**Lab Sheet 3**

**Title:** Implementing and Utilizing Stacks and Queues

**Introduction**

Stacks and queues are fundamental data structures used in computer science for various algorithmic problem-solving tasks. Stacks follow the Last In First Out (LIFO) principle, while queues follow the First In First Out (FIFO) principle. This assignment focuses on implementing these abstract data types using both array-based and linked list-based approaches and exploring their applications in different scenarios.

**Objective**

The objective of this assignment is to understand and implement stack and queue abstract data types using both array-based and linked list-based approaches. Students will explore various applications of stacks and queues in algorithmic problem-solving.

**Problem Description**

1. **Implementing Stacks and Queues:**
   * Implement a stack using both array-based and linked list-based approaches.
   * Implement a queue using both array-based and linked list-based approaches.
2. **Applications:**
   * System Stack: Simulate the call stack of a program using the stack implementation. Write a function to display the state of the stack at any point.
   * Reversing Data: Use a stack to reverse the contents of a given string.
   * Queue Management in Systems: Simulate a basic print queue system where documents are enqueued and dequeued for printing.

**Instructions**

1. **Stack Implementation:**
   * Create a class ArrayStack with methods for push, pop, peek, and isEmpty.
   * Create a class LinkedListStack with methods for push, pop, peek, and isEmpty.
2. **Queue Implementation:**
   * Create a class ArrayQueue with methods for enqueue, dequeue, peek, and isEmpty.
   * Create a class LinkedListQueue with methods for enqueue, dequeue, peek, and isEmpty.
3. **System Stack Simulation:**
   * Write a function that simulates the system stack for a simple program with nested function calls.
4. **Reversing Data:**
   * Write a function reverseString that takes a string as input and uses a stack to reverse its characters.
5. Queue Management Simulation:
   * Write a class PrintQueue to simulate a print queue. Include methods to add documents to the queue and to process them.

**Code:**

#include <iostream>

#include <string>

using namespace std;

// Array-based Stack Implementation

class ArrayStack {

private:

int \*arr;

int top;

int capacity;

public:

ArrayStack(int size) {

arr = new int[size];

capacity = size;

top = -1;

}

void push(int x) {

if (top == capacity - 1) {

cout << "Stack Overflow\n";

return;

}

arr[++top] = x;

}

int pop() {

if (isEmpty()) {

cout << "Stack Underflow\n";

return -1;

}

return arr[top--];

}

int peek() {

if (isEmpty()) {

cout << "Stack is empty\n";

return -1;

}

return arr[top];

}

bool isEmpty() {

return top == -1;

}

~ArrayStack() {

delete[] arr;

}

};

// Linked List-based Stack Implementation

struct Node {

int data;

Node\* next;

};

class LinkedListStack {

private:

Node\* head;

public:

LinkedListStack() {

head = nullptr;

}

void push(int x) {

Node\* newNode = new Node();

newNode->data = x;

newNode->next = head;

head = newNode;

}

int pop() {

if (isEmpty()) {

cout << "Stack Underflow\n";

return -1;

}

Node\* temp = head;

int poppedValue = head->data;

head = head->next;

delete temp;

return poppedValue;

}

int peek() {

if (isEmpty()) {

cout << "Stack is empty\n";

return -1;

}

return head->data;

}

bool isEmpty() {

return head == nullptr;

}

};

// Array-based Queue Implementation

class ArrayQueue {

private:

int \*arr;

int front, rear, capacity;

public:

ArrayQueue(int size) {

arr = new int[size];

capacity = size;

front = 0;

rear = -1;

}

void enqueue(int x) {

if (rear == capacity - 1) {

cout << "Queue Overflow\n";

return;

}

arr[++rear] = x;

}

int dequeue() {

if (isEmpty()) {

cout << "Queue Underflow\n";

return -1;

}

return arr[front++];

}

int peek() {

if (isEmpty()) {

cout << "Queue is empty\n";

return -1;

}

return arr[front];

}

bool isEmpty() {

return front > rear;

}

~ArrayQueue() {

delete[] arr;

}

};

// Linked List-based Queue Implementation

class LinkedListQueue {

private:

struct QNode {

int data;

QNode\* next;

};

QNode\* front;

QNode\* rear;

public:

LinkedListQueue() {

front = rear = nullptr;

}

void enqueue(int x) {

QNode\* newNode = new QNode();

newNode->data = x;

newNode->next = nullptr;

if (rear == nullptr) {

front = rear = newNode;

return;

}

rear->next = newNode;

rear = newNode;

}

int dequeue() {

if (isEmpty()) {

cout << "Queue Underflow\n";

return -1;

}

QNode\* temp = front;

int data = front->data;

front = front->next;

if (front == nullptr)

rear = nullptr;

delete temp;

return data;

}

int peek() {

if (isEmpty()) {

cout << "Queue is empty\n";

return -1;

}

return front->data;

}

bool isEmpty() {

return front == nullptr;

}

};

// Reversing a String using Stack

string reverseString(const string &str) {

ArrayStack stack(str.length());

for (char ch : str)

stack.push(ch);

string reversed = "";

while (!stack.isEmpty())

reversed += stack.pop();

return reversed;

}

// Simulating a System Stack for Function Calls

void functionA();

void functionB();

void functionC();

void systemStackSimulation() {

LinkedListStack stack;

stack.push(1); // Function A

cout << "Function A called\n";

stack.push(2); // Function B

cout << "Function B called\n";

stack.push(3); // Function C

cout << "Function C called\n";

while (!stack.isEmpty()) {

int func = stack.pop();

if (func == 1) cout << "Function A returned\n";

else if (func == 2) cout << "Function B returned\n";

else if (func == 3) cout << "Function C returned\n";

}

}

// Print Queue Simulation

class PrintQueue {

private:

LinkedListQueue queue;

public:

void addDocument(int docID) {

cout << "Document " << docID << " added to the print queue\n";

queue.enqueue(docID);

}

void processDocuments() {

while (!queue.isEmpty()) {

cout << "Printing document " << queue.dequeue() << "\n";

}

}

};

int main() {

// Testing Stack and Queue Implementations

ArrayStack stack(5);

stack.push(10);

stack.push(20);

cout << "Top of stack: " << stack.peek() << endl;

cout << "Popped from stack: " << stack.pop() << endl;

ArrayQueue queue(5);

queue.enqueue(30);

queue.enqueue(40);

cout << "Front of queue: " << queue.peek() << endl;

cout << "Dequeued from queue: " << queue.dequeue() << endl;

// Reversing a String

string str = "OpenAI";

cout << "Original string: " << str << endl;

cout << "Reversed string: " << reverseString(str) << endl;

// System Stack Simulation

systemStackSimulation();

// Print Queue Simulation

PrintQueue printQueue;

printQueue.addDocument(101);

printQueue.addDocument(102);

printQueue.processDocuments();

return 0;

}

**Report on Implementing and Utilizing Stacks and Queues**

**1. Introduction**

Stacks and queues are fundamental data structures in computer science, widely used for various algorithmic and real-world problem-solving tasks. These data structures are essential in managing data that needs to be processed in specific orders:

* **Stacks**: Follow the Last In, First Out (LIFO) principle.
* **Queues**: Follow the First In, First Out (FIFO) principle.

This report details the implementation of stacks and queues using two approaches:

1. **Array-based** implementation
2. **Linked list-based** implementation

Additionally, we explore practical applications, such as reversing strings, simulating system function call stacks, and managing print queues.

**2. Objectives**

The objectives of this project are:

* To implement stack and queue data structures using both array-based and linked list-based approaches.
* To demonstrate the practical applications of stacks and queues:
  + Simulate a system stack for function calls.
  + Reverse a string using a stack.
  + Implement a print queue system.

**3. Problem Description**

The tasks covered in this project include:

1. **Implementing Stacks and Queues**:
   * Implementing a stack using arrays and linked lists.
   * Implementing a queue using arrays and linked lists.
2. **Applications**:
   * Simulating the system stack of a program.
   * Reversing a string using a stack.
   * Managing a print queue system.

**4. Implementation**

**4.1 Stack Implementations**

**4.1.1 Array-based Stack**

The stack is implemented using an array. It supports the following operations:

* **push()**: Adds an element to the top of the stack.
* **pop()**: Removes and returns the top element of the stack.
* **peek()**: Returns the top element without removing it.
* **isEmpty()**: Checks if the stack is empty.

**4.1.2 Linked List-based Stack**

The stack is implemented using a singly linked list. It supports similar operations as the array-based stack:

* **push()**: Adds an element at the head of the linked list.
* **pop()**: Removes and returns the head element.
* **peek()**: Returns the head element.
* **isEmpty()**: Checks if the stack is empty.

**4.2 Queue Implementations**

**4.2.1 Array-based Queue**

The queue is implemented using an array. It supports the following operations:

* **enqueue()**: Adds an element to the rear of the queue.
* **dequeue()**: Removes and returns the front element.
* **peek()**: Returns the front element without removing it.
* **isEmpty()**: Checks if the queue is empty.

**4.2.2 Linked List-based Queue**

The queue is implemented using a linked list. It supports similar operations as the array-based queue:

* **enqueue()**: Adds an element at the rear.
* **dequeue()**: Removes and returns the front element.
* **peek()**: Returns the front element.
* **isEmpty()**: Checks if the queue is empty.

**5. Applications of Stacks and Queues**

**5.1 Reversing a String using a Stack**

A stack is used to reverse the characters of a given string. The characters are pushed onto the stack one by one and then popped off to get them in reverse order.

**5.2 Simulating a System Stack**

We simulate the system call stack using a stack data structure. This is useful in understanding how function calls are managed during program execution. The stack keeps track of function calls, and each function returns in the reverse order of its call.

**5.3 Print Queue Management**

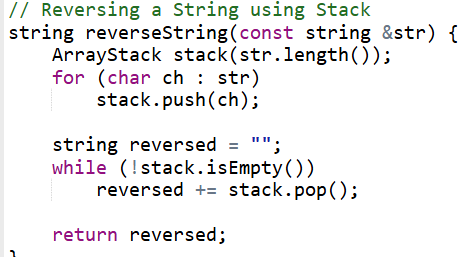
A queue is used to manage a basic print queue system. Documents are added to the queue and processed in the order they are received (FIFO). This is a practical example of using a queue to manage tasks in a real-world scenario.

**6. Code Explanation**

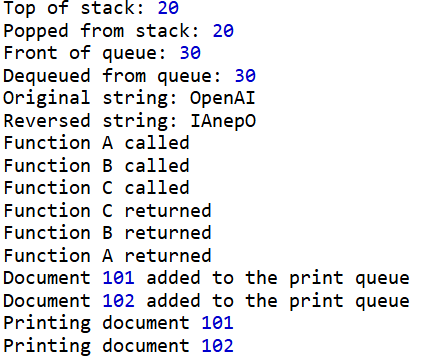
The complete C++ program includes:

* **Classes** for stacks and queues with their respective operations.
* **Functions** to demonstrate the use cases:
  + reverseString() reverses a given string using a stack.
  + systemStackSimulation() simulates a series of function calls.
  + PrintQueue manages a list of print jobs.

**Sample Code**



**Sample Output**



**7. Conclusion**

The project successfully demonstrates the implementation of stack and queue data structures using both array-based and linked list-based approaches. Additionally, it highlights the utility of these data structures in real-world applications such as function call management and queue management systems.

**Key Learnings:**

* Understanding of abstract data types: stacks and queues.
* Gained proficiency in implementing these data structures using arrays and linked lists.
* Demonstrated real-life applications of stacks and queues, reinforcing their importance in computer science.

**8. References**

* Data Structures and Algorithms textbooks
* C++ Standard Documentation