**Java Data structures**

***Iterators***

Iterator and Iterable are two interfaces.

Iterator has hasNext() and next() method and Iterable provides iterator() and next() method.



**Fail Fast and Fail Safe Collections**

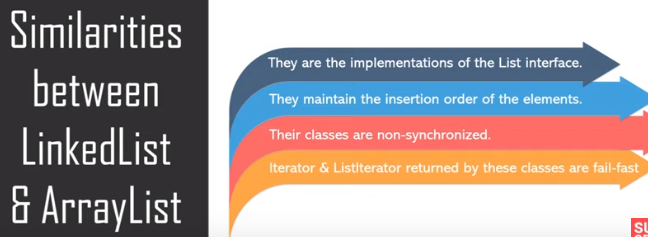
Fail fast collection means that a collection cannot be modified when it is being iterated by iterator. If such an attempt is made, then compiler would throw *ConcurrentModificationException*.

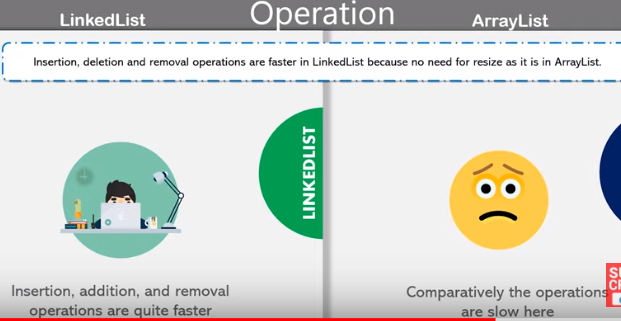
For example, ArrayList, LinkedList are all fail fast collections.

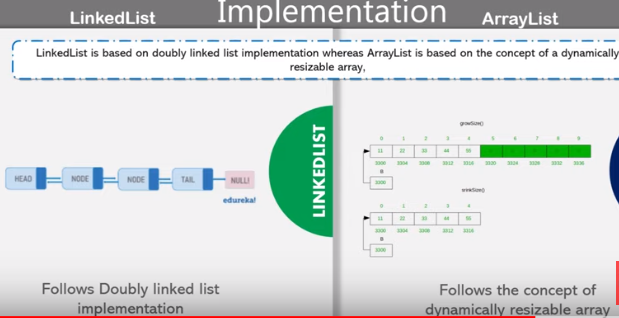
Fail Safe collections are the once that do not throw exceptions even when they are modified during Iteration.

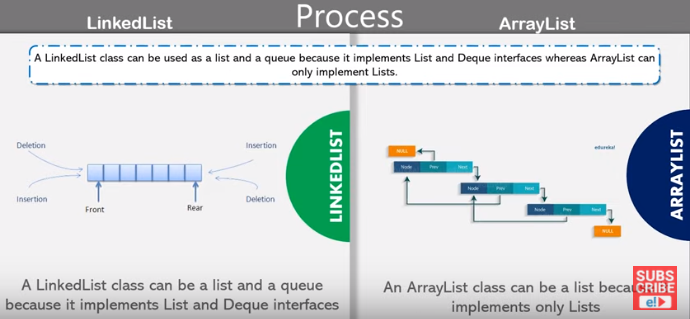
Example: CopyOnWriteArrayList (This class implements List<E>, RandomAccess, Cloneable,Serializable). It is a thread safe variant of ArrayList

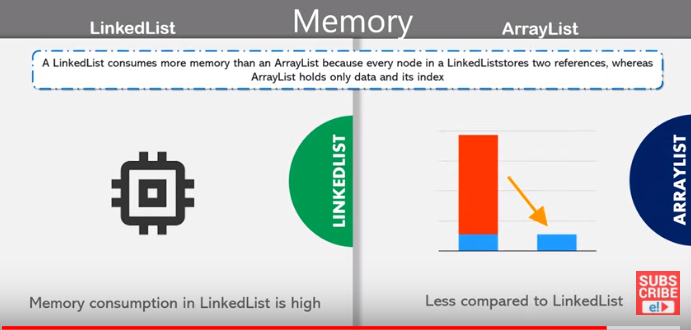
**LinkedList and Arraylist**











|  |  |  |
| --- | --- | --- |
|  | LinkedList | ArrayList |
| **P**rocess | Insert, remove, add are faster since the new node is linked between former and later with the address | Get element is faster because the element is called directly by its index |
| **I**mplementation | Implements List and Deque interfaces | Implements List interface |
| **M**emory | High. Because nodes store value and address of the element | Low. Because it stores only Index and Value of the element. That is why search is also easy |
| **P**erformance | Doubly Linked List | Dynamic Array |

**Note**: in ArrayList and LinkedList, .remove(a) where “a” is Integer removes element at position “a” in the List. Hence to remove an element from list of integers, we use .removeAll(Arrays.asList(a)) or .remove(Integer.valueOf(a))

**Note**: Both are not synchronized (i.e. not thread safe. Two threads can call them and make changes without one waiting for other)

**Note:** LinkedList is called **Doubly linked list** because it can be worked from both sides i.e. elements can be added at first or at last. It can also be traversed from both ends. A circular linked list is the one in which the last element’s next node is the head node.

**Vector**

Vectors are similar to ArrayList. The difference is that they instantiate with default size of 10 and if we insert the 11th element into them, then their size automatically becomes 20. Similary, it becomes 30 when we insert 21st element.

Vectors are similar to arrayList as they implement List interface and they also work like Dynamic Array.

Vector is available since JDK 1.0 and collections came up in JDK 1.2. Since Vector is more memory consuming for its size, ArrayList is preferable.

**Stack**

Stacks extend Vector Class. They too are available since JDK 1.0. They follow the approach of Last in First Out (LIFO). Stack behaves exactly like Vector and ArrayList, but it has two special methods:-

*1. Pop()*

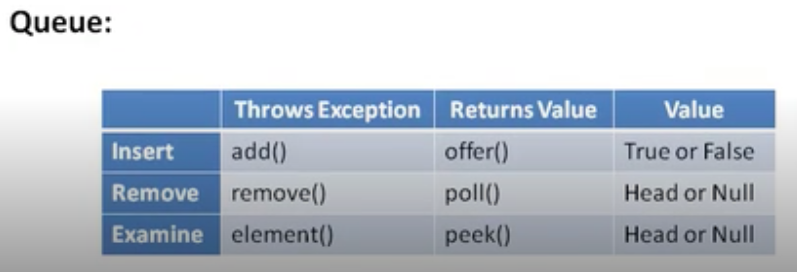
This method is used to remove the lastly added value in the Stack and return the same.

*2.Push(a)*

This method inserts “a” as Latest (uppermost) value in the stack.

**Queue**

Unlike Stack, Queue follows First in First out (or Last in Last Out) approach. Like Stack, it also provides special methods:



**Classes implementing Queue Interface**

[AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html), [ArrayBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [ArrayDeque](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html), [ConcurrentLinkedDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html), [ConcurrentLinkedQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html), [DelayQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html), [LinkedBlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html), [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html), [LinkedTransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html), [PriorityBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html), [PriorityQueue](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html), [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html)

**NOTE:**

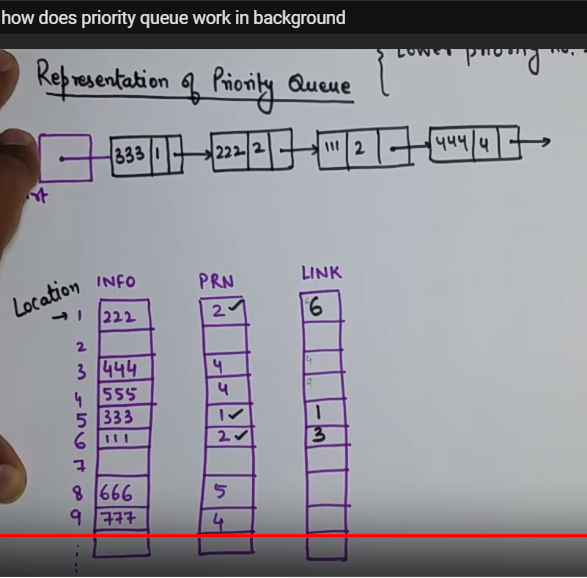
ArrayBlockingQueue is a class implementing Queue interface where the queue capacity can be fixed while initializing.

**Priority Queue**

Priority queue provides elements by priority. By default, it provides(.poll() or .peek()) string queue in alphabetical order and numbers in increasing order. We can provide this priority definition by passing a comparator object.

<https://www.youtube.com/watch?v=FdObb76AmzM>

<https://www.youtube.com/watch?v=OxhYCLWMdHs>



LinkedList implements two interfaces Deque and List. When we use LinkedList implementing List interface,we get all methods of List Interface and when we do that using Deque, we get all methods of Deque.

When we implement LinkedList using LinkedList only i.e. LinkedList list = new LinkedList(), we get methods of both.

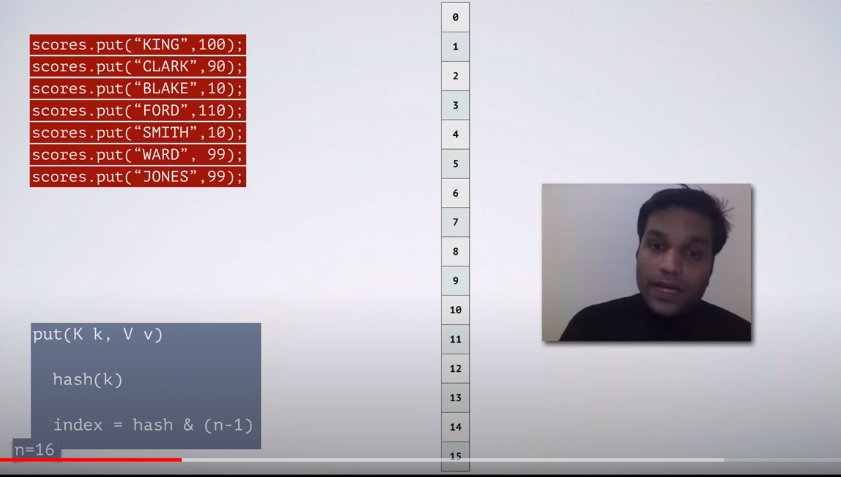
**Map Interface**

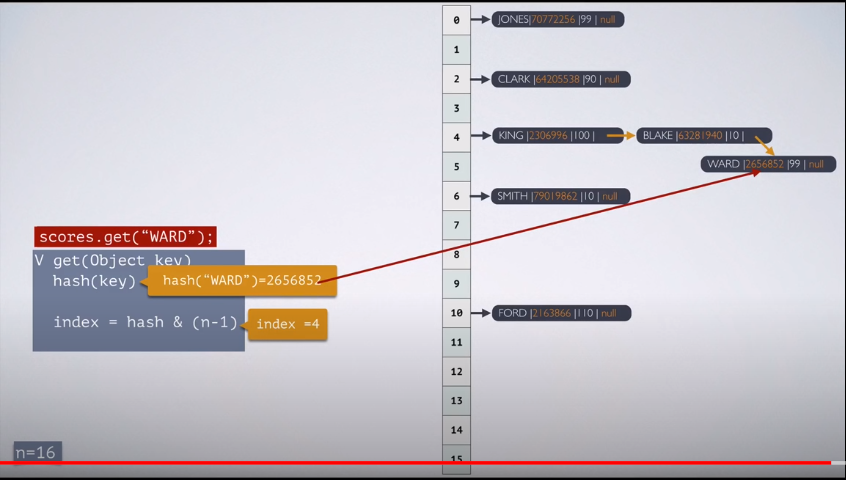
**1.HashMap**

HashMap uses a technique called Hashing. It means using hashcodes to assign index of elements being added. Note: Initial capacity of HashMap is 16 and it increases by 100% (doubles when elements are added). ***hashcode()*** method generates unique hashcode for String/int/Objects. This generated hashcode is divided by the hashmap’s *n value* (current capacity – 1) and the remainder obtained will be the index of the hashmap.

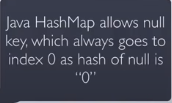
<https://www.youtube.com/watch?v=c3RVW3KGIIE>

If **n** value is same for two entries (and even if their hashcodes are same) , then the new entry is added as next node to already existing entry node with same n value.





**Note:**



Similarly to get a value using **get (key),** n value and hashcode values are found. Using n value, position is identified and then by using hashcode and the Key value using the equals method, the node is identified and then that Node data is taken.

**Note: HashMap** doesn't **allow duplicate keys** but **allows duplicate** values

