

# Trees Cheat Sheet (Theory Only)

---

## 1. Basic Tree Terminologies

**Tree:** A hierarchical data structure consisting of nodes connected by edges.

**Root:** The topmost node in a tree.

**Parent & Child Nodes:** A node directly connected to another node is called a **parent**, and the connected node is a **child**.

**Sibling Nodes:** Nodes sharing the same parent.

**Leaf Node:** A node with no children.

**Height of Tree:** The longest path from the root to a leaf node.

**Depth of Node:** The distance from the root to a given node.

**Subtree:** A tree within another tree.

**Degree of Node:** The number of children a node has.

---

## 2. Types of Trees

### (A) Binary Tree

Each node has at most two children → left and right.

**Types of Binary Trees:**

- **Full Binary Tree:** Every node has 0 or 2 children.
  - **Complete Binary Tree:** All levels are completely filled, except possibly the last level.
  - **Perfect Binary Tree:** All internal nodes have 2 children, and all leaf nodes are at the same level.
  - **Balanced Binary Tree:** The difference in height between left and right subtrees is at most 1.
- 

### (B) Threaded Binary Tree

Uses special pointers to make in-order traversal faster.

Instead of NULL pointers, unused pointers store references to in-order successor/predecessor.

**Types:**

- **Single Threaded** → Only one type of thread (either left or right).
  - **Double Threaded** → Threads for both in-order successor and predecessor.
- 

### (C) Binary Search Tree (BST)

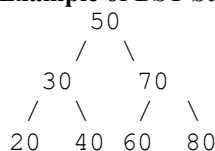
A binary tree with an ordering property:

- **Left subtree** contains values smaller than the root.
- **Right subtree** contains values greater than the root.

**Operations in BST:**

- **Insertion:** Insert elements following BST rules.
- **Searching:** Compare elements with root recursively.
- **Deletion:** Remove a node while maintaining BST structure.

**Example of BST Structure:**



**Example of Searching in BST:**

```
class Node:
```

```

def __init__(self, key):
    self.key = key
    self.left = self.right = None

def search(root, key):
    if not root or root.key == key:
        return root
    if key < root.key:
        return search(root.left, key)
    return search(root.right, key)

# Example Usage
root = Node(50)
root.left = Node(30)
root.right = Node(70)
print(search(root, 30)) # Output: Node with key 30

```

---

### 3. Binary Tree Traversals

Traversals define the way to visit all nodes in a tree.

Types of Tree Traversals:

#### (A) Depth-First Search (DFS) Traversals

1st Inorder (Left  $\rightarrow$  Root  $\rightarrow$  Right)

- Example: (20, 30, 40, 50, 60, 70, 80)
- Used in Binary Search Trees (BSTs).

```

def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

```

2nd

3rd Preorder (Root  $\rightarrow$  Left  $\rightarrow$  Right)

- Example: (50, 30, 20, 40, 70, 60, 80)
- Used in tree cloning, expression trees.

```

def preorder(root):
    if root:
        print(root.key, end=" ")
        preorder(root.left)
        preorder(root.right)

```

4th

5th Postorder (Left  $\rightarrow$  Right  $\rightarrow$  Root)

- Example: (20, 40, 30, 60, 80, 70, 50)
- Used in deleting trees.

```

def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.key, end=" ")

```

---

#### (B) Breadth-First Search (BFS) – Level Order Traversal

**Visits nodes level by level.**

**Implemented using a queue.**

**Example for Level Order Traversal:**

```
from collections import deque

def level_order(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.key, end=" ")
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

---

## 4. AVL Tree (Self-Balancing Binary Search Tree)

**Automatically balances itself after insertion & deletion.**

**For every node, the difference in height of left & right subtrees is at most 1.**

**Rotations are used to balance the tree:**

- **Right Rotation (LL Rotation)**
- **Left Rotation (RR Rotation)**
- **Left-Right Rotation (LR Rotation)**
- **Right-Left Rotation (RL Rotation)**

**Example of AVL Tree Rotations:**

Before Rotation:

```
      30
     /
    20
   /
  10
```

After Right Rotation:

```
      20
     / \
    10  30
```

---

## 5. Red-Black Tree

**A self-balancing BST with extra rules.**

**Each node has a color: Red or Black.**

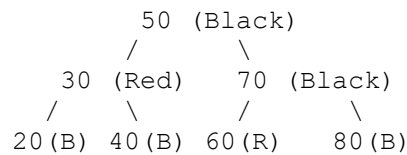
**Rules for Red-Black Trees:**

- Every node is either **Red** or **Black**.
- **Root** is always **Black**.
- **Red** nodes cannot have **Red** children (No two consecutive **Red** nodes).
- Every path from root to leaf must have the same number of **Black** nodes.
- Newly inserted nodes are always **Red**.

**Operations in Red-Black Tree:**

- **Insertion & Deletion** involve rotations to maintain balance.
- **Faster than AVL** in insertion and deletion.

**Example Structure of a Red-Black Tree:**



## 6. Comparison of Trees

Tree Type	Balanced?	Time Complexity (Search, Insert, Delete)	Special Features
Binary Search Tree (BST)	No	$O(n)$ (worst case), $O(\log n)$ (avg case)	Simple but unbalanced
AVL Tree	Yes	$O(\log n)$	Strictly balanced
Red-Black Tree	Yes	$O(\log n)$	Less strict balancing
Threaded Binary Tree	No	$O(n)$ (for traversal)	Faster inorder traversal

## Key Takeaways

**Binary Tree** → Basic hierarchical structure.

**Binary Search Tree (BST)** → Sorted structure for fast searching.

**Threaded Binary Tree** → Uses extra pointers for traversal.

**AVL Tree** → Self-balancing tree using rotations.

**Red-Black Tree** → Self-balancing BST with color rules.

**Tree Traversals:**

- **DFS (Inorder, Preorder, Postorder)**
- **BFS (Level Order)**

This **Tree Data Structure Cheat Sheet** covers **basic terminologies, types of trees, traversal algorithms, AVL trees, and Red-Black trees**. Let me know if you need further explanations!