```
// IntegerListInterface.java
// interface for the IntegerList ADT

public interface IntegerListInterface{

   // isEmpty
   // pre: none
   // post: returns true if this IntgerList is empty, false otherwise
   public boolean isEmpty();

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

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

   // 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;

   // 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;

   // removeAll
   // pre: none
   // post: isEmpty()
   public void removeAll();

}
```

```
// IntegerList.java
// Linked List implementation of the IntegerList ADT

public class IntegerList implements IntegerListInterface {

   // private inner Node class
   private class Node {
      int item;
      Node next;

      Node(int x){
         item = x;
         next = null;
      }
   }

   // Fields for the IntegerList class
   private Node head;     // reference to first Node in List
   private int numItems;  // number of items in this IntegerList

   // find()
   // returns a reference to the Node at position index in this IntegerList
   private Node find(int index){
      Node N = head;
      for(int i=1; i<index; i++) N = N.next;
      return N;
   }

   // IntegerList()
   // constructor for the IntegerList class
   public IntegerList(){
      head = null;
      numItems = 0;
   }

   // Implementation of ADT operations ////////////////////////////////////

   // 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 in this IntegerList
   public int get(int index) throws ListIndexOutOfBoundsException {
      if( index<1 || index>numItems ){
         throw new ListIndexOutOfBoundsException(
            "get(): invalid index: " + index);
      }
      Node N = find(index);
      return N.item;
   }
```

```
// add()
// inserts newItem into 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(): invalid index: " + index);
   }
   if(index==1){
      Node N = new Node(newItem);
      N.next = head;
      head = N;
   }else{
      Node P = find(index-1); // at this point index >= 2
      Node C = P.next;
      P.next = new Node(newItem);
      P = P.next;
      P.next = C;
   }
   numItems++;
}

// remove()
// deletes item at position index from this IntegerList
// 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(): invalid index: " + index);
   }
   if(index==1){
      Node N = head;
      head = head.next;
      N.next = null;
   }else{
      Node P = find(index-1);
      Node N = P.next;
      P.next = N.next;
      N.next = null;
   }
   numItems--;
}

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

// toString
// pre: none
// post:  prints current state to stdout
// Overrides Object's toString() method
public String toString(){
   String s = "";
   for(Node N=head; N!=null; N=N.next){
      s += N.item + " ";
   }
   return s;
}
```

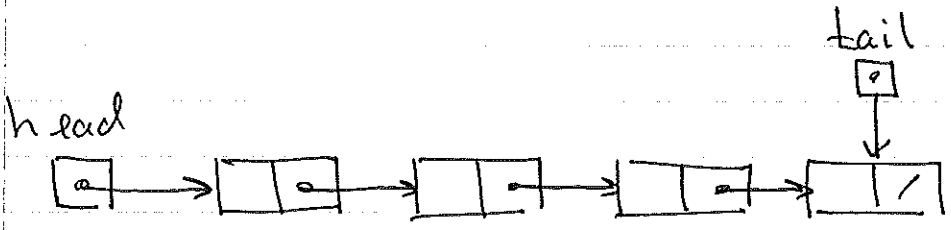
```
    ////////////////////////////////////////////////////////////////////////////
    /
    //
    //  here is a another version of toString() that uses a recursive private
    //  helper function called myString()
    //
    //  private String myString(Node H){
    //     if( H==null ){
    //        return "";
    //     }else{
    //        return (H.item + " " + myString(H.next));
    //     }
    //  }
    //
    //  public String toString(){
    //     return myString(head);
    //  }
    //

    ////////////////////////////////////////////////////////////////////////////
    /

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

       if(rhs instanceof IntegerList){
          R = (IntegerList)rhs;
          eq = ( this.numItems == R.numItems );

          N = this.head;
          M = R.head;
          while(eq && N!=null){
             eq = (N.item == M.item);
             N = N.next;
             M = M.next;
          }
       }
       return eq;
    }

}
```

```
// IntegerListTest.java
// A test client for the IntegerList ADT

public class IntegerListTest{

   public static void main(String[] args){
      IntegerList A = new IntegerList();
      IntegerList B = new IntegerList();
      int i, j;

      for(i=1; i<=100; i++){
         j = i*i;
         A.add(i, j);
         B.add(i, (j+i)/2);
      }

      System.out.println(A);
      System.out.println();
      System.out.println(B);
      System.out.println();
      System.out.println(A.equals(B));
      System.out.println();
      System.out.println(A.size());
      System.out.println();
      System.out.println(B.size());
      System.out.println();

      for(i=1; i<=10; i++){
         A.remove(9*i);
         B.remove(8*i-3);
      }

      System.out.println(A.size());
      System.out.println();
      System.out.println(B.size());
      System.out.println();
      System.out.println(B.get(37));
      System.out.println();
      try{
         System.out.println(A.get(200));
      }catch(ListIndexOutOfBoundsException e){
         System.out.println("Caught Exception " + e);
         System.out.println("Continuing without interuption");
      }
      System.out.println();
      System.out.println(A.get(20));
   }
}
```

VARIATIONS OF LINKED LISTS

TAIL REFERENCES

IT MAY BE USEFUL IN SOME CIRCUMSTANCES TO MAINTAIN A REFERENCE TO THE TAIL OF THE LIST

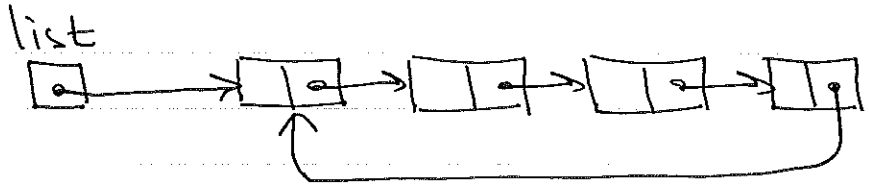


FOR INSTANCE, IF MOST (OR PERHAPS ALL) INSERTIONS WILL BE AT THE END OF THE LIST. RATHER THAN TRAVERSE THE LIST FROM HEAD TO TAIL, WE CAN JUST DO

```
tail.next = new Node( )
tail = tail.next
```

CIRCULAR LINKED LISTS

IT MAY BE CONVENIENT TO HAVE THE LAST NODES next FIELD POINT TO THE FIRST NODE IN THE LIST:



IN THIS CASE WE MAY WISH TO CALL THE EXTERNAL NODE REFERENCE SOMETHING OTHER THAN head. THIS EXTERNAL REFERENCE CAN ALSO POINT TO THE LAST NODE.

IN SUCH A list YOU NEED A DIFFERENT TEST TO DETERMINE WHEN YOU ARE AT THE END. e.g. N=list INSTEAD OF N=null.

```
Node N = list;
do{
    // Process N IN SOME WAY
    N=N.next;
}while (N != list);
```

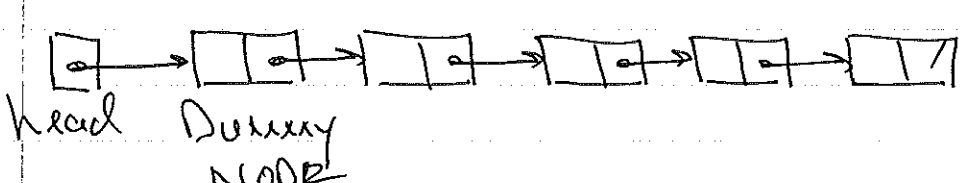
NOTICE THAT:

```
for(Node N= list; N != list; N=N.next ){

    //Process N
}
```

will NOT WORK HERE.

<u>Dummy Head Nodes</u>.
methods which DO INSERTION AND DELETION OFTEN HAVE TO TREAT THE FIRST POSITION IN THE list AS A SPECIAL CASE

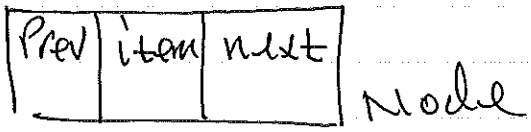
BY INSERTING A "DUMMY" NODE BEFORE THE REAL FIRST NODE WE CAN ELIMINATE THE NEED FOR THESE SPECIAL CASES.



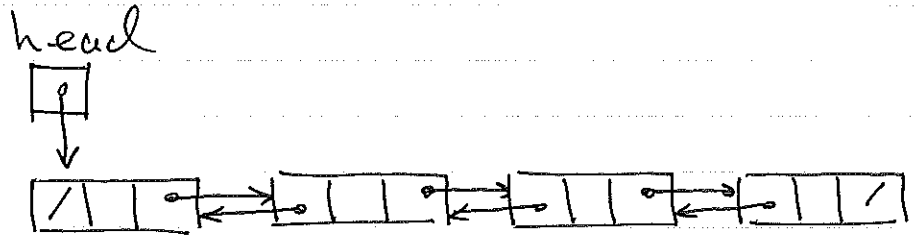
DUMMY NODES CAN BE USED IN CONTEXT OF CIRCULAR LISTS TOO. THEY ARE MOST USEFUL IN DOUBLY LINKED LISTS.

Doubly linked lists

THE DESIGN OF THE Node class CAN BE ALTERED TO INCLUDE A REFERENCE TO THE PREVIOUS NODE IN THE LIST.



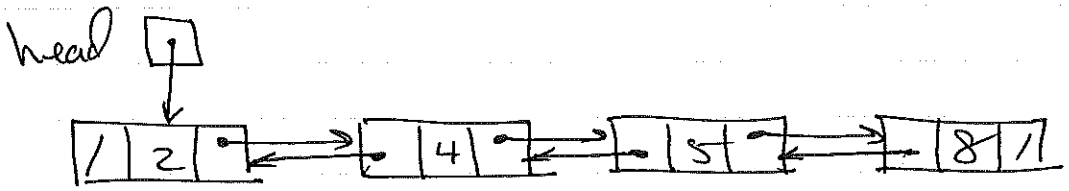
SO A LINKED LIST OF THESE NODES WOULD LOOK LIKE



THIS IS VERY USEFUL IF YOU NEED TO TRAVERSE LIST IN BOTH DIRECTIONS.

IT'S MORE COMPLICATED HOWEVER TO DO INSERTIONS AND DELETIONS.

FOR INSTANCE GIVEN



TO INSERT 6 BETWEEN 4 AND 5:

```
Node N = head;
N = N.next;
Node P = N;
N = N.next;
P.next = new Node(6);
P.next.Prev = P;
P = P.next;
N.Prev = P;
P.next = N;
```

DIFFERENT COMBINATIONS OF THESE TYPES OF LISTS CAN BE USED. e.g. DOUBLY LINKED, WITH DUMMY NODES, CIRCULAR, ETC...

PA2: DICTIONARY ADT. (WEEK MAY 4).