CL2001

Data Structures Lab

[DS]

Lab 08

Binary Trees and their properties, Binary Search Trees, their operations and applications.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2025

Lab Content

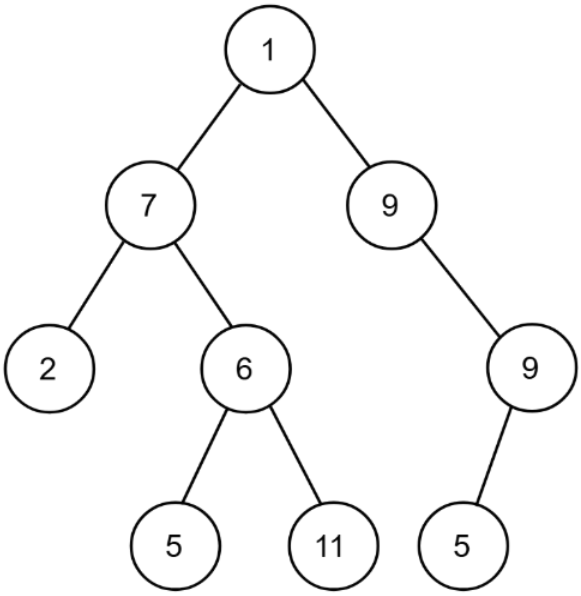
1. Binary Tree
2. Properties of Binary Tree
3. Full Binary Tree
4. Complete Binary Tree
5. Binary Search Trees
6. Operations in BST

Binary Tree

A binary tree in a data structure is a hierarchical model used to organize data efficiently.

A binary tree in DSA (Data Structures and Algorithms) is a way to organize data in a hierarchical structure. In a binary tree, each node has at most two children, called the left child and the right child. The topmost node is called the root, and the nodes with no children are called leaves.

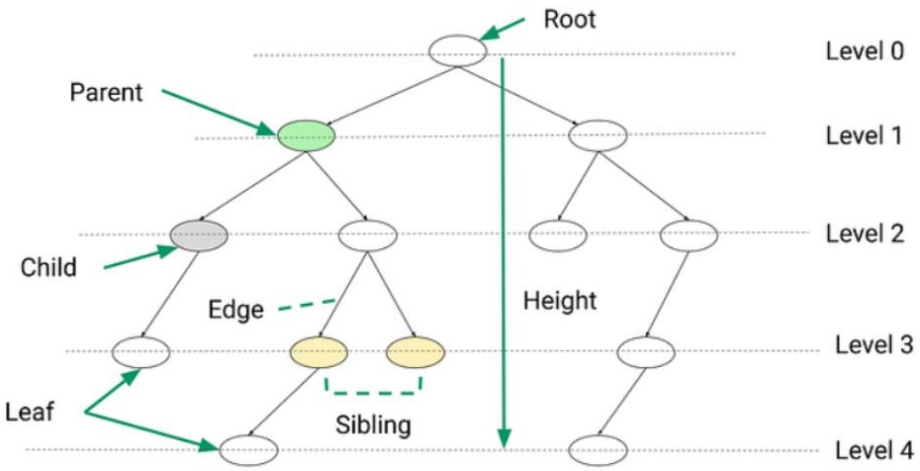
The basic idea of a binary tree is to have a parent-child relationship between nodes. Each node can have a left and a right child, and this pattern continues down the tree. This structure makes it easy to organize and find data quickly.



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

// Define the Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to create a new node
Node* createNode(int data) {
    return new Node(data);
}

int main() {
    // Manually create a simple binary tree
    //
    //         1
    //        / \
    //       2   3
    //      / \
    //     4   5
    //
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    // Just print the root and its children
    cout << "Root: " << root->data << endl;
    cout << "Left Child of Root: " << root->left->data << endl;
    cout << "Right Child of Root: " << root->right->data << endl;

    return 0;
}
```

Terminologies

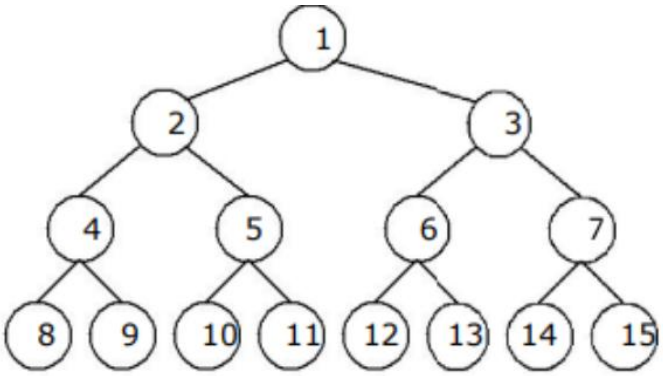
- Root → The topmost node of a tree with no parent.
- Parent → A node that has one or more child nodes.
- Child → A node directly connected below a parent node.
- Edge → The connection (link) between a parent and a child node.
- Leaf → A node with no children (end node of a branch).
- Sibling → Nodes that share the same parent.
- Height → The length of the longest path from the root to any leaf node.
- Level → The distance (in edges) from the root node, starting at Level 0.



Types of Binary Tree in Data Structure

Full Binary Tree

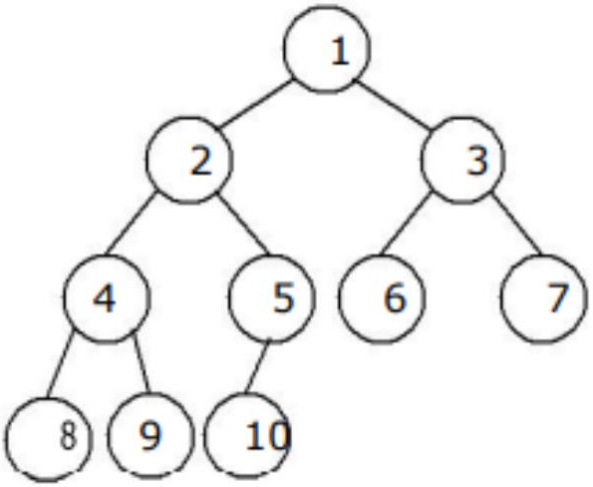
A full binary tree is a tree where every node has either 0 or 2 children.



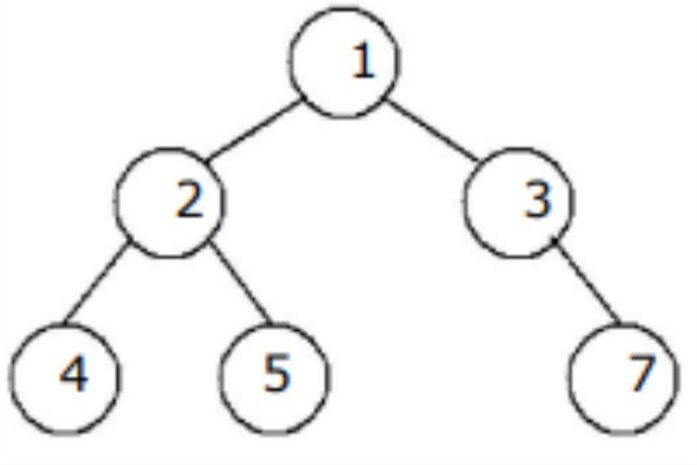
Full Binary Tree

Complete Binary Tree

A complete binary tree is a tree where all levels are fully filled except possibly the last level, which is filled from left to right.



Complete Binary Tree

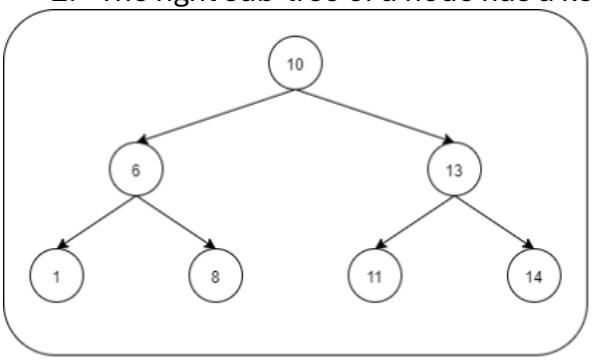


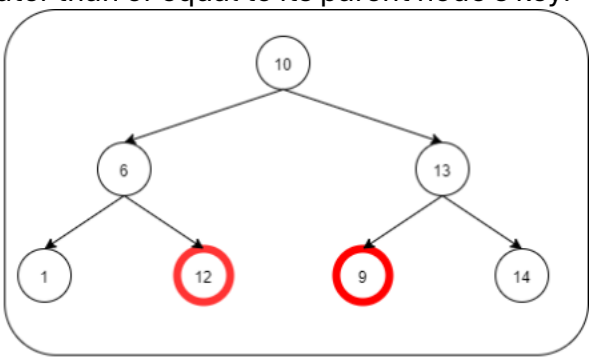
Not Complete Binary Tree

Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow these properties:

1. The left sub-tree of a node has a key less than or equal to its parent node's key.
2. The right sub-tree of a node has a key greater than or equal to its parent node's key.





For Visualization:
https://www.cs.usfca.edu/~galles/visualization/BST.html

Traversals

The binary search tree property allows us to obtain all the keys in a binary search tree in a sorted order by a simple traversing algorithm, called an in order tree walk, that traverses the left sub tree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.
The algorithms are described below, with Node initialized to the tree's root.

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;

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

```
int main()
{
    struct Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    //preorder(root);
    //inorder(root);
    postorder(root);
    return 0;
}
```

Preorder Traversal

1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

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

In-order Traversal

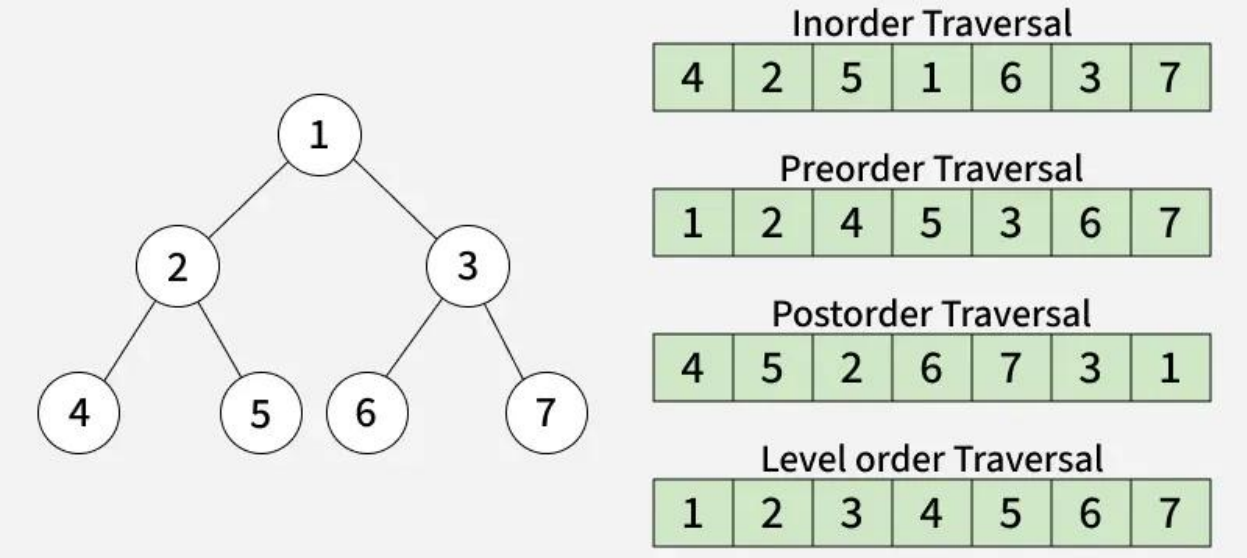
1. Traverse Node's left sub-tree.
2. Visit Node.
3. Traverse Node's right sub-tree

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

Post-order Traversal

1. Traverse Node's left sub-tree.
2. Traverse Node's right sub-tree.
3. Visit Node

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



Operations in BST

Since BST is a type of Binary Tree, the same operations are performed in BST too. Although, insertion and deletion in BST are much stricter with predetermined conventions so that even after performing an operation, the properties of BST are not violated.

1. Searching

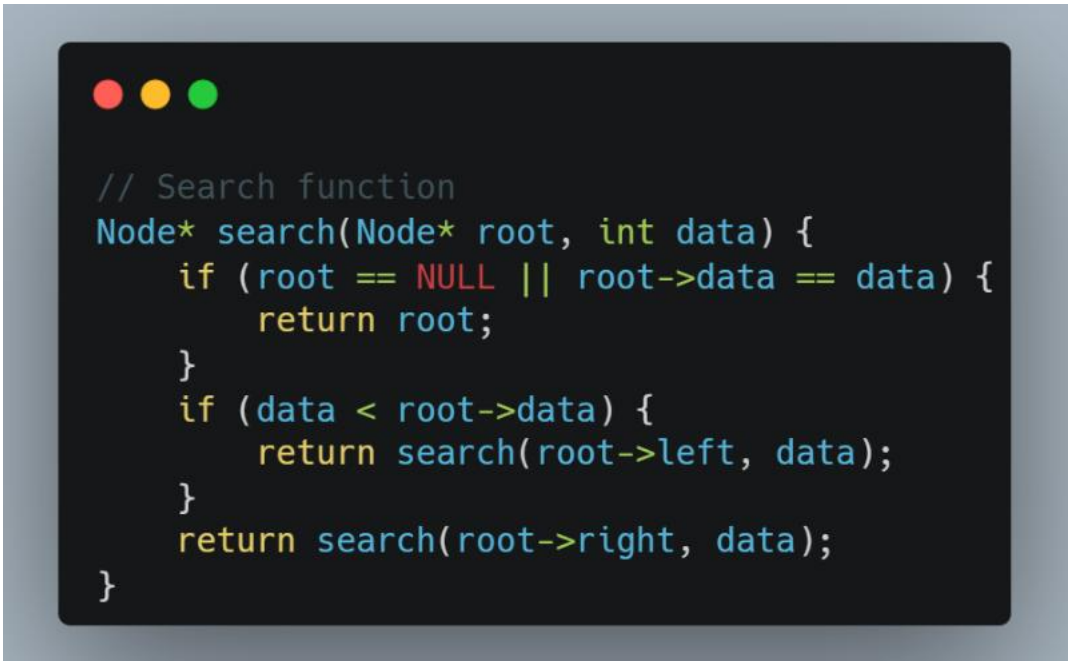
In a normal binary tree, if we need to search for an element, we need to traverse the entire tree (in the worst case). But in BST, we can optimize this by using its properties to our advantage. Any guesses to how we can achieve that?

The basic idea is quite simple really. Let's assume we need to search for value item in a BST. When we compare it with the root of the tree, we are left with three cases.

1. item == root.val: We terminate the search as the item is found
2. item > root.val: We just check the right subtree because all the values in the left subtree are lesser than root.val
3. item < root.val: Now we just check the left subtree as all values in the right subtree are greater than root.val

We keep on recursing in this manner and this decreases our search time complexity up to a great extent as we just need to look at one subtree and reject the other subtree saving us the trouble of comparing a batch of values.

Let's implement this now.



```
// Search function
Node* search(Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    }
    return search(root->right, data);
}
```

2. Insertion:

Insertion in Binary Search Tree, just like Binary Tree is done at the leaf. But since it needs to maintain the properties of BST, the convention is decided in this case.

The basic idea is quite simple really. We need to insert a node in BST with value item and return the root of the new modified tree.

If the root is NULL,

create a new node with value item and return it.

Else, Compare item with root.val

If root.val < item , recurse for right subtree

If root.val > item , recurse for left subtree

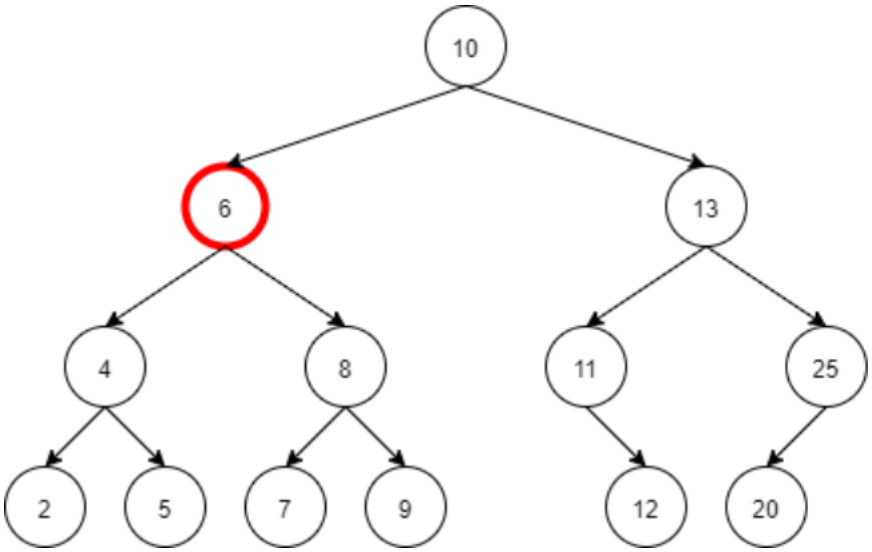
Let's implement this now.

```
// Insertion function
Node* insert(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;
}
```

3. Deletion:

When we delete a node in BST, we may encounter three cases.

1. The node to be deleted is a leaf node : Easiest case, simply remove the node from the tree
2. The node to be deleted has only one child : Replace the node by its child
3. The node to be deleted has both children : The node still needs to be replaced to maintain BST properties, but which node should replace this deleted node?



Suppose we have to delete the node with value 6, which node should be selected ideally to replace this node?

→ The inorder successor of this node would be the aptest choice to replace this node as its inorder successor is the smallest element that is greater than this node.

Note: The inorder predecessor can be used to replace this node too. But conventionally, we use the inorder successor to replace the node.

Let's implement this now.

```
// Deletion function
Node* deleteNode(Node* root, int data) {
    if (root == NULL) {
        return root;
    }
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

Sample Code

Applications

Binary search trees are an essential data structure in computer science and have various applications in various domains. Their efficient search, insertion, and deletion operations make them valuable for solving many problems. Here are some common applications of binary search trees:

1. Searching and Retrieval: BST trees are mainly utilized for effective data retrieval and searching operations. The binary search property helps to guarantee that the search operation can be completed in O(log n) time, where n is the number of nodes in the tree.

2. Database Systems: Binary search trees are used in various databases to search and index large, scattered reports. For example, we can store the names using the BST tree structure in the phonebook.

3. Auto-Complete and Spell Check: A binary search tree can implement auto-complete functionality at various places, like search engines. It can quickly suggest completions while typing based on the prefix entered. They are also used in spell-checking algorithms to suggest corrections for incorrectly spelled words.

4. File Systems: Various current version of file systems use the binary search algorithm to store files in directories.

5. Priority Queues: They can also be used to implement priority queues. The key of each element represents its priority, and you can efficiently extract the element with the highest (or lowest) priority.

6. Optimization Problems: Binary search trees can be used to solve various optimization problems. For instance, in the field of dynamic programming, BSTs can be used to find the optimal solution for specific problems efficiently.

7. Implement Decision Tree: It can implement decision trees in artificial intelligence and machine learning algorithms. It is used to predict outcomes and decisions of the models. These trees can help in various fields like diagnosis, research, and financial analysis work.

8. Encryption Algorithms: It can help encrypt sensitive information using a key encryption algorithm by generating public and private keys.

LAB TASKS