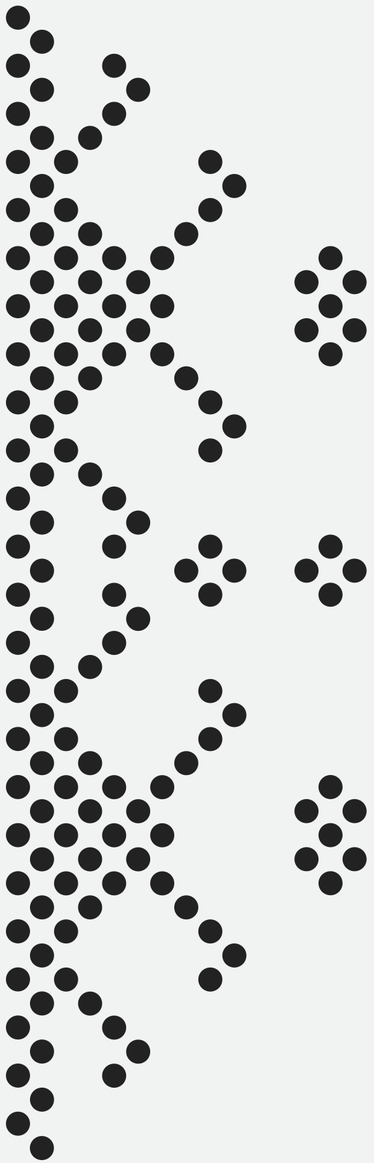


NATIONAL UNIVERSITY
of Computer & Emerging Sciences

Data Structures

Dr. Jawwad A Shamsi

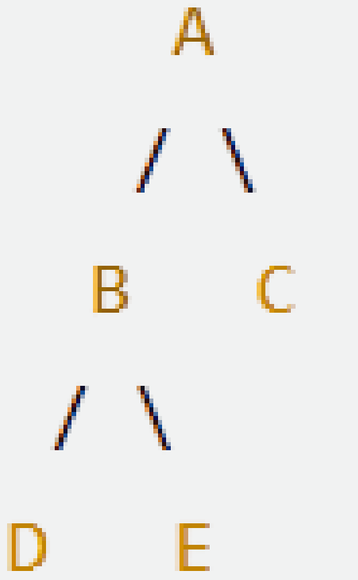
Professor
Dean (Computing)

Trees

- A non-linear data structure consisting of nodes connected by edges.
- Represents hierarchical relationships (e.g., file systems, organization charts).

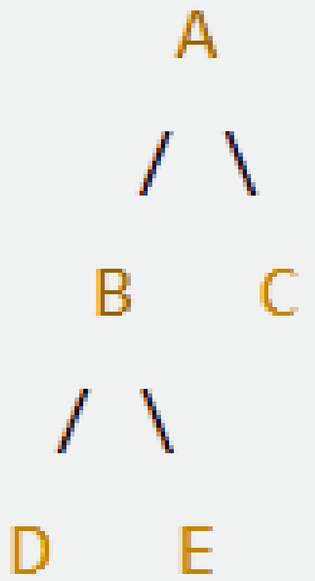
Terminology

- Root: topmost node
- Parent / Child: connected nodes
- Leaf: node with no children
- Edge: link between parent and child
- Path: sequence of nodes from root to a node
- Height: longest path from root to a leaf



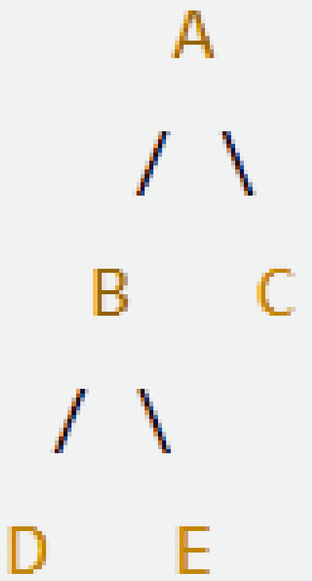
Characteristics

- Has N nodes and N–1 edges.
- No cycles.
- Exactly one path between any two nodes.
- Rooted structure (one node is designated as root).
- Trees can be:
- General trees
- Binary trees
- Binary search trees
- Heaps, etc.



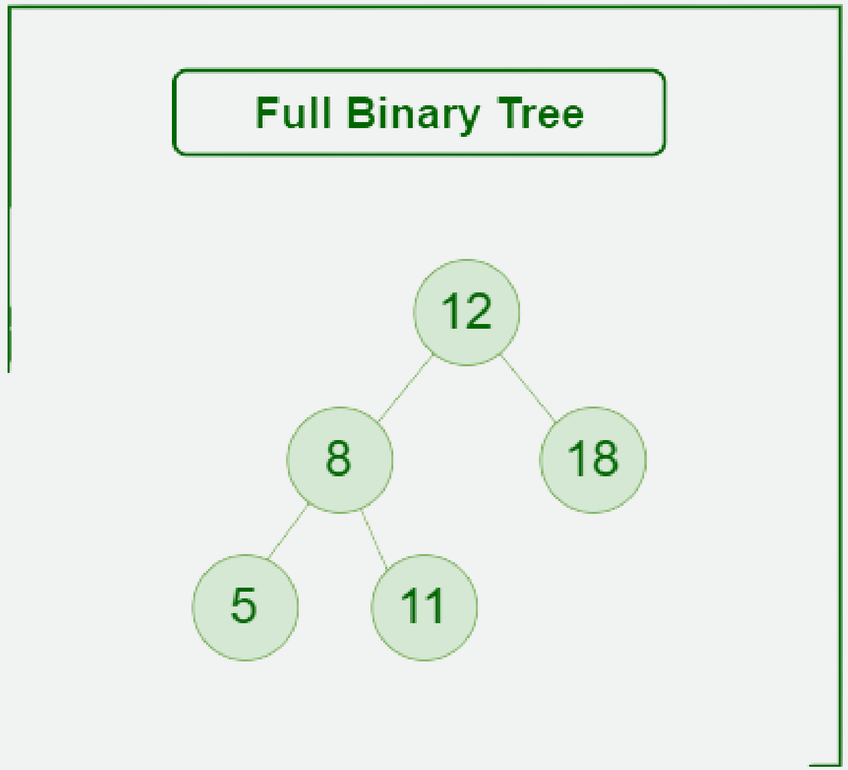
Binary Tree

- A tree where each node can have at most two children.
- Children are called:
  - Left child
  - Right child
- Types:
- Full Binary Tree – every node has 0 or 2 children.
- Complete Binary Tree – all levels filled except possibly last (left to right).
- Perfect Binary Tree – all internal nodes have 2 children and all leaves are at same level.



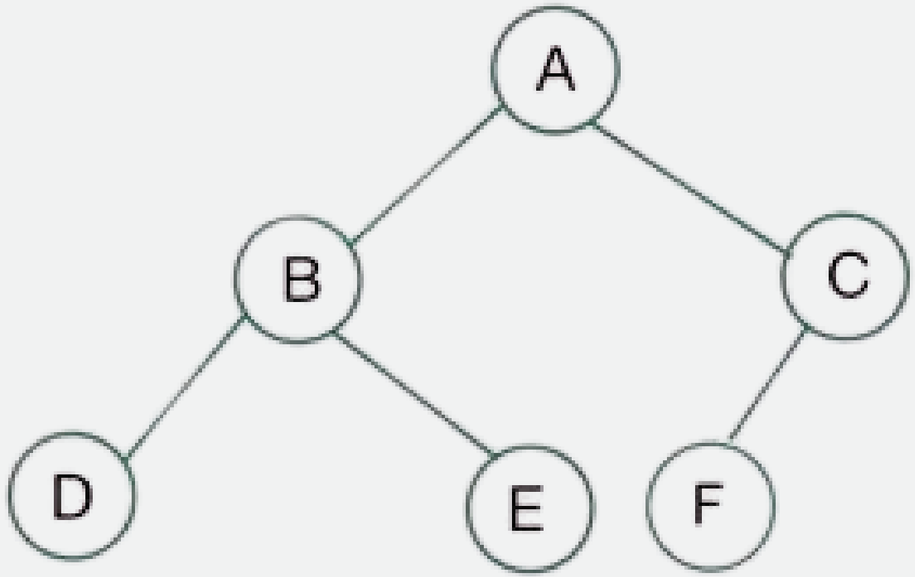
Full Binary Tree

A tree with zero or two children



Complete Binary Tree

A binary tree is said to be a complete binary tree if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible



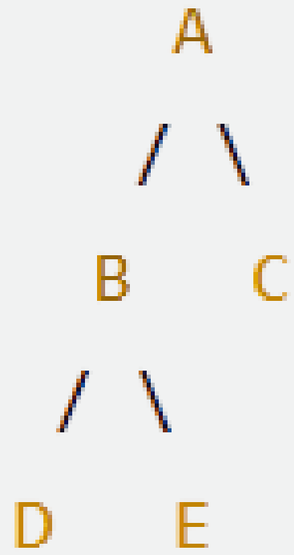
Case	Is Root Internal?
Tree has >1 node	Yes
Tree has only one node	No (root = leaf)

Tree Traversal

- Ways to visit all nodes:
- Inorder (Left, Root, Right)
- Preorder (Root, Left, Right)
- Postorder (Left, Right, Root)

Tree Traversal

Traversal	Output
Preorder	A B D E C
Inorder	D B E A C
Postorder	D E B C A



Trees: Applications

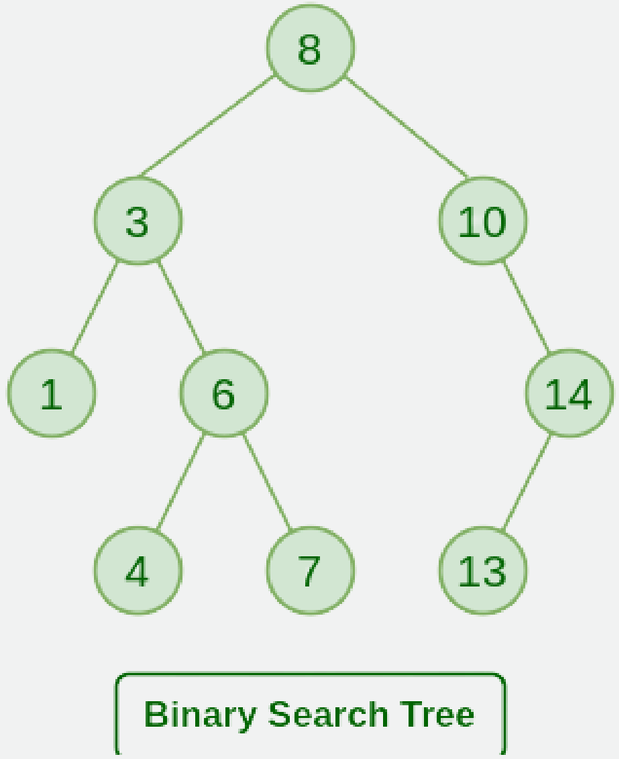
- Hierarchical data (File system, XML/HTML, Organization charts)
- Parsing expressions in compilers
- Binary Search Trees (efficient searching/sorting)
- Huffman Coding (compression)
- Decision Trees (AI/ML)
    -

Domain	Tree Type	Application Example
File Systems	General Tree	Folder structure
Databases	B / B+ Tree	Indexing
Searching	BST / AVL	Fast lookups
Compilers	Syntax Tree	Expression parsing
Networking	Spanning Tree	Loop-free routing
AI / ML	Decision Tree	Classification
Compression	Huffman Tree	Data encoding
Web Dev	DOM Tree	Page rendering

Binary Search Tree (BST)

A special binary tree with the following properties:

- Left subtree of a node contains only nodes with keys less than the node's key.
- Right subtree contains only nodes with keys greater than the node's key.
- Both left and right subtrees are also BSTs.



Binary Search Tree (BST)

- Start from root.
- If key = root → found.
- If key < root → search left subtree.
- If key > root → search right subtree.
- Repeat until key found or subtree empty.

Example: Search 60

→ 60 > 50 → go right

→ 60 < 70 → go left

→ found

Binary Search Tree (BST)- Insertion

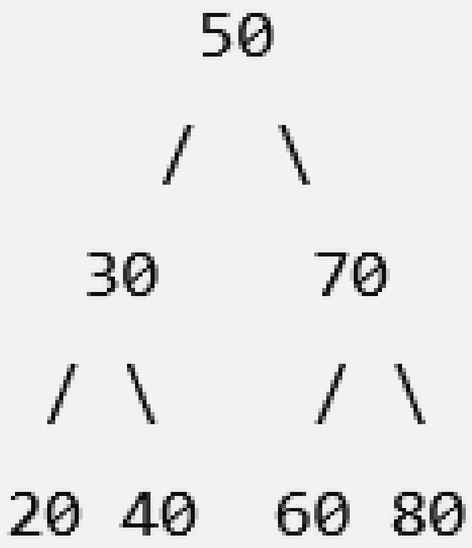
- Start from root.
- Compare key with current node.
- If smaller → go left; if larger → go right.
- Insert when null child found.

Binary Search Tree (BST) - Deletion

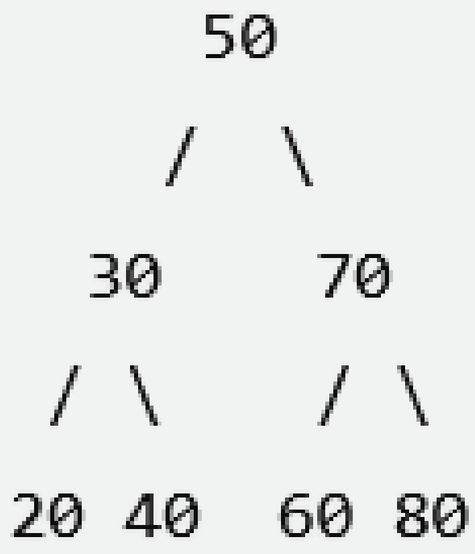
- Leaf node → remove directly.
- Node with one child → link parent directly to child.
- Node with two children →
  - Find inorder successor (smallest node in right subtree).
  - Replace node's key with successor's key.
  - Delete successor node.

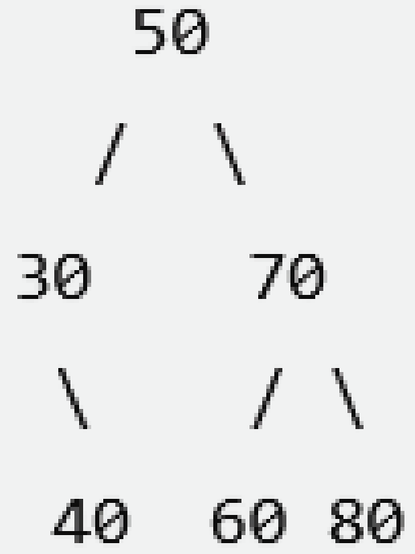
- Example: Delete 50 from the BST (has two children)
- → Find successor = 60
- → Replace 50 with 60
- → Delete original 60 node.

Case	Description	Action
1. Leaf Node	Node to be deleted has no children	Simply delete the node.
2. One Child	Node has one child (left or right)	Delete node and connect child directly to the node’s parent.
3. Two Children	Node has two children	Find inorder successor (smallest node in right subtree), replace the node’s value with it, and then delete that successor.

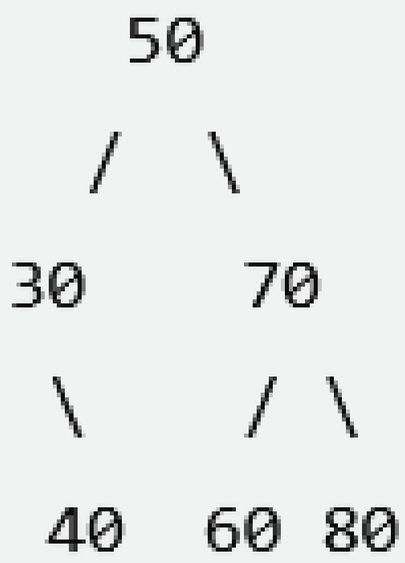

50
30
70
20
40
60
80

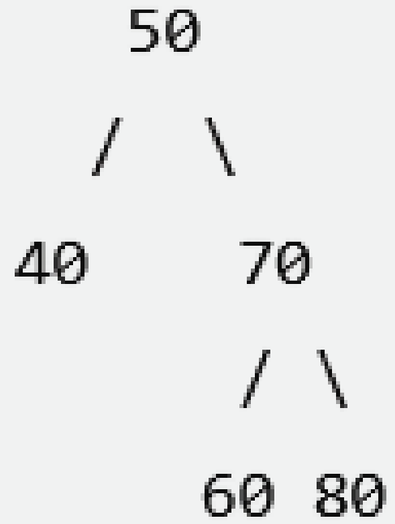
Case 1: Delete 20 → no children → just remove it.



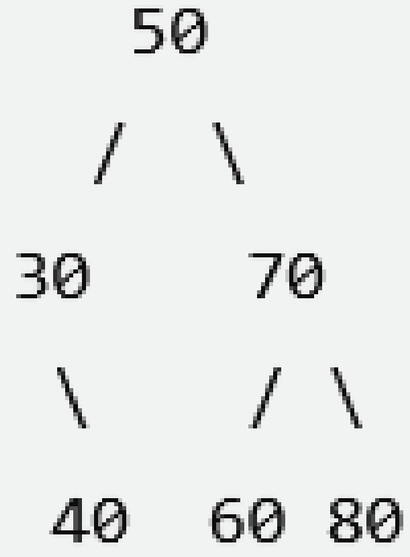


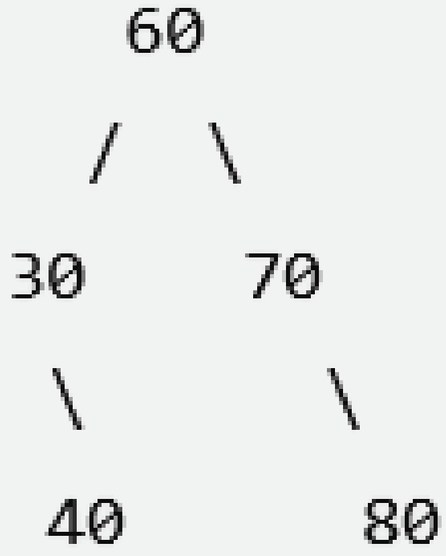
Case 2 – Delete Node with One Child (e.g., 30)





Case 3 – Delete Node with Two Children (e.g., 50)





Coding

Struct

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

In order

```
void inorder(Node* root) {
    if (root == NULL)
        return;

    inorder(root->left);       // Visit left subtree
    cout << root->data << " ";  // Visit root
    inorder(root->right);      // Visit right subtree
}
```

Pre order

```
void preorder(Node* root) {
   if (root == NULL)
      return;

   cout << root->data << " ";  // Visit root
   preorder(root->left);      // Visit left subtree
   preorder(root->right);     // Visit right subtree
}
```

Post order

```
void postorder(Node* root) {
   if (root == NULL)
      return;

   postorder(root->left);     // Visit left subtree
   postorder(root->right);    // Visit right subtree
   cout << root->data << " "; // Visit root
}
```

Insertion

```
Node* insert(Node* root, int key) {
    if (root == NULL)
        return new Node(key); // Create new node if tree is empty or we reached leaf

    if (key < root->data)
        root->left = insert(root->left, key);
    else if (key > root->data)
        root->right = insert(root->right, key);

    // If key == root->data, we do nothing (no duplicates allowed in BST)
    return root;
}
```

Searching

```
bool search(Node* root, int key) {
   if (root == NULL)
      return false; // not found

   if (root->data == key)
      return true; // found

   if (key < root->data)
      return search(root->left, key);
   else
      return search(root->right, key);
}
```

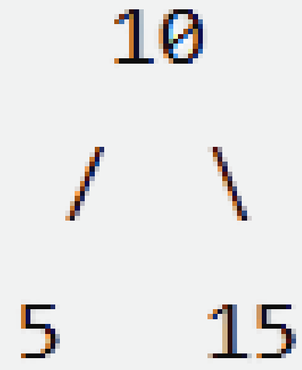
Delete Function

```
Node* deleteNode(Node* root, int key) {
    if (root == NULL) return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node found
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
                // Case 3: Two children
            // Find the inorder successor (smallest
                node in right subtree)
              Node* temp = minValueNode(root-
                >right);

            // Copy the inorder successor's data to
    this node
            root->data = temp->data;

            // Delete the inorder successor
            root->right = deleteNode(root->right,
    temp->data);
        }

        return root;}
```

root = deleteNode(root, 5);

Start with root = 10.

Since 5 < 10, go to the left subtree.

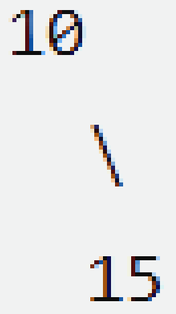


root->left = deleteNode(root->left, 5)

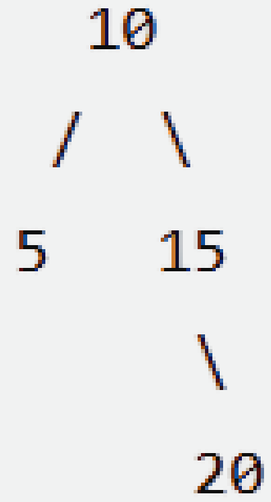
1. Now, root is node 5, and key is 5. Match found.
2. Node 5 has no children → delete it and return NULL.

That NULL goes back up to node 10's left pointer:

;



root = deleteNode(root, 15);



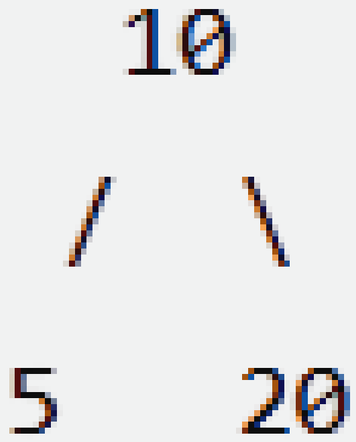
root = 10, key = 15. Since 15 > 10 → go right.

root->right = deleteNode(root->right, 15);

- Now root = 15, key = 15. Found node to delete.
- Node 15 has no left child, but has right child (20).

```
if (!root->left) {
   Node* temp = root->right;  // temp = 20
   delete root;              // delete 15
   return temp;               // return 20
}
```

Returned 20 becomes new root->right for node 10.



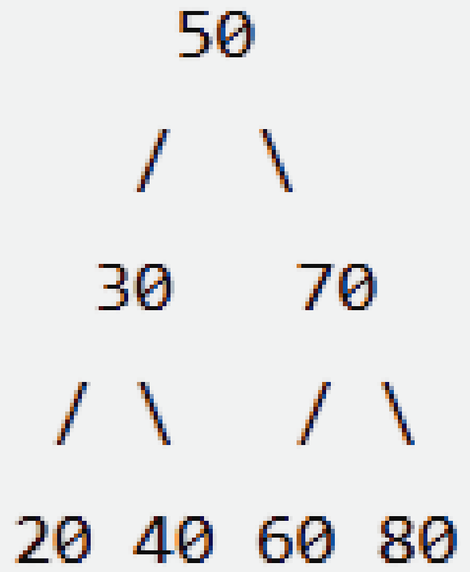
root = deleteNode(root, 50);

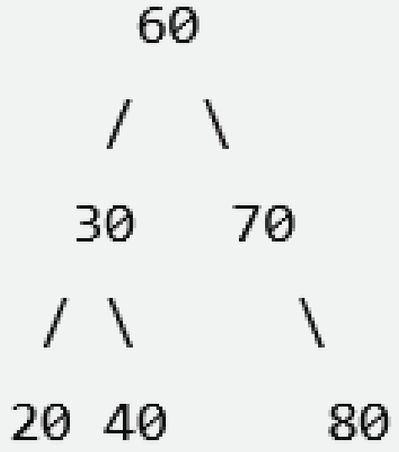
1. Found node 50 → it has two children.
2. Find inorder successor (minimum value in right subtree → 60).
3. Replace root->data with 60.
4. Then recursively delete 60 from the right subtree.

This recursive call looks like:

root->right = deleteNode(root->right, 60);

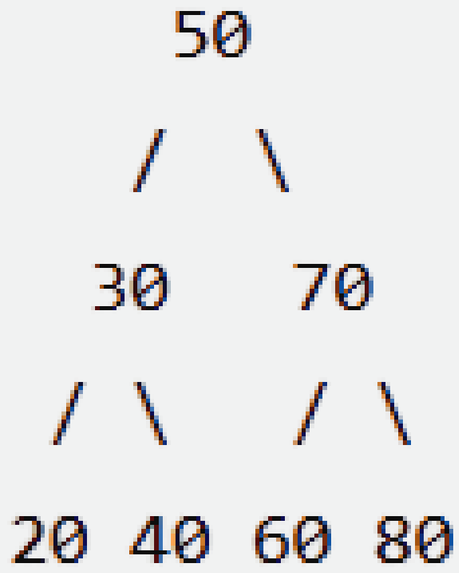
1. First call: root = 50, key = 50
2. Second call (recursive): root = 70, key = 60



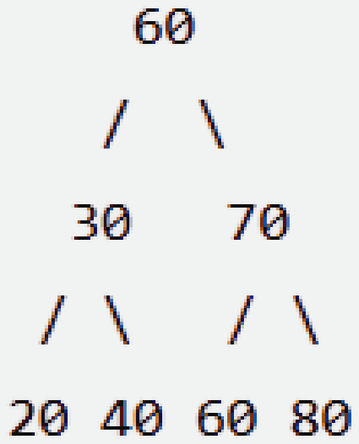


Step 1: root =
deleteNode(root, 50);

root → Node(50)
key = 50



Step 2: The inorder successor is the smallest value in the right subtree (i.e. go right once, then keep going left).



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

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

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

class BST {
public:
    Node* root;

    BST() { root = nullptr; }

    // Insert
    Node* insert(Node* node, int value) {
        if (node == nullptr)
            return new Node(value);
        if (value < node->data)
            node->left = insert(node->left, value);
        else if (value > node->data)
            node->right = insert(node->right, value);
        return node;
    }

    // Search
    bool search(Node* node, int key) {
    if (node == nullptr) return false;
    if (node->data == key) return true;
    if (key < node->data)
    return search(node->left, key);
    else
    return search(node->right, key);
    }
    // Inorder Traversal
    void inorder(Node* node) {
    if (node == nullptr) return;
    inorder(node->left);
    cout << node->data << " ";
    inorder(node->right);
    }
};
int main() {
    BST tree;
    tree.root = tree.insert(tree.root, 50);
    tree.insert(tree.root, 30);
    tree.insert(tree.root, 70);
    tree.insert(tree.root, 20);
    tree.insert(tree.root, 40);
    tree.insert(tree.root, 60);
    tree.insert(tree.root, 80);
    cout << "Inorder traversal: ";
    tree.inorder(tree.root);
    cout << endl;
    cout << "Search 40: " << (tree.search(tree.root, 40) ?
"Found" : "Not Found");
}
```