**ArrayList vs Vector**

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| ArrayList is**not synchronized**, which means multiple threads can work on arrayList at the same time. | Vector is **synchronized**, which means only one thread at a time can access the code. |
| **ArrayList is faster**, since it is non-synchronized | Vector operations give slower performance since they are synchronized (thread-safe). |
| ArrayList increments 50% of the current array size if the number of elements exceeds its capacity | Vector increments 100% – essentially doubling the current array size. |

**HashSet vs SortedSet**

|  |  |
| --- | --- |
| **HashSet** | **SortedSet** |
| HashSet holds a set of objects, but in a way that it allows you to easily and quickly determine whether an object is already in the set or not. | SortedSet is an ordered set collection. You want to store elements in a sorted order and also eliminate all duplicates from the data structure then SortedSet is preferable. |
| HashSet is an unordered collection, but since it uses a hash-based implementation, these operations are O(1). | SortedSet does not include hashing, meaning that it has to do linear searches for lookups. SortedSet is much slower than the HashSet. |
| HashSet uses a **hash-table** data structure to store data. | SortedSet uses a **red-back tree** data structure which is a balanced binary tree. |

**HashSet vs TreeSet**

|  |  |
| --- | --- |
| **HashSet** | **TreeSet** |
| HashSet is faster than TreeSet. For operations like search, insert and delete. It takes constant time for these operations on average. | TreeSet takes O(Log n) for search, insert and delete which is higher than HashSet. But TreeSet keeps sorted data. |
| Elements in HashSet are not ordered. | TreeSet maintains objects in Sorted order. TreeSet elements are sorted in ascending order by default. |
| HashSet allows null object. | TreeSet doesn’t allow null Object and throw NullPointerException, |

**Array vs List**

|  |  |
| --- | --- |
| **Array** | **List** |
| An array is a consecutive segment of memory that occupies n\*size(type) bytes, where n is the length of the array and size(type) is the size in memory required | Most list implementations are a combination of nodes that store: 1.- a single value and, 2.- one or more pointers that keep the nodes connected between them. |
| This means that if you want to create an array of 100 ints, and each int occupies 4 bytes, you will need to have an unused memory segment of at least 400 bytes (100\*4). | This means that you don't need a big chunk of available memory with a size big enough to hold all your data, as the nodes can be scattered through your memory |
| This also means that array are pretty inexpensive to create, release and use because they are actually chunks of memory. | The list structure is defined by the connections created by the pointers. |

**List vs Set**

|  |  |
| --- | --- |
| **List** | **Set** |
| List stores element according to the insertion order. | Sets are stored in random order it uses hashing technique |
| While iterating elements will be fetched in order of insertion. It allows duplicate elements. | Elements are fetched in random order.  It does not allow duplicate values. |
| Any number of NULL values are allowed in the list | Maximum one NULL value is allowed |

**NavigableSet vs NavigableMap**

|  |  |
| --- | --- |
| **NavigableSet** | **NavigableMap** |
| NavigableSet represents a navigable set in Java Collection Framework. The NavigableSet interface inherits from the SortedSet interface. | NavigableMap is an extension of SortedMap which provides convenient navigation method like lowerKey, floorKey, ceilingKey and higherKey, |
| It behaves like a SortedSet with the exception that we have navigation methods available in addition to the sorting mechanisms of the SortedSet. | Along with these popular navigation method it also provide ways to create a Sub Map from existing Map in Java |
| Methods [subSet(E, E)](https://docs.oracle.com/javase/8/docs/api/java/util/NavigableSet.html#subSet-E-E-), [headSet(E)](https://docs.oracle.com/javase/8/docs/api/java/util/NavigableSet.html#headSet-E-), and [tailSet(E)](https://docs.oracle.com/javase/8/docs/api/java/util/NavigableSet.html#tailSet-E-) are specified to return SortedSet to allow existing implementations of SortedSet to be compatibly retrofitted to implement NavigableSet | Methods [subMap(K, K)](https://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html#subMap(K,%20K)), [headMap(K)](https://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html#headMap(K)), and [tailMap(K)](https://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html#tailMap(K)) are specified to return SortedMap to allow existing implementations of SortedMap to be compatibly retrofitted to implement NavigableMap |