**Linked Lists**

Chapter 12.3 in Savitch

```
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 moving 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];
  int a = values[0];
  int b = values[1];
  int c = values[2];
  System.out.println("a = " + a);
  System.out.println("b = " + b);
  System.out.println("c = " + c);
  ```

**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**

```
public class IntegerNode {
    int item;
    IntegerNode next;
}
```

- 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:
The complete IntegerNode class

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

Exercise

**What set of statements turns this list:
```java
list → item
  next → item
  next → item
```**

Into this?

**list → item
  next → item
  next → item
  next → item
**

Exercise

**Let’s write code that creates the following list:
```java
list → item
  next → item
  next → item
```**

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:
```java
list → item
  next → item
  next → item
```**

Into this?

**list → item
  next → item
  next → item
  next → item
**

list = new IntegerNode(30, list);
Exercise

What set of statements turns this list:

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

Into this?

```
list
```

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
```

And we want to print all the items.

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();
```
Printing a linked list

Same thing with a for loop

```java
Node head = ...;
for (Node current = head; current != null; current =
current.getNext()) {
    System.out.println(current.getItem());
}
```

the array version

```java
int[] a = ...;
for (int i = 0; i < a.length; i++) {
    System.out.println(a[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

```
public interface ListInterface {
    public boolean isEmpty();
    public int size();
    public void add(int index, Object item)
        throws ListIndexOutOfBounds;
    public void add(Object item);
    public void remove(int index)
        throws ListIndexOutOfBounds;
    public void remove(Object item);
    public Object get(int index)
        throws ListIndexOutOfBounds;
    public void clear();
}
```

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?

```
Item next Item next Item next Item next
42 — 3 — 17 — 9
```
Implementing add

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

```java
Implementing add
```

```java
The add method
```

```java
public void add(int index, Object item) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("out of bounds");
    } 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
Implementing remove
```

```java
public void remove(int index) {
...
```

```java
Removing a node from a list
```

```java
Removing the first node from a list
```

```java
List with a single element
```

```java
- Before removing element at index 0:
- After:
- 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

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

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) {
        // write the first item
        System.out.println(node.getItem());
        writeList(node.getNext());
    }
}
```

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());
        writeList(node.getNext());
        // 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
  - And
  - Write string \(s\) minus its first character backward
  - Write the first character of string \(s\)

Translated to our problem:

- Write the last node of the list
- Write the list minus its last node backward
- And
- 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 add method

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

```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;
}
```

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?

Image from: http://en.wikipedia.org/wiki/Linked_list