**Experiment No. 8: Binary Search Tree Operations**

**Sachin Rathod SE-03**

Roll no.17

**Aim :** **Implementation of Binary Search Tree ADT using Linked List.**

**Objective:**

1) Understand how to implement a BST using a predefined BST ADT.

2) Understand the method of counting the number of nodes of a binary tree.

**Theory:**

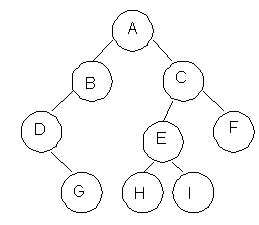
A binary tree is a finite set of elements that is either empty or partitioned into disjoint subsets. In other words node in a binary tree has at most two children and each child node is referred as left or right child.

Traversals in tree can be in one of the three ways : preorder, postorder, inorder.

Preorder Traversal

Here the following strategy is followed in sequence

* Visit the root node R
* Traverse the left subtree of R
* Traverse the right sub tree of R



|  |  |
| --- | --- |
| **Description** | **Output** |
| Visit Root | A |
| Traverse left sub tree – step to B then D | ABD |
| Traverse right sub tree – step to G | ABDG |
| As left subtree is over. Visit root , which is already visited so go for right subtree | ABDGC |
| Traverse the left subtree | ABDGCEH |
| Traverse the right sub tree | ABDGCEHIF |

Inorder Traversal

Here the following strategy is followed in sequence

* Traverse the left subtree of R
* Visit the root node R
* Traverse the right sub tree of R

|  |  |
| --- | --- |
| **Description** | **Output** |
| Start with root and traverse left sub tree from A-B-D | D |
| As D doesn’t have left child visit D and go for right subtree of D which is G so visit this. | DG |
| Backtrack to D and then to B and visit it. | DGB |
| Backtract to A and visit it | DGBA |
| Start with right sub tree from C-E-H and visit H | DGBAH |
| Now traverse through parent of H which is E and then I | DGBAHEI |
| Backtrack to C and visit it and then right subtree of E which is F | DGBAHEICF |

Postorder Traversal

Here the following strategy is followed in sequence

* Traverse the left subtree of R
* Traverse the right sub tree of R
* Visit the root node R

|  |  |
| --- | --- |
| **Description** | **Output** |
| Start with left sub tree from A-B-D and then traverse right sub tree to get G | G |
| Now Backtrack to D and visit it then to B and visit it. | GD |
| Now as the left sub tree is over go for right sub tree | GDB |
| In right sub tree start with leftmost child to visit H followed by I | GDBHI |
| Visit its root as E and then go for right sibling of C as F | GDBHIEF |
| Traverse its root as C | GDBHIEFC |
| Finally a root of tree as A | GDBHIEFCA |

**Algorithm**

Algorithm: PREORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in preorder fashion.

Description : Linked structure of binary tree

* ptr=ROOT
* if ptr!=NULL then

visit(ptr)

PREORDER(LSON(ptr))\

PREORDER(RSON(ptr))

End if

* Stop

Algorithm: INORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in inorder fashion.

Description : Linked structure of binary tree

* ptr=ROOT
* if ptr!=NULL then

INORDER (LSON(ptr))

visit(ptr)

INORDER (RSON(ptr))

End if

* Stop

Algorithm: POSTORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in postorder fashion.

Description : Linked structure of binary tree

* ptr=ROOT
* if ptr!=NULL then

PREORDER(LSON(ptr))

PREORDER(RSON(ptr))

visit(ptr)

End if

* Stop

**Code:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*leftChild, \*rightChild;

};

struct node \*root = NULL;

struct node \*newNode(int item){

struct node \*temp = (struct node \*)malloc(sizeof(struct node));

temp->data = item;

temp->leftChild = temp->rightChild = NULL;

return temp;

}

void insert(int data){

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

//if tree is empty

if(root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while(1) {

parent = current;

//go to left of the tree

if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode;

return;

}

}//go to right of the tree

else {

current = current->rightChild;

//insert to the right

if(current == NULL) {

parent->rightChild = tempNode;

return;

}

}

}

}

}

struct node\* search(int data){

struct node \*current = root;

printf("\n\nVisiting elements: ");

while(current->data != data) {

if(current != NULL) {

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

//go to left tree

if(current->data > data) {

current = current->leftChild;

}//else go to right tree

else {

current = current->rightChild;

}

//not found

if(current == NULL) {

return NULL;

}

}

}

return current;

}

// Inorder Traversal

void inorder(struct node \*root){

if (root != NULL) {

inorder(root->leftChild);

printf("%d -> ", root->data);

inorder(root->rightChild);

}

}

// Preorder Traversal

void preorder(struct node \*root){

if (root != NULL) {

printf("%d -> ", root->data);

preorder(root->leftChild);

preorder(root->rightChild);

}

}

// Postorder Traversal

void postorder(struct node \*root){

if (root != NULL) {

printf("%d -> ", root->data);

postorder(root->leftChild);

postorder(root->rightChild);

}

}

int main(){

insert(10);

insert(14);

insert(19);

insert(26);

insert(27);

insert(31);

insert(33);

insert(35);

insert(42);

insert(44);

printf("Insertion done\n");

printf("\nPreorder Traversal: ");

preorder(root);

printf("\nInorder Traversal: ");

inorder(root);

printf("\nPostorder Traversal: ");

postorder(root);

struct node\* k;

k = search(35);

if(k != NULL)

printf("\nElement %d found", k->data);

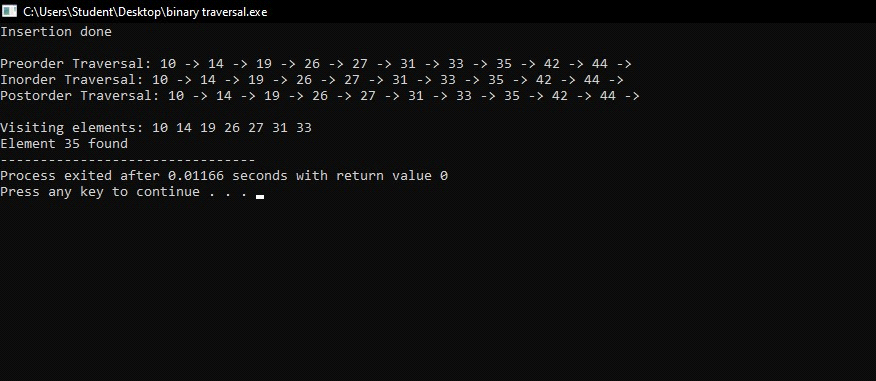
else

printf("\nElement not found");

return 0;

}

**Output**:



**Conclusion:** Binary trees have many applications in computer science, including data storage and retrieval, expression evaluation, network routing, and game AI.