# 3. Chapter 3: Linked lists

A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

**Node**: A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.


**Linked List**: A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.


**Declaring a Linked list:**

In C++ language, a linked list can be implemented using structure and pointers.

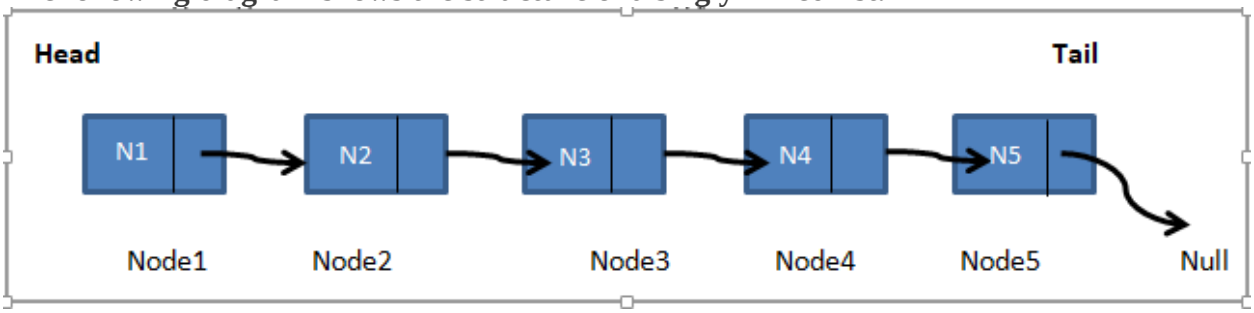
```
struct node{
int data;
struct node *next;
};
```

**Array vs Linked lists:** Arrays are simple and fast but we must specify their size at construction time. This has its own drawbacks. If you construct an array with space for n, tomorrow you may need n+1.Here comes a need for a more flexible system.

**Advantages of Linked Lists**: Flexible space use by dynamically allocating space for each element as needed. This implies that one need not know the size of the list in advance. Memory is efficiently utilized. Deletion and insertion is easy. **Sequential access** and **extra pointer** are drawbacks of linked list

## 3.1.Singly linked lists

**The following diagram shows the structure of a singly linked list.**



As shown above, the first node of the linked list is called “head” while the last node is called “Tail”. As we see, the last node of the linked list will have its next pointer as **null** since it will not have any memory address pointed to.

Since each node has a pointer to the next node, data items in the linked list need not be stored at contiguous locations. The nodes can be scattered in the memory. We can access the nodes anytime as each node will have an address of the next node.

We can add data items to the linked list as well as delete items from the list easily. Thus it is possible to *grow* or *shrink* the linked list dynamically. There is no upper limit on how many data items can be there in the linked list. So as long as memory is available, we can have as many data items added to the linked list.

Apart from easy insertion and deletion, the linked list also doesn’t waste memory space as we need not specify beforehand how many items we need in the linked list. The only space taken by linked list is for storing the pointer to the next node that adds a little overhead.

**Operations**

Just like the other data structures, we can perform various operations for the linked list as well. But unlike arrays, in which we can access the element using subscript directly even if it is somewhere in between, we cannot do the same random access with a linked list.

In order to access any node, we need to traverse the linked list from the start and only then we can access the desired node. Hence accessing the data randomly from the linked list proves to be expensive.

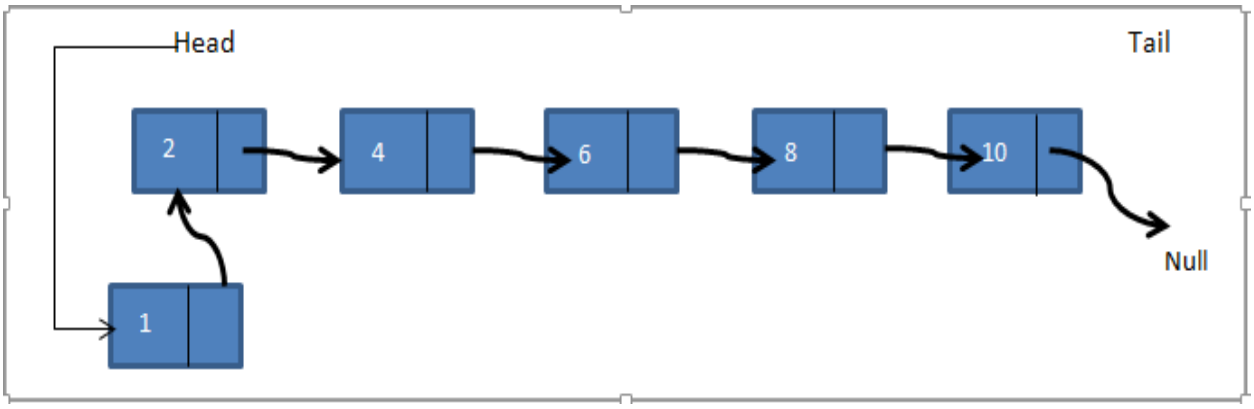
**Insertion**

Insertion operation of linked list adds an item to the linked list. Though it may sound simple, given the structure of the linked list, we know that whenever a data item is added to the linked list, we need to change the next pointers of the previous and next nodes of the new item that we have inserted.

**There are three positions in the linked list where a data item can be added.**

**1) at the beginning of the linked list**

A linked list is shown below 2->4->6->8->10. If we want to add a new node 1, as the first node of the list, then the head pointing to node 2 will now point to 1 and the next pointer of node 1 will have a memory address of node 2 as shown in the below figure.

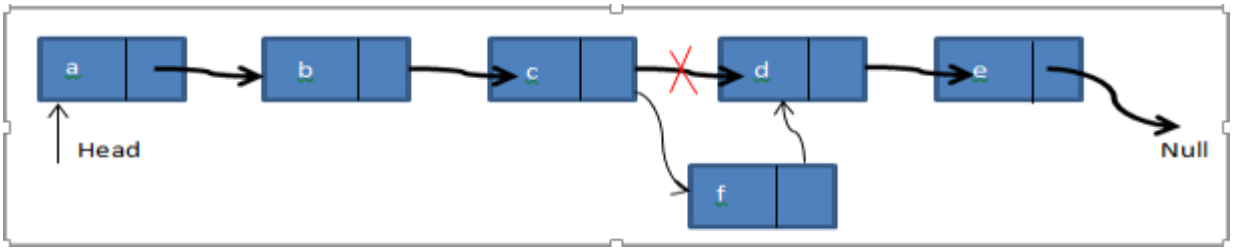



Thus the new linked list becomes 1->2->4->6->8->10.

```
struct Node
 {
 int data;
 struct Node *next;
 };
//insert a new node in front of the list
 void push(struct Node** head, int node_data)
 {
struct Node* newNode = new Node; // 1. Create and allocate node
newNode->data = node_data; // 2. Assign data to node
 newNode->next = (*head);//  3. Set next of new node as head
 (*head) = newNode; //4. Move the head to point to the new node
 }
```

**2) After the given Node**

Here, a node is given and we have to add a new node after the given node. In the below-linked list a->b->c->d ->e, if we want to add a node f after node c then the linked list will look as follows:

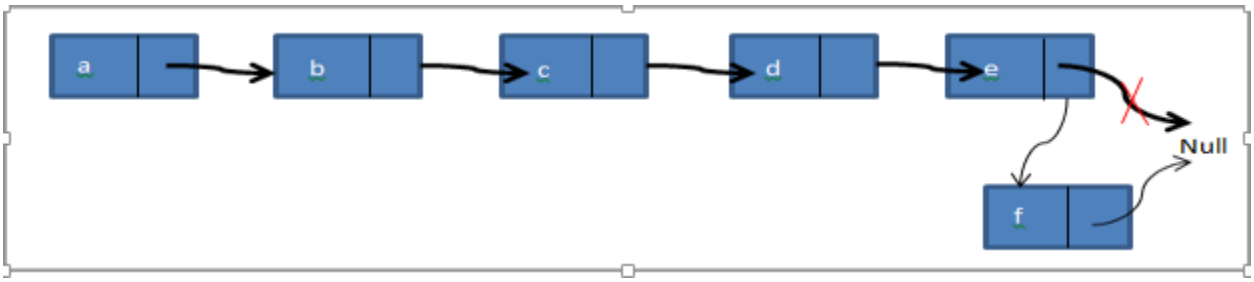


Thus in the above diagram, we check if the given node is present. If it's present, we create a new node f. Then we point the next pointer of node c to point to the new node f. The next pointer of the node f now points to node d.

```
//insert new node after a given node

 void insertAfter(struct Node* prev_node, int node_data)

 { //1. check if the given prev_node is NULL

 if (C == NULL)

 { cout<<"the given pr0evious node is required,cannot be NULL"; return; }

struct Node* f =new Node; //  2. Create and allocate new node

f->data = node_data; // 3. assign data to the node

  f->next = C->next; //4. Make next of new node as next of prev_node

C->next = f; //5. move the next of prev_node as new_node

 }
```

**3) At the end of the Linked List**

In the third case, we add a new node at the end of the linked list. Consider we have the same linked list a->b->c->d->e and we need to add a node f to the end of the list. The linked list will look as shown below after adding the node.




Thus we create a new node f. Then the tail pointer pointing to null is pointed to f and the next pointer of node f is pointed to null. We have implemented all three types of insert functions in the below C++ program.

```
/* insert new node at the end of the linked list */

void append(struct f** head, int node_data)
```

```
 {
 struct Node* f = new Node; // 1. Create and allocate node */
 struct Node *last = *head; // used in step 5
 f->data = node_data; // 2. assign data to the node
f->next = NULL; // 3. set next pointer of new node to null as its the last node
/* 4. if list is empty, new node becomes first node */
 if (*head == NULL)
 {
 *head = f;
 return;
 }
 /* 5. Else traverse till the last node */
 while (last->next != NULL)
 last = last->next;
 /* 6. Change the next of last node */
 last->next = f;
 return;
 }
```

**// display linked list contents**

```
 void displayList(struct Node *node)
 {
 //traverse the list to display each node
 while (node != NULL)
```

```
{
cout<<node->data<<"-->";
node = node->next;
}

if(node== NULL)
cout<<"null";
}
```

**2) Deletion**

Like insertion, deleting a node from a linked list also involves various positions from where the node can be deleted. We can delete the first node, last node or a random kth node from the linked list. After deletion, we need to adjust the next pointer and the other pointers in the linked list appropriately so as to keep the linked list intact.

```
struct Node {

 int data;

 struct Node* next;

 };

 //delete first node in the linked list

 Node* deleteFirstNode(struct Node* head)
 {
 if (head == NULL)
 return NULL;
 // Move the head pointer to the next node
 Node* tempNode = head;
 head = head->next;
 delete tempNode;
 return head;
 }
//delete last node from linked list
 Node* removeLastNode(struct Node* head)
 {
 if (head == NULL)
 return NULL;

 if (head->next == NULL) {
 delete head;
 return NULL;
```

```
}

// first find second last node
Node* second_last = head;
while (second_last->next->next != NULL)
second_last = second_last->next;

// Delete the last node
delete (second_last->next);

// set next of second_last to null
second_last->next = NULL;
return head;
}
```

**Applications**

As arrays and linked lists are both used to store items and are linear data structures, both these structures can be used in similar ways for most of the applications.

**Some of the applications for linked lists are as follows:**

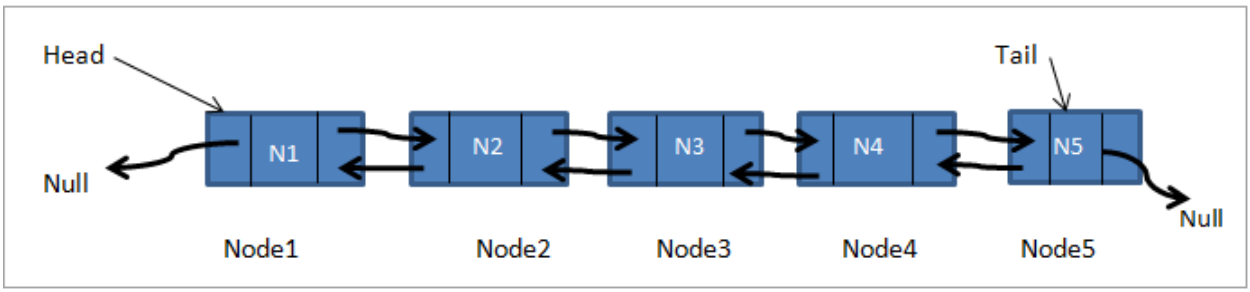
- A linked list can be used to implement stacks and queues.
- A linked list can also be used to implement graphs whenever we have to represent graphs as adjacency lists.
- A mathematical polynomial can be stored as a linked list.
- In the case of hashing technique, the buckets used in hashing are implemented using the linked lists.
- Whenever a program requires dynamic allocation of memory, we can use a linked list as linked lists work more efficiently in this case.

## 3.2.Doubly linked lists

A doubly linked list is a variation of the singly linked list. We are aware that the singly linked list is a collection of nodes with each node having a data part and a pointer pointing to the next node.

A doubly linked list is also a collection of nodes. Each node here consists of a data part and two pointers. One pointer points to the previous node while the second pointer points to the next node.

As in the singly linked list, the doubly linked list also has a head and a tail. The previous pointer of the head is set to NULL as this is the first node. The next pointer of the tail node is set to NULL as this is the last node.




In the above figure, we see that each node has two pointers, one pointing to the previous node and the other pointing to the next node. Only the first node (head) has its previous node set to null and the last node (tail) has its next pointer set to null.

As the doubly linked list contains two pointers i.e. previous and next, we can traverse it into the directions forward and backward. This is the main advantage of doubly linked list over the singly linked list.

**Declaration**

**In C++ style declaration, a node of the doubly linked list is represented as follows:**

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

**Insertion**

Insertion operation of the doubly linked list inserts a new node in the linked list. Depending on the position where the new node is to be inserted, we can have the following insert operations.

- **Insertion at front –** Inserts a new node as the first node.
- **Insertion at the end –** Inserts a new node at the end as the last node.
- **Insertion before a node –** Given a node, inserts a new node before this node.
- **Insertion after a node –** Given a node, inserts a new node after this node.

Deletion

Deletion operation deletes a node from a given position in the doubly linked list.

- **Deletion of the first node –** Deletes the first node in the list
- **Deletion of the last node –** Deletes the last node in the list.
- **Deletion of a node given the data –** Given the data, the operation matches the data with the node data in the linked list and deletes that node.

Traversal

Traversal is a technique of visiting each node in the linked list. In a doubly linked list, we have two types of traversals as we have two pointers with different directions in the doubly linked list.

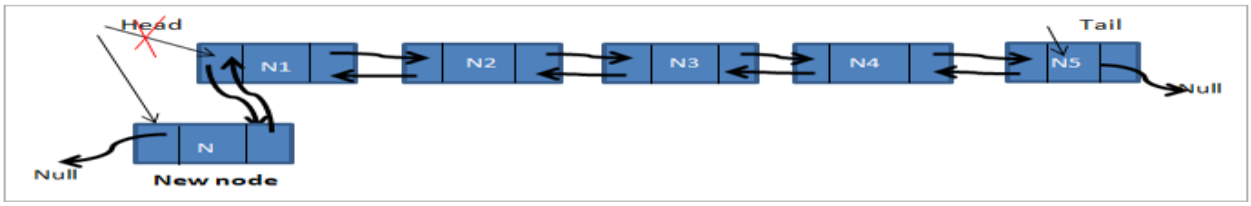
- **Forward traversal –** Traversal is done using the next pointer which is in the forward direction.
- **Backward traversal –** Traversal is done using the previous pointer which is the backward direction.

Reverse

This operation reverses the nodes in the doubly linked list so that the first node becomes the last node while the last node becomes the first node.

**Insertion**

**Insert a node at the front**

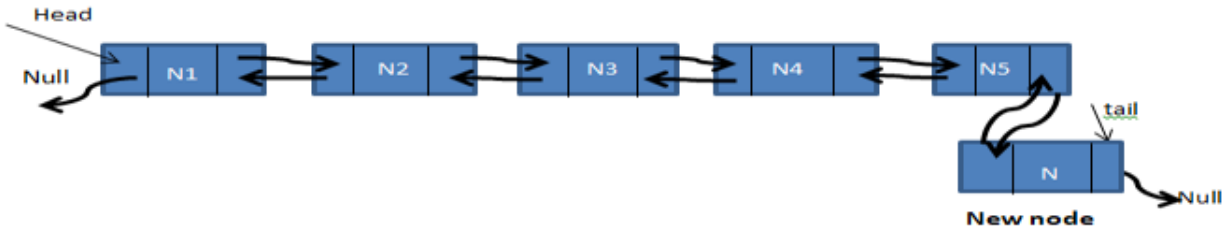



Insertion of a new node at the front of the list is shown above. As seen, the previous new node N is set to null. Head points to the new node. The next pointer of N now points to N1 and previous of N1 that was earlier pointing to Null now points to N.

```
// A doubly linked list node
 struct Node {
 int data;
 struct Node* next;
 struct Node* prev;
 };

 //inserts node at the front of the list
 void insert_front(struct Node** head, int new_data)
 {
 //allocate memory for new node
 struct Node* N = new Node;

 //assign data to new node
 N->data = new_data;

//new node is head and previous is null, since we are adding at the front
 N ->next = (*head);
 N ->prev = NULL;

 //previous of head is N
 if ((*head) != NULL)
 (*head)->prev = N;

 //head points to new node
```

```
(*head) = N;
 }
```

**Insert node at the end**

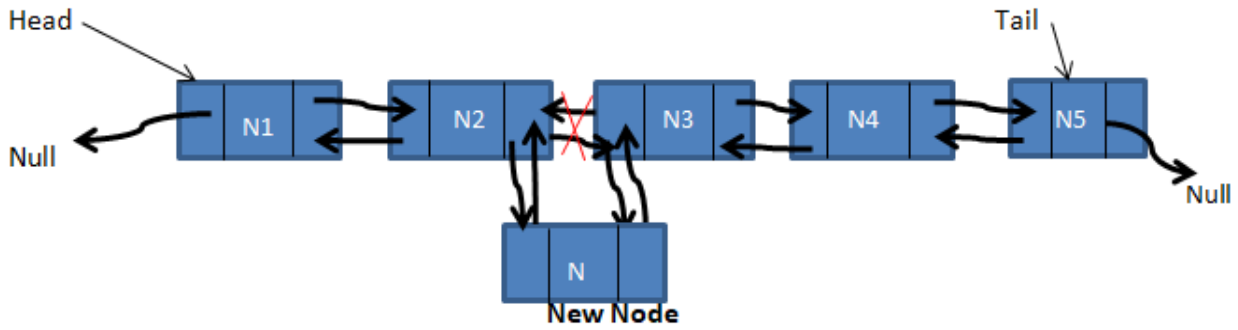



Inserting node at the end of the doubly linked list is achieved by pointing the next pointer of new node N to null. The previous pointer of N is pointed to N5. The 'Next' pointer of N5 is pointed to N.

```
//insert a new node at the end of the list
 void insert_end(struct Node** head, int new_data)

 {
 //allocate memory for node
 struct Node* N = new Node;

 struct Node* last = *head; //set last node value to head

 //set data for new node
 N ->data = new_data;

 //new node is the last node , so set next of new node to null
 newNode->next = NULL;

 //check if list is empty, if yes make new node the head of list
 if (*head == NULL) {
 N ->prev = NULL;
 *head = N;
 return;
 }

 //otherwise traverse the list to go to last node
 while (last->next != NULL)
 last = last->next;

 //set next of last to new node
 last->next = N;

 //set last to prev of new node
```

```
N ->prev = last;
return;
}
```

**Insert node before/after given node**




As shown in the above diagram, when we have to add a node before or after a particular node, we change the previous and next pointers of the before and after nodes so as to appropriately point to the new node. Also, the new node pointers are appropriately pointed to the existing nodes.

```
/* Given a node as prev_node, insert a new node after the given node */
 void insert_After(struct Node* prev_node, int new_data)
 {
 //check if prev node is null
 if (prev_node == NULL) {
 cout<<"Previous node is required , it cannot be NULL";
 return;
 }
 //allocate memory for new node
 struct Node* newNode = new Node;

 //assign data to new node
 newNode->data = new_data;

 //set next of newnode to next of prev node
 newNode->next = prev_node->next;

 //set next of prev node to newnode
 prev_node->next = newNode;

 //now set prev of newnode to prev node
 newNode->prev = prev_node;

 //set prev of new node's next to newnode
 if (newNode->next != NULL)
 newNode->next->prev = newNode;
 }
```

**Deletion**

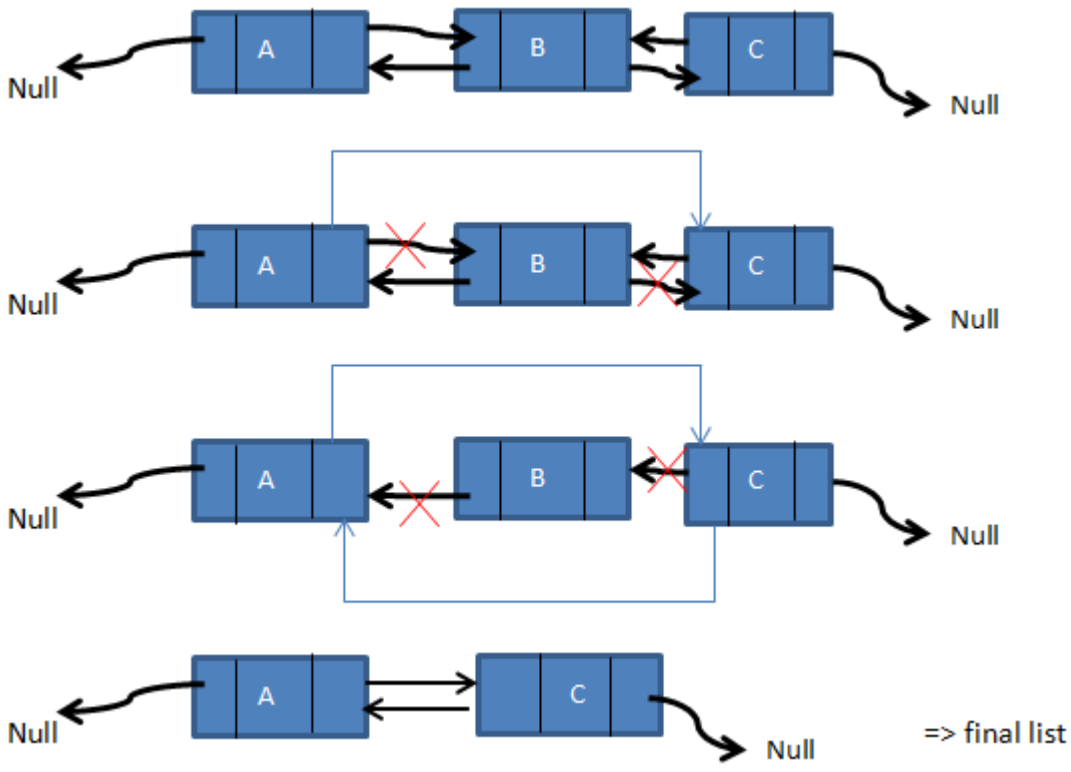
A node can be deleted from a doubly linked list from any position like from the front, end or any other given position or given data.

Algorithm

1. If node to be deleted is head node, then change the head pointer to point to next current head
2. Set next pointer of previous to node to be deleted, if previous exists
3. Set previous of next to node to be deleted, if next exists

When deleting a node from the doubly linked list, we first reposition the pointer pointing to that particular node so that the previous and after nodes do not have any connection to the node to be deleted. We can then easily delete the node.

Consider the following doubly linked list with three nodes A, B, C. Let us consider that we need to delete the node B.



As shown in the above series of the diagram, we have demonstrated the deletion of node B from the given linked list. The sequence of operation remains the same even if the node is first or last. The only care that should be taken is that if in case the first node is deleted, the second node's previous pointer will be set to null.

Similarly, when the last node is deleted, the next pointer of the previous node will be set to null. If in between nodes are deleted, then the sequence will be as above.

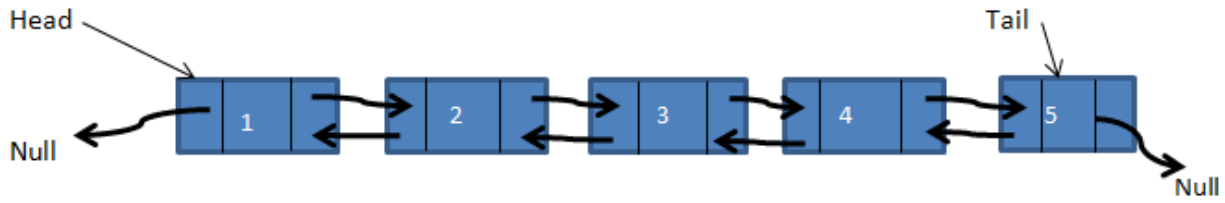
We leave the program to delete a node from a doubly linked list. Note that the implementation will be on the lines of the insertion implementation.

```
/* Function to delete a node in a Doubly Linked List.
head --> pointer to head node pointer.
del --> pointer to node to be deleted. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
        /* base case */
        if (*head_ref == NULL || del == NULL)
                return;

        /* If node to be deleted is head node */
        if (*head == del)
                *head = del->next;

        /* Change next only if node to be deleted is NOT
        the last node */
        if (del->next != NULL)
                del->next->prev = del->prev;

        /* Change prev only if node to be deleted is NOT
        the first node */
        if (del->prev != NULL)
                del->prev->next = del->next;

        /* Finally, free the memory occupied by del*/
        free(del);
}

/* Function to delete the node at the given position
in the doubly linked list */
void deleteNodeAtGivenPos(struct Node** head, int n)
{
```

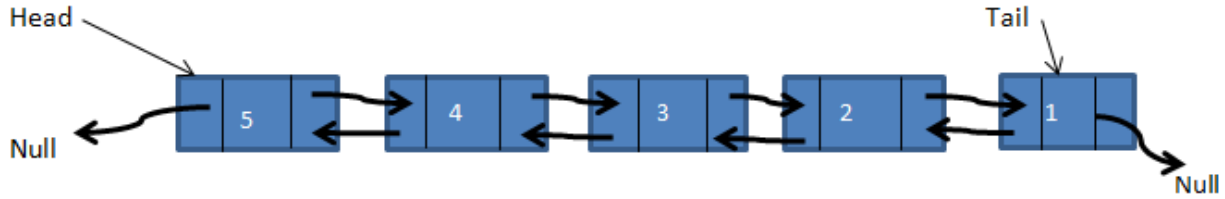
```
        /* if list in NULL or invalid position is given */
        if (*head == NULL || n <= 0)
                return;

        struct Node* current = *head;
        int i;

        /* traverse up to the node at position 'n' from
        the beginning */
        for (int i = 1; current != NULL && i < n; i++)
                current = current->next;

        /* if 'n' is greater than the number of nodes
        in the doubly linked list */
        if (current == NULL)
                return;

        /* delete the node pointed to by 'current' */
        deleteNode(head, current);
}
```

**Reverse Doubly Linked List**

Reversing a doubly linked list is an important operation. In this, we simply swap the previous and next pointers of all the nodes and also swap the head and tail pointers.




After reversing the linked list the resultant doubly linked list is shown below.




**write C++ Program to convert head to tail?**

**Advantages:**

- The doubly linked list can be traversed in forward as well as backward directions, unlike singly linked list which can be traversed in the forward direction only.
- Delete operation in a doubly-linked list is more efficient when compared to singly list when a given node is given. In a singly linked list, as we need a previous node to delete the given node, sometimes we need to traverse the list to find the previous node. This hits the performance.
- Insertion operation can be done easily in a doubly linked list when compared to the singly linked list.

**Disadvantages:**

- As the doubly linked list contains one more extra pointer i.e. previous, the memory space taken up by the doubly linked list is larger when compared to the singly linked list.
- Since two pointers are present i.e. previous and next, all the operations performed on the doubly linked list have to take care of these pointers and maintain them thereby resulting in a performance bottleneck.

**Applications of Doubly Linked List**

A doubly linked list can be applied in various real-life scenarios and applications as discussed below.

- A Deck of cards in a game is a classic example of a doubly linked list. Given that each card in a deck has the previous card and next card arranged sequentially, this deck of cards can be easily represented using a doubly linked list.
- Browser history/cache – The browser cache has a collection of URLs and can be navigated using the forward and back buttons is another good example that can be represented as a doubly linked list.
- Most recently used (MRU) also can be represented as a doubly linked list.
- Other data structures like Stacks, hash table, the binary tree can also be constructed or programmed using a doubly linked list.

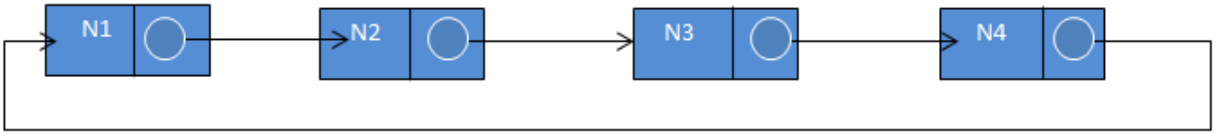
## 3.3.Circular lists

A circular linked list is a variation of the linked list. It is a linked list whose nodes are connected in such a way that it forms a circle.

In the circular linked list, the next pointer of the last node is not set to null but it contains the address of the first node thus forming a circle.

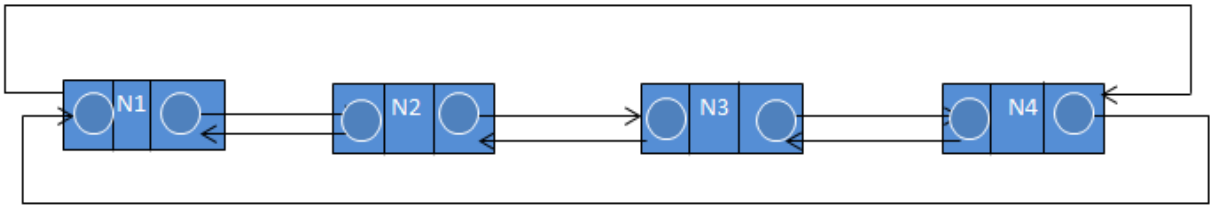
Circular Linked List

**The arrangement shown below is for a singly linked list.**




A circular linked list can be a singly linked list or a doubly linked list. In a doubly circular linked list, the previous pointer of the first node is connected to the last node while the next pointer of the last node is connected to the first node.

**Its representation is shown below.**




**Declaration**

**We can declare a node in a circular linked list as any other node as shown below:**

```
struct Node
{
   int data;
   struct Node *next;
};
```

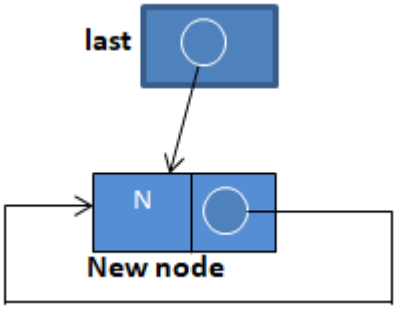
In order to implement the circular linked list, we maintain an external pointer "last" that points to the last node in the circular linked list. Hence last->next will point to the first node in the linked list.

By doing this we ensure that when we insert a new node at the beginning or at the end of the list, we need not traverse the entire list. This is because the last points to the last node while last->next points to the first node. This wouldn't have been possible if we had pointed the external pointer to the first node.

**Insertion**

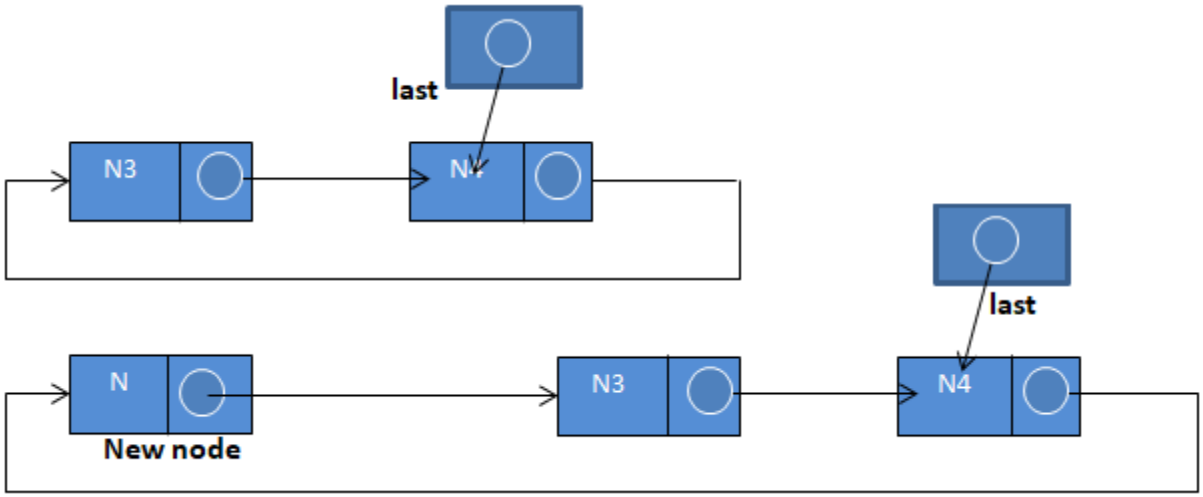
We can insert a node in a circular linked list either as a first node (empty list), in the beginning, in the end, or in between the other nodes. Let us see each of these insertion operations using a pictorial representation below.

**#1) Insert in an empty list**




When there are no nodes in circular list and the list is empty, the last pointer is null, then we insert a new node N by pointing the last pointer to the node N as shown above. The next pointer of N will point to the node N itself as there is only one node. Thus N becomes the first as well as last node in the list.

**#2) Insert at the beginning of the list**



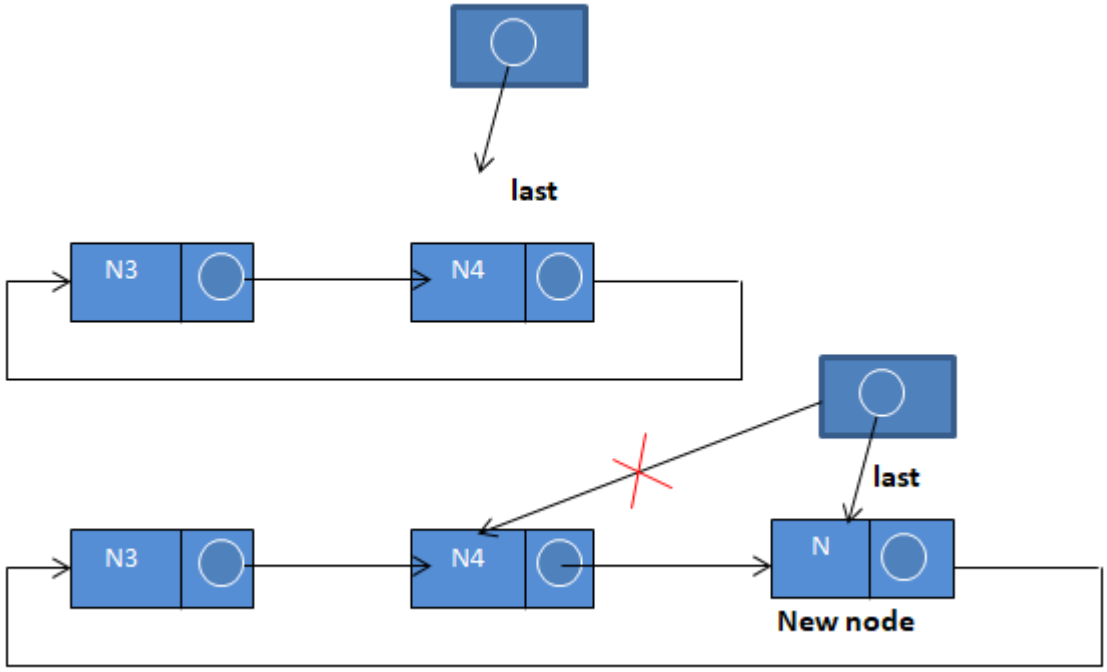

As shown in the above representation, when we add a node at the beginning of the list, the next pointer of the last node points to the new node N thereby making it a first node.

***N->next = last->next***
***Last->next = N***
*//insert new node at the beginning of the list*

```
struct Node *insertAtBegin(struct Node *last, int new_data)

{

//if list is empty then add the node by calling insertInEmpty

if (last == NULL)

return insertInEmpty(last, new_data);

//else create a new node

struct Node *temp = new Node;

//set new data to node

temp -> data = new_data;

temp -> next = last -> next;

last -> next = temp;

return last;
```

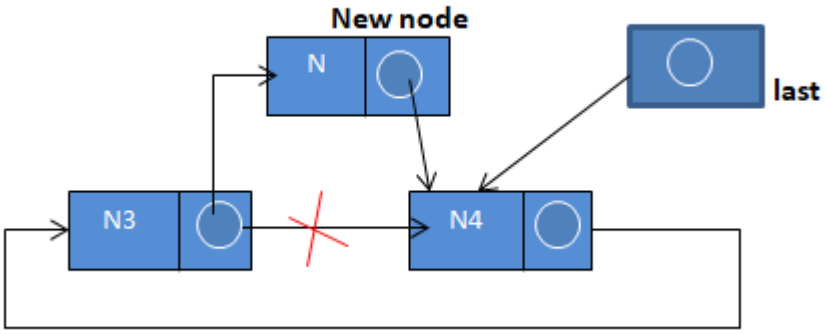
*}*

**3) Insert at the end of the list**




**To insert a new node at the end of the list, we follow these steps:**
***N-> next = last ->next;***
***last ->next = N***
***last = N***

**4) Insert in between the list**




Suppose we need to insert a new node N between N3 and N4, we first need to traverse the list and locate the node after which the new node is to be inserted, in this case, its N3.

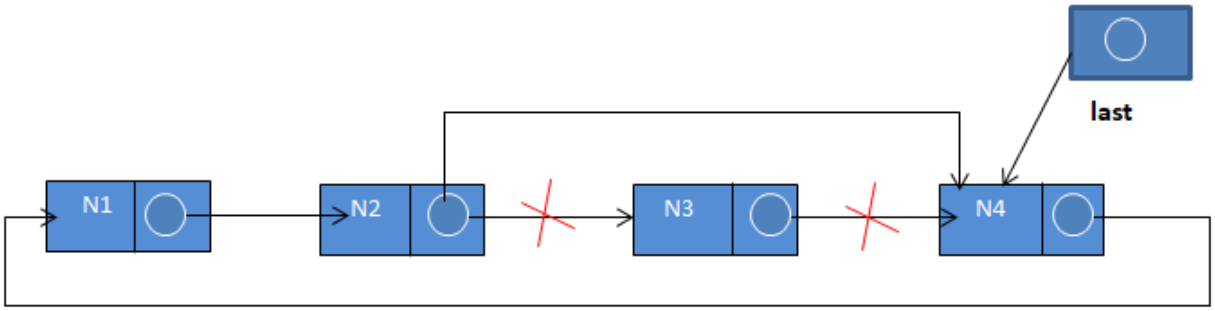
**After the node is located, we perform the following steps.**
***N -> next = N3 -> next;***
***N3 -> next = N***
This inserts a new node N after N3.

Deletion

The deletion operation of the circular linked list involves locating the node that is to be deleted and then freeing its memory.

For this we maintain two additional pointers curr and prev and then traverse the list to locate the node. The given node to be deleted can be the first node, the last node or the node in between. Depending on the location we set the curr and prev pointers and then delete the curr node.

**A pictorial representation of the deletion operation is shown below.**




```
//delete the node from the list

 void deleteNode(Node** head, int key)

 {

 // If linked list is empty retun

 if (*head == NULL)

 return;

 if((*head)->data==key && (*head)->next==*head) {

 free(*head);

 *head=NULL;

 }

 Node *last=*head,*d;
```

```
// If key is the head
if((*head)->data==key) {
while(last->next!=*head) // Find the last node of the list
last=last->next;
// point last node to next of head or second node of the list
last->next=(*head)->next;
free(*head);
*head=last->next;
}
// end of list is reached or node to be deleted not there in the list
while(last->next!=*head&&last->next->data!=key) {
last=last->next;
}
// node to be deleted is found, so free the memory and display the list
if(last->next->data==key) {
d=last->next;
last->next=d->next;
cout<<"The node with data "<<key<<" deleted from the list"<<endl;
free(d);
cout<<endl;
cout<<"Circular linked list after deleting "<<key<<" is as follows:"
<<endl;
```

```
traverseList(last);

}

else

cout<<"The node with data "<< key << " not found in the list"<<endl;

}
```

**Traversal**

Traversal is a technique of visiting each and every node. In linear linked lists like singly linked list and doubly linked lists, traversal is easy as we visit each node and stop when NULL is encountered.

However, this is not possible in a circular linked list. In a circular linked list, we start from the next of the last node which is the first node and traverse each node. We stop when we once again reach the first node.

**Advantages:**

- We can go to any node and traverse from any node. We just need to stop when we visit the same node again.
- As the last node points to the first node, going to the first node from the last node just takes a single step.

**Disadvantages:**

- Reversing a circular linked list is cumbersome.
- As the nodes are connected to form a circle, there is no proper marking for beginning or end for the list. Hence, it is difficult to find the end of the list or loop control. If not taken care, an implementation might end up in an infinite loop.
- We cannot go back to the previous node in a single step. We have to traverse the entire list from first.

**Applications of Circular Linked List**

- Real-time application of circular linked list can be a multi-programming operating system wherein it schedules multiple programs. Each program is given a dedicated timestamp to execute after which the resources are passed to another program. This goes on continuously in a cycle. This representation can be efficiently achieved using a circular linked list.
- Games that are played with multiple players can also be represented using a circular linked list in which each player is a node that is given a chance to play.
- We can use a circular linked list to represent a circular queue. By doing this, we can remove the two pointers front and rear that is used for the queue. Instead, we can use only one pointer.

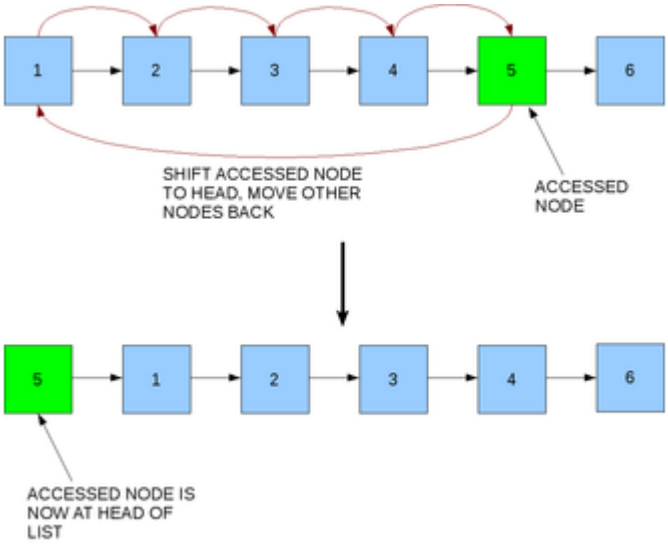
### 3.4.Self-organizing lists

Self-Organizing linked list is a list that re-organizes or re-arranges itself for better performance. In a simple list, an item to be searched is looked for in a sequential manner which gives the time

complexity of O(n). But in real scenario not all the items are searched frequently and most of the time only few items are searched multiple times.
So, a self-organizing list uses this property (also known as locality of reference) that brings the most frequent used items at the head of the list. This increases the probability of finding the item at the start of the list and those elements which are rarely used are pushed to the back of the list.

Move to Front Method (MTF)

This technique moves the element which is accessed to the head of the list. This has the advantage of being easily implemented and requiring no extra memory. This heuristic also adapts quickly to rapid changes in the query distribution. On the other hand, this method may prioritize infrequently accessed nodes—for example, if an uncommon node is accessed even once, it is moved to the head of the list and given maximum priority even if it is not going to be accessed frequently in the future. These 'over rewarded' nodes destroy the optimal ordering of the list and lead to slower access times for commonly accessed elements. Another disadvantage is that this method may become too flexible leading to access patterns that change too rapidly. This means that due to the very short memories of access patterns even an optimal arrangement of the list can be disturbed immediately by accessing an infrequent node in the list.

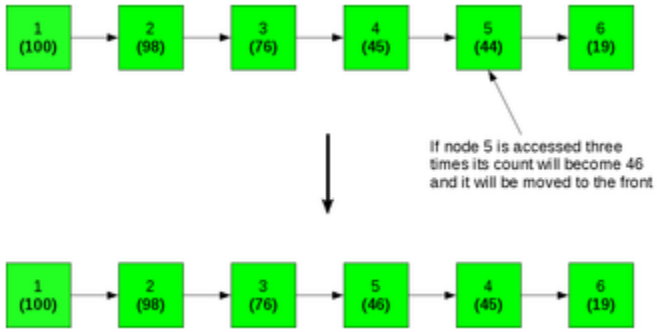



If the 5th node is selected, it is moved to the front

At the t-th item selection:
**if** item i is selected:
move item i to head of the list

## Count Method

In this technique, the number of times each node was searched for is counted i.e. every node keeps a separate counter variable which is incremented every time it is called. The nodes are then rearranged according to decreasing count. Thus, the nodes of highest count i.e. most frequently accessed are kept at the head of the list. The primary advantage of this technique is that it generally is more realistic in representing the actual access pattern. However, there is an added memory requirement, that of maintaining a counter variable for each node in the list. Also, this technique does not adapt quickly to rapid changes in the access patterns. For example: if the count of the head element say A is 100 and for any node after it say B is 40, then even if B becomes the new most commonly accessed element, it must still be accessed at least (100 - 40 = 60) times before it can become the head element and thus make the list ordering optimal.

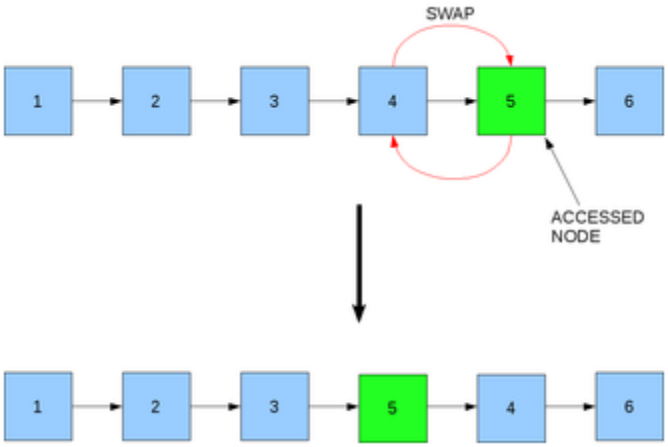



If the 5th node in the list is searched for twice, it will be swapped with the 4th

```
init: count(i) = 0 for each item i
At t-th item selection:
  if item i is searched:
     count(i) = count(i) + 1
     rearrange items based on count
```

## Transpose Method

This technique involves swapping an accessed node with its predecessor. Therefore, if any node is accessed, it is swapped with the node in front unless it is the head node, thereby increasing its priority. This algorithm is again easy to implement and space efficient and is more likely to keep frequently accessed nodes at the front of the list. However, the transpose method is more cautious. i.e. it will take many accesses to move the element to the head of the list. This method also does not allow for rapid response to changes in the query distributions on the nodes in the list.




If the 5th node in the list is selected, it will be swapped with the 4th

```
At the t-th item selection:
    if item i is selected:
        if i is not the head of list:
                swap item i with item (i - 1)
```

### 3.5.Skip lists(Reading Assignment)