

NAME :- Manish Shashikant Jadhav

UID :-

BRANCH :- Comps -B.      BRANCH: B.

EXPERIMENT 1: Implement of given problem statement using Stack.

SUBJECT :- DS (DATA STRUCTURES)

TOPIC 1 :- Implementation of Stack using array.

CODE :-

```
#include<stdio.h>
#include<stdlib.h>
#include <limits.h>

struct Stack
{
    int top;f
    unsigned size;
    int* array;
};

// 1 -> Initialize
// TODO: Complete the function to initialize a new stack with a given size
struct Stack* createStack(unsigned size)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array = (int*)malloc(stack->size * sizeof(int));
    return stack;
}

// 2 -> isFull
// TODO: Complete the function. It should check if the stack provided to it is
full
int isFull(struct Stack* stack)
{
    return stack->top == stack->size -1;
}

// 3 -> isEmpty
// TODO: Complete the function. It should check if the stack provided to it is
empty
int isEmpty(struct Stack* stack)
{

```

```

        return stack->top == -1;
    };

// 4 -> push
void push(struct Stack *stack, int item)
{
    if (isFull(stack))
        return ;
    stack->array[++stack->top] = item;
    printf("%d Pushed to stack\n", item);
}

// 5 -> peek
int peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

// 6 -> pop
// TODO: Complete the function. It should pop an element from the stack and return
// it. For an empty stack, it should throw an error message
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

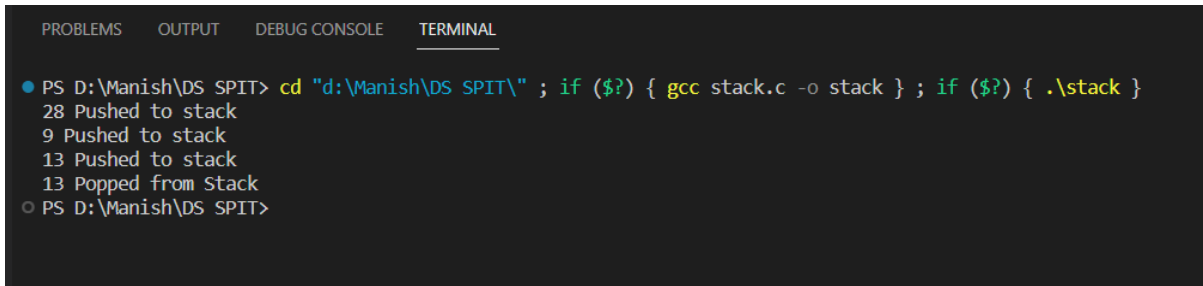
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 28);
    push(stack, 9);
    push(stack, 13);

    printf("%d Popped from Stack\n", pop(stack));

    return 0;
}

```

**OUTPUT :-**


```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
● PS D:\Manish\DS SPIT> cd "d:\Manish\DS SPIT\" ; if ($?) { gcc stack.c -o stack } ; if ($?) { .\stack }
28 Pushed to stack
9 Pushed to stack
13 Pushed to stack
13 Popped from Stack
○ PS D:\Manish\DS SPIT>

```

**Algorithm:-****1. Initialize a stack:**

- Create a function `createStack` that takes an integer `size` as a parameter.
- Allocate memory for a new `struct Stack`.
- Set the stack's size to the given `size`.
- Initialize the stack's top index to -1.
- Allocate memory for the stack's array based on the given size.
- Return the newly created stack.

**2. Check if the stack is full (isFull):**

- Create a function `isFull` that takes a pointer to a `struct Stack` as a parameter.
- Check if the stack's top index is equal to `size - 1`.
- If it is, return true (1); otherwise, return false (0).

**3. Check if the stack is empty (isEmpty):**

- Create a function `isEmpty` that takes a pointer to a `struct Stack` as a parameter.
- Check if the stack's top index is -1.
- If it is, return true (1); otherwise, return false (0).

**4. Push an element onto the stack:**

- Create a function `push` that takes a pointer to a `struct Stack` and an integer `item` as parameters.
- Check if the stack is full using the `isFull` function.
- If the stack is not full, increment the top index and add `item` to the stack's array.
- Optionally, you can print a message indicating that the item was pushed.

**5. Peek at the top element of the stack:**

- Create a function `peek` that takes a pointer to a `struct Stack` as a parameter.

- Check if the stack is empty using the `isEmpty` function.
- If the stack is not empty, return the element at the top of the stack.
- If the stack is empty, return a sentinel value (e.g., INT\_MIN) to indicate an empty stack.

**6. Pop an element from the stack:**

- Create a function `pop` that takes a pointer to a `struct Stack` as a parameter.
- Check if the stack is empty using the `isEmpty` function.
- If the stack is not empty, decrement the top index and return the element removed from the stack.
- If the stack is empty, return a sentinel value (e.g., INT\_MIN) to indicate an empty stack.

**7. In the `main` function:**

- Create a stack using the `createStack` function with a specified size (e.g., 100).
- Push elements onto the stack using the `push` function.
- Optionally, print messages to indicate when elements are pushed and popped.
- Pop an element from the stack using the `pop` function and print the result.

**8. Free memory:**

- In the `main` function, after using the stack, free the allocated memory for the stack and its array using the `free` function.

**9. Terminate the program by returning 0 from the `main` function.**

**TOPIC 2 :- Conversion of Infix Expression to Postfix Expression.****CODE :-**

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Function to check if a character is an operator
int isOperator(char c) {
    return (c == '^' || c == '*' || c == '/' || c == '+' || c == '-');
}

// Function to return the precedence of an operator based on the map
int getPrecedence(char c) {
    switch (c) {
        case '^':
            return 1;
        case '*':
        case '/':
            return 2;
        case '+':
        case '-':
            return 3;
        default:
            return 4;
    }
}

// Function to convert infix to prefix
void infixToPrefix(char infix[], char prefix[]) {
    int len = strlen(infix);
    char stack[len]; // Stack to hold operators
    int top = -1;    // Stack top pointer
    int outputIndex = 0;
    // Reverse the infix expression
    for (int i = len - 1; i >= 0; i--) {
        if (infix[i] == ')') {
            stack[++top] = infix[i];
        } else if (infix[i] == '(') {
            while (top >= 0 && stack[top] != ')') {
                prefix[outputIndex++] = stack[top--];
            }
            if (top >= 0) {
                top--; // Pop the '('
            }
        }
    }
    prefix[outputIndex] = '\0';
}

```

```

    }
    } else if (isOperator(infix[i])) {
        while (top >= 0 && getPrecedence(stack[top]) <
getPrecedence(infix[i])) {
            prefix[outputIndex++] = stack[top--];
        }
        stack[++top] = infix[i];
    } else {
        // Operand
        prefix[outputIndex++] = infix[i];
    }
}
// Pop any remaining operators from the stack
while (top >= 0) {
    prefix[outputIndex++] = stack[top--];
}
// Null-terminate the prefix expression
prefix[outputIndex] = '\0';

// Reverse the prefix expression to get the final result
strrev(prefix);
}

int main() {
    char infix[100], prefix[100];
    printf("Enter an infix expression: ");
    scanf("%s", infix);

    infixToPrefix(infix, prefix);
    printf("Prefix expression: %s\n", prefix);
    return 0;
}

```

**OUTPUT :-**

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
● PS D:\Manish\DS SPIT> cd "d:\Manish\DS SPIT\" ; if ($?) { gcc infix_to_prefix.c -o infix_to_prefix } ; if ($?) { .\infix_to_prefix }
Enter an infix expression: ((A-(B+C))*D)^(E+F)
Prefix expression: ^*-A+BCD+EF
○ PS D:\Manish\DS SPIT> █

```

**Algorithm:-**

**1. Start the program.**

**2. Define the following functions:**

- a. `isOperator(char c)`: Returns 1 if the character 'c' is one of '^', '\*', '/', '+', or '-', else returns 0.
- b. `getPrecedence(char c)`: Returns the precedence of the operator 'c' based on the map.
- c. `infixToPrefix(char infix[], char prefix[])`: Converts the infix expression to a prefix expression.

**3. In the main() function:**

- a. Declare variables ``infix`` and ``prefix`` as character arrays of size 100.
- b. Prompt the user to enter an infix expression and store it in the ``infix`` array using `scanf`.
- c. Call the `infixToPrefix()` function with the ``infix`` and ``prefix`` arrays as arguments.
- d. Print the resulting prefix expression.

**4. Inside the `infixToPrefix()` function:**

- a. Calculate the length of the input infix expression and store it in the variable ``len``.
- b. Create a character array ``stack`` of size ``len`` to hold operators.
- c. Initialize the variable ``top`` to -1 as the stack top pointer.
- d. Initialize the variable ``outputIndex`` to 0 to keep track of the current position in the prefix array.

**5. Reverse the input infix expression by iterating from ``len-1`` to 0:**

- a. If the current character is ')', push it onto the stack.
- b. If the current character is '(', pop elements from the stack and append them to the prefix expression until a ')' is encountered. Pop the ')' from the stack.
- c. If the current character is an operator, pop operators from the stack and append them to the prefix expression until an operator with lower precedence is encountered. Then, push the current operator onto the stack.
- d. If the current character is an operand, append it directly to the prefix expression.

**6. After processing all characters in the infix expression, pop any remaining operators from the stack and append them to the prefix expression.**

**7. Null-terminate the prefix expression by setting ``prefix[outputIndex]`` to ``\0``.**

**8. Reverse the prefix expression using a function like ``strrev(prefix)`` to obtain the final result.**

**9. End the `infixToPrefix()` function.**

## 10. End the main program.

M	T	W	T	F	S	S
Page No.:						YOUVA
Date:						

Experiment No. 1

- \* Problem Statement :- Implementing stack and operations using array.
- \* Theory :-
  - Stack :-
    - A stack is a linear data structure in which the insertion of a new element and removal of existing element takes place at the same end, represented as top of the stack.
    - Stack implements LIFO (Last In First out) strategy that the element that is inserted last will come out first.
  - Basic operation on Stack :-
    - push() to insert an element into stack.
    - pop() to remove an element from stack.
    - top() returns top element of the stack.
    - isEmpty() returns true if stack is empty else false.
    - isFull() returns true if stack is full else false.
    - size() returns the size of stack.
  - Advantages :-
    - Easier to implement.
    - Efficient memory utilization.
    - Provides fast access time.
    - Enables undo/redo operations.





M	T	W	T	F	S	S
Page No.:						YOUVA
Date:						

### • Disadvantages:-

- has limited capacity.
- No random access.
- uses contiguous block of memory, results in memory fragmentation if elements are added and removed frequently.

### • Infix to Prefix Notation:-

a) Infix Expression:- The expression of type a 'operator' b ( $a + b$ ) i.e. when operator is in between two operands.

b) Prefix Expression:- The expression of type 'operator' ab ( $+ab$ ) i.e. when operator is placed before the operands.

### • Steps:-

- Reverse Infix expression.
- Convert reversed infix expression to postfix.
- Reverse Postfix expression.

\*\* Solutions:-

Manish. S. Jadhav.

- Infix Expression:-

$$(A - (B + C)) * D \wedge (E + F)$$

- Prefix Expression:-

$$((A - (B + C)) * D) \wedge (E + F)$$

$$\text{ff} \rightarrow A$$

$$((A - + B C) * D) \wedge (E + F)$$

$$((- A + B C) * D) \wedge (E + F)$$

$$* - A + B C D \wedge (E + F)$$

$$* - A + B C D \wedge + E F$$

$$\wedge * - A + B C D + E F.$$

\* Conclusion:-

Hence, by completing this experiment I came to know about Implementation of stack using array and conversion of Infix expression to prefix expression.