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

```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>();
    testList(list, "LinkedList");
    list = new DoubleLinkedList<Integer>();
    testList(list, "DoubleLinkedList");

  }

  /**
   * Run a list of integers through a pretty good test.
   * @param list The list we are testing
   * @param name What the list is called
   */
  public static void testList(ListInterface<Integer> list, String name) {

    System.out.println("");
    System.out.println("==============================");
    System.out.println("Testing " + name);

    // Fill the list with increasing elements, then
    // remove everything from every index.
    boolean sequential = true;
    boolean size = true;
    boolean containment = true;

    // An array that contains all of the numbers, so that we can test for equality better.
    Integer[] numbers = new Integer[TEST_COUNT];

    for (int i = 0; i < TEST_COUNT; i++) {
      numbers[i] = new Integer(i);
    }

    for (int rIndex = 0; rIndex < TEST_COUNT; rIndex ++) {
      // System.out.println("Testing " + rIndex);
      // Fill the array with increasing numbers
      for (int i = 0; i < TEST_COUNT; i++) {
        list.add(numbers[i]);
        if (list.size() != i + 1) {
          size = false;
        }
      }

      // All of the numbers should still be in sequence
      for (int i = 0; i < TEST_COUNT - rIndex; i++) {
        if (list.remove(rIndex) != numbers[i + rIndex]) {
          sequential = false;
        }
      }

      // It should contain all of the numbers, up to what we removed
      for (int i = 0; i < rIndex; i++) {
        if (!list.contains(numbers[i])) {
          containment = false;
        }
      }

      // It should not contain any of the numbers after what we removed.
      for (int i = rIndex; i < TEST_COUNT; i++) {
        if (list.contains(numbers[i])) {
          containment = false;
        }
      }


      list.clear();
    }
    printTestResult(sequential, name, "removal test");
    printTestResult(containment, name, "containment test");
    printTestResult(size, name, "size test");

    list.clear();

    boolean indexAdd = true;
    boolean indexView = true;

    // Add everything to the start
    for (int i = 0; i < TEST_COUNT; i++) {
      list.add(numbers[i], 0);

      // Check to make sure the size is right
      if (list.size() != i + 1) {
        indexAdd = false;
        // System.out.println(list.size());
      }

      // Check to make sure that the first index is the element we're adding, and the lest is 0
      if (list.view(i) == null || list.view(0) != numbers[i] || list.view(i) != numbers[0]) {
        indexView = false;
      }
    }
    // printArray(list.toArray());

    printTestResult(indexAdd, name, "add(0) test (size)");
    printTestResult(indexView, name, "add(0) test (view)");
    indexAdd = true;
    // All of the numbers should still be in sequence
    for (int i = TEST_COUNT -1; i >= 0; i--) {
      if (list.remove(0) != numbers[i]) {
        indexAdd = false;
      }
    }
    printTestResult(indexAdd, name, "add(0) test (sequence)");

    list.clear();

    for (int i = 0; i < TEST_COUNT; i++) {
      list.add(numbers[TEST_COUNT - i - 1]);
    }

    boolean replaceReturn = true;
    boolean replaceSet = true;

    for (int i = 0; i < TEST_COUNT; i++) {
      Integer c = list.replace(i, numbers[i]);
      if (c != numbers[TEST_COUNT - i - 1]) {
        replaceReturn = false;
      }

      if (list.view(i) != numbers[i]) {
        replaceSet = false;
      }

    }

    printTestResult(replaceReturn, name, "replace return value test");
    printTestResult(replaceSet, name, "replace value set test");

    list.clear();
    try {
      list.remove(0);
      System.err.println("Uh Oh! " + name + " did not throw expected empty stack exception!");
    } catch (EmptyQueueException e) {
      System.out.println(name + " threw expected EmptyQueueException!");
    }

```

```java
  }

  // Take an array of integers, and print them out nicely.
  public static void printArray(Object[] l) {

    System.out.print("[");
    for (int i = 0; i < l.length; i++) {
      System.out.print(l[i]);
      if (i+1 < l.length) {
        System.out.print(", ");
      }
    }
    System.out.println("]");
  }

  /**
   * Print the results of a test
   * @param result Whether the test passed
   * @param name The name of the test
   */
  public static void printTestResult(boolean result, String listName, String name) {
    if (result) System.out.println(listName + " passed " + name +".");
    else System.err.println(listName + " failed " + name + "!");
  }

}
```

Part II
Interfaces

2 src/PriorityQueueInterface.java

```java
/**
   An interface for the ADT priority queue.
   @author Frank M. Carrano
   @author Timothy M. Henry
   @version 4.0
 */
public interface PriorityQueueInterface<T extends Comparable<? super T>>
{
  /** Adds a new entry to this priority queue.
       @param newEntry  An object to be added. */
  public void add(T newEntry);

  /** Removes and returns the entry having the highest priority.
       @return  Either the object having the highest priority or,
                if the priority queue is empty before the operation, null. */
  public T remove();

  /** Retrieves the entry having the highest priority.
       @return  Either the object having the highest priority or,
                if the priority queue is empty, null. */
  public T peek();

  /** Detects whether this priority queue is empty.
       @return  True if the priority queue is empty, or false otherwise. */
  public boolean isEmpty();

  /** Gets the size of this priority queue.
       @return  The number of entries currently in the priority queue. */
  public int getSize();

  /** Removes all entries from this priority queue. */
  public void clear();
} // end PriorityQueueInterface
```

3 src/SortedListInterface.java

```java
/**
 * An interface that allows for implementation of the ADT Sorted List.
 *
 * @author Eli Zupke
 * @param <T> The type of thing the list will contain
 */
public interface SortedListInterface<T> {

  /**
   * Adds a new item to the right place in the list.
   * The list size will be increased by 1.
   * @param item The object to be added
   */
  public void add(T item);

  /**
   * Removes the first occurrence of the provided item, if it exists in the list
   * @param item The item to remove
   * @return Whether it was removed.
   */
  public boolean remove(T item);

  /**
   * Removes the item that is at the given position.
   * The size of the list will decrease by one, and all subsequent entries will also have their positions decreased by one.
   * @param index The index of the item to remove
   * @return The item that was removed
   * @throws EmptyQueueException if the queue is empty
   * @throws IndexOutOfBoundsException if the specified position is outside of the list.
   */
  public T remove(int index);

  /**
   * Gets the position of the first occurence of the provided entry
   * @param entry The entry to locate
   * @return The position of the entry.
   */
  public int getPosition(T entry);

  /**
   * Empties the entire list.
   * After the operation, the size of the list will be 0.
   */
  public void clear();

  /**
   * Checks whether the list contains a specified item.
   * @param item The item to check for
   * @return True if the list contains the item, or false if not.
   */
  public boolean contains(T item);

  /**
   * Returns the number of entries in the list
   * @return The number of entries in the list
   */
  public int size();

  /**
   * Checks whether the list has any items
   * @return True if the list has no items, or false if the list does have items.
   */
  public boolean isEmpty();

  /**
   * Returns an array representation of the list
   * @return An array of the same length as the list, with the same items, in the same positions.
   */
  public T[] toArray();
}
```

4 src/ListInterface.java

```java
/**
 * An interface that allows for implementation of the ADT List. This list starts at zero.
 *
 * @author Eli Zupke
 * @param <T> The type of thing the list will contain
 */
public interface ListInterface<T> {

  /**
   * Adds a new item to the end of the list.
   * The list size will be increased by 1, and other item positions will be unaffected.
   * @param item The object to be added
   */
  public void add(T item);

  /**
   * Adds a new item to the specified position.
   * 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
   * @param item The item to add to the list
   * @param index The position in the list to put the item
   * @throws IndexOutOfBoundsException if the specified position is outside of the list.
   */
  public void add(T item, int index) throws IndexOutOfBoundsException;


  /**
   * Removes the item that is at the given position.
   * The size of the list will decrease by one, and all subsequent entries will also have their positions decreased by one.
   * @param index The index of the item to remove
   * @return The item that was removed
   * @throws EmptyQueueException if the queue is empty
   * @throws IndexOutOfBoundsException if the specified position is outside of the list.
   */
  public T remove(int index);

  /**
```

```java
   * Removes the item at the end of the list.
   * @return The item that was removed
   * @throws EmptyQueueException if the queue is empty
   */
  public T remove();

  /**
   * Empties the entire list.
   * After the operation, the size of the list will be 0.
   */
  public void clear();

  /**
   * Returns the item that is at the given position.
   * The list will remain unchanged.
   * @param index
   * @return The item at the specified index.
   * @throws IndexOutOfBoundsException if the specified position is outside of the list.
   */
  public T view(int index);

  /**
   * Checks whether the list contains a specified item.
   * @param item The item to check for
   * @return True if the list contains the item, or false if not.
   */
  public boolean contains(T item);

  /**
   * Returns the number of entries in the list
   * @return The number of entries in the list
   */
  public int size();

  /**
   * Checks whether the list has any items
   * @return True if the list has no items, or false if the list does have items.
   */
  public boolean isEmpty();

  /**
   * Returns an array representation of the list
   * @return An array of the same length as the list, with the same items, in the same positions.
   */
  public T[] toArray();

  /**
   * Replaces the item at the specified index with the given item.
   * The list size remains the same.
   * @param index The index to replace at
   * @param newItem The item to put in the index.
   * @return The item that was replaced
   */
  public T replace(int index, T newItem);

}
```

Part III

Exceptions

5 src/EmptyQueueException.java

```java

public class EmptyQueueException extends RuntimeException {

  /**
   *
   */
  private static final long serialVersionUID = -25586723978968324L;

}
```

Part IV

Nodes

6 src/Node.java

```java
/**
 *
 * @author Eli Zupke
 *
 * @param <T> The type of data that is to be stored in this node.
 */
public class Node<T> {

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

  private T data;

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

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

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

  public T getData() {
    return data;
  }

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

}
```

7 src/DoubleNode.java

```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) {
        // This is a special case; we must set top to null
        back = null;
      } else {
        prevNode.setNextNode(null);
      }

      return thisNode.getData();
    }
  }

  @Override
  /**  Retrieves the entry at the front of this queue.
    @return  The object at the front of the queue.
    @throws  EmptyQueueException if the queue is empty. */
  public T peek() {
    if (isEmpty()) {
      throw new EmptyQueueException();
    } else {
      ComparableNode<T> thisNode = back;

      // Loop through the entire queue
      while(true) {
        if (thisNode.getNextNode() == null) {
          // We have found the front of the queue
          return thisNode.getData();
        } else {
          // We must continue to the next node
          thisNode = thisNode.getNextNode();
        }
      }
    }
  }
  @Override
  /** Detects whether this queue is empty.
    @return  True if the queue is empty, or false otherwise. */
  public boolean isEmpty() {

    return (back == null);

  }

  @Override
  /** Gets the size of this priority queue.
    @return  The number of entries currently in the priority queue. */
  public int getSize() {
    int size = 0;
    ComparableNode<T> curNode = back;

    // We traverse the linked data until we come across null, which means we reached the front.
    while (curNode != null) {
      size++;
      curNode = curNode.getNextNode();
    }
    return size;
  }

  @Override
  /** Removes all entries from this queue. */
  public void clear() {
    // We can clear the queue by dereferencing the top node.
    back = null;
  }

}
```

10 src/FixedArrayList.java

```java

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

  private T[] array;

  // This number is always the index of the element with the highest index.
  int top = -1;


  public FixedArrayList(int capacity) {

        @SuppressWarnings("unchecked")
        T[] tempArray = (T[])new Object[capacity]; // Unchecked cast
    array = tempArray;
  }


  @Override
  public void add(T item) {
    // Check to see if we have space.
    ensureCapacity();
    top++;
    array[top] = item;

  }

  @Override
  public void add(T item, int index) throws IndexOutOfBoundsException {

    ensureCapacity();
    ensureIndexInAddingBounds(index);

    //Add the item at the right index, and move everything after it down.
    // The element we are currently moving
    T moved = item;
    for (int i = index; i <= top + 1; i++) {
      T temp = array[i];
      array[i] = moved;
      moved = temp;
      // Test.printArray(this.toArray());
    }
    // Now that we’ve added an item, the top counter should move up!
    top++;
    // Test.printArray(this.toArray());
  }

  @Override
  public T remove(int index) {
    ensureNotEmpty();
    ensureIndexInBounds(index);
    // Move everything from the top down one space. By the end, we’ve removed an item.
    // The element we are currently moving
    T moved = null;
    for (int i = top; i >= index; i--) {
      T temp = array[i];
      array[i] = moved;
      moved = temp;
    }
    // We’ve removed something, so the top index is lower now.
    top--;
    // By the end of this, the item we want should be in moved.
    return moved;
  }

  @Override
  public T remove() {
    return remove(top);
  }

  @Override
  public void clear() {
    for (int i = 0; i <= top; i++) {
      array[i] = null;
    }
    top = -1;

  }

  @Override
  public T view(int index) {
    ensureNotEmpty();
    ensureIndexInBounds(index);
    return array[index];
  }

  @Override
  public boolean contains(T item) {
    // Search through the array to see if item is in the array.
    for (int i = 0; i <= top; i++) {
      if (array[i] == item) return true;
    }
    return false;
  }

  @Override
  public int size() {
    return top + 1;
  }

  @Override
  public boolean isEmpty() {
    return top == -1;
  }

  @Override
  public T[] toArray() {
    if (isEmpty()) {
      return null;
    } else {
      // Create an array of the right size
          @SuppressWarnings("unchecked")
          T[] outArray = (T[])new Object[top + 1]; // Unchecked cast

          // Copy the contents of the array into the new array
          for (int i = 0; i < outArray.length; i++) {
            outArray[i] = array[i];
          }
          return outArray;
    }
  }

  @Override
  public T replace(int index, T newItem) {

    // Just swap the two things. Easy.
    T output = array[index];
    array[index] = newItem;
    return output;
  }

  /**
   * Test to see if we have room to add an element.
   * Throws an IndexOutOfBoundsException if the array is full.
```

```java
   */
  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

```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
  public void add(T item) {

    Node<T> newNode = new Node<>(item);

    //special case: the list is empty
    if (front == null) {
      front = newNode;
      size = 1;
      return;
    }

    Node<T> curNode = front;

    // Go through each node, until we get to the end.
    while (curNode.getNextNode() != null) {
      curNode = curNode.getNextNode();
    }

    // Link in the new node to the list
    curNode.setNextNode(newNode);

    // Increment the size counter
    size++;

  }

  @Override
  public void add(T item, int index) throws IndexOutOfBoundsException {

    if (index < 0) throw new IndexOutOfBoundsException();

    Node<T> newNode = new Node<T>(item);

    // Special case: The index is 0
    if (index == 0) {
      // set the new node's next node to front, and front to newNode.
      newNode.setNextNode(front);
      front = newNode;

    } else {
      Node<T> prevNode = null;
      Node<T> curNode = front;

      // Find the node at that index, and set curNode to it
      for(int i = 0; i < index; i++) {
        if (curNode == null) {
          // The index was too large!
          throw new IndexOutOfBoundsException();
        }
        prevNode = curNode;
        curNode = curNode.getNextNode();
      }
      // Link newNode in-between prevNode and newNode
      prevNode.setNextNode(newNode);
      if (curNode != null) {
        newNode.setNextNode(curNode);
      }
    }


    size++;
  }

  @Override
  public T remove(int index) {
    ensureNotEmpty();
    if (index < 0) throw new IndexOutOfBoundsException();

    T data = null;

    // Special case: The index is 0
    if (index == 0) {
      // Remove the front node, and keep its data
      data = front.getData();
      front = front.getNextNode();

    } else {
      Node<T> prevNode = null;
      Node<T> curNode = front;

      // Find the node at that index, and set curNode to it
      for(int i = 0; i < index; i++) {
        if (curNode == null) {
          // The index was too large!
          throw new IndexOutOfBoundsException();
        }
        prevNode = curNode;
        curNode = curNode.getNextNode();
      }

      // We will remove curNode, so keep its data
      data = curNode.getData();

      // Set (the node before curNode)'s nextNode to curNode's NextNode.
      prevNode.setNextNode(curNode.getNextNode());
    }

    // Adjust the known size of the list
    size--;
    return data;
  }

  @Override
  public T remove() {
    // TODO: replace with more efficient implementation;
    return remove(size()-1);
  }

  @Override
  public void clear() {
    // De-reference the entire list
    front = null;
    size = 0;
  }

  @Override
  public T view(int index) {
    if (index < 0) throw new IndexOutOfBoundsException();
    // Find the node at that index, and set curNode to it
    Node<T> curNode = front;

    for(int i = 0; i < index; i++) {
      if (curNode == null) {
        // The index was too large!
        throw new IndexOutOfBoundsException();
      }
      curNode = curNode.getNextNode();
    }
    return curNode.getData();
  }

  @Override
  public boolean contains(T item) {

    Node<T> curNode = front;

    // Go through each node. If we see the right data, return true.
    // If we get to the end, return false.
    while (curNode != null) {
      if (curNode.getData() == item) {
        return true;
      }
      curNode = curNode.getNextNode();
    }

    return false;
  }

  @Override
  public int size() {
    return size;
  }

  @Override
  public boolean isEmpty() {

    return front == null;
  }

  @Override
  public T[] toArray() {

    // If the list is empty, just return null.
    if (isEmpty()) return null;

    @SuppressWarnings("unchecked")
    T[] outArray = (T[])new Object[size]; // Unchecked cast

    Node<T> curNode = front;

    for(int i = 0; i < size; i++) {
      outArray[i] = curNode.getData();
      curNode = curNode.getNextNode();
    }

    if (curNode != null) {
      throw new IllegalStateException("Size of list wasn’t correctly maintained!");
    }

    return outArray;
  }

  @Override
  public T replace(int index, T newItem) {

    if (index < 0) throw new IndexOutOfBoundsException();

    Node<T> curNode = front;

    for(int i = 0; i < index; i++) {
      if (curNode == null) {
        // The index was too large!
        throw new IndexOutOfBoundsException();
      }
      curNode = curNode.getNextNode();
    }

    // Swap the data
    T data = curNode.getData();
    curNode.setData(newItem);

    return data;
  }

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

}
```

12 src/DoubleLinkedList.java

```java

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

  // Point to the front and back of the list, respectively
  DoubleNode<T> front;
  DoubleNode<T> back;

  // Keep track of how long the list is for time's sake.
  int size = 0;

  public DoubleLinkedList() {

    // The single header node
    front = null;
    back = null;

  }


  @Override
  public void add(T item) {

    DoubleNode<T> newNode = new DoubleNode<>(item);

    //special case: the list is empty
    if (front == null) {
      front = newNode;
      back = newNode;
      size = 1;
      return;
    }
    // Link in the new node to the list
    back.setNextNode(newNode);
    newNode.setPreviousNode(back);
    back = newNode;

    // Increment the size counter
    size++;

  }

  @Override
  public void add(T item, int index) throws IndexOutOfBoundsException {

    if (index < 0 || index > size) {
      throw new IndexOutOfBoundsException();
    }

    DoubleNode<T> newNode = new DoubleNode<T>(item);

    if (isEmpty()) {
      // special case; add to start and end of list;
      front = newNode;
      back = newNode;


    } else if (index == 0) {
      // special case; add to start of list
      newNode.setNextNode(front);
      front.setPreviousNode(newNode);
      front = newNode;

    } else if (index == size) {
      // special case; add to end of list
      newNode.setPreviousNode(back);
      back.setNextNode(newNode);
      back = newNode;

    } else {

      // This is the node at the index where we will be adding.
      // its index will increase by one.
      DoubleNode<T> postNode = getNodeAtIndex(index);

      DoubleNode<T> preNode = postNode.getPreviousNode();

      // Link the new node into the list.
      preNode.setNextNode(newNode);
      newNode.setPreviousNode(preNode);

      postNode.setPreviousNode(newNode);
      newNode.setNextNode(postNode);

    }

    size++;
  }

  @Override
  public T remove(int index) {
    ensureNotEmpty();
    if (index < 0 || index >= size) {
      throw new IndexOutOfBoundsException();
    }

    T data = null;

    if (size() == 1) {
      // special case: remove from front and back
      data = front.getData();
      front = null;
      back = null;

    } else if (index == 0) {
      // special case: remove from front
      data = front.getData();
      DoubleNode<T> newFront = front.getNextNode();

      newFront.setPreviousNode(null);
      front = newFront;

    } else if (index == size - 1) {
      // special case: remove from back
      data = back.getData();
      DoubleNode<T> newBack = back.getPreviousNode();

```

```java
      newBack.setNextNode(null);
      back = newBack;
    } else {

      // This is the node at the index where we will be removing.
      DoubleNode<T> exNode = getNodeAtIndex(index);

      data = exNode.getData();

      // These are the nodes around the node we will be removing.
      DoubleNode<T> postNode = exNode.getNextNode();
      DoubleNode<T> preNode = exNode.getPreviousNode();

      // Test.printArray(toArray());

      // Link the new node into the list.
      preNode.setNextNode(postNode);
      postNode.setPreviousNode(preNode);

    }

    size--;
    return data;
  }

  @Override
  public T remove() {
    // TODO: replace with more efficient implementation
    return remove(size()-1);
  }

  @Override
  public void clear() {

    DoubleNode<T> curNode = front;

    // Go through the list, and remove one of each node's connections
    while (curNode != null) {
      curNode.setPreviousNode(null);
      curNode = curNode.getNextNode();
    }

    // Now, every node is only connected one way, so when the ends are
    // de-referenced, the rest will be too.
    front = null;
    back = null;

    size = 0;
  }

  @Override
  public T view(int index) {

    return getNodeAtIndex(index).getData();
  }

  @Override
  public boolean contains(T item) {

    DoubleNode<T> curNode = front;

    // Go through each node. If we see the right data, return true.
    // If we get to the end, return false.
    while (curNode != null) {
      if (curNode.getData() == item) {
        return true;
      }
      curNode = curNode.getNextNode();
    }


    return false;
  }

  @Override
  public int size() {
    return size;
  }

  @Override
  public boolean isEmpty() {
    return (size == 0);
  }

  @Override
  public T[] toArray() {

    // If the array is empty, just return null.
    if (isEmpty()) return null;

    @SuppressWarnings("unchecked")
    T[] outArray = (T[])new Object[size]; // Unchecked cast

    DoubleNode<T> curNode = front;

    // Go through the list and copy the contents of each node to the array
    for(int i = 0; i < size; i++) {
      outArray[i] = curNode.getData();
      curNode = curNode.getNextNode();
    }

    if (curNode != null) {
      throw new IllegalStateException("Size of list wasn't correctly maintained!");
    }

    return outArray;
  }

  @Override
  public T replace(int index, T newItem) {

    // Find the node that contains the data we are replacing
    DoubleNode<T> replacedNode = getNodeAtIndex(index);

    // Grab the data, so that we can return it later
    T data = replacedNode.getData();

    // Replace the data in the node and return the data
    replacedNode.setData(newItem);
    return data;
  }

  /**
   * Returns the node at the given index
   * @param index The index to find the node at
   * @return The given index
   * @throws IndexOutOfBoundsException if the index is not in the list.
   */
  private DoubleNode<T> getNodeAtIndex(int index) {
    if (index < 0 || index >= size) {
      throw new IndexOutOfBoundsException();
    }

    DoubleNode<T> curNode = front;
    // Go through the array until we get to the node at the given index
    for(int i = 0; i < index; i++) {
      if (curNode == null) {
        // The index was too large!
        throw new IndexOutOfBoundsException();
      }
      curNode = curNode.getNextNode();
    }
    return curNode;
  }

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


}
```

13 src/LinkedSortedList.java

```

public class LinkedSortedList<T extends Comparable<? super T>> extends LinkedList<T>
    implements SortedListInterface<T>{

  public LinkedSortedList() {
    super();
  }

  /**
   * Adds the item into the proper, sorted place in the list
   */
  @Override
  public void add(T item) {

    // Get the position of this element
    int position = getPosition(item);
    if (position < 0) position *= -1;

    // Add the item at the position that it belongs in.
    add(item, position);

  }

  @Override
  public boolean remove(T item) {

    int position = getPosition(item);

    // if the position of the item is > 0, then it is in the list, and therefore
    // can be removed.
    if (position > 0) {
      remove(position);
      return true;
    }

    return false;
  }

  @Override
  public int getPosition(T entry) {

    Node<T> curNode = front;
    int index = 0;

    // Go through the list, and increment index.
    while (curNode != null) {
      if (curNode.getData() == entry) {
        // If we find the item we are looking for, return this index
        return index;
      } else if (curNode.getData().compareTo(entry) > 0) {
        // If we find that we have passed the item we are looking for, return this index, but negative.
        return -1 * index;
      }
      curNode = curNode.getNextNode();
      index++;
    }

    return -1 * index;
  }


}
```