Chapter 1 – Introduction to Trees

[Definition and Properties of Trees], [Differences Between Trees and Graphs], [Terminology: Node, Root, Parent, Child, Sibling, Leaf, Subtree, Depth, Height, Path]

Definition and Properties of Trees

- **Definition**: A tree is a hierarchical data structure consisting of nodes connected by edges. Each tree has a single root node and a single path from the root connects every other node.
- **Properties**:
  - **Hierarchical Structure**: Represents a hierarchy where a parent node points to child nodes.
  - **Acyclic**: Trees do not contain cycles.
  - **Connected**: All nodes are connected by edges.

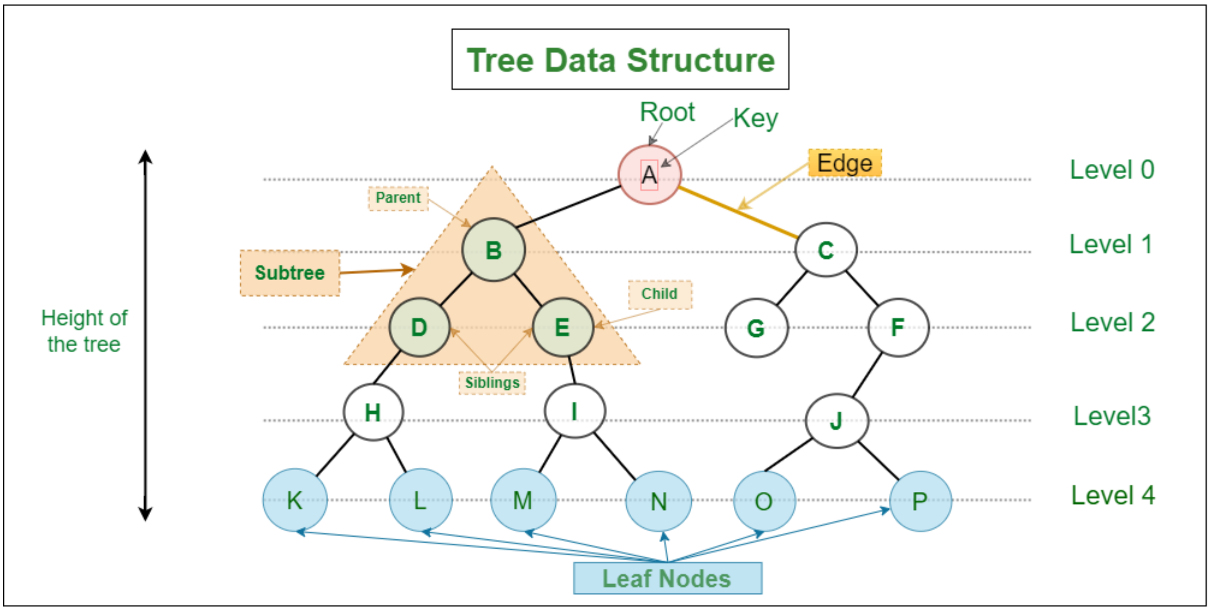
Differences Between Trees and Graphs

- **Trees**:
  - **Acyclic**: Trees do not contain cycles.
  - **Single Path**: There is exactly one path between any two nodes.
  - **Rooted**: Trees have a designated root node.
  - **Hierarchical Structure**: Represents parent-child relationships.
- **Graphs**:
  - **Cycles Allowed**: Graphs can contain cycles.
  - **Multiple Paths**: There can be multiple paths between nodes.
  - **No Root Required**: Graphs do not require a root node.
  - **Network Structure**: Represents general connections between nodes.

Terminology

- **Node**: Basic unit of a tree. Represents an element or a point in the tree.
- **Root**: The topmost node of a tree, where traversal begins. It has no parent.
- **Parent**: A node that has one or more child nodes.
- **Child**: A node that has a parent node.
- **Sibling**: Nodes that share the same parent.
- **Leaf**: A node with no children. It is at the end of a path.
- **Subtree**: A tree consisting of a node and all its descendants.
- **Depth**: The number of edges from the root to a node. The root has a depth of 0.
- **Height**: The number of edges in the longest path from a node to a leaf. The height of a tree is the height of the root.

- **Path**: A sequence of nodes where each adjacent pair is connected by an edge.



Root – A

Child – B, A, D, E, C, G, F, J, O, P, H, I, K, L, M, N

Leaf Nodes – K, L, M, N, O, P

Height – 4

Edges/ Path – 15

Parent – A, B, C, F, D, E, H, I, J

Node – A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P

Siblings – {D, E}, {G, F}, {B, C}, {K, L}, {M, N}, {O, P}

Chapter 2 – Introduction to Trees

[General Tree, Binary Tree, Binary Search Tree (BST), AVL Tree, Red-Black Tree, Splay Tree, B-Tree and B+ Tree, Trie (Prefix Tree), Segment Tree, Fenwick Tree (Binary Indexed Tree), N-ary Tree]

General Tree

- **Definition**: A tree where each node can have an arbitrary number of children.
- **Example**: A family tree where a person can have multiple children.
- **Elaboration**: General trees are flexible and can represent various hierarchical structures, but they lack the constraints and properties of more specific types of trees.

Binary Tree

- **Definition**: A tree where each node has at most two children, referred to as the left child and the right child.
- **Example**: A binary tree can represent hierarchical data, like a company's organizational chart with a CEO, managers, and employees.
- **Elaboration**: Binary trees are foundational in many computer science applications due to their simplicity and versatility.

Subtypes of Binary Tree:

1. **Full Binary Tree**

- **Definition**: Every node has either 0 or 2 children.
- **Example**:

```
      1
     / \
    2   3
   / \
  4   5
```

- **Elaboration**: In a full binary tree, all nodes contribute to the binary structure with either two children or none.

2. **Complete Binary Tree**

- **Definition**: All levels are filled except possibly the last, which is filled from left to right.
- **Example**:

```
    1
   / \
  2   3
 / \
4   5
```

- **Elaboration**: Complete binary trees ensure efficient use of space and are often used in heaps.

3. **Perfect Binary Tree**

- **Definition**: All internal nodes have exactly two children and all leaf nodes are at the same level.
- **Example**:

```
     1
   /   \
  2     3
 / \   / \
4   5 6   7
```

- **Elaboration**: Perfect binary trees are a subset of complete binary trees and are highly symmetrical.

4. **Balanced Binary Tree**

- **Definition**: The height of the tree is minimized to ensure the tree remains balanced.
- **Example**:

```
     1
    / \
   2   3
  /   / \
 4   5   6
```

- **Elaboration**: Balanced trees maintain efficient operations for insertion, deletion, and lookup.

Binary Search Tree (BST)

- **Definition**: A binary tree where the left child contains only nodes with values less than the parent node, and the right child contains only nodes with values greater than the parent node.
- **Example**:

```
    4
   / \
  2   6
 / \ / \
1  3 5  7
```

- **Elaboration**: BSTs allow for efficient searching, insertion, and deletion operations.

AVL Tree

- **Definition**: A self-balancing binary search tree where the difference in heights between the left and right subtrees of any node is at most one.
- **Example**:

```
    3
   / \
  2   4
 /
1
```

- **Elaboration**: AVL trees automatically maintain their balance using rotations, ensuring O (log n) time complexity for operations.

Red-Black Tree

- **Definition**: A self-balancing binary search tree where each node contains an extra bit for denoting the color of the node, either red or black, with specific properties to ensure balance.
- **Example**:

```
     B
    / \
   R   R
  / \   \
 B   B   B
```

- **Elaboration**: Red-Black trees provide a balance between perfect balancing and easier insertion and deletion operations compared to AVL trees.

Splay Tree

- **Definition**: A self-adjusting binary search tree where recently accessed elements are moved to the root using rotations.
- **Example**:

```
Initial Tree:     Access 3:
    4               3
   / \             / \
  2   5           2   4
 / \                 / \
1   3               1   5
```

- **Elaboration**: Splay trees provide amortized O(log n) time complexity by ensuring frequently accessed nodes are quick to reach.

B-Tree and B+ Tree

B-Tree:

- **Definition**: A self-balancing search tree in which nodes can have multiple children, ensuring that the tree remains balanced.
- **Example**: Used in databases and file systems.
- **Elaboration**: B-Trees maintain balance with a higher branching factor, minimizing disk reads.

B+ Tree:

- **Definition:** A type of B-Tree where all values are stored in leaf nodes and internal nodes only store keys.
- **Example:**

```
Internal Nodes:    10
                  /  \
Leaf Nodes:     5 7 10 12
```

- **Elaboration:** B+ Trees are optimized for systems that read large blocks of data.

Trie (Prefix Tree):

- **Definition**: A tree-like data structure that stores a dynamic set of strings, where the keys are usually strings. Each node represents a common prefix shared by some strings.
- **Example**:

```
     Root
     /  \
    a    t
   / \    \
  n   p    r
 /   /      \
d   p        i
            / \
           e   e
```

This trie stores the words: "and", "an", "app", "trie", and "trie".

- **Elaboration**: Tries are useful for implementing dictionaries with quick lookup, insert, and delete operations. Each node in a trie represents a single character of the string, and the path from the root to a particular node represents a prefix of one or more strings.

Chapter 3: Tree Traversals

Depth-First Search (DFS)

DFS explores as far down a branch as possible before backtracking.

1. In-order Traversal

- **Definition**: Traverse the left subtree, visit the root node, then traverse the right subtree.
- **Method**:
    1. Visit the left subtree.
    2. Visit the root node.
    3. Visit the right subtree.
- **Example**:

```
    4
   / \
  2   5
 / \
1   3
```

    - **In-order Traversal**: 1, 2, 3, 4, 5

2. Pre-order Traversal

- **Definition**: Visit the root node, traverse the left subtree, then traverse the right subtree.
- **Method**:
    1. Visit the root node.
    2. Visit the left subtree.
    3. Visit the right subtree.
- **Example**:

```
    4
   / \
  2   5
 / \
1   3
```

    - **Pre-order Traversal**: 4, 2, 1, 3, 5

3. Post-order Traversal

- **Definition**: Traverse the left subtree, traverse the right subtree, then visit the root node.
- **Method**:
    1. Visit the left subtree.
    2. Visit the right subtree.
    3. Visit the root node.
- **Example**:

```
    4
   / \
  2   5
 / \
1   3
```

- **Post-order Traversal**: 1, 3, 2, 5, 4

Breadth-First Search (BFS)

BFS explores all nodes at the present depth level before moving on to nodes at the next depth level.

Level-order Traversal

- **Definition**: Traverse the tree level by level from left to right.
- **Method**:
    1. Start at the root.
    2. Visit all nodes at the current level before moving to the next level.
- **Example**:

```
    4
   / \
  2   5
 / \
1   3
```

- **Level-order Traversal**: 4, 2, 5, 1, 3

Summary:

- **In-order (DFS)**: 1, 2, 3, 4, 5
- **Pre-order (DFS)**: 4, 2, 1, 3, 5
- **Post-order (DFS)**: 1, 3, 2, 5, 4
- **Level-order (BFS)**: 4, 2, 5, 1, 3

These traversal methods allow you to visit and process each node in a tree in a specific order, which is useful for various tree-related algorithms and applications.

Chapter 4 – Binary Search Trees (BST)

Definition and Properties

- **Definition**: A Binary Search Tree is a binary tree in which each node has a key, and it satisfies the following properties:
  - The key in the left subtree of a node is less than the node's key.
  - The key in the right subtree of a node is greater than the node's key.
  - Both the left and right subtrees must also be binary search trees.
- **Properties**:
  - **Ordering**: Left child < Parent < Right child
  - **Uniqueness**: All keys are distinct.
  - **Sorted**: In-order traversal yields sorted keys.

Operations

1. Insertion

- **Method**:
  1. Start at the root.
  2. Compare the key to be inserted with the root's key.
  3. If the key is less, go to the left subtree; if greater, go to the right subtree.
  4. Repeat the process until you find an empty spot.
  5. Insert the new node there.
- **Example**: Inserting `8` into the BST:

```
    10                    10
   /  \                  /  \
  5    15    ->         5    15
 / \                   / \
3   7                 3   7
                           \
                            8
```

2. Deletion

- **Method**:
  1. **Case 1: The node to be deleted has no children** (leaf node).
  2. **Case 2: The node to be deleted has one child**.
  3. **Case 3: The node to be deleted has two children**:
     - Find the in-order successor (smallest node in the right subtree).
     - Replace the node's key with the in-order successor's key.
     - Delete the in-order successor.
- **Example**: Deleting `5` from the BST:

```
    10                    10
   /  \                  /  \
  5    15    ->         7    15
 / \                   /
3   7                 3
```

3. Search

- **Method**:
    1. Start at the root.
    2. Compare the key to be searched with the root's key.
    3. If the key is less, go to the left subtree; if greater, go to the right subtree.
    4. Repeat until the key is found or an empty subtree is reached.
- **Example**: Searching for `7` in the BST:

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

    - Start at `10`, go left to `5`, then right to `7`.

Time Complexity Analysis

- **Best Case**: O (log n) - Occurs when the tree is balanced.
- **Average Case**: O (log n) - Assumes random insertion order.
- **Worst Case**: O(n) - Occurs when the tree becomes a linked list (completely unbalanced).

Summary:

- **Insertion**: O (log n) average, O(n) worst case
- **Deletion**: O (log n) average, O(n) worst case
- **Search**: O (log n) average, O(n) worst case

Binary Search Trees are fundamental data structures that support efficient insertion, deletion, and search operations, making them ideal for implementing dynamic sets and lookup tables.

Chapter 5 – Self-Balancing Trees

AVL Trees

- **Definition**: AVL trees are self-balancing binary search trees where the difference in heights between the left and right subtrees of any node (called the balance factor) is at most 1.
- **Rotations: Single and Double Rotations**
  - **Single Rotation**:
    - **Left Rotation**: Used to balance the tree when the balance factor of a node becomes +2 (right-heavy). It promotes the right child to be the new root, moving the old root to the left of the new root's left child.
    - **Right Rotation**: Used to balance the tree when the balance factor of a node becomes -2 (left-heavy). It promotes the left child to be the new root, moving the old root to the right of the new root's right child.
  - **Double Rotation**:
    - Combines two single rotations to balance the tree in more complex cases.
- **Insertion and Deletion**
  - **Insertion**: Perform a standard BST insertion, then check and adjust the balance factors up the tree. If necessary, perform rotations to maintain AVL balance.
  - **Deletion**: Perform a standard BST deletion, then check and adjust the balance factors up the tree. If necessary, perform rotations to restore AVL balance.
- **Example**: Inserting `15` into an AVL tree:

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

After insertion of 15 and balancing:
    10
   /  \
  5    20
 / \   /
3   7 15
```

Red-Black Trees

- **Definition**: Red-Black trees are self-balancing binary search trees where each node has an extra bit for colour (red or black), satisfying specific properties to ensure balanced structure.
- **Properties and Colouring Rules**
  - **Properties**:
    - The root is black.
    - Red nodes have black children.
    - Every path from a node to its descendant NIL node must have the same number of black nodes (black height).
  - **Insertion and Deletion Operations**:
    - Adjust node colors and perform rotations (left and right) to maintain red-black properties.

- **Example**:

```
Inserting 30 into a Red-Black tree:

    10 (black)
   /  \
  5    20 (black)
 / \   / \
3   7 15 25 (red)
           \
            28

After insertion of 30 and balancing:

    10 (black)
   /  \
  5    20 (black)
 / \   / \
3   7 15 28 (black)
         / \
         25 30 (red)
```

Summary

- **AVL Trees**: Maintain balance based on height difference with rotations.
- **Red-Black Trees**: Maintain balance with color rules and rotations.

Both AVL trees and Red-Black trees ensure efficient operations while maintaining balanced tree structures, crucial for maintaining logarithmic time complexity for operations in worst-case scenarios.

Advanced Trees

Segment Trees (Construction, Query, and Update Operations), Fenwick Trees (Binary Indexed Trees), (Construction, Query, and Update Operations), Tries [Insertion, Deletion, and Search], [Applications: Prefix Matching, Autocomplete], B-Trees and B+ Tree, [Properties and Structure], [Insertion, Deletion, and Search Operations]

Tree Applications

(Expression Trees) [Construction from Infix, Prefix, and Postfix Expressions], (Huffman Trees), [Huffman Coding Algorithm], [Applications in Data Compression], (Decision Trees), [Construction and Use in Machine Learning], (Suffix Trees),[Construction and Applications in String Matching]