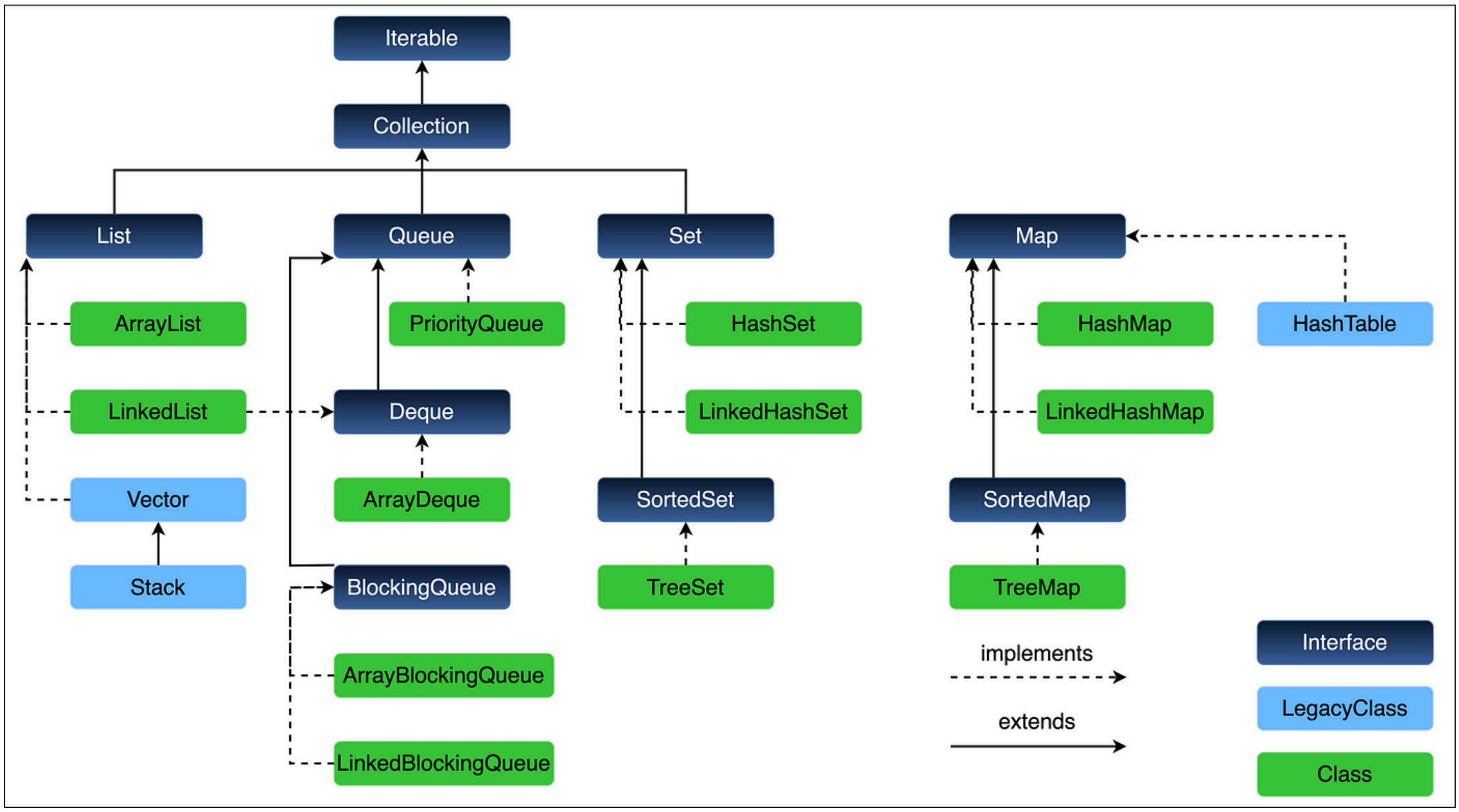
COLLECTION FRAMEWORK IN JAVA

By Mohan Barle...

What is a Collection?

Collections are used to perform operations such as **searching**, **sorting**, **insertion**, **deletion**, and manipulating data efficiently. The Java Collections Framework (JCF) is a set of **classes** and **interfaces** that implement commonly reusable collection **data structures**.



Key Features of Collections in Java:

- Dynamic Size:
- Homogeneous and Heterogeneous:
- Predefined Methods:

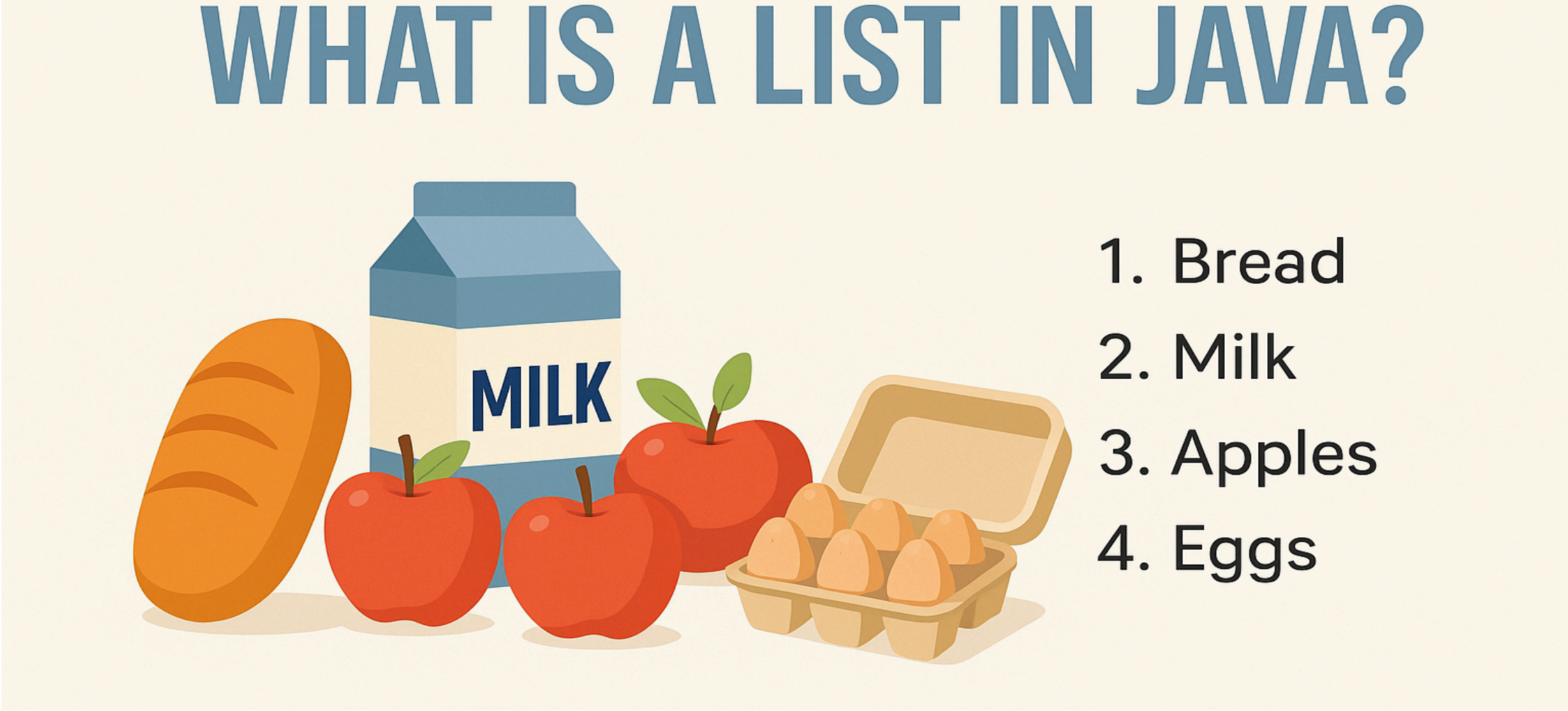
Time Complexity vs Space Complexity

- Time complexity → measures speed
- Space complexity → measures memory usage

Internal Working – How it works inside

It counts **how many operations** run relative to **input size (n)**.
Example:
Loop runs n times → **O(n)**
Nested loop runs n×n → **O(n²)**
Index access in array → **O(1)**

Time complexity cares only about **growth**, not actual speed.



What is a list in java?

- ➢ **List is an interface**
- ➢ **Duplicate elements are allowed**
- ➢ Elements to be accessed by their **index (position)**
- ➢ **Insertion order** to be preserved
- ➢ It is part of **java. util** package.
- ➢ List implementation **classes**: **ArrayList**, **Stack**, **Vector**, and **LinkedList**.

Why do we use List in Java? (Need of List)

- ➢ To **store multiple values** in a single variable.
- ➢ To **add, remove, update, or read** elements easily.
- ➢ It is **dynamic**, meaning it can grow or shrink in size (unlike arrays).

Internal Working – How it works inside

List is an **interface**, not a storage structure.
Storage depends on the implementation:

- **ArrayList** → uses dynamic array
- **LinkedList** → uses doubly linked list
- **Vector** → synchronized dynamic array
- **Stack** → LIFO structure

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("A");
    list.add("B");
    list.add("C");
    list.add("B"); // duplicates allowed
    System.out.println(list.get(0));  // A
    System.out.println(list.get(1));  // B
    for (String s : list) {
      System.out.println(s);
    }
  }}
```

Performance – Big O time complexity

(For ArrayList implementation)

- Access by index → **O(1)**
- Add at end → **O(1)**
- Add/remove in middle → **O(n)**
- Search → **O(n)**

ArrayList is a **resizable array** (also called a **dynamic array**) in Java.
It is a **class** in the java.util package and implements the **List interface**.

It works **like an array**, but the **size can change** automatically — increase when you add elements and decrease when you remove elements.

Non-technical example

Imagine a **shopping bag** that expands when you put more items inside.
No fixed size — it grows as needed.
This is how ArrayList behaves.

Why do we use ArrayList? (Need)

- Maintains **insertion order**.
- Allows **duplicate elements**.
- Not **synchronized** (i.e., not thread-safe).

Coding Example:

```
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
    // Create ArrayList
    ArrayList<String> fruits = new ArrayList<>();
    // Add items
    fruits.add("Apple");
    fruits.add("Banana");
    fruits.add("Mango");
    // Print all fruits
    System.out.println("Fruits List:"+ fruits);
  }}
```

Internal working

ArrayList uses a **dynamic array** internally. When the array becomes full, it creates a **new bigger array** and copies old items into it.

Performance

- Reading item by index → very fast
- Adding at end → fast
- Adding in middle → slow (shifting happens)
- Searching → medium speed

Default Capacity and How Size Grows

➡️ Internally, it creates an **array of size 10** (default capacity is 10).

```
ArrayList<Integer> list = new ArrayList<>();
```

```
ArrayList<String> list = new ArrayList<>(50); // starts with 50 capacity
```

ArrayList Method in

remove(int index): Removes an element at a specific index.

```
  list.remove(1);
```

---

remove(Object o): Removes the first occurrence of the specified element.

```
 list.remove("Java");
```

---

get(int index): Returns the element at the specified index in the arraylist.

```
String lang = list.get(0);
```

---

set(int index, E element): Updates the element at the specified index.

```
  list.set(0, "C++");
```

Searching Elements in ArrayList

---

contains(Object o): Returns true if the element exists in the list

```
boolean exists = list.contains("Python");
```

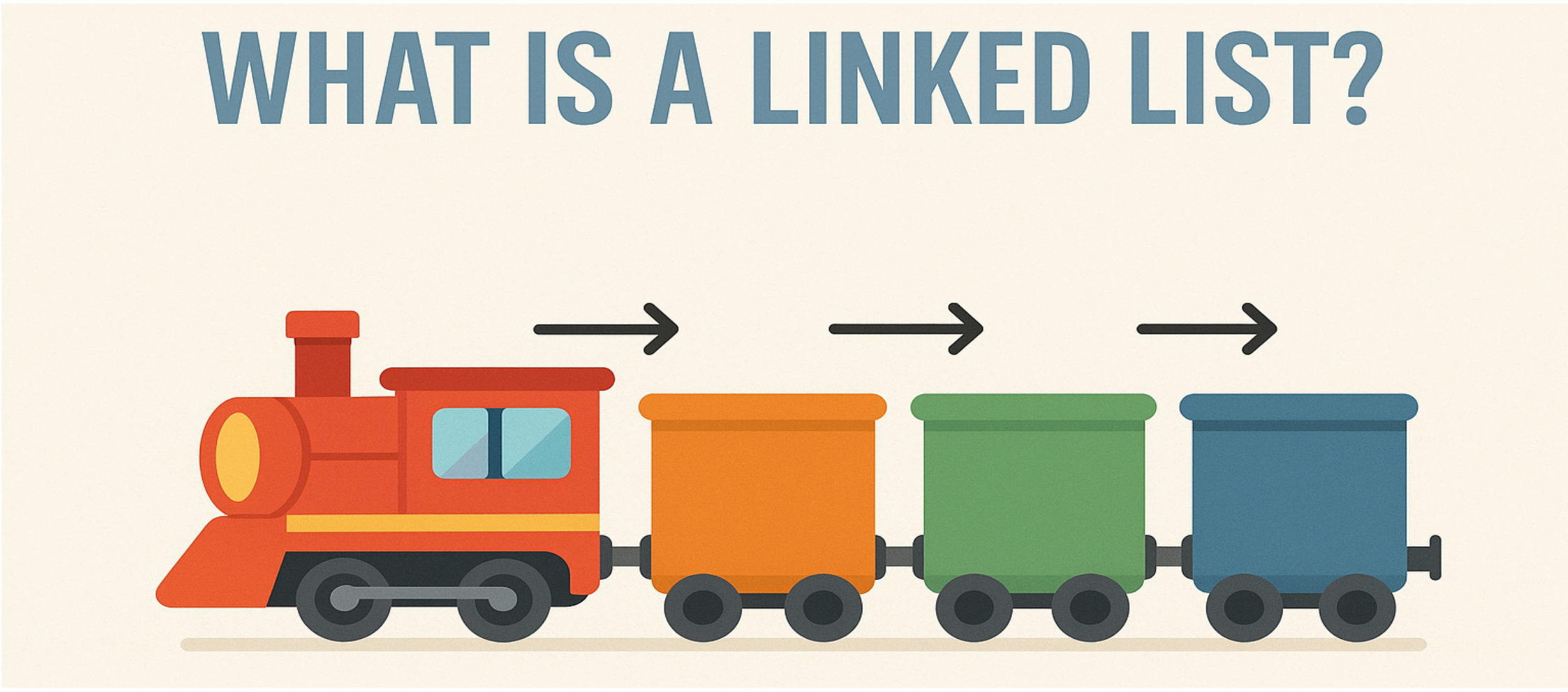
---

isEmpty(): Checks if the list is empty or not.

```
 boolean empty = list.isEmpty();
```

clear(): Removes all elements from the list.

```
list.clear();
```



What is a LinkedList?

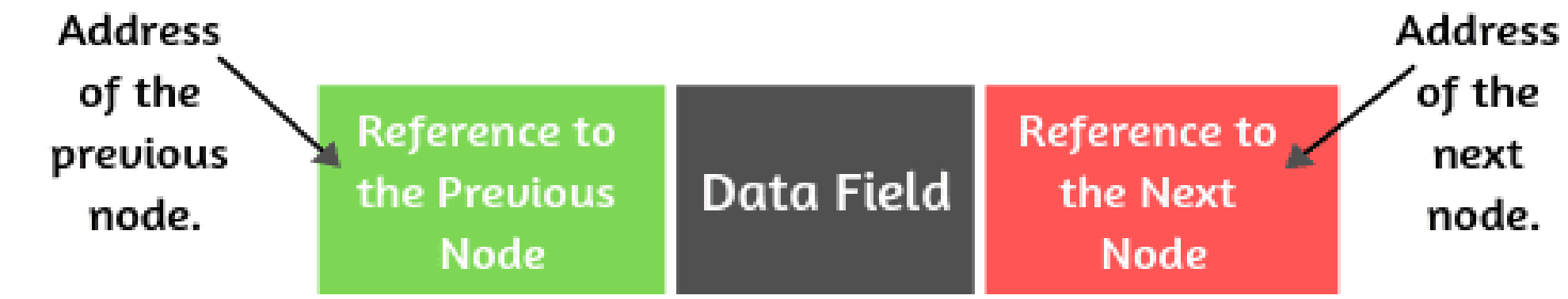
LinkedList is a **class** in Java used to **store a group of elements** (like an array, but more flexible). It is part of the **java.util package**.

A **LinkedList** in Java is a collection where data is stored in small connected parts called **nodes**.Each node stores **data** and **links** (addresses) that connect it to other nodes. This creates a **chain-like structure** in memory.

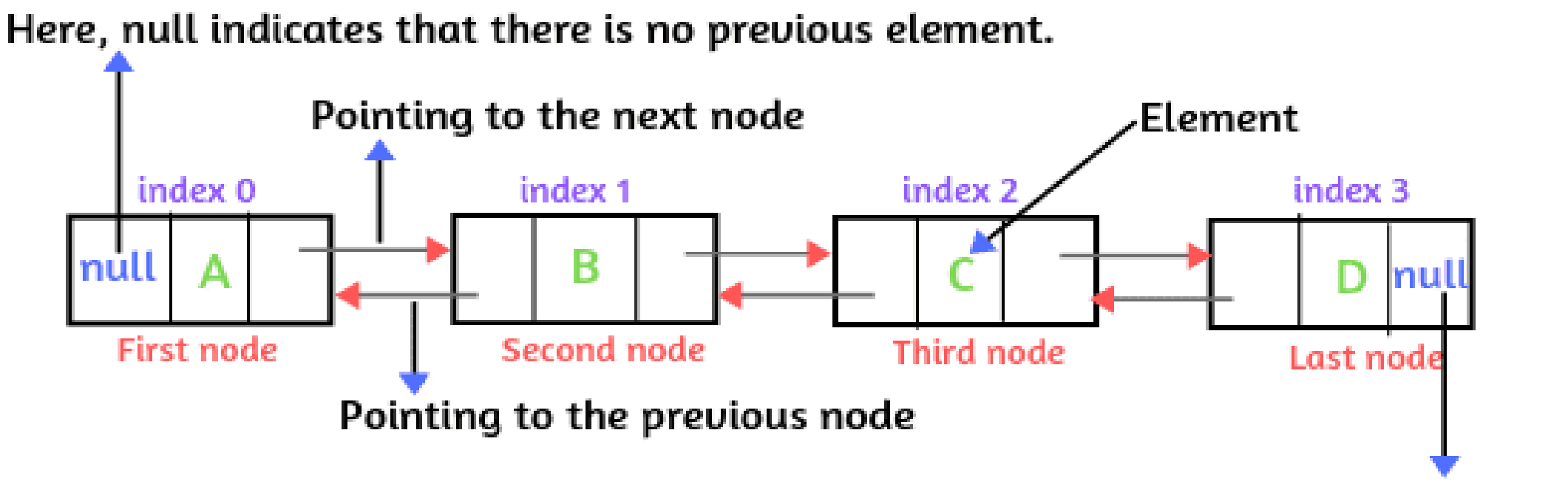
Non-technical example

Think of a **chain of paper clips**: Each clip is connected to the next one.
If you want to remove or insert a clip, you just open that part — you don't need to move the whole chain.

That's exactly how a LinkedList works.
Each node **(paper clip)** connects to the next (and previous in doubly-linked list).



Representation of Java LinkedList Node



A array representation of linear Doubly LinkedList in Java

Key features

- Elements are stored as **nodes**, not in continuous memory
- **Each node has:**
  - **data** → stores value
  - **next** → address of next node
  - **previous** → address of previous node (for doubly linked list)
- Maintains insertion order
- Slower access but faster insertion/removal

Coding Example:

```
import java.util.LinkedList;
public class Main {
  public static void main(String[] args) {
    LinkedList<String> friends = new LinkedList<>();
    // Add friends
    friends.add("Rahul");
    friends.add("Mohan");
    friends.add("Sita");

    // Print all friends
    System.out.println("Friends List:"+ friends);
  }
}
```

Important methods

- add() → add element at end
- addFirst() → add at beginning
- addLast() → add at end
- remove() → remove an element
- get() → access element by index (slow traversal)
- size() → get total number of elements
- clear() → remove all elements

---

7. Performance

- Add at beginning → **O(1)**
- Remove at beginning → **O(1)**
- Add/remove in middle → **O(1)** (after reaching position)
- Search by index → **O(n)** (slow)

⚡ So, LinkedList = Fast **add/remove**, slow access.

---

8. Differences (LinkedList vs ArrayList)

- **LinkedList:** fast add/remove, slow search
- **ArrayList:** fast search, slow add/remove

Feature	LinkedList	ArrayList
Storage	Nodes	Dynamic Array
Speed (Add/Remove)	Fast	Slow (shifting)
Speed (Access)	Slow	Fast
Memory	More (because of links)	Less
Order	Maintains insertion order	Maintains insertion order

Where to use in real projects

- **Messaging apps** → chat history as nodes
- **Undo/Redo** → previous & next actions
- **Music playlist** → next and previous song navigation

LinkedList Methods:

---

addFirst(E element): Adds an element at the beginning of the list.

```
list.addFirst(E);
```

---

addLast(E element): Adds an element at the end of the list.

```
list.addLast(E);
```

---

remove(int index): Removes the element at the specified index.

```
list.remove(int);
```

---

remove(Object o): Removes the first of the specified element.

```
list.remove(Object);
```

---

removeFirst(): Removes the first element from the list.

```
list.removeFirst();
```

---

removeLast(): Removes the last element from the list.

```
list.removeLast();
```

---

clear(): Removes all elements from the list.

```
list.clear();
```

---

getFirst(): Retrieves the first element of the list.

```
list.getFirst();
```

getLast(): Retrieves the last element of the list.

```
list.getLast();
```

---

contains(Object o): Checks if the list contains the specified element.

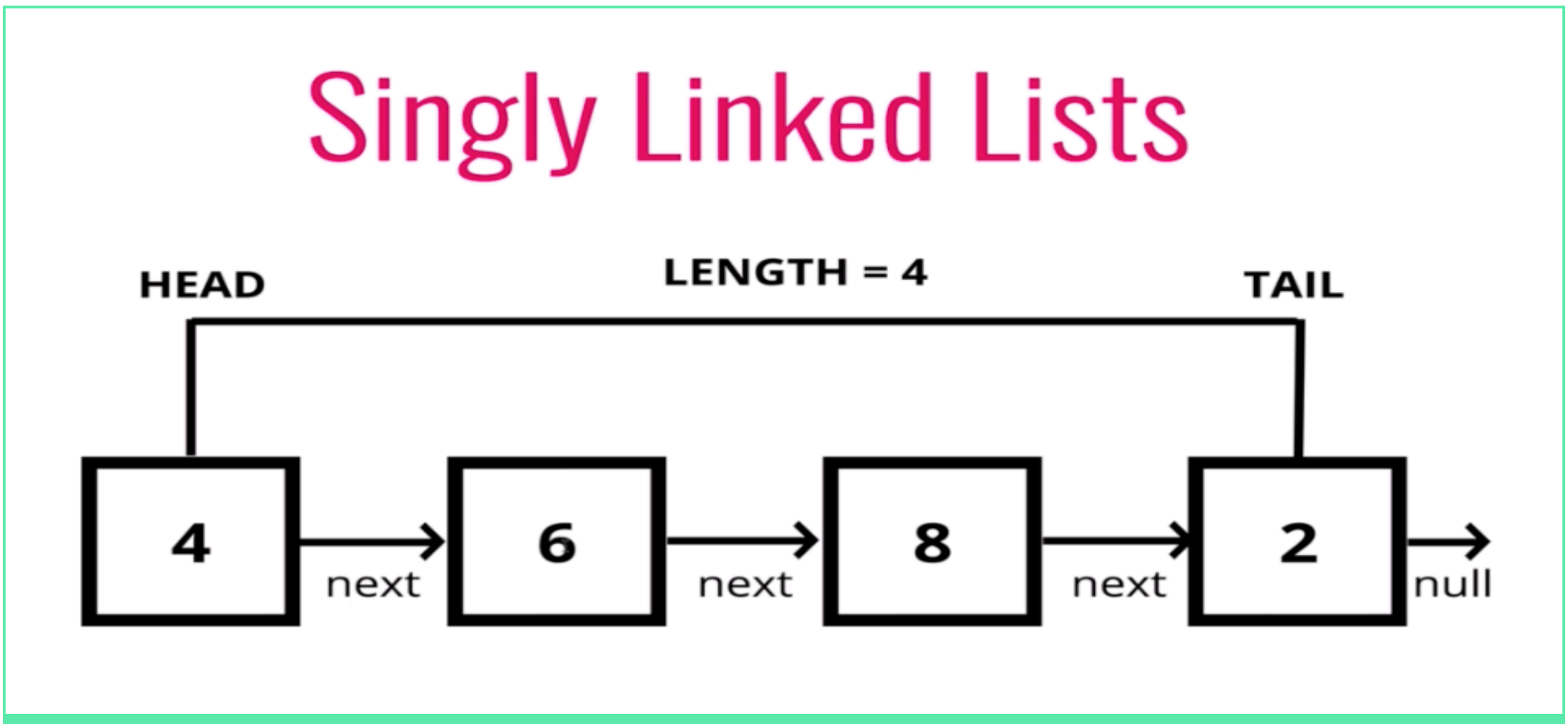
```
list.contains(Object);
```

clone(): Creates a shallow copy of the list. Creates a duplicate list.

```
LinkedList<Integer> clonedList = (LinkedList<Integer>) list.clone();
```

Types of Linked Lists:

- ***Singly Linked List***
- ***Doubly Linked List***
- ***Circular Linked List***



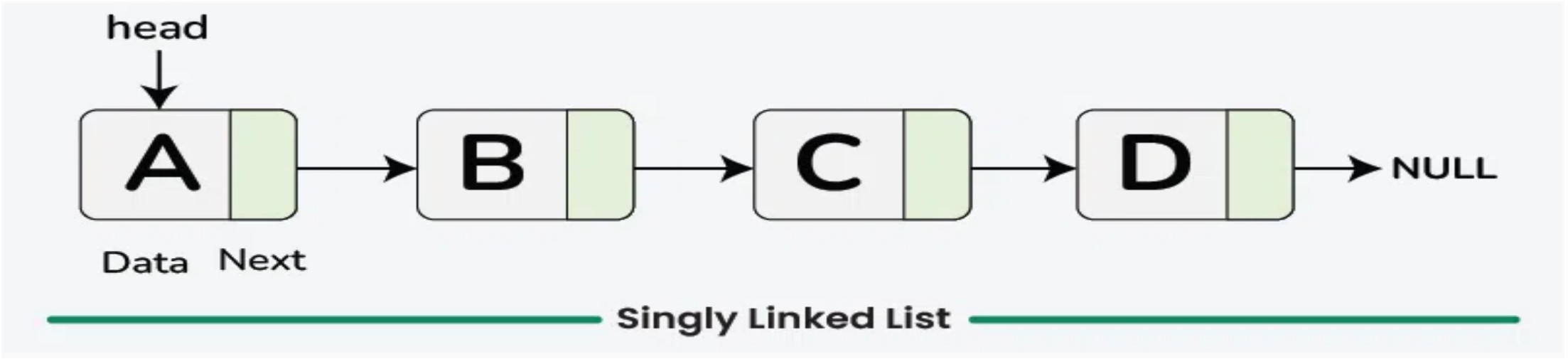
A **Singly Linked List** is a linear data structure where each element (called a **node**) stores **data** and a **link (address)** to the **next node** only.

It connects all nodes in **one direction** — from head to tail.

Non-technical example

Think of **people standing in a queue**, where each person **knows who is next**, but **not who is behind** them.
You can only move forward, not backward.

That's exactly how a Singly Linked List works — one-way connection.



```java
import java.util.LinkedList;

public class Main {
  public static void main(String[] args) {
      LinkedList<Integer> list = new LinkedList<>();
      // Add elements
      list.add(10);
      list.add(20);
      list.add(30);
      // Print all elements
      System.out.println("LinkedList:"+ list);
  }
}
```

Key features

- Each node has two parts:
    1. `data` → value of the node
    2. `next` → address (**reference**) to next node
- Connected in one direction (→).
- Ends with a **null pointer** (tail node).

Important methods

- addFirst(data) → insert at beginning
- addLast(data) → insert at end
- delete(data) → remove specific node
- display() → print all nodes
- search(data) → find a node

When to use (real-life example)

- You only need **forward traversal**
- You need **simple add/delete** from the front

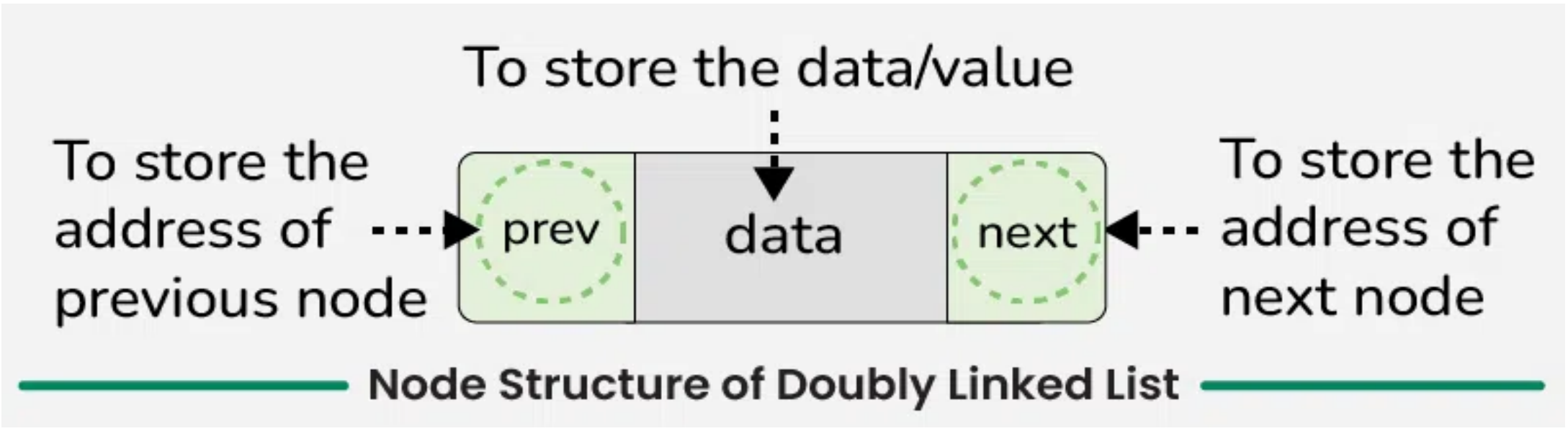
Example: a **print queue** or **playlist playing one direction only**

Doubly Linked List

A **Doubly Linked List (DLL)** is a data structure where:

- Each node contains **three parts**:
  - **data** (value)
  - **next** (pointer to the next node)
  - **prev** (pointer to the previous node)

This allows **forward and backward traversal** of the list.



Non-technical example

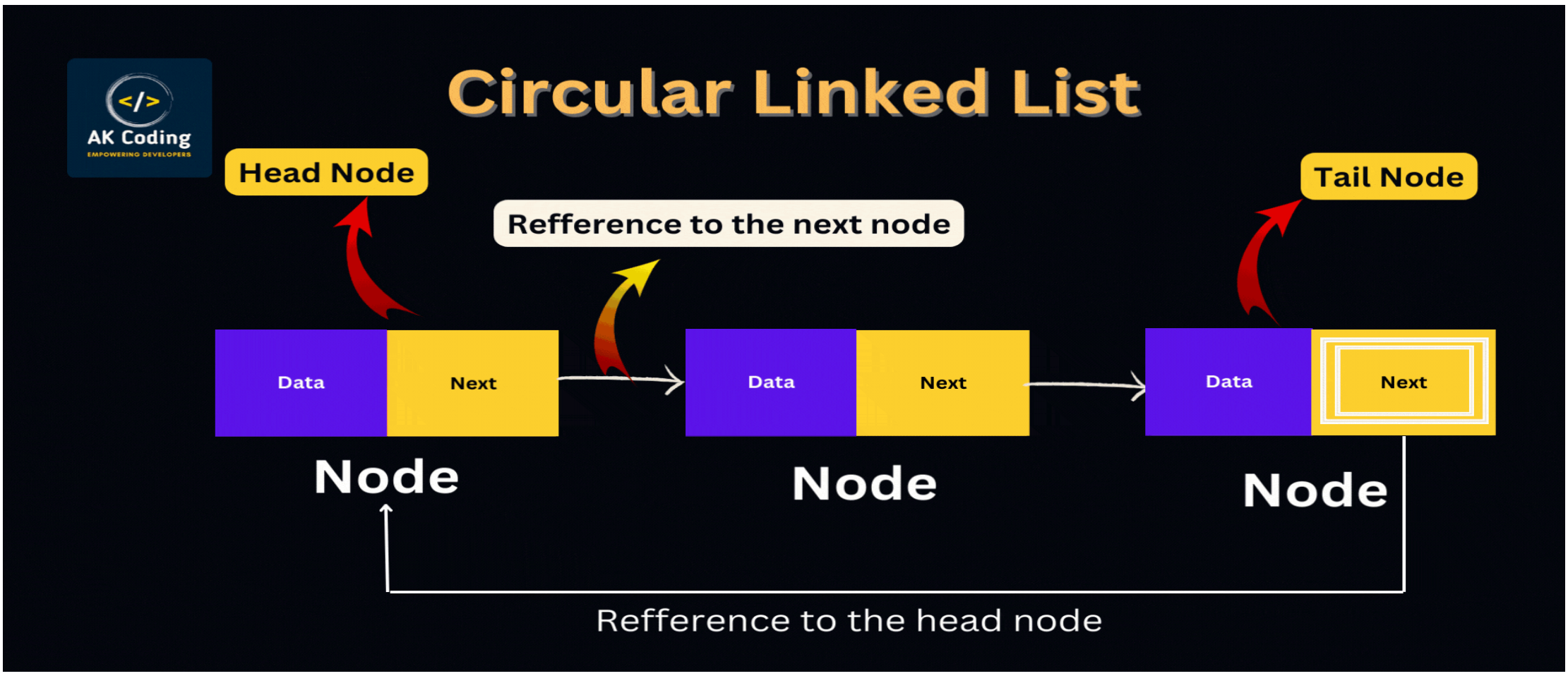
Imagine a **two-way train** where every coach is connected both **front and back**. You can walk **forward or backward** through the coaches easily. That's how a doubly linked list works — two connections per node.

Example: A **music playlist** — you can go "Next Song" or "Previous Song".

```java
public class Main {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();

        list.add("A");
        list.add("B");
        list.add("C");
        System.out.println(list);  // Output: [A, B, C]

        list.addFirst("Start");
        list.addLast("End");
        System.out.println(list);  // Output: [Start, A, B, C, End]
```

🔑 Key Features of Doubly Linked List (via `LinkedList` class):

- ➢ **Easy to delete a node** from the middle or end.
- ➢ **Efficient insertion/deletion** at both ends (front and back).
- ➢ Used when you need to move **backward and forward** in a list.



What is a Circular Linked List

A **Circular Linked List** is a **linear data structure** where the **last node points back to the first node**, forming a **circle**.

Non-technical example

Imagine **people sitting in a circle passing a ball** — when the ball leaves the last person, it goes back to the first one.
That's how data moves in a circular linked list — round and round.

🔑 Key Features:

- The **last node points to the first node**.
- Traversal can start from **any node** and continue forever.

When to Use (Real-life Example)

When you want operations to **repeat continuously** —
like **game turns**, **playlist repeat**, or **CPU scheduling**.

Example: A **music player** that plays songs in a loop.

Where to Use in Real Projects

- **Playlist looping** in media players
- **Circular queues** in networking
- **Game player rotation systems**

What is a Vector?

- **Implements List interface** (so it works like an ArrayList).
- Stores **elements in order** (insertion order is maintained).
- **Thread-safe** – all methods are **synchronized**, which means it is safe to use in **multi-threaded programs**.
- Has an **initial capacity of 10**, and when full, it **doubles its size**.

Non-technical example

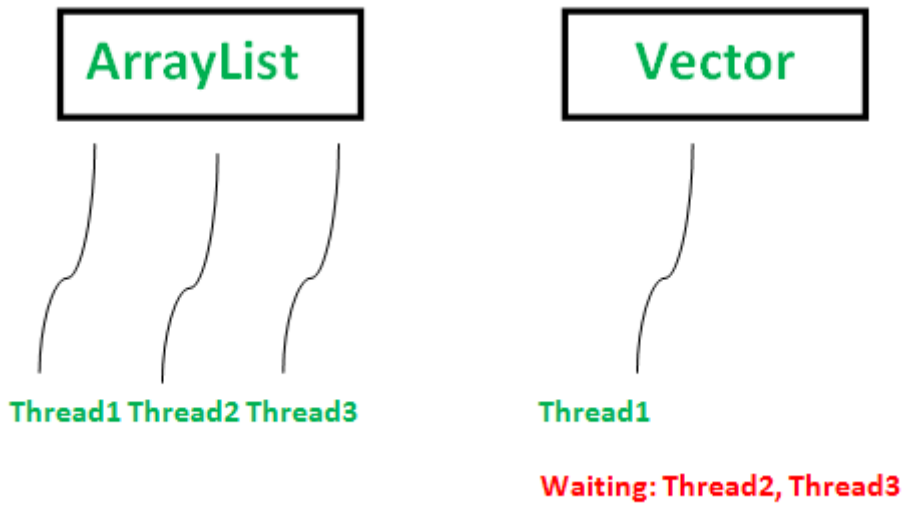
Imagine you have a **locker that can automatically expand** when you add more items. Also, the locker has a **security guard**, so only one person can use it at a time.

This is what Vector does:

- Automatically grows
- Only one thread can use it at a time (synchronized)

Why Use a Vector?

✔ **It is synchronized** (safe for multi-threading).



Key features

- Dynamic size (auto-grows)
- Thread-safe (synchronized)
- Maintains insertion order
- Slower than ArrayList because of synchronization
- Old legacy class

🔍 Interview Summary:

A **Vector** in Java is a **thread-safe**, **dynamic array** from the java.util package. It is similar to ArrayList but **synchronized**, making it suitable for **multi-threaded** environments.

Important methods

- add()
- addElement()
- remove()
- removeElement()
- get()
- size()
- capacity()

Internal working

- Internally uses a **dynamic array**
- When full, it increases in size (usually doubles capacity)
- Each operation is **synchronized** → only one thread uses it at a time

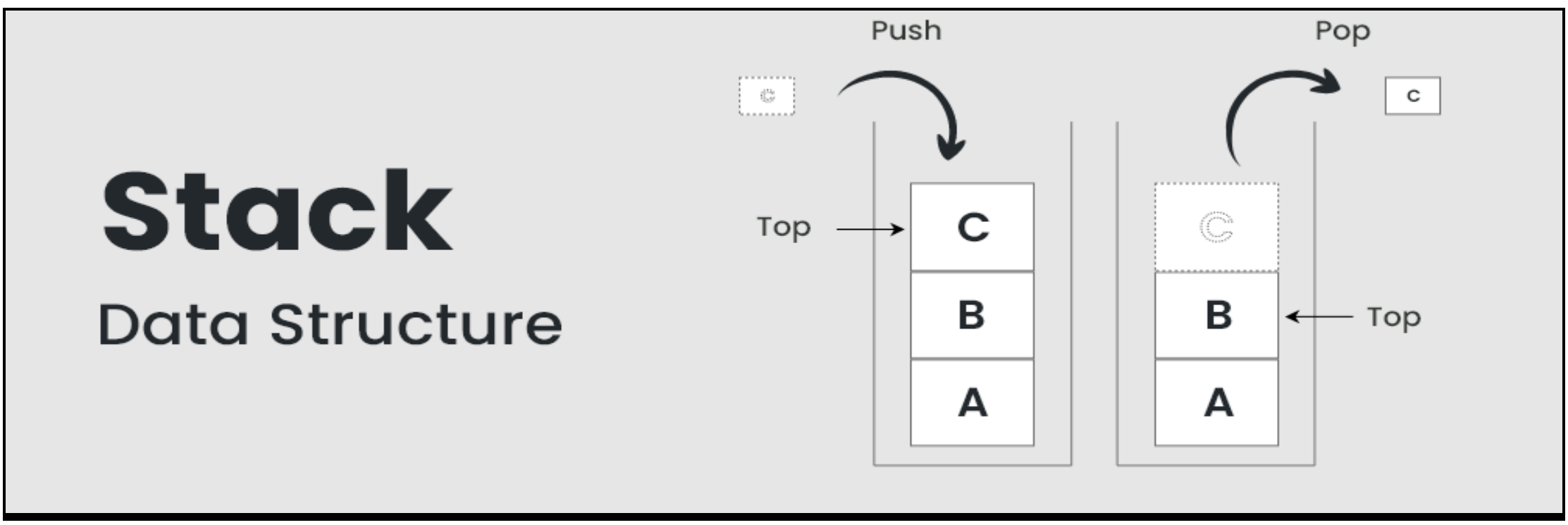
Performance

- Add → **O(1)** (amortized)
- Get → **O(1)**
- Remove → **O(n)** (shifting happens)
- Slower than ArrayList because of synchronization

Differences (Vector vs ArrayList)

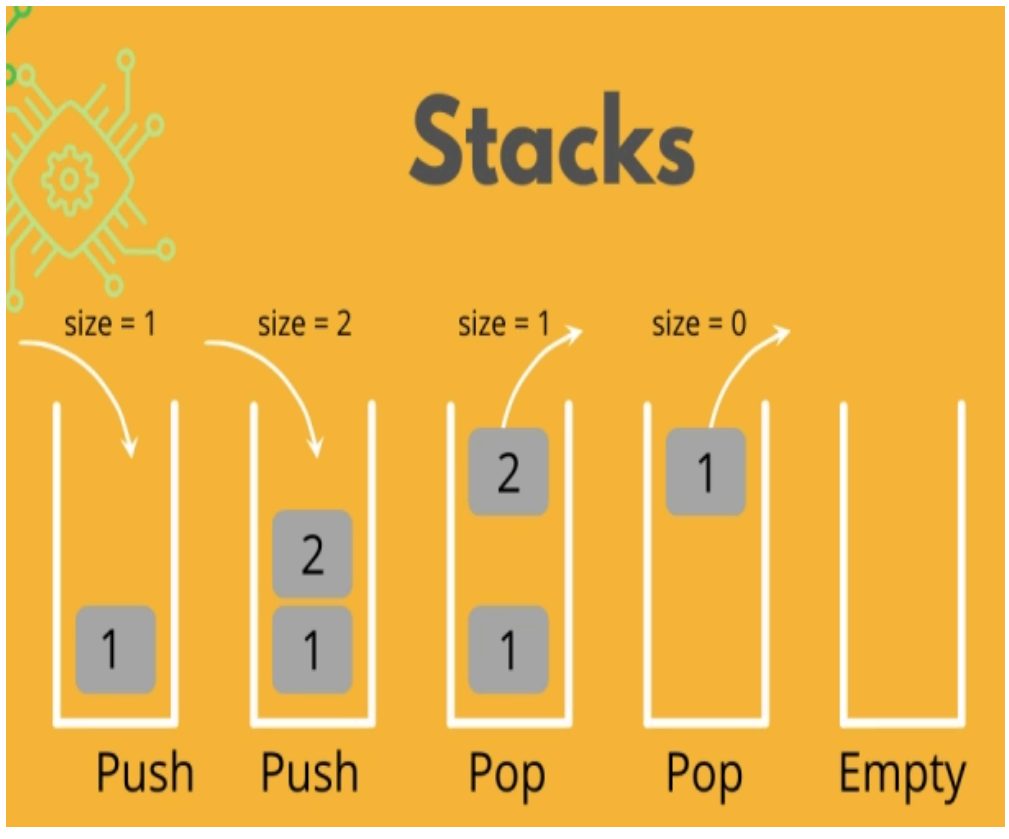
- **Vector:** synchronized (thread-safe), slower
- **ArrayList:** not synchronized (fast), modern

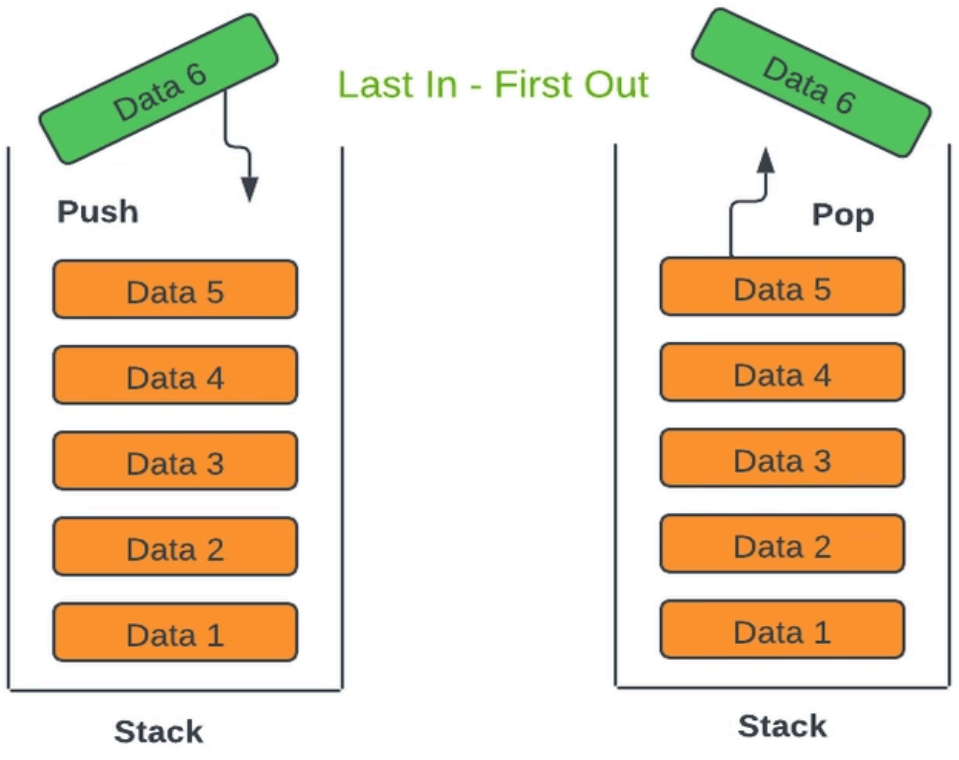
```java
public class Main {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();
        vector.add("Java");
        vector.add("Python");
        vector.add("C++");

        System.out.println(vector);         // Output: [Java, Python, C++]
        System.out.println(vector.get(1)); // Output: Python
    }
```



What is a stack?

- **A stack is a linear data structure** that follows **LIFO** (Last In, First Out) principle. It means the element that is **added last will be removed first**.
- Stack is a class in the `java.util` package.
- It **extends the Vector class**, so it inherits all Vector features.
- Allows **duplicates** and maintains **insertion order**.
- **Thread-safe** – because it inherits Vector's synchronized behavior.

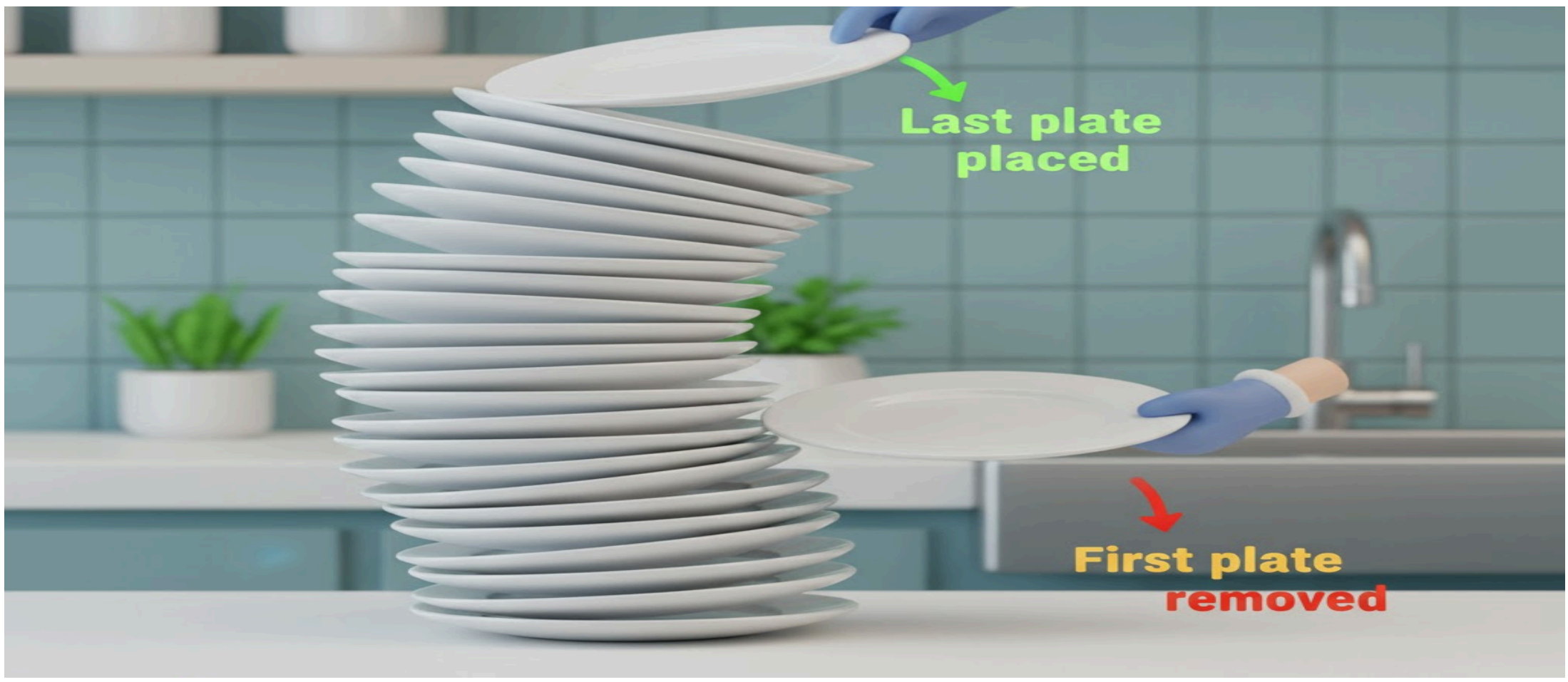




Non-technical example

Imagine a **stack of plates** in a kitchen. You always **put new plates on top**, and when you need one, you **take from the top** only.

✅ Last plate placed → first plate removed. This is exactly how Stack works.



Internal Working (How Stack works inside?)

- Stack stores elements in a **Dynamic Array** because it extends **Vector**.
- Whenever you call `push()`, an element is added at the **end of the array**.
- When you call `pop()`, it removes the **last element** of the array.
- So internally it is not a separate structure → it is a **Vector acting like a stack**.

Order inside memory (last element is the top):

🧠 Core Stack Operations (Methods):

- ➢ **push(E item)** – Adds an item to the top of the stack.
- ➢ **pop()** – Removes and returns the top item.
- ➢ **peek()** – Returns the top item **without removing** it.
- ➢ **empty()** – Checks if the stack is empty (returns true/false).
- ➢ **search(Object o)** – Returns position of element from top (1-based index), or -1 if not found.

Key features

- Works on **LIFO**
- Uses methods like push/pop
- Easy to add/remove from top
- Only top element is accessible
- Based on Vector internally

Performance

- push → **O(1)**
- pop → **O(1)**
- peek → **O(1)**
- search → **O(n)**

Very fast for top-based operations.

---

real-life example

- **Undo/redo** in text editors
- **Browser history** back button
- **Checking parentheses** in expressions

```java
public class Main {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();

        stack.push("Java");
        stack.push("Python");
        stack.push("C++");

        System.out.println(stack);         // Output: [Java, Python, C++]
        System.out.println(stack.pop());   // Output: C++ (removed)
        System.out.println(stack.peek()); // Output: Python
```

Most Commonly Used Stack Methods

1. push(element): Adds (inserts) an element on the top of the stack.

---

2. pop(): Removes and returns the top element. (Stack becomes smaller by one.)

---

3. peek() Returns the top element **without removing it**. Used to "look" at the top.

---

4. isEmpty() Checks whether the stack has no elements. Used before calling pop() to prevent error.

---

5. size() Returns how many elements are present in the stack.

---

6. search(element) Returns the position of the element from the top (1-based). Returns -1 if the element is not found.

⭐ *Bonus: What companies really test?*

✔ Parenthesis matching

✔ Undo operation

✔ Browser forward/back

All these use push, pop, and peek.

🔍 Interview Summary:

A **Stack** in Java is a **synchronized**, LIFO data structure from java.util. Package.
It provides built-in methods like push(), pop(), and peek() for stack operations.
It's commonly used in algorithms like **backtracking** and **recursion**

- **A set** is a **Collection** that **doesn't allow duplicate elements**.
- It represents a group of **unique elements**.
- Belongs to the java.util package.
- It is an **interface** that extends the Collection interface.
- Elements in a Set are **unordered** (no guarantee of insertion order unless using special types like **LinkedHashSet**).

📕 *real life Example*

A list of students present in the classroom.

- If Ganesh is present, you write his name once.
- You **never** write Ganesh multiple times.

Even if someone says "Add Ganesh again",

You say: **"He is already in the list."**

This is EXACTLY how a Set works.

⭐ *Set Uses Hashing Internally (HashSet, LinkedHashSet)*

When you add an item:

1. Java calculates a **hash value** using `hashCode()`.
2. If another item has the **same hash** *and is equal* (`equals()`), Java considers it a **duplicate**.
3. So it does **not insert it again**.

⭐*Because Java Has Another Structure for Duplicates*

Java already provides:

- **List** → allows duplicates
- **ArrayList** → allows duplicates
- **LinkedList** → allows duplicates

So Set was created to offer **something different**.

Technical Reason:

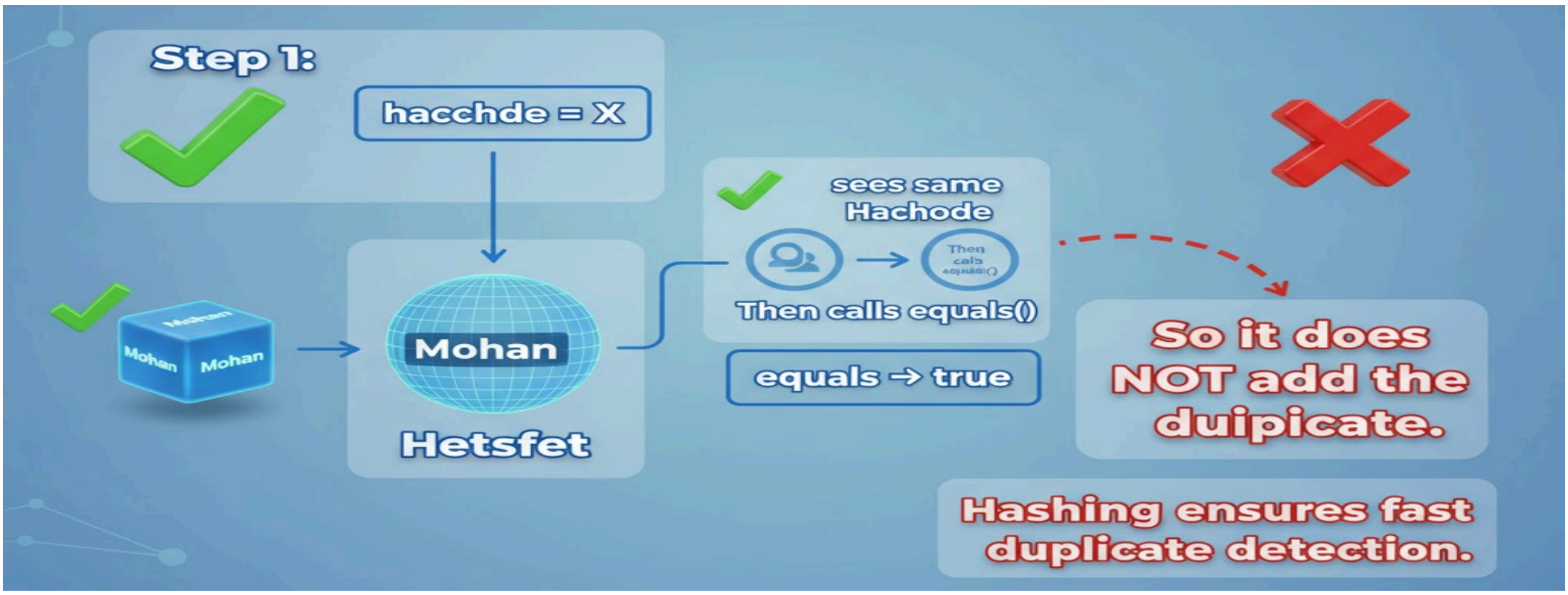
When you add an element:

```
set.add("Mohan");
```

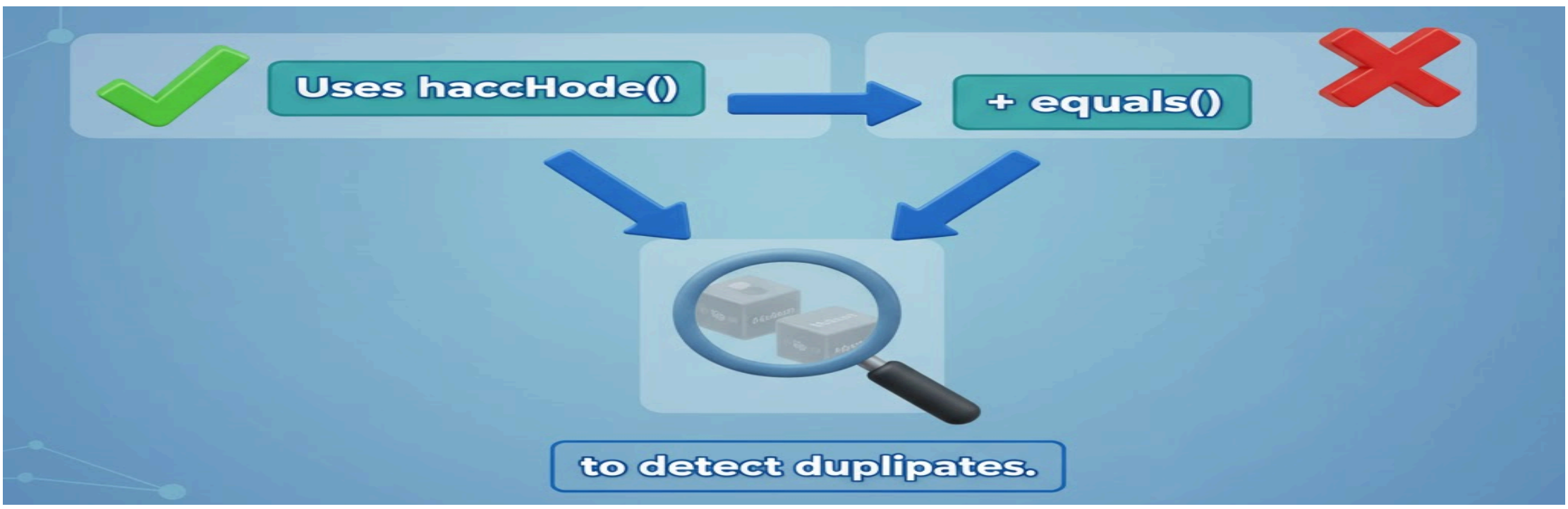
```
set.add("Mohan");
```

During the 2nd addition:

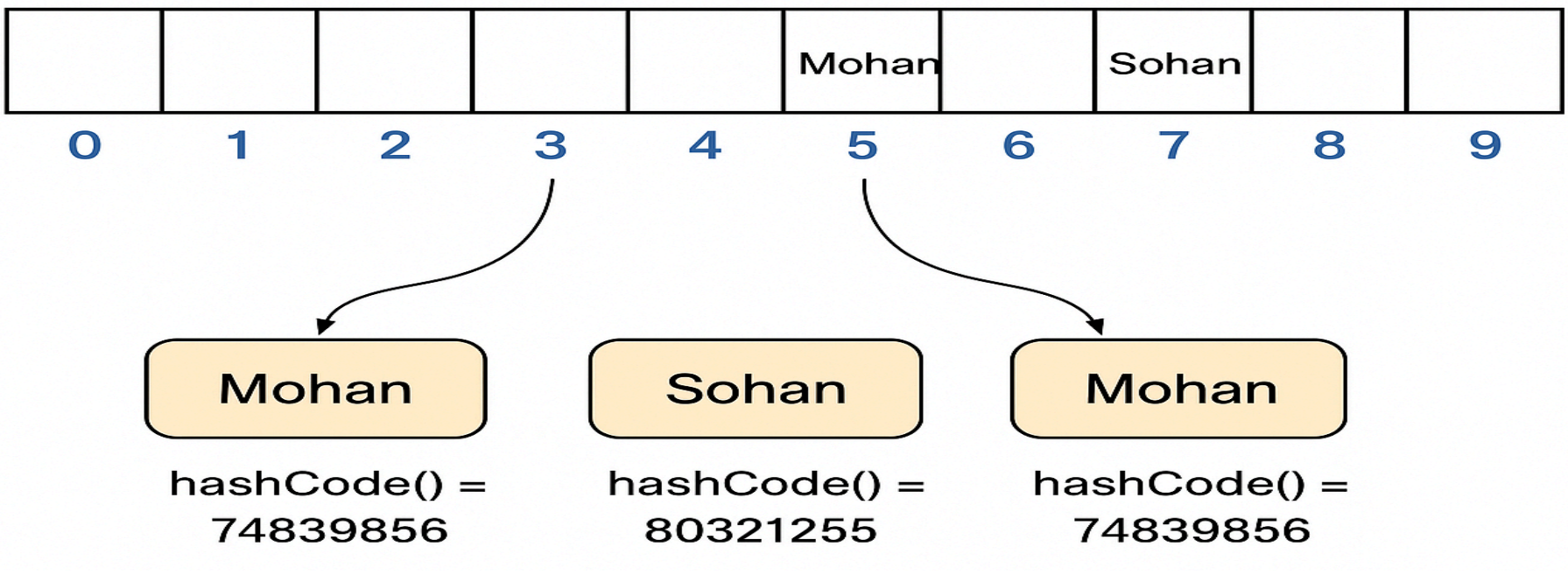
- `hashCode()` generates the same value
- `equals()` returns **true**
- Set says: **"Same element already exists. I will skip it."**

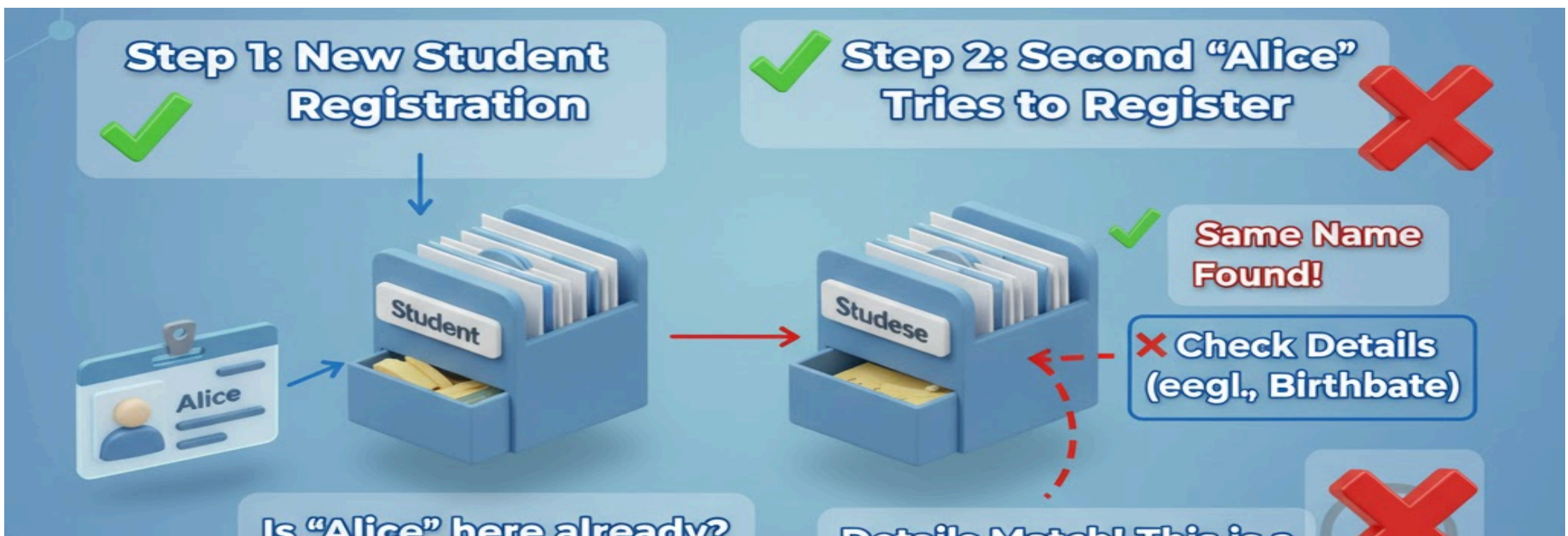


Why Skip duplicates internnaly?



HashSet





⭐ *Final Summary (Very Simple)*

❓ Why doesn't Set allow duplicates?	✔ Because Set's job is to store only unique items.
❓ Why skip duplicates internally?	✔ Uses `hashCode()` + `equals()` to detect duplicates.
❓ Why is this useful?	✔ Faster searching, no repeated data, memory efficient.
❓ Real-life example?	✔ Attendance list: same student cannot be written twice.

Copy code

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

        String a = "Mohan";
        String b = "Sohan";
        String c = "Mohan"; // same as a → duplicate

        System.out.println("hashCode of a: " + a.hashCode());
        System.out.println("hashCode of b: " + b.hashCode());
        System.out.println("hashCode of c: " + c.hashCode()); // same as a

        HashSet<String> set = new HashSet<>();

        set.add(a);
        set.add(b);
        set.add(c);  // HashSet checks hashCode & equals → duplicate → NOT added

        System.out.println("Final Set: " + set);
```

Internal working

Depends on the type of Set:

- **HashSet:** uses **hashing** (hash table)
- **LinkedHashSet:** uses hash table + linked list (keeps order)
- **TreeSet:** uses **Red-Black Tree** (sorted order)

All make sure **no duplicate hash/keys** are stored.

Differences (Set vs List)

- **Set:** no duplicates, no index
- **List:** allows duplicates, has index

When to use (real-life example)

- Storing **unique usernames**
- Storing **unique phone numbers**
- Checking if something already exists

Where to use in real projects

- Login systems (unique emails)
- Token/ID storage

🧠 Common Set Implementations:

Set Type	Ordering	Duplicate Allowed	Null Allowed	Backed By
`HashSet`	No order	❌	✅ (1)	Hash Table
`LinkedHashSet`	Insertion order	❌	✅ (1)	Linked Hash Table
`TreeSet`	Sorted (ascending)	❌	❌ (no null)	Red-Black Tree

What is HashSet?

- **HashSet** is a class that implements the **Set interface**.
- It **stores unique elements only** (no duplicates allowed).
- Belongs to the **java.util** package.
- Based on a **HashTable** (uses hashCode internally to store elements).

Non-technical example

Imagine a **school attendance register**.
Even if a student says their name twice, the teacher marks them **present only once**. Also, the teacher **doesn't care about order**, only uniqueness.

🔥 *MAIN RULE*

1. When you add an object → HashSet calls **hashCode()**
2. HashSet finds a **bucket** using hashCode
3. Inside that bucket it checks with **equals()**
4. If equals() returns true → **Duplicate → Not added**
5. If equals() returns false → **New element → Add**

🧠 Key Features:

- **No Duplicate Elements:** Automatically ignores any duplicate entries.

- **No Guarantee of Order:** Elements are stored in **random/unordered** manner (not in the order you added them).

- **Allows One Null Value:** You can store a single null in HashSet.

- **Not Synchronized (Not Thread-Safe):** Use Collections.synchronizedSet() to make it thread-safe.

- **Efficient Performance:**
  Operations like add(), remove(), contains() are fast (constant time on average).

```
public class Main {
    public static void main(String[] args) {
        HashSet<String> languages = new HashSet<>();

        languages.add("Java");
        languages.add("Python");
        languages.add("C++");
        languages.add("Java"); // Duplicate

        System.out.println(languages); // Output might be: [Java, C++, Python]
```

Performance (Time Complexity)

- add → O(1)
- remove → O(1)
- contains → O(1)

Differences (HashSet vs LinkedHashSet vs TreeSet)

- **HashSet:** fast, no order
- **LinkedHashSet:** maintains insertion order
- **TreeSet:** sorted order, slower

When to use (real-life example)

- You want **unique phone numbers**
- You want **unique emails**
- Login systems (unique usernames)

🔍 *Interview Summary:*

HashSet is a part of Java's Collection Framework that implements the Set interface. It **stores unique elements**, provides **fast performance**, and **does not maintain order**. Internally, it uses a **HashMap** to manage its elements.

What is a linkedHashSet?

- **Maintains insertion order** (elements are stored in the you added them).
- No duplicates.
- It is part of `Java.util` package.
- **Maintains insertion order** (unlike HashSet).
- **Stores only unique elements** (no duplicates).
- **HashSet + LinkedList (for order)**.

Non-Technical Example

Imagine you write names on a **visitor list**:

- You write names **in order**
- If someone comes again with the same name → you **don’t write again**
- But you still show the list in the **same order you wrote**

This is LinkedHashSet.

```java
public class Main {
    public static void main(String[] args) {
        LinkedHashSet<String> set = new LinkedHashSet<>();
        set.add("Java");
        set.add("Python");
        set.add("C++");
        set.add("Java"); // Duplicate, won't be added

        System.out.println(set); // Output: [Java, Python, C++]
```

🧠 Key Features:

✅ **No Duplicates:** Only unique elements are allowed.

🕑 **Maintains Insertion Order:** elements are added is preserved.

❌ **Allows One `null` Element:** Only one `null` is permitted.

⚙️ **Not Thread-Safe:** Use Collections.synchronizedSet() for thread safety.

🔗 **Internally Uses LinkedHashMap:** For storing elements with order tracking.

Internal working

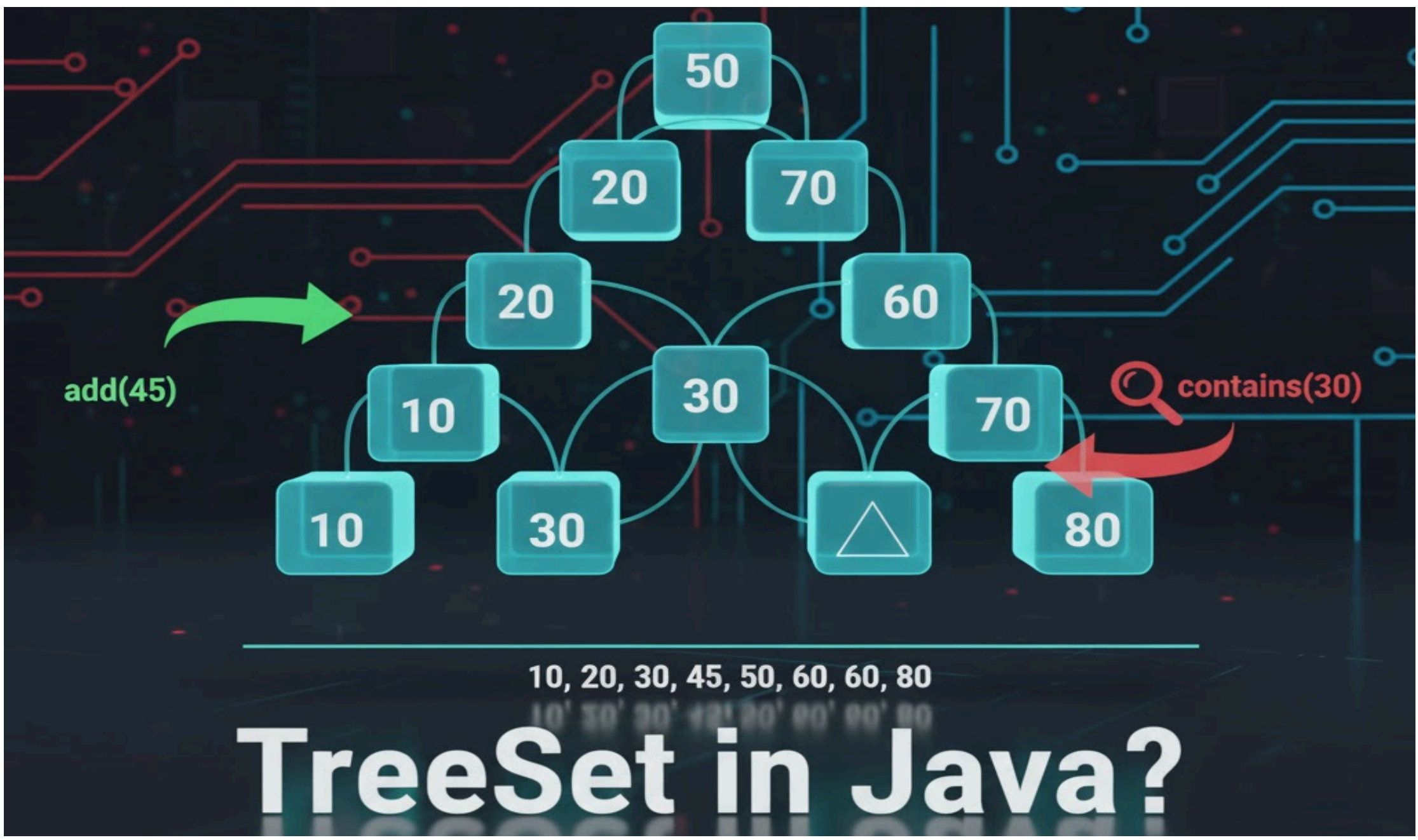
- Uses **HashTable** to store elements
- Uses **Doubly Linked List** to maintain order
- Every element stores:
  - data
  - hash
  - next pointer
  - previous pointer

When you add:

- LinkedList decides order
- Duplicate check using `hashcode()` + `equals()`

Performance

- Add → **O(1)**
- Search → **O(1)**
- Remove → **O(1)**



What is TreeSet in Java?

- **Stores only unique elements** (like HashSet)
- **Automatically sorts elements in natural order (ascending)** 🔼
- It is based on a **Red-Black Tree (Self-balancing BST)** 🌲
- **Stores only unique elements** (no duplicates).
- **Automatically sorts elements** in natural order (Comparable) or custom order (Comparator).

🧠 *Key Features:*

- ✅ No Duplicates Allowed
- 🔢 Sorted Order
- 🌲 Backed by Tree (Red-Black Tree):
- ❌ Does NOT Allow null (if set contains other elements)
- ⚙️ Not Thread-Safe

Non-Technical Example (Very Easy)

Imagine you are writing marks on a paper:

If you write values in any order: 50, 10, 70, 30

But then your friend **automatically arranges them**: 10, 30, 50, 70

This is what TreeSet does — **unique values + sorted order always**.

```java
public class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();

        numbers.add(30);
        numbers.add(10);
        numbers.add(20);
        numbers.add(10); // Duplicate, ignored

        System.out.println(numbers); // Output: [10, 20, 30]
```

Purpose (Why TreeSet exists?)

- **unique values**
- **sorted order**
- **fast searching**

Automatically sorted + no duplicates = TreeSet's job.

Key Features

- Stores **unique** values
- Maintains **sorted order** (default: Ascending)
- Uses a **Balanced Binary Search Tree**
- No null allowed (throws exception)

Important Methods

- **remove**(x)
- **contains**(x)
- **first**() → smallest
- **last**() → largest
- **higher**(x) → next bigger
- **lower**(x) → next smaller
- **ceiling**(x) → ≥ x
- **floor**(x) → ≤ x

---

ceiling(x) — *"Equal or next bigger"*

Imagine you want **at least** a certain mark.

Example: `ceiling(25)`

> "Give me a student who scored **25 or more**.
> If no one has exactly 25, give the **next bigger mark**."

Marks 25 or more → 30, 40, 50
Smallest of these → **30**

ceiling(25) = 30

Example: `ceiling(20)`
20 is present → **20**

---

floor(x) — *"Equal or next smaller"*

Example: `floor(28)`

You say:

> "Give me a student who scored **28 or less**.
> If no one has exactly 28, give the **next smaller mark**."

Marks ≤ 28 → 10, 20
Biggest among them → **20**

---

floor(28) = 20

Example: `floor(40)`
40 is present → **40**

---

higher(x) — *"Strictly bigger"*

Example: `higher(30)`

You say:

> "Give me a student who scored **more than 30**.
> Equal is not allowed."

Marks > 30 → 40, 50
First one → **40**

higher(30) = 40

---

lower(x) — *"Strictly smaller"*

Example: `lower(30)`

> "Give me a student who scored **less than 30**."

Marks < 30 → 10, 20
Highest → **20**

👉 **lower(30) = 20**

pollFirst() — *"Take and remove the smallest number"*

Before:

```
[10, 20, 30, 40, 50]
```

"Give me the student with **lowest marks**, and remove him."

Lowest → **10**

After removal:

```
[20, 30, 40, 50]
```

pollFirst() = **10**

---

14. pollLast() — *"Take and remove the biggest number"*

Before:

```
[10, 20, 30, 40, 50]
```

"Give me the student with **highest marks**, and remove him."

Highest → **50**

After removal:

```
[10, 20, 30, 40]
```

pollLast() = **50**

🌳 *TreeSet Rules*

TreeSet internally uses a **Balanced Binary Search Tree (Red-Black Tree)**.

- **TreeSet Rules**

1. **First element = ROOT**
2. **Smaller → LEFT** if (newValue < currentNodeValue) → ***go left***
3. **Larger → RIGHT** if (newValue > currentNodeValue): → ***go Right***
4. **Equal → IGNORE (no duplicates)**

🎯 *Full Example — Insert in order: 50, 30, 70, 20, 40, 60, 80*

Step-by-step:

1) Insert 50

→ ***root***

```
50
```

2) Insert 30

30 < 50 → left

3) Insert 70

70 > 50 → right

4) Insert 20

20 < 50 → go left

20 < 30 → go left

5) Insert 40

40 < 50 → go left

40 > 30 → go right

6) Insert 60

60 > 50 → go right

60 < 70 → go left

7) Insert 80

80 > 50 → go right

80 > 70 → go right

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

Performance (Time Complexity)

- **add** → O(log n)
- **remove** → O(log n)
- **traversal** → O(n)

Not as fast as **HashSet** (O(1)), but sorted.

Differences (TreeSet vs Others)

- **HashSet** → No order
- **LinkedHashSet** → Insertion order
- **TreeSet** → Sorted order
- **ArrayList** → Allows duplicates + order
- **TreeSet** → Unique + sorted

When to Use in Real Life

- Sorting names automatically
- Sorted list of IDs
- Sorted product prices

Where to Use in a Real Project

- Search suggestions feature
- Autocomplete (sorted dictionary words)
- Sorted leaderboard

---

treeSet.first(): Returns the smallest element.

```
int firstElement = set.first();
```

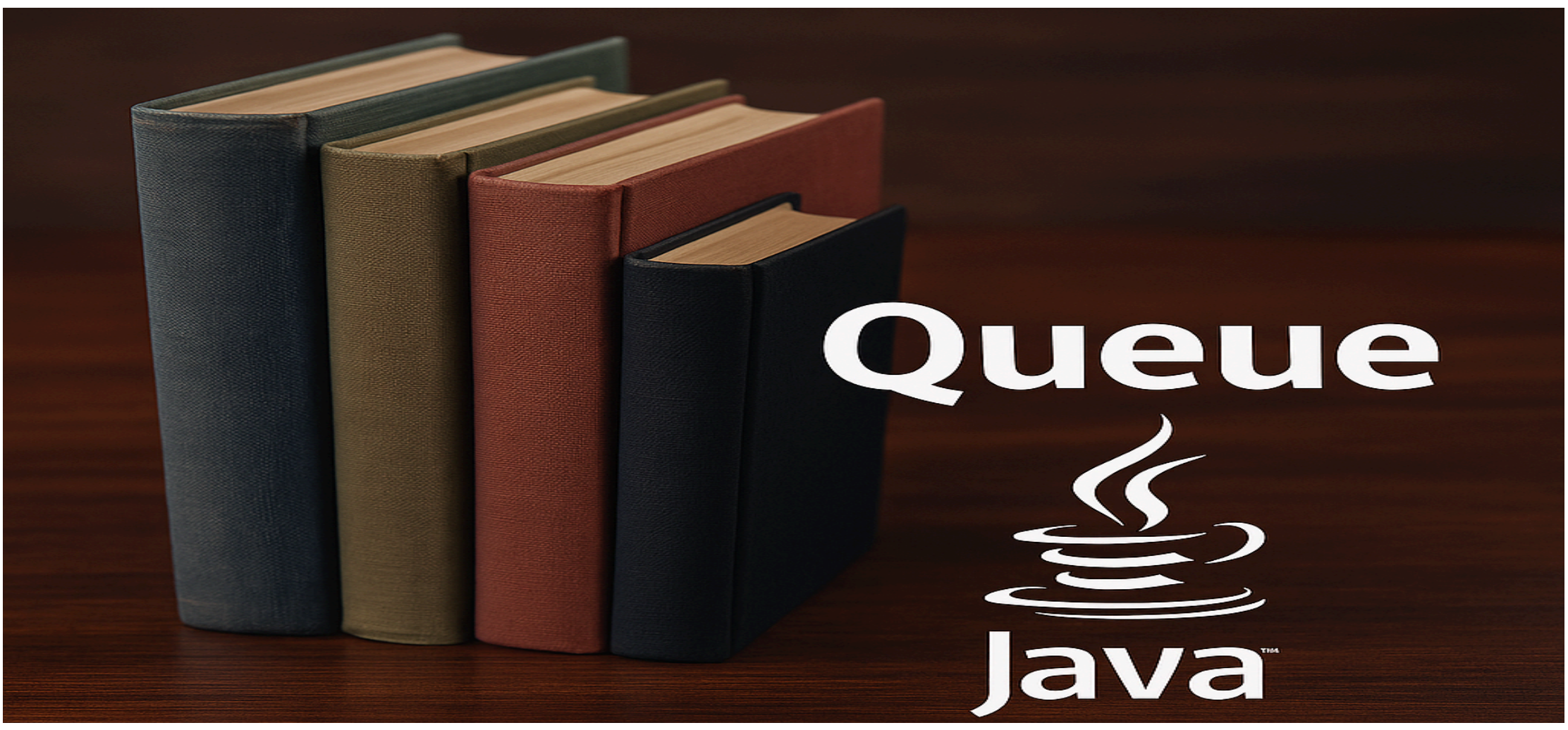
treeSet.last(): Returns the largest element.

```
int lastElement = set.last();
```

treeSet.floor(E element): Returns the largest element ≤ the given element.If exists in the set, it returns . **Returns null** if no such element exists.

```
int floorValue = set.floor(15);  // Returns next equal or lower value
```

Method	Description
`add(E e)`	Adds an element while maintaining sorting
`addAll(Collection c)`	Adds all elements from another collection
`contains(Object o)`	Checks if an element exists
`remove(Object o)`	Removes an element
`size()`	Returns the number of elements
`isEmpty()`	Checks if the set is empty
`clear()`	Removes all elements
`first()`	Returns the smallest element
`last()`	Returns the largest element
`pollFirst()`	Removes and returns the smallest element
`pollLast()`	Removes and returns the largest element
`higher(E e)`	Returns the next greater element



What is a Queue?

- A **Queue** is a **linear data structure** used to **store elements in order**.
- Follow **FIFO**: **First In, First Out** – the element added first is removed first.
- It is part of the **java.util package**.
- **Extends the Collection interface**.
- Follows **FIFO (First In, First Out)** principle.

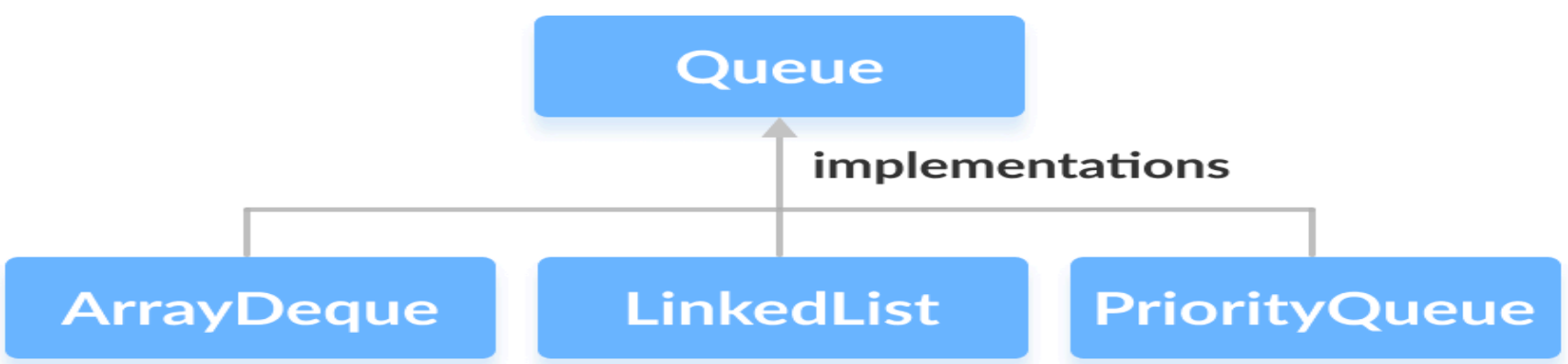
Non-technical Example

Imagine people standing in a **ticket queue**:

- The **first person** who stands in line will get the ticket **first**.
- The **last person** waits until everyone in front is done.

The same thing happens in Java Queue.

Common implementations:



🧠 *Key Features:*

FIFO Order: The first element inserted is the first to be removed.

⚙️ Operations:

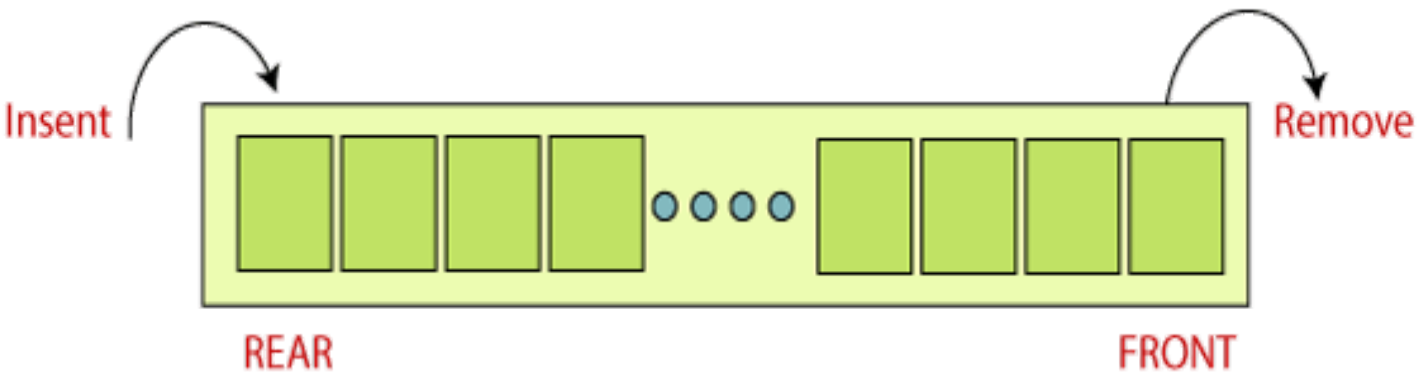
- **add()** / **offer()** – insert element
- **remove()** / **poll()** – remove and return the head
- **element()** / **peek()** – retrieve the head without removing

---

⚠️ Difference between offer() and add():

offer() returns false if the element can't be added, while add() throws an exception.

📌 **Null elements** are **not allowed** in most queue implementations.




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

public class Main {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        queue.offer("Java");
        queue.offer("Python");
        queue.offer("C++");

        System.out.println(queue);         // Output: [Java, Python, C++]
        System.out.println(queue.poll()); // Removes "Java"
        System.out.println(queue.peek()); // Returns "Python"
    }
```

🔥 *How LinkedList Implements Queue*

LinkedList is a **doubly linked list** internally. Each element has:

previous ↔ data ↔ next

So Queue uses:

- **Head node** = front
- **Tail node** = rear

✔ Adding at tail → O(1)
✔ Removing at head → O(1)

That's why LinkedList is good for Queue.

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.offer("A");
        queue.offer("B");
        queue.offer("C");
        System.out.println(queue);
        // [A, B, C]
        System.out.println(queue.peek());
    // A (front element)
        System.out.println(queue.poll());
      // A removed
        System.out.println(queue);
        // [B, C]
    }
}
```

When to Use (Real-time use case)

- You want to process things **in order**
- You want the **oldest** element first
- Example:
  - customer service system
  - message processing

Important Methods (Very simple wording)

- `add()` → Add at end (throws error if full)
- `offer()` = safe add → returns `false` if fails.
- `remove()` deletes front element but **throws error** if queue is empty.
- `poll()` = safe remove → returns `null` when empty.
- `peek()` shows front element safely (returns `null` if empty).
- `isEmpty()` → checks if the queue is empty.

Performance — Big O

- Add → **O(1)**
- Remove → **O(1)**
- Peek → **O(1)**

Queue	Stack
FIFO	LIFO
remove first	remove last
used for tasks	used for undo operations

What is a PriorityQueue in Java?

A PriorityQueue is a special type of queue that gives priority to the **smallest element first** (by default). It does **not** work on FIFO. It works on **priority order**.

Non-Technical Example

Imagine students standing in a line for medicine.
Who gets medicine first?
The student with **highest fever** (highest priority), not the one who came first.

Same in PriorityQueue — The element with **highest priority** goes first.

🧠 Key Features:

- ➢ ❌ Does not allow `null` elements
- ➢ ❌ Not thread-safe
- ➢ ✅ Duplicates are allowed
- ➢ 🧱 **Min-Heap structure** is used internally (by default, smallest element has highest priority)

```java
public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        pq.add(30);
        pq.add(10);
        pq.add(20);

        System.out.println(pq);         // Output: [10, 30, 20] (order not guar
        System.out.println(pq.poll()); // Output: 10 (smallest first)
        System.out.println(pq.peek()); // Output: 20 (next smallest)
```

Key Features – Main characteristics

- Automatically arranges elements by priority.
- Smallest element becomes the **head** (default).

Important Methods – Common functions

- `add()` → insert element
- `offer()` → safe insert
- `peek()` → get highest priority without removing
- `poll()` → remove highest priority
- `size()` → number of elements

⭐ *WHY DOES PRIORITYQUEUE ALWAYS WORK?*

✔ **Small value climbs UP**

✔ **Big value falls DOWN**

These two simple moves keep the smallest element at the **top (index 0)** all the time.

Bubble-up = child goes UP until it becomes bigger than parent.

Bubble-down = parent goes DOWN until children are bigger.

🔥 *Java stores elements in an Array + Heap Structure*

Example:

You add numbers in this order:

```
add(30)
add(10)
add(20)
add(5)
```

Inside queue, steps happen:

📌 **STEP 1 — Add 30 →** Heap: → No problem. Only one element.

```
30
```

📌 **STEP 2 — Add 10 →** Heap tries to keep the **smallest at the top**, so:→ ***Temporary***:

```
  30
 /
10
```

Now heap checks:→ Is 10 < 30 ?→ YES → Swap!→ Final:

```
 10
 /
30
```

📌 **STEP 3 — Add 20 →** Temporary:

```
   10
  /  \
30   20
```

Check: Is 20 < 10? → NO → No swap

📌 **STEP 4 — Add 5** → Temporary:

Heap check: Is 5 < 30? YES → Swap→ Now:

Check again: Is 5 < 10? YES → Swap → Final heap:

🎉 Now **5 is the smallest**, so it stays on top.

FINAL HEAP INTERNAL ARRAY REPRESENTATION

PriorityQueue internally stores this Heap in **array form**:

```
Index: 0   1   2   3

Value: 5  10  20  30
```

⭐ *Summary*

1. **Add()** → put at end → small values move UP
2. **Remove()** → take last → big values move DOWN

PriorityQueue Works Inside

Rule 1 Smallest element at index 0

Rule 2 bubble-up

Rule 3 bubble-down

Real-Life Example: Kids Standing by Height (Smallest in Front)

Smallest height must always stand at the front.

This is exactly how a **Min-Heap / PriorityQueue** works.

⭐ *Bubble-Up (Adding a New Kid)*

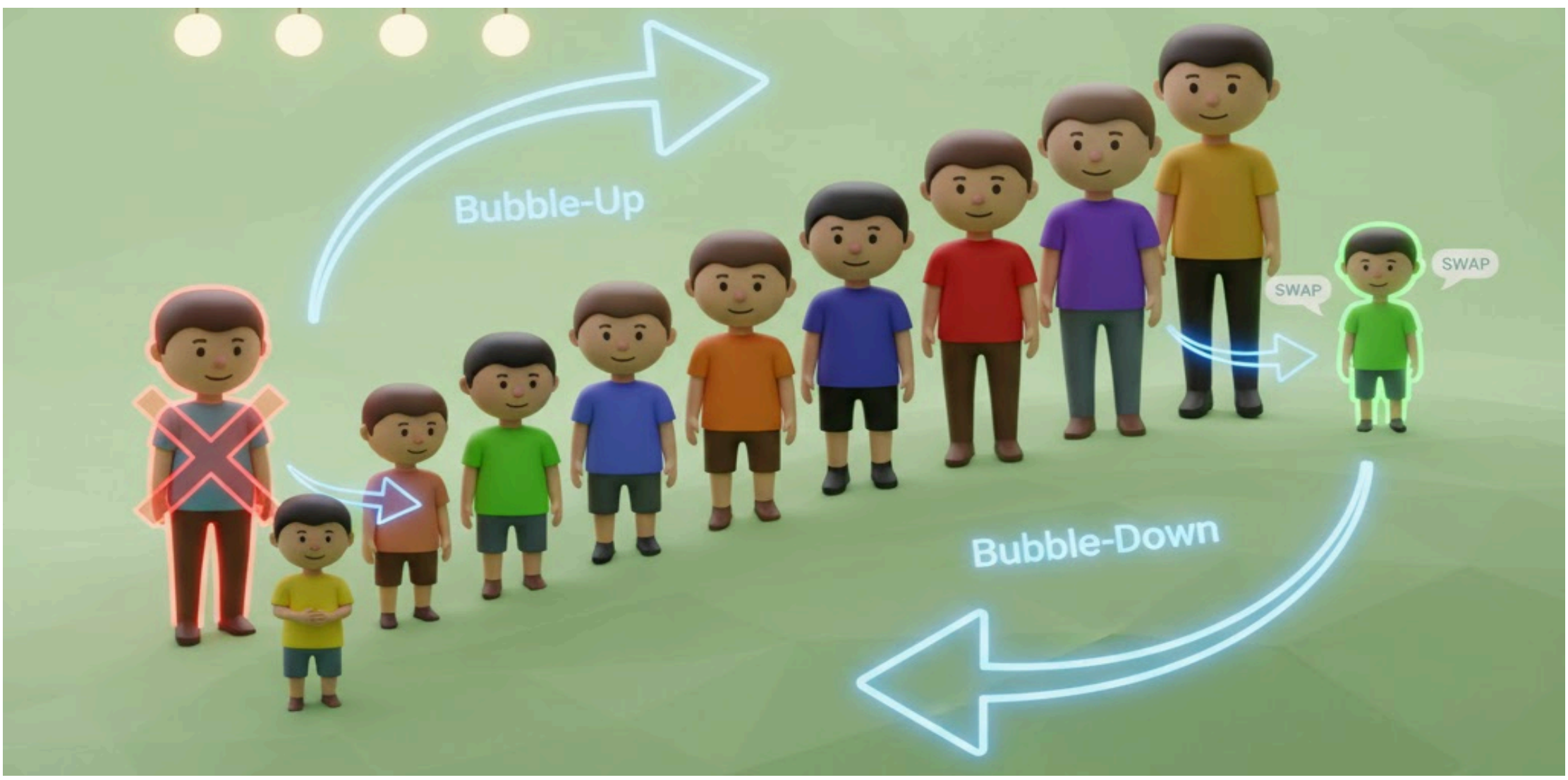
1. A new kid joins at the **end of the line**.
2. If the new kid is **shorter** than the kid before him → swap.
3. Keep swapping until he reaches the correct position.

➡ *Smallest kid moves toward the front.*

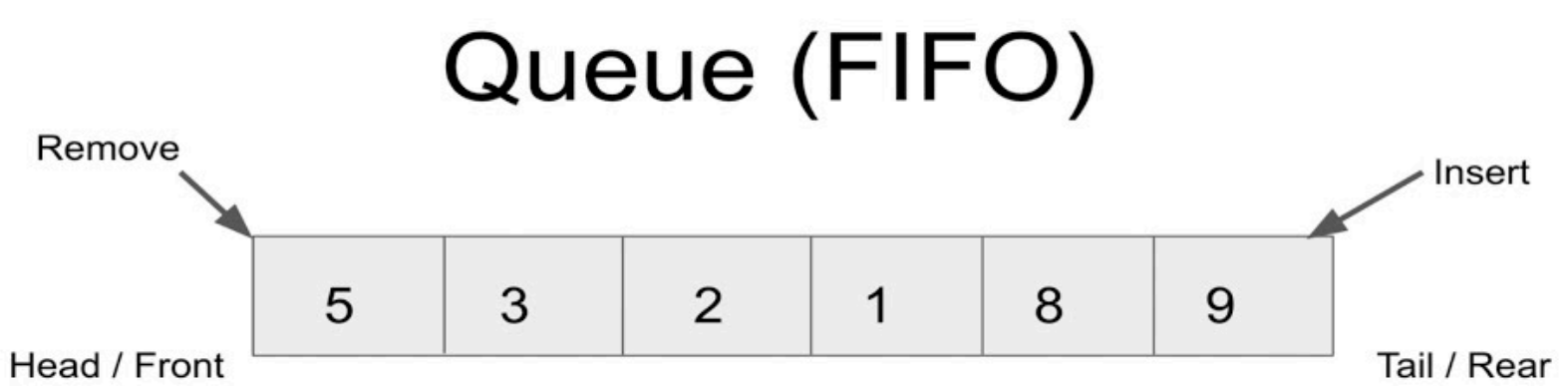
⭐ *Bubble-Down (Removing the Front Kid)*

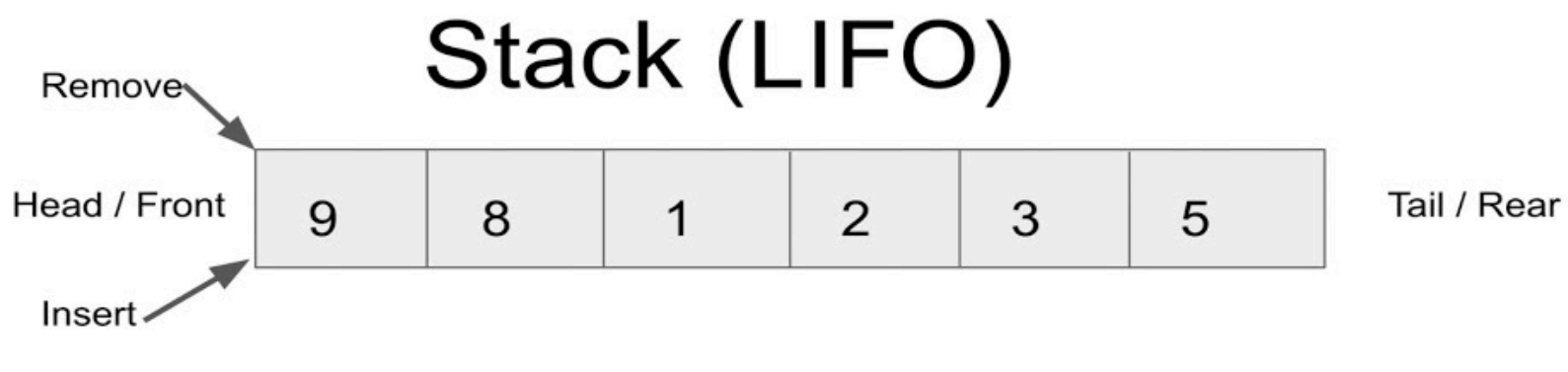
1. The shortest kid at front goes away.
2. The **last kid** in the line comes to the front.
3. If he is **taller** than the kids behind him → swap with the shorter one.
4. Keep swapping until he reaches the correct position.

➡ *Line becomes correct again — the shortest kid reaches front.*



Java Deque





What is Deque in Java?

Deque stands for **Double Ended Queue**. It is a **linear data structure** that allows **insertion and deletion from both ends — front and rear**. It is pronounced as "**deck**".

- ➢ Deque stands for **Double-Ended Queue**
- ➢ It is an **interface** that extends the **Queue** interface

---

Non-Technical Example

Think of a **line of people** where:

- You can enter from the **front**
- You can enter from the **back**
- You can exit from **both sides**

🧠 *Key Features:*

✅ Allows insertion and deletion at both **front** and **rear**

✅ Can act as both a **Stack (LIFO)** and a **Queue (FIFO)**

✅ Can be implemented using **ArrayDeque** or **LinkedList**

Commonly Used Methods:

- addFirst(element) – Add at the front
- addLast(element) – Add at the rear
- removeFirst() – Remove from front
- removeLast() – Remove from rear
- peekFirst() – Retrieve front element
- peekLast() – Retrieve rear element

🧠 *Key Features:*

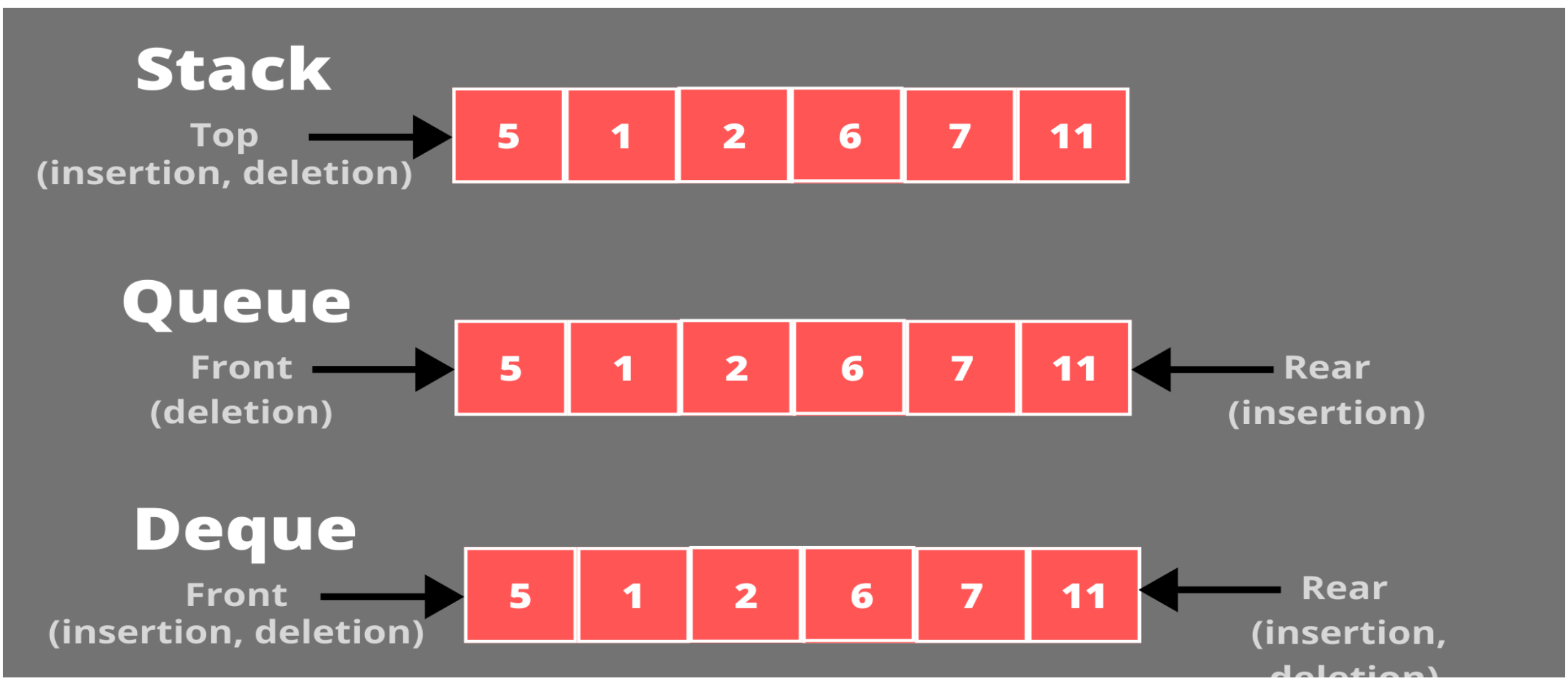
✅ Allows insertion and deletion at both **front** and **rear**

✅ Can act as both a **Stack (LIFO)** and a **Queue (FIFO)**

✅ Can be implemented using **ArrayDeque** or **LinkedList**

🛠️ *Common Implementations:*

- ArrayDeque (most commonly used)
- LinkedList (also implements Deque)




```java
public class Main {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();

        deque.addFirst("Java");
        deque.addLast("Python");
        deque.addFirst("C++");

        System.out.println(deque); // Output: [C++, Java, Python]

        System.out.println(deque.removeLast()); // Output: Python
        System.out.println(deque.peekFirst());  // Output: C++
```

Internal Working – How it works inside

Java uses a **Doubly Linked List** internally. Each node has:

- data
- reference to the next node
- reference to the previous node
- So moving front/back is fast.

Performance – Big-O

- Add/remove at front → **O(1)**
- Add/remove at back → **O(1)**
- Searching → **O(n)**

Differences (vs Queue & Stack)

- Queue → only front remove, back add
- Stack → only top push/pop
- Deque → both sides work

🗺️ *What is a Map in Java?*

In Java, **Map** is **not part of the Collection interface** but is included in the **Collections Framework**. It is used to store **key-value pairs**, where each key is unique, and each key maps to **exactly one value**.

- ➢ Located in the **java.util** package
- ➢ Stores data as **(key, value) pairs**
- ➢ **Keys must be unique**, values can be **duplicates**

🧱 *Common Implementations:*

- **HashMap** – Unordered, fastest, allows one null key
- **LinkedHashMap** – Maintains insertion order
- **TreeMap** – Sorted by key (natural or custom order)
- **Hashtable** – Legacy, synchronized

What it is

A **Map** is a collection in Java that stores data in **key–value pairs**.
Each **key is unique**, and every key points to **one value**.

Non-Technical Example

Think of a **dictionary**:

- A word = **key**
- Meaning = **value**

No two words can be the same → keys must be unique.

Purpose – Why we need it

We use a Map when we want to **find something quickly using a key**,
like searching by ID, name, or code.

Key Features – Main characteristics

- Stores data as: **key → value**
- Keys cannot repeat
- Values can repeat
- Fast lookup using keys
- Allows null keys/values (depends on Map type)

When to Use – Real-time use case

- Store **studentId → studentName**
- Store **username → password**
- Store **countryCode → countryName**
- Store **productId → productDetails**

Where to Use in a Project

- Login systems (map username → hashed password)
- Shopping cart (map productId → quantity)
- Attendance systems (map rollNo → status)

Internal Working – How it works inside

Depends on Map type:

HashMap

- Uses **hashing**
- Stores entries in **buckets**
- Uses hashCode() + equals()

LinkedHashMap

- Same as HashMap but maintains **insertion order**

TreeMap

- Uses **Red-Black Tree**
- Stores keys in **sorted order**

Performance – Big-O

(HashMap average case)

- Insert → **O(1)**
- Search → **O(1)**
- Delete → **O(1)**

(TreeMap is **O(log n)** because tree)

Differences – Compare with Similar Collection

Map vs List:

- Map → key-value
- List → index-based

Map vs Set:

- Set → only values, no keys
- Map → key + value

Example Code – Simple Java Example

```
public class Demo {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();

        map.put(101, "Mohan");
        map.put(102, "Rohit");
        map.put(103, "Amit");

        System.out.println(map.get(102));  // Rohit
        System.out.println(map);           // {101=Mohan, 102=Rohit, 103=Amit}
```

📘 Common Implementations of Map:

Map Type	Ordering	Null Key	Thread-safe	Backed By
`HashMap`	No order	✅ One null key	❌ No	Hash Table
`LinkedHashMap`	Insertion order	✅ One null key	❌ No	Hash Table + Linked List
`TreeMap`	Sorted by keys	❌ (null not allowed)	❌ No	Red-Black Tree
`Hashtable`	No order	❌ Not allowed	✅ Yes	Hash Table

🔧 *Common Methods:*

- **put(key, value)** – Adds a key-value pair
- **get(key)** – Retrieves the value for a given key
- **remove(key)** – Removes the entry by key
- **containsKey(key)** – Checks existence
- **containsValue(value)** – Checks existence
- **keySet()** – Returns a set of all keys
- **values()** – Returns a collection of all values
- **entrySet()** – Returns a set of all (key, value) pairs

---

put(K key, V value):

Inserts a key-value pair into the map. If the key already exists, it updates the value.

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Apple");
map.put(2, "Banana");
map.put(1, "Orange");  // Updates key 1 to "Orange"
```

---

get(Object key): **Returns the value associated with the given key.** If the key does **not exist**, it returns null.

```
String value = map.get(1);  // Returns "Orange"
String notFound = map.get(3);  // Returns null
```

---

containsKey(Object key)

Check if the given key exists in the map. Returns true if the key is present.

```
boolean hasKey = map.containsKey(2);   // true
boolean hasKey2 = map.containsKey(5); // false
```

---

containsValue(Object value)

Checks if the given value exists in the map.Returns true if the value is present.

```
boolean hasValue = map.containsValue("Banana"); // true
boolean hasValue2 = map.containsValue("Grapes"); // false
```

---

values(): Returns a Collection of all values in the map.

```
 Collection<String> values = map.values();
```

---

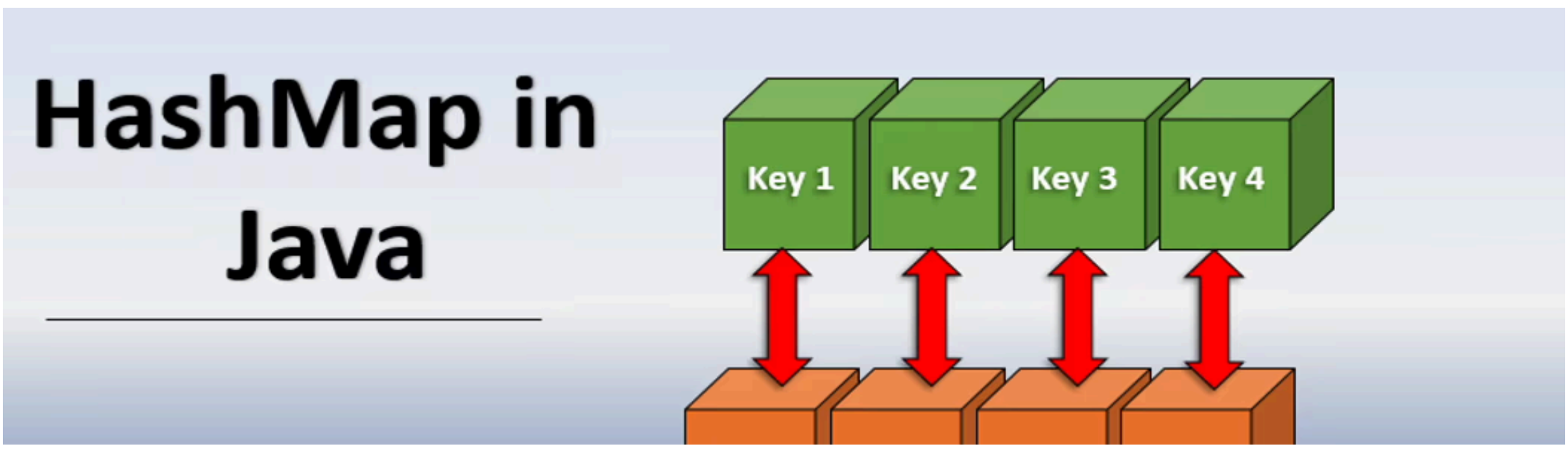
entrySet(): Returns a Set of all key-value pairs (Map.Entry).

```
Set<Map.Entry<Integer, String>> entries = map.entrySet();
```

---

replace(K key, V value): Replaces the value for the specified key if it exists.

```
map.replace(1, "Mango");  // Changes key 1's value to "Mango"
```



What is a HashMap in Java?

HashMap is a class in the **Java Collection Framework** used to store **key-value pairs**,

- **Keys must be unique**
- **Values can be duplicated**
- Belongs to **java.util** package
- Uses a **hashing technique** internally
- **Does not maintain any order** (neither insertion nor sorted)
- **Allows one null key** and **multiple null values**

What it is

A **HashMap** is a Java Map that stores data in **key → value pairs** using **hashing** for very fast search.

Non-Technical Example

Think of **school lockers**:

- Locker number = **key**
- Student's bag = **value**

You find the bag using the locker number instantly → this is how HashMap works.

Purpose – Why we need it

To store and retrieve data **quickly** using a key (ID, code, name).
HashMap gives **near O(1)** performance.

Key Features – Main characteristics

- Stores **unique keys**
- One null key allowed
- Many null values allowed
- Extremely fast (because of hashing)
- No order maintained (random order)

When to Use – Real-time use case

- StudentID → StudentName
- ProductID → ProductInfo
- Username → Password
- CountryCode → CountryName

Where to Use in a Project

- Login module
- E-commerce product data
- API response caching

Important Methods – Common functions

- put(key, value)
- get(key)
- remove(key)
- containsKey(key)
- containsValue(value)
- keySet()
- entrySet()

Internal Working – How it works inside

Step 1: You do put(key, value)

Example:

```
map.put(101, "Mohan");
```

Java calculates:

```
 hash = key.hashCode();
```

Step 2: Hash converted to bucket index

index = hash % capacity

Example:
Capacity = 16
101 % 16 = 5
So entry goes to **bucket 5**.

Step 3: Node stored in bucket

HashMap stores:
(key, value, hash, next)

Step 4: Collision handling

If two keys go to the same bucket:

- If keys **equal** → **update value**
- Else → add to **linked list** inside same bucket

Performance – Big-O

Average:

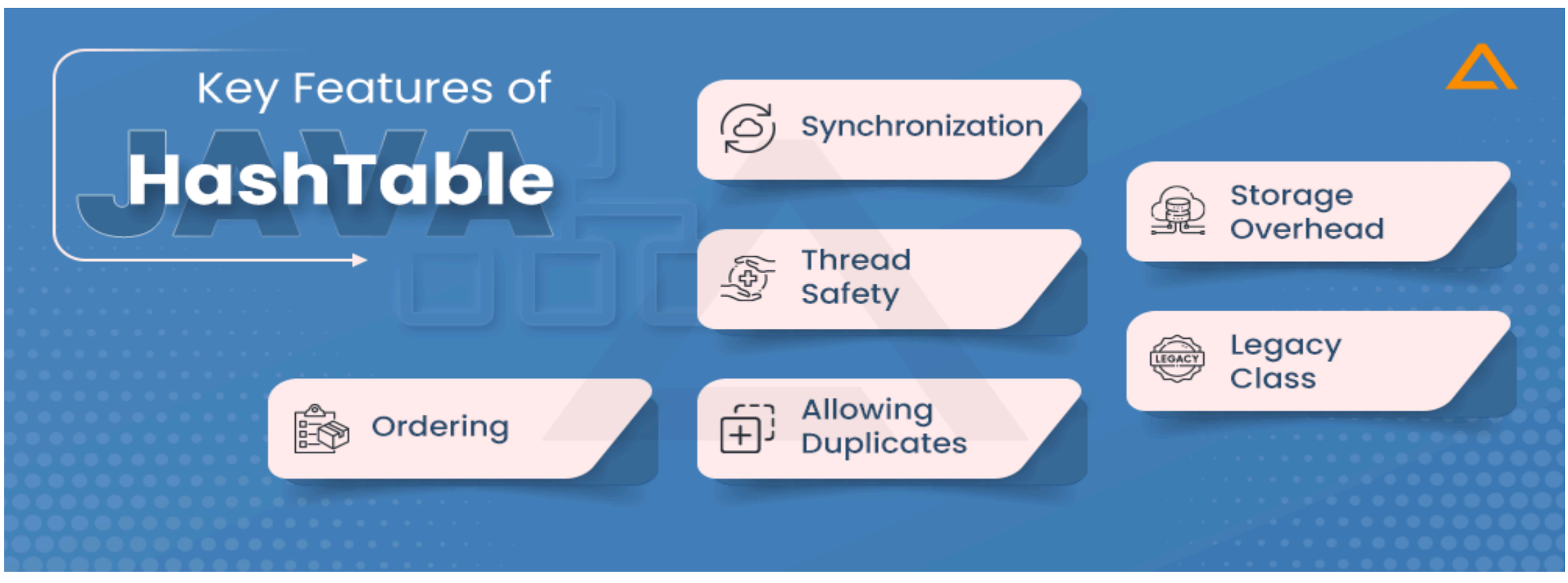
- Insert → **O(1)**
- Search → **O(1)**
- Delete → **O(1)**

Worst case (tree) → **O(log n)**

Differences – Compare with similar collections

- **HashMap vs LinkedHashMap** → No order vs Insertion order
- **HashMap vs TreeMap** → No sort vs Sorted by key
- **HashMap vs Hashtable** → Not synchronized vs synchronized

---



✅ What is a Hashtable in Java?

- Hashtable is a **Map** implementation in Java.
- It stores **key-value** pairs just like a HashMap.
- 🔒 **Hashtable is synchronized**, meaning it is **thread-safe**.
- **Null keys and null values are NOT allowed**.
- It is a **legacy class**, introduced in earlier versions of ***Java (before Java 2).***

✅ *Important Rules*

❌ Does not allow null keys or null values

This will throw a NullPointerException:

```
table.put(null, "test"); // ❌ Not allowed
table.put(4, null);      // ❌ Not allowed
```

Internal Working

Same as HashMap (buckets + hashing)
BUT every operation uses **synchronized lock**.

✅ Important Methods

- ➢ put(K key, V value) – add or update entry
- ➢ get(K key) – get value
- ➢ remove(K key) – remove entry
- ➢ containsKey(K key): Checks whether the specified key exists.
- ➢ containsValue(V value) Checks whether the specified value exists.
- ➢ keySet(), values(), entrySet()

⭐ 4. Hashtable vs HashMap (Very Easy Table)

Feature	HashMap	Hashtable
Null key/value	✔ Allowed	✘ Not allowed
Thread-safe	✘ No	✔ Yes
Speed	Fast	Slow
Legacy class	✘ No	✔ Yes (old class)
Synchronization	Not synchronized	synchronized

🌲 *What is TreeMap?*

- TreeMap is a part of **Java's Collection Framework**.
- It stores **key-value pairs in sorted order(**(ascending).
- It is implemented **Red-Black Tree (self-balancing binary search tree)**.
- Keys are sorted **in natural order** (e.g., numbers: ascending; strings: alphabetical).
- **No null keys are allowed**, but **multiple null values are allowed**.

TreeMap = Sorted Map

Keys are **always kept in increasing order** automatically.

⭐ *Important Rules*

✔ *Rule 1: Keys must be sorted*

- Natural sorting (1,2,3…)
- For strings (A,B,C…)

✔ *Rule 2: Duplicates keys not allowed*

If same key is added → new value replaces old value.

✔ *Rule 3: Null key NOT allowed*

TreeMap cannot compare null → gives error.

✔ *Rule 4: Allows multiple null values*

Values can be null.

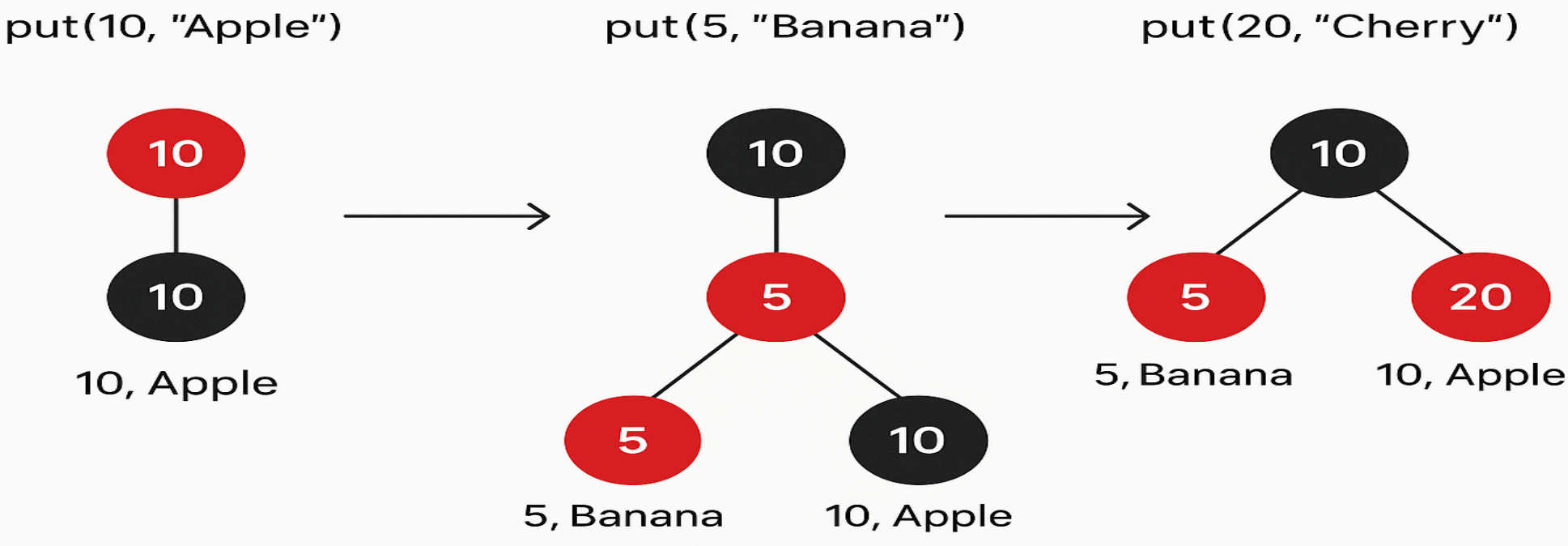
Internal Working (Very Simple)

TreeMap uses **Red-Black Tree**.

✔ ***Automatic sorting***
✔ Fast search
✔ Fast insert
✔ Fast delete

```
public class TreeMapDemo {
    public static void main(String[] args) {
        TreeMap<Integer, String> map = new TreeMap<>();

        map.put(10, "Apple");
        map.put(5, "Banana");
        map.put(20, "Cherry");

        System.out.println(map); // Output: {5=Banana, 10=Apple, 20=Cherry}
```

TreeMap Working in Java



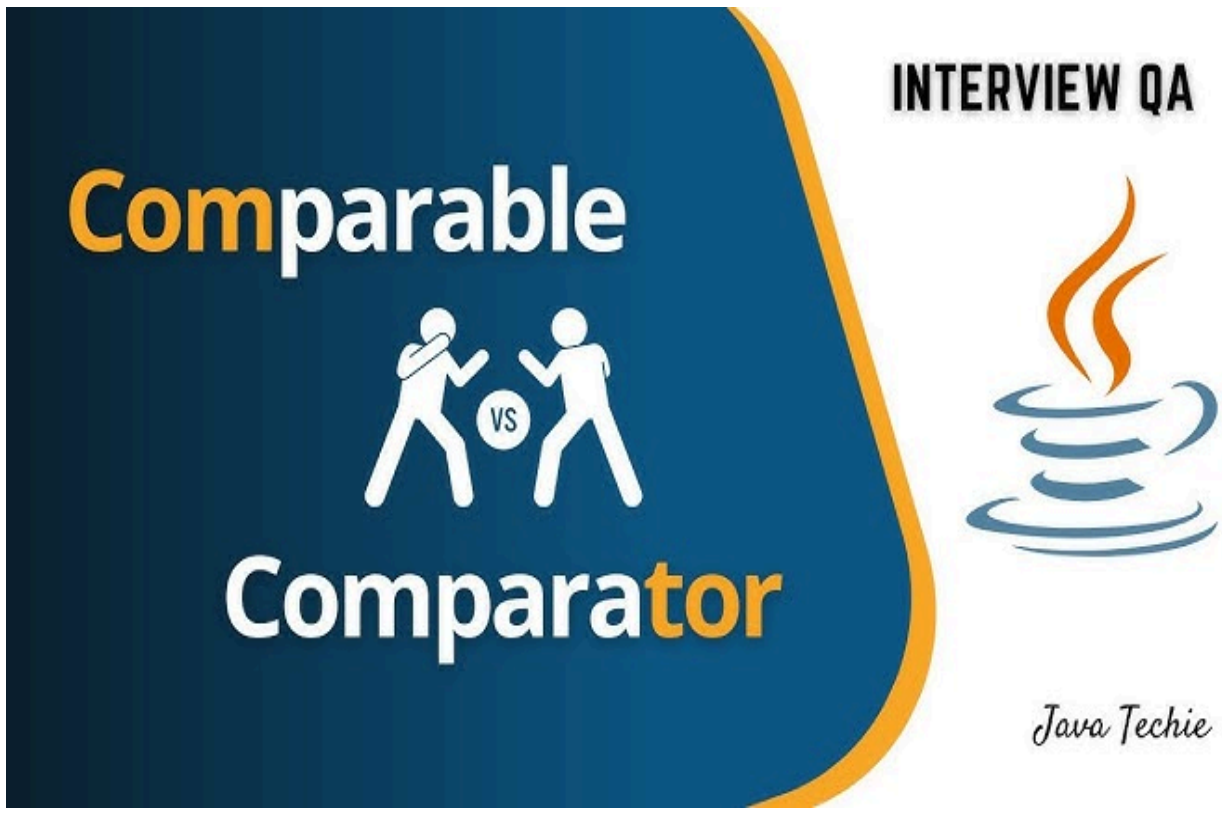
🔍 HashMap vs Hashtable vs LinkedHashMap vs TreeMap:

Feature	HashMap	Hashtable	LinkedHashMap	TreeMap
Thread-safe	❌ No	✅ Yes	❌ No	❌ No
Null key allowed	✅ Yes (1)	❌ No	✅ Yes	❌ No
Ordering	❌ No order	❌ No order	✅ Insertion order	✅ Sorted by key
Performance	✅ Fast	❌ Slower	✅ Good	❌ Slower (O log n)

Use Cases of TreeMap

✔ When you need **sorted data**
✔ When you need **range operations** like:

- **firstKey**() // smallest key
- **lastKey**() // largest key
- **higherKey**() // next higher key
- **lowerKey**()
- **subMap**()



What is Comparable in Java?

- ➢ Comparable is an **interface** in Java.lang package.
- ➢ It is used to **define the natural sorting order** of objects.
- ➢ When a class implements Comparable, it **overrides** the compareTo() method.

Why do we use Comparable?

1. **To sort objects** (like Employee, Student, Product, etc.) based on one specific field (e.g., name, id, age).
2. It tells Java **how to compare two objects** of the same class.
3. It is useful in **collections like TreeSet, TreeMap, and Arrays.sort(), Collections.sort()** where ordering is required.

🔑 Key Points:

- ➢ Comparable is used for **natural sorting**.
- ➢ You **override compareTo()** to define how objects are compared.
- ➢ Used by Collections.sort() and sorted collections like TreeSet, TreeMap.

```
package first;

import java.util.*;

class mohan implements Comparable<mohan> {
        String name;
        mohan(String name) {this.name = name;}

        public int compareTo(mohan s) {
                System.err.println("Comparing " + this.name + " with " + s.name);
                return this.name.compareTo(s.name);
        }
        public String toString() {return name;}
}
public class Main {
        public static void main(String[] args) {
                List<mohan> list = new ArrayList<>();
```

```
                list.add(new mohan("D"));
                list.add(new mohan("C"));
                list.add(new mohan("B"));
                list.add(new mohan("A"));
                Collections.sort(list);
                for (mohan m : list) {
                        System.out.println(m);
        }
}}
```

```
package first;
import java.util.*;
class student implements Comparable<student>{
        String name;
        int id;
        student(int id,String name){this.name = name; this.id = id;}

        public int compareTo(student s){
                System.out.println("Comparing "+ this.id+"  with "+ s.id);
                return this.id  - s.id;}
        public String toString(){      return id +" : "+ name ;}
}
public class Main {
        public static void main(String[] args) {
                List<student> list = new ArrayList<>();
                list.add(new student(3, "Mohan"));
                list.add(new student(2,"Ram"));
                list.add(new student(1,"Rohit"));

                Collections.sort(list);
                for(student obj : list)
                {
                        System.out.println(obj);
                }
    }}
```

Interview Summary:

`Comparable` is used to **sort objects in natural order** using the `compareTo()` method. It is **used when your class defines its own sorting logic**.

Java Comparator Interface

What is a Comparator in Java?

Comparator is an **interface** in Java used to define **custom sorting logic** for **objects**. Unlike Comparable, define **multiple sorting rules** without modifying the class itself. It belongs to the **Java.util** **package**.

```
package first;

import java.util.*;
class Student {
  int id;
  String name;
  Student(int id, String name) { this.id = id; this.name = name; }

 public String toString() {return id + " - " + name;}
}
// Comparator for sorting by name
class NameSort implements Comparator<Student> {
  public int compare(Student s1, Student s2) {
     return s1.name.compareTo(s2.name);
  }
}
public class Main {
  public static void main(String[] args) {
     List<Student> list = new ArrayList<>();
     list.add(new Student(5, "C"));
     list.add(new Student(2, "Z"));
     list.add(new Student(1, "A"));
     Collections.sort(list, new NameSort());
     for (Student s : list) {
        System.out.println(s);
     }
 } }
```

```
import java.util.*;
// ===== Student Class =====
class Student {
  int id;
  String name;
  Student(int id, String name) {
     this.id = id;
     this.name = name;
  }
  public String toString() {
     return "(" + id + ", " + name + ")";
  }
}
// ===== Comparator 1: Sort by ID =====
class SortById implements Comparator<Student> {
  public int compare(Student s1, Student s2) {
     return s1.id - s2.id;  // ascending order
  }
}
// ===== Comparator 2: Sort by Name =====
class SortByName implements Comparator<Student> {
  public int compare(Student s1, Student s2) {
     return s1.name.compareTo(s2.name);  // alphabetical order
  }
```

```
}
// ===== MAIN Program =====
public class Main {
  public static void main(String[] args) {
     List<Student> list = new ArrayList<>();
     list.add(new Student(3, "Mohan"));
     list.add(new Student(1, "Aman"));
     list.add(new Student(2, "John"));
     // ----- Sort by ID -----
     System.out.println("Sorting by ID:");
     Collections.sort(list, new SortById());
     System.out.println(list);
     // ----- Sort by Name -----
     System.out.println("Sorting by Name:");
     Collections.sort(list, new SortByName());
     System.out.println(list);
  }
}
```

🔑 *Key Points:*

- Used to **sort objects in a custom order**
- You implement the compare(T o1, T o2) method
- It's useful when you **can't modify the original class** or need **multiple sorting strategies**
- Often used with Collections.sort() or List.sort()

🔍 Comparable vs Comparator

Feature	Comparable	Comparator
Package	`java.lang`	`java.util`
Method	`compareTo(Object o)`	`compare(Object o1, Object o2)`
Sorting logic	Defined in the class	Defined outside the class
Used for	Natural ordering	Custom ordering
Flexibility	One sort rule only	Multiple sort rules possible

🧠 Interview Summary:

Comparator is used when you want to **sort objects using different or external logic**, especially when. The class does **not implement Comparable**
You want to **sort by different fields** (name, age, salary, etc.)

- ➢ compare() → tells how to compare two objects.
- ➢ reversed() → reverse the order.
- ➢ thenComparing() → add second-level sorting.

⭐ *IMPORTANT DIFFERENCE*

✔ Comparable

- Only **one** sorting rule (inside class).
- Example → sort by id.

✔ Comparator

- You can create **many** sorting rules.
- Example → sort by id, sort by name, sort by age, etc.
