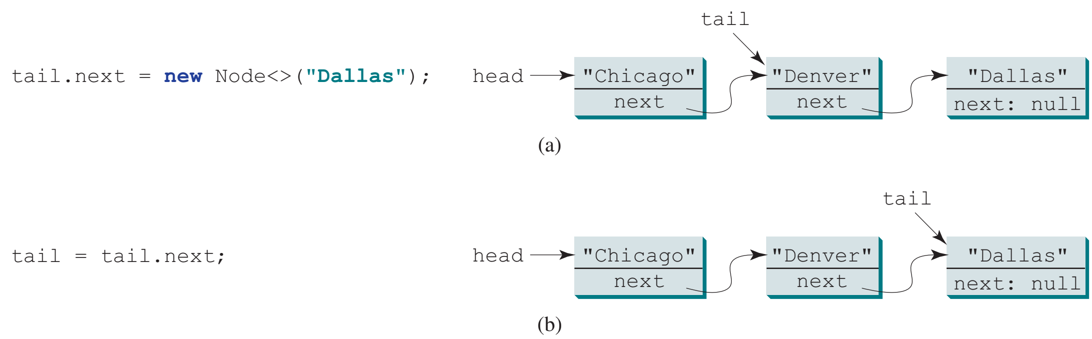


FIGURE 24.10 Append the third node to the list.

Each node contains the element and a data field named **next** that points to the next element. If the node is the last in the list, its pointer data field **next** contains the value **null**. You can use this property to detect the last node. For example, you can write the following loop to traverse all the nodes in the list:

```
1 Node<E> current = head;
2 while (current != null) {
3   System.out.println(current.element);
4   current = current.next;
5 }
```

current pointer
check last node
next node

The variable **current** points initially to the first node in the list (line 1). In the loop, the element of the current node is retrieved (line 3) then **current** points to the next node (line 4). The loop continues until the current node is **null**.

24.4.2 The MyLinkedList Class

The **MyLinkedList** class uses a linked structure to implement a dynamic list. It implements **MyList**. In addition, it provides the methods **addFirst**, **addLast**, **removeFirst**, **removeLast**, **getFirst**, and **getLast**, as shown in Figure 24.11.

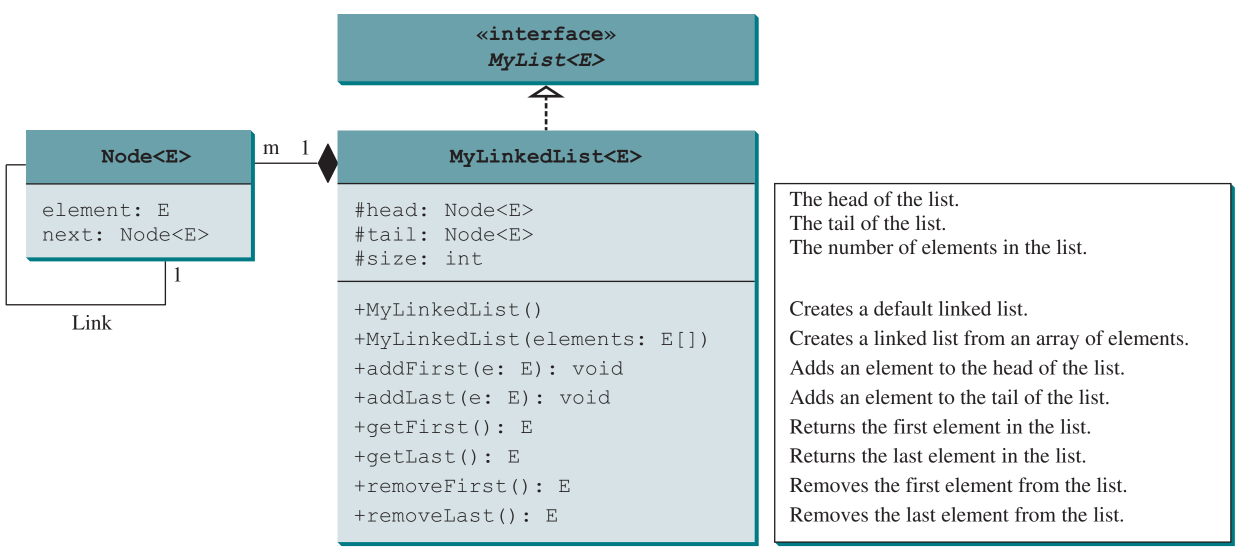


FIGURE 24.11 **MyLinkedList** implements a list using a linked list of nodes.

Assuming the class has been implemented, Listing 24.4 gives a test program that uses the class.

Listing 24.4 TestMyLinkedList.java

create list
append element
print list
insert element
append element
append element
insert element
insert element
insert element
remove element
remove element
remove element
traverse using iterator

```java
public class TestMyLinkedList {
  /** Main method */
  public static void main(String[] args) {
    // Create a list for strings
    MyLinkedList<String> list = new MyLinkedList<>();

    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);

    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);

    list.addLast("France"); // Add it to the end of the list
    System.out.println("(4) " + list);

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);

    list.add(0, "Poland"); // Same as list.addFirst("Poland")
    System.out.println("(7) " + list);

    // Remove elements from the list
    list.remove(0); // Same as list.remove("Poland") in this case
    System.out.println("(8) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(9) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.print("(10) " + list + "\n(11) ");

    for (String s: list)
      System.out.print(s.toUpperCase() + " ");

    list.clear();
    System.out.println("\nAfter clearing the list, the list size is "
      + list.size());
  }
}
```

```
(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [Poland, Canada, America, Germany, Russia, France, Norway]
```

```
(8) [Canada, America, Germany, Russia, France, Norway]
(9) [Canada, America, Russia, France, Norway]
(10) [Canada, America, Russia, France]
(11) CANADA AMERICA RUSSIA FRANCE
After clearing the list, the list size is 0
```

24.4.3 Implementing `MyLinkedList`

Now let us turn our attention to implementing the **`MyLinkedList`** class. We will discuss how to implement the methods **`addFirst`**, **`addLast`**, **`add(index, e)`**, **`removeFirst`**, **`removeLast`**, and **`remove(index)`** and leave the other methods in the **`MyLinkedList`** class as exercises.

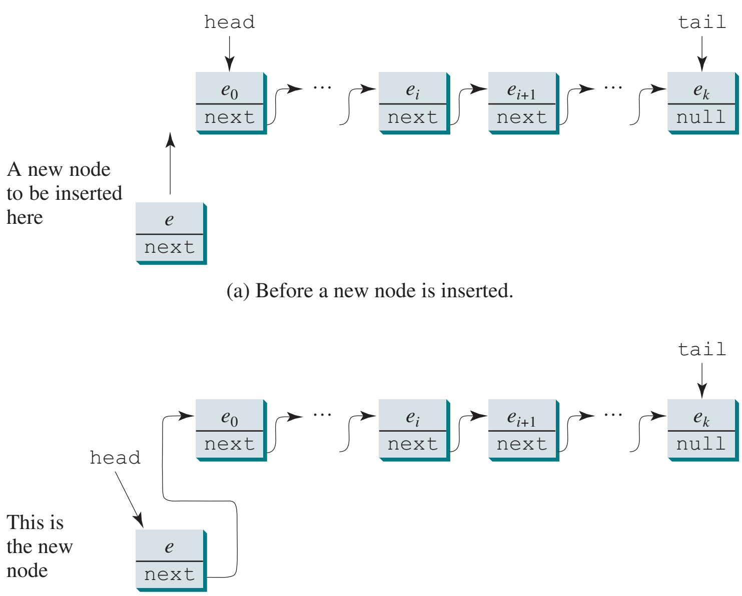
24.4.3.1 Implementing `addFirst(e)`

The **`addFirst(e)`** method creates a new node for holding element **`e`**. The new node becomes the first node in the list. It can be implemented as follows:

```
1 public void addFirst(E e) {
2   Node<E> newNode = new Node<>(e); // Create a new node
3   newNode.next = head; // link the new node with the head
4   head = newNode; // head points to the new node
5   size++; // Increase list size
6
7   if (tail == null) // The new node is the only node in list
8     tail = head;
9 }
```

create a node
link with head
head to new node
increase size
was empty?

The **`addFirst(e)`** method creates a new node to store the element (line 2) and inserts the node at the beginning of the list (line 3), as shown in Figure 24.12a. After the insertion, **`head`** should point to this new element node (line 4), as shown in Figure 24.12b.



(a) Before a new node is inserted.

(b) After a new node is inserted.

FIGURE 24.12 A new element is added to the beginning of the list.

If the list is empty (line 7), both **`head`** and **`tail`** will point to this new node (line 8). After the node is created, **`size`** should be increased by **`1`** (line 5).

24.4.3.2 Implementing `addLast(e)`

The `addLast(e)` method creates a node to hold the element and appends the node at the end of the list. It can be implemented as follows:

create a node

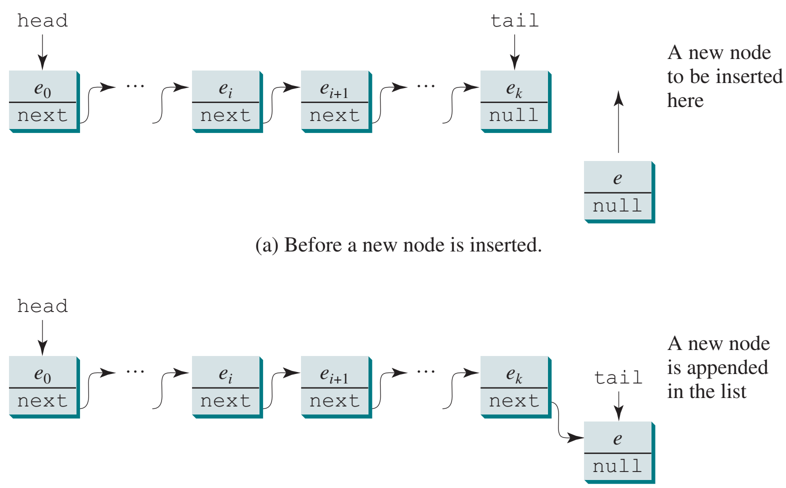
increase size

```
 1  public void addLast(E e) {
 2    Node<E> newNode = new Node<>(e); // Create a new node for e
 3
 4    if (tail == null) {
 5      head = tail = newNode; // The only node in list
 6    }
 7    else {
 8      tail.next = newNode; // Link the new node with the last node
 9      tail = newNode; // tail now points to the last node
10    }
11
12    size++; // Increase size
13  }
```

The `addLast(e)` method creates a new node to store the element (line 2) and appends it to the end of the list. Consider two cases:

1. If the list is empty (line 4), both `head` and `tail` will point to this new node (line 5);
2. Otherwise, link the node with the last node in the list (line 8). `tail` should now point to this new node (line 9). Figures 24.13a and 13b show the new node for element `e` before and after the insertion.

In any case, after the node is created, the `size` should be increased by `1` (line 12).



(a) Before a new node is inserted.

(b) After a new node is inserted.

FIGURE 24.13 A new element is added at the end of the list.

24.4.3.3 Implementing `add(index, e)`

The `add(index, e)` method inserts an element into the list at the specified index. It can be implemented as follows:

insert first
insert last

```
public void add(int index, E e) {
  if (index == 0) addFirst(e); // Insert first
  else if (index >= size) addLast(e); // Insert last
  else { // Insert in the middle
    Node<E> current = head;
```

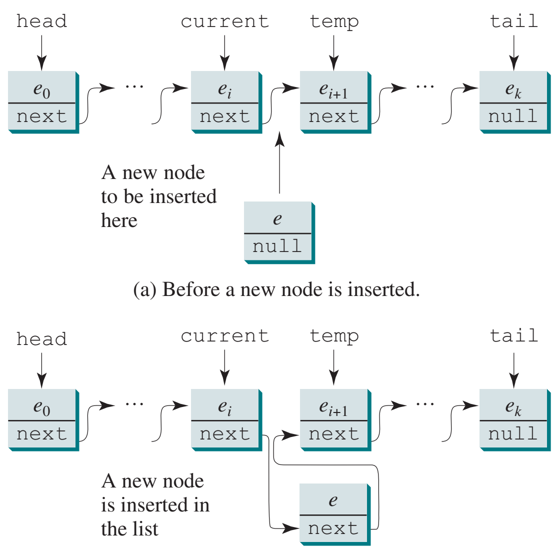
```
6       for (int i = 1; i < index; i++)
7         current = current.next;
8       Node<E> temp = current.next;
9       current.next = new Node<>(e);
10      (current.next).next = temp;
11      size++;
12    }
13  }
```

create a node

increase size

There are three cases when inserting an element into the list:

1. If **index** is **0**, invoke **addFirst(e)** (line 2) to insert the element at the beginning of the list.
2. If **index** is greater than or equal to **size**, invoke **addLast(e)** (line 3) to insert the element at the end of the list.
3. Otherwise, create a new node to store the new element and locate where to insert it. As shown in Figure 24.14a, 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**, as shown in Figure 24.14b. The size is now increased by **1** (line 11).



(a) Before a new node is inserted.

(b) After a new node is inserted.

FIGURE 24.14 A new element is inserted in the middle of the list.

24.4.3.4 Implementing **removeFirst()**

The **removeFirst()** method removes the first element from the list. It can be implemented as follows:

```
public E removeFirst() {
  if (size == 0) return null; // Nothing to delete
  else {
    Node<E> temp = head; // Keep the first node temporarily
    head = head.next; // Move head to point to next node
    size--; // Reduce size by 1
    if (head == null) tail = null; // List becomes empty
```

nothing to remove

keep old head

new head

decrease size

destroy the node

```
 8       return temp.element; // Return the deleted element
 9     }
10   }
```

Consider two cases:

1. If the list is empty, there is nothing to delete, so return **`null`** (line 2).
2. Otherwise, remove the first node from the list by pointing **`head`** to the second node. Figures 24.15a and 15b show the linked list before and after the deletion. The size is reduced by **`1`** after the deletion (line 6). If the list becomes empty, after removing the element, **`tail`** should be set to **`null`** (line 7).

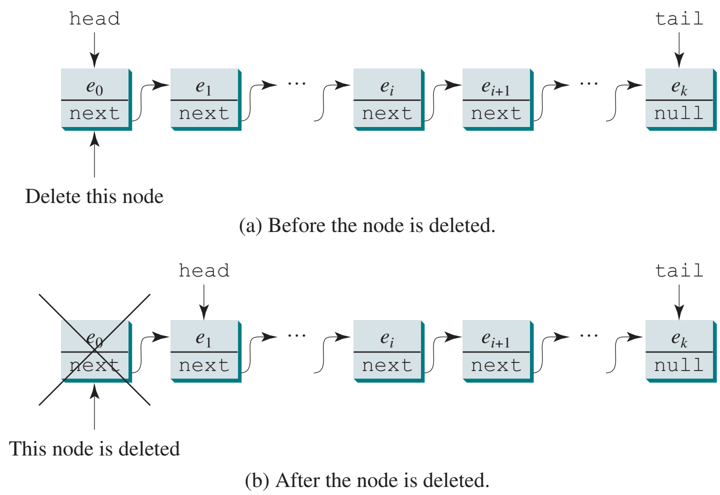


FIGURE 24.15 The first node is deleted from the list.

24.4.3.5 Implementing `removeLast()`

The **`removeLast()`** method removes the last element from the list. It can be implemented as follows:

empty? or size 1?
reduce to removeFirst()
locate current before `tail`
move `tail`
reduce `size`
return deleted element

```
public E removeLast() {
  if (size == 0 || size == 1) {
     return removeFirst();
  }
  else {
    Node<E> current = head;
    for (int i = 0; i < size - 2; i++) {
      current = current.next;
    }

    E temp = tail.element;
    tail = current;
    tail.next = null;
    size--;
    return temp;
  }
}
```

Consider three cases:

1. If the list is empty or has a single element, invoking **`removeFirst()`** will take care of this case (lines 2–4).
2. Otherwise, locate **`current`** to point to the second-to-last node (lines 6–9). Save the value of tail to **`temp`** (line 11). Set **`tail`** to **`current`** (line 12). **`tail`** is now repositioned to point to the second-to-last node and destroy the last node (line 13). The size is reduced by **`1`** after the deletion (line 14) and the element value of the deleted node is returned (line 15).

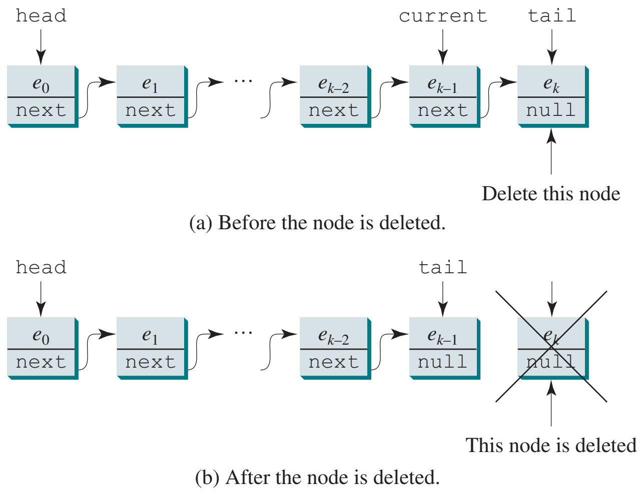


FIGURE 24.16 The last node is deleted from the list.

24.4.3.6 Implementing `remove(index)`

The **`remove(index)`** method finds the node at the specified index then removes it. It can be implemented as follows:

```
1  public E remove(int index) {
2    if (index < 0 || index >= size) return null; // Out of range
3    else if (index == 0) return removeFirst(); // Remove first
4    else if (index == size - 1) return removeLast(); // Remove last
5    else {
6      Node<E> previous = head;
7
8      for (int i = 1; i < index; i++) {
9        previous = previous.next;
10     }
11
12     Node<E> current = previous.next;
13     previous.next = current.next;
14     size--;
15     return current.element;
16   }
17 }
```

out of range
remove first
remove last
locate previous
locate current
remove from list
reduce size
return element

Consider four cases:

1. If **`index`** is beyond the range of the list (i.e., **`index < 0 || index >= size`**), return **`null`** (line 2).

2. If `index` is `0`, invoke `removeFirst()` to remove the first node (line 3).
3. If `index` is `size - 1`, invoke `removeLast()` to remove the last node (line 4).
4. Otherwise, locate the node at the specified `index`. Let `current` denote this node and `previous` denote the node before this node, as shown in Figure 24.17a. Assign `current.next` to `previous.next` to eliminate the current node, as shown in Figure 24.17b.

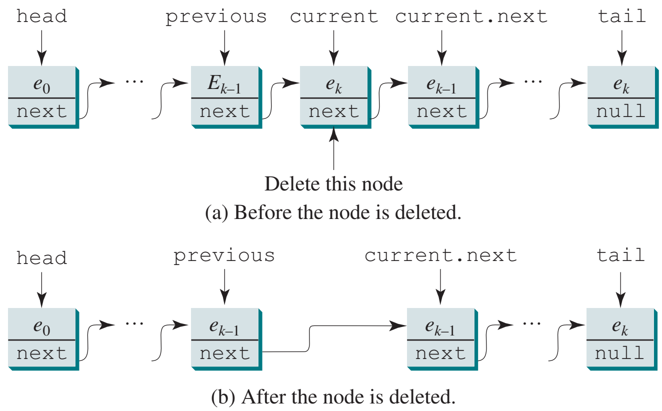


FIGURE 24.17 A node is deleted from the list.

iterator

Listing 24.5 gives the implementation of `MyLinkedList`. The implementation of `get(index)`, `indexOf(e)`, `lastIndexOf(e)`, `contains(e)`, and `set(index, e)` is omitted and left as an exercise. The `iterator()` method defined in the `java.lang.Iterable` interface is implemented to return an instance on `java.util.Iterator` (lines 128–130). The `LinkedListIterator` class implements `Iterator` with concrete methods for `hasNext`, `next`, and `remove` (lines 132–152). This implementation uses `current` to point to the current position of the element being traversed (line 134). Initially, `current` points to the head of the list.

LISTING 24.5 MyLinkedList.java

head, tail
number of elements
no-arg constructor
constructor
getFirst

```
1  public class MyLinkedList<E> implements MyList<E> {
2    private Node<E> head, tail;
3    private int size = 0; // Number of elements in the list
4
5    /** Create an empty list */
6    public MyLinkedList() {
7    }
8
9    /** Create a list from an array of objects */
10   public MyLinkedList(E[] objects) {
11     for (int i = 0; i < objects.length; i++)
12       add(objects[i]);
13   }
14
15   /** Return the head element in the list */
16   public E getFirst() {
17     if (size == 0) {
18       return null;
19     }
20     else {
```

```java
      return head.element;
    }
  }

  /** Return the last element in the list */
  public E getLast() {
    if (size == 0) {
      return null;
    }
    else {
      return tail.element;
    }
  }

  /** Add an element to the beginning of the list */
  public void addFirst(E e) {
    // Implemented in Section 24.4.3.1, so omitted here
  }

  /** Add an element to the end of the list */
  public void addLast(E e) {
    // Implemented in Section 24.4.3.2, so omitted here
  }

  @Override /** Add a new element at the specified index
   * in this list. The index of the head element is 0 */
  public void add(int index, E e) {
    // Implemented in Section 24.4.3.3, so omitted here
  }

  /** Remove the head node and
   * return the object that is contained in the removed node. */
  public E removeFirst() {
    // Implemented in Section 24.4.3.4, so omitted here
  }

  /** Remove the last node and
   * return the object that is contained in the removed node. */
  public E removeLast() {
    // Implemented in Section 24.4.3.5, so omitted here
  }

  @Override /** Remove the element at the specified position in this
   * list. Return the element that was removed from the list. */
  public E remove(int index) {
    // Implemented earlier in Section 24.4.3.6, so omitted
  }

  @Override /** Override toString() to return elements in the list */
  public String toString() {
    StringBuilder result = new StringBuilder("[");

    Node<E> current = head;
    for (int i = 0; i < size; i++) {
      result.append(current.element);
      current = current.next;
      if (current != null) {
        result.append(", "); // Separate two elements with a comma
      }
      else {
        result.append("]"); // Insert the closing ] in the string
```

getLast
addFirst
addLast
add
removeFirst
removeLast
remove
toString

```java
      }
    }

    return result.toString();
  }

  @Override /** Clear the list */
```

clear

```java
  public void clear() {
    size = 0;
    head = tail = null;
  }

  @Override /** Return true if this list contains the element e */
```

contains

```java
  public boolean contains(Object e) {
    // Left as an exercise
    return true;
  }

  @Override /** Return the element at the specified index */
```

get

```java
  public E get(int index) {
    // Left as an exercise
    return null;
  }

  @Override /** Return the index of the first matching element in
   * this list. Return -1 if no match. */
```

indexOf

```java
  public int indexOf(Object e) {
    // Left as an exercise
    return 0;
  }

  @Override /** Return the index of the last matching element in
   * this list. Return -1 if no match. */
```

lastIndexOf

```java
  public int lastIndexOf(E e) {
    // Left as an exercise
    return 0;
  }

  @Override /** Replace the element at the specified position
   * in this list with the specified element. */
```

set

```java
  public E set(int index, E e) {
    // Left as an exercise
    return null;
  }

  @Override /** Override iterator() defined in Iterable */
```

iterator

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

```

LinkedListIterator class

```java
  private class LinkedListIterator
      implements java.util.Iterator<E> {
    private Node<E> current = head; // Current index

    @Override
    public boolean hasNext() {
      return (current != null);
    }

    @Override
```

```
    public E next() {
      E e = current.element;
      current = current.next;
      return e;
    }

    @Override
    public void remove() {
      // Left as an exercise
    }
  }

  private static class Node<E> {
    E element;
    Node<E> next;

    public Node(E element) {
      this.element = element;
    }
  }

  @Override /** Return the number of elements in this list */
  public int size() {
    return size;
  }
}
```

Node inner class

24.4.4 MyArrayList vs. MyLinkedList

Both MyArrayList and MyLinkedList can be used to store a list. MyArrayList is implemented using an array, and MyLinkedList is implemented using a linked list. The overhead of MyArrayList is smaller than that of MyLinkedList. However, MyLinkedList is more efficient if you need to insert elements into and delete elements from the beginning of the list. Table 24.1 summarizes the complexity of the methods in MyArrayList and MyLinkedList. Note MyArrayList is the same as java.util.ArrayList, and MyLinkedList is the same as java.util.LinkedList except that MyLinkedList is implemented using singly a linked list and LinkedList is implemented using a doubly linked list. We will introduce doubly linked lists in Section 24.4.5.

Table 24.1 Time Complexities for Methods in MyArrayList and MyLinkedList

Methods	*MyArrayList*	*MyLinkedList*
add(e: E)	$O(1)$	$O(1)$
add(index: int, e: E)	$O(n)$	$O(n)$
clear()	$O(1)$	$O(1)$
contains(e: E)	$O(n)$	$O(n)$
get(index: int)	O(1)	O(n)
indexOf(e: E)	$O(n)$	$O(n)$
isEmpty()	$O(1)$	$O(1)$
lastIndexOf(e: E)	$O(n)$	$O(n)$
remove(e: E)	$O(n)$	$O(n)$
size()	$O(1)$	$O(1)$
remove(index: int)	$O(n)$	$O(n)$
set(index: int, e: E)	$O(n)$	$O(n)$
addFirst(e: E)	O(n)	O(1)
removeFirst()	O(n)	O(1)

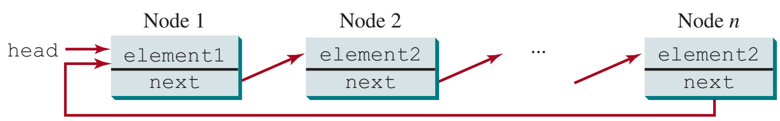
Note that you can implement `MyLinkedList` without using the `size` data field. But then the `size()` method would take `O(n)` time.

> **Note**
> The `MyArrayList` is implemented using an array. In a `MyLinkedList`, every element is wrapped in an object. `MyArrayList` has less overhead than `MyLinkedList`. You should use `MyLinkedList` only if the application involves frequently inserting/deleting the elements from the beginning of a list.

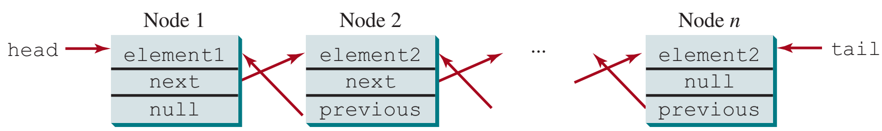
24.4.5 Variations of Linked Lists

The linked list introduced in the preceding sections is known as a *singly linked list*. It contains a pointer to the list's first node, and each node contains a pointer to the next node sequentially. Several variations of the linked list are useful in certain applications.

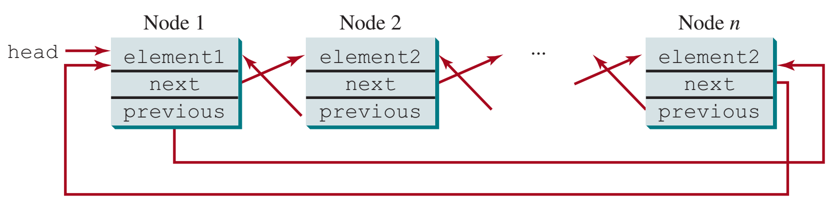
A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node, as shown in Figure 24.18a. Note `tail` is not needed for circular linked lists. `head` points to the current node in the list. Insertion and deletion take place at the current node. A good application of a circular linked list is in the operating system that serves multiple users in a timesharing fashion. The system picks a user from a circular list and grants a small amount of CPU time then moves on to the next user in the list.



(a) Circular linked list



(b) Doubly linked list



(c) Circular doubly linked list

FIGURE 24.18 Linked lists may appear in various forms.

A *doubly linked list* contains nodes with two pointers. One points to the next node and the other to the previous node, as shown in Figure 24.18b. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. Thus, a doubly linked list can be traversed forward and backward. The `java.util.LinkedList` class is implemented using a doubly linked list, and it supports traversing of the list forward and backward using the `ListIterator`.

A *circular*, *doubly linked list* is like a doubly linked list, except that the forward pointer of the last node points to the first node, and the backward pointer of the first pointer points to the last node, as shown in Figure 24.18c.

The implementations of these linked lists are left as exercises.

> **Note**
> In a singly linked list, `removeLast()` takes O(n) time. In a doubly linked list, `removeLast()` can be implemented to take O(1) time. The `LinkedList` in the Java API is implemented using a doubly linked list. See CheckPoint 24.4.11.

Check Point

24.4.1 If a linked list does not contain any nodes, what are the values in `head` and `tail`?

24.4.2 If a linked list has only one node, is `head == tail` true? List all cases in which `head == tail` is true.

24.4.3 Draw a diagram to show the linked list after each of the following statements is executed:

```
MyLinkedList<Double> list = new MyLinkedList<>();
list.add(1.5);
list.add(6.2);
list.add(3.4);
list.add(7.4);
list.remove(1.5);
list.remove(2);
```

24.4.4 When a new node is inserted to the head of a linked list, will the `head` and the `tail` be changed?

24.4.5 When a new node is appended to the end of a linked list, will the `head` and the `tail` be changed?

24.4.6 Simplify the code in lines 77–82 in Listing 24.5 using a conditional expression.

24.4.7 What is the time complexity of the `addFirst(e)` and `removeFirst()` methods in `MyLinkedList`?

24.4.8 Suppose you need to store a list of elements. 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?

24.4.9 If you have to add or delete the elements at the beginning of a list, should you use `MyArrayList` or `MyLinkedList`? If most of the operations on a list involve retrieving an element at a given index, should you use `MyArrayList` or `MyLinkedList`?

24.4.10 Both `MyArrayList` and `MyLinkedList` are used to store a list of objects. Why do we need both types of lists?

24.4.11 Implement the `removeLast()` method in a doubly linked list in O(1) time.

24.5 Stacks and Queues

Key Point

Stacks can be implemented using array lists and queues can be implemented using linked lists.

A stack can be viewed as a special type of list whose elements are accessed, inserted, and deleted only from the end (top), as shown in Figure 10.11. A queue represents a waiting list. It 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), as shown in Figure 24.19.

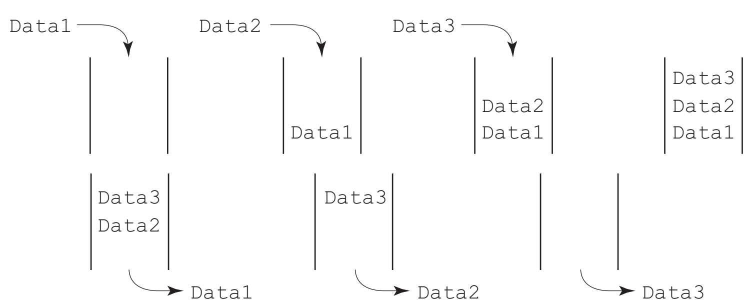
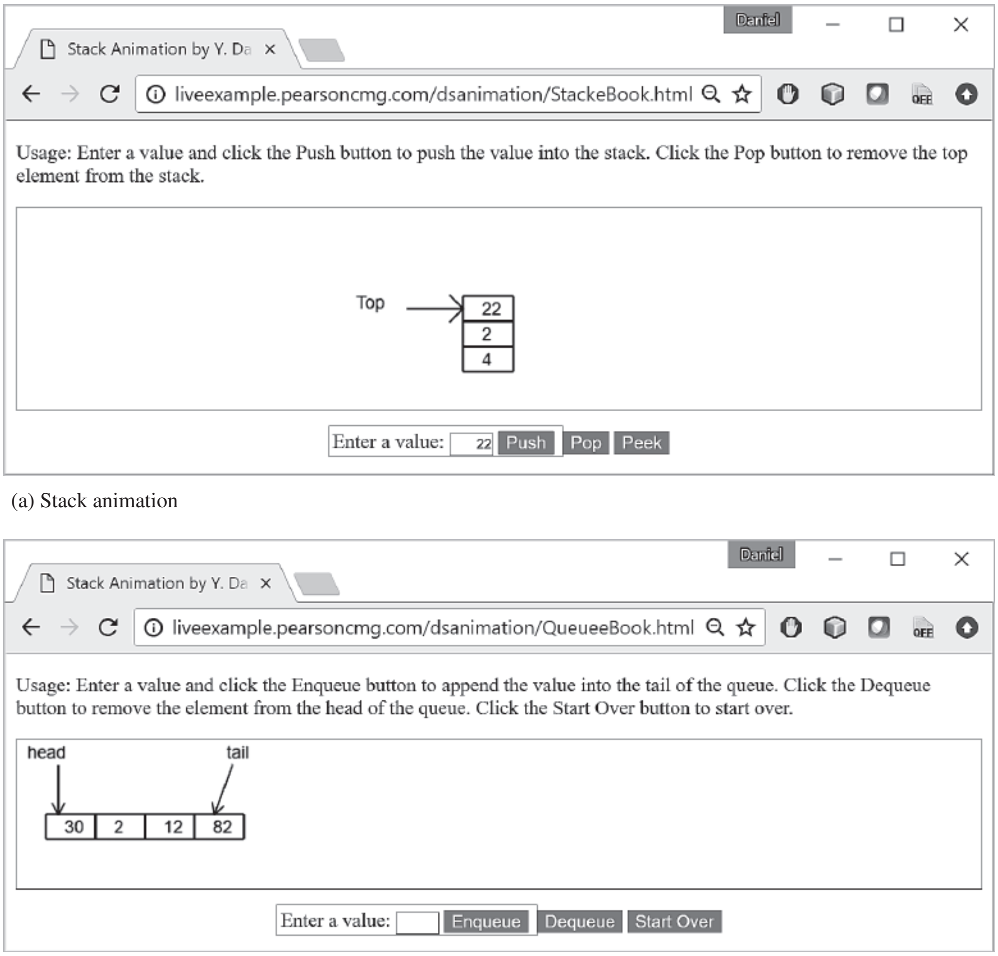


FIGURE 24.19 A queue holds objects in a first-in, first-out fashion.

stack and queue animation on Companion Website

Pedagogical Note

For an interactive demo on how stacks and queues work, go to liveexample.pearsoncmg.com/dsanimation/StackeBook.html, and liveexample.pearsoncmg.com/dsanimation/QueueeBook.html, as shown in Figure 24.20.



(a) Stack animation

(b) Queue animation

FIGURE 24.20 The animation tool enables you to see how queues work. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

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 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 class using a linked list.

There are two ways to design the stack and queue classes:

inheritance

- Using inheritance: You can define a stack class by extending `ArrayList`, and a queue class by extending `LinkedList`, as shown in Figure 24.21a.

composition

- Using composition: You can define an array list as a data field in the stack class and a linked list as a data field in the queue class, as shown in Figure 24.21b.

Both designs are fine, but using composition is better because it enables you to define a completely new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list. The implementation of the stack class using the

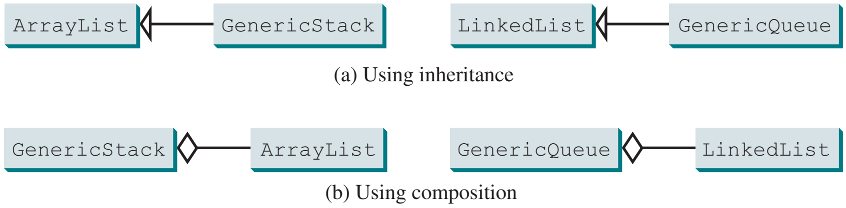


Figure 24.21 `GenericStack` and `GenericQueue` may be implemented using inheritance or composition.

composition approach was given in Listing 19.1, GenericStack.java. Listing 24.6 implements the `GenericQueue` class using the composition approach. Figure 24.22 shows the UML of the class.

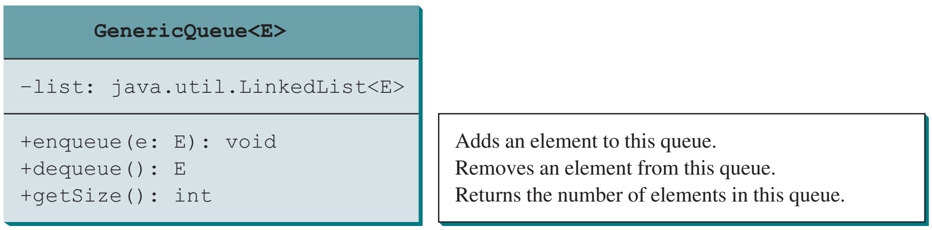


Figure 24.22 `GenericQueue` uses a linked list to provide a first-in, first-out data structure.

Listing 24.6 GenericQueue.java

```
1  public class GenericQueue<E> {
2    private java.util.LinkedList<E> list
3      = new java.util.LinkedList<>();
4
5    public void enqueue(E e) {
6      list.addLast(e);
7    }
8
9    public E dequeue() {
10     return list.removeFirst();
11   }
12
13   public int getSize() {
14     return list.size();
15   }
16
17   @Override
18   public String toString() {
19     return "Queue: " + list.toString();
20   }
21 }
```

linked list (line 2)
`enqueue` (line 5)
`dequeue` (line 9)
`getSize` (line 13)
`toString` (line 18)

A linked list is created to store the elements in a queue (lines 2 and 3). The `enqueue(e)` method (lines 5–7) adds element `e` into the tail of the queue. The `dequeue()` method (lines 9–11) removes an element from the head of the queue and returns the removed element. The `getSize()` method (lines 13–15) returns the number of elements in the queue.

Listing 24.7 gives an example that creates a stack using `GenericStack` and a queue using `GenericQueue`. It uses the `push` (`enqueue`) method to add strings to the stack (queue) and the `pop` (`dequeue`) method to remove strings from the stack (queue).

LISTING 24.7 TestStackQueue.java

```java
public class TestStackQueue {
  public static void main(String[] args) {
    // Create a stack
    GenericStack<String> stack = new GenericStack<>();

    // Add elements to the stack
    stack.push("Tom"); // Push Tom to the stack
    System.out.println("(1) " + stack);

    stack.push("Susan"); // Push Susan to the the stack
    System.out.println("(2) " + stack);

    stack.push("Kim"); // Push Kim to the stack
    stack.push("Michael"); // Push Michael to the stack
    System.out.println("(3) " + stack);

    // Remove elements from the stack
    System.out.println("(4) " + stack.pop());
    System.out.println("(5) " + stack.pop());
    System.out.println("(6) " + stack);

    // Create a queue
    GenericQueue<String> queue = new GenericQueue<>();

    // Add elements to the queue
    queue.enqueue("Tom"); // Add Tom to the queue
    System.out.println("(7) " + queue);

    queue.enqueue("Susan"); // Add Susan to the queue
    System.out.println("(8) " + queue);

    queue.enqueue("Kim"); // Add Kim to the queue
    queue.enqueue("Michael"); // Add Michael to the queue
    System.out.println("(9) " + queue);

    // Remove elements from the queue
    System.out.println("(10) " + queue.dequeue());
    System.out.println("(11) " + queue.dequeue());
    System.out.println("(12) " + queue);
  }
}
```

```
(1) stack: [Tom]
(2) stack: [Tom, Susan]
(3) stack: [Tom, Susan, Kim, Michael]
(4) Michael
(5) Kim
(6) stack: [Tom, Susan]
(7) Queue: [Tom]
(8) Queue: [Tom, Susan]
(9) Queue: [Tom, Susan, Kim, Michael]
(10) Tom
(11) Susan
(12) Queue: [Kim, Michael]
```

stack time complexity

For a stack, the **push(e)** 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. It is easy to see that the time complexity for the **push** and **pop** methods is $O(1)$.

For a queue, the **enqueue(e)** method adds an element to the tail of the queue, and the **dequeue()** method removes the element from the head of the queue. It is easy to see that the time complexity for the **enqueue** and **dequeue** methods is $O(1)$.

queue time complexity

Check Point

24.5.1 You can use inheritance or composition to design the data structures for stacks and queues. Discuss the pros and cons of these two approaches.

24.5.2 If **LinkedList** is replaced by **ArrayList** in lines 2 and 3 in Listing 24.6, GenericQueue.java, what will be the time complexity for the **enqueue** and **dequeue** methods.

24.5.3 Which lines of the following code are wrong?

```
1  List<String> list = new ArrayList<>();
2  list.add("Tom");
3  list = new LinkedList<>();
4  list.add("Tom");
5  list = new GenericStack<>();
6  list.add("Tom");
```

24.6 Priority Queues

Priority queues can be implemented using heaps.

Key Point

An ordinary queue is a first-in, first-out data structure. Elements are appended to the end of the queue and removed from the beginning. In a *priority queue*, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first. For example, the emergency room in a hospital assigns priority numbers to patients; the patient with the highest priority is treated first.

A priority queue can be implemented using a heap, in which the root is the object with the highest priority in the queue. Heaps were introduced in Section 23.6, Heap Sort. The class diagram for the priority queue is shown in Figure 24.23. Its implementation is given in Listing 24.8.

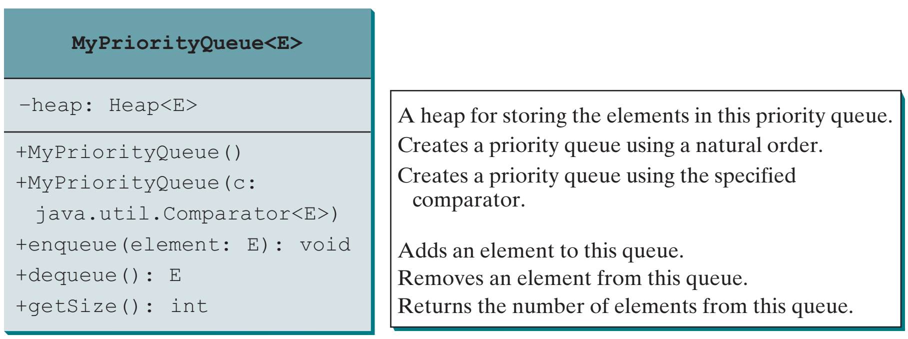


FIGURE 24.23 **MyPriorityQueue** uses a heap to store the elements.

LISTING 24.8 MyPriorityQueue.java

```
public class MyPriorityQueue<E> {
  private Heap<E> heap;                                  // heap for priority queue

  public void MyPriorityQueue<E> {                       // no-arg constructor
    heap.add(new Heap<E>();
  }

  public MyPriorityQueue(java.util.Comparator<E> c) {    // constructor
    heap = new Heap<E>(c);
  }

  public void enqueue(E newObject) {                     // enqueue
```

dequeue

getsize

```java
    heap.add(newObject);
  }

  public E dequeue() {
    return heap.remove();
  }

  public int getSize() {
    return heap.getSize();
  }
}
```

Listing 24.9 gives an example of using a priority queue for patients. The **Patient** class is defined in lines 19–37. Four patients are created with associated priority values in lines 3–6. Line 8 creates a priority queue. The patients are enqueued in lines 10–13. Line 16 dequeues a patient from the queue.

LISTING 24.9 TestPriorityQueue.java

create a patient

create a priority queue

add to queue

remove from queue

inner class Patient

compareTo

```java
1  public class TestPriorityQueue {
2    public static void main(String[] args) {
3      Patient patient1 = new Patient("John", 2);
4      Patient patient2 = new Patient("Jim", 1);
5      Patient patient3 = new Patient("Tim", 5);
6      Patient patient4 = new Patient("Cindy", 7);
7
8      MyPriorityQueue<Patient> priorityQueue
9        = new MyPriorityQueue<>();
10     priorityQueue.enqueue(patient1);
11     priorityQueue.enqueue(patient2);
12     priorityQueue.enqueue(patient3);
13     priorityQueue.enqueue(patient4);
14
15     while (priorityQueue.getSize() > 0)
16       System.out.print(priorityQueue.dequeue() + " ");
17   }
18
19   static class Patient implements Comparable<Patient> {
20     private String name;
21     private int priority;
22
23     public Patient(String name, int priority) {
24       this.name = name;
25       this.priority = priority;
26     }
27
28     @Override
29     public String toString() {
30       return name + "(priority:" + priority + ")";
31     }
32
33     @Override
34     public int compareTo(Patient patient) {
35       return this.priority - patient.priority;
36     }
37   }
38 }
```

```
Cindy(priority:7) Tim(priority:5) John(priority:2) Jim(priority:1)
```

24.6.1 What is a priority queue?

24.6.2 What are the time complexity of the `enqueue`, `dequeue`, and `getSize` methods in `MyPriorityQueue`?

24.6.3 Which of the following statements are wrong?

```
MyPriorityQueue<Object> q1 = new MyPriorityQueue<>();
MyPriorityQueue<Number> q2 = new MyPriorityQueue<>();
MyPriorityQueue<Integer> q3 = new MyPriorityQueue<>();
MyPriorityQueue<Date> q4 = new MyPriorityQueue<>();
MyPriorityQueue<String> q5 = new MyPriorityQueue<>();
```

Chapter Summary

1. You learned how to implement array lists, linked lists, stacks, and queues.

2. To define a data structure is essentially to define a class. The class for a data structure should use data fields to store data and provide methods to support operations such as insertion and deletion.

3. 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 into the data structure or deleting an element from the data structure.

4. You learned how to implement a priority queue using a heap.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

24.1 (*Implement set operations in* `MyList`) The implementations of the methods `addAll`, `removeAll`, `retainAll`, `toArray()`, and `toArray(T[])` are omitted in the `MyList` interface. Implement these methods. Test your new `MyList` class using the code at https://liveexample.pearsoncmg.com/test/Exercise24_01.txt.

***24.2** (*Implement* `MyLinkedList`) The implementations of the methods `contains(E e)`, `get(int index)`, `indexOf(E e)`, `lastIndexOf(E e)`, and `set(int index, E e)` are omitted in the `MyLinkedList` class. Implement these methods. Define a new class named `MyLinkedListExtra` that extends `MyLinkedList` to override these methods. Test your new `MyList` class using the code at https://liveexample.pearsoncmg.com/test/Exercise24_02.txt.

***24.3** (*Implement a doubly linked list*) The `MyLinkedList` class used in Listing 24.5 is a one-way directional linked list that enables one-way traversal of the list. Modify the `Node` class to add the new data field name `previous` to refer to the previous node in the list, as follows:

```
public class Node<E> {
  E element;
  Node<E> next;
  Node<E> previous;

  public Node(E e) {
    element = e;
  }
}
```

Implement a new class named **TwoWayLinkedList** that uses a doubly linked list to store elements. Define **TwoWayLinkedList** to implements **MyList**. You need to implement all the methods defined in **MyLinkedList** as well as the methods **listIterator()** and **listIterator(int index)**. Both return an instance of **java.util.ListIterator<E>** (see Figure 20.4). The former sets the cursor to the head of the list and the latter to the element at the specified index. Test your new class using this code from https://liveexample.pearsoncmg.com/test/Exercise24_03.txt.

24.4 (*Use the **GenericStack** class*) Write a program that displays the first 50 prime numbers in descending order. Use a stack to store the prime numbers.

24.5 (*Implement **GenericQueue** using inheritance*) In Section 24.5, Stacks and Queues, **GenericQueue** is implemented using composition. Define a new queue class that extends **java.util.LinkedList**.

****24.6** (*Revise **MyPriorityQueue***) Listing 24.8, uses a heap to implement the priority queue. Revise the implementation using a sorted array list to store the elements and name the new class **PriorityQueueUsingSortedArrayList**. The elements in the array list are sorted in increasing order of their priority with the last element having the highest priority. Write a test program that generates 5 million integers and enqueues them to the priority and dequeues from the queue. Use the same numbers for **MyPriorityQueue** and **PriorityQueueUsingSortedArraList** and display their execution times.

****24.7** (*Animation: linked list*) Write a program to animate search, insertion, and deletion in a linked list, as shown in Figure 24.1b. The *Search* button searches the specified value in the list. The *Delete* button deletes the specified value from the list. The *Insert* button appends the value into the list if the index is not specified; otherwise, it inserts the value into the specified index in the list.

***24.8** (*Animation: array list*) Write a program to animate search, insertion, and deletion in an array list, as shown in Figure 24.1a. The *Search* button searches the specified value in the list. The *Delete* button deletes the specified value from the list. The *Insert* button appends the value into the list if the index is not specified; otherwise, it inserts the value into the specified index in the list.

***24.9** (*Animation: array list in slow motion*) Improve the animation in the preceding programming exercise by showing the insertion and deletion operations in a slow motion.

***24.10** (*Animation: stack*) Write a program to animate push and pop in a stack, as shown in Figure 24.20a.

***24.11** (*Animation: doubly linked list*) Write a program to animate search, insertion, and deletion in a doubly linked list, as shown in Figure 24.24. The *Search* button searches the specified value in the list. The *Delete* button deletes the specified value from the list. The *Insert* button appends the value into the list if the index is not specified; otherwise, it inserts the value into the specified index in the list. Also add two buttons named *Forward Traversal* and *Backward Traversal* for displaying the elements in a forward and backward order, respectively, using iterators, as shown in Figure 24.24. The elements are displayed in a label.

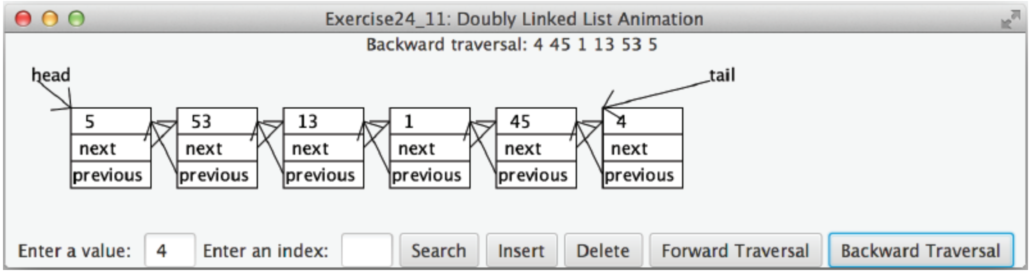


FIGURE 24.24 The program animates the work of a doubly linked list. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***24.12** (*Animation: queue*) Write a program to animate the `enqueue` and `dequeue` operations on a queue, as shown in Figure 24.20b.

***24.13** (*Fibonacci number iterator*) Define an iterator class named `FibonacciIterator` for iterating Fibonacci numbers. The constructor takes an argument that specifies the limit of the maximum Fibonacci number. For example, new `FibonacciIterator(23302)` creates an iterator that iterates Fibonacci numbers less than or equal to `23302`. Write a test program that uses this iterator to display all Fibonacci numbers less than or equal to `100000`.

***24.14** (*Prime number iterator*) Define an iterator class named `PrimeIterator` for iterating prime numbers. The constructor takes an argument that specifies the limit of the maximum prime number. For example, new `PrimeIterator(23302)` creates an iterator that iterates prime numbers less than or equal to `23302`. Write a test program that uses this iterator to display all prime numbers less than or equal to `100000`.

****24.15** (*Test* `MyArrayList`) Design and write a complete test program to test if the `MyArrayList` class in Listing 24.2 meets all requirements.

****24.16** (*Test* `MyLinkedList`) Design and write a complete test program to test if the `MyLinkedList` class in Listing 24.5 meets all requirements.