1) [20%] In a **doubly-linked List**, think of the "next" links as progressing left-to-right, and the "prev" links as right-to-left:

```java
interface ListI {
    abstract public void setItem(Item c);
    abstract public void setNext(List l); // "right" link
    abstract public void setPrev(List l); // "left" link
    abstract public Item getItem();
    abstract public List getNext();
    abstract public List getPrev();
    abstract public int getLength();
}
```

abstract class List implements ListI, Item { ... code omitted

class EmptyList extends List { ... code omitted

class Node extends List { ... code omitted

a) [10%] You’ll construct a doubly-linked list of two Items a and b, one Item at a time. First given Item b (non-null), write code to create a list of only {b}, with "next" and "prev" links referencing EmptyList. Assign this to the variable List l.

b) [10%] Next add Item a to List l, making the list {a,b}, such that the following are true:

1. l.getItem().equals(a)
2. l.getNext().getItem().equals(b)
3. l.getNext().getNext().getPrev().getItem().equals(a)
2a) [10%] Given the above definition of the class `List`, and a valid, non-null `List l`, write code to enumerate the Items in `l` and call `toString()` in each Item. Make sure your code works correctly for all cases (`l` could be an `EmptyList`, or a non-empty `List`).

2b) [10%] Recall that `ListC` implements `CollectionI` where:

```java
public interface CollectionI {
    boolean add(Item i);
    Item get();
    void start();
    boolean step();
    boolean more();
    boolean remove();
    void removeAll();
    void markPlace();
    void restartFromMarker();
    int getCount();
}
```

Given non-null `ListC c`, write code to call `toString()` in each Item contained in `c` (there might be zero or more Items).
3) [5%] Modify the interface Item, so that Items can both accept visitors and be Observers. Assume Observer is an interface. Don’t extend Item, just rewrite it:

public interface Item

4) [15%] If some container class C implements CollectionI, write a subclass CV which can accept instances of class Visitor. Given non-null Visitor v and CV cv, for example, cv.acceptVisitor(v) causes cv itself to visit with v and also each contained Item to individually accept v.

public class CV
5) [10%] Note that our Visitor class must visit both CV and Item. Write the **base class Visitor**.

6) [10%] Write **ToStringVisitor**, a subclass of Visitor, which would invoke **toString()** on the CV itself and on each contained Item.
7) [15%] Write `AddObserverVisitor`, another subclass of `Visitor`, which, given some `Observable o` passed as an argument to the constructor, would add as observers to `o`, any `CV` it visits as well as each contained Item. For example:

```java
Observable o = new SomeKindOfObservable();
AddObserverVisitor aov = new AddObserverVisitor(o);
CV cv = new CV(); // with instances of Items added
cv.acceptVisitor(aov); // cv and any contained items to observe o.
```

8) [5%] Finally, what additional code is needed in `CV` for CV itself to be an `Observer` (and not just its contained Items)? Briefly answer here and go back and add code to your previous code in question 4.