CS61B, Spring 2001
Midterm 2
Professor Clancy and Professor Yelick

Problem #1 (4 points, 8 minutes)

Given below is a framework for an *IntervalEnumeration* class for successively returning *Integer* values in an *Interval* in increasing order. For examples, if this represents the interval [3,5], the enumeration should return first an Integer object representing 3, then an Integer object representing 4, then an *Integer* object representing 5. Complete the method bodies on the next page.

```
// OVERVIEW: This class defines nonempty intervals of consecutive integers,
// such as [1,2,3] or [-2,-1,0,1].  Intervals are immutable.

public class Interval {
        public Interval (int a, int b) throws IllegalArgumentException{
                ...
        }
        // This interval represents all the integers between myLow and myHigh,
        // inclusive, with myLow <= myHigh.
        private int myLow, myHigh;

        // REQUIRES: this interval is not modified while the Enumeration returned
        // is in use.
        // EFFECTS: returns an Enumeration of Integer objects corresponding
        // to the values in this interval, enumerated lowest to highest.
        public Enumeration values ( ) {
                return new IntervalEnumeration ( );
        }

FILL IN THE FOLLOWING DEFINITIONS:

        private class IntervalEnumeration ( ) implements Enumeration {
                // EFFECTS: Initializes an enumeration of elements of this interval.
                public IntervalEnumeration ( ) {



                }
                // EFFECTS: Returns true if there are more elements of this interval
                // to return; returns false otherwise.
                public boolean hasMoreElements ( ) {



                }
                // REQUIRES: hasMoreElements ( ).
                // MODIFIES: this enumeration.
                // EFFECTS: Returns the next element from this interval;
                // throws NoSuchElementException if !hasMoreElements ( ).
                public Object nextElement ( ) {




                }
                // any state variables you need go here




                }
        }
}
```

Problem #2 (5 points, 10 minutes)

Given below is a class representing a singly linked list.

```
public class List {
        private class ListNode {
                public Object myItem;
                public ListNode myNext;
                // EFFECTS: Initializes a one-element linked list.
                public ListNode (Object item) {
                        myItem = item;
                        myNext = null;
                }
                // EFFECTS: Initializes a linked list whose first element is item
                // and whose remaining items are those of remaining.
                public ListNode (Object item, ListNode remaining) {
                        myItem = item;
                        myNext = remaining;
                }
        }
        private ListNode myFirst;     // reference to the first node in the list
        // REQUIRES: myFirst != null and the linked list pointed to by myFirst
        // is noncircular.
        // MODIFIES: this.
        // EFFECTS: exchanges the first two nodes in the linked list pointed to
        // by myFirst; throws NoSuchElementException if there are fewer than two
        // nodes in the list
        private void exchangeFirstTwoNodes ( ) {
                ...
        }
}
```

You are to complete the *exchangeFirstTwoNodes* method. If the REQUIRES clause does not hold, your solution is allowed to crash or exhibit arbitrary behavior. An easy way to solve this problem is to exchange the *myItem* variables of the first two nodes. **DON'T DO THIS. You will receive no credit for a solution that includes an assignment to the *myItem* variable of any *ListNode*.**

```
// REQUIRES: myFirst != null and the linked list pointed to by myFirst
// is noncircular.
// MODIFIES: this.
// EFFECTS: exchanges the first two nodes in the linked list pointed to
// by myFirst; throws NoSuchElementException if there are fewer than two
// nodes in the list
private void exchangeFirstTwoNodes ( ) {





}
```

Problem #3 (4 points, 5 minutes)

Indicate the values stored in the variable b in the main method below by circling true or false in the comment lines. The Interval class for this problem is what you used in project 1 and lab assignment 7.

```
import java.util.*;
public class BetterInterval extends Interval {
        public BetterInterval (int low, int high) {
                super (low,high);
        }
        public boolean equals (BetterInterval intvl) {
                return intvl.high( ) == high( ) && intvl.low( ) == low( );
        }
        public static void main (String [ ] args) {
                BetterInterval intvl1 = new BetterInterval (2,3);
                BetterInterval intvl2 = new BetterInterval (2,3);
                Vector v1 = new Vector ( );
                Vector v2 = new Vector ( ):
                boolean b;
                v1.addElement (intvl1);
                v2.addElement (intvl2);
                b = v1.equals (v2);
                                                          // Contents of b: true or
false?
                b = intvl1.equals (intvl2);
                                                          // Contents of b: true or
false?
                Object intvl3 = v1.elementAt (0);
                Object intvl4 = v2.elementAt (0);
                b = ((BetterInterval) intvl3).equals ((BetterInterval) intvl4);
                                                          // Contents of b: true or
false?
```

```
                  b = intvl3.equals(intvl4);
                                                                  // Contents of b: true or
false?
        }
}
```

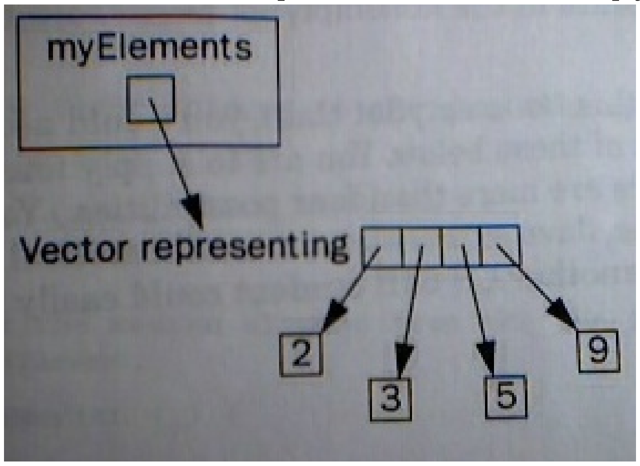
Problem #4 (7 points, 15 minutes)

Consider a class named *NonemptySet* that represents a nonempty set of (distinct) integers and supports two operations:

- add an integer to the set;
- delete the median element from the set.

We assume that, for the purposes of this problem, the median in a set S with an even number of elements is the same as the median in the set that results from removing the largest element of S. For example, the median of {2,3,5,9} is 3. Two implementations for this class are described on the following pages.

Part a
One implementation of the *NonemptySet* class maintains a **sorted** doubly linked list of *Integer objects*, with a reference to the first node in the list, a reference to the node that contains the median element in the list, and a count of the number of nodes in the list. A box-and-pointer diagram of an example set appears below; all the state variables are represented in the diagram. A framework for this implementation of the *NonemptySet* class appears at the end of this exam.



For each operation listed below, circle the estimate that most closely represents the operation's worst-case running time. Assume that the operation is implemented to be as fast as possible.

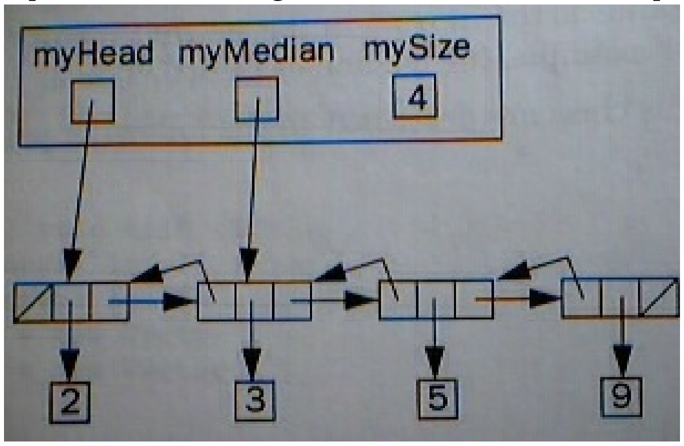
Adding a randomly chosen element k to the set of n elements (the add method):

```
•  finding k or k's proper position in the list,          Θ(n)  Θ(log n)  Θ(1)
•  then inserting k into the list if it's not already     Θ(n)  Θ(log n)  Θ(1)
       there,
•  then updating the median if necessary.                 Θ(n)  Θ(log n)  Θ(1)
```

Deleting the median of a set of n elements (the deleteMedian method):

```
•  locating the median,                                   Θ(n)  Θ(log n)  Θ(1)
•  then removing it from the linked list,                 Θ(n)  Θ(log n)  Θ(1)
•  then updating the median if necessary.                 Θ(n)  Θ(log n)  Θ(1)
```

Part b
Another implementation of the *NonemptySet* class maintains a sorted *Vector*, as shown below. Again, all state variables are represented in the diagram. A framework for this implementation of the *NonemptySet* class appears at the end of this exam.



For each operation listed below, circle the estimate that most closely represents the operation's worst-case running time. Assume that the operation is implemented to be as fast as possible.

Adding a randomly chosen element k to the set of n elements (the add method):

• `finding k or k's proper position in the list,`	Θ(n)	Θ(log n)	Θ(1)
• `then inserting k into the list if it's not already there,`	Θ(n)	Θ(log n)	Θ(1)
• `then updating the median if necessary.`	Θ(n)	Θ(log n)	Θ(1)

Deleting the median of a set of n elements (the deleteMedian method):

• `locating the median,`	Θ(n)	Θ(log n)	Θ(1)
• `then removing it from the linked list,`	Θ(n)	Θ(log n)	Θ(1)
• `then updating the median if necessary.`	Θ(n)	Θ(log n)	Θ(1)

Part c
Circle the estimate that most closely represents the worst-case running time of the entire add method for this implementation. Assume that the method is implemented to be as fast as possible.

Θ(n^2) Θ(n log n) Θ(n) Θ(log n) Θ(1)

Problem #5 (4 points, 11 minutes)

A correctly working implementation for the *NonemptySet* class in part a of problem 4 will have reasonable values for the instance variables *mySize*, *myHead*, and *myMedian*, as well as a proper list of doubly linked *DListNodes* from lab assignment 6. (The declaration of *DListNode* appears in the *NonemptySet* framework that follows this problem.)
To write a *repOk* method for this *NonemptySet* class, you would need to check several properties. We give you some of these below. You are to supply four more interesting and distinct properties. (There are more than four possibilities.) Your properties may be written using English prose, Java expressions, or mathematical notation, but they must be precise enough that another CS61B student could easily translate them into code.
Given:

1. myHead != null.
2. myHead contains a reference to the first node of a proper doubly linked list. That is, myHead.myPrev == null, and for each node n in the myHead list, if n.myNext != null then n.myNext.myPrev == n; if n.myPrev != null then n.myPrev. myNext == n.
3. myHead.myPrev == null.
4. There exists some node n in the myHead list such that n.myNext == null.
5. The myHead list contains no cycles accessible through either the myNext or myPrev references. That is, for every node n in the myHead list, k > 0 applications of myPrev never produces the node n, and k > 0 applications of myNext never produces the node n.
6. myMedian != null.

Your properties:

1.
2.
3.
4.

Code for problem 4, part a

```
public class NonemptySet {
        // EFFECTS: Initialize a one-element set that contains the given value.
        public void NonemptySet (int n) {
                ...
        }
        // MODIFIES: this.
        // EFFECTS: Add the given integer to the set if it's not already there.
        public void add (int n) {
                ...
        }
        // MODIFIES: this.
        // EFFECTS: remove the median element from the set if the set contains
        // more than one element.
        public void deleteMedian ( ) {
                ...
        }
        private class DListNode {
                public Object myItem;
                public DListNode myPrev;
                public DListNode myNext;
                // EFFECTS: Initialize a DListNode with myItem obj
                // and the given values for myPrev and myNext.
                public DListNode (Object obj, DListNode prev, DListNode next) {
                        myItem = obj;
                        myPrev = prev;
                        myNext = next;
                }
        }
        private DListNode myHead;
        private DListNode myMedian;
        private int mySize;
}
```

Code for problem 4, part b

```
public class NonemptySet {
        // EFFECTS: Initialize a one-element set that contains the given value.
        public void NonemptySet (int n) {
                ...
        }
        // MODIFIES: this.
        // EFFECTS: Add the given integer to the set if it's not already there.
        public void add (int n) {
                ...
        }
        // MODIFIES: this.
        // EFFECTS: remove the median element from the set if the set contains
        // more than one elements.
        public void deleteMedian ( ) {
                ...
        }
        private Vector myElements;     // elements are sorted
}
```

Solutions!