Day-9 Interview Questions

1. What is a TreeMap ?

TreeMap class is like HashMap which stores key-value pairs . The major difference is that TreeMap sorts the key in ascending order.

TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.

2. How does TreeMap differ from other map implementations?

A TreeMap is a sorted map implementation in Java that stores key-value pairs in a sorted order based on the natural ordering of keys or a custom comparator. It differs from other map implementations like HashMap in that it maintains the elements in a sorted order, allowing for efficient range queries and operations like getting the smallest or largest key. TreeMap uses a Red-Black Tree as its underlying data structure to maintain the sorting order.

Red-Black Tree: Red-Black tree is a binary search tree in which every node is colored with either red or black. It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity.

3. How do you iterate over the keys, values, and entries in a TreeMap? Is the iteration order guaranteed to be sorted?

You can iterate over the keys using the keySet() method, over the values using the values() method, and over the entries using the entrySet() method of a TreeMap.
The iteration order is guaranteed to be sorted based on the natural ordering of keys or the custom comparator provided during TreeMap instantiation.

4.Why do we need TreeMap when we have sortedMap ?

SortedMap is an interface and TreeMap is the class implementing it .As we know one can not create objects of the interface. Interface tells us which methods a sortedMap implementation should provide .TreeMap is such an implementation.

5.Why Java's TreeMap does not allow an initial size?

HashMap reallocates its internals as the new one gets inserted while TreeMap does not reallocate nodes on adding new ones. Thus, the size of the TreeMap dynamically increases if needed, without shuffling the internals. So it is meaningless to set the initial size of the TreeMap.

6. How does the TreeMap handle duplicate keys? Can you describe the behavior when adding duplicate keys?

TreeMap does not allow duplicate keys. If you attempt to add an element with a key that already exists, the value associated with the existing key will be updated with the new value provided.

7. What happens if you attempt to insert a null key into a TreeMap? How is this different from other map implementations?

TreeMap's sorted nature requires meaningful comparisons between keys to maintain order, and since null cannot be compared with other keys, it's not supported in TreeMap.

8. What is the purpose of the Comparable interface in Java?

The Comparable interface is used to define the natural ordering of objects. It allows objects of a class to be compared and sorted using the default ordering defined by the class itself.

9. Why would you use the Comparator interface over the Comparable interface?

Using the Comparator interface over the Comparable interface offers distinct advantages in certain situations:

- When you need to compare objects from different classes or when you lack access to modify the source code of a class, Comparator allows you to create custom comparison logic externally.

- Comparator permits the definition of multiple comparison criteria for the same class. This flexibility is valuable when you need to sort objects differently based on diverse requirements or user preferences.

- You can dynamically change the sorting behavior at runtime using Comparator, making it suitable for scenarios where sorting criteria need to be adaptable.

In essence, choosing Comparator demonstrates the ability to tailor sorting and comparison logic to specific needs, adapt to changing requirements, and work effectively with diverse classes and scenarios in software development.

10.What is the main difference between the Comparator and Comparable interfaces?

The Comparable interface is used to define the natural ordering of objects within a class itself, while the Comparator interface is used to define external comparison logic that can be applied to classes that may not have implemented Comparable.

11. How can you implement the Comparable interface in a class?
To implement the Comparable interface, the class needs to implement the compareTo() method. This method should compare the current instance with another instance and return a negative, zero, or positive value to indicate their relative ordering.

12. How does the compareTo() method handle null values?

The compareTo() method should handle null values gracefully. If a null value is encountered, the method should be designed to handle it in a consistent and meaningful way. It's important to consider the desired behavior and ensure the method doesn't throw unexpected exceptions.

13. How does the Comparable interface enforce ordering among objects?

The Comparable interface enforces ordering by defining a compareTo() method in the class. This method is responsible for comparing the current object with another object, and it returns a negative integer, zero, or a positive integer depending on whether the current object is less than, equal to, or greater than the other object.

14. Why is it important to override the equals() method in Java?

The equals() method is used to compare the contents of objects for equality. By default, the equals() method in Java checks for reference equality, which may not be

appropriate for all classes. Overriding equals() allows you to define your own logic for determining when two objects are considered equal based on their content.

15. What is the role of the hashCode() method when overriding equals()?

The hashCode() method is used to generate a hash code for an object. When you override equals(), it's a best practice to also override hashCode(). These methods work together to ensure that if two objects are considered equal by equals(), they also produce the same hash code. This is crucial when using objects in hash-based data structures like HashMap or HashSet.

16. What is the difference between equals() method and == operator in java?

equals() Method: It is used to compare the content or attributes of objects for equality based on custom logic defined by the class. It checks if the values within two objects are the same. The behavior of equals() can be customized by overriding it in your class. It's commonly used for comparing the contents of strings, arrays, and user-defined objects.
== Operator: This operator is used to compare primitive data types for their actual values and object references for their memory addresses. It checks if two references point to the exact same object instance in memory. It doesn't consider the content of objects. It's often used to test whether two references refer to the same object in memory, especially when dealing with object references.

17. What is the contract between the equals() and hashCode() methods?

The contract states that if two objects are considered equal according to the equals() method, their hash codes must be equal as well. Conversely, if two objects have the same hash code, they don't necessarily have to be equal according to equals(). Failing to adhere to this contract can result in unexpected behavior when using hash-based collections.

18. How is the default implementation of the hashCode() method in the Object class?

The default hashCode() implementation in the Object class returns a memory address-based hash code, which is typically derived from the internal memory representation of the object. This hash code is not useful for object comparison in most cases.

19. What guidelines should you follow when implementing the hashCode() method?

When implementing hashCode(), follow these guidelines:

- Use the same set of fields used in the equals() method for hash code calculation.
- If a field is used in the equals() comparison, include it in the hash code calculation.
- Choose hashcode computation methods that distribute hash codes well across the hash table to avoid collisions.
- Consider combining hash codes of individual fields in a way that reduces collisions.
- Aim for a consistent hash code value over the object's lifetime, as long as the fields used for hashcode calculation remain constant.

20. Can the hashCode() method return negative values?

Yes, the hashCode() method can return negative values. It's important to note that hash codes are just integers, and negative hash codes are handled correctly by hash-based data structures.