**Assignment No. 02**

Name: Akshay Gangurde

Roll No.: 222022

Batch: B1

**Problem Statement:** Implement threaded binary search tree and perform recursive and non- recursive In-order, pre-order and post-order traversals.

**Objectives:**

1. To understand the concept of Threaded Binary Search Tree.
2. To efficiently traverse the tree in sorted order without using recursion or maintaining an explicit stack.
3. To study types of threaded binary tree and how they are efficient for Inorder traversal.

**Theory:**

A threaded binary tree is a type of binary tree data structure where the empty left and right child pointers in a binary tree are replaced with threads that link nodes directly to their in-order predecessor or successor, thereby providing a way to traverse the tree without using recursion or a stack.

Threaded binary trees can be useful when space is a concern, as they can eliminate the need for a stack during traversal. However, they can be more complex to implement than standard binary trees.

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

static class Node

{

int data;

Node left, right;

boolean rightThread, leftThread;

}

**Algorithm:**

1. **Insert Method Algorithm**:
   * Create a new node with the given data.
   * Traverse the tree:
     + If data is less than the current node, move to the left child.
       - If left child is null, insert new node as left child with a thread pointing to its parent.
     + If data is greater than or equal to the current node:
       - If the current node has a thread, insert new node as its right child, updating threads accordingly.
       - Otherwise, move to the right child and continue traversing.
2. **Main Method Algorithm**:
   * Create a scanner object for user input.
   * Create an instance of the single-threaded binary tree.
   * Display a menu:
     + Option to insert data: Continuously insert data until -1 is entered.
     + Option to print inorder traversal.
     + Option to exit the program.
3. **End of Algorithm**:
   * The program exits when the user chooses to exit.

**Input(Source Code):**

import java.util.Stack;

// Node class to represent each node of the expression tree

class Node {

    String value;

    Node left, right;

    Node(String item) {

        value = item;

        left = right = null;

    }

}

// Class to construct expression tree and perform traversals

class ExpressionTree {

    Node root;

    // Function to construct expression tree from postfix expression

    public void constructTreeFromPostfix(String postfix) {

        Stack<Node> stack = new Stack<>();

        for (int i = 0; i < postfix.length(); i++) {

            char c = postfix.charAt(i);

            // If operand, create a node and push it to the stack

            if (Character.isLetterOrDigit(c)) {

                stack.push(new Node(Character.toString(c)));

            }

            // If operator, pop two operands from stack, create a node with operator as value,

            // make them children of the created node, and push the new node to the stack

            else {

                Node right = stack.pop();

                Node left = stack.pop();

                Node newNode = new Node(Character.toString(c));

                newNode.left = left;

                newNode.right = right;

                stack.push(newNode);

            }

        }

        root = stack.pop();

    }

    // Recursive in-order traversal

    public void recursiveInOrder(Node node) {

        if (node == null) return;

        recursiveInOrder(node.left);

        System.out.print(node.value + " ");

        recursiveInOrder(node.right);

    }

    // Non-recursive in-order traversal using a stack

    public void nonRecursiveInOrder() {

        if (root == null) return;

        Stack<Node> stack = new Stack<>();

        Node current = root;

        while (current != null || !stack.isEmpty()) {

            while (current != null) {

                stack.push(current);

                current = current.left;

            }

            current = stack.pop();

            System.out.print(current.value + " ");

            current = current.right;

        }

    }

    // Recursive pre-order traversal

    public void recursivePreOrder(Node node) {

        if (node == null) return;

        System.out.print(node.value + " ");

        recursivePreOrder(node.left);

        recursivePreOrder(node.right);

    }

    // Non-recursive pre-order traversal using a stack

    public void nonRecursivePreOrder() {

        if (root == null) return;

        Stack<Node> stack = new Stack<>();

        stack.push(root);

        while (!stack.isEmpty()) {

            Node current = stack.pop();

            System.out.print(current.value + " ");

            if (current.right != null) stack.push(current.right);

            if (current.left != null) stack.push(current.left);

        }

    }

    // Recursive post-order traversal

    public void recursivePostOrder(Node node) {

        if (node == null) return;

        recursivePostOrder(node.left);

        recursivePostOrder(node.right);

        System.out.print(node.value + " ");

    }

    // Non-recursive post-order traversal using two stacks

    public void nonRecursivePostOrder() {

        if (root == null) return;

        Stack<Node> stack1 = new Stack<>();

        Stack<Node> stack2 = new Stack<>();

        stack1.push(root);

        while (!stack1.isEmpty()) {

            Node current = stack1.pop();

            stack2.push(current);

            if (current.left != null) stack1.push(current.left);

            if (current.right != null) stack1.push(current.right);

        }

        while (!stack2.isEmpty()) {

            System.out.print(stack2.pop().value + " ");

        }

    }

}

public class Main {

    public static void main(String[] args) {

        String postfixExpression = "ab+c\*";

        ExpressionTree tree = new ExpressionTree();

        // Construct expression tree from postfix expression

        tree.constructTreeFromPostfix(postfixExpression);

        // Perform recursive and non-recursive traversals

        System.out.println("Recursive In-order traversal:");

        tree.recursiveInOrder(tree.root);

        System.out.println("\nNon-recursive In-order traversal:");

        tree.nonRecursiveInOrder();

        System.out.println("\n\nRecursive Pre-order traversal:");

        tree.recursivePreOrder(tree.root);

        System.out.println("\nNon-recursive Pre-order traversal:");

        tree.nonRecursivePreOrder();

        System.out.println("\n\nRecursive Post-order traversal:");

        tree.recursivePostOrder(tree.root);

        System.out.println("\nNon-recursive Post-order traversal:");

        tree.nonRecursivePostOrder();

    }

}

**Output:**

Recursive In-order traversal:

a + b \* c

Non-recursive In-order traversal:

a + b \* c

Recursive Pre-order traversal:

+ a \* b c

Non-recursive Pre-order traversal:

+ a \* b c

Recursive Post-order traversal:

a b c \* +

Non-recursive Post-order traversal:

a b c \* +