Linked List

*A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:*



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

**Why Linked List?**

Arrays can be used to store linear data of similar types, but arrays have the following limitations:

* ***The size of the arrays is fixed****: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.*
* ***Insertion of a new element / Deletion of a existing element in an array of elements is expensive:****The room has to be created for the new elements and to create room existing elements have to be shifted but in Linked list if we have the head node then we can traverse to any node through it and insert new node at the required position.*

***Example:****In a system, if we maintain a sorted list of IDs in an array id[] = [1000, 1010, 1050, 2000, 2040].   
If we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).*

*Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved due to this so much work is being done which affects the efficiency of the code.*

**Advantages of Linked Lists over arrays:**

* Dynamic Array.
* Ease of Insertion/Deletion.
* Insertion at the beginning is a constant time operation and takes O(1) time, as compared to arrays where inserting an element at the beginning takes O(n) time,where n is the number of elements in the array.

**Drawbacks of Linked Lists:**

* Random access is not allowed. We have to access elements sequentially starting from the first node(head node). So we cannot do a [binary search with linked lists](https://www.geeksforgeeks.org/binary-search-on-singly-linked-list/) efficiently with its default implementation.
* Extra memory space for a pointer is required with each element of the list.
* Not cache-friendly. Since array elements are contiguous locations, there is the locality of reference which is not there in the case of linked lists.
* It takes a lot of time in traversing and changing the pointers.
* Reverse traversing is not possible in singly linked lists.
* It will be confusing when we work with pointers.
* Direct access to an element is not possible in a linked list as in an array by index.
* Searching for an element is costly and requires O(n) time complexity.
* Sorting of linked lists is very complex and costly.
* Appending an element to a linked list is a costly operation, and takes O(n) time, where n is the number of elements in the linked list, as compared to arrays that take O(1) time.

**Types of Linked Lists:**

* [**Simple Linked List**](https://www.geeksforgeeks.org/data-structures/linked-list/singly-linked-list/) – In this type of linked list, one can move or traverse the linked list in only one direction. where the next pointer of each node points to other nodes but the next pointer of the last node points to NULL. It is also called *“***Singly Linked List”**.
* [**Doubly Linked List**](https://www.geeksforgeeks.org/doubly-linked-list/) – In this type of linked list, one can move or traverse the linked list in both directions (Forward and Backward)
* [**Circular Linked List**](https://www.geeksforgeeks.org/circular-linked-list/) – In this type of linked list, the last node of the linked list contains the link of the first/head node of the linked list in its next pointer.
* [**Doubly Circular Linked List**](https://www.geeksforgeeks.org/doubly-circular-linked-list-set-1-introduction-and-insertion/) – A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node.
* [**Header Linked List**](https://www.geeksforgeeks.org/header-linked-list-in-c/) – A header linked list is a special type of linked list that contains a header node at the beginning of the list.

**Representation of Singly Linked Lists:**

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head of the linked list. If the linked list is empty, then the value of the head points to NULL.

Each node in a list consists of at least two parts:

* A Data Item (we can store integers, strings, or any type of data).
* Pointer (Or Reference) to the next node (connects one node to another) or An address of another node.
* **class** Node {
* **public**:
* **int** data;
* Node\* next;
* };

## ****Traversal of a Linked List****

In the previous program, we created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general-purpose function printList() that prints any given list.

// A simple C++ program for

// traversal of a linked list

#include <bits/stdc++.h>

**using** **namespace** std;

**class** Node {

**public**:

**int** data;

    Node\* next;

};

// This function prints contents of linked list

// starting from the given node

**void** printList(Node\* n)

{

**while** (n != NULL) {

        cout << n->data << " ";

        n = n->next;

    }

}

// Driver's code

**int** main()

{

    Node\* head = NULL;

    Node\* second = NULL;

    Node\* third = NULL;

    // allocate 3 nodes in the heap

    head = **new** Node();

    second = **new** Node();

    third = **new** Node();

    head->data = 1; // assign data in first node

    head->next = second; // Link first node with second

    second->data = 2; // assign data to second node

    second->next = third;

    third->data = 3; // assign data to third node

    third->next = NULL;

    // Function call

    printList(head);

**return** 0;

}

**Time Complexity:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Time Complexity** | **Worst Case** | **Average Case** |  |  |  |  |  |  |  |  | **Space complexity** |
| **Search** | O(n) | O(n) |  |  |  |  |  |  |  |  | O(1) |
| **Insertion** | O(1) | O(1) |  |  |  |  |  |  |  |  | O(1) |
| **Deletion** | O(1) | O(1) |  |  |  |  |  |  |  |  | O(1) |

* *Search* is O(n) because as data is not stored in contiguous memory locations so we have to traverse one by one.
* Insertion and Deletion are O(1) because we have to just link new nodes for Insertion with the previous and next node and dislink exist nodes for deletion from the previous and next nodes without any traversal.
* **Auxiliary Space:**O(N) [To store dynamic memory]

## Applications of Linked Lists

* ● Linked Lists can be used to implement useful data structures like stacks and queues.
* ● Linked Lists can be used to implement hash tables, each bucket of the hash table can be a linked list.
* ● Linked Lists can be used to implement graphs (Adjacency List representation of graph).
* ● Linked Lists can be used in a refined way in implementing different file systems in one form or another.

# Introduction and Insertion in a Doubly Linked List

A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.



Following is a representation of a DLL node:

/ Node of a doubly linked list

**class** Node {

**public**:

**int** data;

    // Pointer to next node in DLL

    Node\* next;

    // Pointer to previous node in DLL

    Node\* prev;

};

## ****Advantages of DLL over the singly linked list:****

* A DLL can be traversed in both forward and backward directions.
* The delete operation in DLL is more efficient if a pointer to the node to be deleted is given. (already have prev pointer here)
* We can quickly insert a new node before a given node.
* In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

## ****Disadvantages of DLL over the singly linked list:****

* Every node of DLL Requires extra space for a previous pointer. It is possible to implement DLL with a single pointer though (See [this](https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/)and [this](https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/)).
* All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

## Applications of DLL:

* It is used by web browsers for backward and forward navigation of web pages
* LRU ( Least Recently Used ) / MRU ( Most Recently Used ) Cache are constructed using Doubly Linked Lists.
* Used by various applications to maintain undo and redo functionalities.
* In Operating Systems, a doubly linked list is maintained by thread scheduler to keep track of processes that are being executed at that time.

(see implementation of dll using one pointer)

**Insertion in DLL:**

A node can be added in four ways:

* At the front of the DLL
* // Time Complexity: O(1)
* // Auxiliary Space: O(1)
* void InsertatHead(int data,node \* &head){
* node \*newnode = new node(data);
* if(head==NULL){
* head=newnode;
* return;
* }
* newnode->next = head;
* head->prev = newnode;
* head = newnode ;
* }
* After a given node.

// Time Complexity: O(1)

// Auxiliary Space: O(1)

void InsertAfterNode(int data,node \* &head,node \*givenNode){

    node \*newnode = new node(data);

    if(head==NULL){

        head=newnode;

        return;

    }

    newnode->next=givenNode->next;

    givenNode->next=newnode;

    newnode->prev = givenNode;

    if(newnode->next->prev!=NULL){

        newnode->next->prev=newnode;

    }

    return;

}

* At the end of the DLL
* // Time Complexity: O(n)
* // Auxiliary Space: O(1)
* void InsertNodeAtTail(int data,node \* &head){
* node \*newnode = new node(data);
* if(head==NULL){
* head=newnode;
* return;
* }
* node \*temp = head;
* while(temp->next!=NULL){
* temp = temp->next;
* }
* newnode->prev = temp;
* temp->next = newnode;
* return;
* }
* Before a given node.
* // 4) Add a node before the given node
* // Time Complexity: O(1)
* // Auxiliary Space: O(1)
* void InsertBeforeNode(int data,node\* &head,node \*givenNode){
* node \*newnode = new node(data);
* if(givenNode==NULL){
* cout<<"Node cannot be inserted before a NULL";
* return;
* }
* newnode->prev=givenNode->prev;
* givenNode->prev=newnode;
* newnode->next=givenNode;
* if(newnode->prev!=NULL){
* newnode->prev->next = newnode;
* }else{
* head = newnode;
* return;
* }
* return;
* }

Reverse a DLL;

node\* reverseDLL(node \* head)

{

    node \*pre = NULL;

    node \*curr = head;

    while(curr!=NULL){

        node \*temp=curr->next;

        curr->next = pre;

        curr->prev=temp;

        pre=curr;

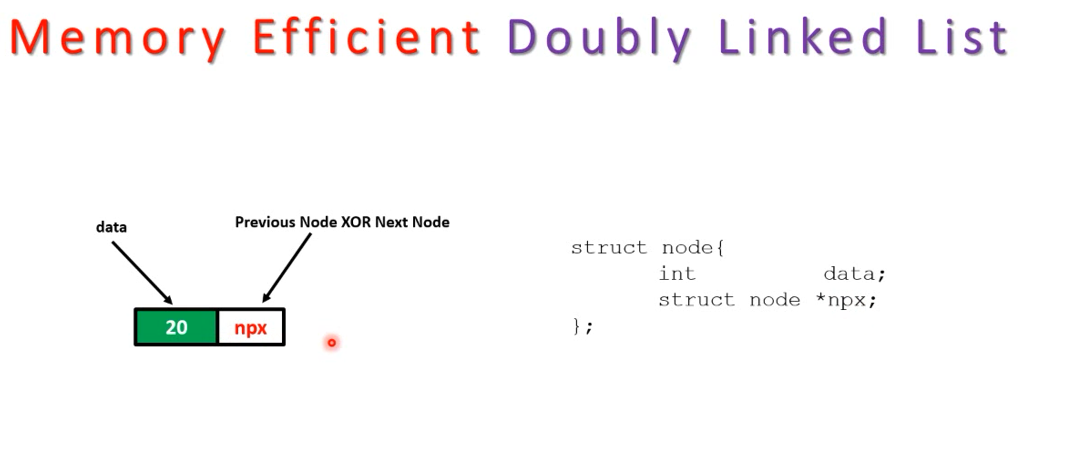
        curr=temp;

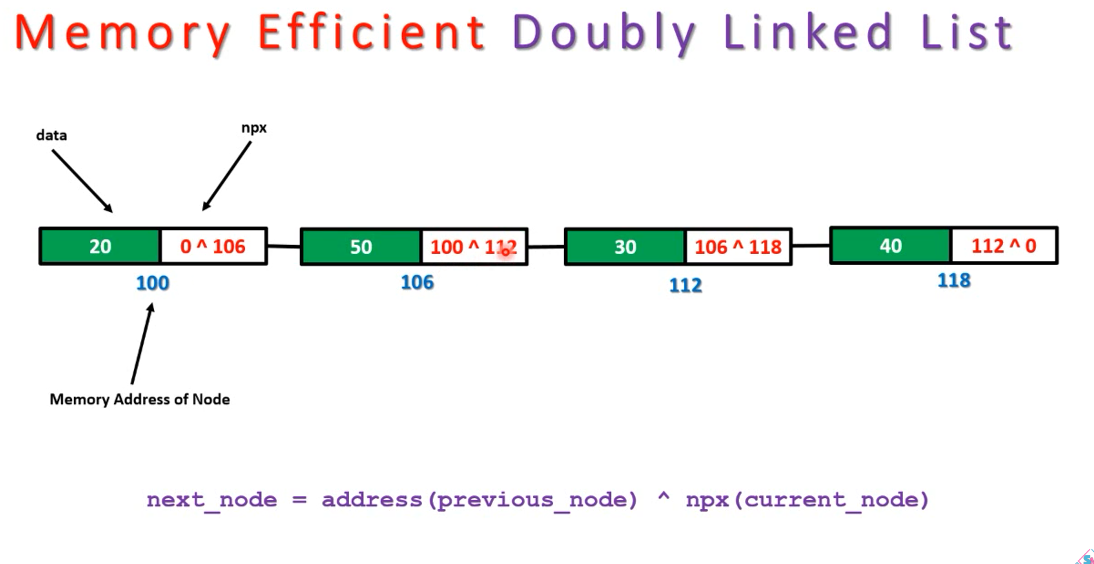
    }

    return pre;

}

We can implement DLL using one pointer.





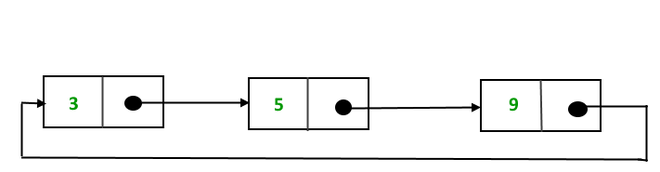
# Introduction to Circular Linked List

*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.*

Circular Linked List

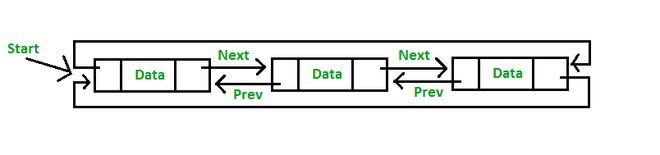
**There are generally two types of circular linked lists:**

* **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.

### 1. [**Insertion in the circular linked list:**](https://www.geeksforgeeks.org/circular-singly-linked-list-insertion/)

A node can be added in three ways:

1. Insertion at the beginning of the list

//insert at begining

// Time complexity: O(n)    //if last node is given instead of head then TC->O(1)

// Auxiliary Space: O(1)

void InsertatFirst(int x,node \*&head){

    node \*newnode = new node(x);

    if(head==NULL){

        head = newnode;

        head->next = head;

        return;

    }

    node \*temp = head;

    while(temp->next!=head){

        temp = temp->next;

    }

    temp->next = newnode;

    newnode->next = head;

    head = newnode;

}

1. Insertion at the end of the list

// Time Complexity: O(n)  //if last node is given instead of head then TC->O(1)

// Auxiliary Space: O(1)

void InsertatLast(int x,node \*&head){

    node \*newnode = new node(x);

    node \*temp = head;

    if(head==NULL){

        head = newnode;

        head->next = head;

        return;

    }

    while(temp->next!=head){

        temp = temp->next;

    }

    newnode->next = head;

    temp->next = newnode;

}

* 1. Insertion in between the nodes

//insert at between

// Time Complexity: O(n)

// Auxiliary Space: O(1)

void InsertInBetween(int x,int pos,node \*&head){

    node \*newnode = new node(x);

    node \*temp = head;

    int c=1; //1 based indexing

    while(temp->next!=head && c!=pos-1){

        temp = temp->next;

        c++;

    }

    newnode->next = temp->next;

    temp->next=newnode;

}

**3) Delete any node from the circular linked list**

// Time Complexity: O(N), Worst case occurs when the element to be deleted is the last element and we need to move through the whole list.

// Auxiliary Space: O(1), As constant extra space is used.

void Deletion(int pos,node \*&head){

    if(pos==1){

        deleteAtHead(pos,head);

        return;

    }

    node \*temp = head;

    int count=1;

    while(count!=pos-1){

        temp = temp->next;

        count++;

    }

    node \*todelete = temp->next;

    temp->next = temp->next->next;

    delete todelete;

}

## ****Advantages of Circular Linked Lists:****

* Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
* Useful for implementation of a queue. Unlike [this](https://www.geeksforgeeks.org/queue-linked-list-implementation/)implementation, we don’t need to maintain two pointers for front and rear if we use a circular linked list. We can maintain a pointer to the last inserted node and the front can always be obtained as next of last.
* Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
* Circular Doubly Linked Lists are used for the implementation of advanced data structures like the [Fibonacci Heap](http://en.wikipedia.org/wiki/Fibonacci_heap).

## Disadvantages of circular linked list:

* Compared to singly linked lists, circular lists are more complex.
* Reversing a circular list is more complicated than singly or doubly reversing a circular list.
* It is possible for the code to go into an infinite loop if it is not handled carefully.
* It is harder to find the end of the list and control the loop.

## Applications of circular linked lists:

* Multiplayer games use this to give each player a chance to play.
* A circular linked list can be used to organize multiple running applications on an operating system. These applications are iterated over by the OS.

## Why circular linked list?

* A node always points to another node, so NULL assignment is not necessary.
* Any node can be set as the starting point.
* Nodes are traversed quickly from the first to the last.