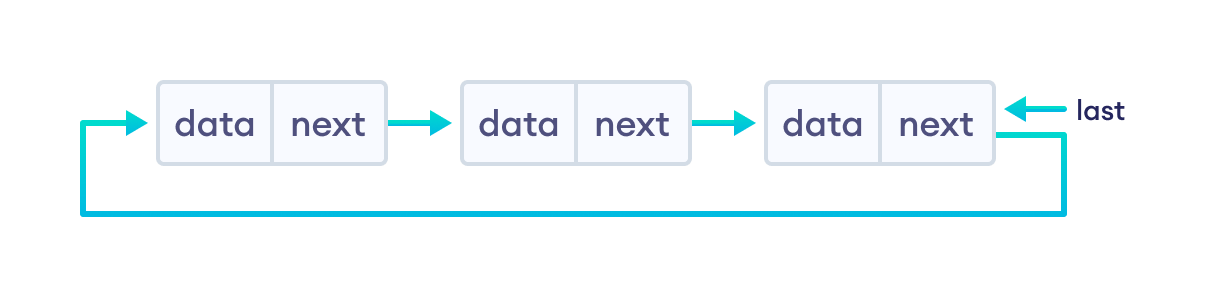
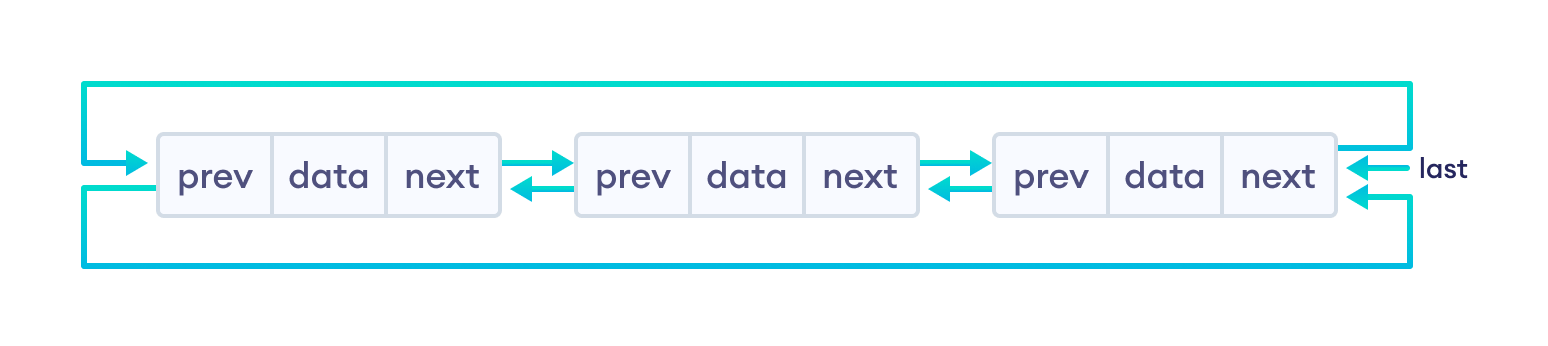
**Circular Linked List**

A circular linked list is a type of [linked list](https://www.programiz.com/dsa/linked-list) in which the first and the last nodes are also connected to each other to form a circle.

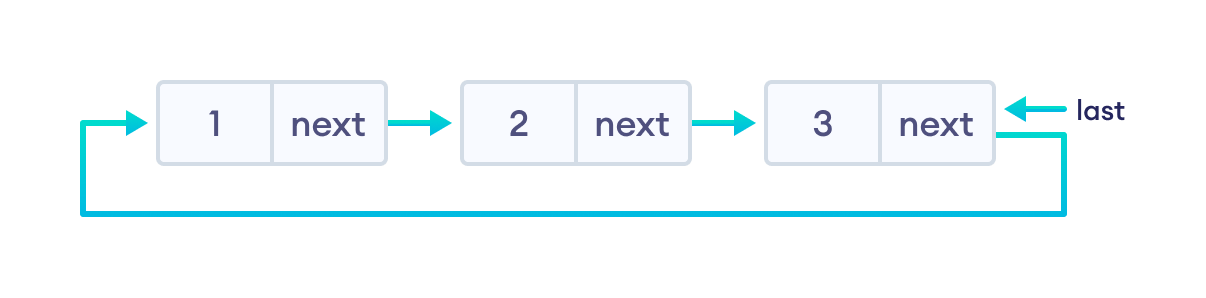
There are basically two types of circular linked list:

**1. Circular Singly Linked List**

**2. Circular Doubly Linked List**



## Representation of Circular Linked List



struct Node {

int data;

struct Node \* next;

};

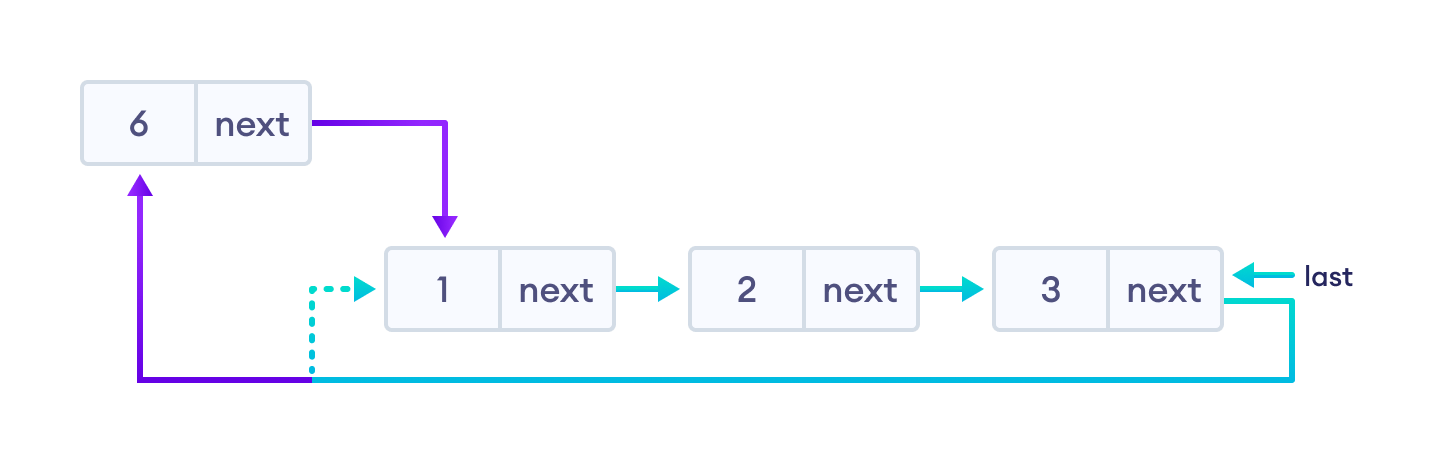
## Insertion on a Circular Linked List

We can insert elements at 3 different positions of a circular linked list:

1. **Insertion at the beginning**
2. **Insertion in-between nodes**
3. **Insertion at the end**

### 1. Insertion at the Beginning

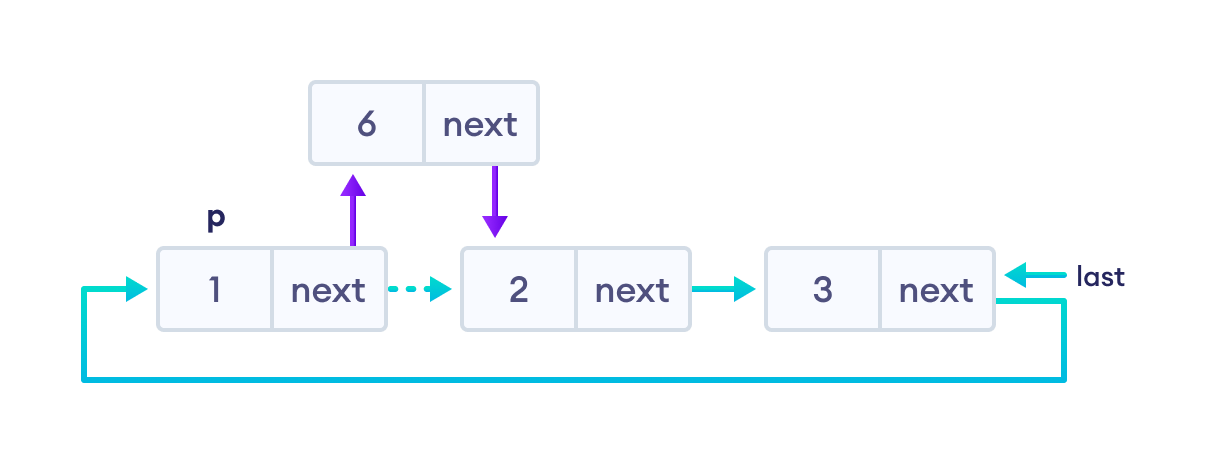
* store the address of the current first node in the newNode (i.e. pointing the newNode to the current first node)
* point the last node to newNode (i.e making newNode as head)



### 2. Insertion in between two nodes

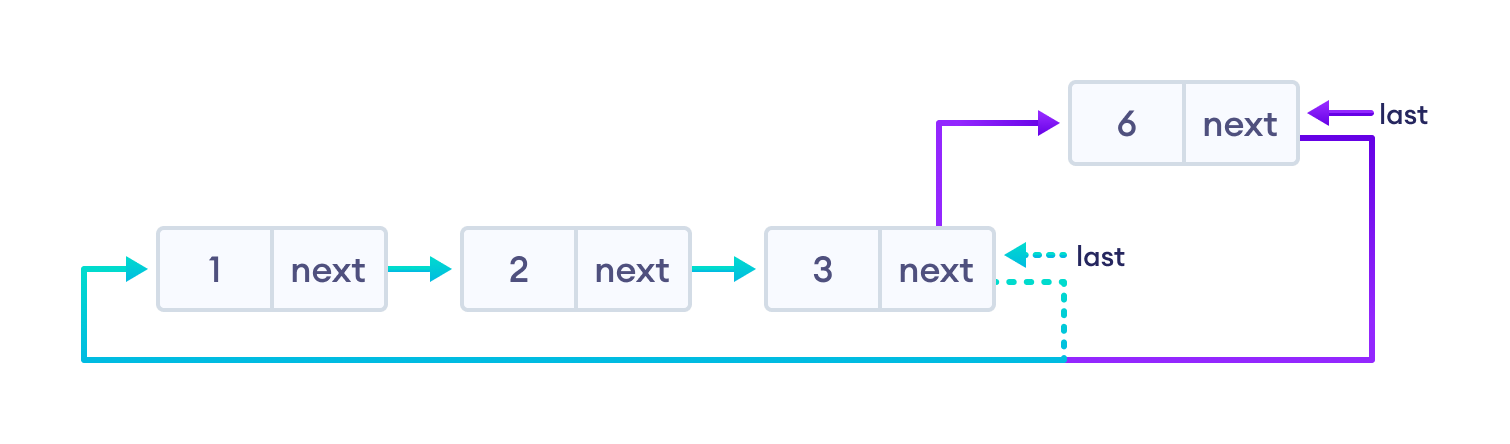
Let's insert newNode after the first node.

* travel to the node given (let this node be p)
* point the next of newNode to the node next to p
* store the address of newNode at next of p

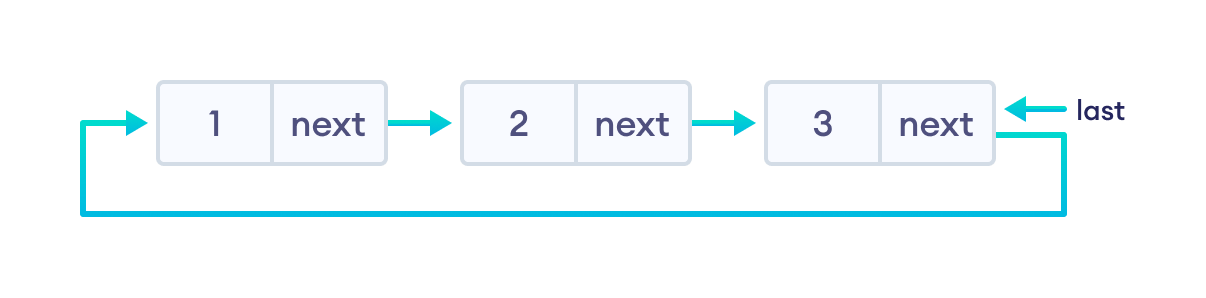


### 3. Insertion at the end

* store the address of the head node to next of newNode (making newNode the last node)
* point the current last node to newNode
* make newNode as the last node



## Deletion on a Circular Linked List

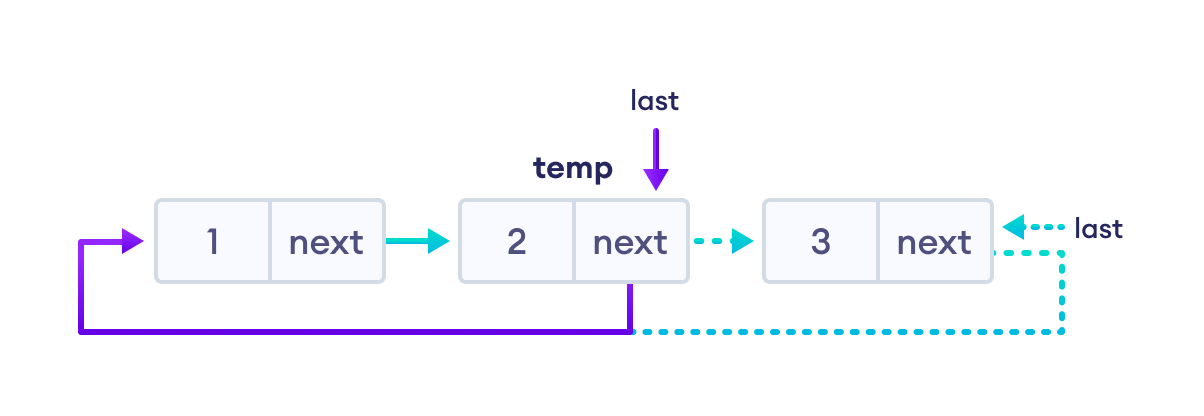


### 1. If the node to be deleted is the only node

* free the memory occupied by the node
* store NULL in last

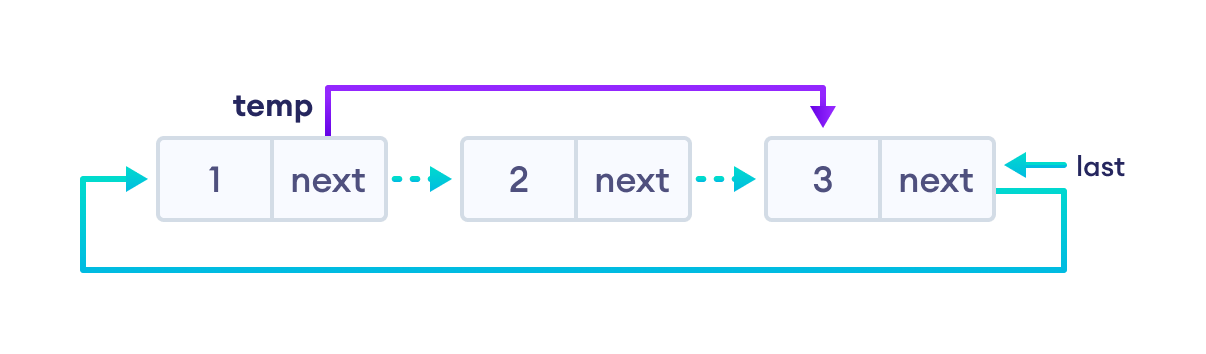
### 2. If last node is to be deleted

* find the node before the last node (let it be temp)
* store the address of the node next to the last node in temp
* free the memory of last
* make temp as the last node



### 3. If any other nodes are to be deleted

* travel to the node to be deleted (here we are deleting node 2)
* let the node before node 2 be temp
* store the address of the node next to 2 in temp
* free the memory of 2



**Program Implementation**

// C++ code to perform circular linked list operations

#include <iostream>

using namespace std;

struct Node {

int data;

struct Node\* next;

};

struct Node\* addToEmpty(struct Node\* last, int data) {

if (last != NULL) return last;

// allocate memory to the new node

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

// assign data to the new node

newNode->data = data;

// assign last to newNode

last = newNode;

// create link to iteself

last->next = last;

return last;

}

// add node to the front

struct Node\* addFront(struct Node\* last, int data) {

// check if the list is empty

if (last == NULL) return addToEmpty(last, data);

// allocate memory to the new node

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

// add data to the node

newNode->data = data;

// store the address of the current first node in the newNode

newNode->next = last->next;

// make newNode as head

last->next = newNode;

return last;

}

// add node to the end

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

// check if the node is empty

if (last == NULL) return addToEmpty(last, data);

// allocate memory to the new node

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

// add data to the node

newNode->data = data;

// store the address of the head node to next of newNode

newNode->next = last->next;

// point the current last node to the newNode

last->next = newNode;

// make newNode as the last node

last = newNode;

return last;

}

// insert node after a specific node

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

// check if the list is empty

if (last == NULL) return NULL;

struct Node \*newNode, \*p;

p = last->next;

do {

// if the item is found, place newNode after it

if (p->data == item) {

// allocate memory to the new node

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

// add data to the node

newNode->data = data;

// make the next of the current node as the next of newNode

newNode->next = p->next;

// put newNode to the next of p

p->next = newNode;

// if p is the last node, make newNode as the last node

if (p == last) last = newNode;

return last;

}

p = p->next;

} while (p != last->next);

cout << "\nThe given node is not present in the list" << endl;

return last;

}

// delete a node

void deleteNode(Node\*\* last, int key) {

// if linked list is empty

if (\*last == NULL) return;

// if the list contains only a single node

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

free(\*last);

\*last = NULL;

return;

}

Node \*temp = \*last, \*d;

// if last is to be deleted

if ((\*last)->data == key) {

// find the node before the last node

while (temp->next != \*last) temp = temp->next;

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

temp->next = (\*last)->next;

free(\*last);

\*last = temp->next;

}

// travel to the node to be deleted

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

temp = temp->next;

}

// if node to be deleted was found

if (temp->next->data == key) {

d = temp->next;

temp->next = d->next;

free(d);

}

}

void traverse(struct Node\* last) {

struct Node\* p;

if (last == NULL) {

cout << "The list is empty" << endl;

return;

}

p = last->next;

do {

cout << p->data << " ";

p = p->next;

} while (p != last->next);

}

int main() {

struct Node\* last = NULL;

last = addToEmpty(last, 6);

last = addEnd(last, 8);

last = addFront(last, 2);

last = addAfter(last, 10, 2);

traverse(last);

deleteNode(&last, 8);

cout << endl;

traverse(last);

return 0;

}

## Circular Linked List Complexity

|  |  |  |
| --- | --- | --- |
| Circular Linked List Complexity | **Time Complexity** | **Space Complexity** |
| **Insertion Operation** | O(1) or O(n) | O(1) |
| **Deletion Operation** | O(1) | O(1) |

**Complexity of Insertion Operation**

* The insertion operations that do not require traversal have the time complexity of O(1).
* And, an insertion that requires traversal has a time complexity of O(n).
* The space complexity is O(1).

**2. Complexity of Deletion Operation**

* All deletion operations run with a time complexity of O(1).
* And, the space complexity is O(1).

## **Why Circular Linked List?**

1. The NULL assignment is not required because a node always points to another node.
2. The starting point can be set to any node.
3. Traversal from the first node to the last node is quick.

## Circular Linked List Applications

* It is used in multiplayer games to give a chance to each player to play the game.
* Multiple running applications can be placed in a circular linked list on an operating system. The os keeps on iterating over these applications.