DSA Lab
Name: Siddhi Parekh
Reg No.: 221071047
Batch: C
SY Comps

Experiment No. : 3

AIM:

Write a Program for Prefix Evaluation, Parenthesis Matching, Prefix to Postfix.

Theory:

1.**Prefix Evaluation:**

In prefix notation, operators appear before the operands.
For example:

- Prefix expression: + 3 4
- Equivalent infix expression: 3 + 4

To Evaluate a Prefix Expression :

- Start from the Rightmost character.
- If it's an operand (a number), push it onto the stack.
- If it's an operator, pop the top two operands from the stack, apply the operator, and push the result back onto the stack.
- First popped is Second Operand, and Second Popped is First Operand.
- Repeat until all characters are processed.
- The final result is the value left on the stack.

Prefix notation eliminates the need for parentheses and ensures unambiguous expression evaluation. It's used in programming languages, calculators, and expression evaluators .

2.**Parenthesis Matching:**

The idea is to put all the opening brackets in the stack. Whenever you hit a closing bracket, search if the top of the stack is the opening bracket of the same nature. If this holds then pop the stack and continue the iteration. In the end if the stack is empty, it means all brackets are balanced or well-formed. Otherwise, they are not balanced.

Time Complexity: O(N)
Space Complexity: O(N)

3**.Prefix to Postfix:**

A prefix expression is one where the operator appears in the expression before the operands.

A postfix expression is one where the operator appears in the expression after the operands. Read the Prefix expression in reverse order (from right to left).

1. If the symbol is an operand, then push it onto the Stack.
2. If the symbol is an operator, then pop two operands from the Stack. Create a string by concatenating the two operands and the operator after them. The string will be of the form operand1 + operand2 + operator. Push the resultant string back to Stack.
3. Repeat the above steps until the end of the Prefix expression.

Algorithm:

1. **Parenthesis checking:**

STEP 1 : Initialize stack r
STEP 2 : Declare string s
STEP 3 : Declare integers o = 0, p = 0

STEP 4 : Prompt user to enter a valid expression s

STEP 5 : For each character c in s:
If c is an opening bracket:
Push c onto the stack r

If c is a closing bracket:
Pop a character y from the stack r
If y and c form a valid pair:
Increment o
If the stack is empty:
Increment p
If p is 1, print "Invalid expression" and exit the loop

STEP 6 : If the stack r is empty:
Increment o
Else:
Increment p
If p is 1, print "Invalid expression"

STEP 7 : Print "Valid expression" if p == 0 and o > 0
Print "Invalid expression" if p > 0

2. **Prefix Evaluation:**
STEP 1 : Initialize rstack s
STEP 2 : Declare string prefixExpression
STEP 3 : Prompt user to enter a prefix expression and store it in prefixExpression

STEP 4 : Function prefixEvaluation(expression):
Initialize integer length to the length of expression
For each character at index i in expression from length - 1 to 0:
If character at index i is a digit:
Convert the character to an integer and push it onto stack s
If character at index i is an operator:
Pop operand1 from stack s
Pop operand2 from stack s
Perform the operation using performOperation function
Push the result back onto stack s
Return the result obtained from pop operation on stack s

STEP 5 : Function performOperation(operation, operand1, operand2):

Switch on operation:
Case '+': Return operand1 + operand2
Case '-': Return operand1 - operand2
Case '*': Return operand1 * operand2
Case '/':
If operand2 is not zero, return operand1 / operand2
Else, print "Error: Division by zero" and return 0
Default:
Print "Error: Invalid operation" and return 0

STEP 6 : Display "Result:" followed by the result obtained from prefixEvaluation function

STEP 7 : End

3. **Prefix to Postfix evaluation**

STEP 1 : Initialize stack s
STEP 2 :Declare string exp
STEP 3 :Prompt user to enter a postfix expression and store it in exp
STEP 4: Reverse the string
STEP 5 :Initialize integers length and index
STEP 6 :Set length to the length of exp
STEP 7 :Set index to length - 1

STEP 8 :While index is greater than or equal to 0:
If exp[index] is not an operator:
Create a string s1 containing exp[index]
Push s1 onto the stack s
If exp[index] is an operator:
Pop op1 and op2 from the stack s
Concatenate op1, op2, and exp[index] to form a new string sub
Push sub onto the stack s
Decrement index by 1

STEP 9 :Display the result by calling the display method of the stack s

Example :

1. Prefix Evaluation

Consider the prefix expression: * + 3 4 5
1. Start with an empty stack.
2. Read the characters from right to left
◦ 5: Operand, push onto the stack.
◦ 4: Operand, push onto the stack.
◦ 3: Operand, push onto the stack.
◦ +: Operator, pop 4 and 3, compute 4 + 3 = 7, and push 7 back.
◦ *: Operator, pop 5 and 7, compute 5 * 7 = 35, and push 35 back.
3. The final result is 35.

2. Parenthesis Matching.

Conclusion:

1. Prefix Evaluation: Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rules.

2. Parenthesis Matching: The stack data structure comes in handy for determining whether or not the syntax has balanced parentheses.

3. Prefix to Postfix Conversion: To convert a prefix to postfix expression, We use a stack to hold the operands. Whenever an operator is found, we pop two operands from the stack and push a new operand. The final element at the top of the stack will be our postfix expression.

CODE:

1. **Parenthesis checking:**

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

class stk
{
  char arr1[100];
  int top;

  public :

  stk()
  {
        top=-1;
  }

  bool isfull()
  {
        if(top==99)
        {
        return 1;
        }

        else
        {
        return 0;
        }
  }

  bool isempty()
  {
        if(top==-1)
        {
        return 1;
        }

        else
```

```
        {
        return 0;
        }
}


void push(char data)
{
        if(isfull())
        {
        cout<<"Stack Overflow"<<endl;
        }

        else
        {
        top = top+1;
        arr1[top]=data;
        }
}

 char pop()
 {
        if(isempty())
        {
        cout<<"Stack Underflow"<<endl;
        return 0;
        }

        else
        {
        int m = arr1[top];
        //arr1[top]=NULL;
        top--;
        return m;
        }
 }
```

```
  char peek()
  {
        if(isempty())
        {
        cout<<"Stack Underflow"<<endl;
        return 0;
        }

        else
        {
        return arr1[top];
        }

  }
};

int main()
{
        stk s;
        string expression;
        int r1=0;
        int r2=0;
        cout<<"Enter the expression"<<endl;
        cin>>expression;
        int x=expression.length();
        for(int i=0;i<x;i++)
        {
        if((expression[i] == '{') || (expression[i] == '(') || (expression[i] == '['))
        {
        s.push(expression[i]);
        r1++;
        }

        else if((expression[i] == '}' && s.peek() == '{') || (expression[i] == ')'
&& s.peek() == '(') || (expression[i] == ']' && s.peek() == '['))
        {
        s.pop();
```

```
        r2++;
        }

        else if( ((expression[i] == '}') || (expression[i] == ')') || (expression[i]
== ']')) && (s.isempty()) )
        {
        r2=0;
        }

        else if((expression[i] == '}' && s.peek() != '{') || (expression[i] == ')' &&
s.peek() != '(') || (expression[i] == ']' && s.peek() != '['))
        {
        s.push(expression[i]);
        r2=0;
        }
        }

        if((r1==r2) && s.isempty())
        {
        cout<<"Valid Expression"<<endl;
        }

        else
        {
        cout<<"Invalid Expression"<<endl;
        }

}
```

2. **Prefix to postfix**

```
#include <iostream>
#include <string>
```

```
#include <bits/stdc++.h>
using namespace std;

class stk
{
        int top = -1;
        int arr[100];

public:
        bool isEmpty()
        {
        return (top == -1);
        }
        bool isFull()
        {
        return (top == sizeof(arr) / sizeof(arr[0]) - 1);
        }
        void push(int data)
        {
        if (isFull())
        {
        cout << "Stack overflow " << endl;
        return;
        }
        arr[++top] = data;
        }
        int getSize()
        {
        return top + 1;
        }
        int pop()
        {
        if (isEmpty())
        {
        cout << "Stack underflow" << endl;
        return -1;
        }
```

```
        return arr[top--];
        }
        int peek()
        {
        if (isEmpty())
        {
        cout << "Stack underflow" << endl;
        return -1;
        }
        return arr[top];
        }
};

int prefix(string s)
{
        for (int i = 0; i < s.size() / 2; i++)
        {
        char temp = s[i];
        s[i] = s[s.size() - i - 1];
        s[s.size() - i - 1] = temp;
        }

        stk st;

        for (int i = 0; i < s.size(); i++)
        {
        if (s[i] == '+')
        {
        int num1 = st.peek();
        st.pop();
        int num2 = st.peek();
        st.pop();

        st.push(num1 + num2);
        }
        else if (s[i] == '-')
        {
```

```
        int num1 = st.peek();
        st.pop();
        int num2 = st.peek();
        st.pop();

        st.push(num1 - num2);
        }
        else if (s[i] == '*')
        {
        int num1 = st.peek();
        st.pop();
        int num2 = st.peek();
        st.pop();

        st.push(num1 * num2);
        }
        else if (s[i] == '/')
        {
        int num1 = st.peek();
        st.pop();
        int num2 = st.peek();
        st.pop();

        st.push(num1 / num2);
        }
        else
        {
        st.push(s[i] - '0');
        }
        }
        return st.peek();
}
int main()
{
        string s;
        cout<<"Enter the string: "<<endl;
        cin>>s;
```

```
        cout <<"Answer is: " <<prefix(s) << endl;

        return 0;
}
```

3. **Prefix Evaluation:**

```
#include <iostream>
#include <string>
#include <bits/stdc++.h>
using namespace std;

class stk
{
  char arr1[100];
  int top;

  public :

  stk()
  {
        top=-1;
  }

  bool isfull()
  {
        if(top==99)
        {
        return 1;
        }

        else
        {
        return 0;
        }
  }
```

```
bool isempty()
{
	if(top==-1)
	{
	return 1;
	}

	else
	{
	return 0;
	}
}


void push(char data)
{
	if(isfull())
	{
	cout<<"Stack Overflow"<<endl;
	}

	else
	{
	top = top+1;
	arr1[top]=data;
	}
}

 char pop()
 {
	if(isempty())
	{
	cout<<"Stock Underflow"<<endl;
	return 0;
	}
```

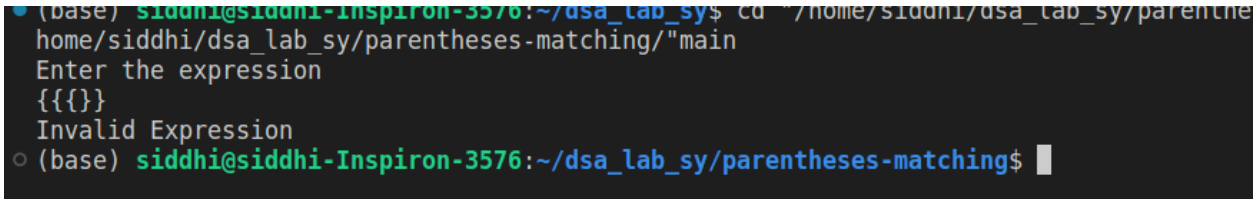
```
        else
        {
        int m = arr1[top];
        //arr1[top]=NULL;
        top--;
        return m;
        }
    }

    char peek()
    {
        if(isempty())
        {
        cout<<"Stack Underflow"<<endl;
        return 0;
        }

        else
        {
        return arr1[top];
        }

    }
};


int prefix(string s)
{
        for (int i = 0; i < s.size() / 2; i++)
        {
        char temp = s[i];
        s[i] = s[s.size() - i - 1];
        s[s.size() - i - 1] = temp;
        }

        stk st;
```

```
for (int i = 0; i < s.size(); i++)
{
if (s[i] == '+')
{
int num1 = st.peek();
st.pop();
int num2 = st.peek();
st.pop();

st.push(num1 + num2);
}
else if (s[i] == '-')
{
int num1 = st.peek();
st.pop();
int num2 = st.peek();
st.pop();

st.push(num1 - num2);
}
else if (s[i] == '*')
{
int num1 = st.peek();
st.pop();
int num2 = st.peek();
st.pop();

st.push(num1 * num2);
}
else if (s[i] == '/')
{
int num1 = st.peek();
st.pop();
int num2 = st.peek();
st.pop();

st.push(num1 / num2);
```

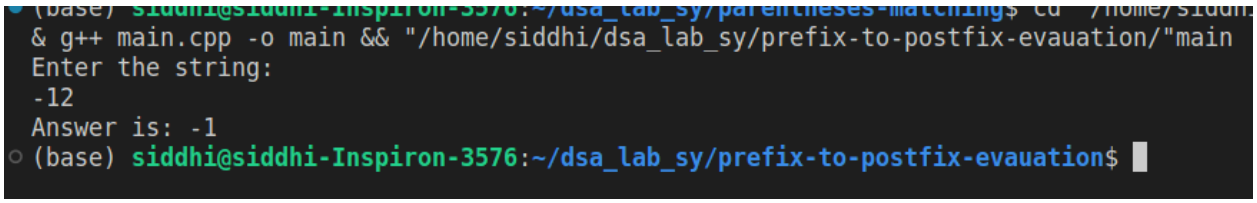
```
            }
            else
            {
            st.push(s[i] - '0');
            }
            }
            return st.peek();
}
int main()
{
            string s;
            cout<<"Enter the string: "<<endl;
            cin>>s;
            cout <<"Answer is: " <<prefix(s) << endl;

            return 0;
}
```

OUTPUT:

Parentheses checking :



Prefix to Postfix evaluation:



Prefix evaluation: