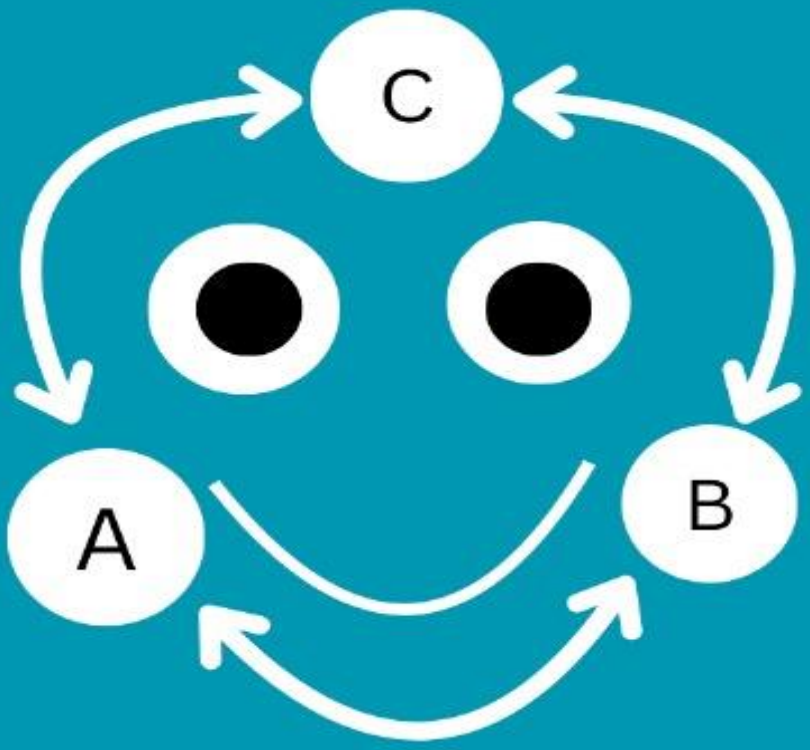


Circular Doubly Linked List

Table of Contents

Hi I am Deadpool.I wanna do somethings Productive So,I am here to teach you the fundamentals of Circular Doubly Linked List

Aarambikalama!!!.

Introduction

Circular linked list:

A circular linked list is a special type of linked list where all the nodes are connected to form a circle. Unlike a regular linked list, which ends with a node pointing to NULL, the last node in a circular linked list points back to the first node. This means that you can keep traversing the list without ever reaching a NULL value.

Doubly linked list:

A doubly linked list is a data structure that consists of a set of nodes, each of which contains a value and two pointers, one pointing to the previous node in the list and one pointing to the next node in the list. This allows for efficient traversal of the list in both directions, making it suitable for applications where

frequent insertions and deletions are required.

Circular Doubly Linked list:

A circular doubly linked list is a data structure that consists of a collection of nodes, where each node contains a data element and two pointers: one that points to the next node in the list and another that points to the previous node in the list. In a circular doubly linked list, the last node points to the first node, creating a circular structure. Here we are going to study the doubly circular linked list introduction and insertion. A doubly circular linked list is a linked list that has the features of both the lists i.e. Doubly linked list and circular linked list. In a doubly circular linked list, two nodes are connected or linked together by the previous and the next pointers, and the last node of the doubly circular linked list is connected to the first node of the circular doubly linked list. Before learning the circular doubly linked list, let us first look at the Features of a Doubly Linked List.

Purpose

Insertion and Deletion: A circular doubly linked list allows for efficient insertion and deletion of nodes, especially compared to arrays. In arrays, inserting or deleting an element requires shifting other elements, which can be time-consuming. In a circular doubly linked list, no shifting is needed, and elements can be added or removed by simply adjusting the pointers, making the operations much faster.

Memory Efficiency: Unlike arrays that require a fixed size, a circular doubly linked list uses memory dynamically. You don't need to pre-allocate memory, and it grows or shrinks based on the number of nodes, saving memory in cases of variable-length data.

Circular Navigation: In comparison to simple linked lists or arrays, the circular nature of this data structure enables seamless looping. For instance, in circular queues or buffers, once the end is reached, it automatically loops back to the beginning, making it ideal for buffering applications in systems like audio/video streaming.

Flexibility in Data Modification: While arrays have fixed indices, a circular doubly linked list offers greater flexibility for inserting, deleting, or moving nodes anywhere in the list, which can be done in constant time if the position is known.

Application

Music or Video Playlists: A circular doubly linked list is often used to manage playlists. You can move forward to the next song or backward to the previous one without reaching the end, as it loops around.

Undo/Redo Features in Software: Many applications use circular doubly linked lists to implement undo/redo features. You can move back to a previous state or forward to a newer one.

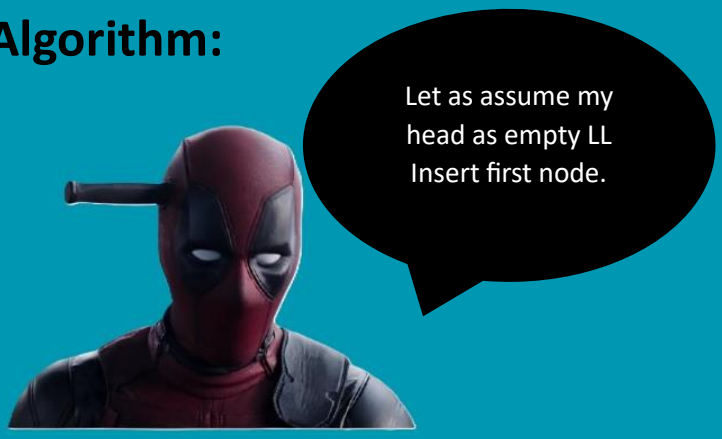
Real-Time Systems: In real-time systems, circular doubly linked lists help in tasks that require continuous, repetitive cycles, like task scheduling or process management.

Navigation Systems: It helps in navigation systems where moving between nodes (locations) in any direction is needed, making the travel seamless.

Basic Operations

Insertion

Algorithm:



i.) Insert at front:

Firstly, we will initialize the start node as NULL, and then we can continue to insert the nodes.The following cases persist if we want to insert the nodes:

a) If the list is empty

We will create a new_node with the given value.

Assign the prev and next of new_node to the new_node itself.

Update new_node as the start node.

b) Insert at begin

We will create a new_node with the given value.

Assign the next of new_node as the start node.

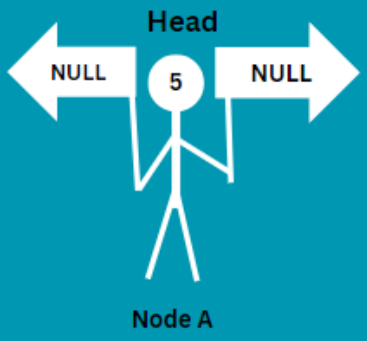
Assign the prev of new_node to the last node of the linked list.

Update the prev of the start node and next of the last node as new_node.

Update new_node as the start node.

Case a.)

List is empty return new node

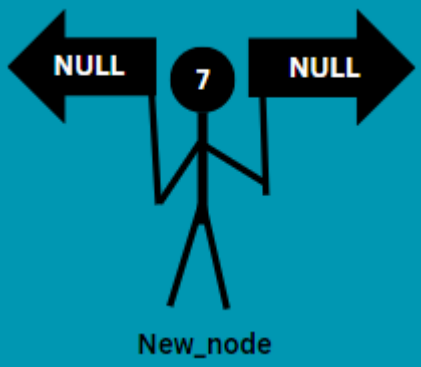


Case b.)

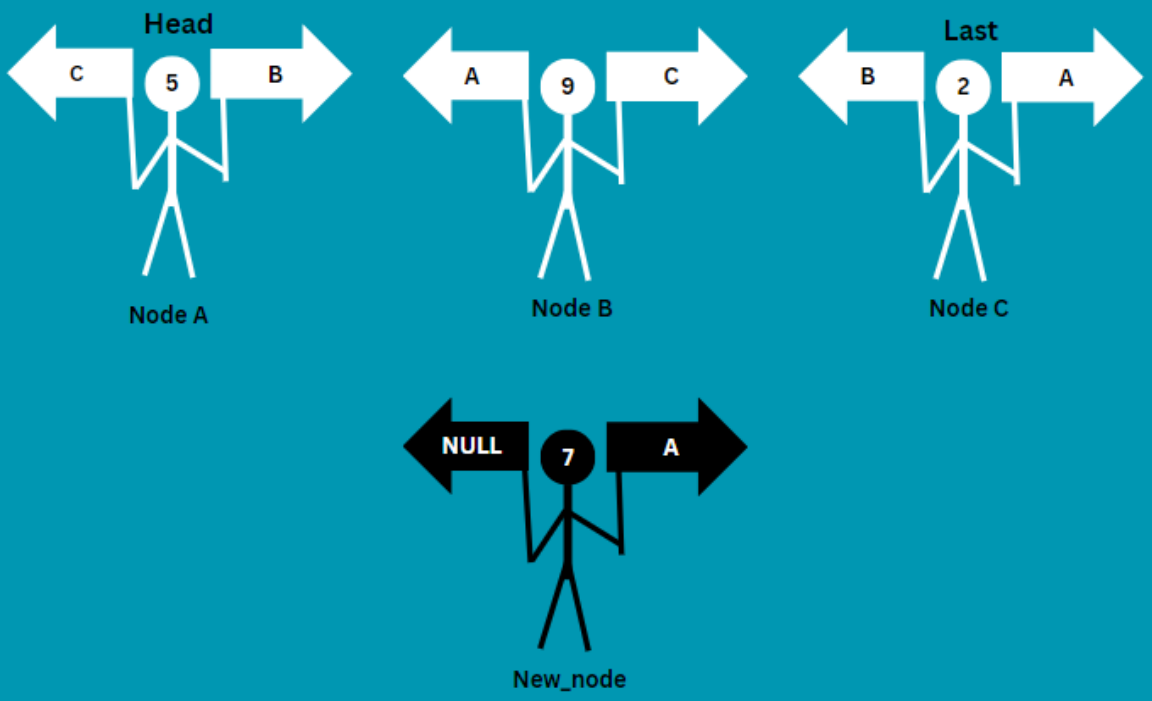
At the beginning

Step 1:

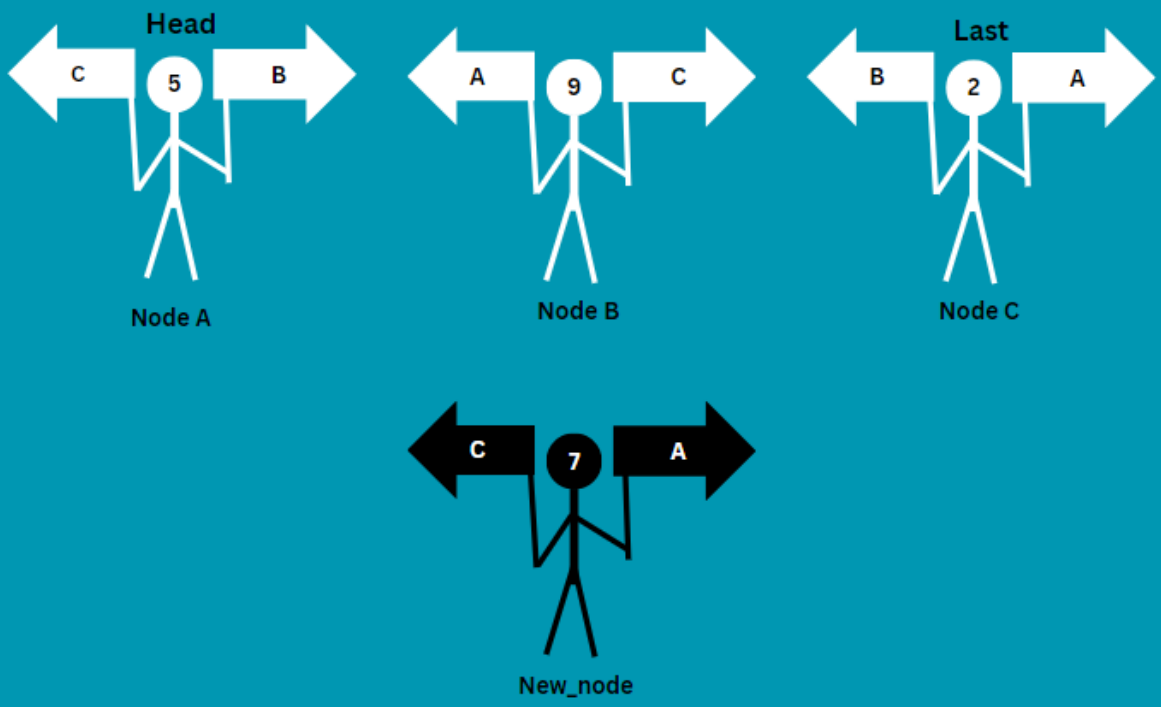
If given value is 7;



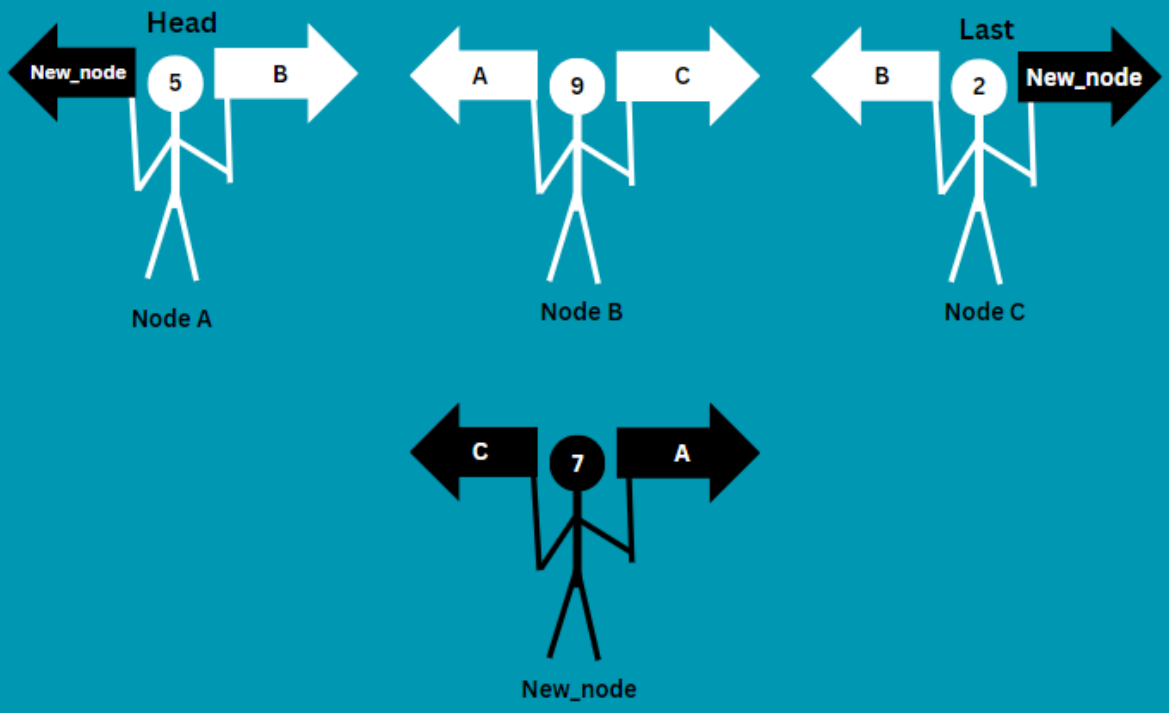
Step 2:


Head
C
5
B
Node A
A
9
C
Node B
Last
B
2
A
Node C
NULL
7
A
New_node

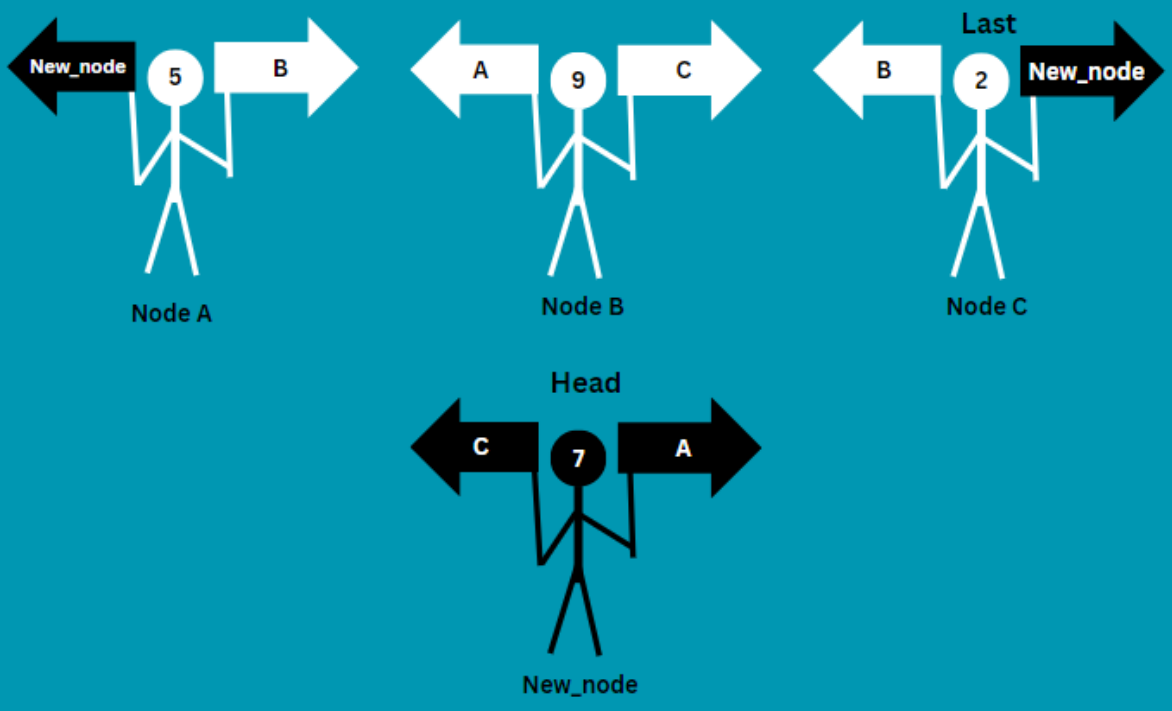
Step 3:


Head
C
5
B
Node A
A
9
C
Node B
Last
B
2
A
Node C
C
7
A
New_node

Step 4:

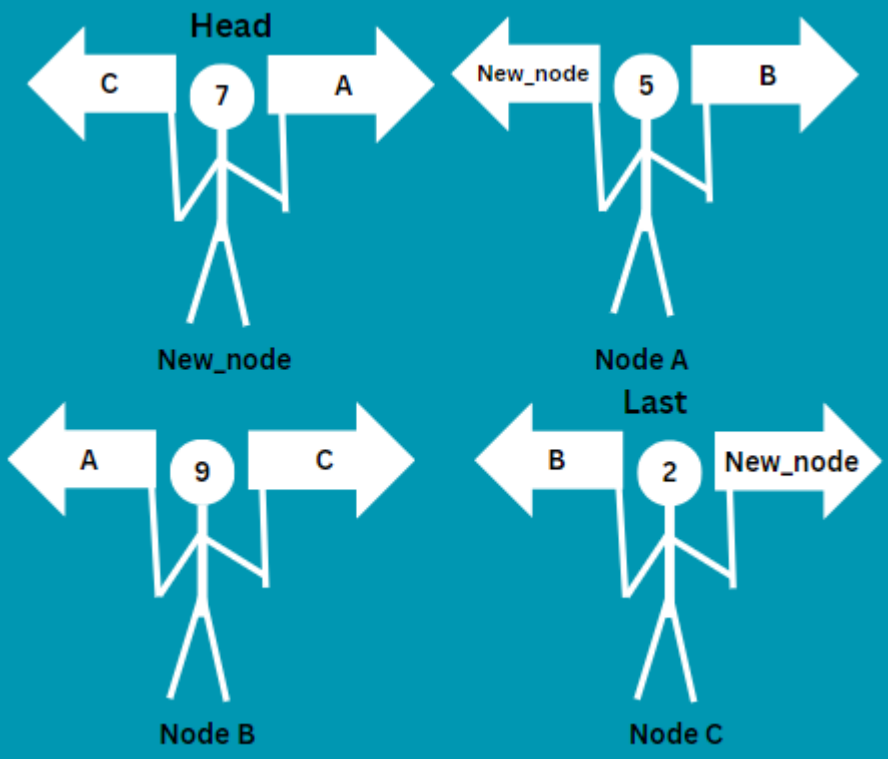

Head
New_node
5
B
Node A
A
9
C
Node B
Last
B
2
New_node
Node C
C
7
A
New_node

Step 5:


New_node
5
B
Node A
A
9
C
Node B
Last
B
2
New_node
Node C
Head
C
7
A
New_node

Step 6:

Return the head node



Code Implementation:

```
Node* insertAtBeginning(Node* head, int newData) {
  Node* newNode = new Node(newData);
   if (!head) {
       newNode->next = newNode->prev = newNode;
       head = newNode;
    } else {

       // List is not empty
         // Last node in the list
       Node* last = head->prev;

       // Insert new node
       newNode->next = head;
       newNode->prev = last;
       head->prev = newNode;
       last->next = newNode;
        head = newNode;
    }

    return head;
}
```

c) Insert at the end

We will create a new_node with the given value.

Assign the next of new_node as the start node.

Assign the prev of new_node to the last node of the list.

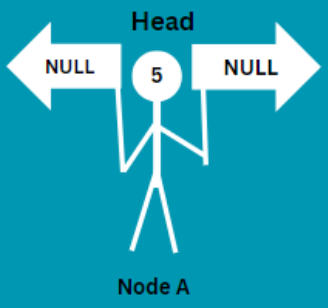
Update the prev of the start node and next of the last node as new_node.

Update new_node as the last node.

Using these above insertion operations, our Doubly Circular Linked List will get created.Now, finally, once we have inserted all the nodes in the list, we can print the list.

Case i.)

List is empty return new node



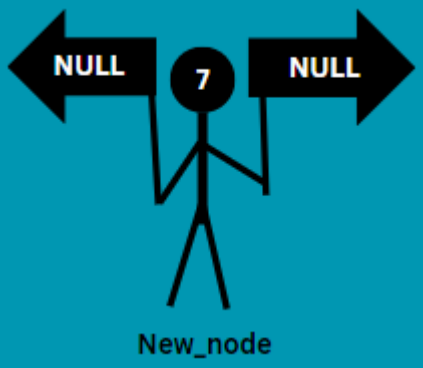
Insert at end(back)

Hence Proved

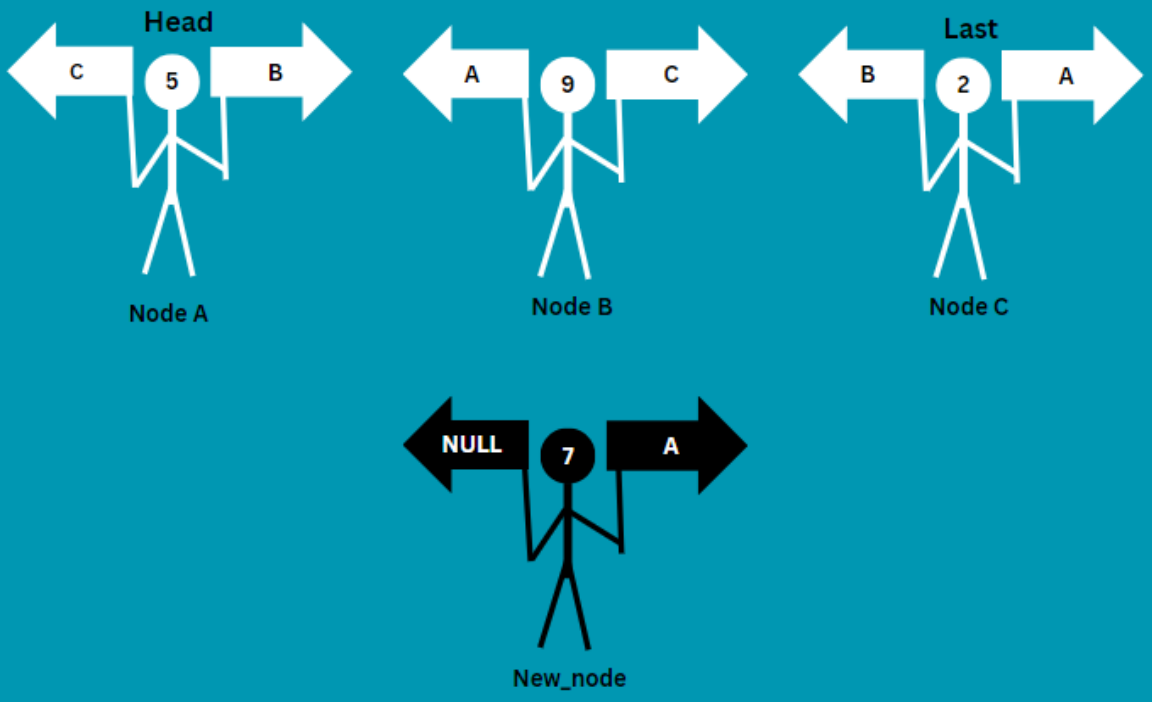
Case ii.)

Assume the value 7 is to be inserted at end;

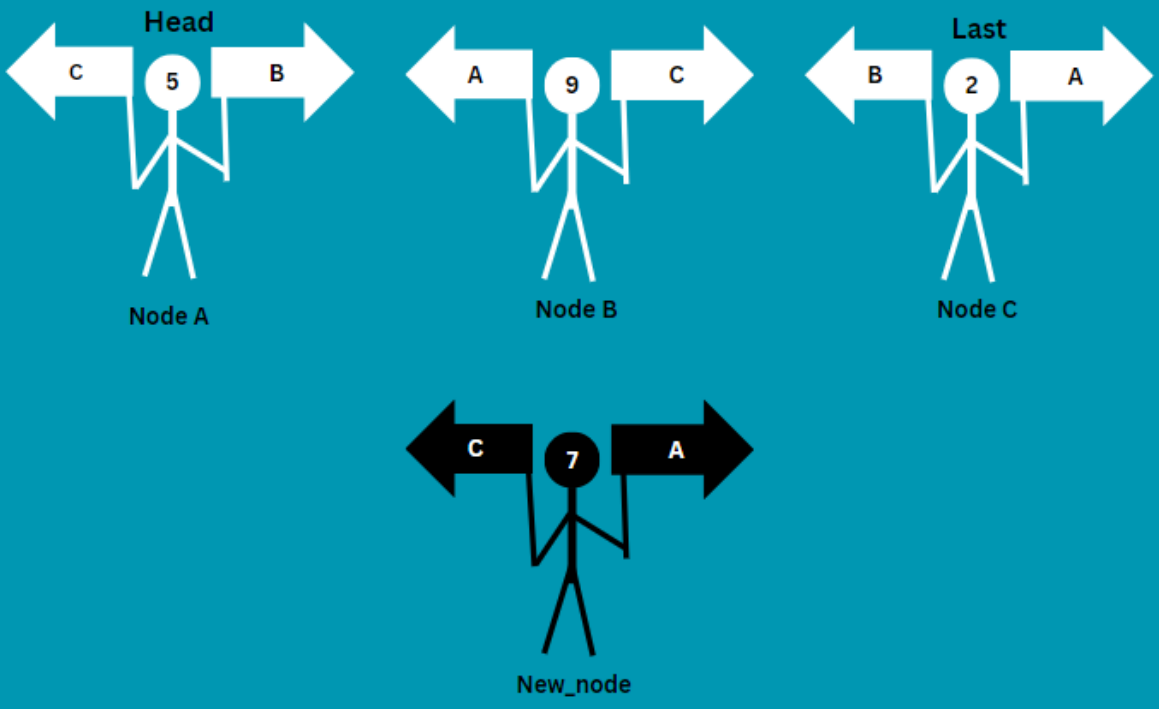
Step 1:



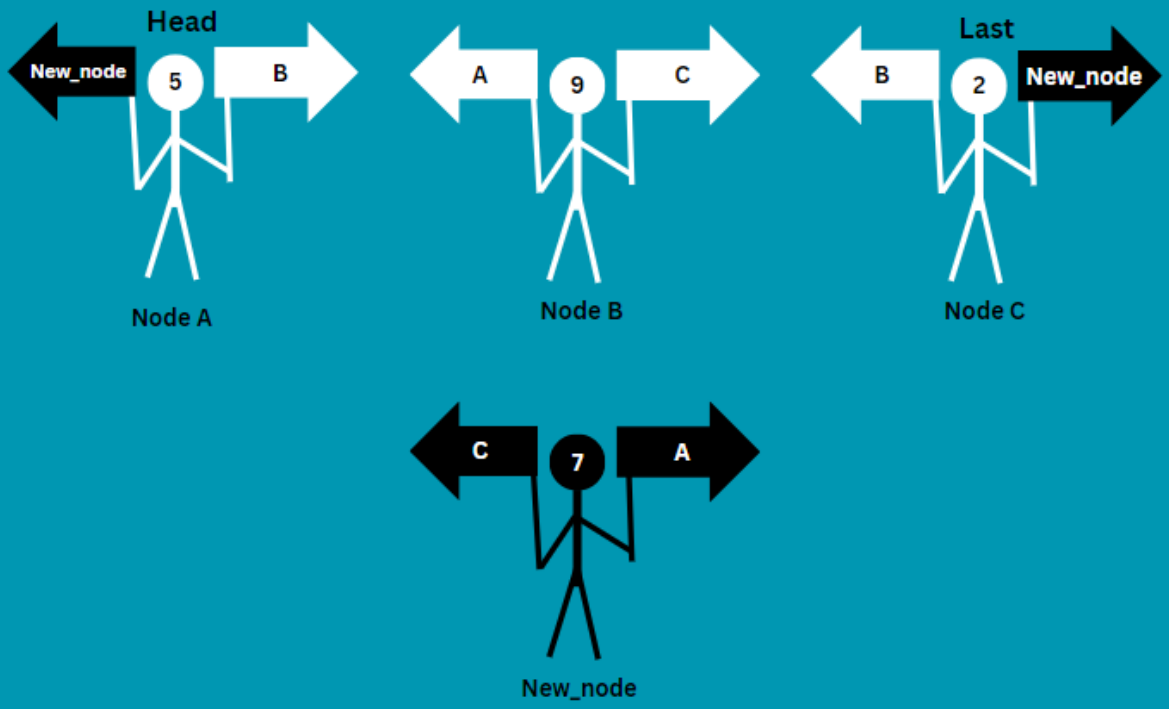
Step 2:



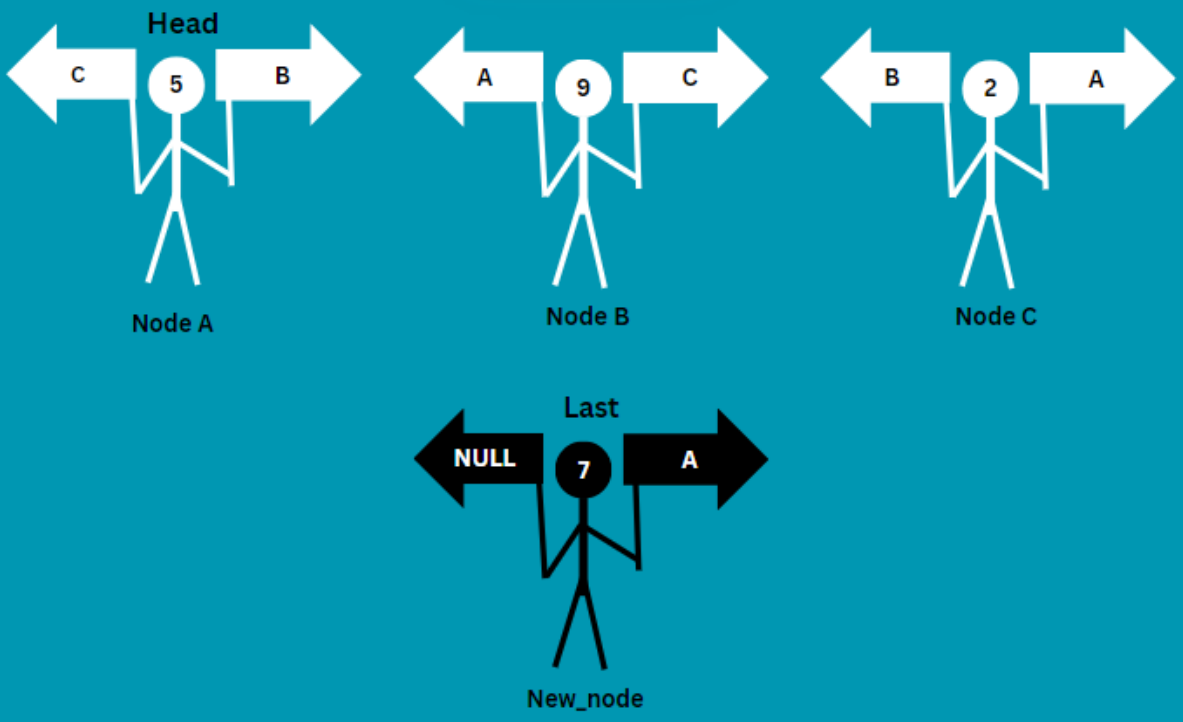
Step 3:


Head
C
5
B
Node A
A
9
C
Node B
Last
B
2
A
Node C
C
7
A
New_node

Step 4:

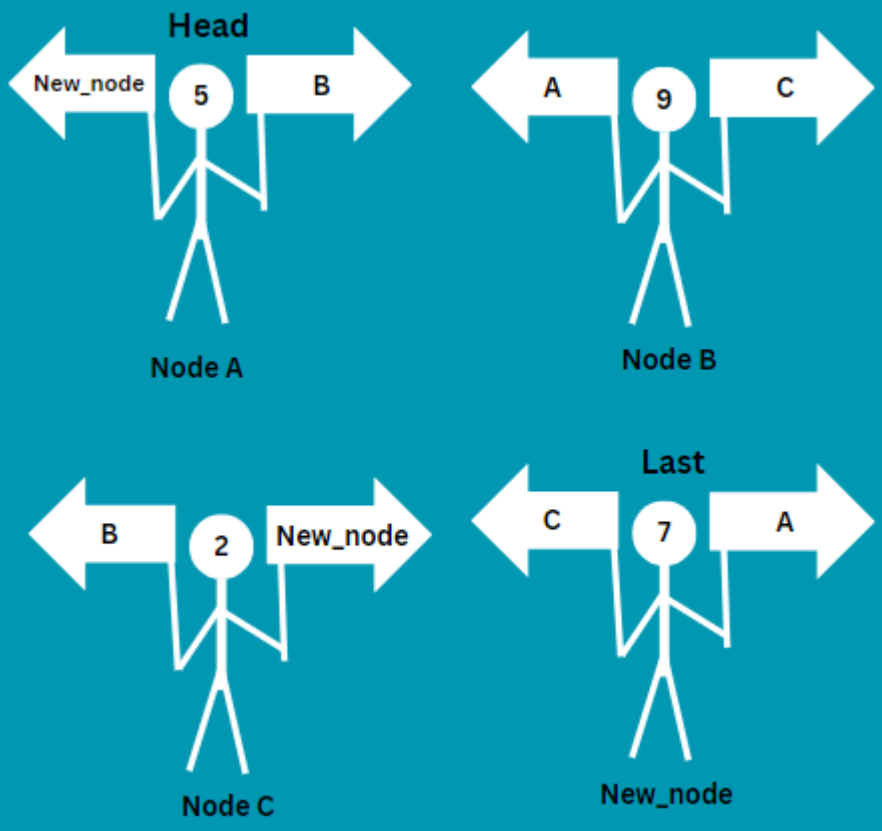

Head
New_node
5
B
Node A
A
9
C
Node B
Last
B
2
New_node
Node C
C
7
A
New_node

Step 5:



Step 6:

Return the head pointer;



Code Implementation:

```
Node* insertAtEnd(Node* head, int newData) {
    Node* newNode = new Node(newData);

    if (!head) {

        // List is empty
        newNode->next = newNode->prev = newNode;
        head = newNode;
    } else {

        // List is not empty
        Node* last = head->prev;

        // Insert new node at the end
        newNode->next = head;
        newNode->prev = last;
        last->next = newNode;
        head->prev = newNode;
    }

    return head;
}
```

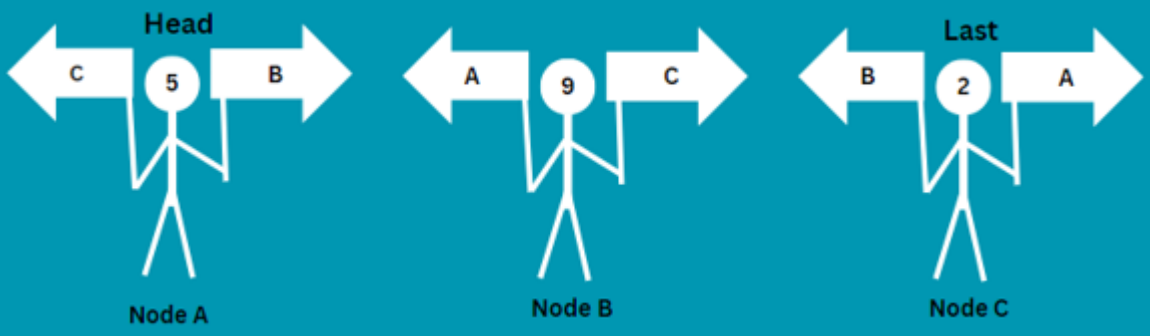
c.)Insert at a specific position:

- Initialize a pointer curr pointer to the head node and start traversing the list we reach the node just before the desired position. Use a counter to keep track of the curr position.
- Insert the New Node:
    - Set newNode->next to curr->next;
    - Set newNode->prev to curr.
    - Update curr->next->prev to newNode.
    - Update current->next to newNode.
- Update Head (if the insertion is at position 0 and the list is empty), set head to newNode.

Case i.)

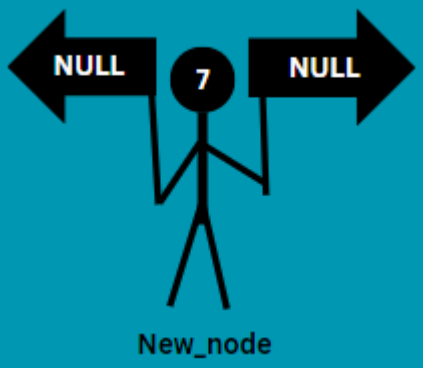
If position is 1 then do the process a[page.no:18];

Case ii.)



We want to insert a element at position 2.

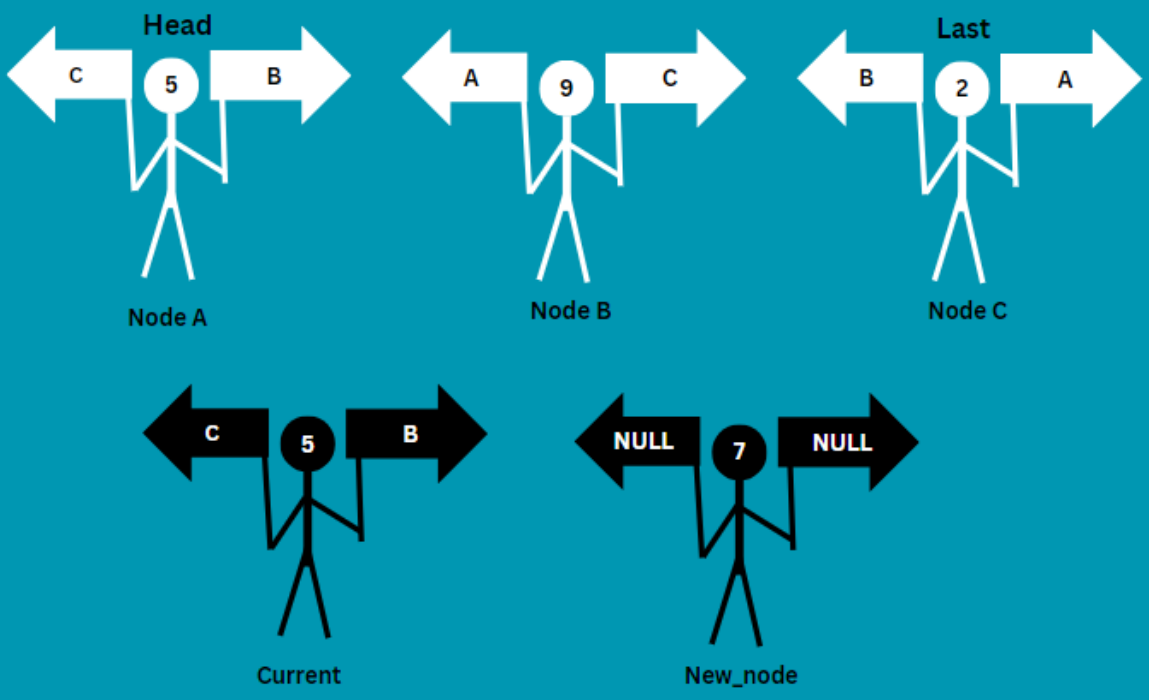
Step 1:



Step 2:

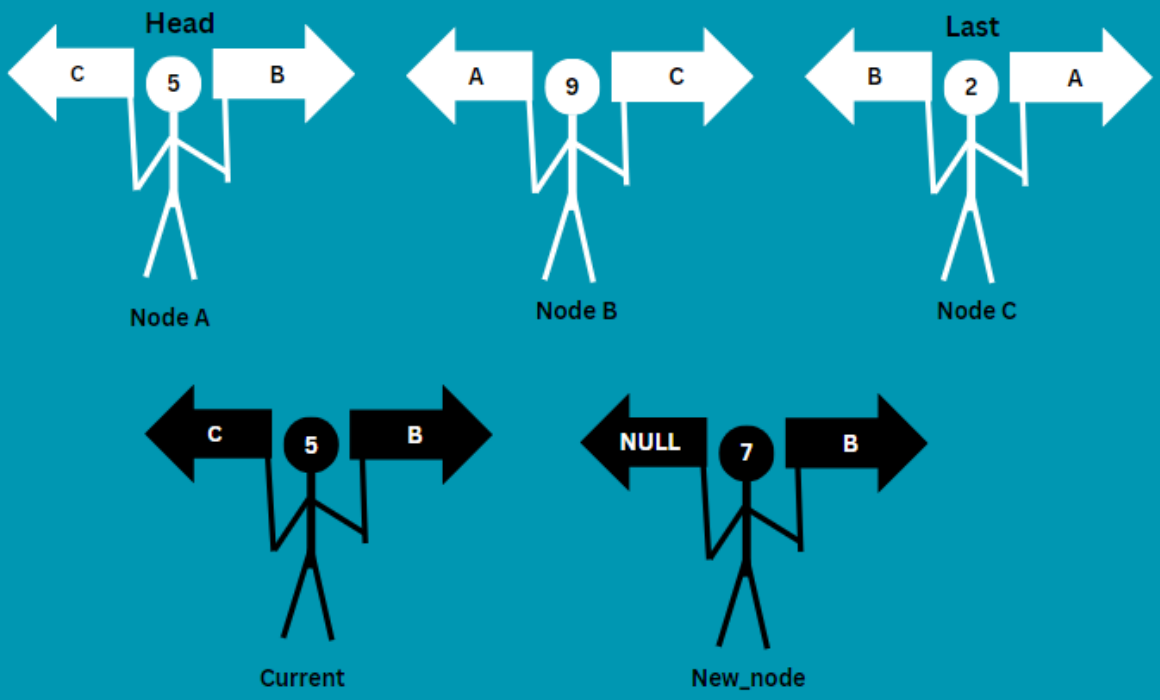
Traversal to the less than one of inserting position then:

Eg:Inserting at pos:2 traversal to pos:1:



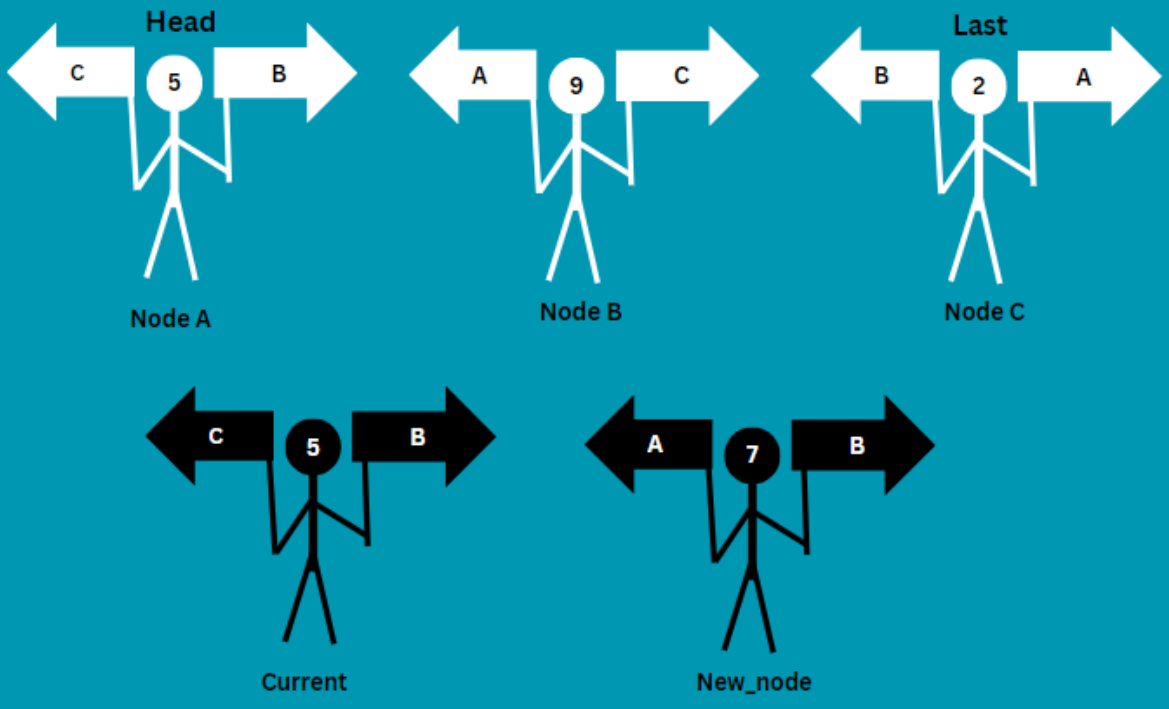
Step 3:

Setting newnode->next to current->next

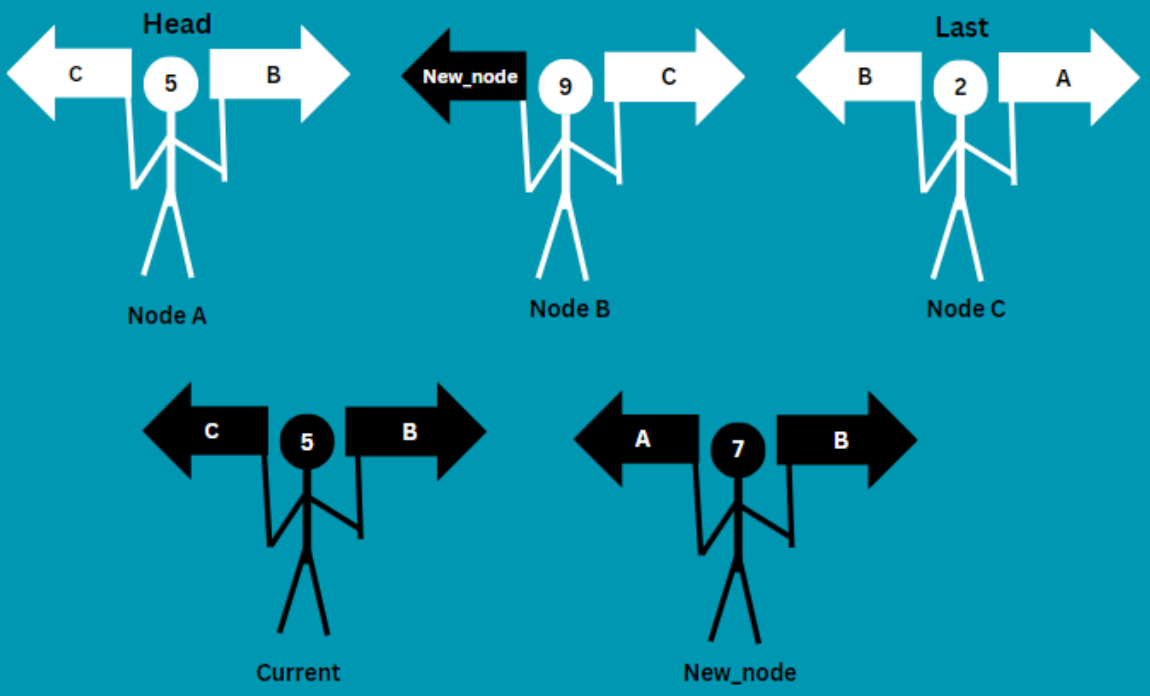


Step 4:

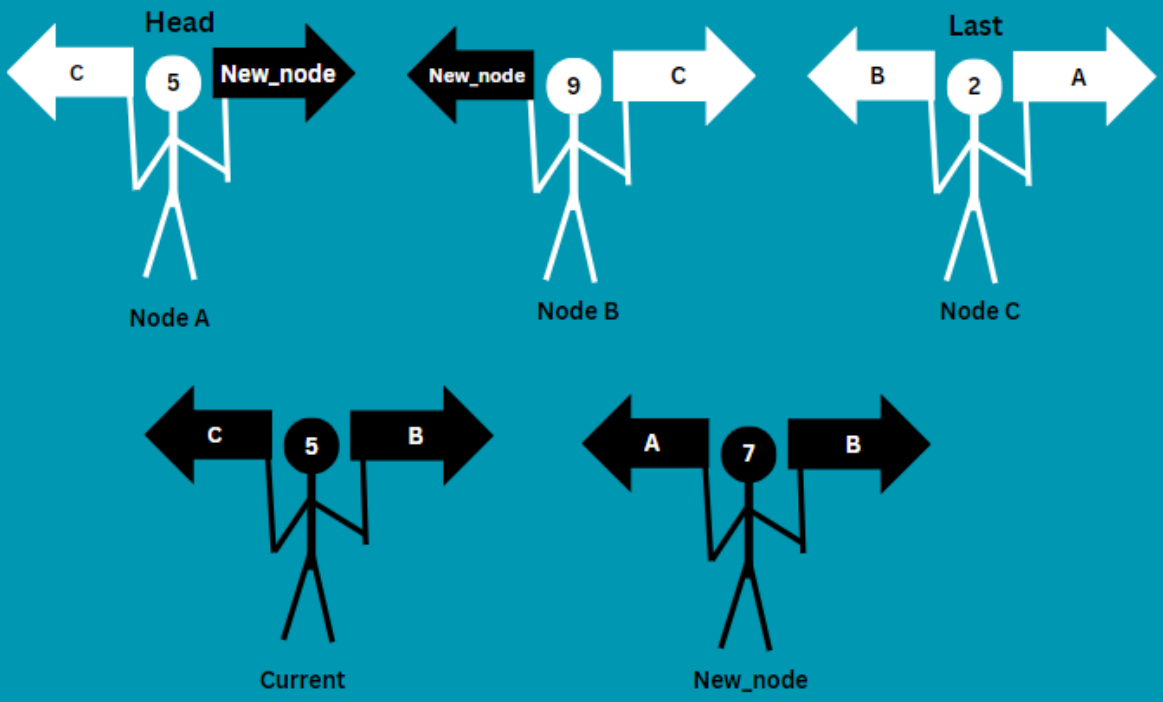
Setting new_node->prev to current;



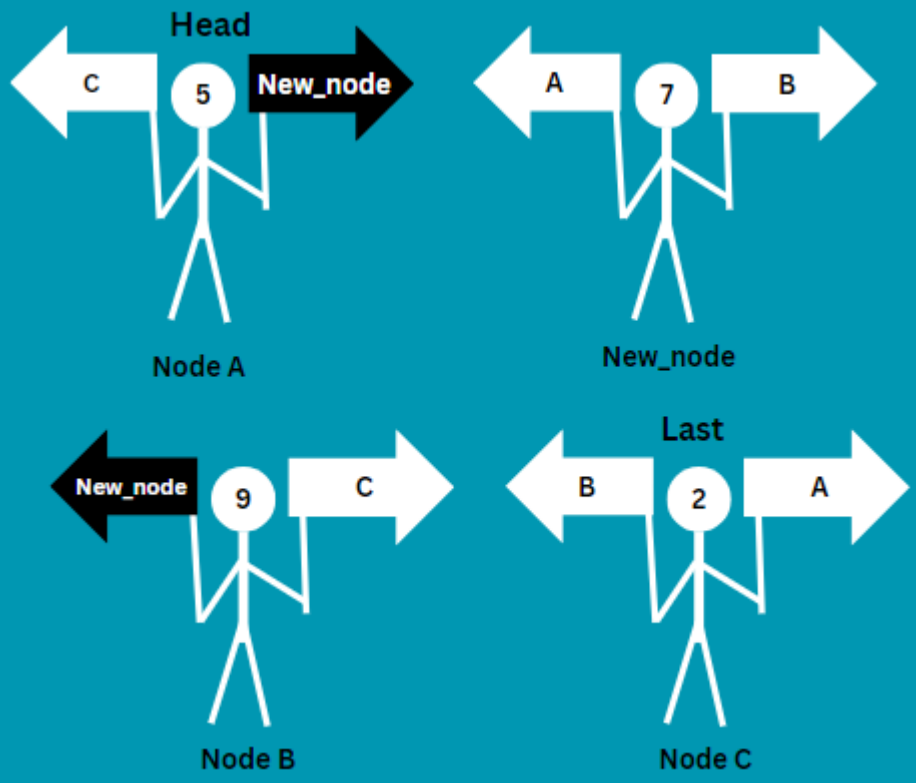
Step 5:


Head
C
5
B
Node A
New_node
9
C
Node B
Last
B
2
A
Node C
C
5
B
Current
A
7
B
New_node

Step 6:


Head
C
5
New_node
Node A
New_node
9
C
Node B
Last
B
2
A
Node C
C
5
B
Current
A
7
B
New_node

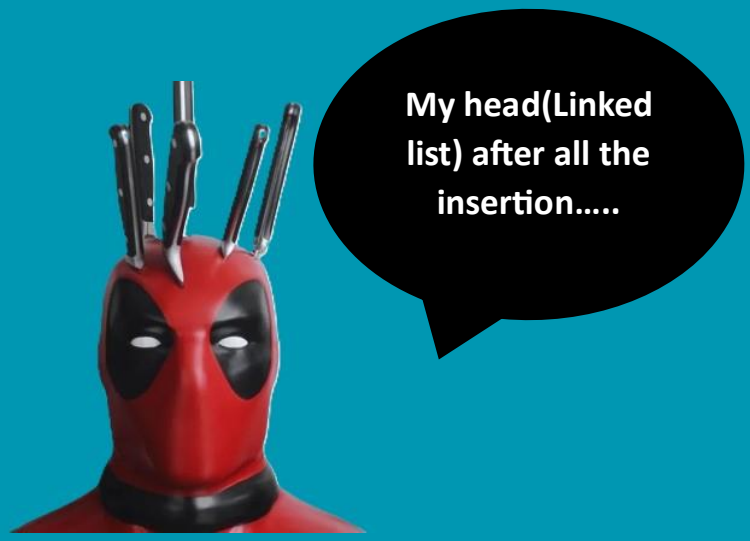
Step 7:



Code Implementation:

```
Node* addNode(Node* head, int pos, int newData) {
   Node* newNode = new Node(newData);
   if (!head) {
       if (pos > 1) {
          return nullptr;
      }
       // New node becomes the only node in the circular list
      newNode->prev = newNode;
       newNode->next = newNode;
       return newNode; }
```

```
if (pos == 1) {
    // Insert at the beginning of the list
    newNode->next = head;
    newNode->prev = head->prev;
    head->prev->next = newNode;
    head->prev = newNode;
    return newNode;
}
Node* curr = head;
for (int i = 1; i < pos - 1; i++) {
    curr = curr->next;
    if (curr == head) {
        cout << "Position out of range!" << endl;
        return head;
    }}
```



Deletion

Algorithm:

i.)**Check if the list is empty**:

- If the list is empty, return immediately because there's nothing to delete.

ii.)**Initialize pointers**:

- If the list is not empty, create two pointers:
  - current will point to the head node.
  - previous will point to NULL initially (it will track the previous node as we traverse).

iii.)**Traverse to find the node to delete**:

- Move through the list using the current pointer to find the node you want to delete.
- During each step of traversal, update the previous pointer to point to the node before current because it's a doubly circular linked list.

iv.)**Check if the node is the only one in the list**:

- If the node to be deleted is the only node in the list (i.e., current == head and current -> next == head), make the head NULL (as the list will be empty after deletion) and return.

v.)**If the node is the head node**:

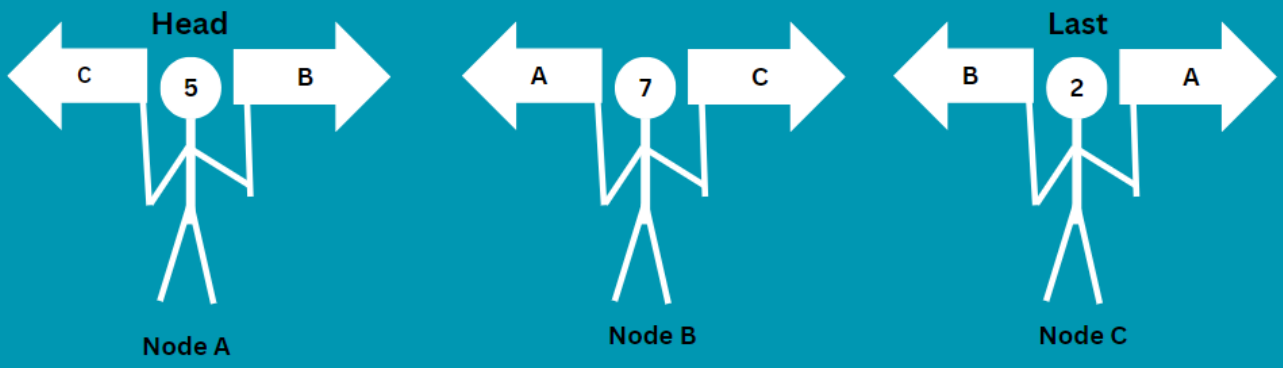
- If the node to delete is the head (i.e., current == head):

- Move the previous pointer to the last node in the list (since it's circular, the last node’s next points to the head).
- Update head to the next node (head = head -> next).
- Now, update the last node (previous) to point to the new head (previous -> next = head).
- Also, set the new head’s prev pointer to the last node (head -> prev = previous).
- Free the memory of the node that was deleted (current).

vi.) **If the node is neither the head nor the tail**:

- If the node is somewhere in the middle:
  - Store the next node after current in a temporary variable temp.
  - Update previous -> next = temp to bypass current.
  - Update temp -> prev = previous to connect temp back to previous.
  - Free the memory of the node pointed to by current.

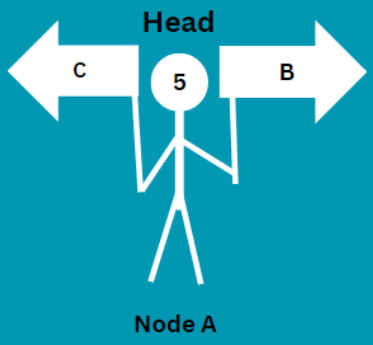
Initial list:



i.) List is empty:

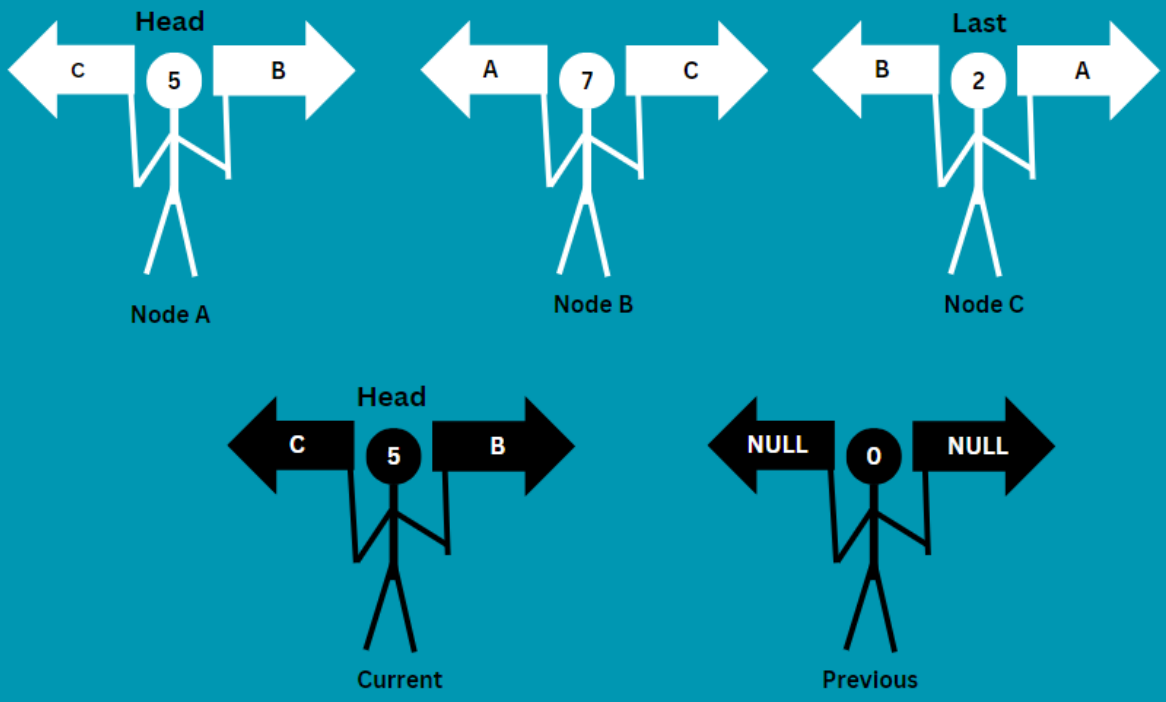
return null;

ii.)if only one Node:



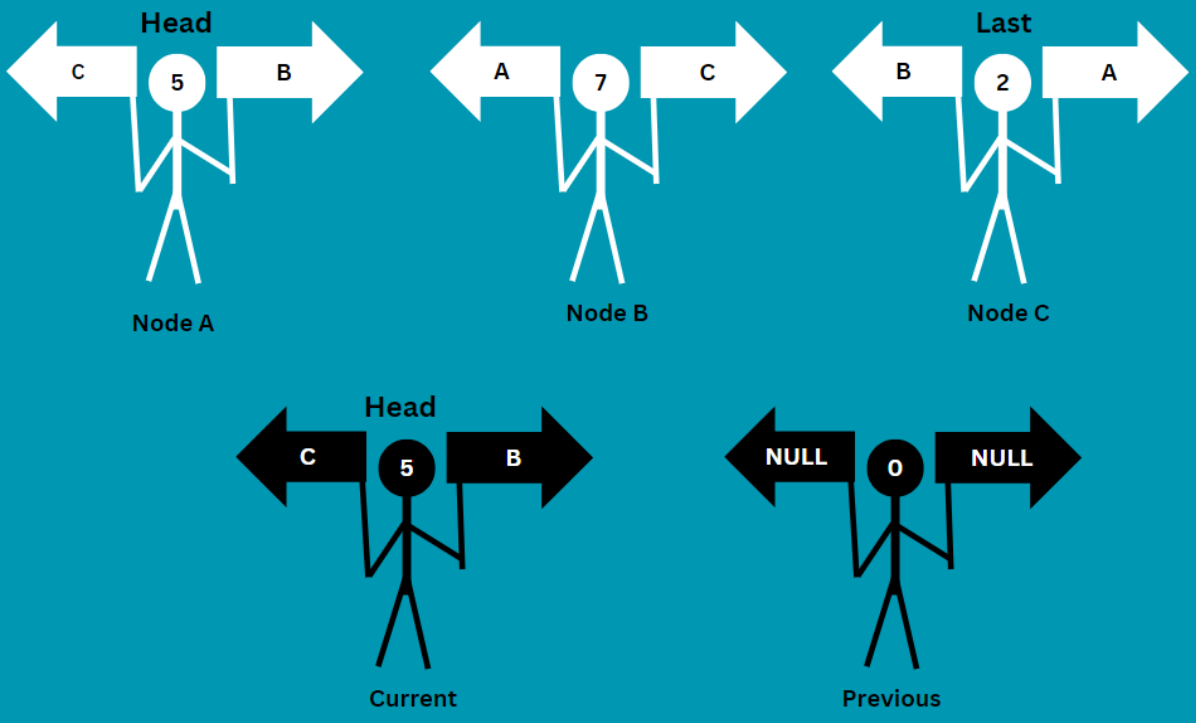
Return null;

iii.)Want to delete head:

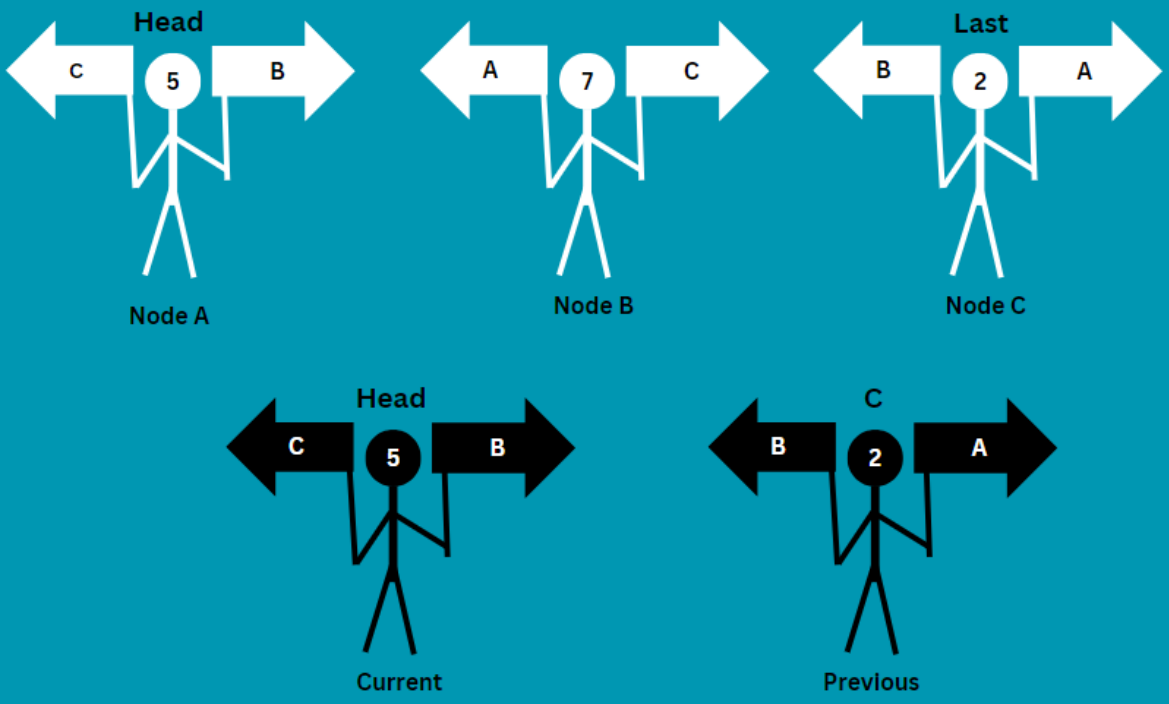


Step 1:

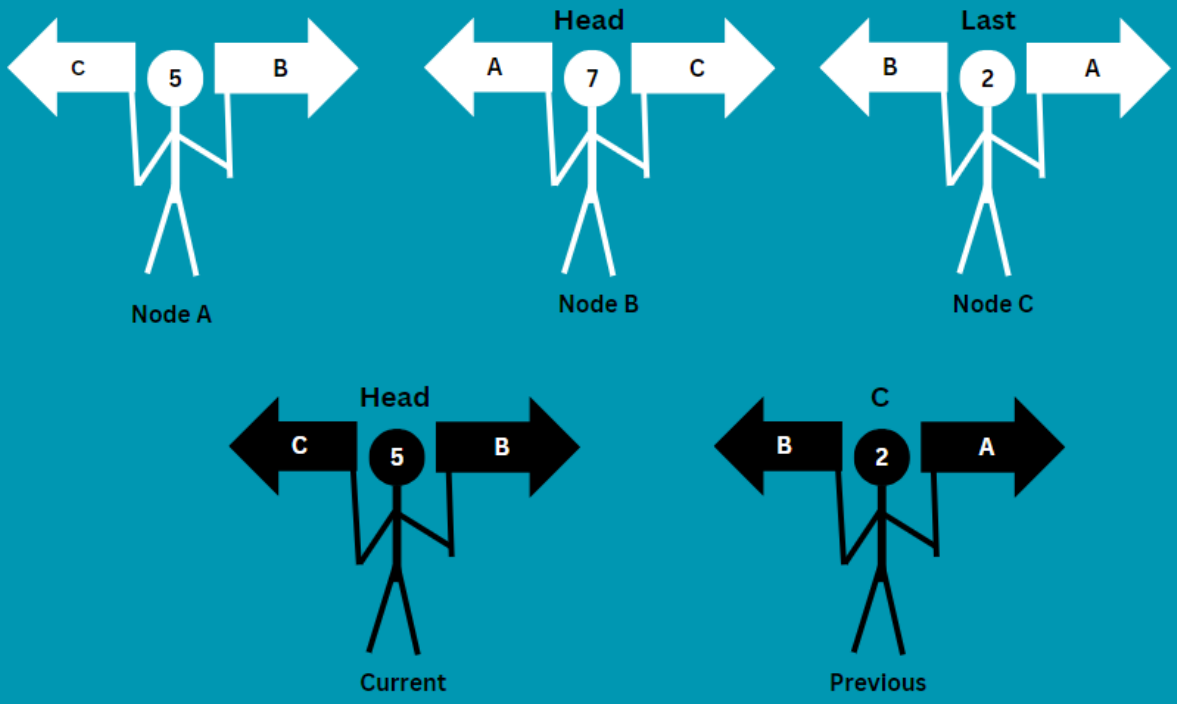
if many nodes are there have to delete the head:



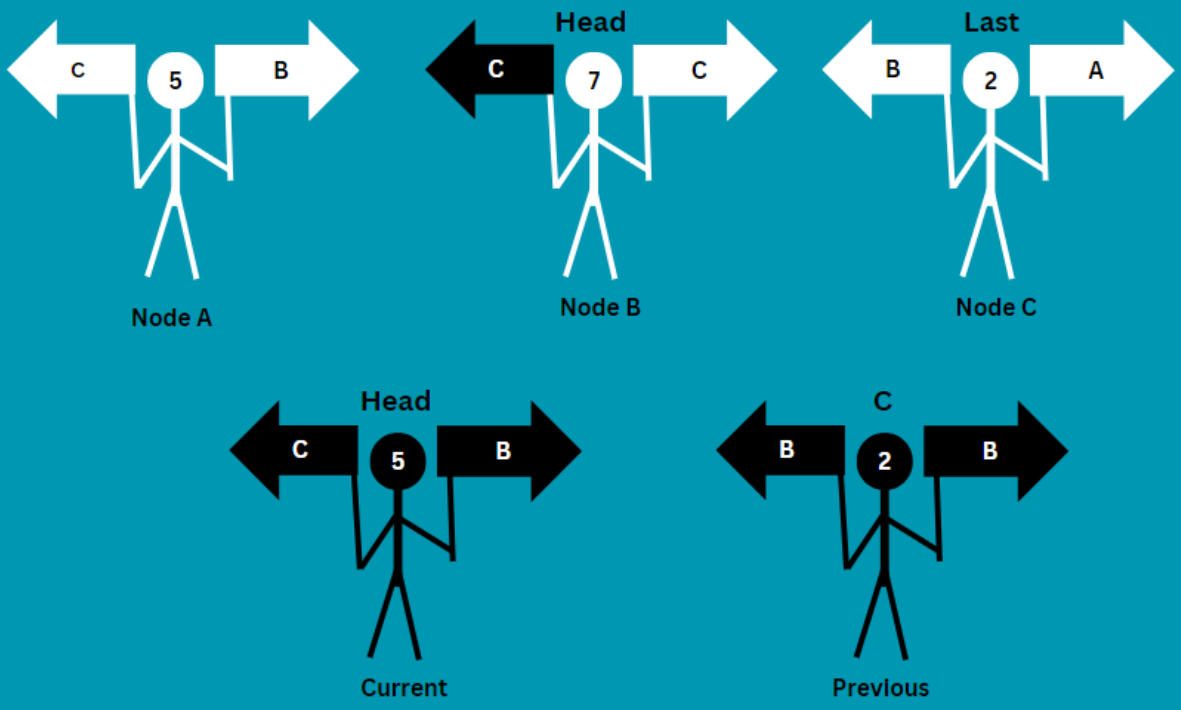
Step 2:


Head
C
5
B
Node A
A
7
C
Node B
Last
B
2
A
Node C
Head
C
5
B
Current
C
B
2
A
Previous

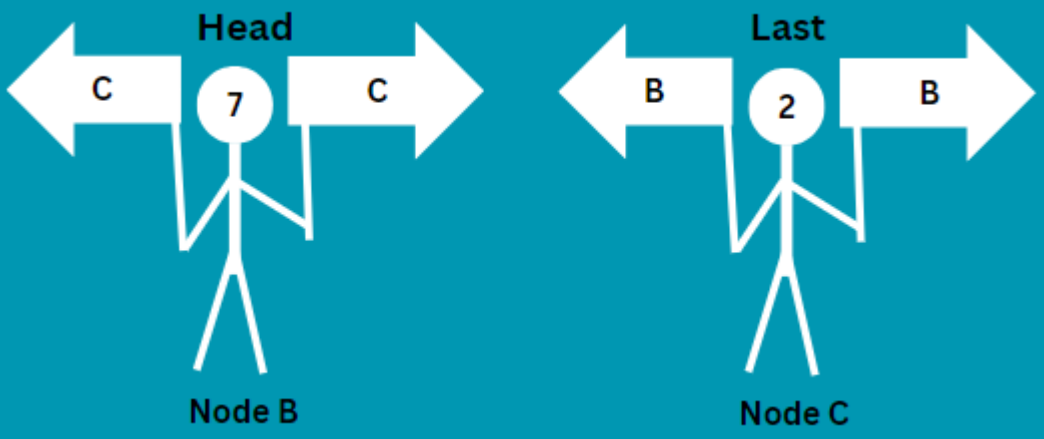
Step 3:


C
5
B
Node A
Head
A
7
C
Node B
Last
B
2
A
Node C
Head
C
5
B
Current
C
B
2
A
Previous

Step 4:

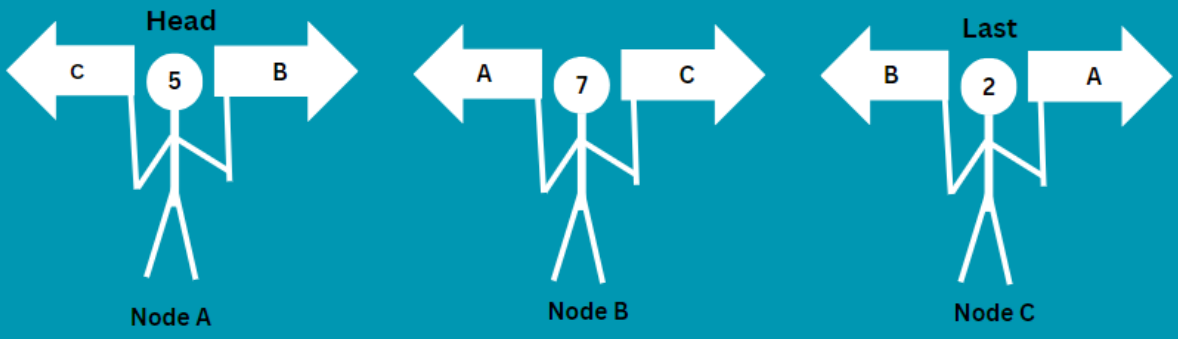

C
5
B
Node A
Head
C
7
C
Node B
Last
B
2
A
Node C
Head
C
5
B
Current
C
B
2
B
Previous

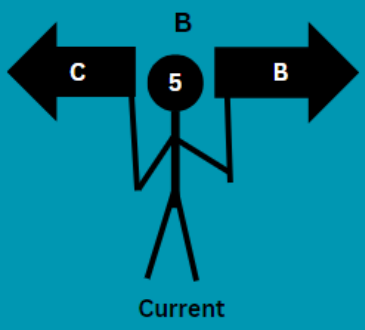
Step 5:


Head
C
7
C
Node B
Last
B
2
B
Node C

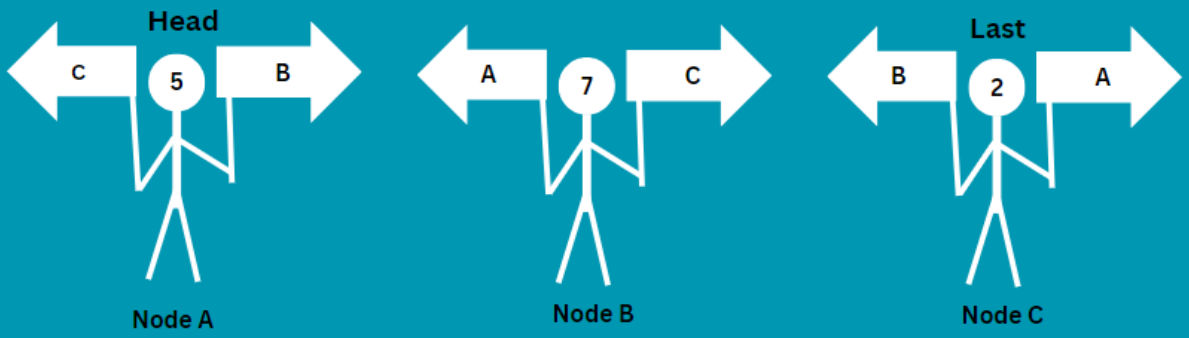
Case iv.)delete at middle and end;

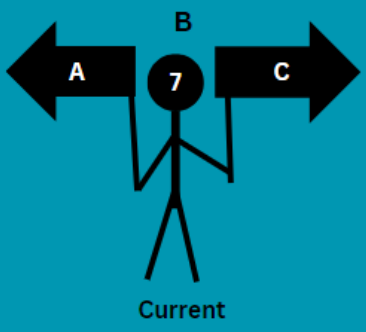
If using traversal we found the deleting element then:



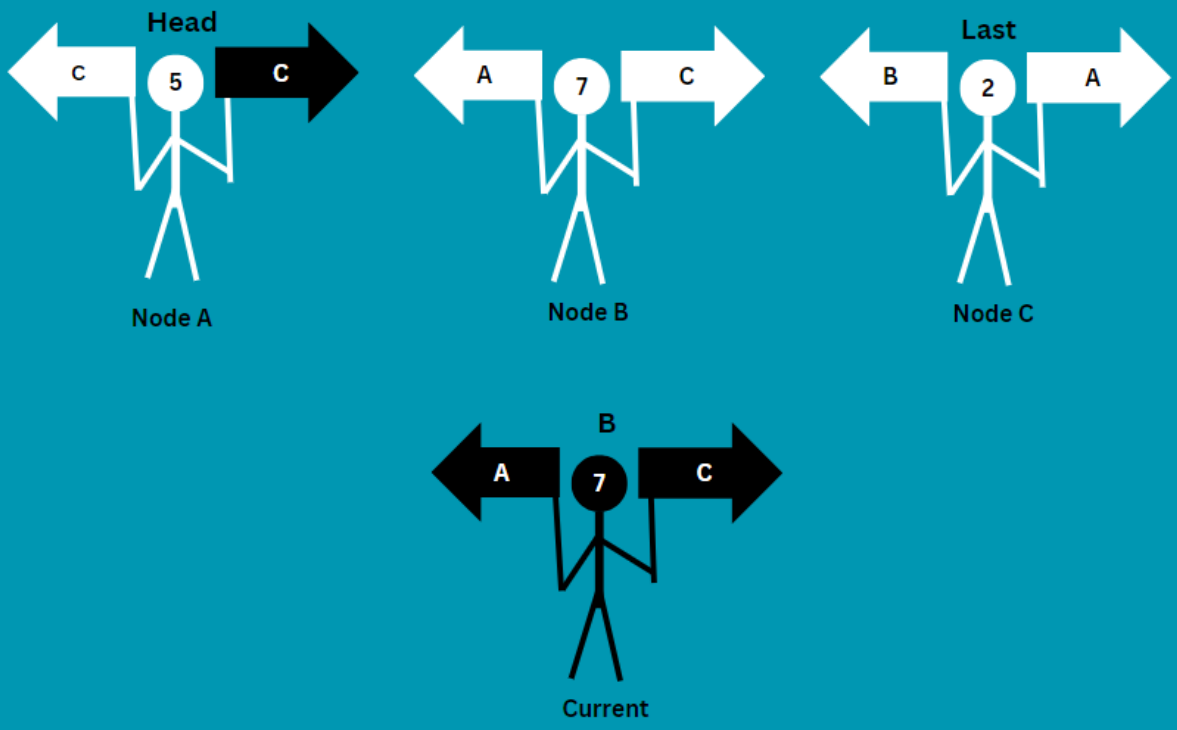


Step 1: traversal:

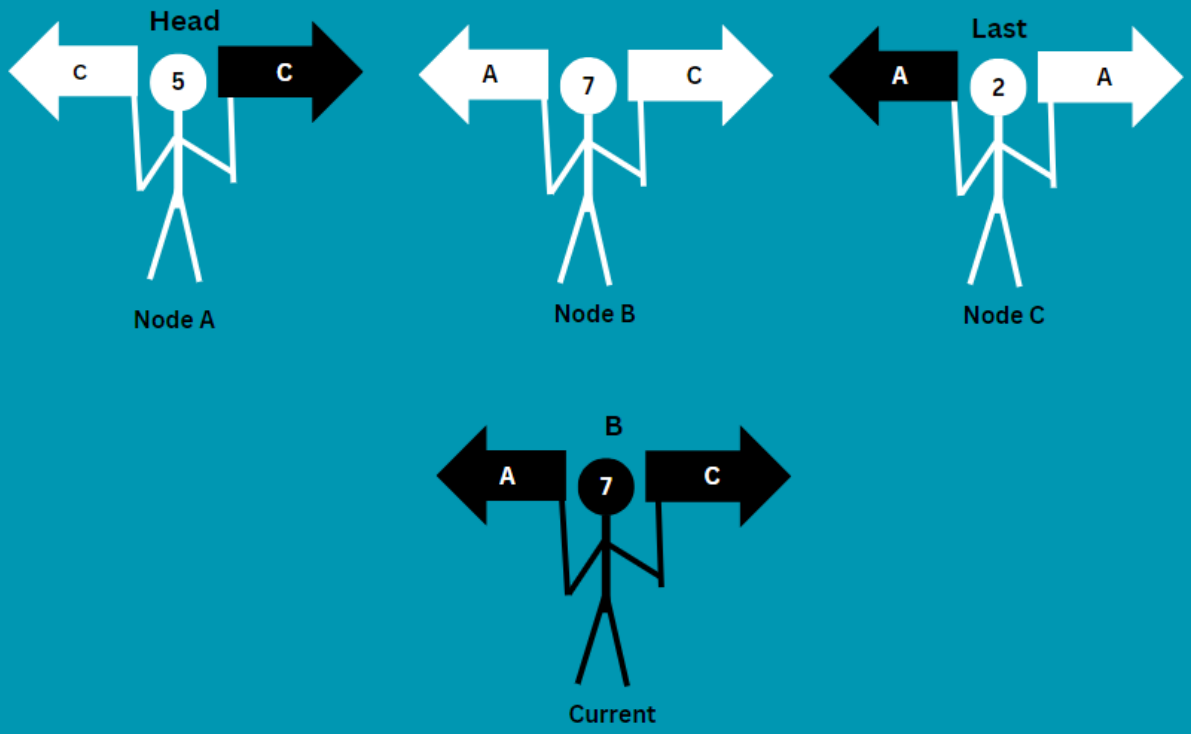




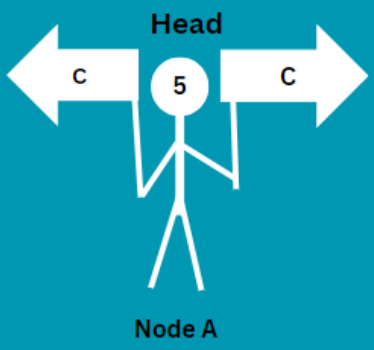
Step-3:

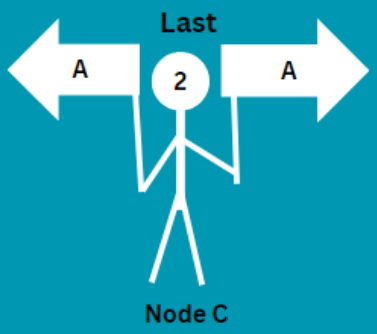

Head
C
5
C
Node A
A
7
C
Node B
Last
B
2
A
Node C
B
A
7
C
Current

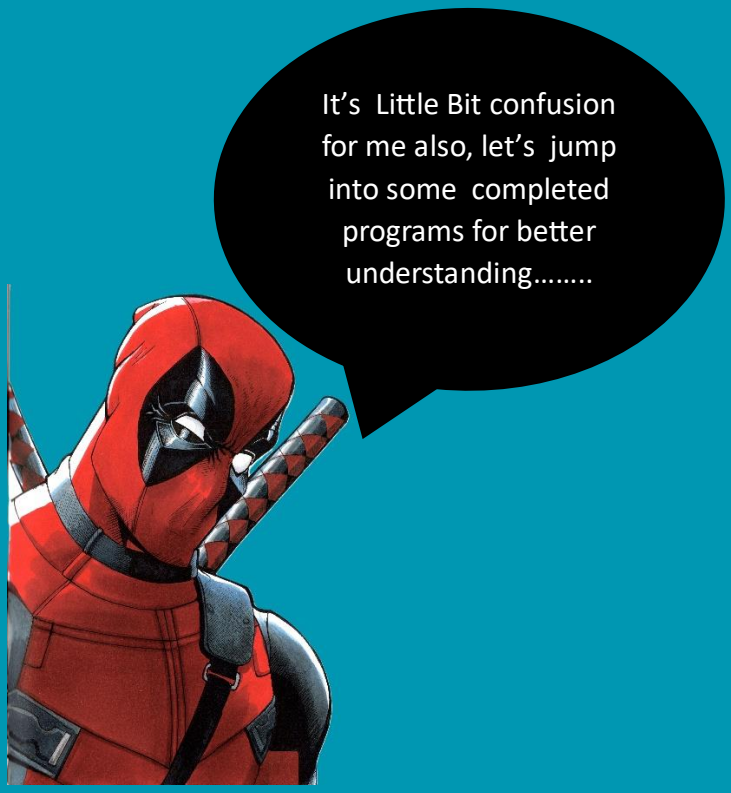
Step-4:


Head
C
5
C
Node A
A
7
C
Node B
Last
A
2
A
Node C
B
A
7
C
Current

Step-5:


Head
C
5
C
Node A


Last
A
2
A
Node C


It's Little Bit confusion for me also, let's jump into some completed programs for better understanding……..

Example Programs

1.)Inserting with a key:

```
#include <iostream>
using namespace std;

// Node structure for Circular Doubly Linked List
struct Node {
    int key;
    Node* next;
    Node* prev;
};

// Function to create a new node
Node* createNode(int key) {
    Node* newNode = new Node();
    newNode->key = key;
    newNode->next = newNode->prev = newNode;
    return newNode;
}

// Insert a node into the list
void insert(Node*& head, int key) {
```

```
    Node* newNode = createNode(key);
    if (!head) {
        head = newNode;
    } else {
        Node* tail = head->prev;
        tail->next = newNode;
        newNode->prev = tail;
        newNode->next = head;
        head->prev = newNode;
    }
}

// Display the list
void display(Node* head) {
    if (!head) {
        cout << "List is empty\n";
        return;
    }
    Node* temp = head;
    do {
        cout << temp->key << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
```

```
}

// Main function
int main() {
    Node* head = nullptr;

    // Insert nodes with keys
    insert(head, 10);
    insert(head, 20);
    insert(head, 30);

    // Display the list
    cout << "Circular Doubly Linked List: ";
    display(head);

    return 0;
}
```

2.)Program for Removing Duplicates:

```
#include <iostream>
using namespace std;
struct Node {
    int key;
    Node* next;
    Node* prev;
};

// Function to create a new node
Node* createNode(int key) {
    Node* newNode = new Node();
    newNode->key = key;
    newNode->next = newNode->prev = newNode;
    return newNode;
}

// Insert a node into the list
void insert(Node*& head, int key) {
    Node* newNode = createNode(key);
    if (!head) {
        head = newNode;
    } else {
        Node* tail = head->prev;
```

```
        tail->next = newNode;
        newNode->prev = tail;
        newNode->next = head;
        head->prev = newNode;
    }
}

// Display the list
void display(Node* head) {
    if (!head) {
        cout << "List is empty\n";
        return;
    }
    Node* temp = head;
    do {
        cout << temp->key << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

// Remove duplicate nodes
void removeDuplicates(Node* head) {
    if (!head) return;
```

```
    Node* current = head;
    do {
        Node* nextNode = current->next;
        while (nextNode != head) {
            if (current->key == nextNode->key) {
                Node* delNode = nextNode;
                nextNode->prev->next = nextNode->next;
                nextNode->next->prev = nextNode->prev;
                nextNode = nextNode->next;
                delete delNode;
            } else {
                nextNode = nextNode->next;
            }
        }
        current = current->next;
    } while (current != head);
}

// Main function
int main() {
    Node* head = nullptr;

    // Insert nodes with duplicates
    insert(head, 10);
```

```
    insert(head, 20);
    insert(head, 30);
    insert(head, 20); // Duplicate

    cout << "Original List: ";
    display(head);

    // Remove duplicates
    removeDuplicates(head);
    cout << "After removing duplicates: ";
    display(head);

    return 0;
}
```

Note:

Store the address of the node,

If the address is not stored then it cannot be find at allllll…….

3.)Program for Splitting Linked list by alternative nodes and sum each one :

```
#include <iostream>
using namespace std;

// Node structure for Circular Doubly Linked List
struct Node {
    int key;
    Node* next;
    Node* prev;
};

// Function to create a new node
Node* createNode(int key) {
    Node* newNode = new Node();
    newNode->key = key;
    newNode->next = newNode->prev = newNode;
    return newNode;
}

// Insert a node into the list
void insert(Node*& head, int key) {
    Node* newNode = createNode(key);
    if (!head) {
```

```
        head = newNode;
    } else {
        Node* tail = head->prev;
        tail->next = newNode;
        newNode->prev = tail;
        newNode->next = head;
        head->prev = newNode;
    }
}

// Display the list
void display(Node* head) {
    if (!head) {
        cout << "List is empty\n";
        return;
    }
    Node* temp = head;
    do {
        cout << temp->key << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
```

```
// Separate alternate nodes into two lists
void separateAlternateNodes(Node*& head, Node*& list1, Node*& list2) {
    if (!head) return;

    Node* temp = head;
    bool toggle = true; // For alternating between lists

    do {
        if (toggle) {
            insert(list1, temp->key); // Add to list1
        } else {
            insert(list2, temp->key); // Add to list2
        }
        toggle = !toggle; // Switch for the next node
        temp = temp->next;
    } while (temp != head);
}

// Main function
int main() {
    Node* head = nullptr;
    Node* list1 = nullptr;
    Node* list2 = nullptr;
```

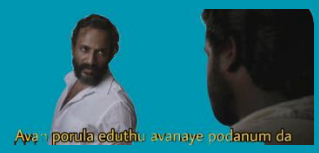
```
    // Insert nodes
    insert(head, 10);
    insert(head, 20);
    insert(head, 30);
    insert(head, 40);

    cout << "Original List: ";
    display(head);

    // Separate alternate nodes
    separateAlternateNodes(head, list1, list2);
    cout << "List 1 (Alternate nodes): ";
    display(list1);
    cout << "List 2 (Alternate nodes): ";
    display(list2);

    return 0;
}
```

Exercise Problems

1. Given a circular doubly linked list consisting of N nodes, the task is to modify every node of the given Linked List such that each node contains the sum of all nodes except that node.
2. Write a C++ program to rotate a circular doubly linked list to the right by k position
3. Implement a C++ program that counts the number of occurrences of a given key in a circular doubly linked list.
4. Write a C++ program that finds the kth node from the end of a circular doubly linked list

Main link:

https://www.youtube.com/@danitha7169



Reference links:

https://youtu.be/3ZrkixbHCTI?si=cgbGhuYiX-fdAr6h

https://youtu.be/Ynjr36a0NFU?si=d9YqiImOt_zepKS1

https://youtu.be/gF52eWekqz4?si=3dpVUITPk2U0VcSt

https://youtu.be/hzvneUGiWj4?si=VZg-D_P4gz4zrLi7

https://youtu.be/xiLKeIHAqdc?si=7mT2HlbOJmnptjM4

Readers After Reading: