```java
/**
 *  Program 2
 *  LinearList uses interface LinearListADT to create a doubly
 linked list.
 *  CS310-01
 *  3/13/2019
 *  @author  Karl Parks cssc1506
 */

package data_structures;

import java.util.Iterator;
import java.util.ConcurrentModificationException;

public class LinearList<E extends Comparable<E>> implements
LinearListADT<E> {
  private Node<E> head;
  private Node<E> tail;
  private int currentSize, modCount;

  //Create node
  @SuppressWarnings("hiding")
  class Node<E> {
    E data;
    Node<E> next;
    Node<E> prev;
    public Node(E obj) {
      data = obj;
      next = null;
      prev = null;
    }
  }

  public LinearList() {
    head = null;
    tail = null;
    modCount = 0; //modification counter
    currentSize = 0;
  }

  @Override
  public boolean addFirst(E obj) { //adds node at front of list
    Node<E> newNode = new Node<E>(obj);
    if (currentSize > 0) { //sets previous pointer
```

```
43        head.prev = newNode;
44      }
45      newNode.next = head; //sets new node next to current head node
46      head = newNode; //changes head to new node
47      if (currentSize == 0) { //empty matrix check
48        tail = head;
49      }
50      currentSize++; //increment size
51      modCount++; //increment modification counter
52      return true;
53    }
54
55    @Override
56    public boolean addLast(E obj) { //similar to addFirst
57      Node<E> newNode = new Node<E>(obj);
58      if (currentSize == 0) {
59        addFirst(obj);
60        return true;
61      }
62      tail.next = newNode;
63      newNode.prev = tail;
64      tail = newNode;
65      currentSize++;
66      modCount++;
67      return true;
68    }
69
70    @Override
71    public E removeFirst() { //removes first node
72      if (currentSize != 0) {
73        E tmp = head.data; //temporary variable for data return
74        head = head.next; //change head to next node
75        if (currentSize != 1) //change prev to null
76          head.prev = null;
77        currentSize--;
78        modCount++;
79        return tmp;
80      }
81      return null;
82    }
83
84    @Override
85    public E removeLast() { //similar to removeFirst
86      if (currentSize != 0) {
```

```
    if (currentSize != 0) {
      E tmp = tail.data;
      tail = tail.prev;
      if (currentSize != 1)
        tail.next = null;
      currentSize--;
      modCount++;
      return tmp;
    }
    return null;
  }

  @Override
  public E remove(E obj) { //remove first object found
    Node<E> tmp = head;
    Node<E> tmpDesired = null;
    for (int i = 0; i < currentSize; i++) {
      if (tmp.data.compareTo(obj) == 0) {
        //found node to delete
        tmpDesired = tmp;
        if (i == 0) { //begining of list
          removeFirst();
        }
        else if (i == currentSize - 1) { //end of list
          removeLast();
        }
        else { //anywhere in the imddle
          tmpDesired.prev.next = tmpDesired.next;
          tmpDesired.next.prev = tmpDesired.prev;
          currentSize--;
          modCount++;
        }
        return obj;
      }
      tmp = tmp.next; //how to iterater through list
    }
    return null;
  }

  @Override
  public E peekFirst() { //reveals first data in first node
    return (currentSize == 0)? null : head.data;
  }

  @
```

```java
    @Override
    public E peekLast() { //reveals last data in last node
      return (currentSize == 0)? null : tail.data;
    }

    @Override
    public boolean contains(E obj) { //returns boolean if object found
      return obj == find(obj); //use find which will return null if no desired value found
    }

    @Override
    public E find(E obj) { //returns object found
      Node<E> tmp = head;
      for (int i = 0; i < currentSize; i++) {
        if (tmp.data == obj) {
          return obj;
        }
        tmp = tmp.next;
      }
      return null;
    }

    @Override
    public void clear() { //sets list to be empty
      head = null;
      tail = null;
      currentSize = 0;
      modCount++;
    }

    @Override
    public boolean isEmpty() { //checks if empty
      return (currentSize == 0);
    }

    @Override
    public boolean isFull() { //checks if full (cannot be full - not array)
      return false;
    }

    @Override
    public int size() { //returns current size
```

```
172       //printList();
173       return currentSize;
174     }
175
176     public void printList() { //used for debugging
177       Node<E> tmp = head;
178       System.out.println("--- Printing List ---");
179       System.out.println("Size: " + currentSize);
180       for (int i = 0; i < currentSize; i++) {
181         System.out.println(tmp.data);
182         tmp = tmp.next;
183       }
184     }
185
186     //@Override
187     public Iterator<E> iterator() { //iterator helper method for
    •    enhanced for-loop
188       return new IteratorHelper();
189     }
190
191     private class IteratorHelper implements Iterator<E> {
192       private int count, expectedMod;
193       Node<E> tmp = head;
194       public IteratorHelper() {
195         expectedMod = modCount; //checks for modifications
196         count = 0;
197       }
198
199       public boolean hasNext() {
200         return count != currentSize; //checks if at end of list
201       }
202
203       public E next() {
204         if (modCount != expectedMod) { //modification error throw here
205               throw new ConcurrentModificationException("Cannot modify
    •               list during enhanced for-loop");
206         }
207         E tempData = tmp.data; //data to return
208         tmp = tmp.next; //iterator using each tmp.next
209         count++;
210         return tempData;
211       }
212
213       public void remove() {
```

```
214         throw new UnsupportedOperationException();
```