Introduction to Trees

Ajay Iyengar
Email: iyengarajay@gmail.com

Introduction to Trees

- Why Trees ? The limitation of Linear Data Structures
- Scenario 1 : Searching in a database
  - You have 1 million records in an array
  - Linear search: O(n) - potentially 1 million comparisons
  - Even with binary search: O(log n) - but insertion/deletion is O(n)
- Scenario 2 : Organizing a File System
  - How to represent folders and subfolders in an array or linked list ?
  - How do you efficiently represent hierarchy ?
  - How do you navigate parent-child relationships ?

Why Trees ?

- Scenario 3 : Auto-complete in Search Engines
  - As you type "prog", it suggests "program", "programming", "progress"
  - Linear structures would require scanning all words
  - Need something that can branch based on prefixes

The Problem & the Solution

Linear data structures (arrays, linked lists, stacks, queues) are excellent for sequential access but struggle with:

- Hierarchical relationships
- Efficient searching with dynamic data
- Representing multi-way branching
- Balancing speed of insertion and search

The Solution: Trees - a non-linear, hierarchical data structure.

Trees : Real World Analogies

Trees are everywhere in computing and real life –

1. File System : Computer's folder structure

```
C:\
 ├── Program Files\
 │   ├── Java\
 │   └── Microsoft\
 └── Users\
     └── Documents\
```

Trees : Real World Analogies

Organizational Hierarchy : Company Structures(CEO - VP's – Managers - Developers)

```
                            CEO
                        (John Wick)
                             |
        +--------------------+------------------+
        |                    |                  |
    VP Engineering        VP Sales           VP Finance
    (Darth Vader)        (Rocky Balboa)      (Gabbar Singh)
         |                     |                   |
    +----+----+           +----+----+       +------+-------+
    |         |           |         |       |              |
Engineering Engineering  Sales    Sales    Accounting  Treasury
Manager    Manager      Manager  Manager   Manager     Manager
(Team A)   (Team B)     (East)   (West)    (Amar)       (Prem)
   |         |            |         |
+--+--+   +-----+      +-----+    +------+
|     |   |     |      |     |    |      |
Dev  Dev Dev   Dev   Sales Sales  Sales Sales
1     2   3     4      1     2    3      4
```

Trees : Real World Analogies

```html
<!DOCTYPE html>
<html>
  <head>
    <title>My Page</title>
    <meta charset="UTF-8">
  </head>
  <body>
    <div id="header">
      <h1>Welcome</h1>
      <nav>
        <a href="/">Home</a>
        <a href="/about">About</a>
      </nav>
    </div>
    <div id="content">
      <p>Hello World!</p>
      <button>Click Me</button>
    </div>
  </body>
</html>
```

```
                     document
                         |
                      <html>
                         |
           +-------------+-------------+
           |                           |
        <head>                      <body>
           |                           |
     +-----+-----+          +----------+----------+
     |           |          |                     |
  <title>     <meta>   <div id="header">    <div id="content">
     |                      |                     |
"My Page"            +------+------+        +-----+-----+
                     |             |        |           |
                    <h1>         <nav>     <p>       <button>
                     |             |        |           |
                "Welcome"    +-----+-----+ "Hello    "Click Me"
                             |           | World!"
                            <a>         <a>
                             |           |
                          "Home"      "About"
```

Dom as a Tree

Trees : Real World Analogies

Decision Tree – AI/ML, Game Theory

```
CURRENT STATE
                        +---+---+---+
                        | X | X |   |
                        +---+---+---+
                        | O | O |   |
                        +---+---+---+
                        |   |   |   |
                        +---+---+---+
                              |
          +----------------+----------------+
          |                                 |
   OPTION 1: Play [0,2]              OPTION 2: Play [1,2]
   (Complete X's row)                (Block O's row)
          |                                 |
   +---+---+---+                     +---+---+---+
   | X | X | X | ✓ WIN!              | X | X |   |
   +---+---+---+                     +---+---+---+
   | O | O |   |                     | O | O | X | ✓ Blocked
   +---+---+---+                     +---+---+---+
   |   |   |   |                     |   |   |   |
   +---+---+---+                     +---+---+---+
          |                                 |
   GAME OVER                         O's Next Move
   X WINS!                           O plays [0,2]
   Score: +10                               |
   ✓ BEST CHOICE                     +---+---+---+
                                     | X | X | O |
                                     +---+---+---+
                                     | O | O | X |
                                     +---+---+---+
                                     |   |   |   |
                                     +---+---+---+
                                            |
                                     Game continues...
                                     X may lose or draw
                                     Score: 0 or -10
                                     X BAD CHOICE
```

Basic Tree Terminology

```
        A (Root)
       / \
      B   C
     / \   \
    D   E   F
   /
  G (Leaf)
```

- **Node**: Each element in the tree (A, B, C, D, E, F, G)
- **Root**: The topmost node with no parent (A)
- **Parent**: Node that has children (A is parent of B and C)
- **Child**: Node that has a parent (B and C are children of A)

Basic Tree Terminology

```
        A (Root)
       / \
      B   C
     / \   \
    D   E   F
   /
  G (Leaf)
```

- **Siblings**: Nodes with the same parent (B and C are siblings)
- **Leaf/External Node**: Node with no children (E, F, G)
- **Edge**: Connection between two nodes (there are 6 edges above)

Basic Tree : Measurement terms

```
        A (Root)
       / \
      B   C
     / \   \
    D   E   F
   /
  G (Leaf)
```

- **Depth of a Node**: Number of edges from root to that node
  - Depth of A = 0
  - Depth of B = 1
  - Depth of G = 3
- **Height of a Node**: Number of edges on longest path from that node to a leaf
  - Height of G = 0 (leaf)
  - Height of D = 1
  - Height of B = 2
  - Height of A = 3

Height of Tree: Height of the root node (3 in above example)

Types of Trees : General Tree(N-ary Tree)

- A tree where each node can have any number of children (0 to N).

```
      A
    / | \
   B  C  D
  /|  |  |\
 E F  G  H I
      |
      J
```

- **Properties**:
  - No restriction on number of children
  - Most flexible tree structure
  - Used in file systems, organizational charts
- **Use Case**: File system directories where a folder can contain any number of subfolders and files.

Binary Trees

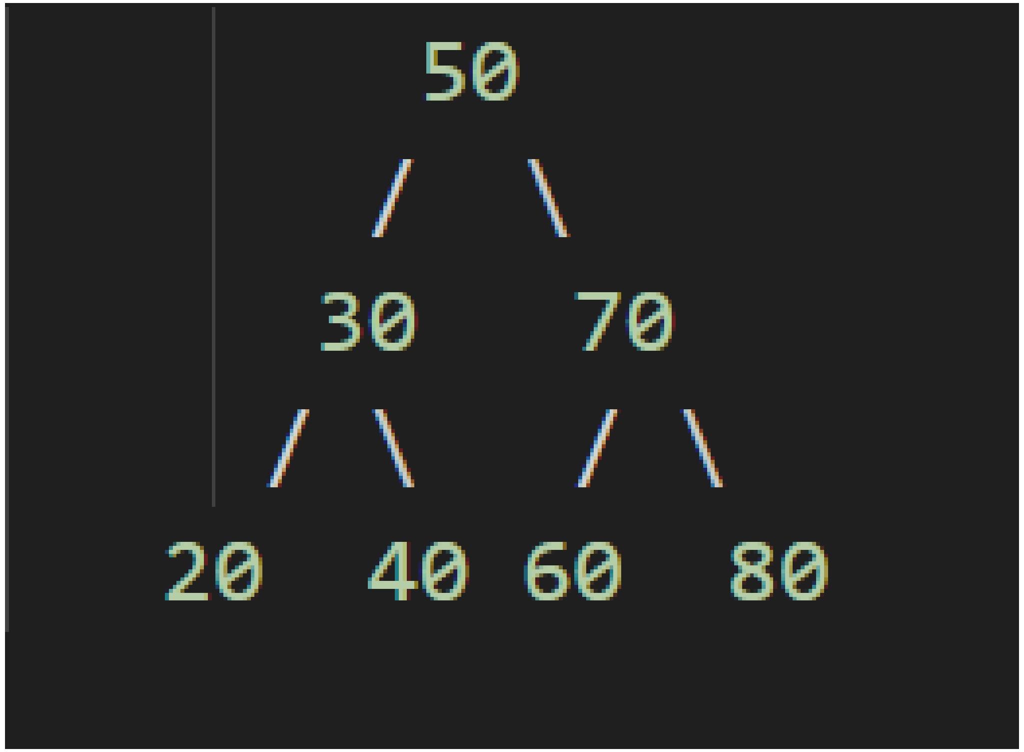
- A tree where each node has at most 2 children (left child and right child).

```
      10
     /  \
    5    15
   / \     \
  3   7     20
 /
1
```

- **Properties**:
  - Each node has 0, 1, or 2 children
  - Children are labeled as left and right
  - Order matters: left ≠ right
- **Use Case**: Expression trees, Huffman coding trees, basic tree operations learning.

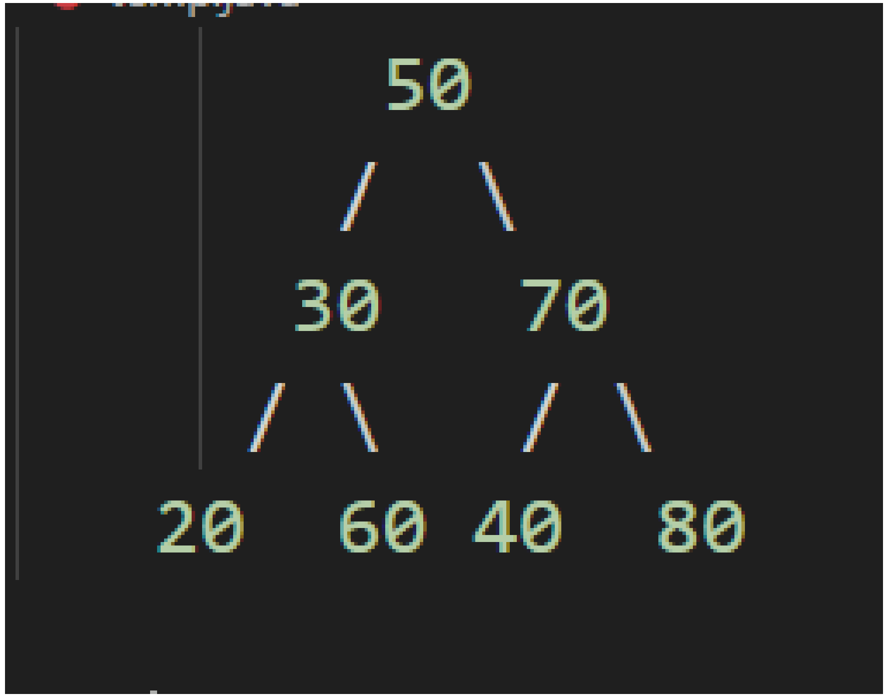
Binary Search Trees*

- A binary tree where for each node:
  - All values in the left subtree are less than the node's value
  - All values in the right subtree are greater than the node's value
  - Both left and right subtrees are also BSTs



Use cases: Databases indexing, searching with dynamic insertions/deletions, symbol tables in compilers.

Not A Binary Search Tree (BST)



60 in left subtree but 60 > 50 - violates BST property &
40 in right subtree – violates BST property

Full Binary Tree (Strict Binary Tree)

- A binary tree where every node has either 0 or 2 children. No node has exactly 1 child
- **Properties**
  - Every internal node has exactly 2 children
  - All leaves can be at different levels
  - If there are L leaves, there are (L-1) internal nodes

```
Valid Full Binary Tree:

    10
   /  \
  5    15
 / \   / \
3   7 12  20

Not a Full Binary Tree:

    10
   /  \
  5    15
 /      \
3       20
(Nodes 5 and 15 have only 1 child each -
violates full binary tree property)
```

Perfect Binary Tree

- A binary tree where all internal nodes have exactly 2 children **AND** all leaf nodes are at the same level.
- **Properties**
  – Most restrictive binary tree type
  – All Perfect trees are also Full and Complete
  – Perfectly balanced

```
Perfect Binary Tree:

      10
     /  \
    5    15
   / \  / \
  3   7 12  20

NOT Perfect (but Full):
     10
    /  \
   5   15
  / \
 3   7
(This is Full but not Perfect -
leaves at different levels)
```

Complete Binary Tree

- A binary tree where **all levels are completely filled except possibly the last level**, and **the last level is filled from left to right**.

```
Complete Binary Tree:
        1
      /   \
     2     3
    / \   /
   4   5 6

Last level filled from left to right

Not a Complete Binary Tree:
        1
      /   \
     2     3
      \
       5
Left child missing, right child exists
```

Some books & articles may refer to the Same as Almost Complete Binary Trees

Array representation : Complete Binary Trees

```
Complete Binary Tree:
        10
      /    \
     20     30
    /  \   /
  40  50  60

Stored in Array (level order):

Index:  0   1   2   3   4   5
Array: [10, 20, 30, 40, 50, 60]

1. Root first
2. Then its children
3. Then grandchildren
4. Exactly how arrays store data
```

```
Index relationship:
Left child  = 2*i + 1
Right child = 2*i + 2
Parent      = (i - 1) / 2

Index:  0   1   2   3   4   5
Array: [10, 20, 30, 40, 50, 60]


Index:      0
           [10]
          /     \
Index:   1        2
       [20]     [30]
       /  \      /
Index: 3    4   5
     [40] [50] [60]
```

Array representation : Works only for Complete Binary Trees

```
Not Complete:

        10
      /    \
     20     30
      \
       50

Attempted Array:
Index:  0   1   2   3   4
Array: [10, 20, 30, ?, 50]

1.Gap at index 3
2.Index formulas break
3.Wasted space
```

Skewed Tree

- A tree where each parent node has only one child. Essentially becomes a linked list.

```
Left Skewed:

        10
       /
      5
     /
    3
   /
  1
```

```
Right Skewed:

10
  \
   15
     \
      20
        \
         25
```

- **Properties:**
  - Worst case scenario for search operations
  - O(n) time complexity for all operations
  - Height = Number of nodes - 1

This is what happens to an unbalanced BST when you insert sorted data. Understanding this motivates the need for self-balancing trees.

Balanced Trees

- Balanced trees maintain height ≈ log(n) to ensure O(log n) operations even in worst case.

```
AVL Tree

      30
     /  \
   20    40
  /        \
10          50
```

- **AVL Tree**:
  - Self-balancing BST where the height difference between left and right subtrees (balance factor) is at most 1 for every node
  - Each node's |height(left) - height(right)| ≤ 1

Balanced Trees

- **Red-Black Tree**:
  - Self-balancing BST with color properties (red/black nodes) ensuring tree remains approximately balanced
- **Use Cases**:
  - Java's TreeMap, TreeSet, Linux kernel scheduling, C++ STL map.

- **Properties**
  - Every node is either red or black
  - Root is always black
  - Red nodes cannot have red children
  - Every path from root to leaf has same number of black nodes

B-Tree

- Self-balancing search tree where nodes can have multiple keys and more than 2 children. Designed for disk storage systems

```
        [30, 60]
       /    |    \
      /     |     \
[10,20]  [40,50]  [70,80,90]
```

- **Properties**
  - Each node can contain multiple keys (not just one)
  - All leaves at the same level
  - Optimized for systems that read/write large blocks of data
  - Minimizes disk I/O operations

- **Use cases:** Database indexing (MySQL, PostgreSQL use B+ Trees), file systems (NTFS, ext4)

Trie

- Tree structure where each node represents a character, used for efficient string storage and retrieval
- Storing "cat", "car", "card", "dog":

```
        root
       /    \
      c      d
     /        \
    a          o
   / \          \
  t   r          g
      |
      d
```

- **Properties**
  - Each path from root to node represents a prefix
  - Common prefixes share paths
  - Fast string lookup: O(L) where L is string length

- **Use cases :** Auto-complete, spell checkers, dictionary implementations.

BST Operations

- A **Binary Search Tree (BST)** is a special type of binary tree where each node follows a simple rule:

  Left child < Parent node < Right child

- This property must be true for **every node** in the tree
- This makes BSTs incredibly efficient for searching, insertion, and deletion operations.
- Unlike arrays or linked lists where searching takes O(n) time, a well-balanced BST allows you to search for elements in O(log n) time - similar to binary search!

Creating a BST

Let's say we want to insert these numbers in order: **50, 30, 70, 20, 40, 60, 80**

```
1. Initial: Insert 50      2. After inserting 30:
        50                         50
                                  /
                                30

3. After inserting 70:     4. After inserting 20:
        50                       50
       /  \                     /  \
     30   70                  30    70
                             /
                           20

5. After inserting 40:     6. Final tree (after 60 and 80):
        50                         50
       /  \                       /  \
     30    70                   30    70
    /        \                 /  \  /  \
   20         40              20  40 60  80
```

Creation of BST : Node class

```
    50
   /  \
  30   70
 /  \  /  \
20  40 60  80
```

```java
class Node {
    int data;
    Node left;
    Node right;

    // Constructor to create a new node
    public Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

Notice how smaller values always go left, larger values always go right!

Example of how iterative insertion works

```
Inserting 40 into the following Binary Search Tree (BST):

       50
      /  \
    30    70
    /
  20

       50
      /  \
    30    70
   /  \
  20   40   ← New node inserted here!

Step-by-step:
1.Create newNode with value 40
2.Start at root (50)
3.40 < 50 → Move to left child (30)
4.40 > 30 → Move to right child (null)
5.Found empty spot! Insert 40 as right child of 30
```

BST Creation : Time & Space Complexity

- **Time Complexity:**
  - **Best Case**: O(log n) - when the tree is balanced
  - **Average Case**: O(log n)
  - **Worst Case**: O(n) - when the tree becomes a skewed line (e.g., inserting sorted data)
- **Space Complexity:**
  - **O(1)** - Iterative approach uses constant extra space (just a few pointers)
  - The tree itself uses O(n) space for n nodes

BST : Inorder Traversal

- Inorder traversal visits nodes in this order:

  Left → Root → Right

- For a BST, this gives us nodes in ****sorted order****!

- Inorder will visit the nodes in this order: 1 3 5 6 7 9

```
      5
     / \
    3   7
   /   / \
  1   6   9
```

BST : Inorder traversal (*Recursion)

```java
public void inorderTraversal(Node root) {
    if (root == null) {
        return;
    }

    inorderTraversal(root.left);   // Visit left subtree
    System.out.print(root.data + " "); // Process current node
    inorderTraversal(root.right);  // Visit right subtree
}
```

Inorder using Recursion
How does it work ?

- Each function call executes ALL its lines
- Recursive calls PAUSE the current function
- Functions RESUME from where they paused
- The recursion naturally goes deep left first
- The call stack remembers where to return AND what to do next

Refer to inorder-traversal-notes for detailed step by step explanation

Inorder Recursion Time & Space Complexity

- **Time Complexity:** O(n) - We visit each node exactly once
- **Space Complexity:** O(h) - Maximum call stack depth is the height of the tree
  - Best case (balanced): O(log n)
  - Worst case (skewed): O(n)

BST : Preorder Traversal

- Preorder traversal visits nodes in this order:

  Root → Left → Right

- The key difference: we process the node first, followed by left subtree & then right subtree
- Preorder will visit the nodes in this order : 5 3 1 7 6 9

```
      5
     / \
    3   7
   /   / \
  1   6   9
```

BST : Postorder Traversal

- Postorder traversal visits nodes in this order:

  Left → Right → Root

- The key difference: we process the node LAST, after both subtrees
- Postorder will visit the nodes in this order : 1 3 6 9 7 5

```
      5
     / \
    3   7
   /   / \
  1   6   9
```

BST : Postorder traversal (*Recursion)

```java
public void postorderTraversal(Node root) {
    if (root == null) {
        return;
    }

    postorderTraversal(root.left);   // Visit left subtree
    postorderTraversal(root.right);  // Visit right subtree
    System.out.print(root.data + " "); // Process current node LAST
}
```

Postorder using Recursion
How does it work ?

- The root prints LAST
- Each function waits through TWO recursive calls
- The "pause and resume" happens TWICE per node
- Children are always processed before parents
- The call stack manages the "waiting"

Refer to postorder-traversal-notes for detailed step by step explanation

Postorder Recursion
Time & Space Complexity

- **Time Complexity:** O(n) - We visit each node exactly once
- **Space Complexity:** O(h) - Maximum call stack depth is the height of the tree
  - Best case (balanced): O(log n)
  - Worst case (skewed): O(n)

Height of a Binary Tree

- The **height of a tree** is the longest path from the root to any leaf node (counting edges).
- This is a classic example of **bottom-up recursion** - we calculate heights from leaves and bubble up to the root!
- The height of the tree is 2

```
    5
   / \
  3   7
 /   / \
1   6   9
```

Code : Height of a Binary Tree

```java
public int height(Node root) {
    if (root == null) {
        return -1;  // Base case: null has height -1
    }

    int leftHeight = height(root.left);     // Get left subtree height
    int rightHeight = height(root.right);   // Get right subtree height

    return 1 + Math.max(leftHeight, rightHeight); // Current height
}
```

The key insight: **Height = 1 + max(left height, right height)**

Why the formula works ?

```
return 1 + Math.max(leftHeight, rightHeight);
       ↑              ↑
       |              └ Which path is longer?
       └ Add current edge/node
```

- If left is taller, we take that path and add 1 for current node
- If right is taller, we take that path and add 1 for current node
- If both equal (or both null), we take either and add 1

Height of a Binary Tree using Recursion

- **This is postorder recursion** - We process children FIRST, then use their results to calculate parent's value
- **Base case is critical** - height(null) = -1 makes the math work for leaf nodes (1 + max(-1, -1) = 0)
- **The + 1 counts the current edge** - We add 1 for the edge from current node to its taller subtree
- **The max picks the taller path** - We want the LONGEST path, so we take the maximum height
- **Each node waits for BOTH children** - Just like postorder traversal, we need both recursive calls to complete

Height of Binary Tree : Time & Space Complexity

- **Time Complexity:** O(n) - We visit each node exactly once
- **Space Complexity:** O(h) – Call stack depth equals tree height
  - Best case (balanced): O(log n)
  - Worst case (skewed): O(n)