Introduction to HashMap in Java

A **HashMap** in Java is a part of the **java.util** package.
It is a **key-value data structure** (like a dictionary/map).

Key Points:

- Stores data in **pairs** → `(key, value)`
- **No duplicate keys** are allowed
- Values **can be duplicated**
- Allows **null key** (only one) and **multiple null values**
- Fast operations → **O(1)** average time for insertion, search, delete
- Elements are **not stored in order**

Why is HashMap Fast?

HashMap uses a technique called **Hashing**.

- Each key is passed through a **hash function**
- Hash function returns a **hash code (index)**
- Value is stored in a **bucket** at that index
- If two keys generate same index → **Collision**
    - Collision is handled using **LinkedList or Tree (Java 8+)**

Internal Structure (Java 8+)

HashMap stores elements in an array of **Node<K, V>[] table**.
Each `Node` contains:

```
key
value
hash
next (linked list OR tree node)
```

If a bucket has many nodes:

- When chain size > 8 → tree (Red-Black Tree)
- When chain size < 6 → back to linked list

Most Used Operations in HashMap

Operation	Meaning
`put(key, value)`	Insert or update
`get(key)`	Fetch value
`remove(key)`	Delete a key
`containsKey(key)`	Check if key exists
`containsValue(value)`	Check if value exists

Operation	Meaning
`isEmpty()`	Check empty
`size()`	Return number of entries
`keySet()`	Returns all keys
`values()`	Returns all values
`entrySet()`	Returns key-value pairs

HashMap Example in Java (Basic Operations)

```
import java.util.*;

public class HashMapDemo {
    public static void main(String[] args) {

        HashMap<String, Integer> map = new HashMap<>();

        // Insert elements
        map.put("Amit", 101);
        map.put("Rahul", 102);
        map.put("Neha", 103);

        // Update value
        map.put("Rahul", 202);

        // Access elements
        System.out.println("Rahul's ID: " + map.get("Rahul"));

        // Check key
        System.out.println(map.containsKey("Amit"));   // true

        // Remove key
        map.remove("Neha");

        // Iterate through Map
        for (Map.Entry<String, Integer> e : map.entrySet()) {
            System.out.println(e.getKey() + " → " + e.getValue());
        }

        // Size
        System.out.println("Total Entries: " + map.size());
    }
}
```

Count Frequency of Each Element

Problem:

Given an array, count how many times each element appears.

Solution Idea:

Use a **HashMap** where:

- **key → element**

- **value → frequency**

Java Code:

```
import java.util.*;

public class FrequencyCount {
    public static void main(String[] args) {
        int[] arr = {1, 2, 2, 3, 1, 4, 2};
        HashMap<Integer, Integer> map = new HashMap<>();

        for(int x : arr) {
            map.put(x, map.getOrDefault(x, 0) + 1);
        }

        System.out.println("Frequency of each element:");
        for(Map.Entry<Integer, Integer> e : map.entrySet()) {
            System.out.println(e.getKey() + " -> " + e.getValue());
        }
    }
}
```

Find the First Non-Repeating Character in a String

Problem:

Given a string `"swiss"`, find the **first character** that does **not repeat**.

Solution Idea:

1. Use HashMap to count frequency
2. Iterate again and return first with count = 1

Java Code:

```
import java.util.*;

public class FirstUniqueChar {
    public static void main(String[] args) {
        String s = "swiss";

        HashMap<Character, Integer> map = new HashMap<>();
        for(char c : s.toCharArray()) {
            map.put(c, map.getOrDefault(c, 0) + 1);
        }

        for(char c : s.toCharArray()) {
            if(map.get(c) == 1) {
                System.out.println("First non-repeating: " + c);
                return;
            }
        }
        System.out.println("No unique character found");
    }
}
```

Check if Two Strings Are Anagrams

Problem:

`"listen"` and `"silent"` → Anagram? (Both have same characters)

Solution Idea:

Count frequency of each character using HashMap.

Java Code:

```
import java.util.*;

public class AnagramCheck {
    public static boolean isAnagram(String s1, String s2) {
        if(s1.length() != s2.length()) return false;

        HashMap<Character, Integer> map = new HashMap<>();
        for(char c : s1.toCharArray()) {
            map.put(c, map.getOrDefault(c, 0) + 1);
        }

        for(char c : s2.toCharArray()) {
            if(!map.containsKey(c)) return false;
            map.put(c, map.get(c) - 1);
            if(map.get(c) < 0) return false;
        }
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isAnagram("listen", "silent"));
    }
}
```

Find Duplicate Elements in Array

Problem:

Print elements that appear more than once.

Solution Idea:

Use HashMap → count frequency → print elements with value > 1.

Java Code:

```
import java.util.*;

public class FindDuplicates {
    public static void main(String[] args) {
        int[] arr = {10, 20, 20, 30, 10, 40, 20};

        HashMap<Integer, Integer> map = new HashMap<>();
```

```
        for(int x : arr) {
            map.put(x, map.getOrDefault(x, 0) + 1);
        }

        System.out.println("Duplicate elements:");
        for(int key : map.keySet()) {
            if(map.get(key) > 1) {
                System.out.println(key + " appears " + map.get(key) + "
times");
            }
        }
    }
}
```

Find the Most Frequent Element

Problem:

Which element occurs the maximum number of times?

Solution Idea:

Maintain max frequency while iterating map.

Java Code:

```
import java.util.*;

public class MostFrequent {
    public static void main(String[] args) {
        int[] arr = {1, 3, 2, 3, 4, 3, 2};

        HashMap<Integer, Integer> map = new HashMap<>();

        for(int x : arr) {
            map.put(x, map.getOrDefault(x, 0) + 1);
        }

        int maxFreq = 0;
        int ans = -1;

        for(int key : map.keySet()) {
            if(map.get(key) > maxFreq) {
                maxFreq = map.get(key);
                ans = key;
            }
        }

        System.out.println("Most Frequent Element: " + ans);
    }
}
```

Custom Implementation of HashMap (Simplified)

This is a **DSA-style implementation** using **array of linked lists** (bucket array).

Node Class

```
class Node {
    String key;
    int value;
    Node next;

    Node(String key, int value) {
        this.key = key;
        this.value = value;
    }
}
```

HashMap Implementation

```
class MyHashMap {

    private Node[] buckets;
    private int capacity = 10;

    MyHashMap() {
        buckets = new Node[capacity];
    }

    private int getHash(String key) {
        return Math.abs(key.hashCode()) % capacity;
    }

    // Insert or Update
    public void put(String key, int value) {
        int index = getHash(key);
        Node head = buckets[index];

        // If key already exists → update
        while (head != null) {
            if (head.key.equals(key)) {
                head.value = value;
                return;
            }
            head = head.next;
        }

        // Insert at beginning
        Node newNode = new Node(key, value);
        newNode.next = buckets[index];
        buckets[index] = newNode;
    }

    // Get value by key
    public Integer get(String key) {
        int index = getHash(key);
        Node head = buckets[index];

        while (head != null) {
            if (head.key.equals(key)) return head.value;
            head = head.next;
        }
        return null;  // not found
    }
```

```
    // Remove key
    public void remove(String key) {
        int index = getHash(key);
        Node head = buckets[index];
        Node prev = null;

        while (head != null) {
            if (head.key.equals(key)) {
                if (prev == null) buckets[index] = head.next;
                else prev.next = head.next;
                return;
            }
            prev = head;
            head = head.next;
        }
    }
}
```

Test the Custom HashMap

```
public class TestMap {
    public static void main(String[] args) {
        MyHashMap map = new MyHashMap();

        map.put("India", 91);
        map.put("USA", 1);
        map.put("Japan", 81);
        map.put("India", 911); // update

        System.out.println(map.get("India"));  // 911
        System.out.println(map.get("USA"));    // 1

        map.remove("Japan");
        System.out.println(map.get("Japan"));  // null
    }
}
```

Summary (Perfect for Notes)

- HashMap stores **key-value pairs**
- Very fast: **O(1)** average
- Uses **hashing and buckets**
- Uses **linked list + tree** for collision handling
- Important methods: `put`, `get`, `remove`, `containsKey`, `size`, `entrySet`

Custom implementation uses **array of linked lists**

Binary Tree Code

We will include:

- Node structure
- Insert (manual insertion)
- Traversals (Inorder, Preorder, Postorder, Level Order)
- Height of tree

- Count nodes
- Search in tree

1. Node Class

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

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

2. Binary Tree Class with Operations

```
import java.util.LinkedList;
import java.util.Queue;

class BinaryTree {
    Node root;

    // Constructor
    BinaryTree() {
        root = null;
    }

    // 1. Insert nodes manually (simple tree)
    public void createSampleTree() {
        root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);

        root.left.left = new Node(4);
        root.left.right = new Node(5);
    }

    // 2. Inorder Traversal (Left, Root, Right)
    public void inorder(Node node) {
        if (node == null) return;

        inorder(node.left);
        System.out.print(node.data + " ");
        inorder(node.right);
    }

    // 3. Preorder Traversal (Root, Left, Right)
    public void preorder(Node node) {
        if (node == null) return;

        System.out.print(node.data + " ");
        preorder(node.left);
        preorder(node.right);
    }

    // 4. Postorder Traversal (Left, Right, Root)
    public void postorder(Node node) {
        if (node == null) return;
```

```
    postorder(node.left);
    postorder(node.right);
    System.out.print(node.data + " ");
}

// 5. Level Order Traversal (BFS)
public void levelOrder(Node node) {
    if (node == null) return;

    Queue<Node> q = new LinkedList<>();
    q.add(node);

    while (!q.isEmpty()) {
        Node curr = q.poll();
        System.out.print(curr.data + " ");

        if (curr.left != null) q.add(curr.left);
        if (curr.right != null) q.add(curr.right);
    }
}

// 6. Search a value in tree
public boolean search(Node node, int key) {
    if (node == null) return false;

    if (node.data == key) return true;

    // search in left or right subtree
    return search(node.left, key) || search(node.right, key);
}

// 7. Find height of tree
public int height(Node node) {
    if (node == null) return 0;

    int l = height(node.left);
    int r = height(node.right);

    return Math.max(l, r) + 1;
}

// 8. Count total nodes
public int countNodes(Node node) {
    if (node == null) return 0;

    return 1 + countNodes(node.left) + countNodes(node.right);
}

// Main method
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();

    tree.createSampleTree();

    System.out.println("Inorder Traversal:");
    tree.inorder(tree.root);

    System.out.println("\nPreorder Traversal:");
    tree.preorder(tree.root);
```

```
        System.out.println("\nPostorder Traversal:");
        tree.postorder(tree.root);

        System.out.println("\nLevel Order Traversal:");
        tree.levelOrder(tree.root);

        System.out.println("\n\nSearch 5: " + tree.search(tree.root, 5));
        System.out.println("Height of tree: " + tree.height(tree.root));
        System.out.println("Total Nodes: " + tree.countNodes(tree.root));
    }
}
```

Output Example

```
Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Postorder: 4 5 2 3 1
Level Order: 1 2 3 4 5
Search 5: true
Height: 3
Total Nodes: 5
```

Most Used Operations Summary

Operation	Description
Inorder	Used to get sorted output in BST
Preorder	Used to create copy of tree
Postorder	Used to delete tree bottom-up
Level Order (BFS)	Used in shortest path, BFS
Search	Find any element
Height	Balanced/unbalanced tree check
Count Nodes	Total nodes in tree

Binary Search Tree (BST)

A **Binary Search Tree (BST)** is a special type of binary tree where the data is arranged in a sorted manner.

BST Property

For every node:

- **Left subtree contains values smaller** than the node.
- **Right subtree contains values greater** than the node.
- No duplicate values (generally).

Why BST is Important?

Operation	Time Complexity (Average)
Search	**O(log n)**
Insert	**O(log n)**
Delete	**O(log n)**

If BST becomes skewed (linked-list-like), the time becomes **O(n)** → reason AVL/Red-Black Trees are used.

Basic Operations in BST

1. **Insert a node**
2. **Search a value**
3. **Delete a node**
4. **Inorder Traversal (gives sorted order)**
5. **Preorder & Postorder** (optional)

BST Traversals

Inorder:

```
Left → Root → Right
```

✓ gives sorted output

Preorder:

```
Root → Left → Right
```

✓ used to create copy of tree

Postorder:

```
Left → Right → Root
```

✓ used to delete tree safely

BST Code in Java (Most Important Operations)

Node Structure

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

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

Binary Search Tree Class

```
class BST {
    Node root;

    BST() {
        root = null;
    }

    // 1. Insert a value in BST
    Node insert(Node node, int value) {
        if (node == null) {
            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; // unchanged root
    }

    // 2. Search a value in BST
    boolean search(Node node, int key) {
        if (node == null) return false;

        if (key == node.data) return true;

        if (key < node.data)
            return search(node.left, key);
        else
            return search(node.right, key);
    }

    // 3. Find minimum node (used in deletion)
    Node findMin(Node node) {
        while (node.left != null)
            node = node.left;
        return node;
    }

    // 4. Delete a node from BST
    Node delete(Node node, int key) {
        if (node == null) return null;

        if (key < node.data) {
            node.left = delete(node.left, key);
        } else if (key > node.data) {
            node.right = delete(node.right, key);
        } else {

            // Case 1: Node has no child
            if (node.left == null && node.right == null)
                return null;

            // Case 2: One child
            else if (node.left == null)
                return node.right;

            else if (node.right == null)
                return node.left;
```

```
            // Case 3: Two children
            Node minRight = findMin(node.right);
            node.data = minRight.data; // replace value
            node.right = delete(node.right, minRight.data);
        }
        return node;
    }

    // 5. Inorder Traversal (sorted order)
    void inorder(Node node) {
        if (node == null) return;

        inorder(node.left);
        System.out.print(node.data + " ");
        inorder(node.right);
    }

    // 6. Preorder Traversal
    void preorder(Node node) {
        if (node == null) return;

        System.out.print(node.data + " ");
        preorder(node.left);
        preorder(node.right);
    }

    // 7. Postorder Traversal
    void postorder(Node node) {
        if (node == null) return;

        postorder(node.left);
        postorder(node.right);
        System.out.print(node.data + " ");
    }

    // Main method
    public static void main(String[] args) {
        BST tree = new BST();

        int[] values = {50, 30, 20, 40, 70, 60, 80};

        for (int v : values) {
            tree.root = tree.insert(tree.root, v);
        }

        System.out.println("Inorder (Sorted):");
        tree.inorder(tree.root);

        System.out.println("\nSearch 40: " + tree.search(tree.root, 40));

        tree.root = tree.delete(tree.root, 30);
        System.out.println("Inorder after deleting 30:");
        tree.inorder(tree.root);
    }
}
```

Example Output

```
Inorder (Sorted):
```

```
20 30 40 50 60 70 80
Search 40: true
Inorder after deleting 30:
20 40 50 60 70 80
```

BST Deletion Summary (Easy)

Case	Explanation
0 Child	Delete node directly
1 Child	Replace node with its single child
2 Children	Replace with **inorder successor** (minimum in right subtree)

AVL Tree – Notes

An **AVL Tree** is a **self-balancing Binary Search Tree**.

It was invented by **Adelson-Velsky and Landis** (hence the name AVL).

Why AVL Tree?

In a normal BST, if the tree becomes skewed, operations (search, insert, delete) become **O(n)**.

AVL tree **keeps the height balanced**, ensuring operations always take:

Time Complexity: O(log n)

(for Search, Insert, Delete)

Balance Factor (VERY Important)

Each node stores:

```
Balance Factor = height(left subtree) – height(right subtree)
```

Allowed values: `-1, 0, +1`

If balance factor becomes:

- < –1
- +1
  → **Tree is unbalanced** → perform rotations.

Rotations in AVL Tree

To fix unbalanced cases, AVL performs **rotations**.

Four imbalance cases:

Case	Condition	Rotation
LL (Left-Left)	Insert in left subtree of left child	**Right Rotation**
RR (Right-Right)	Insert in right subtree of right child	**Left Rotation**
LR (Left-Right)	Insert in right subtree of left child	**Left Rotation on Left Child + Right Rotation**
RL (Right-Left)	Insert in left subtree of right child	**Right Rotation on Right Child + Left Rotation**

Rotation Summary (Easy to Remember)

LL → Right Rotate

RR → Left Rotate

LR → Left Rotate → Right Rotate

RL → Right Rotate → Left Rotate

AVL Tree Properties

- Self-balancing BST
- Height always **O(log n)**
- No skewed structure
- Uses rotations to maintain balance

AVL Tree Code in Java (Insert + Rotations)

This is clean, standard, easy-to-understand code.

Node Structure

```
class Node {
    int key, height;
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}
```

AVL Tree Class with Insert Operation

```
class AVLTree {

    Node root;

    // Get height of node
```

```
int height(Node N) {
    if (N == null)
        return 0;
    return N.height;
}

// Get balance factor
int getBalance(Node N) {
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}

// Right Rotation (LL Case)
Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    // Rotation
    x.right = y;
    y.left = T2;

    // Update heights
    y.height = Math.max(height(y.left), height(y.right)) + 1;
    x.height = Math.max(height(x.left), height(x.right)) + 1;

    return x; // new root
}

// Left Rotation (RR Case)
Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    // Rotation
    y.left = x;
    x.right = T2;

    // Update heights
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    y.height = Math.max(height(y.left), height(y.right)) + 1;

    return y; // new root
}

// Insert node
Node insert(Node node, int key) {

    // 1. Normal BST insert
    if (node == null)
        return new Node(key);

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else
        return node; // duplicates not allowed

    // 2. Update height
    node.height = 1 + Math.max(height(node.left), height(node.right));
```

```
        // 3. Get balance factor
        int balance = getBalance(node);

        // 4. Check imbalance cases

        // LL Case
        if (balance > 1 && key < node.left.key)
            return rightRotate(node);

        // RR Case
        if (balance < -1 && key > node.right.key)
            return leftRotate(node);

        // LR Case
        if (balance > 1 && key > node.left.key) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        // RL Case
        if (balance < -1 && key < node.right.key) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        return node; // unchanged root
    }

    // Inorder Traversal (sorted)
    void inorder(Node node) {
        if (node != null) {
            inorder(node.left);
            System.out.print(node.key + " ");
            inorder(node.right);
        }
    }

    public static void main(String[] args) {

        AVLTree tree = new AVLTree();

        int[] values = { 10, 20, 30, 40, 50, 25 };

        for (int v : values)
            tree.root = tree.insert(tree.root, v);

        System.out.println("Inorder Traversal:");
        tree.inorder(tree.root);
    }
}
```

Output

```
Inorder Traversal:
10 20 25 30 40 50
```

AVL Tree Summary (One-Page Notes)

- AVL = Self-balancing BST
- Balance Factor must be **-1, 0, +1**
- Height is always **log n**
- Uses **rotations** to maintain balance
- Four cases:
    - LL → Right Rotate
    - RR → Left Rotate
    - LR → Left Rotate + Right Rotate
    - RL → Right Rotate + Left Rotate

Red-Black Tree (RBT)

A **Red-Black Tree** is a **self-balancing Binary Search Tree** where each node has an extra color property (Red or Black).
It ensures the tree height remains **O(log n)** for insertion, deletion, and searching.

Why Red-Black Tree?

- Maintains **balanced height** automatically
- Performs all operations in **O(log n)**
- Used in **Java's TreeMap, TreeSet**, Linux process scheduling etc.

Properties of Red-Black Tree

Every RBT must follow these 5 rules:

1. **Every node is either RED or BLACK**
2. **Root is always BLACK**
3. **All NULL (leaf) nodes are BLACK**
4. **A RED node must not have a RED parent**
   (No two consecutive RED nodes)
5. **Every path from a node to its descendant NULL nodes has the same number of BLACK nodes**
   (Black-height is same)

These ensure the tree is balanced.

Key Operations That Maintain Properties

- **Rotations**
    - Left Rotation
    - Right Rotation
- **Recoloring**
- **Rebalancing after Insert/Delete**

Red-Black Tree – Insert Case Handling

When you insert a node, tree properties may break. To fix:

Case 1: Node is Root:

Recolor it to black.

Case 2: Red-Red Violation:

If the new node's parent is red

Case 2.1: Uncle is Red:

Recolor the parent and the uncle to black, and the grandparent to red. Then, repeat the fix-up process from the grandparent.

Case 2.2: Uncle is Black or Null:

Perform rotations to balance the tree.
Recolor the nodes accordingly

Delete Case Handling

When a BLACK node is deleted, “double-black” problem arises.

Fix via:

- Recoloring sibling
- Rotations
- Propagating double-black upward

Most Used Methods in RBT

(Also used generally in Binary Trees)

Method	Use
`insert(key)`	Insert data maintaining RBT rules
`delete(key)`	Delete node with rebalancing
`search(key)`	Search element (O(log n))
`inorder()`	Sorted order traversal
`preorder()`	Root-left-right
`postorder()`	Left-right-root
`minimum()`	Smallest value
`maximum()`	Largest value
`successor(node)`	Next higher value
`predecessor(node)`	Next lower value

JAVA CODE – Red-Black Tree Implementation

Below is a **clean & understandable** version suitable for learning and interviews.

Node Class

```
class Node {
    int data;
    Node left, right, parent;
    boolean color; // true = RED, false = BLACK

    Node(int data) {
        this.data = data;
        this.color = true; // new node is RED
    }
}
```

Red-Black Tree Class

```
class RedBlackTree {

    private Node root;
    private final boolean RED = true;
    private final boolean BLACK = false;

    // Left Rotation
    private void leftRotate(Node x) {
        Node y = x.right;
        x.right = y.left;

        if (y.left != null)
            y.left.parent = x;

        y.parent = x.parent;

        if (x.parent == null)
            root = y;
        else if (x == x.parent.left)
            x.parent.left = y;
        else
            x.parent.right = y;

        y.left = x;
        x.parent = y;
    }

    // Right Rotation
    private void rightRotate(Node x) {
        Node y = x.left;
        x.left = y.right;

        if (y.right != null)
            y.right.parent = x;

        y.parent = x.parent;
```

```
    if (x.parent == null)
        root = y;
    else if (x == x.parent.right)
        x.parent.right = y;
    else
        x.parent.left = y;

    y.right = x;
    x.parent = y;
}

// Fix Violations After Insert
private void fixInsert(Node k) {
    Node parent = null, grandparent = null;

    while (k != root && k.color == RED && k.parent.color == RED) {
        parent = k.parent;
        grandparent = parent.parent;

        // Parent is left child of grandparent
        if (parent == grandparent.left) {
            Node uncle = grandparent.right;

            // Case 1: Uncle is RED
            if (uncle != null && uncle.color == RED) {
                grandparent.color = RED;
                parent.color = BLACK;
                uncle.color = BLACK;
                k = grandparent;
            }
            else {
                // Case 2: k is right child
                if (k == parent.right) {
                    leftRotate(parent);
                    k = parent;
                    parent = k.parent;
                }
                // Case 3: k is left child
                rightRotate(grandparent);
                boolean temp = parent.color;
                parent.color = grandparent.color;
                grandparent.color = temp;
                k = parent;
            }
        }
        // Parent is right child
        else {
            Node uncle = grandparent.left;

            if (uncle != null && uncle.color == RED) {
                grandparent.color = RED;
                parent.color = BLACK;
                uncle.color = BLACK;
                k = grandparent;
            }
            else {
                if (k == parent.left) {
                    rightRotate(parent);
                    k = parent;
                    parent = k.parent;
```

```
                }
                leftRotate(grandparent);
                boolean temp = parent.color;
                parent.color = grandparent.color;
                grandparent.color = temp;
                k = parent;
            }
        }
    }
    root.color = BLACK;
}

// Insert node
public void insert(int data) {
    Node newNode = new Node(data);
    root = bstInsert(root, newNode);
    fixInsert(newNode);
}

// Normal BST insert
private Node bstInsert(Node root, Node node) {
    if (root == null)
        return node;

    if (node.data < root.data) {
        root.left = bstInsert(root.left, node);
        root.left.parent = root;
    }
    else if (node.data > root.data) {
        root.right = bstInsert(root.right, node);
        root.right.parent = root;
    }

    return root;
}

// Inorder Traversal (sorted order)
public void inorder(Node root) {
    if (root != null) {
        inorder(root.left);
        System.out.print(root.data + " ");
        inorder(root.right);
    }
}

public Node getRoot() {
    return root;
}
}
```

Main Method to Test RBT

```
public class Main {
    public static void main(String[] args) {
        RedBlackTree rbt = new RedBlackTree();

        rbt.insert(10);
        rbt.insert(20);
        rbt.insert(30);
        rbt.insert(15);
```

```
        System.out.print("Inorder: ");
        rbt.inorder(rbt.getRoot());
    }
}
```

Output

```
Inorder: 10 15 20 30
```

HEAP TREE

A **Heap** is a **complete binary tree** used mainly for **priority queues**.

1. What is a Heap?

A **Heap** is a special complete binary tree that satisfies the **heap property**.

Complete Binary Tree

All levels must be filled completely
(except last level – filled left to right)

2. Types of Heaps

(1) Max Heap

- Root has **maximum** value
- Every parent ≥ children
- `arr[parent] >= arr[left/right child]`

(2) Min Heap

- Root has **minimum** value
- Every parent ≤ children
- `arr[parent] <= arr[left/right child]`

3. Why Heap?

Used in:

- **Priority Queue**
- **Heap Sort**
- **Dijkstra's Algorithm**
- **Scheduling**
- **Memory management**

4. Heap Representation

Heaps are usually stored in **arrays**, not linked nodes.

For any index `i`:

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

5. Basic Heap Operations

Operation	Time Complexity
Insert	**O(log n)**
Delete root	**O(log n)**
Get min/max	**O(1)**
Build heap	**O(n)**

6. Heapify Function (Most Important)

Heapify = Bring subtree into heap order.

Two types:

- **Max-Heapify**
- **Min-Heapify**

Java Code: Max Heap (Insert + Delete + Build)

```
class MaxHeap {
    int[] heap;
    int size;

    MaxHeap(int capacity) {
        heap = new int[capacity];
        size = 0;
    }

    // Get parent index
    int parent(int i) { return (i - 1) / 2; }

    // Get left child
    int left(int i) { return 2 * i + 1; }

    // Get right child
    int right(int i) { return 2 * i + 2; }

    // Insert value
    void insert(int val) {
        heap[size] = val;
        int i = size;
        size++;

        // Fix max heap property by moving up
        while (i != 0 && heap[parent(i)] < heap[i]) {
            int temp = heap[i];
            heap[i] = heap[parent(i)];
            heap[parent(i)] = temp;
```

```
            i = parent(i);
        }
    }

    // Max-Heapify (fix downward)
    void maxHeapify(int i) {
        int largest = i;
        int l = left(i);
        int r = right(i);

        if (l < size && heap[l] > heap[largest])
            largest = l;

        if (r < size && heap[r] > heap[largest])
            largest = r;

        if (largest != i) {
            int temp = heap[i];
            heap[i] = heap[largest];
            heap[largest] = temp;

            maxHeapify(largest);
        }
    }

    // Remove max element (root)
    int extractMax() {
        if (size <= 0) return -1;
        if (size == 1) return heap[--size];

        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;

        maxHeapify(0);

        return root;
    }

    // Print heap
    void printHeap() {
        for (int i = 0; i < size; i++)
            System.out.print(heap[i] + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        MaxHeap mh = new MaxHeap(10);
        mh.insert(20);
        mh.insert(15);
        mh.insert(30);
        mh.insert(40);

        System.out.println("Max Heap:");
        mh.printHeap();

        System.out.println("Extract Max = " + mh.extractMax());
        mh.printHeap();
    }
}
```

Output Example

```
Max Heap:
40 30 20 15
Extract Max = 40
30 15 20
```

Min Heap Code (Simple Version)

```
class MinHeap {
    int[] heap;
    int size;

    MinHeap(int capacity) {
        heap = new int[capacity];
        size = 0;
    }

    int parent(int i) { return (i - 1) / 2; }
    int left(int i) { return 2 * i + 1; }
    int right(int i) { return 2 * i + 2; }

    void insert(int val) {
        heap[size] = val;
        int i = size++;

        // Move upward to restore min-heap
        while (i != 0 && heap[parent(i)] > heap[i]) {
            int temp = heap[i];
            heap[i] = heap[parent(i)];
            heap[parent(i)] = temp;
            i = parent(i);
        }
    }

    // Fix downward
    void minHeapify(int i) {
        int smallest = i;
        int l = left(i);
        int r = right(i);

        if (l < size && heap[l] < heap[smallest])
            smallest = l;

        if (r < size && heap[r] < heap[smallest])
            smallest = r;

        if (smallest != i) {
            int temp = heap[i];
            heap[i] = heap[smallest];
            heap[smallest] = temp;

            minHeapify(smallest);
        }
    }

    int extractMin() {
        if (size <= 0) return -1;
        if (size == 1) return heap[--size];
```

```
        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;

        minHeapify(0);

        return root;
    }

    void printHeap() {
        for (int i = 0; i < size; i++)
            System.out.print(heap[i] + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        MinHeap mh = new MinHeap(10);

        mh.insert(40);
        mh.insert(20);
        mh.insert(30);
        mh.insert(10);

        System.out.println("Min Heap:");
        mh.printHeap();

        System.out.println("Extract Min = " + mh.extractMin());
        mh.printHeap();
    }
}
```

Output

```
Min Heap:
10 20 30 40
Extract Min = 10
20 40 30
```