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.

Copy the Homework 5 directory by doing the following, starting from your home directory. Don’t forget the “-r” switch in the cp command.

```
mkdir hw5
cd hw5
cp -r $master/hw/hw5/*
```

The list package contains encapsulated DLList and SLList 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 any ListNode x in a List l, x.mylist = l (even if x is a sentinel). This allows us to move some of the methods from the List class to the ListNode class.

### Methods of List

- `public boolean isEmpty()`  
- `public int length()`  
- `public void insertFront(Object item)`  
- `public void insertBack(Object item)`  
- `public ListNode front()`  
- `public ListNode back()`

### Methods of ListNode

- `public boolean isEmpty()`  
- `public Object item()`  
- `public void setItem(Object item)`  
- `public ListNode next()`  
- `public ListNode prev()`  
- `public void insertAfter(Object item)`  
- `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, whereas here they return an invalid node. 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 manipulation of 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 DLList implementation, every sentinel is also an invalid node (a choice that simplifies the implementations of front(), back(), next(), and prev()), even though its "myList" field is not null. (Take a look at DLListNode.isValidNode.)

### Part I (2 points)

--------------

Complete the implementation of the DLList and DLListNode classes.

In DList.java, you will need to implement insertFront(), insertBack(), and the DLList() constructor. You should be able to cut and paste your solutions from Homework 4 unchanged. 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 DLListNode.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 DLList to DLListNode.

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 decide on fields and implement the following methods.

- `public Set()`  
- `public int cardinality()`  
- `public void insert(Comparable c)`  
- `public void union(Set s)`  
- `public void intersect(Set s)`  
- `public String toString()`

Unlike in previous assignment, each method comes with prescribed time bounds that you must meet when your Set uses DLLists (but not when it uses SLLists).

For example, `union()` and `intersect()` must run in time proportional to `this.cardinality() + s.cardinality()`. This means you don’t have time to make a pass through "this" for each element of s; that would take time proportional to `this.cardinality() * s.cardinality()`. You must take advantage of the fact that Sets are sorted to achieve this time bound. 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.

Another constraint is that `union()` and `intersect()` may not change the Set s. Furthermore, `intersect()` may not create any new ListNodeS (it only needs to remove ListNodeS from "this" List), and `union()` should reuse all the ListNodeS in the Set "this", creating 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 DLList, DLListNode, SLList, or SListNode. Set.java should be able to switch between using DLLists and using SLLists by changing one constructor call in the Set() constructor. (In fact, you can use SLList to help debug Set if you have trouble getting DLList working. But be sure to use a DLList in your final submission unless you can’t get it working.)

Do not modify List.java, ListNode.java, SLList.java, or SListNode.java. Do not modify the prototypes in Set.java, DLList.java, or DLListNode.java.

Afterthought (for your own introspection only)

- If you use SLLists instead of DLLists, 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

------------------------
Change (cd) to your hw5 directory, which should contain the list directory. The list directory should contain DList.java and DListNode.java. You're not allowed to change the other files, so you can’t submit them. You shouldn’t need any other classes, but you can submit them if you want.

Make sure that your code compiles and runs on the _lab_ machines. Then, from your hw5 directory, type "submit hw5". (Note that "submit" will not work if you are inside the list directory!) 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.