

ArrayList and LinkedList

* Both are implementation of List Interface with their own use cases and own properties
* ArrayList using an array to implement list Interface.
* Arrays are fixed size in java with initial capacity.
* Let’s evaluate this data structure with respect to 3 min operation
  + Add
  + Getting one by Index
  + Removing by index

Add

The time complexity of add operation is O (1) – Pretty Fast.

To add a new item, we need to initialize a band new array with more capacity (grow method) and copy all existing items to the new Array. Then time complexity becomes O(n) in the worst case because of a copy.

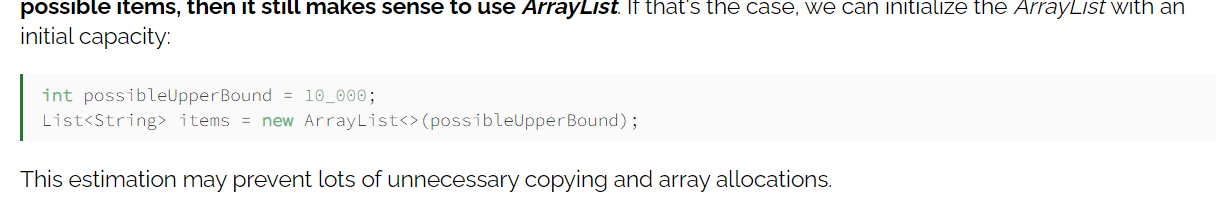
Access by Index:

* Accessing items by indices is where the ArrayList shines.
* Time Complexity for accessing index operation is always O (1).

Remove by Index:

* Obviously the nearer the element to the start of the array, the more elements we should move.
* Time complexity is O (1) in the best case. O(n) in the average and worst case.

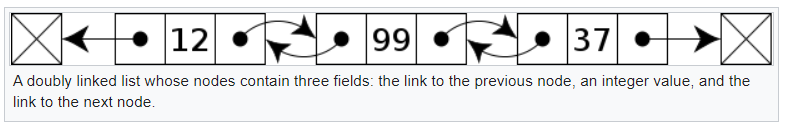
It will be a sensible choice when the number of reads is far more than the number of writes.



It’s not possible to store more than 2power 32 elements in Java array and consequently in ArrayList.

LinkedList:

* It uses a collection of linked nodes to store and retrieve elements
* Each nodes have 2 pointers:
  + One pointing to next element
  + Another referring to previous one.



Add:

* Add a new node first and then link the last node to the current node and update the pointer.
* Time complexity of adding an element is O (1).

Accessing by Index:

* It does not support random access. We need to traverse some portion of the list manually.
* When the requested item is near the start or end of the list then time complexity is O (1).

Removing Index:

* We should find the item and then remove it from the list.
* O (1) in the best case and O (n) in the worst case.

LinkedList is more suitable when the addition rate is much higher than the reading rate.

* Linked List also implements Deque Interface – supports efficient access of both ends of the collection.

List VS Set

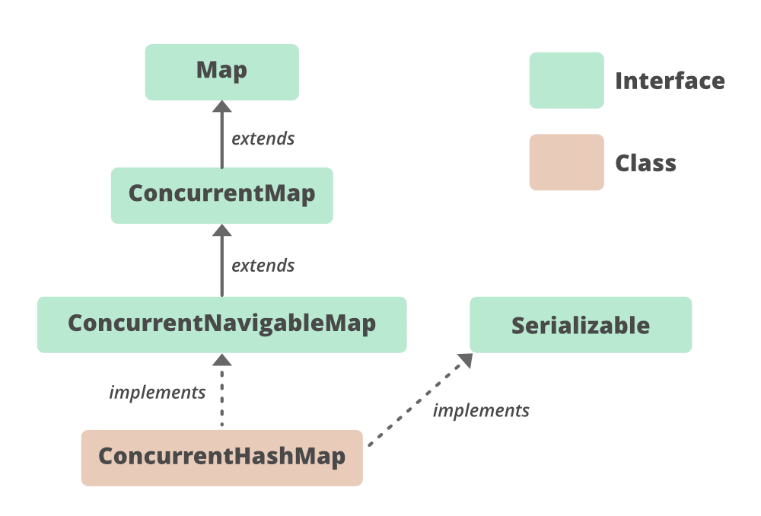
* Interface extends Collection
* Implemented by LinkedList and ArrayList, Vector and Stack
* Ordered Collection
* User can access elements using an integer index
* It allows duplicate elements
* List has Iterator, add, remove and equals and hashcode method
* It has provided a special Iterator – ListIterator (allows insertion and replacement, bidirectional access in addition to a normal Iterator interface.

Set

* Interface extends Collection
* Implemented by HashSet(unordered) , LinkedHashSet(ordered) , TreeSet(sorted by natural order or by provided comparator)
* No duplicates
* It doesn’t allow null value and duplicate value

<https://www.javainuse.com/java/javaConcurrentHashMap>

Concurrent hashmap Internal working



* Hashtable is synchronized , it has to wait until other thread completes it task.
* It isthread-safe without synchronizing the whole map.
* Does not allow null as key. Read is much faster.
* Hashmap is not thread-safe and gives parallel access to multiple threads reading hashmap simultaneously.
* Concurrenthashmap introduced in Java 5. Enhancement of HashMap
* Hashmap and hashtable both uses arrayand Linked List
* Tree Structure instead of Linked List in HashMap.
* Read / Write / Delete operation
  + Identify the segment
  + Identify the bucket/array of hashmap to find the exact position of insertion.
  + Iterate over the LinkedList to check the key-value pair.
  + Insert 🡪Replace if already key exists
  + Read🡪 return value if exists or return null.
  + Delete 🡪 if the key matches the link.
* Part of the map is call segment (internal data structure)
* **Segment** is only locked while updating or adding the map.
* **Allow concurrent threads** to read the value without locking at all
* Concurrenthashmap was introduced to improve performance.

Concurreny Level :

* Maximum of 32 threads can operate at max which simply maintains 32 locks to guard each bucket of concurrenthashmap.

Comparable and Comparator difference

* Comparable is an interface defining the strategy of comparing an object with other objects of the same type.

This is called classes naturing ordering.

* Integer.compareTo(x,y) returns -1.
* X< y = 0
* X=y the nreturn 1.
* We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as an argument.
* Both interfaces use Generics for compile time type checking.

HashTable

* Extends Dictionary and implements Map
* **Synchronized**
* **Does not allow** the key to be **null**
* Internally contains bucket in which key and value pair is stored.
* Hashcode is determined to which bucket the key-value pair should be.

Complexity:

* O(n) for most common operations like *get(), put(), contains()*

<https://senoritadeveloper.medium.com/data-structures-when-to-use-which-79e2006d4ee1>

HashMap

* Extends AbstractMap
* Has an array of nodes ( Node has K key ,V value and int hash ,Node next )
* Hashing – process of converting an object into integer form by using the Hashcode method.
* Bucket is an element of HashMap array to store the node
* Single bucket can have more than one node (depending on Hashcode)
* Buckets are different in capacities.
* Capacity = number of Bucket \* capacity
* Index = Hashcode (key) & (N-1)
* Allows null key
* Initial size =16
* Load factor =0.75 (75 % of capacity)
* Threshold = current capacity \* load factor
* The number of entries in HashTable that exceeds the Map is rehashed, twice the number of buckets as before.
* Rehashing 🡪 recalculating the Hashcode value of already stored entries.
* Collision occurs when the hash function returns the same bucket location with 2 different keys.
* While Collision it checks for *Hashcode() and equals()* method that both the key are same.
  + If keys are the same, replace the value with the current value.
  + Otherwise, connect the node object to the previous node object through a linked list so both are stored at the same index.
* Hashmap initially uses the linkedList
  + When the entries crosses certain threshold , it will replace the linkedlist with balanced binary tree(red-black Tree)

Methods :

*Get*

*Put*

*Getor Default*

*Remove*

*Containskey*

*containsValue*

*isEmpty*

*size*

complexity

O(1) for insertion and lookup

LinkedHashMap

* Added in JDK 1.4
* Treemap and HashMap added in JDK 1.2
* Fail-fast iterator – throws *concurrent modification exception* while adding elements while iteration.
* Unique elements with insertions order
* Non-synchronized
* Doubly LinkedList is also maintained to handle insertion order.
* To maintain insertion order , 2 more points (before and after ) at every node.

Complexity :

O(1) for insertion for lookup

Constant-time performance of Linkedhashmap is likely to be little worse than the constant time of HashMap