**Batch: A3 Roll No.: 1911034**

**Experiment / assignment / tutorial No. 8**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title:**  Implementation of BST & Binary tree traversal techniques. |

**Objective:** To Understand and Implement Binary Search Tree, Preorder, Postorder and Inorder Traversal Techniques.

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **CO** | **Outcome** |
| 1 | Explain the different data structures used in problem solving |

**Books/ Journals/ Websites referred:**

1. *Fundamentals Of Data Structures In C –* Ellis Horowitz, Satraj Sahni, Susan Anderson-Fred
2. *An Introduction to data structures with applications –* Jean Paul Tremblay,

Paul G. Sorenson

1. *Data Structures A Pseudo Approach with C –* Richard F. Gilberg & Behrouz A. Forouzan
2. <https://www.geeksforgeeks.org/binary-tree-data-structure/>
3. <https://www.thecrazyprogrammer.com/2015/03/c-program-for-binary-search-tree-insertion.html>

**Abstract**:

**A tree** is a non- linear data structure used to represent hierarchical relationship existing among several data items. It is a finite set of one or more data items such that, there is a special data item called the root of the tree. Its remaining data items are partitioned into number of mutually exclusive subsets, each of which is itself a tree, and they are called subtrees.

**A binary tree** is a finite set of nodes. It is either empty or It consists a node called root with two disjoint binary trees-Left subtree, Right subtree. The Maximum degree of any node is 2

**A Binary Search Tree** is a node-based binary tree data structure in which the left subtree of a node contains only nodes with keys lesser than the node’s key. The right subtree of a node contains only nodes with keys greater than the node’s key. The left and right subtree each must also be a binary search tree.

**Related Theory: -**

**Preorder Traversal of BST**

**Algorithm for Preorder :**

**PREORDER(Tree, Data)**

Step1: Print(PREORDER(Tree, Data))

Step2: Print(PREORDER(Tree->Left ,Data))

Step3: Print(PREORDER(Tree->Right, Data))

(This algorithm recursively prints the root node of each subtree, visits it’s left subtree , in the recursive order of root-left-right, then visits it’s right subtree in the recursive order of root-left-right)

Uses : Preorder Traversal is used to create a copy of the tree. It is also used to get the prefix expression from the tree.

**Post order Traversal of BST**

**Algorithm for Post order :**

**POSTORDER(Tree, Data)**

Step1: Print(POSTORDER(Tree->Left, Data))

Step2: Print(POSTORDER(Tree->Right ,Data))

Step3: Print(POSTORDER(Tree, Data))

(This algorithm recursively visits the left sub child of each parent, until it reaches a node which has no left sub child. It prints that element, then recursively visits the right subtree in the order of left-right-root, and finally visits the root node.)

Uses : Post order traversal is used to delete the tree, it is also used to get the postfix expression of the Tree.

**In order Traversal of BST**

**Algorithm for In order Traversal :**

Step1: Print(INORDER(Tree->Left, Data))

Step2: Print(INORDER(Tree ,Data))

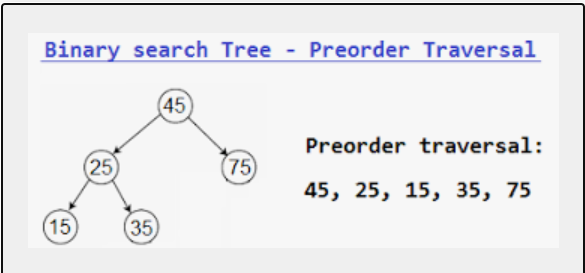
Step3: Print(INORDER(Tree->Right, Data))

(This algorithm recursively visits the left sub child of each parent, until it reaches a node which has no left sub child. It prints that element , then prints it’s parent then , recursively visits it’s right child in the order , left-root-right)

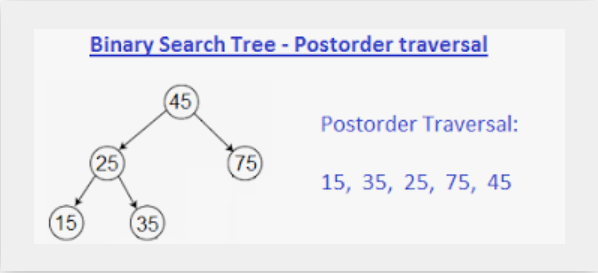
Uses : In order Traversal can be used to print the elements of a tree in the non-decreasing order.

**Diagram for :**

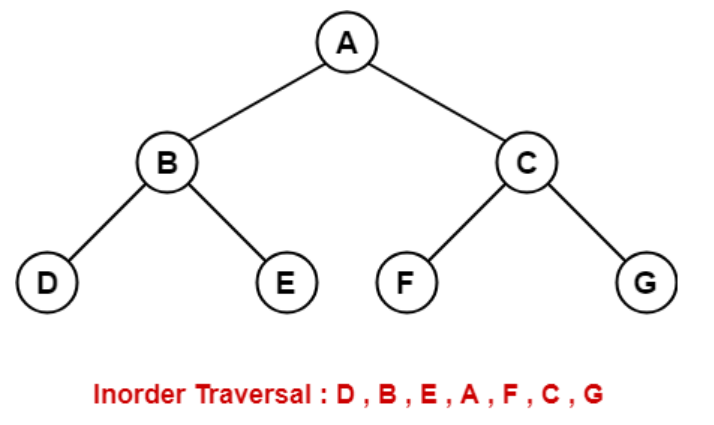
**Preorder Traversal of BST**



**Postorder Traversal of BST**



**Inorder Traversal of BST**



**Algorithm for Implementation of BST & Binary tree traversal techniques:**

**Algorithm for Creation of a Binary Search Tree by Insertion of Elements (InsertNode Function):**

let ‘root’ represent the pointer to root node of the tree, every node, contains a DATA part that contains the data, the LEFT pointer which points to the left child and RIGHT pointer which points to the right child of the node.

Step1: accept the value to inserted from the user (val), store it in a node ‘temp’.Set temp->left = NULL and temp->right = NULL.

Step2: If root=NULL, then set root=temp. set temp->left= NULL and

temp->right=NULL

Step3: else if root! = NULL, initialize a pointer p = root, and t1.

Step4 : repeat steps 5 to 7 while p!=NULL

Step5: set t1=p

Step6: if val > p->data, set p = p->right.

Step7: else if val < p->data, set p = p->left.

[END OF WHILE]

Step7: At this step, t1 holds the value of the node which will be the parent of the node to be inserted. If t1->data < val , set t1->right = temp, else set t1->left = temp.

**Algorithm to search for an element in a BST**

Step1: Accept the value to be searched from the user, and pass that value and the root node to the function void search (tree , data)

Step2: initialize two pointers t1, and p of type struct node (which will be of the same type as that of an individual node in a tree).

Step3: ‘p’ will be used to traverse the tree, while t1, holds the value of a particular node visited by p.

Step4: repeat steps 5to 6 , while (p!=NULL) OR while (p->data !=val)

Step5: set t1 = p

Step6 if (p-> data) > val , set p= p->left , else set p= p->right.

Step7: if the value is found , print(‘found’) , else if p==NULL, print(‘value exists’)

END

**(Recursive Algorithms to find the Inorder , Preorder and Postorder traversals have been given above)**

**Implementation Details:**

1. **Enlist all the Steps followed and various options explored.**

**Steps followed :**

1. We have defined the nodes of the tree , in the form of a struct , with three values , one , which holds the data of the node, and two other node pointers to it’s left and right subtrees.
2. We then ask the user to enter the number of elements, and finally the elements , which are then arranged in such a way that the left subtree of each node has a value less than the parent , and the right subtree of each node has a value greater than the parent.
3. We have then provided the user a menu based program, which allows the user to print the Inorder , Preorder or PostOrder traversals according to recursive algorithms as described above
4. We have finally provided an option to search for an element in a BST , which utilizes two pointers , which are of type node , which can be used for searching for an element in the tree**.**

**Options explored :**

1. In printing tree traversals , it is possible to print the elements of a tree , either iteratively or recursively , but here we have provided recursive based implementation
2. We have provided the option to the user to implement the program in a menu driven way , which allows for dynamic insertion or searching for the elements too.

**Assumptions made for Input:**

1. It is assumed , that the value held by each node will be an integer.
2. As there is no limit to the allocation of nodes (apart from free memory available), it is assumed that user can enter any number of nodes as they wish.

**Built-In Functions Used:**

1. Malloc() : used to dynamically allocate memory to each node as and when required.

**Program source code for Implementation of BST & Binary tree traversal techniques :**

#include <stdio.h>

#include<stdlib.h>

struct tnode

{

int data;

struct tnode\*left;

struct tnode\*right;

};

struct tnode\*insertnode(int val, struct tnode\*root)

{

struct tnode\* p=root;

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

temp->data=val;

temp->right=NULL;

temp->left = NULL;

struct tnode\*t1;

if(root==NULL)//when the root is to be created

{

root=temp;

}

else if(root!=NULL)

{

while(p!=NULL)

{t1=p;

if(p->data>val)

{

p=p->left;

}

else if(p->data<val)

{

p=p->right;

}

}

if(val>t1->data)

{

t1->right=temp;

}

else{

t1->left=temp;

}

}

return root;

}

void inordertraversal(struct tnode\*root)

{

struct tnode \*p=root;

if(p!=NULL)

{

inordertraversal(p->left);

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

inordertraversal(p->right);

}

}

void preordertraversal(struct tnode\*root)

{

struct tnode \*p=root;

if(p!=NULL)

{

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

preordertraversal(p->left);

preordertraversal(p->right);

}

}

void postordertraversal(struct tnode\*root)

{

struct tnode \*p=root;

if(p!=NULL)

{

preordertraversal(p->left);

preordertraversal(p->right);

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

}

}

void search(struct tnode\*root,int val)

{

struct tnode\*p=root;

while(p!=NULL)

{

if(val==root->data)

{

printf("The value to be found exists\n");

return;

}

else if(val>root->data)

{

search(p->right,val);

return;

}

else if(val<root->data)

{

search(p->left,val);

return;

}

}

if(p==NULL) //in case we reach null and value does not exist

{

printf("value does not exist\n");

}

}

int main(void) {

printf("Hello World\n");

struct tnode\*root = NULL;

int ch,val;

do{

printf("enter any of the following choices\n");

printf("Enter 1 to insert a node\n 2 to display the inorder traversal\n3 to display the preorder traversal\n,4 to display the postorder traversal\n, 5 to check whther an element exists or not\n");

scanf("%d",&ch);

switch(ch)

{

case 1:

{

printf("Enter the value\n");

scanf("%d",&val);

root=insertnode(val,root);

}break;

case 2:

{

inordertraversal(root);

printf("\n");

}break;

case 3:

{

preordertraversal(root);

printf("\n");

}break;

case 4:

{

postordertraversal(root);

printf("\n");

}break;

case 5:

{

printf("Enter the value\n");

scanf("%d",&val);

search(root,val);

}}}

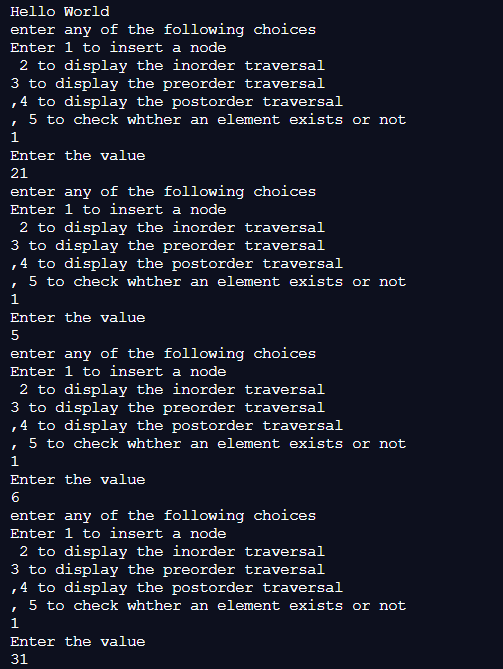
while(ch!=-1);

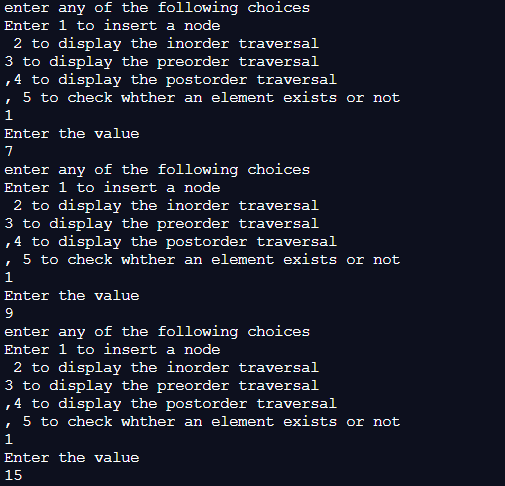
return 0;

}

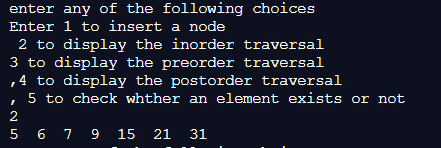
**Output Screenshots for Each Operation:**

**Inserting elements in a Binary Search tree:**

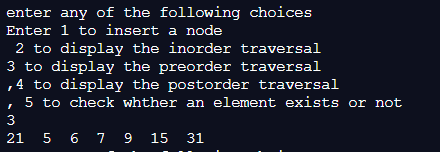




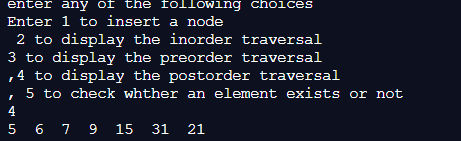
**Displaying the In order Traversal of the tree created :**



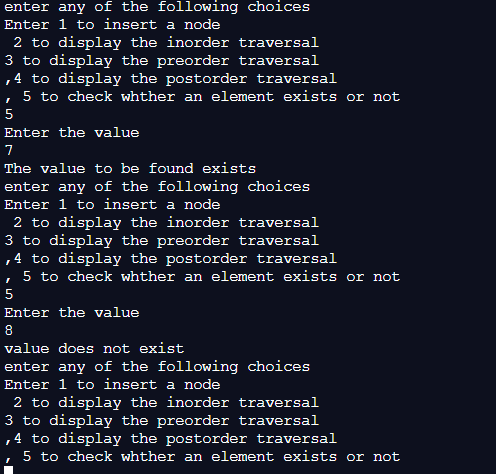
**Displaying the Pre order traversal of tree created :**



**Displaying the Post order Traversal of the tree created :**



**Checking whether a particular element exists or not :**



**Explain the Importance of the approach followed by you**

.

1. In this approach , we have designed a menu driven program , which not only allows the user to enter , search or display the traversals in any order , but it is also successful in storing the values of the nodes , such that the previous order of the BST is not disturbed.
2. We have also learnt about the various tree traversals, and how they can be implemented recursively as well as iteratively , which may be useful for different applications.
3. Finally ,we have implemented the concept of the BST as a non linear data structure which allows us to store the elements in a particular order , according to their values of the elements , so that the searching time for any element is reduced , by nearly half , as compared to linear data structures. (The **worst case** searching time for a tree is O(N) which is similar to that of a linear data structure like arrays).

**Conclusion:-**

**Through this experiment, we have been able to perform the following:**

1. We have been able to understand the concept of Tree as a Non Linear Data Structure, and of Binary Search Tree as a Tree that stores elements which makes it easier to search for them.
2. We have implemented the Binary Search Tree , by inserting the nodes in a way that does not disturb the relative order of the BST , as well as performed the three traversals , and have understood how to search for any element in the BST**.**

**PostLab Questions:**

1. **Illustrate 2 Applications of Trees.**

**Applications of Tree Data Structure are as follows:**

1. Binary Search Tree is a tree which allows for fast search , insert , delete on a sorted data. It also allows us to find a closest item. For this reason , the trees can be used to store hierarchical data like folder structure , organization structure or XML/HTML files.
2. The B-Trees are self balancing and hence the average and worst complexities is logarithmic. We opt for these when the data is too huge to fit in main memory. These structures are used in database indexing and help in faster operations on disk.
3. **Compare and Contrast between B Tree and B+ Tree?**

|  |  |
| --- | --- |
| **B Tree** | **B+ Tree** |
| 1. A B tree is an organizational structure for information storage and retrieval in the form of a tree in which all terminal nodes are at the same distance from the base, and all non-terminal nodes have between n and 2 n sub-trees or pointers (where n is an integer). | B+ tree is an n-array tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children. |
| In a B tree, search keys and data stored in internal or leaf nodes. | In a B+ tree, data stored only in leaf nodes. |
| The leaf nodes of the three store pointers to records rather than actual records. | The leaf nodes of the tree stores the actual record rather than pointers to records. |
| Here in B tree the search is not that easy as compared to a B+ tree. | Here in B+ tree the searching becomes easy. |
| It might have fewer nodes compared to B+ tree as each node will have data. | Automatically Adjust the nodes to fit the new record. Similarly it re-organizes the nodes in the case of delete, if required. Hence it does not alter the definition of B+ tree. |