Batch: C1	**Roll. No.: 16010122257**
Experiment:7	
Grade: AA / AB / BB / BC / CC / CD /DD	

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

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

At first visit the root then traverse left subtree and then traverse the right subtree.

Follow the below steps to implement the idea:

Visit the root and print the data.
Traverse left subtree
Traverse the right subtree

Postorder Traversal of BST

At first traverse left subtree then traverse the right subtree and then visit the root.

Follow the below steps to implement the idea:

Traverse left subtree
Traverse the right subtree
Visit the root and print the data.

Inorder Traversal of BST

At first traverse left subtree then visit the root and then traverse the right subtree.

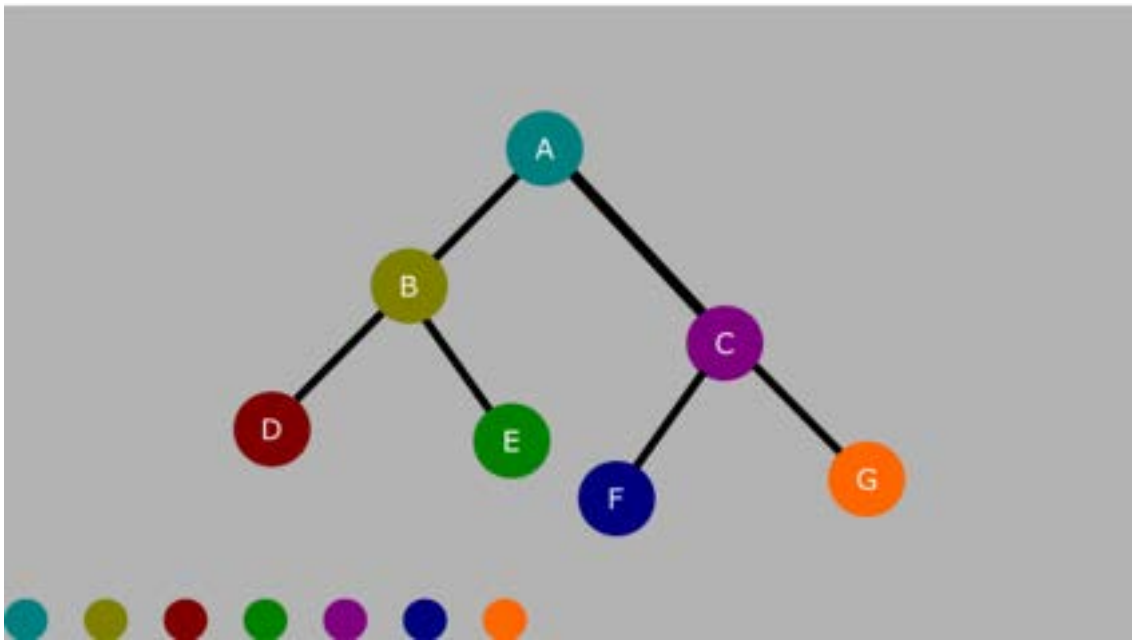
Follow the below steps to implement the idea:

Traverse left subtree
Visit the root and print the data.
Traverse the right subtree
The inorder traversal of the BST gives the values of the nodes in sorted order. To get the decreasing order visit the right, root, and left subtree.

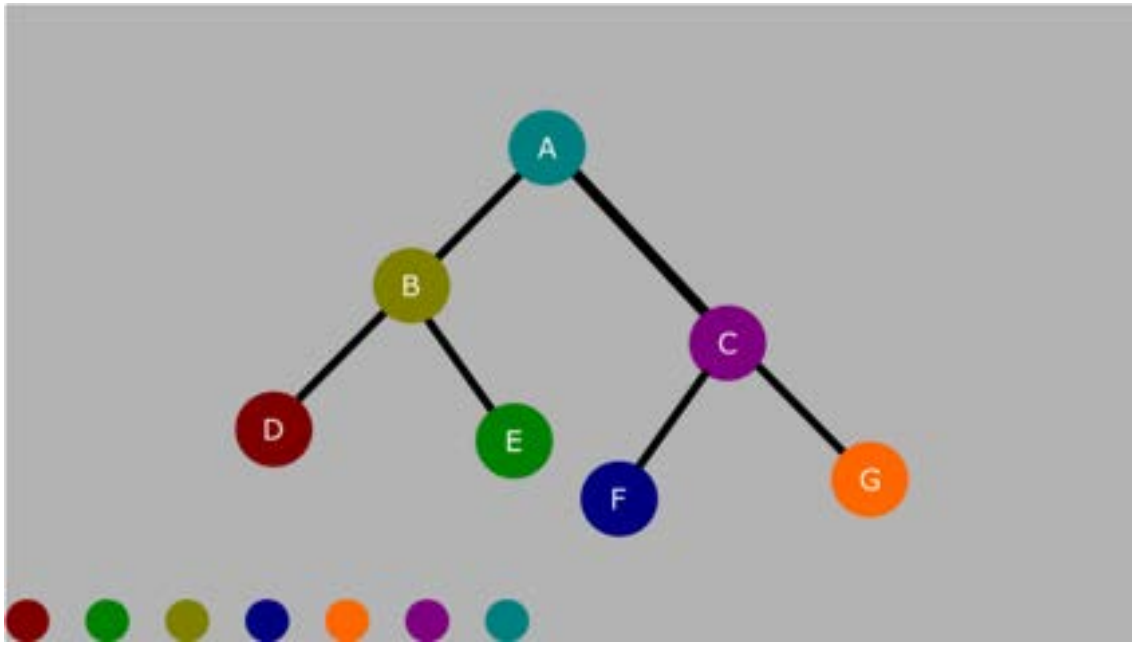
Source: https://www.geeksforgeeks.org/binary-search-tree-traversal-inorder-preorder-post-order/

Diagram for :

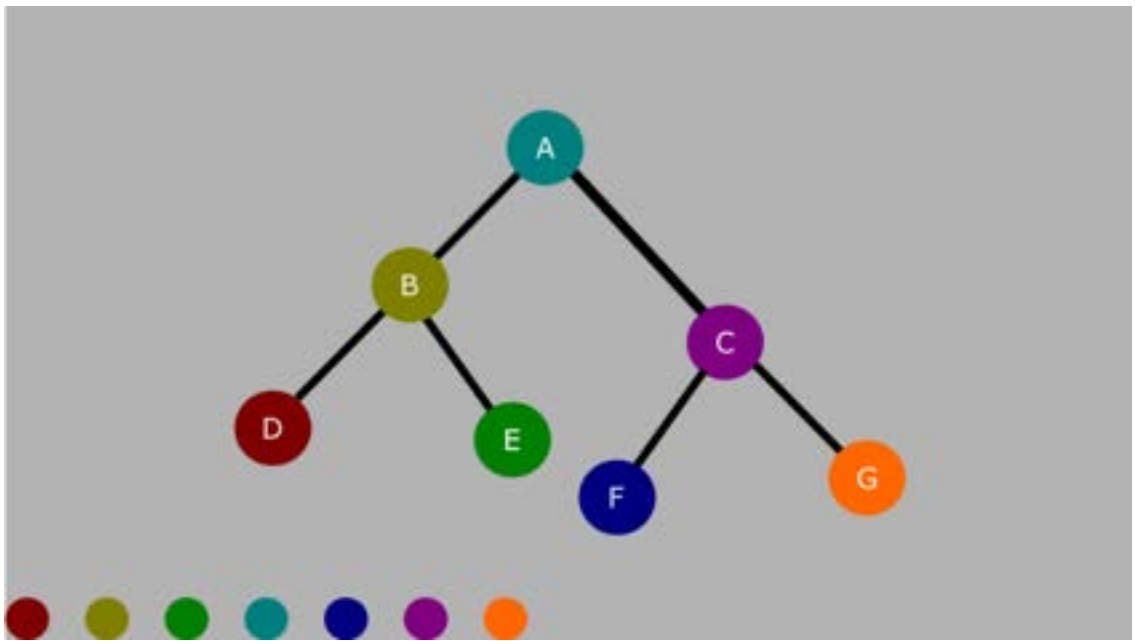
Preorder Traversal of BST



Postorder Traversal of BST



Inorder Traversal of BST



Algorithm for Implementation of BST & Binary tree traversal techniques:

BINARY TREE:

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)

/* inorder tree traversal */

{

if (ptr) {

inorder(ptr->left_child);

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

indorder(ptr->right_child);

}

}
```

A / B * C * D + E

#

Preorder Traversal (recursive version)

void preorder(tree_pointer ptr)

/* preorder tree traversal */

{

if (ptr) {

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

preorder(ptr->left_child);

predorder(ptr->right_child);

}

}

+ * * / A B C D E

#

Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)

/* postorder tree traversal */

{

if (ptr) {

postorder(ptr->left_child);

postdorder(ptr->right_child);

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

}

}
```

BINARY SEARCH TREE:

```
Struct tree{

int data;
```

```
struct tree *left;

struct tree *right;

}

Struct tree *t;
```

Binary Search implementation

Insertion in BST

```
Treetype insert(TreeType *root, int key)

{ CreateNode(NewNode)

// find the position to insert the new node

Treetype *temp = root;

// Pointer parent maintains the trailing pointer of temp

Treetype *parent = NULL;
```

```
while (temp != NULL) {

parent = temp;

if (key < temp->data) temp = temp->left;

else
temp = temp->right;

}

// If the root is NULL i.e the tree is empty The new node is the root node

if (parent == NULL) root = newnode;

// If the new key is less then the leaf node key Assign the new node to be its left child

else if (key < parent->data)
parent->left = newnode;

// else assign the new node its right child

else
parent->right = newnode;
```

return root;

}

Count nodes

int countNodes(TreeType t)

{

If (t==Null)

Print “tree is empty”

Else if (Left(t)==Null AND Right(t)==Null)

return 1

Else

return(CountNodes(Left(t)+CountNodes(Right(t))+1

}

Cases:

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

```
#include <iostream>

using namespace std;


class TreeNode {

public:

   int data;

   TreeNode* left;

   TreeNode* right;

   TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}

};


class BinarySearchTree {

private:

   TreeNode* root;


public:

   BinarySearchTree() : root(nullptr) {}


   void insert(int val) {

      root = insertRec(root, val);

   }


   TreeNode* insertRec(TreeNode* node, int val) {

      if (node == nullptr) {
```

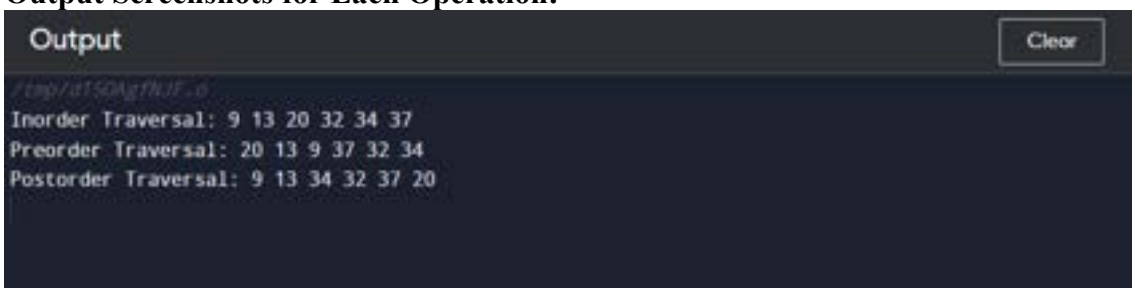
```
        return new TreeNode(val);
    }

    if (val < node->data) {
        node->left = insertRec(node->left, val);
    } else if (val > node->data) {
        node->right = insertRec(node->right, val);
    }

    return node;
}

void inorderTraversal(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    inorderTraversal(node->left);
    cout << node->data << " ";
    inorderTraversal(node->right);
}

void preorderTraversal(TreeNode* node) {
    if (node == nullptr) {
        return;
```

```
    }

    cout << node->data << " ";
    preorderTraversal(node->left);
    preorderTraversal(node->right);
}

void postorderTraversal(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    postorderTraversal(node->left);
    postorderTraversal(node->right);
    cout << node->data << " ";
}

void printInorder() {
    cout << "Inorder Traversal: ";
    inorderTraversal(root);
    cout << endl;
}

void printPreorder() {
    cout << "Preorder Traversal: ";
```

```
        preorderTraversal(root);
        cout << endl;
    }

    void printPostorder() {
        cout << "Postorder Traversal: ";
        postorderTraversal(root);
        cout << endl;
    }
};

int main() {
    BinarySearchTree bst;
    bst.insert(20);
    bst.insert(37);
    bst.insert(32);
    bst.insert(13);
    bst.insert(9);
    bst.insert(34);

    bst.printInorder();
    bst.printPreorder();
    bst.printPostorder();

    return 0;
```

}

Output Screenshots for Each Operation:



Conclusion:-Thus,we've successfully implemented Binary Search Tree and Binary Tree after understanding the various techniques and operations performed on them. . We studied inorder, preorder and postorder traversal. All these techniques have different applications. I implemented these techniques in C++.**PostLab Questions:**

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

1. File Systems:
In computer science, file systems on a computer's hard drive are often organized in a tree-like structure. Each directory (or folder) can contain multiple files or other directories. This hierarchical organization is well-suited for representation using trees.
The top-level directory represents the root of the tree, subdirectories represent child nodes, and files represent leaf nodes. Tree structures make it easy to navigate and manage files and directories efficiently.

2. Binary Search Trees (BSTs) in Databases:
Binary Search Trees are fundamental data structures used in database systems. In a BST, each node has at most two children: a left child with a value less than its parent and a right child with a value greater than its parent. BSTs are used to implement search, insert, delete, and update operations in databases efficiently. When data is stored in a BST, search operations have an average time complexity of O(log n), making it much faster than linear search operations on unsorted data. This efficiency is crucial in applications where large datasets need to be searched or queried rapidly, such as in database management systems.

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

B-Tree:

Structure:

Each node in a B-tree can have between t-1 and 2t-1 keys, where t is the minimum degree of the tree.

All keys in a node are stored in sorted order.

Child pointers are stored for each key, allowing for efficient traversal and search operations.

Search Operation:

B-trees support efficient search operations with a time complexity of O(log_t(n)), where t is the minimum degree of the tree and n is the number of nodes.

Insertion and Deletion:

Insertion and deletion operations can lead to restructuring of the tree, including splitting or merging nodes.

B-trees maintain balance by redistributing keys among siblings or merging nodes when necessary.

Node Splitting:

During insertion, if a node overflows (contains more than 2t-1 keys), it is split into two nodes.

Data Storage:

B-trees can store data in both internal nodes and leaf nodes.

B+ Tree:

Structure:

B+ trees are similar to B-trees, but they have additional restrictions:

All data is stored in the leaf nodes, not in internal nodes.

Leaf nodes are linked together in a linked list, allowing for efficient range queries.

Search Operation:

B+ trees also support efficient search operations with a time complexity of O(log_t(n)), similar to B-trees.

Insertion and Deletion:

Insertion and deletion operations in B+ trees are also similar to B-trees.

Deletion of keys does not affect the leaf structure, ensuring that the linked list of leaf nodes remains intact.

Node Splitting:

During insertion, if a leaf node overflows, it is split into two nodes. However, this split does not affect the overall structure of the tree.

Data Storage:

All data is stored in the leaf nodes, and internal nodes only store keys for routing purposes.

B+ trees are ideal for range queries due to their linked list structure of leaf nodes.

Comparison:

Data Storage:

B-trees can store data in both internal and leaf nodes.

B+ trees store all data in leaf nodes, making them efficient for range queries and sequential access.

Range Queries:

B+ trees are more suitable for range queries due to their linked list structure of leaf nodes.

B-trees can handle range queries but might require traversal of multiple levels.

Node Structure:

B-trees have a more flexible internal structure, allowing for data storage in internal nodes.

B+ trees have a uniform structure with all data stored in leaf nodes.

Pointer Efficiency:

B+ trees have fewer pointers in internal nodes since they only point to leaf nodes.

B-trees have more pointers in internal nodes due to the possibility of data storage in internal nodes.