EXPERIMENT 5

Aim:

1. Write a program to evaluate the following given postfix expressions: a. 2 3 1 $*$ + 9 – Output: -4 b. 2 2 + 2 / 5 $*$ 7 + Output: 17
2. Convert the given infix expression into postfix expression using stack. Example- Input: a + b $*$(c^d −e)^(f +$g$$*$$h$)−$i$ Output: $abcd$^e − fgh $*$ +^$*$+i −
3. 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 = "[( )]{ }{[( )( )]( )}" Input: exp = "[( ])" Output: Balanced Output: Not Balanced

Theory (Evaluation of Postfix Expression):

Postfix (Reverse Polish Notation) expressions place operators after operands, removing the need for parentheses. To evaluate:

1. **Scan left to right**.
2. **Push operands** onto a stack.
3. **When encountering an operator**, pop two operands, apply the operator, and push the result back.
4. **Final result** will be on the stack after processing the entire expression.

Program:

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

typedef struct Stack
{
   int top;
   int capacity;
   int *array;
} Stack;

Stack *createStack(int n);
int isEmpty(Stack *s);
int isFull(Stack *s);
int pop(Stack *s);
void push(Stack *s, int value);
int evaluatePostfix(char *postfix);
int performOperation(int a, int b, char op);

int main(void)
{
   char postfix[100];
   printf("Enter Postfix Expression: ");
   fgets(postfix, sizeof(postfix), stdin);
```

```
    size_t len = strlen(postfix);
    if (len > 0 && postfix[len - 1] == '\n')
    {
        postfix[len - 1] = '\0';
    }

    int result = evaluatePostfix(postfix);
    printf("The result is: %d\n", result);

    return 0;
}

Stack *createStack(int n)
{
    Stack *s = (Stack *)malloc(sizeof(Stack));
    s->capacity = n;
    s->top = -1;
    s->array = (int *)malloc(s->capacity * sizeof(int));
    return s;
}

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

int isFull(Stack *s)
{
    return s->top == s->capacity - 1;
}

int pop(Stack *s)
{
    if (isEmpty(s))
    {
        printf("Stack is Empty\n");
        return -1;
    }
    return s->array[s->top--];
}

void push(Stack *s, int value)
{
    if (isFull(s))
    {
        printf("Stack Overflow!\n");
        return;
    }
    s->array[++s->top] = value;
}
```

```
int evaluatePostfix(char *postfix)
{
   int strlength = strlen(postfix);
   Stack *mystack = createStack(strlength);

   for (int i = 0; postfix[i] != '\0'; i++)
   {
      char ch = postfix[i];

      if (isdigit(ch))
      {
         push(mystack, ch - '0');
         // value of 0 in ASCII is 48
      }
      else if (ch == ' ')
      {
         continue;
      }
      else
      {
         int val2 = pop(mystack);
         int val1 = pop(mystack);
         int result = performOperation(val1, val2, ch);
         push(mystack, result);
      }
   }

   int finalResult = pop(mystack);
   free(mystack->array);
   free(mystack);

   return finalResult;
}

int performOperation(int a, int b, char op)
{
   switch (op)
   {
   case '+':
      return a + b;
   case '-':
      return a - b;
   case '*':
      return a * b;
   case '/':
      return a / b;
   default:
      printf("Invalid operator encountered: %c\n", op);
      return 0;
   }
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab6> gcc postfixeval.c -o postfixeval
PS B:\sem3\ds\23bcp153_dsa\lab6> ./postfixeval
Enter Postfix Expression: 2 3 1 * + 9 -
The result is: -4
PS B:\sem3\ds\23bcp153_dsa\lab6> ./postfixeval
Enter Postfix Expression: 2 2 + 2 / 5 * 7 +
The result is: 17
```

Theory (Infix to Postfix):

Infix notation (e.g., A + B) requires parentheses for precedence, while postfix (e.g., AB+) doesn't. To convert:

1. **Scan the infix expression** left to right.
2. **Push operators** to a stack and **add operands** to the output.
3. **Handle parentheses**: Push (, pop to the output until) is found.
4. **Pop remaining operators** to the output at the end.

Program:

```
// Also always follow alphabetical order
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Stack
{
   int top;
   int capacity;
   char *array;
} Stack;

Stack *createStack(int n);
int isEmpty(Stack *s);
int isFull(Stack *s);
char pop(Stack *s);
void push(Stack *s, int value);
char peek(Stack *s);
void removeSpaces(const char *input, char *output);
void infixToPostfix(char *infix, char *postfix);
int precedence(char ch);
int isOperator(char ch);

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

```
    printf("Enter Your Operation String: ");
    fgets(inputstr, sizeof(inputstr), stdin);
    removeSpaces(inputstr, infix);
    size_t len = strlen(infix);
    if (len > 0 && infix[len - 1] == '\n') {
        infix[len - 1] = '\0';
    }
    infixToPostfix(infix, postfix);

    printf("Your Postfix Expression:-\n%s", postfix);

    return 0;
}

Stack *createStack(int n)
{
    Stack *s = (Stack *)malloc(sizeof(Stack));
    s->capacity = n;
    s->top = -1;
    s->array = (char *)malloc(s->capacity * sizeof(int));
    return s;
}

char pop(Stack *s)
{
    if (isEmpty(s))
    {
        // Underflow
        printf("Stack is Empty\n");
        return '\0';
    }
    char popped = s->array[s->top];
    s->top--;
    return popped;
}

void push(Stack *s, int value)
{
    // Overflow (as we do in algo in class)
    if (isFull(s))
    {
        printf("Stack is Full!\n");
        return;
    }

    s->top++;
    s->array[s->top] = value;
    return;
}

int isFull(Stack *s)
```

```
{
   return s->top == s->capacity - 1;
}

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

void removeSpaces(const char *input, char *output)
{
   // Index for output string
   int j = 0;
   for (int i = 0; input[i] != '\0'; i++)
   {
      if (input[i] != ' ')
      {
         output[j++] = input[i];
      }
   }
   // Null terminating o/p string
   output[j] = '\0';
}

void infixToPostfix(char *infix, char *postfix)
{
   int strlength = strlen(infix);
   // printf("%d\n", strlength);
   // printf("%s", infix);
   Stack *mystack = createStack(strlength);

   int i, j = 0;

   push(mystack, '(');
   // Double quotes here in strcat are necessary becasue function takes string type argument only
   strcat(infix, ")");

   for (i = 0; infix[i] != '\0'; i++)
   {
      char ch = infix[i];

      if (isalnum(ch))
      {
         postfix[j++] = ch;
      }

      else if (ch == '(')
      {
         push(mystack, ch);
      }
```

```
        else if (isOperator(ch))
        {
            while (isOperator(peek(mystack)) && precedence(peek(mystack)) >= precedence(ch))
            {
                postfix[j++] = pop(mystack);
            }
            push(mystack, ch);
        }

        else if (ch == ')')
        {
            while (peek(mystack) != '(')
            {
                postfix[j++] = pop(mystack);
            }
            // Then pop '('
            pop(mystack);
        }
    }

    postfix[j] = '\0';

    free(mystack->array);
    free(mystack);
}

char peek(Stack *s)
{
    if (isEmpty(s))
    {
        return '\0';
    }
    return s->array[s->top];
}

int precedence(char ch)
{
    if (ch == '+' || ch == '-')
    {
        return 1;
    }

    if (ch == '*' || ch == '/')
    {
        return 2;
    }

    if (ch == '^')
    {
        return 3;
    }
```

```
    return 0;
}

int isOperator(char ch)
{
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab5> ./postfix
Enter Your Operation String:  A + ( B * C - ( D / E ^ F ) * G ) * H
Your Postfix Expression:-
ABC*DEF^/G*-H*+
PS B:\sem3\ds\23bcp153_dsa\lab5> ./postfix
Enter Your Operation String: A + B * C
Your Postfix Expression:-
ABC*+
PS B:\sem3\ds\23bcp153_dsa\lab5> ./postfix
Enter Your Operation String: (A + B) * (C + D)
Your Postfix Expression:-
AB+CD+*
PS B:\sem3\ds\23bcp153_dsa\lab5> ./postfix
Enter Your Operation String: A * (B + C) / D
Your Postfix Expression:-
ABC+*D/
PS B:\sem3\ds\23bcp153_dsa\lab5> ./postfix
Enter Your Operation String: A + B * C - D / E
Your Postfix Expression:-
ABC*+DE/-
PS B:\sem3\ds\23bcp153_dsa\lab5> ./postfix
Enter Your Operation String: A * (B + C * D) - E
Your Postfix Expression:-
ABCD*+*E-
PS B:\sem3\ds\23bcp153_dsa\lab5> ./postfix
Enter Your Operation String: (A + B) * C - D / (E + F)
Your Postfix Expression:-
AB+C*DEF+/-
PS B:\sem3\ds\23bcp153_dsa\lab5> ./postfix
Enter Your Operation String: A ^ B + C * D - E
Your Postfix Expression:-
AB^CD*+E-
```

Theory (Balanced Brackets):

Check if all brackets (()), {}, []) are correctly paired and nested. Steps:

1. **Push opening brackets** onto a stack.
2. **Pop the stack** when encountering a closing bracket; ensure it matches the most recent opening bracket.
3. If the stack is empty at the end, the brackets are **balanced**.

Program:

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

typedef struct Stack
{
   int top;
   int capacity;
   char *array;
} Stack;

Stack *createStack(int n);
int isEmpty(Stack *s);
void push(Stack *s, char value);
char pop(Stack *s);
char peek(Stack *s);
int isMatchingPair(char left, char right);
int isBalanced(char *exp);

int main(void)
{
   char exp[100];

   printf("Enter expression: ");
   fgets(exp, sizeof(exp), stdin);

   size_t len = strlen(exp);
   if (len > 0 && exp[len - 1] == '\n')
   {
      exp[len - 1] = '\0';
   }

   if (isBalanced(exp))
      printf("Balanced\n");
   else
      printf("Not Balanced\n");

   return 0;
}

Stack *createStack(int n)
```

```
{
   Stack *s = (Stack *)malloc(sizeof(Stack));
   s->capacity = n;
   s->top = -1;
   s->array = (char *)malloc(s->capacity * sizeof(char));
   return s;
}

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

void push(Stack *s, char value)
{
   s->array[++s->top] = value;
}

char pop(Stack *s)
{
   if (isEmpty(s))
   {
      printf("Stack Underflow\n");
      return '\0';
   }
   return s->array[s->top--];
}

char peek(Stack *s)
{
   if (isEmpty(s))
   {
      return '\0';
   }
   return s->array[s->top];
}

int isMatchingPair(char left, char right)
{
   return (left == '(' && right == ')') ||
         (left == '{' && right == '}') ||
         (left == '[' && right == ']');
}

int isBalanced(char *exp)
{
   int n = strlen(exp);
   Stack *stack = createStack(n);

   for (int i = 0; exp[i] != '\0'; i++)
   {
```

```
        char ch = exp[i];

        if (ch == '(' || ch == '{' || ch == '[')
        {
            push(stack, ch);
        }
        else if (ch == ')' || ch == '}' || ch == ']')
        {
            if (isEmpty(stack) || !isMatchingPair(pop(stack), ch))
            {
                free(stack->array);
                free(stack);
                return 0; // Not Balanced
            }
        }
    }

    int balanced = isEmpty(stack);
    free(stack->array);
    free(stack);
    return balanced;
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab6> ./brackets
Enter expression: [( )]{ }{[( )( )]( )}
Balanced
PS B:\sem3\ds\23bcp153_dsa\lab6> ./brackets
Enter expression: [( ])
Not Balanced
```