**Batch: B-1 Roll No.: 16010122104**

**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** |
| 2 | Apply linear and non-linear data structure in application development |

**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**

* Step 1: Visit the root node.
* Step 2: Traverse the left sub tree, i.e., traverse recursively.
* Step 3: Traverse the right sub tree, i.e., traverse recursively.
* Step 4: Repeat steps 2 and 3 until all the nodes are visited.

**Postorder Traversal of BST**

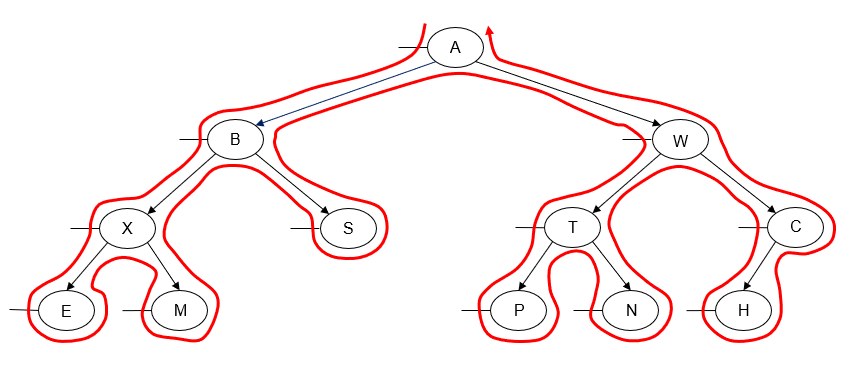
* Step 1: Traverse the left subtree, i.e., traverse recursively.
* Step 2: Traverse the right subtree, i.e., traverse recursively.
* Step 3: Finally, visit the root node.

**Inorder Traversal of BST**

* Step 1: The first thing to do is to traverse all the nodes of the left sub tree in the manner where the first left node is visited then the main and then the right node.
* Step 2: Now, visit the root node of the tree.
* Step 3: It’s time to visit the right subtree after left subtree traversing and root visiting.

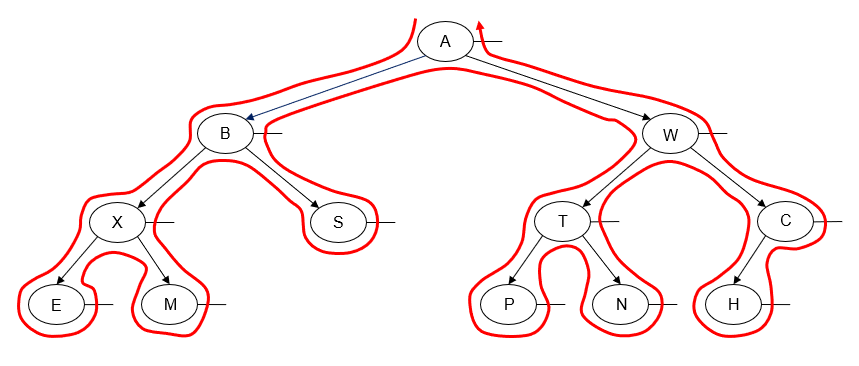
**Diagram for :**

**Preorder Traversal of BST**



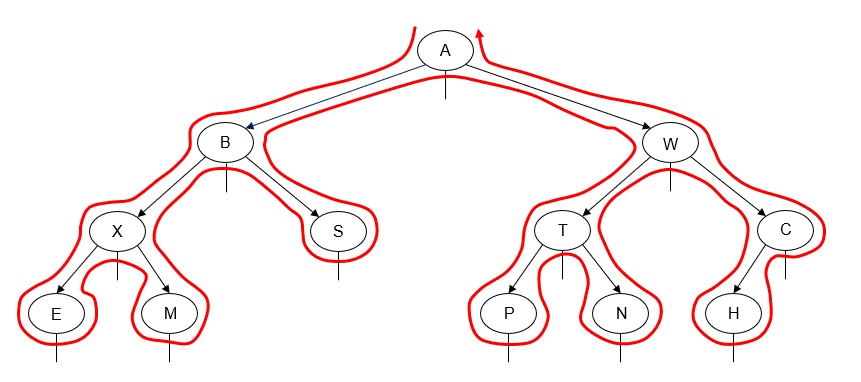
A B X E M S W T P N C H

**Postorder Traversal of BST**



E M X S B P N T H C W A

**Inorder Traversal of BST**

 E X M B S A P T N W H C

**Algorithm for Implementation of BST (insertion and Search) & Binary tree traversal techniques:**

Structure Node:

Data: integer

Left: Node

Right: Node

Initialize BST:

Root = null

**Insert(Node root, int data):**

If root is null:

Create a new node with data

Return the new node

If data < root.data:

root.left = Insert(root.left, data)

Else:

root.right = Insert(root.right, data)

Return root

**Search(Node root, int key):**

If root is null or root.data is equal to key:

Return root

If key < root.data:

Return Search(root.left, key)

Else:

Return Search(root.right, key)

**InOrderTraversal(Node root):**

If root is not null:

InOrderTraversal(root.left)

Print root.data

InOrderTraversal(root.right)

**PreOrderTraversal(Node root):**

If root is not null:

Print root.data

PreOrderTraversal(root.left)

PreOrderTraversal(root.right)

**PostOrderTraversal(Node root):**

If root is not null:

PostOrderTraversal(root.left)

PostOrderTraversal(root.right)

Print root.data

**Program source code for Implementation of BST (insertion and Search) & Binary tree traversal techniques :**

#include <stdio.h>

#include <stdlib.h>

// Node structure for BST

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new BST node

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

// Function to insert a node into BST

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

} else if (data > root->data) {

root->right = insert(root->right, data);

}

return root;

}

// Function to search for a node in BST

struct Node\* search(struct Node\* root, int key) {

if (root == NULL || root->data == key) {

return root;

}

if (key < root->data) {

return search(root->left, key);

} else {

return search(root->right, key);

}

}

// In-order traversal of BST

void inOrderTraversal(struct Node\* root) {

if (root != NULL) {

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

// Pre-order traversal of BST

void preOrderTraversal(struct Node\* root) {

if (root != NULL) {

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

preOrderTraversal(root->left);

preOrderTraversal(root->right);

}

}

// Post-order traversal of BST

void postOrderTraversal(struct Node\* root) {

if (root != NULL) {

postOrderTraversal(root->left);

postOrderTraversal(root->right);

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

}

}

int main() {

struct Node\* root = NULL;

int choice, data;

int n;

while (1) {

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

printf("1. Insert a node\n");

printf("2. Search for a node\n");

printf("3. In-order Traversal\n");

printf("4. Pre-order Traversal\n");

printf("5. Post-order Traversal\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the number of nodes you want to insert: ");

scanf("%d", &n);

printf("Enter data for each node:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &data);

root = insert(root, data);

}

printf("%d nodes inserted successfully.\n", n);

break;

case 2:

printf("Enter data to search: ");

scanf("%d", &data);

if (search(root, data) != NULL) {

printf("Node found in the BST.\n");

} else {

printf("Node not found in the BST.\n");

}

break;

case 3:

printf("In-order Traversal: ");

inOrderTraversal(root);

printf("\n");

break;

case 4:

printf("Pre-order Traversal: ");

preOrderTraversal(root);

printf("\n");

break;

case 5:

printf("Post-order Traversal: ");

postOrderTraversal(root);

printf("\n");

break;

case 6:

exit(0);

default:

printf("Invalid choice! Please enter a valid option.\n");

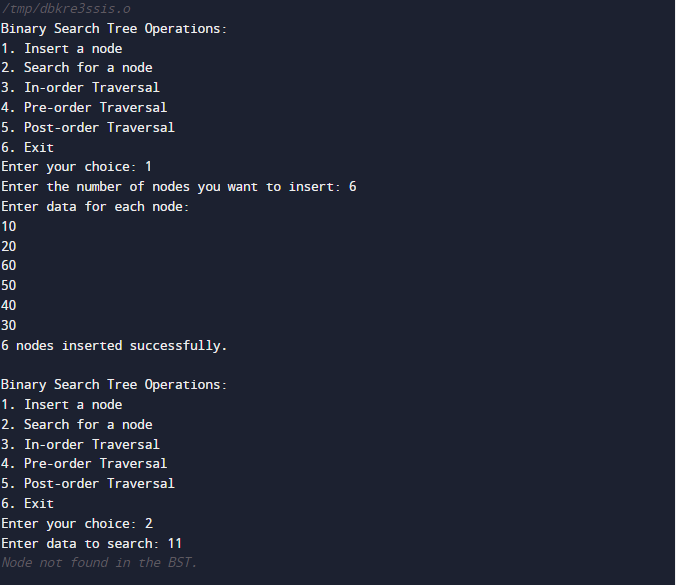
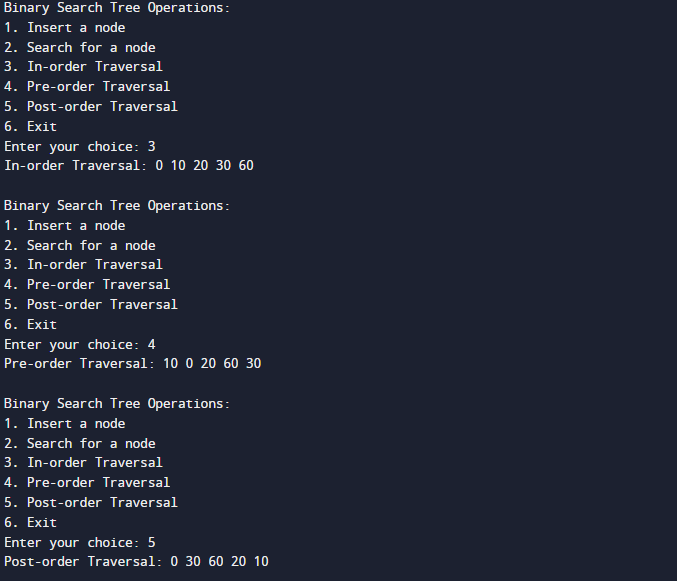
}

}

return 0;

}

**Output Screenshots for Each Operation:**





**Conclusion:-**

We implemented Binary Search Tree, Preorder, Postorder and Inorder Traversal Techniques.

**PostLab Questions:**

1. **Illustrate 2 Applications of Trees.**
2. File System Directory Structure:
   * Example: The directory structure in your computer's file system.
   * How Trees Are Used: In a file system, directories (or folders) and files are organized hierarchically. The top-level directory is the root, and beneath it, there can be multiple subdirectories and files. This hierarchical structure can be effectively represented as a tree, where each node represents a directory or file, and edges represent the containment relationship. Trees make it easy to navigate, search, and manage files and directories within the file system.
3. Binary Search Trees (BSTs) for Searching and Sorting:
   * Example: Database indexing, phone books, and dictionary applications.
   * How Trees Are Used: Binary Search Trees are a specific type of tree data structure that is used for efficient searching, sorting, and retrieval of data. In a BST, each node has at most two children, and data is organized such that elements less than the current node are in the left subtree, and elements greater are in the right subtree. This hierarchical organization allows for fast searching (O(log n) on average) and sorting (in-order traversal) of data. BSTs are used in various applications where efficient data retrieval is crucial, such as database management systems and dictionary applications.
4. **Compare and Contrast between B Tree and B+ Tree.**

B-trees and B+ trees are both tree data structures used for indexing and organizing data, primarily in databases and file systems. While they share some similarities, they have distinct differences in terms of structure and use cases. Here's a comparison and contrast between B-trees and B+ trees:

1. Structure:

* B-Tree:
  + Each node in a B-tree can have multiple child nodes (often referred to as branching factor or order).
  + Both keys and data can be stored in internal nodes.
  + Leaf nodes contain data pointers or actual data.
* B+ Tree:
  + In a B+ tree, only data is stored in leaf nodes.
  + All internal nodes only contain keys and are used for routing to the appropriate leaf node.
  + B+ trees typically have a higher fanout (more children per node) compared to B-trees.

2. Search Operations:

* B-Tree:
  + B-trees are well-suited for systems where data access time is critical and where data can be found in internal nodes, which may reduce the number of disk accesses.
  + B-tree nodes can contain data, which can be beneficial when data access involves multiple fields.
* B+ Tree:
  + B+ trees are optimized for range queries and sequential access because they store data only in leaf nodes.
  + Range queries are efficient because all data lies in the leaves, which form a linked list.

3. Insertion and Deletion:

* B-Tree:
  + B-trees are less efficient than B+ trees for insertions and deletions because a modification may propagate up to the root node.
* B+ Tree:
  + B+ trees are preferred when insertions and deletions are frequent because they only affect the leaf nodes. Leaf node splits or merges don't affect the internal structure.

4. Use Cases:

* B-Tree:
  + B-trees are used in databases, file systems, and filesystem indexing structures where data access speed is crucial.
  + They are suitable for databases with high insert and delete operations as well.
* B+ Tree:
  + B+ trees are commonly used in databases and file systems where range queries and sequential access are essential.
  + They are particularly well-suited for databases with frequent insertions and deletions.

5. Leaf Node Structure:

* B-Tree:
  + In B-trees, data and keys may be interspersed in leaf nodes.
* B+ Tree:
  + B+ trees have a clear separation between keys in internal nodes and data in leaf nodes, which simplifies node splitting and merging.

6. Memory Usage:

* B-Tree:
  + B-trees might use more memory due to data storage in internal nodes.
* B+ Tree:
  + B+ trees tend to use memory more efficiently because only keys are stored in internal nodes.

In summary, B-trees are optimized for scenarios where data access time is critical and data is frequently found in internal nodes. B+ trees, on the other hand, excel in scenarios requiring efficient range queries, sequential access, and high insert/delete operations. The choice between them depends on the specific use case and the trade-offs between access patterns and data modification requirements.