EXPERIMENT 8

Aim:

1. Implement the Binary Tree and perform any one of the following three types of traversals: (Implement iterative method of traversal, not recursive) a. Pre-order Traversal b. In-order Traversal c. Post-order Traversal

Theory (Inorder Traversal):

Inorder Traversal (Left, Root, Right): In iterative inorder traversal, we use a stack to simulate the recursive process. We traverse the left subtree first, push nodes onto the stack, and process each node after visiting its left child, before moving to the right child.

Time Complexity:

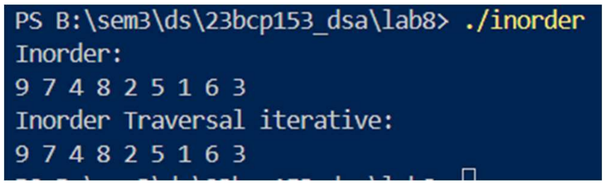
$O(n)$ - We visit each node exactly once, where n is the number of nodes in the tree.

Program:

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

// left root right - inorder

typedef struct Tree {
   int info;
   struct Tree *left;
   struct Tree *right;
} Tree;

typedef struct Stacktree
{
   int top;
   int capacity;
   Tree **array;
} Stacktree;

Tree *makeTnode(int info);
void printInorder(Tree *root);
void freetree(Tree *root);
void printInorderiter(Tree *root);
Stacktree *createStackt(int n);
Tree *popt(Stacktree *s);
void pusht(Stacktree *s, Tree *node);
int isFullt(Stacktree *s);
int isEmptyt(Stacktree *s);
Tree *peekt(Stacktree *s);

int main(void)
{
   Tree *A = makeTnode(1);
   Tree *B = makeTnode(2);
   Tree *C = makeTnode(3);
```

```
    Tree *D = makeTnode(4);
    Tree *E = makeTnode(5);
    Tree *F = makeTnode(6);
    Tree *G = makeTnode(7);
    Tree *H = makeTnode(8);
    Tree *I = makeTnode(9);
    A->left = B;
    A->right = C;
    B->left = D;
    B->right = E;
    C->left = F;
    D->left = G;
    D->right = H;
    G->left = I;

    printf("Inorder:\n");
    printInorder(A);
    printf("\n");
    printInorderiter(A);
    freetree(A);

    return 0;
}

void printInorder(Tree *root)
{
    if (root == NULL)
    {
        return;
    }

    printInorder(root->left);
    printf("%d ", root->info);
    printInorder(root->right);
}

// Make these below type of dynamic functions later

// Tree *insert()
// {

// }

Tree *makeTnode(int info)
{
    Tree *node = (Tree *)malloc(sizeof(Tree));
    if (node == NULL)
    {
        printf("Memory Allocation Failed!\n");
        return NULL;
    }
    node->info = info;
    node->left = NULL;
    node->right = NULL;

    return node;
}

void freetree(Tree *root)
{
    if (root == NULL)
    {
        return;
    }

    freetree(root->left);
    freetree(root->right);
    free(root);
}

void printInorderiter(Tree *root)
{
    // use array based stack here for simplicity - remove this comment - don't use libraires - straight
away ds and implementation
    if (root == NULL)
    {
        return;
    }
    Stacktree *mystack = createStackt(9);
    Tree *current = root;
    printf("Inorder Traversal iterative:\n");
    while (current != NULL || !isEmptyt(mystack))
    {
        while (current != NULL)
        {
            pusht(mystack, current);
            current = current->left;
        }

        current = popt(mystack);
        printf("%d ", current->info);
        current = current->right;
    }

    free(mystack->array);
    free(mystack);
}

Stacktree *createStackt(int n)
{
    Stacktree *s = (Stacktree *)malloc(sizeof(Stacktree));
    s->capacity = n;
    s->top = -1;
    s->array = (Tree **)malloc(s->capacity * sizeof(Tree *));
    return s;
}

Tree *popt(Stacktree *s)
{
    if (isEmptyt(s))
    {
        // Underflow
        printf("Stack is Empty\n");
        return NULL;
    }
    Tree *popped = s->array[s->top];
    s->top--;
    return popped;
}

void pusht(Stacktree *s, Tree *node)
{
    // Overflow (as we do in algo in class)
    if (isFullt(s))
    {
        printf("Stack is Full!\n");
        return;
    }

    s->top++;
    s->array[s->top] = node;
    return;
}

int isFullt(Stacktree *s)
{
    return s->top == s->capacity - 1;
}

int isEmptyt(Stacktree *s)
{
    return s->top == -1;
}

Tree *peekt(Stacktree *s)
{
    if (isEmptyt(s))
    {
        return NULL;
    }
    return s->array[s->top];
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab8> ./inorder
Inorder:
9 7 4 8 2 5 1 6 3
Inorder Traversal iterative:
9 7 4 8 2 5 1 6 3
```

Theory (Preorder Traversal):

Preorder Traversal (Root, Left, Right): In iterative preorder traversal, a stack is used to maintain nodes, starting with the root. The root is processed first, then the right child is pushed, followed by the left child, ensuring left is processed before right.

Time Complexity:

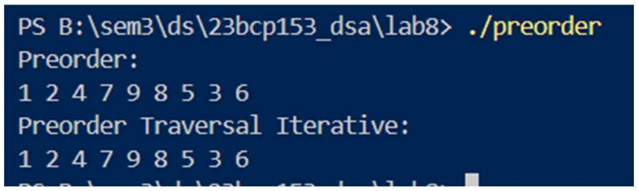
$O(n)$ — Each node is visited once.

Program:

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

// root left right - Preorder

typedef struct Tree {
   int info;
   struct Tree *left;
   struct Tree *right;
} Tree;

typedef struct Stacktree
{
   int top;
   int capacity;
   Tree **array;
} Stacktree;

Tree *makeTnode(int info);
void printPreorder(Tree *root);
void freetree(Tree *root);
void printPreorderiter(Tree *root);
Stacktree *createStackt(int n);
Tree *popt(Stacktree *s);
void pusht(Stacktree *s, Tree *node);
int isFullt(Stacktree *s);
int isEmptyt(Stacktree *s);
Tree *peekt(Stacktree *s);
```

```
int main(void)
{
    Tree *A = makeTnode(1);
    Tree *B = makeTnode(2);
    Tree *C = makeTnode(3);
    Tree *D = makeTnode(4);
    Tree *E = makeTnode(5);
    Tree *F = makeTnode(6);
    Tree *G = makeTnode(7);
    Tree *H = makeTnode(8);
    Tree *I = makeTnode(9);
    A->left = B;
    A->right = C;
    B->left = D;
    B->right = E;
    C->left = F;
    D->left = G;
    D->right = H;
    G->left = I;

    printf("Preorder:\n");
    printPreorder(A);
    printf("\n");
    printPreorderiter(A);
    freetree(A);

    return 0;
}

void printPreorder(Tree *root)
{
    if (root == NULL)
    {
        return;
    }

    printf("%d ", root->info);
    printPreorder(root->left);
    printPreorder(root->right);
}

Tree *makeTnode(int info)
{
    Tree *node = (Tree *)malloc(sizeof(Tree));
    if (node == NULL)
    {
        printf("Memory Allocation Failed!\n");
        return NULL;
    }
    node->info = info;
    node->left = NULL;
    node->right = NULL;

    return node;
}

void freetree(Tree *root)
{
    if (root == NULL)
    {
        return;
    }

    freetree(root->left);
    freetree(root->right);
    free(root);
}

void printPreorderiter(Tree *root)
{
    if (root == NULL)
    {
        return;
    }

    Stacktree *mystack = createStackt(9);
    Tree *current = root;
    printf("Preorder Traversal Iterative:\n");
    pusht(mystack, root);
    while (!isEmptyt(mystack))
    {
        Tree *current = popt(mystack);

        printf("%d ", current->info);

        if (current->right != NULL)
        {
            pusht(mystack, current->right);
        }
        if (current->left != NULL)
        {
            pusht(mystack, current->left);
        }
    }
    printf("\n");

    free(mystack->array);
    free(mystack);
}

Stacktree *createStackt(int n)
{
    Stacktree *s = (Stacktree *)malloc(sizeof(Stacktree));
    s->capacity = n;
    s->top = -1;
    s->array = (Tree **)malloc(s->capacity * sizeof(Tree *));
    return s;
}

Tree *popt(Stacktree *s)
{
    if (isEmptyt(s))
    {
        // Underflow
        printf("Stack is Empty\n");
        return NULL;
    }
    Tree *popped = s->array[s->top];
    s->top--;
    return popped;
}

void pusht(Stacktree *s, Tree *node)
{
    // Overflow (as we do in algo in class)
    if (isFullt(s))
    {
        printf("Stack is Full!\n");
        return;
    }

    s->top++;
    s->array[s->top] = node;
    return;
}

int isFullt(Stacktree *s)
{
    return s->top == s->capacity - 1;
}

int isEmptyt(Stacktree *s)
{
    return s->top == -1;
}

Tree *peekt(Stacktree *s)
{
    if (isEmptyt(s))
    {
        return NULL;
    }
    return s->array[s->top];
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab8> ./preorder
Preorder:
1 2 4 7 9 8 5 3 6
Preorder Traversal Iterative:
1 2 4 7 9 8 5 3 6
```

Theory (Postorder Traversal):

Postorder Traversal (Left, Right, Root): Iterative postorder traversal uses two stacks. The first stack helps to visit nodes in a modified postorder, and the second stack stores the nodes in reverse order to finally print them in correct postorder sequence.

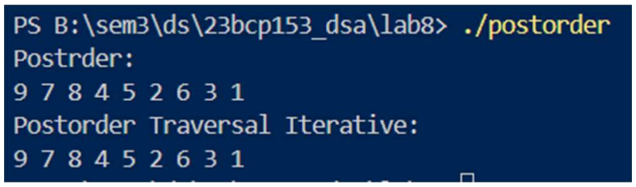
Time Complexity:

$O(n)$ — Nodes are pushed and popped from the stack twice, but still results in linear time.

Program:

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

// left right root - Postorder

typedef struct Tree {
    int info;
    struct Tree *left;
    struct Tree *right;
} Tree;

typedef struct Stacktree
{
    int top;
    int capacity;
    Tree **array;
} Stacktree;

Tree *makeTnode(int info);
void printPostorder(Tree *root);
void freetree(Tree *root);
void printPostorderiter(Tree *root);
Stacktree *createStackt(int n);
Tree *popt(Stacktree *s);
void pusht(Stacktree *s, Tree *node);
int isFullt(Stacktree *s);
int isEmptyt(Stacktree *s);
Tree *peekt(Stacktree *s);

int main(void)
{
    Tree *A = makeTnode(1);
    Tree *B = makeTnode(2);
    Tree *C = makeTnode(3);
    Tree *D = makeTnode(4);
    Tree *E = makeTnode(5);
    Tree *F = makeTnode(6);
    Tree *G = makeTnode(7);
    Tree *H = makeTnode(8);
    Tree *I = makeTnode(9);
    A->left = B;
    A->right = C;
    B->left = D;
    B->right = E;
    C->left = F;
    D->left = G;
    D->right = H;
    G->left = I;

    printf("Postrder:\n");

    printPostorder(A);
    printf("\n");
    printf("Postorder Traversal Iterative:\n");
    printPostorderiter(A);
    printf("\n");
    freetree(A);

    return 0;
}

void printPostorder(Tree *root)
{
    if (root == NULL)
    {
        return;
    }

    printPostorder(root->left);
    printPostorder(root->right);
    printf("%d ", root->info);
}

// Make these below type of dynamic functions later

// Tree *insert()
// {

// }

Tree *makeTnode(int info)
{
    Tree *node = (Tree *)malloc(sizeof(Tree));
    if (node == NULL)
    {
        printf("Memory Allocation Failed!\n");
        return NULL;
    }
    node->info = info;
    node->left = NULL;
    node->right = NULL;

    return node;
}

void freetree(Tree *root)
{
    if (root == NULL)
    {
        return;
    }

    freetree(root->left);
    freetree(root->right);
    free(root);
}

void printPostorderiter(Tree *root)
{
    if (root == NULL)
    {
        return;
    }

    Stacktree *mainstack = createStackt(9);
    Stacktree *outputstack = createStackt(9);

    pusht(mainstack, root);

    while (!isEmptyt(mainstack))
    {
        Tree *current = popt(mainstack);
        pusht(outputstack, current);

        if (current->left != NULL)
        {
            pusht(mainstack, current->left);
        }
        if (current->right != NULL)
        {
            pusht(mainstack, current->right);
        }
    }
    while (!isEmptyt(outputstack))
    {
        Tree *current = popt(outputstack);
        printf("%d ", current->info);
    }

    free(mainstack->array);
    free(mainstack);
    free(outputstack->array);
    free(outputstack);
}

Stacktree *createStackt(int n)
{
    Stacktree *s = (Stacktree *)malloc(sizeof(Stacktree));
    s->capacity = n;
    s->top = -1;
    s->array = (Tree **)malloc(s->capacity * sizeof(Tree *));
    return s;
}

Tree *popt(Stacktree *s)
{
    if (isEmptyt(s))
    {
        // Underflow
        printf("Stack is Empty\n");
        return NULL;
    }
    Tree *popped = s->array[s->top];
    s->top--;
    return popped;
}

void pusht(Stacktree *s, Tree *node)
{
    // Overflow (as we do in algo in class)
    if (isFullt(s))
    {
        printf("Stack is Full!\n");
        return;
    }

    s->top++;
    s->array[s->top] = node;
    return;
}

int isFullt(Stacktree *s)
{
    return s->top == s->capacity - 1;
}

int isEmptyt(Stacktree *s)
{
    return s->top == -1;
}

Tree *peekt(Stacktree *s)
{
    if (isEmptyt(s))
    {
        return NULL;
    }

    return s->array[s->top];
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab8> ./postorder
Postrder:
9 7 8 4 5 2 6 3 1
Postorder Traversal Iterative:
9 7 8 4 5 2 6 3 1
```