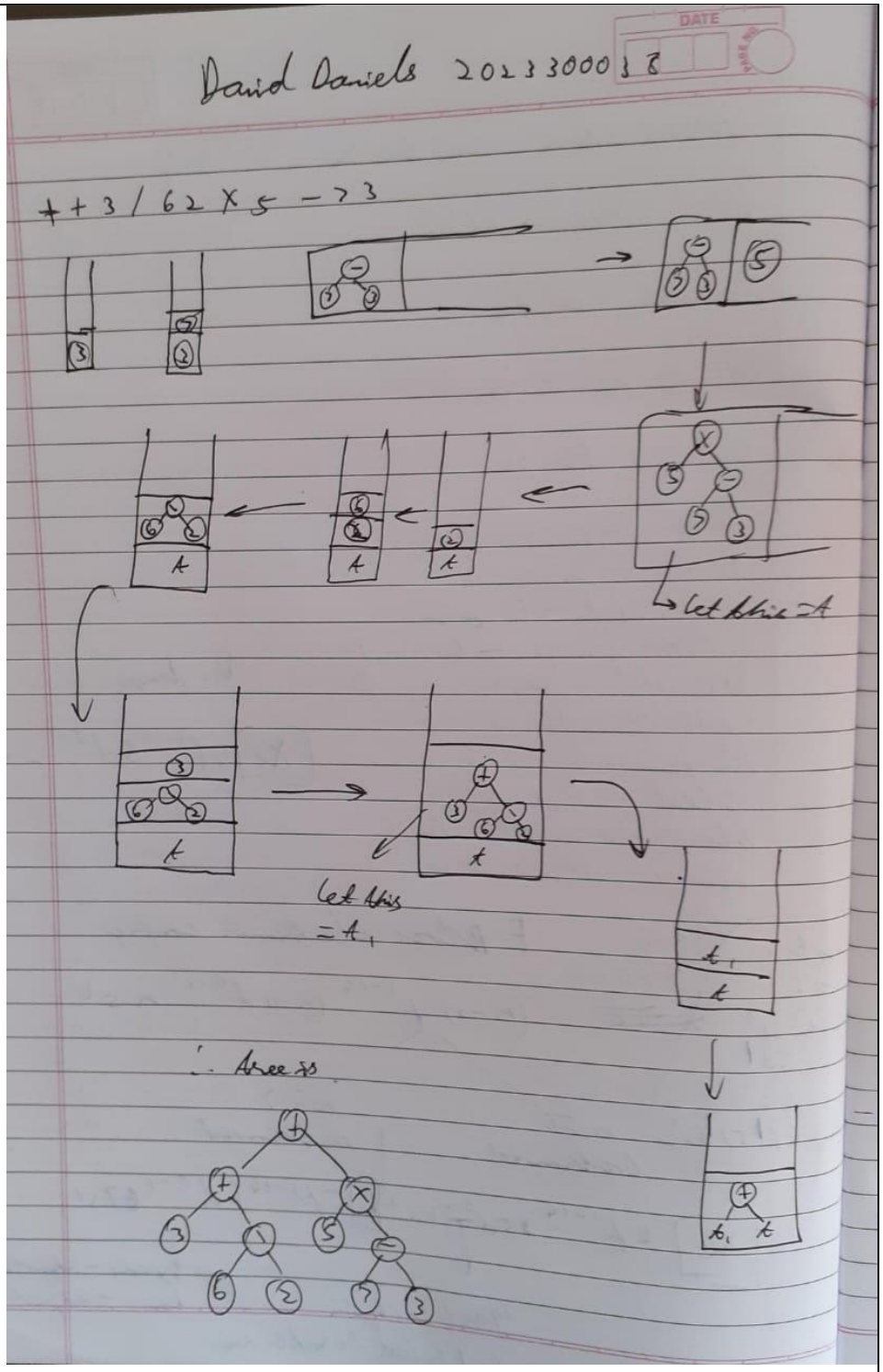
Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
COMPS Department

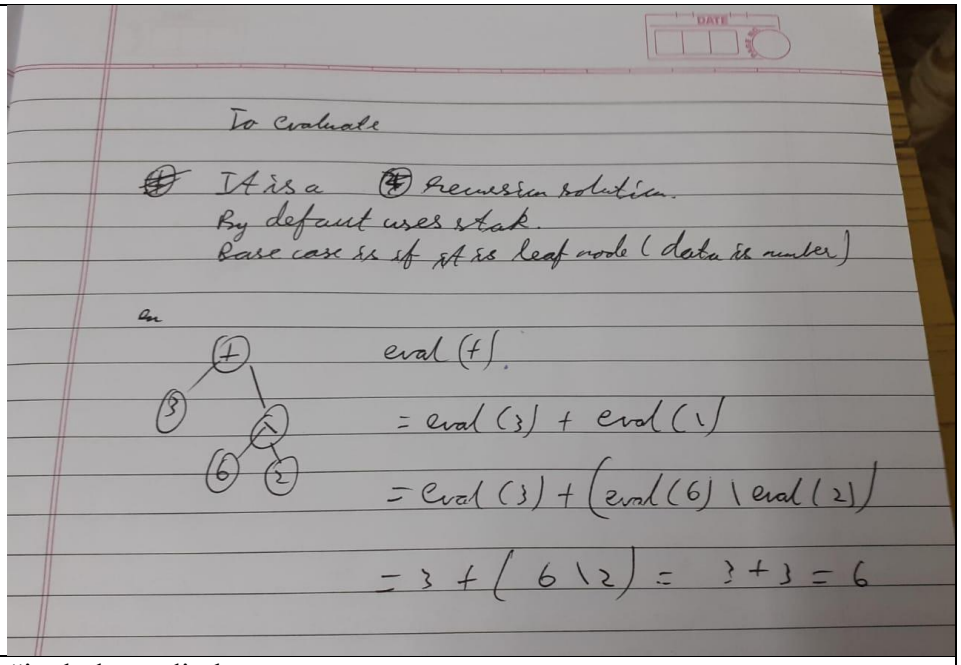
Experiment	6
Aim	**Create an expression tree from a given preorder expression traversal and perform the evaluation.**
Name	David Daniels
UID	2023300038
Class	Div -A
Batch	C
Date of Submission	7-10-24

Theory	A data structure called an expression tree is used to describe expressions that take the shape of trees. After creating an expression's expression tree, we may execute inorder traversal to produce an infix expression. Postfix expressions can also be produced by traversing the expression tree in postorder. It is a special type of Binary Tree. A mathematical expression can be expressed as a binary tree using expression trees. Expression trees are binary trees with each leaf node serving as an operand and each internal (non-leaf) node serving as an operator. **Properties of Expression Tree in Data Structure** • The operands are always represented by the leaf nodes. These operands are always used. • The operators are placed in a node. This node must always have two children. The children may be another operator or operands • The priority of operators can be seen by the height of the node. • The expression tree can be traversed to evaluate prefix expressions, postfix expressions, and infix expressions. • The main use of these expression trees is that it is used to **evaluate, analyze** and **modify** the various expressions. • It is also used to find out the associativity of each operator in the expression. In summary, the value present at the depth of the tree has the highest priority when compared to the other operators located at the top of the tree. The expression tree is immutable, and once built, we cannot change or modify it further, so to make any changes, we must completely construct the new expression tree. **Implementation of an Expression tree**

	To implement the expression tree and write its program, we will be required to use a stack data structure. We can create an expression tree using prefix or postfix expression. For constructing an expression tree we use a stack. We loop through input expression and do the following for every character. 1. If a character is an operand push that into the stack 2. If a character is an operator pop two values from the stack make them its child and push the current node again. 3. In the end, the only element of the stack will be the root of an expression tree. **Use of Expression tree** 1. The main objective of using the expression trees is to make complex expressions and can be easily be evaluated using these expression trees. 2. It is also used to find out the associativity of each operator in the expression. 3. It is also used to solve the postfix, prefix, and infix expression evaluation.
Algorithm	# 1. Create Expression Tree from Prefix String **Processing the Input**: • Loop through the input string in reverse (from last character to first) as it is prefix expression o If the character is not an operator (i.e., it's an operand): ▪ Create a new node for the operand and push it onto the stack. o If the character is an operator: ▪ Creates a new node for the operator. ▪ Pops the top two nodes from the stack (these become the left and right children of the operator node). ▪ Sets the left child of the operator node to the first popped node, and the right child to the second popped node. ▪ Pushes the newly created operator node back onto the stack. **Final Node Extraction**: • After processing all characters, pops the top node from the stack, which represents the root of the expression tree. **Stack Validation**: • Pops another node to check if the stack is empty:

	○ If the stack is empty, the function returns the root node of the expression tree. ○ If the stack is not empty, it indicates an error in the input expression and returns NULL. **2. Evaluate** **1) Base Case**: • The first condition checks if the current node is a leaf node (i.e., it has no left or right children) • If it is a leaf, it means this node represents a number. The code converts the character to its integer value. **2) Recursive Evaluation**: • If the node is not a leaf, it contains an operator. The function uses a switch statement to determine which operator is stored in root->data. • For each operator ('/', '*', '-', '+'), the function recursively evaluates the left and right subtrees and applies the corresponding operation

Problem Solving	



Program(Code)

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

typedef struct node
{
   char data;
   struct node * left;
   struct node * right;
}node;

void inorder (node * root)
{
   if (root!=NULL)
   {
      inorder(root->left);
      printf("%c ",root->data);
      inorder(root->right);
   }

}

node * create_node(char d)
{
   struct node * ptr = (struct node * ) malloc(sizeof(struct node));
   ptr->left=NULL;
   ptr->right=NULL;
   ptr->data=d;
   return ptr;
}
```

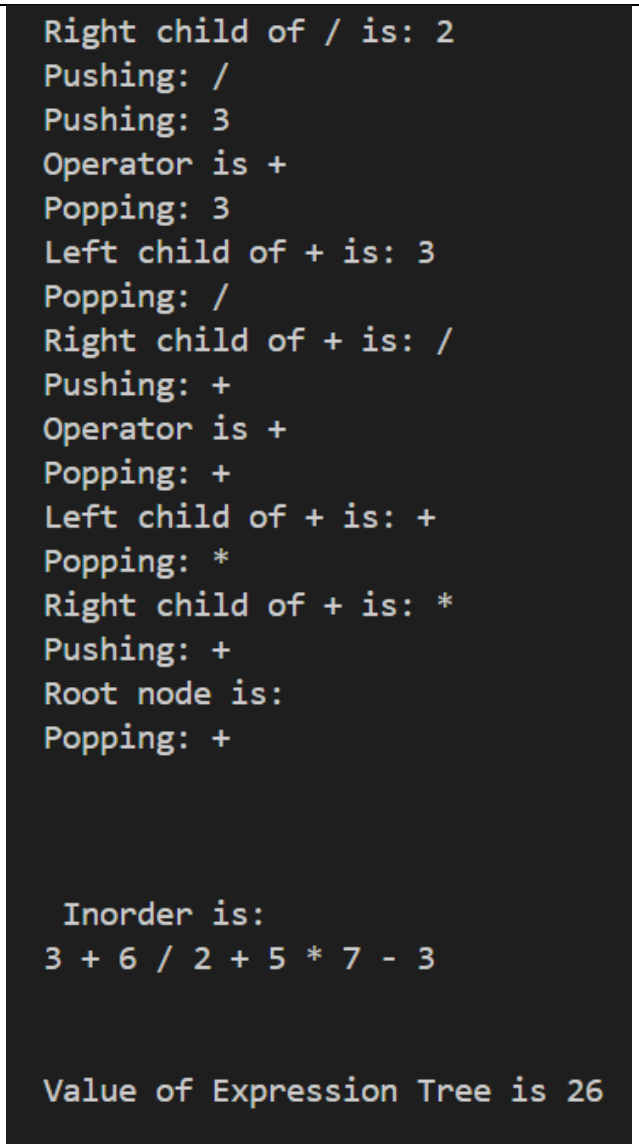
```
typedef struct stack
{
   int top,size;
   struct node * * arr;
}stack;

bool is_empty_st(stack * s)
{
   if (s->top==-1)
   {
      return true;
   }

   return false;
}

bool is_full (stack *s)
{
   if (s->top ==(s->size-1))
   {
      return true;
   }

   return false;
}

void push(stack * s1,node * data)
{
   if (is_full(s1))
   {
      printf("FAIL");
      return;
   }
   s1->top++;
   s1->arr[s1->top]=data;
   printf("Pushing: %c \n", data->data);
}

node * pop (stack * s)
{
   if (is_empty_st(s))
   {
      return NULL;
   }
   node * temp = s->arr[s->top--];
   printf("Popping: %c \n", temp->data);
   return temp;
}

bool is_operator (char op)
```

```
{
   if (op=='+' ||op=='-' ||op=='*' ||op=='/')
   {
      return true;
   }
   else
   {
      return false;
   }
}

node * prefix_to_xt( char * input )
{
   int str_len = strlen(input);

   stack * s= (stack *)malloc(sizeof(stack));
   s->top=-1;
   s->size=str_len;
   s->arr = (node * *)  malloc(str_len*sizeof(node *));

   for (int i = str_len-1; i >= 0; i--)
   {
      if (!is_operator(input[i]))
      {
         node * temp = create_node(input[i]);
         push(s,temp);
      }
      else
      {
         node * op = create_node(input[i]);
         printf("Operator is %c \n",op->data);
         node * n1 = pop(s);
         op->left=n1;
         printf("Left child of %c is: %c \n",op->data ,op->left->data);
         node * n2 = pop(s);
         op->right=n2;
         printf("Right child of %c is: %c \n",op->data, op->right->data);
         push(s,op);
      }
   }

   printf("Root node is: \n");
   node * output =pop(s);
   pop(s);
   if (is_empty_st(s))
   {
      return output;
   }
   printf("Input is wrong");
   return NULL;
```

```
}

int evaluate(node * root)
{
   if (root->left==NULL && root->right==NULL)
   {
      return( (int) root->data -'0');
   }

   switch (root->data)
   {
   case'/':
         {
         return evaluate(root->left) / evaluate(root->right);
         break;
         }
   case'*':
         {
         return evaluate(root->left) * evaluate(root->right);
         break;
         }
   case'-':
         {
         return evaluate(root->left) - evaluate(root->right);
         break;
         }
   case'+':
         {
         return evaluate(root->left) + evaluate(root->right);
         break;
         }
   default:
         {
         printf("Wrong operator or tree \n");
         break;
         }
   }
}

int main(int argc, char const *argv[])
{
   char * input;
   int size;


   printf("Enter Size of String: \n");
   scanf("%d",&size);
   printf("Enter Preorder: \n");
   getchar();
   input=(char *) malloc(size*sizeof(char));
```

```
    scanf("%[^\n]%*c", input);
    input[size+1]='\0';

    //input="-+*12*/3523";
    //len=11
    node * root = prefix_to_xt(input);
    printf("\n\n\n Inorder is: \n");
    inorder(root);
    printf("\n\n\n");
    int ans =evaluate(root);
    printf("Value of Expression Tree is %d \n",ans);
    printf("\n\n\n");

    return 0;
}
```

Output	Enter Size of String: 11 Enter Preorder: ++3/62*5-73 Pushing: 3 Pushing: 7 Operator is - Popping: 7 Left child of - is: 7 Popping: 3 Right child of - is: 3 Pushing: - Pushing: 5 Operator is * Popping: 5 Left child of * is: 5 Popping: - Right child of * is: - Pushing: * Pushing: 2 Pushing: 6 Operator is / Popping: 6 Left child of / is: 6 Popping: 2	

	
Conclusion	Thus we have learnt how to convert a prefix expression into an expression tree using implementation of a systematic approach to parse and construct expression trees from prefix notation. By utilizing a stack-based method, we were able to effectively traverse the prefix expression, identifying operators and operands to build the tree structure. The resulting expression tree facilitates efficient computation.