Collection framework:

Custom immutable class with arraylist as reference.

A, To create an immutable class in Java that contains an ArrayList as a reference, you need to ensure that:

1. The class itself is declared as final to prevent subclassing.
2. The fields are declared as private final.
3. There are no setter methods to modify fields after object creation.
4. Defensive copying is used when passing mutable objects (like ArrayList) into and out of the class.

Here's an example implementation:

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public final class ImmutableClass {

private final List<String> items;

// Constructor with defensive copy

public ImmutableClass(List<String> items) {

this.items = new ArrayList<>(items); // Create a defensive copy

}

// Getter with defensive copy

public List<String> getItems() {

return Collections.unmodifiableList(new ArrayList<>(items)); // Return unmodifiable copy

}

@Override

public String toString() {

return "ImmutableClass{" + "items=" + items + '}';

}

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Item1");

list.add("Item2");

ImmutableClass immutableObj = new ImmutableClass(list);

System.out.println("Original: " + immutableObj);

// Attempt to modify the original list

list.add("Item3");

System.out.println("After modifying original list: " + immutableObj);

// Attempt to modify the list from getter

List<String> fetchedList = immutableObj.getItems();

// fetchedList.add("Item4"); // This will throw UnsupportedOperationException

System.out.println("Final state: " + immutableObj);

}

}

**Key Aspects:**

* **Defensive Copy in Constructor:** The constructor takes a List<String> and creates a new ArrayList to ensure external modifications don't affect the internal state.
* **Immutable Getter:** The getItems() method returns an **unmodifiable copy** of the list to prevent external modifications.
* **Final Class & Final Fields:** Ensures that no subclass can override behavior and fields remain constant after initialization.

This guarantees immutability while still allowing safe interaction with the ArrayList. 🚀

}

Syncronised hashmap vs concurrent hashmap

A,  
Here’s a **comparison table** for **Synchronized HashMap vs ConcurrentHashMap**:

| **Feature** | **Synchronized HashMap** | **ConcurrentHashMap** |
| --- | --- | --- |
| **Thread Safety** | Yes (locks entire map) | Yes (locks only segments/buckets) |
| **Locking Mechanism** | Locks the **whole map** | Uses **bucket-level locking** (Java 8+) |
| **Concurrency** | **Single-thread access** at a time | **Multiple threads** can read/write concurrently |
| **Performance** | **Slower** due to full map locking | **Faster**, as only segments are locked |
| **Iteration Safety** | Must manually synchronize during iteration | **Thread-safe iteration** (no extra synchronization needed) |
| **Null Keys/Values** | ✅ Allows null keys and values | ❌ **Does not allow null keys or values** |
| **Concurrent Modification Exception** | May throw ConcurrentModificationException | **Does not throw** (iterators are weakly consistent) |
| **Best Use Case** | **Single-threaded or low concurrency** | **High-concurrency applications** |

**Summary**

* Use **Synchronized HashMap** if:
  + You have **low concurrency**.
  + You need to store **null keys or values**.
  + You can handle **explicit synchronization** while iterating.
* Use **ConcurrentHashMap** if:
  + You have a **multi-threaded application**.
  + You need **high performance**.
  + You don’t need **null keys or values**.

For multi-threaded environments, **ConcurrentHashMap is recommended**. 🚀

Interview questions on collection frame work?

**Collection Framework Hierarchy**

mathematica

CopyEdit

Iterable

|

Collection

-----------------------------------

| | |

List Set Queue

| | |

ArrayList HashSet PriorityQueue

LinkedList TreeSet LinkedList

Vector

Stack

**Level-Wise Questions and Answers**

**Beginner Level**

1. **Q: What is the Java Collection Framework? Why is it used?**  
   **A:**  
   The Java Collection Framework is a unified architecture for representing and manipulating collections. It provides classes like ArrayList, HashSet, and HashMap to store and process data efficiently. It simplifies data handling, improves code quality, and reduces development effort.
2. **Q: What is the difference between ArrayList and LinkedList?**  
   **A:**
   * **ArrayList**:
     + Backed by a dynamic array.
     + Faster for random access (get operations).
   * **LinkedList**:
     + Backed by a doubly linked list.
     + Faster for insertions and deletions in the middle.
3. **Q: What is the difference between List and Set?**  
   **A:**
   * **List**: Allows duplicates and maintains the insertion order.
   * **Set**: Does not allow duplicates and does not guarantee order (except LinkedHashSet and TreeSet).

**Intermediate Level**

1. **Q: Explain the difference between HashMap and TreeMap.**  
   **A:**
   * **HashMap**:
     + Stores key-value pairs in no specific order.
     + Allows one null key and multiple null values.
   * **TreeMap**:
     + Maintains keys in natural (ascending) order.
     + Does not allow null keys.
2. **Q: What is the difference between ArrayList and Vector?**  
   **A:**
   * **ArrayList**:
     + Not synchronized (not thread-safe).
     + Faster as it does not require synchronization.
   * **Vector**:
     + Synchronized (thread-safe).
     + Slower due to synchronization overhead.
3. **Q: How does a HashSet ensure uniqueness?**  
   **A:**  
   HashSet uses a HashMap internally. It calculates the hash code of objects and compares them to ensure no two objects have the same hash code or content.

**Advanced Level**

1. **Q: What is the internal working of a HashMap?**  
   **A:**
   * A HashMap uses an array of buckets to store key-value pairs.
   * It calculates the hash code of the key to determine the bucket.
   * In case of collisions (same hash code), it uses a linked list or binary tree (from Java 8) to store multiple entries in a single bucket.
2. **Q: What is the difference between fail-fast and fail-safe iterators?**  
   **A:**
   * **Fail-fast**: Throws a ConcurrentModificationException if the collection is modified while iterating (e.g., ArrayList, HashMap).
   * **Fail-safe**: Does not throw an exception as it works on a copy of the collection (e.g., CopyOnWriteArrayList).
3. **Q: Explain the difference between Comparable and Comparator.**  
   **A:**
   * **Comparable**:
     + Used to define natural ordering of objects.
     + Implemented in the class itself via compareTo().
   * **Comparator**:
     + Used for custom ordering.
     + Implemented as a separate class or lambda via compare().

**Example**:

java

CopyEdit

import java.util.\*;

class Student implements Comparable<Student> {

int id;

String name;

public Student(int id, String name) {

this.id = id;

this.name = name;

}

public int compareTo(Student other) {

return this.id - other.id;

}

}

public class Main {

public static void main(String[] args) {

List<Student> students = new ArrayList<>();

students.add(new Student(2, "Alice"));

students.add(new Student(1, "Bob"));

Collections.sort(students);

}

}

4,what is difference of linked list and arraylist?

A,

| **Aspect** | **List** | **ArrayList** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Definition** | List is an **interface** in the Java Collections Framework. | ArrayList is a **class** that implements the List interface. |

|  |  |  |
| --- | --- | --- |
| **Implementation** | List is a blueprint; it cannot be instantiated. | ArrayList is a concrete implementation of List backed by a dynamic array. |

|  |  |  |
| --- | --- | --- |
| **Type** | Abstract concept. | Specific implementation of the List interface. |

|  |  |  |
| --- | --- | --- |
| **Instantiation** | You cannot directly create an instance of List. Example: List<E> | You can directly create an instance of ArrayList. Example: new ArrayList<E>() |

|  |  |  |
| --- | --- | --- |
| **Flexibility** | Can point to any class that implements List (e.g., ArrayList, LinkedList, Vector). | Strictly tied to the dynamic array implementation. |

|  |  |  |
| --- | --- | --- |
| **Performance** | Depends on the implementation class being used (e.g., ArrayList or LinkedList). | Performance is based on the dynamic array's characteristics. |

|  |  |  |
| --- | --- | --- |
| **Memory Management** | Depends on the implementation class. | Memory is managed as a resizable array. |

|  |  |  |
| --- | --- | --- |
| **Methods** | Defines methods like add(), remove(), get(), etc., in a generic way. | Inherits and implements all List methods specifically for an array-based structure. |

5,what is hash map vs treemap?  
a,

| **Aspect** | **HashMap** | **TreeMap** |
| --- | --- | --- |
| **Data Structure** | Uses a hash table. | Uses a red-black tree (self-balancing binary tree). |
| **Ordering** | No guarantee of order (entries may appear unordered). | Maintains entries in **sorted order** by keys. |
| **Null Keys** | Allows **one null key** and multiple null values. | Does **not allow null keys**, but allows null values. |
| **Performance** | Faster for most operations: O(1) for put/get. | Slower: O(log n) for put/get due to tree traversal. |
| **Use Case** | Suitable for quick lookups without concern for order. | Suitable when sorted order of keys is required. |

**Example: HashMap**

java

Copy code

import java.util.HashMap;

public class HashMapExample {

public static void main(String[] args) {

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

hashMap.put(3, "Three");

hashMap.put(1, "One");

hashMap.put(2, "Two");

hashMap.put(null, "Null Key");

System.out.println("HashMap:");

for (Integer key : hashMap.keySet()) {

System.out.println("Key: " + key + ", Value: " + hashMap.get(key));

}

}

}

**Output** (Order may vary):

yaml

Copy code

HashMap:

Key: null, Value: Null Key

Key: 1, Value: One

Key: 2, Value: Two

Key: 3, Value: Three

**Example: TreeMap**

java

Copy code

import java.util.TreeMap;

public class TreeMapExample {

public static void main(String[] args) {

TreeMap<Integer, String> treeMap = new TreeMap<>();

treeMap.put(3, "Three");

treeMap.put(1, "One");

treeMap.put(2, "Two");

// treeMap.put(null, "Null Key"); // Throws NullPointerException

System.out.println("TreeMap:");

for (Integer key : treeMap.keySet()) {

System.out.println("Key: " + key + ", Value: " + treeMap.get(key));

}

}

}

**Output** (Keys sorted):

yaml

Copy code

TreeMap:

Key: 1, Value: One

Key: 2, Value: Two

Key: 3, Value: Three

6,’what is hashmap vs list?

A,

| **Aspect** | **HashMap** | **ArrayList** |
| --- | --- | --- |
| **Purpose** | A HashMap is a key-value pair data structure. | An ArrayList is a resizable array for storing elements. |
| **Structure** | Uses a hash table internally. | Uses a dynamic array internally. |
| **Data Organization** | Stores data in key-value pairs (Map.Entry<K,V>). | Stores elements sequentially in an indexed manner. |
| **Access Time** | Fast lookup by key: O(1) (average case). | Fast random access by index: O(1). |
| **Order** | Does not guarantee the order of elements. | Maintains insertion order. |
| **Duplicates** | Keys must be unique; values can be duplicated. | Allows duplicate elements. |
| **Null Handling** | Allows one null key and multiple null values. | Allows multiple null elements. |
| **Use Case** | Used when you need key-value mapping and fast lookup. | Used when you need an ordered collection of elements. |
| **Implementation** | Implements the Map interface. | Implements the List interface. |

7,explain internal working of hashmap?

A,

i,what is hashcode?

A **hash code** is a numerical value that is generated by a hash function from an object, commonly used in data structures like hash tables to quickly locate data.

This numerical value is used in arraing index bucket in hash table.

Ex:

String str1 = "Hello"; String str2 = "World"; String str3 = "Hello";

System.out.println("Hash code for str1 (\"Hello\"): " + str1.hashCode());

ii , what is hashing ?

a,

**Hashing** is a process of converting data (like strings, numbers, or objects) into a fixed-size numerical value called a hash value or hash code, using a function known as a **hash function.**

**Code:**

**class Employee {**

**int id;**

**String name;**

**public Employee(int id, String name) {**

**this.id = id;**

**this.name = name;**

**}**

**@Override**

**public int hashCode() {**

**// Custom hash function: Combine id and name's hash code**

**return id \* 31 + name.hashCode();**

**}**

**@Override**

**public boolean equals(Object obj) {**

**if (this == obj) return true;**

**if (obj == null || getClass() != obj.getClass()) return false;**

**Employee employee = (Employee) obj;**

**return id == employee.id && name.equals(employee.name);**

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**HashMap<Employee, String> employeeMap = new HashMap<>();**

**Employee emp1 = new Employee(1, "John");**

**Employee emp2 = new Employee(2, "Jane");**

**employeeMap.put(emp1, "HR");**

**employeeMap.put(emp2, "Finance");**

**System.out.println("John's department: " + employeeMap.get(emp1));**

**}**

**}**

**Iii ,what is hash table?**

**A ,**

**A hash table is a data structure that stores key-value pairs. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found efficiently.**

**8,how put method works in hashmap and equals method is used?**

**A,after providing key,value**

**It finds hash code using hash function ex:n%10**

**It will store data in that buket index.**

**If other key got the same index it will form linked list and continue adding.**

**But linked list o(n) traversal.**

**So if elements>8 it will form balanced tree.**

**In balanced tree less hashcode values are left remainin are right.**

**While travelling it checks equal function if exists it replaces value keeping key same.**

**If not exist attach at end.**

**9,when a linked list changed to tree in hashmap?**

**If threshold value>8**

**List changes to tree.**

**10,how delete function works?**

**If equal is true it deletes eventually.**

**11,how contains function works?**

**A,Use equal method and returns true.**

**12,what is collision in hashmap?**

**A,**

**A collision in a hash map occurs when two different keys generate the same hash value and are assigned to the same index in the hash table. This happens because the hash function, which maps keys to indices, may produce the same output for multiple inputs due to the finite size of the hash table.**

**Why Collisions Occur:**

1. **Finite Hash Table Size: A hash table has a limited number of slots, but the number of possible keys is often much larger.**
2. **Hash Function Limitations: A hash function maps keys to a fixed range of indices, so two different keys can map to the same index.**

**How chaining applies:**

**Chaining (Separate Chaining):**

* **Each index in the hash table points to a linked list (or another data structure) that holds all key-value pairs with the same hash value.**
* **If a collision occurs, the new key-value pair is added to the linked list at that index.**
* **Example: hash\_table[index] = [key1, key2, ...].**

**13, State the differences between a Hashmap and a Hashtable in Java.**

**HashMap Vs HashTable?**

**14, Which is the best technique to handle collision in a hashmap.**

**As part of its collision handling, HashMap employs chaining. In chaining, a linked list is used for placing the key-value pairs inserted into the map with the value already present to avoid collision in the map at a bucket location as the newly inserted value is placed in front of the linked list.**

**16,**

**Distinguish between a Hashmap and ConcurrentHashMap in Java along with the implementation of both.**

**Hashmap vs ConcurrentHashMap:**

1. **ThreadSafe: One of the most significant differences between both is that a Cocurrenthashmap is synchronized internally and thread-safe which makes it suitable for a multithreaded environment whereas a hashmap is non-synchronized as well non-thread-safe which makes it unsuitable for a multithreaded environment.**
2. **Null Keys And Null Values: As we all know, data is kept in the form of key and value pairs in a hashmap and we can store as many null values in a hashmap as we want but only 1 null key can be stored whereas in a concurrent hashmap null keys, as well as null values, are not permitted.**
3. **import** java.util.\*;
4. **import** java.io.\*;
5. **public** **class** **interviewBit** {
6. **public** **static** **void** **main**(String[] args)
7. {
8. HashMap hmap=**new** HashMap();
9. hmap.put(91,"Hashmap Implementation");
10. hmap.put(92,"in ");
11. hmap.put(93,"Java");
12. hmap.put(**null**,"InterviewBit");
13. System.out.println(hmap);
14. }
15. }
16. **OUTPUT:**
17. java -cp /tmp/ZPv88JJ0D2 interviewBit
18. {null=InterviewBit, 91=Hashmap Implementation, 92=in , 93=Java}

* **ConcurrentHashMap implementation:**

**For implementing ConcurrentHashmap we have to import the concurrent package.**

**import java.util.concurrent.ConcurrentHashMap;**

**public class interviewBit {**

**public static void main(String[] args)**

**{**

**ConcurrentHashMap hmap=new ConcurrentHashMap();**

**hmap.put(91,"Hashmap Implementation");**

**hmap.put(92,"in ");**

**hmap.put(93,"Java");**

**hmap.put(null,"InterviewBit");**

**System.out.println(hmap);**

**}**

**}**

**OUTPUT:**

**java -cp /tmp/ZPv88JJ0D2 interviewBit**

**Exception in thread "main" java.lang.NullPointerException**

**at java.base/java.util.concurrent.ConcurrentHashMap.putVal(ConcurrentHashMap.java:1011)**

**at java.base/java.util.concurrent.ConcurrentHashMap.put(ConcurrentHashMap.java:1006)**

**at interviewBit.main(interviewBit.java:12)**

4. HashCode Consistency

• String objects are often used as keys in hash-based collections like HashMap. Immutability guarantees the hash code remains consistent, preventing issues with hash-based lookups.

• Example:

String key = "user";

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

map.put(key, "John");

// If String were mutable, changing the value of 'key' would break the map

Some missing collection framework questions ?

The interview questions you've provided are quite comprehensive in terms of testing knowledge about Java Collections Framework, particularly focusing on core concepts like HashMap, TreeMap, ArrayList, LinkedList, etc. However, there are several other important concepts and scenarios related to the Collection Framework that could be included to give a more complete understanding. Here are some missing concepts that you can consider adding to your set of interview questions:

**Missing Concepts:**

1. **Performance and Time Complexity**:
   * **Question**: What is the time complexity of the most common operations (add, remove, contains, get) for different collection classes?
   * **Answer**:
     + ArrayList:
       - add() is O(1) (amortized), get() is O(1), remove() is O(n) (in the worst case).
     + HashMap:
       - put(), get() are O(1) (average), O(n) in the worst case (due to collisions).
     + TreeMap:
       - put(), get() are O(log n) because of the balanced tree structure.
     + LinkedList:
       - add(), remove() are O(1) for the beginning or end of the list, O(n) for the middle.
2. **Synchronized Collection Classes**:
   * **Question**: What are some of the synchronized collection classes available in Java, and when should they be used?
   * **Answer**:
     + Vector, Hashtable, and the methods in Collections.synchronizedList() or Collections.synchronizedMap() provide synchronized versions of the basic collections.
     + These should be used when working with collections in a multi-threaded environment where thread safety is required, but they may have performance overhead due to synchronization.
3. **List Iterators**:
   * **Question**: What is the difference between a regular Iterator and a ListIterator?
   * **Answer**:
     + A ListIterator extends the Iterator interface and provides additional methods such as add(), set(), previous(), and hasPrevious(). It also allows bidirectional traversal, meaning it can iterate both forwards and backwards through the list, whereas a regular Iterator can only move forward.
4. **Concurrent Collections**:
   * **Question**: What are the differences between ConcurrentHashMap and HashMap regarding concurrency and performance?
   * **Answer**:
     + ConcurrentHashMap is thread-safe and allows concurrent access by multiple threads. It is optimized for high concurrency by dividing the map into segments, allowing multiple threads to operate on different segments simultaneously.
     + HashMap is not thread-safe and can lead to data corruption when accessed concurrently by multiple threads.
5. **WeakHashMap**:
   * **Question**: What is a WeakHashMap, and when would you use it?
   * **Answer**:
     + A WeakHashMap is a special type of map that allows its keys to be garbage-collected when they are no longer in use. This is particularly useful for caching data where you want the cache to automatically release entries that are no longer referenced.
6. **NavigableMap and NavigableSet**:
   * **Question**: What is the difference between NavigableMap and SortedMap?
   * **Answer**:
     + NavigableMap extends SortedMap and provides additional methods for navigating through the map, such as lowerKey(), higherKey(), ceilingKey(), floorKey(), etc.
     + SortedMap only provides sorting functionalities like firstKey(), lastKey(), headMap(), tailMap(), etc.
7. **Stream API with Collection Framework**:
   * **Question**: How do you use Java Streams with collections? Provide an example.
   * **Answer**:
     + Streams can be used to process collections in a functional style. Example:
     + List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
     + List<Integer> evenNumbers = numbers.stream()
     + .filter(n -> n % 2 == 0)
     + .collect(Collectors.toList());
     + System.out.println(evenNumbers); // Output: [2, 4]
     + Streams provide methods like filter(), map(), reduce(), etc., which can help manipulate the data in the collection.
8. **Custom Collection Classes**:
   * **Question**: How would you implement a custom collection in Java? Can you extend AbstractList, AbstractSet, etc.?
   * **Answer**:
     + Java allows you to extend existing collection classes like AbstractList, AbstractSet, or AbstractMap to create your own collection types. You need to implement the abstract methods and ensure proper behavior.
     + Example: Implementing a custom List by extending AbstractList.
9. **CopyOnWrite Collections**:
   * **Question**: What is a CopyOnWriteArrayList, and when should you use it?
   * **Answer**:
     + CopyOnWriteArrayList is a thread-safe version of ArrayList. It makes a copy of the underlying array every time a modification is made, thus providing thread-safety but with performance overhead for write operations. It's suitable for situations where reads are far more frequent than writes.
10. **EnumSet and EnumMap**:
    * **Question**: What are EnumSet and EnumMap, and when would you use them?
    * **Answer**:
      + EnumSet is a specialized set for storing enum types. It is more efficient than HashSet for enums and allows bitwise operations on sets.
      + EnumMap is a specialized map for enum keys. It is more efficient than using a HashMap with enums as keys.
11. **Priority Queue**:
    * **Question**: How does a PriorityQueue work in Java, and what is its use case?
    * **Answer**:
      + A PriorityQueue is a collection that orders elements according to their natural ordering or by a comparator provided at the time of creation. It is useful in scenarios like implementing a heap or priority scheduling algorithms.
12. **ArrayList vs. Vector**:
    * **Question**: What is the difference between ArrayList and Vector in terms of synchronization and resizing?
    * **Answer**:
      + Vector is synchronized, making it thread-safe but slower in performance. ArrayList is not synchronized, offering better performance in single-threaded environments.
13. **Cloneable Interface**:
    * **Question**: How does the clone() method work with collections in Java?
    * **Answer**:
      + Collections like ArrayList or HashMap implement the Cloneable interface. The clone() method creates a shallow copy of the collection. If deep copies are required, you would need to implement custom logic.
14. **ArrayList vs LinkedList**:
    * **Question**: How does ArrayList compare to LinkedList in terms of performance?
    * **Answer**:
      + ArrayList provides fast random access and is better when the number of insertions and deletions is low.
      + LinkedList is better when insertions and deletions are frequent, especially in the middle of the list, but slower for random access.

These additional questions touch on different aspects of Java's Collection Framework that may be asked in advanced interviews to evaluate an in-depth understanding of the concepts.

Here are the answers to your questions about HashMap in Java:

**Q1. How HashMap works internally?**

A **HashMap** works by storing key-value pairs in an array of **buckets**. The key is hashed using its hashCode(), and this hash value determines which bucket the key-value pair will be placed in. The value is then stored in the appropriate bucket. If multiple keys have the same hash code (a hash collision), they will be stored in the same bucket, typically in a **linked list** or **tree** (after Java 8). The get() and put() operations involve calculating the hash value of the key and searching for the appropriate bucket.

**Q2. Can we use Custom Object as key in HashMap?**

Yes, you can use a custom object as a key in a **HashMap**. However, for the map to function correctly, the custom object should override the hashCode() and equals() methods. This ensures that keys with the same data are treated as equal and are placed in the same bucket when necessary.

**Q3. What is HashMap in Java?**

A **HashMap** is a class in the java.util package that implements the **Map** interface. It is used to store key-value pairs, where each key is unique, and it allows fast retrieval of values based on keys. Internally, it uses an array of buckets and hashing to store and access the elements efficiently.

**Q4. Why is String considered a good key for HashMap?**

String is considered a good key for HashMap because:

1. It is **immutable** (it can't change once created), which means its hash code won't change during its lifetime.
2. The String class overrides both hashCode() and equals() methods effectively, ensuring proper key comparison and bucket distribution.
3. The **hashCode** of a String is computed based on its content, making it well-distributed across different keys.

**Q5. Is HashMap thread-safe in Java?**

No, **HashMap** is **not thread-safe** in Java. Multiple threads modifying the HashMap concurrently can lead to inconsistent data. To make it thread-safe, you can use **ConcurrentHashMap** or synchronize the HashMap manually using Collections.synchronizedMap().

**Q6. Explain HashMap before and after Java 8 changes?**

Before Java 8:

* HashMap used a simple **linked list** to resolve hash collisions.
* If multiple keys had the same hash code, they would be chained in a linked list.

After Java 8:

* If a bucket contains more than 8 entries, the linked list is converted into a **balanced tree** (specifically a **Red-Black Tree**) for better performance (O(log n) time complexity for lookups and insertions).
* This change improves the performance of get(), put(), and containsKey() operations when dealing with hash collisions.

**Q7. How to retrieve value if two obj keys have same hashcode?**

If two keys have the same hash code, they will end up in the same bucket. The HashMap will then use the equals() method to compare the keys. If the keys are equal (according to equals()), the corresponding value will be returned. If the keys are not equal, the map will continue searching in the linked list or tree (depending on Java version).

**Q8. Explain Equals and Hashcode Method with Example - PART 1?**

The hashCode() method returns an integer value used to place the object in a specific bucket in the HashMap. The equals() method compares two objects for equality. When using custom objects as keys in a HashMap, it’s crucial to override both methods to ensure correct behavior.

Example:

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int hashCode() {

return Objects.hash(name, age);

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Person person = (Person) obj;

return age == person.age && Objects.equals(name, person.name);

}

}

**Q9. Explain Equals and Hashcode Method with Example - PART 2?**

Continuing from PART 1, if two objects are considered equal according to the equals() method, their hashCode() must return the same value. This is essential for the proper functioning of a HashMap.

Example:

* If two Person objects with the same name and age are stored in a HashMap, their hashCode() method will return the same hash value, ensuring they are placed in the same bucket. The equals() method will then check if the objects are truly equal.

**Q10. How to Sort a HashMap in Java?**

To sort a HashMap, you typically need to sort it by its keys or values. Here's how to sort it by values:

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

map.put("A", 5);

map.put("B", 3);

map.put("C", 8);

// Sorting by values

Map<String, Integer> sortedMap = map.entrySet()

.stream()

.sorted(Map.Entry.comparingByValue())

.collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,

(e1, e2) -> e1, LinkedHashMap::new));

**Q11. Explain difference between Fail-fast Vs Fail-safe Iterators in Java?**

* **Fail-fast Iterators**: These iterators throw a ConcurrentModificationException if the collection is modified while iterating (except through the iterator itself).
* **Fail-safe Iterators**: These iterators do not throw an exception if the collection is modified during iteration. Instead, they operate on a clone or copy of the collection.

Example:

* **ArrayList** uses a fail-fast iterator.
* **CopyOnWriteArrayList** uses a fail-safe iterator.

**Q12. What is Stack class in Java?**

Stack is a class in Java that models a stack data structure (LIFO). It provides methods like push(), pop(), and peek() to manage elements in the stack. However, it is considered outdated, and the Deque interface (with implementations like ArrayDeque) is preferred for stack-like operations.

**Q13. What is Vector class in Java?**

Vector is a resizable array implementation of the List interface. It is synchronized, which makes it thread-safe but slower than other non-synchronized collections like ArrayList. However, it is considered outdated and generally avoided in modern Java in favor of ArrayList.

**Q14. What is HashMap class in Java?**

HashMap is a collection class that implements the Map interface. It stores key-value pairs and allows efficient lookups and insertions based on the hash value of the key. It doesn't maintain the order of elements and doesn't allow duplicate keys.

**Q15. What is WeakHashMap class in Java?**

WeakHashMap is a variation of HashMap where the keys are weakly referenced. If a key is no longer referenced elsewhere, it can be garbage collected, even if it is still present in the map. This is useful for caches and similar use cases.

**Q16. What is IdentityHashMap in Java?**

IdentityHashMap is a variant of HashMap where key comparisons are done using reference equality (==) rather than object equality (equals()). This means two different objects with the same data will be treated as different keys.

**Q17. How concurrent hashmap works before & after Java 8?**

Before Java 8, **ConcurrentHashMap** used **segment locking** to allow for thread-safe concurrent access to different portions of the map. After Java 8, it switched to using **synchronized blocks** and **CAS (Compare-And-Swap)** for more efficient thread safety.

**Q18. What is ConcurrentHashMap?**

ConcurrentHashMap is a thread-safe map that allows multiple threads to concurrently read and write to the map without blocking each other. It divides the map into segments and locks only the segment being modified, enabling more concurrency compared to Hashtable.

Let me know if you'd like further clarification on any specific question!

Sure! Let’s continue with the next set of questions.

**Q19. What is hash collision?**

A **hash collision** occurs when two different keys produce the same hash code. Since a hash map uses the hash code of the key to determine the bucket in which to store the entry, when two keys have the same hash code, they will end up in the same bucket. The map then needs a way to distinguish between these keys (such as using the equals() method to compare them).

**Q20. Does HashMap allow duplicate values?**

Yes, **HashMap** allows duplicate values. The values in a HashMap do not need to be unique; only the keys need to be unique. If you attempt to insert a new key-value pair with an existing key, the new value will overwrite the old one.

**Q21. Can you store null keys in HashMap?**

Yes, a **HashMap** allows **one null key** and **multiple null values**. However, you can only have a single null key because keys must be unique. If you try to insert another entry with the same key (null or otherwise), the new value will overwrite the existing one.

**Q22. What will happen if two different objects have the same hashcode?**

If two different objects have the same hash code, they will be placed in the same bucket in the HashMap. The HashMap will then use the equals() method to determine whether the keys are actually the same. If the equals() method returns true, the second object will replace the first one. If equals() returns false, the second object will be added to the bucket as a collision.

**Q23. How will you retrieve the Value object if two Keys have the same hashcode?**

If two keys have the same hash code, they will be stored in the same bucket, usually in a linked list or a tree structure (after Java 8). To retrieve the value associated with a key, the HashMap will first calculate the hash code of the key, then compare the key using equals() to find the exact match within the bucket. If a match is found, it retrieves the corresponding value.

**Q24. How many entries you can put into a HashMap?**

The number of entries you can store in a **HashMap** is theoretically **unlimited** (as long as there is available memory). However, in practice, the maximum capacity will depend on the JVM and the system memory available. Internally, **HashMap** dynamically resizes as the number of entries grows.

**Q25. What is the difference between HashMap and Hashtable?**

The main differences between **HashMap** and **Hashtable** are:

1. **Thread-safety**: Hashtable is synchronized and thread-safe, meaning only one thread can access it at a time. HashMap, on the other hand, is **not thread-safe**.
2. **Null keys and values**: HashMap allows **one null key** and multiple null values, while Hashtable does not allow **null keys or values**.
3. **Performance**: Since Hashtable is synchronized, it tends to be slower than HashMap.

**Q26. Difference between HashMap and ConcurrentHashMap?**

* **Thread Safety**: HashMap is **not thread-safe**, while ConcurrentHashMap is designed for **thread-safe** operations in concurrent environments.
* **Locking Mechanism**: ConcurrentHashMap uses **segment-level locking** or **fine-grained locking**, which allows multiple threads to read/write concurrently. HashMap uses no locking mechanism.
* **Performance**: ConcurrentHashMap performs better in multi-threaded scenarios compared to HashMap.

**Q27. How does HashMap handle null keys in Java?**

HashMap allows exactly **one null key** and multiple **null values**. The null key is placed in the first bucket (index 0). When you insert a null key, the map uses null for the hash code and places it in the appropriate bucket. The value associated with the null key can also be a valid object (null or non-null).

**Q28. What is the difference between HashMap and HashSet?**

* **HashMap**: It is a collection of key-value pairs, where each key is unique. It allows retrieval of values by keys.
* **HashSet**: It is a collection of unique elements (just like a set in mathematics) and does not store any key-value pairs. It only stores individual objects without associating them with a value.

**Q29. How do you remove an entry from HashMap while iterating over it?**

You can remove an entry from a HashMap while iterating using the iterator's remove() method to avoid ConcurrentModificationException. Here's an example:

Iterator<Map.Entry<String, Integer>> iterator = map.entrySet().iterator();

while (iterator.hasNext()) {

Map.Entry<String, Integer> entry = iterator.next();

if (entry.getValue() == someValue) {

iterator.remove(); // safely remove the entry during iteration

}

}

**Q30. Does the iterator returned by HashMap is fail-safe or fail-fast?**

The iterator returned by **HashMap** is **fail-fast**, meaning if the map is modified structurally while the iterator is being used (outside of the iterator itself), it will throw a ConcurrentModificationException.

**Q31. Can you store duplicate keys in HashMap? What happens if you try to put a key which already exists in HashMap?**

No, HashMap does not allow duplicate keys. If you try to insert a key that already exists in the map, the old value will be replaced with the new value associated with that key.

**Q32. Difference between HashMap and WeakHashMap?**

* **HashMap**: Stores strong references to keys and values. The objects referenced as keys will not be garbage collected as long as they are present in the map.
* **WeakHashMap**: Stores weak references to keys. This means that if a key is no longer referenced outside of the WeakHashMap, it can be garbage collected even if it is still in the map.

**Q33. How do you sort HashMap by values in Java?**

To sort a HashMap by values, you can use a List of Map.Entry objects and sort it based on values. Here’s an example:

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

map.put("A", 5);

map.put("B", 3);

map.put("C", 8);

// Sort by values

Map<String, Integer> sortedMap = map.entrySet()

.stream()

.sorted(Map.Entry.comparingByValue())

.collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,

(e1, e2) -> e1, LinkedHashMap::new));

**Q34. Is HashMap thread-safe?**

No, **HashMap** is not thread-safe. If multiple threads access the HashMap concurrently and modify it, it can lead to inconsistent data. To make it thread-safe, you can use **ConcurrentHashMap** or **Collections.synchronizedMap()**.

**Q35. What is the time complexity of get() and put() in HashMap?**

* The time complexity of both get() and put() operations is **O(1)** on average, assuming a good distribution of hash codes and minimal hash collisions. In the worst case (due to collisions), the time complexity can degrade to **O(n)**, but this is rare.

**Q36. How does the get() method of HashMap work?**

The get() method in a HashMap works by calculating the hash code of the key, determining the bucket index, and then looking for the corresponding key-value pair in the bucket. If the key matches, the value is returned.

**Q37. How does the put() method of HashMap work?**

The put() method works by calculating the hash code of the key, determining the appropriate bucket, and placing the key-value pair in that bucket. If a key already exists in the map, its value is replaced with the new value. If not, a new entry is created.

**Q38. As we know that hashcodes are generated for each key, but what happens when the same hashcode is generated for distinct keys?**

When two distinct keys have the same hash code (a hash collision), they will be stored in the same bucket. The HashMap will then use the equals() method to distinguish between them. If the equals() method returns true, they are considered the same key; otherwise, they are treated as distinct keys and stored in a linked list or a tree structure inside the bucket.

Let me know if you'd like further details on any specific topic!

Some string interview questions from back end developer?

**Q1. Why is String immutable in Java?** Strings are immutable in Java for several reasons:

* **Security**: Immutable objects are inherently thread-safe, which is crucial for security-sensitive applications.
* **Caching**: String literals are cached in the String Constant Pool, allowing for reuse and reducing memory overhead.
* **Performance**: Immutable objects can be shared freely without copying, improving performance.
* **Hashcode**: The immutability of strings ensures that their hashcode remains constant, which is important for their use in collections like HashMap.

**Q2. String Constant Pool (SCP) in Java?** The String Constant Pool (SCP) is a special memory region where Java stores string literals. When a string literal is created, the JVM checks the SCP to see if an identical string already exists. If it does, the reference to the existing string is returned; otherwise, a new string is created and added to the pool.

**Q3. Is it possible to compare Strings using the == operator?** Yes, it is possible to compare strings using the == operator, but it compares the reference addresses, not the actual content. To compare the content of two strings, the equals() method should be used.

**Q4. What does the String intern() method do?** The intern() method returns a canonical representation of the string. It checks if the string is already in the String Constant Pool, and if it is, it returns the reference from the pool. If not, it adds the string to the pool and returns the reference.

**Q5. StringBuffer vs StringBuilder?**

* **StringBuffer**: Synchronized, thread-safe, and slower. Suitable for use in a multi-threaded environment.
* **StringBuilder**: Not synchronized, not thread-safe, and faster. Suitable for use in a single-threaded environment.

**Q6. What is the use of StringTokenizer?** StringTokenizer is used to break a string into tokens based on specified delimiters. It is a legacy class and has been largely replaced by the split() method and Scanner class.

**Q7. Difference between String a = "ravi" and String a2 = new String("ravi")?**

* String a = "ravi": Creates a string literal and stores it in the String Constant Pool.
* String a2 = new String("ravi"): Creates a new String object in the heap memory, even if an identical string exists in the String Constant Pool.

**Q8. Difference between String, StringBuffer, StringBuilder?**

* **String**: Immutable, stored in the String Constant Pool.
* **StringBuffer**: Mutable, synchronized, thread-safe, slower.
* **StringBuilder**: Mutable, not synchronized, not thread-safe, faster.

**Q9. StringBuilder Scenario Question 1?** Please provide more details about the specific scenario you have in mind for StringBuilder.

**Q10. In what scenario should StringBuilder be preferred over String class?** StringBuilder should be preferred over String when you need to perform multiple modifications (like appending, inserting, or deleting characters) to a string. Using StringBuilder avoids the creation of multiple intermediate String objects, improving performance.

**Q11. Why is a char array preferred to store passwords rather than strings?** A char array is preferred to store passwords because it can be explicitly cleared from memory after use, reducing the risk of sensitive data being exposed. Strings are immutable and their contents remain in memory until garbage collected, which can be a security risk.

**Q12. What is the use of the toString() method in Java?** The toString() method provides a string representation of an object. It is commonly overridden in custom classes to provide meaningful output when the object is printed or logged.

**Q13. Why does Java use the String literal concept?** Java uses the String literal concept to optimize memory usage and improve performance. By storing string literals in the String Constant Pool, Java ensures that identical strings are reused, reducing the number of String objects created and the memory footprint.

Feel free to ask if you have more questions or need further clarification on any of these topics!

Some interview questions about collection frame work from backenddeveloper?  
**Q1. How does HashMap work internally?** HashMap in Java works on the principle of hashing. It uses an array of buckets to store key-value pairs. When a key-value pair is added, the hash code of the key is computed, and the bucket index is determined using the hash code. If multiple keys map to the same bucket (hash collision), a linked list or a balanced tree (since Java 8) is used to store the entries.

**Q2. Can we use a custom object as a key in HashMap?** Yes, you can use a custom object as a key in HashMap. However, you must override the equals() and hashCode() methods in your custom class to ensure that the keys are compared correctly and the hash codes are generated consistently.

**Q3. What is HashMap in Java?** HashMap is a part of Java's collection framework and is used to store key-value pairs. It allows for fast retrieval, insertion, and deletion of elements based on keys. HashMap permits null values and one null key.

**Q4. Why is String considered a good key for HashMap?** String is considered a good key for HashMap because it is immutable and final, meaning its hash code is cached at the time of creation and does not change. This ensures consistent behavior when used as a key in HashMap.

**Q5. Is HashMap thread-safe in Java?** No, HashMap is not thread-safe. If multiple threads access a HashMap concurrently and at least one of the threads modifies the map structurally, it must be synchronized externally.

**Q6. Explain HashMap before and after Java 8 changes.** Before Java 8, HashMap used a linked list to handle hash collisions. This could lead to performance degradation in the case of many collisions. After Java 8, HashMap uses a balanced tree (Red-Black tree) to handle collisions when the number of entries in a bucket exceeds a certain threshold, improving performance.

**Q7. How to retrieve a value if two object keys have the same hash code?** If two keys have the same hash code, they will be placed in the same bucket. To retrieve the value, HashMap uses the equals() method to compare the keys and find the correct entry.

**Q8. Explain equals() and hashCode() methods with an example - PART 1.** The equals() method checks if two objects are equal, while the hashCode() method returns an integer representation of the object. Both methods must be overridden to ensure correct behavior in collections like HashMap.

class Person {

private String name;

private int age;

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Person person = (Person) obj;

return age == person.age && Objects.equals(name, person.name);

}

@Override

public int hashCode() {

return Objects.hash(name, age);

}

}

**Q9. Explain equals() and hashCode() methods with an example - PART 2.** Continuing from the previous example, the equals() method ensures that two Person objects are considered equal if they have the same name and age. The hashCode() method generates a hash code based on the name and age, ensuring that equal objects have the same hash code.

**Q10. How to sort a HashMap in Java?** You can sort a HashMap by converting it to a list of entries and then sorting the list using a comparator. Finally, you can create a new LinkedHashMap to maintain the sorted order.

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

mapA", 10);

map.put("B", 30);

map.put("C", 20);

List<Map.Entry<String, Integer>> list = new ArrayList<>(map.entrySet());

list.sort(Map.Entry.comparingByValue());

Map<String, Integer> sortedMap = new LinkedHashMap<>();

for (Map.Entry<String, Integer> entry : list) {

sortedMap.put(entry.getKey(), entry.getValue());

}

System.out.println("Sorted HashMap: " + sortedMap);

**Q11. Explain the difference between Fail-fast and Fail-safe Iterators in Java.**

* **Fail-fast Iterators**: These iterators throw a ConcurrentModificationException if the collection is modified while iterating over it. Examples include iterators of ArrayList, HashMap, etc.
* **Fail-safe Iterators**: These iterators do not throw an exception if the collection is modified while iterating. They operate on a clone of the collection. Examples include iterators of ConcurrentHashMap, CopyOnWriteArrayList, etc.

**Q12. What is the Stack class in Java?** The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends the Vector class and provides methods like push(), pop(), peek(), empty(), and search().

**Q13. What is the Vector class in Java?** The Vector class implements a growable array of objects. It is synchronized and can be used in a multi-threaded environment. It provides methods for adding, removing, and accessing elements.

**Q14. What is the HashMap class in Java?** The HashMap class is a part of Java's collection framework and is used to store key-value pairs. It allows for fast retrieval, insertion, and deletion of elements based on keys. HashMap permits null values and one null key.

**Q15. What is the WeakHashMap class in Java?** WeakHashMap is a HashMap implementation that uses weak references for its keys. This means that if a key is no longer referenced elsewhere, it can be garbage collected, allowing the entry to be removed from the map.

**Q16. What is the IdentityHashMap in Java?** IdentityHashMap is a HashMap implementation that uses reference equality (==) instead of object equality (equals()) for comparing keys. It is useful when you need to distinguish between different instances of the same object.

**Q17. How did ConcurrentHashMap work before and after Java 8?** Before Java 8, ConcurrentHashMap used a segmented locking mechanism, where the map was divided into segments, each with its own lock. After Java 8, it uses a combination of lock-free and fine-grained locking techniques, including CAS (Compare-And-Swap) operations, to improve performance and scalability.

**Q18. What is ConcurrentHashMap?** ConcurrentHashMap is a thread-safe implementation of the Map interface. It allows concurrent read and write operations without locking the entire map, providing better performance in multi-threaded environments.

**Q19. What is a hash collision?** A hash collision occurs when two different keys produce the same hash code, resulting in them being placed in the same bucket in a hash table. HashMap handles collisions using linked lists or balanced trees (since Java 8).

**Q20. Does HashMap allow duplicate values?** Yes, HashMap allows duplicate values but not duplicate keys. If a key already exists in the map, the old value is replaced with the new value.

**Q21. Can you store null keys in HashMap?** Yes, HashMap allows one null key and multiple null values.

**Q22. What will happen if two different objects have the same hash code?** If two different objects have the same hash code, they will be placed in the same bucket. HashMap uses the equals() method to distinguish between the objects and store them as separate entries.

**Q23. How will you retrieve the Value object if two Keys have the same hash code?** To retrieve the value, HashMap first finds the bucket using the hash code and then uses the equals() method to find the correct key-value pair within the bucket.

**Q24. How many entries can you put into a HashMap?** The number of entries you can put into a HashMap is limited by the maximum capacity, which is 230230 (approximately 1 billion entries). However, practical limits are determined by available memory.

**Q25. What is the difference between HashMap and Hashtable?**

* **HashMap**: Not synchronized, allows null keys and values, better performance.
* **Hashtable**: Synchronized, does not allow null keys or values, slower performance.

**Q26. Difference between HashMap and ConcurrentHashMap?**

* **HashMap**: Not thread-safe, allows null keys and values.
* **ConcurrentHashMap**: Thread-safe, does not allow null keys or values, better performance in multi-threaded environments.

**Q27. How does HashMap handle null keys in Java?** HashMap handles null keys by storing them in the first bucket (index 0). The put() method checks for null keys and handles them separately.

**Q28. What is the difference between HashMap and HashSet?**

* **HashMap**: Stores key-value pairs, allows one null key, and multiple null values.
* **HashSet**: Stores unique elements, backed by a HashMap, allows one null element.

**Q29. How do you remove an entry from HashMap while iterating over it?** You can use the Iterator's remove() method to safely remove entries while iterating.

Iterator<Map.Entry<String, Integer>> iterator = map.entrySet().iterator();

while (iterator.hasNext()) {

Map.Entry<String, Integer> entry = iterator.next();

if (entry.getValue() < 20) {

iterator.remove();

}

}

**Q30. Does the iterator returned by HashMap is fail-safe or fail-fast?** The iterator returned by HashMap is fail-fast. This means that if the HashMap is structurally modified at any time after the iterator is created, except through the iterator's own remove() method, it will throw a ConcurrentModificationException.

**Q31. Can you store duplicate keys in HashMap? What happens if you try to put a key which already exists in HashMap?** No, you cannot store duplicate keys in HashMap. If you try to put a key that already exists in the HashMap, the old value associated with that key will be replaced by the new value.

**Q32. Difference between HashMap and WeakHashMap?**

* **HashMap**: Uses strong references for keys, meaning the keys are not eligible for garbage collection as long as the map holds a reference to them.
* **WeakHashMap**: Uses weak references for keys, meaning the keys can be garbage collected if there are no strong references to them elsewhere. This allows entries to be removed automatically when the keys are no longer in use.

**Q33. How do you sort a HashMap by values in Java?** You can sort a HashMap by converting it to a list of entries and then sorting the list using a comparator. Finally, you can create a new LinkedHashMap to maintain the sorted order.

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

map.put("A", 10);

map.put("B", 30);

map.put("C", 20);

List<Map.Entry<String, Integer>> list = new ArrayList<>(map.entrySet());

list.sort(Map.Entry.comparingByValue());

Map<String, Integer> sortedMap = new LinkedHashMap<>();

for (Map.Entry<String, Integer> entry : list) {

sortedMap.put(entry.getKey(), entry.getValue());

}

System.out.println("Sorted HashMap: " + sortedMap);

**Q34. Is HashMap thread-safe?** No, HashMap is not thread-safe. If multiple threads access a HashMap concurrently and at least one of the threads modifies the map structurally, it must be synchronized externally.

**Q35. What is the time complexity of get() and put() in HashMap?** The average time complexity of get() and put() operations in HashMap is O(1). However, in the worst case, when there are many hash collisions, the time complexity can degrade to O(n).

**Q36. How does the get() method of HashMap work?** The get() method works by computing the hash code of the key and then finding the corresponding bucket index. It then traverses the linked list or tree at that bucket to find the entry with the matching key using the equals() method.

**Q37. How does the put() method of HashMap work?** The put() method works by computing the hash code of the key and then finding the corresponding bucket index. If the bucket is empty, it creates a new entry and adds it to the bucket. If the bucket already contains entries, it traverses the linked list or tree to check if the key already exists. If it does, it updates the value; otherwise, it adds a new entry to the bucket.

**Q38. As we know that hash codes are generated for each and every key, but what happens when the same hash code is generated for distinct keys?** When the same hash code is generated for distinct keys (hash collision), the keys are placed in the same bucket. HashMap uses a linked list or a balanced tree (since Java 8) to store the entries in the bucket. The equals() method is used to distinguish between the keys and find the correct entry.