CS 240: Homework 4

Eli Zupke

November 9, 2017

The GitHub folder can be found at https://github.com/Trainzack/CS240/tree/master/Homework%204%20Priority%20Queue%20List%20and%20Sorted%20List.

Contents

Part I
Testing

1 src/Test.java

```
import java.util.Random;


public class Test {

  private static final int TEST_COUNT = 20;

  public static void main(String[] args) {

    // Note: This test is unbearably slow above 5000 entries or so.
    LinkedPriorityQueue<Integer> queue = new LinkedPriorityQueue<>();
    Random r = new Random();

    boolean sizeCorrect = true;

    for (int i = 0; i < TEST_COUNT; i++) {

      if (queue.getSize() != i) sizeCorrect = false;

      queue.add(new Integer(r.nextInt(TEST_COUNT)));

    }

    if (!sizeCorrect) {
      System.out.println("LinkedPriorityQueue returned incorrect size!");
    } else {
      System.out.println("LinkedPriorityQueue returned correct size!");
    }

    int errors = 0;

    // Now we need to make sure that the list is sorted.
    int min = queue.remove();
    while (!queue.isEmpty()) {
      int next = queue.remove();
      if (next > min) {
        errors++;
      } else {
        min = next;
      }
    }

    System.out.println("LinkedPriorityQueue Sorting errors found: " + errors);

    try {
      queue.remove();
      System.out.println("LinkedPriorityQueue isEmpty, but remove returns values!");
    } catch (EmptyQueueException e) {
      System.out.println("LinkedPriorityQueue threw expected EmptyQueueException.");
    }

    // Test the lists.

    ListInterface<Integer> list = new FixedArrayList<>(TEST_COUNT);
    testList(list, "FixedArrayList");
    list = new LinkedList<Integer>();
```

```
57      testList(list, "LinkedList");
58      list = new DoubleLinkedList<Integer>();
59      testList(list, "DoubleLinkedList");
60
61    }
62
63    /**
64     * Run a list of integers through a pretty good test.
65     * @param list The list we are testing
66     * @param name What the list is called
67     */
68    public static void testList(ListInterface<Integer> list, String name) {
69
70      System.out.println("");
71      System.out.println("==============================");
72      System.out.println("Testing " + name);
73
74      // Fill the list with increasing elements, then
75      // remove everything from every index.
76      boolean sequential = true;
77      boolean size = true;
78      boolean containment = true;
79
80      // An array that contains all of the numbers, so that we can test for equality better.
81      Integer[] numbers = new Integer[TEST_COUNT];
82
83      for (int i = 0; i < TEST_COUNT; i++) {
84        numbers[i] = new Integer(i);
85      }
86
87      for (int rIndex = 0; rIndex < TEST_COUNT; rIndex ++) {
88        // System.out.println("Testing " + rIndex);
89        // Fill the array with increasing numbers
90        for (int i = 0; i < TEST_COUNT; i++) {
91          list.add(numbers[i]);
92          if (list.size() != i + 1) {
93            size = false;
94          }
95        }
96
97        // All of the numbers should still be in sequence
98        for (int i = 0; i < TEST_COUNT - rIndex; i++) {
99          if (list.remove(rIndex) != numbers[i + rIndex]) {
100           sequential = false;
101         }
102       }
103
104       // It should contain all of the numbers, up to what we removed
105       for (int i = 0; i < rIndex; i++) {
106         if (!list.contains(numbers[i])) {
107           containment = false;
108         }
109       }
110
111       // It should not contain any of the numbers after what we removed.
112       for (int i = rIndex; i < TEST_COUNT; i++) {
113         if (list.contains(numbers[i])) {
114           containment = false;
115         }
116       }
117
118
119       list.clear();
120     }
121     printTestResult(sequential, name, "removal test");
122     printTestResult(containment, name, "containment test");
123     printTestResult(size, name, "size test");
124
```

```
125     list.clear();
126
127     boolean indexAdd = true;
128     boolean indexView = true;
129
130     // Add everything to the start
131     for (int i = 0; i < TEST_COUNT; i++) {
132       list.add(numbers[i], 0);
133
134       // Check to make sure the size is right
135       if (list.size() != i + 1) {
136         indexAdd = false;
137         // System.out.println(list.size());
138       }
139
140       // Check to make sure that the first index is the element we're adding, and the lest
    is 0
141       if (list.view(i) == null || list.view(0) != numbers[i] || list.view(i) != numbers[0])
    {
142         indexView = false;
143       }
144     }
145     // printArray(list.toArray());
146
147     printTestResult(indexAdd, name, "add(0) test (size)");
148     printTestResult(indexView, name, "add(0) test (view)");
149     indexAdd = true;
150     // All of the numbers should still be in sequence
151     for (int i = TEST_COUNT -1; i >= 0; i--) {
152       if (list.remove(0) != numbers[i]) {
153         indexAdd = false;
154       }
155     }
156     printTestResult(indexAdd, name, "add(0) test (sequence)");
157
158     list.clear();
159
160     for (int i = 0; i < TEST_COUNT; i++) {
161       list.add(numbers[TEST_COUNT - i - 1]);
162     }
163
164     boolean replaceReturn = true;
165     boolean replaceSet = true;
166
167     for (int i = 0; i < TEST_COUNT; i++) {
168       Integer c = list.replace(i, numbers[i]);
169       if (c != numbers[TEST_COUNT - i - 1]) {
170         replaceReturn = false;
171       }
172
173       if (list.view(i) != numbers[i]) {
174         replaceSet = false;
175       }
176
177     }
178
179     printTestResult(replaceReturn, name, "replace return value test");
180     printTestResult(replaceSet, name, "replace value set test");
181
182     list.clear();
183     try {
184       list.remove(0);
185       System.err.println("Uh Oh! " + name + " did not throw expected empty stack exception!"
    );
186     } catch (EmptyQueueException e) {
187       System.out.println(name + " threw expected EmptyQueueException!");
188     }
189
```

```
190   }
191
192   // Take an array of integers, and print them out nicely.
193   public static void printArray(Object[] l) {
194
195     System.out.print("[");
196     for (int i = 0; i < l.length; i++) {
197       System.out.print(l[i]);
198       if (i+1 < l.length) {
199         System.out.print(", ");
200       }
201     }
202     System.out.println("]");
203   }
204
205   /**
206    * Print the results of a test
207    * @param result Whether the test passed
208    * @param name The name of the test
209    */
210   public static void printTestResult(boolean result, String listName, String name) {
211     if (result) System.out.println(listName + " passed " + name +".");
212     else System.err.println(listName + " failed " + name + "!");
213   }
214
215 }
```

Part II
Interfaces

2 src/PriorityQueueInterface.java

```
1  /**
2     An interface for the ADT priority queue.
3     @author Frank M. Carrano
4     @author Timothy M. Henry
5     @version 4.0
6   */
7  public interface PriorityQueueInterface<T extends Comparable<? super T>>
8  {
9    /** Adds a new entry to this priority queue.
10        @param newEntry  An object to be added. */
11   public void add(T newEntry);
12
13   /** Removes and returns the entry having the highest priority.
14        @return  Either the object having the highest priority or,
15                 if the priority queue is empty before the operation, null. */
16   public T remove();
17
18   /** Retrieves the entry having the highest priority.
19        @return  Either the object having the highest priority or,
20                 if the priority queue is empty, null. */
21   public T peek();
22
23   /** Detects whether this priority queue is empty.
24        @return  True if the priority queue is empty, or false otherwise. */
25   public boolean isEmpty();
```

```
26
27   /** Gets the size of this priority queue.
28        @return  The number of entries currently in the priority queue. */
29   public int getSize();
30
31   /** Removes all entries from this priority queue. */
32   public void clear();
33 } // end PriorityQueueInterface
```

3 src/SortedListInterface.java

```
1  /**
2   * An interface that allows for implementation of the ADT Sorted List.
3   *
4   * @author Eli Zupke
5   * @param <T> The type of thing the list will contain
6   */
7  public interface SortedListInterface<T> {
8
9    /**
10    * Adds a new item to the right place in the list.
11    * The list size will be increased by 1.
12    * @param item The object to be added
13    */
14   public void add(T item);
15
16   /**
17    * Removes the first occurrence of the provided item, if it exists in the list
18    * @param item The item to remove
19    * @return Whether it was removed.
20    */
21   public boolean remove(T item);
22
23   /**
24    * Removes the item that is at the given position.
25    * The size of the list will decrease by one, and all subsequent entries will also have
       their positions decreased by one.
26    * @param index The index of the item to remove
27    * @return The item that was removed
28    * @throws EmptyQueueException if the queue is empty
29    * @throws IndexOutOfBoundsException if the specified position is outside of the list.
30    */
31   public T remove(int index);
32
33   /**
34    * Gets the position of the first occurence of the provided entry
35    * @param entry The entry to locate
36    * @return The position of the entry.
37    */
38   public int getPosition(T entry);
39
40   /**
41    * Empties the entire list.
42    * After the operation, the size of the list will be 0.
43    */
44   public void clear();
45
46   /**
47    * Checks whether the list contains a specified item.
48    * @param item The item to check for
49    * @return True if the list contains the item, or false if not.
50    */
```

```
51    public boolean contains(T item);
52
53    /**
54     * Returns the number of entries in the list
55     * @return The number of entries in the list
56     */
57    public int size();
58
59    /**
60     * Checks whether the list has any items
61     * @return True if the list has no items, or false if the list does have items.
62     */
63    public boolean isEmpty();
64
65    /**
66     * Returns an array representation of the list
67     * @return An array of the same length as the list, with the same items, in the same
       positions.
68     */
69    public T[] toArray();
70  }
```

4 src/ListInterface.java

```
1   /**
2    * An interface that allows for implementation of the ADT List. This list starts at zero.
3    *
4    * @author Eli Zupke
5    * @param <T> The type of thing the list will contain
6    */
7   public interface ListInterface<T> {
8
9     /**
10     * Adds a new item to the end of the list.
11     * The list size will be increased by 1, and other item positions will be unaffected.
12     * @param item The object to be added
13     */
14    public void add(T item);
15
16    /**
17     * Adds a new item to the specified position.
18     * The list size will be increased by 1, and all items that are at or after the specified
       position will also be increased by 1
19     * @param item The item to add to the list
20     * @param index The position in the list to put the item
21     * @throws IndexOutOfBoundsException if the specified position is outside of the list.
22     */
23    public void add(T item, int index) throws IndexOutOfBoundsException;
24
25
26    /**
27     * Removes the item that is at the given position.
28     * The size of the list will decrease by one, and all subsequent entries will also have
       their positions decreased by one.
29     * @param index The index of the item to remove
30     * @return The item that was removed
31     * @throws EmptyQueueException if the queue is empty
32     * @throws IndexOutOfBoundsException if the specified position is outside of the list.
33     */
34    public T remove(int index);
35
36    /**
```

```
37      * Removes the item at the end of the list.
38      * @return The item that was removed
39      * @throws EmptyQueueException if the queue is empty
40      */
41     public T remove();
42
43     /**
44      * Empties the entire list.
45      * After the operation, the size of the list will be 0.
46      */
47     public void clear();
48
49     /**
50      * Returns the item that is at the given position.
51      * The list will remain unchanged.
52      * @param index
53      * @return The item at the specified index.
54      * @throws IndexOutOfBoundsException if the specified position is outside of the list.
55      */
56     public T view(int index);
57
58     /**
59      * Checks whether the list contains a specified item.
60      * @param item The item to check for
61      * @return True if the list contains the item, or false if not.
62      */
63     public boolean contains(T item);
64
65     /**
66      * Returns the number of entries in the list
67      * @return The number of entries in the list
68      */
69     public int size();
70
71     /**
72      * Checks whether the list has any items
73      * @return True if the list has no items, or false if the list does have items.
74      */
75     public boolean isEmpty();
76
77     /**
78      * Returns an array representation of the list
79      * @return An array of the same length as the list, with the same items, in the same
        positions.
80      */
81     public T[] toArray();
82
83     /**
84      * Replaces the item at the specified index with the given item.
85      * The list size remains the same.
86      * @param index The index to replace at
87      * @param newItem The item to put in the index.
88      * @return The item that was replaced
89      */
90     public T replace(int index, T newItem);
91
92 }
```

Part III
Exceptions

5 src/EmptyQueueException.java

```
1
2 public class EmptyQueueException extends RuntimeException {
3
4   /**
5    *
6    */
7   private static final long serialVersionUID = -25586723978968324L;
8
9 }
```

Part IV
Nodes

6 src/Node.java

```
1  /**
2   *
3   * @author Eli Zupke
4   *
5   * @param <T> The type of data that is to be stored in this node.
6   */
7  public class Node<T> {
8
9    /**
10    * The node that this node links to.
11    */
12   private Node<T> nextNode;
13
14   private T data;
15
16   public Node(T data) {
17     this.data = data;
18   }
19
20   public Node<T> getNextNode() {
21     return nextNode;
22   }
23
24   public void setNextNode(Node<T> nextNode) {
25     this.nextNode = nextNode;
26   }
27
28   public T getData() {
29     return data;
30   }
31
32   public void setData(T data) {
33     this.data = data;
```

```
  }

}
```

7 src/DoubleNode.java

```
/**
 *
 * @author Eli Zupke
 *
 * @param <T> The type of data we want to store in this node
 */
public class DoubleNode<T> {

  /**
   * The node that this node links to.
   */
  private DoubleNode<T> nextNode;

  // The node that links to this one.
  private DoubleNode<T> previousNode;

  private T data;

  public DoubleNode(T data) {
    this.data = data;
  }

  public DoubleNode<T> getNextNode() {
    return nextNode;
  }

  public void setNextNode(DoubleNode<T> nextNode) {
    this.nextNode = nextNode;
  }

  public T getData() {
    return data;
  }

  public void setData(T data) {
    this.data = data;
  }

  public DoubleNode<T> getPreviousNode() {
    return previousNode;
  }

  public void setPreviousNode(DoubleNode<T> previousNode) {
    this.previousNode = previousNode;
  }

}
```

8 src/ComparableNode.java

```java
/**
 *
 * @author Eli Zupke
 *
 * @param <T> The object that is to be stored (Must implement Comparable)
 */
public class ComparableNode<T extends Comparable<? super T>> implements Comparable<
    ComparableNode<T>> {

  /**
   * The node that this node links to.
   */
  private ComparableNode<T> nextNode;

  private T data;

  public ComparableNode(T data) {
    this.data = data;
  }

  public ComparableNode<T> getNextNode() {
    return nextNode;
  }

  public void setNextNode(ComparableNode<T> nextNode) {
    this.nextNode = nextNode;
  }

  public T getData() {
    return data;
  }

  public void setData(T data) {
    this.data = data;
  }

  @Override
  /**
   * When we compare one node with another, we really want to compare their contents, so do
    that.
   */
  public int compareTo(ComparableNode<T> n) {
    return this.getData().compareTo(n.getData());
  }}
```

Part V
Implementations

9 src/LinkedPriorityQueue.java

```java
/**
 * An implementation of the ADT Priority Queue using singly linked data.
 *
 * @author Eli Zupke
 * @param <T> The type of data to store in this queue. T must extend Comparable.
 */
public class LinkedPriorityQueue<T extends Comparable<? super T>> implements
    PriorityQueueInterface<T> {

  // This variable always points at the back of the queue
  private ComparableNode<T> back;

  @Override
  /** Adds a new entry to the back of this queue.
    @param newEntry  An object to be added. */
  public void add(T newEntry) {//TODO FIX THIS

    // Create the new node that we will add
    ComparableNode<T> addedNode = new ComparableNode<T>(newEntry);

    ComparableNode<T> prevNode = null;
    ComparableNode<T> nextNode = back;
    // We need to loop through the queue to find the first node that is better than this one
    .
    while(nextNode != null
        && nextNode.compareTo(addedNode) < 0) {
      // We are still in the middle of the queue; we must go further
      prevNode = nextNode;
      nextNode = nextNode.getNextNode();
    }

    // We have found the front of the queue
    // We remove this node from the chain by setting the previous node's next node to null
    if (prevNode == null) {
      // This is a special case; we must set top to null
      addedNode.setNextNode(back);
      back = addedNode;
    } else {
      addedNode.setNextNode(nextNode);
      prevNode.setNextNode(addedNode);
    }

  }

  @Override
  /** Removes and returns the entry at the front of this queue.
    @return  The object at the front of the queue.
    @throws  EmptyQueueException if the queue is empty before the operation. */
  public T remove() {
    if (isEmpty()) {
      throw new EmptyQueueException();
    } else {
      ComparableNode<T> prevNode = null;
      ComparableNode<T> thisNode = back;

      // We need to loop through the entire queue to find the front.
      while(thisNode.getNextNode() != null) {
        // We are still in the middle of the queue; we must go further
        prevNode = thisNode;
        thisNode = thisNode.getNextNode();
      }

      // We have found the front of the queue
      // We remove this node from the chain by setting the previous node's next node to null
      if (prevNode == null) {
```

```
64          // This is a special case; we must set top to null
65          back = null;
66        } else {
67          prevNode.setNextNode(null);
68        }
69
70        return thisNode.getData();
71      }
72    }
73
74    @Override
75    /**  Retrieves the entry at the front of this queue.
76      @return  The object at the front of the queue.
77      @throws  EmptyQueueException if the queue is empty. */
78    public T peek() {
79      if (isEmpty()) {
80        throw new EmptyQueueException();
81      } else {
82        ComparableNode<T> thisNode = back;
83
84        // Loop through the entire queue
85        while(true) {
86          if (thisNode.getNextNode() == null) {
87            // We have found the front of the queue
88            return thisNode.getData();
89          } else {
90            // We must continue to the next node
91            thisNode = thisNode.getNextNode();
92          }
93        }
94      }
95    }
96    @Override
97    /** Detects whether this queue is empty.
98      @return  True if the queue is empty, or false otherwise. */
99    public boolean isEmpty() {
100
101     return (back == null);
102
103   }
104
105   @Override
106   /** Gets the size of this priority queue.
107     @return  The number of entries currently in the priority queue. */
108   public int getSize() {
109     int size = 0;
110     ComparableNode<T> curNode = back;
111
112     // We traverse the linked data until we come across null, which means we reached the
      front.
113     while (curNode != null) {
114       size++;
115       curNode = curNode.getNextNode();
116     }
117     return size;
118   }
119
120   @Override
121   /** Removes all entries from this queue. */
122   public void clear() {
123     // We can clear the queue by dereferencing the top node.
124     back = null;
125   }
126
127 }
```

10 src/FixedArrayList.java

```
1
2  public class FixedArrayList<T> implements ListInterface<T> {
3
4    private T[] array;
5
6    // This number is always the index of the element with the highest index.
7    int top = -1;
8
9
10   public FixedArrayList(int capacity) {
11
12         @SuppressWarnings("unchecked")
13         T[] tempArray = (T[])new Object[capacity]; // Unchecked cast
14     array = tempArray;
15   }
16
17
18   @Override
19   public void add(T item) {
20     // Check to see if we have space.
21     ensureCapacity();
22     top++;
23     array[top] = item;
24
25   }
26
27   @Override
28   public void add(T item, int index) throws IndexOutOfBoundsException {
29
30     ensureCapacity();
31     ensureIndexInAddingBounds(index);
32
33     //Add the item at the right index, and move everything after it down.
34     // The element we are currently moving
35     T moved = item;
36     for (int i = index; i <= top + 1; i++) {
37       T temp = array[i];
38       array[i] = moved;
39       moved = temp;
40       // Test.printArray(this.toArray());
41     }
42     // Now that we've added an item, the top counter should move up!
43     top++;
44     // Test.printArray(this.toArray());
45   }
46
47   @Override
48   public T remove(int index) {
49     ensureNotEmpty();
50     ensureIndexInBounds(index);
51     // Move everything from the top down one space. By the end, we've removed an item.
52     // The element we are currently moving
53     T moved = null;
54     for (int i = top; i >= index; i--) {
55       T temp = array[i];
56       array[i] = moved;
57       moved = temp;
58     }
59     // We've removed something, so the top index is lower now.
60     top--;
61     // By the end of this, the item we want should be in moved.
62     return moved;
63   }
64
65   @Override
```

```
66  public T remove() {
67    return remove(top);
68  }
69
70  @Override
71  public void clear() {
72    for (int i = 0; i <= top; i++) {
73      array[i] = null;
74    }
75    top = -1;
76
77  }
78
79  @Override
80  public T view(int index) {
81    ensureNotEmpty();
82    ensureIndexInBounds(index);
83    return array[index];
84  }
85
86  @Override
87  public boolean contains(T item) {
88    // Search through the array to see if item is in the array.
89    for (int i = 0; i <= top; i++) {
90      if (array[i] == item) return true;
91    }
92    return false;
93  }
94
95  @Override
96  public int size() {
97    return top + 1;
98  }
99
100 @Override
101 public boolean isEmpty() {
102   return top == -1;
103 }
104
105 @Override
106 public T[] toArray() {
107   if (isEmpty()) {
108     return null;
109   } else {
110     // Create an array of the right size
111         @SuppressWarnings("unchecked")
112         T[] outArray = (T[])new Object[top + 1]; // Unchecked cast
113
114         // Copy the contents of the array into the new array
115         for (int i = 0; i < outArray.length; i++) {
116           outArray[i] = array[i];
117         }
118         return outArray;
119   }
120 }
121
122 @Override
123 public T replace(int index, T newItem) {
124
125   // Just swap the two things. Easy.
126   T output = array[index];
127   array[index] = newItem;
128   return output;
129 }
130
131 /**
132  * Test to see if we have room to add an element.
133  * Throws an IndexOutOfBoundsException if the array is full.
```

```
   */
  private void ensureCapacity() {
    if (top + 1 >= array.length) {
      throw new IndexOutOfBoundsException("Max array size reached!");
    }
  }

  /**
   * Test to see if an index is in the bounds of the array,
   * and that there is an element at that index.
   * Throws an IndexOutOfBoundsException if not.
   * @param index The index to test
   */
  private void ensureIndexInBounds(int index) {
    if (index > top || index < 0) {
      throw new IndexOutOfBoundsException();
    }
  }


  /**
   * Test to see if an index is in the bounds of the array,
   * and that there is an element at that index, or one before it.
   * This is used to test if we can add an element at a given index.
   * Throws an IndexOutOfBoundsException if not.
   * @param index The index to test
   */
  private void ensureIndexInAddingBounds(int index) {
    if (index > top + 1 || index >= array.length || index < 0) {
      throw new IndexOutOfBoundsException();
    }
  }

  /**
   * Test to see if the queue is not empty.
   * If it is empty, throw an EmptyQueueException
   */
  private void ensureNotEmpty() {
    if (isEmpty()) {
      throw new EmptyQueueException();
    }
  }

}
```

11 src/LinkedList.java

```

public class LinkedList<T> implements ListInterface<T> {

  // Points to the front of the list
  protected Node<T> front;

  // Keep track of how big the list is to save time
  private int size;

  public LinkedList() {
    front = null;
    size = 0;
  }

  @Override
```

```
16  public void add(T item) {
17
18    Node<T> newNode = new Node<>(item);
19
20    //special case: the list is empty
21    if (front == null) {
22      front = newNode;
23      size = 1;
24      return;
25    }
26
27    Node<T> curNode = front;
28
29    // Go through each node, until we get to the end.
30    while (curNode.getNextNode() != null) {
31      curNode = curNode.getNextNode();
32    }
33
34    // Link in the new node to the list
35    curNode.setNextNode(newNode);
36
37    // Increment the size counter
38    size++;
39
40  }
41
42  @Override
43  public void add(T item, int index) throws IndexOutOfBoundsException {
44
45    if (index < 0) throw new IndexOutOfBoundsException();
46
47    Node<T> newNode = new Node<T>(item);
48
49    // Special case: The index is 0
50    if (index == 0) {
51      // set the new node's next node to front, and front to newNode.
52      newNode.setNextNode(front);
53      front = newNode;
54
55    } else {
56      Node<T> prevNode = null;
57      Node<T> curNode = front;
58
59      // Find the node at that index, and set curNode to it
60      for(int i = 0; i < index; i++) {
61        if (curNode == null) {
62          // The index was too large!
63          throw new IndexOutOfBoundsException();
64        }
65        prevNode = curNode;
66        curNode = curNode.getNextNode();
67      }
68      // Link newNode in-between prevNode and newNode
69      prevNode.setNextNode(newNode);
70      if (curNode != null) {
71        newNode.setNextNode(curNode);
72      }
73    }
74
75
76    size++;
77  }
78
79  @Override
80  public T remove(int index) {
81    ensureNotEmpty();
82    if (index < 0) throw new IndexOutOfBoundsException();
83
```

```
84      T data = null;
85
86      // Special case: The index is 0
87      if (index == 0) {
88        // Remove the front node, and keep its data
89        data = front.getData();
90        front = front.getNextNode();
91
92      } else {
93        Node<T> prevNode = null;
94        Node<T> curNode = front;
95
96        // Find the node at that index, and set curNode to it
97        for(int i = 0; i < index; i++) {
98          if (curNode == null) {
99            // The index was too large!
100           throw new IndexOutOfBoundsException();
101         }
102         prevNode = curNode;
103         curNode = curNode.getNextNode();
104       }
105
106       // We will remove curNode, so keep its data
107       data = curNode.getData();
108
109       // Set (the node before curNode)'s nextNode to curNode's NextNode.
110       prevNode.setNextNode(curNode.getNextNode());
111     }
112
113     // Adjust the known size of the list
114     size--;
115     return data;
116   }
117
118   @Override
119   public T remove() {
120     // TODO: replace with more efficient implementation;
121     return remove(size()-1);
122   }
123
124   @Override
125   public void clear() {
126     // De-reference the entire list
127     front = null;
128     size = 0;
129   }
130
131   @Override
132   public T view(int index) {
133     if (index < 0) throw new IndexOutOfBoundsException();
134     // Find the node at that index, and set curNode to it
135     Node<T> curNode = front;
136
137     for(int i = 0; i < index; i++) {
138       if (curNode == null) {
139         // The index was too large!
140         throw new IndexOutOfBoundsException();
141       }
142       curNode = curNode.getNextNode();
143     }
144     return curNode.getData();
145   }
146
147   @Override
148   public boolean contains(T item) {
149
150     Node<T> curNode = front;
151
```

```
152     // Go through each node. If we see the right data, return true.
153     // If we get to the end, return false.
154     while (curNode != null) {
155       if (curNode.getData() == item) {
156         return true;
157       }
158       curNode = curNode.getNextNode();
159     }
160 
161     return false;
162   }
163 
164   @Override
165   public int size() {
166     return size;
167   }
168 
169   @Override
170   public boolean isEmpty() {
171 
172     return front == null;
173   }
174 
175   @Override
176   public T[] toArray() {
177 
178     // If the list is empty, just return null.
179     if (isEmpty()) return null;
180 
181     @SuppressWarnings("unchecked")
182     T[] outArray = (T[])new Object[size]; // Unchecked cast
183 
184     Node<T> curNode = front;
185 
186     for(int i = 0; i < size; i++) {
187       outArray[i] = curNode.getData();
188       curNode = curNode.getNextNode();
189     }
190 
191     if (curNode != null) {
192       throw new IllegalStateException("Size of list wasn't correctly maintained!");
193     }
194 
195     return outArray;
196   }
197 
198   @Override
199   public T replace(int index, T newItem) {
200 
201     if (index < 0) throw new IndexOutOfBoundsException();
202 
203     Node<T> curNode = front;
204 
205     for(int i = 0; i < index; i++) {
206       if (curNode == null) {
207         // The index was too large!
208         throw new IndexOutOfBoundsException();
209       }
210       curNode = curNode.getNextNode();
211     }
212 
213     // Swap the data
214     T data = curNode.getData();
215     curNode.setData(newItem);
216 
217     return data;
218   }
219 
```

```
220   /**
221    * Test to see if the queue is not empty.
222    * If it is empty, throw an EmptyQueueException
223    */
224   private void ensureNotEmpty() {
225     if (isEmpty()) {
226       throw new EmptyQueueException();
227     }
228   }
229 
230 }
```

12 src/DoubleLinkedList.java

```
1 
2 public class DoubleLinkedList<T> implements ListInterface<T> {
3 
4   // Point to the front and back of the list, respectively
5   DoubleNode<T> front;
6   DoubleNode<T> back;
7 
8   // Keep track of how long the list is for time's sake.
9   int size = 0;
10 
11  public DoubleLinkedList() {
12 
13    // The single header node
14    front = null;
15    back = null;
16 
17  }
18 
19 
20  @Override
21  public void add(T item) {
22 
23    DoubleNode<T> newNode = new DoubleNode<>(item);
24 
25    //special case: the list is empty
26    if (front == null) {
27      front = newNode;
28      back = newNode;
29      size = 1;
30      return;
31    }
32    // Link in the new node to the list
33    back.setNextNode(newNode);
34    newNode.setPreviousNode(back);
35    back = newNode;
36 
37    // Increment the size counter
38    size++;
39 
40  }
41 
42  @Override
43  public void add(T item, int index) throws IndexOutOfBoundsException {
44 
45    if (index < 0 || index > size) {
46      throw new IndexOutOfBoundsException();
47    }
48 
```

```
49      DoubleNode<T> newNode = new DoubleNode<T>(item);
50
51      if (isEmpty()) {
52        // special case; add to start and end of list;
53        front = newNode;
54        back = newNode;
55
56
57      } else if (index == 0) {
58        // special case; add to start of list
59        newNode.setNextNode(front);
60        front.setPreviousNode(newNode);
61        front = newNode;
62
63      } else if (index == size) {
64        // special case; add to end of list
65        newNode.setPreviousNode(back);
66        back.setNextNode(newNode);
67        back = newNode;
68
69      } else {
70
71        // This is the node at the index where we will be adding.
72        // its index will increase by one.
73        DoubleNode<T> postNode = getNodeAtIndex(index);
74
75        DoubleNode<T> preNode = postNode.getPreviousNode();
76
77        // Link the new node into the list.
78        preNode.setNextNode(newNode);
79        newNode.setPreviousNode(preNode);
80
81        postNode.setPreviousNode(newNode);
82        newNode.setNextNode(postNode);
83
84      }
85
86      size++;
87    }
88
89    @Override
90    public T remove(int index) {
91      ensureNotEmpty();
92      if (index < 0 || index >= size) {
93        throw new IndexOutOfBoundsException();
94      }
95
96      T data = null;
97
98      if (size() == 1) {
99        // special case: remove from front and back
100       data = front.getData();
101       front = null;
102       back = null;
103
104     } else if (index == 0) {
105       // special case: remove from front
106       data = front.getData();
107       DoubleNode<T> newFront = front.getNextNode();
108
109       newFront.setPreviousNode(null);
110       front = newFront;
111
112     } else if (index == size - 1) {
113       // special case: remove from back
114       data = back.getData();
115       DoubleNode<T> newBack = back.getPreviousNode();
116
```

```
117       newBack.setNextNode(null);
118       back = newBack;
119     } else {
120
121       // This is the node at the index where we will be removing.
122       DoubleNode<T> exNode = getNodeAtIndex(index);
123
124       data = exNode.getData();
125
126       // These are the nodes around the node we will be removing.
127       DoubleNode<T> postNode = exNode.getNextNode();
128       DoubleNode<T> preNode = exNode.getPreviousNode();
129
130       // Test.printArray(toArray());
131
132       // Link the new node into the list.
133       preNode.setNextNode(postNode);
134       postNode.setPreviousNode(preNode);
135
136     }
137
138     size--;
139     return data;
140   }
141
142   @Override
143   public T remove() {
144     // TODO: replace with more efficient implementation
145     return remove(size()-1);
146   }
147
148   @Override
149   public void clear() {
150
151     DoubleNode<T> curNode = front;
152
153     // Go through the list, and remove one of each node's connections
154     while (curNode != null) {
155       curNode.setPreviousNode(null);
156       curNode = curNode.getNextNode();
157     }
158
159     // Now, every node is only connected one way, so when the ends are
160     // de-referenced, the rest will be too.
161     front = null;
162     back = null;
163
164     size = 0;
165   }
166
167   @Override
168   public T view(int index) {
169
170     return getNodeAtIndex(index).getData();
171   }
172
173   @Override
174   public boolean contains(T item) {
175
176     DoubleNode<T> curNode = front;
177
178     // Go through each node. If we see the right data, return true.
179     // If we get to the end, return false.
180     while (curNode != null) {
181       if (curNode.getData() == item) {
182         return true;
183       }
184       curNode = curNode.getNextNode();
```

```
185     }
186
187
188     return false;
189   }
190
191   @Override
192   public int size() {
193     return size;
194   }
195
196   @Override
197   public boolean isEmpty() {
198     return (size == 0);
199   }
200
201   @Override
202   public T[] toArray() {
203
204     // If the array is empty, just return null.
205     if (isEmpty()) return null;
206
207     @SuppressWarnings("unchecked")
208     T[] outArray = (T[])new Object[size]; // Unchecked cast
209
210     DoubleNode<T> curNode = front;
211
212     // Go through the list and copy the contents of each node to the array
213     for(int i = 0; i < size; i++) {
214       outArray[i] = curNode.getData();
215       curNode = curNode.getNextNode();
216     }
217
218     if (curNode != null) {
219       throw new IllegalStateException("Size of list wasn't correctly maintained!");
220     }
221
222     return outArray;
223   }
224
225   @Override
226   public T replace(int index, T newItem) {
227
228     // Find the node that contains the data we are replacing
229     DoubleNode<T> replacedNode = getNodeAtIndex(index);
230
231     // Grab the data, so that we can return it later
232     T data = replacedNode.getData();
233
234     // Replace the data in the node and return the data
235     replacedNode.setData(newItem);
236     return data;
237   }
238
239   /**
240    * Returns the node at the given index
241    * @param index The index to find the node at
242    * @return The given index
243    * @throws IndexOutOfBoundsException if the index is not in the list.
244    */
245   private DoubleNode<T> getNodeAtIndex(int index) {
246     if (index < 0 || index >= size) {
247       throw new IndexOutOfBoundsException();
248     }
249
250     DoubleNode<T> curNode = front;
251     // Go through the array until we get to the node at the given index
252     for(int i = 0; i < index; i++) {
```

```
253       if (curNode == null) {
254         // The index was too large!
255         throw new IndexOutOfBoundsException();
256       }
257       curNode = curNode.getNextNode();
258     }
259     return curNode;
260   }
261 
262   /**
263    * Test to see if the queue is not empty.
264    * If it is empty, throw an EmptyQueueException
265    */
266   private void ensureNotEmpty() {
267     if (isEmpty()) {
268       throw new EmptyQueueException();
269     }
270   }
271 
272 
273 }
```

13 src/LinkedSortedList.java

```
1 
2 public class LinkedSortedList<T extends Comparable<? super T>> extends LinkedList<T>
      implements SortedListInterface<T>{
3 
4   public LinkedSortedList() {
5     super();
6   }
7 
8   /**
9    * Adds the item into the proper, sorted place in the list
10   */
11  @Override
12  public void add(T item) {
13 
14    // Get the position of this element
15    int position = getPosition(item);
16    if (position < 0) position *= -1;
17 
18    // Add the item at the position that it belongs in.
19    add(item, position);
20 
21  }
22 
23  @Override
24  public boolean remove(T item) {
25 
26    int position = getPosition(item);
27 
28    // if the position of the item is > 0, then it is in the list, and therefore
29    // can be removed.
30    if (position > 0) {
31      remove(position);
32      return true;
33    }
34 
35    return false;
36  }
37 
```

```
38    @Override
39    public int getPosition(T entry) {
40
41      Node<T> curNode = front;
42      int index = 0;
43
44      // Go through the list, and increment index.
45      while (curNode != null) {
46        if (curNode.getData() == entry) {
47          // If we find the item we are looking for, return this index
48          return index;
49        } else if (curNode.getData().compareTo(entry) > 0) {
50          // If we find that we have passed the item we are looking for, return this index,
      but negative.
51          return -1 * index;
52        }
53        curNode = curNode.getNextNode();
54        index++;
55      }
56
57      return -1 * index;
58    }
59
60
61  }
```