NAME: ADWAIT S PURAO
UID: 2021300101
EXP NO. :6
AIM: To convert an infix expression to prefix and the convert that prefix expression to expression tree and show the inorder traversal of the tree

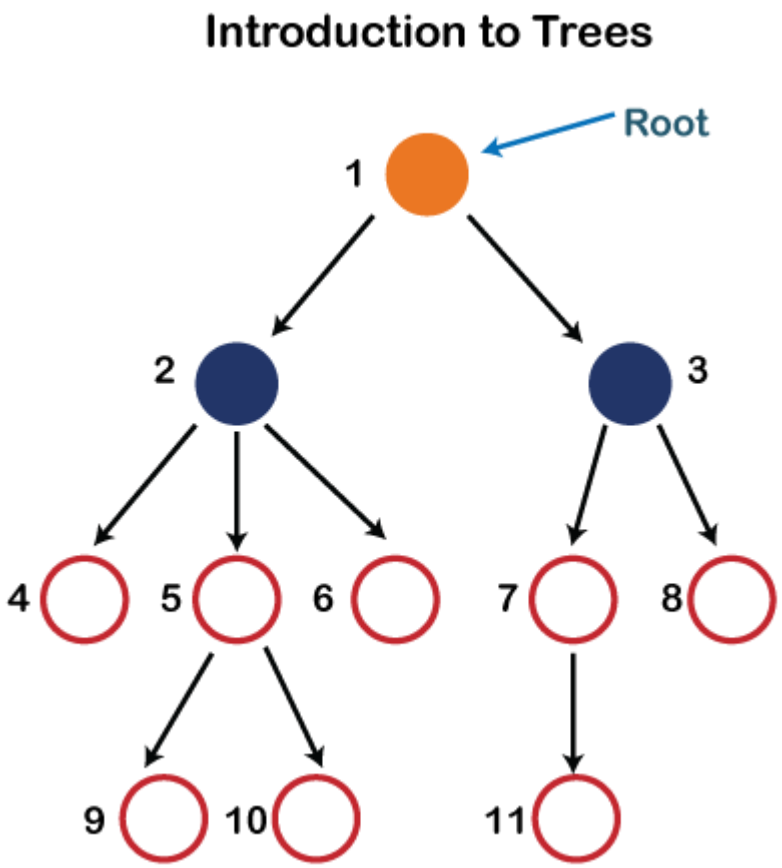
THEORY:

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



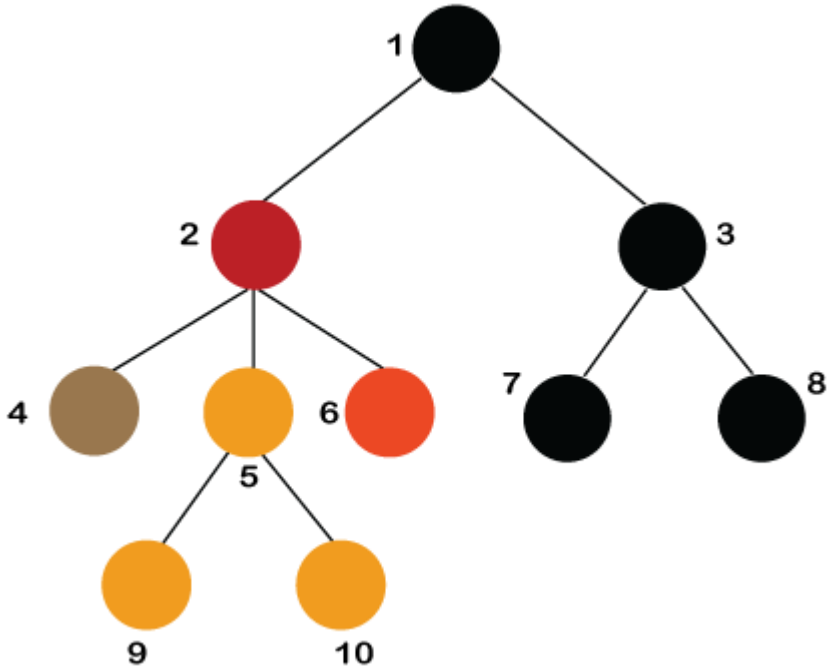
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***

- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.



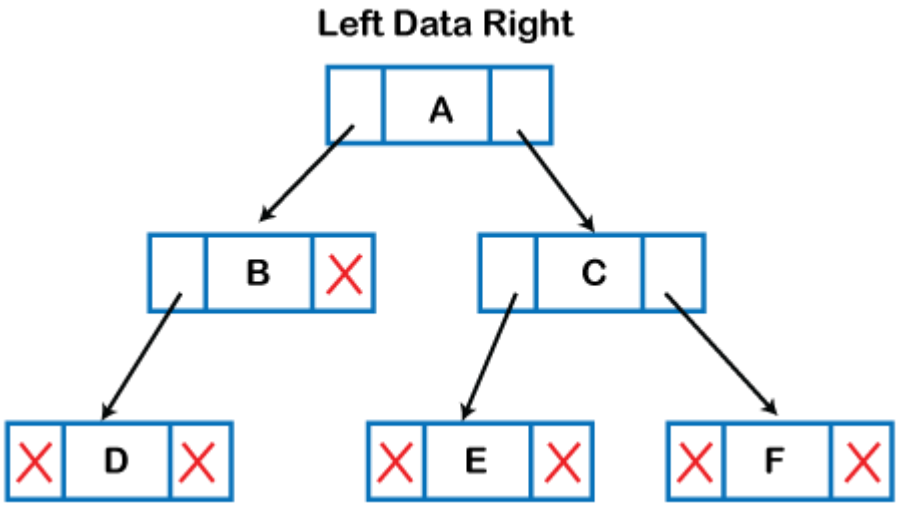
- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.

- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



Applications of trees

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.

- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

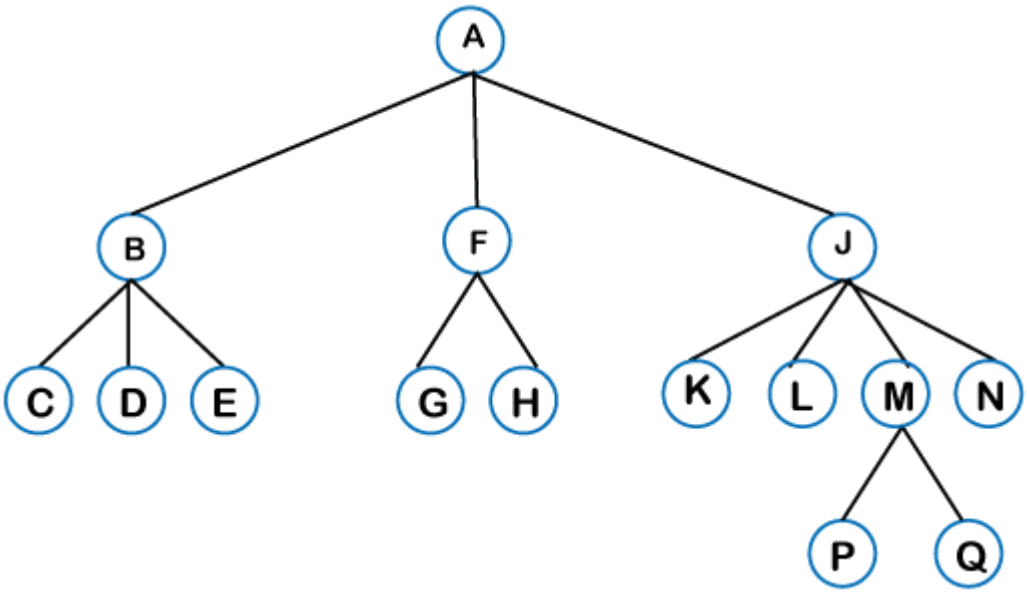
Types of Tree data structure

The following are the types of a tree data structure:

1. General tree:

A general tree is a type of tree data structure that has no constraints on the hierarchical structure. Every node can have an infinite number of children in a general tree.

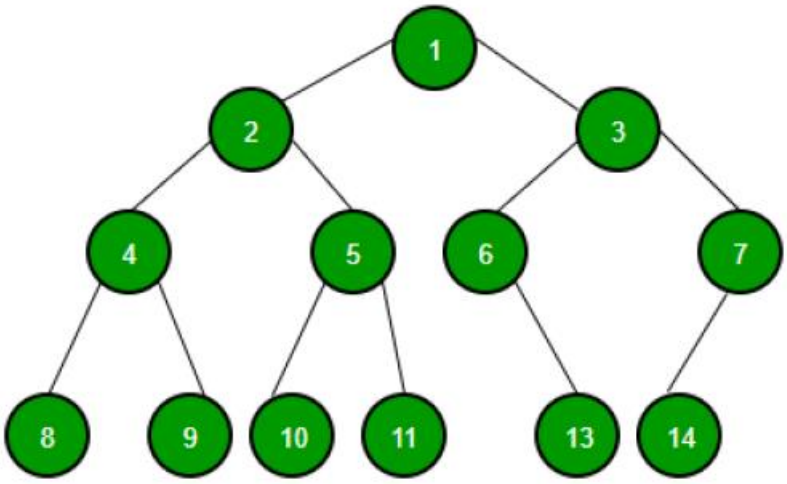
A general tree stores hierarchical structures, such as a folder structure in a computer system.



Binary tree:

Binary Tree is defined as a Tree data structure with at most 2 children.

These trees are highly functional and help serve many purposes in the data structure. Using a binary tree, data scientists and analysts can create a representation of data through a bifurcating structure and they can easily access nodes and label them as per convenience.

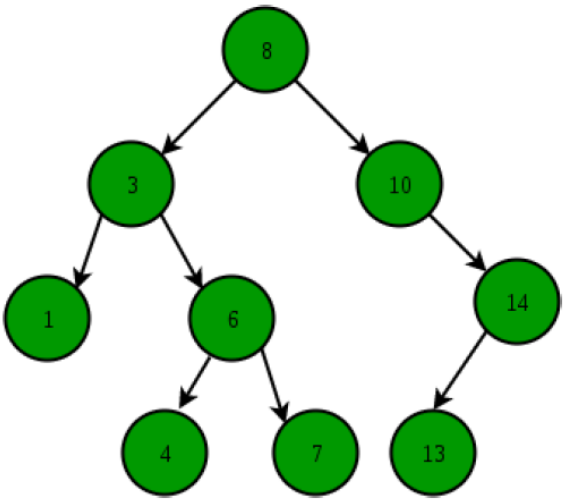


Binary Search Tree:

A binary search tree is an extension of a binary tree with several optional restrictions.

It is a binary tree in which the left subtree has all elements less than its root node and the right subtree has elements greater than root node. Each inserted value is compared to the parent node and then inserted into the tree.

BST finds usage in search applications where you constantly add and remove data.



4. AVL Tree: AVL - Adelson-Velsky and Landis, named after its founders.

An AVL tree is a self-balancing BST.

This tree can automatically balance its height because each node stores a value called “balance factor," representing the difference in the height of a right sub-tree and left sub-tree. The AVL nodes can have a balance factor of minus one, zero and one. It enjoys all properties of a BST.

AVL trees are used while performing lookup operations and situations where frequent data insertion is necessary. AVL finds its usage in memory management subsystems of the Linux kernel.

5. Red-Black Tree:

A red-black tree is a self-balancing BST. Each node is either red or black. the colour of these nodes ensures that the tree remains self-balanced every time someone inserts or adds a value.

Computation geometry uses this type of tree data structure.

6. B- Tree:

B-tree is another self-balancing search tree that comprises many nodes to keep data stored in a particular order. Each node has over two child nodes and each node comprises multiple keys.

Larger storage systems like disks use B-tree data structures

Binary Tree: A Binary tree is represented by a pointer to the topmost node of the tree. If the tree is empty, then the value of the root is NULL. Binary Tree node contains the following parts:

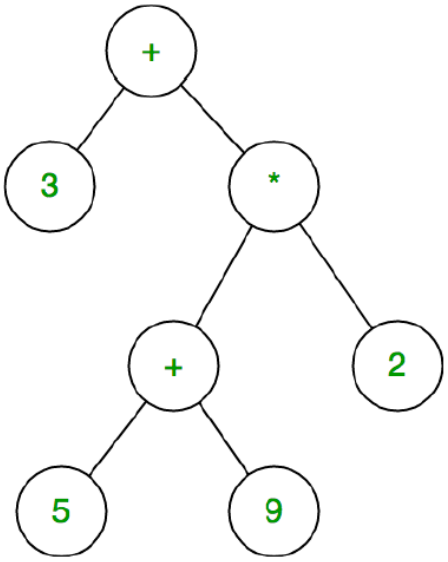
1. Data

2. Pointer to left child

3. Pointer to right child

Expression Tree:

It is a special application of Binary Tree. The expression tree is a tree used to represent the various expressions. The tree data structure is used to represent the expressional statements. In this tree, the internal node always denotes the operators. The operations are always performed on these operands. The value present at a depth of the tree is at the highest priority compared with the other operators present at the top of the tree.



An example of expression tree

Uses of expression tree:

1. The main use of these expression trees is that it is used to evaluate, analyze and modify the various expressions

2. It is also used to find out the associativity of each operator in the expression. For example, the + operator is the left-associative and / is the right-associative.

3. It is one of the major parts of compiler design and belongs to the semantic analysis phase. In this semantic analysis, we will use the syntax-directed translations, and in the form of output, we will produce the annotated parse tree

4. Expression trees play a very important role in representing the language-level code in the form of the data, which is mainly stored in the tree-like structure. It is also used in the memory representation of the lambda expression. Using the tree data structure, we can express the lambda expression more transparently and explicitly. It is first created to convert the code segment onto the data segment so that the expression can easily be evaluated. Implementation of expression tree:

1. For creating an expression tree we make use of a combination of the binary tree and stack data structure.

2. Basically, the node of a tree data structure is used as the data type for the stack array. So, the stack takes in and out nodes instead of characters or digits.

3. So, the operations such as pop and push will be similar to stack.

4. First, we convert the infix expression to prefix or postfix expression.

5. Here we will be converting it to prefix expression

6. After converting it, we will iterate it from right to left.

7. Whenever we encounter an operand, we create a node of it and push it into the stack.

8. When we encounter an operator, we pop two nodes from the stack. We create a new node of the operator and assign the pop nodes as left and right child of it. Then we push the new node into stack.

9. After the iterations end, we pop the last element of the stack and assign it as the root.

Operations used while creating expression tree:

1. Push: When we insert an element in a stack then the operation is known as a push. Before inserting we check whether the stack is full or not. If the stack is full then the overflow condition occurs. When we initialize a stack, we set the value of top as -1 to check that the stack is empty. When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top+=1, and the element will be placed at the new position of the top. The elements will be inserted until we reach the max size of the stack.

2. Pop: When we delete an element from the stack, the operation is known as a pop. Before removing we check whether the stack is empty or not. If the stack is empty means that no element exists in the stack, this state is known as an underflow state. If the stack is not empty, we first access the element which is pointed by the top Once the pop operation is performed, the top is decremented by 1, i.e., top-=1

ALGORITHM:

1)Struct stack

Data members

Int size

Int top

Char * arr

2)Struct sll

Data members

Char data

Struct sll * next

3)Struct tree

Char data

Struct tree * left

Struct tree * right

Struct tree * next (Next pointer to maintain the stack)

4)Header pointer of the tree node to maintain the stack

Struct tree * head

5)Function int isFull(struct stack * s)

if(s->top==s->size-1)

return 1

return -1

6)Function int isEmpty(struct stack * s)

if(s->top==-1)

return 1

return -1

7)Function void push(char data,struct stack * s)

if(isFull(s)==1)

print stack is overflowing

Increment s->top by 1

Store data in s->arr[s->top]

Print pushed element data

8)Function void pop(struct stack * s)

if(isEmpty(s)==1)

Print stack is underflowing

Return

Print Popped element s->arr[s->top]

Decrement s->top by 1

9)Function char peek(struct stack * s)

return s->arr[s->top]

10)Function int precedence(char c)

if(c=='^')

return 3

else if(c=='*' || c=='/')

return 2

else if(c=='+' || c=='-')

return 1

return -1

11)Function int Bracket_swap(struct sll*head,int n)

Set integer variable k to n

Set struct sll * ptr to head

Iterate a while loop until ptr is not equal to null

if(ptr->data==')')

k--;

ptr->data='('

ptr=ptr->next

else if(ptr->data=='(')

k--;

ptr->data=')'

ptr=ptr->next

else

ptr=ptr->next

return k

12)Function struct sll*InsertAtEnd(struct sll * head,char data)

Allocate memory for struct sll * ptr

Store data in ptr->data

Set ptr-> next to null

if(head==NULL)

head=ptr;

return head

else

Set struct sll * p to head

Iterate a while loop until next of p is not null

Store next of p in p

After completing the iterations

Set next of p as ptr

Return head

13) struct sll*InsertAtFirst(struct sll*head,char data)

Allocate memory for struct sll * ptr

Store data in ptr->data

Set next of ptr to head

Store ptr in head

Return head

14)Function void LinkedListTraversal(struct sll*head)

Set struct sll * ptr to head

Iterate a while loop until ptr is not equal to NULL

Print ptr->data

Store ptr->next in ptr

15)Function struct tree *tree_pop()

Store head in struct tree * ptr

Make head= head->next

Return ptr

16)Function void tree_push(struct tree *top)

if (head == NULL)

head = top

else

Make top->next = head;

head = top

17)Function struct tree *creation(char data)

Allocate memory for struct tree * new

Store data in new->data

Set new->left = null

Set new->right=null

Set new->next=null

Return new

18)Function void Inorder(struct tree *root)

if (root != NULL)

Inorder(root->left);

printf("%c", root->data);

Inorder(root->right)

19) Main function

Allocate memory for struct stack

Set s->top=-1

Set s->size =50

Allocate memory for s->arr

Set struct sll * head1=NULL

Declare integer variable i, n

Take input n

Declare character array inf

Take input the infix expression

Convert the infix expression to a linked list with the help of a loop

Call function Bracket swap to swap the brackets and Store the integer variable returned by the function in k

Call function head1=InsertAtFirst(head1,'(')

Call function head1=InsertAtEnd(head1,')')

Call function LinkedListTraversal(head1)

Declare character array final[n+2]

```
Set struct sll * ptr to head
Re-initialize I =0
Store the entire linked list in the final array with the help of a while loop
Print the final array
Declare a character array prefix[k]
Declare an integer variable c =-1
Run a for loop for i=0 to i<n+2
        if(isalpha(final[i]) || isdigit(final[i]))
                Increment c by 1
                prefix[c]=final[i]
        else if(final[i]=='(')
                Call function push(final[i],s)
        else if(final[i]==')')
                Print Popped element : )
                while(isEmpty(s)!=1 && peek(s)!='(')
                        Increment c by one
                        prefix[c]=peek(s)
                        pop(s)
        if(peek(s)!='(' && isEmpty(s)!=1)
                exit(1)
        else
                Call function pop(s)
Else
        while(isEmpty(s)!=1 && precedence(final[i])<=precedence(peek(s)))
                Increment c by one
                prefix[c]=peek(s)
                Call function pop(s)
        Call function push(final[i],s)
```

```
while(isEmpty(s)!=1)
        Increment c by one
        prefix[c]=peek(s)
        pop(s)
prefix[k]='\0'
Call function strrev(prefix) to reverse the string prefix
Print the resultant prefix expression i.e. the string prefix
struct tree *right_ins;
struct tree *left_ins;
struct tree *top
declare integer variable j
Run a for loop from j=k-1 uptill j>=0
        if (prefix[j] == '+' || prefix[j] == '-' || prefix[j] == '*' || prefix[j] == '/' || prefix[j] == '^')
                top = creation(prefix[j]);
                left_ins = tree_pop(head)
                right_ins = tree_pop(head)
                top->left = left_ins
                top->right = right_ins
                tree_push(top)
        else
                struct tree *ptr = creation(prefix[j])
                tree_push(ptr)
Call function Inorder traversal(top) to print the inorder traversal of the tree
```

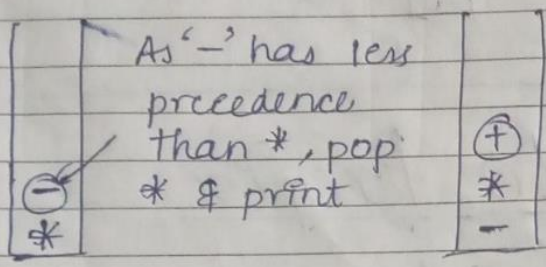
PROBLEM SOLVING ON CONCEPT:

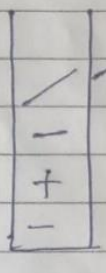
E.g. Adwait S Purao 2021300101

Infix: a/b-c+d*e-a*c

o/p: Prefix

- + - / abc * de * ac





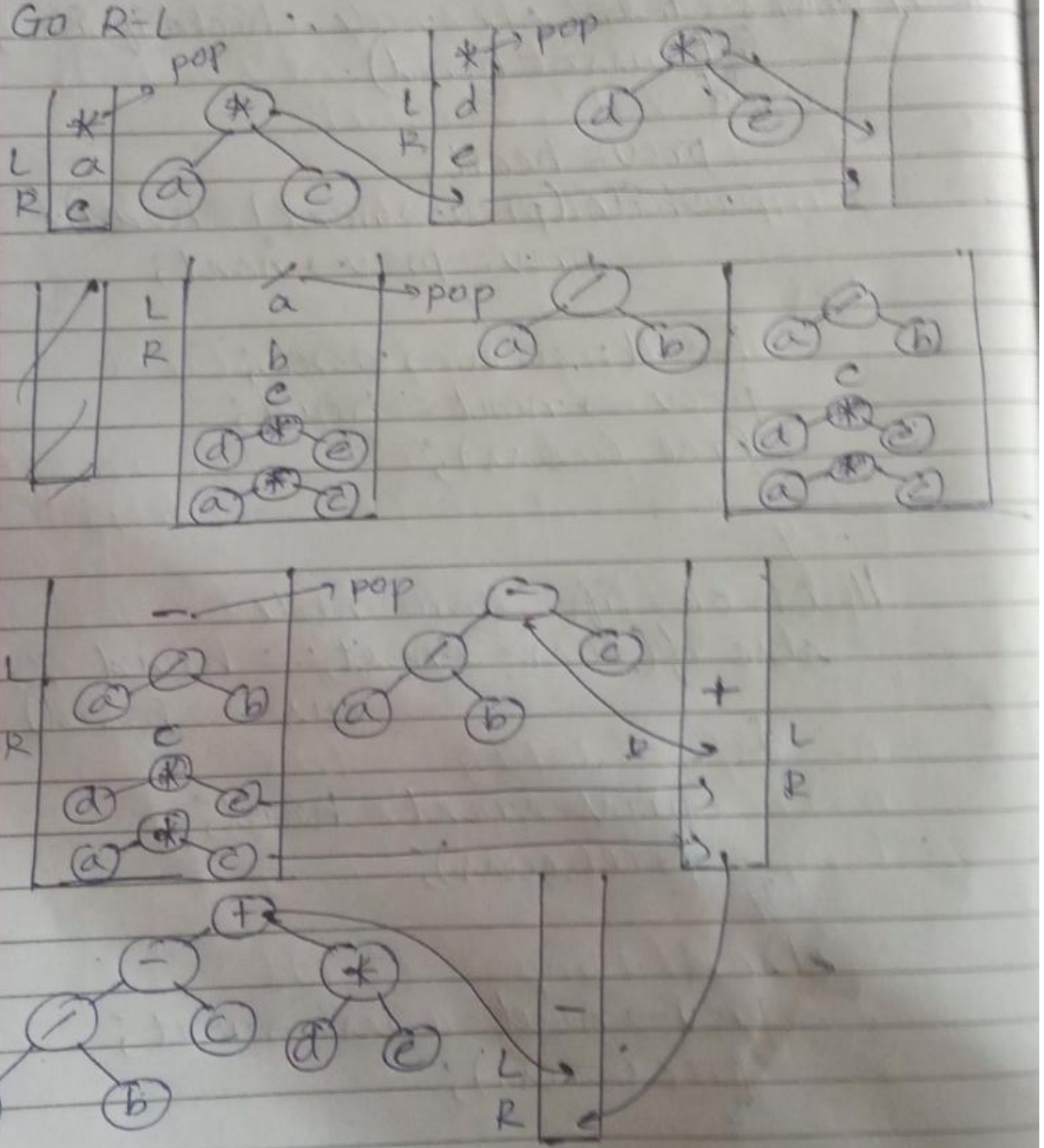
Adwait S. Purao B2
20213001O1

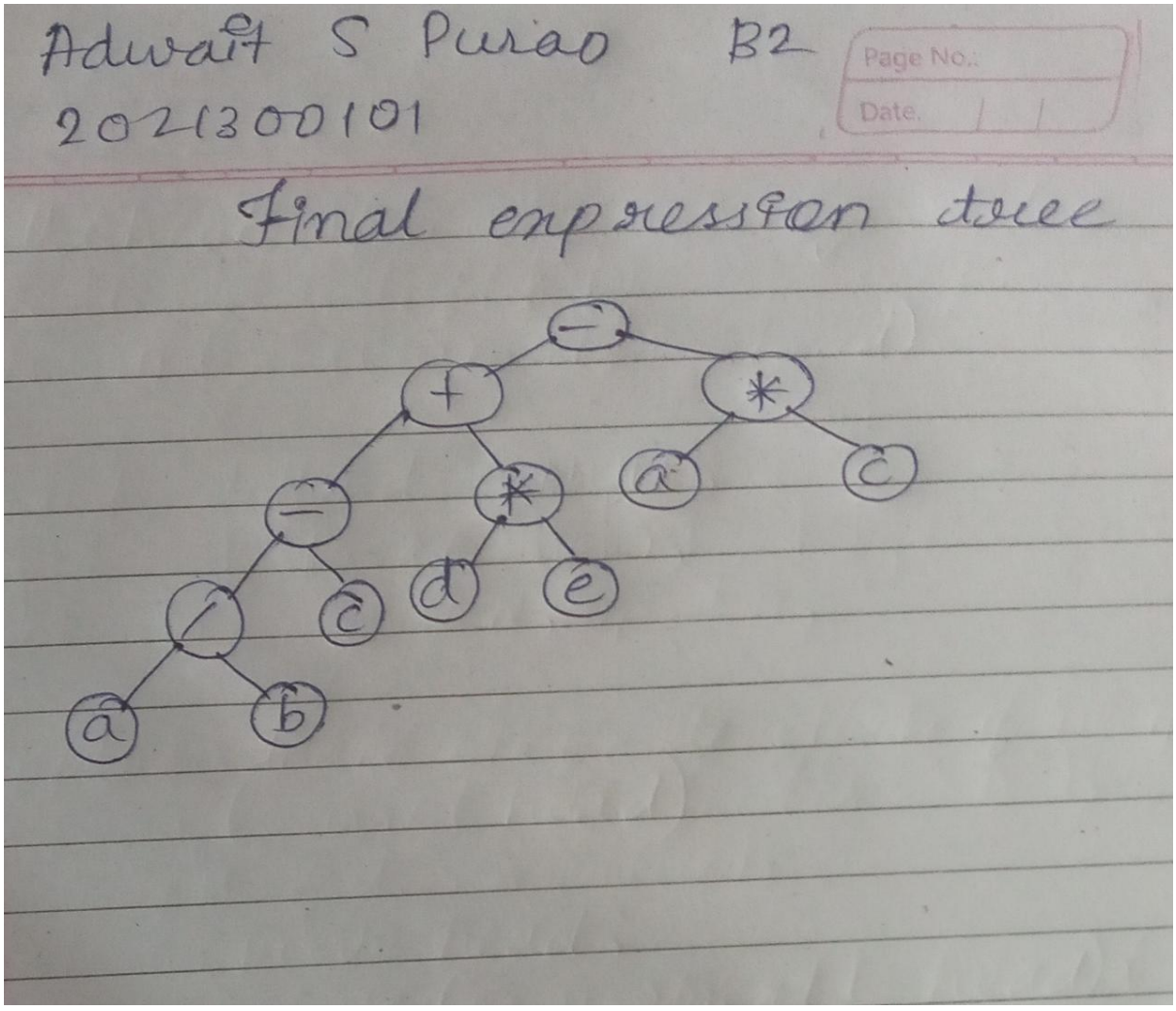
Expression tree from infix:

Prefix:

Prefix:
- + - / abc * de * ac

Go R-L





CODE:

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

// stack to convert infix to prefix
struct stack{
    int size;
    int top;
    char * arr;
};

//Linked list to take the string as input
struct sll{
    char data;
    struct sll * next;
};
```

```c
//Node for expression tree
struct tree
{
    char data;
    struct tree *left;
    struct tree *right;
    struct tree *next;
};

struct tree *head = NULL;

// Functions of stack
int isFull(struct stack * s){
    if(s->top==s->size-1){
        return 1;
    }
    return -1;
}

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

void push(char data,struct stack * s){
    if(isFull(s)==1){
        printf("Stack is Overflowing\n");
        return ;
    }
    s->top++;
    s->arr[s->top]=data;
    printf("Pushed element:%c\n",data);
}

void pop(struct stack * s){
    if(isEmpty(s)==1){
        printf("Stack is underflowing\n");
        return ;
    }
    printf("Popped element:%c\n",s->arr[s->top]);
    s->top--;
}

char peek(struct stack * s){
    return s->arr[s->top];
```

```c
}

//Function to get the precedence of operators while converting from infix to
prefix
int precedence(char c){
    if(c=='^'){
        return 3;
    }
    else if(c=='*' || c=='/'){
        return 2;
    }
    else if(c=='+' || c=='-'){
        return 1;
    }
    return -1;
}

//Function to swap ) with ( and vice-versa
int Bracket_swap(struct sll*head,int n){
    int k=n;
    struct sll * ptr=head;
    while(ptr!=NULL){
        if(ptr->data==')'){
            k--;
            ptr->data='(';
            ptr=ptr->next;
        }
        else if(ptr->data=='('){
            k--;
            ptr->data=')';
            ptr=ptr->next;
        }
        else{
            ptr=ptr->next;
        }
    }
    return k;
}

// Function to insert at end in linked list
struct sll*InsertAtEnd(struct sll * head,char data){
    struct sll*ptr=(struct sll*)malloc(sizeof(struct sll));

    ptr->data=data;
    ptr->next=NULL;

    if(head==NULL){
        head=ptr;
```

```
        return head;
    }

    else{
        struct sll*p=head;
        while(p->next!=NULL){
            p=p->next;
        }

        p->next=ptr;
        return head;
    }
}

//Function to insert at first
struct sll*InsertAtFirst(struct sll*head,char data){
    struct sll*ptr=(struct sll*)malloc(sizeof(struct sll));
    ptr->data=data;
    ptr->next=head;
    head=ptr;
    return head;
}

//Function to traverse a linked list
void LinkedListTraversal(struct sll*head){
    struct sll*ptr=head;
    printf("Traversal of entire linked list\n");
    while(ptr!=NULL){
        printf("%c ",ptr->data);
        ptr=ptr->next;
    }

}

//Functions of stack to convert tree to expression tree

//Function for popping a node in a stack
struct tree *tree_pop()
{
    struct tree *ptr = head;
    head = head->next;
    return ptr;
}

//Function for pushing a node in a stack
void tree_push(struct tree *top)
{
    if (head == NULL)
```

```
    {
        head = top;
    }
    else
    {
        top->next = head;
        head = top;
    }
}

//Function to create node
struct tree *creation(char data)
{
    struct tree *new = (struct tree *)malloc(sizeof(struct tree));
    new->data = data;
    new->left = NULL;
    new->right = NULL;
    new->next = NULL;
    return new;
}

//Function for inorder traversal
void Inorder(struct tree *root)
{
    if (root != NULL)
    {
        Inorder(root->left);
        printf("%c", root->data);
        Inorder(root->right);
    }
}

int main()
{
    struct stack * s=(struct stack *)malloc(sizeof(struct stack));
    s->top=-1;
    s->size=50;
    s->arr=(char *)malloc(s->size * sizeof(char));
    int n;

    struct sll * head1=NULL;
    int i;


    printf("Enter the length of the string:\n");
    scanf("%d",&n);

    char inf[n];
```

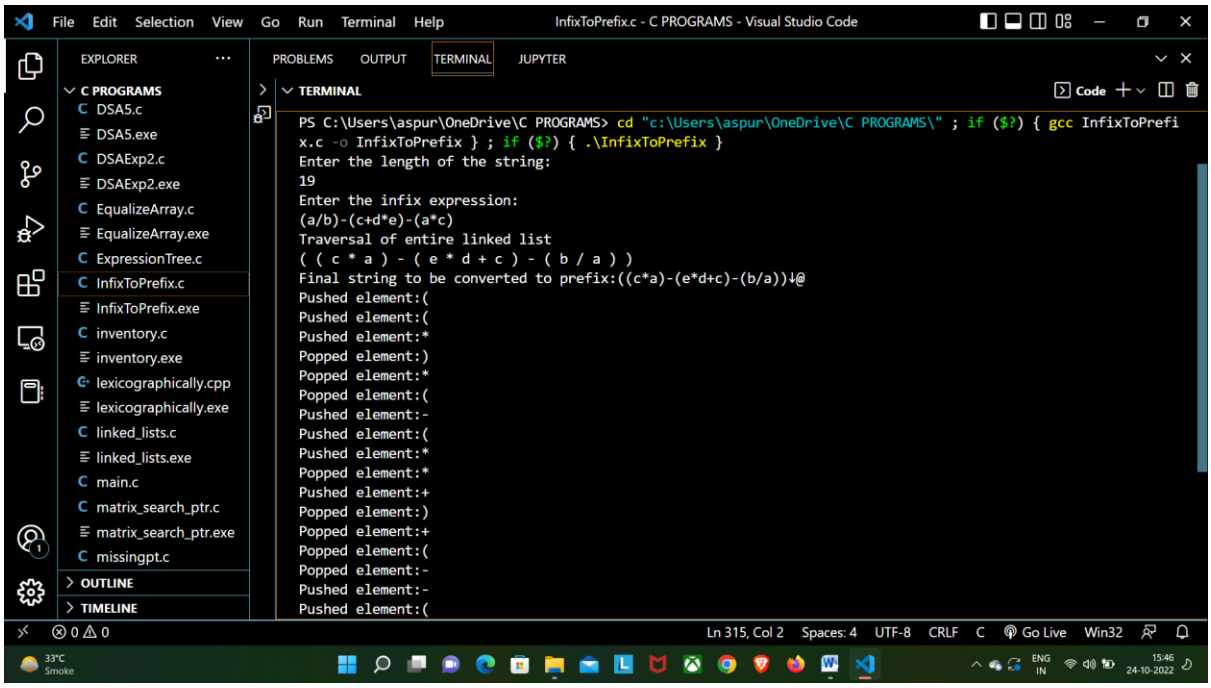
```c
printf("Enter the infix expression:\n");
scanf("%s",inf);

for(i=n-1;i>=0;i--){
    head1=InsertAtEnd(head1,inf[i]);
}

int k=Bracket_swap(head1,n);

head1=InsertAtFirst(head1,'(');
head1=InsertAtEnd(head1,')');

LinkedListTraversal(head1);
printf("\n");

char final[n+2];

struct sll * ptr = head1;

i=0;

while(ptr!=NULL){
    final[i]=ptr->data;
    ptr=ptr->next;
    i++;
}

//Now convert final string to postfix
printf("Final string to be converted to prefix:%s\n",final);
char prefix[k];

int c=-1;

for(i=0;i<n+2;i++){

    if(isalpha(final[i]) || isdigit(final[i])){
        ++c;
        prefix[c]=final[i];
    }

    else if(final[i]=='('){
        push(final[i],s);
    }

    else if(final[i]==')'){
        printf("Popped element:)\n");
        while(isEmpty(s)!=1 && peek(s)!='('){
```

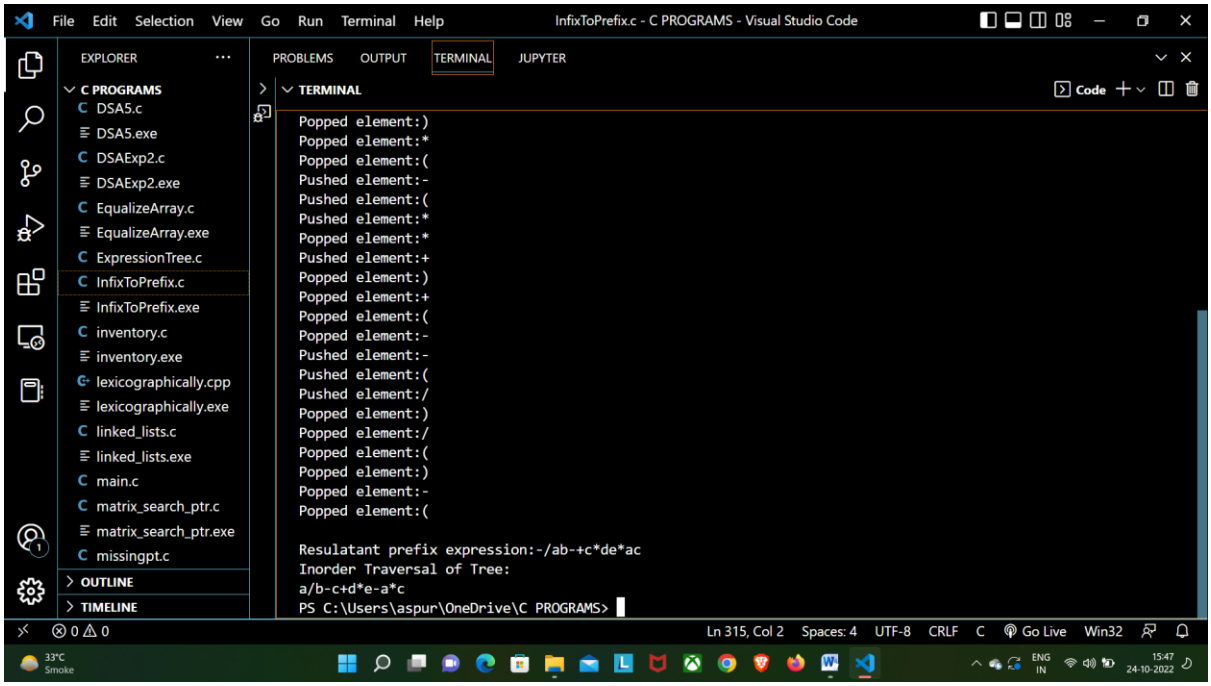
```
                ++c;
                prefix[c]=peek(s);
                pop(s);
            }
            if(peek(s)!='(' && isEmpty(s)!=1){
                exit(1);
            }
            else{
                pop(s);
            }
        }

        else{
            while(isEmpty(s)!=1 && precedence(final[i])<=precedence(peek(s))){
                ++c;
                prefix[c]=peek(s);
                pop(s);
            }
            push(final[i],s);
        }
    }
    while(isEmpty(s)!=1){
        ++c;
        prefix[c]=peek(s);
        pop(s);
    }
    prefix[k]='\0';
    //Reverse the obtained prefix expression to get a prefix expression
    strrev(prefix);

    printf("\nResulatant prefix expression:%s\n",prefix);

    struct tree *right_ins;
    struct tree *left_ins;
    struct tree *top;

    int j;
    for (j = k - 1; j >= 0; j--)
    {
        if (prefix[j] == '+' || prefix[j] == '-' || prefix[j] == '*' ||
prefix[j] == '/' || prefix[j] == '^')
        {
            top = creation(prefix[j]);
            left_ins = tree_pop(head);
            right_ins = tree_pop(head);
            top->left = left_ins;
            top->right = right_ins;
            tree_push(top);
```

```
        }
        else
        {
            struct tree *ptr = creation(prefix[j]);
            tree_push(ptr);
        }
    }
    printf("Inorder Traversal of Tree:\n");
    Inorder(top);
    printf("\n");
return 0;
}
```

OUTPUT SCREENSHOT:

Input





CONCLUSION:

In this experiment we learnt the concept of Expression trees and how to convert a Prefix and Postfix expression to an Expression tree. We understood the internal structure of a node, which contains a data part another node to the right and a node to the left and a next pointer to operate the expression tree as a stack. We understood that which node becomes left and right while popping out from stack.We also learnt how to convert an infix expression to a prefix expression . We performed various operations on stack like push, pop etc.