Experiment 4

Aim:

Implementing stack in various ways.

Theory:

Stack is a data structure which is based on LIFO (Last In First Out) system. To access the items of the stack we have top of the stack. We can add or delete items from the top of the stack only. The operations that can be performed on the stack are push(adding an item to the stack) and pop(deleting an item from the stack).

Assignment 1

Problem:

Convert the given infix expression into postfix expression using stack.

Example-

Input: $a + b * (c\hat{}d - e)\hat{}(f + g * h) - i$

Output: $abcd\hat{}e - fgh * +\hat{} * +i -$

SOURCE CODE:

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


struct Stack {
   int top;
   char items[100];
};
```

```
void initialize(struct Stack *s) {
    s->top = -1;
}


int isEmpty(struct Stack *s) {
    return (s->top == -1);
}

void push(struct Stack *s, char c) {
    if (s->top == 99) {
        printf("Stack overflow\n");
        return;
    }
    s->items[++(s->top)] = c;
}

char pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        return '\0'; // Assuming '\0' represents an error value
    }
    return s->items[(s->top)--];
}
```

```
int precedence(char op) {
    if (op == '^')
        return 3;
    else if (op == '*' || op == '/')
        return 2;
    else if (op == '+' || op == '-')
        return 1;
    else
        return 0;
}


void infixToPostfix(char *infix, char *postfix) {
    struct Stack stack;
    initialize(&stack);
    int i = 0;
    int j = 0;

    while (infix[i] != '\0') {
        char c = infix[i];

        if (isalnum(c)) {
            postfix[j++] = c;
        } else if (c == '(') {
            push(&stack, c);
        } else if (c == ')') {
            while (!isEmpty(&stack) && stack.items[stack.top] != '(') {
                postfix[j++] = pop(&stack);
```

```
            }
            if (!isEmpty(&stack) && stack.items[stack.top] != '(') {
                printf("Invalid expression\n");
                return;
            } else {
                pop(&stack);
            }
        } else {
            while (!isEmpty(&stack) && precedence(c) <= precedence(stack.items[stack.top])) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, c);
        }

        i++;
    }

    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }

    postfix[j] = '\0';
}

int main() {
    char infix[100];
    char postfix[100];
```

```
    printf("Enter an infix expression: ");
    scanf("%s", infix);

    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

OUTPUT:

```
Enter an infix expression: a+b*(c^d-e)^(f+g*h)
Postfix expression: abcd^e-fgh*+^*+
```

Assignment 2

Problem:

Write a program to evaluate the following given postfix expressions:

2 3 1 ∗ + 9 –

Output: -4

2 2 + 2 / 5 ∗ 7 +

Output: 17

SOURCE CODE:

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

struct Stack {
    int top;
    int items[100];
};

void initialize(struct Stack *s) {
    s->top = -1;
```

```
}

int isEmpty(struct Stack *s) {
   return (s->top == -1);
}

int pop(struct Stack *s) {
   if (isEmpty(s)) {
      printf("Stack underflow\n");
      return -1; // Assuming -1 represents an error value
   }
   return s->items[s->top--];
}

void push(struct Stack *s, int value) {
   if (s->top == 99) {
      printf("Stack overflow\n");
      return;
   }
   s->items[++(s->top)] = value;
}

int evaluatePostfix(char *expression) {
   struct Stack stack;
   initialize(&stack);

   int length = strlen(expression);
   for (int i = 0; i < length; i++) {
      char token = expression[i];
      if (token >= '0' && token <= '9') {
         push(&stack, token - '0'); // Convert char to int
      } else if (token == '+' || token == '-' || token == '*' || token == '/') {
         int operand2 = pop(&stack);
         int operand1 = pop(&stack);
         int result;
         switch (token) {
            case '+':
               result = operand1 + operand2;
               break;
            case '-':
               result = operand1 - operand2;
               break;
            case '*':
               result = operand1 * operand2;
               break;
            case '/':
               result = operand1 / operand2;
               break;
```

```
        }
        push(&stack, result);
      }
    }

    return pop(&stack);
}

int main() {
    char expression[] = "22 + 2 / 5 * 7 +";
    int result = evaluatePostfix(expression);
    printf("Result: %d\n", result);

    return 0;
}
```

OUTPUT:

Result: 17

Assignment 3

Problem:

Given an expression, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in the expression or not.

Example: Input: exp = “[( )]{ }{[( )( )]( )}”

Output: Balanced

Input: exp = “[( ])”

Output: Not Balanced

SOURCE CODE:

```
#include <stdio.h>

#include <stdbool.h>

#include <string.h>


struct Stack {

    char data;
```

```
    struct Stack* next;
};

struct Stack* newNode(char data) {
    struct Stack* node = (struct Stack*)malloc(sizeof(struct Stack));
    node->data = data;
    node->next = NULL;
    return node;
}

bool isEmpty(struct Stack* stack) {
    return stack == NULL;
}

void push(struct Stack** stack, char data) {
    struct Stack* node = newNode(data);
    node->next = *stack;
    *stack = node;
}

char pop(struct Stack** stack) {
    if (isEmpty(*stack)) {
        return '\0'; // Null character to indicate an empty stack
    }
    struct Stack* temp = *stack;
    char popped = temp->data;
    *stack = temp->next;
    free(temp);
    return popped;
```

```
}
bool areParenthesesAndBracesBalanced(const char* expression) {
    struct Stack* stack = NULL;

    for (int i = 0; expression[i] != '\0'; i++) {
        char currentChar = expression[i];
        if (currentChar == '(' || currentChar == '{' || currentChar == '[') {
            push(&stack, currentChar);
        } else if (currentChar == ')' || currentChar == '}' || currentChar == ']') {
            char topChar = pop(&stack);
            if (isEmpty(&stack) || (currentChar == ')' && topChar != '(') ||
                (currentChar == '}' && topChar != '{') ||
                (currentChar == ']' && topChar != '[')) {
                return false;
            }
        }
    }
    return isEmpty(stack);
}

int main() {
    const char* expression1 = "[( )]{ }{[( )( )]( )}";
    const char* expression2 = "[( ])";

    if (areParenthesesAndBracesBalanced(expression1)) {
        printf("Expression 1: Balanced\n");
    } else {
        printf("Expression 1: Not Balanced\n");
    }
```

```
    if (areParenthesesAndBracesBalanced(expression2)) {
        printf("Expression 2: Balanced\n");
    } else {
        printf("Expression 2: Not Balanced\n");
    }

    return 0;
}
```

OUTPUT:

Expression 1: Balanced

Expression 2: Not Balanced