Name	**PRATIK PUJARI**
UID no.	**2020300054**
Experiment No.	6

AIM:	To implement Expression Tree
Program	

THOERY:

What is an Expression Tree?

A binary expression tree is a specific kind of a binary tree used to represent expressions. Two common types of expressions that a binary expression tree can represent are algebraic and boolean. These trees can represent expressions that contain both unary and binary operators

Overview:

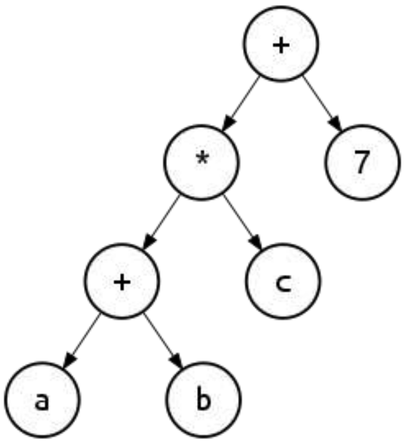
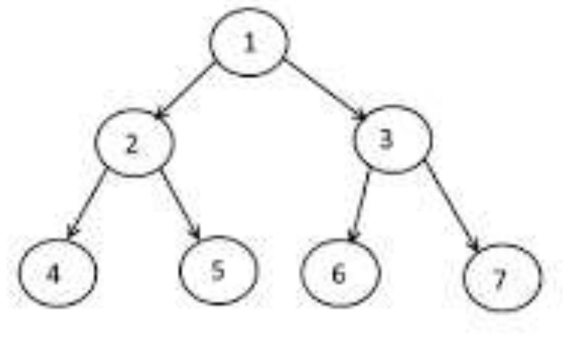
- The leaves of a binary expression tree are operands, such as constants or variable names, and the other nodes contain operators.
- These particular trees happen to be binary, because all of the operations are binary, and although this is the simplest case, it is possible for nodes to have more than two children.
- It is also possible for a node to have only one child, as is the case with the unary minus operator.
- An expression tree, *T*, can be evaluated by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees
-

How to Construct a Expression Tree?

Now 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.

In the end, the only element of the stack will be the root of an expression tree.

	Diagram of Expression Tree:  *(credits to en.wikipedia.org)* **Traversal in Expression Tree:** An algebraic expression can be produced from a binary expression tree by recursively producing a parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression. **Different Types of Traversal:**  This expression tree will serve as example for all traversals • **In-order Traversal** This general strategy (left, node, right) is known as an in-order traversal. Eg: In-order Traversal: 4 2 5 1 6 3 7 • **Post-order Traversal** An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator. This traversal strategy is generally known as post-order traversal. Eg : Post-order Traversal: 7 6 3 5 4 2 1

- **Pre-order Traversal**

A third strategy is to print out the operator first and then recursively print out the left and right subtree known as pre-order traversal.
Eg : Pre-order Traversal: 1 2 4 5 3 6 7

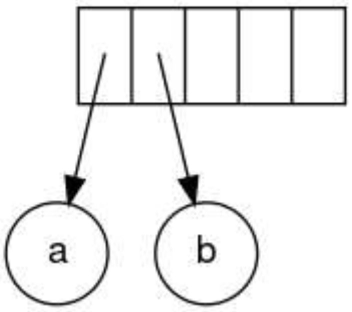
How to Construct a Binary Expression Tree:

Now For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

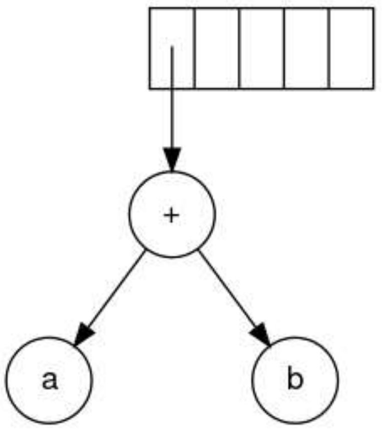
1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

Consider an example where an expression (a+b)*(c*(d+e)) where we have to convert it into an expression tree
First, we convert it into postfix expression which comes out to be: **ab+cde+****

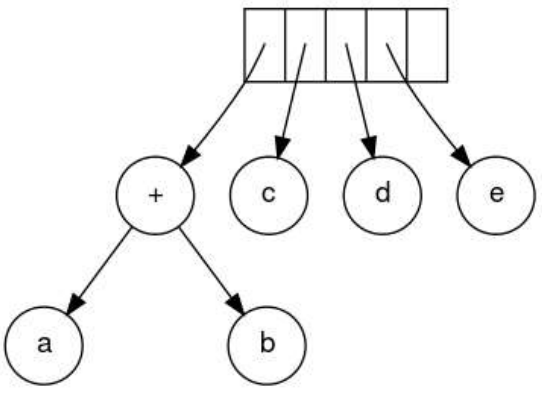
Step 1: Since the first two symbols are operands, one-node trees are created and pointers to them are pushed onto a stack. For convenience the stack will grow from left to right



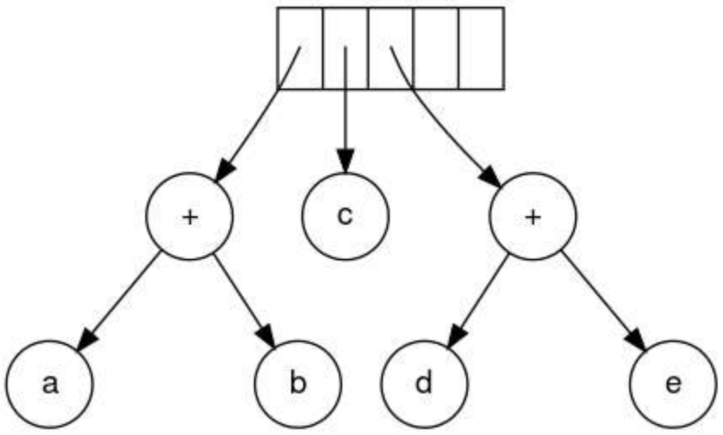
Step 2: The next symbol is a '+'. It pops the two pointers to the trees, a new tree is formed, and a pointer to it is pushed onto the stack.



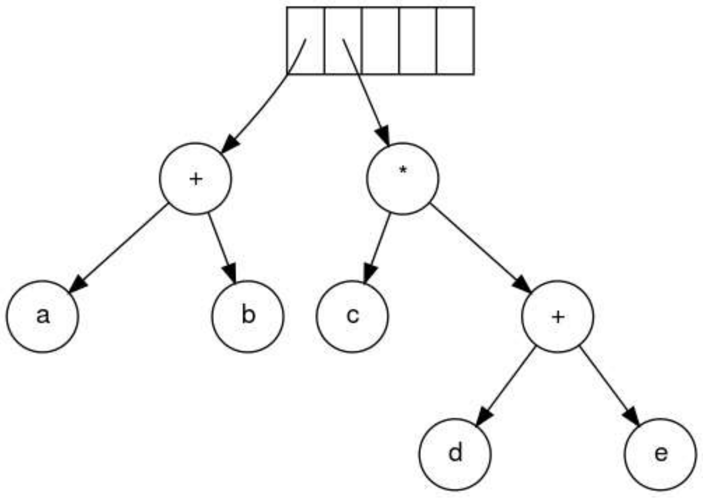
Step 3: Next, c, d, and e are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.



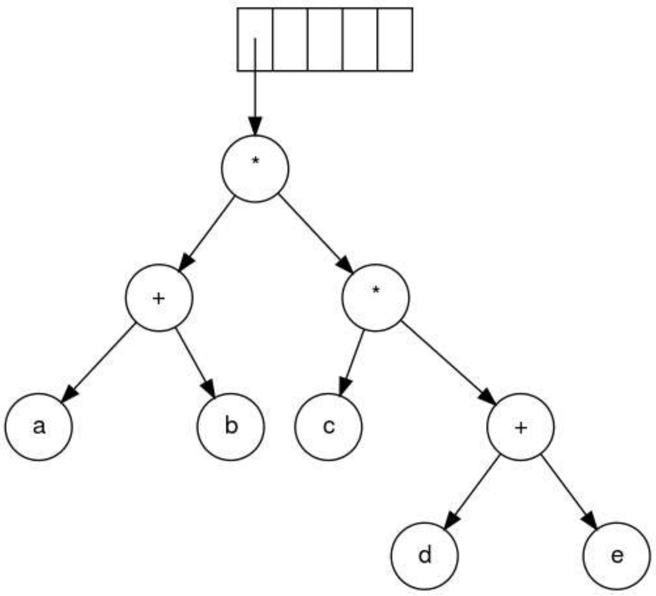
Step 4: Continuing, a '+' is read, and it merges the last two trees.



Step 5: Now, a '*' is read. The last two tree pointers are popped and a new tree is formed with a '*' as the root.



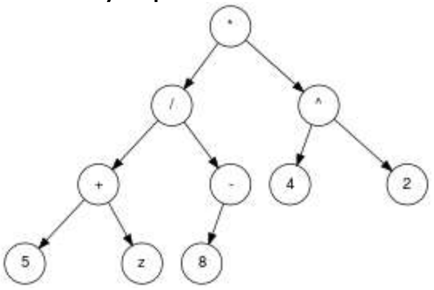
Step 6: Finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack



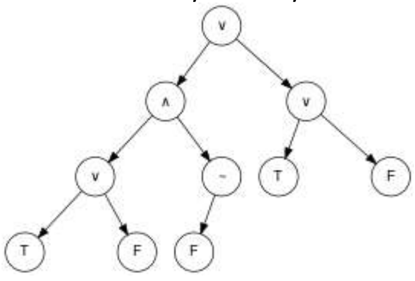
Step 7: End

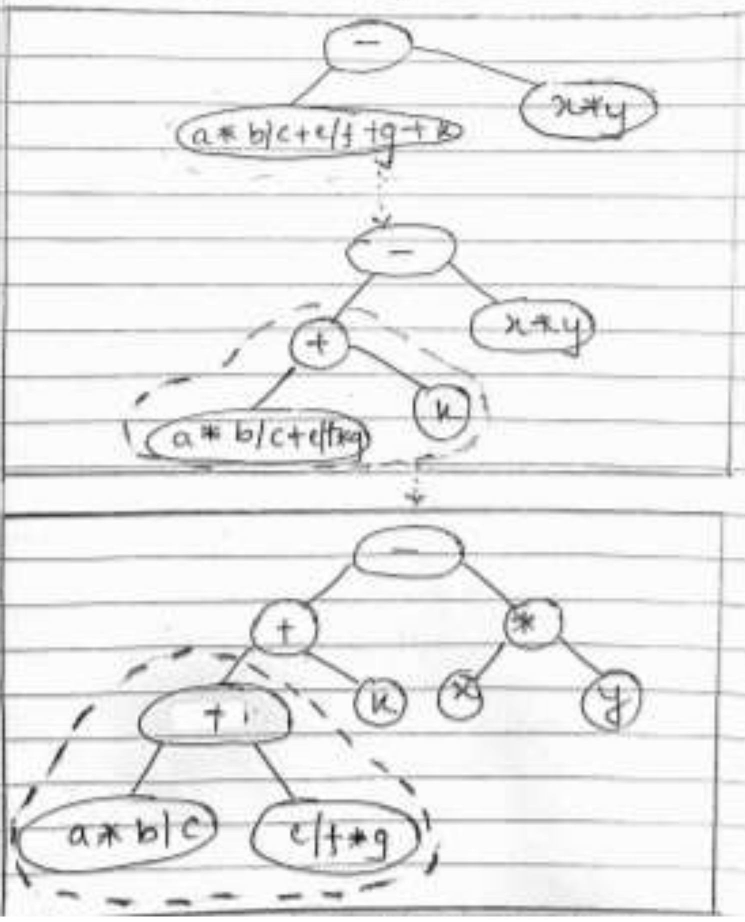
Applications of Expression Tree:

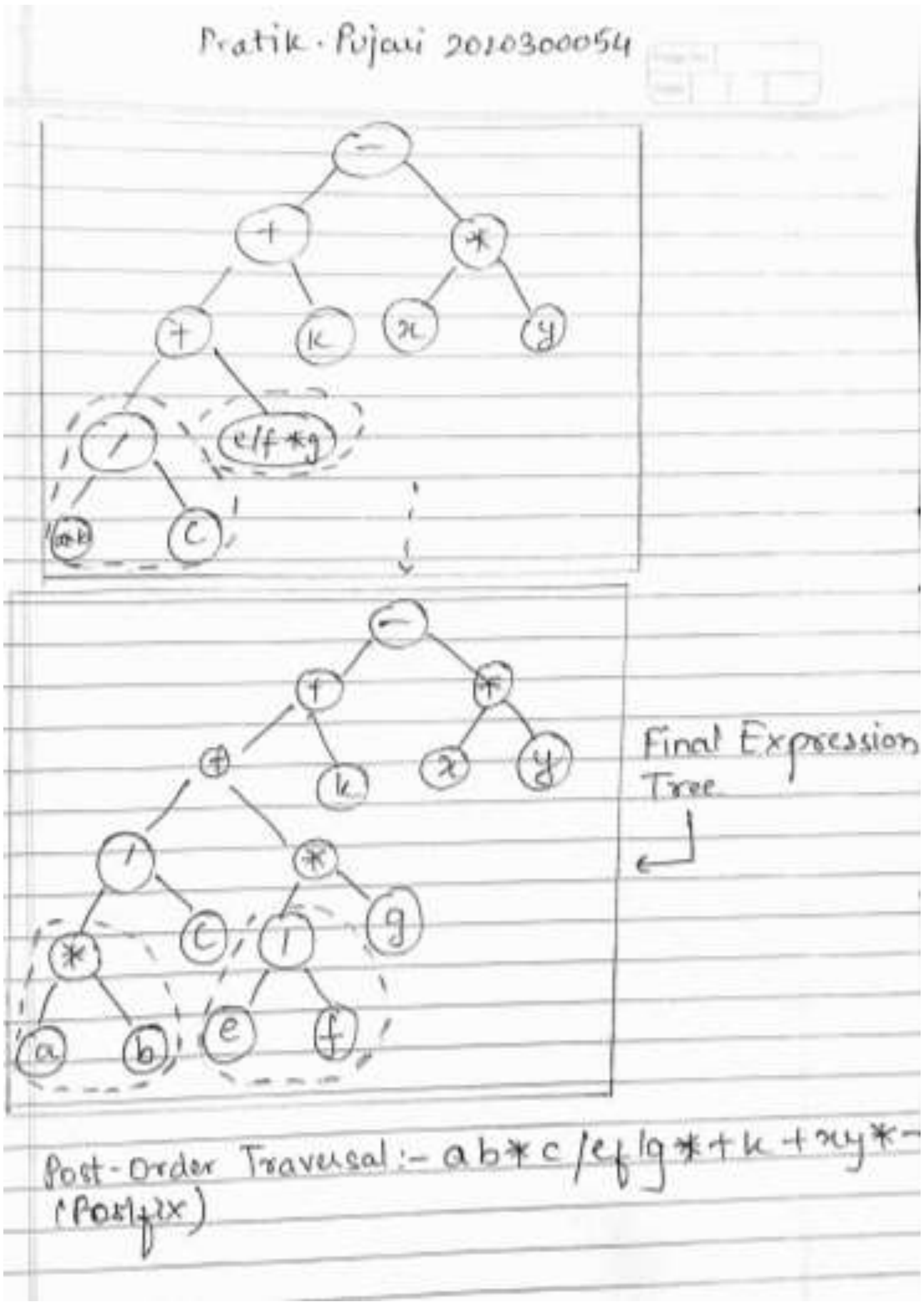
- Algebraic expression trees represent expressions that contain numbers, variables, and unary and binary operators. Some of the common operators are × (multiplication), ÷ (division), + (addition), − (subtraction), ^ (exponentiation), and - (negation). The operators are contained in the internal nodes of the tree, with the numbers and variables in the leaf nodes. The nodes of binary operators have two child nodes, and the unary operators have one child node.



- Boolean expressions are represented very similarly to algebraic expressions, the only difference being the specific values and operators used. Boolean expressions use true and false as constant values, and the operators "AND", "OR", "NOT"



PROBLEM STATEMENT :	Create an expression tree from a given post-order traversal and perform the evaluation.
PROBLEM SOLVING:	Pratik · Pujari 2020300054 Expression :- a * b/c + e/f * g + k - x*y Most Proirity :- ^ , *, /, -, + $\rightarrow$ Decreasing order Infix Expression :- a* b/c + e/f* g+k-x*y $\rightarrow$ Find operator with min precendance from right to left  − a*b/c+e/f*g+k x*y − + x*y a*b/c+e/f*g k − + * + k x y a*b/c e/f*g



Post-Order Traversal :- ab*c/ef/g*+k+xy*-
(Postfix)

PROGRAM:

ALGORITHM:

In-Order Traversal:

```
infix(tree):
if (tree not empty)
    if (tree token is operator)
       print (open parenthesis)
    end if
    infix (tree left subtree)
    print (tree token)
    infix (tree right subtree)
    if (tree token is operator)
       print (close parenthesis)
    end if
 end if
end infix
```

Post-order Traversal:

```
postfix(tree):
if (tree not empty)
    postfix (tree left subtree)
    postfix (tree right subtree)
    print (tree token)
 end if
end postfix
```

Pre-order Traversal:

```
prefix(tree):
if (tree not empty)
    print (tree token)
    prefix (tree left subtree)
    prefix (tree right subtree)
 end if
end prefix
```

PROGRAM:

Node class:

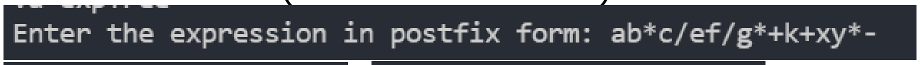
```
public class Node {
    char data;
    Node left, right;
    Node link;

    Node(char data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }

    Node(char c, Node left, Node right) {
        this.data = c;
        this.left = left;
        this.right = right;
    }
}
```

Class Binary Tree:

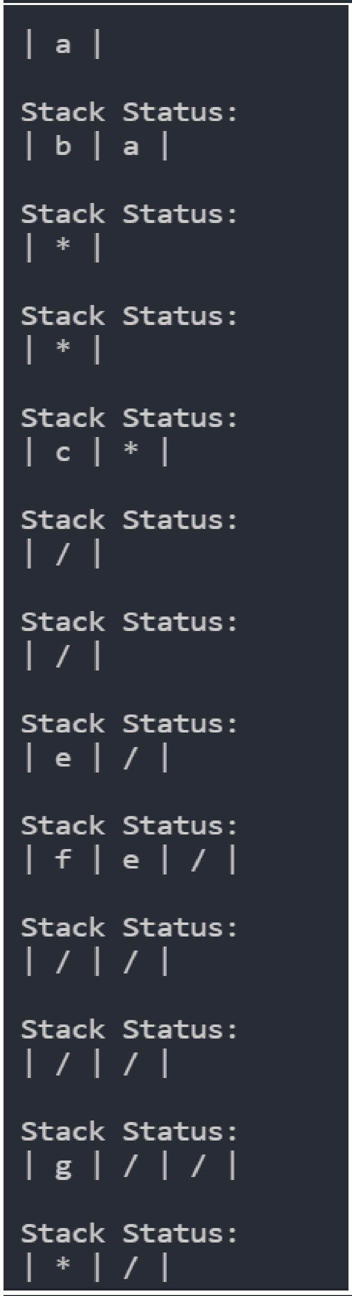
```
import java.util.*;
import java.util.Stack;

public class expTree {

    public static boolean istreeNode(char c) {
        return (c == '+' || c == '-' || c == '*' || c == '/' || c
== '^');
    }

    public void postorder(Node root) {
        if (root == null) {
            return;
        }
        postorder(root.left);
        postorder(root.right);
        System.out.print(root.data);
    }

    public void inorder(Node root) {
        if (root == null) {
            return;
        }

        if (istreeNode(root.data)) {
            System.out.print("(");
        }

        inorder(root.left);
        System.out.print(root.data);
        inorder(root.right);

        if (istreeNode(root.data)) {
            System.out.print(")");
        }
    }

    public Node makePostfix(String postfix) {
        if (postfix == null || postfix.length() == 0) {
            return null;
        }
        Stack<Node> s = new Stack<>();
        StackUsingLL stack = new StackUsingLL();

        Node treeNode, x, y;
        for (char c : postfix.toCharArray()) {
            // String postfix = "ab+cde+**"
            if (istreeNode(c)) {
                treeNode = new Node(c);
                x = s.pop();
                y = s.pop();

                stack.pop();
                stack.pop();

                treeNode.right = x;
                treeNode.left = y;
                s.push(treeNode);
                stack.push(treeNode);
                stack.display();

            } else {
                treeNode = new Node(c);
                s.push(treeNode);
                stack.push(treeNode);
            }
        }
        treeNode = (Node) s.peek();
        s.pop();
        return treeNode;
    }

    public int findDigit(char ch) {
        return ch - '0';
    }

    public boolean isDigit(char ch) {
        if (ch >= '0' && ch <= '9')
            return true;
        return false;
    }

    public double evaluateTree(Node node) {
        if (node.left == null && node.right == null)
            return findDigit(node.data);
        else {
            double result = 0.0;
            double left = evaluateTree(node.left);
            double right = evaluateTree(node.right);
            char treeNode = node.data;

            switch (treeNode) {
            case '+':
                result = left + right;
                break;
            case '-':
                result = left - right;
                break;
            case '*':
                result = left * right;
                break;
            case '/':
                result = left / right;
                break;
            default:
                result = left + right;
                break;
            }
            return result;
        }
    }

    public boolean checkNumber(String postfix) {
        for (char c : postfix.toCharArray()) {
            if (isDigit(c)) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the expression in postfix
form: ");
        String postfix = input.next();

        expTree tree = new expTree();
        Node root = tree.makePostfix(postfix);
        BinaryTree newTree = new BinaryTree();

        System.out.print("\nPostfix Expression: ");
        newTree.printPostorder(root);

        System.out.print("\nInfix Expression: ");
        tree.inorder(root);
        if (tree.checkNumber(postfix)) {
            double result = tree.evaluateTree(root);
            System.out.print("\nThe evaluation of the binary
tree: " + result);
            input.close();
        }
    }
}
```

SCREENSHOT:

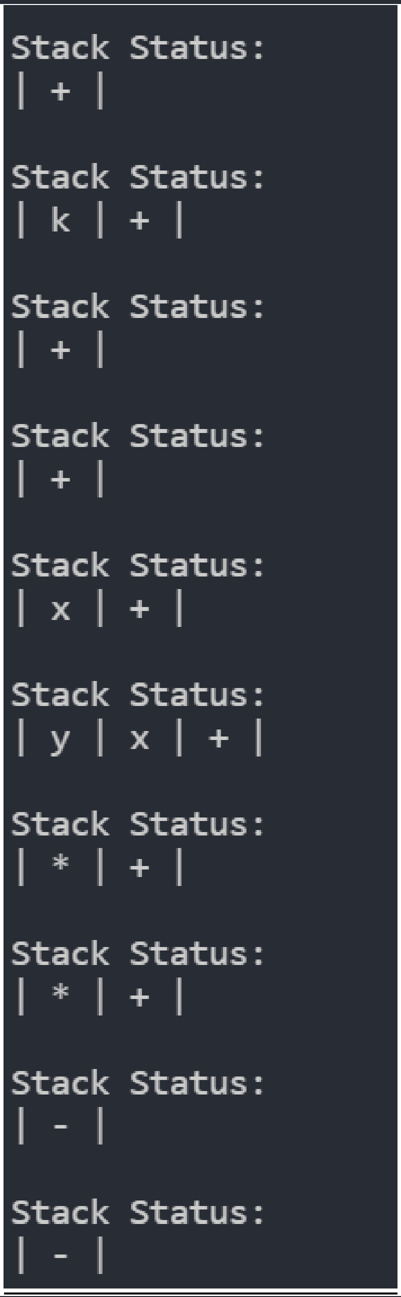
Normal Traversal(Without evaluation):



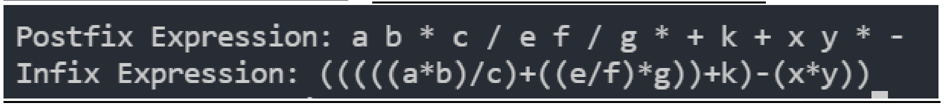
```
Enter the expression in postfix form: ab*c/ef/g*+k+xy*-
```



```
| a |

Stack Status:
| b | a |

Stack Status:
| * |

Stack Status:
| * |

Stack Status:
| c | * |

Stack Status:
| / |

Stack Status:
| / |

Stack Status:
| e | / |

Stack Status:
| f | e | / |

Stack Status:
| / | / |

Stack Status:
| / | / |

Stack Status:
| g | / | / |

Stack Status:
| * | / |
```

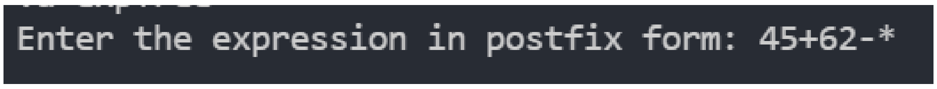


```
Stack Status:
| + |

Stack Status:
| k | + |

Stack Status:
| + |

Stack Status:
| + |

Stack Status:
| x | + |

Stack Status:
| y | x | + |

Stack Status:
| * | + |

Stack Status:
| * | + |

Stack Status:
| - |

Stack Status:
| - |
```

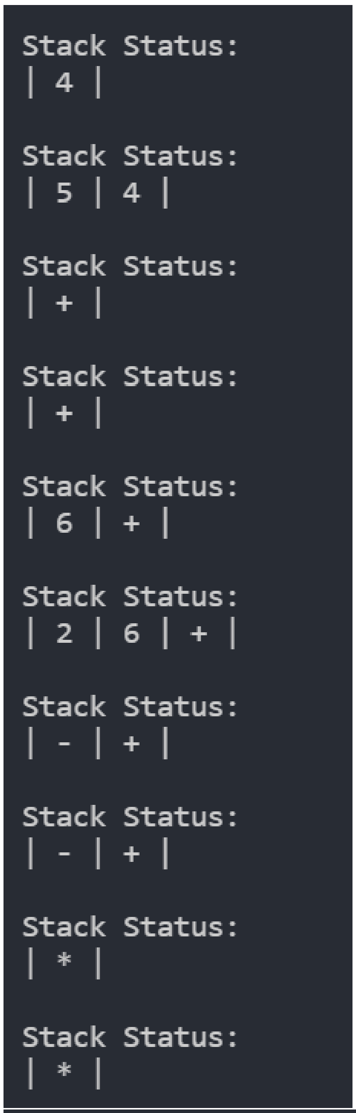
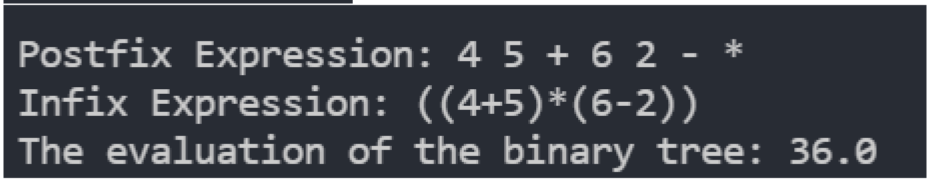


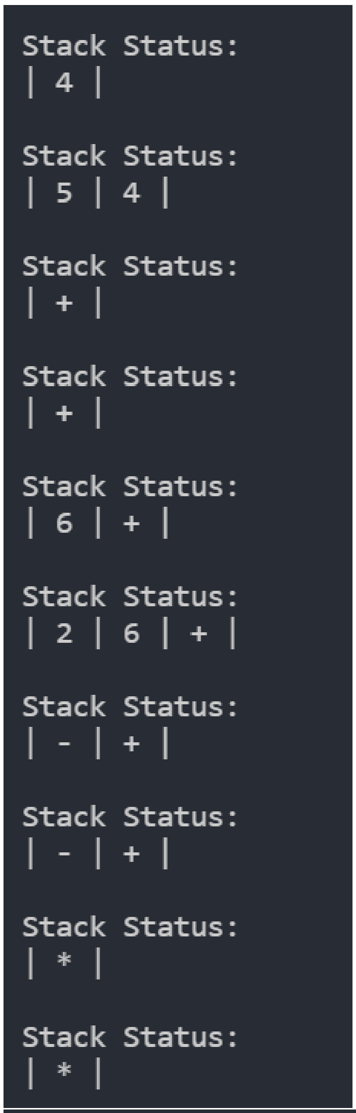
```
Postfix Expression: a b * c / e f / g * + k + x y * -
Infix Expression: (((((a*b)/c)+((e/f)*g))+k)-(x*y))
```

Evaluation:

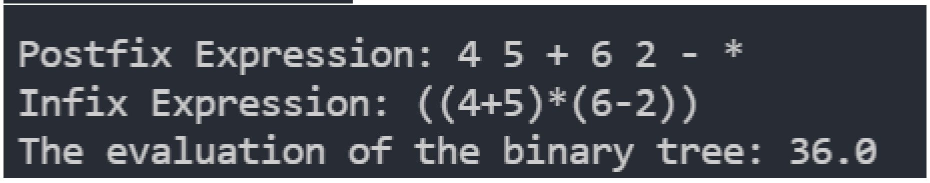


```
Enter the expression in postfix form: 45+62-*
```



```
Stack Status:
| 4 |

Stack Status:
| 5 | 4 |

Stack Status:
| + |

Stack Status:
| + |

Stack Status:
| 6 | + |

Stack Status:
| 2 | 6 | + |

Stack Status:
| - | + |

Stack Status:
| - | + |

Stack Status:
| * |

Stack Status:
| * |
```



```
Postfix Expression: 4 5 + 6 2 - *
Infix Expression: ((4+5)*(6-2))
The evaluation of the binary tree: 36.0
```

CONCLUSION: Things learnt during procedural programming of the program

- Learnt that Binary trees traversals can be used as to calculate and evaluate expression
- Learnt to use Nodes of Stack to form a Binary tree
- Learnt to generate and evaluate an Expression Tree using the postfix
- Learnt to combine to different data structures to perform difficult tasks that can’t be performed with one data structure(conditional*)