|  |  |
| --- | --- |
| **LinkedList** | **ArrayList** |
| Get(i) cost O(N), use doubly linked list. | Get(i) cost O(1), use array |
| remove() cost O(1) | Remove cost O(N) |
| Add() cost O(1) | Add() cost O(N) |
| Higher memory consumption | Lower memory consumption |
| Implement List interface | |
| Maintain elements insertion order | |
| Non-synchronized, could be synchronized by Collections.synchronizedList | |
| Fail-fast | |

**Explain**

**1) Get**: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching. LinkedList implements doubly linked list which requires the traversal through all the elements for searching.

**2) Remove & Add**: LinkedList’s each element maintains two pointers which points to the both neighbor elements. Hence removal only requires change in the pointer location in the two neighbor nodes of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

**3) Memory Overhead**: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes.

**When to use LinkedList and ArrayList?**

If there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| Stores primitive data types and also objects | Stores only objects |
| Defined in Java language itself as a fundamental data structure | Belongs to collections framework |
| Fixed size | Resizable. Elements can be added or removed |
| Stores similar data of one type | Can store heterogeneous data types |
| It is not a class | It is a class with many methods |
| Cannot be synchronized | Can be obtained a synchronized version |
| Elements retrieved with for loop | Can be retrieved with for loop and iterators |
| Elements accessible with index number | Accessing methods like get() etc. are available |
| Can be multidimensional | — |

|  |  |
| --- | --- |
| **HashSet** | **TreeSet** |
| Add, remove, contains, size cost O(1) | Add remove contains size cost O(logN) |
| Not maintain insertion order | Maintain ascending order |
| Not allow duplicate elements | |
| Use HashSet firstly and then add elements to TreeSet is faster than use TreeSet directly | |
| Non-synchronized and not thread-safe | |

**List VS Set VS Map Interfaces**

1. **Duplicity:**

List allows duplicate elements. Any number of duplicate elements can be inserted into the list without affecting the same existing values and their indexes.  
Set doesn’t allow duplicates. Set and all of the classes which implements Set interface should have unique elements.  
Map stored the elements as key & value pair. Map doesn’t allow duplicate keys while it allows duplicate values.

1. **Null values:**

List allows any number of null values.  
Set allows single null value at most.  
Map can have single null key at most and any number of null values.

1. **Order**:

List and all of its implementation classes maintains the insertion order.  
Set doesn’t maintain any order; still few of its classes sort the elements in an order such as LinkedHashSet maintains the elements in insertion order.  
Similar to Set Map also doesn’t stores the elements in an order, however few of its classes does the same. For e.g. TreeMap sorts the map in the ascending order of keys and LinkedHashMap sorts the elements in the insertion order, the order in which the elements got added to the LinkedHashMap.

1. **Commonly used classes:**

List: [**ArrayList**](http://beginnersbook.com/2013/12/java-arraylist/), [**LinkedList**](http://beginnersbook.com/2014/08/java-linkedlist-class/) etc.  
Set: [**HashSet**](http://beginnersbook.com/2013/12/hashset-class-in-java-with-example/), [**LinkedHashSet**](http://beginnersbook.com/2013/12/linkedhashset-class-in-java-with-example/), [**TreeSet**](http://beginnersbook.com/2013/12/treeset-class-in-java-with-example/), SortedSet etc.  
Map: [**HashMap**](http://beginnersbook.com/2014/08/java-hashmap-class/), [**TreeMap**](http://beginnersbook.com/2013/12/treemap-in-java-with-example/), WeakHashMap, [**LinkedHashMap**](http://beginnersbook.com/2013/12/linkedhashmap-in-java/), IdentityHashMap etc.

**5) When to use List, Map, Set?**

1. If you do not want to have duplicate values in the database then Set should be your first choice as all of its classes do not allow duplicates.  
2. If there is a need of frequent search operations based on the index values then List (ArrayList) is a better choice.  
3. If there is a need of maintaining the insertion order then also the List is a preferred collection interface.  
4. If the requirement is to have the key & value mappings in the database then Map is your best bet.

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| **Non** **synchronized** and **not** **thread** **safe**, could be synchronized by Collections.synchronizedMap(hashMap) | **Synchronized** and **thread** **safe** |
| Allow one **null** key and any **null** value | Not allow **null** key and value |
| Use **iterator** to iterator | Use **enumerator** to iterator like vector |
| **Iterator** is **Fail-fast**: if is structurally modified at any time after the iterator is created in any way except the iterator's own remove method, the iterator will throw ConcurrentModification Exception. | **Enumerator** is **Not fail-fast** |
| **Faster**, **less** **memory** is single threat environment because of **unsynchronized** | **Slower**, **more** **memory** |
| Subclass of **AbstractMap** class | Subclass of **Dictionary** class(Obsolete) |
| **Insertion Order** is not guaranteed | |
| Implements **Map** interface | |
| Put and Get methods cost constant time assuming that the objects are distributed uniformly across the bucket. | |
| Works on the principle of Hashing | |

**When to use HashMap and Hashtable?**  
**1. Single Threaded Application**: HashMap should be preferred over Hashtable for the non-threaded applications. In simple words, use HashMap in unsynchronized or single threaded applications.  
**2. Multi-Threaded Application**: We should avoid using Hashtable, as the class is now obsolete in latest Jdk 1.8. Oracle has provided a better replacement of Hashtable named ConcurrentHashMap. For multithreaded application prefer ConcurrentHashMap instead of Hashtable.

|  |  |
| --- | --- |
| **Binary Search Tree** | **Heap** |
| Guarantee order of left and right part | Guarantee order of higher level and lower level |
| Cost O(logN) in searching and insertion | Cost O(1) in search min/max, O(logN) in insertion |
| If want to sort elements then use BST | |
| If care more about min or max then use heap | |

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| No guarantee order | Sorted according to the natural ordering of keys according to the compareTo() method |
| Get/Put/Remove/contains O(1) | O(logN) |
| Map Interface | Map, SortedMap, NavigableMap Interfaces |
| One null key and any null value allowed | Only values |
| Fail-fast | |
| Bucket | Red-black tree |
| Not synchronized | |

|  |  |
| --- | --- |
| **HashTable** | **Binary Tree** |
| Complicate to implement | Easy to implement |
| Add, get, remove cost O(1) | Add get remove cost O(logN) |
| No orders | BST has orders |

|  |  |
| --- | --- |
| **Quick Sort** | **Merge Sort** |
| Worst case O(N2), AVE O(NlogN) | O(NlogN) |
| Cost more compares but less space | Cost less compares but more space |
| Shuffle is important, faster | Needn’t shuffle, slower |

|  |  |
| --- | --- |
| **Stack** | **Queue** |
| Insert to end and remove from end | Insert to end and remove from head |
| Use one pointer “Top” | Use two pointers “Front”, “Rear” |
| No space wastage | Have space wastage |
| Cost O(1) in add and remove | |