Trees : Definition, Terminologies & Binary Trees

A tree is a non-linear data structure that represents data in a hierarchical form.
It consists of nodes connected by edges, with a single root node at the top.

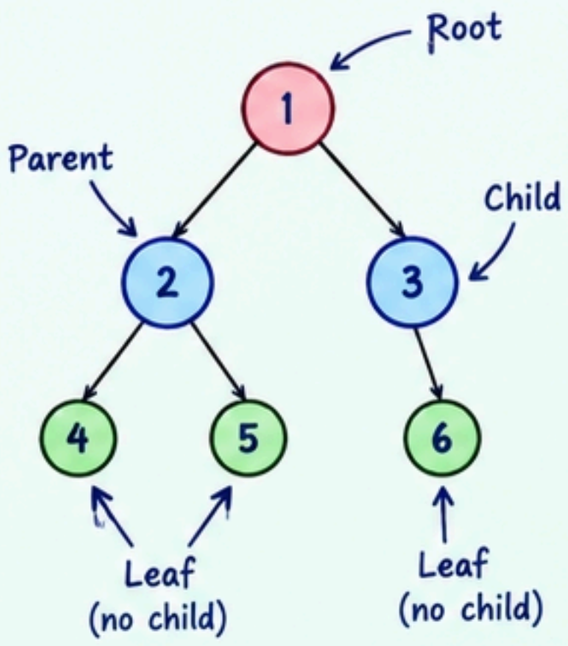
Hierarchical Structure
Not Linear!

1. Definition

- A tree is a collection of nodes where one node is designated as the **root** and the remaining nodes are partitioned into disjoint **subtrees.**
- It is a hierarchical structure with a top-down relationship between nodes.

In simple words:
A tree is a connected structure with no cycles.

2. Terminologies



Root	: The topmost node with no parent.
Parent	: A node that has child(ren).
Child	: A node that has a parent.
Leaf	: A node with no children.
Internal Node	: A node with at least one child (not a leaf).
Subtree	: A tree formed by a node and its descendants.
Height	: The longest path from root to a leaf.
Level	: Distance from root (root is level 0).

3. Types of Trees (Context)

General Tree
- Each node can have any number of children.

Binary Tree
- Each node has at most two children (left and right).

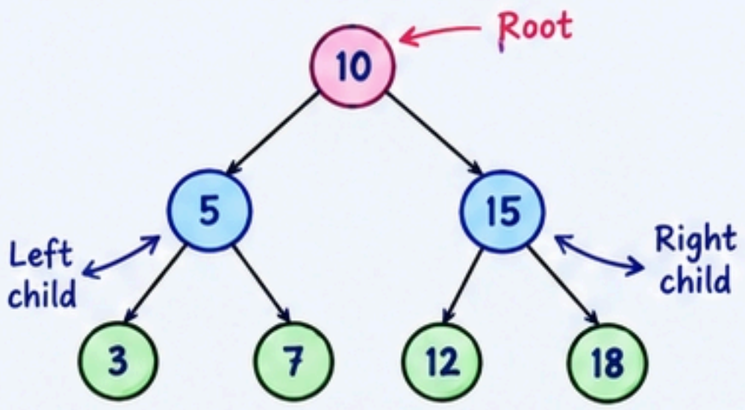
Binary Search Tree (BST)
- Left child < node < right child (for all nodes).

Balanced Tree
- Height difference between left and right subtrees is small.

4. Binary Trees

- A binary tree is a tree in which each node has at most two children, usually called the **left child** and the **right child.**

Example of a Binary Tree



Properties of Binary Trees

- Each node has at most two children.
- The maximum number of nodes at level i is 2^i (starting from level 0).
- The maximum number of nodes in a tree of height h is $2^{h+1} - 1$.
- In a full binary tree, every internal node has exactly two children or is a leaf.
- In a perfect binary tree, all leaf nodes are at the same level.

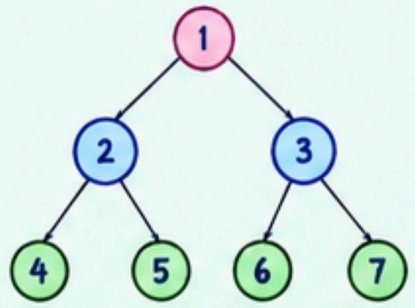
Key Points to Remember

- Binary tree = max 2 children per node.
- Left and right are important (order matters).
- Used in many applications: searching, sorting, expression trees, etc.

Types of Binary Trees

1. Full Binary Tree

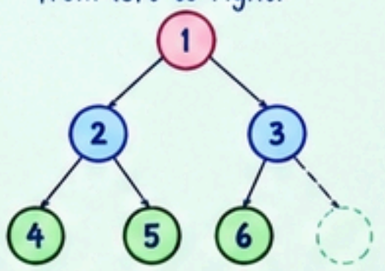
Every node has 0 or 2 children.



(All internal nodes have 2 children)

2. Complete Binary Tree

All levels are completely filled except possibly the last level, which is filled from left to right.



(Last level filled left to right)

In a Binary Tree, the order of traversal is important:

- **Preorder** : Root → Left → Right
- **Inorder** : Left → Root → Right
- **Postorder** : Left → Right → Root

Traversal helps in visiting all nodes in a specific order!

Memory Representation of Trees using Array

A tree can be stored in an array using a simple indexing scheme.

In a binary tree, for any node at index i:

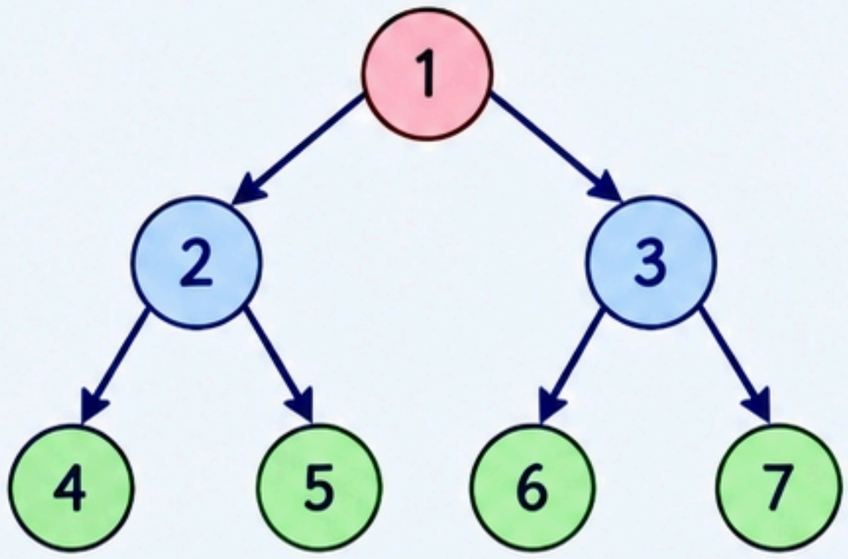
- Left child is stored at index $2i + 1$
- Right child is stored at index $2i + 2$

Where,

i	=	index of current node
$2i + 1$	=	index of left child
$2i + 2$	=	index of right child

Example

Consider the following binary tree:



Array Representation

Using the indexing rule, we store the nodes in an array.

For the above tree, the array will be:

Index	0	1	2	3	4	5	6	7
Value	1	2	3	4	5	6	7	

So, the tree is stored as:

arr = [1, 2, 3, 4, 5, 6, 7]

How It Works?

1. Root (1) is stored at index 0.
2. Left child of 1 (2) is at index $2 \times 0 + 1 = 1$.
3. Right child of 1 (3) is at index $2 \times 0 + 2 = 2$.
4. Left child of 2 (4) is at index $2 \times 1 + 1 = 3$.
5. Right child of 2 (5) is at index $2 \times 1 + 2 = 4$.
6. Left child of 3 (6) is at index $2 \times 2 + 1 = 5$.
7. Right child of 3 (7) is at index $2 \times 2 + 2 = 6$.

Key Points

- This method works well for complete binary trees or nearly complete binary trees.
- If a node doesn't have a child, the corresponding array position is left empty (or set to a special value like -1 or NULL).
- It is simple and uses only one array.
- For sparse trees (where many nodes are missing), this method can waste memory.

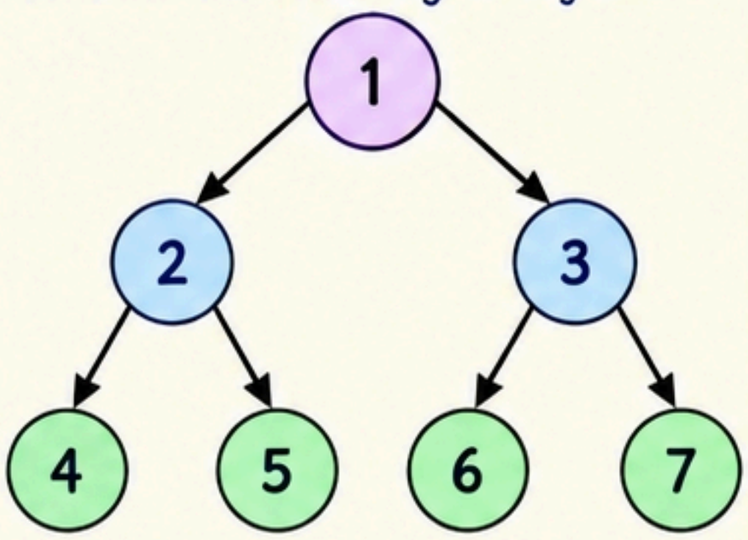
In short: Array index → $2i + 1$ (left child), $2i + 2$ (right child)

Memory Representation of Trees using Linked List

In a linked list representation, each node of the tree is stored in a linked list node with pointers to its left and right child (and optionally to its parent).

1. Binary Tree Example

Consider the following binary tree:



2. Linked List Node Structure

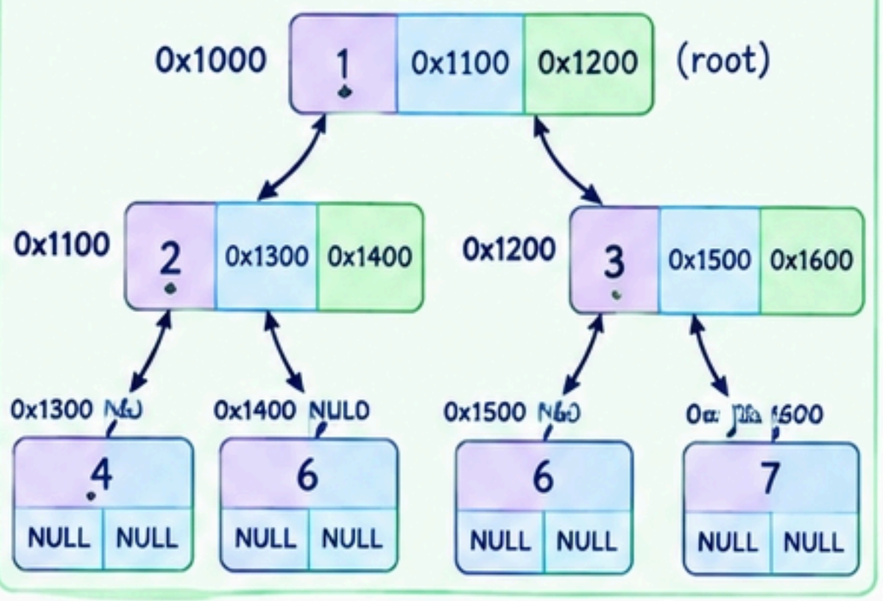
Each node contains:

data (value)	left (pointer)	right (pointer)	(optional) parent (pointer)

```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
    struct Node *parent; // optional
};
```

3. Memory Representation

Each node is a separate memory block, linked using pointers.



4. How It Works

1. The root node is stored at a memory location (e.g., 0x1000).
2. Its left and right pointers point to the memory addresses of its children (0x1100 and 0x1200).
3. Each child node simillarly contains pointers to its own left and right children.
4. Leaf nodes have their left and right pointers set to NULL.
5. The tree is built using dynamic memory allocation (e.g., malloc in C).

Pointer View (Example)

Each node stores the addresses of its children, not the actual child values.

Advantages

- Dynamic size (grows as needed).
- Does not require a fixed number of nodes.
- Easy to insert or delete nodes.
- Works well for sparse and irregular trees.

Disadvantages

- Uses more memory (pointers + data).
- Less efficient for complete binary trees (compared to array representation).
- More complex implementation.

Key Points to Remember

- Each node has data + pointers (to left, right, and optional parent).
- NULL is used when a child is missing.
- Tree is built using dynamic memory allocation.
- Works well for general (non-complete) trees.

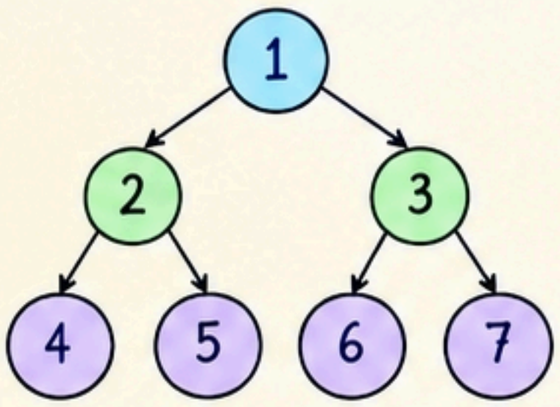
In Short: In linked list representation, a tree node is stored in a linked list node with pointers to its left and right children (and optionally to its parent).

Recursive and Non-Recursive Traversal of Binary Tree
(In Data Structure)

1. Recursive Traversal

- In recursive traversal, the function calls itself to visit the left and right subtrees.

Binary Tree Example



Traversal Orders:

- Preorder : 1 2 4 5 3 6 7
- Inorder : 4 2 5 1 6 3 7
- Postorder : 4 5 2 6 7 3 1

Recursive Functions (Pseudocode)

1. Preorder (Root → Left → Right)

```
void preorder(Node *root)
{
   if (root == NULL)
      return;
   printf("%d ", root->data);
   preorder(root->left);
   preorder(root->right);
}
```

2. Inorder (Left → Root → Right)

```
void inorder(Node *root){
{
   if (root == NULL)
      return;
   inorder(root->left);
   printf("%d ", root->data);
   inorder(root->right);
}
```

3. Postorder (Left → Right → Root)

```
void postorder(Node *root){
{
   if (root == NULL)
      return;
   postorder(root->left);
   postorder(root->right);
   printf("%d ", root->data);
}
```

Key Point:

- Each function first checks if the node is NULL.
- Then it visits the nodes in the order specified.
- It uses function calls (recursion) to go deeper into the tree.

Summary

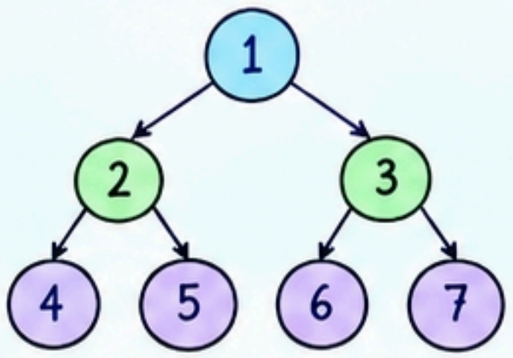
Traversal	Recursive (Function Call)	Non-Recursive (Stack)
Preorder	Root → Left → Right	Push right first, then left
Inorder	Left → Root → Right	Leftmost push, pop, then right
Postorder	Left → Right → Root	Use stack + last visited node

Both methods give the same result.
The choice depends on the requirement and the use case.

2. Non-Recursive Traversal

- In non-recursive traversal, we use a stack (or queue) instead of function calls to keep track of nodes.
- It avoids recursion and gives the same traversal order.

Binary Tree (same as above)



Traversal Orders (same):

- Preorder : 1 2 4 5 3 6 7
- Inorder : 4 2 5 1 6 3 7
- Postorder : 4 5 2 6 7 3 1

Non-Recursive Algorithms (Using Stack)

1. Preorder (Root → Left → Right)

```
push(root);
while (!stack.isEmpty()) {
   node = stack.pop();
   print(node->data);
   if (node->right) push(node->right);   // right first
   if (node->left) push(node->left);     // then left
}
```

Idea:
- Use a stack.
- Push right child first (so left is processed first).

2. Inorder (Left → Root → Right)

```
stack<Node*> s;
node = root;
while (node != NULL || !s.isEmpty()) {
   while (node != NULL) {
         s.push(node);
         node = node->left;
   }
   node = s.pop();
   print(node->data);
   node = node->right;
}
```

Idea:
- Go to the leftmost node and push all ancestors.
- Pop from stack, visit node, then go to right subtree.

3. Postorder (Left → Right → Root)

```
stack<Node*> s;
Node *lastVisited = NULL, *node = root;
while (node != NULL || !s.isEmpty()) {
   while (node != NULL) {
      s.push(node);
      node = node->left;
   }
   node = s.top();
   if (node->right == NULL || node->right == lastVisited) {
      print (node->data);
      lastVisited = node;
      s.pop();
   } else {
      node = node->right;
} }
```

Idea:
- Use a stack and a last visited pointer.
- Visit the node only after both subtrees are processed.

Key Point:

- Non-recursive traversal uses an explicit stack to maintain the traversal order.
- It produces the same output as the recursive approach.

Threaded Binary Tree
(In Data Structure)

1. Introduction

- A threaded binary tree is a binary tree in which NULL pointers are replaced by special links called **threads**.
- These threads point to the inorder **predecessor** or **inorder successor** of a node.
- It helps in traversing the tree without using recursion or a stack.

2. Why Threaded Binary Tree?

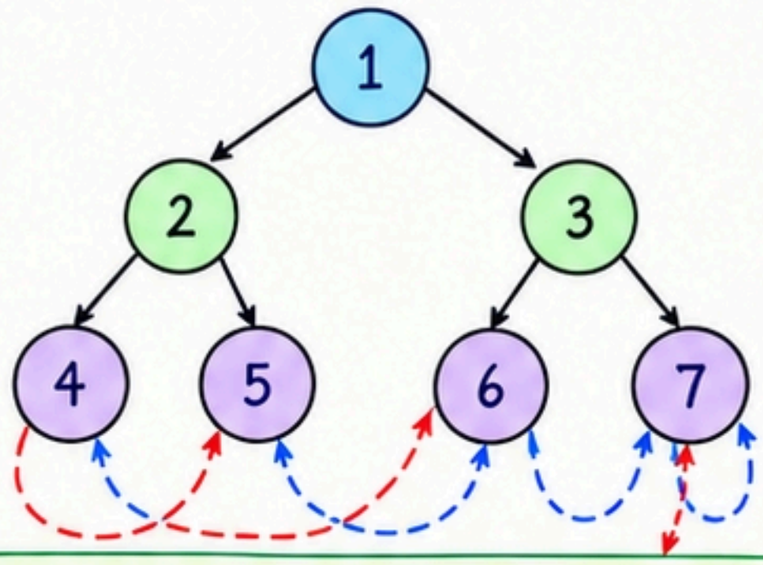
- In a normal binary tree, when we want to find the inorder successor or predecessor, we need to use a stack or recursion.
- Threaded binary tree allows us to get them directly using the threads, which **saves space and time.**
- It is useful for **inorder traversal** and other traversals.

3. Types of Threaded Binary Tree

1. **Single Threaded Binary Tree**
   - Only one type of thread is used (either for inorder predecessor or inorder successor).
2. **Double Threaded Binary Tree**
   - Two types of threads are used (one for inorder predecessor and one for inorder successor).

4. Example of Threaded Binary Tree

Consider the following binary tree:



Inorder traversal sequence: 4 2 5 1 6 3 7

5. Legend

- Left / Right child link
- Inorder predecessor thread (left thread)
- Inorder successor thread (right thread)
- X → Node
- → Thread link (replaces NULL)

6. Threaded Tree Representation

Node	Left Pointer	Right Pointer
1	2 (child)	3 (child)
2	4 (child)	5 (child)
3	6 (child)	7 (child)
4	2 (thread)	2 (thread)
5	2 (thread)	1 (thread)
6	3 (thread)	3 (thread)
7	3 (thread)	1 (thread)

7. Inorder Traversal Using Threads

1. Start from the leftmost node of the tree.
2. If the right pointer is a thread, follow it to the next node in inorder.
3. If the right pointer is a child, move to the leftmost node of that right subtree.
4. Repeat the process until all nodes are visited.

This way, we can perform inorder traversal without using a stack or recursion.

8. Advantages

- No extra memory for stack or recursion.
- Faster inorder traversal.
- Uses NULL pointers efficiently.
- Simple to implement.

9. Disadvantages

- Slightly complex insertion and deletion.
- Extra space for storing thread information.

10. Key Takeaway

A threaded binary tree replaces NULL pointers with threads to inorder predecessor or successor, allowing traversal without recursion or a stack. It is memory efficient and useful for inorder traversal.

Threads connect where NULL ends!

Binary Search Tree (BST)
(Inserting, Deleting and Searching in BST)

1. What is a Binary Search Tree?

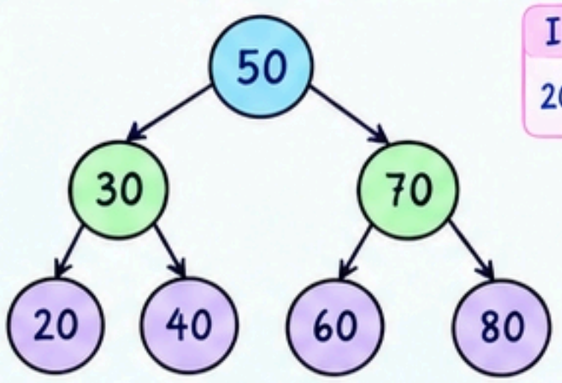
A Binary Search Tree (BST) is a binary tree in which each node has at most two children and follows the BST property:

> For any node, all values in the left subtree are smaller and all values in the right subtree are greater than the node's value.

Key Characteristics:

- Each node has at most two children.
- Left child < node < right child.
- Inorder traversal gives values in ascending order.
- No duplicate values (in a standard BST).

2. Example of a BST



Inorder Traversal:
20 30 40 50 60 70 80

BST Property Check:

- Left subtree of 50: 30, 20, 40 (all < 50)
- Right subtree of 50: 70, 60, 80 (all > 50)
- Left of 30: 20 < 30, 40 > 30 ✓
- Left of 70: 60 < 70, 80 > 70 ✓

3. Basic Operations in BST

1. **Insertion** – Add a new node while maintaining the BST property.
2. **Deletion** – Remove a node while maintaining the BST property.
3. **Searching** – Find a node with a given value.

Time Complexity (for balanced tree)

- Insertion : O(h)
- Deletion : O(h)
- Searching : O(h)

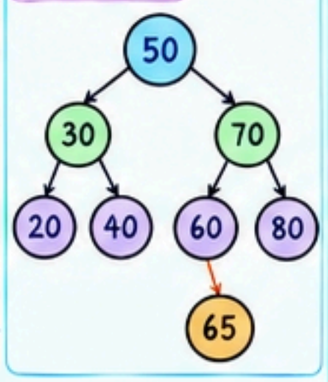
(h = height of tree; in balanced BST, h = O(log n))

4. Inserting a Node in BST

Steps:

1. Start from the root.
2. If the value is less, go to the left subtree.
3. If the value is greater, go to the right subtree.
4. Repeat until a NULL position is found, then insert the node.

Insert 65



Pseudocode (Insertion)

```
insert(root, key):
    if root == NULL:
        return new node(key)
    if key < root->data:
        root->left = insert(root->left, key)
    else if key > root->data:
        root->right = insert(root->right, key)
    return root
```

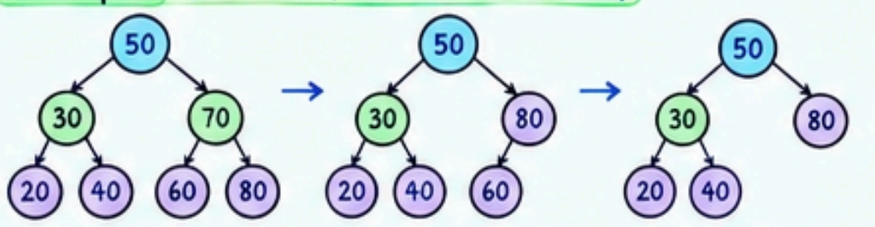
Note: Duplicate values are usually not inserted.

5. Deleting a Node in BST

Cases:

1. Leaf node (no children) – Simply delete it.
2. One child – Replace node with its child.
3. Two children – Replace with inorder successor (smallest in right subtree) or inorder predecessor (largest in left subtree), then delete that successor/predecessor.

Example: Delete 70 (node with two children)



Pseudocode (Deletion - simplified)

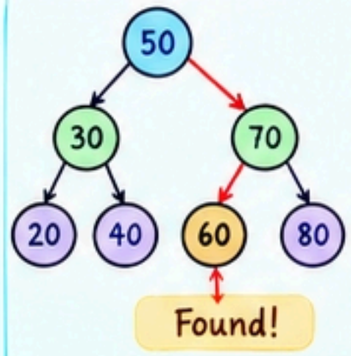
```
delete(root, key):
    if root == NULL: return NULL
    if key < root->data: root->left = delete(root->left, key)
    else if key > root->data: root->right = delete(root->right, key)
    else: // found the node
        handle 0, 1 or 2 children cases
    return root
```

6. Searching in BST

Steps:

1. Start from the root.
2. If the value is equal to the node, return found.
3. If the value is less, go to the left subtree.
4. If the value is greater, go to the right subtree.
5. Repeat until the value is found or NULL is reached.

Search for 60



Pseudocode (Searching)

```
search(root, key):
    if root == NULL:
        return false
    if key == root->data:
        return true
    if key < root->data:
        return search(root->left, key)
    else:
        return search(root->right, key)
```

Key Takeaways:

- BST maintains a sorted order (left < node < right).
- Insertion, deletion and searching are all based on comparison.
- Inorder traversal gives the sorted sequence.
- Time complexity is O(h) (O(log n) for balanced BST).