CL2001

Data Structures Lab

Lab 6

Stack and Queues

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2025

Lab Content

1. Stacks
2. Queues
3. Prefix and Postfix Conversions
4. Circular Queue
5. Applications of stacks and queue

Stack and Its Implementation

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle — the last item added is the first one removed. It supports two main operations: push (to add an element) and pop (to remove the top element). Below are some sample example codes of how to use stack with arrays and linked list.

1. Stack with Array – To-Do List Example

```
class Stack {
    int top;

public:
    int a[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}

int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
}
```

2. **Stack with Linked List – Web History Tracker**

```
struct Node
{
    int data;
    struct Node* link;
};

struct Node* top;

// Utility function to add an element
// data in the stack insert at the beginning
void push(int data)
{

    // Create new node temp and allocate memory
    struct Node* temp;
    temp = new Node();

    // Check if stack (heap) is full.
    // Then inserting an element would
    // lead to stack overflow
    if (!temp)
    {
        cout << "\nHeap Overflow";
        exit(1);
    }

    // Initialize data into temp data field
    temp->data = data;

    // Put top pointer reference into temp link
    temp->link = top;

    // Make temp as top of Stack
    top = temp;
}
```

Applications of Stack (Convert infix expression to postfix)

The infix expression is the standard mathematical notation (e.g., A + B * C). The prefix expression (also known as Polish notation) writes the operator before the operands (e.g., + A * B C).

To handle the conversion correctly, the following precedence and associativity rules apply:

- **^** (Exponent) : Highest precedence, Right to Left
- ***, /** : Medium precedence, Left to Right
- **+, -** : Lowest precedence, Left to Right

Rules to pop and push operators in a stack:
When an operand is encountered in the input, it is added to the output queue.

- When an operator is encountered in the input, the following rules are applied:
- If the stack is empty or the top of the stack contains a left parenthesis, the operator is pushed onto the stack.
- If the operator has higher precedence than the top of the stack, it is pushed onto the stack.
- If the operator has lower or equal precedence than the top of the stack, the top of the stack is popped and added to the output queue. This continues until the operator has higher precedence than the top of the stack, or the stack is empty or contains a left parenthesis.
- If the incoming operator is the right parenthesis, operators are popped from the stack and added to the output queue until a left parenthesis is encountered. The left parenthesis is popped from the stack and discarded.
- After all the tokens have been processed, any remaining operators in the stack are popped and added to the output queue.

Code Explanation: The algorithm reads each character of the input string and performs the following actions based on the type of the character:

- If the character is an operand, append it to the output string.
- If the character is a left parenthesis, push it onto the stack.
- If the character is a right parenthesis, pop operators from the stack and append them to the output string until a left parenthesis is encountered. Pop the left parenthesis from the stack and discard it.
- If the character is an operator, pop operators from the stack and append them to the output string until an operator of lower precedence is encountered, or the stack is empty, or a left parenthesis is encountered. Push the operator onto the stack.
- After all the characters have been processed, pop any remaining operators from the stack and append them to the output string.

Code Snippet

```
int precedence(char c) {
    if (c == '^') {
        return 3;
    } else if (c == '*' || c == '/') {
        return 2;
    } else if (c == '+' || c == '-') {
        return 1;
    } else {
        return -1;
    }
}
```

```
string infixToPostfix(string infix) {
    string postfix = "";
    Stack s(infix.length());

    for (int i = 0; i < infix.length(); i++) {
        char c = infix[i];
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
            postfix += c;
        } else if (c == '(') {
            s.push(c);
        } else if (c == ')') {
            while (!s.isEmpty() && s.top() != '(') {
                char op = s.pop();
                postfix += op;
            }
            if (s.top() == '(') {
                s.pop();
            }
        } else {
            while (!s.isEmpty() && precedence(c) <= precedence(s.top())) {
                char op = s.pop();
                postfix += op;
            }
            s.push(c);
        }
    }
    while (!s.isEmpty()) {
        char op = s.pop();
        postfix += op;
    }
    return postfix;
}
```

Infix to Prefix Conversion:

```
string infixToPrefix(string infix) {
    string prefix = "";
    Stack st(infix.length());

    for (int i = infix.length() - 1; i >= 0; i--) {
        char c = infix[i];

        if (isalnum(c)) {
            prefix = c + prefix;
        } else if (c == ')') {
            st.push(c);
        } else if (c == '(') {
            while (!st.isEmpty() && st.pop() != ')') {
                prefix = st.pop() + prefix;
            }
        } else if (isOperator(c)) {
            while (!st.isEmpty() && getPrecedence(st.peek()) >= getPrecedence(c)) {
                prefix = st.pop() + prefix;
            }
            st.push(c);
        }
    }

    while (!st.isEmpty()) {
        prefix = st.pop() + prefix;
    }

    return prefix;
}
```

Explanations:

- The infixToPrefix function takes an infix expression as input and returns the corresponding prefix expression. The function first reverses the input string using the reverse function, so that it can be processed from right to left.
- The function initializes a stack operator to hold the operators encountered during processing, and a string prefix to hold the prefix expression.
- The function iterates through the reversed infix expression, one character at a time.
- If the current character is an operand (i.e., not an operator or parenthesis), it is added to the prefix string.
- If the current character is an operator, it is pushed onto the operator's stack if it has higher precedence than the top operator in the stack, or if the stack is empty. If the current operator has lower precedence than the top operator in the stack, the top operator is popped from the stack and added to the prefix string, and the process is repeated until the top operator has lower precedence or the stack is empty. Finally, the current operator is pushed onto the stack.
- If the current character is a left parenthesis ((), it is pushed onto the operator's stack.
- If the current character is a right parenthesis (), the top operator is popped from the stack and added to the prefix string until a left parenthesis is encountered. The left parenthesis is then discarded, and the process continues with the next character.
- After all characters have been processed, any remaining operators on the stack are popped and added to the prefix string.
- The prefix string is then reversed back to its original order using the reverse function and returned as the result of the function.

Circular Queue

A circular queue is a type of data structure that allows you to implement a queue where the front and rear elements are linked together to form a circular chain. In a circular queue, the last element is connected to the first element, forming a circle. This means that when the queue becomes full, we can start adding new elements at the beginning of the queue, provided there is space available there.

The circular queue has four primary operations:

- enqueue, which adds an element to the rear of the queue
- dequeue, which removes an element from the front of the queue
- isFull, which checks whether the queue is full
- isEmpty, which checks whether the queue is empty

Example:

```
//Initialization
int arr[4];
int rear = -1;
int front = -1;
```

```
enqueue(value)
{
    if(isFull())
        return;
    else if(isEmpty)
    {
        rear = front = 0;
    }
    else
    {
        rear = (rear+1)%N;
    }
    arr[rear] = value;
}

isEmpty()
{
    if(front == -1 && rear == -1)
        return true;
    else
        return false;
}

isFull()
{
    if((rear+1)%N == front)
        return true;
    else
        return false;
}
```

```
enqueue(value)
{
    if(isFull())
        return;
    else if(isEmpty)
    {
        rear = front = 0;
    }
    else
    {
        rear = (rear+1)%N;
    }
    arr[rear] = value;
}

dequeue()
{
    int x = 0;
    if(isEmpty())
        return;
    else if(front == rear)
    {
        x = arr[front];
        front = rear = -1;
    }
    else
    {
        x = arr[front];
        front = (front+1)%4;
    }
    return x;
}
```

Linear Queue

A linear queue is a type of queue where the elements are stored in a contiguous memory location, similar to an array. In a linear queue, the insertion of elements is done at one end (rear) and the deletion of elements is done from the other end (front).

```
#include <iostream>
using namespace std;
int front = -1, rear = -1;
int arr[MAX_SIZE];

bool isFull() {
    return rear == MAX_SIZE - 1;
}

bool isEmpty() {
    return front == -1 || front > rear;
}

void enqueue(int value) {
    if (isFull()) {
        cout << "Queue is full, cannot enqueue " << value << endl;
        return;
    }
    if (front == -1) {
        front = 0;
    }
    arr[++rear] = value;
}
```

```
int dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty, cannot dequeue" << endl;
        return -1;
    }
    return arr[front++];
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    enqueue(50);
    cout << dequeue() << endl;
    cout << dequeue() << endl;
    cout << dequeue() << endl;
    enqueue(60);
    enqueue(70);
    cout << dequeue() << endl;
    cout << dequeue() << endl;
    cout << dequeue() << endl;
    cout << dequeue() << endl;
    cout << dequeue() << endl;
    cout << dequeue() << endl;
    return 0;
}
```

LAB TASKS

1. Write a program to check whether all the opening and closing tags in an HTML/XML snippet are properly nested — for example, <html><body></body></html> is valid, but <html><body></html></body> is not. Use a stack to validate matching start and end tags in the given HTML code.
2. Develop a mini calculator that evaluates postfix expressions entered by the user (like a part of a compiler's expression parser). Input a postfix expression such as "45*62/+9-" and output the evaluated result using a stack.
3. Simulate a check-in counter at an airport where passengers queue up for boarding passes. Implement enqueue (add passenger) and dequeue (serve passenger) operations using a queue, displaying the current waiting line after each operation.
4. In a food delivery app, new orders are added continuously while completed ones are removed. To manage limited order slots efficiently, use a circular queue. Implement a circular queue where each order has an Order ID and number of items; perform enqueue, dequeue, and display operations.
5. In stock analysis, the "span" of stock's price today is the number of consecutive days before today with price less than or equal to today's price. Implement a function vector<int> calculateSpan(int price[], int n) that returns span for each day using a stack.
   Example Input: price[] = {100, 80, 60, 70, 60, 75, 85}
   Output: {1, 1, 1, 2, 1, 4, 6}