**Java Interview Questions**

**1-Sort map based on values**

**public** **class** SortMapByValues {

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

HashMap<String, Integer> hm = **new** HashMap<String, Integer>();

hm.put("Math", 98);

hm.put("Data Structure", 85);

hm.put("Database", 91);

hm.put("Java", 95);

Map<String, Integer> hm1 = *sortByValueLambdaSteam*(hm);

**for** (Map.Entry<String, Integer> en : hm1.entrySet()) {

System.***out***.println("Key = " + en.getKey() + ", Value = " + en.getValue());

}

}

**public** **static** HashMap<String, Integer> sortByValueOld(HashMap<String, Integer> hm) {

// Create a list from elements of HashMap

List<Map.Entry<String, Integer>> list = **new** LinkedList<Map.Entry<String, Integer>>(hm.entrySet());

// Sort the list

Collections.*sort*(list, **new** Comparator<Map.Entry<String, Integer>>() {

**public** **int** compare(Map.Entry<String, Integer> obj1, Map.Entry<String, Integer> obj2) {

**return** (obj1.getValue()).compareTo(obj2.getValue());

}

});

// put data from sorted list to LinkedHashMap so insertion order is preserved

HashMap<String, Integer> valueSortedMap = **new** LinkedHashMap<String, Integer>();

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

valueSortedMap.put(currentEntry.getKey(), currentEntry.getValue());

}

**return** valueSortedMap;

}

**public** **static** HashMap<String, Integer> sortByValueLambda(HashMap<String, Integer> inputMap) {

// Create a list from elements of HashMap

List<Map.Entry<String, Integer>> list = **new** LinkedList<Map.Entry<String, Integer>>(inputMap.entrySet());

// Sort the list using lambda expression

Collections.*sort*(list, (i1, i2) -> i1.getValue().compareTo(i2.getValue()));

// put data from sorted list to HashMap

HashMap<String, Integer> valueSortedMap = **new** LinkedHashMap<String, Integer>();

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

valueSortedMap.put(aa.getKey(), aa.getValue());

}

**return** valueSortedMap;

}

**public** **static** HashMap<String, Integer> sortByValueLambdaSteam(HashMap<String, Integer> hm) {

HashMap<String, Integer> valueSortedMap = hm.entrySet()

.stream()

.sorted((i1, i2) -> i1.getValue().compareTo(i2.getValue()))

.collect(Collectors.*toMap*(

Map.Entry::getKey,

Map.Entry::getValue,

(e1, e2) -> e1,

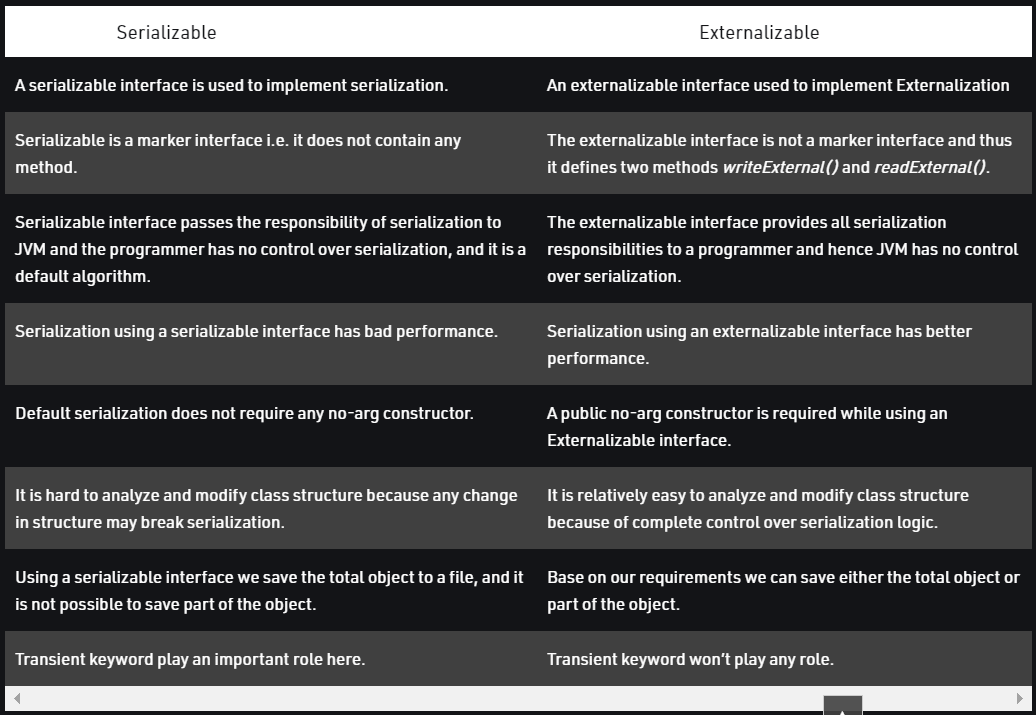
LinkedHashMap::**new**));

**return** valueSortedMap;

}

}

**2-Serialization vs Externalization**



The process of writing the state of an object to a file is called serialization, but strictly speaking, it is the process of converting an object from java supported form into a file supported form or network supported form. By using fileOutputStream and objectOutputStream classes we can implement serialization.

**class** serializableDemo **implements** Serializable {

String name;

**int** age;

**int** jobId;

// Default constructor

**public** serializableDemo(String name, **int** age, **int** jobId) {

**this**.name = name;

**this**.age = age;

**this**.jobId = jobId;

}

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

// Java Object

serializableDemo t1 = **new** serializableDemo("Ram", 34, 2364);

// Serialization -> Saving of object in a file

FileOutputStream fos = **new** FileOutputStream("abc1.ser");

ObjectOutputStream oos = **new** ObjectOutputStream(fos);

oos.writeObject(t1);

System.***out***.println("Object has been serialized");

//Deserialization -> Reading the object from a file

FileInputStream fis = **new** FileInputStream("abc1.ser");

ObjectInputStream ois = **new** ObjectInputStream(fis);

serializableDemo t2 = (serializableDemo) ois.readObject();

System.***out***.println("Object has been deserialized ");

//Printing Deserialized Object

System.***out***.println("Name:" + t2.name + "\n"

+ "Age:" + t2.age + "\n"

+ t2.jobId);

}

}

**public** **class** ExternalizableDemo **implements** Externalizable {

String name;

**int** age;

**int** jobId;

// No-argument constructor

**public** ExternalizableDemo() {

System.***out***.println("Public no-argument constructor");

}

// Default constructor

**public** ExternalizableDemo(String name, **int** age, **int** jobId) {

**this**.name = name;

**this**.age = age;

**this**.jobId = jobId;

}

// Implementing write external method

**public** **void** writeExternal(ObjectOutput out) **throws** IOException {

out.writeObject(name);

out.writeInt(age);

}

// Implementing readExternal method

**public** **void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException {

name = (String) in.readObject();

age = in.readInt();

}

// Main method

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

// Java Object

ExternalizableDemo t1 = **new** ExternalizableDemo("Ram", 35, 23675);

// Serialization -> Saving of object in a file

FileOutputStream fos = **new** FileOutputStream("abc.ser");

ObjectOutputStream oos = **new** ObjectOutputStream(fos);

oos.writeObject(t1);

// Deserialization -> Reading the object from a file

FileInputStream fis = **new** FileInputStream("abc.ser");

ObjectInputStream ois = **new** ObjectInputStream(fis);

ExternalizableDemo t2 = (ExternalizableDemo) ois.readObject();

// Printing Deserialized Object

System.***out***.println("Name :"

+ " " + t2.name + " "

+ "Age :"

+ " " + t2.age);

}

}

**3-Transient vs volatile**

A volatile keyword is used in a multithreading environment where two threads reading and writing the same variable simultaneously. The volatile keyword flushes the changes directly to the main memory instead of the CPU cache (i.e. in the thread stack).

On the other hand, the transient keyword is used during serialization. Fields that are marked as transient cannot be part of the serialization and deserialization. We don't want to save the value of any variable then we use transient keyword with that variable.

| **Sr. No.** | **Key** | **Volatile** | **Transient** |
| --- | --- | --- | --- |
| 1 | Basic | Volatile keyword is used to flush changes directly to the main memory | The transient keyword is used to exclude variable during serialization |
| 2. | Default value | Volatile are not initialized with a default value. | During deserialization, transient variables are initialized with a default value |
| 3 | Static | Volatile can be used with a static variable. | Transient cannot be used with the static keyword |
| 4 | Final | Volatile can be used with the final keyword | Transient cannot be used with the final keyword |

// A sample class that uses transient keyword to skip their serialization.

**class** TransientExample **implements** Serializable {

**transient** **int** age;

// serialize other fields

**private** String name;

**private** String address;

// other code

}

**public** **class** VolatileExmaple **extends** Thread {

**volatile** **boolean** isRunning = **true**;

@Override

**public** **void** run() {

**long** count = 0;

**while** (isRunning) {

count++;

}

System.***out***.println("Thread terminated with count= " + count);

}

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

VolatileExmaple t = **new** VolatileExmaple();

t.start();

Thread.*sleep*(2000);

t.isRunning = **false**;

t.join();

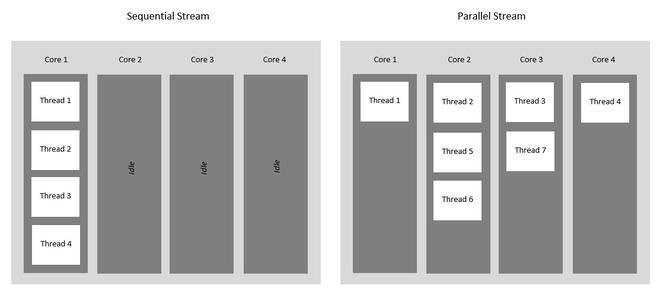
System.***out***.println("isRunning set to " + t.isRunning);

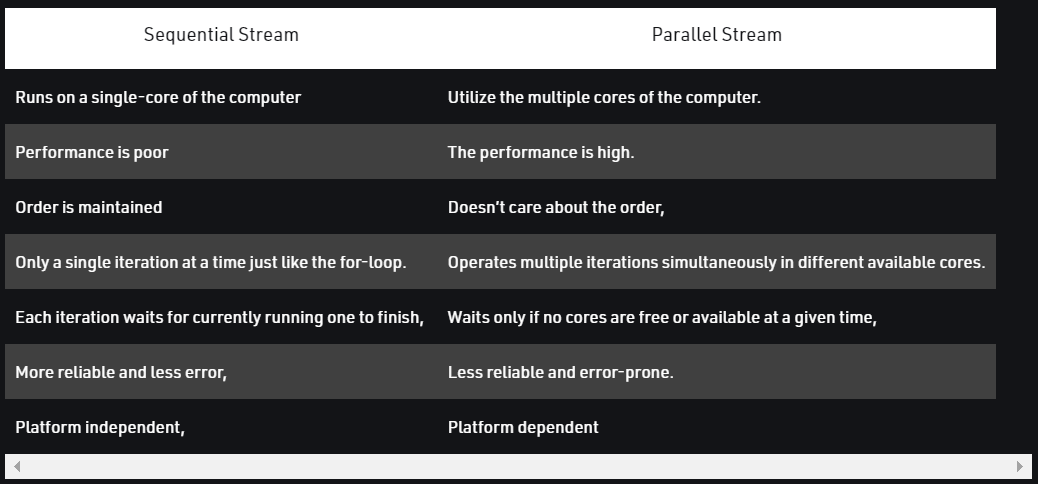
}

}

**4-Stream vs parallel stream**

A stream in Java is a sequence of objects which operates on a data source such as an array or a collection and supports various methods. It was introduced in Java 8’s java.util.stream package. Stream supports many aggregate operations like filter, map, limit, reduce, find, and match to customize the original data into a different form according to the need of the programmer. The operations performed on a stream do not modify its source hence a new stream is created according to the operation applied to it. The new data is a transformed copy of the original form.



****

**class** SequentialStreamDemo {

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

// create a list

List<String> list = Arrays.*asList*("Hello ", "G", "E", "E", "K", "S!");

list.stream().forEach(System.***out***::print);

}

}

Output: -

**Hello GEEKS!**

**class** ParallelStreamExample {

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

// create a list

List<String> list = Arrays.*asList*("Hello ", "G", "E", "E", "K", "S!");

**list.parallelStream().forEach(System.*out*::print);**

}

}

Output: -

**EGS!KEHello**

**class** ParallelStreamWithOrderedIteration {

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

// create a list

List<String> list = Arrays.*asList*("Hello ", "G", "E", "E", "K", "S!");

**list.parallelStream().forEachOrdered(System.*out*::print);**

}

}

Output: -

**Hello GEEKS!**

**5-Java is pass by value or pass by reference?**

Data is shared between functions by passing parameters. Now, there are 2 ways of passing parameters:

1. Pass By Value: The pass by value method copies the value of actual parameters. The called function creates its own copy of argument values and then uses them. Since the work is done on a copy, the original parameter does not see the changes.
2. Pass By Reference: The pass by reference method passes the parameters as a reference(address) of the original variable. The called function does not create its own copy, rather, it refers to the original values only. Hence, the changes made in the called function will be reflected in the original parameter as well.

Java follows the following rules in storing variables:

1. Local variables like primitives and object references are created on Stack memory.
2. Objects are created on Heap memory.

Now coming to the main question: Is Java Pass by Value or Pass by Reference?

**Java Always follows Pass by Value**

**public** **class** PassByValueReference {

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

**int** data = 10;

List<String> fruits = **new** ArrayList<>();

fruits.add("apple");

fruits.add("mango");

fruits.add("grapes");

System.***out***.println("Before processing pass by value");

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

System.***out***.println("After processing pass by value");

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

System.***out***.println("Before processing pass by ref");

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

System.***out***.println("After processing pass by ref");

*processPassByRef*(fruits);

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

}

**public** **static** **void** processPassByValue(**int** data) {

data = data \* 10;

}

**public** **static** **void** processPassByRef(List<String> fruitsRef) {

fruitsRef.add("Banana");

}

}

Before processing pass by value

10

After processing pass by value

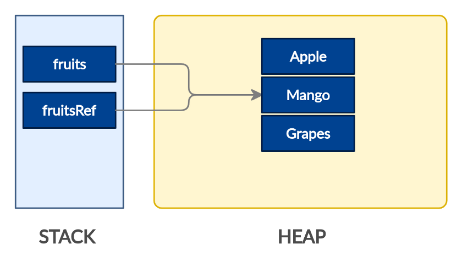
10

Before processing pass by ref

[apple, mango, grapes]

After processing pass by ref

[apple, mango, grapes, Banana]



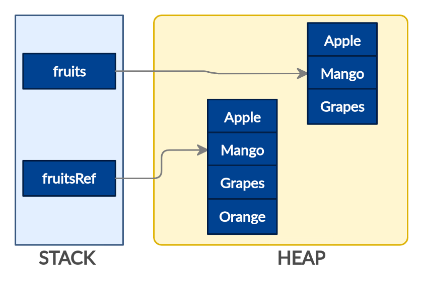
**In Below case:**

**public** **static** **void** processPassByRef(List<String> fruitsRef) {

fruitsRef = **new** ArrayList<>();

fruitsRef.add("Banana");

}



**In the above program, “fruits” is passed to the processData function. “fruitsRef” is a copy of the “fruits” param. Both fruits and fruitsRef are created on Stack. They are two different references. But the interesting point is, it points to the same underlying object in Heap. So, any change that you make using one reference is going to impact the common object.**

**6- Employees Sorting question**

**public** **class** EmployeeSortingQuestion {

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

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

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

Employee emp3 = **new** Employee(3, "abc");

Employee emp4 = **new** Employee(4, "xyz");

List<Employee> empList = **new** ArrayList<>();

empList.add(emp1);

empList.add(emp2);

empList.add(emp3);

empList.add(emp4);

System.***out***.println("Before Sorting");

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

System.***out***.println("After EmpNameSorter");

Collections.*sort*(empList, **new** EmpNameSorter());

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

System.***out***.println("After Java8 EmployeeSorting");

empList.sort(Comparator.*comparing*(e -> e.getEmpName()));

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

System.***out***.println("After Java8 EmployeeSorting reversed");

Comparator<Employee> empReverseComparator = Comparator.*comparing*(e -> e.getEmpName());

empList.sort(empReverseComparator.reversed());

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

}

}

**class** EmpNameSorter **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee obj1, Employee obj2) {

**return** obj1.getEmpName().compareTo(obj2.getEmpName());

}

}

**class** Employee {

**int** empId;

String empName;

**public** Employee(**int** empId, String empName) {

**super**();

**this**.empId = empId;

**this**.empName = empName;

}

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getEmpName() {

**return** empName;

}

**public** **void** setEmpName(String empName) {

**this**.empName = empName;

}

@Override

**public** String toString() {

**return** "[empId=" + empId + "empName=" + empName +"]";

}

}

Before Sorting

[[empId=1empName=ghi], [empId=2empName=def], [empId=3empName=abc], [empId=4empName=xyz]]

After EmpNameSorter

[[empId=3empName=abc], [empId=2empName=def], [empId=1empName=ghi], [empId=4empName=xyz]]

After Java8 EmployeeSorting

[[empId=3empName=abc], [empId=2empName=def], [empId=1empName=ghi], [empId=4empName=xyz]]

After Java8 EmployeeSorting reversed

[[empId=4empName=xyz], [empId=1empName=ghi], [empId=2empName=def], [empId=3empName=abc]]

**7- Remove duplicates in ArrayList of Employees**

**public** **class** RemoveArrayListDuplicates {

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

List<Employee> employeeList = **new** ArrayList<>();

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

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

Employee emp3 = **new** Employee(1, "abc");

employeeList.add(emp1);

employeeList.add(emp2);

employeeList.add(emp3);

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

List<Employee> listWithoutDuplicates = **new** ArrayList<>(**new** HashSet<>(employeeList));

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

}

}

[[empId=1empName=abc], [empId=2empName=def], [empId=1empName=abc]]

[[empId=1empName=abc], [empId=2empName=def]]

Note: hashCode() and equals() should be implemented in Employee class if this should work.

**8- Singleton Class**

**public** **class** SingletonClass **implements** Serializable, Cloneable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

// Lazy initialization

**private** **static** SingletonClass *singleInstance* = **null**;

// Prevent from Reflection.

**private** SingletonClass() {

**if** (*singleInstance* != **null**) {

**throw** **new** InstantiationError("Creating of this object is not allowed.");

}

}

// Use of double locking.

**public** **static** SingletonClass getInstance() {

**if** (*singleInstance* == **null**) {

**synchronized** (SingletonClass.**class**) {

**if** (*singleInstance* == **null**) {

*singleInstance* = **new** SingletonClass();

}

}

}

**return** *singleInstance*;

}

// Clone prevention

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

**return** *getInstance*();

}

// Deserialization prevention

**protected** Object readResolve() {

**return** *getInstance*();

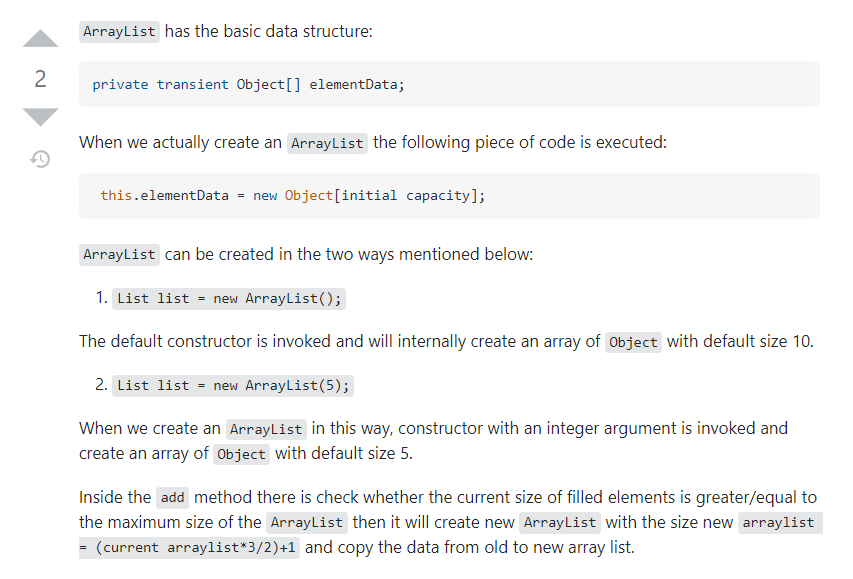
}

}

**Double Locking:**

Here, we run into a problem. Suppose that there are two threads running. Both can get inside of the if statement concurrently when the instance is null. Then, one thread enters the synchronized block to initialize the instance, while the other is blocked. When the first thread exits in the synchronized block, the waiting thread enters and creates another singleton object. Note that when the second thread enters the synchronized block, it does not check to see if the instance is non-null.

**9-Internal working of ArrayList**

****

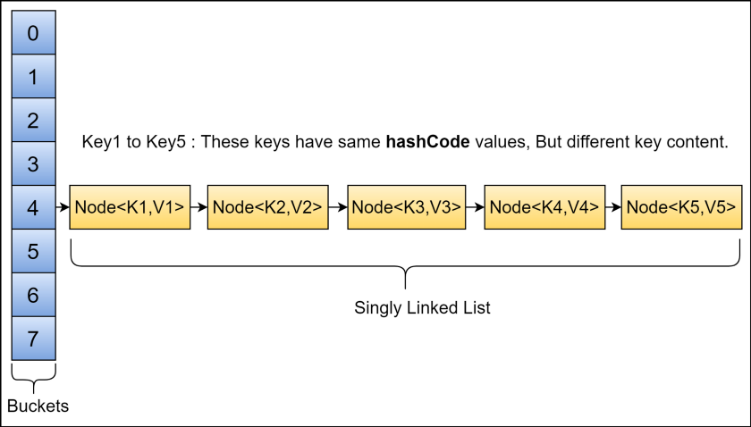
**10- Internal working of HashMap**

***Few important points to about HashMap:***

1. HashMap uses its static inner class Node<K,V> for storing the entries into the map.
2. HashMap allows at most one null key and multiple null values.
3. The HashMap class does not preserve the order of insertion of entries into the map.
4. HashMap has multiple buckets or bins which contain a head reference to a singly linked list. That means there would be as many linked lists as there are buckets. Initially, it has a bucket size of 16 which grows to 32 when the number of entries in the map crosses the 75%. (That means after inserting in 12 buckets bucket size becomes 32)
5. HashMap is almost similar to Hashtable except that it’s unsynchronized and allows at max one null key and multiple null values.
6. HashMap uses hashCode() and equals() methods on keys for the get and put operations. So HashMap key objects should provide a good implementation of these methods.
7. That’s why the Wrapper classes like Integer and String classes are a good choice for keys for HashMap as they are immutable and their object state won’t change over the course of the execution of the program.

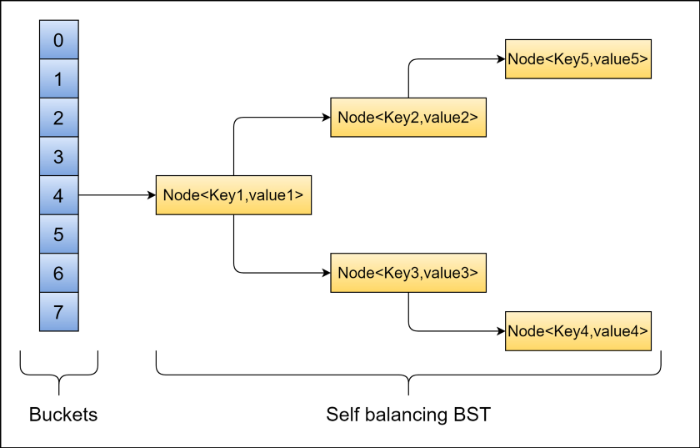
***Now Let’s Look at the Internal Working of HashMap:***

* HashMap uses its static inner class Node<K,V> for storing map entries. That means each entry in hashMap is a Node. Internally HashMap uses a hashCode of the key Object and this hashCode is further used by the hash function to find the index of the bucket where the new entry can be added.
* HashMap uses multiple buckets and each bucket points to a Singly Linked List where the entries (nodes) are stored.
* Once the bucket is identified by the hash function using hashcode, then hashCode is used to check if there is already a key with the same hashCode or not in the bucket (singly linked list).
* If there already exists a key with the same hashCode, then the equals() method is used on the keys. If the equals method returns true, that means there is already a node with the same key and hence the value against that key is overwritten in the entry(node), otherwise, a new node is created and added to this Singly Linked List of that bucket.
* If there is no key with the same hashCode in the bucket found by the hash function then the new Node is added into the bucket found.



***A Very Important Note That You Must Know:***

Before java 8, singly-linked lists were used for storing the nodes. But this implementation has changed to self-balancing BST after a threshold is crossed (static final int TREEIFY\_THRESHOLD = 8;). The motive behind this change is that HashMap buckets normally use linked lists, but for the linked lists the worst-case time is O(n) for lookup. Also note that Ordinary binary search trees have pathological cases where they become O(n) [basically BST becomes skewed], but red-black/AVL trees are specifically designed to prevent these cases. In a HashMap with linked lists, if we have a really awful hash function, we could end up with all the items hashing to the same bucket and get O(n) lookup, But it seems like with this red-black/AVL tree scheme, even if all the items hashed into the same bucket, we would get O(log n) lookup in worst of worst scenario.



**Re-Hashing:**

Whenever the number of entries in the HashMap crosses the threshold value then the bucket size of the HashMap is doubled and rehashing is performed and all the already existing entries of the map are copied and new entries are added to this increased HashMap.

Threshold value = Bucket size \* Load factor

Eg. If bucket size is 16 and the load factor is 0.75 then the threshold value is 12.

**Time Complexity:**

1. In a fairly distributed HashMap where the entries go to all the buckets in such a scenario, the HashMap has O(1) time for search, insertion, and deletion operations.
2. In the worst case, where all the entries go to the same bucket and the singly linked list stores these entries, O (n) time is required for operations like search, insert, and delete.
3. In a case where the threshold for converting this linked list to a self-balancing binary search tree (i.e., AVL/Red black) is used then for the operations, search, insert and delete O(log N) is required as AVL/Red Black tree has a max length of log N in the worst case.

**11- Why HashMap key should be immutable?**

1. If key's hash code changes after the key-value pair (Entry) is stored in HashMap, the map will not be able to retrieve the Entry.
2. **Key's hashcode can change if the key object is mutable.**
3. **Mutable keys in HashMap can result in data loss.**
4. String, Integer and other wrapper classes are natural candidates of HashMap key, and String is most frequently used key as well because String is immutable and final, and overrides equals and hashcode() method .

**12- Use of private constructor in java**

1. While implementing Singleton class.
2. If you have a utility class that generally has only public static methods and you do not want to create an object to that utility class.

13- Immutable Class implementation.