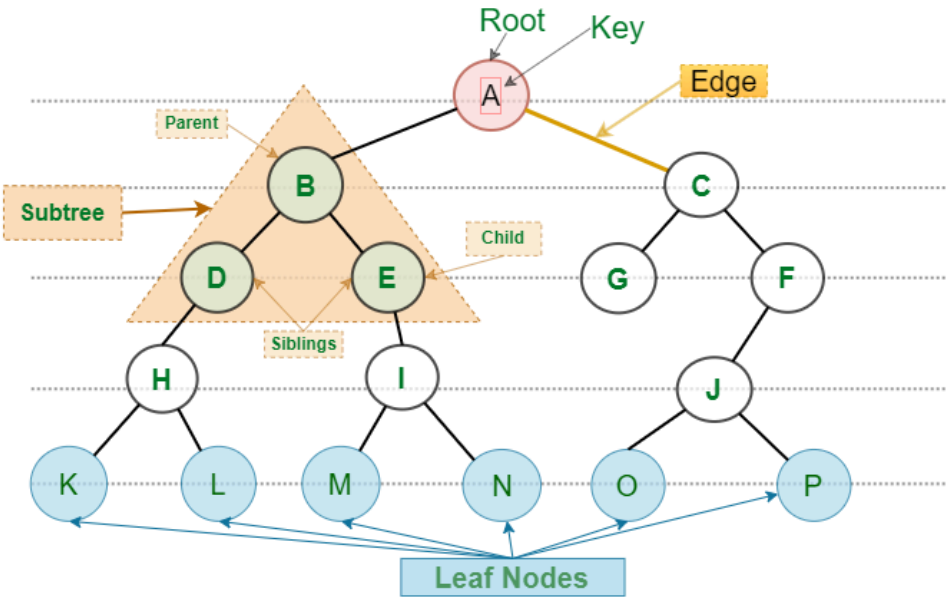
Non Linear Data Structures

Trees

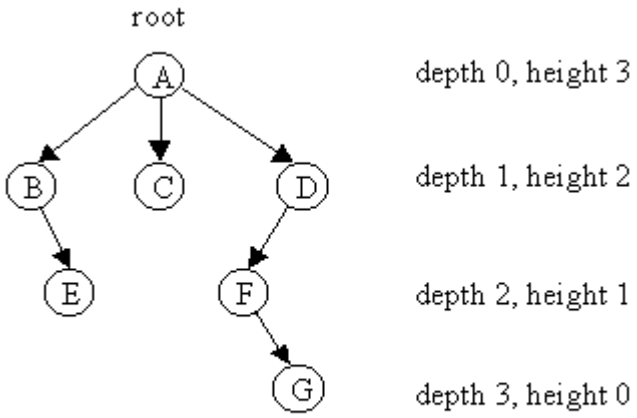
- A **hierarchical** data structure.
- Consists of **nodes** connected by **edges**.
- Topmost node: **Root**.
- Each node has **0 or more child nodes**.



Basic Terminology

- **Root**: First node of the tree (no parent)
- **Node**: Each element in the tree.
- **Edge**: Connection between parent and child.
- **Parent**: A node that has children.
- **Child**: A node that descends from another node.
- **Siblings:** Nodes that share the same parent
- **Leaf**: A node with no children.
- **Subtree**: A tree formed by a node and its descendants.

- **Height**: Max number of edges from root to a leaf.
- **Depth**: Number of edges from root to the node.



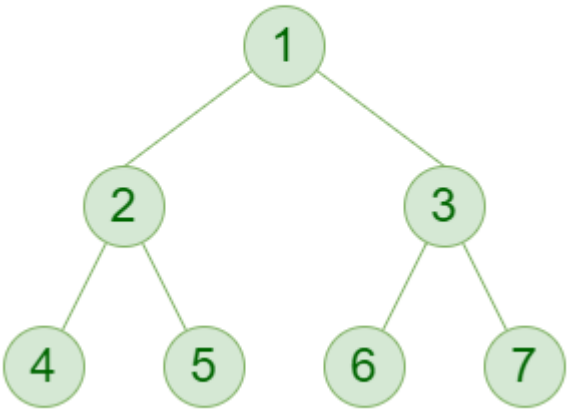
A tree of height 3

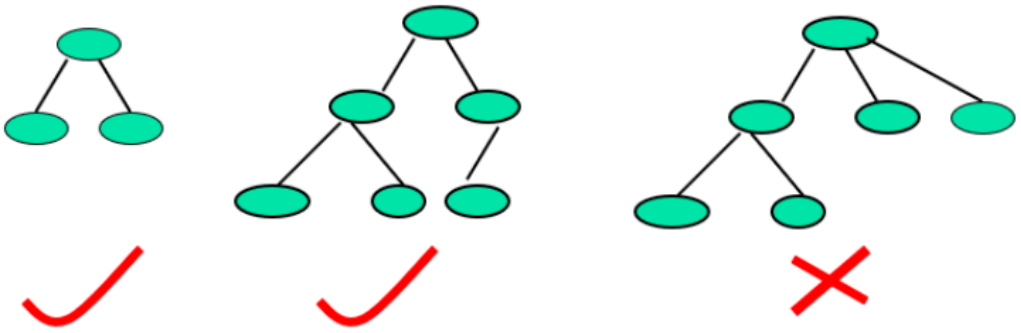
Binary Trees

A **tree** where **each node has at most two children**.(0,1 or 2 children)

Children are usually called:

- **Left child**
- **Right child**





Properties of Binary Tree

- **Maximum nodes in tree with height h**

$$2^{(h+1)} - 1 \text{ nodes}$$

Example:

If height = 2,

Then no. of nodes = $2^{(2+1)} - 1$

$= 7$ nodes

- **Minimum possible height with n nodes** = $\lceil \log_2(n+1) \rceil - 1$

Applications of Binary Trees

- Hierarchical data (e.g. file systems)
- Routing tables in networking
- Used as base for:
    - **Binary Search Trees**
    - **Heaps**
    - **AVL Trees**

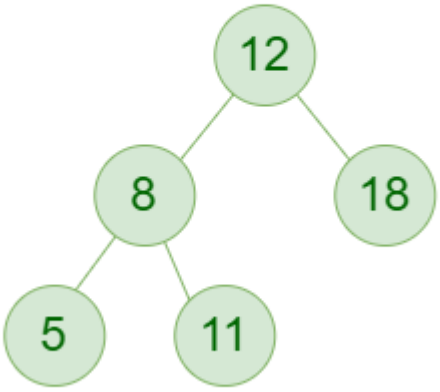
Operations in Binary Tree

- Traverse the tree
- Insert a node
- Delete a node
- Search for a value
- Find height/depth
- Count nodes or leaves

Types of Binary Trees

1. Full Binary Tree

- Every node has **0 or 2 children**
- No node has only one child

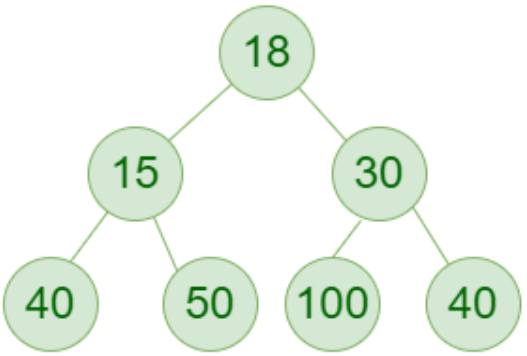


Node 8 has **2 children**

Node 18 has **0 children**

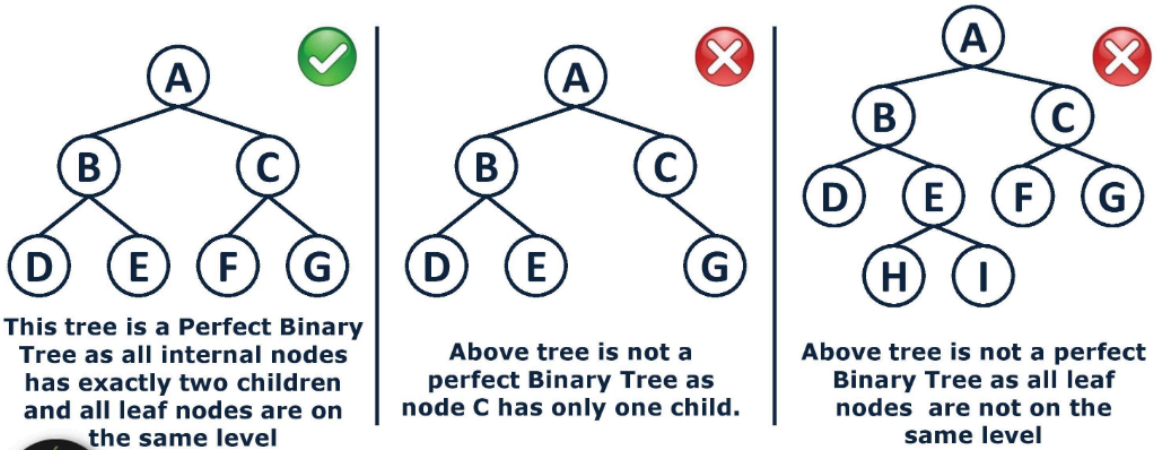
2. Perfect Binary Tree

- All internal **nodes have 2 children**
- All leaves are at the same level



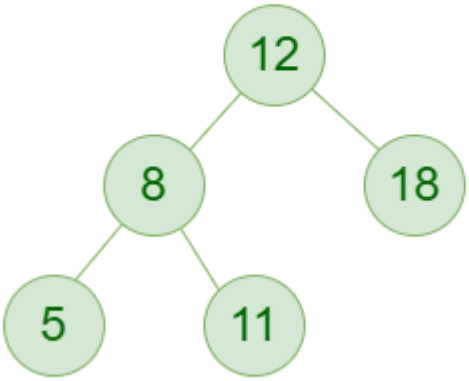
All nodes are completely filled

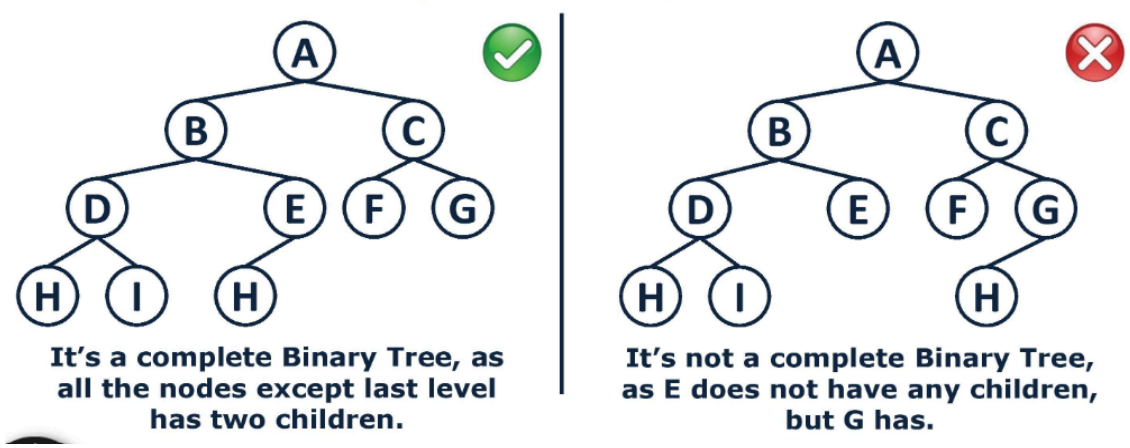
All leaves (40, 50, 100, 40) are on the same level



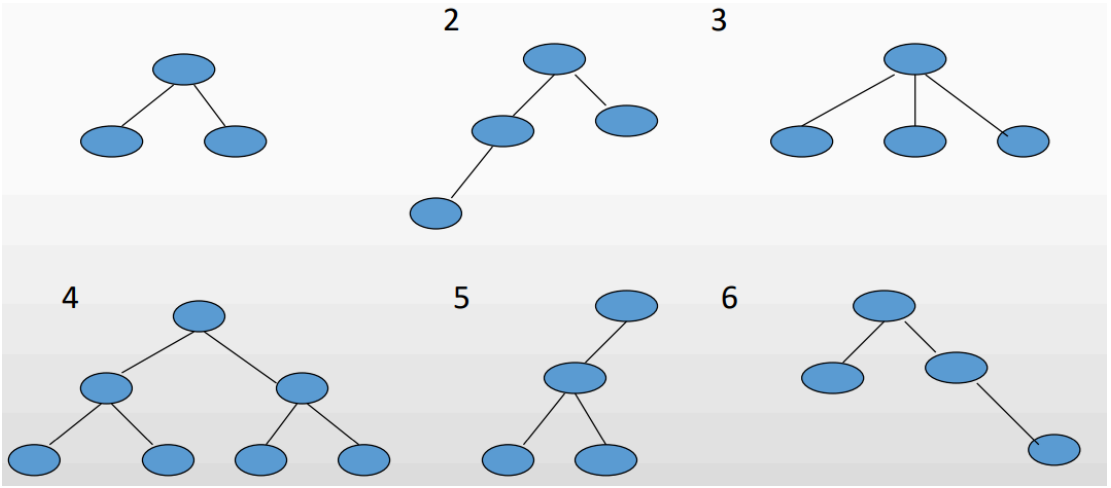
3. Complete Binary Tree

- All levels are completely filled **except the last**
- The last level is filled **from left to right**



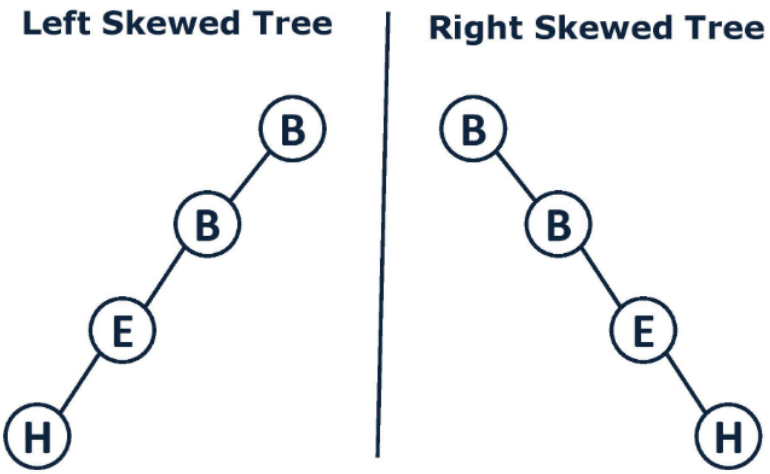


Question: Find the Complete Binary Trees from the following diagrams.



4. Degenerate (Skewed) Binary Tree

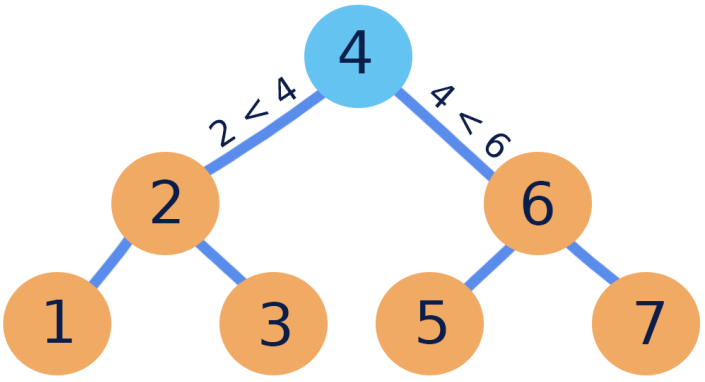
- Every parent has only one child
- Becomes a linked list



5.Binary Search Trees(BST)

A **Binary Search Tree** is a special type of **Binary Tree** where:

- Each node has at most **two children**
- The **left child** has a **smaller** value than the parent
- The **right child** has a **greater** value than the parent



Operations on BST

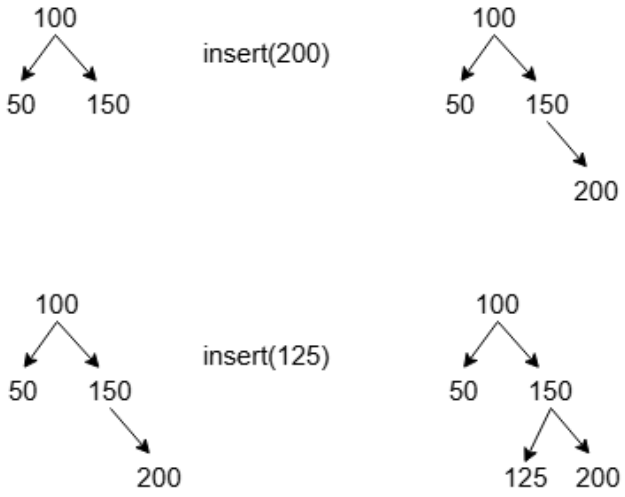
1. **Insertion**
2. **Search**
3. **Traversal**
4. **Deletion**

Operations - Insertion

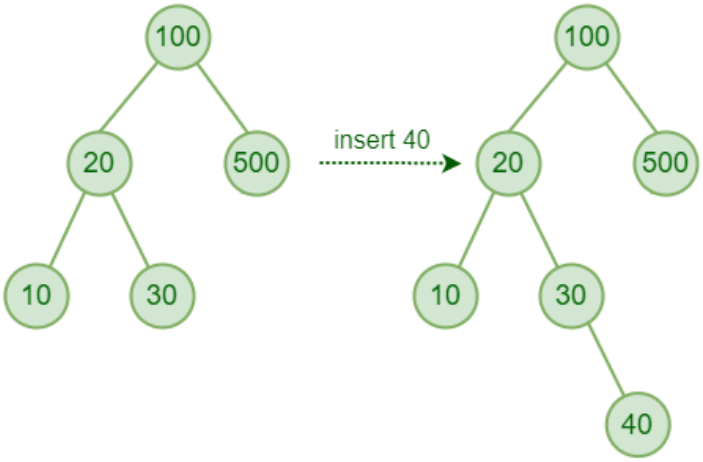
Purpose: Add a new node in the correct position so that the BST property is maintained.

How it works:

- If the tree is empty → new node becomes root
- If **value > root** → go to the **right subtree**



- If **value < root** → go to the **left subtree**



- Recursively insert at the correct position.

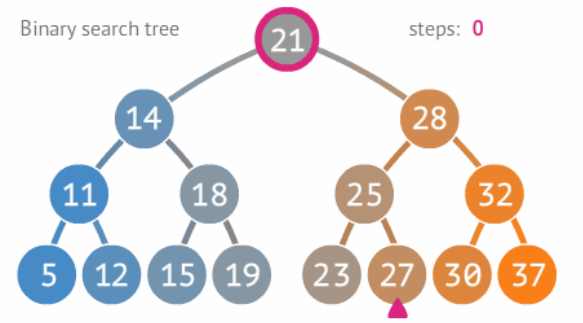
Operations - Search

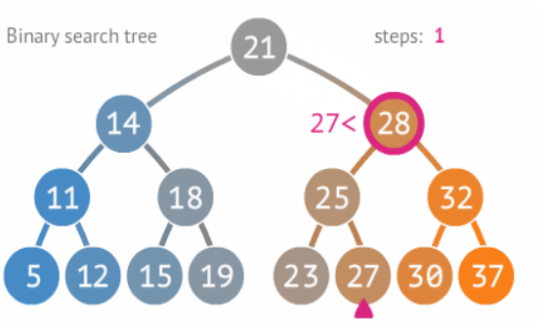
Purpose: Check whether a value exists in the BST.

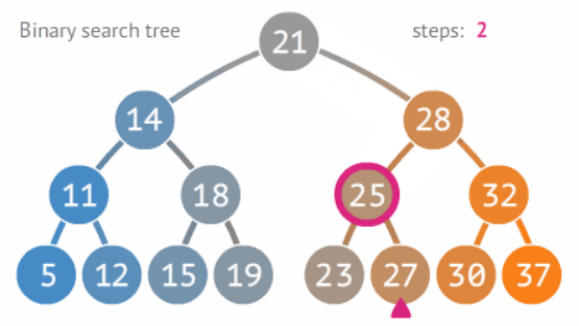
How it works:

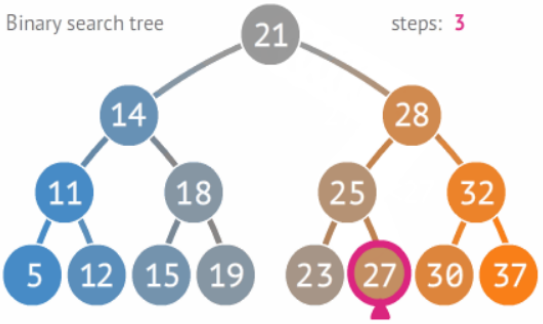
- If value == root → Found
- If **value < root** → Search **left**
- If **value > root** → Search **right**
- Continue until node is found or `null`.

Find 27







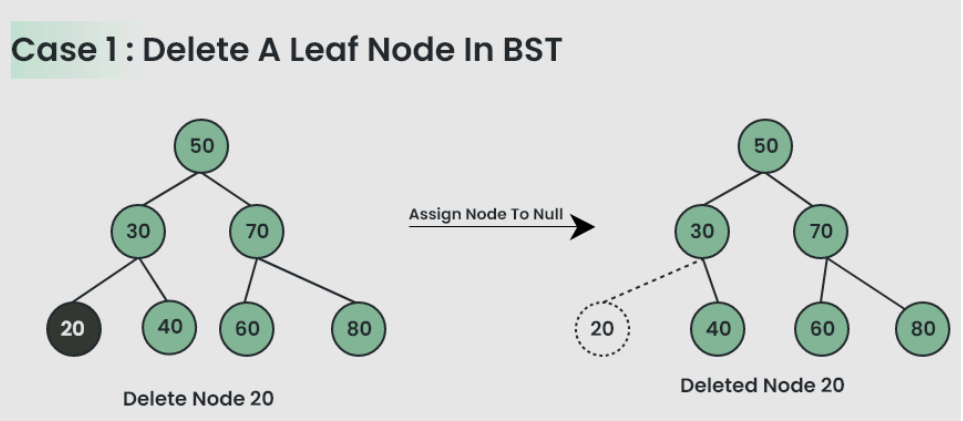


Operations - Deletion

Purpose: Remove a node while keeping BST structure intact.

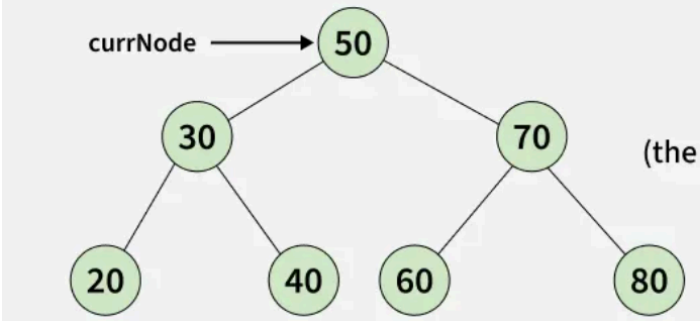
Cases:

- **Leaf node(no children)**: Just delete it.



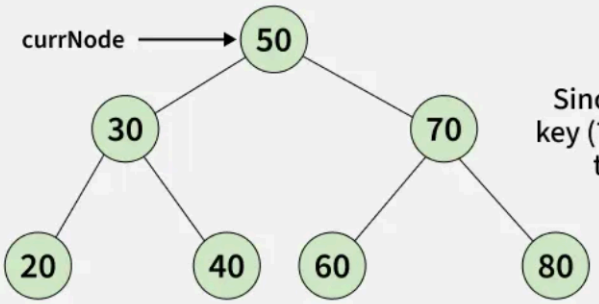
- **One child**: Replace node with its child.

01 Step | Consider the following BST



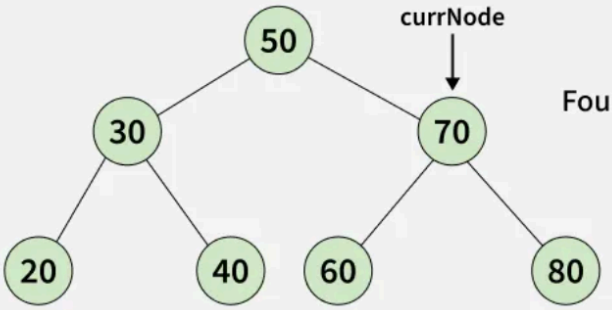
key = 70
(the node to be deleted)

02 Step | Comparing key with root node

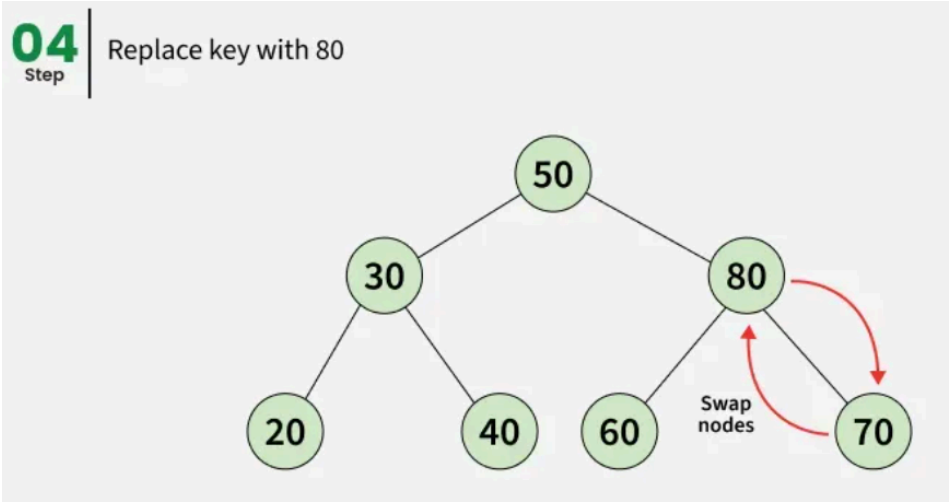


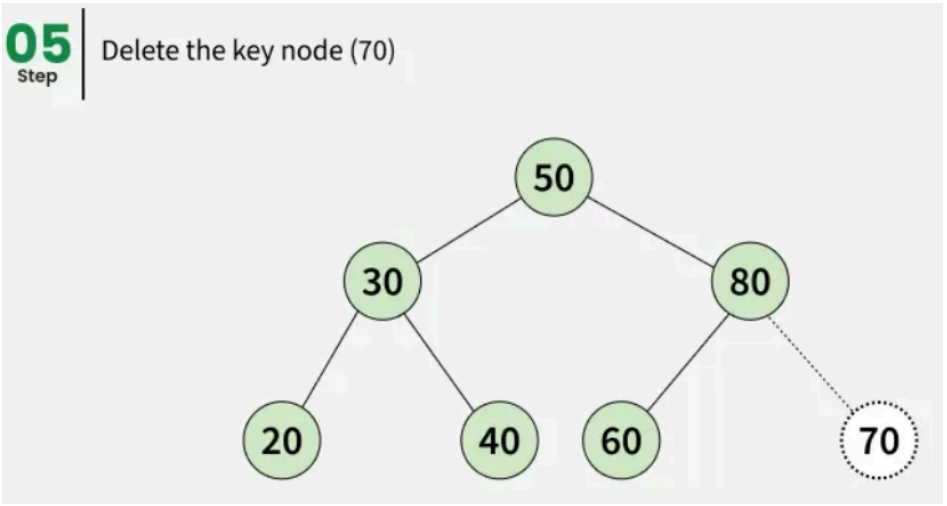
Since 50 is smaller than key (70), move the pointer to the right child.

03 Step | Comparing key with right child root node

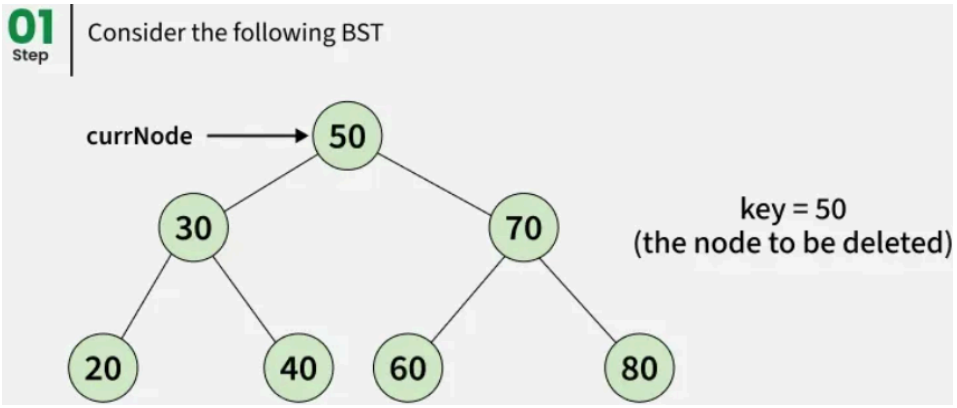


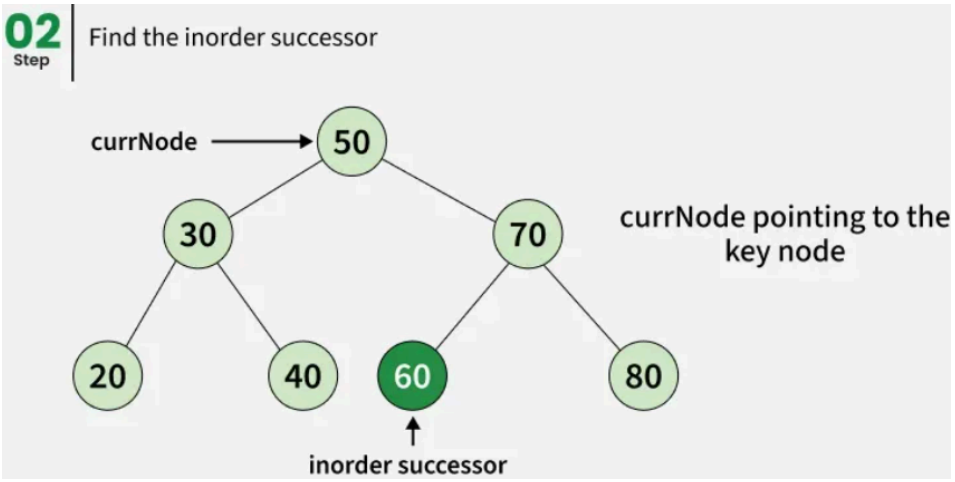
Found the key node(70)

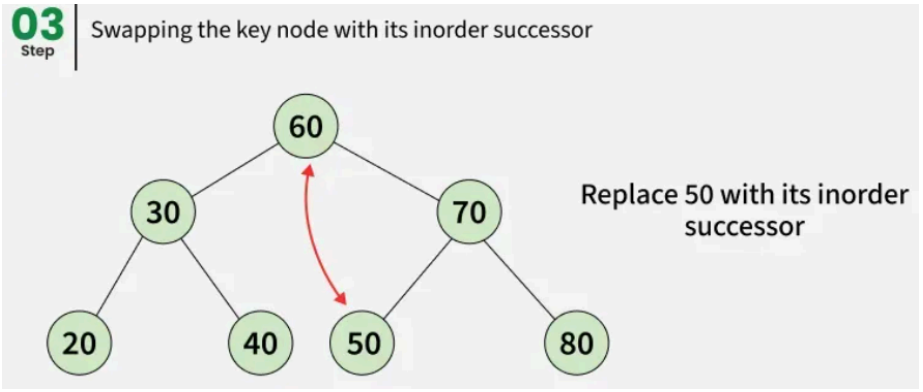


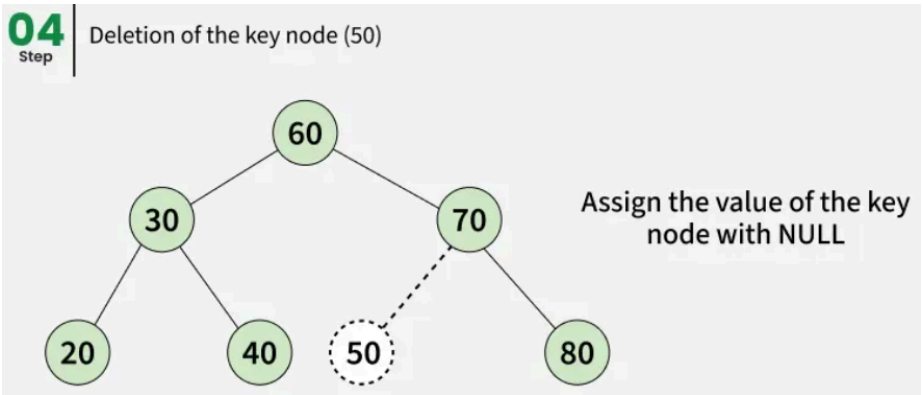


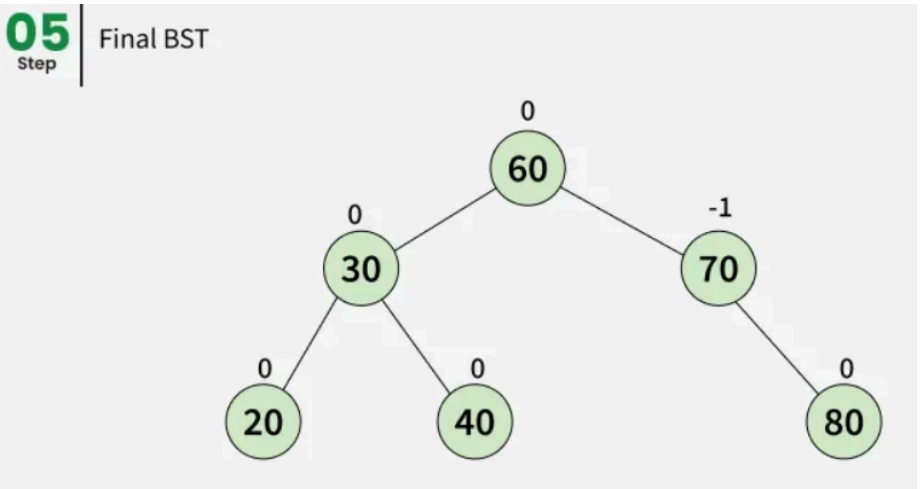
- **Two children**: Replace node with **inorder successor** (smallest value in right subtree).




02
Step
Find the inorder successor
currNode
50
30
70
20
40
60
80
currNode pointing to the key node
inorder successor


03
Step
Swapping the key node with its inorder successor
60
30
70
20
40
50
80
Replace 50 with its inorder successor


04
Step
Deletion of the key node (50)
60
30
70
20
40
50
80
Assign the value of the key node with NULL



Operations - Traversal

- Traversing a binary tree means **visiting each node** in a specific order.
- There are two main types:

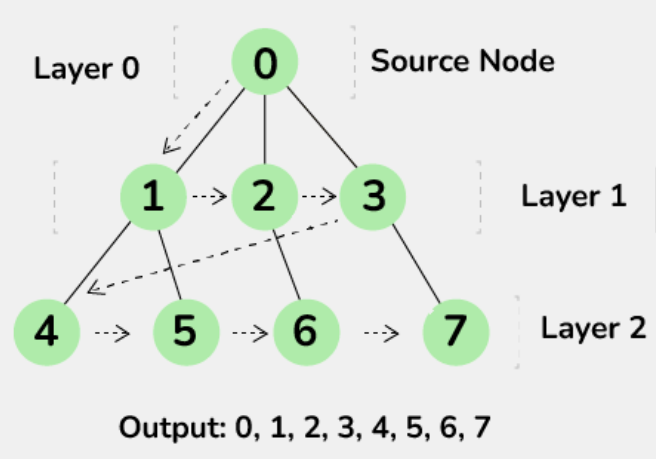
  • **Breadth First Traversal / Level Order**

  • **Depth First Traversals**

    1. Inorder Traversal (Left-Root-Right)
    2. Preorder Traversal (Root-Left-Right)
    3. Postorder Traversal (Left-Right-Root)

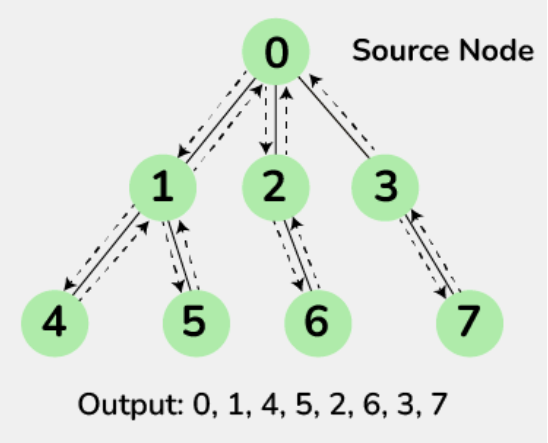
Breadth First Traversal

- Visits nodes level by level (uses a queue)
- Starts from the root and goes to the next level



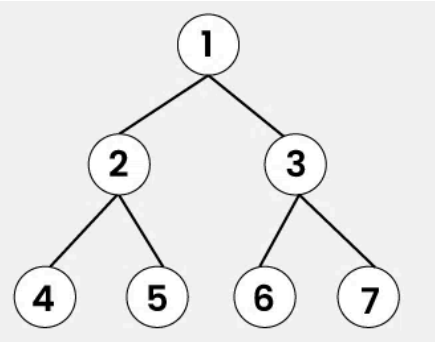
Depth First Traversal

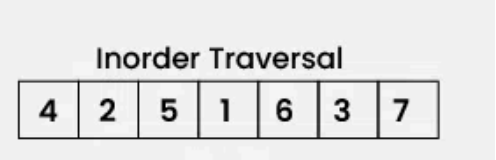
- Traverses as deep as possible along one branch before backtracking.



1. Inorder Traversal (LNR)

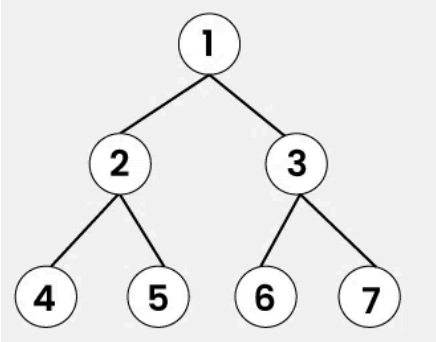
- Left → Node → Right

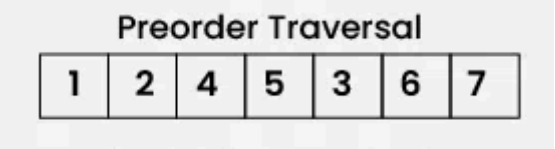




2. Preorder Traversal (NLR)

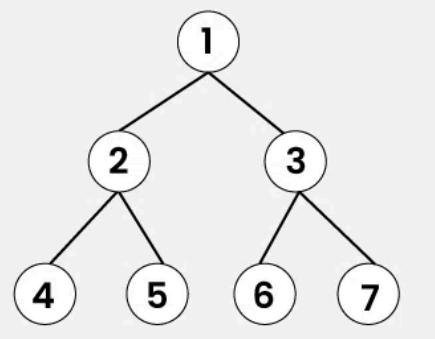
- Node → Left → Right

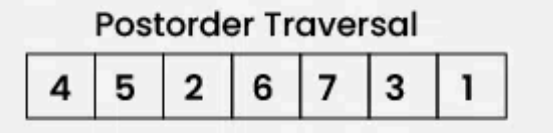




3. Postorder Traversal (LRN)

- Left → Right → Node





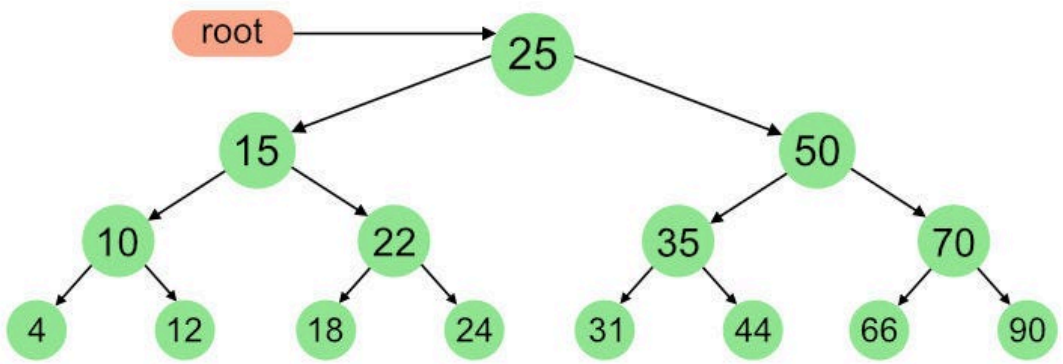
`Question:`

Find

i)Inorder

ii)Preorder

iii)Postorder traversals of the following tree.



Question:

Write a Java program to create a binary tree and implement the following Depth-First Traversals:

1. Inorder Traversal (Left → Node → Right)
2. Preorder Traversal (Node → Left → Right)
3. Postorder Traversal (Left → Right → Node)

You should define a Node class, build a sample binary tree and implement each traversal.

Sample Tree Structure:

Build this binary tree in your code:

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

Expected Output:

```
Inorder Traversal: 4 2 5 1 3
Preorder Traversal: 1 2 4 5 3
Postorder Traversal: 4 5 2 3 1
```

Java code for BST operations

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

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

public class BinarySearchTree {
    // Insert a node
    Node insert(Node root, int 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;
    }

    // Search for a value
    boolean search(Node root, int key) {
        if (root == null) return false;

        if (key == root.data) return true;
        else if (key < root.data)
            return search(root.left, key);
        else
            return search(root.right, key);
    }

    // Delete a node
    Node delete(Node root, int 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 {
            // Case 1: no child or one child
            if (root.left == null) return root.right;
            else if (root.right == null) return root.left;

            // Case 2: two children
            root.data = minValue(root.right);
            root.right = delete(root.right, root.data);
        }
        return root;
    }

    // Helper to find minimum value node
    int minValue(Node root) {
        int min = root.data;
        while (root.left != null) {
            min = root.left.data;
            root = root.left;
        }
        return min;
    }

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

    // Preorder Traversal (NLR)
    void preorder(Node root) {
        if (root != null) {
            System.out.print(root.data + " ");
            preorder(root.left);
            preorder(root.right);
        }
    }

    // Postorder Traversal (LRN)
    void postorder(Node root) {
        if (root != null) {
            postorder(root.left);
            postorder(root.right);
            System.out.print(root.data + " ");
        }
```

```
    }

    // Main method
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        Node root = null;

        // Insert nodes
        int[] values = {50, 30, 20, 40, 70, 60, 80};
        for (int val : values) {
            root = bst.insert(root, val);
        }

        // Traversals
        System.out.print("Inorder: ");
        bst.inorder(root);
        System.out.println();

        System.out.print("Preorder: ");
        bst.preorder(root);
        System.out.println();

        System.out.print("Postorder: ");
        bst.postorder(root);
        System.out.println();

        // Search
        int key = 60;
        System.out.println("Search " + key + ": " +
(bst.search(root, key) ? "Found" : "Not Found"));

        // Delete node
        root = bst.delete(root, 70);
        System.out.print("Inorder after deleting 70: ");
        bst.inorder(root);
    }
}
```

Output:

```
Inorder: 20 30 40 50 60 70 80
Preorder: 50 30 20 40 70 60 80
Postorder: 20 40 30 60 80 70 50
Search 60: Found
Inorder after deleting 70: 20 30 40 50 60 80
```

Heaps and Priority Queues

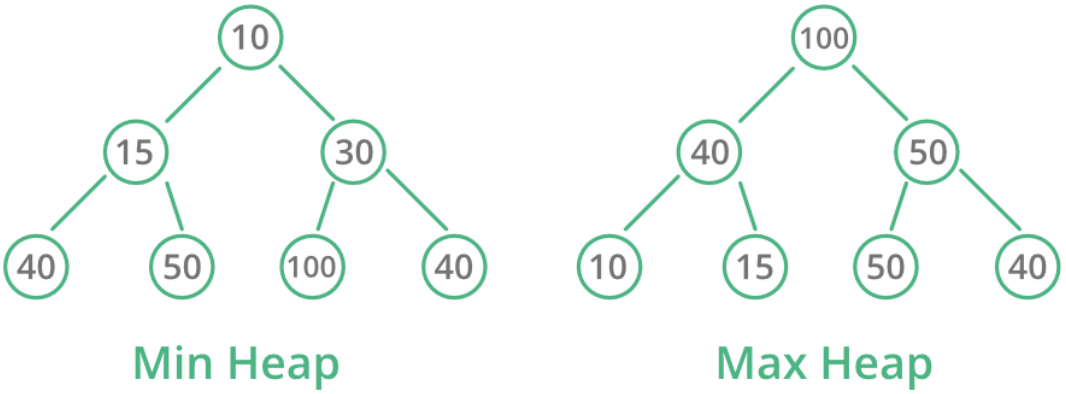
Heaps

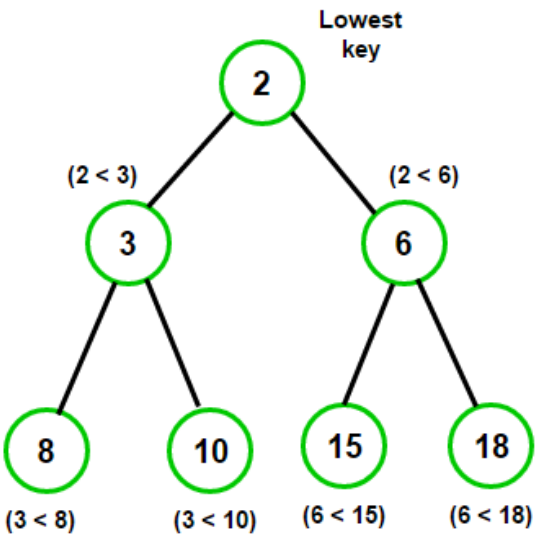
Definition

- A special tree-based data structure
- Follows the heap property
    - Max-Heap: Parent ≥ children
    - Min-Heap: Parent ≤ children
- Implemented as a complete binary tree
- Usually stored as an array

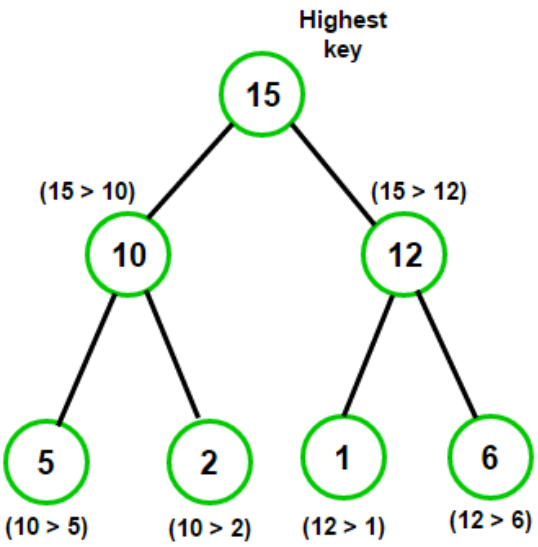
Types of Heaps

- **Min-Heap**: Smallest element at the root
- **Max-Heap**: Largest element at the root



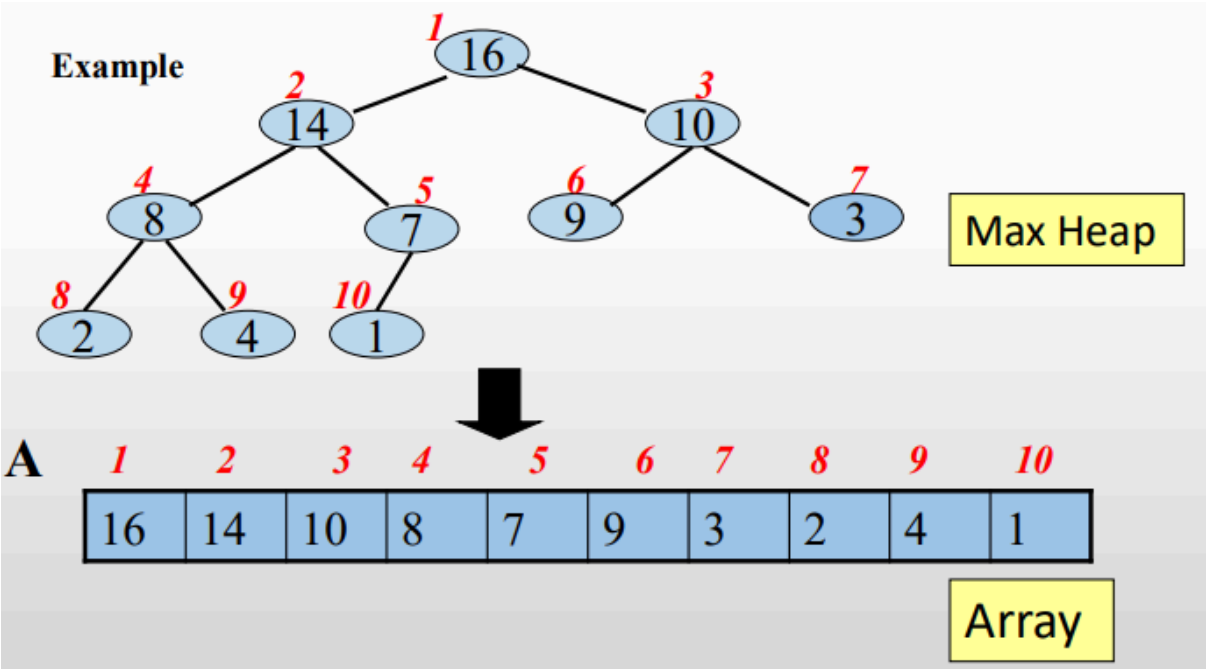


Min Heap
(Parent key is less than or equal to (≤) the child key)



Max Heap
(Parent key is greater than or equal to (≥) the child key)

• A heap can be represented in a one-dimensional array



After representing a heap using an array **A**

- Root of the tree is **A[1]**

Given node with index *i*,

- ***PARENT(i)*** is the index of parent of ***i***;

  $PARENT(i) = \lfloor i/2 \rfloor$

- ***LEFT_CHILD(i)*** is the index of left child of ***i*** ;

  $LEFT\_CHILD(i) = 2 \times i$

- ***RIGHT_CHILD(i)*** is the index of right child of ***i***;

  $RIGHT\_CHILD(i) = 2 \times i + 1$

Example:

To find the parent of node 8:

Parent(4)= 4/2 =2

Index 2 =14

Therefore, parent of node 8 is 14

Max Heapify

What is Max Heapify?

Max Heapify is a process used to maintain the **Max-Heap Property**:

> 🔺 **In a max-heap, every parent node is greater than or equal to its children.**

If a node violates this property (e.g., it is smaller than one of its children), `maxHeapify()` is used to **"push it down"** the tree so that the property is restored.

When is Max Heapify Used?

- After removing the **root** of a max-heap (e.g., in heap sort)
- After **building** the heap from an array
- After **swapping** elements

How Max Heapify Works

1. Compare the current node with its **left** and **right child**.
2. Find the **largest** among the three.
3. If the current node is **not the largest**, swap it with the largest child.
4. Recursively apply `maxHeapify()` on the affected child node.

Max Heapify Algorithm (Pseudo-code)

```
MAX-HEAPIFY(A, i)
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i

    if left < heapSize and A[left] > A[largest]
        largest = left

    if right < heapSize and A[right] > A[largest]
        largest = right

    if largest != i
        swap A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

Java Code Example for Max Heapify

```java
public class MaxHeapifyExample {
    static void maxHeapify(int[] arr, int n, int i) {
        int largest = i; // Initialize largest as root
        int left = 2 * i + 1; // left child index
        int right = 2 * i + 2; // right child index

        // If left child is larger than root
        if (left < n && arr[left] > arr[largest])
            largest = left;

        // If right child is larger than largest so far
        if (right < n && arr[right] > arr[largest])
            largest = right;

        // If largest is not root
        if (largest != i) {
            // Swap
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;

            // Recursively heapify the affected subtree
            maxHeapify(arr, n, largest);
        }
    }

    public static void main(String[] args) {
        int[] heap = {3, 5, 9, 6, 8, 20, 10, 12, 18, 9};
        int n = heap.length;

        // Apply maxHeapify starting from the last non-leaf node
        for (int i = n / 2 - 1; i >= 0; i--)
            maxHeapify(heap, n, i);

        System.out.print("Max Heap: ");
```

```
        for (int val : heap)
            System.out.print(val + " ");
    }
}
```

Output:

```
Max Heap: 20 18 10 12 9 9 3 5 6 8
```

Time Complexity

- **O(log n)**

Applications of Heaps

- **Priority Queues**
- **Heap Sort**
- **Dijkstra's Algorithm** (shortest path)
- **Scheduling systems**

Priority Queues

What is a Priority Queue?

- A **priority queue** is a special type of **queue** where each element is assigned a **priority**.

- Unlike a normal queue (FIFO – First In First Out), in a priority queue:
    - The **element with the highest priority** is served **first**.
    - If two elements have the **same priority**, they follow **FIFO** order.

Key Characteristics

- Elements are **not dequeued in insertion order**, but by **priority**.
- Commonly implemented using **heaps** for efficiency.

Types of Priority Queues

1. Min-Priority Queue

- **Highest priority = smallest value**
- Elements with **lower values** are dequeued first

Example:

Let's insert the following values:

```
40, 10, 30, 20
```

We expect the element with the **smallest value (10)** to be dequeued first.

Queue after insertion:

```
[10, 20, 30, 40]   ← priority order (lowest value = highest priority)
```

2. Max-Priority Queue

- **Highest priority = largest value**
- Elements with **higher values** are dequeued first

Example:

Insert:

```
40, 10, 30, 20
```

Now the **largest value (40)** should be served first.

Queue after insertion:

```
[40, 30, 20, 10]   ← priority order (highest value = highest priority)
```

Java Code: Priority Queue using Max-Heap

```
public class MaxPriorityQueue {
    private int[] heap;
    private int size;
    private int capacity;

    public MaxPriorityQueue(int capacity) {
        this.capacity = capacity;
        heap = new int[capacity];
        size = 0;
    }
    // Insert element
    public void insert(int value) {
        if (size == capacity) {
            System.out.println("Heap is full");
            return;
        }
        heap[size] = value;
        heapifyUp(size);
        size++;
    }
    // Remove max (root)
    public int remove() {
        if (size == 0) {
            System.out.println("Heap is empty");
            return -1;
        }
        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;
```

```
        heapifyDown(0);
        return root;
    }
    // Heapify up (after insert)
    private void heapifyUp(int index) {
        int parent = (index - 1) / 2;
        if (index > 0 && heap[index] > heap[parent]) {
            swap(index, parent);
            heapifyUp(parent);
        }
    }
    // Heapify down (after remove)
    private void heapifyDown(int index) {
        int left = 2 * index + 1;
        int right = 2 * index + 2;
        int largest = index;

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

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

        if (largest != index) {
            swap(index, largest);
            heapifyDown(largest);
        }
    }

    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }

    public void print() {
```

```
        for (int i = 0; i < size; i++)
            System.out.print(heap[i] + " ");
        System.out.println();
    }
  // Main method
    public static void main(String[] args) {
        MaxPriorityQueue pq = new MaxPriorityQueue(10);
        pq.insert(50);
        pq.insert(30);
        pq.insert(40);
        pq.insert(10);
        pq.insert(20);

        System.out.println("Max-Heap (Priority Queue):");
        pq.print();

        System.out.println("Removed (highest priority): " +
pq.remove());
        pq.print();
    }
}
```

Output:

```
Max-Heap (Priority Queue):
50 30 40 10 20
Removed (highest priority): 50
40 30 20 10
```

Advantages

- Efficient access to **highest priority** element
- Good for **real-time systems** that need urgency-based processing
- Helps optimize **resource usage**

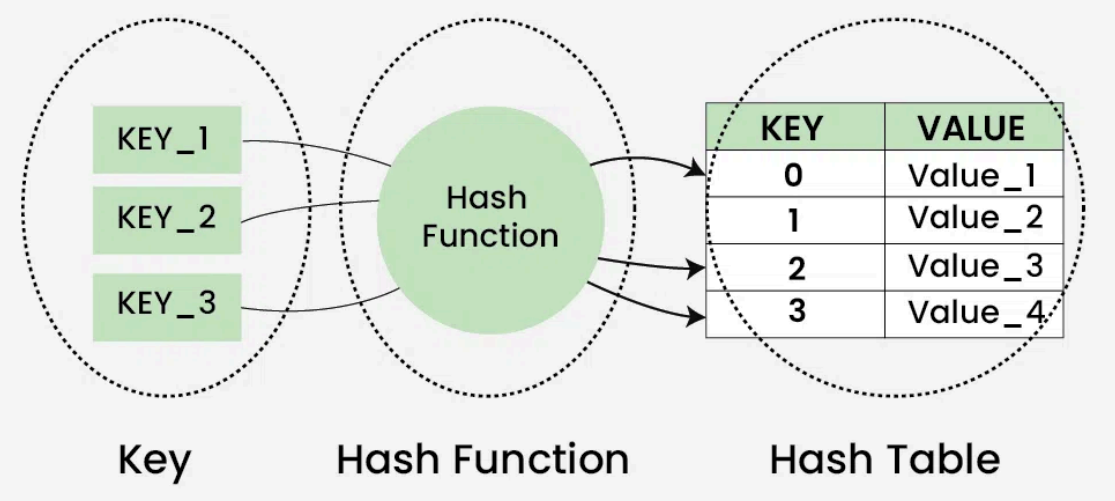
Disadvantages

- More complex than regular queues
- If not implemented with a heap, performance can be poor for large data

Hash Tables

Definition

- A **hash table** is a data structure that **stores key-value pairs**.
- It uses a **hash function** to compute an index (hash code), where the value is stored.



How It Works

1. **Key** is passed to a **hash function**.
2. Hash function returns an **index**.
3. The **value is stored** at that index.
4. When retrieving, the key is hashed again to find the value.

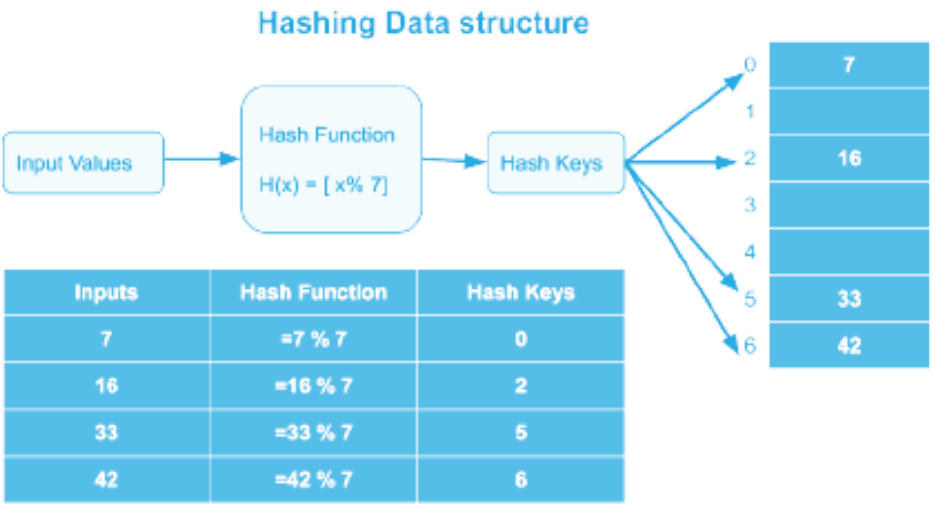
Hash Functions

What is a Hash Function?

A **hash function** is an algorithm that takes **input (key)** and returns a **fixed-size integer (hash code)** — usually used as an **index** in a hash table.

Example:

```
Input key: "apple"
→ Hash code: 3
→ Stored in table[3]
```



Inputs	Hash Function	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	6

Purpose of a Hash Function

- Map keys to **array indices**
- Ensure **fast access** to data
- Distribute data **evenly** to minimize **collisions**

Properties of a Good Hash Function

1. **Deterministic**: Same input always gives the same output.
2. **Efficient**: Should compute quickly.
3. **Uniform Distribution**: Spread keys evenly across buckets.
4. **Minimize Collisions**: Reduce chances of different keys mapping to the same index.

Simple Hash Functions (Examples)

🔹 1. Division Method

```
index = key % table_size;
```

- Simple for integer keys.
- Example: `key = 105`, `table_size = 10` → index = `105 % 10 = 5`

🔹 2. Multiplicative Method

```
index = floor(table_size * (key * A % 1))
```

- A = constant (0 < A < 1), often A = 0.618...

🔹 3. String Hash Function (Common in Java)

Java's `String.hashCode()` method (simplified):

```
int hash = 0;
for (int i = 0; i < str.length(); i++) {
    hash = 31 * hash + str.charAt(i);
}
```

Example:

```
String key = "abc";
hash = ((97 * 31 + 98) * 31 + 99) = 96354
```

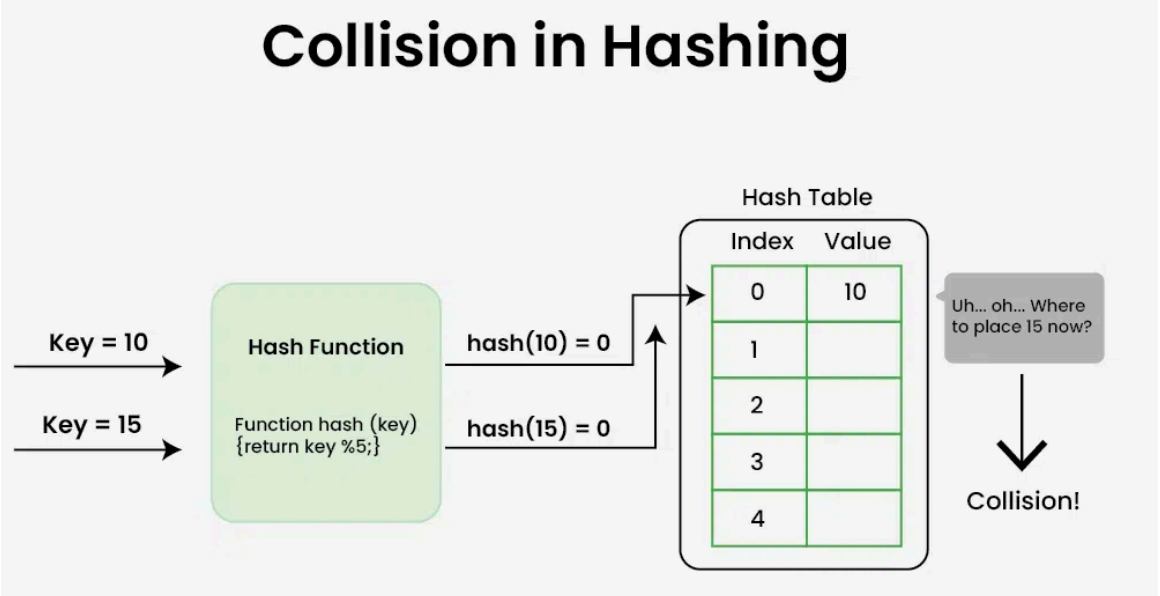
Then we do:

```
index = hash % table_size;
```

Hash Collisions

What are Hash Collisions?

- **When two different keys produce the same hash index**
- Example: `"abc"` and `"acb"` might map to the same slot



- **Handled by:**
    - **Open addressing** (e.g., linear probing)
    - **Chaining** (linked list at each index)

Open Addressing

When two keys hash to the **same index**, **Open Addressing** finds another **empty slot** in the table to store the new value — all elements are stored **within the hash table itself**.

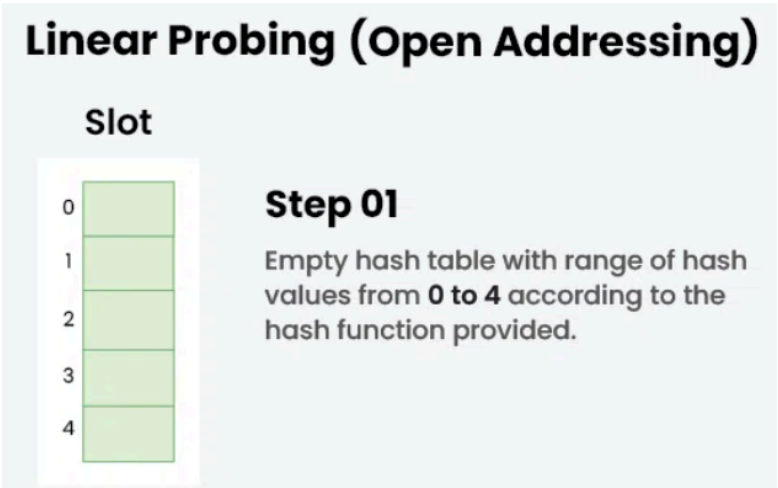
Main Types of Open Addressing:

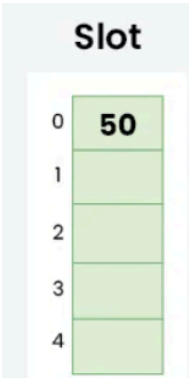
1. Linear Probing

- Check the **next** slot: `(hash + 1) % tableSize`, then `(hash + 2) % tableSize`, etc.

- **Simple but can lead to clustering**

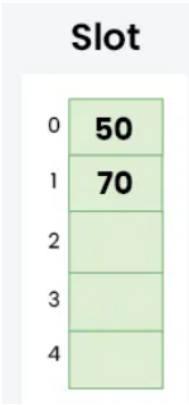
Example: Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.





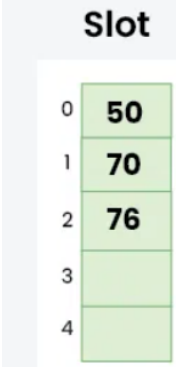
Step 02

The first key to be inserted is **50** which is mapped to **slot 0 (50%5=0)**



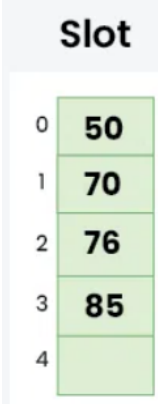
Step 03

The next key is **70** which is mapped to **slot 0 (70%5=0)** but **50** is already at **slot 0** so, search for the next empty slot and insert it.



Step 04

The next key is **76** which is mapped to **slot 1 (76%5=1)** but **70** is already at **slot 1** so, search for the next empty slot and insert it.



Step 05

The next key is **85** which is mapped to **slot 0 (85%5=0)**, but **50** is already at **slot number 0** so, search for the next empty slot and insert it. So insert it into **slot number 3**.

Open Addressing (Linear Probing) – Java Code

```
public class OpenAddressingHashTable {
    private int[] table;
    private int size;
    private final int EMPTY = -1;

    public OpenAddressingHashTable(int size) {
        this.size = size;
        table = new int[size];
        for (int i = 0; i < size; i++)
            table[i] = EMPTY;
    }

    private int hash(int key) {
        return key % size;
    }

    public void insert(int key) {
        int index = hash(key);
        int startIndex = index;

        while (table[index] != EMPTY) {
```

```
            index = (index + 1) % size;
            if (index == startIndex) {
                System.out.println("Table full!
Cannot insert " + key);
                return;
            }
        }
        table[index] = key;
    }

    public boolean search(int key) {
        int index = hash(key);
        int startIndex = index;

        while (table[index] != EMPTY) {
            if (table[index] == key)
                return true;
            index = (index + 1) % size;
            if (index == startIndex)
                break;
        }
        return false;
    }

    public void delete(int key) {
        int index = hash(key);
        int startIndex = index;

        while (table[index] != EMPTY) {
            if (table[index] == key) {
                table[index] = EMPTY;
                return;
            }
            index = (index + 1) % size;
```

```
                if (index == startIndex)
                    return;
            }
        }

        public void display() {
            for (int i = 0; i < size; i++)
                System.out.println("Index " + i + ": " +
(table[i] == EMPTY ? "empty" : table[i]));
        }

        public static void main(String[] args) {
            OpenAddressingHashTable ht = new
OpenAddressingHashTable(7);
            ht.insert(10);
            ht.insert(17);
            ht.insert(24);

            ht.display();


            System.out.println("Search 17: " +
ht.search(17));
            ht.delete(17);
            ht.display();
        }
    }
```

Output:

```
Index 0: empty
Index 1: empty
Index 2: empty
Index 3: 10
```

```
Index 4: 17
Index 5: 24
Index 6: empty
Search 17: true
Index 0: empty
Index 1: empty
Index 2: empty
Index 3: 10
Index 4: empty
Index 5: 24
Index 6: empty
```

Code Summary:

```
Inserted keys: 10, 17, 24
 Table size: 7
 Hash function: key % 7
```

Hash calculations:

- `10 % 7 = 3 → index 3`
- `17 % 7 = 3 → collision → next available index = 4`
- `24 % 7 = 3 → collision → index 4 taken → next = 5`

2. Quadratic Probing

- Jumps increase quadratically.
- Reduces clustering but might skip available slots.

Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i^2 . Insert = 22, 30, and 50.

Quadratic Probing (Open Addressing)

Step 01

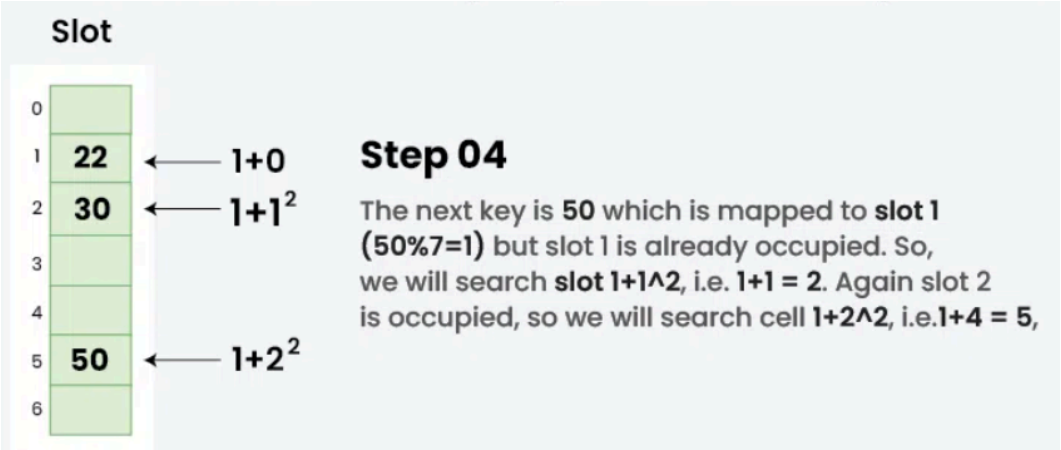
Empty hash table with range of hash values from **0 to 6** according to the hash function provided.

Step 02

The first key to be inserted is **22** which is mapped to **slot 1 (22%7=1)**

Step 03

The next key is **30** which is mapped to **slot 2 (30%7=2)**



2. Double Hashing

- **Uses second hash function to calculate step size:**

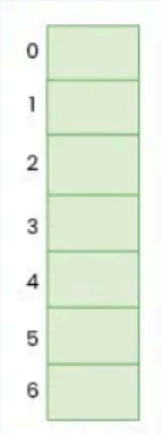
```
index = (hash1(key) + i * hash2(key)) % tableSize
```

- **Best at avoiding clustering and spreading entries well.**

Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is h1(k) = k mod 7 and second hash-function is h2(k) = 1 + (k mod 5)

Double Hashing (Open Addressing)

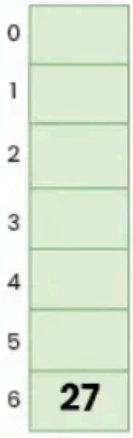
Slot



Step 01

Empty hash table with range of hash values from **0 to 6** according to the hash function provided.

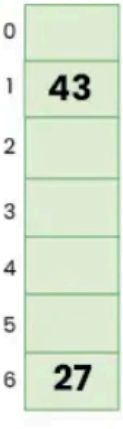
Slot



Step 02

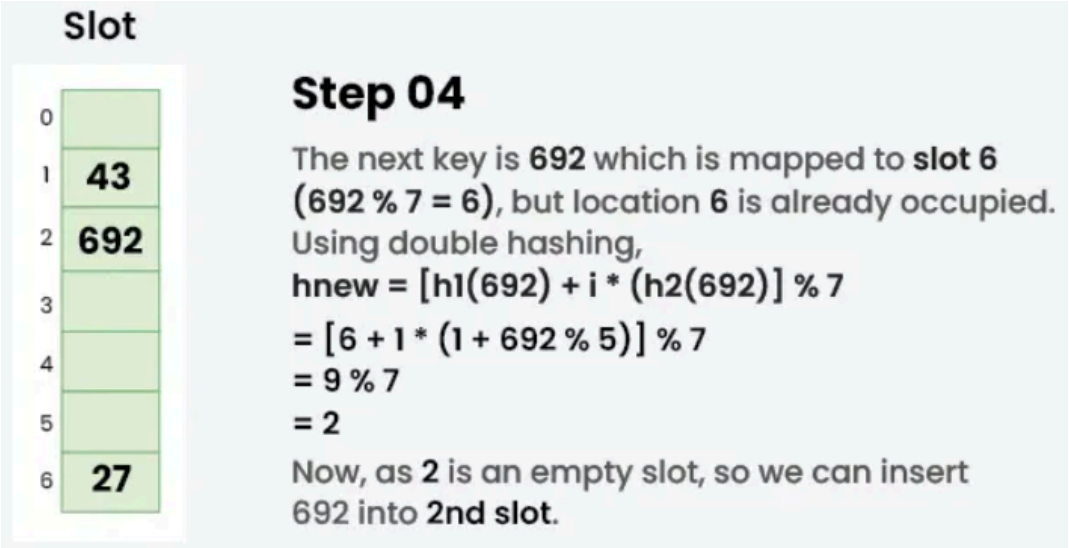
The first key to be inserted is **27** which is mapped to **slot 6 (27%7=6).**

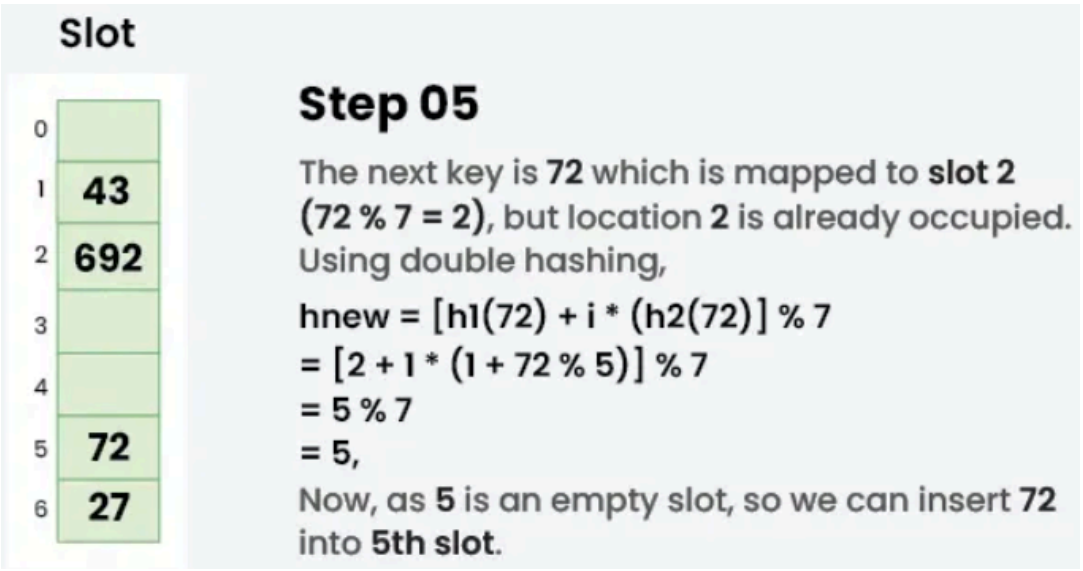
Slot



Step 03

The next key is **43** which is mapped to **slot 1 (43%7=1).**





Separate Chaining

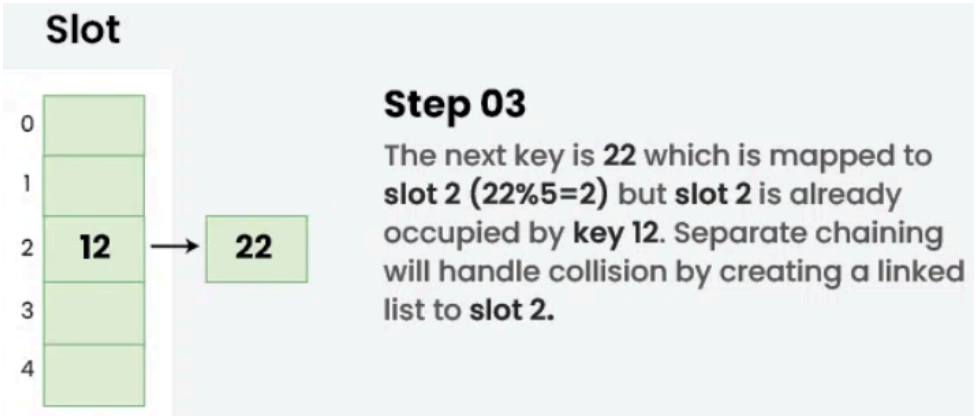
- **Separate Chaining** is a **collision resolution** method where each slot in the hash table holds a **linked list** (or any other dynamic structure) of all elements that hash to the same index.

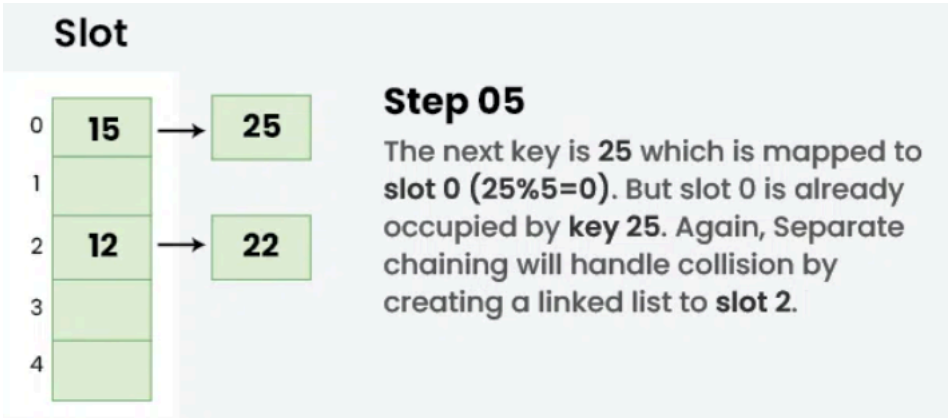
If multiple keys produce the **same hash index**, they are stored in a **chain (list)** at that index.

How It Works:

1. Use a **hash function** to map the key to an index.
2. At that index, store a **linked list (chain)** of key-value pairs.
3. On collision, just **append the new key-value pair** to the list at that index.

Example: Let us consider a simple hash function as "**key mod 5**" and a sequence of keys as 12, 22, 15, 25





Separate Chaining – Java Code

```
import java.util.*;

public class SeparateChainingHashTable {
    private int size;
    private LinkedList<Integer>[] table;
```

```
@SuppressWarnings("unchecked")
public SeparateChainingHashTable(int size) {
    this.size = size;
    table = new LinkedList[size];
    for (int i = 0; i < size; i++)
        table[i] = new LinkedList<>();
}

private int hash(int key) {
    return key % size;
}

public void insert(int key) {
    int index = hash(key);
    table[index].add(key);
}

public boolean search(int key) {
    int index = hash(key);
    return table[index].contains(key);
}

public void delete(int key) {
    int index = hash(key);
    table[index].remove((Integer) key);
}

public void display() {
    for (int i = 0; i < size; i++) {
        System.out.print("Index " + i + ": ");
        for (int key : table[i]) {
            System.out.print(key + " -> ");
        }
        System.out.println("null");
```

```
        }
    }

    public static void main(String[] args) {
        SeparateChainingHashTable ht =
newSeparateChainingHashTable(5);
        ht.insert(10);
        ht.insert(15);
        ht.insert(20);
        ht.insert(7);

        ht.display();
        System.out.println("Search 15: " + ht.search(15));
        ht.delete(15);
        ht.display();
    }
}
```

Output:

```
Index 0: 10 -> 15 -> 20 -> null
Index 1: null
Index 2: 7 -> null
Index 3: null
Index 4: null
Search 15: true
Index 0: 10 -> 20 -> null
Index 1: null
Index 2: 7 -> null
Index 3: null
Index 4: null
```

Code Summary:

Inserted keys: `10, 15, 20, 7`

Table size: `5`

Hash function: `key % 5`

Hash calculations:

- `10 % 5 = 0` → index 0
- `15 % 5 = 0` → index 0 (collision)
- `20 % 5 = 0` → index 0 (collision)
- `7 % 5 = 2` → index 2

Advantages:

- Simple to implement
- Hash table never fills up, we can always add more elements to the chain
- Less sensitive to has function and load factors
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted

Disadvantages:

- The cache performance is not good
- Wastage of space
- Use extra space for links

Comparison of Open Addressing & Separate Chaining

Feature	Open Addressing	Separate Chaining

Structure	Array only	Array of linked lists
Memory usage	Less	More
Insertion	Probing	Append to chain
Deletion	Complex	Easy
Performance degrade	Rapid	Gradual
Implementation	More cache-friendly	Simpler conceptually

Hashing Efficiency

Hashing is considered one of the most efficient techniques for storing and retrieving data — especially when fast insertions, deletions, and lookups are needed.

Why Is Hashing Efficient?

Average-Case Time Complexity

Operation	Time (Average Case)
Insert	O(1)
Search	O(1)
Delete	O(1)

This makes hashing **faster than arrays, lists, binary trees**, and even balanced BSTs (which are O(log n)) for key-based access.

When Hashing Becomes Inefficient

- **Poor hash function** → causes **collisions**
- **High load factor** (too many items in small table) → performance degrades
- **Collision resolution strategy** (like probing or chaining) affects performance

Load Factor (α)

```
α = number of elements / table size
```

- A **low load factor** (e.g., 0.5) = fast operations

- A **high load factor** (e.g., > 0.7) = more collisions → degraded performance

Factors Affecting Hashing Efficiency

1. **Hash Function Quality**

   - Good hash functions spread keys **uniformly**
   - Poor functions lead to **clustering/collisions**

2. **Table Size**

   - Prime number table sizes reduce patterns
   - Dynamic resizing helps avoid overflow

3. **Collision Resolution Method**

   - **Separate Chaining** handles collisions well even at high load
   - **Open Addressing** is efficient but degrades quickly if the table is too full

Tips to Maximize Efficiency

- Use a **good hash function**

- **Resize (rehash)** the table when the load factor is too high
- Choose a **collision resolution method** that suits your data size and access pattern