Collection

✅ 1. THEORY

ArrayList

- Backed by dynamic array.
- Allows random access (index-based operations are O(1)).
- Adding at end is O(1) amortized; adding in middle is O(n).
- Not thread-safe.
- Grows internally by 1.5x (in Java 8+).
- Allows duplicates + maintains insertion order.

LinkedList

- Backed by doubly-linked list.
- No random access (get(i) = O(n)).
- Insert/remove at head/middle/tail is O(1) after reaching the node.
- Better than ArrayList when frequent insert/delete operations occur.
- Can act as Queue, Deque.

HashSet

- Backed by HashMap (keys only).
- No duplicates, no order.
- Insert, search = O(1) average.
- Uses hashCode(), equals() to check duplicates.
- Loads factor = 0.75, triggers rehash.

LinkedHashSet

- Extends HashSet + maintains insertion order.
- Implemented via doubly-linked list running through HashMap entries.
- Slightly slower than HashSet.

TreeSet

- Backed by TreeMap → Red-Black Tree.
- Sorted order (natural / custom comparator).
- Insert, search = O(log n).
- Cannot insert null (NullPointerException).

✅ 2. INTERNAL WORKING OF hashCode(), equals(), Bucket & bitwise operations

How HashSet identifies duplicates

1. **Call hashCode() → get integer hash.**
2. **Hash is converted to a bucket index using:**

bucketIndex = (hashCode) & (capacity – 1)

✓ Bitwise AND (&) is used because it is faster than modulus (%).
✓ Works only when capacity is always a power of 2 (16, 32, 64…).

Example:
capacity = 16 → binary = 00010000
capacity–1 = 15 → binary = 00001111

So bucket = hash & 00001111.

Why bitwise & instead of modulus?

- Faster (CPU instruction level).
- No division required.
- Easy to distribute values uniformly when capacity is power of two.

Collision Handling

If two different elements generate same bucket:

Java 8+ uses:

- Linked List for small collisions.
- Converts to a Red-Black Tree when chain size > 8.

---

equals() is called after hash match:

- If bucket is same → compare with equals()
    - equals() = true → duplicate → ignore
    - equals() = false → add in same bucket

✅ 3. TOP 20 CONCEPT INTERVIEW QUESTIONS (WITH ANSWERS)

Q1. Difference between ArrayList and LinkedList?

ArrayList uses dynamic array → fast random access, slow insert/delete in middle.
LinkedList uses nodes → fast insert/delete, slow random access.

Q2. When will you prefer LinkedList over ArrayList?

When the application has frequent insert/delete operations, especially in the middle.

Q3. How does ArrayList grow internally?

When array is full → new capacity = old * 1.5, old elements copied to new array.

Q4. Why LinkedList is slower in iteration?

Because data is in scattered memory → CPU cache misses increase.

Q5. How HashSet internally works?

Uses HashMap → stores elements as keys → bucket decided by
hashCode() & (capacity – 1) → resolves collision via linked list / tree.

Q6. Why does HashSet not allow duplicate elements?

Because HashMap does not allow duplicate keys → equals + hashCode detect duplicates.

Q7. Difference between HashSet and LinkedHashSet?

HashSet → no order.
LinkedHashSet → insertion order, because linked list is maintained.

Q8. Why TreeSet is slower than HashSet?

TreeSet uses Red-Black tree → O(log n).
HashSet is O(1).

Q9. Can TreeSet store null? Why not?

No → NullPointerException because comparison is needed for sorting.

Q10. What happens if hashCode() is not overridden?

The object is hashed based on memory address → duplicate detection fails.

Q11. What if equals() overridden but hashCode() not?

Contract breaks → duplicates may appear in HashSet.

Q12. What load factor in HashSet?

0.75 → When 75% buckets fill, resize happens.

Q13. What is rehashing?

Creating new buckets and redistributing all elements.

Q14. Why HashSet performance degrades when many collisions?

Elements go into Linked List → O(n) search instead of O(1).

Q15. When does HashSet use tree nodes?

When a bucket size > 8 (Java 8+) → change to Red-Black Tree.

Q16. Difference between capacity and load factor?

Capacity → total buckets
Load factor → threshold to resize

Q17. Why HashMap/HashSet capacity is always power of two?

To enable efficient hashing using bitwise & and reduce collisions.

Q18. How LinkedHashSet maintains order?

Each node contains pointers to prev and next, forming a doubly-linked list.

Q19. How to sort a HashSet?

Convert to list and use Collections.sort, or use TreeSet.

Q20. Can we synchronize ArrayList or HashSet?

Yes → Collections.synchronizedList(list)
or use CopyOnWriteArrayList, ConcurrentHashMap for real multithreading.

✅ 4. 20 CODING INTERVIEW QUESTIONS WITH ANSWERS

Coding Q1: Remove duplicates from ArrayList

```
List<Integer> list = Arrays.asList(1,2,2,3,4,4,5);
List<Integer> unique = new ArrayList<>(new HashSet<>(list));
```

Coding Q2: Find frequency of each element (ArrayList)

```
Map<Integer, Long> freq = list.stream()
    .collect(Collectors.groupingBy(i->i, Collectors.counting()));
```

Coding Q3: Reverse a LinkedList

```
LinkedList<Integer> list = new LinkedList<>();
Collections.reverse(list);
```

Coding Q4: Find 1st repeated element

```
Set<Integer> set = new HashSet<>();
for(int n : arr){
   if(!set.add(n)) return n;
}
```

Coding Q5: Sort HashSet

```
Set<Integer> sorted = new TreeSet<>(hashSet);
```

Coding Q6: Iterate LinkedHashSet

```
for(Integer i : linkedHashSet) System.out.println(i);
```

Coding Q7: Find intersection of two sets

```
set1.retainAll(set2);
```

Coding Q8: Find union of two sets

set1.addAll(set2);

Coding Q9: Compare two ArrayLists

boolean isEqual = list1.equals(list2);

Coding Q10: Convert ArrayList to array

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

Coding Q11: Remove even numbers from HashSet

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

Coding Q12: Check if LinkedList is palindrome

boolean isPal = list.equals(new LinkedList<>(list).descendingIterator());

Coding Q13: Find largest element in ArrayList

int max = Collections.max(list);

Coding Q14: Convert List to Set

Set<Integer> set = new HashSet<>(list);

Coding Q15: Convert Set to List

List<Integer> list = new ArrayList<>(set);

Coding Q16: Remove duplicates while preserving order

List<Integer> output = new ArrayList<>(new LinkedHashSet<>(list));

Coding Q17: TreeSet with custom comparator (descending)

Set<Integer> set = new TreeSet<>((a,b) -> b - a);

Coding Q18: Find 2nd largest using TreeSet

```
TreeSet<Integer> set = new TreeSet<>(list);

set.pollLast();

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

Coding Q19: Get first and last element of LinkedList

```
list.getFirst();

list.getLast();
```

Coding Q20: Iterate TreeSet in descending order

```
for(Integer i : treeSet.descendingSet()) System.out.println(i);
```