**Lab Sheet 4**

**Title:** Linked List Implementations and Advanced Applications

**Introduction**

Linked lists are fundamental data structures that allow for efficient insertion and deletion of elements. They come in various forms, including singly linked lists, doubly linked lists, and circular linked lists. This assignment focuses on implementing these types of linked lists and exploring their applications in advanced data structure scenarios.

**Objective**

The objective of this assignment is to implement various types of linked lists and explore their applications in advanced data structure scenarios.

**Problem Description**

1. Implementing Linked Lists:
   * Implement singly linked lists with methods for insertion, deletion, and traversal.
   * Implement doubly linked lists with methods for insertion, deletion, and traversal.
   * Implement circular linked lists with methods for insertion, deletion, and traversal.
2. Applications:
   * Postfix Calculator: Implement a stack-based solution for evaluating postfix expressions.
   * Queue-Based System Simulation: Simulate a queue-based ticketing system.
   * Priority Queue Using Heaps: Implement a priority queue using a heap data structure.

**Instructions**

1. Linked List Implementation:
   * Create classes SinglyLinkedList, DoublyLinkedList, and CircularLinkedList.
   * Each class should have methods for insertion, deletion, and traversal.
2. Postfix Calculator:
   * Write a class PostfixCalculator that evaluates postfix expressions using a stack.
3. Queue-Based Ticketing System:
   * Write a class TicketQueue that simulates a ticketing system using a queue. Include methods to add tickets to the queue and to process them.
4. Priority Queue Implementation:
   * Write a class PriorityQueue that uses a heap to manage elements based on their priority.

**Code:**

#include <iostream>

#include <stack>

#include <queue>

#include <sstream>

#include <vector>

using namespace std;

// Singly Linked List

class SinglyLinkedList {

private:

struct Node {

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

Node\* head;

public:

SinglyLinkedList() : head(nullptr) {}

void insert(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next)

temp = temp->next;

temp->next = newNode;

}

}

void deleteNode(int value) {

if (!head) return;

if (head->data == value) {

Node\* temp = head;

head = head->next;

delete temp;

return;

}

Node\* temp = head;

while (temp->next && temp->next->data != value)

temp = temp->next;

if (temp->next) {

Node\* toDelete = temp->next;

temp->next = toDelete->next;

delete toDelete;

}

}

void traverse() {

Node\* temp = head;

while (temp) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

};

// Doubly Linked List

class DoublyLinkedList {

private:

struct Node {

int data;

Node\* next;

Node\* prev;

Node(int val) : data(val), next(nullptr), prev(nullptr) {}

};

Node\* head;

public:

DoublyLinkedList() : head(nullptr) {}

void insert(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next)

temp = temp->next;

temp->next = newNode;

newNode->prev = temp;

}

}

void deleteNode(int value) {

if (!head) return;

Node\* temp = head;

while (temp && temp->data != value)

temp = temp->next;

if (temp) {

if (temp->prev)

temp->prev->next = temp->next;

if (temp->next)

temp->next->prev = temp->prev;

if (temp == head)

head = temp->next;

delete temp;

}

}

void traverse() {

Node\* temp = head;

while (temp) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

};

// Circular Linked List

class CircularLinkedList {

private:

struct Node {

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

Node\* head;

public:

CircularLinkedList() : head(nullptr) {}

void insert(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

newNode->next = head;

} else {

Node\* temp = head;

while (temp->next != head)

temp = temp->next;

temp->next = newNode;

newNode->next = head;

}

}

void deleteNode(int value) {

if (!head) return;

Node\* temp = head;

Node\* prev = nullptr;

do {

if (temp->data == value) break;

prev = temp;

temp = temp->next;

} while (temp != head);

if (temp->data == value) {

if (prev) {

prev->next = temp->next;

if (temp == head)

head = head->next;

} else {

Node\* tail = head;

while (tail->next != head)

tail = tail->next;

head = head->next;

tail->next = head;

}

delete temp;

}

}

void traverse() {

if (!head) return;

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

};

// Postfix Calculator

class PostfixCalculator {

public:

int evaluate(const string& expression) {

stack<int> stk;

stringstream ss(expression);

string token;

while (ss >> token) {

if (isdigit(token[0])) {

stk.push(stoi(token));

} else {

int b = stk.top(); stk.pop();

int a = stk.top(); stk.pop();

switch (token[0]) {

case '+': stk.push(a + b); break;

case '-': stk.push(a - b); break;

case '\*': stk.push(a \* b); break;

case '/': stk.push(a / b); break;

}

}

}

return stk.top();

}

};

// Queue-Based Ticketing System

class TicketQueue {

private:

queue<int> ticketQueue;

int ticketNumber = 1;

public:

void addTicket() {

ticketQueue.push(ticketNumber++);

cout << "Ticket " << ticketNumber - 1 << " added to queue.\n";

}

void processTicket() {

if (!ticketQueue.empty()) {

cout << "Processing ticket " << ticketQueue.front() << endl;

ticketQueue.pop();

} else {

cout << "No tickets to process.\n";

}

}

void displayQueue() {

queue<int> tempQueue = ticketQueue;

while (!tempQueue.empty()) {

cout << tempQueue.front() << " ";

tempQueue.pop();

}

cout << endl;

}

};

// Priority Queue Using Heap

class PriorityQueue {

private:

priority\_queue<int> pq;

public:

void insert(int value) {

pq.push(value);

}

void deleteMax() {

if (!pq.empty()) pq.pop();

}

int getMax() {

return pq.top();

}

void display() {

priority\_queue<int> temp = pq;

while (!temp.empty()) {

cout << temp.top() << " ";

temp.pop();

}

cout << endl;

}

};

// Main Function

int main() {

// Test Singly Linked List

SinglyLinkedList sll;

sll.insert(10);

sll.insert(20);

sll.insert(30);

sll.traverse();

sll.deleteNode(20);

sll.traverse();

// Test Postfix Calculator

PostfixCalculator calc;

cout << "Postfix Result: " << calc.evaluate("3 4 + 2 \* 7 /") << endl;

// Test Ticket Queue

TicketQueue tq;

tq.addTicket();

tq.addTicket();

tq.processTicket();

tq.displayQueue();

// Test Priority Queue

PriorityQueue pq;

pq.insert(15);

pq.insert(5);

pq.insert(30);

pq.display();

return 0;

}

**Report on Linked List Implementations and Advanced Applications**

**1. Introduction**

Linked lists are fundamental data structures that provide efficient insertion and deletion operations. This project explores various types of linked lists (singly linked list, doubly linked list, and circular linked list) and delves into practical applications of data structures, such as stack-based postfix calculators, queue-based ticketing systems, and priority queues using heaps.

**2. Objectives**

The objectives of this project are:

* To implement different types of linked lists with methods for insertion, deletion, and traversal.
* To explore advanced data structure applications using stacks, queues, and priority queues.
* To understand the practical utility of these data structures in various real-life scenarios.

**3. Problem Statement**

The assignment consists of two main parts:

1. **Linked List Implementation**:
   * Implement singly, doubly, and circular linked lists with operations such as insertion, deletion, and traversal.
2. **Applications**:
   * Implement a stack-based solution to evaluate postfix expressions.
   * Implement a queue-based ticketing system.
   * Implement a priority queue using a heap data structure.

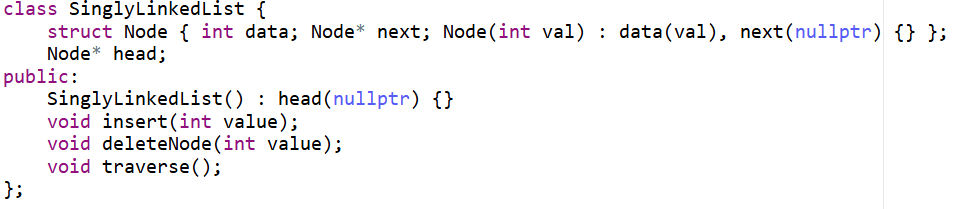
**4. Implementation**

**4.1. Singly Linked List**

The singly linked list consists of nodes where each node stores data and a pointer to the next node in the sequence. The operations implemented include:

* **Insertion**: Adds a new node to the end of the list.
* **Deletion**: Removes a node by value.
* **Traversal**: Prints all the elements of the list.

**Code:**

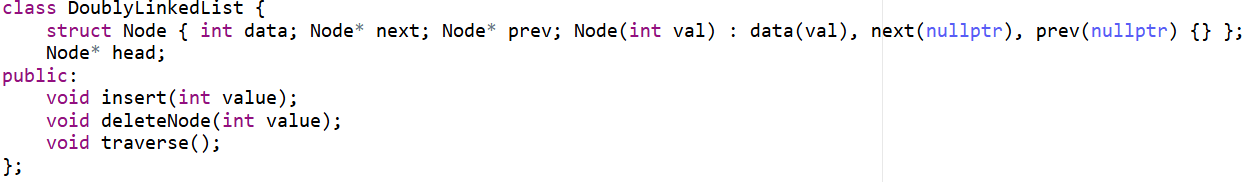
****

**4.2. Doubly Linked List**

A doubly linked list contains nodes with pointers to both the previous and next nodes, allowing bidirectional traversal.

* **Insertion**: Adds a new node to the end.
* **Deletion**: Removes a node by value.
* **Traversal**: Prints elements from the head.

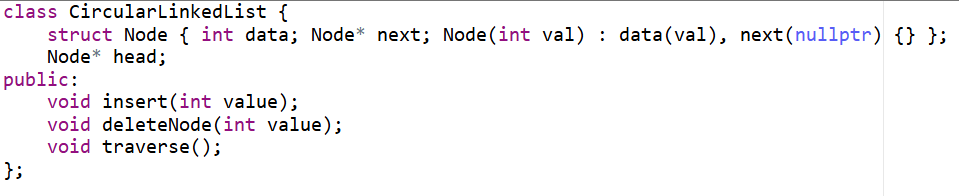
**Code:**

**4.3. Circular Linked List**

In a circular linked list, the last node points back to the first node, forming a circular structure.

* **Insertion**: Adds nodes to the list while maintaining the circular link.
* **Deletion**: Deletes nodes by value.
* **Traversal**: Continuously traverses from the head until it loops back.

**Code:**

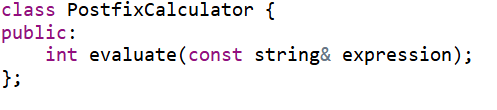


**5. Applications**

**5.1. Postfix Calculator**

A stack-based implementation is used to evaluate postfix expressions, which are mathematical expressions where the operator follows the operands (e.g., 3 4 + instead of 3 + 4). The algorithm uses a stack to handle operands and applies operators sequentially.

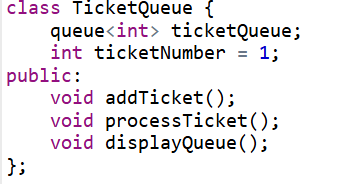
**Code:**



**5.2. Queue-Based Ticketing System**

A ticketing system is simulated using a queue. The system manages ticket issuance and processing. New tickets are added to the end, and the oldest ticket is processed first, following the First-In-First-Out (FIFO) principle.

**Code:**

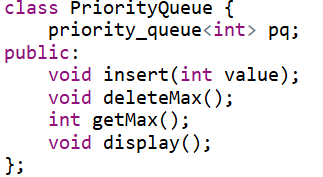


**5.3. Priority Queue Using Heaps**

A priority queue is implemented using a max-heap to manage elements based on priority. The heap ensures that the highest-priority element is always at the top.

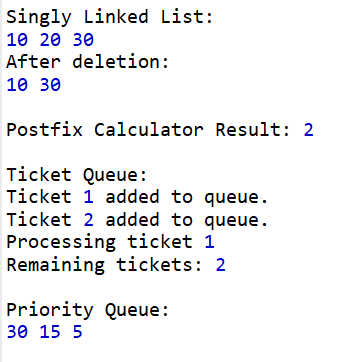
* **Insertion**: Adds a new element to the heap.
* **DeleteMax**: Removes the element with the highest priority.
* **Display**: Displays all elements in descending priority.

**Code:**



**6. Example Output**

**Sample Output from the Program:**



**7. Conclusion**

In this project, we explored various linked list implementations and their real-world applications using advanced data structures like stacks, queues, and heaps. By implementing these structures, we observed their efficiency in handling operations like insertion, deletion, and priority-based processing.

**Key Takeaways:**

* Linked lists are efficient for dynamic memory allocation.
* Stacks are useful for evaluating expressions and managing function calls.
* Queues are ideal for processing tasks in a sequential manner.
* Heaps provide an efficient way to implement priority queues.