Experiment No: 03

Aim:

Implement stack using a linked list. Use this stack to perform evaluation of a postfix expression.

Objective:

To understand the concept of abstraction of data type. How different data structures such as arrays and a stacks are represented as an ADT.

Theory:

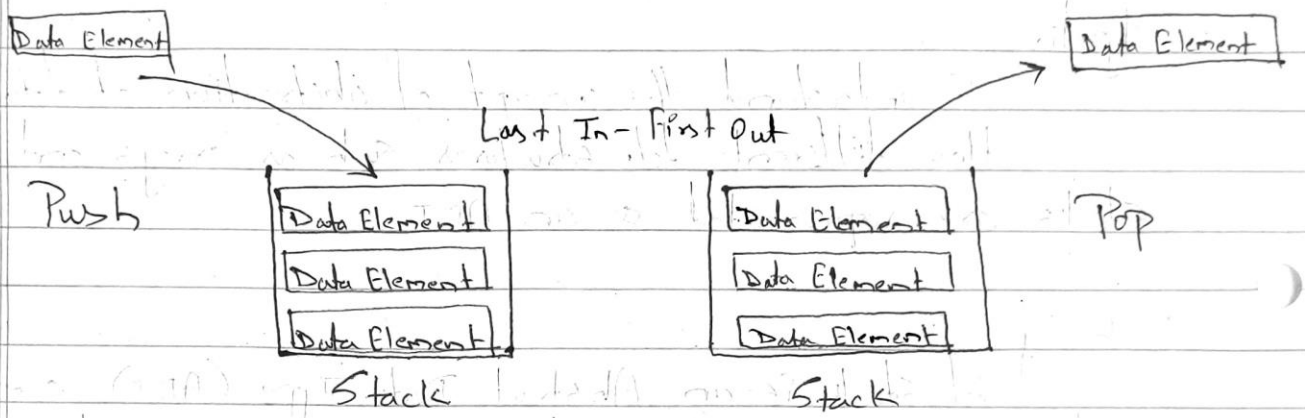
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example - a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can ~~replace~~ or remove a card or plate from the top of stack only. Likewise, stack ADT allows all data operation at one end only. At any given time, we can access the top element of stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-First-out. Here, the element which is placed last, is accessed first. In stack terminology, insertion operation is called PUSH operation & removal operation is called POP operation.

Stack Representation:

The following diagram depicts a stack & its operation.



A stack can be implemented by mean of Array, Structure, Pointer and Linked list. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using array, which makes it a fixed size stack implementation.

Basic Operations:

Stack operation may involve initializing the stack, using it & then de-initializing it. Apart from these basic stuff, a stack is used for following two primary operations.

push() - Pushing an element on stack.

pop() - Removing an element from stack.

When data is PUSHed onto stack.

To use a stack efficiency, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks.

peek() - get the top data element of stack, without removing it.

isFull() - check if stack is full.

isEmpty() - check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on stack. As this pointer always represents the top of stack, hence named top. The top pointer provides top value of stack without actually removing it.

First we should learn about procedure to support stack functions:

```
bool isfull() {
    if (top == MAXSIZE)
        return true;
    else
        return false;
}
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array start from 0. So, we check if the top is below zero or -1 to determine if stack is empty. Here is the code-

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

Push Operation:

The process of putting a new data element onto stack is known as push operation.

```
void push(int data){
 if(!isfull()){
   top = top + 1;
   stack[top] = data;
 }else{
   printf("Couldnot insert data. Stack is full.\n");
 }
}
```

Pop Operation:

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in stack to point the next value. But, in linked-list implementation, Pop() actually removes data element & deallocates memory space.

```
int pop (int data){
 if (! isempty()){
    data = stack[top];
    top = top-1;
    return data;
 }else{
    printf("Couldnot retrieve data. Stack is empty.\n");
 }
}
```

Program Code:

```
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

using namespace std;
struct node {
  int data;
  struct node * next;
};
struct node * top = NULL;
/* create a new node with the given data */
struct node * createNode(int data) {
  struct node * ptr = (struct node * ) malloc(sizeof(struct node));
  ptr -> data = data;
  ptr -> next = NULL;
}
/* push the input data into the stack */
void push(int data) {
  struct node * ptr = createNode(data);
  if (top == NULL) {
    top = ptr;
    return;
  }
  ptr -> next = top;
  top = ptr;
}
/* pop the top element from the stack */
int pop() {
  int data;
  struct node * temp;
  if (top == NULL)
    return -1;
  data = top -> data;
  temp = top;
  top = top -> next;
  free(temp);
  return (data);
}
int main() {
  // 6 2 * 3 4 10 / - +
  char str[100];
  int i, data = -1, operand1, operand2, result;
  /* i/p postfix expr from the user */
  cout << "Enter your postfix expression: ";
  fgets(str, 100, stdin);
  for (i = 0; i < strlen(str); i++) {
```

```
    if (isdigit(str[i])) {
      /* if the i/p char is digit, parse character by character to get complete operand*/
      data = (data == -1) ? 0 : data;
      data = (data * 10) + (str[i] - 48);
      continue;
    }
    /* push the operator into the stack */
    if (data != -1) {
      push(data);
    }
    if (str[i] == '+' || str[i] == '-' || str[i] == '*' || str[i] == '/') {
      /*
       * if the i/p character is an operator,
       * then pop two elements from the stack,
       * apply operator and push the result into
       * the stack
       */
      operand2 = pop();
      operand1 = pop();
      if (operand1 == -1 || operand2 == -1)
        break;
      switch (str[i]) {
      case '+':
        result = operand1 + operand2;
        /* pushing result into the stack */
        push(result);
        break;
      case '-':
        result = operand1 - operand2;
        push(result);
        break;
      case '*':
        result = operand1 * operand2;
        push(result);
        break;
      case '/':
        result = operand1 / operand2;
        push(result);
        break;
      }
    }
    data = -1;
  }
  if (top != NULL && top -> next == NULL)
    cout << "Output:" << top -> data;
  else
    cout << "You have entered wrong expression\n";
  return 0;
}
```

Program Output: