**Assignment-5**

**Name of Student: Ayush Sanjay Dhangar**

**Batch: 02 Class: SY\_IT-A Roll No: 42**

**PRN: 12210406**

**Subject: IT2265 Advanced Data Structures**

**Problem Statement:** Write C or C++ program to implement **non-recursive tree traversal**(inorder, preorder and post order)

**Code:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

    int data;

    struct Node \*left, \*right;

} Node;

typedef struct Stack {

    Node \*\*s;

    int top;

    int capacity;

} Stack;

void push(Stack \*s, Node \*t);

Node \*pop(Stack \*s);

int isEmpty(Stack \*s);

void inorder(Node \*root);

void preorder(Node \*root);

void postorder(Node \*root);

Node \*createNode(int data);

Stack \*createStack(int capacity);

void freeTree(Node \*root);

void insertNode(Node \*\*root, int data);

int main() {

    int choice, data;

    Node \*root = NULL;

    do {

        printf("\nBinary Tree Operations\n");

        printf("1. Insert Node\n");

        printf("2. Inorder Traversal\n");

        printf("3. Preorder Traversal\n");

        printf("4. Postorder Traversal\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter data for the new node: ");

                scanf("%d", &data);

                insertNode(&root, data);

                break;

            case 2:

                printf("Inorder Traversal: ");

                inorder(root);

                printf("\n");

                break;

            case 3:

                printf("Preorder Traversal: ");

                preorder(root);

                printf("\n");

                break;

            case 4:

                printf("Postorder Traversal: ");

                postorder(root);

                printf("\n");

                break;

            case 5:

                printf("Exiting...\n");

                freeTree(root);

                break;

            default:

                printf("Invalid choice. Please enter a number between 1 and 5(both inclusive).\n");

        }

    } while (choice != 5);

    return 0;

}

void push(Stack \*s, Node \*t) {

    if (s->top == s->capacity - 1) {

        printf("Stack Overflow\n");

        return;

    }

    s->top++;

    s->s[s->top] = t;

}

Node \*pop(Stack \*s) {

    if (s->top == -1) {

        printf("Stack Underflow\n");

        return NULL;

    }

    Node \*t = s->s[s->top];

    s->top--;

    return t;

}

int isEmpty(Stack \*s) {

    return s->top == -1;

}

void inorder(Node \*root) {

    if (root == NULL) return;

    Stack \*s = createStack(100);

    Node \*T = root;

    while (T != NULL || !isEmpty(s)) {

        while (T != NULL) {

            push(s, T);

            T = T->left;

        }

        T = pop(s);

        printf("%d ", T->data);

        T = T->right;

    }

}

void preorder(Node \*root) {

    if (root == NULL) return;

    Stack \*s = createStack(100);

    Node \*T = root;

    while (T != NULL || !isEmpty(s)) {

        while (T != NULL) {

            printf("%d ", T->data);

            push(s, T);

            T = T->left;

        }

        T = pop(s);

        T = T->right;

    }

}

void postorder(Node \*root) {

    if (root == NULL) return;

    Stack \*s1 = createStack(100);

    Stack \*s2 = createStack(100);

    push(s1, root);

    while (!isEmpty(s1)) {

        root = pop(s1);

        push(s2, root);

        if (root->left != NULL)

            push(s1, root->left);

        if (root->right != NULL)

            push(s1, root->right);

    }

    while (!isEmpty(s2)) {

        root = pop(s2);

        printf("%d ", root->data);

    }

}

Node \*createNode(int data) {

    Node \*newNode = (Node \*)malloc(sizeof(Node));

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(EXIT\_FAILURE);

    }

    newNode->data = data;

    newNode->left = newNode->right = NULL;

    return newNode;

}

Stack \*createStack(int capacity) {

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

    if (stack == NULL) {

        printf("Memory allocation failed\n");

        exit(EXIT\_FAILURE);

    }

    stack->s = (Node \*\*)malloc(capacity \* sizeof(Node \*));

    if (stack->s == NULL) {

        printf("Memory allocation failed\n");

        exit(EXIT\_FAILURE);

    }

    stack->top = -1;

    stack->capacity = capacity;

    return stack;

}

void freeTree(Node \*root) {

    if (root != NULL) {

        freeTree(root->left);

        freeTree(root->right);

        free(root);

    }

}

void insertNode(Node \*\*root, int data) {

    Node \*newNode = createNode(data);

    if (\*root == NULL) {

        \*root = newNode;

    } else {

        Node \*current = \*root;

        Node \*parent = NULL;

        while (1) {

            parent = current;

            if (data < current->data) {

                current = current->left;

                if (current == NULL) {

                    parent->left = newNode;

                    return;

                }

            } else {

                current = current->right;

                if (current == NULL) {

                    parent->right = newNode;

                    return;

                }

            }

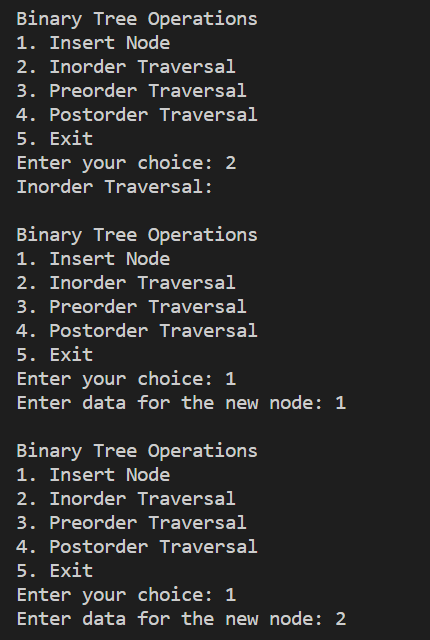
        }

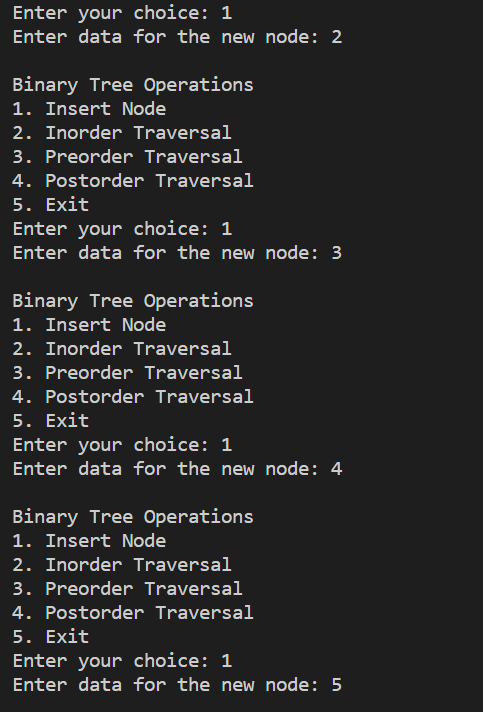
    }

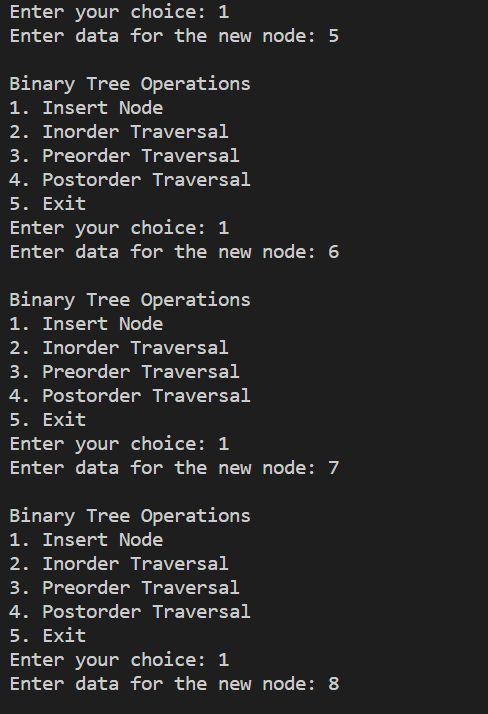
}

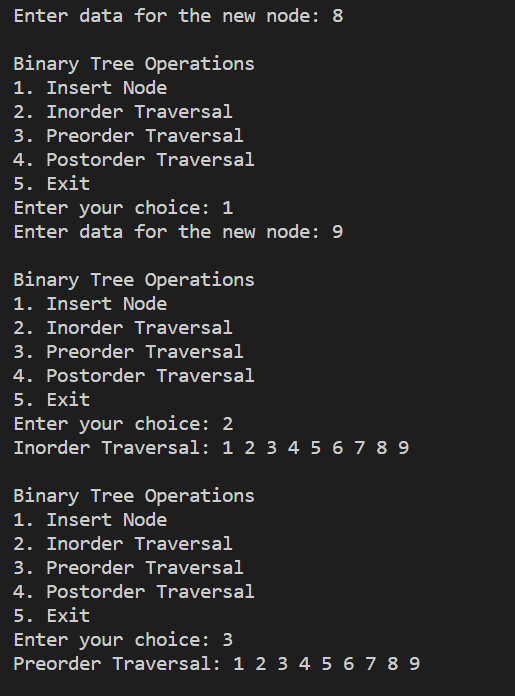
**Result:**

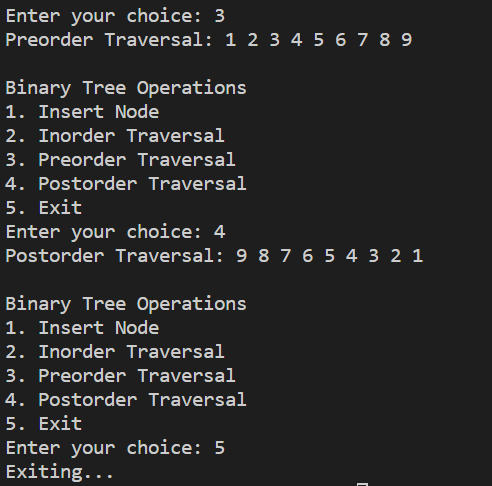
**Actual Output:**

****

****

****

****

****