Collection Framework

08 August 2025 17:26

Introduction:

- **Framework** is pre-built classes/instructions that we can use easily
- **Collection Framework** was introduced in JDK 1.2 by Josh Bloch and team(when Java was opensource(it was open source till Java 6 under sun microsystems and later became Closed when it was taken by Oracle))
- Collection Framework is example for Abstraction.
- Its collection of classes and Interfaces
- Collections (e.g., ArrayList, HashMap) can store only objects, not primitive data types hence we use wrapper class for it.

---

1. Definition

- **Collection Framework** in Java is a unified architecture for storing, retrieving, and manipulating groups of objects.
- It provides **interfaces**, **classes**, and **algorithms** to handle collections of data efficiently.

2. Need for Collection Framework

Before Java 2 (JDK 1.2):

- Data was stored using **arrays**, **Vector**, **Hashtable**, etc.
- Problems with old approach:
  - Fixed size arrays → Cannot grow/shrink dynamically.
  - Different APIs for similar tasks → No uniform interface.
  - No in-built algorithms (sorting, searching) for data structures.

Collection Framework solves these problems by:

- Providing **dynamic data structures**.
- Offering **common interfaces** and **implementations**.
- Including ready-made algorithms.

3. Key Features

- **Dynamic size**: Collections can grow/shrink as needed.
- **Type safety**: Achieved using **Generics**.
- **Polymorphic behavior**: Code written with interfaces works for multiple implementations.
- **Built-in algorithms**: Sorting, searching, shuffling, etc.
- **Reduces development time**: Pre-built classes.

4. Hierarchy Overview

The **java.util** package contains the main interfaces and classes.

Root Interface:

- **Collection** (extends **Iterable**)

Main branches:

List (Ordered, allows duplicates)
  - Implementations: **ArrayList**, **LinkedList**, **Vector**, **Stack**

Set (No order, No duplicates)
  - Implementations: **HashSet**, **LinkedHashSet**, **TreeSet**

Queue (FIFO order, can be priority-based)
  - Implementations: **PriorityQueue**, **ArrayDeque**

Separate branch:

- **Map** (Key-Value pairs, no duplicate keys) – Not a true child of Collection
  - Implementations: **HashMap**, **LinkedHashMap**, **TreeMap**, **Hashtable**

5. Interfaces in Collection Framework

- **Iterable** – Root interface (iterator() method).
- **Collection** – Base interface for all collections (add(), remove(), size(), etc.).
- **List** – Ordered collection (can contain duplicates).
- **Set** – Unordered collection (no duplicates).
- **Queue** – For processing elements in a specific order.
- **Map** – Key-value mapping interface.
- **SortedSet**, **SortedMap** – Maintain sorted order.
- **NavigableSet**, **NavigableMap** – Extended sorting/navigation methods.

6. Common Methods in Collection Interface

- boolean add(E e)
- boolean remove(Object o)
- boolean contains(Object o)
- int size()
- void clear()
- boolean isEmpty()
- Iterator<E> iterator()

7. Advantages of Collection Framework

- Reusable classes → Less coding.
- Well-tested library → Fewer bugs.
- Consistent API → Easy to learn & use.
- Ready-made algorithms → Faster development.

8. Drawbacks

- Slightly **slower** than primitive arrays in raw performance.
- **Type casting** required in non-generic legacy code.

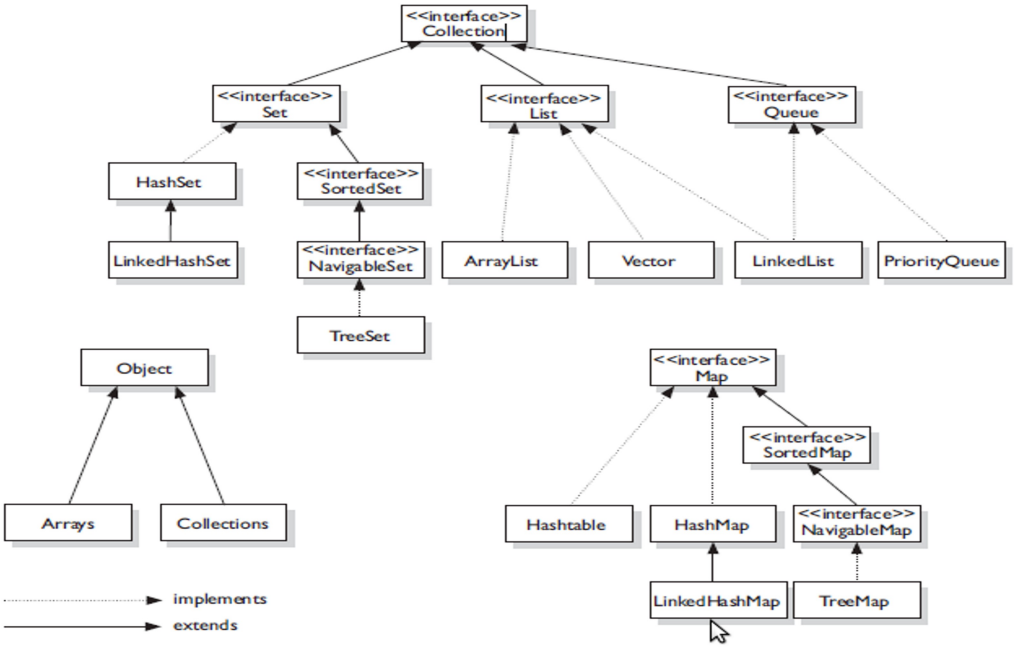
9. Real-world Analogy

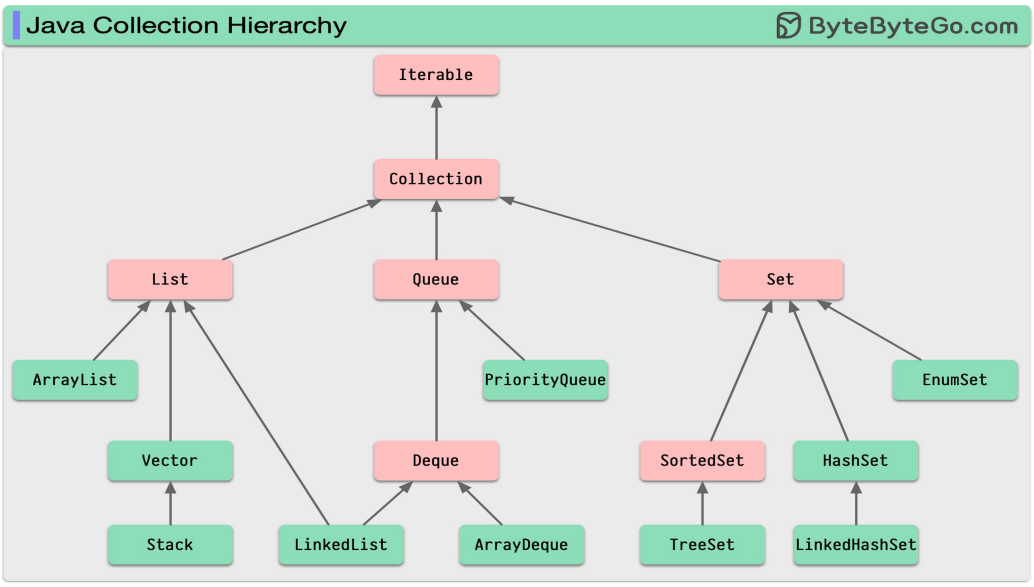
Think of the Collection Framework as a **toolbox**:

- **Interfaces** = Tool categories (hammer, screwdriver)
- **Classes** = Specific tools (Philips screwdriver, claw hammer)
- **Algorithms** = Techniques you can apply (tightening, loosening, sorting)

10. Example Usage in Java

```
java
CopyEdit
import java.util.*;
public class CollectionExample {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>();
      list.add("Apple");
      list.add("Banana");
      list.add("Cherry");
for (String fruit : list) {
         System.out.println(fruit);
      }
   }
}
```





ARRAY LIST:

==========

Array list can grow/shrink unlike array which has fixed size.
Both array and arraylist use contagiuos memory allocation.

Hierarchy :

java.lang.Object
  ↑
java.util.AbstractCollection<E> (class)
  ↑
java.util.AbstractList<E> (class)
  ↑
java.util.ArrayList<E> (class)
  ↳ implements List<E>
  ↑

SequencedCollection<E>
↑
Collection<E>
↑
Iterable<E>

Additional interfaces implemented by ArrayList:
↳ RandomAccess
↳ Cloneable
↳ Serializable

Properties:

- Default capacity of Array list is 10 until we give initial capacity which will create that size of array list.
- It can store Heterogeneous data(different data type in same array list).
- Null insertions are allowed.
- Duplicates are allowed.(even null).
- Preserves the order of insertion
- There are 3 constructors in this class -
  - new ArrayList() - which will have default capacity of 10
  - new ArrayList(int initialCapacity) - We can give the size of array if we already know , which will save memory
  - new ArrayList(Collection c) -
- Internal Data Structure - Array List is resizable.
  - Whenever we add more than 10 elements JVM will create new array List of size 16(0-15). [Has formula: currentCapacity * 3/2 + 1 = 10 * 1.5 + 1 = 16.]
  - And the old array list with 10 elements will be copied to the new Array list of increased size. [old one will be removed by garbage collector].

Methods:

- Add() - to add data to arrayList at last
- Add(index, element) - to add data at particular index
- addAll() - to add other ArrayList's all elements
- addAll(index, Collection c) - to add other Arraylist's elements same as above but starting from index.
- retainAll(Collection c) - to tell one arrayList to keep the duplicate/common elements from other ArrayList. (returns only duplicates from array1)
- removeAll(Collection c) - to remove duplicates of arrayList1 thats also in arrayList2 , and keep only non duplicates from arrayList1- (returns only non duplicates form arr1).
- remove(int index) - remove the element at a particular index,
  (syso(remove(2)) - prints the element at index that has been removed
- Remove(Object o) -
- RemoveIf(Predicate filter)- boolean
- removeFirst();
- removeLast();

Except syso(remove(int index)) : that prints the element, other all methods if put in syso - it returns boolean value true - Eg: syso(array.add(2)); //true

- set(index, element/Collection c) : It replaces the element at a particular index not add. If arr(2) - is 1, set(2,2) - this will replace 1 with 2
- Get(int index) - This will get us the element at a index - syso(arr.get(int index));
- getFast(), getLast(); - get first or last element.
- If we try to get from the index out of bound-we get IndexOutOfBoundException
- Size(Collection c) - will give the size of the arrayList.
- isEmpty() - returns true if arrayList is empty and return false if its not empty

- trimToSize() - since arrayList initial capacity is 10, and we know that we want only 4 spaces among them, then this method can be used to trim the arrayList to its size and save memory.

DIFFERENT WAYS TO ACCESS THE ELEMENTS OF ARRAYLIST:

1. Normal printing the ref - syso(a);
2. For loop:

```
//2. For loop - we can access customized index
        for(int i=0; i<a.size(); i++) {
            System.out.println(a.get(i));
        }
```

3. For each loop/ Enhanced for loop:

```
    //3. For each / enhanced for loop: cant access index based elements
        for(Object e : ar2) { //Since Collection can never store Primitive data types only objects, we must use wrapper class to store objects - like Integer for int, Character for char, but since here it stores heterogeneous data- we must use Object
            System.out.println(e);
        }
```

4. Iterator : it can traverse only in the forward directions, Hence the next method

```
        //4. Iterator - to iterate over array list by checking if there is element next or not as below
        Iterator cursor = a.iterator();
        while(cursor.hasNext()) {
            System.out.println(cursor.next());
        }
        //cannot traverse array list in reverse direction using for each and iterator hence List iterator was introduced
```

5. ListIterator : can traverse array in reverse order also using List Iterator .

```
        //5. ListIterator
        ListIterator cursorl = ar2.listIterator(ar2.size()); //ar2.size wll make start from the index of size and we can access backward
        while(cursorl.hasPrevious()) {
            System.out.println(cursorl.previous());
        }
```

==========

When to use Array List?

- Heterogeneous data
- When we want to create a array not knowing size - create resizable array
- When we want to store duplicates
- When we want to store null values and
- Preserve order of insertion
- When we want to add a element without resizing - since time complexity is O(1)
- Retrieving data from a index - Time complexity is O(1).

Time complexity in array list:

- To add a number at last and array size less than 10**(without resizing) - O(1)**
- When we want to add a number to index 10(array size>10) then JVM will create new array copy elements and add the no. And old array will be collected by garbage collector: **Hence with resizing - time complexity is O(n)**
- To add a no. at index : This will move all the elements from that index to all elements to the right of it to right - So **to add at any random index time complexity is O(n)**
- Array.get(int index): **Retrieving data from a index - Time complexity is O(1)**

DIFFERENCE BETWEEN ARRAY AND ARRAY LIST:

Feature	Array	ArrayList
Definition	Fixed-size data structure that stores elements of the same type (primitive or objects).	Resizable collection class from java.util package that can store objects only.
Size	Fixed at creation; cannot grow or shrink.	Dynamic; can grow or shrink automatically.
Type	Can store both primitives (int, char, etc.) and objects.	Can store only objects (primitives are stored via wrapper classes like Integer).
Heterogeneous	Not allowed	Allowed
Performance	Faster in terms of direct access and lower memory overhead.	Slightly slower due to dynamic resizing and object handling.
Memory Location	Elements are stored in contiguous memory locations.	Internally uses a dynamic array (Object[]) but manages resizing automatically.
Syntax Example	int[] arr = new int[5];	ArrayList<Integer> list = new ArrayList<>();
Length/Size Retrieval	Use .length property (e.g., arr.length).	Use .size() method (e.g., list.size()).
Insertion/Deletion	Manual shifting of elements required.	Automatically shifts elements internally when adding/removing.
Generics Support	Not applicable.	Fully supports Generics for type safety.
Null Values	Can store null in object arrays.	Can store null as an element.
Utility Methods	No built-in utility methods except Arrays class helpers.	Has many built-in methods (add(), remove(), contains(), clear(), etc.).
Utility class	Arrays	Collections
Multidimensional Support	Directly supports multidimensional arrays (int[][] arr).	No direct multidimensional structure (can have nested ArrayList<ArrayList<>>).
Resizing	Must create a new array and copy elements manually.	Automatically resizes internally when needed.
Import:	No importing externally	Importing externally required.
Use Case	Best when size is fixed and performance is critical.	Best when size changes frequently or when using collections framework utilities.

==========

Contiguous memory means that all elements are stored **next to each other in a single continuous block of memory addresses** without gaps.
Example:
If the first element is at memory address 1000,

- 2nd element will be at 1004 (if int = 4 bytes),
- 3rd at 1008, and so on — all in sequence.

📌 **Key points:**

- Easy & fast **index-based access** (direct calculation using address).
- But requires **one large continuous free block**, which can be a problem if memory is fragmented.

Drawbacks of ArrayList:

- **Slow insert/delete in middle** → shifting elements takes O(n) time.
- **Fixed growth pattern** → resizing array internally costs performance.
- **Not memory efficient** → stores extra capacity beyond current size.
- **Contiguous Memory Location Drawback (Array & ArrayList):**
  - Both store elements in **contiguous memory blocks**.
  - Requires finding a **large continuous chunk of memory** → can fail if memory is fragmented.
  - **Resizing cost**: When full, they create a new bigger array and **copy all elements** → costly in time.
  - Makes **insertion/deletion in middle** slow because shifting elements is required.

Why use LinkedList over ArrayList:

- **Faster insert/delete in middle** → O(1) if position known.
- **No resizing cost** → grows dynamically by adding nodes.
- **Better for frequent modifications** → no shifting required.

==========

1. Theory / Concept Questions

Q1. What is ArrayList in Java?
Answer:

- ArrayList is a **resizable array implementation** of the List interface in java.util package.
- Stores elements in **insertion order** and allows duplicates.
- Can store **null** values.
- Backed internally by a **dynamic array**.

Q2. How does ArrayList differ from an array?
Answer:

- Array is fixed size; ArrayList is dynamic.
- Array can store primitives & objects; ArrayList stores only objects.
- Array uses length, ArrayList uses size() method.

Q3. Is ArrayList synchronized?
Answer:

- No. ArrayList is **not synchronized** by default → not thread-safe for concurrent modifications.
- Use Collections.synchronizedList() for thread safety.

Q4. What is the initial capacity of an ArrayList?
Answer:

- Default initial capacity = **10** (Java 8+).
- When full, capacity increases by **50%** (new capacity = old + old/2).

Q5. Can ArrayList store null values?
Answer:

- Yes, ArrayList can store multiple null values.

Q6. How to make an ArrayList read-only?
Answer:

```
List<String> list = new ArrayList<>();
list.add("A");
list = Collections.unmodifiableList(list);
```

Q7. How do you remove duplicates from an ArrayList?
Answer:

```
List<Integer> list = new ArrayList<>(Arrays.asList(1,2,2,3));
list = new ArrayList<>(new LinkedHashSet<>(list));
```

Q8. How does ArrayList grow internally?
Answer:

- Internally uses an Object[] array.
- When adding beyond capacity, it creates a **new array with 1.5x capacity** and copies old elements.

Q9. Is ArrayList fail-fast?
Answer:

- Yes, iterators throw **ConcurrentModificationException** if modified structurally during iteration (except through iterator's remove()).

Q10. How to clone an ArrayList?
Answer:

```
ArrayList<String> copy = (ArrayList<String>) list.clone();
```

2. Coding Questions
Q1. Reverse an ArrayList in Java.

```
Collections.reverse(list);
```

Q2. Sort an ArrayList of strings in descending order.

```
list.sort(Collections.reverseOrder());
```

Q3. Find the maximum element in an ArrayList of integers.

```
int max = Collections.max(list);
```

Q4. Remove all even numbers from an ArrayList.

```
list.removeIf(n -> n % 2 == 0);
```

Q5. Merge two ArrayLists without duplicates.

```
List<Integer> merged = new ArrayList<>(list1);
merged.addAll(list2);
merged = new ArrayList<>(new LinkedHashSet<>(merged));
```

Q6. Iterate over an ArrayList using different methods.

```
for (String s : list) {}
list.forEach(System.out::println);
Iterator<String> it = list.iterator();
while (it.hasNext()) {}
```

Q7. Convert ArrayList to array.

```
String[] arr = list.toArray(new String[0]);
```

3. MCQs

Q1. What is the time complexity of get(index) in an ArrayList?

a) O(1) ✅
b) O(n)
c) O(log n)
d) Depends on size

Q2. Which package contains ArrayList?

a) java.array
b) java.util ✅
c) java.collection
d) java.arrays

Q3. Which is true for ArrayList?

a) Allows null values ✅
b) Stores only primitives
c) Is synchronized by default
d) Fixed size

Q4. What happens if you access an invalid index in ArrayList?

a) NullPointerException
b) ArrayIndexOutOfBoundsException
c) IndexOutOfBoundsException ✅
d) Compilation error

Q5. How to create a thread-safe ArrayList?

a) Collections.synchronizedList(new ArrayList<>()) ✅
b) Use synchronized block on ArrayList object
c) Both
d) None

4. Performance Notes

Operation	Time Complexity
get(index)	O(1)
set(index)	O(1)
add(E e) at end	Amortized O(1)
Insert at index	O(n)
Remove at index	O(n)
Search	O(n)

5. Tricky Interview Scenarios

Scenario 1 – Null Handling

```
ArrayList<String> list = new ArrayList<>();
list.add(null);
System.out.println(list.size()); // 1
```

Scenario 2 – Concurrent Modification

```
for (String s : list) {
   list.remove(s); // ConcurrentModificationException
}
```

Scenario 3 – Initial Capacity

```
ArrayList<Integer> list = new ArrayList<>(5); // Capacity = 5
```

Scenario 4 – Immutable List

```
List<String> list = Arrays.asList("A","B");
// list.add("C"); // UnsupportedOperationException
```

LINKED LIST

==========

1. Introduction

A **Linked List** is a linear data structure where elements (nodes) are stored at non-contiguous memory locations(not continuous memory allocation).

Each node contains:

Data – the value stored.

Pointer/Reference – the address of the next node.

Hierarchy:

```
java.lang.Object
  ↑
java.util.AbstractCollection<E>       (class)
  ↑
java.util.AbstractList<E>             (class)
  ↑
java.util.AbstractSequentialList<E>   (class)
  ↑
java.util.LinkedList<E>               (class)
  ↳ implements List<E>
        ↑
    SequencedCollection<E>
        ↑
    Collection<E>
```

↑
Iterable<E>

↳ implements Deque<E>
↑
Queue<E>
↑
Collection<E>
↑
Iterable<E>

Additional interfaces implemented by LinkedList:
↳ Cloneable
↳ Serializable
==========

Singly Linked list: Only moves forward not possible backwards

[10 | next → 1040] → [20 | next → 2056] → [30 | next → 3020] → [40 | next → null]
↑ ↑ ↑ ↑
5000 1040 2056 3020

Doubly linked list: Can move frwd and backward.(In java this is what Linked list is created)

NULL ← [Prev:null | Data:10 | Next:1040] ⇄ [Prev:5000 | Data:20 | Next:2056] ⇄ [Prev:1040 | Data:30 | Next:3020] ⇄ [Prev:2056 | Data:40 | Next:null] → NULL

2. Key Features

- **Dynamic Size**: Can grow/shrink during runtime (unlike arrays).
- **No memory wastage**: No pre-allocation required.
- **Insertion/Deletion**: Faster than arrays for middle elements.
- **Access Time**: Slower than arrays for random access (O(n)).

3. Types of Linked List

Singly Linked List – Each node points to the next.
Doubly Linked List – Nodes have pointers to both next and previous nodes.
Circular Linked List – Last node points to the first node.
Circular Doubly Linked List – Combination of circular + doubly.

4. Structure of a Node in Java

```
class Node {
  int data;
  Node next;

  Node(int data) {
    this.data = data;
    this.next = null;
  }
}
```

◆ **Properties of Linked List**

- **Default capacity :** 0, Everytime we add data new linked list is created.
- Heterogeneous data , duplicates and null insertion, duplicate null are allowed.
- **Order of insertion** is preserved.
- **Internal data structure :** Doubly linked list
- **Dynamic Size** – Can grow or shrink at runtime (no fixed size like arrays).
- **There are 2 Constructors**
  - New LinkedList() - Created linked list of 0 size, empty
  - New LinkedList(Collection c) -

- **Non-Contiguous Memory** – Nodes are stored at random memory locations, connected via pointers.
- **Node Structure** – Each node contains **data** and a **reference (pointer)** to the next (and/or previous) node.
- **Efficient Insert/Delete** – Insertion and deletion are faster than arrays, especially in the middle.
- **Sequential Access** – Elements must be accessed one by one (no direct index-based access).
- **Multiple Types** – Singly Linked List, Doubly Linked List, Circular Linked List.
- **Extra Memory Overhead** – Needs extra space for storing pointers.
- **Head/Tail Pointers** – Access to the list starts from a head node (and optionally a tail node).

==========

Methods in Linkedlist

--The address will change no need of shifting - hence O(1) is the complexity.

--unique methods like peek, poll, addFirst, last, offer, offerFirst, offerLast that is not in arraylist, we get these because linked list also implements dequeue interface which extends to queue interface

1. addFirst(Object o) - add an object at starting of the list - O(1)
2. addLast(Object o) - add at last
3. removeFirst()
4. removeLast()
5. getFirst() , getLast()-
   doubly linked list implementation):
   - **getFirst()** → **O(1)**
     - Just returns the head node's element. No traversal needed.
   - **getLast()** → **O(1)**
     - Just returns the tail node's element. Java's LinkedList stores a direct reference to the last node, so it doesn't traverse.
   - If you implemented your own **singly linked list** without a tail pointer:
     - getFirst() would still be **O(1)**
     - getLast() would be **O(n)** because you'd have to traverse to the end.
6. get(int index) → O(n) in worst case
   - Because it has to **traverse** from either the head or tail until it reaches the given index.
   - If index is in the **first half**, it starts from the head.
   - If index is in the **second half**, it starts from the tail.

So the actual traversal steps ≈ **O(min(index, size-index))**

But in Big-O notation, that's **O(n)**.

7. Peek(), peekFirst() - just prints the first element
8. peekLast() - prints the last element
9. Poll(), pollFirst() - removes and prints the first element
10. pollLast() - removes and prints the last element
11. Add() - adds the element at last of linked list
12. Offer() - adds the element same as add() - difference is In LinkedList offer() adds the element at the end just like add(), but offer() returns false instead of throwing an exception if capacity is full — though LinkedList is unbounded

==========

Different ways of accessing elements in LinkedList:

---

1. For loop

```
for(int i=0; i<li.size(); i++) {
                System.out.println(li.get(i));
            }
```

2. For each Loop

```
for(Object e: li) {
            System.out.println(e);
        }
```

3. Iterator

```
Iterator cursor = li.iterator();
        while(cursor.hasNext()) {
            System.out.println(cursor.next());
        }
```

List Iterator(child interface) extends Iterator(parent interface) - hasNext(), next() are inherited to ListInterface and previous(), hasPrevious() - specialized methods in list iterator

4. ListIterator

```
ListIterator cursor2 = li.listIterator(li.size());
        while(cursor2.hasNext()) {
            System.out.println(cursor2.next());
        }

        while(cursor2.hasPrevious()) {
            System.out.println(cursor2.previous());
        }
```

==========

5. Time Complexity

Operation	**Singly LL**	**Doubly LL**
Access (by index)	O(n)	O(n)
Search	O(n)	O(n)
Insert at beginning	O(1)	O(1)
Insert at end	O(n)	O(1) if tail ref maintained
Delete at beginning	O(1)	O(1)
Delete at end	O(n)	O(1) if tail ref maintained
Insert in middle	O(n)	O(n)

6. Advantages

- Dynamic size allocation.
- Efficient insertions/deletions.
- No memory wastage.

Disadvantages

- Uses extra memory for storing pointers.
- Sequential access only (slow random access).
- Reverse traversal difficult in singly LL.

7. Use Cases - When to use Linked list

- Heterogeneous data, duplicate elements , null insertions, preserve order of insertion
- Best for Insertion at any end(head and tail -O(1), but in middle - O(n) because this requires traversing the list from the head until the desired insertion point is found.) (Though no shifting like array list, only address change)
- Linked list can Implement and create dequeue, queue and stacks.
- Dynamic memory allocation.
- Undo/redo functionality.
- Adjacency lists in graphs.

==========

Stack Creation using Linked list: (FILO/LIFO)

--> Use push() and pop() instead of add() and remove()

--> Eg Where we use stack - Type abcd , last typed d, ctrl+z will remove the last typed first
Web browser - last url will be the first one to go when we give backspace

```
            LinkedList li = new LinkedList();

            li.push(10);
            li.push(1);
            li.push(20);
            li.push(3);

            System.out.println(li); //3 20 1 10 //The last will be the first to come out hence its at
    first

            li.pop();
            System.out.println(li); //20 1 10 //3 is removes - Last in First Out
```

Queue creation using Linkedlist: (FIFO/LILO)

-->Use add(), removeFirst(); linked list already behaves like queue

--> Eg: queue for tickets

```
        //Queue(FIFO/LILO) creation using linked list - use  not add() ,  removeFirst() , removeLast()

          LinkedList li = new LinkedList(); //it already behaves like queue

          li.add(10);
          li.add(20);
          li.add(30);
          li.add(40);

          System.out.println(li);

          System.out.println(li.removeFirst()); //10

          System.out.println(li);
```

Dequeue creation using Linked list:

--> Can remove and add elements from both side

--> That's from where we get add first last, remove first last

--> Eg: OS scheduling - we can toggle the priority of tasks in task manager

```
          //Dequeue - Double ended q - can add and remove elements from both sides - from this is why and
    where linked list got addFirst and last , removeFirst and last : To add and remove elements on both sides

          LinkedList li = new LinkedList(); //it already behaves like queue

          li.addFirst(10); //10
          li.addFirst(20); //20 10
          li.addLast(30); //20 10 30

          System.out.println(li);

          System.out.println(li.removeFirst()); //10

          System.out.println(li);
```

==========

DIFFERENCE BETWEEN ARRAYLIST AND LINKED LIST:

Feature / Parameter	ArrayList	LinkedList
Package	java.util	java.util
Implements	List, RandomAccess, Cloneable, Serializable	List, Deque, Cloneable, Serializable
Default Size (No-arg Constructor)	**Capacity = 10** elements initially.	**No predefined capacity** (size = 0; grows as nodes are added).
Constructors	3 - noargs, initial capacity, collection c	2 - noargs, collection c
Internal Data Structure	**Resizable Array** (Object[])	**Doubly Linked List** (each node has data, prev, next references).
Memory Units Used	1 -Stores only actual data objects in array slots.	3 -Each element stored in a separate **Node** object containing: 1. Data 2. Reference to previous node 3. Reference to next node.
Memory	Contiguous	Disperssed
Chance of Memory Wastage	**Yes** – due to unused capacity in the array when it's not full.	**Yes (different type)** – extra memory overhead per element due to node object + two references (prev & next).
Capacity Increment Formula	When capacity exceeded: **new capacity = old capacity + (old capacity / 2)** (i.e., 1.5x growth).	No capacity concept; grows dynamically node by node.

Insertion – Time Complexity	**Average**: O(1) at end (amortized). **Worst**: O(n) when inserting at arbitrary index (shift elements).	**At beginning or end**: O(1). **At arbitrary index**: O(n) (need to traverse to index).
Deletion – Time Complexity	**At end**: O(1) amortized. **At arbitrary index**: O(n) (shift elements).	**At beginning or end**: O(1). **At arbitrary index**: O(n) (traversal).
Access (get/set) – Time Complexity	**O(1)** – Direct access via index (random access).	**O(n)** – Must traverse from start or end to reach index.
Search – Time Complexity	O(n) for linear search (unless additional indexing done).	O(n) for linear search.
Memory Efficiency	Better for storing large amounts of primitive wrapper types due to contiguous memory block.	Worse for large amounts of data because each node requires extra reference memory.
Iteration Performance	**Faster** – better cache locality (elements stored contiguously).	**Slower** – due to scattered memory allocation of nodes.
Best Use Case	When frequent access (get, set) is needed and size changes are infrequent.	When frequent insertion/deletion at start/middle is needed, and random access is less important.
Thread Safety	Not synchronized (use Collections.synchronizedList() for thread-safe version).	Not synchronized (needs explicit synchronization for thread-safe use).
Null Elements	Allows multiple nulls.	Allows multiple nulls.
Fail-fast Iterator	Yes (throws ConcurrentModificationException if modified during iteration without iterator).	Yes (same behavior).
Example Initialization	ArrayList<String> list = new ArrayList<>();	LinkedList<String> list = new LinkedList<>();

==========

INTERVIEW QUESTIONS WITH ANSWERS

Q1: Difference between Linked List and Array?

✅ **Answer**:

Linked List	Array
Dynamic size	Fixed size
Non-contiguous memory	Contiguous memory
Insertion/Deletion faster (O(1) for ends)	Insertion/Deletion slower (O(n))
Sequential access only	Random access allowed
Extra memory for pointers	No extra memory for links

Q2: Why is insertion in Linked List O(1) in the beginning?

✅ Because we only change the head pointer to the new node, without shifting elements.

Q3: Why is accessing an element in Linked List O(n)?

✅ Because we must traverse from the head node to the required position.

Q4: How to detect a cycle in a Linked List?

✅ Use **Floyd's Cycle Detection Algorithm (Tortoise and Hare)**.

Q5: How to reverse a Linked List?

✅ Change the direction of all next pointers using iterative or recursive methods.

CODING QUESTIONS

1. Reverse a Linked List

```
Node reverse(Node head) {
   Node prev = null, current = head, next = null;
   while (current != null) {
      next = current.next;
      current.next = prev;
      prev = current;
```

```
        current = next;
    }
    return prev;
}
```

2. Detect Cycle

```
boolean hasCycle(Node head) {
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true;
    }
    return false;
}
```

3. Find Middle Node

```
Node findMiddle(Node head) {
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
```

4. Merge Two Sorted Linked Lists

```
Node mergeTwoLists(Node l1, Node l2) {
    if (l1 == null) return l2;
    if (l2 == null) return l1;

    if (l1.data < l2.data) {
        l1.next = mergeTwoLists(l1.next, l2);
        return l1;
    } else {
        l2.next = mergeTwoLists(l1, l2.next);
        return l2;
    }
}
```

5. Remove Nth Node from End

```
Node removeNthFromEnd(Node head, int n) {
    Node dummy = new Node(0);
    dummy.next = head;
    Node first = dummy, second = dummy;
    for (int i = 0; i <= n; i++) first = first.next;
    while (first != null) {
        first = first.next;
        second = second.next;
    }
    second.next = second.next.next;
    return dummy.next;
}
```

MCQs

Q1: What is the time complexity to insert an element at the beginning of a singly linked list?
a) O(1)
b) O(n)
c) O(log n)
d) O(n^2)
✅ **Answer:** a) O(1)

Q2: Which of these allows constant time insertion/deletion at both ends?
a) Array
b) Singly Linked List
c) Doubly Linked List
d) Queue
✅ **Answer:** c) Doubly Linked List

Q3: In a singly linked list, deleting the last node requires:
a) O(1) time
b) O(n) time
c) O(log n) time
✅ **Answer:** b) O(n)

Q4: Which pointer is null in a singly linked list's last node?
a) Head
b) Next
c) Prev
✅ **Answer:** b) Next

Q5: Which is more memory efficient for large datasets if frequent random access is required?
a) Linked List
b) Array
✅ **Answer:** b) Array

MASTER INTERVIEW Q&A: JAVA LINKEDLIST (LIST INTERFACE IMPLEMENTATION)

SECTION 1 — THEORY QUESTIONS & ANSWERS

Q: What is Java's LinkedList class?
A: It is a doubly-linked list implementation of the List, Deque, and Queue interfaces, allowing fast insertions and deletions but slower random access compared to ArrayList.
Q: Which package contains LinkedList?
A: java.util.
Q: Is LinkedList in Java singly-linked or doubly-linked?
A: Doubly-linked.
Q: Does LinkedList implement RandomAccess interface?
A: No, because random access is not efficient for linked lists.
Q: What is the time complexity of adding an element at the end of a LinkedList?
A: O(1).
Q: What is the time complexity of accessing an element by index in a LinkedList?
A: O(n).
Q: How does LinkedList store elements internally?
A: As Node objects containing data, a reference to the previous node, and a reference to the

next node.
Q: What are the advantages of LinkedList over ArrayList?
A: Faster insertions/deletions in the middle, no need for resizing arrays.
Q: What are the disadvantages of LinkedList over ArrayList?
A: More memory per element, slower random access.
Q: Is LinkedList synchronized?
A: No.
Q: How to make LinkedList thread-safe?
A: Use Collections.synchronizedList(new LinkedList<>()) or use ConcurrentLinkedDeque.
Q: Does LinkedList allow null elements?
A: Yes, it allows multiple nulls.
Q: How does LinkedList handle capacity?
A: It has no fixed capacity; it grows dynamically.
Q: Can LinkedList store duplicate elements?
A: Yes.
Q: What happens if you insert into the middle of a LinkedList?
A: The previous and next references of neighboring nodes are updated to include the new node.
Q: Is LinkedList a good choice for stack implementation?
A: Yes, it can be used as a stack via push(), pop(), and peek().
Q: Is LinkedList a good choice for queue implementation?
A: Yes, via offer(), poll(), and peek().
Q: Can LinkedList be traversed backwards?
A: Yes, using descendingIterator() or ListIterator.
Q: Difference between size() and capacity() in LinkedList?
A: LinkedList has size() but no concept of capacity().
Q: Which iterators does LinkedList support?
A: Iterator, ListIterator, and descendingIterator.

SECTION 2 — MCQ QUESTIONS

What is the default capacity of LinkedList?
a) 10
b) 0
c) No capacity limit
d) 16
Answer: c) No capacity limit
Time complexity of addFirst() in LinkedList?
a) O(1)
b) O(n)
c) O(log n)
d) O(n^2)
Answer: a) O(1)
Which method removes and returns the first element?
a) remove()
b) poll()
c) pop()
d) All of the above
Answer: d) All of the above
Which interface does LinkedList NOT implement?
a) List
b) Deque
c) RandomAccess
d) Cloneable
Answer: c) RandomAccess

In LinkedList, get(index) has complexity:
a) O(1)
b) O(log n)
c) O(n)
d) O(n log n)
Answer: c) O(n)

SECTION 3 — CODING QUESTIONS

Create a LinkedList, add elements, and print:

pgsql
CopyEdit
LinkedList<String> list = new LinkedList<>();
list.add("A");
list.add("B");
list.add("C");
System.out.println(list);
Insert at specific position:

pgsql
CopyEdit
LinkedList<Integer> list = new LinkedList<>();
list.add(1);
list.add(2);
list.add(3);
list.add(1, 99);
System.out.println(list);
Remove first and last elements:

pgsql
CopyEdit
LinkedList<Integer> list = new LinkedList<>(Arrays.asList(1,2,3,4));
list.removeFirst();
list.removeLast();
System.out.println(list);
Iterate using ListIterator:

arduino
CopyEdit
LinkedList<String> list = new LinkedList<>(Arrays.asList("X", "Y", "Z"));
ListIterator<String> it = list.listIterator();
while(it.hasNext()) {
  System.out.println(it.next());
}
Implement a method to reverse a LinkedList:

lua
CopyEdit
Collections.reverse(list);

SECTION 4 — INTERNAL WORKING QUESTIONS

Q: How is a Node defined in LinkedList?
A: As a static inner class with fields: E item, Node<E> next, Node<E> prev.
Q: How does LinkedList keep track of first and last nodes?

A: It maintains head and tail references.
Q: How is element removal implemented internally?
A: It adjusts the next and prev references of neighboring nodes and clears the removed node's references.
Q: Why does LinkedList not support efficient random access?
A: Because it requires sequential traversal from head or tail to the index.
Q: How is fail-fast behavior implemented in LinkedList iterators?
A: By maintaining a modCount and throwing ConcurrentModificationException if the list is structurally modified during iteration.

SECTION 5 — TRICKY INTERVIEW QUESTIONS

Q: What happens if you call get(-1) on LinkedList?
A: Throws IndexOutOfBoundsException.
Q: What is the complexity of contains(Object o)?
A: O(n).
Q: Can LinkedList have heterogeneous elements?
A: Yes, if not using generics.
Q: If two threads modify LinkedList without synchronization, what happens?
A: May cause data corruption and unpredictable behavior.
Q: Difference between remove(Object) and remove(int index)?
A: remove(Object) removes the first occurrence of the object, remove(int index) removes the element at that index.

STACK in COLLECTION:

==========

OVERVIEW

- **Definition**:
  Stack is a **class** in java.util package that represents a **last-in, first-out (LIFO) or First-in Last-out** (FILO) stack of objects.
- **Hierarchy**:
  Object → AbstractCollection → AbstractList → Vector → Stack
- **Implements**: Serializable, Cloneable, Iterable, Collection, List
- **Thread-safety**: Yes (inherited from Vector — all public methods are synchronized).
- **Type**: Generic (from Java 1.5).

HEIRARCHY:

```
java.lang.Object
  ↑
java.util.AbstractCollection<E>   (class)
  ↑
java.util.AbstractList<E>        (class)
  ↑
java.util.Vector<E>              (class)
  ↑
java.util.Stack<E>               (class)
  ↳ implements List<E>
        ↑
    SequencedCollection<E>
        ↑
    Collection<E>
        ↑
    Iterable<E>
```

Additional interfaces implemented by Stack (inherited from Vector):
↳ RandomAccess
↳ Cloneable
↳ Serializable

KEY CHARACTERISTICS

- LIFO order: Last element pushed is the first to be popped.
- Extends Vector → dynamic array internally.
- Allows duplicates and null values.
- Backed by an Object array (via Vector).
- Capacity automatically increases when full.
- Default initial capacity (inherited from Vector) is **10**.

INTERNAL WORKING

- Internally uses Vector's array to store elements.
- push() = append element at the end of Vector.
- pop() = remove last element from Vector.
- peek() = return last element without removing.
- Synchronization: All methods are synchronized (one thread at a time).

CONSTRUCTORS

Stack(): Creates an empty Stack with default Vector capacity (10).
Example:
Stack<Integer> s = new Stack<>();

METHODS — WORKING + TIME COMPLEXITY

push(E item)
- Adds the element to the top of the stack (end of Vector).
- Internally calls addElement(item) from Vector.
- If array is full → capacity doubled → elements copied to new array.
- **Time Complexity**: O(1) amortized (O(n) if resize happens).

pop()
- Removes and returns the top element (last element of Vector).
- Internally calls removeElementAt(size() - 1).
- Throws EmptyStackException if empty.
- **Time Complexity**: O(1).

peek()
- Returns the top element without removing.
- Accesses element at elementAt(size() - 1).
- Throws EmptyStackException if empty.
- **Time Complexity**: O(1).

empty()
- Returns true if stack has no elements.
- Internally checks size() == 0.
- **Time Complexity**: O(1).

search(Object o)
- Returns **1-based position** of object from top of stack.
- If element at top is o → returns 1.
- Traverses from top down, compares using equals().
- If not found → returns -1.

- **Time Complexity**: O(n).

add(E e) *(inherited from Vector)*
- Adds element to end of stack.
- Equivalent to push() for Stack.
- **Time Complexity**: O(1) amortized.

add(int index, E element) *(inherited from Vector)*
- Inserts element at specific position.
- Shifts elements to make space.
- **Time Complexity**: O(n - index) for shifting.

remove(int index) *(inherited from Vector)*
- Removes element at specific index.
- Shifts elements after index left.
- **Time Complexity**: O(n - index).

remove(Object o) *(inherited from Vector)*
- Removes first occurrence of object.
- Shifts elements left.
- **Time Complexity**: O(n).

contains(Object o) *(inherited from Vector)*
- Checks if element exists.
- Linear search using equals().
- **Time Complexity**: O(n).

size() *(inherited from Vector)*
- Returns current number of elements.
- **Time Complexity**: O(1).

clear() *(inherited from Vector)*
- Removes all elements.
- Nullifies each array element to help GC.
- **Time Complexity**: O(n).

iterator() *(inherited from Vector)*
- Returns Iterator over elements in order from bottom to top.
- **Time Complexity**: O(1) to create iterator, O(n) to traverse.

capacity() *(inherited from Vector)*
- Returns current array capacity.
- **Time Complexity**: O(1).

trimToSize() *(inherited from Vector)*
- Reduces capacity to match current size.
- **Time Complexity**: O(n) for array copy.

BEHAVIORAL DETAILS

- **LIFO Principle**: push → push → pop returns the last pushed.
- **Search numbering**: Top is position 1, next is 2, and so on.
- **Null elements**: Allowed — search(), peek(), and pop() work with null values.
- **Iteration order**: Bottom to top (Vector's order), not reverse.
- **Thread-safety**: Suitable for multi-threaded use without additional synchronization.

ADVANTAGES

- Built-in synchronization.
- Simple implementation of stack.
- Dynamic resizing.
- Backed by mature, well-tested Vector class.
- Easy to use with minimal methods to remember.

DISADVANTAGES

- Slower in single-threaded cases due to synchronization overhead.
- Legacy design — modern Java prefers ArrayDeque for stack behavior.
- Can consume more memory than minimal array-based stacks.

COMMON INTERVIEW QUESTIONS + ANSWERS

Q: What is Stack in Java?
A: A class in java.util implementing LIFO stack of objects, extending Vector.
Q: Is Stack synchronized?
A: Yes, because it inherits Vector's synchronized methods.
Q: Which exception is thrown if pop() is called on empty stack?
A: EmptyStackException.
Q: What is the difference between peek() and pop()?
A: peek() returns top element without removing, pop() removes and returns it.
Q: Is Stack generic?
A: Yes, from Java 1.5 onwards.
Q: What is the complexity of push and pop?
A: O(1) amortized for push, O(1) for pop.
Q: Why is ArrayDeque preferred over Stack in modern Java?
A: It is faster in single-threaded environments because it's non-synchronized.
Q: Can you store null in Stack?
A: Yes, but be careful when using equals().
Q: How does search() number positions?
A: From top of stack as position 1.
Q: Does iteration return elements from top to bottom?
A: No, iteration starts from bottom to top.

TRICKY / EDGE CASES

- pop() on empty stack → EmptyStackException, not IndexOutOfBoundsException.
- search(null) works — returns position if null is present.
- push(null) is valid — pop() will return null.
- If multiple equal objects exist, search() returns position of first match from top.
- Iteration order is bottom to top (unlike pop order).
- Capacity can be manually increased using ensureCapacity().
- Cloning a stack creates shallow copy.

SIMPLE CODE EXAMPLES

```
Stack<Integer> s = new Stack<>();
s.push(10); // [10]
s.push(20); // [10, 20]
s.push(30); // [10, 20, 30]
System.out.println(s.peek()); // 30
System.out.println(s.pop()); // 30 → stack now [10, 20]
System.out.println(s.search(10)); // position 2 from top
while(!s.empty()) {
System.out.println(s.pop());
}
```

USE CASES

- Undo/Redo functionality in editors.
- Browser back button.
- Parsing expressions (postfix, prefix).
- Function call management in recursion.
- Syntax checking (balanced parentheses).

==========

A) Theory Questions – With Answers

1. What is a Stack? Explain LIFO.
A stack is a linear data structure that follows the **LIFO (Last In, First Out)** principle — the last element pushed into the stack is the first one to be removed. Imagine a stack of plates: the last plate placed on top is the first one taken off.

2. Which Java class implements Stack?
The java.util.Stack class implements a stack in Java. It extends Vector and provides stack-specific methods like push(), pop(), peek(), empty(), and search().

3. Stack vs Queue — difference?
- **Stack**: LIFO order (Last In, First Out)
- **Queue**: FIFO order (First In, First Out)
  Example: Stack is like a stack of plates, Queue is like a line at a ticket counter.

4. Why does Stack extend Vector instead of ArrayList?
Historically, Stack was introduced before ArrayList existed in Java 1.2. Vector was the dynamic array implementation at that time, so Stack was built on it.

5. Is Stack synchronized? How?
Yes. Since Stack extends Vector, all its methods are synchronized, meaning they are thread-safe for concurrent access.

6. Explain push(), pop(), peek() in Stack.
- **push(E item)** → Inserts item at the top.
- **pop()** → Removes and returns top item. Throws EmptyStackException if empty.
- **peek()** → Returns top item without removing it.

7. What is returned by search() method if element not found?
The search() method returns **-1** if the element is not found in the stack.

8. Can Stack store null values?
Yes, Stack can store null values because it does not have restrictions on elements (unless using generics with primitives via wrappers).

9. How does peek() differ from pop()?
peek() returns the top element without removing it.
pop() removes and returns the top element.

10. What is the time complexity of push and pop in Stack?
Both push() and pop() have **O(1)** time complexity on average.

11. Why is ArrayDeque preferred over Stack in modern Java?

ArrayDeque is preferred because:

- It's faster (no synchronization overhead).
- It implements Deque, allowing cleaner stack methods (push, pop, peek).
- Stack is considered a legacy class.

12. Explain how Stack works internally in Java.
Internally, Stack uses a dynamic array provided by Vector. Elements are stored in contiguous memory, and the top of the stack corresponds to the last index of the array.

13. How is Stack related to List interface?
Stack indirectly implements the List interface because it extends Vector, which implements List.

14. Is Stack a generic class in Java?
Yes. You can create a type-safe stack using generics, e.g., Stack<Integer> stack = new Stack<>();.

15. Can Stack hold heterogeneous objects?
Yes, if generics are not used (raw type), a Stack can store heterogeneous objects. With generics, all elements must be of the specified type.

16. Is Stack ordered?
Yes, elements in a stack maintain insertion order, but access is only allowed from the top.

17. How does capacity grow in Stack when using push()?
When the internal array (from Vector) is full, capacity grows by **doubling the size** of the array.

18. What happens if you call pop() on an empty stack?
An EmptyStackException is thrown.

19. Can you iterate through a Stack? How?
Yes, you can use:

- For-each loop
- Iterator
- Stream API

20. Does Stack support random access?
Yes, because it extends Vector, but random access is generally discouraged for stacks as it violates LIFO principles.

B) Coding Questions – With Answers

1. Reverse a string using Stack

java
CopyEdit

```
import java.util.Stack;
public class ReverseString {
   public static String reverse(String s) {
      Stack<Character> stack = new Stack<>();
      for (char c : s.toCharArray()) stack.push(c);
      StringBuilder sb = new StringBuilder();
      while (!stack.isEmpty()) sb.append(stack.pop());
      return sb.toString();
   }
}
```

2. Check for balanced parentheses using Stack

```
java
CopyEdit
public boolean isBalanced(String str) {
   Stack<Character> stack = new Stack<>();
   for (char c : str.toCharArray()) {
      if (c == '(') stack.push(c);
      else if (c == ')') {
         if (stack.isEmpty()) return false;
         stack.pop();
      }
   }
   return stack.isEmpty();
}
```

3. Convert infix to postfix using Stack

```
java
CopyEdit
// Shunting-yard algorithm
```
(You can expand this fully for implementation in notes.)

4. Evaluate postfix expression using Stack

```
java
CopyEdit
public int evalPostfix(String exp) {
   Stack<Integer> stack = new Stack<>();
   for (char c : exp.toCharArray()) {
      if (Character.isDigit(c)) stack.push(c - '0');
      else {
         int b = stack.pop(), a = stack.pop();
         switch (c) {
            case '+': stack.push(a + b); break;
            case '-': stack.push(a - b); break;
            case '*': stack.push(a * b); break;
            case '/': stack.push(a / b); break;
         }
      }
   }
   return stack.pop();
}
```

5. Implement two stacks in one array
(Requires maintaining two pointers from both ends.)

6. Sort a stack using another stack

```
java
CopyEdit
public Stack<Integer> sortStack(Stack<Integer> s) {
   Stack<Integer> temp = new Stack<>();
```

```
    while (!s.isEmpty()) {
        int val = s.pop();
        while (!temp.isEmpty() && temp.peek() > val) s.push(temp.pop());
        temp.push(val);
    }
    return temp;
}
```

7. Implement min-stack

```
java
CopyEdit
class MinStack {
    Stack<Integer> stack = new Stack<>();
    Stack<Integer> minStack = new Stack<>();
    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek()) minStack.push(x);
    }
    public int pop() {
        int val = stack.pop();
        if (val == minStack.peek()) minStack.pop();
        return val;
    }
    public int getMin() { return minStack.peek(); }
}
```

8. Stack palindrome check

```
java
CopyEdit
public boolean isPalindrome(String str) {
    Stack<Character> stack = new Stack<>();
    for (char c : str.toCharArray()) stack.push(c);
    for (char c : str.toCharArray()) {
        if (c != stack.pop()) return false;
    }
    return true;
}
```

9. Stack using queues
(Classic implementation using two queues.)

10. Next Greater Element using Stack

```
java
CopyEdit
public int[] nextGreater(int[] arr) {
    int[] res = new int[arr.length];
    Stack<Integer> stack = new Stack<>();
    for (int i = arr.length - 1; i >= 0; i--) {
        while (!stack.isEmpty() && stack.peek() <= arr[i]) stack.pop();
        res[i] = stack.isEmpty() ? -1 : stack.peek();
        stack.push(arr[i]);
```

```
    }
    return res;
}
```

11. Stock Span Problem

```
java
CopyEdit
public int[] stockSpan(int[] prices) {
    int[] span = new int[prices.length];
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < prices.length; i++) {
        while (!stack.isEmpty() && prices[stack.peek()] <= prices[i]) stack.pop();
        span[i] = stack.isEmpty() ? i + 1 : i - stack.peek();
        stack.push(i);
    }
    return span;
}
```

12. Undo feature simulation
(Push previous states into a stack, pop to revert.)

13. Reverse a linked list using Stack
(Push nodes into stack, pop to reconstruct list.)

14. Tower of Hanoi using Stack
(Simulate recursive moves using explicit stack of states.)

C) MCQs – With Answers

1. Stack follows which principle?
b) LIFO ✅
2. Which method checks if stack is empty?
b) empty() ✅
3. What does search() return if element not found?
b) -1 ✅
4. Time complexity of push in Stack:
a) O(1) ✅
5. What is thrown by pop() on empty stack?
b) EmptyStackException ✅
6. Which is preferred in modern Java for stack operations?
c) ArrayDeque ✅

ArrayDeque

1. Introduction

- **ArrayDeque** (short for *Array Double Ended Queue*) is a **resizable-array implementation** of the **Deque interface**.
- Supports **insertion and removal** from **both ends** in **O(1)** amortized time.

- Unlike LinkedList, it **does not store node references**, so it's **more memory-efficient** and generally **faster**.
- **No capacity restrictions** — grows automatically as needed.
- **Not thread-safe** — must be manually synchronized for concurrent access.
- Can be used **as a Queue**, **Deque**, or **Stack**.
- **Introduced in Java 6**. - 1.6

Hierarchy:

```
java.lang.Object
  ↑
java.util.AbstractCollection<E>    (class)
  ↑
java.util.ArrayDeque<E>          (class)
    ↳ implements Deque<E>
          ↑
       Queue<E>
          ↑
       Collection<E>
          ↑
       Iterable<E>

Additional interfaces implemented by ArrayDeque:
  ↳ Cloneable
  ↳ Serializable
```

---

2. Key Features

- Heterogeneous data, duplicates are allowed
- **Default Capacity:** 16
- **Constructors**: 3 - default, (int numElements)- initial capacity, Collection c
- **Internal data structure : Resizable Array**: Internally backed by a circular array.
  - When we add elements more than 16 , increment formula is doubling - 16*2=32
- **Double-Ended Operations**: Can add/remove from both front and rear efficiently.
- **Better than Stack / LinkedList**: Faster for stack and queue operations in single-threaded use.
- **Does not allow null elements**. - NullPointerException
- **Not fixed size** — expands dynamically.
- **No random access by index** like ArrayList. - Array Deque has no indexing

---

3. Internal Working

- Uses a **circular buffer** array internally.
- Maintains **head** (front index) and **tail** (end index) pointers.
- When head/tail reaches the end of the array, it **wraps around**.
- Resizing happens when the array becomes full — new capacity = old capacity × 2.

Different ways of accessing elements in Array deque:

1. For loop : Not possible since array deque has no indexing.
2. For each loop : No indexing, still this method works because for-each loop doesn't need index.

```
for(Object e: arr) {
     System.out.println(e);
}
```

3. Iterator :

```
Iterator cursor = arr.iterator();
while(cursor.hasNext()) {
     System.out.println(cursor.next());
}
```

4. List Iterator : This is not possible because This method is only for classes that implement List interface, and queue is not a list. Hence not possible.
5. Descending Iterator:

```
//5. Descending Iterator - this will traverse in both backward and forward direction, pointer will be
at last by default and we say hasNext()- it moves in backward direction - hasNext will work like
```

```
hasPrevious()
        Iterator cursor2 = arr.descendingIterator();
        while(cursor2.hasNext()) { //hasNext() works like hasPreviuos
            System.out.println(cursor2.next());
        }
```

When to use Array Deque ?

- Heterogeneous data , duplicate entries, no null insertions, preserve order of insertion.
- Best for insertion at first and last - O(1), (accessing elements - O(n) since traversing is must)
- Create - stack , queue, deque

Create - stack , queue, deque:

```
        //Stack:
        ArrayDeque arr = new ArrayDeque();
        arr.push(1);
        arr.push(11);
        arr.push(12);

        System.out.println(arr);//stack is created - Last element displayed first, since it will be the
first one to be out
        System.out.println(arr.pop()); //12 //last element is removed first

        //-------------------------------------------------

        //Queue:
        ArrayDeque ar = new ArrayDeque();
        ar.add(1);
        ar.add(11);
        ar.add(12);

        System.out.println(ar);//
        System.out.println(ar.pop()); //1 - first element will be removed

        //-------------------------------------------------------

        //Deque:
        ArrayDeque a = new ArrayDeque();
        a.addFirst(1);
        a.add(11);
        a.addLast(12);

        System.out.println(a);
        System.out.println(a.pop()); //1
```

---

4. Constructors

ArrayDeque()

Creates an empty deque with initial capacity (default = 16).

ArrayDeque(int numElements)

Creates an empty deque with enough capacity for numElements.

ArrayDeque(Collection<? extends E> c)

Creates a deque containing the elements of the given collection, in order.

---

5. Important Methods and Working

Insertion

Add(), addAll()

addFirst(E e) – Inserts at front (throws exception if full — practically won't happen since resizes).

addLast(E e) – Inserts at end.

offerFirst(E e) – Inserts at front (returns false if fails instead of throwing).

offerLast(E e) – Inserts at end.

Time Complexity: O(1) amortized.

Removal

removeFirst() – Removes and returns first element (throws exception if empty).
removeLast() – Removes and returns last element.
pollFirst() – Removes and returns first element (returns null if empty).
pollLast() – Removes and returns last element.

Time Complexity: O(1) amortized.

Retrieval (No Removal)

getFirst() – Returns first element (throws exception if empty).
getLast() – Returns last element.
peekFirst() – Returns first element (returns null if empty).
peekLast() – Returns last element.

Time Complexity: O(1).

Stack-Like Methods (LIFO)

push(E e) – Adds to front (like addFirst).
pop() – Removes from front (like removeFirst).

Time Complexity: O(1) amortized.

Other Utility Methods

size() – Returns number of elements.
isEmpty() – Checks if empty.
contains(Object o) – O(n) search.
iterator() – Returns forward iterator.
descendingIterator() – Returns reverse iterator.
clear() – Removes all elements.

---

6. Null Handling

- **Null values not allowed** — any null insertion will throw NullPointerException.

---

7. Performance

- **Add/Remove at ends**: O(1) amortized.
- **Contains/Search**: O(n).
- **Iteration**: O(n).
- **Better performance than LinkedList** in most use-cases due to contiguous memory.

---

8. Usage Examples

Basic Queue

```
ArrayDeque<Integer> queue = new ArrayDeque<>();
queue.offer(10);
queue.offer(20);
queue.offer(30);
System.out.println(queue.poll()); // 10
System.out.println(queue.peek()); // 20
```

Basic Stack

```
ArrayDeque<String> stack = new ArrayDeque<>();
stack.push("A");
stack.push("B");
stack.push("C");
System.out.println(stack.pop()); // C
System.out.println(stack.peek()); // B
```

Deque Operations

```
ArrayDeque<Integer> deque = new ArrayDeque<>();
deque.addFirst(1);
deque.addLast(2);
deque.offerFirst(0);
deque.offerLast(3);
System.out.println(deque); // [0, 1, 2, 3]
deque.removeFirst(); // Removes 0
deque.removeLast();  // Removes 3
System.out.println(deque); // [1, 2]
```

---

9. Common Interview Questions (Conceptual)

What is ArrayDeque? How is it different from ArrayList and LinkedList?

Answer: ArrayDeque is a resizable array-based implementation of a double-ended queue, allowing insertion/removal from both ends in O(1) time. Unlike ArrayList, it doesn't allow random index access. Unlike LinkedList, it is more memory-efficient and faster in most cases.

Why is ArrayDeque preferred over Stack class?

Answer: Stack is synchronized (slower) and inherits from Vector, making it outdated. ArrayDeque is faster and more efficient for LIFO operations.

Can ArrayDeque store null?

Answer: No, null values are not allowed.

What is the time complexity of offerFirst/offerLast?

Answer: O(1) amortized.

Is ArrayDeque thread-safe?

Answer: No, it must be synchronized manually for multi-threading.

10. Common Interview Questions (Coding)

Q1: Reverse a string using ArrayDeque

```
String str = "hello";
ArrayDeque<Character> stack = new ArrayDeque<>();
for(char c : str.toCharArray()) {
   stack.push(c);
}
StringBuilder reversed = new StringBuilder();
while(!stack.isEmpty()) {
   reversed.append(stack.pop());
}
System.out.println(reversed.toString()); // olleh
```

Q2: Implement Queue using ArrayDeque

```
ArrayDeque<Integer> q = new ArrayDeque<>();
q.offer(1);
q.offer(2);
q.offer(3);
System.out.println(q.poll()); // 1
System.out.println(q.peek()); // 2
```

Q3: Implement Stack using ArrayDeque

```
ArrayDeque<Integer> s = new ArrayDeque<>();
s.push(1);
s.push(2);
s.push(3);
System.out.println(s.pop()); // 3
System.out.println(s.peek()); // 2
```

11. Advantages
- Fast for queue/stack operations.
- No memory overhead like LinkedList.
- Dynamic resizing.
- Can be used as queue, deque, or stack.

12. Disadvantages
- No random access.
- Not thread-safe.
- Slower for mid-position insertions (O(n)).

1️⃣ ArrayList vs ArrayDeque

Aspect	ArrayList	ArrayDeque
Default Size (No-arg Constructor)	10 elements	16 elements
Internal Data Structure	Resizable dynamic array	Resizable circular array
Chances of Memory Wastage	Possible if list has unused capacity after removals	Possible due to unused slots in circular buffer after deque shrink/expansion
Incrementing Formula (Resizing)	newCapacity = oldCapacity + (oldCapacity >> 1) → grows by 50%	newCapacity = oldCapacity * 2 (doubles size)
Insertion (Amortized Time Complexity)	**End**: O(1) amortized**Middle**: O(n) due to shifting	**Front/End**: O(1) amortized**Middle**: Not supported efficiently
Access (get/set)	O(1)	O(1)
Insertion at Front	O(n) — requires shifting	O(1) amortized — designed for fast front insertion
Deletion at Front	O(n) — requires shifting	O(1) amortized
Random Access	Very efficient (O(1))	Efficient (O(1)) but not designed for random access use
Null Elements Allowed?	Yes (any number)	No null elements allowed
Usage Preference	Best for indexed random access and append-heavy lists	Best for queue/deque operations at both ends

==========

2️⃣ LinkedList vs ArrayDeque

Aspect	LinkedList	ArrayDeque
Default Size (No-arg Constructor)	0 (empty list, grows dynamically)	16 elements
Internal Data Structure	Doubly linked list	Resizable circular array
Memory Units per Element	Higher — stores data + 2 references (prev, next)	Lower — stores only data in array slot
Chances of Memory Wastage	Higher — overhead for node objects even if few elements	Possible unused slots in circular buffer
Insertion (Front/End)	O(1) — just re-link nodes	O(1) amortized
Insertion in Middle	O(n) — traverse to position, then re-link	Not efficient — no direct middle insertion method
Access (by Index)	O(n) — must traverse nodes	O(1) — index calculation in array

Deletion (Front/End)	O(1) — re-link nodes	O(1) amortized
Random Access	Inefficient (O(n))	Efficient (O(1))
Null Elements Allowed?	Yes	No
Usage Preference	Best for frequent insertions/deletions anywhere in list	Best for queue/deque operations at both ends with low memory overhead

==========

3️⃣ Array vs ArrayDeque

Aspect	Array	ArrayDeque
Default Size (No-arg Constructor)	Not applicable — fixed size must be given at creation	16 elements
Internal Data Structure	Fixed-size array	Resizable circular array
Memory Wastage	None if used fully; wasted slots if unused	Possible unused slots after resizing
Resizing	Not possible — new array must be created manually	Automatic resizing — doubles capacity
Insertion at End	O(1) if space available, else new array creation O(n)	O(1) amortized
Insertion at Front	O(n) — requires shifting elements	O(1) amortized
Access (by Index)	O(1)	O(1)
Deletion at Front	O(n) — shift elements left	O(1) amortized
Fixed or Dynamic Size	Fixed	Dynamic
Null Elements Allowed?	Yes	No
Usage Preference	Best for fixed-size collections and primitive storage	Best for dynamic double-ended queue operations

---

I. Theory / Technical Interview Questions on ArrayDeque

1. What is an ArrayDeque in Java?
Answer:
ArrayDeque (Array Double-Ended Queue) is a resizable array-based implementation of the Deque interface in Java that allows insertion and removal from both ends in **O(1) amortized time**. It does **not allow null elements** and is **not thread-safe**.

2. How is ArrayDeque different from other Deque implementations?
Answer:
- Unlike LinkedList, ArrayDeque is **array-backed** instead of node-based.
- It is faster than LinkedList for **most operations** due to better cache locality and less memory overhead.
- Unlike PriorityQueue, it does not maintain a priority ordering; it's FIFO/LIFO based.

3. What is the initial capacity of an ArrayDeque?
Answer:
By default, 16 elements. You can specify capacity using the constructor.

4. How does ArrayDeque grow internally when it becomes full?
Answer:
When full, it **doubles** the current capacity (capacity × 2) and rearranges elements so that the head is at index 0.

5. Why is ArrayDeque preferred over Stack?

Answer:
- Stack is synchronized (slower for single-threaded use).
- ArrayDeque is faster and more modern for stack-like operations (push, pop).

6. Why is ArrayDeque preferred over LinkedList for queue operations?
Answer:
- Fewer memory allocations (array vs node objects).
- Better locality of reference.
- Typically **O(1)** faster for enqueuing and dequeuing.

7. Can ArrayDeque contain null elements? Why or why not?
Answer:
No. Nulls are not allowed because they are used internally to mark empty slots.

8. Is ArrayDeque thread-safe? How to make it thread-safe?
Answer:
No. To make it thread-safe, wrap it using:

java
CopyEdit
Deque<Integer> dq = Collections.synchronizedDeque(new ArrayDeque<>());

9. What interfaces does ArrayDeque implement?
Answer:
- Deque<E>
- Queue<E>
- Iterable<E>
- Collection<E>

10. How does ArrayDeque handle wrap-around indexing?
Answer:
It uses **modular arithmetic** (index & (capacity - 1)) since capacity is always a power of 2, making modulo operation efficient via bit masking.

11. What happens when you remove elements from an ArrayDeque? Does it shrink?
Answer:
No automatic shrinking; capacity remains fixed after expansion.

12. Is ArrayDeque faster than ArrayList for FIFO/LIFO operations?
Answer:
Yes, because ArrayDeque avoids shifting elements on front insertions/removals.

13. What is the time complexity of insertion/removal from both ends in ArrayDeque?
Answer:
- Amortized **O(1)** for both ends.
- No shifting like ArrayList.

14. What's the difference between addFirst() and offerFirst() in ArrayDeque?
Answer:
- addFirst() throws an exception if the deque is full (in fixed capacity scenarios).
- offerFirst() returns false instead of throwing an exception.

15. Can we use ArrayDeque for implementing stack and queue both?
Answer:

Yes.

- Stack: push() / pop()
- Queue: offer() / poll() / peek()

II. Important Coding Questions on ArrayDeque

1. Implement a stack using ArrayDeque:

```
java
CopyEdit
import java.util.*;
public class StackUsingArrayDeque {
   public static void main(String[] args) {
      ArrayDeque<Integer> stack = new ArrayDeque<>();
// Push
      stack.push(10);
      stack.push(20);
      stack.push(30);
// Pop
      System.out.println(stack.pop()); // 30
// Peek
      System.out.println(stack.peek()); // 20
   }
}
```

2. Implement a queue using ArrayDeque:

```
java
CopyEdit
import java.util.*;
public class QueueUsingArrayDeque {
   public static void main(String[] args) {
      ArrayDeque<Integer> queue = new ArrayDeque<>();
// Enqueue
      queue.offer(10);
      queue.offer(20);
      queue.offer(30);
// Dequeue
      System.out.println(queue.poll()); // 10
// Peek
      System.out.println(queue.peek()); // 20
   }
}
```

3. Reverse a queue using ArrayDeque:

```
java
CopyEdit
import java.util.*;
public class ReverseQueue {
   public static void main(String[] args) {
      ArrayDeque<Integer> queue = new ArrayDeque<>();
      queue.offer(1);
```

```
        queue.offer(2);
        queue.offer(3);
ArrayDeque<Integer> reversed = new ArrayDeque<>();
        while (!queue.isEmpty()) {
            reversed.push(queue.poll());
        }
        System.out.println(reversed); // [3, 2, 1]
    }
}
```

4. Check if a string is palindrome using ArrayDeque:

```
java
CopyEdit
import java.util.*;
public class PalindromeCheck {
    public static void main(String[] args) {
        String str = "madam";
        ArrayDeque<Character> deque = new ArrayDeque<>();
for (char c : str.toCharArray()) {
            deque.addLast(c);
        }
boolean isPalindrome = true;
        while (deque.size() > 1) {
            if (deque.removeFirst() != deque.removeLast()) {
                isPalindrome = false;
                break;
            }
        }
System.out.println(isPalindrome ? "Palindrome" : "Not Palindrome");
    }
}
```

5. Rotate elements in a deque:

```
java
CopyEdit
import java.util.*;
public class RotateDeque {
    public static void main(String[] args) {
        ArrayDeque<Integer> dq = new ArrayDeque<>(Arrays.asList(1, 2, 3, 4, 5));
        int k = 2; // rotate right by 2
        for (int i = 0; i < k; i++) {
            dq.addFirst(dq.removeLast());
        }
        System.out.println(dq); // [4, 5, 1, 2, 3]
    }
}
```

III. MCQs on ArrayDeque

1. Which of the following statements is true for ArrayDeque?
A) Allows null elements
B) Is thread-safe

C) Has no capacity limit other than memory
D) Maintains elements in sorted order
✅ Answer: **C**

2. Time complexity of offerFirst() in ArrayDeque?
A) O(n)
B) O(log n)
C) O(1) amortized
D) O(n^2)
✅ Answer: **C**

3. What is the default capacity of ArrayDeque?
A) 8
B) 10
C) 16
D) 32
✅ Answer: **C**

4. Which data structure does ArrayDeque internally use?
A) Linked List
B) Circular Array
C) Doubly Linked List
D) Hash Table
✅ Answer: **B**

5. Which of these is NOT a valid ArrayDeque method?
A) push()
B) pop()
C) peekFirst()
D) removeMiddle()
✅ Answer: **D**

TreeSet

1. Introduction

- TreeSet is a **NavigableSet** implementation based on a **Red-Black Tree**.
- Belongs to java.util package.
- Tree Set is good for ranging operations.
- Internal Data Structure is Red black tree / Binary search tree
- Stores elements in **sorted (ascending) order** according to their **natural ordering** or a **custom comparator**.
- It sorts elements **according to their natural ordering** (via Comparable) or a custom ordering (via Comparator).
- **No duplicates** allowed. No heterogeneous data is allowed(`ClassCastException`).
- **Null elements**: `NullPointerException`
  - Not allowed for comparison-based ordering (throws NullPointerException if null is inserted after first element).
  - null is allowed only if the set is empty **and** no custom comparator is used — but inserting more elements may cause errors.
- Default capacity - 0
- Constructors: TreeSet(){}, TreeSet(Comparator){}, TreeSet(Collection c), TreeSet(SortedSet s),TreeSet(NavigableMap m) - Total 5

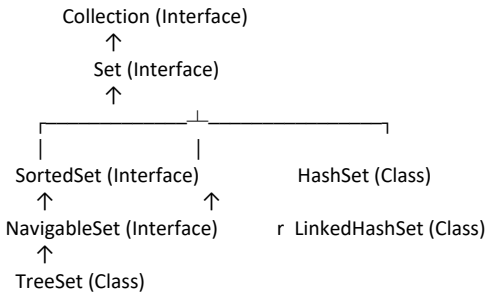
---

2. Class Hierarchy

```
java.lang.Object
  ↑
java.util.AbstractCollection<E>     (class)
  ↑
java.util.AbstractSet<E>          (class)
  ↑
java.util.TreeSet<E>              (class)
    ↳ implements NavigableSet<E>
          ↑
       SortedSet<E>
          ↑
       Set<E>
          ↑
       Collection<E>
          ↑
       Iterable<E>
```

Additional interfaces implemented by TreeSet:
↳ Cloneable
↳ Serializable

Implements:
Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>, NavigableSet<E>, SortedSet<E>



---

3. Key Features

- **Sorted**: Maintains elements in ascending order by default.
- **Self-balancing**: Uses **Red-Black Tree** (balanced binary search tree) internally.
- **Navigable methods**: Supports range searches and navigation methods (higher(), lower(), ceiling(), floor(), subSet(), etc.).
- **No index-based access**: Unlike ArrayList, elements are accessed via iterators or navigable methods.
- **Thread safety**: Not synchronized. Can be made synchronized via Collections.synchronizedSortedSet().

---

4. Internal Implementation

- Uses **TreeMap** internally, with elements stored as **keys** and a dummy constant object as the value.
- **Red-Black Tree properties**:
  Each node is either red or black.
  Root is always black.
  No two consecutive red nodes.
  Every path from a node to a null leaf has the same number of black nodes.
- Ensures **O(log n)** for add, remove, search.

How Sorting Happens Internally in TreeSet

- TreeSet stores elements in **sorted order** because it uses a **Red-Black Tree** internally (a type of self-balancing binary search tree).
- **Rules of storage:**
  Natural ordering (via Comparable) if no custom comparator is provided.

Example: Integers sorted ascending, Strings sorted lexicographically.
Custom ordering if a Comparator is given at creation.

- **When inserting an element**:
  - The Red-Black Tree compares it with existing nodes.
  - It places it **left** if smaller, **right** if larger.
  - Balances the tree to maintain **O(log n)** operations.

Example Code

```
import java.util.*;
public class TreeSetExample {
   public static void main(String[] args) {
      TreeSet<Integer> set = new TreeSet<>();
      set.add(50);
      set.add(30);
      set.add(70);
      set.add(20);
      set.add(40);
      set.add(60);
      set.add(80);
System.out.println("TreeSet elements: " + set);
   }
}
```

How It's Stored Internally (Red-Black Tree Visual)
Step-by-step insertion order:

1. Insert **50** → becomes root.
   50 (Black)
2. Insert **30** → smaller than 50 → goes left.
   50 (Black)
   /
   30 (Red)
3. Insert **70** → greater than 50 → goes right.
   50 (Black)
   / \
   30(R) 70(R)
4. Insert **20** → goes left of 30.
   50 (Black)
   / \
   30(R) 70(R)
   /
   20(R)

(Here balancing happens: recoloring/restructuring to maintain Red-Black properties.)

5. Insert **40** → goes right of 30.

   50 (Black)
   / \
   30(B) 70(R)
   / \
   20(R) 40(R)
6. Insert **60** → goes left of 70.
   50 (Black)
   / \

30(B) 70(B)
/ \ /
20(R)40(R)60(R)

7. Insert **80** → goes right of 70.

50 (Black)
/ \
30(B) 70(B)
/ \ / \
20(R)40(R)60(R)80®

Final Sorted Order

When you print:

TreeSet elements: [20, 30, 40, 50, 60, 70, 80]

➡️ This is **in-order traversal** of the Red-Black Tree.

Tree Traversals:

1. Inorder (Left → Root → Right) used in tree set

- Visit **left subtree**, then **root**, then **right subtree**.
- For **BST**, this gives elements in **sorted order**.
- Example (BST with 20, 30, 40): Output → 20 30 40

2. Preorder (Root → Left → Right)

- Visit **root first**, then left subtree, then right subtree.
- Useful for **copying a tree** or getting prefix expression.
- Example: Output → 30 20 40

3. Postorder (Left → Right → Root)

- Visit **left subtree**, then **right subtree**, then root.
- Used in **deleting tree** or getting postfix expression.
- Example: Output → 20 40 30

Example code (all 3 in one):

```
void inorder(Node n)  { if(n != null){ inorder(n.left);  System.out.print(n.data+" "); inorder(n.right);} }
void preorder(Node n) { if(n != null){ System.out.print(n.data+" "); preorder(n.left); preorder(n.right);} }
void postorder(Node n){ if(n != null){ postorder(n.left); postorder(n.right); System.out.print(n.data+" ");}
```

ACCESSING ELEMENTS IN TREESET:

==========

1. For loop : not possible since tree set has no indexing
2. For each loop

```
for(Object e: set) {
    System.out.println(e); //prints sorted order
}
```

3. Iterator

```
Iterator cursor = set.iterator();
while(cursor.hasNext()) {
    System.out.println(cursor.next());
}
```

4. ListIterator : Its not even list and there is no indexing so not possible
5. Descending Iterator

```
Iterator cursor1 = set.descendingIterator();
while(cursor1.hasNext()) {
    System.out.println(cursor1.next()); //prints descending sorted order
}
```

---

5. Constructors

1. **TreeSet()**
   - Natural ordering, empty set.
2. **TreeSet(Collection<? extends E> c)**
   - Creates a set containing elements of collection c.
3. **TreeSet(SortedSet<E> s)**
   - Creates a set from another sorted set.
4. **TreeSet(Comparator<? super E> comparator)**
   - Custom ordering.
5. **TreeSet(NavigableMap<E, Object> m)**
   -

---

6. Time Complexity

Operation	Complexity	Reason
add()	O(log n)	Tree traversal + possible rebalancing
remove()	O(log n)	Tree traversal + rebalancing
contains()	O(log n)	Tree search
size()	O(1)	Maintains element count internally
iterator()	O(1)	Retrieves iterator, actual traversal O(n)

---

7. Memory Usage

- Slightly more memory than HashSet because it maintains tree nodes with:
  - Key (element)
  - Color bit
  - Links to parent, left, right children
- **Extra ~40–50 bytes** per element compared to a hash-based set.

---

8. Important Methods

Method	Description
add(E e)	Adds element in sorted order.
first()	Returns smallest element.
last()	Returns largest element.
higher(E e)	Smallest element > given element.
lower(E e)	Largest element < given element.
ceiling(E e)	Smallest element ≥ given element.
floor(E e)	Largest element ≤ given element.
subSet(E from, E to)	Returns elements between range exclusive of E to
subSet(E from,boolean, E to, boolean)	Returns elements between range whether inclusive or not decided by boolean values
headSet(E to)	Elements less than given element excluding given element.
headSet(E to, boolean)	Elements less than given element including given element if boolean set to true, excludes if boolean is given as false.
tailSet(E from)	Elements greater than or equal to given element including given element.
tailSet(E from,boolean)	Elements greater than given element excluding given element if boolean Is set to false, if set to true it includes given element also
descendingSet()	Elements in reverse order.

```
        //Methods
        System.out.println(set.first()); //returns smallest number since treeset is sorted
        System.out.println(set.last()); //returns largest number

        System.out.println(set.headSet(6)); //returns elements less than or left of 6(excluding if present in set)
        System.out.println(set.headSet(6, true)); //returns elements less than 6 including 6 when boolean argument is
given true, excludes if given false

        System.out.println(set.tailSet(20)); //returns elements greater than 20 including 20
        System.out.println(set.tailSet(20, false)); //returns elements greater than 20 excluding 20 if boolean set to
false

        System.out.println(set.descendingSet());//gives descending order sorted set

        System.out.println(set.subSet(6, 75));// returns subset form 6 to 75(exclusive)
        System.out.println(set.subSet(2, false, 75, true)); //returns subset from 2 to 75 - whether inclusive or not
is based on boolean argument given

        System.out.println(set.ceiling(7)); //returns smallest element i.e greater than or equal to 7 - //20 is the
smallest element which is greater than 7
        System.out.println(set.floor(3)); //returns largest element i.e smaller than or equal to 3 - //2 is the
largest which is smaller than 3

        System.out.println(set.higher(2)); //returns the 1st higher value than given element 2 even if given element
is present //6
        System.out.println(set.lower(7)); //returns the 1st lower value than given element 7 even if given element is
present or not //6

        System.out.println(set);
```

---

9. Advantages

- Maintains sorted order automatically.
- Provides efficient range search.
- Logarithmic time for basic operations.

---

10. Disadvantages

- Slower than HashSet for basic add/search/remove due to tree operations.
- No index-based access.
- Higher memory overhead due to tree nodes.

WHEN TO USE TREESET?

- Want sorted order , no duplicate entries, range operations
- Insertion-O(log n), Accessing - O(log n) - because tree has diff levels with many nodes as level increases

---

11. TreeSet vs HashSet vs LinkedHashSet

Feature	TreeSet (Red-Black Tree)	HashSet (Hash Table)	LinkedHashSet (Hash Table + Linked List)
Ordering	Sorted	No order	Insertion order preserved
Null elements	Only one, but restricted	One allowed	One allowed
Time complexity	O(log n)	O(1) avg, O(n) worst	O(1) avg, O(n) worst
Memory usage	Higher	Medium	Higher

---

12. Common Interview Traps

- **Q:** Why is insertion O(log n) in TreeSet?
  A: Because it's backed by a balanced binary search tree (Red-Black Tree), which requires traversal and possible rebalancing.
- **Q:** Can you store heterogeneous objects in a TreeSet?
  A: No, all elements must be mutually comparable, otherwise ClassCastException occurs.

- **Q:** Why is null insertion restricted?
  A: Comparison with null causes NullPointerException in natural ordering.

---

13. Example Code

java
CopyEdit

```
import java.util.*;
public class TreeSetExample {
   public static void main(String[] args) {
      TreeSet<Integer> ts = new TreeSet<>();
      ts.add(10);
      ts.add(5);
      ts.add(20);
      ts.add(15);
System.out.println(ts);  // [5, 10, 15, 20]
      System.out.println(ts.first());  // 5
      System.out.println(ts.last());   // 20
      System.out.println(ts.higher(10)); // 15
      System.out.println(ts.lower(10));  // 5
      System.out.println(ts.ceiling(15)); // 15
      System.out.println(ts.floor(14));   // 10
   }
}
```

TreeSet – All Possible Interview Questions, MCQs, Coding Questions with Answers

A) Theory Questions with Answers

1. What is a TreeSet in Java?
A TreeSet is a collection that implements the NavigableSet interface and stores unique elements in a **sorted** order. Internally, it uses a **Red-Black Tree** for ordering and search efficiency.

2. Does TreeSet allow duplicate elements?
No, duplicates are not allowed. If you attempt to insert a duplicate, the insertion is ignored.

3. What is the internal data structure of TreeSet?
A **self-balancing Red-Black Tree**.

4. What is the default sorting order in TreeSet?
Natural ascending order according to the element's compareTo() method.

5. How can we sort TreeSet in descending order?
Use the descendingSet() method or provide a custom Comparator during construction.

6. Does TreeSet allow null elements?
- **Java 6 and below** → Allows one null element if inserted first.
- **Java 7+** → Throws NullPointerException if null is inserted.

7. Is TreeSet synchronized?
No, but it can be synchronized using:

java
CopyEdit
Set s = Collections.synchronizedSet(new TreeSet());

8. What is the time complexity of basic operations in TreeSet?
- **add(E)** → O(log n)
- **remove(Object)** → O(log n)
- **contains(Object)** → O(log n)

9. Difference between HashSet and TreeSet?
- **HashSet** → Unordered, O(1) operations, uses HashMap internally.
- **TreeSet** → Ordered, O(log n) operations, uses TreeMap internally.

10. Can TreeSet store heterogeneous objects?
No, if elements are not mutually comparable, it throws ClassCastException.

11. How is iteration order defined in TreeSet?
It follows the **sorted order** of elements (either natural or custom comparator).

12. How do you create a TreeSet with a custom sorting order?

java
CopyEdit
TreeSet<Integer> ts = new TreeSet<>((a, b) -> b - a);
This sorts elements in **descending order**.

13. What happens if you modify an object after adding it to TreeSet?
If the modification changes the ordering, TreeSet's ordering can become inconsistent, leading to undefined behavior.

14. How is TreeSet different from LinkedHashSet?
- **TreeSet** → Sorted order, O(log n)
- **LinkedHashSet** → Insertion order, O(1)

15. Is TreeSet backed by TreeMap?
Yes, internally TreeSet is implemented using a TreeMap where elements are stored as keys and a constant dummy object is stored as the value.

B) MCQs with Answers

1. Which interface does TreeSet implement?
a) List
b) Set

c) NavigableSet ✅
d) Map

2. Time complexity of add() in TreeSet?
a) O(1)
b) O(log n) ✅
c) O(n)
d) O(log log n)

3. TreeSet allows null values in Java 8?
a) Yes, always
b) Only first null
c) No ✅
d) Depends on Comparator

4. What is the internal data structure of TreeSet?
a) Hash table
b) Linked list
c) Red-Black Tree ✅
d) Binary Heap

5. What happens when adding a non-comparable object to TreeSet without comparator?
a) Added successfully
b) Throws ClassCastException ✅
c) Throws NullPointerException
d) Ignores the object

C) Coding Questions with Answers

1. Create a TreeSet of integers and print them in descending order.

java
CopyEdit
```
TreeSet<Integer> ts = new TreeSet<>();
ts.add(10);
ts.add(5);
ts.add(20);
System.out.println(ts.descendingSet()); // [20, 10, 5]
```

2. Find the greatest element ≤ a given element in TreeSet.

java
CopyEdit
```
TreeSet<Integer> ts = new TreeSet<>(Arrays.asList(10, 20, 30, 40));
System.out.println(ts.floor(25)); // 20
```

3. Find the smallest element ≥ a given element.

java
CopyEdit
```
System.out.println(ts.ceiling(25)); // 30
```

4. Remove and return the smallest element.

```
java
CopyEdit
System.out.println(ts.pollFirst()); // Removes and returns 10
```

5. Remove and return the largest element.

```
java
CopyEdit
System.out.println(ts.pollLast()); // Removes and returns 40
```

6. Iterate TreeSet in reverse order.

```
java
CopyEdit
for(Integer num : ts.descendingSet()) {
   System.out.println(num);
}
```

7. Create a TreeSet with custom comparator (length of string).

```
TreeSet<String> ts = new TreeSet<>((a, b) -> a.length() - b.length());
ts.add("apple");
ts.add("banana");
ts.add("kiwi");
System.out.println(ts); // [kiwi, apple, banana]
```

8. Store Employee objects in TreeSet sorted by salary.

```
java
CopyEdit
class Employee {
   String name;
   int salary;
   Employee(String n, int s) { name = n; salary = s; }
}
TreeSet<Employee> ts = new TreeSet<>((e1, e2) -> e1.salary - e2.salary);
ts.add(new Employee("John", 5000));
ts.add(new Employee("Jane", 7000));
```

9. Use subSet() to get range of elements.

```
java
CopyEdit
TreeSet<Integer> ts = new TreeSet<>(Arrays.asList(10, 20, 30, 40, 50));
System.out.println(ts.subSet(20, 50)); // [20, 30, 40]
```

10. Find common elements of two TreeSets.

```
java
CopyEdit
TreeSet<Integer> a = new TreeSet<>(Arrays.asList(1, 2, 3, 4));
TreeSet<Integer> b = new TreeSet<>(Arrays.asList(3, 4, 5, 6));
a.retainAll(b);
```

```
System.out.println(a); // [3, 4]
```

HashSet in Java

==========

1. Introduction

- **Definition**: HashSet is a part of the Java Collections Framework that implements the **Set** interface.
- **Key property**: Stores **unique elements** (no duplicates allowed).
- **Package:** java.util
- **Implements:** Set interface
- **Internal Data Structure:** Backed by a **HashMap** (keys = elements, values = a constant dummy object).
- **Duplicates:** ❌ Not allowed (only one null allowed).
- **Order:** ❌ No guarantee of insertion and sort order (insertion order not preserved).
- **Thread-safety:** ❌ Not synchronized (use Collections.synchronizedSet() for thread-safe version).
- **Null Handling**: Allows **only one** null element.
- **Default capacity:** 16
- Resizing will happen based on load factor- 0.75 of 16 memory locations, if we add data after 75% of 16 memory locations, resizing will happen when we try to add 13th element.
- No indexing, internal data structure : hash table along with hash function(formula is confidential)
- **Memory location in HashSet is called as bucket location**

🔹 Hash Function & Data Insertion in HashSet

Purpose of Hash Function : Takes i/p and gives o/p as address

- A hash function takes an object's data and produces an integer value called **hash code** (hashCode() in Java).
- The same object should always return the same hash code during a program's execution (unless its state changes in a way that affects equality).

Mapping to Buckets

- The **hash code** is then processed using a modulo operation with the **capacity** of the underlying array (hash table) to determine the **bucket index**.
- Formula:
  index = hashCode(object) & (capacity - 1)
  (bitwise AND is used instead of modulo for performance).

Handling Collisions

- If two objects have the same hash code **or** map to the same bucket index, this is a **collision**.
- Java's HashSet handles this using **LinkedList (pre-Java 8)** or **balanced tree (Red-Black Tree, post-Java 8)** inside that bucket for faster lookups in high-collision scenarios.

Adding an Element

- Step 1: Calculate hash code using hashCode().
- Step 2: Find bucket index from hash code.
- Step 3: Check if the element already exists in that bucket (using equals() method).
- Step 4: If not found, insert it into the bucket (either in linked list form or tree structure).

Uniqueness Guarantee

- HashSet uses both hashCode() and equals() to ensure that **duplicate elements** are not stored

Rehashing

- When the **load factor** (size / capacity) exceeds the threshold (default 0.75), the table's capacity is **doubled**, and all elements are **rehashed** to new bucket positions.

Performance

- Average time complexity for add, remove, and contains is **O(1)**, but in worst-case collision scenarios, it can degrade to **O(log n)** (since Java 8).

---

2. Class Declaration & Hierarchy

```
public class HashSet<E> extends AbstractSet<E>
   implements Set<E>, Cloneable, Serializable
```

Hierarchy:

```
        Collection (Interface)
             ↑
           Set (Interface)
             ↑
       AbstractSet (Abstract Class) extends
             ↑
          HashSet implements Set interface
```

---

3. Key Features

No Duplicates: Automatically checks using equals() and hashCode().
Allows heterogeneous data.
Fast Performance: Constant-time complexity for add, remove, contains — **O(1)** average case.
Allows null: Only one null element.
Backed by HashMap: Each element is a **key** in the map.
No order of insertion - Unordered: Iteration order may differ from insertion order.
Internal DS : Hashtable with hash functions
Load Factor: Default **0.75** — when 75% full, capacity doubles.
Default capacity - 16
Constructors - 5

---

4. Constructors

- **Default constructor**:

```
HashSet<Integer> set = new HashSet<>();
```

- **With initial capacity**:

```
HashSet<Integer> set = new HashSet<>(50);
```

- **With initial capacity & load factor**:

```
HashSet<Integer> set = new HashSet<>(50, 0.80f);
```

- **From another collection**:

```
HashSet<Integer> set = new HashSet<>(Arrays.asList(1,2,3));
```

- **Boolean dummy(for dummy values since it follows hash map):**

HashSet(int initialCapacity, float loadFactor, boolean dummy)

---

5. Internal Working

Hashing Mechanism:
- On adding an element, hashCode() is computed.
- Element is placed into a **bucket** based on hash value.
- If two elements have the same hash, **equals()** decides uniqueness.

HashMap Relation:
- Internally: private transient HashMap<E,Object> map;

- Dummy object value: private static final Object PRESENT = new Object();

Collision Handling:

- Uses **Linked List** or **Balanced Tree (Red-Black Tree)** in buckets (Java 8+).

---

6. Time Complexity (Average Case)

Add → O(1)
Remove → O(1)
Contains → O(1)
Iteration → O(n)

Worst case (all hash collisions): O(n).

---

7. Important Methods

add(E e) – Adds element if not already present.
remove(Object o) – Removes if present.
contains(Object o) – Checks presence.
isEmpty() – Checks if set is empty.
size() – Returns number of elements.
clear() – Removes all elements.
iterator() – Returns iterator for traversal.

Accessing the elements in Hashset:

1. For loop - not possible
2. For each loop

```
for(Object e: h) {
        System.out.println(e);
    }
```

3. Iterator -

```
Iterator cursor = h.iterator();
while(cursor.hasNext()) {
    System.out.println(cursor.next());
}
```

4. ListIterator - not possible
5. DescendingIterator - not possible- since no fixed order for hash set

---

8. Example

```
import java.util.*;
public class HashSetExample {
   public static void main(String[] args) {
      HashSet<String> set = new HashSet<>();
      set.add("Apple");
      set.add("Banana");
      set.add("Apple"); // duplicate ignored
      set.add(null);

      for (String s : set) {
         System.out.println(s);
      }
   }
}
```

Output (unordered):

```
null
Apple
Banana
```

---

9. Null Handling

- First null is accepted.
- Further null insertions ignored.
- HashSet checks null specially before hashing.

---

10. Advantages

Fast performance for basic operations.
No duplicates — automatically enforced.
Allows one null.

11. Limitations

Unordered — no predictable iteration order.
Not thread-safe — use Collections.synchronizedSet() for thread-safety.
Poor performance in high collision scenarios.

---

12. Thread Safety

- Not synchronized.
- Make synchronized:

```
Set<Integer> syncSet = Collections.synchronizedSet(new HashSet<>());
```

---

13. When to Use HashSet?

- **Best for Insertion -** O(1) - since Hash function will generate the hash code, it will go and sit in that location
- Accessing - O(1)
- No insertion order - (to overcome we can use LinkedHashSet)
- Random o/p
- When you need **fast lookups** and **don't care about order**.
- When you need **unique elements - no duplicate**.

---

14. Differences from Other Sets

Feature	HashSet	LinkedHashSet	TreeSet
Order	No order	Maintains insertion order	Sorted order (natural/custom)
Speed	Fastest	Slightly slower	Slowest
Null Allowed	One null	One null	No null (except comparator allows)
Internal DS	HashMap	LinkedHashMap	TreeMap (RB Tree)

A) THEORY QUESTIONS WITH ANSWERS (HASHSET)

1. What is a HashSet in Java?

- **Answer:** A HashSet is a collection class in Java that implements the Set interface, backed by a HashMap. It stores **unique elements only**, does **not allow duplicates**, and **does not maintain insertion order**.

2. Internal data structure of HashSet?

- **Answer:** Internally uses a HashMap to store elements as keys with a constant dummy value (PRESENT). Elements are stored in hash table buckets based on their hash code.

3. How does HashSet ensure uniqueness?

- **Answer:**
  - Computes **hashCode()** for the element to find the bucket.
  - Uses **equals()** to check if the element already exists.
  - If both hashCode and equals indicate equality, duplicate is rejected.

4. Does HashSet maintain insertion order?

- **Answer:** No. Order depends on hash codes and may change when elements are added/removed.

5. Can HashSet store null?

- **Answer:** Yes, one null element is allowed.

6. Is HashSet synchronized?

- **Answer:** No, it's **not thread-safe**. Must be synchronized manually via Collections.synchronizedSet().

7. Time complexity of operations in HashSet?

- **Answer:**
  Add → **O(1)** average, **O(n)** worst case (hash collisions).
  Remove → **O(1)** average.
  Contains → **O(1)** average.

8. Difference between HashSet and TreeSet?

- **Answer:**
  - **HashSet:** Unordered, faster (O(1)), allows null.
  - **TreeSet:** Sorted, slower (O(log n)), no null in natural ordering.

9. How HashSet handles collisions?

- **Answer:** Uses **chaining** (linked list or balanced tree in modern Java) in the bucket.

10. Why does HashSet use HashMap internally instead of array?

- **Answer:** HashMap provides efficient hashing, collision handling, and uniqueness checks.

11. What happens when load factor is exceeded?

- **Answer:** Capacity is increased (doubled) and all elements are **rehashed** into new buckets.

12. Can HashSet have heterogeneous objects?

- **Answer:** Yes, but mixing different types can lead to **ClassCastException** during comparisons.

13. How are elements stored internally?

- **Answer:** As keys in a HashMap with a constant dummy value (PRESENT = new Object()).

14. When to use HashSet?

- **Answer:** When you need **unique elements** and **fast lookup/add/remove** without caring about order.

15. Difference between HashSet and LinkedHashSet?

- **Answer:**
  - **HashSet:** No ordering.
  - **LinkedHashSet:** Maintains **insertion order** using a linked list along with hashing.

16. Why is equals() important in HashSet?

- **Answer:** Used to check element equality along with hashCode(). Without it, duplicates may appear.

17. How does HashSet handle null's hashCode()?

- **Answer:** Special handling in HashMap to store null in bucket 0.

18. Is iteration order of HashSet predictable?

- **Answer:** No, changes with resizing or internal rearrangement.

19. How to iterate through HashSet?

- **Answer:** Using Iterator, enhanced for-loop, or Java Streams.

20. Why HashSet is faster than List for search?

- **Answer:** List search is **O(n)**, HashSet lookup is **O(1)** average.

B) CODING QUESTIONS WITH ANSWERS

1. Program to remove duplicates from a list using HashSet

```
java
CopyEdit
import java.util.*;
class RemoveDuplicates {
   public static void main(String[] args) {
      List<Integer> list = Arrays.asList(1,2,3,2,4,1);
      Set<Integer> set = new HashSet<>(list);
      System.out.println(set);
   }
}
```

2. Program to find union of two sets

```
java
CopyEdit
Set<Integer> set1 = new HashSet<>(Arrays.asList(1,2,3));
Set<Integer> set2 = new HashSet<>(Arrays.asList(3,4,5));
set1.addAll(set2);
System.out.println("Union: " + set1);
```

3. Program to find intersection of two sets

```
java
CopyEdit
Set<Integer> set1 = new HashSet<>(Arrays.asList(1,2,3));
Set<Integer> set2 = new HashSet<>(Arrays.asList(2,3,4));
set1.retainAll(set2);
System.out.println("Intersection: " + set1);
```

4. Program to find difference between two sets

```
java
CopyEdit
Set<Integer> set1 = new HashSet<>(Arrays.asList(1,2,3,4));
Set<Integer> set2 = new HashSet<>(Arrays.asList(3,4,5));
set1.removeAll(set2);
System.out.println("Difference: " + set1);
```

5. Check if two sets are equal

```
java
CopyEdit
Set<Integer> set1 = new HashSet<>(Arrays.asList(1,2,3));
Set<Integer> set2 = new HashSet<>(Arrays.asList(3,2,1));
System.out.println(set1.equals(set2)); // true
```

6. Program to convert HashSet to array

java
CopyEdit
```
Set<String> set = new HashSet<>(Arrays.asList("A", "B", "C"));
String[] arr = set.toArray(new String[0]);
System.out.println(Arrays.toString(arr));
```

7. Program to clone a HashSet

java
CopyEdit
```
HashSet<String> set = new HashSet<>();
set.add("Java");
set.add("Python");
HashSet<String> clone = (HashSet<String>) set.clone();
System.out.println(clone);
```

C) MCQs WITH ANSWERS

1. HashSet allows duplicates?
a) Yes
b) No ✅

2. HashSet maintains order?
a) Yes
b) No ✅

3. Which class is used internally by HashSet?
a) HashTable
b) HashMap ✅
c) TreeMap
d) LinkedList

4. Default initial capacity of HashSet?
a) 8
b) 16 ✅
c) 32
d) 64

5. Default load factor of HashSet?
a) 0.5
b) 0.75 ✅
c) 1.0
d) 0.25

6. Time complexity of add() in HashSet (average case)?
a) O(n)
b) O(1) ✅
c) O(log n)
d) O(n log n)

7. Which exception thrown if null element inserted multiple times in HashSet?

a) NullPointerException
b) No Exception ✅

8. Can HashSet contain heterogeneous objects?
a) Yes ✅
b) No

9. Which interface does HashSet implement?
a) Set ✅
b) List
c) Map
d) Collection only

10. Method to check if HashSet contains an element?
a) search()
b) contains() ✅
c) find()
d) has()

LinkedHashSet :

1. Overview

- **Package:** java.util
- **Inheritance:**
  java.lang.Object
  ↳ java.util.AbstractCollection
  ↳ java.util.AbstractSet
  ↳ java.util.HashSet
  ↳ java.util.LinkedHashSet extends hashSet
- **Implements:** Set<E>, Cloneable, Serializable
- **Introduced in:** Java 1.4
- **Internal Data Structure:**
  - **Hash table** (like HashSet) **+ Doubly Linked List** (for predictable iteration order).
- **Key Feature:** Maintains **insertion order** (or access order if configured).
- No duplicates allowed
- Heterogeneous data , one null is allowed

2. How It Works

- Internally, LinkedHashSet extends **HashSet** but uses a special **LinkedHashMap** to store elements.
- Each entry in the hash table is part of a **doubly linked list** that connects all entries in their insertion order.
- Two pointers are maintained for each node:
  before → points to previous entry.
  after → points to next entry.
- This allows iteration in **the same order in which elements were inserted**.

3. Characteristics

No Duplicates

- Like all Sets, it rejects duplicate elements.

Allows Null
- Only **one null element** allowed.

Order Maintained
- Elements are returned in **insertion order** during iteration.

Performance
- Slightly slower than HashSet due to linked list overhead.

Access Order Option
- If using the LinkedHashSet's parent LinkedHashMap constructor, you can set accessOrder = true to order elements based on last access.

4. Constructors

LinkedHashSet()
Creates an empty set with default capacity (16) and load factor (0.75).
LinkedHashSet(int initialCapacity)
LinkedHashSet(int initialCapacity, float loadFactor)
LinkedHashSet(Collection<? extends E> c)

5. Common Methods
(Inherited from HashSet & AbstractSet, but maintain order during iteration)

boolean add(E e)
- Adds element if not already present.
- Returns true if added, false otherwise.
- Average Time Complexity: **O(1)**.

boolean remove(Object o)
- Removes element if present.
- Average Time Complexity: **O(1)**.

boolean contains(Object o)
- Checks if element exists.
- Average Time Complexity: **O(1)**.

void clear()
- Removes all elements.

Iterator<E> iterator()
- Returns an iterator maintaining **insertion order**.

int size()
- Returns element count.

Object clone()
- Returns shallow copy.

6. Time Complexity
(Average Case — with good hash function)

Insertion (add) → O(1)
Deletion (remove) → O(1)
Search (contains) → O(1)
Iteration → O(n) (in insertion order)

7. Internal Working Flow (Insertion Example)
When you do:
```
LinkedHashSet<String> set = new LinkedHashSet<>();
set.add("A");
set.add("B");
set.add("C");
```
"A"
- hashCode() calculated → index in hash table.

- New entry created in table and added to linked list tail.

"B"
- Hash index calculated.
- New entry linked after "A" in linked list.

"C"
- Linked after "B".

🔹 **Iteration Order** → A → B → C.

8. Advantages over HashSet
- Predictable iteration order.
- Useful in caches, ordered sets, and where element sequence matters.

9. Disadvantages
- Slightly higher memory overhead than HashSet (due to doubly linked list).
- Slightly slower operations than HashSet.

10. Example Code
```
import java.util.LinkedHashSet;
public class Example {
   public static void main(String[] args) {
      LinkedHashSet<String> set = new LinkedHashSet<>();
      set.add("Apple");
      set.add("Banana");
      set.add("Orange");
      set.add("Apple"); // duplicate, ignored
for (String fruit : set) {
         System.out.println(fruit);
      }
   }
}
```
Output:
Apple
Banana
Orange

11. Real-World Uses
- Caching (LRU Cache implementation with access order).
- Maintaining unique items while preserving order.
- Ordered menus, playlists, recent-history lists.

12. Key Differences from HashSet

Feature	HashSet	LinkedHashSet
Order	No	Maintains insertion order
Performance	Faster	Slightly slower
Memory Usage	Less	More (due to linked list)
Internal DS	Hash table	Hash table + doubly linked list

LINKED HASHSET — ALL POSSIBLE INTERVIEW Q&A

A) Theory Questions with Answers

1. What is LinkedHashSet in Java?
A LinkedHashSet is a **HashSet** implementation that maintains **insertion order** using a **doubly-linked list** running through all its entries.

- Introduced in **Java 1.4**.
- Extends HashSet and implements Set interface.

2. Difference between HashSet and LinkedHashSet?

- **HashSet:** No order guarantee.
- **LinkedHashSet:** Maintains insertion order.
- Internal structure: Hash table + doubly-linked list in LinkedHashSet.

3. How does LinkedHashSet maintain insertion order?

- Each entry in the underlying HashMap is linked to the **previous** and **next** entry through a **doubly-linked list**.
- The linked list preserves the order in which elements were inserted.

4. Is LinkedHashSet synchronized?

- No. Needs external synchronization using Collections.synchronizedSet() if used in multi-threaded context.

5. Can LinkedHashSet store null values?

- Yes, only **one** null value is allowed (because it's a Set).

6. What is the load factor and capacity in LinkedHashSet?

- Same as HashMap: Default capacity = 16, load factor = 0.75.
- Load factor controls resizing.

7. Internal working of LinkedHashSet?

- Backed by LinkedHashMap internally.
- On insertion, a hash is calculated using hashCode() → index determined → stored in hash table bucket.
- Additionally, a doubly-linked list pointer is updated to maintain insertion order.

8. When should you use LinkedHashSet over HashSet?

- When you need uniqueness **and** predictable iteration order.

9. Is LinkedHashSet faster than HashSet?

- Almost same performance for most operations.
- Slightly more memory usage and small overhead due to linked list pointers.

10. Time Complexity for operations in LinkedHashSet:

- add() → O(1) average
- remove() → O(1) average
- contains() → O(1) average
- iteration → O(n)

11. Can LinkedHashSet contain heterogeneous objects?

- Yes, but **not recommended** because equals() and hashCode() consistency might break.

12. How does resizing happen in LinkedHashSet?

- When size exceeds capacity × load factor, capacity is doubled.
- All elements are rehashed and linked list is rebuilt.

13. How is iteration different from HashSet?

- HashSet: Unpredictable order.

- LinkedHashSet: Order exactly as inserted.

14. How to remove duplicates from a list using LinkedHashSet?
- Add all list elements to a LinkedHashSet → automatically removes duplicates while preserving order.

15. What is the difference between insertion order and sorted order?
- Insertion order: Order of elements as they were added (LinkedHashSet).
- Sorted order: Elements arranged according to comparator or natural order (TreeSet).

B) Coding Questions with Answers

1. Remove duplicates from ArrayList using LinkedHashSet
```
import java.util.*;
class Main {
   public static void main(String[] args) {
      List<Integer> list = Arrays.asList(1, 2, 3, 2, 1, 4);
      LinkedHashSet<Integer> set = new LinkedHashSet<>(list);
      System.out.println(set); // [1, 2, 3, 4]
   }
}
```

2. Maintain insertion order of unique strings
```
LinkedHashSet<String> set = new LinkedHashSet<>();
set.add("Java");
set.add("Python");
set.add("C++");
set.add("Java"); // duplicate ignored
System.out.println(set); // [Java, Python, C++]
```

3. Convert LinkedHashSet to Array
```
LinkedHashSet<String> set = new LinkedHashSet<>();
set.add("One");
set.add("Two");
String[] arr = set.toArray(new String[0]);
System.out.println(Arrays.toString(arr));
```

4. Iterate through LinkedHashSet
```
LinkedHashSet<Integer> set = new LinkedHashSet<>(Arrays.asList(1, 2, 3));
for (Integer num : set) {
   System.out.println(num);
}
```

5. Intersection of two LinkedHashSets
```
LinkedHashSet<Integer> set1 = new LinkedHashSet<>(Arrays.asList(1, 2, 3, 4));
LinkedHashSet<Integer> set2 = new LinkedHashSet<>(Arrays.asList(3, 4, 5, 6));
set1.retainAll(set2);
System.out.println(set1); // [3, 4]
```

6. LinkedHashSet to remove duplicates from a String
```
String str = "programming";
LinkedHashSet<Character> set = new LinkedHashSet<>();
for (char c : str.toCharArray()) {
   set.add(c);
}
```

```
StringBuilder sb = new StringBuilder();
for (char c : set) sb.append(c);
System.out.println(sb.toString()); // progamin
```

C) MCQs with Answers

1. LinkedHashSet maintains which order?
a) Random
b) Sorted
c) Insertion ✅
d) None

2. Underlying data structure of LinkedHashSet:
a) HashTable
b) HashMap + LinkedList ✅
c) TreeMap
d) Array

3. Can LinkedHashSet store multiple nulls?
a) Yes
b) No ✅

4. Time complexity of contains() in LinkedHashSet:
a) O(n)
b) O(1) ✅
c) O(log n)
d) O(n log n)

5. Default load factor of LinkedHashSet:
a) 0.5
b) 0.75 ✅
c) 1.0
d) 0.25

6. Is LinkedHashSet synchronized?
a) Yes
b) No ✅

7. LinkedHashSet internally uses which class?
a) HashMap
b) LinkedHashMap ✅
c) TreeMap
d) None

Vector in Java - ArrayList Replacement back then

---

1. Introduction

- **Vector** is a **legacy class** in Java, introduced in **JDK 1.0**.

- It is part of the **java.util** package.
- Implements:
  - **List** interface (ordered, allows duplicates)
  - **RandomAccess** (fast index-based access)
  - **Cloneable**
  - **Serializable**
- **Underlying Data Structure: Resizable array** (like ArrayList).
- **Key Feature: Synchronized** → thread-safe for multiple threads.
- **Drawback:** Synchronized methods make it **slower** than ArrayList in single-threaded contexts.

---

2. Class Hierarchy

java.lang.Object
↳ java.util.AbstractCollection<E>
↳ java.util.AbstractList<E>
↳ java.util.Vector<E>
↳ java.util.Stack<E>

3. Key Features

Resizable Array — Grows automatically when capacity exceeded.
Synchronized — All public methods are synchronized.
Allows Duplicates & Nulls — Can store multiple same elements and null values.
Indexed Access — Elements accessed via **0-based index**.
Legacy Class — Before Collections Framework, later retrofitted to implement List.

4. Constructors

Vector()
- Default capacity: **10** elements.

Vector(int initialCapacity)
- Creates vector with given initial capacity.

Vector(int initialCapacity, int capacityIncrement)
- Creates vector with given initial capacity & custom growth size.

Vector(Collection<? extends E> c)
- Creates vector from another collection.

5. Capacity Growth

- If capacityIncrement is **> 0** → new capacity = old capacity + capacityIncrement.
- If capacityIncrement is **0** (default) → new capacity = old capacity × **2**.

6. Important Methods

Adding Elements

add(E e) — Appends element to end.
add(int index, E e) — Inserts at specific index.
addAll(Collection c) — Adds all from collection.

Accessing Elements

get(int index) — Retrieves element at index.
elementAt(int index) — Same as get() (legacy).

Updating Elements

set(int index, E e) — Replaces element.
setElementAt(E e, int index) — Legacy equivalent.

Removing Elements

remove(int index) — Removes element by index.
remove(Object o) — Removes first occurrence.
removeAllElements() — Clears all (legacy).

Capacity & Size

capacity() — Returns current capacity.
ensureCapacity(int minCapacity) — Increases capacity if needed.
trimToSize() — Reduces capacity to current size.

Searching

contains(Object o) — Checks presence.
indexOf(Object o) — Returns index of first occurrence.
lastIndexOf(Object o) — Returns index of last occurrence.

Iteration

iterator() — Modern iterator.
elements() — Legacy Enumeration.

7. Time Complexity

Access by Index (get) → **O(1)** (RandomAccess)
Insert at End → **O(1)** (Amortized)
Insert at Middle → **O(n)** (Shift elements)
Remove at Middle → **O(n)** (Shift elements)
Search by Value → **O(n)**

8. Internal Working

- Uses an **Object[] array** internally.
- Index starts from **0**.
- On insertion when capacity is full:
  New array created with increased capacity.
  Old elements copied into the new array.
  New element added.

9. Advantages

- Thread-safe.
- Allows random access like an array.
- Can store null & duplicates.
- Can be easily converted to ArrayList.

10. Disadvantages

- Slower than ArrayList due to synchronization.
- Legacy API → newer code often prefers ArrayList.

11. When to Use

- Use when:
  - Multiple threads need to modify the list.
  - Legacy code requires it.
- Otherwise, prefer **ArrayList** + manual synchronization.

12. Example Code

java
CopyEdit

```
import java.util.*;
public class VectorExample {
   public static void main(String[] args) {
      Vector<String> vec = new Vector<>(2, 3); // capacity=2, increment=3
      vec.add("A");
      vec.add("B");
      vec.add("C"); // capacity grows from 2 → 5
```

```
System.out.println("Vector Elements: " + vec);
      System.out.println("Capacity: " + vec.capacity());
      System.out.println("First Element: " + vec.firstElement());
      System.out.println("Last Element: " + vec.lastElement());
   }
}
```

---

1. THEORY INTERVIEW QUESTIONS (WITH ANSWERS)

Q1. What is Vector in Java?
Answer:
Vector is a **legacy class** in Java that implements the **List interface**, is **synchronized**, and stores elements in a **resizable array**. It allows **duplicate and null elements** and provides **random access** to elements via index.

Q2. How is Vector different from ArrayList?

Feature	**Vector**	**ArrayList**
Synchronization	**Yes** (thread-safe)	**No**
Performance	Slower in single-threaded env	Faster
Legacy Status	Legacy (since JDK 1.0)	Modern (JDK 1.2)
Growth	Double capacity (default)	50% increase (default)
Iteration	Enumeration, Iterator, ListIterator	Iterator, ListIterator
	No Race Condition	Race Condition

Q3. What is the default capacity of a Vector?

- **10** elements (if no initial capacity is specified).

Q4. How does capacity grow in a Vector?

- **Default:** Capacity doubles when exceeded.
- **Custom:** If constructed with (initialCapacity, capacityIncrement), capacity grows by capacityIncrement instead of doubling.

Q5. Is Vector ordered?

- Yes. Elements are stored **in insertion order**.

Q6. Can Vector store null values?

- Yes. Vector can store **multiple null values**.

Q7. Why is Vector considered a legacy class?

- It was introduced **before the Collections Framework** (JDK 1.0).
- It was later **retrofitted** to implement List in JDK 1.2.

Q8. How do you iterate over a Vector?
Example:

java
CopyEdit

```
Vector<String> v = new Vector<>();
v.add("A");
v.add("B");
// Using Iterator
Iterator<String> it = v.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
// Using Enumeration (legacy)
Enumeration<String> e = v.elements();
while (e.hasMoreElements()) {
   System.out.println(e.nextElement());
}
```

Q9. Is Vector fail-fast?

- **Iterator/ListIterator** → Fail-fast (throws ConcurrentModificationException if modified while iterating).
- **Enumeration** → **Not** fail-fast (can iterate even if modified).

Q10. Can Vector be made unsynchronized?

- Yes, using:

java
CopyEdit
List list = Collections.synchronizedList(new ArrayList<>());
But this creates an **unsynchronized alternative** rather than modifying Vector itself.

2. MCQs ON VECTOR

Q1. What is the default capacity of a Vector in Java?
A) 5
B) 8
C) 10
D) 16
✅ **Answer:** C) 10

Q2. Which interface does Vector NOT implement?
A) List
B) RandomAccess
C) Cloneable
D) SortedSet
✅ **Answer:** D) SortedSet

Q3. Which method removes all elements from a Vector?
A) deleteAll()
B) removeAll()
C) clear()
D) removeAllElements()
✅ **Answer:** D) removeAllElements() (Legacy) and clear() (Modern)

Q4. Which iteration method in Vector is **not fail-fast**?
A) Iterator
B) ListIterator
C) Enumeration
D) Stream

✅ **Answer:** C) Enumeration

Q5. If a Vector has capacity 10 and you add the 11th element without setting capacityIncrement, what will be the new capacity?
A) 11
B) 15
C) 20
D) 21
✅ **Answer:** C) 20 (doubles)

3. CODING QUESTIONS ON VECTOR

Q1. Reverse a Vector

```
java
CopyEdit
import java.util.*;
public class ReverseVector {
   public static void main(String[] args) {
      Vector<Integer> v = new Vector<>();
      v.add(1); v.add(2); v.add(3);

      Collections.reverse(v);
      System.out.println(v); // [3, 2, 1]
   }
}
```

Q2. Remove duplicates from Vector

```
java
CopyEdit
import java.util.*;
public class RemoveDuplicates {
   public static void main(String[] args) {
      Vector<Integer> v = new Vector<>(Arrays.asList(1,2,2,3,4,4));
      Set<Integer> set = new LinkedHashSet<>(v);
      v.clear();
      v.addAll(set);
      System.out.println(v); // [1, 2, 3, 4]
   }
}
```

Q3. Find the maximum element in a Vector

```
java
CopyEdit
import java.util.*;
public class MaxVector {
   public static void main(String[] args) {
      Vector<Integer> v = new Vector<>(Arrays.asList(10, 20, 5, 30));
      int max = Collections.max(v);
      System.out.println(max); // 30
   }
}
```

Q4. Iterate Vector using Enumeration

```
java
CopyEdit
import java.util.*;
public class EnumerationExample {
   public static void main(String[] args) {
      Vector<String> v = new Vector<>();
      v.add("A"); v.add("B");
Enumeration<String> e = v.elements();
      while (e.hasMoreElements()) {
         System.out.println(e.nextElement());
      }
   }
}
```

Q5. Custom capacity increment demonstration

```
java
CopyEdit
import java.util.*;
public class CapacityIncrement {
   public static void main(String[] args) {
      Vector<Integer> v = new Vector<>(2, 3); // cap=2, increment=3
      v.add(1);
      v.add(2);
      v.add(3); // capacity increases by 3 → 5
      System.out.println(v.capacity()); // 5
   }
}
```


📜 **Legacy Classes in Java (Complete Notes)**

==========

1. Introduction

- **Legacy classes** are classes from **earlier versions of Java** (before Java 2 / JDK 1.2) that were later **retrofitted into the Collections Framework**.
- These classes are **part of java.util package** but **not part of the modern Collection interfaces hierarchy** at first — they were later adapted.
- Legacy classes are **synchronized** by default (thread-safe), but synchronization makes them **slower** than modern alternatives.
- Mostly replaced by **ArrayList**, **HashMap**, **ArrayDeque**, etc., in modern Java.

2. List of Main Legacy Classes

Vector
Stack
Hashtable
Properties (extends Hashtable)
Dictionary (abstract class — parent of Hashtable)
Enumeration (legacy iterator interface)

3. Detailed Notes

A) Vector

- **Package:** java.util
- **Implements:** List, RandomAccess, Cloneable, Serializable
- **Internal Structure: Dynamic array**
- **Thread-safety:** ✅ Synchronized methods (slower than ArrayList)
- **Default Capacity:** 10
- **Growth Formula:** If capacity exceeded → new capacity = old capacity × 2
- **Null Values:** ✅ Allowed multiple times
- **Insertion Order:** ✅ Maintained
- **Access Time Complexity:**
  Access (get/set): **O(1)**
  Insert at end: **O(1)** (amortized)
  Insert/remove in middle: **O(n)**

Important Methods:

add(E e)
add(int index, E e)
get(int index)

remove(int index)
set(int index, E e)
size()
capacity()
clear()

B) Stack

- **Package:** java.util
- **Extends:** Vector
- **Implements:** LIFO (Last In First Out)
- **Thread-safety:** ✅ Inherited from Vector
- **Null Values:** ✅ Allowed
- **Insertion Order:** ✅ Maintained

Key Methods:

java
CopyEdit
push(E e) // Add element on top
pop() // Remove and return top element
peek() // Return top element without removing
search(Object o) // Position from top (1-based), or -1 if not found
empty() // Returns true if stack is empty

C) Hashtable

- **Package:** java.util
- **Extends:** Dictionary<K, V>
- **Implements:** Map<K, V>, Cloneable, Serializable
- **Internal Structure: Hash table**
- **Thread-safety:** ✅ All methods synchronized
- **Null Keys:** ❌ Not allowed
- **Null Values:** ❌ Not allowed
- **Initial Capacity:** 11 (by default)
- **Load Factor:** 0.75
- **Resizing:** capacity = (old capacity × 2) + 1
- **Order:** ❌ No insertion order maintained

Key Methods:

java
CopyEdit
put(K key, V value)
get(Object key)
remove(Object key)
containsKey(Object key)
containsValue(Object value)
size()
clear()
keys() // Returns Enumeration of keys
elements() // Returns Enumeration of values

D) Properties

- **Extends:** Hashtable<Object, Object>
- **Purpose:** To store string-based key-value pairs (usually for configuration)
- **Key Features:**

- Often used to store application settings.
- Can **read/write from .properties files** using load() and store() methods.

Example:

java
CopyEdit
Properties p = new Properties();
p.setProperty("username", "admin");
p.setProperty("password", "12345");
String user = p.getProperty("username");

E) Dictionary (Abstract Class)

- **Package:** java.util
- **Purpose:** Parent of Hashtable
- **Status:** Obsolete — replaced by Map interface.
- Defines abstract methods:

java
CopyEdit
V get(Object key)
V put(K key, V value)
V remove(Object key)
Enumeration<K> keys()
Enumeration<V> elements()

F) Enumeration (Interface)

- **Purpose:** Legacy way to iterate through collections before Iterator existed.
- **Methods:**

boolean hasMoreElements()
E nextElement()

- **Differences from Iterator:**
  Enumeration has **only read** capability — no remove().
  Not part of the Collection framework.
  Faster than Iterator but less safe.

Example:

```
Vector<Integer> v = new Vector<>();
v.add(1); v.add(2);
Enumeration<Integer> e = v.elements();
while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
}
```

PROPERTIES:

- Used for traversal Vector and Stack

---

4. Other Notes

- **Synchronized Nature:** All these legacy classes are **thread-safe** but slower compared to modern non-synchronized classes.
- **Modern Replacements:**
  - Vector → ArrayList
  - Stack → ArrayDeque
  - Hashtable → HashMap / ConcurrentHashMap

- Enumeration → Iterator / ListIterator

5. Quick Comparison Table (Legacy vs Modern)

Legacy Class	Modern Alternative	Thread-Safe?	Allows Null Key?	Maintains Order?
Vector	ArrayList	Yes	Yes	Yes
Stack	ArrayDeque	Yes	Yes	Yes
Hashtable	HashMap	Yes	No	No
Enumeration	Iterator	No	N/A	N/A

Collections Utility Class in Java (java.util.Collections)

==========

1. Introduction

- **java.util.Collections** is a **utility class** that contains **static methods** for working with **collections** (e.g., Lists, Sets, Maps).
- It **cannot be instantiated** — constructor is private.
- It provides **algorithms**, **wrappers**, and **helpers** for:
  - Searching
  - Sorting
  - Synchronizing
  - Immutable collections
  - Frequency counting
  - Reversing/shuffling
  - Creating empty or singleton collections

2. Key Points

Static Methods Only — you don't create objects of Collections.
Works **only** on objects that implement **Collection Framework interfaces**.
Supports both **mutable** and **immutable** views of collections.
Algorithms like **sort**, **binarySearch** work on **List** (not Set/Map).
Many methods throw **NullPointerException** if the collection contains null where comparisons are needed (e.g., sorting).

3. Commonly Used Methods

A) Sorting

```
Collections.sort(list);              // Natural order - sorts only same type, not if there's heterogeneous data
Collections.sort(list, comparator);  // Custom order
```

- Uses **Modified Merge Sort (TimSort)** for Lists.
- **Time Complexity:** O(n log n)
- Works on List, requires elements to be **Comparable** or have a Comparator.

B) Searching

java
CopyEdit

```
int index = Collections.binarySearch(list, key);
int index = Collections.binarySearch(list, key, comparator);
```

- **Precondition:** List must be **sorted** in the same order as search.
- Returns:

- >=0 → index found
- <0 → insertion point = (-insertionPoint - 1)

C) Min / Max

java
CopyEdit
Collections.min(collection);
Collections.max(collection);
Collections.min(collection, comparator);
Collections.max(collection, comparator);

- Finds smallest/largest element based on natural or custom order.

D) Reversing / Shuffling / Rotating

java
CopyEdit
Collections.reverse(list); // Reverses order in-place
Collections.shuffle(list); // Randomizes order
Collections.rotate(list, 2); // Shifts right by k positions

- shuffle() uses a **Random** object internally.
- rotate() uses array reversal trick.

E) Frequency & Disjoint

java
CopyEdit
Collections.frequency(collection, element); // Count occurrences
Collections.disjoint(c1, c2); // True if no common elements

F) Fill & Copy

java
CopyEdit
Collections.fill(list, value); // Replace all elements
Collections.copy(dest, src); // dest must be >= size of src

G) Unmodifiable Collections

java
CopyEdit
List<String> unmodifiableList = Collections.unmodifiableList(list);
Set<String> unmodifiableSet = Collections.unmodifiableSet(set);

- Any modification attempt → UnsupportedOperationException.

H) Synchronized Collections

java
CopyEdit
List<String> syncList = Collections.synchronizedList(new ArrayList<>());

- Makes a collection **thread-safe** using synchronized wrapper.
- Synchronizes each method call.

I) Singleton & Empty Collections

java
CopyEdit
```
Collections.singleton(element);        // Immutable set with 1 element
Collections.emptyList();               // Immutable empty list
Collections.emptySet();
Collections.emptyMap();
```
- Used for **constant collections** without overhead.

J) N Copies

java
CopyEdit
```
Collections.nCopies(count, element);
```
- Returns immutable list with same element repeated.

4. Internal Working Insights

Sorting
- Uses **TimSort** → hybrid of merge sort + insertion sort.
- Stable and adaptive.

Synchronization
- Wrapper object uses synchronized keyword for every method.

Unmodifiable
- Returns a wrapper that throws exceptions for modification methods.

Empty Collections
- Returns shared **singleton immutable objects** to avoid memory waste.

5. Time Complexities
- **sort()** — O(n log n)
- **binarySearch()** — O(log n)
- **reverse(), rotate(), shuffle()** — O(n)
- **min()/max()** — O(n)
- **frequency()** — O(n)
- **disjoint()** — O(n × m) in worst case.

6. Advantages
- Centralized utility methods for common collection operations.
- Reduces boilerplate code.
- Provides thread-safe & immutable wrappers easily.
- Highly optimized algorithms (TimSort, binarySearch).

7. Limitations
- Works **only** with Java Collections Framework.
- Immutable/unmodifiable collections are **shallow** — underlying objects can still be changed if they are mutable.
- Synchronization wrappers add overhead.
- null handling varies per method.

8. Interview Tip

Common Questions:
- Difference between **Collections** and **Collection** (interface vs utility class).
- Why does Collections.sort() work only on List, not Set?
- How binarySearch() works and preconditions.
- Thread-safety: Collections.synchronizedList() vs CopyOnWriteArrayList.

- Immutable vs Unmodifiable collections.

✅ **Example Code**

java
CopyEdit

```
import java.util.*;
public class CollectionsDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<>(Arrays.asList(5, 3, 8, 1));
Collections.sort(list);
      System.out.println("Sorted: " + list);
Collections.reverse(list);
      System.out.println("Reversed: " + list);
Collections.shuffle(list);
      System.out.println("Shuffled: " + list);
int index = Collections.binarySearch(list, 3);
      System.out.println("Index of 3: " + index);
System.out.println("Max: " + Collections.max(list));
      System.out.println("Frequency of 3: " + Collections.frequency(list, 3));
List<Integer> syncList = Collections.synchronizedList(list);
      System.out.println("Thread-safe list: " + syncList);
   }
}
```

A) Theory & Technical Interview Questions with Answers

What is the Collections class in Java?

- It's a **utility class** in java.util containing **static methods** for operating on collections (sorting, searching, shuffling, reversing, synchronization, etc.).
- It **cannot be instantiated** because it has a **private constructor**.
- Works with interfaces like **List, Set, Map** (indirectly for maps via collections view).

Difference between Collection interface and Collections class?

- Collection: Root interface for Java's collection hierarchy.
- Collections: Final utility class with static methods to operate on collections.

Why are all methods in Collections class static?

- So they can be used without creating an object.
- This makes them easy to use as helper functions.

How does Collections.sort() work internally?

- For List of objects implementing Comparable: uses **TimSort** (Hybrid of Merge Sort + Insertion Sort).
- Time complexity: **O(n log n)**.
- Stable sort: Yes.

What's the difference between Collections.sort() and List.sort()?

- Collections.sort(list) → available since Java 1.2, calls list.sort(null) internally (Java 8+).
- list.sort(comparator) → available since Java 8, more concise.

How does Collections.binarySearch() work?

- Works on **sorted lists** using **binary search algorithm**.
- Time complexity: **O(log n)**.
- Returns:
  - **index ≥ 0** → element found.
  - **negative value** → -(insertionPoint) - 1 if not found.

What does Collections.reverse() do?

- Reverses the order of elements in a list **in place** (no new list created).

What is Collections.shuffle() used for?

- Randomizes the order of list elements using **Fisher–Yates shuffle**.

What does Collections.unmodifiableList() do?

- Returns an **unmodifiable (read-only)** view of the given list.
- Any modification operation throws UnsupportedOperationException.

What is the difference between Collections.synchronizedList() and CopyOnWriteArrayList?

- Collections.synchronizedList() → wraps list with synchronized methods (thread-safe, but locks on every access).
- CopyOnWriteArrayList → thread-safe without locking reads (uses copy-on-write strategy).

What is Collections.singletonList()?
- Returns an immutable list containing **exactly one element**.

What is Collections.nCopies()?
- Returns an immutable list containing **n copies** of the same object reference.

Why can't you instantiate Collections class?
- It has a **private constructor** to prevent object creation; it's only meant to provide static methods.

How does Collections.max() and Collections.min() work?
- Iterates through elements using natural ordering or a provided comparator.
- Time complexity: **O(n)**.

What is the difference between Arrays and Collections utility classes?
- Arrays → works on arrays.
- Collections → works on collection objects (List, Set, etc.).

B) Coding Questions with Answers

1. Sort a list of integers in descending order
```
import java.util.*;
class Test {
   public static void main(String[] args) {
      List<Integer> list = Arrays.asList(5, 1, 8, 3);
      Collections.sort(list, Collections.reverseOrder());
      System.out.println(list); // [8, 5, 3, 1]
   }
}
```

2. Find maximum element in a list
```
List<Integer> nums = Arrays.asList(10, 20, 5, 30);
int max = Collections.max(nums);
System.out.println(max); // 30
```

3. Binary search for an element
```
List<Integer> nums = Arrays.asList(1, 3, 5, 7, 9);
int index = Collections.binarySearch(nums, 5);
System.out.println(index); // 2
```

4. Shuffle elements
```
List<String> list = Arrays.asList("A", "B", "C", "D");
Collections.shuffle(list);
System.out.println(list); // Random order
```

5. Reverse a list
```
List<Integer> nums = Arrays.asList(1, 2, 3, 4);
Collections.reverse(nums);
System.out.println(nums); // [4, 3, 2, 1]
```

6. Create a read-only list
```
List<String> list = Collections.unmodifiableList(Arrays.asList("X", "Y"));
System.out.println(list);
```

// list.add("Z"); // Throws UnsupportedOperationException

C) MCQs with Answers

Which package contains Collections class?
a) java.lang
b) java.util ✅
c) java.io
d) java.collection

Collections.sort() uses which algorithm?
a) QuickSort
b) MergeSort
c) TimSort ✅
d) HeapSort

Which method is used to make a list read-only?
a) Collections.unmodifiableList() ✅
b) Collections.freeze()
c) Collections.lockList()
d) Collections.readOnly()

Time complexity of Collections.binarySearch()?
a) O(n)
b) O(log n) ✅
c) O(1)
d) O(n log n)

Which method reverses a list?
a) Collections.reverse() ✅
b) Collections.rotate()
c) Collections.swap()
d) Collections.shuffle()

What happens if you modify an unmodifiable collection?
a) Nothing
b) Returns false
c) Throws UnsupportedOperationException ✅
d) Compiles but fails at runtime

📚 Generics in Java

1. What are Generics?

- **Definition**: Generics allow you to create **classes, interfaces, and methods** with **type parameters**, enabling type safety at compile-time.
- Introduced in **Java 5** to eliminate type casting and increase code reusability.
- Syntax:

java
CopyEdit
class Box<T> { T item; }

2. Why Use Generics?

- **Type Safety**: Prevents runtime ClassCastException.
- **Code Reusability**: Single class/method works for multiple data types.
- **Eliminates Explicit Casting**: No need to manually cast objects after retrieval.
- **Cleaner Code**: Self-documenting.

3. Generic Syntax

java
CopyEdit
class ClassName<T> { ... }
T – Type
E – Element (used in collections)
K – Key
V – Value
N – Number

4. Example

java
CopyEdit
List<String> list = new ArrayList<>();
list.add("Java");
// list.add(10); // Compile error
String name = list.get(0); // No casting needed

5. Multiple Type Parameters

java
CopyEdit
class Pair<K, V> {
  K key;
  V value;
}

6. Bounded Type Parameters

- **Upper Bound**: T extends Number
- **Lower Bound**: ? super Integer
- Example:

java
CopyEdit
public <T extends Number> void display(T num) {
  System.out.println(num);
}

7. Wildcards

- <?> — Unknown type
- <? extends T> — Upper bound

- <? super T> — Lower bound

java
CopyEdit
List<? extends Number> nums = new ArrayList<Integer>();
List<? super Integer> ints = new ArrayList<Number>();

8. Generic Methods

java
CopyEdit
public static <T> void printArray(T[] array) {
  for (T item : array) System.out.println(item);
}

9. Diamond Operator (<>)
- Introduced in **Java 7**.
- Allows compiler to infer type parameters automatically.

java
CopyEdit
List<String> list = new ArrayList<>(); // No need to write <String> twice

10. Restrictions of Generics
❌ No primitive types (use wrapper classes like Integer, Double).
❌ No static generic type parameters.
❌ No instance of generic type (new T()).
❌ Cannot create arrays of generic type (new T[]).

11. Generics in Collections Framework
- **List<T>**, **Set<T>**, **Map<K, V>**
- Example:

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

💡 **How Generics Work Internally**
- Generics use **Type Erasure**: Type info is **removed** at compile-time, replaced with **Object** (or bounded type).
- Example:

java
CopyEdit
List<Integer> list = new ArrayList<>();
Internally becomes:

java
CopyEdit
List list = new ArrayList(); // after type erasure

⚡ **All Possible Interview Questions with Answers — Generics & Diamond Operator**

Theory

What are Generics in Java?

→ Generics provide compile-time type safety and eliminate the need for casting by parameterizing types in classes, interfaces, and methods.

Why were Generics introduced?

→ To enable type safety, reduce runtime errors, and improve code reusability.

What is the Diamond Operator?

→ Introduced in Java 7, <> allows type inference for constructors, avoiding duplicate type declarations.

What is Type Erasure?

→ The process by which the compiler removes generic type information during compilation, replacing them with Object or bounded type.

Can Generics work with primitive types? Why?

→ No, only wrapper classes can be used because generics work with objects.

What are Bounded Type Parameters?

→ Generics can restrict types using extends (upper bound) or super (lower bound).

Difference between <? extends T> and <? super T>?

→ extends allows reading as T but not writing; super allows writing but not reading as T.

Can static methods use generic types?

→ Yes, but they must declare their own type parameters.

Can we overload a method with same erasure?

→ No, type erasure removes type parameters making them identical to the compiler.

Why is List<Object> not the same as List<?>?

→ List<Object> can store any object; List<?> is read-only without type casting.

MCQs

Generics were introduced in which Java version?

✅ Java 5

Diamond Operator introduced in:

✅ Java 7

What does List<?> mean?

✅ List of unknown type.

Can we create an array of generic type?

✅ No

What does <? super Integer> mean?

✅ Any type that is Integer or its superclasses.

Coding Questions

Q1: Generic Class

```
java
CopyEdit
class Box<T> {
   T item;
   void set(T item) { this.item = item; }
   T get() { return item; }
}
public class Main {
   public static void main(String[] args) {
      Box<String> box = new Box<>();
      box.set("Java");
      System.out.println(box.get());
   }
}
```

Q2: Generic Method

java
CopyEdit
```
public static <T> void printArray(T[] array) {
   for (T element : array) System.out.print(element + " ");
}
```
Q3: Upper Bound

java
CopyEdit
```
public <T extends Number> void show(T value) {
   System.out.println(value.doubleValue());
}
```
Q4: Wildcard Example

java
CopyEdit
```
public static void display(List<?> list) {
   for (Object obj : list) System.out.println(obj);
}
```

Custom Sorting in Java:

----

1. What is Custom Sorting?

- **Custom Sorting** means defining our own sorting logic instead of relying on Java's **natural ordering** (default order given by Comparable).
- Used when:
  - Sorting in **descending order**.
  - Sorting **by multiple fields** (e.g., sort employees by salary, then name).
  - Sorting objects that do **not implement Comparable**.

2. Approaches for Custom Sorting

(a) Using Comparable Interface

- Defines **natural order** of objects.
- Implemented by the class whose objects need sorting.
- Method to override:
  public int compareTo(T o)
- Returns:
  - **Negative** → Current object < given object - no swap
  - **Zero** → Equal - No swap
  - **Positive** → Current object > given object - Then swap

Example:
```
class Student implements Comparable<Student> {
   int age;
   String name;
   public Student(int age, String name) {
      this.age = age;
      this.name = name;
   }
   @Override
```

```
    public int compareTo(Student s) {
        return this.age - s.age; // ascending order by age
    }
}
```

```
package sorting;

import java.util.ArrayList;
import java.util.Collections;

class Employee implements Comparable<Employee>{ //we pass Employee object which is our base here
	private int id;
	private String name;
	private int salary;
	public int getId() {
		return id;
	}

	public Employee() {

	}

	public Employee(int id, String name, int salary) {
		super();
		this.id = id;
		this.name = name;
		this.salary = salary;
	}

	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getSalary() {
		return salary;
	}
	public void setSalary(int salary) {
		this.salary = salary;
	}

	@Override
	public String toString() {
		return id + " " + name + " " + salary;
	}

//	@Override
//	public int compareTo(Employee e2) {
//		Employee e1 = this;
//		if(e1.salary>e2.salary) { //we can get without getters even if salary is private because we are accessing in the same class its declared
//			return 1;
//		}else if (e2.salary>e1.salary) {
//			return -1;
//		}else {
//			return 0;
//		}
//	}

	//Since we already have compareTo for Integer we can use it to compare according to ids as below
//	@Override
//	public int compareTo(Employee e2) {
//		Employee e1 = this;
//
//		Integer id1 = e1.getId();
//		Integer id2 = e2.getId();
//
//		return id1.compareTo(id2);
//	}

	//Sorting with name
//	@Override
//	public int compareTo(Employee e2) {
//		Employee e1 = this;
//
//		String n1 = e1.getName();
//		String n2 = e2.getName();
//
//		return n1.compareTo(n2);
//	}

	//to sort employees based on length of their name
	@Override
	public int compareTo(Employee e2) {
		Employee e1 = this;

		Integer n1 = e1.getName().length();
		Integer n2 = e2.getName().length();

		return n1.compareTo(n2);
	}

	//since we cant change code in compareTo for each different comparing methods, 2nd option is comparator interface
}

public class CustomSorting  {

	public static void main(String[] args) {

		ArrayList<Integer> ar = new ArrayList<Integer>();
		ar.add(100);
		ar.add(200);

		System.out.println(ar);
```

```
            Collections.sort(ar); //This calls compareTo method from Integer class which takes single object parameter and is from Comparable Interface, val1>val2 -> +ve, val1<val2 -> -ve, val1==val2 -> 0
            //Every wrapper class has compareTo method for that particular object, compareTo for Integer, Long, String...

            //-------------------------------------------------------------

            //Our own class Object - complex object
            Employee e = new Employee(1, "Varshini C", 2400000);
            Employee em = new Employee(2, "Sanjay C", 2400000);

            ArrayList<Employee> emp = new ArrayList<Employee>();
            emp.add(em);
            emp.add(e);

            System.out.println(emp); //return address of e and em if toString() is not used

            //Now to sort the employees based then their salaries
            Collections.sort(emp); //we have to make our own compareTo , to that make Employee class implement functional Interface Comparable(where compareTo is abstract method) and override the compareTo method and make it our own implementation

            System.out.println(emp);

      }

}
```

(b) Using Comparator Interface

- Defines an **external** comparison logic.
- Used when:
  - We **cannot modify the original class**.
  - We need **multiple sorting criteria**.
- Method to override:
  public int compare(T o1, T o2)

Example:

```
import java.util.*;
class Student {
   int age;
   String name;
   public Student(int age, String name) {
      this.age = age;
      this.name = name;
   }
}
class SortByName implements Comparator<Student> {
   public int compare(Student a, Student b) {
      return a.name.compareTo(b.name);
   }
}
```

```
package sorting;

import java.util.Comparator;
import java.util.TreeSet;

class Sorting implements Comparator<Employee1>{

      @Override
      public int compare(Employee1 e1, Employee1 e2) {
            Integer id1 = e1.getId();
            Integer id2 = e2.getId();

            return id1.compareTo(id2);
      }

}

class Sorting1 implements Comparator<Employee1>{

      @Override
      public int compare(Employee1 e1, Employee1 e2) {
            Integer sal1 = e1.getSalary();
            Integer sal2 = e2.getSalary();

            return sal1.compareTo(sal2);
      }

}

//make sorting in descending order - i.e by multiplying by -1 : which will make +ve to -ve and -ve to +ve, then -ve will swap and , +ve will No swap
```

```
class SortingDesc implements Comparator<Employee1>{

	@Override
	public int compare(Employee1 e1, Employee1 e2) {
		Integer sal1 = e1.getSalary();
		Integer sal2 = e2.getSalary();

		return -1*sal1.compareTo(sal2);
	}

}

class SortingName implements Comparator<Employee1>{

	@Override
	public int compare(Employee1 e1, Employee1 e2) {
		String n1 = e1.getName();
		String n2 = e2.getName();

		return n1.compareTo(n2);
	}

}

//sorting so that if name is same of 2 employees then we should compare with id
class Sorting2 implements Comparator<Employee1>{

	@Override
	public int compare(Employee1 e1, Employee1 e2) {
		String n1 = e1.getName();
		String n2 = e2.getName();

		if(n1.equals(n2)) { //even if we use n1 == n2 still works the same because we care comparing both refs which is not
created using new keyword, hence in SCP - duplicates are not allowed
			Integer id1 = e1.getId();
			Integer id2 = e2.getId();

			return id1.compareTo(id2);
		}else {
			return n1.compareTo(n2);
		}
	}

}

//compare names, if same compare salaries, if same compare id
class SortingC implements Comparator<Employee1>{

	@Override
	public int compare(Employee1 e1, Employee1 e2) {
		String n1 = e1.getName();
		String n2 = e2.getName();

		if(n1.equals(n2)) { //even though salary is integer we are comparing using equals() because we are comparing object-
Integer not int
			Integer s1 = e1.getSalary();
			Integer s2 = e2.getSalary();

			if(s1.equals(s2)) {
				Integer i1 = e1.getId();
				Integer i2 = e2.getId();

				return i1.compareTo(i2);
			}else {
				return s1.compareTo(s2);
			}
		}else {
			return n1.compareTo(n2);
		}
	}
}

class Employee1 {
	private int id;
	private String name;
	private int salary;
	public int getId() {
		return id;
	}

	public Employee1() {

	}

	public Employee1(int id, String name, int salary) {
		super();
		this.id = id;
		this.name = name;
		this.salary = salary;
	}

	public void setId(int id) {
		this.id = id;
	}
```

```
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getSalary() {
		return salary;
	}
	public void setSalary(int salary) {
		this.salary = salary;
	}

	@Override
	public String toString() {
		return id + " " + name + " " + salary;
	}



	//since we can't change code in compareTo for each different comparing methods, 2nd option is function interface - comparator interface - compare() method which takes 2 parameters
	//we shouldn't use our main class to implement Comparator interface, we have to create separate class to implement it
}

public class CustomSorting2  {

	public static void main(String[] args) {

		//By default sorting order is ascending
		Sorting s = new Sorting(); // we have to say this is our sorting to tree set by passing s to tree set which will call internally compare which is inside Sorting.
		Sorting1 s1 = new Sorting1(); //salary sorting ,pass the ref to tree set
		SortingDesc s2 = new SortingDesc();
		SortingName s3 = new SortingName();
		Sorting2 s4 = new Sorting2();
		SortingC s5 = new SortingC();

		//Our own class Object - complex object
		Employee1 e = new Employee1(1, "Varshini C", 2400000);
		//Employee1 em = new Employee1(2, "Sanjay C", 2400000);
		Employee1 em = new Employee1(2, "Varshini C", 24000000); //when name is same and we compare based on name, then as per set properties , duplicate are not allowed and Will return only one

//		TreeSet<Employee1> emp = new TreeSet<Employee1>(s); //TreeSet(Comparator<? super E> comparator), s will be given to comparator
		//We can sort the employees based on salary and name also, for that we dont have to change the method like we did for Comparable - compareTo,
		//In this Comparator interface, we can implement other class to comparator and write custom sorting for Salary, or names. and pass that here

//		TreeSet<Employee1> emp = new TreeSet<Employee1>(s1);//sorting based on salary - if salary same returns only one since duplicates are not allowed.
//		TreeSet<Employee1> emp = new TreeSet<Employee1>(s2); //sorting salary but descending
//		TreeSet<Employee1> emp = new TreeSet<Employee1>(s3);//sorting based on name - if name same , will return 1 because no duplicates allowed
//		TreeSet<Employee1> emp = new TreeSet<Employee1>(s4);
		TreeSet<Employee1> emp = new TreeSet<Employee1>(s5); //sort if name same on salary, salary same compare ids
		emp.add(em);
		emp.add(e);

		System.out.println(emp); //return address of e and em if toString() is not used
		//Collections.sort(emp); //no need of sort since tree set will order itself
		//System.out.println(emp);

	}

}
----------------



package sorting;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.TreeSet;

public class CustomSortingArrayList  {

	public static void main(String[] args) {

		//By default sorting order is ascending
		Sorting s = new Sorting();
		Sorting1 s1 = new Sorting1();
		SortingDesc s2 = new SortingDesc();
		SortingName s3 = new SortingName();
		Sorting2 s4 = new Sorting2();
		SortingC s5 = new SortingC();

		Employee1 e = new Employee1(1, "Varshini C", 2400000);
		//Employee1 em = new Employee1(2, "Sanjay C", 2400000);
```

```
            Employee1 em = new Employee1(2, "Varshini C", 2400000);

            ArrayList<Employee1> emp = new ArrayList<Employee1>();
            emp.add(em);
            emp.add(e);

            //Since ArrayList has no constructor that takes Comparator as parameter,
            //we must use Collections.sort(emp, ) - which has constructor that takes Comparator - public static <T> void
sort(List<T> list, Comparator<? super T> c)
            Collections.sort(emp, s5); //sort method is overloaded

            System.out.println(emp);

        }

}
```

---

3. Comparator vs Comparable

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo(_)	compare(_ , _)
Sorting logic	Defined inside the class	Defined outside the class
Multiple sorting	No(natural sorting)	Yes(custom sorting)
Modifies class	Yes	No

4. Sorting in Reverse Order

Collections.sort(list, Collections.reverseOrder());
Or with Comparator:
list.sort((a, b) -> b.age - a.age);

5. Sorting Collections of Primitives

- Primitives (like int[], double[]) → Use Arrays.sort() with lambda for custom order.

```
Integer[] arr = {5, 2, 8};
Arrays.sort(arr, (a, b) -> b - a); // Descending
```

6. Java 8+ Shortcuts

- **Lambda Expressions**:

```
list.sort((a, b) -> a.name.compareTo(b.name));
```

- **Method References**:

```
list.sort(Comparator.comparing(Student::getName));
```

- **Chaining Comparators**:

```
list.sort(Comparator.comparing(Student::getAge)
                .thenComparing(Student::getName));
```

7. Internal Working

- Collections.sort() uses **TimSort** (hybrid of merge sort + insertion sort) with **O(n log n)** time complexity.
- Custom sorting works by repeatedly calling the provided comparator or compareTo().

💡 **All Possible Interview Questions with Answers – Custom Sorting**

Theory Questions

What is custom sorting in Java?

Custom sorting means defining our own sorting logic using Comparable or Comparator instead of default natural ordering.

Difference between Comparable and Comparator?

- Comparable: Defines natural ordering, modifies class, one sorting logic.
- Comparator: External class, multiple sorting logics possible.

When to use Comparable over Comparator?

- Use Comparable when you can modify the class and only need one default sorting order.

Can we sort objects without modifying their class?

- Yes, by using Comparator.

What happens if compare() returns zero?

- Objects are considered equal in terms of sorting.

What is the time complexity of Collections.sort()?

- **O(n log n)** using TimSort.

Can we sort null values?

- Yes, using Comparator.nullsFirst() or Comparator.nullsLast().

Is custom sorting stable?

- Yes, TimSort used in Java is stable.

Coding Questions

Sort list of strings in reverse order:

```
List<String> list = Arrays.asList("Apple", "Banana", "Cherry");
list.sort(Collections.reverseOrder());
```

Sort employees by salary, then name:

```
list.sort(Comparator.comparing(Employee::getSalary)
                .thenComparing(Employee::getName));
```

Sort using lambda (descending age):

```
list.sort((a, b) -> b.age - a.age);
```

Sort an array of integers in custom order:

```
Integer[] arr = {4, 1, 7};
Arrays.sort(arr, (a, b) -> a % 2 - b % 2); // even first
```

Sort using anonymous Comparator:

```
Collections.sort(list, new Comparator<Student>() {
   public int compare(Student a, Student b) {
      return a.name.compareTo(b.name);
   }
});
```

MCQs

Which package contains Comparator?

a) java.lang
b) java.util ✅
c) java.io
d) java.sort

Which method is used in Comparator?

a) compareTo()
b) compare() ✅
c) sort()
d) compareList()

What is the default sorting order for strings?

a) Reverse order
b) Lexicographical ✅
c) Length-based
d) None

Which sorting algorithm is used in Collections.sort()?

a) QuickSort
b) MergeSort
c) TimSort ✅
d) HeapSort

Can a Comparator be a lambda?

a) No
b) Yes ✅

HASHMAP

==========

1. Introduction

- **HashMap** is a part of java.util package.
- Implements **Map<K, V>** interface.
- Stores **key-value** pairs.
- **No duplicate keys** allowed (values can be duplicated).
- **Null keys**: only 1 allowed; **null values**: multiple allowed.
- **Order**: **Not ordered** — no guarantee of insertion order.
- **Internal data structure**: **Array of buckets** + **Linked List/Balanced Tree** (Red-Black Tree for high collisions).
- **Default load factor**: 0.75
- **Default capacity**: 16

2. Internal Working

- **Hashing mechanism** is used to determine where to store key-value pairs.
- Steps when inserting (put(K key, V value)):
  hashCode() of the key is computed.
  Hash is processed to reduce collisions (e.g., (hash & (n - 1))).
  Index = (hash & (capacity - 1))
  If bucket empty → new node is inserted.
  If collision → compares keys via equals().
  - If key exists → replace value.
  - Else add new node at bucket end (Java 7: linked list head insertion, Java 8+: tail insertion + possible tree conversion).

  If bucket's linked list size > 8 → convert to **Red-Black Tree**.

3. Load Factor & Rehashing

- **Load Factor**: Ratio of number of elements to capacity.
  - Default: **0.75** → when exceeded, capacity doubled (rehashing).
- **Rehashing**: Recomputing bucket positions for existing entries.

4. Performance

- **Average case**:
  - get() → O(1)
  - put() → O(1)
  - remove() → O(1)
- **Worst case**: O(log n) (due to tree structure in bucket).

5. Key Points

- Uses **hashCode()** & **equals()** for comparison.
- **Fail-fast** iterator — throws ConcurrentModificationException if modified during iteration.
- **Not synchronized** — use Collections.synchronizedMap() or ConcurrentHashMap for thread safety.
- **Iteration order** is random.

6. When to Use

- When you need fast lookups and don't care about ordering.
- When you can handle non-thread-safety externally.

- For caching, indexing, and associative mapping.

INTERVIEW QUESTIONS – THEORY

Q1. Difference between HashMap and Hashtable?
- HashMap: Not synchronized, allows 1 null key, multiple null values, faster.
- Hashtable: Synchronized, no null key, no null values, slower.

Q2. How does HashMap handle collisions?
- Uses Linked List in each bucket.
- If list size exceeds 8 in Java 8 → converted to Red-Black Tree.

Q3. Why is load factor important?
- Controls when to rehash. Too low → memory waste, too high → performance drop.

Q4. Can HashMap store heterogeneous keys and values?
- Yes (if no generics used).

Q5. Why O(1) for get()?
- Because index is computed directly via hashing, avoiding iteration.

Q6. Difference between HashMap and LinkedHashMap?
- LinkedHashMap maintains insertion/access order; HashMap does not.

Q7. Is HashMap ordered?
- No. Iteration order can change when rehashing occurs.

Q8. Why equals() & hashCode() must be overridden?
- To ensure that logically equal keys are stored/retrieved properly.

Q9. What is treeification in HashMap?
- Converting bucket's linked list to Red-Black Tree for faster lookup when collisions are high.

Q10. Can we have duplicate keys in HashMap?
- No. Existing key will overwrite the old value.

MCQs

1. Default capacity of HashMap?
a) 8
b) 16 ✅
c) 32
d) 64

2. Default load factor of HashMap?
a) 0.5
b) 0.65
c) 0.75 ✅
d) 1.0

3. Which data structure is used for high collision buckets?
a) AVL Tree
b) Red-Black Tree ✅
c) B-Tree
d) Skip List

4. Time complexity of get() in average case?
a) O(1) ✅
b) O(n)
c) O(log n)
d) O(n log n)

5. Which interface does HashMap implement?
a) List
b) Set
c) Map ✅
d) Collection

CODING QUESTIONS WITH ANSWERS

1. Basic Usage

```
java
CopyEdit
import java.util.*;
public class HashMapExample {
   public static void main(String[] args) {
      HashMap<String, Integer> map = new HashMap<>();
      map.put("Apple", 2);
      map.put("Banana", 5);
      map.put("Orange", 3);
System.out.println("Map: " + map);
      System.out.println("Apple count: " + map.get("Apple"));
   }
}
```

2. Count Frequency of Words

```
java
CopyEdit
import java.util.*;
public class WordFrequency {
   public static void main(String[] args) {
      String[] words = {"apple", "banana", "apple", "orange", "banana", "apple"};
      Map<String, Integer> freq = new HashMap<>();
for (String w : words)
         freq.put(w, freq.getOrDefault(w, 0) + 1);
System.out.println(freq);
   }
}
```

3. Check Anagrams

```
java
CopyEdit
import java.util.*;
public class AnagramCheck {
   public static boolean isAnagram(String s1, String s2) {
      if (s1.length() != s2.length()) return false;
      Map<Character, Integer> map = new HashMap<>();
for (char c : s1.toCharArray())
         map.put(c, map.getOrDefault(c, 0) + 1);
for (char c : s2.toCharArray()) {
         if (!map.containsKey(c)) return false;
         map.put(c, map.get(c) - 1);
         if (map.get(c) == 0) map.remove(c);
      }
      return map.isEmpty();
   }
public static void main(String[] args) {
      System.out.println(isAnagram("listen", "silent")); // true
   }
}
```

4. Remove Duplicates from Array

```
java
CopyEdit
import java.util.*;
public class RemoveDuplicates {
   public static void main(String[] args) {
      int[] arr = {1, 2, 2, 3, 4, 4, 5};
      Map<Integer, Boolean> map = new HashMap<>();
for (int num : arr) map.put(num, true);
System.out.println(map.keySet());
   }
}
```

5. First Non-Repeating Character

```
java
CopyEdit
import java.util.*;
public class FirstNonRepeating {
   public static void main(String[] args) {
      String s = "swiss";
      Map<Character, Integer> map = new LinkedHashMap<>();
for (char c : s.toCharArray())
         map.put(c, map.getOrDefault(c, 0) + 1);
for (Map.Entry<Character, Integer> e : map.entrySet()) {
         if (e.getValue() == 1) {
            System.out.println("First non-repeating: " + e.getKey());
            break;
         }
      }
   }
}
```

LinkedHashMap

==========

1. Definition

- LinkedHashMap<K, V> is a **HashMap** implementation that maintains a **doubly linked list** running through its entries.
- It preserves **insertion order** (default) or **access order** (if enabled).
- Introduced in **Java 1.4**.

2. Key Features

- **Order**: Maintains predictable iteration order.
  - *Insertion order* → Order in which keys are inserted.
  - *Access order* → Least Recently Used (LRU) ordering if accessOrder = true.
- **Nulls**: Allows **one null key** and **multiple null values**.
- **Not synchronized** (must use Collections.synchronizedMap() for thread safety).

- **Performance**:
  - Lookup, insert, delete → **O(1)** average (same as HashMap).
  - Slightly slower than HashMap due to linked list overhead.
- **Duplicates**: Keys must be unique; values can be duplicated.

3. Internal Working

- **Underlying Data Structures**:
  - **Hash table** → for O(1) access.
  - **Doubly linked list** → to maintain iteration order.
- **Entry Node**: Each entry is a subclass of HashMap.Node<K, V> with before and after references for the linked list.
- **Hashing**: Same as HashMap → hash(key) % capacity.
- **Load Factor**: Default 0.75 (resize when 75% full).
- **Capacity**: Default 16.

4. Constructors

```
LinkedHashMap(); // default capacity 16, load factor 0.75, insertion order
LinkedHashMap(int initialCapacity);
LinkedHashMap(int initialCapacity, float loadFactor);
LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder); // true for access order
LinkedHashMap(Map<? extends K, ? extends V> m);
```

5. Special Use – LRU Cache

- By enabling accessOrder = true and overriding removeEldestEntry(), LinkedHashMap can be used as an **LRU Cache**.
- Example:

```
java
CopyEdit
LinkedHashMap<Integer, String> lruCache = new LinkedHashMap<>(16, 0.75f, true) {
   protected boolean removeEldestEntry(Map.Entry eldest) {
      return size() > 5; // remove oldest when size exceeds 5
   }
};
```

6. Differences with HashMap

Feature	HashMap	LinkedHashMap
Order	Unordered	Maintains order
Performance	Slightly faster	Slightly slower
Memory Usage	Lower	Higher (extra linked list)
Use Case	Fast lookups	Predictable iteration

All Possible Interview Questions + Answers – LinkedHashMap

Theory Questions

What is LinkedHashMap in Java?

- A HashMap with predictable iteration order, implemented using a doubly linked list along with a hash table.

How does LinkedHashMap maintain insertion order?

- Each entry is linked to its predecessor and successor in a doubly linked list.

How do you maintain access order instead of insertion order?

- Pass true for the accessOrder parameter in the constructor.

How does LinkedHashMap handle null keys and values?
- Allows one null key, multiple null values.

Is LinkedHashMap synchronized?
- No, it is not thread-safe.

Can LinkedHashMap be used as a cache?
- Yes, with accessOrder = true and overriding removeEldestEntry().

What is the complexity of get(), put(), and remove()?
- **O(1)** average time.

Does LinkedHashMap rehash like HashMap?
- Yes, when size exceeds capacity × load factor.

What is the difference between LinkedHashMap and TreeMap?
- LinkedHashMap preserves insertion/access order; TreeMap sorts keys.

Why is LinkedHashMap slower than HashMap?
- Due to maintaining a doubly linked list.

MCQs

LinkedHashMap preserves:
- a) Sorted order
- b) Insertion order ✅
- c) Random order
- d) None

Default accessOrder in LinkedHashMap is:
- a) true
- b) false ✅
- c) depends
- d) null

Which data structures does LinkedHashMap use internally?
- a) Hash table + Doubly Linked List ✅
- b) Array only
- c) Tree
- d) Queue

LinkedHashMap is:
- a) Synchronized
- b) Not synchronized ✅
- c) Always thread-safe
- d) None

Time complexity of get() in LinkedHashMap is:
- a) O(n)
- b) O(log n)
- c) O(1) ✅
- d) Depends

Coding Questions

Basic LinkedHashMap Example

java
CopyEdit
```
LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
map.put(1, "A");
map.put(2, "B");
map.put(3, "C");
System.out.println(map); // {1=A, 2=B, 3=C}
```

Access Order Example

java
CopyEdit
```
LinkedHashMap<Integer, String> map = new LinkedHashMap<>(16, 0.75f, true);
map.put(1, "A");
map.put(2, "B");
map.put(3, "C");
map.get(1);
System.out.println(map); // {2=B, 3=C, 1=A}
```

LRU Cache Implementation

java
CopyEdit
```
class LRUCache<K, V> extends LinkedHashMap<K, V> {
   private int capacity;
   LRUCache(int capacity) {
      super(capacity, 0.75f, true);
      this.capacity = capacity;
   }
   protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
      return size() > capacity;
   }
}
```

Iterating LinkedHashMap

java
CopyEdit
```
for (Map.Entry<Integer, String> entry : map.entrySet()) {
   System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

Removing Elements

java
CopyEdit
```
map.remove(2); // removes key 2
```

📌 TreeMap in Java

==========

1. Overview

- **Definition:** TreeMap is a **NavigableMap** implementation based on a **Red-Black Tree** (a self-balancing binary search tree).
- **Package:** java.util
- **Ordering:** Keys are stored in **sorted order** (natural ordering or custom comparator).
- **Null Handling:**
  - **Null Key:** ❌ Not allowed (throws NullPointerException if attempted).
  - **Null Values:** ✅ Allowed.
- **Duplicate Keys:** ❌ Not allowed; inserting an existing key replaces its value.

- **Thread-Safety:** ❌ Not synchronized. Use Collections.synchronizedSortedMap() for thread-safe version.
- **Internal Working:**
  - Keys are stored in a **Red-Black Tree**.
  - Balanced height → O(log n) time complexity for insert, delete, search.

2. Constructors

java
CopyEdit
TreeMap<K, V> map1 = new TreeMap<>(); // Natural ordering
TreeMap<K, V> map2 = new TreeMap<>(Comparator.reverseOrder()); // Custom ordering
TreeMap<K, V> map3 = new TreeMap<>(existingMap); // From another map
TreeMap<K, V> map4 = new TreeMap<>(anotherSortedMap); // From another SortedMap

3. Key Features

- **SortedMap Implementation:** Maintains ascending order by default.
- **NavigableMap Features:**
  - firstKey(), lastKey()
  - ceilingKey(k) → Smallest ≥ k
  - floorKey(k) → Largest ≤ k
  - higherKey(k) → Smallest > k
  - lowerKey(k) → Largest < k
- **Views:**
  - headMap(), tailMap(), subMap()
  - descendingMap()
- **Time Complexity:**
  - Insert: O(log n)
  - Search: O(log n)
  - Delete: O(log n)

4. Internal Working – Red-Black Tree

- Self-balancing BST with extra **color (Red/Black)** property.
- Balances itself on insert/delete to ensure **height ≈ $\log_2(n)$**.
- Guarantees performance even in worst-case.

5. Example Code

java
CopyEdit
```
import java.util.*;
public class TreeMapExample {
   public static void main(String[] args) {
      TreeMap<Integer, String> map = new TreeMap<>();
      map.put(3, "Banana");
      map.put(1, "Apple");
      map.put(2, "Cherry");
System.out.println(map); // {1=Apple, 2=Cherry, 3=Banana} - Sorted by key
   }
}
```

📌 **All Possible Interview Questions – TreeMap**

A) Theory Questions

What is a TreeMap in Java?

→ A NavigableMap implementation that stores keys in sorted order using a Red-Black Tree.

How does TreeMap maintain order?

→ Uses natural ordering or a custom comparator.

Can TreeMap store null keys?

→ No, null keys are not allowed.

Can TreeMap store null values?

→ Yes, multiple null values are allowed.

What is the time complexity of operations?

→ O(log n) for put, get, remove.

What is the difference between TreeMap and HashMap?

→ TreeMap is sorted (log n), HashMap is unsorted (O(1) average).

Is TreeMap synchronized?

→ No. Use Collections.synchronizedSortedMap() for synchronization.

How is TreeMap different from LinkedHashMap?

→ TreeMap sorts by keys, LinkedHashMap maintains insertion order.

When to use TreeMap?

→ When you need a sorted map or range queries.

How does Red-Black Tree work internally in TreeMap?

→ Balances tree using rotations and recoloring after insert/delete.

B) MCQs

1. TreeMap follows which sorting by default?

a) Insertion order

b) Natural order ✅

c) Reverse order

d) None

2. Time complexity of insertion in TreeMap:

a) O(1)

b) O(log n) ✅

c) O(n)

d) Depends

3. Null key allowed in TreeMap?

a) Yes

b) No ✅

c) Only once

d) Depends

4. Which internal data structure does TreeMap use?

a) Hash table

b) Red-Black Tree ✅

c) Array

d) Linked list

C) Coding Questions

1. Sort map by keys automatically:

java

CopyEdit

```
TreeMap<Integer, String> map = new TreeMap<>();
map.put(5, "Five");
map.put(1, "One");
map.put(3, "Three");
System.out.println(map); // {1=One, 3=Three, 5=Five}
```

2. Custom sorting by descending order:

```
java
CopyEdit
TreeMap<Integer, String> map = new TreeMap<>(Comparator.reverseOrder());
map.put(5, "Five");
map.put(1, "One");
map.put(3, "Three");
System.out.println(map); // {5=Five, 3=Three, 1=One}
```

3. Find floor and ceiling keys:

```
java
CopyEdit
TreeMap<Integer, String> map = new TreeMap<>();
map.put(10, "A");
map.put(20, "B");
map.put(30, "C");
System.out.println(map.floorKey(25)); // 20
System.out.println(map.ceilingKey(25)); // 30
```

4. SubMap Example:

```
java
CopyEdit
TreeMap<Integer, String> map = new TreeMap<>();
map.put(1, "A");
map.put(2, "B");
map.put(3, "C");
System.out.println(map.subMap(1, 3)); // {1=A, 2=B}
```

Feature	HashMap	LinkedHashMap	TreeMap
Ordering	No order – elements are stored in **hash table** order (unpredictable).	Maintains **insertion order** (or **access order** if enabled).	Maintains **sorted order of keys** (natural or custom Comparator).
Internal Structure	Array of buckets + Linked Lists / Red-Black Trees (since Java 8).	Extends HashMap + **doubly-linked list** across entries.	**Red-Black Tree**.
Time Complexity (Avg)	Insertion: **O(1)** Search: **O(1)** Deletion: **O(1)**	Insertion: **O(1)** Search: **O(1)** Deletion: **O(1)**	Insertion: **O(log n)** Search: **O(log n)** Deletion: **O(log n)**
Null Keys	**1 null key allowed**.	**1 null key allowed**.	**Null key not allowed** (NullPointerException).
Null Values	Multiple null values allowed.	Multiple null values allowed.	Multiple null values allowed.
Duplicates	Keys must be unique, values can repeat.	Same as HashMap.	Same as HashMap.
Thread-Safety	Not synchronized. Use Collections.synchronizedMap() for sync.	Same as HashMap.	Same as HashMap.
When to Use	Best for **fast lookups** without caring about order.	Best when **order of insertion or access order** is required along with	Best when **sorted order of keys** is required.

		fast lookups.	
Memory Overhead	Lowest memory usage.	Higher than HashMap (due to linked list).	Higher than both HashMap & LinkedHashMap (due to tree structure).
Iteration Order	Unpredictable.	Predictable (insertion/access order).	Sorted by keys.
Interfaces Implemented	Map, Cloneable, Serializable.	Map, Cloneable, Serializable.	Map, NavigableMap, SortedMap, Cloneable, Serializable.
Performance Impact of Resizing	Rehashing may occur when load factor exceeded.	Same as HashMap but also maintains linked list.	Tree balancing occurs automatically on inserts/deletes.
Best Use-Case	Quick key-based access where order doesn't matter.	Maintain predictable iteration order with fast access.	Maintain keys in sorted order for range queries & navigation methods.

ArrayList — Methods, Purpose, and Time Complexity

Method	Purpose	Time Complexity (Average)	Notes
add(E e)	Append element at end	O(1) amortized	O(n) if resize occurs
add(int index, E e)	Insert at specific index	O(n)	Shifts elements
addAll(Collection c)	Append all elements	O(m) amortized	m = size of c
addAll(int index, Collection c)	Insert collection at index	O(n + m)	Shifts existing
get(int index)	Access element by index	O(1)	Direct array access
set(int index, E e)	Replace element	O(1)	Direct array access
remove(int index)	Remove element by index	O(n)	Shifts elements
remove(Object o)	Remove first occurrence	O(n)	Search + shift
removeAll(Collection c)	Remove all elements in c	O(n × m) worst	Depends on c.contains
retainAll(Collection c)	Keep only elements in c	O(n × m) worst	Depends on c.contains
clear()	Remove all elements	O(n)	Nulls array slots
contains(Object o)	Check if element exists	O(n)	Linear search
indexOf(Object o)	First occurrence index	O(n)	Linear search
lastIndexOf(Object o)	Last occurrence index	O(n)	Linear search
size()	Number of elements	O(1)	Stores size variable
isEmpty()	Check if empty	O(1)	size == 0
iterator()	Returns iterator	O(1)	
listIterator()	Returns ListIterator	O(1)	
subList(int from, int to)	View of part of list	O(1)	Changes affect original
toArray()	Copy to Object[]	O(n)	
toArray(T[] a)	Copy to typed array	O(n)	

2. LinkedList — Methods, Purpose, Time Complexity

Method	Purpose	Time Complexity (Average)	Notes
add(E e)	Add at end	O(1)	Maintains tail pointer
add(int index, E e)	Insert at index	O(n)	Traverse to index
addFirst(E e)	Insert at front	O(1)	Head pointer update
addLast(E e)	Insert at end	O(1)	Tail pointer update
addAll(Collection c)	Add all at end	O(m)	m = size of c
addAll(int index, Collection c)	Insert all at index	O(n + m)	
get(int index)	Get element at index	O(n)	Sequential traversal
getFirst()	First element	O(1)	
getLast()	Last element	O(1)	
set(int index, E e)	Replace element	O(n)	
remove()	Remove first element	O(1)	
remove(int index)	Remove at index	O(n)	
removeFirst()	Remove first element	O(1)	
removeLast()	Remove last element	O(1)	
remove(Object o)	Remove first occurrence	O(n)	
removeAll(Collection c)	Remove all matching	O(n × m) worst	
retainAll(Collection c)	Keep only matching	O(n × m) worst	
clear()	Remove all	O(n)	
contains(Object o)	Check if exists	O(n)	
indexOf(Object o)	First occurrence index	O(n)	
lastIndexOf(Object o)	Last occurrence index	O(n)	

size()	Count elements	O(1)
isEmpty()	Check if empty	O(1)
iterator()	Sequential iterator	O(1)
descendingIterator()	Reverse order iterator	O(1)

3. Vector — Methods, Purpose, Time Complexity

Method	Purpose	Time Complexity	Notes
add(E e)	Append at end	O(1) amortized	Synchronized
add(int index, E e)	Insert at index	O(n)	
addElement(E e)	Append at end	O(1) amortized	Legacy method
addAll(Collection c)	Append all	O(m)	
capacity()	Current capacity	O(1)	
copyInto(Object[] anArray)	Copy elements	O(n)	
elementAt(int index)	Get element	O(1)	
firstElement()	First element	O(1)	
lastElement()	Last element	O(1)	
remove(int index)	Remove at index	O(n)	
remove(Object o)	Remove occurrence	O(n)	
removeAll(Collection c)	Remove all matching	O(n × m) worst	
removeAllElements()	Clear vector	O(n)	
set(int index, E e)	Replace element	O(1)	
setElementAt(E e, int index)	Legacy replace	O(1)	
size()	Count	O(1)	
trimToSize()	Reduce capacity	O(n)	

4. Stack — Methods, Purpose, Time Complexity

Method	Purpose	Time Complexity	Notes
push(E e)	Add element to top	O(1) amortized	Uses Vector add
pop()	Remove and return top	O(1)	Throws EmptyStackException
peek()	Return top element	O(1)	
empty()	Check if stack empty	O(1)	
search(Object o)	Position from top	O(n)	

4. PriorityQueue (java.util.PriorityQueue)

(implements Queue) — Min-heap

Method	Purpose	Time Complexity
add(E e)	Insert	**O(log n)**
offer(E e)	Insert	**O(log n)**
peek()	Get smallest element	**O(1)**
poll()	Remove smallest	**O(log n)**
remove(Object o)	Remove specific	**O(n)**
contains(Object o)	Check if present	**O(n)**
size()	Count	**O(1)**
clear()	Remove all	**O(n)**

5. ArrayDeque (java.util.ArrayDeque)

(implements Deque)

Method	Purpose	Time Complexity
addFirst(E e)	Insert at head	**O(1)** amortized
addLast(E e)	Insert at tail	**O(1)** amortized
offerFirst(E e)	Insert at head	**O(1)** amortized
offerLast(E e)	Insert at tail	**O(1)** amortized
removeFirst()	Remove head	**O(1)**
removeLast()	Remove tail	**O(1)**
pollFirst()	Remove head	**O(1)**
pollLast()	Remove tail	**O(1)**
peekFirst()	Get head	**O(1)**
peekLast()	Get tail	**O(1)**
size()	Count	**O(1)**
contains(Object o)	Search	**O(n)**
remove(Object o)	Remove match	**O(n)**

6. HashSet (java.util.HashSet)

(implements Set) — Hash table, no duplicates, unordered

Method	Purpose	Time Complexity
add(E e)	Insert	**O(1)** average, **O(n)** worst
remove(Object o)	Delete	**O(1)** average
contains(Object o)	Search	**O(1)** average
size()	Count	**O(1)**
clear()	Remove all	**O(n)**
iterator()	Iterate	**O(1)** creation

7. LinkedHashSet

(Maintains insertion order)

Method	Purpose	Time Complexity
(Same as HashSet)		*(Same complexities)*

8. TreeSet (java.util.TreeSet)

(implements NavigableSet) — Red-Black Tree

Method	Purpose	Time Complexity
add(E e)	Insert	**O(log n)**
remove(Object o)	Delete	**O(log n)**
contains(Object o)	Search	**O(log n)**
first()	Min element	**O(log n)**
last()	Max element	**O(log n)**
higher(E e)	Least greater than e	**O(log n)**
lower(E e)	Greatest less than e	**O(log n)**
subSet()	Range view	**O(log n)**
headSet()	Elements before	**O(log n)**
tailSet()	Elements after	**O(log n)**

9. HashMap (java.util.HashMap)

(Key-Value pairs)

Method	Purpose	Time Complexity
put(K key, V value)	Insert/update	**O(1)** avg, **O(n)** worst
get(Object key)	Retrieve	**O(1)** avg
remove(Object key)	Delete	**O(1)** avg
containsKey(Object key)	Check key	**O(1)** avg
containsValue(Object value)	Check value	**O(n)**
size()	Count	**O(1)**
clear()	Remove all	**O(n)**
keySet()	Keys view	**O(1)**
values()	Values view	**O(1)**
entrySet()	Entries view	**O(1)**

1 Hashtable

Method	Purpose	Time Complexity
put(K key, V value)	Inserts or updates value for the key	O(1) avg, O(n) worst
get(Object key)	Retrieves value for key	O(1) avg, O(n) worst
remove(Object key)	Removes key-value pair	O(1) avg, O(n) worst
containsKey(Object key)	Checks if key exists	O(1) avg
containsValue(Object value)	Checks if value exists	O(n)
isEmpty()	Checks if table is empty	O(1)
size()	Returns number of entries	O(1)
clear()	Removes all entries	O(n)
keys()	Returns enumeration of keys	O(1)
elements()	Returns enumeration of values	O(1)
clone()	Creates shallow copy	O(n)
rehash() *(protected)*	Resizes table	O(n)

2 LinkedHashMap

Method	Purpose	Time Complexity
put(K key, V value)	Inserts or updates value	O(1)
get(Object key)	Retrieves value for key	O(1)
remove(Object key)	Removes key-value mapping	O(1)

containsKey(Object key)	Checks if key exists	O(1)
containsValue(Object value)	Checks if value exists	O(n)
size()	Returns entry count	O(1)
clear()	Clears map	O(n)
isEmpty()	Checks if empty	O(1)
keySet()	Returns set of keys (in insertion order)	O(1)
values()	Returns collection of values (in insertion order)	O(1)
entrySet()	Returns set of entries	O(1)
forEach(BiConsumer)	Performs action for each entry	O(n)
replace(K, V)	Replaces value for key	O(1)
replace(K, oldV, newV)	Conditional replace	O(1)
replaceAll(BiFunction)	Replaces all values	O(n)
getOrDefault(K, defaultV)	Gets value or default	O(1)
putIfAbsent(K, V)	Inserts only if absent	O(1)
remove(K, V)	Removes entry if matches value	O(1)
compute(K, BiFunction)	Computes new value	O(1)
computeIfAbsent(K, Function)	Computes if absent	O(1)
computeIfPresent(K, BiFunction)	Computes if present	O(1)
merge(K, V, BiFunction)	Merges value	O(1)

3️⃣ TreeMap

Method	Purpose	Time Complexity
put(K key, V value)	Inserts or updates value (sorted order)	O(log n)
get(Object key)	Retrieves value for key	O(log n)
remove(Object key)	Removes key-value mapping	O(log n)
containsKey(Object key)	Checks if key exists	O(log n)
containsValue(Object value)	Checks if value exists	O(n)
firstKey()	Returns smallest key	O(log n)
lastKey()	Returns largest key	O(log n)
ceilingKey(K key)	Returns smallest ≥ given key	O(log n)
floorKey(K key)	Returns largest ≤ given key	O(log n)
higherKey(K key)	Returns next higher key	O(log n)
lowerKey(K key)	Returns next lower key	O(log n)
headMap(K toKey)	Returns view of keys < toKey	O(log n)
tailMap(K fromKey)	Returns view of keys ≥ fromKey	O(log n)
subMap(K fromKey, K toKey)	Returns view between keys	O(log n)
size()	Entry count	O(1)
clear()	Removes all	O(n)
isEmpty()	Checks if empty	O(1)
keySet()	Returns sorted keys	O(1)
values()	Returns values in sorted order	O(1)
entrySet()	Returns sorted entries	O(1)
replace(K, V)	Replaces value	O(log n)
replace(K, oldV, newV)	Conditional replace	O(log n)
replaceAll(BiFunction)	Replace all values	O(n)
putIfAbsent(K, V)	Adds if absent	O(log n)
remove(K, V)	Removes if matches value	O(log n)
getOrDefault(K, defaultV)	Gets value or default	O(log n)
compute, computeIfAbsent, computeIfPresent, merge		

📜 Master Java Collections & Maps Time Complexity Cheat Sheet

Class	Method	Purpose	Average Time Complexity	Worst Case
ArrayList	add(E)	Append element at end	O(1) amortized	O(n) (resize)
	add(int, E)	Insert at index	O(n)	O(n)
	get(int)	Get element at index	O(1)	O(1)
	set(int, E)	Replace element	O(1)	O(1)
	remove(int)	Remove by index	O(n)	O(n)
	remove(Object)	Remove by value	O(n)	O(n)
	contains(Object)	Search	O(n)	O(n)
	indexOf(Object)	Find index	O(n)	O(n)
	size()	Get size	O(1)	O(1)
	iterator()	Get iterator	O(1)	O(1)
LinkedList	add(E)	Append at end	O(1)	O(1)
	addFirst(E)	Insert at head	O(1)	O(1)
	addLast(E)	Insert at tail	O(1)	O(1)
	get(int)	Get element by index	O(n)	O(n)
	getFirst()	Head element	O(1)	O(1)
	getLast()	Tail element	O(1)	O(1)

	removeFirst()	Remove head	O(1)	O(1)
	removeLast()	Remove tail	O(1)	O(1)
	remove(int)	Remove by index	O(n)	O(n)
	contains(Object)	Search	O(n)	O(n)
Vector	add(E)	Append element	O(1) amortized	O(n)
	add(int, E)	Insert at index	O(n)	O(n)
	get(int)	Get element	O(1)	O(1)
	set(int, E)	Replace element	O(1)	O(1)
	remove(int)	Remove by index	O(n)	O(n)
	contains(Object)	Search	O(n)	O(n)
	capacity()	Capacity of vector	O(1)	O(1)
Stack	push(E)	Add to top	O(1) amortized	O(n) (resize)
	pop()	Remove from top	O(1)	O(1)
	peek()	Top element	O(1)	O(1)
	search(Object)	Search in stack	O(n)	O(n)
PriorityQueue	add(E)	Insert element	O(log n)	O(log n)
	offer(E)	Insert element	O(log n)	O(log n)
	peek()	Get min element	O(1)	O(1)
	poll()	Remove min element	O(log n)	O(log n)
	remove(Object)	Remove by value	O(n)	O(n)
	contains(Object)	Search	O(n)	O(n)
ArrayDeque	addFirst(E)	Add to head	O(1)	O(1)
	addLast(E)	Add to tail	O(1)	O(1)
	removeFirst()	Remove head	O(1)	O(1)
	removeLast()	Remove tail	O(1)	O(1)
	peekFirst()	First element	O(1)	O(1)
	peekLast()	Last element	O(1)	O(1)
	contains(Object)	Search	O(n)	O(n)
HashSet	add(E)	Add element	O(1)	O(n)
	remove(Object)	Remove element	O(1)	O(n)
	contains(Object)	Search	O(1)	O(n)
	size()	Size of set	O(1)	O(1)
LinkedHashSet	add(E)	Add element (maintains order)	O(1)	O(n)
	remove(Object)	Remove element	O(1)	O(n)
	contains(Object)	Search	O(1)	O(n)
TreeSet	add(E)	Insert element (sorted)	O(log n)	O(log n)
	remove(Object)	Remove element	O(log n)	O(log n)
	contains(Object)	Search	O(log n)	O(log n)
	first()	Smallest element	O(1)	O(1)
	last()	Largest element	O(1)	O(1)
HashMap	put(K, V)	Insert key-value	O(1)	O(n)
	get(K)	Get value	O(1)	O(n)
	remove(K)	Remove entry	O(1)	O(n)
	containsKey(K)	Check key	O(1)	O(n)
	containsValue(V)	Check value	O(n)	O(n)
	keySet()	Get all keys	O(1)	O(1)
LinkedHashMap	put(K, V)	Insert (maintains order)	O(1)	O(n)
	get(K)	Get value	O(1)	O(n)
	remove(K)	Remove entry	O(1)	O(n)
	containsKey(K)	Check key	O(1)	O(n)
	containsValue(V)	Check value	O(n)	O(n)
TreeMap	put(K, V)	Insert (sorted keys)	O(log n)	O(log n)
	get(K)	Get value	O(log n)	O(log n)
	remove(K)	Remove entry	O(log n)	O(log n)
	containsKey(K)	Check key	O(log n)	O(log n)
	firstKey()	Smallest key	O(1)	O(1)
	lastKey()	Largest key	O(1)	O(1)
Hashtable	put(K, V)	Insert	O(1)	O(n)
	get(K)	Get value	O(1)	O(n)
	remove(K)	Remove entry	O(1)	O(n)
	containsKey(K)	Check key	O(1)	O(n)
	containsValue(V)	Check value	O(n)	O(n)

Feature	**ArrayList**	**LinkedList**	**Vector**	**Stack**	**ArrayDeque**	**HashSet**	**LinkedHashSet**	**TreeSet**	**PriorityQueue**	**HashMap**	**LinkedHashMap**	**TreeMap**	**Hashtable**
Type	List	List, Deque	List	List (LIFO)	Deque	Set	Set	Set	Queue	Map	Map	Map	Map
Order Maintained?	Insertion order	Insertion order	Insertion order	Insertion order	Insertion order	No	Insertion order	Sorted order	Priority order	No	Insertion order	Sorted order	No
Duplicates	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Keys: No,	Keys: No,	Keys: No,	Keys: No,

Allowed?										Values: Yes	Values: Yes	Values: Yes	Values: Yes
Null Allowed?	Yes (multiple)	Yes (multiple)	Yes (multiple)	Yes (multiple)	Yes (multiple)	One null	One null	No	One null	One key null, many value nulls	One key null, many value nulls	No key null, many value nulls	No key null, no value null
Underlying DS	Dynamic array	Doubly linked list	Dynamic array (synchronized)	Vector (synchronized)	Resizable circular array	HashMap	LinkedHashMap	TreeMap	Binary heap	Hash table + linked list buckets	Hash table + linked list buckets	Red-Black Tree	Hash table
Time Complexity – Access	O(1)	O(n)	O(1)	O(1)	O(1)	—	—	O(log n)	O(1) (peek)	O(1)	O(1)	O(log n)	O(1)
Time Complexity – Insert	O(1) amortized	O(1) at ends, O(n) middle	O(1) amortized	O(1) amortized	O(1) amortized	O(1)	O(1)	O(log n)	O(log n)	O(1) amortized	O(1) amortized	O(log n)	O(1)
Time Complexity – Remove	O(n)	O(1) at ends, O(n) middle	O(n)	O(1)	O(1)	O(1)	O(1)	O(log n)	O(log n)	O(1)	O(1)	O(log n)	O(1)
Synchronized?	No	No	Yes	Yes	No	No	No	No	No	No	No	No	Yes
When to Use	Random access, frequent reads	Frequent inserts/deletes	Thread-safe ArrayList	Thread-safe LIFO	Double-ended queue	Unique unordered elements	Unique ordered elements	Unique sorted elements	Min/max retrieval	Key-value unordered	Key-value ordered	Key-value sorted	Thread-safe key-value

📌 Java PriorityQueue — Deep Notes

==========

1️⃣ Definition

- **PriorityQueue** is a class in java.util that implements the **Queue** interface.
- It is an **unbounded, ordered, heap-based** queue where elements are arranged according to **natural ordering** (Comparable) or a **Comparator** provided at construction.
- By default → **Min-Heap** (smallest element at the head).
- **Does not allow null elements**.
- **Duplicates are allowed**.
- **Not thread-safe** → use PriorityBlockingQueue for concurrent usage.

2️⃣ Hierarchy

java.lang.Object
  ↑
java.util.AbstractCollection<E> (class)
  ↑
java.util.AbstractQueue<E> (class)
  ↑
java.util.PriorityQueue<E> (class)
  ↳ implements Serializable
  ↳ implements Iterable<E>
  ↳ implements Collection<E>
  ↳ implements Queue<E>

Example code:

```
package priorityqueueDemo;

import java.util.PriorityQueue;
import java.util.Comparator;
import java.util.Iterator;
```

```java
public class PriorityQueueBasics {

    public static void main(String[] args) {

        // ================================
        // 1️⃣ Natural Ordering (Min-Heap)
        // ================================
        PriorityQueue<Integer> pq = new PriorityQueue<>(); // Default: min-heap

        // Adding elements
        pq.add(30);     // add() throws exception if fails
        pq.offer(10);   // offer() returns false if fails
        pq.add(20);

        System.out.println("Min-Heap PriorityQueue: " + pq); // Order not sorted, but head is smallest

        // ================================
        // 2️⃣ Accessing head element
        // ================================
        System.out.println("Peek (head): " + pq.peek()); // returns head without removing
        System.out.println("Poll (remove head): " + pq.poll()); // removes and returns head
        System.out.println("After poll: " + pq);

        // ================================
        // 3️⃣ Removing specific element
        // ================================
        pq.remove(30); // removes object, returns boolean
        System.out.println("After removing 30: " + pq);

        // ================================
        // 4️⃣ Checking size and contains
        // ================================
        System.out.println("Contains 20? " + pq.contains(20));
        System.out.println("Size: " + pq.size());

        // ================================
        // 5️⃣ Iterating over PriorityQueue
        // ================================
        pq.add(40);
        pq.add(5);
        pq.add(15);

        System.out.print("Iterating: ");
        Iterator<Integer> it = pq.iterator();
        while (it.hasNext()) {
            System.out.print(it.next() + " "); // Not guaranteed to be in order
        }
        System.out.println();

        // ================================
        // 6️⃣ Custom Ordering (Max-Heap)
        // ================================
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
        maxHeap.add(30);
        maxHeap.add(10);
        maxHeap.add(20);

        System.out.println("Max-Heap PriorityQueue: " + maxHeap);
        System.out.println("Poll from Max-Heap: " + maxHeap.poll());
        System.out.println("After poll (Max-Heap): " + maxHeap);

        // ================================
        // 7️⃣ Clearing the PriorityQueue
        // ================================
        pq.clear();
        System.out.println("After clear(), is empty? " + pq.isEmpty());
    }
}
```

3️⃣ Internal Working

- Internally uses a **resizable array** as a **binary heap**.
- **Binary heap properties**:
  Complete binary tree (filled level by level).
  Heap order property — root is the smallest (min-heap) or largest (max-heap if custom comparator).
- Elements are compared using:
  - **Natural ordering** via Comparable<T> (compareTo() method), or
  - Custom ordering via Comparator<T> passed to the constructor.
- Index calculation:
  - Parent index: (i - 1) / 2
  - Left child: 2*i + 1
  - Right child: 2*i + 2

4️⃣ Constructors

PriorityQueue() // default capacity 11, natural ordering
PriorityQueue(int initialCapacity)
PriorityQueue(int initialCapacity, Comparator<? super E> comparator)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)

5️⃣ Important Methods & Complexities

Method	Description	Time Complexity
boolean add(E e)	Inserts element, throws exception if fails	O(log n)
boolean offer(E e)	Inserts element, returns false if fails	O(log n)
E peek()	Retrieves head without removing	O(1)
E poll()	Retrieves and removes head	O(log n)
boolean remove(Object o)	Removes a single instance of the object	O(n)
boolean contains(Object o)	Checks if element exists	O(n)
int size()	Returns number of elements	O(1)
void clear()	Removes all elements	O(n)
Iterator<E> iterator()	Returns iterator (not sorted order)	O(n)

6️⃣ Properties & Limitations

- **Not sorted when iterated** — Iterator order is arbitrary, **only the head** follows priority rules.
- **Nulls not allowed**.
- **Capacity grows automatically**.
- Thread safety → **NOT synchronized**.
- Allows duplicates.
- Default = **Min Heap**; for Max Heap, use:
  PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());

7️⃣ When to Use

- When you need **fast retrieval of smallest/largest element** without sorting the entire list.
- Task scheduling, Dijkstra's algorithm, Huffman encoding, A* pathfinding, etc.

💡 Common Interview Questions

Theory Questions

What is the default ordering of PriorityQueue?

- Min-heap, natural ordering of elements.

How does PriorityQueue maintain order internally?

- Uses a binary heap with either Comparable or Comparator.

Is PriorityQueue synchronized?

- No; use PriorityBlockingQueue for thread-safe use.

Does PriorityQueue allow null elements?

- No.

Is iteration over PriorityQueue sorted?

- No, only the head is guaranteed to be the smallest/largest.

Difference between PriorityQueue and TreeSet?

- PriorityQueue allows duplicates, TreeSet doesn't.
- PriorityQueue is heap-based, TreeSet is a Red-Black Tree.

Coding Questions

Q1: Implement Max-Heap using PriorityQueue

```
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
maxHeap.addAll(Arrays.asList(3, 1, 4, 1, 5, 9));
System.out.println(maxHeap.poll()); // 9
```

Q2: Find K smallest elements

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.addAll(Arrays.asList(7, 10, 4, 3, 20, 15));
int k = 3;
for (int i = 0; i < k; i++) {
   System.out.println(pq.poll());
}
```

Q3: Merge K sorted arrays

- Use PriorityQueue of size k storing (value, arrayIndex, elementIndex).

Q4: Top K frequent elements

- Use HashMap for frequency + PriorityQueue for ordering.

Q5: Dijkstra's Algorithm

- Use PriorityQueue for picking the vertex with the smallest distance.