**List:**

Lists maintain the order of elements based on their insertion order.

Elements can be accessed, inserted, or removed by their index.

Lists allow duplicate elements.

Common implementation classes are ArrayList, LinkedList, Vector, Stack.

**Set:**

Sets do not maintain insertion order (except for LinkedHashSet).

TreeSet maintains elements in their natural ordering or by a comparator provided at set creation.

Sets do not allow duplicate elements.

Common implementation classes are HashSet, LinkedHashSet,TreeSet.

**Map:**

Map is an interface that represents a collection of key-value pairs.

Common implementation classes are HashMap, TreeMap, LinkedHashMap, Hashtable, and ConcurrentHashMap.

No specific ordering it is depends on implementation.

Performance depends on the specific implementation.

Null handling also depends on the specific implementation.

**ArrayList:**

A resizable array implementation of the List interface.

it allows direct access via indices.

Provides fast random access to elements but slow insertion and removal (except at the end).

May waste memory due to array resizing.

Not synchronized. Use Collections.synchronizedList if thread safety is required.

O(1) time complexity for acess and O(n) time complexity for insertion and deletion.

**LinkedList:**

A doubly-linked list implementation of both List and Deque interfaces.

Provides fast insertion and removal but slower random access.

Slower access because it requires traversing the list from the beginning or end to reach an element.

No need for resizing operations.

Less memory due to storage overhead for node pointers.

Not synchronized. Use Collections.synchronizedList if thread safety is required.

O(n) time complexity for acess and O(1) time complexity for insertion and deletion.

**Vector:**

A synchronized, resizable array implementation.

**Stack:**

A subclass of Vector that implements a last-in-first-out (LIFO) stack.

**HashSet:**

Implements the Set interface, backed by a hash table.

Offers constant time performance for basic operations (add, remove, contains).

Does not maintain any order for the elements.

The order of elements is determined by the hash codes of the elements.

Provides constant-time performance for basic operations like add, remove.

Allows one null element.

O(1) time complexity for basic operations.

**TreeSet:**

Implements the Set interface, backed by a red-black tree.

TreeSet is an implementation of the NavigableSet interface that uses a red-black tree (a self-balancing binary search tree).

It orders its elements based on their natural ordering or by a comparator provided at set creation time.

Maintains elements in sorted order, either by natural order (using Comparable) or by a specified Comparator.

Does not allow null elements. Attempting to add null will result in a NullPointerException.

O(log n) time complexity for basic operations.

**LinkedHashSet:**

Maintains insertion order because doubly-linked list running through all of its entries.

Slightly slower than HashSet due to the additional overhead of maintaining the linked list.

Allows one null element.

O(1) time complexity for basic operations.

**HashMap:**

Implements the Map interface, backed by a hash table.

Uses a hash table to store key-value pairs.

Does not maintain any order of its entries. The order can change over time.

Allows one null key and multiple null values.

Not synchronized. External synchronization is needed for concurrent access.

O(1) time complexity for basic operations.

**TreeMap:**

TreeMap is a concrete implementation of the NavigableMap interface that uses a red-black tree data structure for storage.

Maintains the keys in sorted order, either by natural ordering or by a specified comparator.

Does not allow null keys. Allows multiple null values.

Not synchronized. External synchronization is needed for concurrent access.

O(log n) time complexity for basic operations.

**LinkedHashMap:**

Uses a hash table and a linked list data structures internally.

Maintains insertion order or access order.

Allows one null key and multiple null values.

O(1) time complexity for basic operations.

**ConcurrentHashMap:**

ConcurrentHashMap is a thread-safe implementation of a hash table in Java, designed for use in concurrent environments.

ConcurrentHashMap allows multiple threads to read and write without locking the entire map.

ConcurrentHashMap uses a technique called lock striping, which divides the map into segments, each protected by its own lock. This allows multiple threads to access different segments simultaneously, reducing contention and improving concurrency.

ConcurrentHashMap provides atomic operations like putIfAbsent, remove, replace, compute, merge, and other bulk operations.

Provides better concurrency performance compared to HashMap in a multithreaded environment due to its internal locking strategy.

Does not maintain any specific order of entries. The order can change over time.

Does not allow null keys or null values. Any attempt to insert a null key or value will result in a NullPointerException.

Designed for concurrent use. Multiple threads can read and write without causing ConcurrentModificationException.

**How Hashmap Works Internally?**

A HashMap consists of an array of buckets.

Each bucket is essentially a linked list from Java 8 each bucket is balanced tree that holds entries like key-value pairs.

Each entry contains a key, a value, a hash value, and a reference to the next entry in the bucket.

When you add an entry to the HashMap, the key’s hashCode() method is called to compute an integer hash code.

This hash code is then transformed into a bucket index using another hashing function to reduce collisions and evenly distribute entries.

The index is calculated using (n - 1) & hash, where n is the number of buckets i.e the length of the array. This ensures the index falls within the bounds of the array.

HashMap handles collisions using chaining. Multiple entries can be stored in a single bucket by linking them in a list or tree.

Starting from Java 8, if the number of entries in a bucket exceeds a certain threshold (default is 8), the linked list is transformed into a balanced tree (Red-Black tree) to improve performance.

HashMap has a load factor that is used for hashmap resizing, a measure of how full the hash table can get before it needs to be resized. The default load factor is 0.75.

When the number of entries exceeds the product of the load factor and the current capacity (number of buckets), the HashMap is resized. The capacity is doubled, and the existing entries are rehashed and distributed into the new array of buckets.

So finally while adding entries into hashmap we use put method that time hashcode method of key is computed. That hash is transformed into a bucket index.If the bucket at that index is empty, a new entry is created and placed in the bucket.If the bucket is not empty (collision), the HashMap checks for the existence of the key. If the key exists, the value is updated. If the key does not exist, the new entry is appended to the linked list or tree in that bucket.

To retrieving entries we use get method. The hascode method of the key is computed. The hash is transformed into a bucket index.The HashMap looks through the linked list or tree in the bucket to find the entry with the matching key. If found, the value is returned.

**what is hash table?**

A hash table is a data structure that provides fast insertion, deletion, and lookup of 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.

The core structure of a hash table is an array where each position (bucket) can store one or more entries.

Each entry typically contains a key, a value, and a hash value.

This is a function that takes a key and computes an integer hash code. The hash code is then mapped to an index in the array of buckets.

The index is usually calculated by taking the modulo of the hash code with the array size: index = hashCode % arraySize. This ensures the index falls within the array's bounds.

While insert(put) operation it computes the hash code of the key. Determine the bucket index using the hash code. If the bucket is empty, place the entry there. If the bucket is occupied, handle the collision by chaining or open addressing.

Collisions occur when two keys hash to the same index. There are several strategies to handle collisions like chaining, open addressing etc.

In chaining, each bucket contains a linked list (or other collection) of entries. When a collision occurs, the new entry is added to the list in that bucket.

In open addressing, all entries are stored in the array itself. When a collision occurs, the hash table searches for the next available bucket (e.g., linear probing, quadratic probing, double hashing).

While lookup(get) operation it retrieves a value by key. For that it compute the hash code of the key. Determine the bucket index using the hash code. Search through the bucket to find the entry with the matching key. Return the corresponding value.

While deletion(remove) operation it remove a key-value pair. For that it compute the hash code of the key. Determine the bucket index using the hash code. Search through the bucket to find the entry with the matching key. Remove the entry.

The load factor is a measure of how full the hash table is. It’s calculated as the number of entries divided by the number of buckets. A high load factor means many collisions, while a low load factor means many empty buckets.

When the load factor exceeds a certain threshold, the hash table is resized (typically doubled in size). All existing entries are rehashed and redistributed into the new array.

Advantages are fast access and efficient space usage. It is fast in insert, delete, and lookup operations. It is properly designed hash functions and load factors lead to efficient space utilization.

Disadvantages are collisions, memory overhead and non-sequential access. Handling collisions can be complex and may degrade performance. Maintaining a large array with many empty buckets can be inefficient. Hash tables do not maintain any order of the keys.