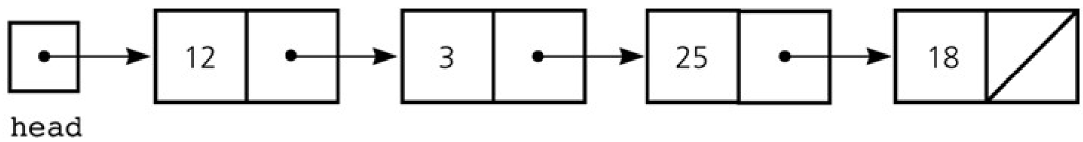
Linked List

Algorithm - abstract way to perform computation tasks
Data Structure - abstract way to organize information

Linked List:



Node:

Data
Next → null

Node code:

```
public class Node<T> {
        private T data;
        private Node<T> next;

        public Node(T data) { this.data = data;  }

        public void setData(T data) {  this.data = data;  }
        public T getData() {  return data;  }

        public Node<T> getNext() {  return next;  }
        public void setNext(Node<T> next) {  this.next = next;  }
}
```

Linked List Code:

```
public class LinkedList<T> {
        private Node<T> head;
}
```

Inserting a new node:

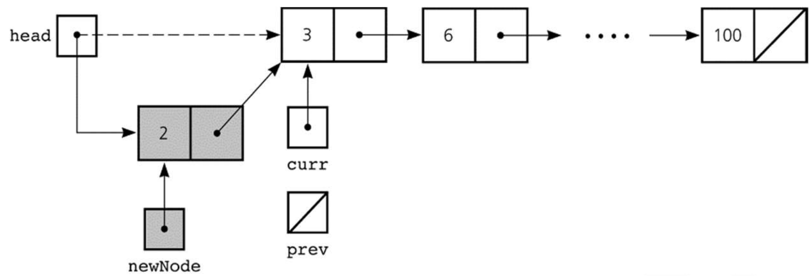
Inserting a Node into a Specified Position of a Linked List:

Three steps to insert a new node into a linked list

- Determine the point of insertion
- Create a new node and store the new data in it
- Connect the new node to the linked list by changing references

Case 1: To insert a node at the beginning of a linked list: (**curr == head**)

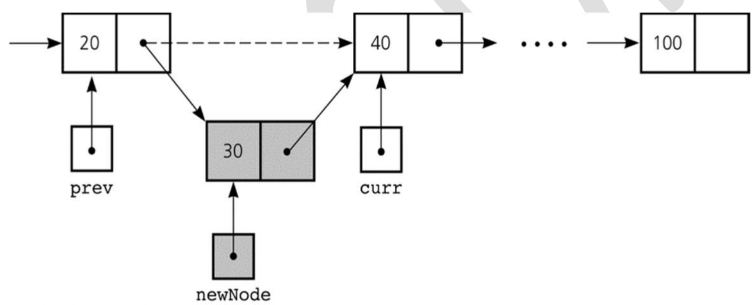
newNode.next = head;
head = newNode;



What's the time complexity of inserting an item to the head? ➔ **O(1)**

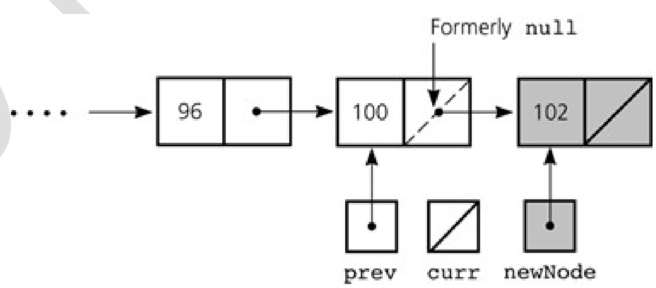
Case 2: To insert a node between two nodes:

newNode.next = curr;
prev.next = newNode;



Case 3: Inserting at the end of a linked list is a special case if **curr** is **null**:

newNode.next = curr;
prev.next = newNode;



Determining **curr** and **prev**

Determining the point of insertion or deletion for a sorted linked list of objects

```
for (    prev = null , curr = head    ;
        (curr != null)  &&  (newNode.data.compareTo(curr.data) > 0);
          prev = curr , curr = curr.next )      ;       // end for
```

Time Complexity ➔ **O(n)**

H.W. ➔ implement insert into a sorted linked list

Create a driver class to test linked list classes.
Override the ***toString*** methods first

Node *toString*:

```
@Override
public String toString() {  return data.toString();  }
```

LinkedList *toString*:

```
@Override
public String toString() {
        String res = "➔";
        Node<T> curr = head;
        while (curr != null) {
                res += curr + "→ ";
                curr = curr.next;
        }
        return res + “NULL”;
}
```

Length of Linked List?

Case 1: If it's empty: head == null ➔ length = 0

Case 2: If not: Make a pointer and move over all the nodes and maintain a **counter**

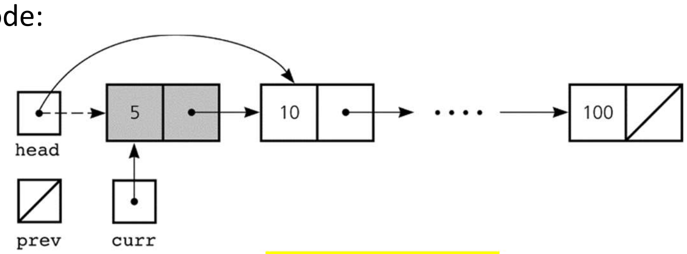
Length code:

```
public int length() {
        int length = 0;
        Node<T> curr = head;
        while (curr != null) {
                length++;
                curr = curr.next;
        }
        return length;
}
```

Time Complexity ➔ O(n)

Deleting Nodes:

Case 1: Deleting the head node:



Simply move the **head** to the **head.next**: head = head.next;

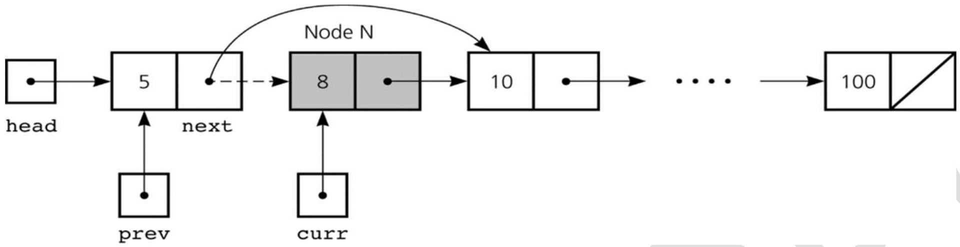
Now first Node has no reference to it ➔ **Garbage**

Time Complexity ➔ **O(1)**

Delete at head code: // make sure linked list is not empty

```
public Node<T> deleteAtStart() {
        Node<T> toDel =head;
        head = head.next;
        return toDel;
}
```

Case 2: Delete node **N** which **curr** references:



Set **next** in the node that precedes **N** to reference the node that follows **N**

```
prev.next = curr.next;      // prev.next = prev.next.next;
```

Searching for an Item in a Linked List:

Search code:

```
public Node<T> find(T data) {
        Node<T> curr =  head;
        while (curr != null) {
                if (curr.getData() == data)    //  if (curr.getData().equals(data))
                        return curr;
                curr = curr.next;
        }
        return null;
}
```

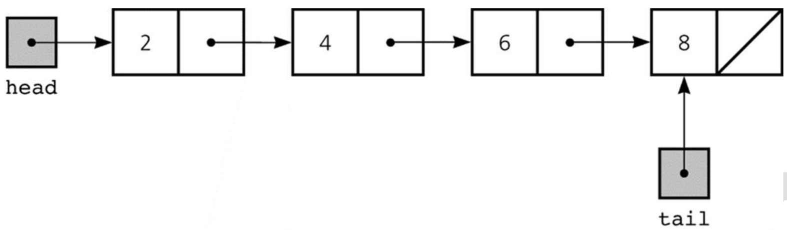
Time Complexity: linear growth ➔ **O(n)**

Variations of the Linked List:

1- Tail References (Doubly Ended Linked List)

- Remembers where the end of the linked list is.
- Therefore, we can add and delete at both ends.
- To add a node to the end of a linked list

  tail.next = new Node(request, null);



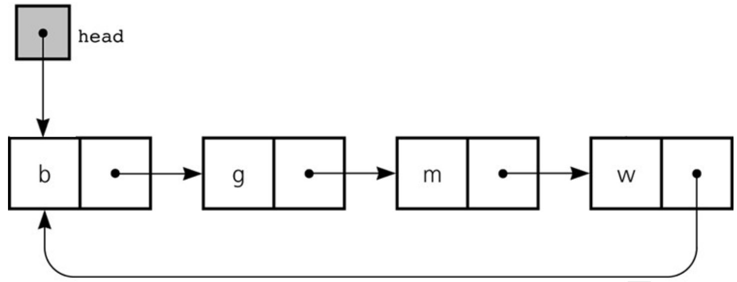

```
public class DoubleEndedList<T> extends LinkedList<T> {
        private Node<T> tail;
        public Node<T> getTail() {      return tail;   }

        public void addAtEnd(T data) {
                Node<T> newNode = new Node<T>(data);
                if (head == null) {  // empty
                        head = newNode;
                        tail = newNode;
                }
                        else {
                        tail.setNext(newNode);
                        tail = newNode;
                }
        }
}
```

Make sure to override addAtStart to set the tail pointer correctly:

```
@Override
public void addAtStart(T data) {
      Node<T> newNode = new Node<T>(data);
      if (head == null) {  // empty
         head = newNode;
         tail = newNode;
      }
      else{
         newNode.setNext(head);
         head = newNode;
      }
}
```

2- Circular Linked List

- Last node references the first node
- Every node has a successor

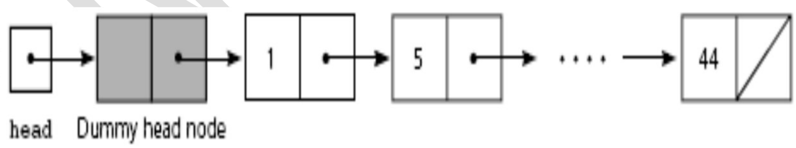


Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

3- Dummy Head Nodes

- Always present, even when the linked list is empty
- Insertion and deletion algorithms initialize **prev** to reference the dummy head node, rather than **null**



Processing Linked Lists Recursively:

- **Traversal**
  - Recursive strategy to display a list
    Write the first node of the list
    Traversal the list minus its first node

```
public static void traversList(Node curr) {
   if(curr == null)
      System.out.println("NULL");
   else {
       System.out.print("[" + curr + "]-->");
       traversList(curr.next);
   }
}
```

- Recursive strategies to display a list backward
  - writeListBackward strategy

    Write the last node of the list

    Write the list minus its last node backward

```
public static void traversListBackward(Node curr) {
   if(curr == null)
      System.out.print("NULL");
   else {
      traversListBackward(curr.next);
      System.out.print("<--[" + curr + "]");
   }
}
```

How to reverse a linked list:

Iterative:

Logic for this would be:

- Have three references i.e. prevNode, currNode and nextNode.
- When currNode is starting node, then prevNode will be null.
- Assign currNode.next to prevNode to reverse the link.
- In each iteration move currNode and prevNode by 1 node.

```
public static Node reverseLinkedList(Node currNode)   {
  Node prevNode=null;    // For first node, prevNode will be null
  Node nextNode;
  while(currNode!=null)    {
    nextNode=currNode.next;
    currNode.next=prevNode;    // reversing the link
    prevNode=currNode;    // moving currNode and prevNode by 1 node
    currNode=nextNode;
  }
  return prevNode;
 }
```

Recursive:

Base case for this would be either node is null or node.next is null.

```
public static Node reverseLinkedList(Node node) {
    if (node == null || node.next == null)
       return node;

    Node remaining = reverseLinkedList(node.next);
    node.next.next = node;
    node.next = null;
    return remaining;
 }
```