## **LEC-5 Circular Linked List**

# **Circular Linked List: Overview**

A Circular Linked List is a variant of the traditional Linked List where the last node points back to the first node, forming a circular structure. This data structure offers unique advantages and use cases compared to linear linked lists.

# **Advantages of Circular Linked List**

Circular Linked Lists provide specific benefits owing to their circular nature:

* **Efficient Rotations:** Elements can be rotated within a Circular Linked List by adjusting pointers, making it useful for applications like round-robin scheduling.
* **Constant-time Insertions and Deletions**: Insertions and deletions at the beginning and end of the list are consistent operations, unlike linear linked lists where end operations can be costly.
* **Circular Data Representation**: Circular data relationships are naturally represented using Circular Linked Lists.

# **Node Structure**

Each node in a Circular Linked List retains the structure of a standard linked list node, including:

* **Data:** The actual value or information stored within the node.
* **Next Pointer:** A pointer to the next node in the list.

# **Circular Linked List Operations**

**1. Insertion**

Circular Linked List insertion cases include:

* **Insertion at the Beginning:** Adjust the new node's next pointer and update the last node's next pointer to the new node.
* **Insertion at the End:** Update the new node's next pointer to the first node and update the last node's next pointer to the new node.

**2. Deletion**

Deletion operations are similar to linear linked lists:

* **Deletion from the Beginning:** Update the head pointer and adjust the last node's next pointer if necessary.
* **Deletion from the End:** Traverse to the node before the last node, update its next pointer to the first node, and update the last node pointer.

**3. Traversal**

Traversal in a Circular Linked List is similar to linear linked lists, but it involves a termination condition based on reaching the starting node again.

**4. Rotations**

Circular Linked Lists are uniquely suited for rotation operations:

* **Right Rotation:** Update the head pointer to the next node and adjust the last node pointer accordingly.
* **Left Rotation:** Update the last node's next pointer to the second node, update the head pointer to the second node, and adjust the second node's previous pointer.

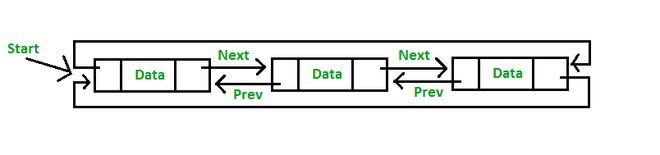
# **Use Cases**

Circular Linked Lists have applications in various scenarios:

* **Round-Robin Scheduling:** Circular Linked Lists are used in scheduling algorithms where tasks are rotated in a circular manner.
* **Music Playlists:** Circular Linked Lists can model playlists where the last song connects back to the first, creating a seamless loop.
* **Memory Management:** They can be used to manage memory blocks in a circular buffer.

# **Conclusion**

In conclusion, a Circular Linked List offers unique advantages such as efficient rotations, constant-time insertions/deletions at both ends, and natural circular data representation. Its circular structure allows for interesting use cases, making it a valuable variation of the traditional linked list.



using namespace std;

#include <iostream>

class circularlinklist; // Forward declaration

class Node {

    friend class circularlinklist;

    int data;

    Node\* next;

};

class circularlinklist {

private:

    Node\* head;

public:

    circularlinklist() {

        head = nullptr;

    }

    ~circularlinklist() {

        // TODO: Implement destructor to free memory used by the list nodes

    }

    bool insertatstart(int val) {

        Node\* newNode = new Node();

        newNode->data = val;

        newNode->next = newNode; // Circular connection

        if (head == nullptr) {

            head = newNode;

        } else {

            Node\* current = head;

            while (current->next != head) {

                current = current->next;

            }

            current->next = newNode;

            newNode->next = head;

            head = newNode;

        }

        return true;

    }

    bool insertatend(int val) {

        Node\* newNode = new Node();

        newNode->data = val;

        newNode->next = newNode; // Circular connection

        if (head == nullptr) {

            head = newNode;

        } else {

            Node\* current = head;

            while (current->next != head) {

                current = current->next;

            }

            current->next = newNode;

            newNode->next = head;

        }

        return true;

    }

    bool deletefromstart() {

        if (head == nullptr) {

            return false;

        }

        Node\* temp = head;

        if (head->next == head) { // Only one node

            head = nullptr;

        } else {

            Node\* current = head;

            while (current->next != head) {

                current = current->next;

            }

            current->next = head->next;

            head = head->next;

        }

        delete temp;

        return true;

    }

    bool deletefromend() {

        if (head == nullptr) {

            return false;

        }

        if (head->next == head) { // Only one node

            delete head;

            head = nullptr;

            return true;

        }

        Node\* prev = nullptr;

        Node\* current = head;

        while (current->next != head) {

            prev = current;

            current = current->next;

        }

        prev->next = head;

        delete current;

        return true;

    }

    void display() {

        if (head == nullptr) {

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

            return;

        }

        Node\* current = head;

        do {

            cout << current->data << " ";

            current = current->next;

        } while (current != head);

        cout << endl;

    }

};

int main() {

    circularlinklist list;

    list.insertatstart(3);

    list.insertatstart(2);

    list.insertatstart(1);

    list.insertatend(4);

    list.insertatend(5);

    cout << "Original List: ";

    list.display();

    list.deletefromstart();

    list.deletefromend();

    cout << "Modified List: ";

    list.display();

    return 0;

}