Linked Lists

Chapter 12.3 in Savitch
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];
  ```

```
values -> 5 7 10 6 3
```

```
int x = 1;
```
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
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>9</td>
</tr>
</tbody>
</table>
Consider the following class:

```java
public class Node {
    String name;
    Node next;
}
```

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 list node

- IntegerNodes 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.data = 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 int getItem() {
        return item;
    }
    public void setItem(int item) {
        this.item = item;
    }
}
```
public class IntegerNode {
    private int item;
    private IntegerNode next;

    public IntegerNode(int item) {...}
    public IntegerNode(int item, IntegerNode next) {...}
    public void setNext(IntegerNode nextNode) {...}
    public IntegerNode getNext() {...}
}

Exercise: Write code to produce the following list

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
<tr>
<td></td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>
Exercise

- What set of statements turns this list:

```
list →
```

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

```
item next
20
```

- Into this?

```
list →
```

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

```
item next
10
```

```
item next
20
```
Exercise

- What set of statements turns this list:

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Into this?

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Let’s write code that creates the following list:

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 →

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

  →

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

- Into this?

  list →

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

  →

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

  →

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
Exercise

- What set of statements turns this list:

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

- Into this?
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 Integer
Printing a linked list

- Suppose we have a chain of nodes:

```
head → [item 10] → [item 20] → ... → [item 990]
```

- And we want to print all the items.
Printing a linked list

- Start at the **head** of the list.
- While (there are more nodes to print):
  - Print the current node's **item**.
  - Go to the **next** node.

How do we walk through the nodes of the list?

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

![Diagram of a linked list](image)
Printing a linked list

- **Important:** A `Node` variable is NOT a `Node` object!

  ```java
  Node current = head;
  ```

- Move along a list by advancing a `Node` reference:

  ```java
  current = current.getNext();
  ```
Printing a linked list

Code for printing the nodes of a list:

```java
Node head = ...;

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

Similar to array code:

```java
int[] a = ...;

int i = 0;
while (i < a.length) {
    System.out.println(a[i]);
    i++;
}
```
Printing a linked list

Same thing with a for loop

Node head = ...

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

the array version

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

<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>index 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>object remove(index)</code></td>
<td>Removes the element at the specified position (and returns it)</td>
</tr>
<tr>
<td><code>object remove(object)</code></td>
<td>Removes the element that corresponds to the given object (and returns it)</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>returns the size of the list</td>
</tr>
<tr>
<td><code>boolean 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*
The list interface

class ListInterface {
    public boolean isEmpty();
    public int size();
    public void add(int index, Object item)
        throws ListIndexOutOfBoundsException;
    public void add(Object item);
    public void remove(int index)
        throws ListIndexOutOfBoundsException;
    public void remove(Object item);
    public Object get(int index)
        throws ListIndexOutOfBoundsException;
    public void clear();
}
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?

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>null</td>
</tr>
</tbody>
</table>
Implementing add

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

```
<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>
```
```
<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>
```
```
<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>
```
```
<table>
<thead>
<tr>
<th>item</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>null</td>
</tr>
</tbody>
</table>
```
The add method

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

// Removes value at a given index
public void remove(int index) {
    ...
}

- How do we remove a node?
Removing a node from a list

- Before removing element at index 1:
  - Head = 3
  - Size = 3

- After:
  - Head = 2
  - Size = 2
Removing the first node from a list

- Before removing element at index 0:

  - head = 42
  - size = 3

- After:

  - head = -3
  - size = 2
List with a single element

- Before:

  - head = element 0
  - size = 1

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

- After:

  - head = null
  - 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;
    size = 0
}
```

- 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?

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

b) private void writeList (Node node) {
    if (node != null) {
        writeList(node.getNext());
        System.out.println(node.getItem());
    }
}
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
  And
  - 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
  
  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 backward traversal

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());
    }
}
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

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