About ME:
I come from Bihar Muzaffarpur and currently working in Bangalore as Senior software development engineer. I have done my Bachelor of engineering from BIT Bangalore in Mechanical department. and stated my career as Test Engineer.

https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/

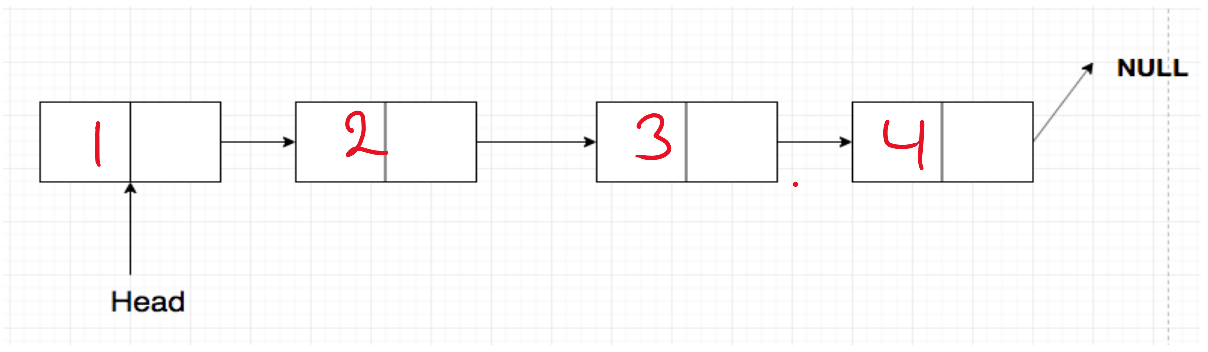
Data structure Topics

What is Linked List

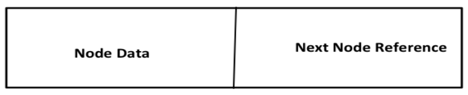
Linked list is data structure and used to store collection of data. It has following properties

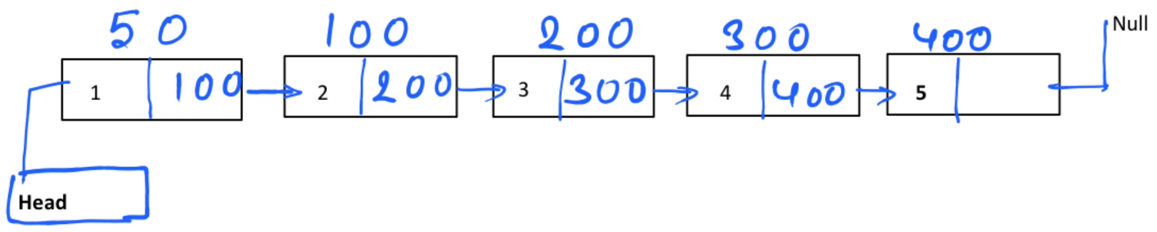
- Each element is connected by pointer.
- Last Element point to NULL
- Can grow and shrink in size during execution of program.
- Can be made as long as required.
- It does not waste memory space but takes some space for processing

Pictorial Representations



Node



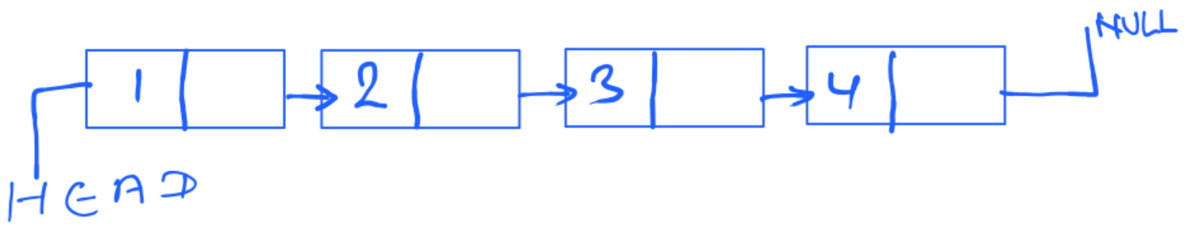


Main Linked List Operation
Insertion: Insert data into the list
Deletion: Removed and returns the specified location from list.

Types of linked list.

- Singly Linked list.
- Doubly Linked list.
- Circular Linked list.

Singly Linked List.
In general, Linked list means singly linked list. This list consists of nodes in which each node has a data and Pointer to Next node. Last node of linked list always points to NULL and Head always points to first Node.



https://www.youtube.com/c/learnbybhanu
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/user/bhanu-pratap-15/

```
package linkedList;

public class ListNode {

	private int data;
	private ListNode next;

	public int getData() {
		return data;
	}
	public void setData(int data) {
		this.data = data;
	}
	public ListNode getNext() {
		return next;
	}
	public void setNext(ListNode next) {
		this.next = next;
	}

}
```

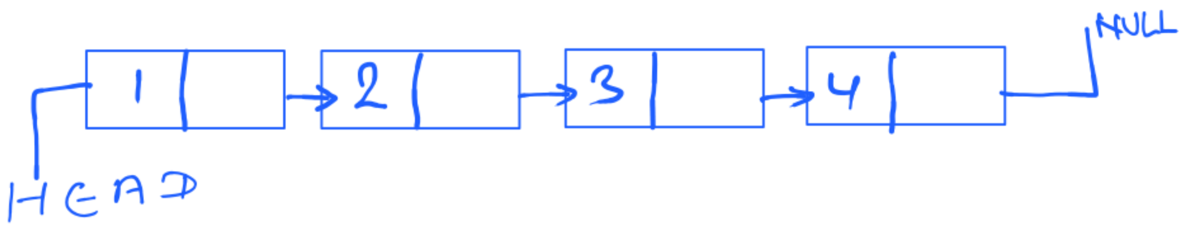
Basic Operation on list

- Traversing the list.
- Insertion element into list.
- Delete element from list.

Traversing the list.

As We know that head points to first Node of the list. To transverse the list we need to do following steps.

- Follow the pointers.
- Display the content of node while traversing.
- Transverse till you encounter node pointing to NULL



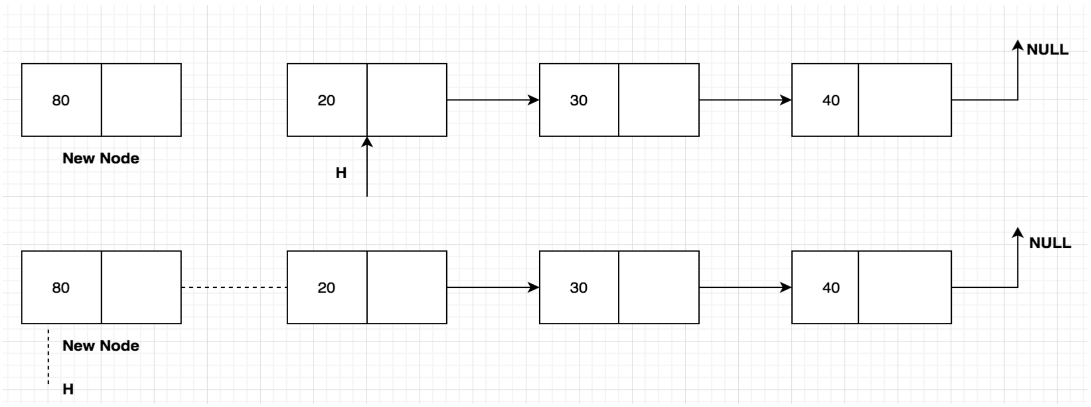
```
package linkedList;
```

```
public class TranverseTheList {

	public int findListLength(ListNode head) {
		int length = 0;
		ListNode currentNode = head;
		while (currentNode != null) {
			currentNode = currentNode.getNext();
			length++;
		}
		return length;
	}
}
```

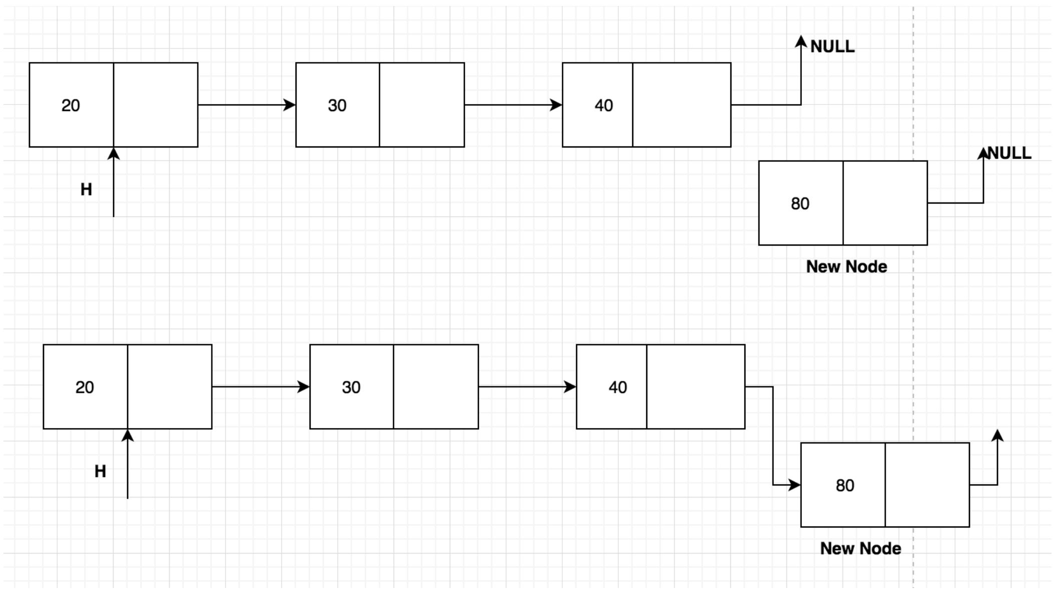
Inserting elements into list.
This can be achieved by three ways.

- Insert Node at beginning.
- Insert Node at end.
- Insert Node in middle.

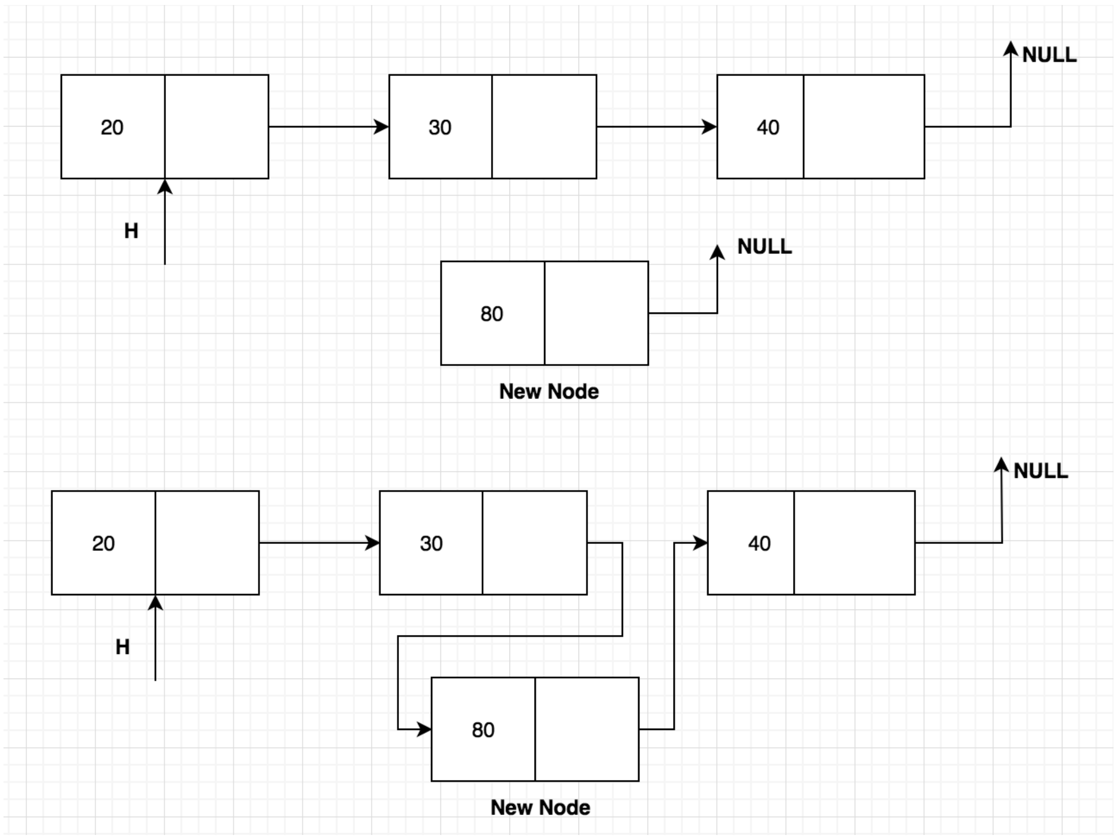
Insert Node at beginning.



Insert Node at End.



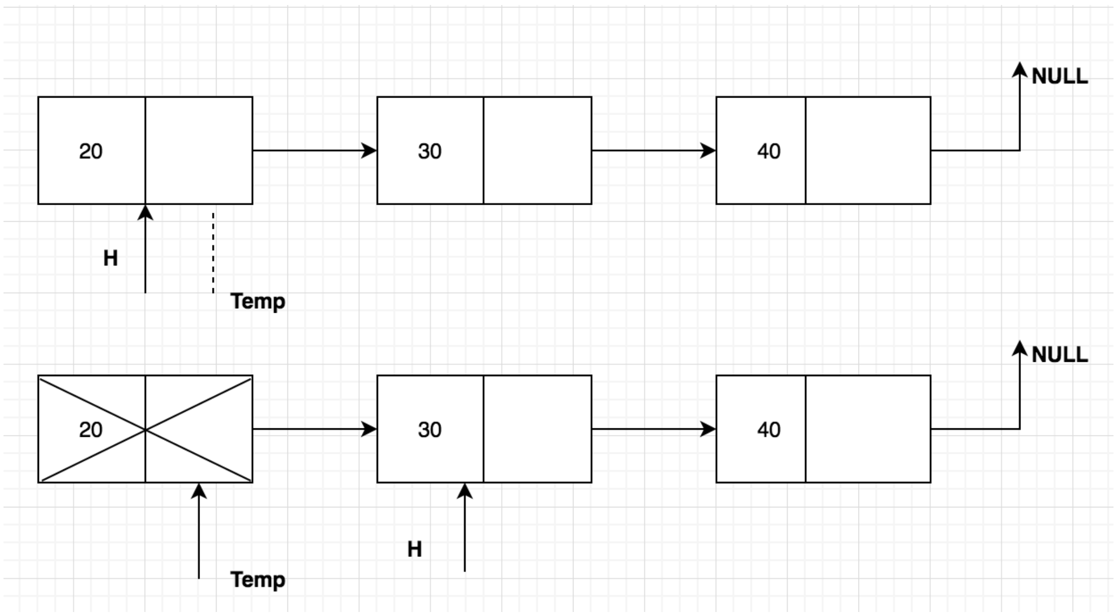
Insert Node in Middle of List



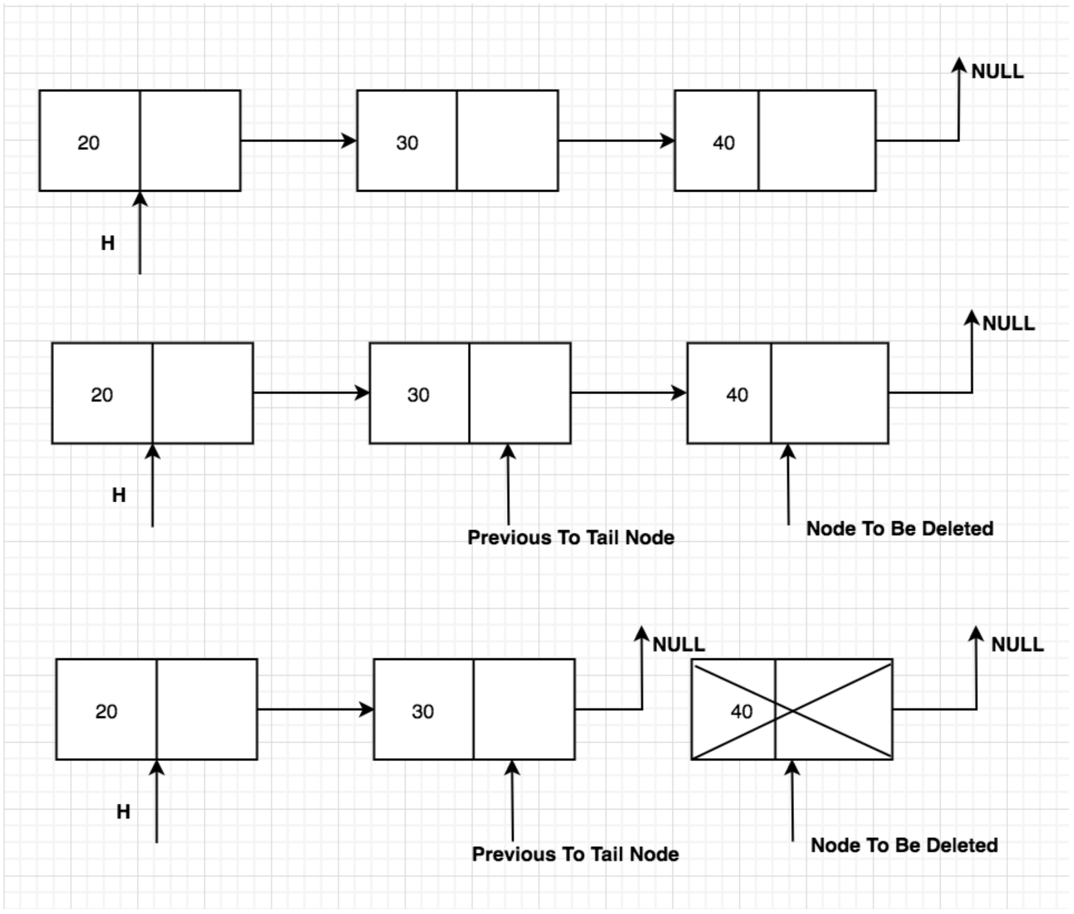
Delete element from list.

- Delete First Node.
- Delete Last Node.
- Delete Middle Node.

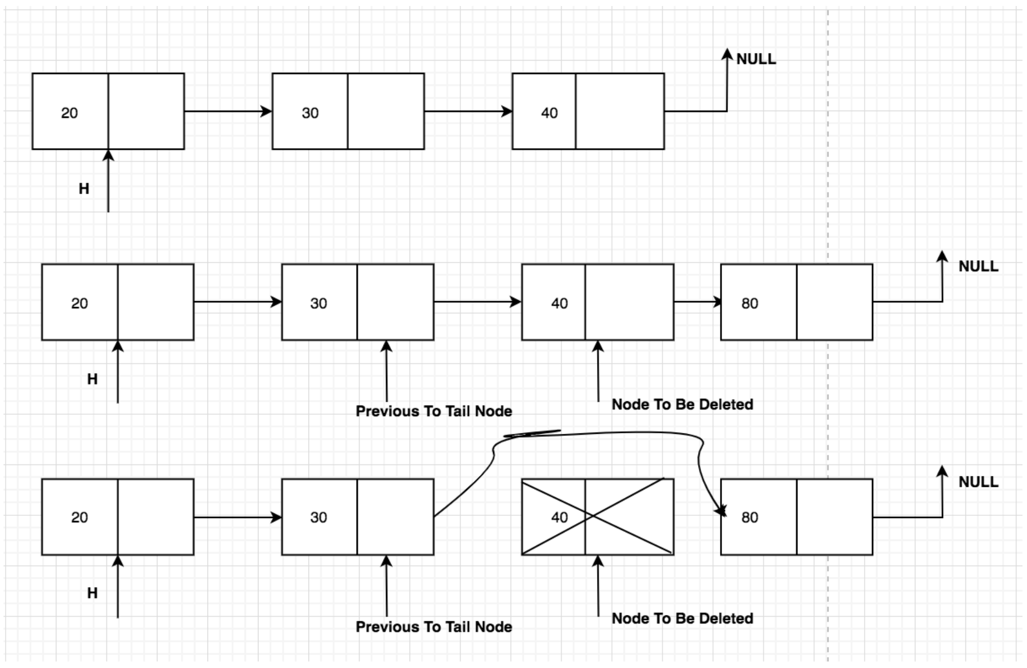
Delete First Node.



Delete Last Node.



Delete Middle Node.



```
package linkedList;

public class ListNode {

	private int data;
	private ListNode next;

	ListNode(){
	}

	/**
	 * Constructor which will take node data as an argument
	 * @param data
	 */
	ListNode(int data){
		this.data = data;
	}

	public int getData() {
		return data;
	}
	public void setData(int data) {
		this.data = data;
	}
	public ListNode getNext() {
		return next;
	}
	public void setNext(ListNode next) {
		this.next = next;
	}


	@Override
	public String toString() {
		return "data--->" + data + " and next--->" + next;
	}

}

package linkedList;

/**
 *
 * @author Bhanu Pratap Singh
 *
 */
public class LinkedList {
	private int length;
	ListNode head;

	LinkedList() {
		length = 0;
	}

	public ListNode getHead() {
		return head;
	}

	// At First Head is null
	public void insertAtBegin(ListNode node) {
		node.setNext(head);
		head = node;
		length++;
	}

	public void insertAtEnd(ListNode newNode) {
		if (head == null) {
			head = newNode;
		} else {
			ListNode start;
			ListNode nextNode;
```

```
                        for (start = head; ((nextNode = start.getNext()) != null); start =
nextNode) {
                        }
                        start.setNext(newNode);
                        // here we don't need to set next for newNode,
                        // since we are adding node at end so newNode
                        // next would be already pointing to NULL
                        length++;
                }
        }

        public void insertAtPosition(int data, int position) {
                if (position > length) {
                        position = length;
                }
                if (head == null) {
                        head = new ListNode(data);
                }
                // here position 0 does not mean we don't have node
                // in Linked list. and we know that first Node in Linked
                // list is Head node. so first temp should point to head
                // and then we need to change the head to point to temp
                else if (position == 0) {
                        ListNode temp = new ListNode(data);
                        temp.setNext(head);
                        head = temp;
                } else {
                        ListNode temp = head;
                        for (int i = 1; i < position; i++) {
                                temp = temp.getNext();
                        }
                        ListNode newNode = new ListNode(data);
                        // first set NewNode next pointer to temp next pointer
                        newNode.setNext(temp.getNext());
                        temp.setNext(newNode);
                        length++;
                }
        }

        public ListNode removeNodeFromBeging() {
                ListNode temp = head;
                if (temp != null) {
                        head = temp.getNext();
                        temp.setNext(null);
                }
```

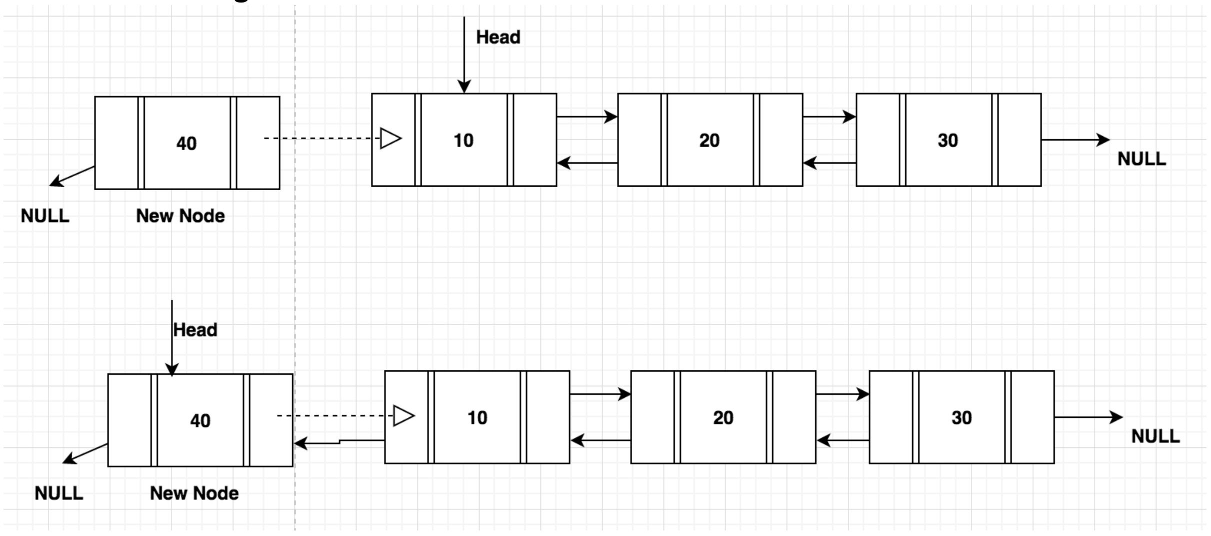
```
		return temp;
	}

	public ListNode removeFromEnd() {
		if (head == null) {
			return null;
		}
		ListNode temp = head;
		if (temp.getNext() == null) {
			return head;
		}
		ListNode node1 = null;
		ListNode node2 = null;
		while ((node1 = temp.getNext()) != null) {
			node2 = temp;
			temp = node1;
		}
		node2.setNext(null);
		return temp;
	}

	public ListNode removeMatchedNode(ListNode node) {
		if (head == null) {
			return null;
		}
		if (node.equals(head)) {
			ListNode temp = head;
			head = head.getNext();
			return temp;
		}
		ListNode temp = head;
		ListNode temp1;
		while ((temp1 = temp.getNext()) != null) {
			if (node.equals(temp1)) {
				temp.setNext(temp1.getNext());
				return temp1;
			}
			temp = temp1;
		}
		return null;

	}

	public ListNode removePosition(int position) {
		if (position > length) {
			position = length;
		}
		if (position == 0) {
			ListNode temp = head;
			head = head.getNext();
			return temp;
		} else {
			ListNode a = head, b = head;
			for (int i = 1; i < position; i++) {
				a = b;
				b = b.getNext();
			}
			a.setNext(a.getNext().getNext());
			return a;
		}
	}

	@Override
	public String toString() {
		return "LinkedList [head=" + head + "]";
	}

	public static void main(String[] args) {
		LinkedList linkedList = new LinkedList();
		linkedList.insertAtBegin(new ListNode(5));
		linkedList.insertAtBegin(new ListNode(3));
		linkedList.insertAtBegin(new ListNode(2));
		ListNode node = new ListNode(50);
		linkedList.insertAtBegin(node);
		linkedList.insertAtPosition(200, 3);
		linkedList.insertAtEnd(new ListNode(60));
		linkedList.insertAtEnd(new ListNode(70));
		System.out.println(linkedList);
		// linkedList.removeMatchedNode(node);
		// linkedList.removeNodeFromBeging();
		// linkedList.removeFromEnd();
		// linkedList.removeFromEnd();
		// linkedList.removeFromEnd();
		System.out.println(linkedList);
		linkedList.removePosition(4);
		System.out.println(linkedList);
	}

}
```

Doubly LinkedList
In case of doubly LinkedList we can navigate in both directions. Whereas in case of singly Linked list we can navigate only in forward direction. We cannot remove any node in singly linked list unless we have the pointer to its predecessors. But in doubly linked list we can delete the node even if we don't have previous node address (since in doubly linked list each node has left pointer to previous node and we can move backward).

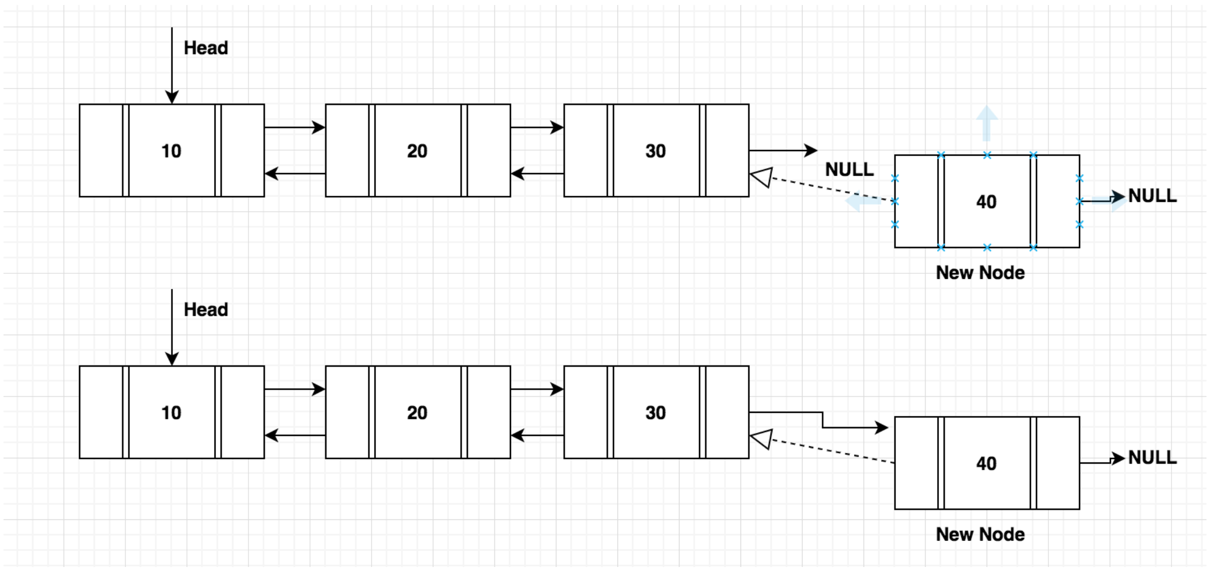
Doubly Linked list Insertion.

- Insert Node at begin.
- Insert Node at end.
- Insert Node in Middle.

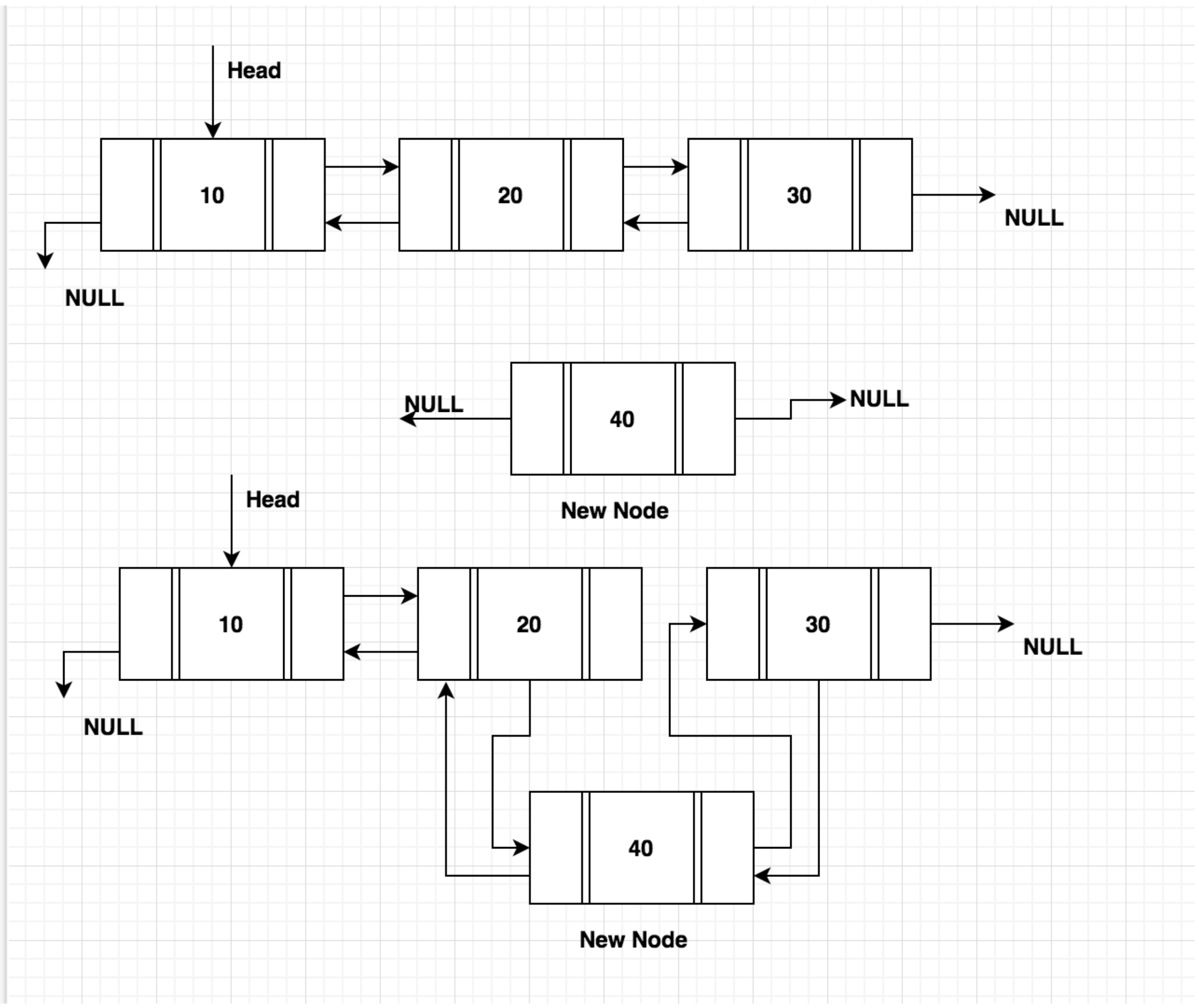
Insert Node at begin.



Insert Node at end.



Insert Node in Middle.



https://www.youtube.com/c/learnbybhanu
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/user/bhanu-pratap-15/

```
package linkedList;

/**
 *
 * @author Bhanu Pratap Singh
 *
 */
public class DoublyLinkedList {

	private DLLNode head;
	private DLLNode tail;
	private int length;

	private DoublyLinkedList(){
		head = new DLLNode(Integer.MAX_VALUE, null, null);
		tail = new DLLNode(Integer.MAX_VALUE, head, null);
		head.setNext(tail);
		length++;
	}

	public void insertAtBeging(int data){
		DLLNode newNode = new DLLNode(data, head, head.getNext());
		newNode.getNext().setPrev(newNode);
		head.setNext(newNode);
		length++;
	}

	public void insertAtTail(int data){
		DLLNode newNode = new DLLNode(data, tail.getPrev(), tail);
		newNode.getPrev().setNext(newNode);
		tail.setPrev(newNode);
		length++;
	}

	public void insertAtPosition(int data, int position){
		if(position > length){
			position = 0;
		}
		if(position == 0){
			insertAtBeging(data);
		}
		DLLNode temp = head;
		for (int i = 1; i < position; i++) {
			temp = temp.getNext();
		}
```

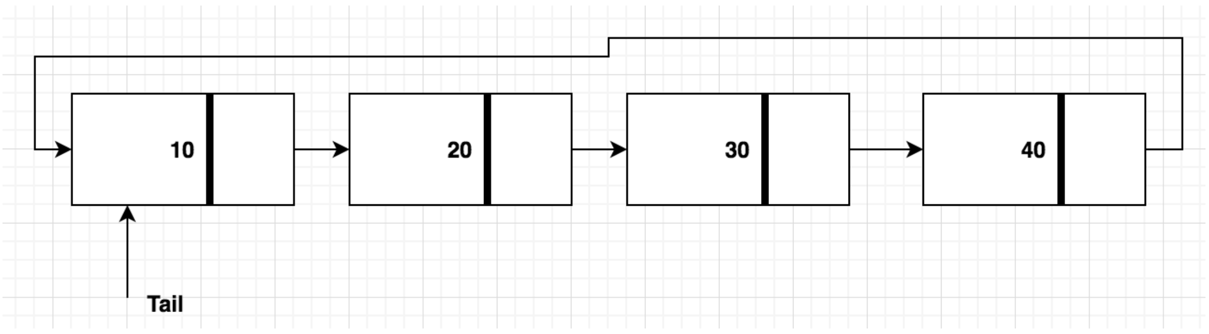
```
        DLLNode newNode = new DLLNode(data);
        newNode.setNext(temp.getNext().getNext());
        temp.getNext().getNext().setPrev(newNode);
        temp.setNext(newNode);
        newNode.setPrev(temp);
        length++;

    }

    public int getPosition(int data){
        DLLNode temp = head;
        int position = 0;
        while(temp!=null){
            if(data == temp.getData()){
                return position;
            }
            position++;
            temp = temp.getNext();
        }
        return Integer.MAX_VALUE;

    }

    public int removeFromBeging(){
        if(length == 0){
            return Integer.MAX_VALUE;
        }
        DLLNode temp = head.getNext();
        head.setNext(temp.getNext());
        temp.getNext().setPrev(head);
        length--;
        return temp.getData();
    }

    public int removeFromEnd(){
        if(length == 0){
            return Integer.MAX_VALUE;
        }
        DLLNode temp = tail.getPrev();
        tail.setPrev(temp.getPrev());
        temp.getPrev().setNext(tail);
        length--;
        return temp.getData();
    }
```

```
public int removePosition(int position){
        if(position > length){
                position = length;
        }
        if(position == 0){
                removeFromBeging();
        }
        DLLNode temp = head;
        for (int i = 1; i < position; i++) {
                temp = temp.getNext();
        }
        temp.getNext().getNext().setPrev(temp);
        temp.setNext(temp.getNext().getNext());
        length--;
        return temp.getData();
}

@Override
public String toString() {
        String result = "[]";
        if(length == 0){
                return null;
        }
        result = "["+head.getData();
        DLLNode temp = head.getNext();
        while(temp!=null){
                result = result+","+temp.getData();
                temp = temp.getNext();
        }
        return result+"]";
}

public static void main(String[] args) {
        DoublyLinkedList doublyLinkedList = new DoublyLinkedList();
        doublyLinkedList.insertAtBeging(4);
        doublyLinkedList.insertAtBeging(5);
        doublyLinkedList.insertAtBeging(6);
        doublyLinkedList.insertAtTail(8);
        doublyLinkedList.insertAtPosition(100, 2);
        doublyLinkedList.insertAtPosition(1000, 0);
        System.out.println(doublyLinkedList);
        System.out.println(doublyLinkedList.removeFromBeging());
        System.out.println(doublyLinkedList);
        System.out.println(doublyLinkedList.removeFromEnd());
        System.out.println(doublyLinkedList);
```

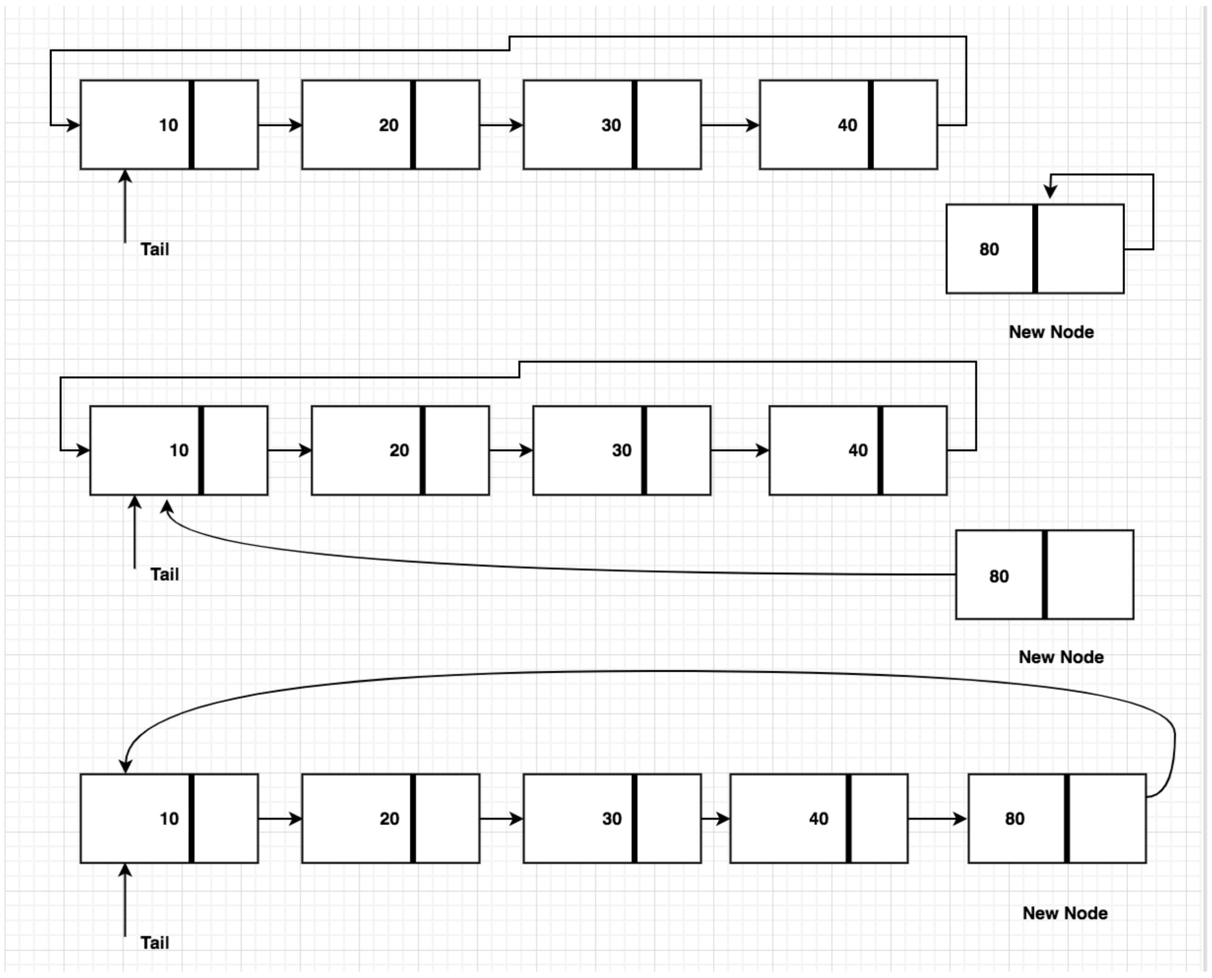
```
                System.out.println(doublyLinkedList.removePosition(3));
                System.out.println(doublyLinkedList);
        }


}
```

Circular linked list

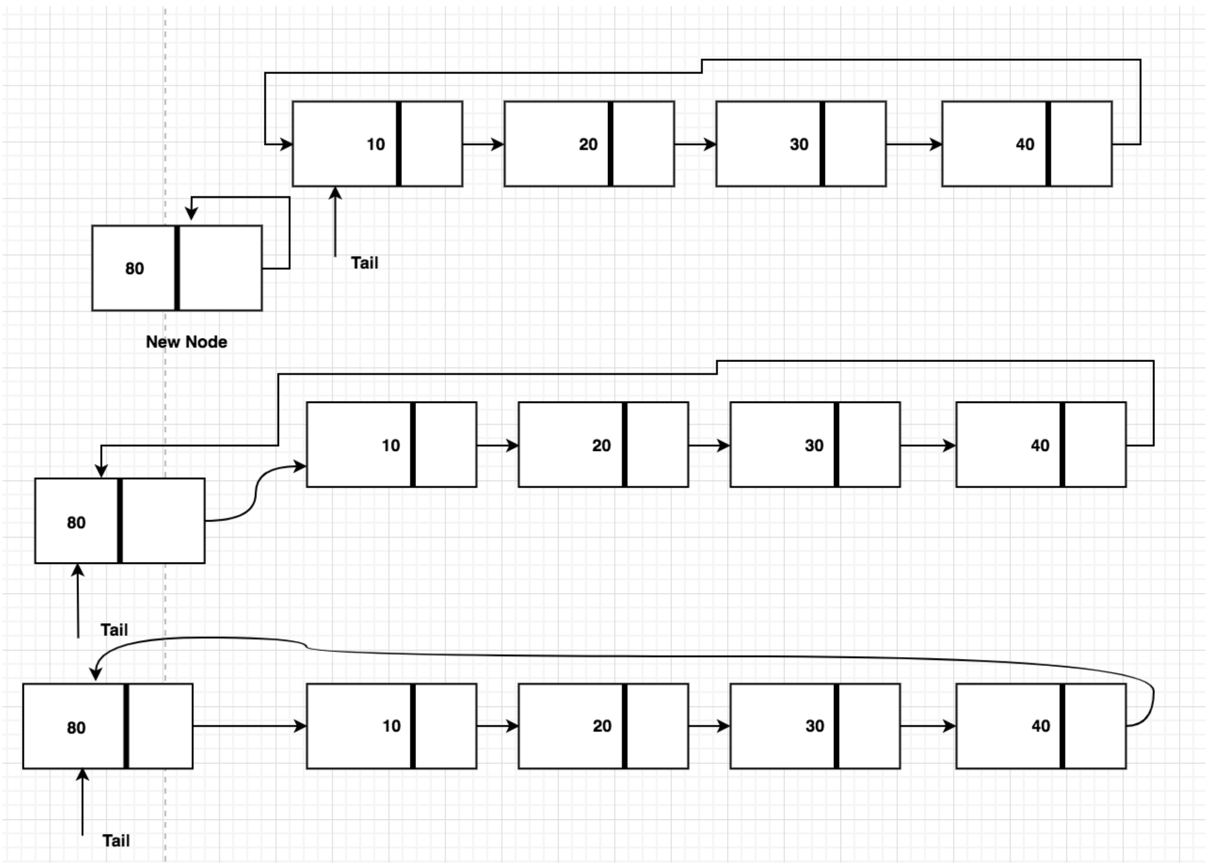
In single and doubly linked list end is indicated by NULL value. Whereas in case of circular linked list we don't have end with NULL value. While traversing the circular linked list, we should to be careful, otherwise we will end up with infinite loop.



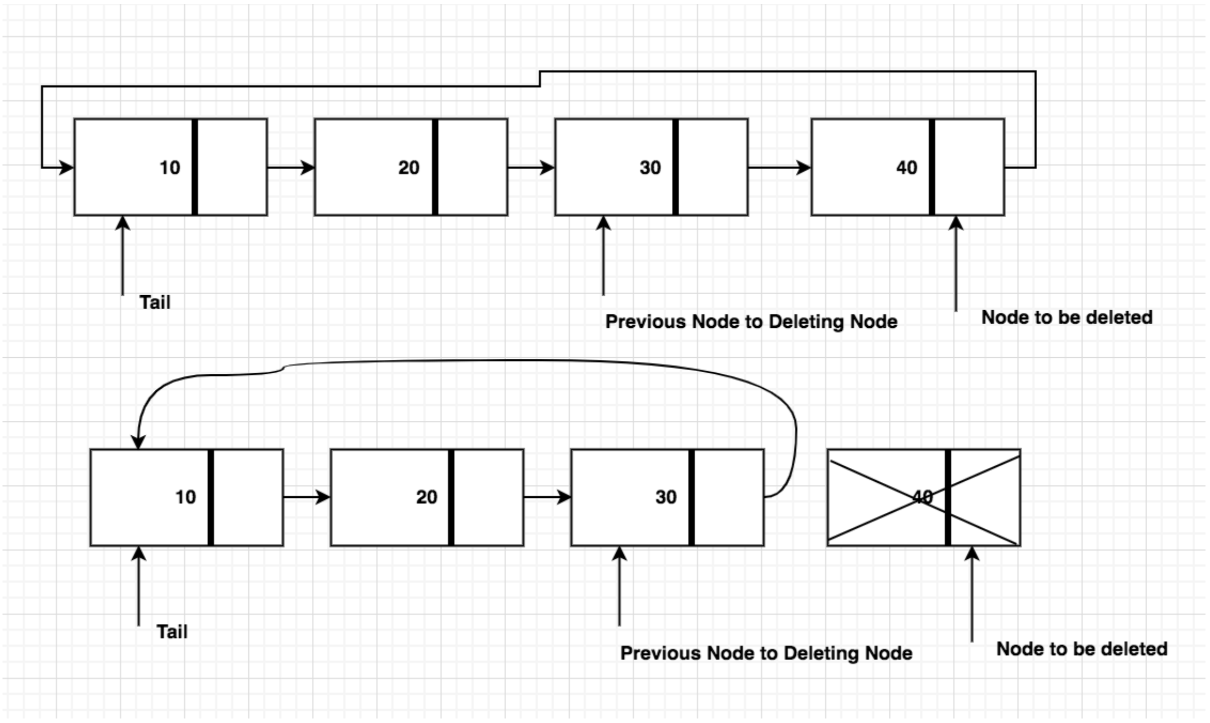
Insert The data at End of Circular linked list



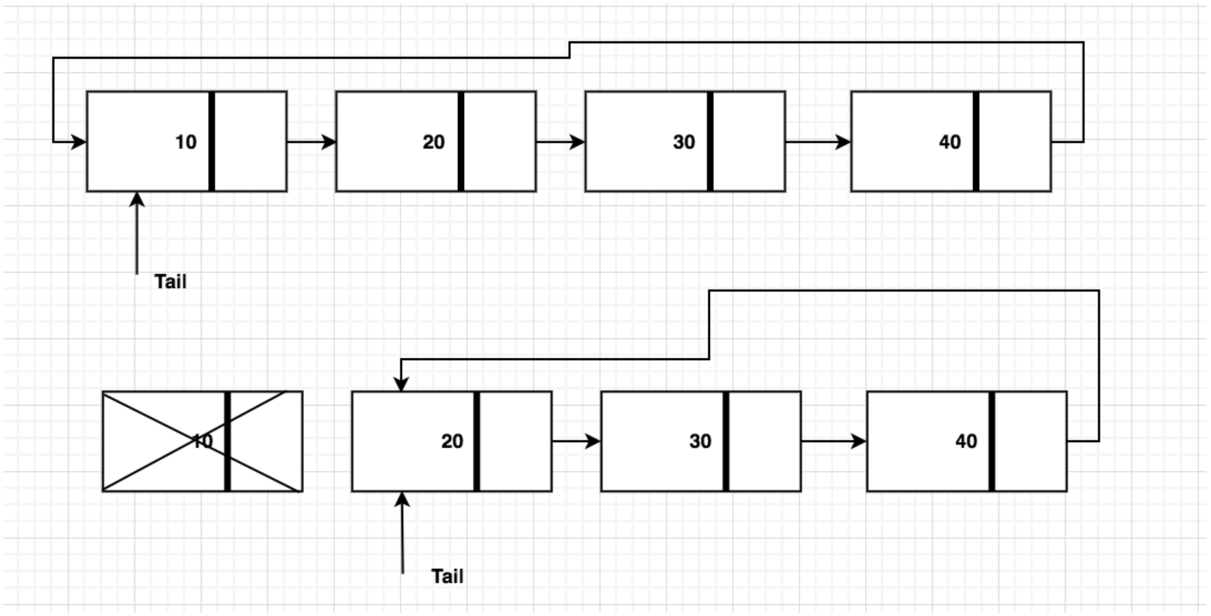
Insert the Data at beginning of circular linked list



Delete the last Node of Circular Linked list.


10
20
30
40
Tail
Previous Node to Deleting Node
Node to be deleted
10
20
30
40
Tail
Previous Node to Deleting Node
Node to be deleted

Delete the first Node of Circular Linked list.



```
package linkedList;

public class CircularLinkedList {

	private CLLNode tail;
	private int length;

	CircularLinkedList() {
		tail = null;
		length = 0;
	}

	public CLLNode getTail() {
		return tail;
	}

	public int getLength() {
		return length;
	}

	public void addHead(int data) {
		CLLNode newNode = new CLLNode(data);
		if (tail == null) {
			// here don't do tail.setNext(newNode); you will get NPE
			// since tail is null in beginning
			tail = newNode;
```

```
            newNode.setNext(tail);
        } else {
            // first change newnode point to tail
            newNode.setNext(tail.getNext());
            // then tail should point ot newnode
            tail.setNext(newNode);

        }
        length++;
    }

    public void addTail(int data) {
        addHead(data);
        tail = tail.getNext();
    }

    public int returnTailData() {
        return tail.getData();
    }

    public int returnHeadData() {
        return tail.getNext().getData();
    }

    public int removeFromhead() {
        // Just after tail node we have head node
        CLLNode temp = tail.getNext();
        if (tail == tail.getNext()) {
            tail = null;
        } else {
            // we need to sift tail node to point ahead of head node
            tail.setNext(temp.getNext());
            // set temp point to null
            temp.setNext(null);
        }
        return temp.getData();
    }

    public int removeFromEnd() {
        CLLNode temp = tail;
        while (temp.getNext() != tail) {
            temp = temp.getNext();
        }
        // this condition will work when there is only one node in
        // circular linked list
```

```
		if (temp == tail) {
			tail = null;
		} else {
			temp.setNext(tail.getNext());
			tail = temp;
		}
		length--;
		return temp.getData();

	}

	public int removeMatched(int data) {
		if (tail == null) {
			return Integer.MAX_VALUE;
		}
		// store temp with tail.next
		CLLNode temp = tail.getNext();
		CLLNode preTemp = tail;
		// check data with temp data
		for (int i = 0; i < length && (temp.getData() != data); i++) {
			preTemp = temp;
			temp = temp.getNext();
		}
		if(temp.getData() == data){
			if(tail == tail.getNext()){
				tail = null;
			}
			else{
				// if temp points to tail then we need
				// to remove current tail and make previous
				// to tail node as tail
				if(temp == tail){
					tail = preTemp;
				}
				preTemp.setNext(temp.getNext());
			}
			temp.setNext(null);
			length--;
			return temp.getData();
		}
		preTemp.setNext(temp.getNext());
		preTemp.setNext(null);
		return preTemp.getData();
	}
	@Override
	public String toString() {
		String result = "[";
		if (tail == null) {
			result = result + "]";
		}
		CLLNode temp = tail;
		result = result + temp.getData();
		temp = temp.getNext();
		while (temp != tail && temp != null) {
			result = result + "," + temp.getData();
			temp = temp.getNext();
		}
		return result + "]";

	}

	public static void main(String[] args) {
		CircularLinkedList circularLinkedList = new CircularLinkedList();
		circularLinkedList.addHead(1);
		circularLinkedList.addHead(2);
		circularLinkedList.addHead(3);
		circularLinkedList.addHead(4);
		circularLinkedList.addHead(5);
		circularLinkedList.addHead(6);
		System.out.println(circularLinkedList);
		System.out.println(circularLinkedList.removeMatched(1));
		System.out.println(circularLinkedList);
	}
}
```

Algorithm Example

Program for n'th node from the end of a Linked List ...

```
package linkedList;

/**
 *
 * @author Bhanu Pratap Singh
 *
 */
public class FindNthNodeFromEndOfLinkedList {
        int length;

        public int findNthNodeFromLast(ListNode head, int nthNode){
                ListNode temp = head;
                // get total number of length
                while(temp!=null){
                        length++;
                        temp = temp.getNext();
                }

                // length-nthNode plus 1 will be equal to
                // nth node from last
                // e.g if linked list has 5 node and when we want 2 node
                // from last means (5-2)+1 = will be 4th node from begin.
                int nodeFromBeging = (length-nthNode)+1;
                ListNode node = head;
                for (int i = 0; i <= nodeFromBeging; i++) {
                        node = node.getNext();
                }
                return node.getData();
        }

        public static void main(String[] args) {
                LinkedList linkedList = new LinkedList();
                linkedList.insertAtBegin(new ListNode(5));
                linkedList.insertAtBegin(new ListNode(3));
                linkedList.insertAtBegin(new ListNode(2));
                linkedList.insertAtBegin(new ListNode(20));
                linkedList.insertAtBegin(new ListNode(200));
                System.out.println(linkedList);
                FindNthNodeFromEndOfLinkedList findNthNodeFromEndOfLinkedList = new
FindNthNodeFromEndOfLinkedList();
                int data =
findNthNodeFromEndOfLinkedList.findNthNodeFromLast(linkedList.getHead(), 4);
                System.out.println(data);
        }
}
```

Can we improve the complexity of above program?

Answer: Yes, by using hash Table. By the time we transverse the complete list we can find length of linked list. then we can get (length-NthNode)+1 key from hash Table