Linked Lists

Chapter 12.3 in Savitch

```java
public class StrangeObject {
    String name;
    StrangeObject other;
}
```

Preliminaries

- Arrays are not always the optimal data structure:
  - An array has fixed size – needs to be copied to expand its capacity
  - Adding in the middle of an array requires copying all subsequent elements
- ArrayLists have the same issues since they use arrays to store their data.

Objects and references

- Object variables do not actually store an object; they store the address of an object’s location in the computer's memory (references / pointers).
- Example:
  ```java
  int [] values = new int[5];
  values[0] = 7;
  System.out.println(values[0]); // 7
  ```

Java References

- When one reference variable is assigned to another, the object is **not** copied; both variables refer to the same object.
  ```java
  int[] a1 = {4, 5, 2, 12, 14, 14, 9};
  int[] a2 = a1; //refers to same array as a1
  a2[0] = 7;
  System.out.println(a1[0]); // 7
  ```
Self references

- Consider the following class:
  ```java
  public class StrangeObject {
      String name;
      StrangeObject other;
  }
  ```
  Will this compile?

Linking self-referential nodes

- Each node object stores:
  - one piece of integer data
  - a reference to another node
- IntegerNode objects can be "linked" into chains to store a list of values:
  ```java
  IntegerNode
    // item 42
    // item -3
    // item 17
    // item 9
    null
  ```

The complete IntegerNode class

```java
public class IntegerNode {
    private int item;
    private IntegerNode next;
    public IntegerNode(int item) {
        this.data = item;
        this.next = null;
    }
    public IntegerNode(int item, IntegerNode next) {
        this.data = item;
        this.next = next;
    }
    public void setNext(IntegerNode nextNode) {
        next = nextNode;
    }
    public IntegerNode getNext() {
        return next;
    }
    public void setItem(Object item) {
        this.item = item;
    }
    public Object getItem() {
        return item;
    }
}
```

Exercise

- Exercise: Write code to produce the following list:
  ```java
  IntegerNode
    // item 42
    // item -3
    // item 17
    // item 9
    null
  ```
Exercise

What set of statements turns this list:

list → item next
10 → 20

Into this?

list → item next
30 → 10 → 20

Exercise

Let’s write code that creates the following list:

list → item next
10 → 20

Which is correct?

a) list = new IntegerNode(10, new IntegerNode(20));
b) list = new IntegerNode(20, new IntegerNode(10));
c) Neither will correctly produce that list

Exercise

What set of statements turns this list:

list → item next
10 → 20

Into this?

list → item next
30 → 10 → 20

list = new IntegerNode(30, list);
**Exercise**

- What set of statements turns this list:

  ```
  list 10 --- 20
  ```

- Into this?

  ```
  list 10 --- 20 --- 30
  ```

```
list.getNext().setNext(new IntegerNode(30));
```

**A more flexible version**

```java
public class Node {
    private Object item;
    private Node next;
    public Node(Object item) {
        this.item = item;
        this.next = null;
    }
    public Node(Object item, Node next) {
        this.item = item;
        this.next = next;
    }
    public void setNext(Node nextNode) {
        next = nextNode;
    }
    public Node getNext() {
        return next;
    }
    public Object getItem() {
        return item;
    }
    public void setItem(Object item) {
        this.item = item;
    }
}
```

Node node = new Node(5);
Java will convert 5 to an instance of class Integer

**Printing a linked list**

- Suppose we have a chain of nodes:

  ```
  head 10 --- 20 --- ... 990
  ```

- And we want to print all the items.

```java
head = head.getNext(); // is this a good idea?
```
Printing a linked list

To not lose the reference to this first node:
Node current = head;

Move along a list by advancing a Node reference:
current = current.getNext();

Code for printing the nodes of a list:
Node head = ...;
Node current = head;
while (current != null){
    System.out.println(current.getItem());
    current = current.getNext();
}

Similar to array code:
int[] a = ...;
int i = 0;
while (i < a.length) {
    System.out.println(a[i]);
    i++;
}

Interim summary – why should I care?

- Linked list:
  - a self referential structure
- Advantage over arrays – no bound on capacity – can grow/shrink as needed (a dynamic structure)
- Linked lists are the basis for a lot of data structures:
  - stacks, queues, trees
- The primary alternative to arrays
### The list interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object get(index)</code></td>
<td>Returns the element at the given position</td>
</tr>
<tr>
<td><code>indexOf(object)</code></td>
<td>Returns the index of the first occurrence of the specified element</td>
</tr>
<tr>
<td><code>add(object)</code></td>
<td>Appends an element to the list</td>
</tr>
<tr>
<td><code>add(index, object)</code></td>
<td>Inserts given value at given index, shifting subsequent values right</td>
</tr>
<tr>
<td><code>remove(index)</code></td>
<td>Removes the element at the specified position and returns it</td>
</tr>
<tr>
<td><code>remove(object)</code></td>
<td>Removes the element that corresponds to the given object (and returns it)</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>Returns the size of the list</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Indicates if the list is empty</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Removes all elements from the list</td>
</tr>
</tbody>
</table>

Index is an `int`, and `object` is of type `Object`.

### Linked List: constructor

```java
public class LinkedList {
    private Node head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }
    ...
}
```

### Implementing add

- How do we add to a linked list at a given index?

```java
public void add(int index, Object item) throws ListIndexOutOfBoundsException {
    ...
}
```
Implementing add

- How do we add to a linked list at a given index?
- Did we consider all the possible cases?

```java
public void add(int index, Object item) {
    if (index<0 || index>size)
        throw new IndexOutOfBoundsException("out of bounds");
    if (index == 0) {
        head = new Node(item, head);
    } else { // find predecessor of node
        Node curr = head;
        for (int i=0; i<index-1; i++)
            curr = curr.getNext();
        curr.setNext(new Node(item, curr.getNext()));
    }
    size++;
}
```

Implementing remove

```java
public void remove(int index) {
    // Removes value at a given index
    head = ...
    size = ...
    item next
    42 ——- -3 ——- 17 ——- 9 head

    Before removing element at index 1:
    head = size = 3
    item next
    42 ——- -3 ——- 17 ——- 9

    After:
    head = size = 2
    item next
    42 ——- -3 ——- 20
```
Removing the first node from a list

Before removing element at index 0:

- head
- size = 3

After:

- head
- size = 2

List with a single element

Before:

- head
- size = 1

After:

- head
- size = 0

We must change head to null.
Do we need a special case to handle this?

The remove method

```java
public void remove(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException("List index out of bounds");
    if (index == 0) {
        // special case: removing first element
        head = head.getNext();
    } else {
        // removing from elsewhere in the list
        Node current = head;
        for (int i = 0; i < index - 1; i++) {
            current = current.getNext();
        }
        current.setNext(current.getNext().getNext());
        size--;
    }
}
```

The clear method

- How do you implement a method for removing all the elements from a linked list?
The clear method

```java
public void clear() {
    head = null;
}
```

- Where did all the memory go?
- Java’s garbage collection mechanism takes care of it!
- An object is eligible for garbage collection when it is no longer accessible (cyclical references don’t count!)
- In C/C++ the programmer needs to release unused memory explicitly

Linked lists recursively

- We would like to print the elements in a linked list recursively.
  - What would be the signature of the method?
  - Base case?
  - Recursive case?

Recursive linked list traversal – which is correct?

```java
private void writeList(Node node) {
    if (node != null) {
        System.out.println(node.getItem());
        writeList(node.getNext());
    }
}
```

- a) private void writeList(Node node) {
- b) private void writeList(Node node) {

Recursive linked list traversal

```java
private void writeList(Node node) {
    // precondition: linked list is referenced by node
    // postcondition: list is displayed, list is unchanged
    if (node != null) {
        // write the first item
        System.out.println(node.getItem());
        // write the rest of the list
        writeList(node.getNext());
    }
}
```
Recursive backward traversal

- We have two ways for recursively traversing a string backwards:
  - Write the last character of the string s
  - Write string s minus its last character backward
  - Write string s minus its first character backward
  - Write the first character of string s

Recursive backward traversal

- Translated to our problem:
  - Write the last node of the list
  - Write the list minus its last node backward
  - Write the list minus its first node backward
  - Write the first node of the list

Which of these strategies is better for linked lists?

---

Recursive backward traversal

```java
private void writeListBackward(Node node) {
    //precondition: linked list is referenced by node
    //postcondition: list is displayed. list is unchanged
    if (node != null) {
        // write the rest of the list
        writeListBackward(node.getNext());
        // write the first item
        System.out.println(node.getItem());
    }
}
```

Recursive add method

```java
public void add(Object item) {
    head = addRecursive(head, item);
}

private Node addRecursive(Node node, Object item) {
    if (node == null) {
        node = new Node(item, node);
    } else {// insert into the rest of the linked list
        node.setNext(addRecursive(node.getNext(), item));
    }
    return node;
}
```
Proof of correctness

```java
private Node addRecursive(Node node, Object item) {
    if (node == null) {
        node = new Node(item, node);
    } else {// insert into the rest of the linked list
        node.setNext(addRecursive(node.getNext(), item));
    }
    return node;
}
```

- **Base case:** If we have reached the end of the list, it correctly returns a link to the newly inserted node.
- **Recursive case:** Assuming that the recursive call correctly returns a reference to the rest of the list with the element added, then setting that reference results in correctly adding the node.

Variations

- **Circular linked list**

- **Doubly linked list**

- What are the advantages and disadvantages of a doubly linked list?

Inner classes

- **Inner class:** defined inside another class
- If declared private it can’t be used by other classes
- The methods of the inner and outer classes have access to each other’s methods and instance variables, even if declared private.
- How would we use an inner class in implementing a linked list?