Practical – 6

AIM: Implementation of binary tree and its traversal (preorder, inorder, postorder)

Program

```
#include <iostream>
#include <queue>
using namespace std;

class Node
{
public:
    int data;
    Node *right;
    Node *left;

    Node(int value)
    {
        data = value;
        right = NULL;
        left = NULL;
    }
};

Node *insertNode(Node *root, int value)
{
    if (root == NULL)
    {
        root = new Node(value);
        return root;
    }

    Node *temp = root;
    while (true)
    {
        if (value > temp->data)
        {
            if (temp->right == NULL)
            {
                temp->right = new Node(value);
                break;
            }
            temp = temp->right;
        }
```

```
        else
        {
            // for value<root->data
            if (temp->left == NULL)
            {
                temp->left = new Node(value);
                break;
            }
            temp = temp->left;
        }
    }
    return root;
}

void displayPreorder(Node *root)
{
    if (root == NULL)
    {
        return;
    }
    cout << root->data << ",";
    displayPreorder(root->left);
    displayPreorder(root->right);
}

void displayInorder(Node *root)
{
    if (root == NULL)
    {
        return;
    }
    displayInorder(root->left);
    cout << root->data << ",";
    displayInorder(root->right);
}

void displayPostorder(Node *root)
{

    if (root == NULL)
    {
        return;
    }
    displayPostorder(root->left);
    displayPostorder(root->right);
    cout << root->data << ",";
}
```

```
vector<vector<int>> levelorder(Node *&root)
{
    cout << endl;
    vector<int> level;
    vector<vector<int>> ans;
    queue<Node *> q;

    q.push(root);
    q.push(NULL);
    while (!q.empty())
    {
        Node *temp = q.front();
        q.pop();
        if (temp == NULL)
        {
            cout << endl;
            if (!q.empty())
            {
                q.push(NULL);
            }
        }
        else
        {
            if (temp->left != NULL)
                q.push(temp->left);

            if (temp->right != NULL)
                q.push(temp->right);

            cout << temp->data << " ";
        }
    }

    return ans;
}

int main()
{
    Node *root = NULL;
    root = insertNode(root, 100);
    root = insertNode(root, 20);
    root = insertNode(root, 200);
    root = insertNode(root, 10);
    root = insertNode(root, 30);
    root = insertNode(root, 150);
    root = insertNode(root, 300);

    cout << "Preorder : ";
    displayPreorder(root);
    cout << endl;
```

```
    cout << "inorder : ";
    displayInorder(root);
    cout << endl;
    cout << "Postorder : ";
    displayPostorder(root);

    cout << endl<< endl;
    cout << "-------------level order traversel------------------";
    vector<vector<int>> x = levelorder(root);
    return 0;
}
```

OUTPUT

```
Preorder : 100,20,10,30,200,150,300,
inorder : 10,20,30,100,150,200,300,
Postorder : 10,30,20,150,300,200,100,

------------------level order traversel-------------------
100
20 200
10 30 150 300
```

Time analysis

Operation	Best Case	Average Case	Worst Case
Search	O(1)	O(log n)	O(n)
Insertion	O(1)	O(log n)	O(n)
Deletion	O(1)	O(log n)	O(n)
Inorder Traversal	O(n)	O(n)	O(n)
Preorder Traversal	O(n)	O(n)	O(n)
Postorder Traversal	O(n)	O(n)	O(n)

Applications

1. Symbol Tables:
   BSTs are frequently used to implement symbol tables in compilers and interpreters. In a symbol table, identifiers (such as variable names) are stored along with their associated information, and the BST structure allows for efficient retrieval.

2. Database Indexing:
   In database management systems, BSTs can be employed for indexing. For example, a BST can be used to index records based on certain attributes, enabling faster search operations.

3. File Systems:
   BSTs are utilized in file systems for organizing and searching directories and files. The hierarchical structure of file systems can be represented effectively using BSTs.

4. Router Tables in Networking:
   In networking, BSTs can be used to implement router tables. IP addresses or routing information can be stored in a BST for efficient routing lookups.

5. Compression Algorithms:
   Huffman coding, a widely used compression algorithm, often employs binary trees. Binary trees, including BSTs, can be used to represent the hierarchical structure of Huffman codes.

6. Auto-Completion in Text Editors:
   Binary Search Trees can be employed to implement auto-completion functionality in text editors. The tree structure allows for quick search and retrieval of suggestions.

7. Caching:
   BSTs are utilized in caching mechanisms. Items with the highest or lowest priority can be efficiently identified and removed using the binary search property.

8. Priority Queues:
   BSTs can be used to implement priority queues, where items with higher priority are accessed more quickly. The root of the BST typically represents the highest-priority item.

9. Game Trees in Artificial Intelligence:
   BSTs are employed in game trees to represent possible moves and outcomes in game-playing algorithms. The tree structure facilitates decision-making in games.

10. Code Optimization:
    In compilers, BSTs can be used for code optimization. Symbolic expressions and intermediate code representations can be efficiently manipulated using BSTs.