Assignment

**Question 1:**

Implementing binary tree traversals and displaying in tree form.

**Source\_Code:**

#include<stdio.h>

#include<stdlib.h>

#define spaceVal 10

struct Node

{

int key;

struct Node \*l\_child, \*r\_child;

};

//Creating the new node in tree

struct Node \*newNode(int value)

{

struct Node \*node = (struct Node \*)malloc(sizeof(struct Node \*));

node->key = value;

node->l\_child = NULL;

node->r\_child = NULL;

return node;

}

//insert the node

struct Node\* insert\_node(struct Node\* node, int value)

{

if (node == NULL)

{

return newNode(value);

}

if (value < node->key)

{

node->l\_child = insert\_node(node->l\_child, value);

}

else if (value > node->key)

{

node->r\_child = insert\_node(node->r\_child, value);

}

return node;

}

//Recursive post order traversal

void Postorder\_Traverse(struct Node\* root)

{

if (root == NULL)

{

return;

}

Postorder\_Traverse(root->l\_child);

Postorder\_Traverse(root->r\_child);

printf("%d ", root->key);

}

// Recursive inorder traversal

void Inorder\_Traverse(struct Node\* root)

{

if (root == NULL)

{

return;

}

Inorder\_Traverse(root->l\_child);

printf("%d ", root->key);

Inorder\_Traverse(root->r\_child);

}

// Recursive preorder traversal

void Preorder\_Traverse(struct Node\* root)

{

if (root == NULL)

{

return;

}

printf("%d ", root->key);

Preorder\_Traverse(root->l\_child);

Preorder\_Traverse(root->r\_child);

}

//Display in tree form

void displayNode(struct Node \*node, int space)

{

if (node == NULL)

return;

space += spaceVal;

displayNode(node->r\_child, space);

printf("\n");

for (int i = spaceVal; i < space; i++)

printf(" ");

printf("%d\n", node->key);

displayNode(node->l\_child , space);

}

void displayTree(struct Node \*node)

{

displayNode(node, 0);

}

//Finds the depth of tree

int treeHeight(struct Node\* node)

{

if (node == NULL)

return 0;

else

{

/\* compute the depth of each subtree \*/

int l\_height = treeHeight(node->l\_child);

int r\_height = treeHeight(node->r\_child);

/\* use the larger one \*/

if (l\_height > r\_height)

return (l\_height + 1);

else

return (r\_height + 1);

}

}

int main()

{

struct Node \*root\_node = NULL;

int no\_of\_nodes,a[30];

printf("Enter number of Nodes: ");

scanf("%d",&no\_of\_nodes);

printf("Enter elements ");

for(int i=0;i<no\_of\_nodes;i++)

{

scanf("\n %d",&a[i]);

}

//Taking first node as root node

root\_node = insert\_node(root\_node, a[0]);

//Inserting the remaining nodes

for(int i=1;i<no\_of\_nodes;i++)

{

insert\_node(root\_node, a[i]);

}

//printf("\n Height of tree is %d", treeHeight(root\_node));

printf("\n Inorder\_Traverse is: ");

Inorder\_Traverse(root\_node);

printf("\n Preorder\_Traverse is : ");

Preorder\_Traverse(root\_node);

printf("\n Postorder\_Traverse is : ");

Postorder\_Traverse(root\_node);

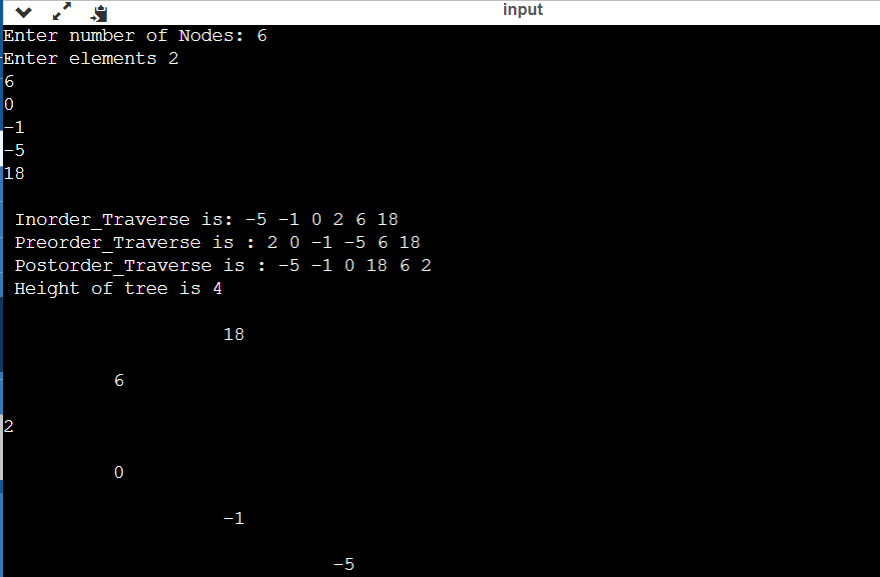
printf("\n Height of tree is %d \n", treeHeight(root\_node));

displayTree(root\_node);

return 0;

}

**Output:**



**Question 2:**

Validate the infix expression and convert them to prefix, postfix expression.

**Source\_Code**:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

char stack[MAX];

int top = -1;

int stack1[MAX];

int top1 = -1;

//Checking for operator

int isOperator (char symbol)

{

if (symbol == '+' || symbol == '-' || symbol == '\*' || symbol == '/'

|| symbol == '^')

{

return 1;

}

return 0;

}

//Finding the priority of operators

int precedenceValue (char symbol)

{

if (symbol == '+' || symbol == '-')

{

return 1;

}

else if (symbol == '\*' || symbol == '/')

{

return 2;

}

else if (symbol == '^')

{

return 3;

}

else

{

return 0;

}

}

//push the operators to stack

void push (char value)

{

if (top >= MAX - 1)

{

printf ("\nStack Overflow");

}

else

{

top++;

stack[top] = value;

}

}

//poping from the stack

char pop ()

{

char value;

if (top < 0)

{

printf ("\nStack Underflow");

getchar ();

exit (1);

}

else

{

value = stack[top];

top = top - 1;

return (value);

}

}

//convert infix to postfix

void infixToPostfix (char infix[], char postfix[])

{

push ('('); /\* push '(' onto stack \*/

strcat (infix, ")");

//char convert[MAX];

int index = 0;

int postfixIndex = 0;

//int cIndex= 0;

char value = infix[index];

char holder;

while (value != '\0')

{

if (value == '(')

{

push (value);

}

else if (isOperator (value) == 1)

{

holder = pop (); //element on top of stack, this is used to compare with parsing element

while (isOperator(holder) == 1 && precedenceValue(holder) >= precedenceValue(value))

{

postfix[postfixIndex] = holder;

postfixIndex++;

holder = pop ();

}

push(holder);

push(value);

postfix[postfixIndex] = ' ';

postfixIndex++;

}

else if (value == ')')

{

holder = pop ();

while (holder != '(')

{

postfix[postfixIndex] = holder;

postfixIndex++;

holder = pop ();

}

}

else if (!isdigit (value))

{

postfix[postfixIndex] = value;

postfixIndex++;

}

else

{

printf ("\n Invalid infix Expression"); /\* When it is illegal symbol \*/

getchar ();

exit (1);

}

index++;

value = infix[index];

}

postfix[postfixIndex] = '\0';

}

//Validating the infix expression

int validate(char expr[])

{

int n = strlen(expr);

//return if you find operator at start or end of expression

if(isOperator(expr[0]) || isOperator(expr[n-1]) )

{

return 0;

}

//return if you find two consecutive operator in the expression

for(int i=0; i<n-1; i++)

{

if(isOperator(expr[i]) && isOperator(expr[i+1]) )

{

return 0;

}

}

return 1;

}

//Main program

int main ()

{

char infix[MAX];

char postfix[MAX];

int length, i = 0, j = 0;

char temp;

length = sizeof (postfix) / sizeof (int);

printf("\nEnter the expression \n");

gets(infix);

//Checking whether given infix is valid or not

if(validate(infix))

{

printf("\nInfix expression is Valid");

infixToPostfix (infix, postfix);

infix[strlen(infix)-1] = '\0';

printf("\nInfix expression is ");

puts(infix);

printf ("\nPostfix Expression: ");

puts (postfix);

//Reversing the infix expression

j = strlen(infix) - 1;

while(i<j)

{

temp = infix[i];

infix[i] = infix[j] ;

infix[j] = temp;

i++;

j--;

}

//Converting the reverse of infix to postfix

infixToPostfix(infix,postfix);

//Reversing the converted postfix expression

int k = 0,l = 0;

l = strlen(postfix) - 1;

while(k<l)

{

temp = postfix[k];

postfix[k] = postfix[l] ;

postfix[l] = temp;

k++;

l--;

}

printf("Prefix expression is: ");

puts(postfix);

}

else

{

printf("\nInfix expression is Invalid");

}

return 0;

}

**Output**:

