TRIBHUVAN UNIVERSITY

Paschimanchal Campus

Lamachaur 16, Pokhara

Department of Computer Engineering

Lab Report On:

Lab No: 3

Title:

Submitted By

Name: Anish Karki

Roll No: PAS080BCT007

Submitted To: HPB Sir

Date of submission: 03/07/2025

PRACTICAL-1
INFIX TO POSTFIX CONVERSION

THEORY

Infix to postfix conversion is a method of rewriting arithmetic expressions so that they can be easily evaluated by computers without needing to consider operator precedence or parentheses. In infix notation, operators are written between operands (e.g., A + B), which is how humans commonly write expressions. However, postfix notation, also known as Reverse Polish Notation (e.g., A B +), places operators after their operands and is more suitable for computer-based evaluation. The conversion process uses a stack to temporarily hold operators and ensures that they are placed in the correct order based on precedence and associativity. Operands are directly added to the output, while operators and parentheses are managed using stack operations. This conversion simplifies the evaluation process and is commonly used in compilers and expression parsers.

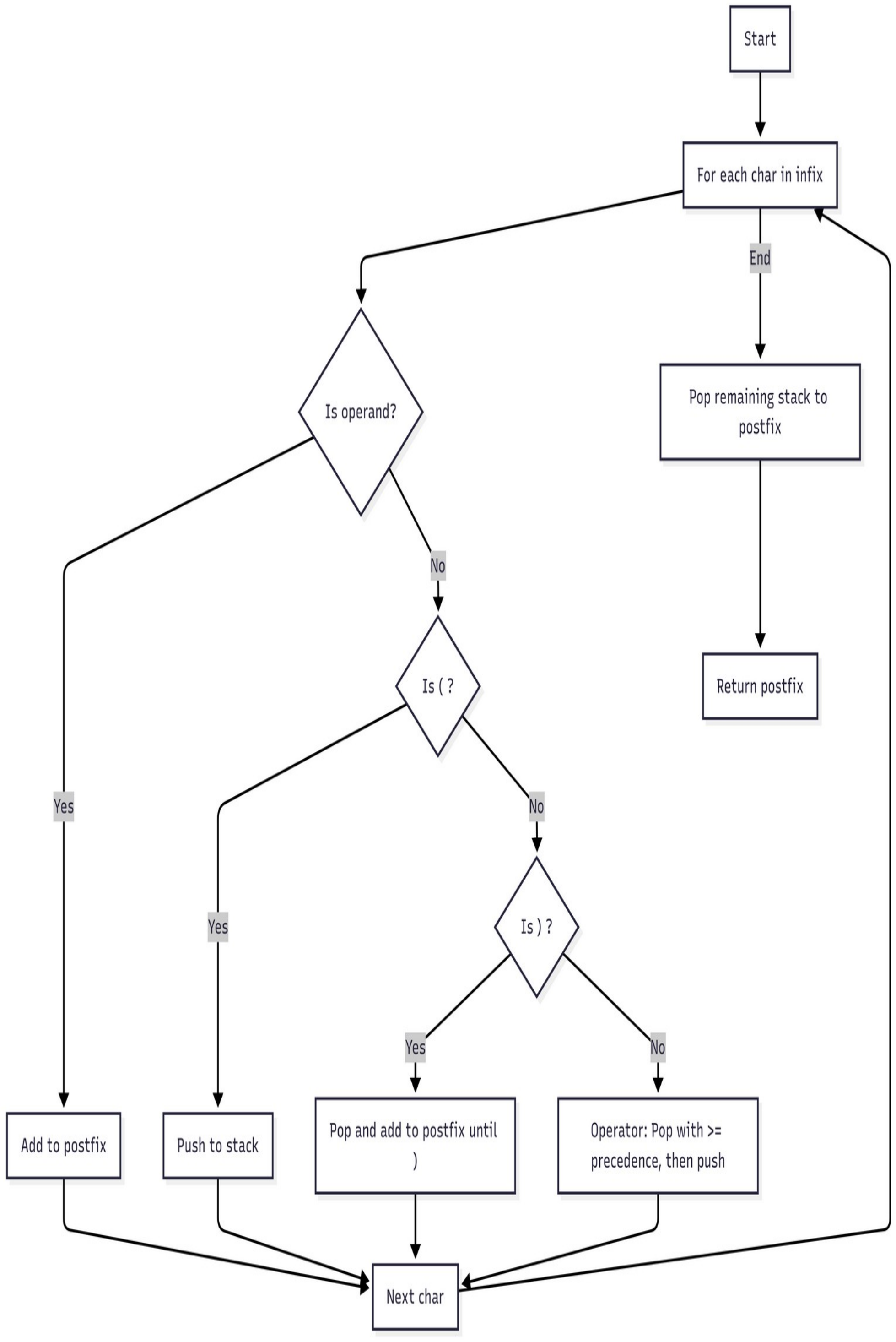
Time Complexity:

O(n)

ALGORITHM

```
1) Start
2) Initialize an empty stack s for operators.
3) Initialize an empty string postfix for the result.
4) Read the infix expression as a string infix.
5) For each character ch in infix, do:
        1.If ch is an operand, append ch to postfix.
        2.Else if ch is '(', push ch onto stack s.
        3.Else if ch is ')':
            1.While the top of stack s is not '(', pop from s and append to postfix.
            2.Pop '(' from stack s.
        4.Else if ch is an operator:
            1.While stack s is not empty and precedence of top of s is greater than or
            equal to precedence of ch:
                •Pop from s and append to postfix.
            2.Push ch onto stack s.
6) While stack s is not empty:
        5.Pop from s and append to postfix.
7) Output postfix as the postfix expression.
8) Stop
```

FLOWCHART


Start
For each char in infix
End
Pop remaining stack to postfix
Return postfix
Is operand?
Yes
No
Is (?
Yes
No
Is) ?
Yes
No
Add to postfix
Push to stack
Pop and add to postfix until)
Operator: Pop with >= precedence, then push
Next char

CODE:

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

int precedence(char op)
{
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    if (op == '^')
        return 3;
    return 0;
}

bool isOperator(char ch)
{
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}

bool isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9');
}

string infixToPostfix(const string &infix)
{
    stack<char> s;
    string postfix;

    for (int i = 0; i < infix.length(); i++)
    {
        char ch = infix[i];

        if (isOperand(ch))
        {
            postfix += ch;
        }
        else if (ch == '(')
        {
            s.push(ch);
        }
        else if (ch == ')')
```

```
            {
                while (!s.empty() && s.top() != '(')
                {
                    postfix += s.top();
                    s.pop();
                }
                if (!s.empty() && s.top() == '(')
                    s.pop();
            }
            else if (isOperator(ch))
            {
                while (!s.empty() && precedence(s.top()) >= precedence(ch))
                {
                    postfix += s.top();
                    s.pop();
                }
                s.push(ch);
            }
        }

        while (!s.empty())
        {
            postfix += s.top();
            s.pop();
        }

        return postfix;
    }

    int main()
    {
        string infix;
        cout << "Enter an infix expression: ";
        cin >> infix;

        string postfix = infixToPostfix(infix);
        cout << "Postfix expression: " << postfix << endl;

        return 0;
    }
```

PRACTICAL-2
ENQUEUE AND DEQUEUE OPERATION

THEORY

A **queue** is a linear data structure that follows the **FIFO** (First In, First Out) principle, meaning the first element added is the first one to be removed.

- **Enqueue Operation:**
  This operation adds an element to the **rear (end)** of the queue. When you enqueue, you insert the new element after the last element currently in the queue.
- **Dequeue Operation:**
  This operation removes an element from the **front (beginning)** of the queue. When you dequeue, you remove the element that has been in the queue the longest.

These two operations allow the queue to function as a waiting line where elements enter at the rear and exit from the front in the order they arrived.

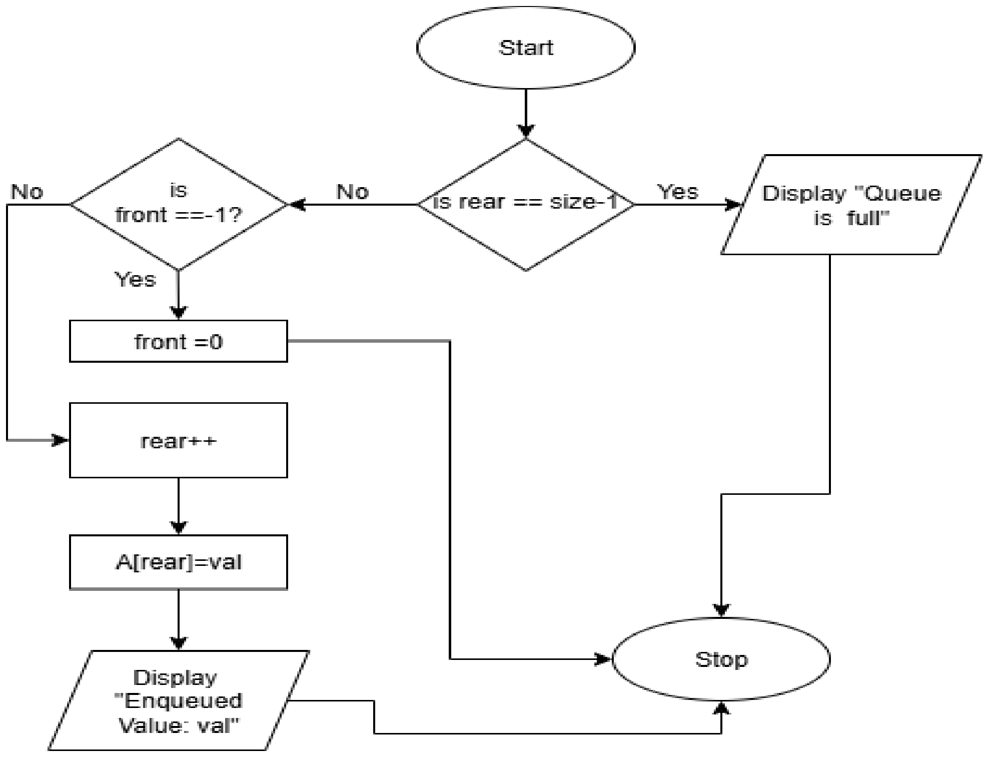
Time Complexity:

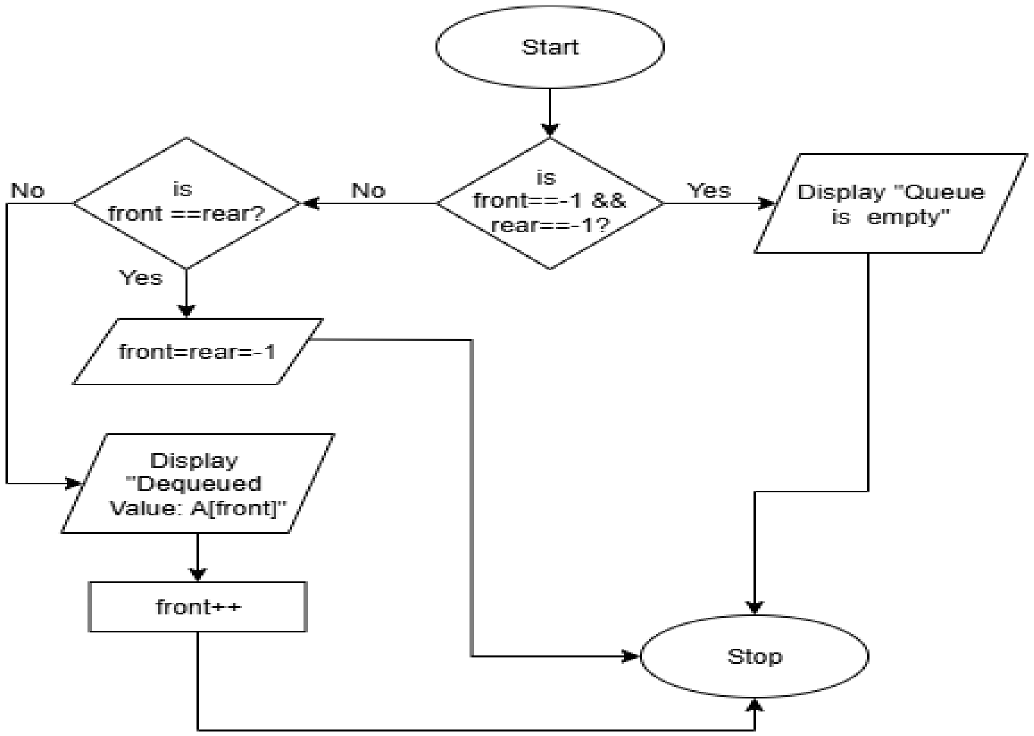
Enqueue: *O(1)*
Dequeue*:O(1)*

ALGORITHM

```
Start
    1.Initialize an array A of size SIZE.
    2.Set front = -1 and rear = -1.
    3.Enqueue Operation (enqueue(val)):
        1.If rear is equal to SIZE - 1, print "Queue is full" and return.
        2.If front is -1, set front = 0.
        3.Increment rear by 1.
        4.Set A[rear] = val.
        5.Print the enqueued value.
    4.Dequeue Operation (dequeue()):
        1.If front == -1 and rear == -1, print "Queue is empty" and return.
        2.If front == rear, set both front and rear to -1.
        3.Else,
            1.Print the dequeued value A[front].
            2.Increment front by 1.
    5.Main Function:
        1.Call dequeue().
        2.Call enqueue(2).
        3.Call enqueue(4).
        4.Call enqueue(6).
        5.Call dequeue().
```

FLOWCHART


Start
is rear == size-1
Yes
Display "Queue is full"
No
is front ==-1?
No
Yes
front =0
rear++
A[rear]=val
Display "Enqueued Value: val"
Stop


Start
is front==-1 && rear==-1?
Yes
Display "Queue is empty"
No
is front ==rear?
No
Yes
front=rear=-1
Display "Dequeued Value: A[front]"
front++
Stop

CODE:

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

#define SIZE 2

int A[SIZE];
int front = -1, rear = -1;

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

void enqueue(int val)
{
   if (rear == SIZE - 1)
   {
      cout << "Queue is full" << endl;
      return;
   }

   if (front == -1)
   {
      front = 0;
   }

   rear++;
   A[rear] = val;
   cout << "Enqueued Value : " << val << endl;
}

void dequeue()
{
   if (isEmpty())
   {
      cout << "Queue is empty" << endl;
      return;
   }
```

```
    if (front == rear)
    {
        front = rear = -1;
    }
    else
    {
        if (front == rear)
        {
            front = rear = -1;
        }
        else
        {
            cout << "Dequeued Value : " << A[front] << endl;
            front++;
        }
    }
}

int main()
{
    dequeue();
    enqueue(2);
    enqueue(4);
    enqueue(6);
```

CONCLUSION

In this lab, we implemented the infix to postfix conversion using stacks and performed enqueue and dequeue operations using a queue. The infix to postfix conversion demonstrated how stacks handle operator precedence and associativity to convert expressions for easier evaluation. The enqueue and dequeue operations helped us understand the working of a queue data structure, following the First-In-First-Out (FIFO) principle. Overall, this lab enhanced our understanding of both stack and queue operations and their real-world applications in expression evaluation and data handling.