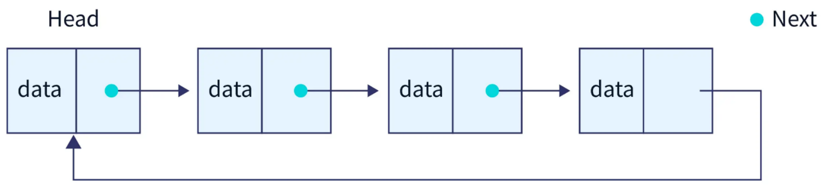
# Practical – 7
# Implementation of Circular and Doubly Linked List

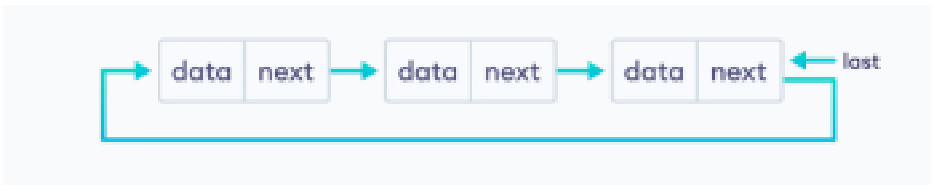
## Circular Linked list

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or a doubly circular 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 can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.





There are many applications in real life(as we will see later) that need to circle the list of elements multiple times. And as we have seen, in Linked List, the last node points to Null, and therefore to get the starting Node we need to rely on the Head pointer, which points to the starting Node.

Circular linked lists avoids this overhead of traversing till the Node points to null, and then using a head pointer to circle back from start, but it's the intrinsic property of Circular Linked lists that provides this property to circle the Linked List, by default.

We can traverse a circular linked list until we reach the same node where we started. The circular linked list has no beginning and no ending and there is no null value present in the next part of any of the nodes. These are some points that differentiate it from the linked list.

A circular linked list is represented similarly to the Linked List, with one additional care to connect the last node to the first Node.

So a Node in the Circular Linked List can be represented as:

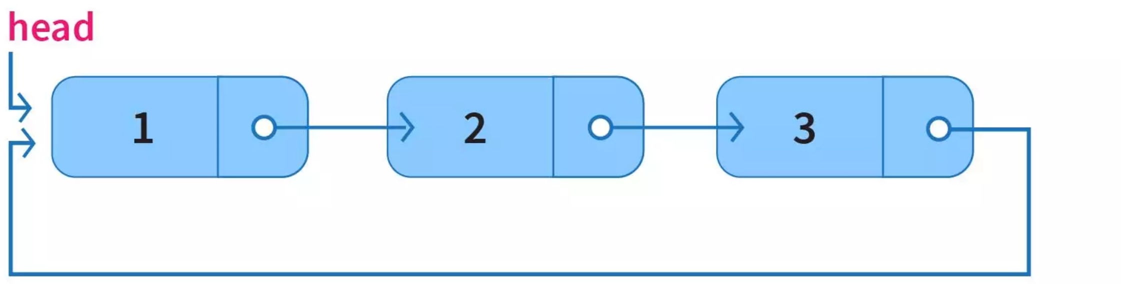
```
// Node structure is similar to what we have in Linked Lists.
class Node{
    int value;
    Node next; // Points to the next node.
}
```

We have seen how the Circular Linked List can be represented. To perform other operations on it, we need a pointer to any of the Nodes in the Circular Linked List, from where we can traverse the entire List.

Usually, in linked lists, we maintain the pointer to the first Node, known as the head pointer. Let's see if the same Node would work the best in the Circular Linked list.

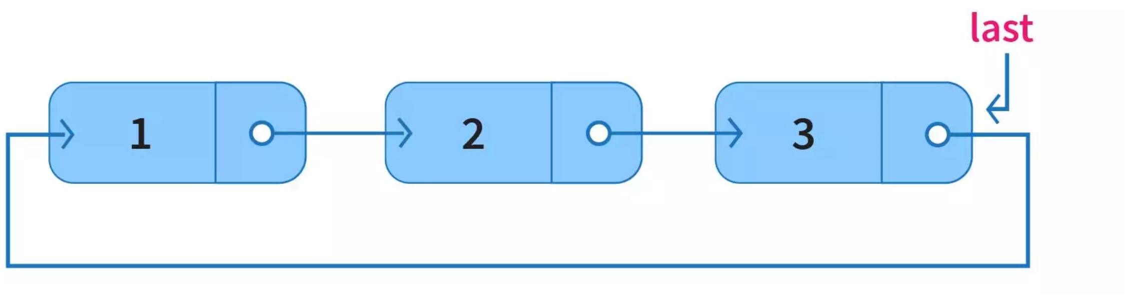
**Insertion**

We should be able to insert any new Node in the given Circular Linked List. Let's see if the head pointer would work with insertions. If we want to insert a new Node at the beginning of the Circular Linked List, and if we have the Head Pointer maintained, as shown:

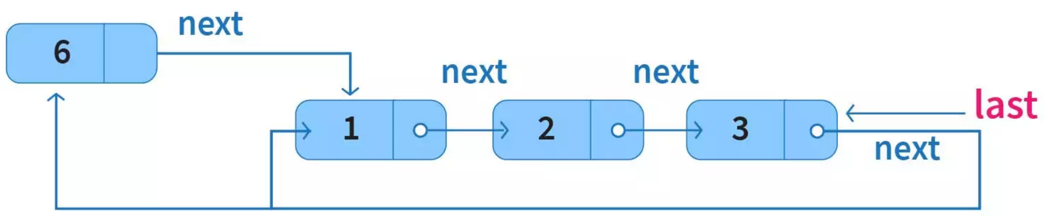


- Let's say we want to insert a new Node (say 4) at the beginning. How can we adjust the Nodes we already have in the list to accommodate this new Node?
- Since we maintain the head pointer(to 1), the next of a new Node (i.e. 4) can point to Head, and this takes care of including 4 in the list.
- But we are missing one thing here. The circular property is not yet adjusted.
- Last Node(i.e. 3) is still pointing to 1, whereas, with the addition of 4, 3’s next should point to 4 instead. How can this be done?
- We can transverse the list, reach the last Node in the circular linked list(i.e. To 3), and make its next point to the new Node(i.e. 4).
- But as you can guess, it is a slow process. Consider the scenario where we have 1000 nodes in the circular linked list, and just to add one Node we are traversing these 1000 Nodes to reach the last Node.
- Similarly, when we need to insert a new Node at the end(i.e. After 3), again we need to traverse the list again from head to reach the last node.

- So it seems the head pointer is not the best pointer for circular linked list in Data Structure.
- But what if we maintain the pointer to the Last Node instead?
- If instead of a start pointer, we take a pointer to the last node, then in both cases there won't be any need to traverse the whole list.
- Since we can reach head by just one jump from last using last.next, the head pointer is not much required. So in Circular Linked lists, the pointer to the last Node can be maintained, instead of the head Node(which is done for linked lists).
- So the circular linked list in Data Structure can be visualized as:



- Insertion at the Beginning: Now as we maintain the pointer to Last Node, we can get the first Node directly using Last.next.
- We can insert the Node at the beginning as:
    - Get the current First Node(by last.next), and the New Node can point to this node as its next. This takes care of adjusting pointers of the New Node.
    - Point the Last Node to the new Node, thus honouring the circular property.



```
Node last; // Pointer to the last Node of the Circular
Linked List.

void insertNodeAtBeginning(int value) {
    // Creating a new Node with the value.
    Node newNode = new Node(value);

    // Adjusting the links
    Node oldFirstNode = last.next;
    newNode.next = oldFirstNode;
    last.next = newNode;
}
```

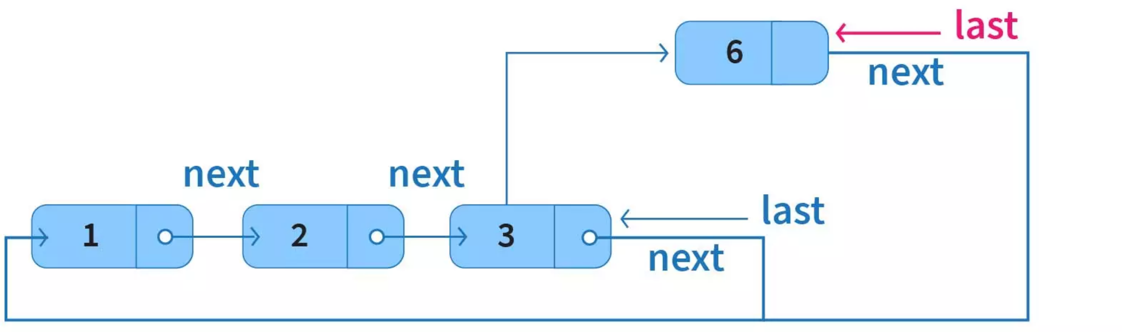
As we can see, we need not loop over the list, but we just need to change the pointers of the last Node and the new Node. So it takes constant time operation if we have the pointer to the last Node.

And as we have not used any extra space, the space complexity is constant i.e. O(1) Space complexity to insert a new Node at the beginning of the circular linked list.

**Insertion at the end:**

Let's see how the New Node can be inserted in-between 2 Nodes.

1. New Node will have its next pointing to the last.next.
2. Last.next will point to this new node.
3. Now since we have a new Node at the end, change the last Node pointer to point to this New node.



```
void addAtEnd(int value) {
    Node newNode = new Node(value);

    // Adjusting the links.
    newNode.next = last.next;
    last.next = newNode;
    last = newNode;
}
```

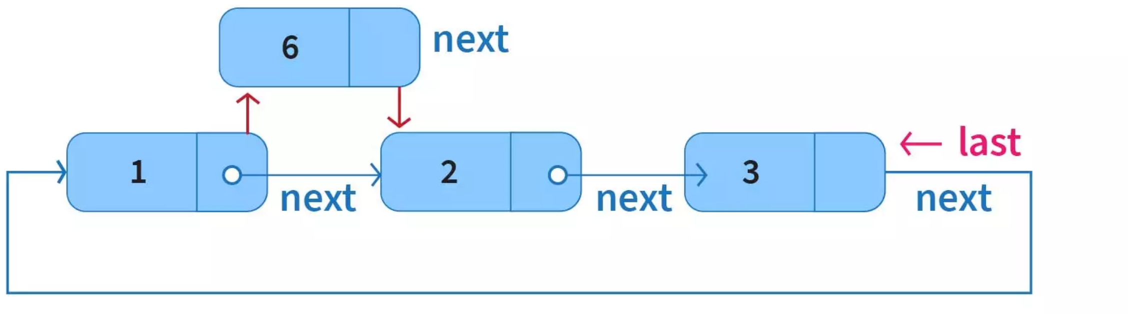
As we can see, to insert a new Node at the end of the circular linked list, we only need to adjust the pointers of the last node, hence it takes constant time to insert a new node at the end of the circular linked list.

As we have not used any extra space, the space complexity is constant i.e. O(1) space complexity to insert a new Node at the end of the circular linked list.

**Insertion After Another Node:**

Let's see how the New Node can be inserted in between 2 Nodes.

1. Transverse till we reach the given node(let this node be X).
2. Point the next of NewNode to the Next of X.
3. Point next to X to this new Node.



```
void addAfter(int newValue, Node nodeBefore) {
    if (nodeBefore == null) {
        return;
    }
    Node newNode;
    Node transversalNode = last.next; // Transverse the List
from last Node onwards.
    do {
        if (transversalNode.value == nodeBefore.value) { // We
found the node.
            newNode = new Node(newValue);

            // Adjusting the links
            newNode.next = transversalNode.next;
            transversalNode.next = newNode;

            if (transversalNode == last) {
                // If the nodeBefore was LastNode itself,
meaning we are inserting this node at the end,
                // adjust the last pointer to point to this
node now.
                last = newNode;
            }
        }

        transversalNode = transversalNode.next;
        // Keep transversing until we reach the Node after
which the new node has to be inserted.
    } while (transversalNode != last.next);
}
```

As we can see, we need to iterate over the list to reach the Node after which a new Node has to be inserted, which means it takes as much time as many elements are there in the List.

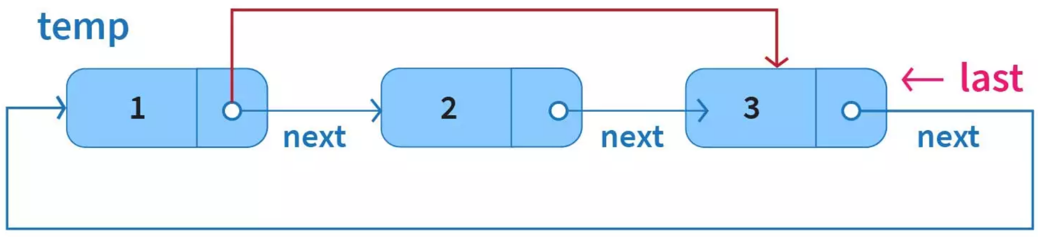
Thus if there are 1000 elements, and if the insert has to be done after the 999th Node, we would need to iterate over these 999 nodes, hence this insertion has the Time complexity of O(N), N being the number of elements in the circular linked list.

And as we have not used any extra space, the space complexity is constant i.e. O(1) space complexity to insert a new Node after a given node of the circular linked list.

**Deletion**

To delete a given node from a Circular Linked List, we can have 3 different scenarios:

1. If it's the only node in the Circular Linked List.
    - In this case, the last which was pointing to the only Node can now point to null, making the circular linked list empty.
2. If the Node to be deleted is the last node in the Circular Linked List:
    - Iterate the circular linked list to find the Node before the Node to delete. Let it be referred to as X.
    - Point the X.next to null, removing last from the circular linked list.
    - Point last pointer to X, signifying that X is the new last Node now.
3. Deleting any other node in the Circular Linked List:
    - Iterate the Circular Linked List to find the Node before the Node to delete. Let it be referred to as X.
    - Let the Node to be deleted be referred to as Y.
    - To remove Y, we need to change X.next to point to Y.next.



```
Node deleteNode(int valueOfNodeToDelete) {
    // if Circular Linked List is empty
    if (last == null)
        return null;

    // If the Circular Linked List contains only a single node
    // -- Scenario 1.
    if (last.value == valueOfNodeToDelete && last.next == last)
{
        // If Circular Linked List has only one Node, last.next
will point to last,
        // as last is itself the starting node in this case.
        last = null;
        return last;
    }

    Node temp = last;

    // If last is to be deleted --  Scenario 2.
    if (last.value == valueOfNodeToDelete) {
        // find the node before the last node
        while (temp.next != last) {
            temp = temp.next;
        }

        // point temp node to the next of last i.e. first node
        temp.next = last.next;
        last = temp.next;
    }

    // travel to the node to be deleted
    while (temp.next != last && temp.next.value !=
valueOfNodeToDelete) {
        temp = temp.next;
    }

    // If node to be deleted was found -- Scenario 3.
    if (temp.next.value == valueOfNodeToDelete) {
        Node toDelete = temp.next;
        temp.next = toDelete.next;
    }
    return last;
}
```

As we can observe, we might need to transverse the circular linked list to delete the node. This is required to find the Previous node, so that its next can be adjusted to remove the culprit node from the circular linked list chain.

So deletion takes O(N) time complexity, and as it takes no extra space to execute, its space complexity is O(1) i.e. content space complexity.

**Display**

To display the circular linked list, we need to transverse from the last Pointer until we reach back to it. As we transverse over the Nodes, we can print their values.

```
Node last; // pointer to last Node.

void displayList() {
    Node temp = last;

    if (last != null) {
        // Keep printing nodes till we reach the last node
again.
        do {
            System.out.print(temp.value + " ");
            temp = temp.next;
        } while (temp != last);
    }
}
```

**Applications of Circular Linked List**

Circular linked lists find their application in many real live systems where we can easily circle back to the starting Node from the last one. Some of those can be:

- In multiplayer games, each player is represented as a Node in the circular linked list. This way, each player is given a chance to play, as we can easily shuffle back to the first player from the last player quickly, utilizing the circular property.
- Similarly, operation systems utilize this concept for running multiple applications, where all these applications are placed in the circular linked list, and without worrying about reaching to the end of the running applications, it can easily circle back the applications at the start.

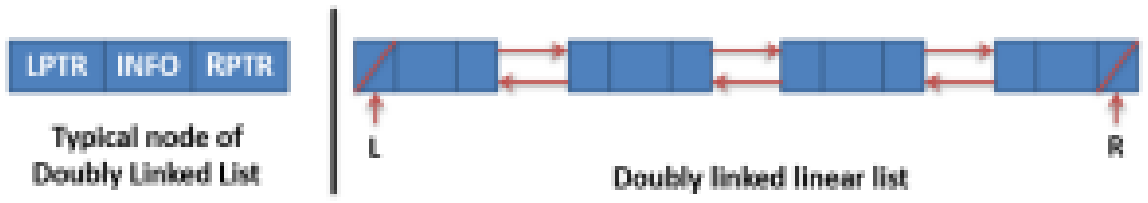
**Illustration of one Approach**:

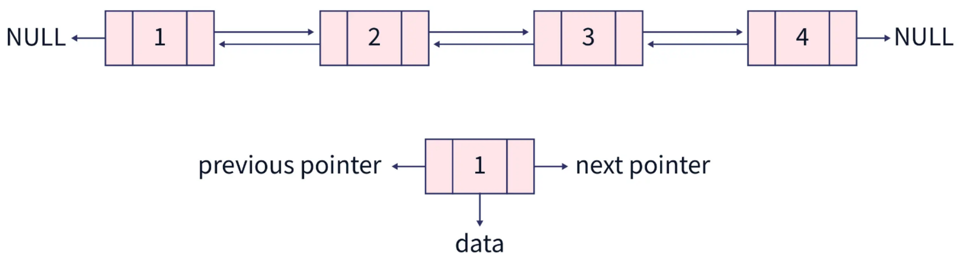
```
typedef struct Node
{
      int info;
      struct Node *next;
}node;

node *first=NULL,*last=NULL,*temp;
```

```
void create()
{
      node *newnode;
      newnode=(node*)malloc(sizeof(node));
      printf("\nEnter the node value : ");
      scanf("%d",&newnode->info);
      newnode->next=NULL;
      if(last==NULL)
            first=last=newnode;
      else
      {
            last->next=newnode;
            last=newnode;
      }
      last->next=first;
}

void del()
{
      temp=first;
      if(first==NULL)
      printf("\nUnderflow :");
      else
      {
      if(first==last)
      { printf("\n%d",first->info);
            first=last=NULL;
      }
      else
      { printf("\n%d",first->info);
            first=first->next;
            last->next=first;
      }
       temp->next=NULL;
      free(temp);
      }
}

…

…

…
```

**Doubly linked list**

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).





Doubly Linked List is a Data Structure, which is a variation of the Linked List, in which the transversal is possible in both the directions, forward and backward easily as compared to the Singly Linked List, or which is also simply called as a Linked List.

You might be aware of the Tabs in Windows/Mac. You can basically loop over all the opened applications, and can switch between these applications in both the directions. This can be thought of as all these Applications are linked via a Doubly Linked List Data structure, and you can switch to the applications on both sides of the current application.

A doubly linked list contains pointers to the previous node and the next node in the sequence. This is the biggest advantage of a doubly linked list over a singly linked list. Whenever the application requires navigation in both directions, we need a doubly linked list.

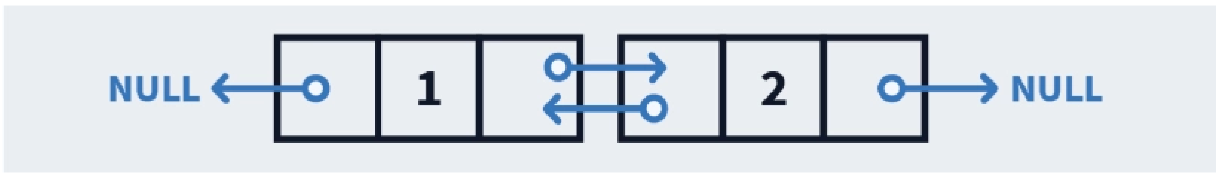
For example, a doubly linked list program in C can be used in a web browser for managing link navigation. Whenever the user clicks a new hyperlink, the link is inserted at the end into the doubly linked list. Now, if the user wants to go back and forth between the links, then this feature can be implemented by the forward and backward movement of the head pointer in a doubly linked list.

**So if a Linked List ⇒ A → B →. C**

**Then a Doubly Linked List ⇒ A ⇆ B ⇆ C**

The Previous pointer is the fundamental difference between the Doubly Linked List and the

Linked List, which enables one to transverse back also in the Doubly Linked List.



As per the above illustration, following are the important points to be considered.

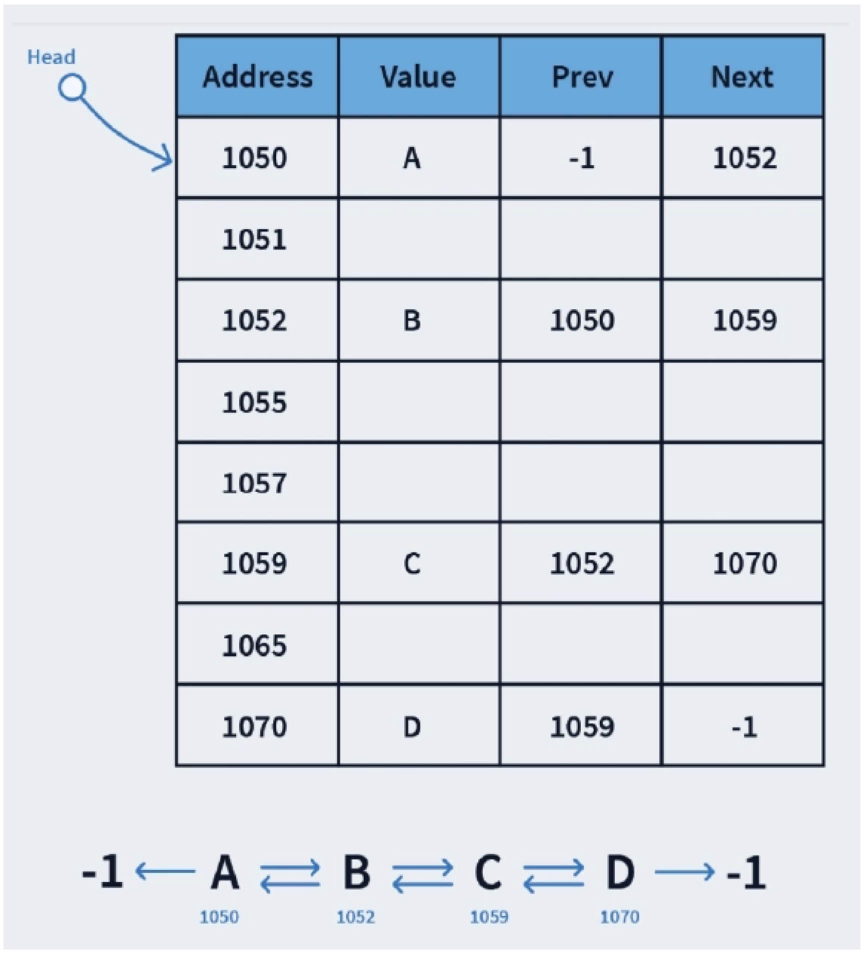
- Each Doubly Linked List Element contains pointers to next and Previous Elements and the data field.
- The First Element will have its Previous pointer as Null, to mark the start of the List.
- The Last Element will have its Next pointer as Null, to mark the end of the List.

From the Code Perspective, the Doubly Linked List can be represented as:

```
class Node {

   Node previous;  // Pointer to the Previous Element

   int value;      // Value stored in this element.

   Node next;      // Pointer to the Next Element

}
```

We also use the concept of "head" node, which points to the first Node in the Doubly Linked List. Using the head node we know where the Doubly Linked List starts from.

A Doubly Linked List is represented using the Linear Arrays in memory where each memory address stores 3 parts: Data Value, Memory address of next Element and Memory Address of the previous Element.



| Address | Value | Prev | Next |
|---|---|---|---|
| 1050 | A | -1 | 1052 |
| 1051 | | | |
| 1052 | B | 1050 | 1059 |
| 1055 | | | |
| 1057 | | | |
| 1059 | C | 1052 | 1070 |
| 1065 | | | |
| 1070 | D | 1059 | -1 |

**Traversing**

Since the Doubly Linked List has both next and previous pointers, this allows us to transverse in both the directions from the given Node. We can go to the previous node by following the previous pointer, and similarly go to the next nodes by following the next pointers.

```
 public void transverseForward(Node start) {
         Node current = start;

         //Checks whether the list is empty
         if (start == null) {
             System.out.println("Node to start from is
NULL");
             return;
         }

         while (current != null) {
             System.out.println("Node : " + current.value);
```

```
            current = current.next;
        }
    }


public void transverseBackward(Node start) {
   Node current = start;

   //Checks whether the list is empty
   if (start == null) {
       System.out.println("Node to start from is NULL");
       return;
   }

   while (current != null) {
       System.out.println("Node : " + current.value);
       current = current.previous;
   }
}
```

**Searching**

To search in the given doubly linked list, we would basically need to traverse the entire doubly linked list from the first node, and keep moving to next nodes using the next pointer. We can compare each transversed element against the one we are searching for.

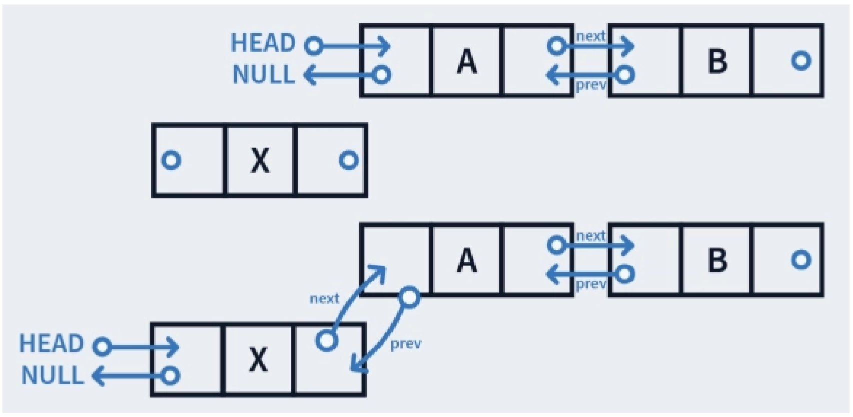
```
public void searchNode(int value) {
   int i = 1;
   //Node current will point to head
   Node current = head;

   //Checks whether the list is empty
   if (head == null) {
       System.out.println("List is empty");
       return;
   }

   // Traversing the List.
   while (current != null) {
       //Compare value to be searched with each node in
the list
       if (current.value == value) {
           System.out.println("Node is present in the list
at the position : " + i);
           return;
       }
       current = current.next;
       i++;
   }
   System.out.println("Node is not present in the list");
}
```

**Insertion**

Insertion into the Doubly Linked List is fast and can be done in constant times, as we just need to change the pointers of adjacent elements we are inserting this element between.

Insert First

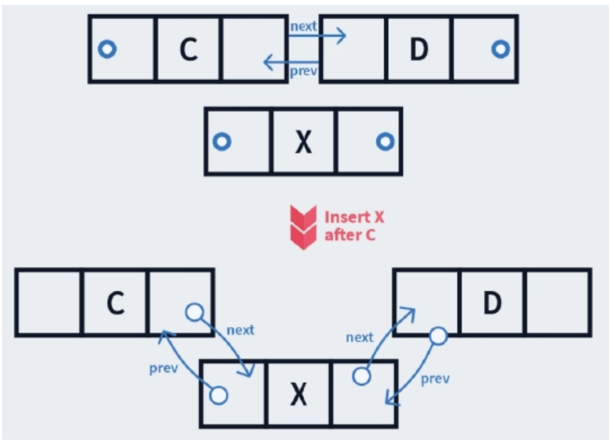
- A Node can be inserted as the first element in the Doubly Linked List.
- Earlier Doubly Linked List: A ⇆ B ⇆
- Now after we insert X inside this Doubly Linked List: X ⇆ A ⇆ B ⇆ C



```
// Adding a node at the front of the list
public void insertFront(int newValue) {

            // Create New Node and put data in it.
            Node newNode = new Node(newValue);

            // Update next of the new node as head and previous
     as NULL.
            newNode.next = head; // i.e. X.next = A. Head
     points to A yet.

            // Since X = first Node, so its prev should point
     to NULL.
            newNode.previous = null;

            // Change prev of head node to new node
            // Head points to A, so change A Previous to be X.
            if (head != null)
                head.previous = newNode;

            // Since X and A pointers are altered, we can not
     point Head to X which is the new start of the DLL.
            head = newNode;
}
```

Insert After a Node

- If we are given a Node(say C) and we want to insert a new Node(say X) after it, then the algorithm would look like:
- DLL Before ⇒ B ⇆ C ⇆ D ⇆ E
- After Insert X after C⇒ B ⇆ C ⇆ X ⇆ D ⇆ E



```
// Given a node as previousNode, insert a new node after it.
public void InsertAfter(Node previousNode, int newValue) {

   // Check if the given prev_node is NULL
   if (previousNode == null) {
       System.out.println("The given previous node cannot be
NULL ");
       return;
   }

   // Create New Node and put data in it.
   Node newNode = new Node(newValue);
   // Make next of new node as next of previousNode
   // X --> C.next i.e. X --> D
   newNode.next = previousNode.next;
   // Make the next of previousNode as newNode
   // C --> X
   previousNode.next = newNode;
   // Make previousNode as previous of newNode
   // C <-- X
   newNode.previous = previousNode;
   // Change the previous of D to point to X.
   // newNode.next = D, D.previous = X OR set X <-- D
   if (newNode.next != null)
       newNode.next.previous = newNode;
}
```

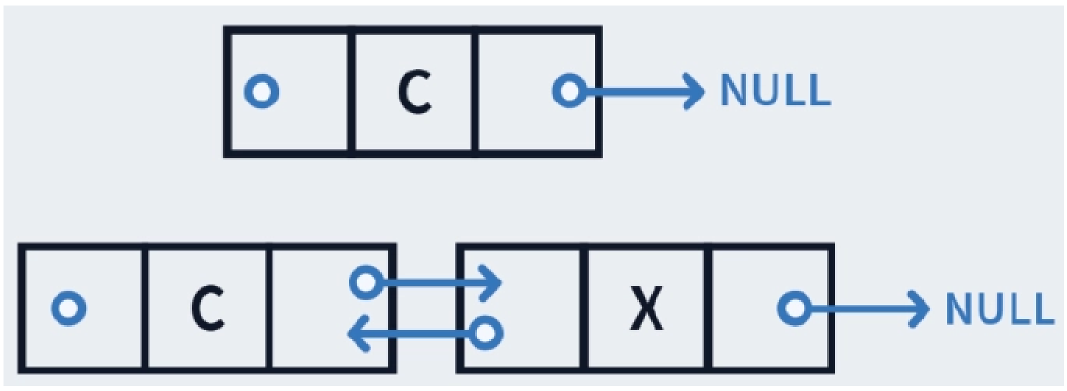
Insert Before a Node in Linked List

- If we are given a Node(say C) and we want to insert a new Node(say X) before it
- DLL Before ⇒ B ⇆ C ⇆ D ⇆ E
- After Insert X after C ⇒ B ⇆ X ⇆ C ⇆ D ⇆E

```
void insertBefore(Node head, Node nextNode, int newValue) {

    // Check if the given nextNode is NULL
    if (nextNode == null) {
        System.out.print("The given next node cannot be NULL");
        return;
    }

    // Create New Node and put data in it.
    Node newNode = new Node(newValue);

    // Make prev of new node as prev of nextNode
    // C.Prev <-- X OR B <-- X
    newNode.previous = nextNode.previous;

    // Make the prev of nextNode as newNode
    // X <-- C
    nextNode.previous = newNode;

    // Make nextNode as next of newNode
    // X --> C
    newNode.next = nextNode;

    // Change next of newNode's previous node
    // X.previous = B so B.next = X i.e. B --> X
    if (newNode.previous != null)
        newNode.previous.next = newNode;
    else
        // If the prev of newNode is NULL, it will be the new
head node
        head = newNode;

  }
```

<u>Insert last</u>

- If we want to insert a ⇆ Xnew Node(say X) at the end of the Doubly Linked List
- DLL Before ⇒ A ⇆ B ⇆ C
- After Insert X at end ⇒ A ⇆ B ⇆ C



```
// Add a node at the end of the list
void insertAtEnd(int newValue) {
   // Create New Node and put data in it.
   Node newNode = new Node(newValue);

   // Since this Node = Last Node, its Next needs to be Null.
   newNode.next = null;

   // If the Linked List is empty, then make this new Node as
head
   if (head == null) {
       newNode.previous = null;
       head = newNode;
       return;
   }

   Node last = head;
   // Else traverse till the last node
   while (last.next != null)
       last = last.next;

   // Change the next of last node
   // C --> X
   last.next = newNode;

   // Make last node as previous of new node
   // C <-- X
   newNode.previous = last;
}
```

**Deletion**

- Delete any Given Node
- Delete a Given Position
- Delete First
- Delete Last etc.

<u>Sample:</u>

```
struct node  {
 int data;
 struct node *prev, *next;
};
struct node* head = NULL;
struct node* tail = NULL;
void insertFront()
{
      int val;
      struct node* temp;
      temp=(struct node*)malloc(sizeof(struct node));
      printf(" Enter value= ");
      scanf("%d",&val);
      temp->data=val;
      temp->prev=NULL;
      temp->next=head;
      if(head!=NULL)
            head->prev=temp;
      head=temp;

}
void insertPosition()
{
      int val,pos,i=1;
      struct node *temp, *newnode;
      newnode=malloc(sizeof(struct node));
      newnode->next=NULL;
      newnode->prev=NULL;
      printf("Enter position= ");
      scanf("%d",&pos);
      printf("Enter value= ");
      scanf("%d",&val);
      newnode->data=val;
      temp=head;
      if(head==NULL)
      {
            head=newnode;
            newnode->prev=NULL;
            newnode->next=NULL;
      }
      else if(pos==1)
      {
            newnode->next=head;
            newnode->next->prev=newnode;
            newnode->prev=NULL;
            head=newnode;
      }
      else
      {
            while(i<pos-1)
             {
              temp = temp->next;
              i++;
             }
            newnode->next=temp->next;
            newnode->prev=temp;
            temp->next=newnode;
            temp->next->prev=newnode;
 }
}
void deleteFirst()
{ struct node* temp;
      if(head==NULL)
      { printf("List is empty\n"); }
      else
      { temp=head;
            head=head->next;
            if(head!=NULL)
             {
       head->prev=NULL;
       }
            free(temp);
      }
}
void deleteEnd()
{ struct node* temp;
      if(head==NULL)
      { printf("List is empty\n"); }
      temp=head;
      while(temp->next!=NULL)
      { temp=temp->next;
      }
      if(head->next==NULL)
      { head=NULL;
      }
      else
      { temp->prev->next=NULL;
            free(temp);
      }
}
void deletePosition()
{ int pos,i=1;
      struct node *temp, *position;
      temp=head;
      if(head==NULL)
      {
            printf("List is empty\n");
      }
      else
      { printf("Enter position= ");
            scanf("%d",&pos);
            if(pos==1)
            { position=head;
             head=head->next;
             if(head!=NULL)
             {
                  head->prev=NULL;
             }
             free(position);
             return;
             }
             while(i<pos-1)
             {
       temp=temp->next;
       i++;
             }
             position=temp->next;
             if(position->next!=NULL)
             {
       position->next->prev=temp;
             }
             temp->next=position->next;
             free(position);
      }
}
```

**Exercise**

1. Write a program to implement Enqueue and Dequeue operations of circular queue using  circular link list.
2. Write a program for all operations of a circular singly linked list.
    a. Inserting Node – as First Node, at specific location, as Last Node
    b. Deleting Node – at First, at Last, specific node
    c. Display List
3. Write a program for all operations of doubly linked list.
    a. Inserting Node – as First Node, at specific location, as Last Node
    b. Deleting Node – at First, at Last, specific node
    c. Display List
4. Write a program for all operations of circular doubly linked list.
    a. Inserting Node – as First Node, at specific location, as Last Node
    b. Deleting Node – at First, at Last, specific node
    c. Display List