

Experiment No. – 2

Aim:

Implementation of Stack Using Array, Parenthesis Validation, Infix to Postfix

Theory:

1.Stack Using ARRAY:

A stack is a fundamental data structure that follows the last in, first out (lifo) principle. When implemented using an array in c++, it involves a collection of elements where operations like push (adding an element), pop (removing the top element), and peek (viewing the top element without removal) are performed on the same end - typically referred to as the top of the stack.

Operations:

1)push operation:

to add an element onto the stack.

Increment the top index and assign the new element to the array at that index.

Check for overflow conditions.

2)Pop operation:

to remove the top element from the stack.

Return the element at the top index and decrement the top index.

Check for underflow conditions.

3)top operation:

to view the top element without removing it.

Return the element at the top index.

Check for an empty stack condition.

4)Isempty operation:

to check if the stack is empty.

Return true if the top index is -1; otherwise, return false.

5)size operation:

to find the size of the stack.

Return true if the top index is equal to the maximum size of the stack minus one; otherwise, return false.

2. Parenthesis Validation:

Utilizing a stack-based approach, the strategy involves tracking all opening brackets. Upon encountering a closing bracket, the algorithm checks if the top of the stack corresponds to the matching opening bracket. In such cases, the stack is popped, and the iteration continues. At the conclusion of the process, an empty stack signifies well-formed and balanced brackets. Conversely, a non-empty stack indicates an imbalance.

Time Complexity: $O(N)$

Space Complexity: $O(N)$

This methodology operates with a linear time complexity of $O(N)$ and a space complexity of $O(N)$, ensuring efficient bracket balancing checks.

3. Postfix Evaluation:

In postfix expression, operators follow their respective operands. For example:

Postfix expression: 5 7 +

Equivalent infix expression: 5 + 7

To assess a postfix expression:

1. Initiate processing from the leftmost character.
2. When encountering an operand (a numeric value), push it onto the stack.
3. If an operator is encountered, pop the top two operands from the stack, perform the operation, and push the result back onto the stack.
4. The first operand popped represents the second operand, and the second operand popped corresponds to the first operand.
5. Repeat this procedure until all characters are processed.
6. The ultimate outcome is the value remaining on the stack.

Postfix notation, similar to its prefix counterpart, eradicates the necessity for parentheses and ensures an unequivocal approach to expression

evaluation. It is employed in diverse applications, including programming languages, calculators, and expression evaluators.

Algorithm:-

- Stack Implementation

- Infix to Postfix

Algorithm: InfixToPostfixConversion

Input:

- infixExpression: String containing the infix expression

Output:

- postfixExpression: String containing the equivalent postfix expression

Steps:

1. Create an empty string postfixExpression to store the resulting postfix expression.
2. Create an empty stack s1 to hold operators during conversion.
3. Loop through each character c in infixExpression:
 - a. If c is an operand (a letter from 'a' to 'z'):
 - Append c to postfixExpression.
 - b. If c is '(':
 - Push '(' onto s1.
 - c. If c is ')':
 - While s1 is not empty and the top of s1 is not '':
 - Append the top of s1 to postfixExpression.
 - Pop from s1.
 - Pop '(' from s1 (to discard).
 - d. If c is an operator (+, -, *, /):
 - While s1 is not empty and precedence(c) <= precedence(top of s1):
 - Append the top of s1 to postfixExpression.
 - Pop from s1.

- Push c onto s1.
- 4. While s1 is not empty:
 - Append the top of s1 to postfixExpression.
 - Pop from s1.
- 5. Return postfixExpression as the resulting postfix expression.

Function precedence(operator):

Define the precedence of operators: '*' and '/' > '+' and '-'

If operator is '*' or '/', return 2.

If operator is '+' or '-', return 1.

Otherwise, return -1 (for unknown operators).

Example Usage:

- infixExpression = Input from user or predefined infix expression
- postfixExpression = Call InfixToPostfixConversion(infixExpression)
- Display postfixExpression as the equivalent postfix expression.

Example :

- Infix to Postfix

Infix Expression: $a - b / c * d + e$

Infix Expression: $a + b / c$

Answer:

1. Infix Expression: $a - b / c * d + e$

Step-by-Step Conversion:

Reading characters:

- a: Operand, append to output.
- -: Operator, push onto stack.
- b: Operand, append to output.
- /: Operator, push onto stack.
- c: Operand, append to output.
- *: Operator, precedence higher than '/'. Push onto stack.

- d: Operand, append to output.
- +: Operator, precedence lower than ". Pop " and append to output, then push '+'.- e: Operand, append to output.

After processing:

- Pop remaining operators from stack and append to output.

Resulting Postfix Expression:

abc/d*-e+

2. Infix Expression: $a + b / c$

Step-By-Step Conversion:

Reading characters:

- a: Operand, append to output.
- +: Operator, push onto stack.
- b: Operand, append to output.
- /: Operator, push onto stack.
- c: Operand, append to output.

After processing:

- Pop remaining operators from stack and append to output.

Resulting Postfix Expression:

abc/+

- Parenthesis Matching

Algorithm: Check for Balanced Expression

Input:

- expression: String containing the expression to be checked for balance.

Output:

- Returns true if the expression is balanced (all opening brackets have corresponding closing brackets in the right order), otherwise returns false.

Steps:

Create an empty stack s to store opening brackets.

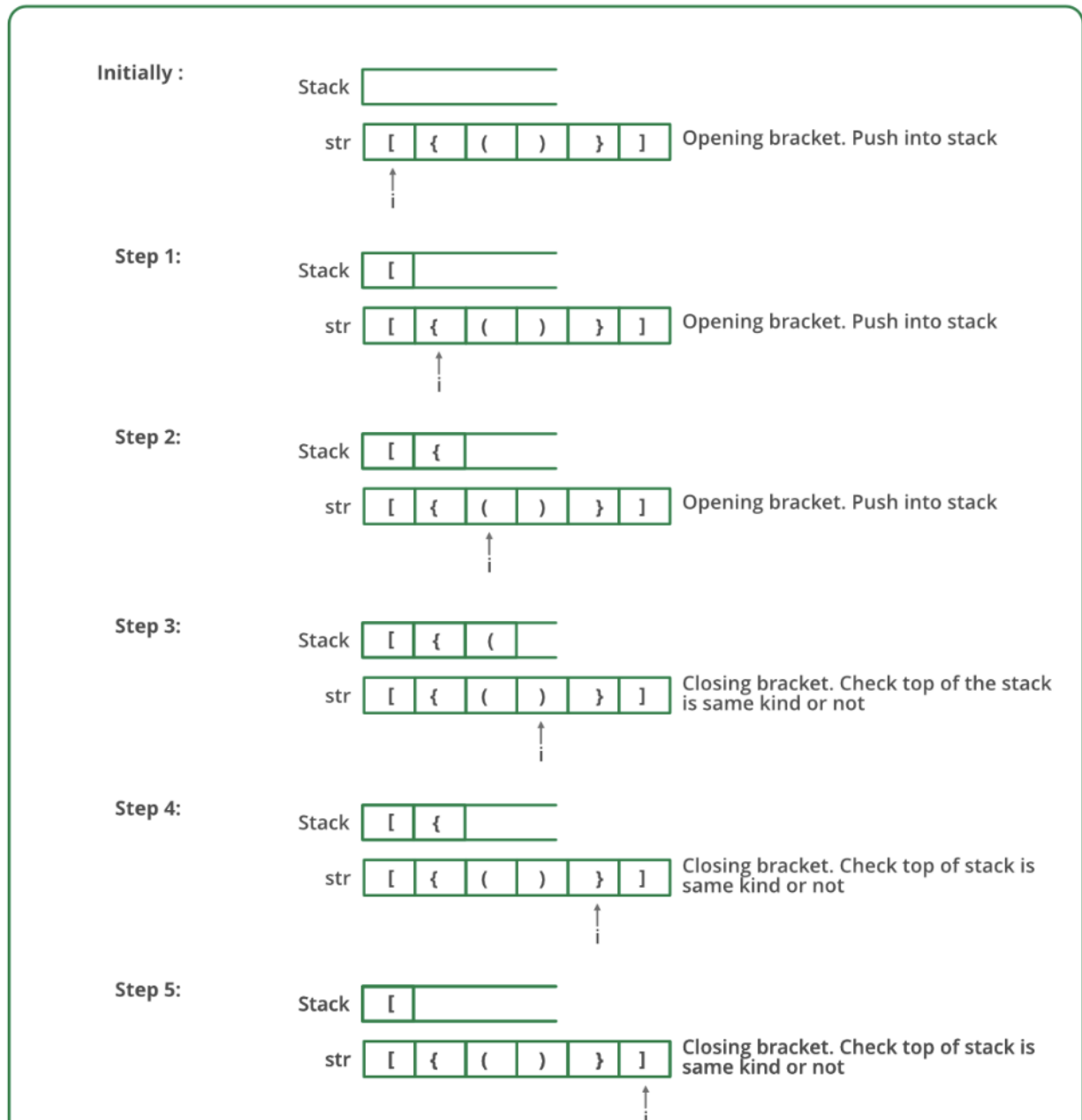
Loop through each character c in the expression:

- If c is an opening bracket ('(', '[', '{'):
 - Push c onto the stack s .
- If c is a closing bracket (')', ']', '}'):
 - If the stack s is empty, return false (imbalance, a closing bracket without a corresponding opening bracket).
 - If c matches the top element of the stack:
 - Pop the top element from the stack.
 - Else, return false (mismatched brackets).

After processing all characters:

- If the stack s is empty, return true (expression is balanced).

- Else, return false (some opening brackets don't have corresponding closing brackets).



Code :-

Stack Implementation

Algorithm: Stack Implementation using Array

Input:

- capacity: Integer value defining the initial capacity of the stack.

Data Structures:

- data[]: Array to store stack elements.
- nextindex: Integer representing the next available index in the stack.
- capacity: Integer representing the total capacity of the stack.

Class: StackUsingArray

Constructor: StackUsingArray(capacity)

- Initialize data array with size capacity.
- Set nextindex to 0 and capacity to the given value.

Function: size()

- Returns the value of nextindex representing the current size of the stack.

Function: isEmpty()

- Returns true if nextindex is 0 (indicating an empty stack), otherwise returns false.

Function: push(element)

- Check if the stack is full (nextindex equals capacity).
- If full, create a new array (twice the capacity), copy existing elements, delete old array, update data pointer and capacity.
- Add the element at data[nextindex] and increment nextindex.

Function: pop()

- Check if the stack is empty.
- If empty, display "Stack is Empty" and return INT_MIN (indicating underflow).
- Decrement nextindex and return data[nextindex].

Function: top()

- Check if the stack is empty.
- If empty, display "Stack is Empty" and return INT_MIN (indicating underflow).
- Return data[nextindex-1].

Code : -

```
#include<iostream>

using namespace std;

class Stack{

    int N;

    queueusingarray q1;

    queueusingarray q2;
```

```
public:

Stack() {

    N = 0;

}

void push(int element) {

    q2.enqueue(element);

    N++;

    while(!q1.isEmpty()) {

        q2.enqueue(q1.front());

        q1.dequeue();

    }

    queueusingarray temp = q2;

    q2 = q1;

    q1 = temp;

}

void pop() {

    q1.dequeue();

    N--;

}

int top() {

    return q1.front();

}

int size() {

    return N;

}
```



```
    }  
  
};  
  
int main() {  
  
    Stack s;  
  
    s.push(1);  
  
    s.push(2);  
  
    s.push(3);  
  
    s.push(4);  
  
    s.push(5);  
  
    s.push(6);  
  
    s.push(7);  
  
    s.push(8);  
  
    s.push(9);  
  
    s.push(10);  
  
    cout<<s.size()<<endl;  
  
    int size = s.size();  
  
    for(int i=0;i<size;i++){  
  
        cout<<s.top()<<" ";  
  
        s.pop();  
  
    }  
  
}
```

Parenthesis Checker

```
bool isBalanced(string expression)
{
    stack<char> s;

    for(int i=0;i<expression.size();i++){

        if(expression[i]=='(' || expression[i]=='[' || expression[i]=='{'){

            s.push(expression[i]);

        }

        else if(expression[i]==')'){

            if(i==0){

                return false;

            }

            if(s.empty()){

                return false;

            }

            if(s.top()=='('){

                s.pop();

            }

            else{

                return false;

            }

        }

        else if(expression[i]==']'){

            if(i==0){

                return false;

            }

            if(s.empty()){

                return false;

            }

            if(s.top()=='['){

                s.pop();

            }

            else{

                return false;

            }

        }

        else if(expression[i]=='}'){

            if(i==0){

                return false;

            }

            if(s.empty()){

                return false;

            }

            if(s.top()=='{'){

                s.pop();

            }

            else{

                return false;

            }

        }

    }

    if(s.empty()){

        return true;

    }

    else{

        return false;

    }

}
```

```
    }

    if(s.empty()){

        return false;

    }

    if(s.top()=='['){

        s.pop();

    }

    else{

        return false;

    }

}

else if(expression[i]=='}'){

    if(i==0){

        return false;

    }

    if(s.empty()){

        return false;

    }

    if(s.top()=='{'){

        s.pop();

    }

    else{

        return false;

    }

}
```

```
    }

    }

    if(s.empty()){

        return true;

    }

    else{

        return false;

    }

}
```

Infix to postfix

```
#include<iostream>

#include<climits>

using namespace std;

class stackusingarray{

int *data;

int nextindex;

int capacity;

public:

stackusingarray() {

    data=new int[5];

    nextindex=0;
```

```
        capacity=5;
    }

    int size() {

        return nextindex;

    }

    bool isEmpty() {

        return nextindex==0;

    }

    void push(int element) {

        if(nextindex==capacity){

            int *newdata=new int[2*capacity];

            for(int i=0;i<capacity;i++){

                newdata[i]=data[i];

            }

            delete [] data;

            data=newdata;

            capacity=2*capacity;

        }

        data[nextindex]=element;

        nextindex++;

    }

    int pop() {

        if(isEmpty()) {
```

```
        cout<<"Stack is Empty"<<endl;

        return INT_MIN;

    }

    nextindex--;

    return data[nextindex];
}

int top(){

    if(isEmpty()){

        cout<<"Stack is Empty"<<endl;

        return INT_MIN;

    }

    return data[nextindex-1];

}

};

int priority(char i){

    switch(i){

        case '+':

        case '-':

            return 1;

            break;

        case '*':

        case '/':
```

```
        return 2;

        break;

    default:

        return -1;

        break;

    }

}

int main() {

    stackusingarray s1;

    string infix;

    cout<<"Enter the infix Expression: "<<endl;

    cin>>infix;

    string postfix;

    for (int i = 0; i < infix.length(); i++) {

        char c = infix[i];

        if (c >= 'a' && c <= 'z')

            postfix += c;

        else if (c == '(')

            s1.push('(');

        else if (c == ')') {

            while (s1.top() != '(') {

                postfix += s1.top();

                s1.pop();

            }

        }

    }

    postfix += c;

    cout<<postfix<<endl;

    return 0;

}
```

```
    }

    s1.pop();

    continue;

}

else {

    while (!s1.isEmpty() && priority(infix[i]) <= priority(s1.top())) {

        postfix += s1.top();

        s1.pop();

    }

    s1.push(c);

}

}

while (!s1.isEmpty()) {

    postfix += s1.top();

    s1.pop();

}

cout << "The Postfix Expression is: "<<postfix << endl;

}

////////// Sample Test Cases //////////

/*

Input -> a-b/c*d+e

Expected Output -> abc/d*-e+
```



```
Input -> a+b/c
```

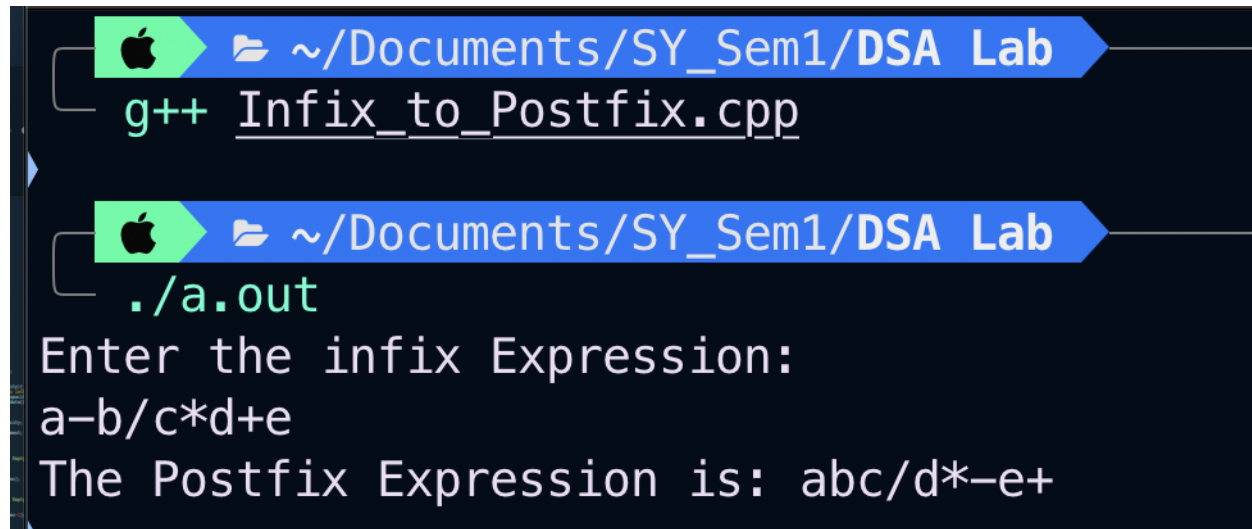
```
Expected Output -> abc/+
```

```
*/
```

Output:-

```
g++ StackUsingArray.cpp
./a.out
```

```
g++ Parenthesis_Checker.cpp
./a.out
()((()))()
true%
```



```
~/Documents/SY_Sem1/DSA Lab
g++ Infix_to_Postfix.cpp

~/Documents/SY_Sem1/DSA Lab
./a.out
Enter the infix Expression:
a-b/c*d+e
The Postfix Expression is: abc/d*-e+
```

Conclusion:

1. **Stack Using Array:** a stack implemented using an array in C++ offers a flexible and efficient solution for managing elements in a Last In, First Out (LIFO) fashion. The push and pop operations, along with proper initialization and boundary checks, provide a reliable structure for various applications. This simple yet powerful data structure proves valuable in scenarios where orderly element processing is essential.
2. **Parenthesis Validation:** the parenthesis validation experiment showcased the reliability of a stack-based algorithm in efficiently checking the balanced structure of expressions. The algorithm, with a time and space complexity of $O(N)$, proved to be a practical and scalable solution, emphasizing the versatility of stack-based approaches for parentheses validation in mathematical expressions.
3. **Infix to postfix :** Postfix expression evaluation employs a systematic approach, scanning the expression from left to right. Utilizing a stack to manage operands and perform operations, this

process ensures an efficient and accurate computation of the postfix expression. The final result is derived by iteratively applying operators to operands, providing a straightforward and effective method for expression evaluation.