Homework 5

This homework will teach you a more secure way to encapsulate lists than the method used in Homework 4, and give you practice using it to accomplish tasks quickly. This is an individual assignment; you may not share code with other students.

The list package contains encapsulated DList and SList classes (both of which inherit from an abstract List class). These classes differ from those we have seen before in a critical way: each ListNode knows which List it is in. A new invariant in our Lists is that for every ListNode x in a List l, x.myList = l, UNLESS x is the sentinel. For any sentinel node x, x.myList = null. Because every ListNode knows its List, we can move some of the methods from the List class to the ListNode class.

Methods of List	Methods of ListNode
public boolean isEmpty()	public Object item()
public int length()	public void setItem(Object item)
public void insertFront(Object item)	public ListNode next()
public void insertBack(Object item)	public ListNode prev()
public ListNode front()	public void insertAfter(Object item)
public ListNode back()	public void insertBefore(Object item)
	public void remove()
	public boolean isValidNode()

One innovation of these classes is the existence of "invalid nodes," which can be identified by the isValidNode() method. In Homework 4, the methods next() and prev() return null when there is no node to return. Here in Homework 5, they return an invalid node instead. A node that has been removed from a List is also invalid. With the exception of isValidNode(), any method called on an invalid node will throw an InvalidNodeException.

The item field of ListNode is no longer public, to prevent applications from storing items in invalid nodes.

Recall that every ListNode knows what List it is in. An invalid node is represented by any ListNode whose "myList" field is null. In the DList implementation, the sentinel is an invalid node, which simplifies the implementations of front(), back(), next(), and prev(). (Take a look at the code for DListNode.isValidNode.)

Part I (2 points)

Complete the implementation of the DList and DListNode classes.

In DList.java, you will need to implement insertFront(), insertBack(), and the DList() constructor. You should be able to cut and paste your solutions from Homework 4 with just a small change. The implementations of front() and back() are already provided; observe that they are simpler than in Homework 4 because we use sentinels as invalid nodes.

In DListNode.java, you will need to implement insertAfter(), insertBefore(), and remove(). Your Homework 4 solutions will be a good start, but you'll need to make changes to accommodate these methods' move from DList to DListNode.

The main() method of list.DList contains code to help test your work.

Part II (8 points)

Your main assignment is to implement a Set ADT in Set.java. Your Set class must use a List to store the elements of the set. Your Sets should behave like mathematical sets, which means they should not contain duplicate items. To make set union and intersection operations run quickly, your Sets will contain only Comparable elements, and you will keep them sorted in order from least to greatest element. (You will want to review the Comparable interface on the Java API Web page.)

You will need to declare some fields and implement the following methods.

```
  public Set()                          // Constructs an empty Set.
  public int cardinality()              // Number of elements in this Set.
  public void insert(Comparable c)      // Insert c into this Set.
  public void union(Set s)              // Assign this = (this union s).
  public void intersect(Set s)          // Assign this = (this intersect s).
  public String toString()              // Express this Set as a String.
```

Two items o1 and o2 are considered duplicates if o1.compareTo(o2) == 0. By convention, Java classes are supposed to have o1.compareTo(o2) == 0 if and only if o1.equals(o2). (Of course, it's always possible for some idiot to break this convention, so it would be safest not to depend on equals() working.)

Unlike most previous assignments, each method comes with prescribed time bounds that you must meet when your Set uses DLists (but not when it uses SLists). For example, union() and intersect() must run in time proportional to this.cardinality() + s.cardinality(). This means you do NOT have time to make a pass through "this" list for every element of s; that would take time proportional to this.cardinality() * s.cardinality(). To achieve this time bound, you must take advantage of the fact that Sets are sorted. This time bound is one reason why Sets may not store duplicate items in their Lists.

On the other hand, insert() need not run in constant time. Since each Set uses a sorted representation, insert() may need time proportional to the cardinality of the Set to find the right place to insert a new element, and to ensure that the new element doesn't duplicate an old one.

Another constraint is that union() and intersect() may NOT change the Set s. Furthermore, intersect() may not construct any new ListNodes (it only needs to remove ListNodes from "this" List), and union() should reuse all the ListNodes in the Set "this", constructing new nodes only for elements of s that "this" List lacks. We will deduct points for failing to meet the time bounds or failing to obey these constraints.

Be sure to declare variables of static type List and ListNode in Set.java, not variables of type DList, DListNode, SList, or SListNode. Set.java should be able to switch between using DLists and using SLists by changing one constructor call in the Set() constructor. (In fact, you can use SList to help you debug Set if you have trouble getting DList working. But be sure to use a DList in your final submission unless you can't get it working.)

Do not modify List.java, ListNode.java, SList.java, or SListNode.java. Do not modify the prototypes in Set.Java, DList.java, or DListNode.java.

Afterthought (for your own introspection only)

If you use SLists instead of DLists, do your union() and intersect() methods still run within the time bounds? If not, how easy would it be to fix them so that they do?

```
Submitting your solution
------------------------
Make sure that your code compiles and runs. After submitting, if you realize
your solution is flawed, you may fix it and submit again.  You may submit as
often as you like.  Only the last version you submit before the deadline will
be graded.
```