-

Batch: B2	**Roll No.: 16010122151**
Experiment / assignment / tutorial No. 7	
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
3. *Data Structures A Pseudo Approach with C* – Richard F. Gilberg & Behrouz A. Forouzan
4. https://www.geeksforgeeks.org/binary-tree-data-structure/
5. https://www.thecrazyprogrammer.com/2015/03/c-program-for-binary-search-tree-insertion.html

GEEKFORGEEKS.COM

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

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

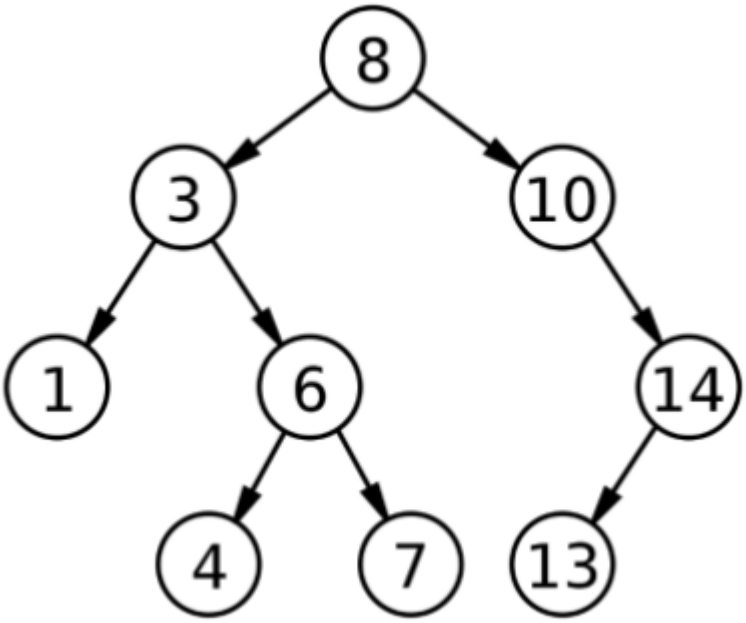
Postorder Traversal of BST

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

Inorder Traversal of BST

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Diagram for traversal :



Preorder Traversal of BST: 8 3 1 6 4 7 10 14 13

Postorder Traversal of BST: 1 3 4 6 7 8 10 13 14

Inorder Traversal of BST: 1 4 7 6 3 13 14 10 8

Algorithm for Implementation of BST & Binary tree traversal techniques:

Search in BST:

1. Start from the root.

2. Compare the searching element with root, if less than root, then recurse for left, else recurse for right.

3. If the element to search is found anywhere, return true, else return false.

Insertion in BST:

1. Start from the root.

2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.

3. After reaching the end, just insert that node at left (if less than current) else right.

Deletion in BST:

1. Search the node that to be deleted in BST.

2. If node has left or right subtree then find the inorder predecessor or inorder successor respectively (call it new node) and replace the node data value with its data. Then make recursive call to the delete function for that new node.

3 Else delete the node and assign the parent's child pointer(which is node) to NULL.

Implementation Details:

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

A structure called Node is created which has two struct pointers (left and right) and 1 int variable (data).

A class called BST is created with functions for insertion, deletion and searching.

For insertion in BST: -

• If root is null create new node and assign the data to it.

• Else traverse in tree according to the data of node.

For searching: -

• If node is null then the node with required data isn't there in the BST.

• Else traverse in tree according to the data of node.

For deletion: -

• Search the node to be deleted.

• If the node has left or right child then replace its data value with inorder successor(if right child is present) or predecessor of current node. Then search of that node in the corresponding subtree.

• Else delete the node and assign its parents pointer (which points to current node) as NULL.

An object of BST class is created and a menu driven program is made for the user to choose an operation.

Assumptions made for Input:

The value of all nodes are integers and |value|<=INTMAX

Built-In Functions Used:

void Inorder(node *root)

void preorder(node *root)

void postorder(node *root)

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

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

typedef struct bstnode {
    int data;
    struct bstnode* left;
    struct bstnode* right;
} node;

// Inserting a node
```

```
void Insert(node** root, int n) {
   node* temp, * p, * parent;
   temp = (node*)malloc(sizeof(node));
   if (temp == NULL)
      printf("Out of memory\n");
   else {
      temp->data = n;
      temp->left = NULL;
      temp->right = NULL;
      p = (*root);
      if ((*root) == NULL)
         (*root) = temp;
      else {
         parent = NULL;
         p = (*root);
         while (p != NULL) {
            parent = p;
            if (p->data > n)
               p = p->left;
            else
               p = p->right;
         }
         if (parent->data > n)
            parent->left = temp;
         else
```

```
            parent->right = temp;
        }
    }
}

// Display Inorder
// Left Root Right
void Inorder(node* root) {
    if (root != NULL) {
        Inorder(root->left);
        printf("%d\t", root->data);
        Inorder(root->right);
    }
}

// Display Preorder
// Root Left Right
void preorder(node* root) {
    if (root != NULL) {
        printf("%d\t", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

```
// Display Postorder
// Left Right Root
void postorder(node* root) {
   if (root != NULL) {
      postorder(root->left);
      postorder(root->right);
      printf("%d\t", root->data);
   }
}

// Search for an element
int search(node* root, int n) {
   if (root == NULL)
      return 0;
   else if (root->data == n)
      return 1;
   else if (root->data > n)
      return search(root->left, n);
   else
      return search(root->right, n);
}

int main() {
   node* root = NULL;
   int op = 0, n, m;
```

```
while (op != 6) {
    printf("\n1. Insert a node.\n2. Display in Inorder.\n3. Display in Preorder.\n4. Display in Postorder.\n5. Search for an element\n6. Exit\n");
    scanf("%d", &op);

    switch (op) {
    case 1:
        printf("Enter the number of elements to be inserted: ");
        scanf("%d", &n);
        for (int i = 0; i < n; i++) {
            printf("Enter data: ");
            scanf("%d", &m);
            Insert(&root, m);
        }
        break;
    case 2:
        Inorder(root);
        break;
    case 3:
        preorder(root);
        break;
    case 4:
        postorder(root);
        break;
    case 5:
```

```
            printf("Enter element to be searched: ");
            scanf("%d", &m);
            int flag = search(root, m);
            if (flag == 1)
               printf("Element exists\n");
            if (flag == 0)
               printf("Element does not exist\n");
            break;
         case 6:
            exit(1);
         }
      }
}
```

Output Screenshots for Each Operation:

Output Clear

```
/tmp/6sggvFNR7p.o
1. Insert a node.
2. Display in Inorder.
3. Display in Preorder.
4. Display in Postorder.
5. Search for an element
6. Exit
1
Enter the number of elements to be inserted: 5
Enter data: 12
Enter data: 42
Enter data: 6
Enter data: 8
Enter data: 91
1. Insert a node.
2. Display in Inorder.
3. Display in Preorder.
4. Display in Postorder.
5. Search for an element
6. Exit
2
6   8   12  42  91
1. Insert a node.
2. Display in Inorder.
3. Display in Preorder.
4. Display in Postorder.
5. Search for an element
6. Exit
3
12  6   8   42  91
1. Insert a node.
2. Display in Inorder.
3. Display in Preorder.
4. Display in Postorder.
5. Search for an element
6. Exit
4
8   6   91  42  12
1. Insert a node.
2. Display in Inorder.
3. Display in Preorder.
4. Display in Postorder.
5. Search for an element
6. Exit
5
Enter element to be searched: 16
Element does not exist

1. Insert a node.
2. Display in Inorder.
3. Display in Preorder.
4. Display in Postorder.
5. Search for an element
6. Exit
5
Enter element to be searched: 91
Element exists
```

Explain the Importance of the approach followed by you:

1.Data Structure:

Binary Search Trees are fundamental data structures used in computer science and programming. They provide efficient methods for insertion, deletion, and retrieval of elements. This code provides a practical example of working with BSTs.

2.Search Efficiency:

BSTs have efficient search operations. In the best-case scenario (a balanced tree), the time complexity of searching for an element is O(log n), which is significantly faster than searching in a linear data structure like an array (O(n)).

3.Insertion and Deletion:

The code demonstrates how to insert nodes into a BST, ensuring that the tree remains balanced. Balancing is crucial for maintaining the efficient search times. It also shows the basics of the binary search tree properties.

4.Traversal Algorithms:

The code includes three common tree traversal algorithms - Inorder, Preorder, and Postorder. These are essential techniques for processing the elements of a tree in a specific order, and they find applications in various tree-based algorithms.

5.Dynamic Memory Allocation:

It demonstrates dynamic memory allocation for creating tree nodes using malloc. Understanding memory management is crucial for writing efficient and reliable code.

6.User Interaction:

The menu-driven interface allows a user to interact with the BST, demonstrating how to create user-friendly programs. Such interactive programs are often used in real-world applications.

7.Error Handling:

The code includes error handling for scenarios where memory allocation fails. Handling errors gracefully is an important aspect of software development.

8.Searching for Elements:

It shows how to search for elements within the tree, which can be useful in a wide range of applications, including database systems and information retrieval.

9.Modularity:

The code is modular, with separate functions for insertion, traversal, and search. This promotes code organization and reusability.

10.Teaching and Learning:

This code can serve as a teaching tool for those learning about data structures, algorithms, and programming in the C language.

.

Conclusion: In this experiment we learnt about various operation and traversals in BST.

PostLab Questions:

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

File System Representation:

Trees are commonly used to represent file systems in operating systems. In this context, a tree structure can be used to represent directories (folders) and files. Each node in the tree represents either a directory or a file, and the hierarchy of directories is established through parent-child relationships. Here's how it works:

Each node in the tree represents a directory or a file.
Directories are represented as internal (non-leaf) nodes, while files are represented as leaf nodes.
Parent-child relationships between nodes represent the containment of files and directories within one another.
This tree structure allows for efficient file and directory management, easy navigation, and searching within a file system. It also enables operations like creating, deleting, moving, and copying files and directories.

Database Indexing:

Trees, specifically balanced search trees like Binary Search Trees (BSTs) and B-Trees, are used for indexing in database management systems. In this context, a tree structure allows for efficient data retrieval and storage. Here's how it works:

Each node in the tree represents a data record or an index entry.
Nodes are organized in a hierarchical structure based on the values of the data they represent.
The tree structure ensures that data is sorted and stored in a way that enables fast searching, insertion, and deletion operations.

2) Compare and Contrast between B Tree and B+ Tree?

Basis of Comparison	B tree	B+ tree
Pointers	All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
Search	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and more accurate.
Redundant Keys	No duplicate of keys is maintained in the tree.	Duplicate of keys are maintained and all nodes are present at the leaf.
Insertion	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
Deletion	Deletion of the internal node is very complex and the tree has to undergo a lot of transformations.	Deletion of any node is easy because all node are found at leaf.
Leaf Nodes	Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.
Access	Sequential access to nodes is not possible	Sequential access is possible just like linked list
Height	For a particular number nodes height is larger	Height is lesser than B tree for the same number of nodes
Application	B-Trees used in Databases, Search engines	B+ Trees used in Multilevel Indexing, Database indexing
Number of Nodes	Number of nodes at any intermediary level 'l' is 2^l.	Each intermediary node can have n/2 to n children.