Data Structure Algorithms & Applications (CT-159)
Lab 03

Types of Linked List

Objectives

The objective of the lab is to get students familiar with Circular and Doubly Linked List.

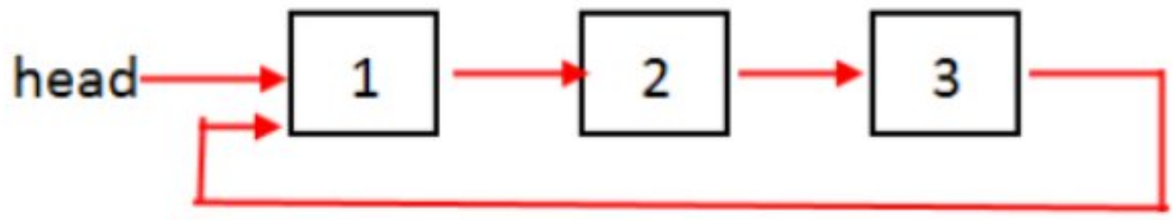
Tools Required

Dev C++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Circular Linked List

A circular linked list is a variation of a linked list where the last node points back to the first node, creating a circular structure. This allows for continuous traversal of the list without ever encountering a nullptr, as is the case with linear linked lists.



Circular Linked List

Properties of a Circular Linked List:

- Circular Nature: The next pointer of the last node points to the first node, forming a circular loop.
- No nullptr at the End: Unlike a singly linked list, where the last node points to nullptr, a circular linked list loops back to the first node.
- Traversal: You can start at any node and traverse the entire list. It's common to use an additional condition to stop traversing (such as visiting the starting node again).
- Single Entry Point: Usually, a circular linked list is referenced by a pointer to the last node (so that both the head and tail can be accessed efficiently) or the head node.
- Efficient Operations: Insertion and deletion operations are efficient, particularly if the list is used in scenarios like a queue or round-robin scheduling.

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

// Node structure
class Node {
        public:
   int data;
   Node* next;
   Node(int d){
        data=d;
        next=NULL;
        }
};

// Class to represent a circular linked list
class CircularLinkedList {
private:
   Node* tail; // Points to the tail node in the list

public:
   CircularLinkedList() : tail(NULL) {}

   void insertAtEnd(int value) {
      Node* newNode = new Node(value);
      if (tail == NULL) {     // List is empty, create a single node circular list
         tail = newNode;
         tail->next = tail;
      } else {
         // Insert after the tail node
         newNode->next = tail->next;  // Point new node to head
         tail->next = newNode;       // Update tail node's next
         tail = newNode;            // Update the tail pointer
      }
   }

   void insertAtBeginning(int value) {
      Node* newNode = new Node(value);
      if (tail == NULL) {           // List is empty, create a single node circular list
         tail = newNode;
         tail->next = tail;
```

```
    } else {          // Insert new node after the tail node and update head
        newNode->next = tail->next;
        tail->next = newNode;
    }
}

void display() const {
    if (tail == NULL) {
        cout << "List is empty." << endl;
        return;
    }

    Node* temp = tail->next;  // Start from the head
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != tail->next);  // Stop when we reach the head again

    cout << endl;
}

void deleteNode(int key) {
    if (tail == NULL) {
        cout << "List is empty, cannot delete." << endl;
        return;
    }

    // Single node case
    if (tail == tail->next && tail->data == key) {
        delete tail;
        tail = NULL;
        return;
    }

    // Traverse the list to find the node to be deleted
    Node* temp = tail->next;
    Node* prev = tail;

    // Case: Deleting the head node
    if (temp->data == key) {
        prev->next = temp->next;
        delete temp;
        return;
    }

    // General case: find the node to delete
    do {
        prev = temp;
        temp = temp->next;
    } while (temp != tail->next && temp->data != key);

    if (temp->data == key) {
        prev->next = temp->next;
        if (temp == tail) {
            tail = prev;  // If the tail node is deleted, update the tail pointer
        }
        delete temp;
    } else {
        cout << "Node with value " << key << " not found." << endl;
    }
}

~CircularLinkedList() {
    if (tail != NULL) {
        Node* current = tail->next;
        Node* nextNode;
        while (current != tail) {
            nextNode = current->next;
            delete current;
```

```
            current = nextNode;
        }
        delete tail;
    }
  }
};

int main() {
  CircularLinkedList cll;

  cll.insertAtEnd(10);
  cll.insertAtEnd(20);
  cll.insertAtEnd(30);
  cll.insertAtBeginning(5);

  cout << "Circular Linked List: ";
  cll.display();

  cll.deleteNode(20);
  cout << "After deleting 20: ";
  cll.display();

  return 0;
}
```

Circular vs. Singly Linked List

The circular linked list offers several advantages over a singly linked list in certain use cases, mainly because of its circular nature, which provides continuous traversal and efficient operations. Here are some key advantages:

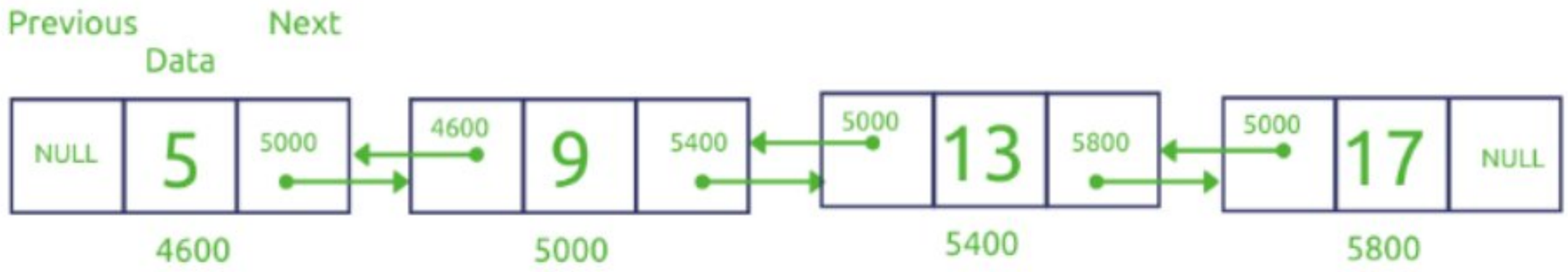
- Efficient Continuous Traversal:
  - In a circular linked list, after reaching the last node, traversal doesn't have to stop at a nullptr; instead, it loops back to the first node. This makes it useful in scenarios where continuous cycling through the list is required, such as: Round-robin scheduling (e.g., time-sharing in operating systems), Buffer management (e.g., circular buffers)
  - In a singly linked list, after reaching the end, traversal stops, requiring an explicit loop restart from the head.
- Easier Insertions at Both Ends:
  - In a circular linked list, maintaining a pointer to the last node allows easy access to both the head (first node) and the tail (last node) of the list. This simplifies insertion at both the beginning and the end of the list without needing to traverse the entire list to reach the last node.
  - In a singly linked list, you would need to traverse the list to insert at the end, making it less efficient, especially for long lists.
- Efficient Queue Implementation:
  - A circular linked list is ideal for implementing queues (especially circular queues) because it naturally supports enqueue and dequeue operations at both ends. By having the last node point to the first, you avoid the need to reset pointers manually or handle edge cases when the list reaches the end.
  - In a singly linked list, implementing a queue may require additional bookkeeping for managing the head and tail.

Example Use Cases: In summary, circular linked lists offer advantages in scenarios requiring continuous or repetitive processing, efficient insertions at both ends, and fixed memory usage without losing data integrity.

- Circular queues in memory-constrained systems (e.g., real-time operating systems).
- Round-robin task scheduling (e.g., operating system schedulers, multiplayer games).
- Circular buffers for data streams or buffering audio/video.

Doubly Linked List

A **doubly linked list** is a type of linked list in which each node contains a data element and two pointers: one pointing to the next node and another pointing to the previous node. This allows traversal in both forward and backward directions, offering more flexibility than a singly linked list.



Properties of a Doubly Linked List:

- **Two Pointers**: Each node has two pointers, one to the next node and one to the previous node, allowing bi-directional traversal.
- **Bi-directional Traversal**: You can traverse the list both forwards and backwards.
- **Efficient Insertions and Deletions**: You can insert or delete nodes more efficiently at any position, especially when you have a reference to the node, because you have access to the previous node without needing to traverse the list.
- **More Memory Overhead**: Since each node stores an extra pointer (prev), the memory overhead is higher than a singly linked list.

Advantages of a Doubly Linked List:

- **Bi-directional Traversal**: You can traverse the list in both directions (from head to tail or tail to head), making it more versatile for certain algorithms and data structures (e.g., implementing a deque).
- **Efficient Deletion and Insertion**: When deleting or inserting a node at an arbitrary position, a doubly linked list allows quick access to both neighboring nodes, avoiding the need to traverse the list to find the previous node.
- **Backtracking**: You can easily backtrack to the previous node, which is useful in certain applications like undo features or iterators that require two-way movement.

Disadvantages:

- **Memory Overhead**: Each node requires extra memory for the prev pointer.
- **Slightly More Complex**: Maintaining the prev pointer in insertion and deletion operations adds complexity compared to a singly linked list.

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

// Node structure for the doubly linked list
class Node {
   public:
            int data;
   Node* next;
   Node* prev;
   Node(int d){
            data=d;
            next=NULL;
            prev=NULL;
            }
};

class DoublyLinkedList {
private:
   Node* head;  // Pointer to the head of the list

public:
   // Constructor
   DoublyLinkedList() : head(NULL) {}

   void insertAtFront(int value) {
      Node* newNode = new Node(value);
      newNode->next = head;
      newNode->prev = NULL;
```

```
    if (head != NULL) {
        head->prev = newNode;  // Update previous pointer of the old head
    }
    head = newNode;
}

void insertAtEnd(int value) {
    Node* newNode = new Node(value);
    newNode->next = NULL;

    if (head == NULL) {            // If the list is empty, make the new node the head
        newNode->prev = NULL;
        head = newNode;
        return;
    }

    // Otherwise, find the last node
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

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

void deleteNode(int value) {      // Delete a node from the list by value
    Node* temp = head;

    // Traverse the list to find the node with the given value
    while (temp != NULL && temp->data != value) {
        temp = temp->next;
    }

    if (temp == NULL) {        // Node not found
        cout << "Node with value " << value << " not found." << endl;
        return;
    }

    if (temp->prev != NULL) {              // Update the previous node's next pointer
        temp->prev->next = temp->next;
    } else {
        // Node is the head
        head = temp->next;
    }

    if (temp->next != NULL) {              // Update the next node's previous pointer
        temp->next->prev = temp->prev;
    }

    // Free the memory of the node
    delete temp;
}

void displayForward() const {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

void displayBackward() const {
    Node* temp = head;

    // Traverse to the end of the list
```

```
    if (temp == NULL) return;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    // Now traverse backward from the last node
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << endl;
}

    // Destructor to free the memory
    ~DoublyLinkedList() {
        Node* temp = head;
        while (temp != NULL) {
            Node* next = temp->next;
            delete temp;
            temp = next;
        }
    }
};

int main() {
    DoublyLinkedList dll;

    dll.insertAtFront(10);
    dll.insertAtFront(20);
    dll.insertAtEnd(30);
    dll.insertAtEnd(40);

    cout << "Doubly Linked List (Forward): ";
    dll.displayForward();

    cout << "Doubly Linked List (Backward): ";
    dll.displayBackward();

    dll.deleteNode(20);
    cout << "After deleting 20, list (Forward): ";
    dll.displayForward();

    return 0;
}
```

Exercise

1. Implement class of a Circular Queue using a circular Linked List.
2. Implement class of a double ended queue using doubly Linked List.
3. Create two doubly link lists, say L and M . List L should be containing all even elements from 2 to 10 and list M should contain all odd elements from 1 to 9. Create a new list N by concatenating list L and M.
4. Sort the contents of list N created in Q4 in descending order.
5. You have a browser of one tab where you start on the homepage and you can visit another url, get back in the history number of steps or move forward in the history number of steps.

- Implement the BrowserHistory class: BrowserHistory(string homepage) Initializes the object with the homepage of the browser.
- void visit(string url) Visits url from the current page. It clears up all the forward history.
- string back(int steps) Move steps back in history. If you can only return x steps in the history and steps > x, you will return only x steps. Return the current url after moving back in history at most steps.
- string forward(int steps) Move steps forward in history. If you can only forward x steps in the history and steps > x, you will forward only x steps. Return the current url after forwarding in history at most steps.

Example:

```
BrowserHistory browserHistory = new BrowserHistory("leetcode.com");
browserHistory.visit("google.com");       // You are in "leetcode.com". Visit "google.com"
browserHistory.visit("facebook.com");     // You are in "google.com". Visit "facebook.com"
browserHistory.visit("youtube.com");      // You are in "facebook.com". Visit "youtube.com"
browserHistory.back(1);                   // You are in "youtube.com", move back to "facebook.com" return "facebook.com"
browserHistory.back(1);                   // You are in "facebook.com", move back to "google.com" return "google.com"
browserHistory.forward(1);                // You are in "google.com", move forward to "facebook.com" return "facebook.com"
browserHistory.visit("linkedin.com");     // You are in "facebook.com". Visit "linkedin.com"
browserHistory.forward(2);                // You are in "linkedin.com", you cannot move forward any steps.
browserHistory.back(2);                   // You are in "linkedin.com", move back two steps to "facebook.com" then to "google.com". return "google.com"
browserHistory.back(7);                   // You are in "google.com", you can move back only one step to "leetcode.com". return "leetcode.com"
```

Lab 07 Evaluation		
Student Name:	**Student ID:**	**Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		