## Double Linked List

Doubly Linked List in C++ is very similar to a linked list, except that each node also contains a pointer to the node previous to the current one. This means that in a doubly linked list in C++ we can travel not only in the forward direction, but also in the backward direction, which was not possible with a palin linked list in C++.

Here, we have the same node class, the only difference is that now we have a prev pointer which points to the previous node.

```
class Node
{
        public:
                int val;
                Node *next;
                Node *prev;

                Node(int val)
                {
                        this->val=val;
                        next=NULL;
                        prev=NULL;
                }
};
```

### Destructor

```
~LinkedList()

{

        Node *curr=head;

        while(head!=NULL)

        {

                curr=head;

                head=head->next;

                delete curr;

        }

}
```

## Insertion in a Doubly linked List:

1. Input Validation: The insert method begins by checking if the given pos (position) is within a valid range. If pos is less than 1 or greater than the current length of the list plus 1, it's considered an invalid position, and the method displays an error message indicating that the position is invalid.
2. Insertion at the Beginning (pos == 1): If the position is valid and equal to 1, it means you want to insert a new node at the beginning of the list. In this case, a new node temp is created with the specified val. The next pointer of the new node (temp->next) is set to point to the current head of the list (head). Then, the head pointer is updated to point to the newly created node temp, effectively making it the new head of the list. This part ensures that the new node becomes the first node in the list. In case the list is empty initially so here we don't need to set the prev pointer.
3. Insertion at Other Positions (pos > 1): When inserting at a position other than the beginning, a new node temp is created similarly. To insert it at the desired position, a loop is used to traverse the list to the node immediately before the specified position. The variable curr is used to keep track of the current node during this traversal. The loop runs for (pos - 1) iterations, positioning curr at the node before the target position.
4. Adjusting Pointers for Insertion: Once curr is at the node before the target position, the next pointer of the new node (temp->next) is set to point to the node that was originally at the target position, which becomes the next node of the new node. Then, a check is made to ensure that the next node (curr->next) is not NULL. If it's not NULL, the prev pointer of the next node is adjusted to point back to the new node (temp) to maintain the doubly linked list structure. Afterward, the next pointer of curr is updated to point to the new node (temp), and the prev pointer of the new node is set to point back to curr. These operations effectively insert the new node into the list at the specified position while preserving the doubly linked list connections.
5. Length Update: Finally, regardless of where the insertion occurs, the length of the list is incremented by 1 to reflect the addition of the new node.

```
void insert(int pos, int val) {
        if(pos<1 || pos> (length+1)) {
                cout<<"Invalid position"<<endl;
                return;
        }

        else if(pos==1) { // insert at top
                Node *temp=new Node(val);
                if(isEmpty()) { // if list was empty
                        head=temp;
                } else { // if list was not empty
                        temp->next=head;
                        head->prev=temp;
                        head=temp;
                }
        } else {
                Node *temp=new Node(val);
                Node *curr=head;
                for(int i=1; i<(pos-1); i++) {
                        curr=curr->next;
                }
                temp->next=curr->next;
                if(curr->next!=NULL) {
                        curr->next->prev=temp;
                }
                curr->next=temp;
                temp->prev=curr;
        }
        length++;
}
```

## Deletion in a Doubly Linked List:

1. List Empty Check: The deleteNode method starts by checking whether the linked list is empty. If head is NULL, it means the list is empty, and there are no nodes to delete. In this case, the method displays a message indicating that the list is empty.
2. Delete Node at the Beginning (pos == 1): If the list is not empty and the target position is 1, it means you want to delete the first node in the list. In this case, a temporary pointer temp is created and initialized to the current head of the list. Then, the head pointer is updated to point to the second node in the list (temp->next). Finally, the node pointed to

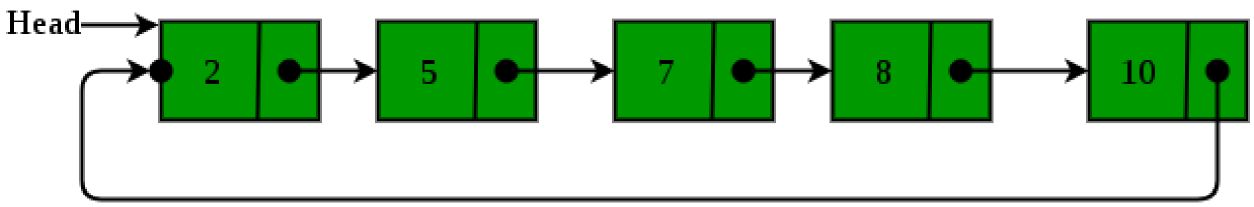
by temp is deleted using the delete keyword to free up the memory associated with it. This operation effectively removes the first node from the list.

3. Delete Node at Other Positions (pos > 1): When you want to delete a node at a position other than the beginning, the method uses a loop to traverse the list and position curr at the node to be deleted. The loop runs for pos iterations, positioning curr at the target node.
4. Adjusting Pointers for Deletion: Once curr is at the target node to be deleted, the method adjusts the pointers to maintain the integrity of the doubly linked list. Specifically, it updates the next pointer of the previous node (curr->prev->next) to skip over curr and point directly to the node after curr. Additionally, it checks if curr->next is not NULL, indicating that there is a node following curr. If so, it updates the prev pointer of the following node (curr->next->prev) to point back to the node before curr. These pointer adjustments effectively remove curr from the linked list while ensuring that the remaining nodes are correctly connected.
5. Memory Deallocation: Finally, after adjusting the pointers and effectively removing the node from the list, the method uses the delete keyword to free the memory occupied by the node curr. This step is essential to prevent memory leaks.
6. Decrement the length.

```
void deleteNode(int pos) {
        if(pos<1 || pos>length) {
                cout<<"Invalid position"<<endl;
                return;
        } else if(pos==1) {
                Node *curr=head;
                head=curr->next;
                delete curr;
        } else {
                Node *curr=head;
                for(int i=1; i<pos; i++) {
                        curr=curr->next;
                }
                curr->prev->next=curr->next;
                if(curr->next!=NULL) {
                        curr->next->prev=curr->prev;
                }
                delete curr;
        }
        length--;
}
```

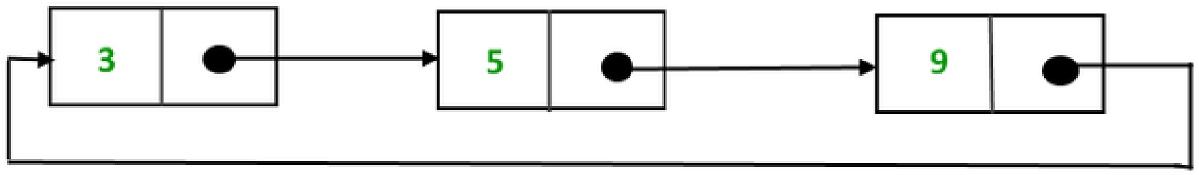
## Circular Linked List:

*The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.*



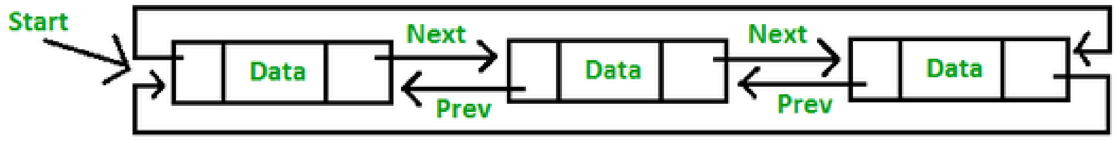
**There are generally two types of circular linked lists:**

- **Circular singly linked list:** In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



*Representation of Circular singly linked list*

- **Circular 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 and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



*Representation of circular doubly linked list*

**Note:** We will be using the singly circular linked list to represent the working of the circular linked list.

**Singly Circular Linked List:**

```
class Node {
        public:
                int val;
                Node *next;
                Node(int val) {
                        this->val=val;
                        next=NULL;
                }
};
```

Here we will take a tail pointer that keeps a track of the last node.

```
class LinkedList {
            int length;
            Node* head;
      public:
            LinkedList() {
                  length=0;
                  head=NULL;s
            }
};
```

**Destructor:**

```
~LinkedList() {

      Node* current = head;
      Node* nextNode;

      do {
            nextNode = current->next; // Store the next node
            delete current; // Delete the current node
            current = nextNode; // Move to the next node
      } while (current != head);
      head = nullptr;
}
```

## Insertion in a circular Linked List:

1. Position Validation: First, it checks whether the specified position pos is valid. If pos is less than 1 or greater than the length of the list plus one, it prints an "Invalid position" message and returns, indicating that no insertion will take place.
2. Insert at the Beginning: If pos is 1, it means you want to insert a new node at the beginning of the circular list. The code creates a new node temp with the provided value. If the list is empty (isEmpty() is true), it makes temp point to itself and sets head to temp. If the list is not empty, it connects temp to the current head, then iterates through the list to find the last node (curr) and makes it point to temp. Finally, it updates head to point to temp.
3. Insert at Any Other Location: If pos is not 1, it means you want to insert a new node at some other position. The code traverses the list to find the node just before the insertion position (pos - 1) and creates a new node temp with the provided value. It connects temp to the node at the insertion position and updates the previous node to point to temp.
4. Increment Length: Regardless of whether the insertion was successful or not, the length of the list is incremented by 1.

```
void insert(int pos, int val) {
        if(pos<1 || pos> (length+1)) {
                cout<<"Invalid position"<<endl;
                return;
        }

        else if(pos==1) { // insert at top
                Node *temp=new Node(val);

                if(isEmpty()) {
                        temp->next=temp;
                        head=temp;
                } else {
                        temp->next=head;
                        Node *curr=head;
                        while(curr->next!=head) {
                                curr=curr->next;
                        }
                        curr->next=temp;
                        head=temp;
                }

        } else { // insert at any other location
                Node *curr=head;
                for(int i=1; i<(pos-1); i++) {
                        curr=curr->next;
                }
                Node *temp=new Node(val);
                temp->next=curr->next;
                curr->next=temp;
        }
        length++;
}
```

## Printing a Circular Linked List:

In order to avoid an infinite loop problem while printing the list, we will check if the curr is not equal to head again. So here we will use a do while loop.

```
void printList() {
        if (isEmpty()) {
                cout << "List is empty" << endl;
                return;
        }
        Node* curr = head;
        do {
                cout << curr->val << "  ";
                curr = curr->next;
        } while (curr != head);
        cout << endl;
}
```

**Note: Now handle the deletion in the circular linked list yourself.**