Lab Assignment 3 (Stacks)

1. Develop a menu driven program demonstrating the following operations on a Stack using array:
(i) push(), (ii) pop(), (iii) isEmpty(), (iv) isFull(), (v) display(), and (vi) peek().

2. Given a string, reverse it using STACK. For example “DataStructure” should be output as “erutcurtSataD.”

3. Write a program that checks if an expression has balanced parentheses.

4. Write a program for the evaluation of a Postfix expression.

5. Write a program to convert an Infix expression into a Postfix expression.

1. Develop a menu driven program demonstrating the following operations on a Stack using array:
push(), (ii) pop(), (iii) isEmpty(), (iv) isFull(), (v) display(), and (vi) peek().

Stacks_Array (Menu).c

2. Given a string, reverse it using STACK. For example “DataStructure” should be output as “erutcurtSataD.”

- ✔ *The idea is to create an empty stack and push all the characters from the string into it.*
- ✔ *Then for each character in the stack*
  Extract it top () and append to the string then perform pop().

Character Extraction From String

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
	char str[20];
	printf("Enter the string: ");
	scanf("%s",str);

	// character extraction
	printf("Printing the characters:: \n");
	for (int i = 0; str[i] != '\0'; i++) {
			printf("%c\n", str[i]);  // printing each characters in a new line
		}

	return 0;
}
```

Concatenation

```
#include <string.h>
int main()
{
    char str[100]="";
    printf("String: %s\n", str);

     char ch = 'a';
     int len = strlen(str); // Find the length of the string
     str[len] = ch;
     str[len + 1] = '\0';   // Null-terminate the string
    printf("Concatenated String: %s\n", str);


    ch = 'b';
    len = strlen(str); // Find the length of the string
    str[len] = ch;         // Null-terminate the string
    str[len + 1] = '\0';
    printf("Concatenated String: %s\n", str);
       return 0;
}
```

3. Write a program that checks if an expression has balanced parentheses.

Input: exp = “[()]{}{[()()]()}”
Output: Balanced
Explanation: all the brackets are well-formed

Input: exp = “[(])”
Output: Not Balanced
Explanation: 1 and 4 brackets are not balanced because there is a closing ‘]’ before the closing ‘(‘

- Store the Expression into String

- Now traverse the string exp.

  - If the current character is a starting bracket (**‘(‘ or ‘{‘ or ‘[‘**) then push it to stack.

```
if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
        push()
```

  - If the current character is a closing bracket (**‘)’ or ‘}’ or ‘]’**) then top () + pop () from the stack and if the popped character is the matching starting bracket then fine.

```
if (exp[i] == '}' || exp[i] == ')'  || exp[i] == ‘]’)
        top()
        pop ()
        if ( isMatchingPair(char1,char2) )
        Else brackets are Not Balanced.
```

- After complete traversal, if some starting brackets are left in the stack then the expression is **Not balanced**, else **Balanced**.

```
bool isMatchingPair(char character1, char character2)
{
    if (character1 == '(' && character2 == ')')
        return 1;
    else if (character1 == '{' && character2 == '}')
        return 1;
    else if (character1 == '[' && character2 == ']')
        return 1;
    else
        return 0;
}
```

4. Write a program for the evaluation of a Postfix expression.

Postfix Evaluation Algorithm

For every token in the postfix expression (scanned from left to right):

a) If the token is an operand (push it on the stack)

b) Otherwise, if the token is an operator (or function):

- Check if the stack contains the sufficient number of values (usually two) for given operator. If there are not enough values, finish the algorithm with an

A: Top element
B: Next to top element
Result = B operator A

- Else Pop the appropriate number of values from the stack and evaluate the operator using the popped values and push the single result on the stack.

At last if the stack contains only one value, return it as a final result of the calculation Otherwise, finish the algorithm with an error

char exp[] = "231*+9-“ ;

```
for ( ) // Scan all characters one by one
  {
     if (isdigit(scanned character))   // If the scanned character is an operand push it to the stack.
        push();
    // If the scanned character is an operator, pop two elements from stack apply the operator
     else {
        int val1 = top();
        pop();
        int val2 = top();
        pop();
        switch (scanned character)
{
        case '+':
           push(stack, val2 + val1);
           break;
        case '-':
           push(stack, val2 - val1);
           break;  } } }
```

C isdigit() Return value

Return Value	Remarks
Non-zero integer (x > 0)	Argument is a numeric character.
Zero (0)	Argument is not a numeric character.

```
#include <stdio.h>
int main()
{
    char c;

    printf("Enter a character: ");
    scanf("%c",&c);

    if (isdigit(c) == 0)
        printf("%c is not a digit.",c);
    else
        printf("%c is a digit.",c);
    return 0;
}
```

5. Write a program to convert an Infix expression into a Postfix expression

Infix to Postfix Conversion

1.Add opening and closing bracket to the given infix expression if already not there then scan given Input from left to right and repeat step 2 to 5 for each element until
the STACK is empty.

2. If an **operand** is encountered add it to OUTPUT.
3. If a **left parenthesis** is encountered push it onto the STACK.
4. If an **operator** is encountered then:
    i. Repeatedly pop from STACK and add to OUTPUT each operator which has same or higher precedence than the operator encountered.
    ii. Push the encountered operator onto the STACK.
5. If a **right parenthesis** is encountered, then

i. Repeatedly pop from the STACK and add to OUTPUT each operator until a left parenthesis is encountered.
ii. Remove the left parenthesis; do not add it to OUTPUT.

Input	Stack	Output
A + (B * (C – D) / E)		
A + (B * (C – D) / E))	(	
+ (B * (C – D) / E))	(	A
(B * (C – D) / E))	(+	A
B * (C – D) / E))	(+ (	A
* (C – D) / E))	(+ (	A B
(C – D) / E))	(+ (*	A B
C – D) / E))	(+ (* (	A B

Input	Stack	Output
– D) / E))	(+ (* (	A B C
D) / E))	(+ (* (–	A B C
) / E))	(+ (* (–	A B C D
/ E))	(+ (*	A B C D –
E))	(+ (/	A B C D – *
))	(+ (/	A B C D – * E
)	(+	A B C D – * E /
		A B C D – * E / +