CHAP. 5 : LINKED LISTS

ONE DRAWBACK TO OUR INTEGERLIST ADT IS THE FACT THAT THE UNDERLYING ARRAY IS OF FIXED SIZE.

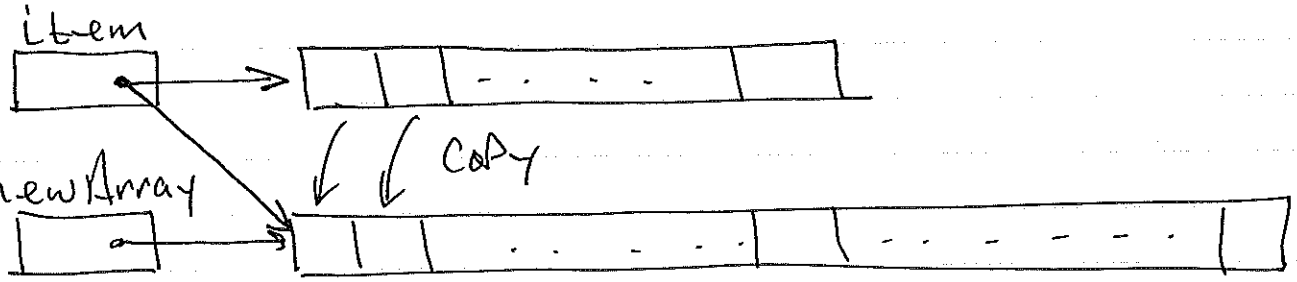
THIS CAN BE EASILY FIXED BY INCORPORATING A PROCEDURE TO INCREASE THE SIZE OF THE ARRAY item[] WHENEVER THE PHYSICAL LIMIT IS REACHED.

DEFINE A NEW FIELD

```
Private int PhysicalSize;
```

AND INTRODUCE A NEW PRIVATE METHOD:

```
Private void doubleItemArray(){
    PhysicalSize *= 2;
    int[] newArray = new int[PhysicalSize];
    for(int i=0; i< numItems; i++){
        newArray[i] = item[i];
    }
    item = newArray;
}
```



```
// IntegerList.java
// Array based implementation of IntegerList ADT (with array doubling)

public class IntegerList implements IntegerListInterface{

   private static final int INITIAL_SIZE = 1;
   private int physicalSize;       // current length of underlying array
   private int[] item;             // array of IntegerList items
   private int numItems;           // number of items in this IntegerList

   // arrayIndex
   // transforms a List index to an Array index
   private int arrayIndex(int listIndex){
      return listIndex-1;
   }

   // doubleItemArray
   // doubles the physical size of the underlying array item[]
   private void doubleItemArray(){
      physicalSize *=2;
      int[] newArray = new int[physicalSize];
      for(int i=0; i<numItems; i++) newArray[i] = item[i];
      item = newArray;
   }

   // IntegerList
   // default constructor for the IntegerList class
   public IntegerList(){
      physicalSize = INITIAL_SIZE;
      item = new int[physicalSize];
      numItems = 0;
   }

   // isEmpty
   // pre: none
   // post: returns true if this IntgerList is empty, false otherwise
   public boolean isEmpty(){
      return(numItems == 0);
   }

   // size
   // pre: none
   // post: returns the number of elements in this IntegerList
   public int size() {
      return numItems;
   }

   // get
   // pre: 1 <= index <= size()
   // post: returns item at position index
   public int get(int index) throws ListIndexOutOfBoundsException {

      if( index<1 || index>numItems ){
         throw new ListIndexOutOfBoundsException("get() precondition
         violated");
      }
      return item[arrayIndex(index)];
   }
```

```
// add
// inserts newItem in this IntegerList at position index
// pre: 1 <= index <= size()+1
// post: !isEmpty(), items to the right of newItem are renumbered
public void add(int index, int newItem) throws
ListIndexOutOfBoundsException{

   if( index<1 || index>(numItems+1) ){
      throw new ListIndexOutOfBoundsException("add() precondition
      violated");
   }

   if( numItems == physicalSize ) {
      doubleItemArray();
   }

   for(int i=numItems; i>=index; i--) {
      item[arrayIndex(i+1)] = item[arrayIndex(i)];
   }
   item[arrayIndex(index)] = newItem;
   numItems++;
}

// remove
// deletes item from position index
// pre: 1 <= index <= size()
// post: items to the right of deleted item are renumbered
public void remove(int index)
   throws ListIndexOutOfBoundsException{

   if( index<1 || index>numItems ){
      throw new ListIndexOutOfBoundsException("remove() precondition
      violated");
   }

   for(int i=index+1; i<=numItems; i++){
      item[arrayIndex(i-1)] = item[arrayIndex(i)];
   }
   numItems--;
}

// removeAll
// pre: none
// post: isEmpty()
public void removeAll(){
   numItems = 0;
}

// toString
// pre: none
// post:  prints current state to stdout
// Overrides Object's toString() method
public String toString(){
   int i;
   String s = "";

   for(i=0; i<numItems; i++) s += item[i] + " ";
   return s;
}
```

```
    // equals
    // pre: none
    // post: returns true if this IntegerList matches rhs, false otherwise
    // Overrides Object's equals() method
    public boolean equals(Object rhs){
       int i = 0;
       boolean eq = false;
       IntegerList R = null;

       if(rhs instanceof IntegerList){
          R = (IntegerList)rhs;
          eq = (this.numItems == R.numItems);
          while(eq && i<numItems){
             eq = (this.item[i] == R.item[i]);
             i++;
          }
       }
       return eq;
    }

}
```

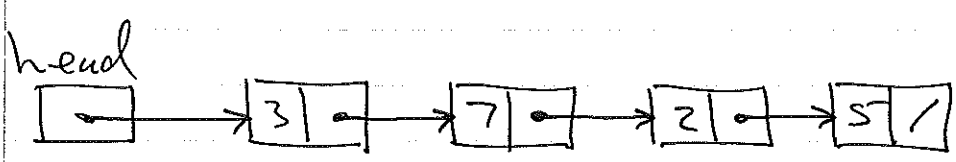
```
// ListIndexOutOfBoundsException.java

public class ListIndexOutOfBoundsException extends IndexOutOfBoundsException{
   public ListIndexOutOfBoundsException(String s){
      super(s);
   }
}
```

```
#
# makefile for IntegerList ADT
#

JAVASRC    = IntegerList.java IntegerListInterface.java IntegerListTest.java\
             ListIndexOutOfBoundsException.java
MAINCLASS  = IntegerListTest
CLASSES    = IntegerList.class IntegerListInterface.class IntegerListTest.
class\
             ListIndexOutOfBoundsException.class
JARFILE    = IntegerListTest
JARCLASSES = $(CLASSES)


all: $(JARFILE)

$(JARFILE): $(CLASSES)
   echo Main-class: $(MAINCLASS) > Manifest
   jar cvfm $(JARFILE) Manifest $(JARCLASSES)
   rm Manifest
   chmod +x $(JARFILE)

$(CLASSES): $(JAVASRC)
   javac $(JAVASRC)
# Note: no -Xlint option

clean:
   rm -f $(CLASSES) $(JARFILE)
```

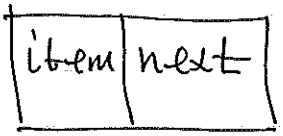
THE DOUBLING OPERATION CAN BE COSTLY, BUT FORTUNATELY IT DOESN'T HAPPEN VERY OFTEN (ONCE THE ARRAY REACHES A SUFFICIENT SIZE.)

STILL DOUBLING SEEMS WASTEFUL IF ALL YOU NEED IS ONE MORE ITEM IN THE LIST.

ANOTHER APPROACH IS TO HAVE EACH LIST ITEM STORED IN A SPECIAL CLASS WHICH ALSO MAINTAINS A REFERENCE TO THE next (AND POSSIBLY PREVIOUS) ITEM IN THE LIST. THIS DATA STRUCTURE IS CALLED A <u>LINKED LIST</u> AND THE CLASS WHICH HOLDS A SINGLE ITEM IS CALLED A Node



NODE:



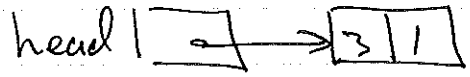

```
Public class Node {
    Public int item;
    Public Node next;
}
```

```
// Node.java

class Node {

    int item;
    Node next;

    Node(int x) {

        item = x;
        next = null;
    }

}
```

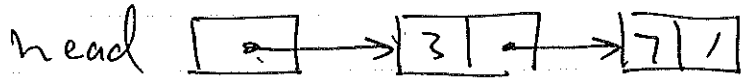
A SPECIAL REFERENCE CALLED head POINTS TO THE FIRST Node IN THE List

In main say: (in file: NodeTest.java)

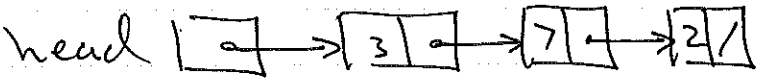
```
Node head = new Node(3);
```



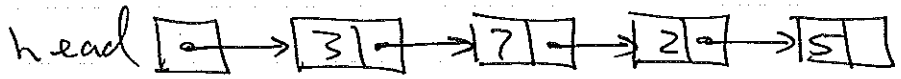

```
head.next = new Node(7);
```




```
head.next.next = new Node(2);
```

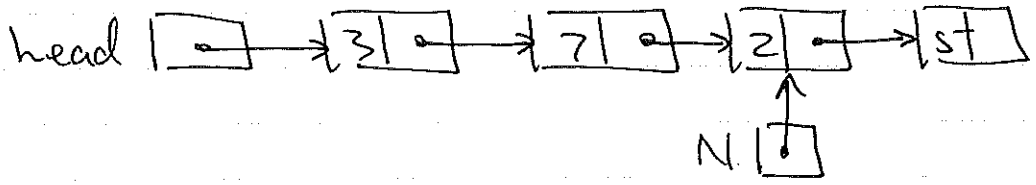



```
head.next.next.next = new Node(5);
```



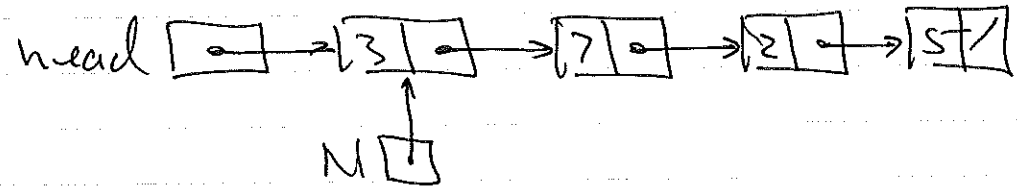
ANOTHER WAY:

```
Node head = new Node(3);
Node N = head;
  N.next = new Node(7);
     N = N.next;
  N.next = new Node(2);
     N = N.next;
  N.next = new Node(5);
```



IT'S A LITTLE BASIER TO INSERT NODES INTO THE FRONT OF THE LIST:

```
Node head = new Node(5);
Node N = new Node(2);
  N.next = head;
    head = N
      N = new Node(7);
  N.next = head;
    head = N;
      N = new Node(3);
  N.next = head
    head = N;
```



WE COULD REMEDY THIS BY CHANGING THE 5 TO A 9 FOR INSTANCE BY DOING.

```
N=N.next;
N=N.next;
N=N.next;
N.setItem(9);
```

NOTICE THAT WE CAN ONLY TRAVERSE THE LIST IN ONE DIRECTION. IT LOOKS LIKE