# JAVA NOTES

# DATA STRUCTURES AND ALGORITHMS

Terry Marris July 2001

## 14 LINKED LISTS

### 14.1 LEARNING OUTCOMES

By the end of this lesson the student should be able to

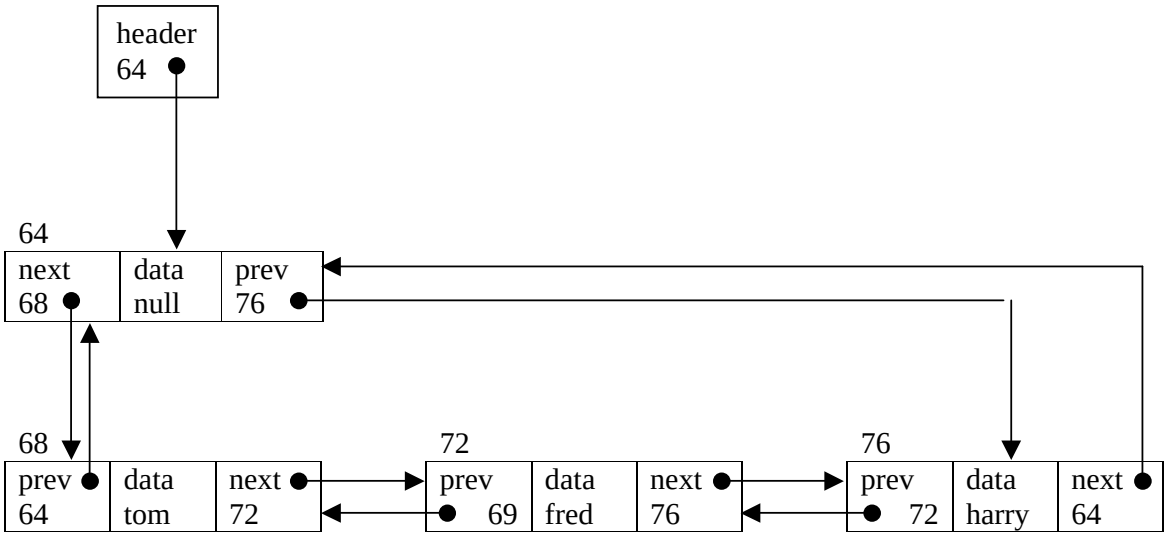
- explain, in diagrams and words, how list methods add, remove and find objects in a doubly linked indexed list.
- test given *LinkedList* methods

### 14.2 PRE-REQUISITES

The student should be comfortable with manipulating a linked list of nodes - §9 Dynamic Stacks and §12 Dynamic Queues.

## 14.3 LIST

We picture a linked list (or just list for short) as a linear sequence of nodes. Each node has two link fields. One link field refers to the next node in the sequence. One link field refers to the previous node in the sequence. We show a list containing three data objects, *tom*, *dick* and *harry*.

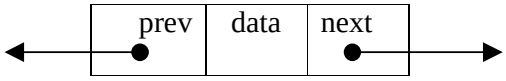



The node referred to by header (64) has a next link field that refers to the first node in the list, and a previous link field that refers to the last node in the list. Both the first node in the list (68) and the last node (76) refer back to the node referred to be *header* (64).

Starting from any node and following the arrows you can reach any node you like.

## 14.4 THE NODE CLASS

A node has a data field and two link fields, *previous* and *next*. Both *previous* and *next* refer to nodes.

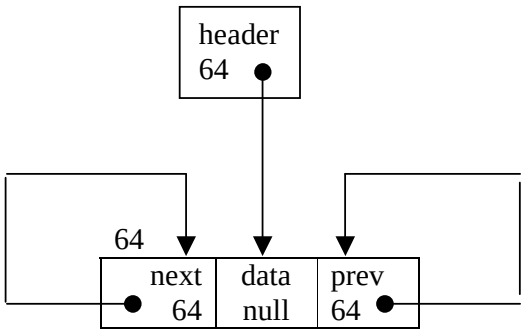



```
private static class Node {
/* Defines a node, used by LinkedList
*/
  Object data;
  Node next;
  Node previous;

  Node(Object data, Node next, Node previous)
  {
    this.data = data;
    this.next = next;
    this.previous = previous;
  }
}
```

The *Node* class is an inner, *private static* class. It is defined inside the *LinkedList class*. It is a helper class - it helps the *LinkedList* class methods to implement their behaviour. Because it is a *private* inner class, only *LinkedList* methods may create and use *Node* objects. Because it is a *static* class, *Node* objects cannot access *private LinkedList* components. Only inner classes can be *private* and *static*.

## 14.5 THE EMPTY LIST

An empty list has no data objects.



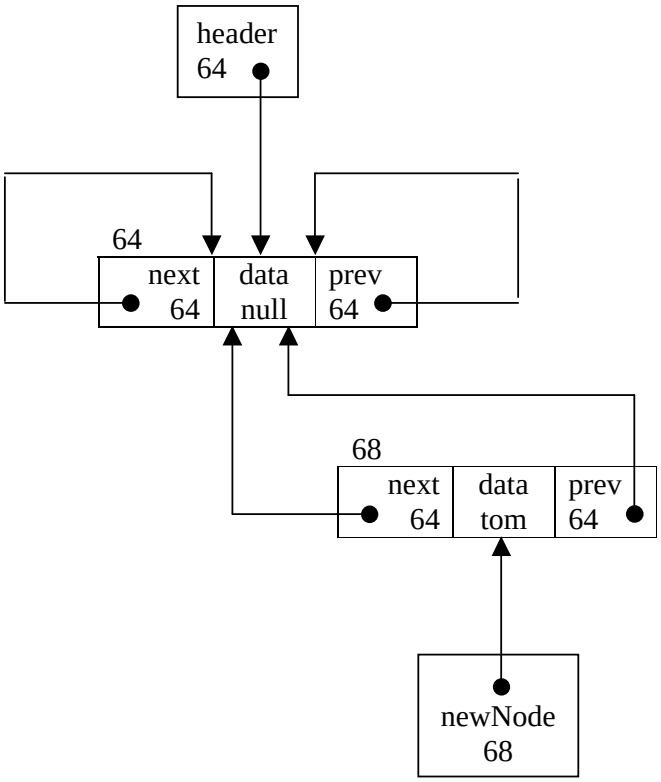

```
private Node header = new Node(null, null, null);
private int size = 0;


public LinkedList()
/* initialises a new, empty linked list.
*/
{
  header.next = header.previous = header;
}
```

A list has two fields. The *header* field is a link into the list. *size* contains the number of nodes in the list.

Initially, *size* is zero and the *header next* and *previous* link fields both refer back to the *header*.

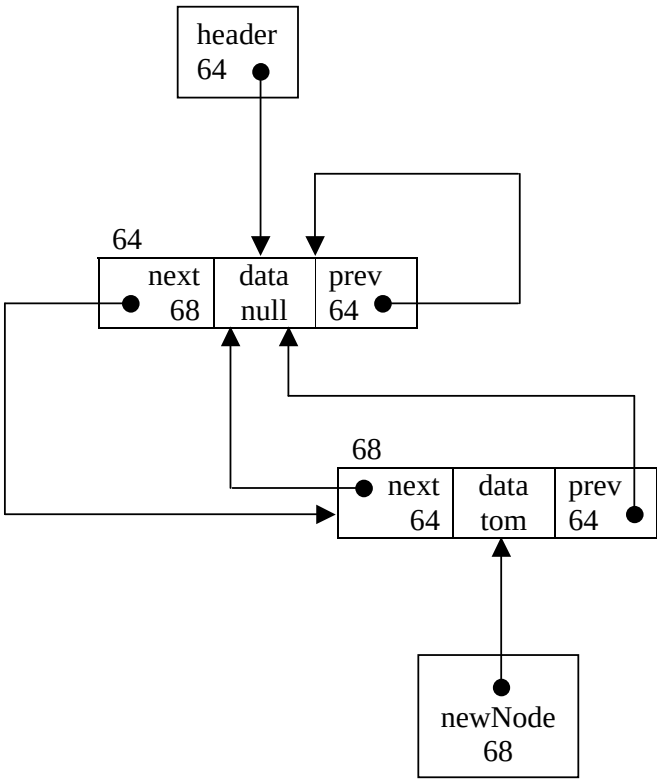
## 14.6 ADD(OBJECT)

We add a given object to an empty list. We create a new node with its data field referring to the given object, its *next* field referring to the *header* node, and its *previous* field with the same value as the *header* node's *previous* value.



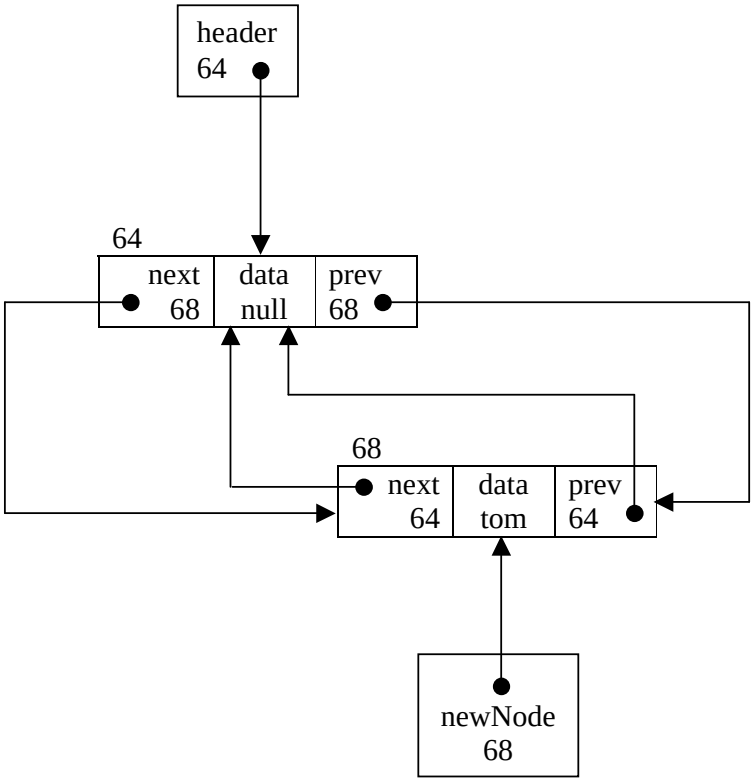

```
Node newNode = new Node(obj, header, header.previous);
```

Then we update *header*'s link fields. Into *newNode.previous.next* (just follow the arrows) we put the value contained in *newNode*.




```
newNode.previous.next = newNode;
```

Into *newNode.next.previous* we put the value contained in *newNode*.

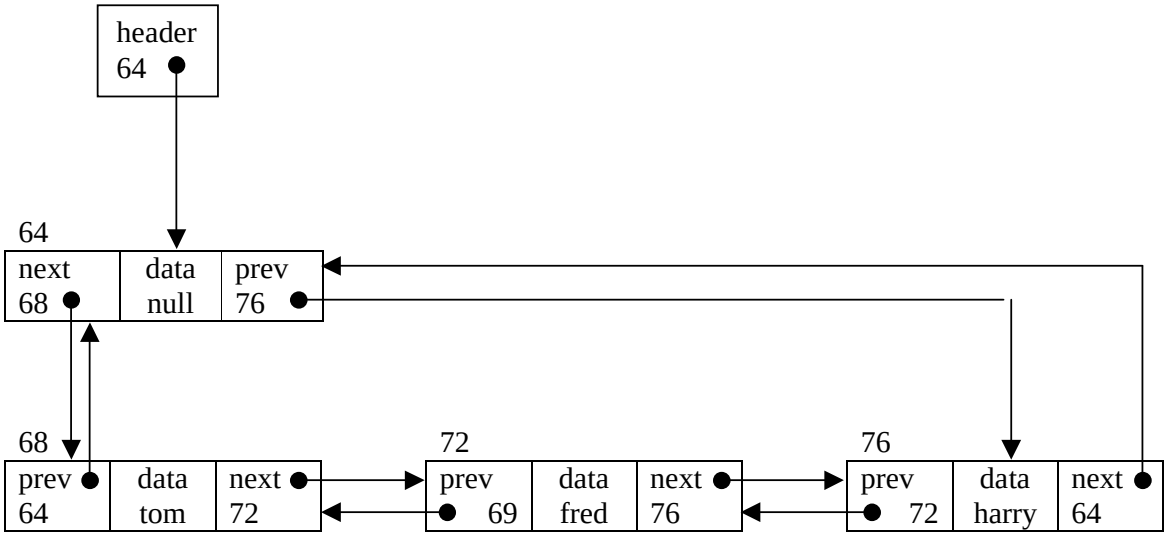



```
   newNode.next.previous = newNode;

public String add(Object obj)
/*  adds the given object to the end of this list and
    returns success.
*/
{
  return addBefore(obj, header);
}

private String addBefore(Object obj, Node n)
/* inserts the given object before the given node and
   returns success.
*/
{
  Node newNode = new Node(obj, n, n.previous);
  newNode.previous.next = newNode;
  newNode.next.previous = newNode;
  size++;
  return "success";
}
```

**Exercise**: by drawing a sequence of diagrams, supported by descriptive prose and snippets of Java code, show how you would add a new data item (*fred*) to a list containing just one data item.

## 14.7 REMOVE(OBJECT)

We start with a list containing three data items, *tom*, *fred* and *harry*. We show how to remove object *fred* from the list.

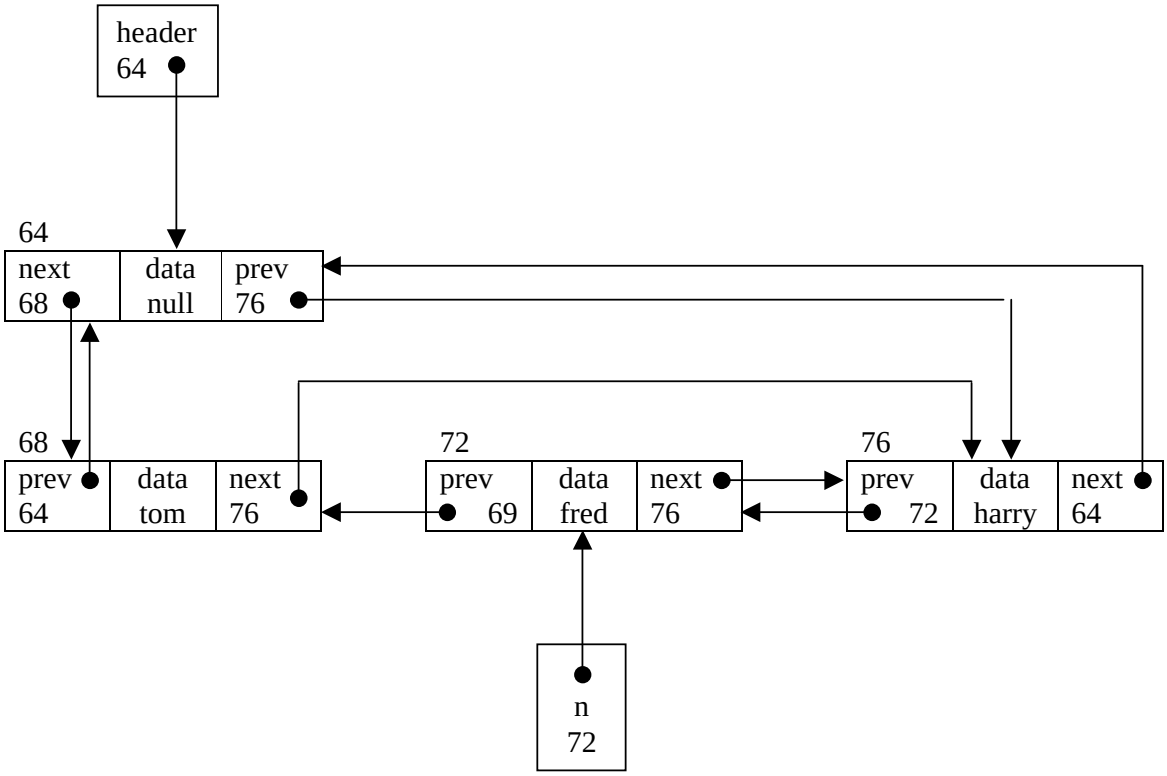



First, we find the node to be removed. Into node *n* we repeatedly put *n.next* until *n.data* equals the object we want to remove.

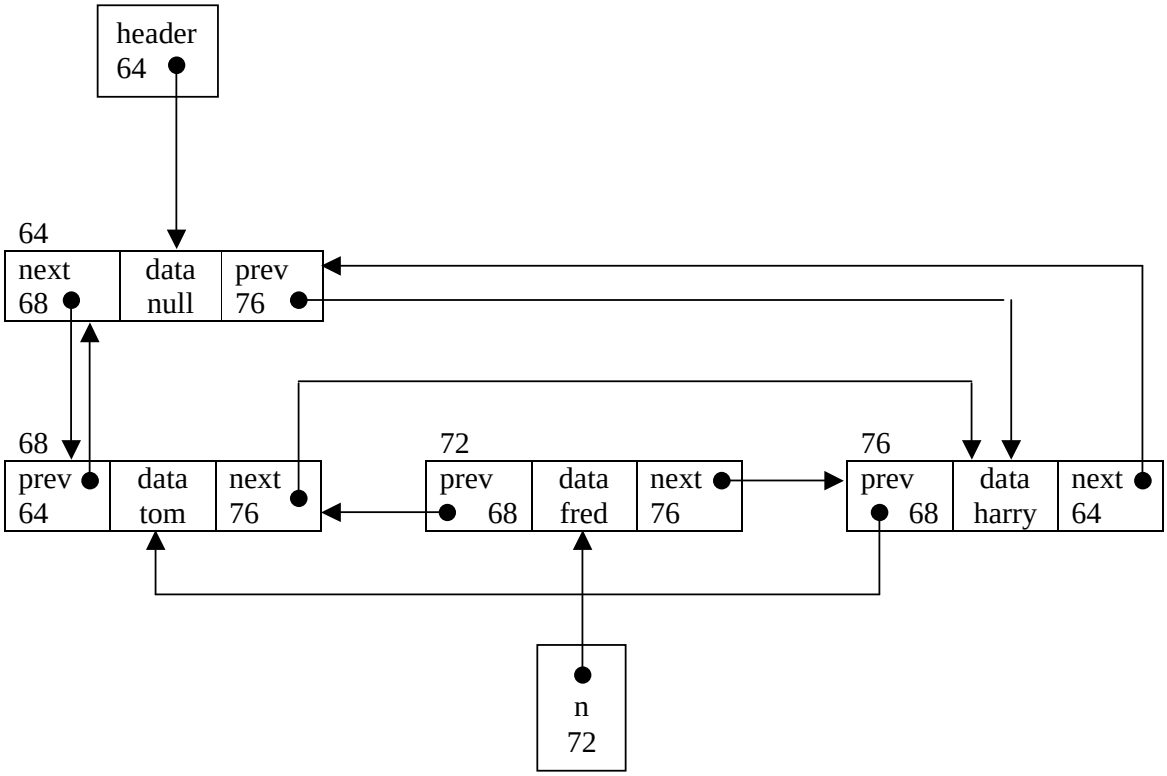
```
for (Node n = header.next; n != header; n = n.next) {
  if (obj.equals(n.data)) {
    remove(n);
  }
}
```

To remove a node is simple; we just update the previous node's next link field to bypass the node.

Having found the node, *n*, to be removed: into *n.previous.next* we put *n.next*



Then into *n.next.previous* we put *n.previous*.




```
private String remove(Node n)
{
  n.previous.next = n.next;
  n.next.previous = n.previous;
```

The methods to remove a given object are shown in their entirety below.

```
private String remove(Node n)
/* removes the given node from this list and returns success
   if the given node is in this list, otherwise
   returns failure.
*/
{
  if (n == header)
    return "failure - no such node";
  n.previous.next = n.next;
  n.next.previous = n.previous;
  size--;
  return "success";
}


public String remove(Object obj)
/* removes the given object from this list and
   returns success if the given object is in this list,
   otherwise returns failure.
   Requires equals to be defined for the given object.
*/
{
  for (Node n = header.next; n != header; n = n.next) {
    if (obj.equals(n.data)) {
      return remove(n);
    }
  }
  return "failure - given object not in list";
}
```

**Exercise**. Start with a list containing two data items (*tom* and *harry*). By drawing a sequence of diagrams, supported by descriptive prose and snippets of Java code, show how you would remove **(a)** *harry* then **(b)** *tom*.

## 14.8 LINKED LIST IMPLEMENTATION

```
/* LinkedList.java
   Terry Marris  30 July 2001
*/

public class LinkedList implements List {
  private static class Node {
  /* Defines a node, used by LinkedList
  */
    Object data;
    Node next;
    Node previous;

    Node(Object data, Node next, Node previous)
    {
      this.data = data;
      this.next = next;
      this.previous = previous;
    }
  }


  private Node header = new Node(null, null, null);
  private int size = 0;


  public LinkedList()
  /* initialises a new, empty linked list.
  */
  {
    header.next = header.previous = header;
  }


  public int size()
  /* returns the number of elements in this list.
  */
  {
    return size;
  }
```

```
private String remove(Node n)
/* removes the given node from this list and returns success
   if the given node is in this list, otherwise
   returns failure.
*/
{
  if (n == header)
    return "failure - no such node";
  n.previous.next = n.next;
  n.next.previous = n.previous;
  size--;
  return "success";
}


private String addBefore(Object obj, Node n)
/* inserts the given object before the given node and
   returns success.
*/
{
  Node newNode = new Node(obj, n, n.previous);
  newNode.previous.next = newNode;
  newNode.next.previous = newNode;
  size++;
  return "success";
}


public String add(Object obj)
/*  adds the given object to the end of this list and
    returns success.
*/
{
  return addBefore(obj, header);
}
```

```
public String remove(Object obj)
/* removes the given object from this list and
   returns success if the given object is in this list,
   otherwise returns failure.
   Requires equals to be defined for the given object.
*/
{
  for (Node n = header.next; n != header; n = n.next) {
    if (obj.equals(n.data)) {
      return remove(n);
    }
  }
  return "failure - given object not in list";
}


private Node node(int index)
/* returns the indexed node.
   requires index to be in range 0..size-1
*/
{
  if (index < 0 || index >= size)
    return null;
  Node n = header;
  for (int i = 0; i <= index; i++) {
    n = n.next;
  }
  return n;
}


public Object get(int index)
/* returns the object at the given indexed node.
   index must be within 0..size()-1.
*/
{
  if (index < 0 || index >= size)
    return null;
  return node(index).data;
}
```

```
  public String add(int index, Object obj)
  /* inserts the given objects at the given index position
     in this list.
  */
  {
    if (index < 0 || index > size)
      return "failure - index out of range";
    if (index == size)
      addBefore(obj, header);
    else
      addBefore(obj, node(index));
    return "success";
  }


  public String remove(int index)
  /* removes the node at the given index and returns success.
     requires index to be in the range 0..size()-1
  */
  {
    if (index < 0 || index >= size)
      return "failure - index out of range";
    Node n = node(index);
    return remove(n);
  }


  public int indexOf(Object obj)
  /* returns the index of the given object, or -1 if the given
     object is not in this list.
  */
  {
    int index = 0;
    for (Node n = header.next; n != header; n = n.next) {
      if (obj.equals(n.data))
        return index;
      index++;
    }
    return -1;
  }
}
```

```
/* TestLinkedList.java
   Terry Marris  30 July 2001
*/

public class TestLinkedList {
  public static void main(String[] s)
  {
    LinkedList list = new LinkedList();
    list.add(new String("dunlop"));
    list.add(new String("michelin"));
    list.add(new String("goodyear"));

    for (int index = 0; index < list.size(); index++) {
      System.out.println(list.get(index));
    }
  }
}
```

**Output**

```
dunlop
michelin
goodyear
```

## 14.9 REVIEW

## 14.10 FURTHER READING

*c:\jdk1.2\src\java\util\LinkedList.java*

This is the source code for the Java library version of *LinkedList*. You might need to unpack the contents of *src.jar*. To do so use

*jar xvf src.jar*

These library source files are provided for you to learn from and should not be changed in any way!

## 14.11 EXERCISES

**1** Draw a linked list containing three objects. Explain how the method *private Node node(int index)* works. Hence (or otherwise) explain how the method

**(a)** *public object get(int index)*
**(b)** *public String remove(int index)*

**2** Draw a linked list containing two objects. By a sequence of diagrams, supported by descriptive prose and snippets of Java code, show how *public String add(int index, Object obj)* works with an index argument value of two and object *anna*.

**3** Draw a linked list containing two objects. By a sequence of diagrams, supported by descriptive prose and snippets of Java code, show how *public String remove(int index)* works to remove the last object in the list.

**4** Draw a linked list containing one object. By a sequence of diagrams, supported by descriptive prose and snippets of Java code, show how *public String remove(int index)* works to remove that object from the list.

**5** Comprehensively test every method of the *LinkedList* class described in §14.8 above. Pay particular attention to boundaries e.g. beginning and end of lists containing zero, one, two and three elements. For example, given a list of three elements can you (a) insert a new element in index location zero? (b) add a new element at index location three?