Singly Linked Lists

- Representation
- Space Analysis
- Creation and Insertion
- Traversal
- Search
- Deletion
Representation

- We are using a representation in which a linked list has both head and tail references.

```java
public class MyLinkedList{
    protected Element head;
    protected Element tail;
    public final class Element{
        Object data;
        Element next;
        Element(Object obj, Element element){
            data = obj;
            next = element;
        }
        public Object getData(){return data;}
        public Element getNext(){return next;}
    }
}
```
Representation: Space Analysis

• Now, we can take a look at the space requirements:

\[ S(n) = \text{sizeof(MyLinkedList)} + n \times \text{sizeof(MyLinkedList.Element)} \]
\[ = 2 \times \text{sizeof(MyLinkedList.Element ref)} + n \left[ \text{sizeof(Object ref)} + \text{sizeof(MyLinkedList.Element ref)} \right] \]
\[ = (n + 2) \times \text{sizeof(MyLinkedList.Element ref)} + n \times \text{sizeof(Object ref)} \]

<table>
<thead>
<tr>
<th>Space Require</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof(MyLinkedList)</td>
<td>The list reference has two fields: head (type: Element) and tail (type: Element)</td>
</tr>
<tr>
<td></td>
<td>= 2 \times \text{sizeof(MyLinkedList.Element ref)}</td>
</tr>
<tr>
<td>( n ) \times \text{sizeof(MyLinkedList.Element)}</td>
<td>The list has ( n ) elements of type \text{Element}. Each element has two fields-- data (type Object) and next (type Element).</td>
</tr>
</tbody>
</table>
List Creation and Insertion

• An empty list is created as follows:

```java
MyLinkedList list = new MyLinkedList();
```

• Once created, elements can be inserted into the list using either the `append` or `prepend` methods

```java
for (int k = 0; k < 10; k++)
    list.append(new Integer(k));
```

• Also if we have reference to a node (an element), we can use `insertAfter` or `InsertBefore` of the Element class.
public void append(Object obj) {
    Element element = new Element(obj, null);
    if (head == null)
        head = element;
    else
        tail.next = element;
    tail = element;
}

Complexity is $O(1)$
public void prepend(Object obj) {
    Element element = new Element(obj, head);
    if (head == null)
        tail = element;
    head = element;
}

Complexity is O(1)
public void insertBefore(Object obj) {
    Element element = new Element(obj, this);
    if(this == head) {
        head = element;
        return;
    }
    Element previous = head;
    while (previous.next != this) {
        previous = previous.next;
    }
    previous.next = element;
}

public void insertAfter(Object obj) {
    next = new Element(obj, next);
    if(this == tail)
        tail = next;
}
Traversing

To move a reference $e$ from one node to the next:

```java
public int countNodes()
{
    int count = 0;
    Element e = head;
    while (e != null) {
        count++;
        e = e.next;
    }
    return count;
}
```

Example: Count the number of nodes in a linked list.

Complexity is $O(n)$
Searching

- To search for an element, we traverse from head until we locate the object.
  
  Example: Count the number of nodes with data field equal to a given object.

```java
public int countNodes(Object obj){
    int count = 0;
    Element e = head;
    while(e != null){
        if(e.data.equals(obj))
            count++;
        e = e.next;
    }
    return count;
}
```

Complexity is $O(n)$
Deletion

• To delete an element, we use either the `extract` method of `MyLinkedList` or that of the `Element` inner class.

```java
public void extract(Object obj) {
    Element element = head;
    Element previous = null;
    while(element != null && ! element.data.equals(obj)) {
        previous = element;
        element = element.next;
    }

    if(element == null)
        throw new IllegalArgumentException("item not found");
    if(element == head)
        head = element.next;
    else
        previous.next = element.next;
    if(element == tail)
        tail = previous;
}
```

Complexity is $O(n)$
Deletion - Difference between the MyLinkedList and the Element extracts

• To delete an element, we use either the `extract` method of MyLinkedList or that of the Element inner class.

```java
try{
    list.extract(obj1);
} catch(IllegalArgumentException e){
    System.out.println("Element not found");
}

MyLinkedList.Element e = list.find(obj1);
if(e != null)
    e.extract();
else
    System.out.println("Element not found");
```
Deletion – Deleting First and Last Element

```java
public void extractFirst() {
    if (head == null)
        throw new IllegalArgumentException("item not found");
    head = head.next;
    if (head == null)
        tail = null;
}

Complexity is O(1)
```

```java
public void extractLast() {
    if (tail == null)
        throw new IllegalArgumentException("item not found");
    if (head == tail)
        head = tail = null;
    else {
        Element previous = head;
        while (previous.next != tail)
            previous = previous.next;
        previous.next = null;
        tail = previous;
    }
}

Complexity is O(n)
```
Exercises

• For the MyLinkedList class, implement each of the following methods:
  – String toString()
  – Element find(Object obj)
  – void insertAt(int n) //counting the nodes from 1.

State the complexity of each method.

• Which methods are affected if we do not use the *tail* reference in *MyLinkedList* class.