❖ Circular Linked List

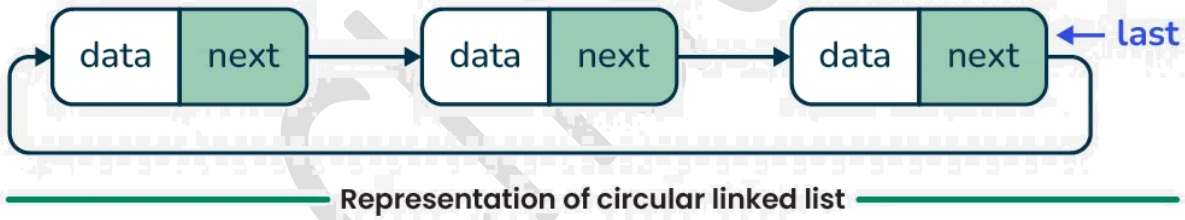
→ A circular linked list is a data structure where the last node connects back to the first, forming a loop. This structure allows for continuous traversal without any interruptions. Circular linked lists are especially helpful for tasks like scheduling and managing playlists, allowing for smooth navigation.

❖ What is a Circular Linked List?

→ A circular linked list is a special type of linked list where all the nodes are connected to form a circle. Unlike a regular linked list, which ends with a node pointing to NULL, the last node in a circular linked list points back to the first node. This means that you can keep traversing the list without ever reaching a NULL value.

❖ Circular Singly Linked List

→ In Circular Singly Linked List, each node has just one pointer called the "next" pointer. The next pointer of the last node points back to the first node and this results in forming a circle. In this type of Linked list, we can only move through the list in one direction.



Representation of circular linked list

❖ Representation of a Circular Singly Linked List

→ Let's take a look on the structure of a circular linked list.

Node structure of circular linked list

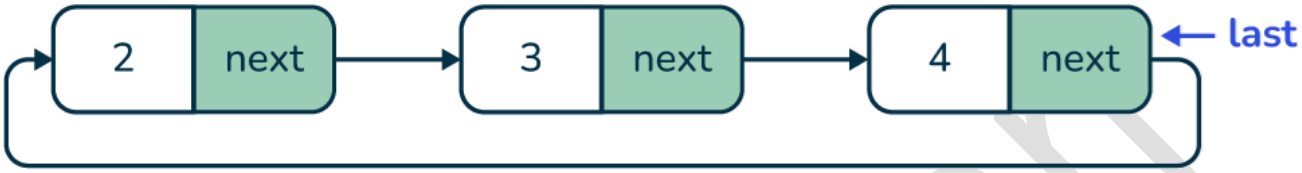
❖ Create/Declare a Node of Circular Linked List

```
class Node {
   int data;
   Node next;

Node(int x)
   {
```

```
        data = x;
    }
}
```

❖ Example of Creating a Circular Linked List

Here's an example of creating a circular linked list with three nodes (2, 3, 4):




```
// Initilize and allocate memory for nodes

Node first = new Node(2);

Node second = new Node(3);

Node last = new Node(4);


// Connect nodes

first.next = second;

second.next = last;

last.next = first;
```

→ In the above code, we have created three nodes first, second, and last having values 2, 3, and 4 respectively.

- After creating three nodes, we have connected these node in a series.
- Connect the first node "first" to "second" node by storing the address of "second" node into first's next
- Connect the second node "second" to "third" node by storing the address of "third" node into second's next
- After connecting all the nodes, we reach the key characteristic of a circular linked list: linking the last node back to the first node. Therefore, we store the address of the "first" node in the "last" node.

Operations on the Circular Linked list

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

1. Insertion

- Insertion at the empty list
- Insertion at the beginning
- Insertion at the end
- Insertion at the given position

2. Deletion

- Delete the first node
- Delete the last node
- Delete the node from any position

3. Searching

```
class node
{
   int data;
   node next;
   node(int x)
   {
      data=x;
   }
}
class CLL
{ node head;
   void insertfirst(int x)
   {
      node n = new node(x);
      if(head==null)
      {
         head=n;
         n.next=n;
      }
      else
      {
         n.next=head;
         node temp=head;
         while(temp.next!=head)
```

```
        {
            temp=temp.next;
        }
        temp.next=n;
        head=n;
    }
}

void display()
{
    if(head==null)
    {
        System.out.println("Empty");
    }
    else {
        node temp=head;
        while(temp.next!=head)
        {
            System.out.print(temp.data+"-->");
            temp=temp.next;
        }
        System.out.println(temp.data);
    }
}

void insertlast(int x)
{
    node n = new node(x);
    if(head==null)
    {
        head=n;
        n.next=n;
    }
    else
    {
        node temp=head;
```

```
        while(temp.next!=head)
        {
           temp=temp.next;
        }
        temp.next=n;
        n.next=head;
    }
}

void deletefirst()
{
    if(head==null)
    {
        System.out.println("underflow");
    }
    else if(head.next==head)
    {
         head.next=null;
        head=null;
    }
    else
    {
        node del=head;
        node temp=head;
        while(temp.next!=head)
        {
           temp=temp.next;
        }
        temp.next=head.next;
        head=head.next;
        del.next=null;
        del=null;
    }
}
```

```
void deletelast()
{
   if(head==null)
   {
      System.out.println("underflow");
   }
   else if(head.next==head)
   {
       head.next=null;
        head=null;
   }
   else
   {
      node temp=head;
      while(temp.next.next!=head)
      {
         temp=temp.next;
      }
      node del=temp.next;
      temp.next=head;
      del.next=null;
     del=null;
   }
}

void inserbeforevalue(int x,int data)
{
   node  n = new node(data);
   int flag=0;
   if(head==null)
   {
      System.out.println("CLL is empty");
   }
   else
```

```
{
   node temp1 = head;
   do {
      if(temp1.data==x)
      {
         flag=1;
      }
      temp1=temp1.next;
   }while(temp1!=head);
   if(flag==1)
   {
      if(head.data==x)
      {
         n.next=head;
         node temp2=head;
         while(temp2.next!=head){
            temp2=temp2.next;
         }
         temp2.next=n;
         head=n;
      }
      else
      {
         node temp3=head;
         while(temp3.next.data!=x)
         {
            temp3=temp3.next;
         }
         n.next=temp3.next;
         temp3.next=n;
      }
   }
   else
   {
      System.out.println("Value not found");
   }
```

```
    }
}


void insertaftervalue(int x,int data)
{
    node  n = new node(data);
    int flag=0;
    if(head==null)
    {
        System.out.println("CLL is empty");
    }
    else
    {
        node temp1 = head;
        do {
            if(temp1.data==x)
            {
                flag=1;
            }
            temp1=temp1.next;
        }while(temp1!=head);
        if(flag==1)
        {
                node temp3=head;
                while(temp3.data!=x)
                {
                    temp3=temp3.next;
                }
                n.next=temp3.next;
                temp3.next=n;

        }
        else
        {
            System.out.println("Value not found");
```

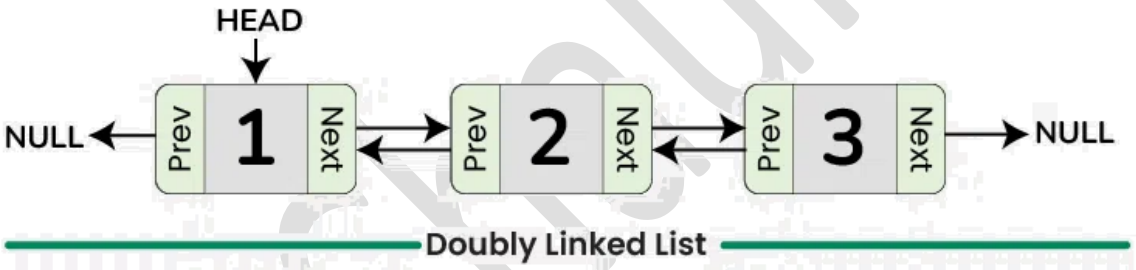
```
        }
    }

}
void deletevalue(int x)
{
    int flag=0;
    if(head==null)
    {
        System.out.println("CLL is empty");
    }
    else
    {
        node temp1 = head;
        do {
            if(temp1.data==x)
            {
                flag=1;
            }
            temp1=temp1.next;
        }while(temp1!=head);
        if(flag==1)
        {
            if(head.data==x && head.next==head){
                head.next=null;
                head=null;
            }
            else if(head.data==x && head.next!=head)
            {
                node del=head;
                node temp2=head;
                while(temp2.next!=head){
                    temp2=temp2.next;
                }
                temp2.next=head.next;
                head=head.next;
```

```
            del.next=null;
            del=null;
         }
         else {
            node temp3=head;
            while(temp3.next.data!=x){
               temp3=temp3.next;
            }
            node del=temp3.next;
            temp3.next=temp3.next.next;
            del.next=null;
            del=null;
         }
      }
      else
      {
         System.out.println("Value not found");
      }
   }
}
class A
{
   public static void main(String[] args) {
      CLL c = new CLL();
      c.insertfirst(1);
      c.insertfirst(2);
      c.insertlast(3);
      c.insertlast(4);
      c.display();
      c.deletefirst();
      c.display();
      c.deletelast();
      c.display();
      c.inserbeforevalue(3,5);
      c.inserbeforevalue(300,5);
```

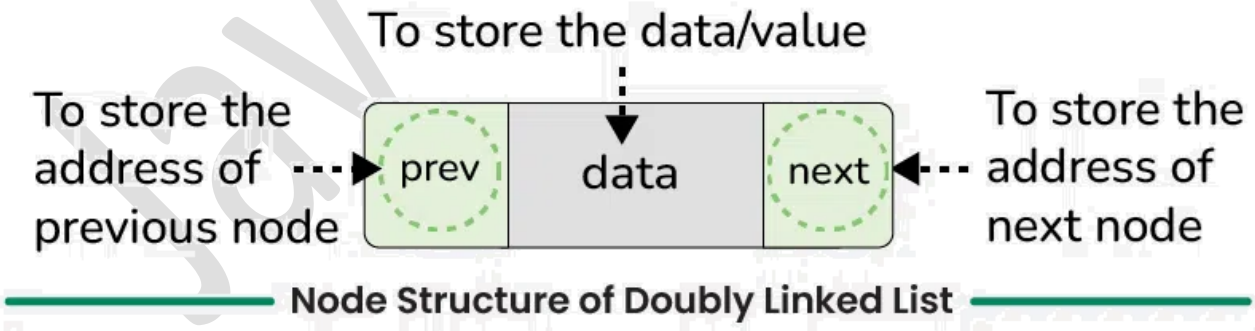
```
        c.display();
        c.insertaftervalue(3,4000000);
        c.display();
    }
}
```

❖ Doubly Linked List

→A doubly linked list is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.



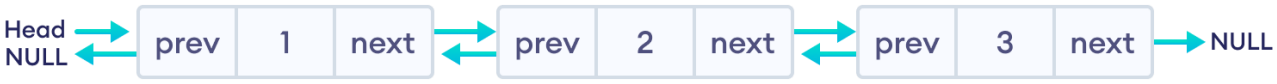
❖ Representation of Doubly Linked List in Data Structure



❖ Insertion on a Doubly Linked List

- **Insertion at the beginning**

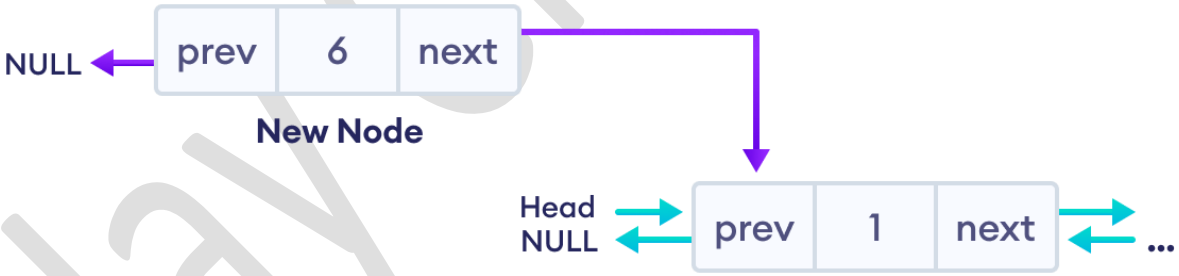
→Suppose we have a double-linked list with elements 1, 2, and 3.



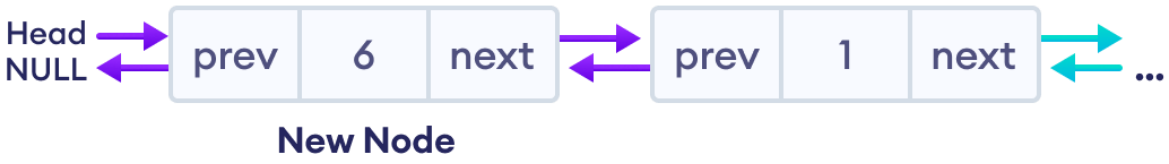
Let's add a node with value 6 at the beginning of the doubly linked list we made above.

1. Create a new node
   → allocate memory for newNode
   → assign the data to newNode.

2. Set prev and next pointers of new node
   → point next of newNode to the first node of the doubly linked list
   → point prev to null



3. Make new node as head node
   → Point prev of the first node to newNode (now the previous head is the second node)
   → Point head to newNode



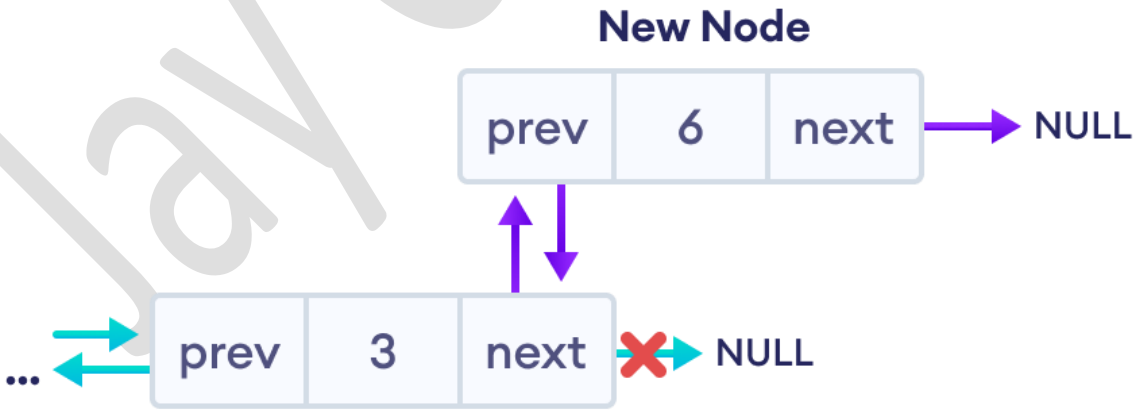
- **Insertion at the End**

→ Let's add a node with value 6 at the end of the doubly linked list.

1. Create a new node

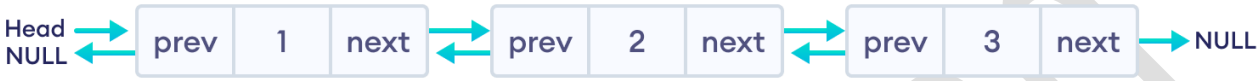
2. Set prev and next pointers of new node and the previous node

→ If the linked list is empty, make the newNode as the head node. Otherwise, traverse to the end of the doubly linked list and



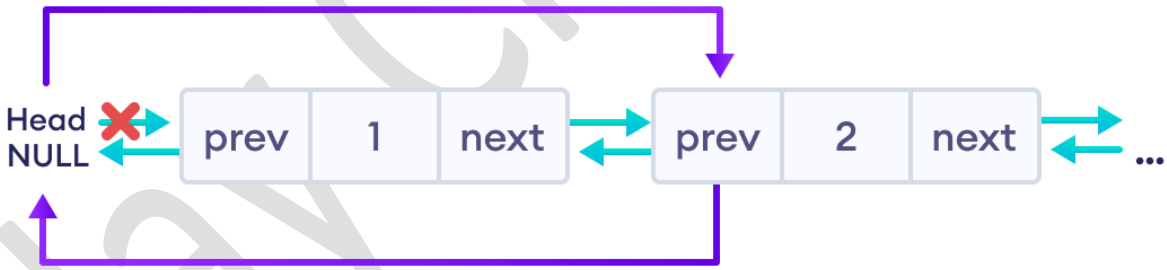
The final doubly linked list looks like this.

❖ Deletion from a Doubly Linked List

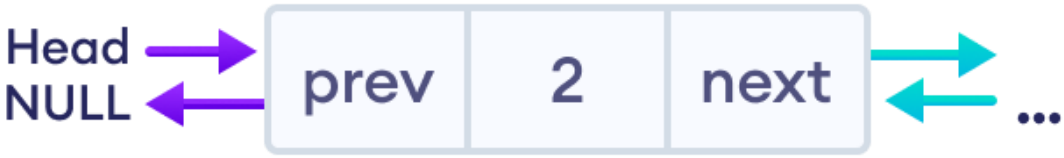


- **Delete the First Node of Doubly Linked List**

→ If the node to be deleted (i.e. del_node) is at the beginning
→ Reset value node after the del_node (i.e. node two)



Finally, free the memory of del_node. And, the linked will look like this.



Free the space of the first node

- **Delete the Last Node of Doubly Linked List**

→ In this case, we are deleting the last node with value **3** of the doubly linked list.

→ Here, we can simply delete the del_node and make the next of node before del_node point to NULL.