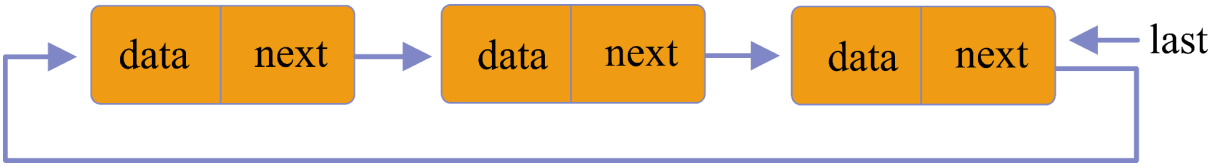
Circular Linked List

What is Circular Linked List?

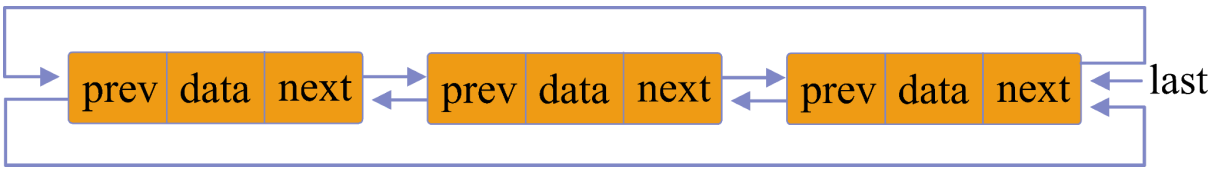
The circular linked list is a linked list in which all nodes form a circle. The initial and last nodes in a circular linked list are linked to each other, forming a circle. There is no NULL at the end.

There are two types of Circular Linked List.

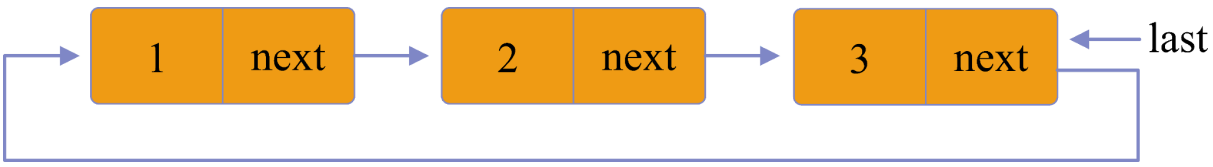
- **Singly Circular Linked List**: The last node of a circular Singly linked list contains a pointer to the beginning node of the list. We go around the circular singly linked list till we get back to where we started. There is no beginning or finish to the circular singly linked list. There is no null value in the next section of any of the nodes.



- **Doubly Linked List**: Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous. Next, pointer and the last node points to the first node by the next pointer and the first node points to the last node by the previous pointer.



Representation of Circular Linked List




```
public class Node{
```

```
    int data;
    Node next;
    public Node(int data) {
       this.data = data;
    }
}
```

Operations on the circular linked list:

We can do some operations on the circular linked list similar to the singly linked list which are:

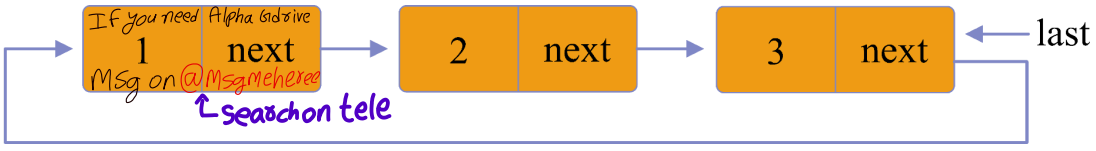
1. Insertion
2. Deletion

Insertion into a Circular Linked List

We can introduce elements into a circular linked list at three different positions:

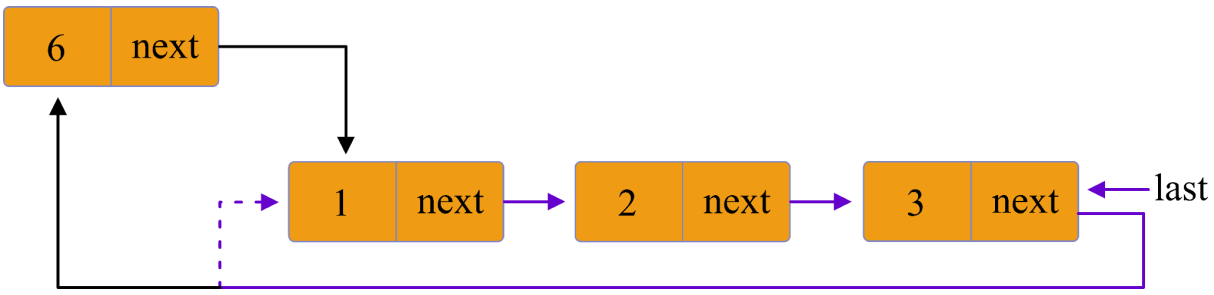
- Insertion in the beginning
- Insertion between nodes
- Insertion toward the end

Assume we have a circular linked list with the elements 1, 2, and 3.



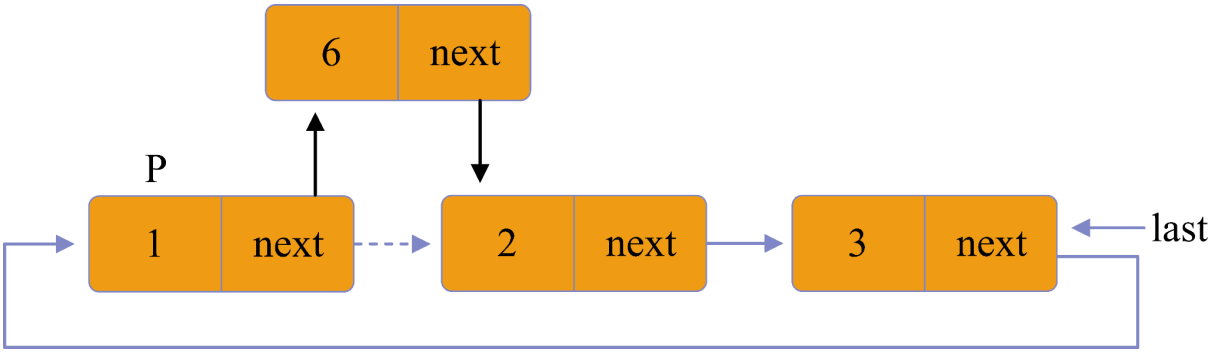
1. Insertion at the Start

=> Store the address of the current first node in the newNode (i.e. directing the newNode to the current first node) point the last node to the newNode (i.e making newNode as head)



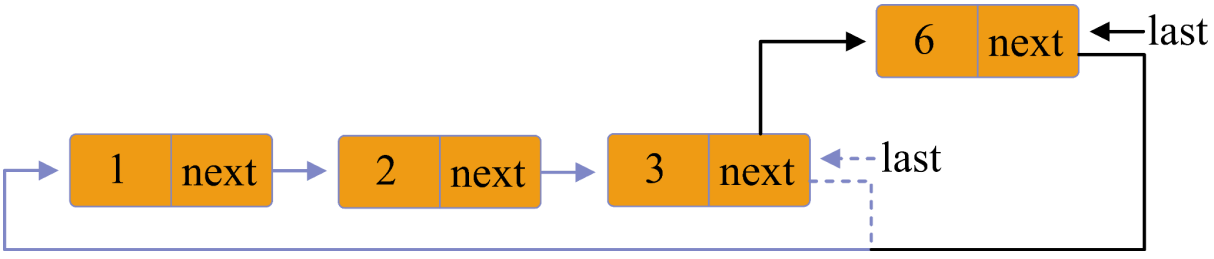
2. **Insertion between two nodes**

=> Insert newNode after the first node.travel to the supplied node (let this node be p) direct the next of newNode to the node next to p put the address of newNode at next of p.



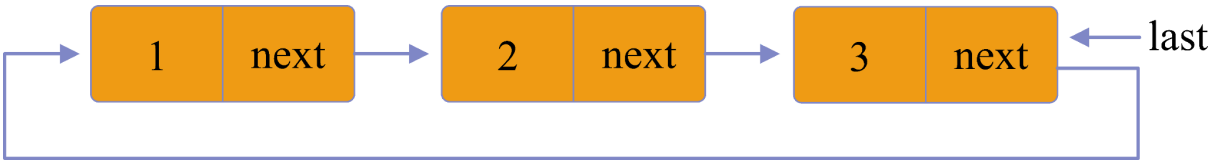
3. **Insert at the End**

=> Insert the address of the head node to the next of newNode (making newNode the last node) make newNode the final node by pointing the current last node to newNode.



Deletion in a Circular Linked List

Let's say we have a singly circular linked list.

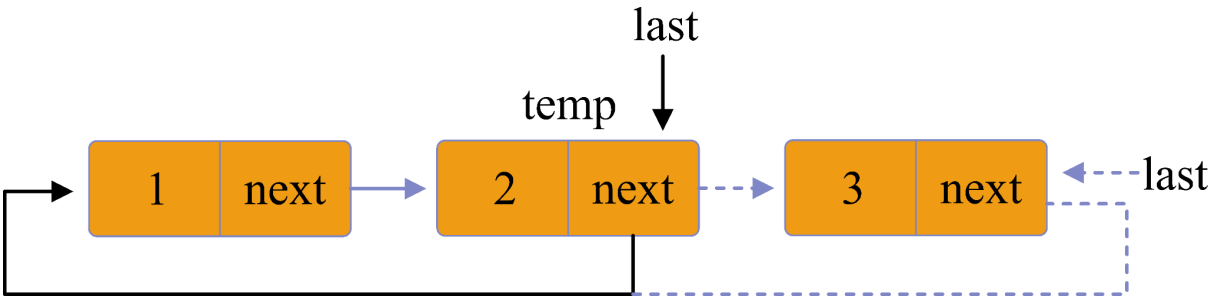


1. **Delete Single Node**

=> If the node to be destroyed is the sole node, free the node's memory and store NULL in the final node.

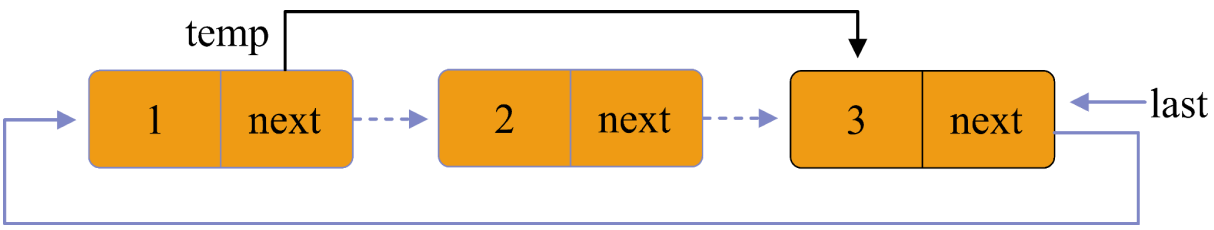
2. **Delete Last Node**

=> If the last node is to be eliminated, locate the node preceding the last node (let it be temp), save the address of the node following the last node in temp free the memory of the last make temp as the last node



3. **Delete Any Other Node**

=> If any additional nodes are to be removed, go to the node to be deleted (in this case, node 2), let the node before node 2 be temp store the address of the node next to 2 in temp free the memory of 2



Implementing Insertion, Deletion and Printing in Circular Linked List

```
class Solution {
  static class Node {
    int data;
    Node next;
  };

  static Node addToEmpty(Node last, int data) {
    if (last != null)
      return last;
    Node newNode = new Node();
    newNode.data = data;
    last = newNode;
    newNode.next = last;
    return last;
  }
  static Node addFront(Node last, int data) {
    if (last == null)
      return addToEmpty(last, data);
    Node newNode = new Node();
    newNode.data = data;
    newNode.next = last.next;
    last.next = newNode;
    return last;
  }

  static Node addEnd(Node last, int data) {
    if (last == null)
      return addToEmpty(last, data);
    Node newNode = new Node();
    newNode.data = data;
    newNode.next = last.next;
    last.next = newNode;
    last = newNode;
    return last;
```

```
}

static Node addAfter(Node last, int data, int item) {
  if (last == null)
    return null;

  Node newNode, p;
  p = last.next;
  do {
    if (p.data == item) {
      newNode = new Node();
      newNode.data = data;
      newNode.next = p.next;
      p.next = newNode;
      if (p == last)
        last = newNode;
      return last;
    }
    p = p.next;
  } while (p != last.next);
  System.out.println(item + "The given node is not present in the list");
  return last;
}

static Node deleteNode(Node last, int key) {
  if (last == null)
    return null;
  if (last.data == key && last.next == last) {
    last = null;
    return last;
  }

  Node temp = last, d = new Node();
  if (last.data == key) {
    while (temp.next != last) {
      temp = temp.next;
    }
```

```
    temp.next = last.next;
    last = temp.next;
  }
  while (temp.next != last && temp.next.data != key) {
    temp = temp.next;
  }
  if (temp.next.data == key) {
    d = temp.next;
    temp.next = d.next;
  }
  return last;
}

static void traverse(Node last) {
  Node p;
  if (last == null) {
    System.out.println("List is empty.");
    return;
  }
  p = last.next;
  do {
    System.out.print(p.data + " ");
    p = p.next;
  }
  while (p != last.next);
}

public static void main(String[] args) {
  Node last = null;
  last = addToEmpty(last, 6);
  last = addEnd(last, 8);
  last = addFront(last, 2);
  last = addAfter(last, 10, 2);
  traverse(last);
  deleteNode(last, 8);
  traverse(last);
}
```

```
}
```

Thanks for reading this article till the end.