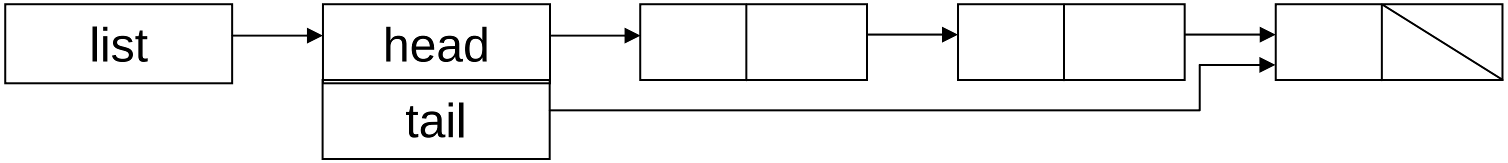
Singly Linked Lists

- 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 .**

```
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) = sizeof(MyLinkedList) + n sizeof(MyLinkedList.Element)

= 2 sizeof(MyLinkedList.Element ref) + n [sizeof(Object ref) + sizeof(MyLinkedList.Element ref)]

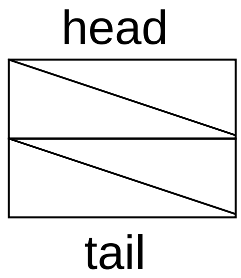
= (n + 2) sizeof(MyLinkedList.Element ref) + n sizeof(Object ref)

Space Require	Explanation
sizeof(MyLinkedList)	The list reference has two fields: ***head*** (type: *Element*) and ***tail*** (type: *Element*) = 2 sizeof(MyLinkedList.Element ref)
n sizeof(MyLinkedList.Element)	The list has n elements of type ***Element***. Each element has two fields-- ***data*** (type *Object*) and ***next*** (type *Element*).

List Creation and Insertion

- An empty list is created as follows:

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



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

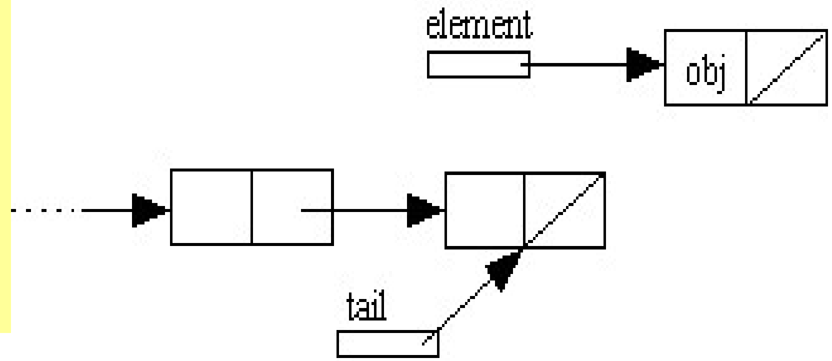
```
for (int k = 0; k < 10; k++)
    list.append(new Integer(k));
```

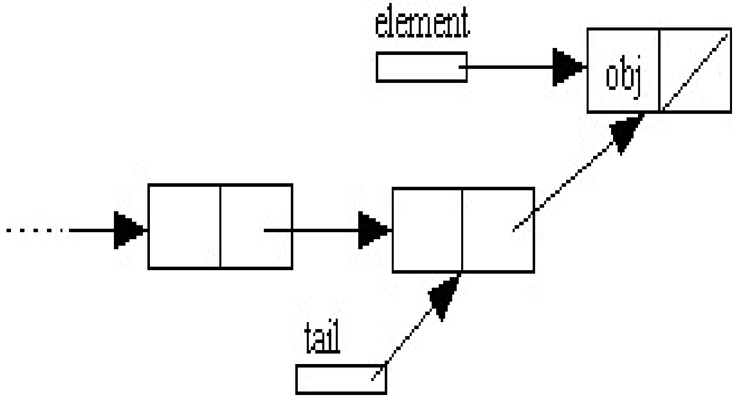
- Also if we have reference to a node (an element), we can use `insertAfter` or `InsertBefore` of the Element class.

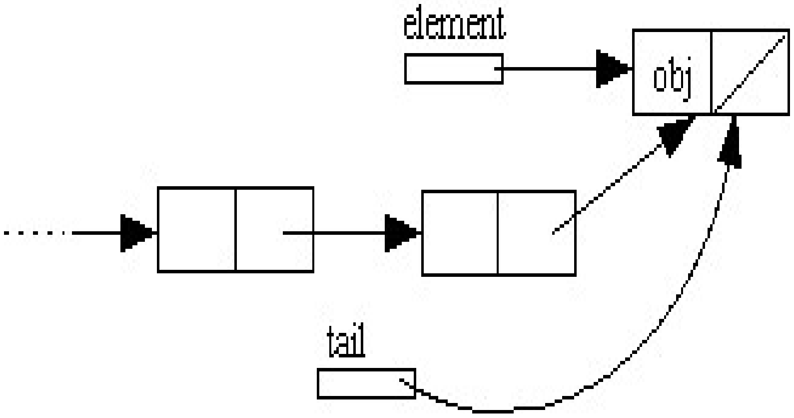
Insertion at the end (Append)

```
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)



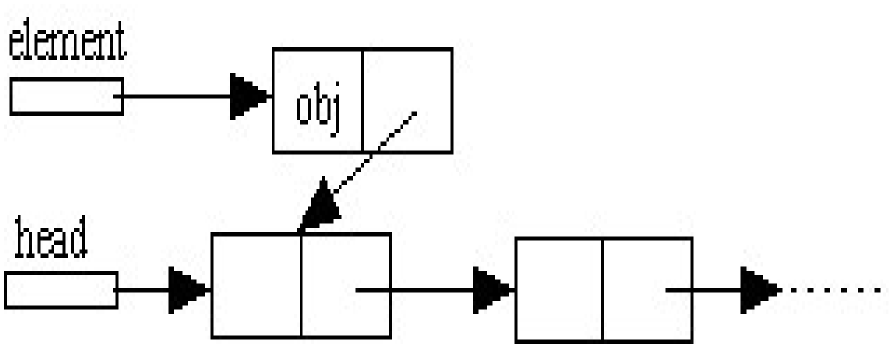


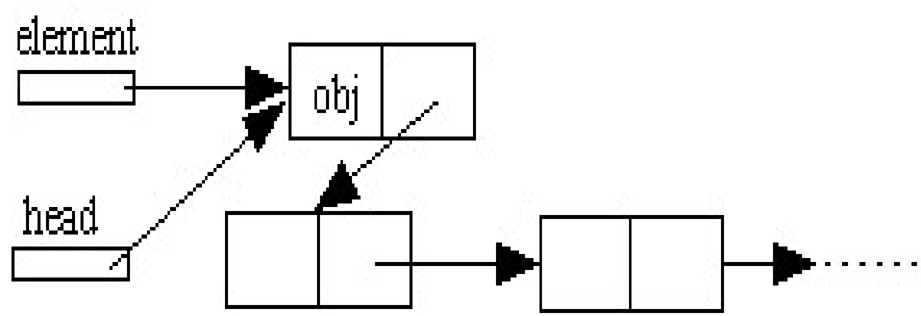


Insertion at the beginning (Prepend)

```
public void prepend(Object obj) {
   Element element = new Element(obj, head);
   if(head == null)
      tail = element;
  head = element;
}
```

Complexity is O(1)





Insertion before and after an element

```
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;
}
```

Complexity is O(n)

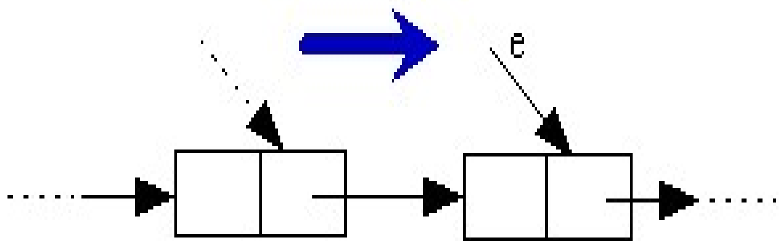
```
public void insertAfter(Object obj) {
   next = new Element(obj, next);
   if(this == tail)
      tail = next;
}
```

Complexity is O(1)

Traversal

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

```
e = e.next;
```



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

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

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.

```
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)

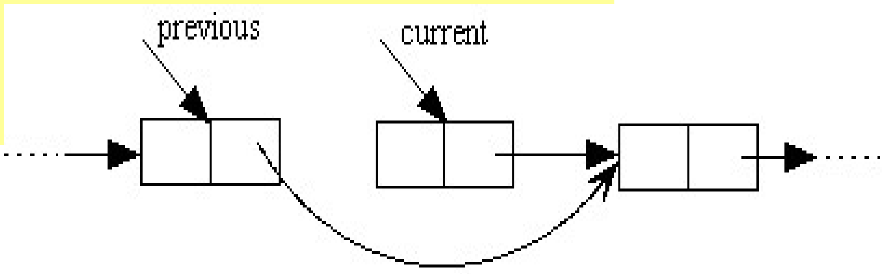
Deletion

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

```
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.

```
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

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

Complexity is O(1)

```
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.