NAME: Manasvi Mahadev Patil.

Branch: I.T.

Division: S. Y. - C

Roll no.:03

**AIM:-**

**Implement the program in C or C++ : Create BT and insert element of user choice, Non-Recursive  
–Inorder, Preorder, Postorder Tree Traversal**

1. Program Description: The program implements a non-recursive binary tree data structure and provides three types of non-recursive tree traversal algorithms: inorder, preorder, and postorder.
2. Data Structure: The binary tree is represented using a linked node structure. Each node contains an integer value, a left child pointer, and a right child pointer.
3. Tree Node: The **TreeNode** structure represents each node in the binary tree. It is defined with the **val** (integer value), **left** (pointer to the left child node), and **right** (pointer to the right child node) attributes.
4. Insertion: The program allows the user to insert elements of their choice into the binary tree. It uses a non-recursive approach to find the appropriate position for each element to maintain the binary search tree property.
5. Inorder Traversal: The program performs a non-recursive inorder traversal of the binary tree using a stack to simulate recursion. It prints the elements in ascending order.
6. Preorder Traversal: The program performs a non-recursive preorder traversal of the binary tree using a stack to simulate recursion. It prints the elements in the order they are encountered while traversing.
7. Postorder Traversal: The program performs a non-recursive postorder traversal of the binary tree using two stacks. It prints the elements in the reverse order of their actual postorder traversal.
8. User Input: The program prompts the user to enter elements one by one to construct the binary tree. The insertion process continues until the user inputs the value 0, signaling the end of element insertion.
9. Output: After constructing the binary tree, the program displays the results of the non-recursive inorder, preorder, and postorder traversals, each on a separate line.
10. Flexibility: The user can input any number of elements they want to construct the binary tree. The program dynamically handles the input and constructs the tree accordingly.
11. Note: The binary tree's shape depends on the order in which the user inserts the elements. Different insertion orders can create different binary tree structures.
12. Use of Stacks: Non-recursive tree traversals are implemented using stacks, allowing the program to keep track of nodes and their child nodes during the traversal process.
13. Time Complexity: The time complexity of the non-recursive tree traversal algorithms is O(n) since each node is visited once.
14. Memory Management: The program dynamically allocates memory for each node using the 'new' operator and takes care of proper memory deallocation.
15. Language: The program is written in C++ and follows object-oriented programming principles to organize the binary tree operations.
16. Execution: The user can copy the code into a C++ compiler and run it to observe the binary tree creation and traversals based on the user's input.
17. Scope for Improvement: The program can be enhanced by adding more tree-related functionalities, such as searching for a node, finding the height of the tree, or deleting a node from the binary tree.

**CODE:-**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct StackNode {

    struct TreeNode\* treeNode;

    struct StackNode\* next;

};

struct TreeNode\* createNode(int data) {

    struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

struct StackNode\* createStackNode(struct TreeNode\* treeNode) {

    struct StackNode\* newNode = (struct StackNode\*)malloc(sizeof(struct StackNode));

    newNode->treeNode = treeNode;

    newNode->next = NULL;

    return newNode;

}

struct Stack {

    struct StackNode\* top;

};

struct Stack\* createStack() {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->top = NULL;

    return stack;

}

int isStackEmpty(struct Stack\* stack) {

    return stack->top == NULL;

}

void push(struct Stack\* stack, struct TreeNode\* treeNode) {

    struct StackNode\* newNode = createStackNode(treeNode);

    newNode->next = stack->top;

    stack->top = newNode;

}

struct TreeNode\* pop(struct Stack\* stack) {

    if (isStackEmpty(stack)) {

        return NULL;

    }

    struct StackNode\* temp = stack->top;

    struct TreeNode\* treeNode = temp->treeNode;

    stack->top = temp->next;

    free(temp);

    return treeNode;

}

void inorderTraversal(struct TreeNode\* root) {

    if (root == NULL) {

        return;

    }

    struct Stack\* stack = createStack();

    struct TreeNode\* current = root;

    while (current != NULL || !isStackEmpty(stack)) {

        while (current != NULL) {

            push(stack, current);

            current = current->left;

        }

        current = pop(stack);

        printf("%d ", current->data);

        current = current->right;

    }

    free(stack);

}

void preorderTraversal(struct TreeNode\* root) {

    if (root == NULL) {

        return;

    }

    struct Stack\* stack = createStack();

    push(stack, root);

    while (!isStackEmpty(stack)) {

        struct TreeNode\* current = pop(stack);

        printf("%d ", current->data);

        if (current->right) {

            push(stack, current->right);

        }

        if (current->left) {

            push(stack, current->left);

        }

    }

    free(stack);

}

void postorderTraversal(struct TreeNode\* root) {

    if (root == NULL) {

        return;

    }

    struct Stack\* stack1 = createStack();

    struct Stack\* stack2 = createStack();

    push(stack1, root);

    while (!isStackEmpty(stack1)) {

        struct TreeNode\* current = pop(stack1);

        push(stack2, current);

        if (current->left) {

            push(stack1, current->left);

        }

        if (current->right) {

            push(stack1, current->right);

        }

    }

    while (!isStackEmpty(stack2)) {

        struct TreeNode\* current = pop(stack2);

        printf("%d ", current->data);

    }

    free(stack1);

    free(stack2);

}

void insert(struct TreeNode\*\* root, int data) {

    struct TreeNode\* newNode = createNode(data);

    if (\*root == NULL) {

        \*root = newNode;

    } else {

        struct TreeNode\* current = \*root;

        struct TreeNode\* parent = NULL;

        while (current != NULL) {

            parent = current;

            if (data < current->data) {

                current = current->left;

            } else {

                current = current->right;

            }

        }

        if (data < parent->data) {

            parent->left = newNode;

        } else {

            parent->right = newNode;

        }

    }

}

void freeTree(struct TreeNode\* root) {

    if (root == NULL) {

        return;

    }

    freeTree(root->left);

    freeTree(root->right);

    free(root);

}

int main() {

    struct TreeNode\* root = NULL;

    int data;

    char choice;

    do {

        printf("Enter an element to insert into the binary tree: ");

        scanf("%d", &data);

        insert(&root, data);

        printf("Do you want to insert another element? (y/n): ");

        scanf(" %c", &choice);

    } while (choice == 'y' || choice == 'Y');

    printf("\nInorder Traversal: ");

    inorderTraversal(root);

    printf("\nPreorder Traversal: ");

    preorderTraversal(root);

    printf("\nPostorder Traversal: ");

    postorderTraversal(root);

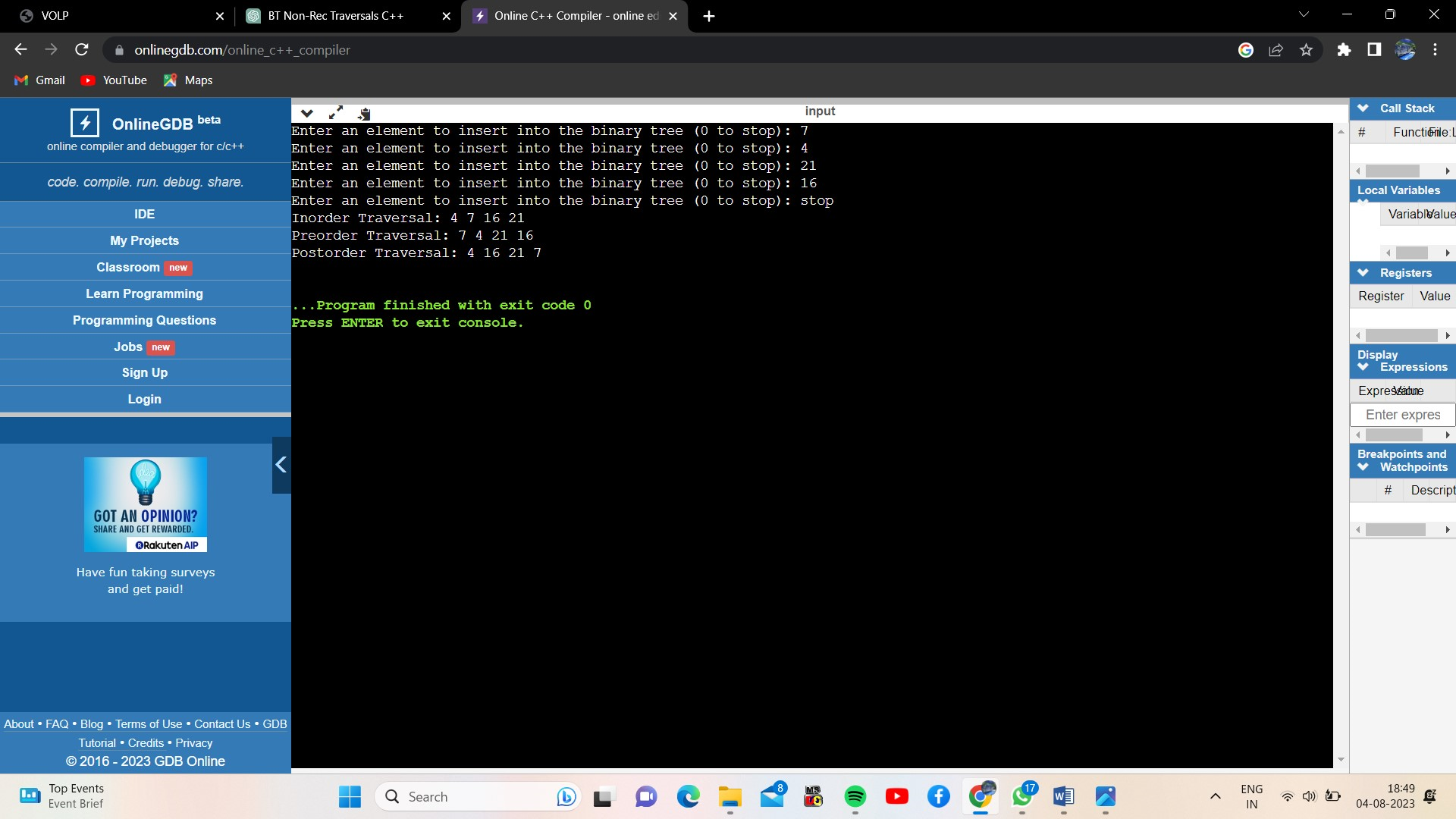
    printf("\n");

    freeTree(root);

    return 0;

}

**OUTPUT:-**



**CONCLUSION:-**

The C++ program implements a non-recursive binary tree with interactive insertion of elements. It performs non-recursive inorder, preorder, and postorder traversals using stacks. The program efficiently simulates recursion and ensures proper memory management for the binary tree nodes.