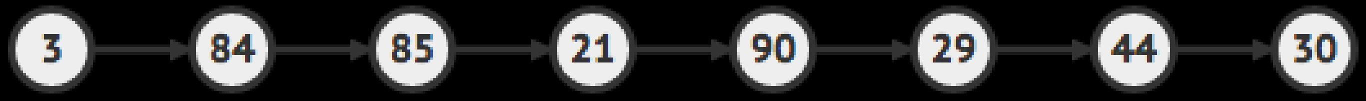
Chapter 5

Lists-PartII

Previous class

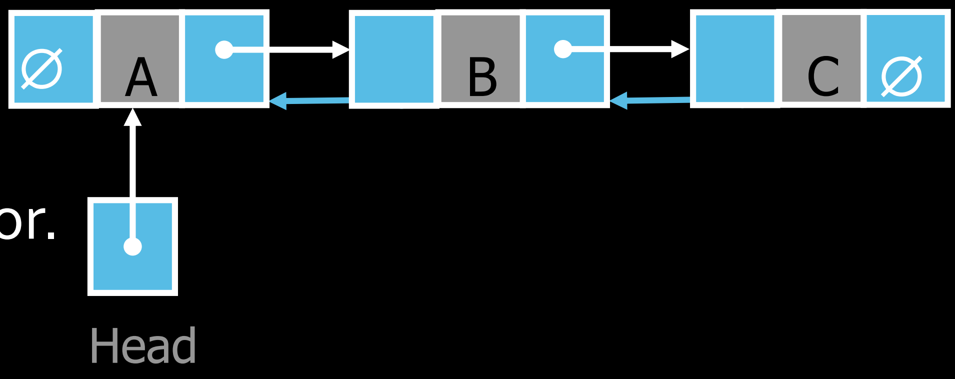
- Singly linked lists hold a reference only to the next node.



- In the implementation you always maintain a reference to the **head** to the linked list and optionally, a reference to the **tail** node for quick additions/removals.
- Today:
  - Doubly linked list
  - Circular linked list

Doubly linked lists

- A doubly linked list is one in which all nodes are linked together by multiple links
  - Each node points to not only successor but to the predecessor as well.
- There are two `NULL`: at the first and last nodes in the list
- Every nodes in the doubly linked list has minimum of three fields:
  1. `LeftPointer(previous)`: Point to the prevous node
  2. `RightPointer(next)`: Points to the next node in the list
  3. `DATA`: Actual data the node stores



- **Advantage:**
  - Given a node, it is easy to visit its predecessor.
  - Convenient to traverse lists backwards

DLL Operations

- In a doubly linked list we perform all valid operations of the List ADT
  1. Insertion
  2. Deletion
  3. Display
  4. Search

Empty List

- Before we implement actual operations, first we need to setup empty list.
- First perform the following steps before implementing actual operations.
  - **Step 1:** Include all the **header files** which are used in the program.
  - **Step 2:** ***Define a Node structure*** with members **data** , **next** and **previous pointers**
  - **Step 3:** Define a Node pointer '**head**' and **'tail'** set to **NULL**.

```cpp
#include <iostream>

using namespace std;
struct Node
{
    int data;
    Node *next;
    Node *previous;
}*head=NULL,*tail=NULL;

int main()
{

    return 0;
}
```

Insertion

- In a doubly linked list, the insertion operation can be performed in three different locations.
    1. Inserting At Beginning of the list
    2. Inserting At End of the list
    3. Inserting At Specific location in the list

Inserting At Beginning of the list

- We can use the following steps to insert a new node at beginning of the **doubly linked list**
  - **Step 1:** Create a **newNode** with given value and **newNode** ⟶ **previous** as **NULL**
  - **Step 2:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 3:** If it is **Empty** then,set
    - **newNode→next = NULL, head =newNode** and **tail =newNode.**
  - **Step 4:** If it is **Not Empty** then, set **newNode→next = head , head→previous = newNode** and **head =newNode**

```
Node *newNode=new Node;
newNode->previous=NULL;
newNode->data=value;
if(head==NULL)
{
    newNode->next=NULL;
    head=newNode;
    tail=newNode;
}
else
{
    newNode->next=head;
    head->previous=newNode;
    head=newNode;
}
```

Inserting At End of the list

- We can use the following steps to insert a new node at end of the **doubly linked list**
  - **Step 1:** Create a **newNode** with given value and **newNode → next** as **NULL**
  - **Step 2:** Check whether list is **Empty** (**head == NULL**)
  - **Step 3:** If it is **Empty** then,set
    - **newNode→next** = **NULL, head** =**newNode** and **tail** =**newNode**.
  - **Step 4:** If it is **Not Empty** then, set **newNode→previous** = **tail** , **tail→next** = **newNode** and **tail** =**newNode**

```
Node *newNode=new Node;
newNode->data=value;
newNode->next=NULL;
if(head==NULL)
{
    newNode->next=NULL;
    head=newNode;
    tail=newNode;
}
else
{
    newNode->previous=tail;
    tail->next=newNode;
    tail=newNode;
}
```

Inserting At Specific location in the list (After a Node)

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then, set
  - **newNode→next** = **NULL**, **head** =**newNode** and **tail** =**newNode**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Cont…

- **Step 6:** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7:** Check whether **temp → data** is equal to **location, if it is TRUE** go to step 8 otherwise terminate the function
- **Step 8:** Check whether temp is the last node, if yes **tail=newNode**, if no **(temp->next)->previous=newNode**
- **Step 9**: Finally, Set **'newNode → next = temp → next'** ,, **'temp → next = newNode'** and **newNode->previous=temp**

```
Node *newNode=new Node;
newNode->data=value;
if(head==NULL)
{
    newNode->next=NULL;
    head=newNode;
    tail=newNode;
}
else
{
    Node *temp=head;
    while(temp->data!=after)
    {
        if(temp->next==NULL)
        {
            cout<<"Given node is not found in the list!"<<endl;
            break;
        }
        temp=temp->next;
    }
    if(temp->data==after)
    {
        if(temp->next==NULL)
        {
            tail=newNode;
        }
        else
        {
            (temp->next)->previous=newNode;
        }

        newNode->next=temp->next;
        newNode->previous=temp;
        temp->next=newNode;
    }
}
```

Deletion

- In a doubly linked list, the deletion operation can be performed from three different locations.
  1. Deleting from Beginning of the list
  2. Deleting from End of the list
  3. Deleting a Specific Node

Deleting from Beginning of the list

- We can use the following steps to delete a node from beginning of the doubly linked list
  - **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
  - **Step 4:** Check whether list is having only one node (**temp** → **next** == **NULL**)
  - **Step 5:** If it is **TRUE** then set **head** = **NULL** , **tail** = **NULL** and delete **temp** (Setting **Empty** list conditions)
  - **Step 6:** If it is **FALSE** then set **head** = **temp** → **next**, (**temp-> next)->previous=NULL** and delete **temp**.

```
if(head==NULL)
{
    cout<<"List is empty"<<endl;
}
else
{
    Node *temp=head;
    if(temp->next==NULL)
    {
        head=NULL;
        tail=NULL;
    }
    else
    {
        head=temp->next;
        temp->next->previous=NULL;
    }
    delete temp;
}
```

Deleting from End of the list

- We can use the following steps to delete a node from end of the doubly linked list
  - **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **tail**.
  - **Step 4:** Check whether list is having only one node (**temp** → **previous** == **NULL**)
  - **Step 5:** If it is **TRUE** then set **head** = **NULL** , **tail** = **NULL** and delete **temp** (Setting **Empty** list conditions)
  - **Step 6:** If it is **FALSE** then set **tail** = **temp** → **previous**, (**temp-> previous)->next=NULL** and delete **temp**.

```
if(head==NULL)
{
    cout<<"List is empty"<<endl;
}
else
{
    Node *temp=tail;
    if(temp->previous==NULL)
    {
        head=NULL;
        tail=NULL;
    }
    else
    {
        tail=temp->previous;
        (temp->previous)->next=NULL;
    }
    delete temp;
}
```

Deleting a Specific Node from the list

- We can use the following steps to delete a specific node from the doubly linked list...
  - **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - **Step 3:** If it is **Not Empty** then, define Node pointers **'temp'** initialize with **head**.
  - **Step 4:** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node. And every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
  - **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Cont…

- **Step 7:** If list has only one node and that is the node to be deleted, then set **head** = **NULL, tail** = **NULL** and delete **temp**.
- **Step 8:** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9:** If **temp** is the first node then move the **head** to the next node (**head = head** → **next**) , **(temp->next)->previous=NULL** and delete **temp**.
- **Step 10:** If **temp** is not first node then check whether it is last node in the list (**temp == tail**).
- **Step 11:** If **temp** is last node (**tail= temp**→ **previous**) , **(temp->previous)->next =NULL** and delete **temp**.
- **Step 12:** If **temp** is not first node and not last node then set **(temp->previous)->next=temp->next**, **(temp->next)->previous=temp->previous** and delete **temp**.

```
if(head==NULL)
{
    cout<<"List is empty"<<endl;
}
else
{
    Node *temp=head;
    while(temp->data!=key)
    {
        if(temp->next==NULL)
        {
            cout<<"Given node is not found in the list!"<<endl;
            break;
        }
        temp=temp->next;
    }
    if(temp->data==key)
    {
        if(head->next==NULL)
        {
            head=NULL;
            tail=NULL;
        }
        else
        {
            if(temp==head)
            {
                head=head->next;
                (temp->next)->previous=NULL;
            }
            else if(temp==tail)
            {
                tail=temp->previous;
                (temp->previous)->next=NULL;
            }
            else
            {
                (temp->previous)->next=temp->next;
                (temp->next)->previous=temp->previous;
            }
        }
    }
}
```

Display

– In a doubly linked list, the display operation can be performed in two ways. They are as follows

1. **Display forward**: Displays the complete list in a forward manner.
2. **Deleting backward**: Displays the complete list in a backward manner.

Displaying forward

- We can use the following steps to display forward the elements of a doubly linked list
  - **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 2:** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
  - **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
  - **Step 4:** Keep displaying **temp** → **data** with an arrow (**--->**) until **temp==NULL,** every time move the **temp** to next node.

```
if(head==NULL)
{
    cout<<"List is empty"<<endl;
}
else
{
    Node *temp=head;
    while(temp!=NULL)
    {
        cout<<temp->data<<endl;
        temp=temp->next;
    }
}
```

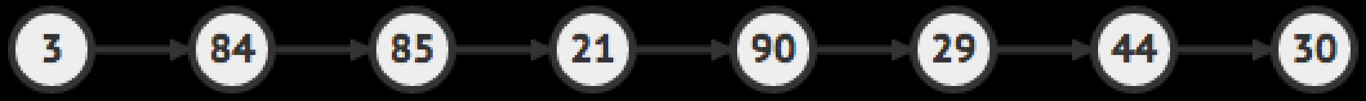
Displaying Backward

- We can use the following steps to display backward the elements of a doubly linked list
  - **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 2:** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
  - **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **tail**.
  - **Step 4:** Keep displaying **temp** → **data** with an arrow (**-→**) until **temp==NULL,** every time move the **temp** to **previous** node.

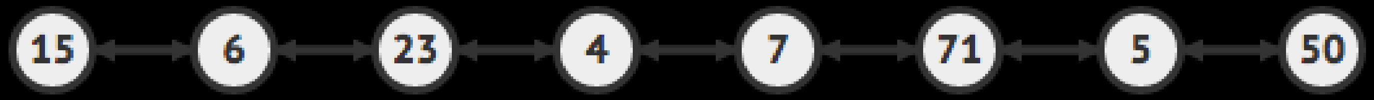
```
if(head==NULL)
{
    cout<<"List is empty"<<endl;
}
else
{
    Node *temp=tail;
    while(temp!=NULL)
    {
        cout<<temp->data<<endl;
        temp=temp->previous;
    }
}
```

Singly vs Doubly Linked Lists

- Singly linked lists only hold a reference to the next node. In the implementation you always maintain a reference to the **head** to the linked list and optionally, a reference to the **tail** node for quick additions/removals.



- With a doubly linked list each node holds a reference to the next and previous node. In the implementation you always maintain a reference to the **head** and the **tail** of the doubly linked list to do quick additions/removals from both ends of your list.



DLLs compared to SLLs

- **Advantages:**
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier

- **Disadvantages:**
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Complexity

	Singly Linked	Doubly Linked
Search	O(n)	O(n)
Insert at head	O(1)	O(1)
Insert at tail	O(1)(if tail is used) O(n) otherwise	O(1) (if tail is used) O(n) otherwise

Complexity

	Singly Linked	Doubly Linked
`Remove at head`	O(1)	O(1)
`Remove at tail`	O(n)	O(1) (if tail is used), O(n) otherwise.
`Remove in middle`	O(n)	O(n)

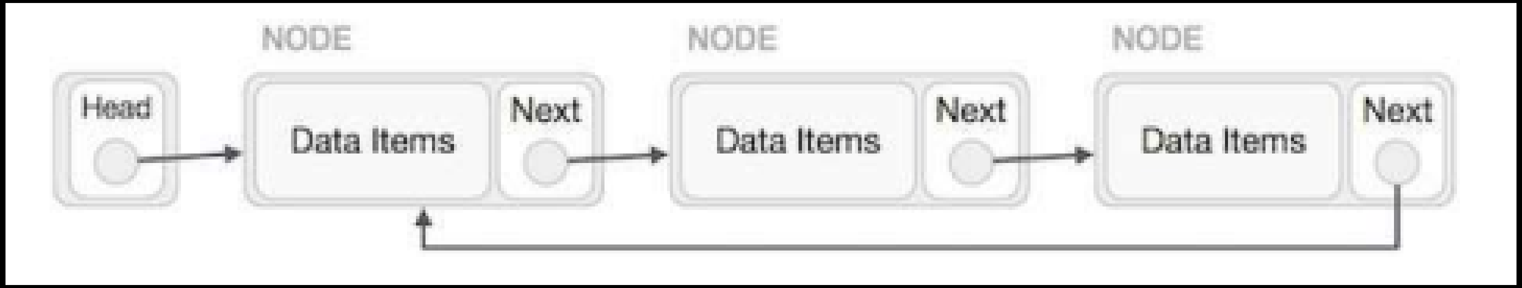
Circular List

Introduction

- Circular Linked List is a variation of Linked list in which
  - The **first** element points to the **last** element and
  - The **last** element points to the **first** element.
- Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

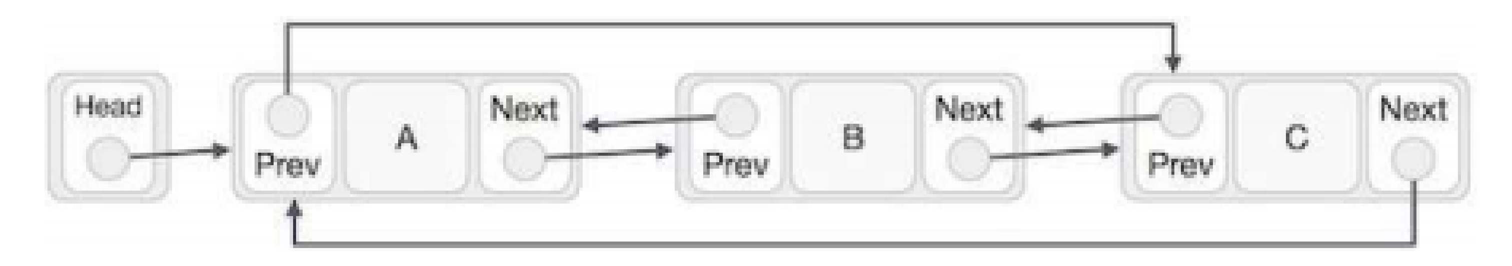
Singly Linked List as Circular List

- In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

- In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



Note:

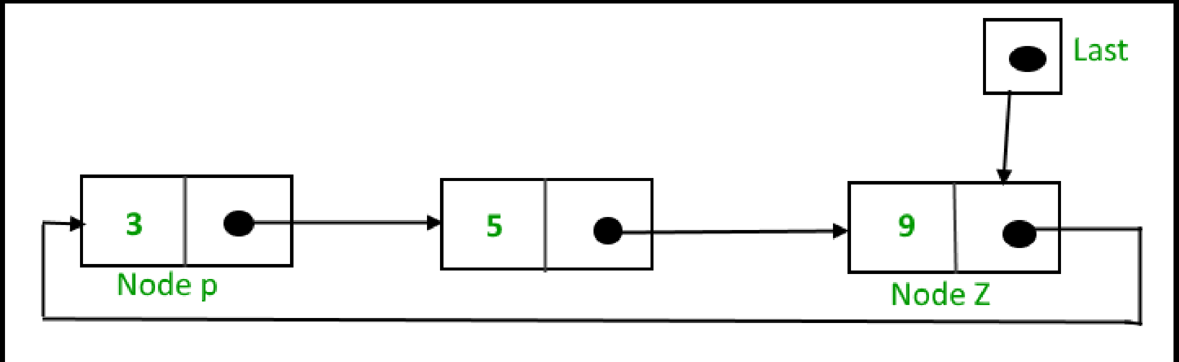
The **last node's next** points to **the first node** of the list in both cases of singly as well as doubly linked list.
The **first node's previous** points to the **last node of the list** in case of doubly linked list.

Application of Circular Linked List

- Multitasking OS
  - All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Multiplayer games.
  - All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Queue.
  - In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Implementation

- To implement a circular singly linked list, we use a pointer that points to the **last node** of the list.
  - If we have a pointer **last** pointing to the last node, then last -> next will point to the first node.
- ***Why we use a pointer that points to the last node instead of first node ?***
  - Because insertion in the beginning or at the end of the list takes constant time irrespective of the length of the list i.e O(1).
  - However, if we use head pointer instead, we need to traverse the whole list i.e O(n).



Operations

- Following are the important operations supported by a circular list.
  1. Insertion
  2. Deletion
  3. Display

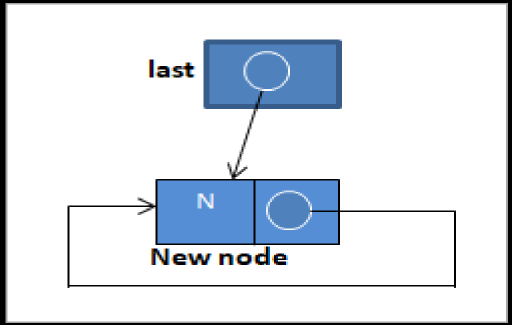
Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

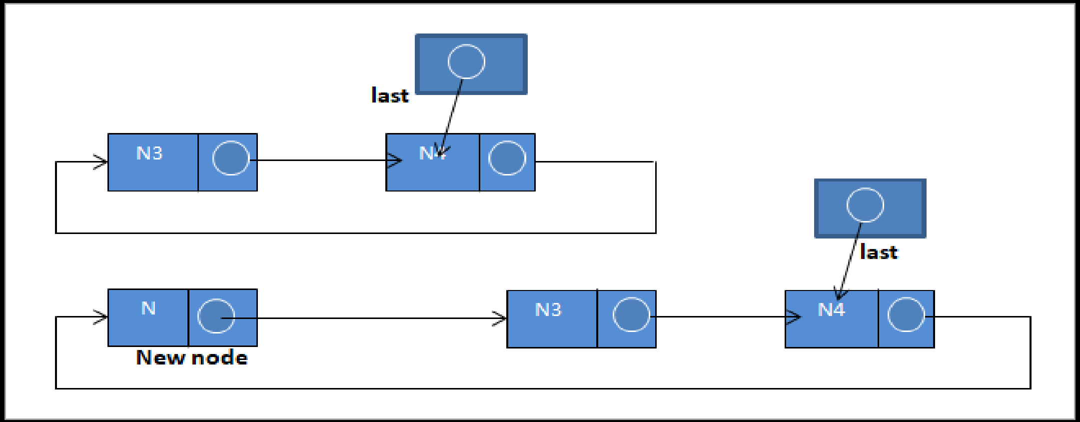
- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

Inserting At Beginning of the list

- We can use the following steps to insert a new node at beginning of the circular linked list...



- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**last** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **last** = **newNode** and **newNode→next** = **last** .
- **Step 4 -** If it is **Not Empty** then,
  - Set '**newNode → next =last->next**',
  - **last->next** = **newNode**'



Sample code to add node to front the list

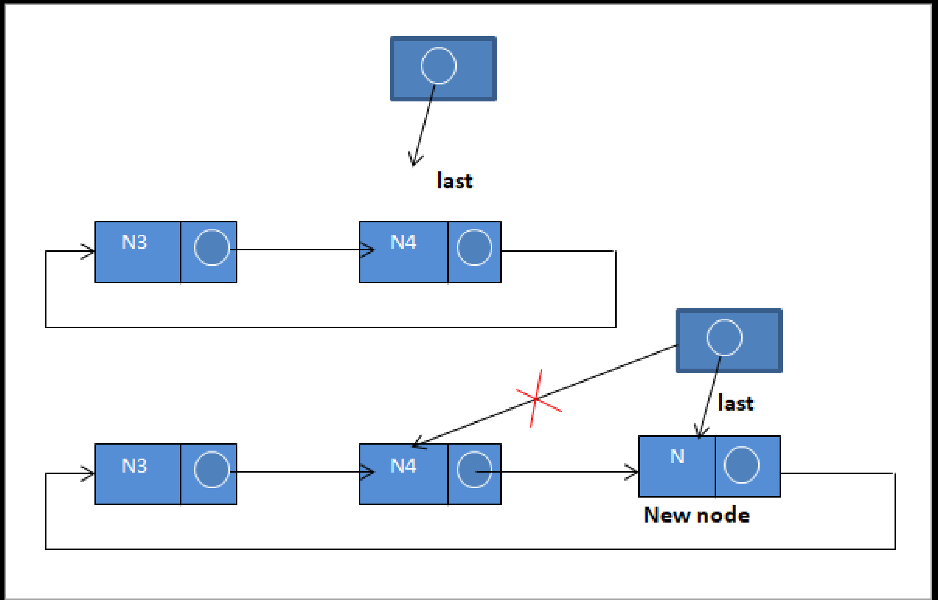
```
void addFront(){
    // Create new node and populate it with data
    Node *newNode= new Node;
    cout<<"Enter value for the node\n";
    cin>>newNode->d;

    //Check if list is empyty
if(last==NULL){
    last=newNode;
    newNode->next=last;
}
else{
    newNode->next=last->next; //make the new node point to what last pointing earlier (1st node)
    last->next=newNode; //Make the last node point to the new 1st node
    //last=newNode;
}
}
```

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list…

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **last** = **newNode** and **newNode** → **next** = **last**.
- **Step 4 -** If it is **Not Empty** then, Set
  - **newNode** → **next** = **last->next**.
  - **Last->next** = **newNode** and
  - Last= newNode



Sample code to add node to the rear of CSL

```
void addLast(){
 // Create new node and populate it with data
    Node *newNode= new Node;
    cout<<"Enter value for the node\n";
    cin>>newNode->d;

//Check if list is empyty. if so, make it point to itself.
    if(last==NULL){
        last=newNode;
        newNode->next=last;
    }
else{
        // if not empty List, then rearage the pointers to add the node last
        newNode->next=last->next;
        last->next=newNode;
        last=newNode;
}
```

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

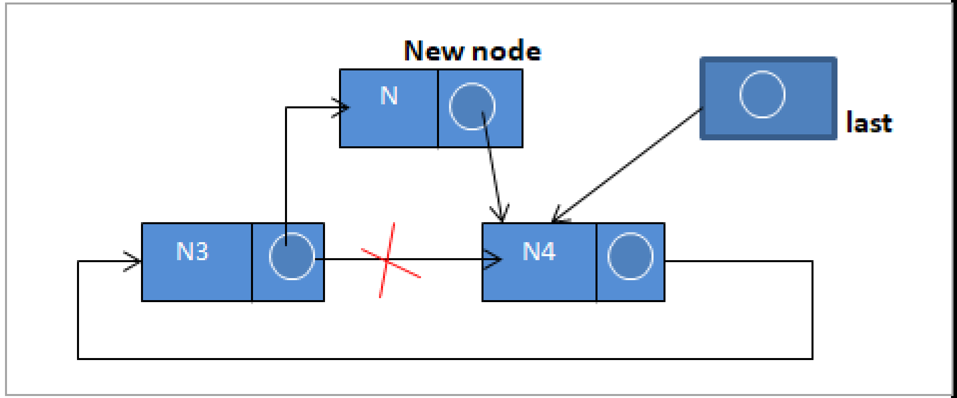
- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** define a node pointer **temp** and make it point to the 1st node **(temp= last->next**.)
- **Step 3 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
  - **In each iteration,** check whether **temp** is reached to the last node or location is found.
- **Step 4**- If temp reached to last node then
  - display **'Given location is not found in the list!!! Insertion not possible!!!'** and terminate the function.
- **Step 5**- If **temp** is reached to the exact node after which we want to insert the newNode
  - **newNode → next** = **temp->next**
  - **temp → next** = **newNode**
- **Step 6 -**Then check whether location is at the last node. If so move **last pointer** to the new last node that is **newNode**
  - last == newNode.

Sample code: addAfter(Location)

```cpp
void addAfter(int location)
{
    if(last==NULL){
        cout<<"'Given location "<<location<<" is not found in the list!!! Insertion not possible!!!\n";
        return;
    }
     // Create new node and populate it with data
    Node *newNode= new Node;
    cout<<"\n Enter value for the node\n";
    cin>>newNode->d;

    Node *temp=last->next;
    // Searching the item.
    do
    {
        if (temp ->d == location)
        {
            newNode -> next = temp -> next;// Adjusting the links.
            temp -> next = newNode;// Adding newly allocated node after temp.

            if (temp == last)// Checking for the last node.
                last = newNode;
            return ;
        }
        temp = temp -> next;
    } while (temp != last -> next);
    cout<<"'Given location "<<location<<" is not found in the list!!! Insertion not possible!!!\n";
```



Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

- **Step 1 -** Check whether list is **Empty** (**last** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **last**.
- **Step 4 -** Keep displaying **temp** → **data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp** → **data** with arrow pointing to **head** → **data**.

Sample Code: Display CLL

```
void diplayList(){
    struct Node* ptr;
    if(last==NULL){
     cout<<"List is empty !!\n";
     return;
    }
    ptr = last->next;
    do {
        cout<<ptr->d <<" ";
        ptr = ptr->next;
    } while(ptr != last->next);

}
```

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

Delete first

```cpp
//delete the first node
void deleteFirst(){

    Node* temp1, *temp2;
    temp1=temp2=NULL;
    //check if the list is emty. if so display valid message
    if(last==NULL)
        cout<<"List is empty. Deleting is no allowed !!\n";
    else if(last->next==last){ // check the list is having only one node
        temp1=last;
        last=NULL;
        delete temp1;
    }
    else{
        temp1=last->next;
        last->next=temp1->next;
        delete temp1;
    }
}
```

Delete last

```
//Delete the last node in the cll
void deleteLast(){

    Node* temp1, *temp2;
    temp2=NULL;
    temp1=last->next;
    if(last==NULL){
        cout<<"List is empty deleting is not allowed \n";
        return;
    }
    while(temp1!=last){ // traverse the list until the last node
        temp2=temp1;
        temp1=temp1->next;
    }
    if(temp2!=NULL){ //if list is having morethan one node temp2 should no be NULL
        temp2->next=temp1->next;
        last=temp2;
        delete temp1;
    }
    else {//if list is having only a single node
        last=NULL;
        delete temp1, temp2;
    }
}
```

Delete specific node

```
//Delete a specific node with value
void deleteNode(int value)
{
    Node* templ, *temp2;
    templ=temp2=NULL;
    if(last==NULL){
        cout<<"List is empty deleting is not allowed \n";
        return;
    }
    templ=last->next;
    do{// Searching the target node.
        if (templ ->d == value){
            if(temp2!=NULL)
                temp2 -> next = templ -> next;// Adjusting the links.
            else{
                deleteFirst(); return;
            }
            if (templ == last)// Checking for the last node.
                last = temp2;
            cout<<"The node with value "<<templ->d<<" is deleted successfully \n";
            delete templ;return ;
        }
        temp2=templ; templ = templ -> next;
    } while (templ != last -> next);
    cout<<"Node with value "<<value<<" is not found in the list!!! Deleting not possible!!!\n";
```

Linked Lists Benefits & Drawbacks

- Benefits
  - Easy to insert and delete in O(1) time (front and end)
  - Don't need to estimate total memory needed
- Drawbacks
  - Hard to search in less than O(n) time(e.g. binary search doesn't work)
  - Hard to jump to the middle
- Skip Lists and Self-organizing lists
  - Tries to address these drawbacks

Next Time!!

Ch6 – stack and queue