**Introduction to Circular Linked List**

## What is Circular linked list?

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

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

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

*Representation of Circular singly linked list*

* **Circular Doubly linked list:**Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.

*Representation of circular doubly linked list*

## Representation of circular linked list:

Circular linked lists are similar to single Linked Lists with the exception of connecting the last node to the first node.

Node representation of a Circular Linked List:

// Class Node, similar to the linked list

class Node{

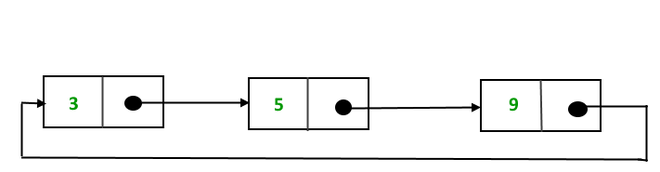
    int value;

  // Points to the next node.

    Node next;

}

Example of Circular singly linked list:



*Example of  circular linked list*

The above  Circular singly linked list can be represented as:

// Initialize the Nodes.

Node one = new Node(3);

Node two = new Node(5);

Node three = new Node(9);

// Connect nodes

one.next = two;

two.next = three;

three.next = one;

**Explanation:** In the above program one, two, and three are the node with values 3, 5, and 9 respectively which are connected in a circular manner as:

* **For Node One:**The Next pointer stores the address of Node two.
* **For Node Two:**The Next stores the address of Node three
* **For Node Three:** TheNext points to node one.

## Operations on the circular linked list:

We can do some operations on the circular linked list similar to the singly linked list which are:

1. Insertion
2. Deletion

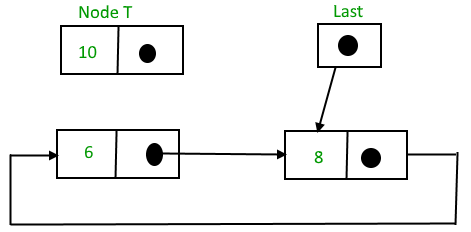
### 1. [**Insertion in the circular linked list:**](https://www.geeksforgeeks.org/circular-singly-linked-list-insertion/)

A node can be added in three ways:

1. Insertion at the beginning of the list
2. Insertion at the end of the list
3. Insertion in between the nodes

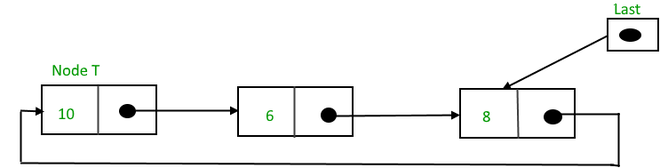
1) **Insertion at the beginning of the list:**To insert a node at the beginning of the list, follow these steps:

* Create a node, say T.
* Make T -> next = last -> next.
* last -> next = T.



*Circular linked list before insertion*

And then,



*Circular linked list after insertion*

Below is the code implementation to insert a node at the beginning of the list:

struct Node \*addBegin(struct Node \*last, int data)

{

if (last == NULL)

    return addToEmpty(last, data);

// Creating a node dynamically.

struct Node \*temp

        = (struct Node \*)malloc(sizeof(struct Node));

// Assigning the data.

temp -> data = data;

// Adjusting the links.

temp -> next = last -> next;

last -> next = temp;

return last;

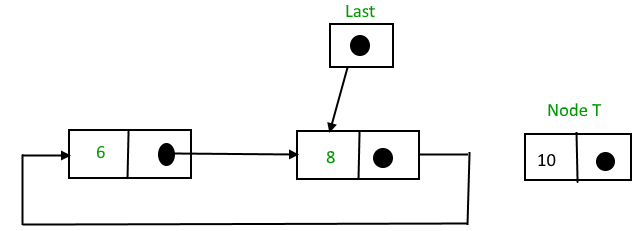
}

**Time complexity:**O(1) to insert a node at the beginning no need to traverse list it takes constant time   
**Auxiliary Space:** O(1)

2) **Insertion at the end of the list:**To insert a node at the end of the list, follow these steps:

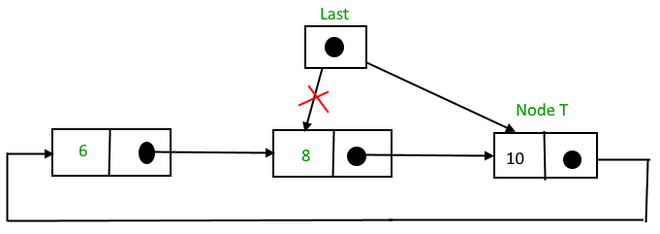
* Create a node, say T.
* Make T -> next = last -> next;
* last -> next = T.
* last = T.

Before insertion,



*Circular linked list before insertion of node at the end*

After insertion,



*Circular linked list after insertion of node at the end*

Below is the code implementation to insert a node at the beginning of the list:

struct Node \*addEnd(struct Node \*last, int data)

{

if (last == NULL)

    return addToEmpty(last, data);

// Creating a node dynamically.

struct Node \*temp =

        (struct Node \*)malloc(sizeof(struct Node));

// Assigning the data.

temp -> data = data;

// Adjusting the links.

temp -> next = last -> next;

last -> next = temp;

last = temp;

return last;

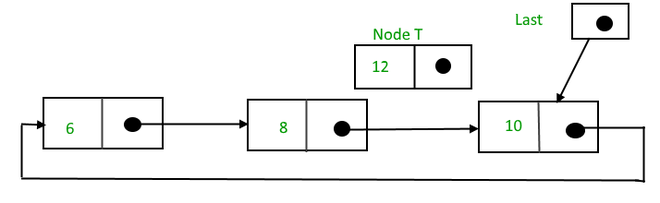
}

**Time Complexity:** O(1) to insert a node at the end of the list. No need to traverse the list as we are utilizing the last pointer, hence it takes constant time.  
**Auxiliary Space:** O(1)

3) **Insertion in between the nodes:**To insert a node in between the two nodes, follow these steps:

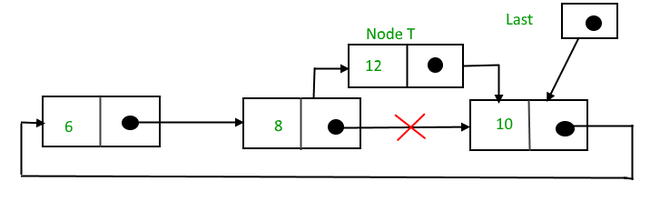
* Create a node, say T.
* Search for the node after which T needs to be inserted, say that node is P.
* Make T -> next = P -> next;
* P -> next = T.

Suppose 12 needs to be inserted after the node has the value 10,



*Circular linked list before insertion*

After searching and insertion,



*Circular linked list after  insertion*

Below is the code to insert a node at the specified position of the List:

struct Node \*addAfter(struct Node \*last, int data, int item)

{

    if (last == NULL)

    return NULL;

    struct Node \*temp, \*p;

    p = last -> next;

    // Searching the item.

    do

    {

        if (p ->data == item)

        {

            // Creating a node dynamically.

            temp = (struct Node \*)malloc(sizeof(struct Node));

            // Assigning the data.

            temp -> data = data;

            // Adjusting the links.

            temp -> next = p -> next;

            // Adding newly allocated node after p.

            p -> next = temp;

            // Checking for the last node.

            if (p == last)

                last = temp;

            return last;

        }

        p = p -> next;

    } while (p != last -> next);

    cout << item << " not present in the list." << endl;

    return last;

}

**Time Complexity:** O(N)  
**Auxiliary Space:**O(1)

### 2. Deletion in a circular linked list:

**1) Delete the node only if it is the only node in the circular linked list:**

* Free the node’s memory
* The last value should be NULL 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.

**2) Deletion of the last node:**

* Locate the node before the last node (let it be temp)
* Keep the address of the node next to the last node in temp
* Delete the last memory
* Put temp at the end

**3) Delete any node from the circular linked list:**We will be given a node and our task is to delete that node from the circular linked list.

**Algorithm:**  
**Case 1**: List is empty.

* If the list is empty we will simply return.

**Case 2**:List is not empty

* If the list is not empty then we define two pointers **curr**and **prev** and initialize the pointer **curr** with the **head**node.
* Traverse the list using **curr** to find the node to be deleted and before moving to curr to the next node, every time set prev = curr.
* If the node is found, check if it is the only node in the list. If yes, set head = NULL and free(curr).
* If the list has more than one node, check if it is the first node of the list. Condition to check this( curr == head). If yes, then move prev until it reaches the last node. After prev reaches the last node, set head = head -> next and prev -> next = head. Delete curr.
* If curr is not the first node, we check if it is the last node in the list. Condition to check this is (curr -> next == head).
* If curr is the last node. Set prev -> next = head and delete the node curr by free(curr).
* If the node to be deleted is neither the first node nor the last node, then set prev -> next = curr -> next and delete curr.
* If the node is not present in the list return head and don’t do anything.

// Function to delete a given node

// from the list

void deleteNode(Node\*\* head, int key)

{

    // If linked list is empty

    if (\*head == NULL)

        return;

    // If the list contains only a

    // single node

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

        free(\*head);

        \*head = NULL;

        return;

    }

    Node \*last = \*head, \*d;

    // If head is to be deleted

    if ((\*head)->data == key) {

        // Find the last node of the list

        while (last->next != \*head)

            last = last->next;

        // Point last node to the next of

        // head i.e. the second node

        // of the list

        last->next = (\*head)->next;

        free(\*head);

        \*head = last->next;

        return;

    }

    // Either the node to be deleted is

    // not found or the end of list

    // is not reached

    while (last->next != \*head && last->next->data != key) {

        last = last->next;

    }

    // If node to be deleted was found

    if (last->next->data == key) {

        d = last->next;

        last->next = d->next;

        free(d);

    }

    else

        cout << "Given node is not found in the list!!!\n";

}

**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.

## **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.

# 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.
* 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.

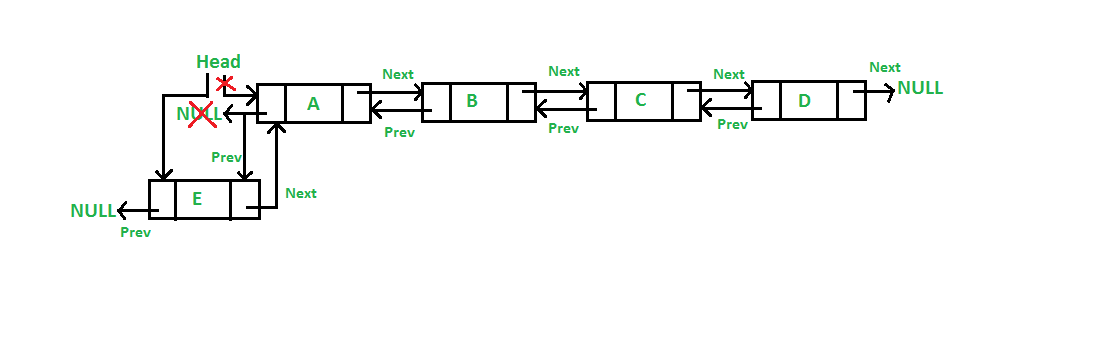
## **Insertion** in DLL:

A node can be added in four ways:

* At the front of the DLL
* After a given node.
* At the end of the DLL
* Before a given node.

### **1) Add a node at the front:**

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example, if the given Linked List is **1->0->1->5**and we add an item **5**at the front, then the Linked List becomes **5->1->0->1->5**. Let us call the function that adds at the front of the list push(). The push() must receive a pointer to the head pointer because the push must change the head pointer to point to the new node (See [this](https://www.geeksforgeeks.org/how-to-write-functions-that-modify-the-head-pointer-of-a-linked-list/))



/\* Given a reference (pointer to pointer)

to the head of a list

and an int, inserts a new node on the

front of the list. \*/

void push(Node\*\* head\_ref, int new\_data)

{

    /\* 1. allocate node \*/

    Node\* new\_node = new Node();

    /\* 2. put in the data \*/

    new\_node->data = new\_data;

    /\* 3. Make next of new node as head

    and previous as NULL \*/

    new\_node->next = (\*head\_ref);

    new\_node->prev = NULL;

    /\* 4. change prev of head node to new node \*/

    if ((\*head\_ref) != NULL)

        (\*head\_ref)->prev = new\_node;

    /\* 5. move the head to point to the new node \*/

    (\*head\_ref) = new\_node;

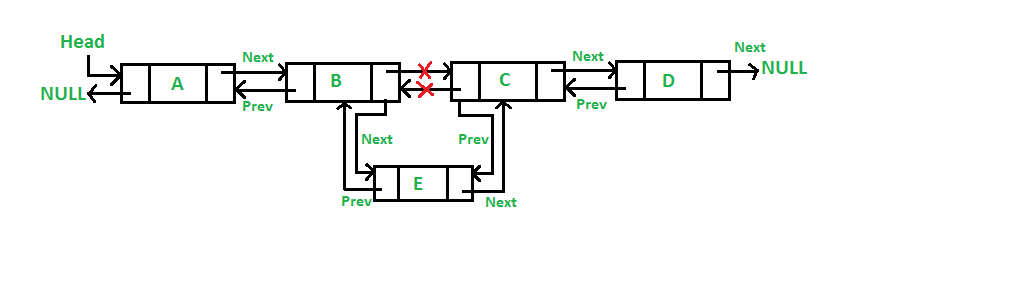
}

**Time Complexity: O(1)**  
**Auxiliary Space: O(1)**

**Note:** Four steps of the above five steps are the same as [the 4 steps used for inserting at the front in the singly linked list](https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/). The only extra step is to change the previous head.

### **2) Add a node after a given node:**

We are given a pointer to a node as prev\_node, and the new node is inserted after the given node.



Below is the implementation of the 7 steps to insert a node after a given node in the linked list:

/\* Given a node as prev\_node, insert

a new node after the given node \*/

void insertAfter(Node\* prev\_node, int new\_data)

{

    /\*1. check if the given prev\_node is NULL \*/

    if (prev\_node == NULL) {

        cout << "the given previous node cannot be NULL";

        return;

    }

    /\* 2. allocate new node \*/

    Node\* new\_node = new Node();

    /\* 3. put in the data \*/

    new\_node->data = new\_data;

    /\* 4. Make next of new node as next of prev\_node \*/

    new\_node->next = prev\_node->next;

    /\* 5. Make the next of prev\_node as new\_node \*/

    prev\_node->next = new\_node;

    /\* 6. Make prev\_node as previous of new\_node \*/

    new\_node->prev = prev\_node;

    /\* 7. Change previous of new\_node's next node \*/

    if (new\_node->next != NULL)

        new\_node->next->prev = new\_node;

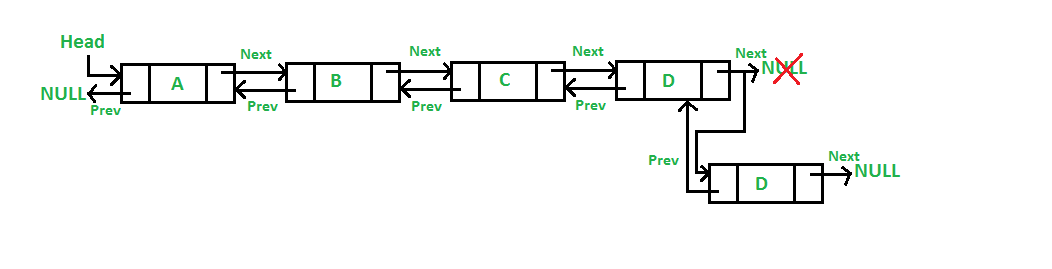
}

**Time Complexity: O(1)**  
**Auxiliary Space: O(1)**

**Note:** Five of the above steps step process are the same as [the 5 steps used for inserting after a given node in the singly linked list](https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/). The two extra steps are needed to change the previous pointer of the new node and the previous pointer of the new node’s next node.

### **3) Add a node at the end:**

The new node is always added after the last node of the given Linked List. For example, if the given DLL is **5->1->0->1->5->2** and we add item **30** at the end, then the DLL becomes **5->1->0->1->5->2->30**. Since a Linked List is typically represented by its head of it, we have to traverse the list till the end and then change the next of last node to the new node.



/\* Given a reference (pointer to pointer) to the head

of a DLL and an int, appends a new node at the end \*/

void append(Node\*\* head\_ref, int new\_data)

{

    /\* 1. allocate node \*/

    Node\* new\_node = new Node();

    Node\* last = \*head\_ref; /\* used in step 5\*/

    /\* 2. put in the data \*/

    new\_node->data = new\_data;

    /\* 3. This new node is going to be the last node, so

        make next of it as NULL\*/

    new\_node->next = NULL;

    /\* 4. If the Linked List is empty, then make the new

        node as head \*/

    if (\*head\_ref == NULL) {

        new\_node->prev = NULL;

        \*head\_ref = new\_node;

        return;

    }

    /\* 5. Else traverse till the last node \*/

    while (last->next != NULL)

        last = last->next;

    /\* 6. Change the next of last node \*/

    last->next = new\_node;

    /\* 7. Make last node as previous of new node \*/

    new\_node->prev = last;

    return;

}

**uxiliary Space: O(1)**

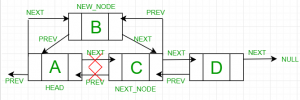
**Note:** Six of the above 7 steps are the same as [the 6 steps used for inserting after a given node in the singly linked list](https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/). One extra step is needed to change the previous pointer of the new node.

### **4) Add a node before a given node:**

Follow the below steps to solve the problem:

Let the pointer to this given node be next\_node and the data of the new node be added as new\_data.

* Check if the next\_node is NULL or not. If it’s NULL, return from the function because any new node can not be added before a NULL
* Allocate memory for the new node, let it be called new\_node
* Set new\_node->data = new\_data
* Set the previous pointer of this new\_node as the previous node of the next\_node, new\_node->prev = next\_node->prev
* Set the previous pointer of the next\_node as the new\_node, next\_node->prev = new\_node
* Set the next pointer of this new\_node as the next\_node, new\_node->next = next\_node;
* If the previous node of the new\_node is not NULL, then set the next pointer of this previous node as new\_node, new\_node->prev->next = new\_node
* Else, if the prev of new\_node is NULL, it will be the new head node. So, make (\*head\_ref) = new\_node.



/\* Given a node as prev\_node, insert

a new node after the given node \*/

void insertAfter(Node\* prev\_node, int new\_data)

{

    /\*1. check if the given prev\_node is NULL \*/

    if (prev\_node == NULL) {

        cout << "the given previous node cannot be NULL";

        return;

    }

    /\* 2. allocate new node \*/

    Node\* new\_node = new Node();

    /\* 3. put in the data \*/

    new\_node->data = new\_data;

    /\* 4. Make next of new node as next of prev\_node \*/

    new\_node->next = prev\_node->next;

    /\* 5. Make the next of prev\_node as new\_node \*/

    prev\_node->next = new\_node;

    /\* 6. Make prev\_node as previous of new\_node \*/

    new\_node->prev = prev\_node;

    /\* 7. Change previous of new\_node's next node \*/

    if (new\_node->next != NULL)

        new\_node->next->prev = new\_node;

}

/\* Given a reference (pointer to pointer) to the head

of a DLL and an int, appends a new node at the end \*/

void append(Node\*\* head\_ref, int new\_data)

{

    /\* 1. allocate node \*/

    Node\* new\_node = new Node();

    Node\* last = \*head\_ref; /\* used in step 5\*/

    /\* 2. put in the data \*/

    new\_node->data = new\_data;

    /\* 3. This new node is going to be the last node, so

        make next of it as NULL\*/

    new\_node->next = NULL;

    /\* 4. If the Linked List is empty, then make the new

        node as head \*/

    if (\*head\_ref == NULL) {

        new\_node->prev = NULL;

        \*head\_ref = new\_node;

        return;

    }

    /\* 5. Else traverse till the last node \*/

    while (last->next != NULL)

        last = last->next;

    /\* 6. Change the next of last node \*/

    last->next = new\_node;

    /\* 7. Make last node as previous of new node \*/

    new\_node->prev = last;

    return;

}

# Insertion in Doubly Circular 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.

Following is the representation of a Circular doubly linked list node in C/C++:

static class node {

  int data;

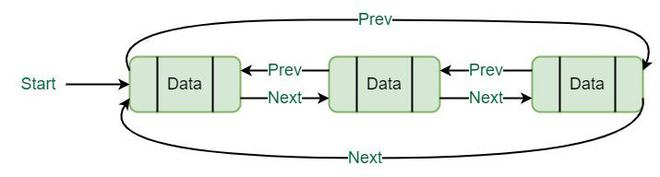
  // pointer to next node

  node next;

  // pointer to prev node

  node prev;

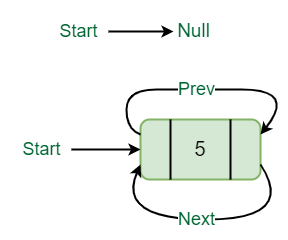
}



## **Insertion in Circular Doubly Linked List:**

### 1. Insertion at the end of the list or in an empty list:

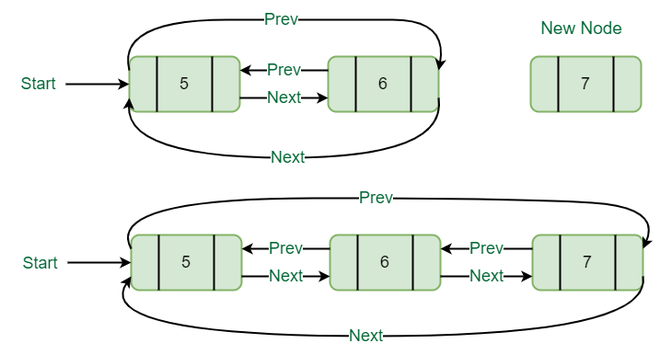
A node(Say **N**) is inserted with **data = 5**. So, the previous pointer of N points to N and the next pointer of N also points to N. But now start pointer points to the first node of the list.



*Insertion in an empty list*

### 2.**List initially contains some nodes, start points to the first node of the List:**

A node(Say M) is inserted with data = 7, so the previous pointer of M points to the last node, the next pointer of M points to the first node and the last node’s next pointer points to this M node, and first node’s previous pointer points to this M node.



*Insertion at the end of list*

Below is the implementation of the above operations:

// Function to insert at the end

void insertEnd(struct Node\*\* start, int value)

{

    // If the list is empty, create a single node

    // circular and doubly list

    if (\*start == NULL) {

        struct Node\* new\_node = new Node;

        new\_node->data = value;

        new\_node->next = new\_node->prev = new\_node;

        \*start = new\_node;

        return;

    }

    // If list is not empty

    /\* Find last node \*/

    Node\* last = (\*start)->prev;

    // Create Node dynamically

    struct Node\* new\_node = new Node;

    new\_node->data = value;

    // Start is going to be next of new\_node

    new\_node->next = \*start;

    // Make new node previous of start

    (\*start)->prev = new\_node;

    // Make last previous of new node

    new\_node->prev = last;

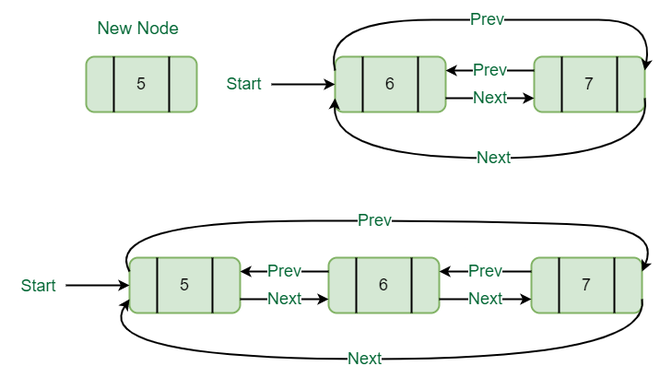
    // Make new node next of old last

    last->next = new\_node;

}

### 3. **Insertion at the beginning of the list:**

To insert a node at the beginning of the list, create a node(Say T) with data = 5, T next pointer points to the first node of the list, T previous pointer points to the last node of the list, last node’s next pointer points to this T node, first node’s previous pointer also points this T node and at last don’t forget to shift ‘Start’ pointer to this T node.



*Insertion at the beginning of the list*

Below is the implementation of the above operation:

// Function to insert Node at the beginning

// of the List,

void insertBegin(struct Node\*\* start, int value)

{

    // Pointer points to last Node

    struct Node\* last = (\*start)->prev;

    struct Node\* new\_node = new Node;

    new\_node->data = value; // Inserting the data

    // setting up previous and next of new node

    new\_node->next = \*start;

    new\_node->prev = last;

    // Update next and previous pointers of start

    // and last.

    last->next = (\*start)->prev = new\_node;

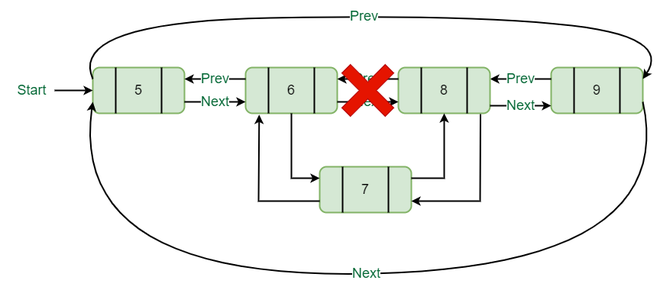
    // Update start pointer

    \*start = new\_node;

}

### **4. Insertion in between the nodes of the list**:

To insert a node in between the list, two data values are required one after which new node will be inserted and another is the data of the new node.



*Insertion in between other nodes*

Below is the implementation of the above operation:

// Function to insert node with value as value1.

// The new node is inserted after the node with

// with value2

void insertAfter(struct Node\*\* start, int value1,

                 int value2)

{

    struct Node\* new\_node = new Node;

    new\_node->data = value1; // Inserting the data

    // Find node having value2 and next node of it

    struct Node\* temp = \*start;

    while (temp->data != value2)

        temp = temp->next;

    struct Node\* next = temp->next;

    // insert new\_node between temp and next.

    temp->next = new\_node;

    new\_node->prev = temp;

    new\_node->next = next;

    next->prev = new\_node;

}

## **Advantages of circular doubly linked list:**

* The list can be traversed from both directions i.e. from head to tail or from tail to head.
* Jumping from head to tail or from tail to head is done in constant time O(1).
* Circular Doubly Linked Lists are used for the implementation of advanced data structures like the [Fibonacci Heap](https://en.wikipedia.org/wiki/Fibonacci_heap).

## **Disadvantages of circular doubly linked list:**

* It takes slightly extra memory in each node to accommodate the previous pointer.
* Lots of pointers are involved while implementing or doing operations on a list. So, pointers should be handled carefully otherwise data of the list may get lost.

## **Applications of Circular doubly linked list:**

* Managing song playlists in media player applications.
* Managing shopping carts in online shopping.

# Deletion in Doubly Circular Linked List

We have discussed the doubly circular linked list introduction and its insertion.

Let us formulate the problem statement to understand the deletion process. Given a ‘key’, delete the first occurrence of this key in the circular doubly linked list.

***Algorithm:***

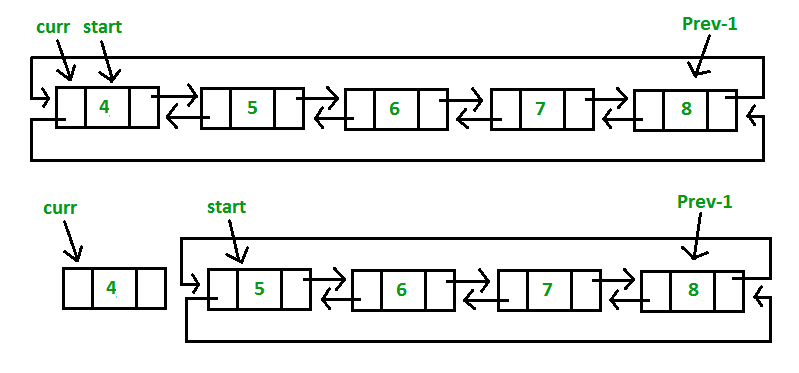
***Case 1:****Empty List(start = NULL)*

* *If the list is empty, simply return it.*

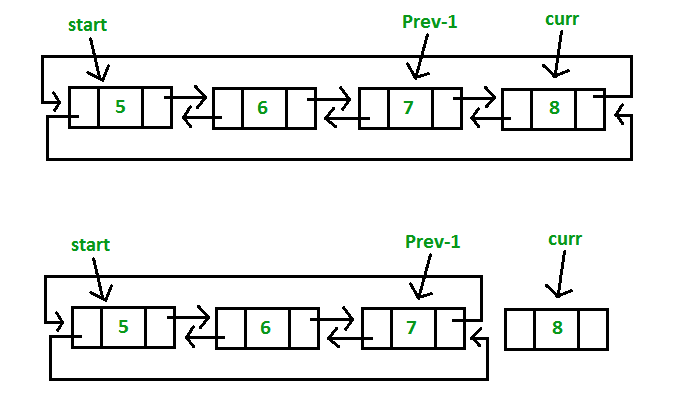
***Case 2:****The List initially contains some nodes, start points at the first node of the List*

1. *If the list is not empty, then we define two pointers****curr****and****prev\_1****and initialize the pointer****curr****points to the first node of the list, and prev\_1 = NULL.*
2. *Traverse the list using the curr pointer to find the node to be deleted and before moving from curr to the next node, every time set prev\_1 = curr.*
3. *If the node is found, check if it is the only node in the list. If yes, set start = NULL and free the node pointing by curr.*
4. *If the list has more than one node, check if it is the first node of the list. The condition to check this is (curr == start). If yes, then move prev\_1 to the last node(prev\_1 = start -> prev). After prev\_1 reaches the last node, set start = start -> next and prev\_1 -> next = start and start -> prev = prev\_1. Free the node pointing by curr.*
5. *If curr is not the first node, we check if it is the last node in the list. The condition to check this is (curr -> next == start). If yes, set prev\_1 -> next = start and start -> prev = prev\_1. Free the node pointing by curr.*
6. *If the node to be deleted is neither the first node nor the last node, declare one more pointer****temp****and initialize the pointer****temp****points to the next of curr pointer (temp = curr->next). Now set, prev\_1 -> next = temp and temp ->prev = prev\_1. Free the node pointing by curr.*

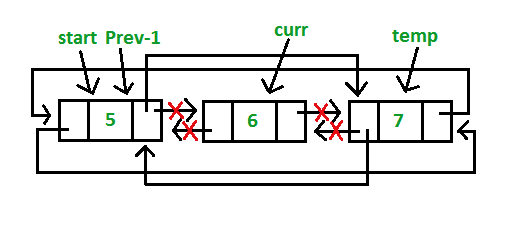
* If the given *key*(Say 4) matches with the first node of the list(Step 4):



* If the given *key*(Say 8) matches with the last node of the list(Step 5):



* If the given *key*(Say 6) matches with the middle node of the list(Step 6):



// Function to delete a given node from the list

void deleteNode(struct Node\*\* start, int key)

{

    // If list is empty

    if (\*start == NULL)

        return;

    // Find the required node

    // Declare two pointers and initialize them

    struct Node \*curr = \*start, \*prev\_1 = NULL;

    while (curr->data != key) {

        // If node is not present in the list

        if (curr->next == \*start) {

            printf("\nList doesn't have node with value = %d", key);

            return;

        }

        prev\_1 = curr;

        curr = curr->next;

    }

    // Check if node is the only node in list

    if (curr->next == \*start && prev\_1 == NULL) {

        (\*start) = NULL;

        free(curr);

        return;

    }

    // If list has more than one node,

    // check if it is the first node

    if (curr == \*start) {

        // Move prev\_1 to last node

        prev\_1 = (\*start)->prev;

        // Move start ahead

        \*start = (\*start)->next;

        // Adjust the pointers of prev\_1 and start node

        prev\_1->next = \*start;

        (\*start)->prev = prev\_1;

        free(curr);

    }

    // check if it is the last node

    else if (curr->next == \*start) {

        // Adjust the pointers of prev\_1 and start node

        prev\_1->next = \*start;

        (\*start)->prev = prev\_1;

        free(curr);

    }

    else {

        // create new pointer, points to next of curr node

        struct Node\* temp = curr->next;

        // Adjust the pointers of prev\_1 and temp node

        prev\_1->next = temp;

        temp->prev = prev\_1;

        free(curr);

    }

}

# Program to implement Singly Linked List in C++ using class

// C++ program for the above approach

#include <iostream>

using namespace std;

// Node class to represent

// a node of the linked list.

class Node {

public:

    int data;

    Node\* next;

    // Default constructor

    Node()

    {

        data = 0;

        next = NULL;

    }

    // Parameterised Constructor

    Node(int data)

    {

        this->data = data;

        this->next = NULL;

    }

};

// Linked list class to

// implement a linked list.

class Linkedlist {

    Node\* head;

public:

    // Default constructor

    Linkedlist() { head = NULL; }

    // Function to insert a

    // node at the end of the

    // linked list.

    void insertNode(int);

    // Function to print the

    // linked list.

    void printList();

    // Function to delete the

    // node at given position

    void deleteNode(int);

};

// Function to delete the

// node at given position

void Linkedlist::deleteNode(int nodeOffset)

{

    Node \*temp1 = head, \*temp2 = NULL;

    int ListLen = 0;

    if (head == NULL) {

        cout << "List empty." << endl;

        return;

    }

    // Find length of the linked-list.

    while (temp1 != NULL) {

        temp1 = temp1->next;

        ListLen++;

    }

    // Check if the position to be

    // deleted is greater than the length

    // of the linked list.

    if (ListLen < nodeOffset) {

        cout << "Index out of range"

             << endl;

        return;

    }

    // Declare temp1

    temp1 = head;

    // Deleting the head.

    if (nodeOffset == 1) {

        // Update head

        head = head->next;

        delete temp1;

        return;

    }

    // Traverse the list to

    // find the node to be deleted.

    while (nodeOffset-- > 1) {

        // Update temp2

        temp2 = temp1;

        // Update temp1

        temp1 = temp1->next;

    }

    // Change the next pointer

    // of the previous node.

    temp2->next = temp1->next;

    // Delete the node

    delete temp1;

}

// Function to insert a new node.

void Linkedlist::insertNode(int data)

{

    // Create the new Node.

    Node\* newNode = new Node(data);

    // Assign to head

    if (head == NULL) {

        head = newNode;

        return;

    }

    // Traverse till end of list

    Node\* temp = head;

    while (temp->next != NULL) {

        // Update temp

        temp = temp->next;

    }

    // Insert at the last.

    temp->next = newNode;

}

// Function to print the

// nodes of the linked list.

void Linkedlist::printList()

{

    Node\* temp = head;

    // Check for empty list.

    if (head == NULL) {

        cout << "List empty" << endl;

        return;

    }

    // Traverse the list.

    while (temp != NULL) {

        cout << temp->data << " ";

        temp = temp->next;

    }

}

// Driver Code

int main()

{

    Linkedlist list;

    // Inserting nodes

    list.insertNode(1);

    list.insertNode(2);

    list.insertNode(3);

    list.insertNode(4);

    cout << "Elements of the list are: ";

    // Print the list

    list.printList();

    cout << endl;

    // Delete node at position 2.

    list.deleteNode(2);

    cout << "Elements of the list are: ";

    list.printList();

    cout << endl;

    return 0;

}