Java Collections

AMOL R PATIL

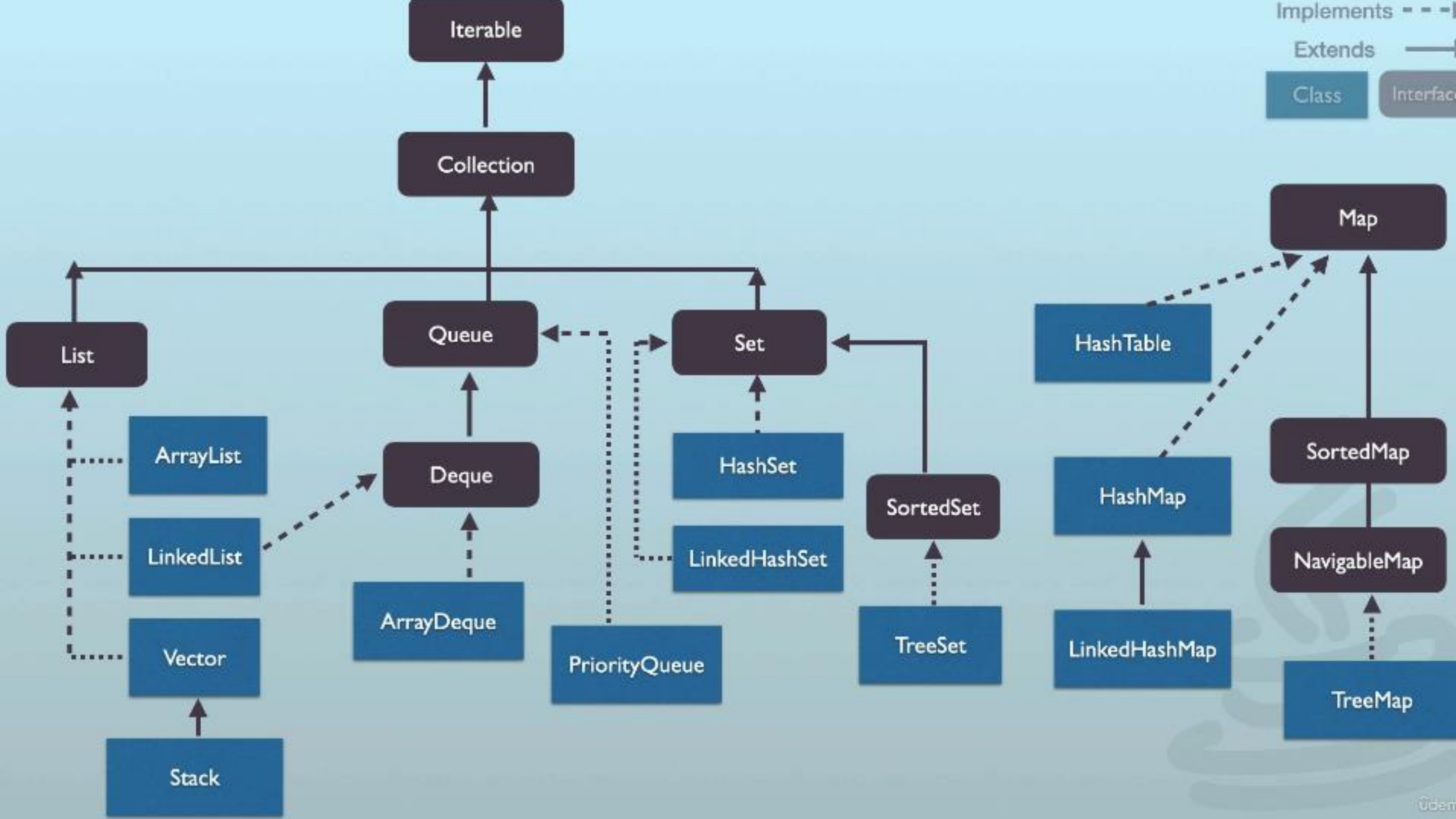
Collection Framework

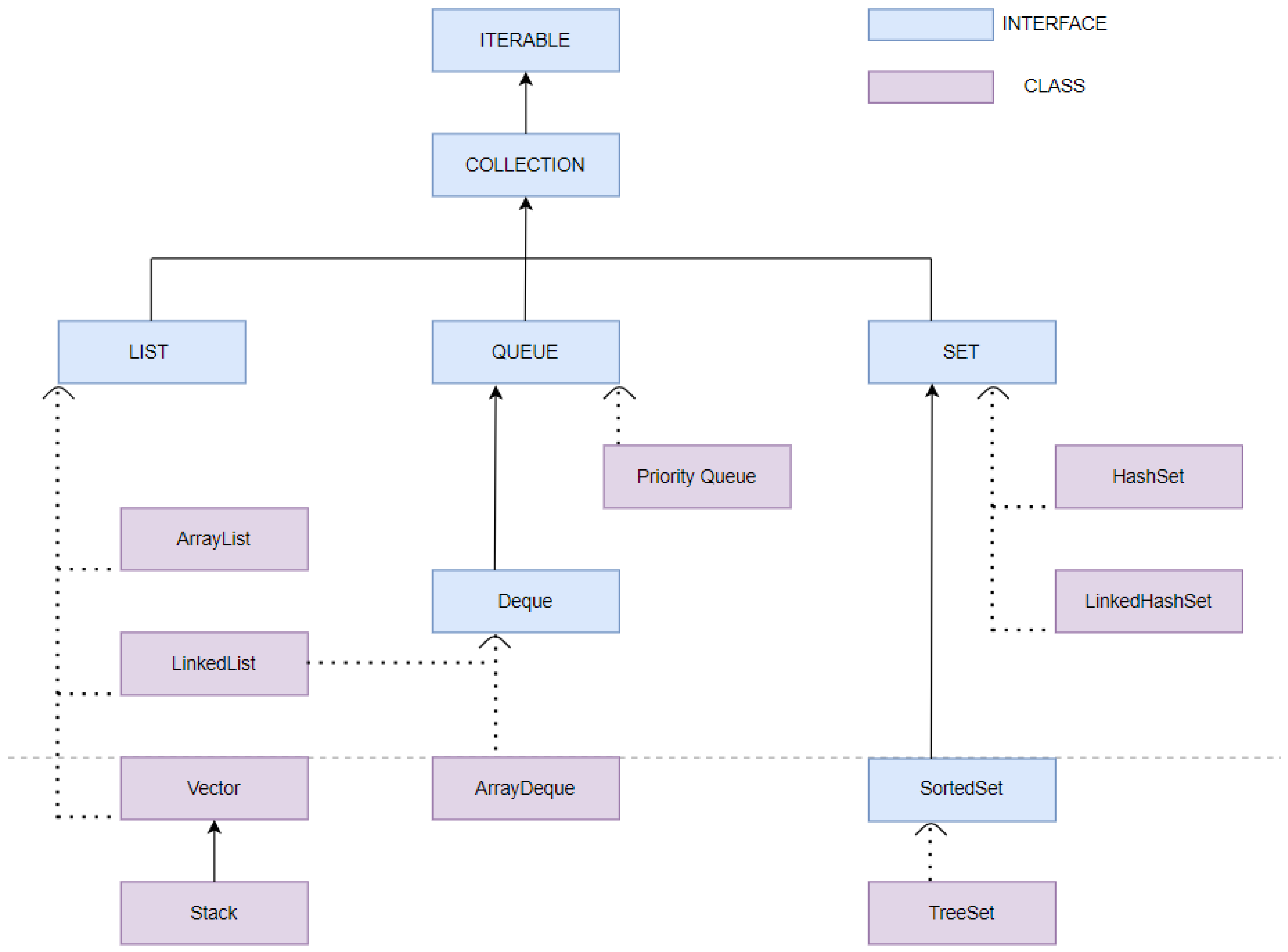
The Java collections framework is a set of classes and interfaces that implement commonly reusable collection data structures.

•Although referred to as a framework, it works in a manner of a library. The collections framework provides both interfaces that define various collections and classes that implement them.

The Java Collection Framework package (java.util) contains:

- A set of interfaces,
- Implementation classes, and
- Algorithms (such as sorting and searching).


Implements
Extends
Class
Interfac
Iterable
Collection
List
Queue
Set
Map
ArrayList
LinkedList
Vector
Stack
Deque
ArrayDeque
PriorityQueue
HashSet
LinkedHashSet
SortedSet
TreeSet
HashTable
HashMap
LinkedHashMap
SortedMap
NavigableMap
TreeMap



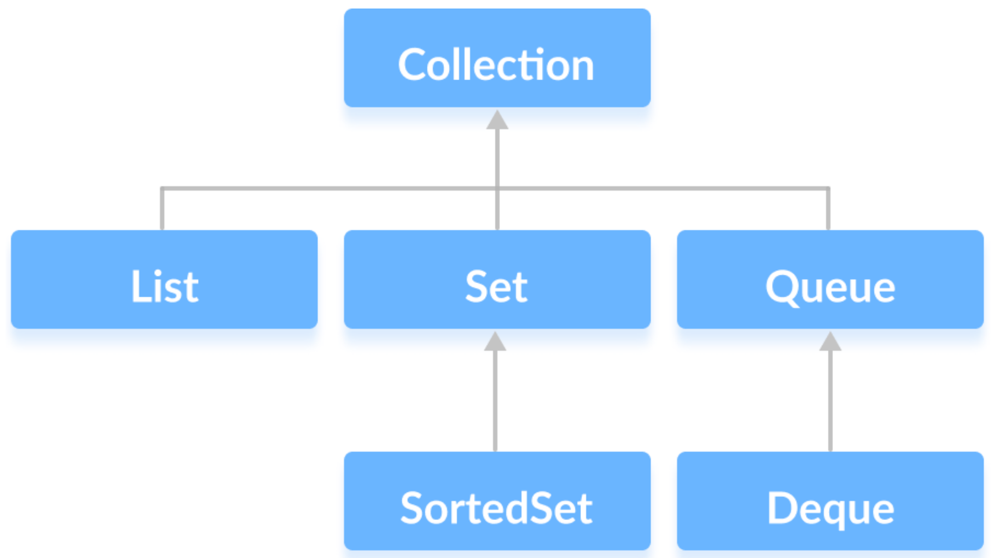
ITERABLE
COLLECTION
LIST
QUEUE
SET
Priority Queue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet
INTERFACE
CLASS

Array	**Collection Framework**
Fixed-size (not growable)	Growable in nature
If the size is 10 and only 5 elements store, then it is a waste of memory.	It adjusts size according to elements.
Arrays can hold only homogeneous data elements.	Collection can hold homogeneous as well as heterogeneous data elements.
Memory management is poor.	Memory management is effective.

Iterable Interface

Implementing this interface allows an object to be the target of the "for-each loop" statement. See For-each Loop

Collection Interface



add() - inserts the specified element to the collection

size() - returns the size of the collection

remove() - removes the specified element from the collection

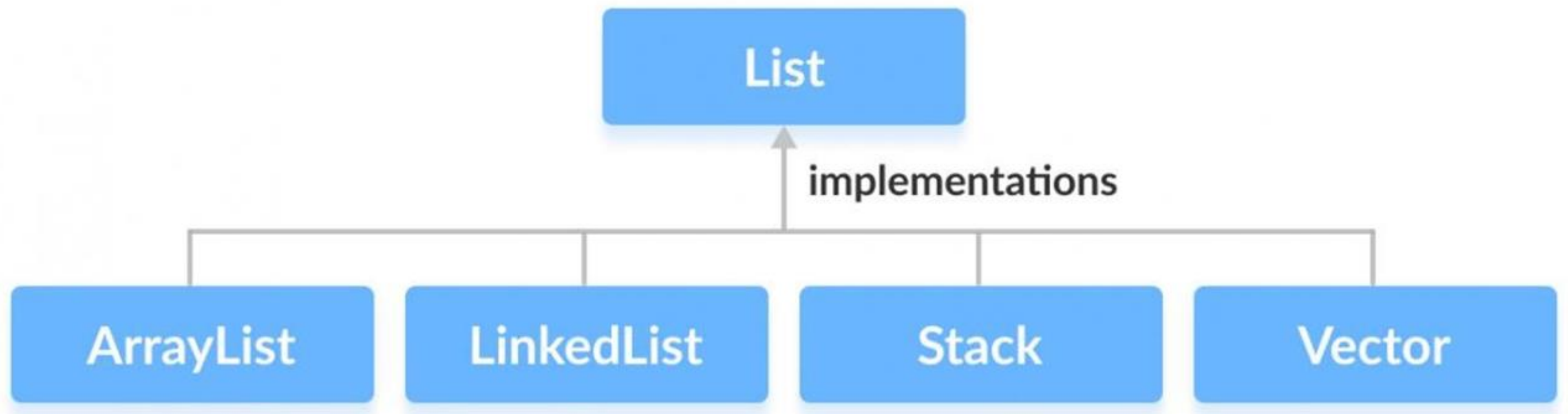
iterator() - returns an iterator to access elements of the collection

addAll() - adds all the elements of a specified collection to the collection

removeAll() - removes all the elements of the specified collection from the collection

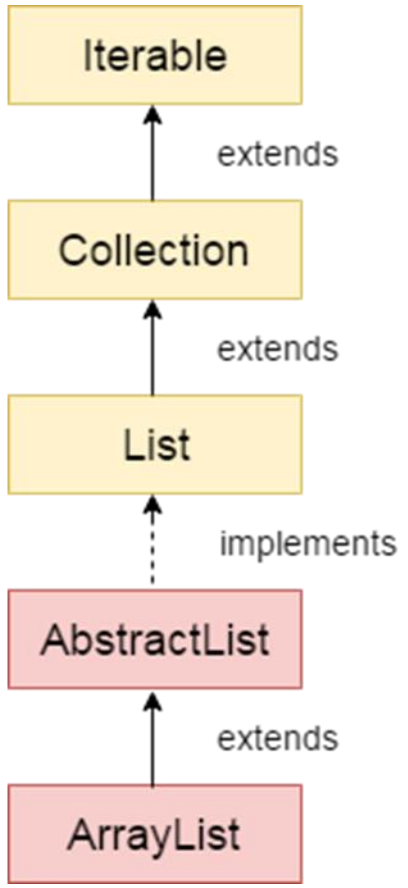
clear() - removes all the elements of the collection

List Interface



- List can **contain duplicate elements.**
- List **maintains insertion order.**
- List **Allow Multiple null elements**

ArrayList class



- It provides a **dynamic array** for storing the element.
- It is an array but there is no size limit.
- Stores null and duplicate elements
- Java ArrayList class is **non** synchronized.
- Java ArrayList **allows fast random access** because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of **shifting** needs to occur if any element is removed from the array list.

ArrayList – Java Docs

- Resizable-array implementation of the List interface.
- This class is roughly equivalent to Vector, except that it is unsynchronized.
- Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically.
- An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation

```
private static final int DEFAULT_CAPACITY = 10;
```

How ArrayList is resized in Java ?

In Java, when an ArrayList needs more space to accommodate additional elements beyond its current capacity, it resizes itself dynamically to prevent running out of space. The process of resizing involves creating a new array with a larger capacity and copying the elements from the old array into the new one.

Here's a simplified overview of how the resizing of an ArrayList typically occurs:

Initial Capacity: When you create an ArrayList without specifying an initial capacity, it starts with a default capacity (often 10 elements). If you specify an initial capacity, the ArrayList starts with that size.

Adding Elements: As you add elements to the ArrayList using the add() method, it keeps track of the number of elements and compares it to its current capacity.

Capacity Check: When the number of elements reaches the current capacity, the ArrayList needs more space to accommodate additional elements.

Resizing: At this point, the ArrayList creates a new, larger array (typically 1.5 times the current size) to accommodate more elements.

Copying Elements: It then copies all the existing elements from the old array into the new, larger array. This step ensures that the order and content of the elements remain unchanged.

Updating Reference: After copying the elements, the ArrayList updates its internal reference to point to the new array, making the new array the underlying storage for the ArrayList.

This process of resizing and copying elements is transparent to the developer using the ArrayList. However, it's essential to note that this resizing process, although efficient, involves allocating new memory and copying elements, which can be relatively expensive for large lists. To mitigate frequent resizing, it's often a good practice to estimate the number of elements the ArrayList might hold initially by using an appropriate initial capacity when creating the list, if possible.

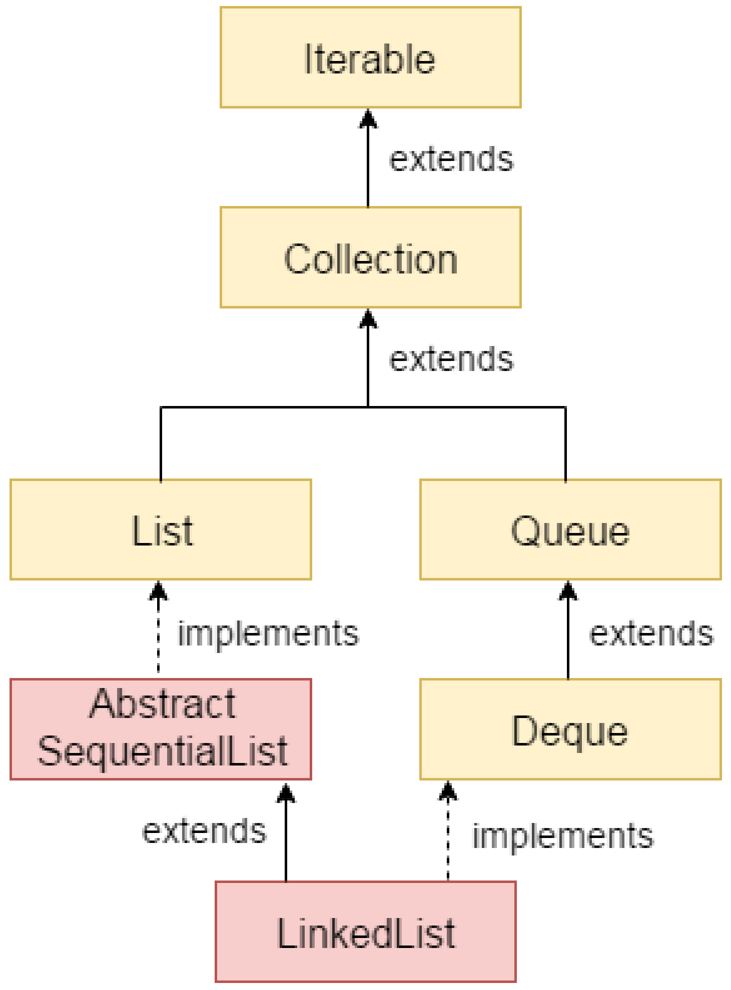
modCount Property

- The number of times this list has been structurally modified. Structural modifications are those that change the size of the list, or otherwise perturb it in such a fashion that iterations in progress may yield incorrect results.

- This field is used by the iterator and list iterator implementation returned by the iterator and listIterator methods. If the value of this field changes unexpectedly, the iterator (or list iterator) will throw a ConcurrentModificationException in response to the next, remove, previous, set or add operations. This provides fail-fast behavior, rather than non-deterministic behavior in the face of concurrent modification during iteration.

- Use of this field by subclasses is optional. If a subclass wishes to provide fail-fast iterators (and list iterators), then it merely has to increment this field in it add(int, E) and remove(int) methods (and any other methods that it overrides that result in structural modifications to the list). A single call to add(int, E) or remove(int) must add no more than one to this field, or the iterators (and list iterators) will throw bogus ConcurrentModificationExceptions. If an implementation does not wish to provide fail-fast iterators, this field may be ignored.

LinkedList



- LinkedList class uses a **doubly LinkedList** to store element. i.e., the user can add data at the first position as well as the last position.
- The **dequeue interface** is implemented using the LinkedList class.
- Null insertion is possible.
- **If we need to perform insertion /Deletion operation the LinkedList is preferred.**
- **LinkedList is used to implement Stacks and Queues.**

LinkedList – Java Docs

- Doubly-linked list implementation of the List and Deque interfaces.

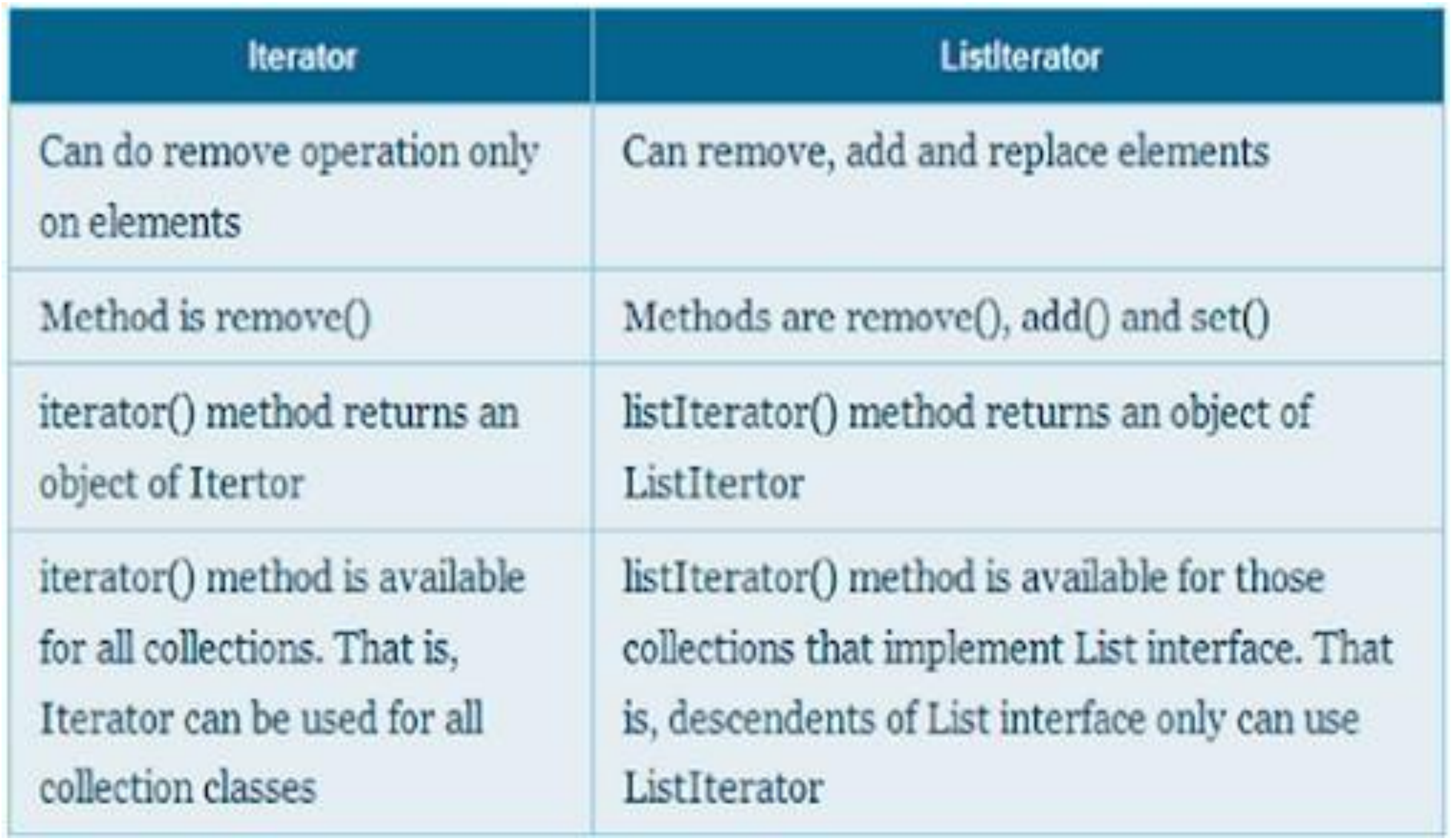
- Note that this implementation is not synchronized.

- The iterators returned by this class's iterator and listIterator methods are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

ArrayList Vs. LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.

Iterator Vs. ListIterator

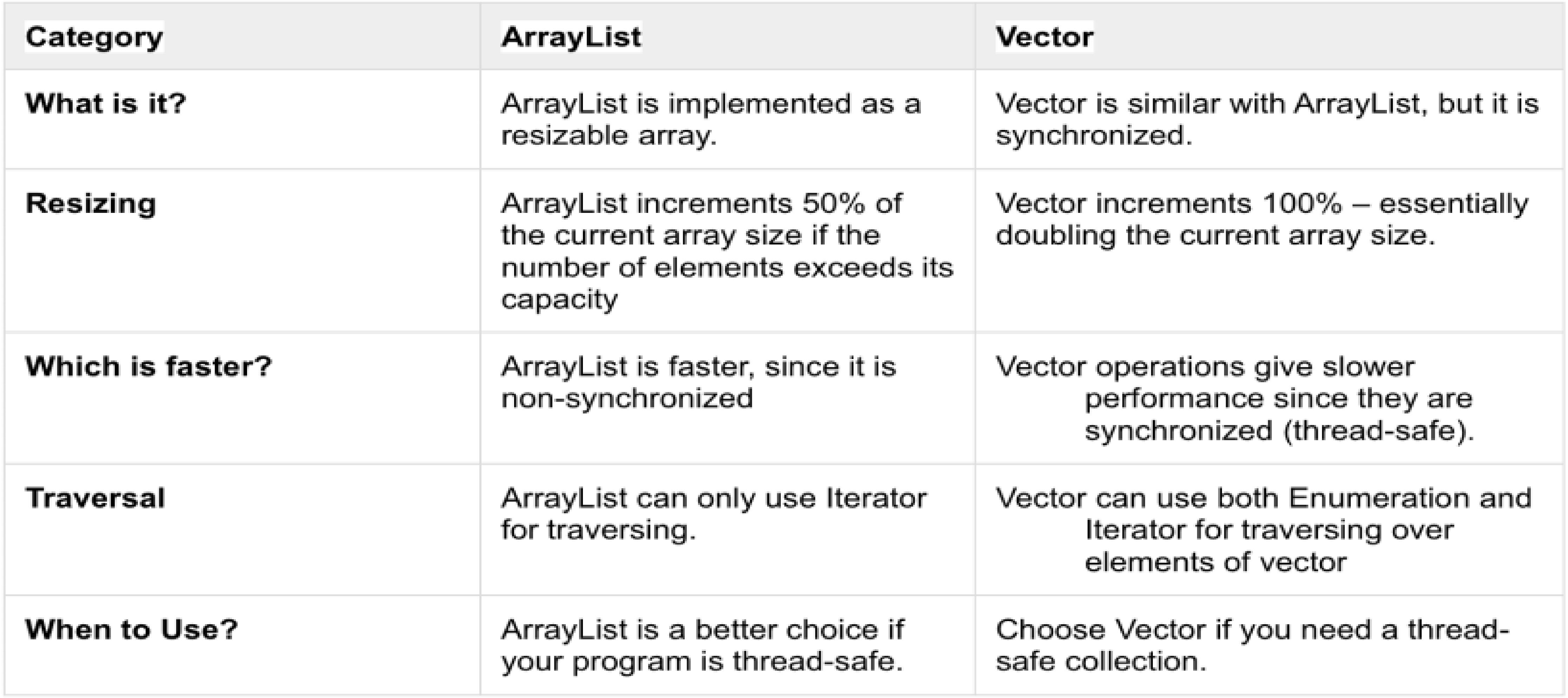


Iterator	ListIterator
Can do remove operation only on elements	Can remove, add and replace elements
Method is remove()	Methods are remove(), add() and set()
iterator() method returns an object of Itertor	listIterator() method returns an object of ListItertor
iterator() method is available for all collections. That is, Iterator can be used for all collection classes	listIterator() method is available for those collections that implement List interface. That is, descendents of List interface only can use ListIterator

Vector class

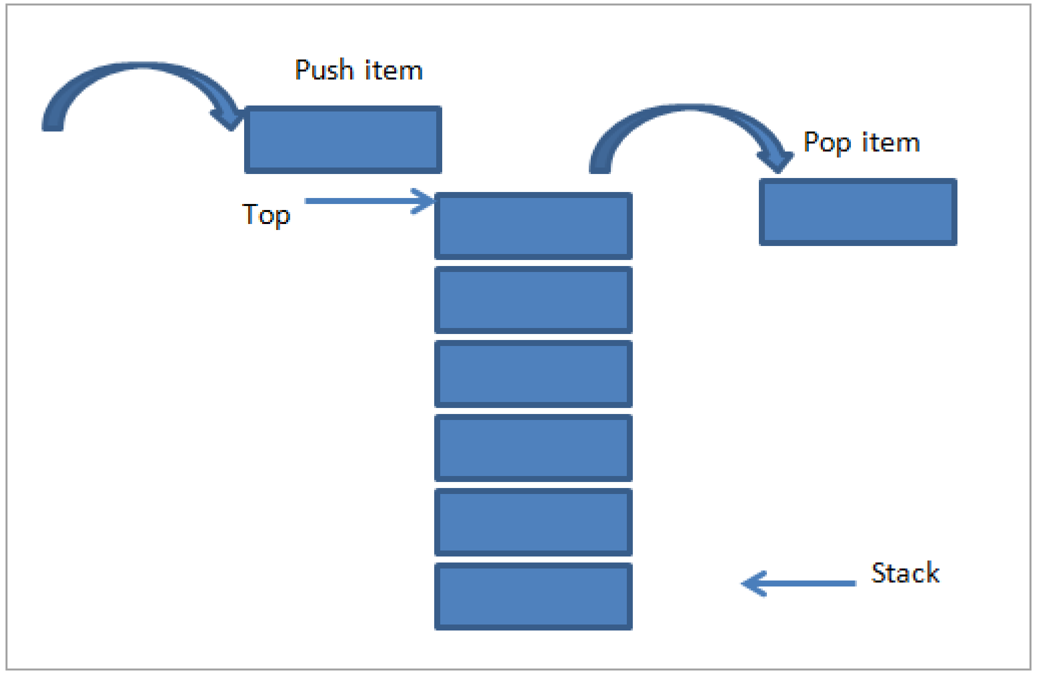
- The Vector is a class in the java.util package.
- The Vector implements List interface.
- The Vector is a legacy class.
- The Vector is synchronized.
- In ArrayList in jdk 1.7 it increases by 50% after jdk 1.8 its not specified
- In Vector The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity. If the capacity increment is less than or equal to zero, the capacity of the vector is doubled each time it needs to grow.

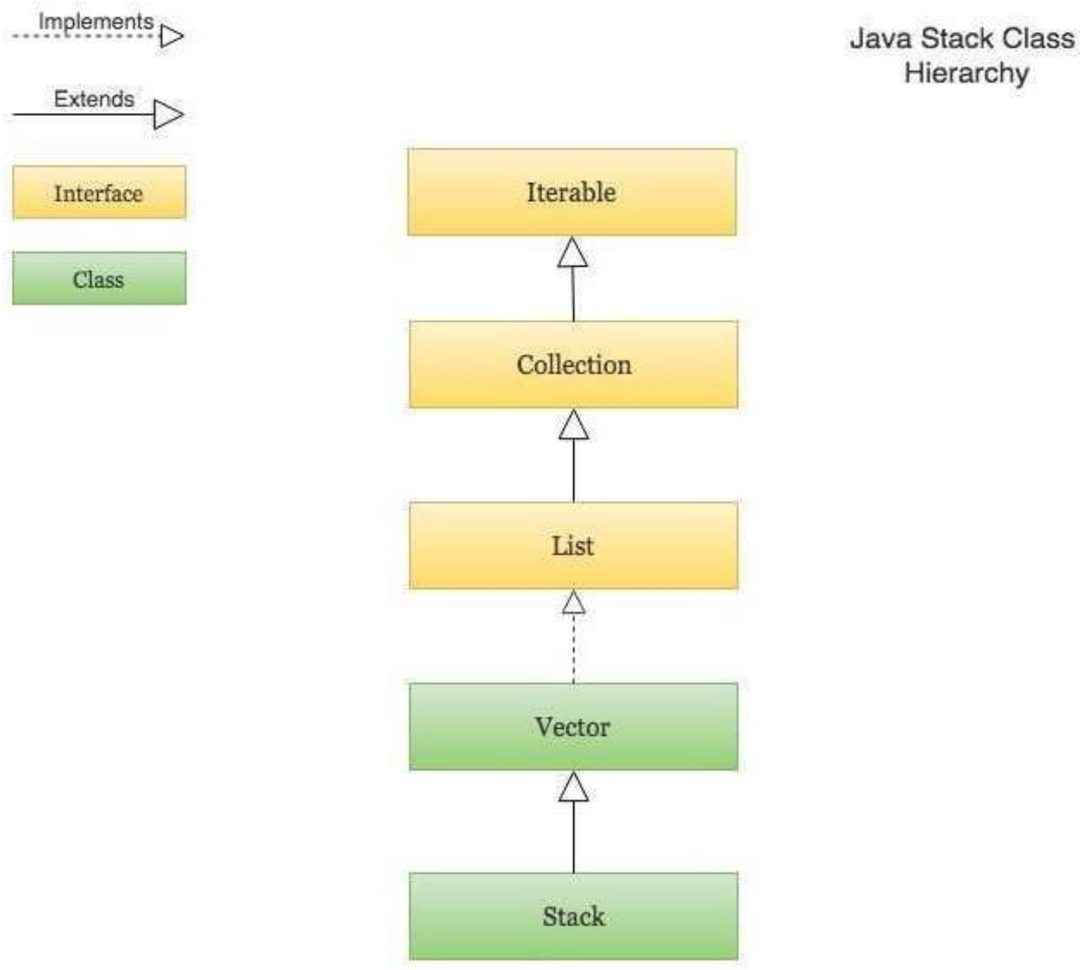
ArrayList Vs. Vector



Category	ArrayList	Vector
What is it?	ArrayList is implemented as a resizable array.	Vector is similar with ArrayList, but it is synchronized.
Resizing	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.
Which is faster?	ArrayList is faster, since it is non-synchronized	Vector operations give slower performance since they are synchronized (thread-safe).
Traversal	ArrayList can only use Iterator for traversing.	Vector can use both Enumeration and Iterator for traversing over elements of vector
When to Use?	ArrayList is a better choice if your program is thread-safe.	Choose Vector if you need a thread-safe collection.

Stack


Push item
Pop item
Top
Stack


Implements
Extends
Interface
Class
Java Stack Class Hierarchy
Iterable
Collection
List
Vector
Stack

Java Stack Operations

- **Push:** Adds an element to the stack. As a result, the value of the top is incremented.

- **Pop:** An element is removed from the stack. After the pop operation, the value of the top is decremented.

- **Peek:** This operation is used to look up or search for an element. The value of the top is not modified.

Queue Interface

java.util.Queue

- **<E> element( )**
  - Returns the **head** element of the Queue.
  It throws **NoSuchElementException** if the queue is empty.
- **<E> peek( )**
  - Returns the **head** element of the Queue.
  It returns **null** if the queue is empty.
- **boolean offer(E *obj*)**
  - Adds **obj** to the invoking Queue. Returns **true** on success and **false** on failure.
- **<E> remove( )**
  - Removes the **head** from the invoking Queue and returns the same.
  It throws **NoSuchElementException** if the queue is empty.
- **<E> poll( )**
  - Removes the **head** from the invoking Queue and returns the same.
  It returns **null** if the queue is empty.

Queue Interface Methods

- Each Queue method exists in two forms:
  - One throws an exception if the operation fails
  - The other returns a special value if the operation fails (either null or false, depending on the operation)

Queue Interface

- Following are the concrete subclasses of the Queue interface:
  - ArrayDeque
  - PriorityQueue
  - PriorityblockingQueue
  - LinkedBlockingQueue

PriorityQueue

- An unbounded priority queue based on a priority heap.
- The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
- A priority queue does not permit null elements.
- A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).
- The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.

PriorityQueue

- A priority queue is unbounded but has an internal capacity governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

- This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces. The Iterator provided in method iterator() is not guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray()).

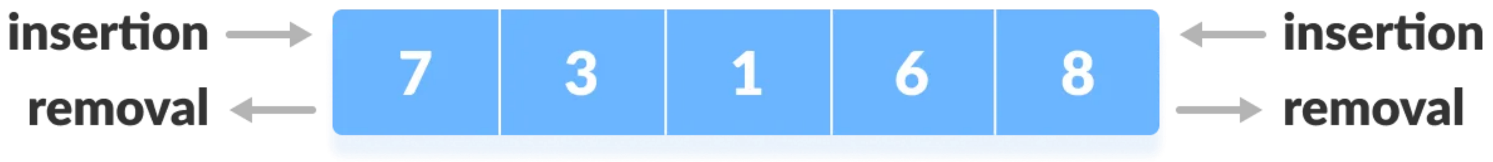
- Note that this implementation is not synchronized. Multiple threads should not access a PriorityQueue instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe PriorityBlockingQueue class.

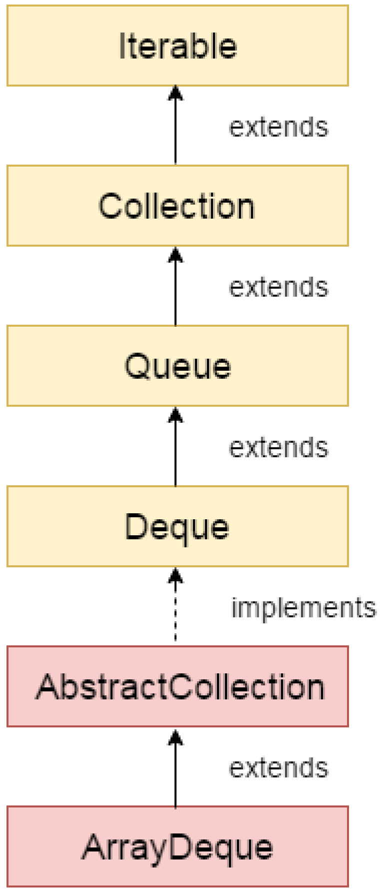
Deque and ArrayDeque

- Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First In First Out).

- The interface called Deque is present in java.util package. It is the subtype of the interface queue.

- The Deque supports the addition as well as the removal of elements from both ends of the data structure. Therefore, a deque can be used as a stack or a queue.





The important points about ArrayDeque class are:

- Resizable-array implementation of the Deque interface
- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

ArrayDeque Interface Methods

Type of Operation	First Element (Beginning of the Queue)	Last Element (End of the Queue instance)

Insert	addFirst(e)
	offerFirst(e)
		addLast(e)
		offerLast(e)

Remove	removeFirst(e)
	pollFirst(e)
		removeLast(e)
		pollLast(e)

Examine	getFirst(e)
	peekFirst(e)
		getLast(e)
		peekLast(e)

ArrayDeque Vs. LinkedList

- Both ArrayDeque and Java LinkedList implements the Deque interface. However, there exist some differences between them.

- LinkedList supports null elements, whereas ArrayDeque doesn’t.

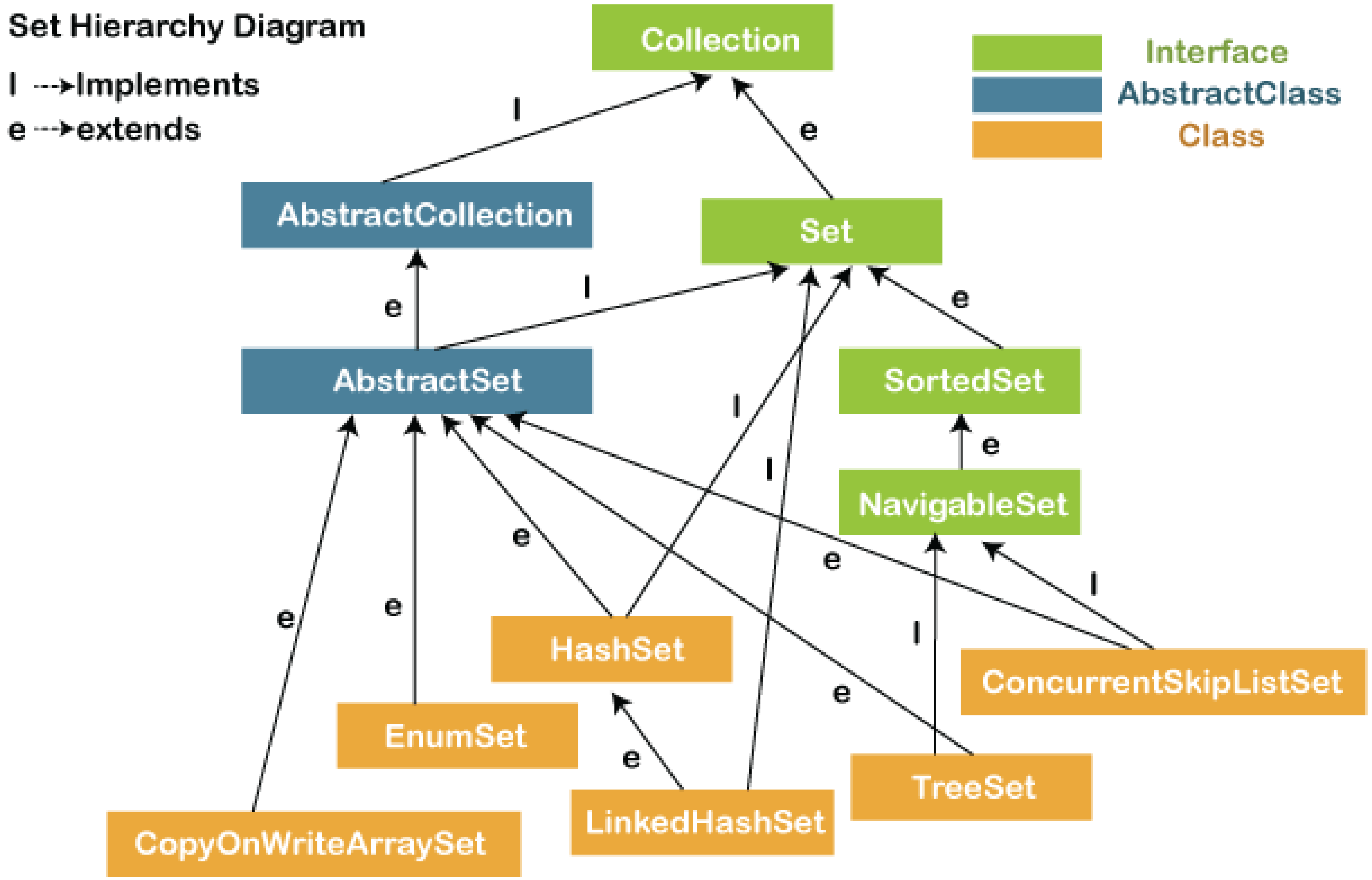
- Each node in a linked list includes links to other nodes. That's why LinkedList requires more storage than ArrayDeque.

- If you are implementing the queue or the deque data structure, an ArrayDeque is likely to faster than a LinkedList.

List Vs. Set

List	Set
The list implementation allows us to add the same or duplicate elements.	The set implementation doesn't allow us to add the same or duplicate elements.
The insertion order is maintained by the List.	It doesn't maintain the insertion order of elements.
List allows us to add any number of null values.	Set allows us to add at least one null value in it.
The List implementation classes are LinkedList and ArrayList.	The Set implementation classes are TreeSet, HashSet and LinkedHashSet.
We can get the element of a specified index from the list using the get() method.	We cannot find the element from the Set based on the index because it doesn't provide any get method().
It is used when we want to frequently access the elements by using the index.	It is used when we want to design a collection of distinct elements.
The method of List interface listiterator() is used to iterate the List elements.	The iterator is used when we need to iterate the Set elements.

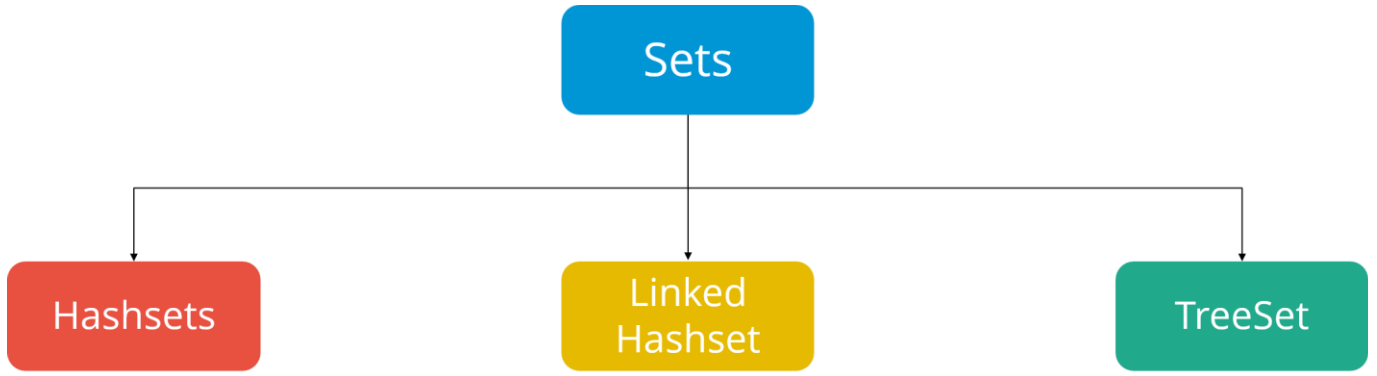
Set in Java


Set Hierarchy Diagram
I --->Implements
e --->extends
Collection
Interface
AbstractClass
Class
AbstractCollection
Set
AbstractSet
SortedSet
NavigableSet
HashSet
ConcurrentSkipListSet
EnumSet
TreeSet
LinkedHashSet
CopyOnWriteArraySet
I
e

Set in Java

- It is an unordered collection of objects in which duplicate values cannot be stored.
- It is an interface that implements the mathematical set.
- Unlike List, Set DOES NOT allow you to add duplicate elements.
- Set allows you to add at most one null element only.
- Set interface got one default method in Java 8: spliterator.
- Unlike List and arrays, Set does NOT support indexes or positions of its elements.

Set in Java



- HashSet does not provide any guarantee about the order of the elements while iterating the set.
- LinkedHashSet on the other hand, provides a guarantee about the order of the elements while iterating them.
- TreeSet provides guarantee, but the set is sorted according to the natural order, or by a specific comparator implementation.

Hashset class

- Java HashSet class creates a collection that **use a hash table** for storage.
- Hashset only contain **unique elements** and it inherits the AbstractSet class and implements Set interface. Also, it uses a mechanism hashing to store the elements.
- The HashSet initial capacity is **16 elements.**
- The HashSet is **best suitable for search** operations.

- This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element

```
private transient HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    /**
     * Constructs a new, empty set; the backing {@code HashMap} instance has
     * default initial capacity (16) and load factor (0.75).
     */
    public HashSet() {
        map = new HashMap<>();
    }
```

Hashset Internal Working

- HashSet uses HashMap internally to store data in the form of Key-Value pair.
- Elements are stored using ***hashing technique***, therefore elements are not stored in an ordered fashion and the elements will be returned in random order.
- Elements are stored in the form of key-value pair (internally by HashMap) where key will the actual element value and value will be the ***PRESENT*** Constant.
- This class permits at most ***one null element*** because HashMap allows only one null key.
- HashSet has default initial capacity of 16. HashSet has default load factor of 0.75 or 75%.
- The only way to retrieve objects from the HashSet is through iterating the entire HashSet. This can be achieved by using iterator, for, for-each ,etc. The iterators returned by HashSet class iterator method are fail-fast:

Hashset Internal Working – add function

- Adds the specified element to the set if it is not already present and then it returns true. If this set already contains the specified element, no operation would be performed, and it returns false.

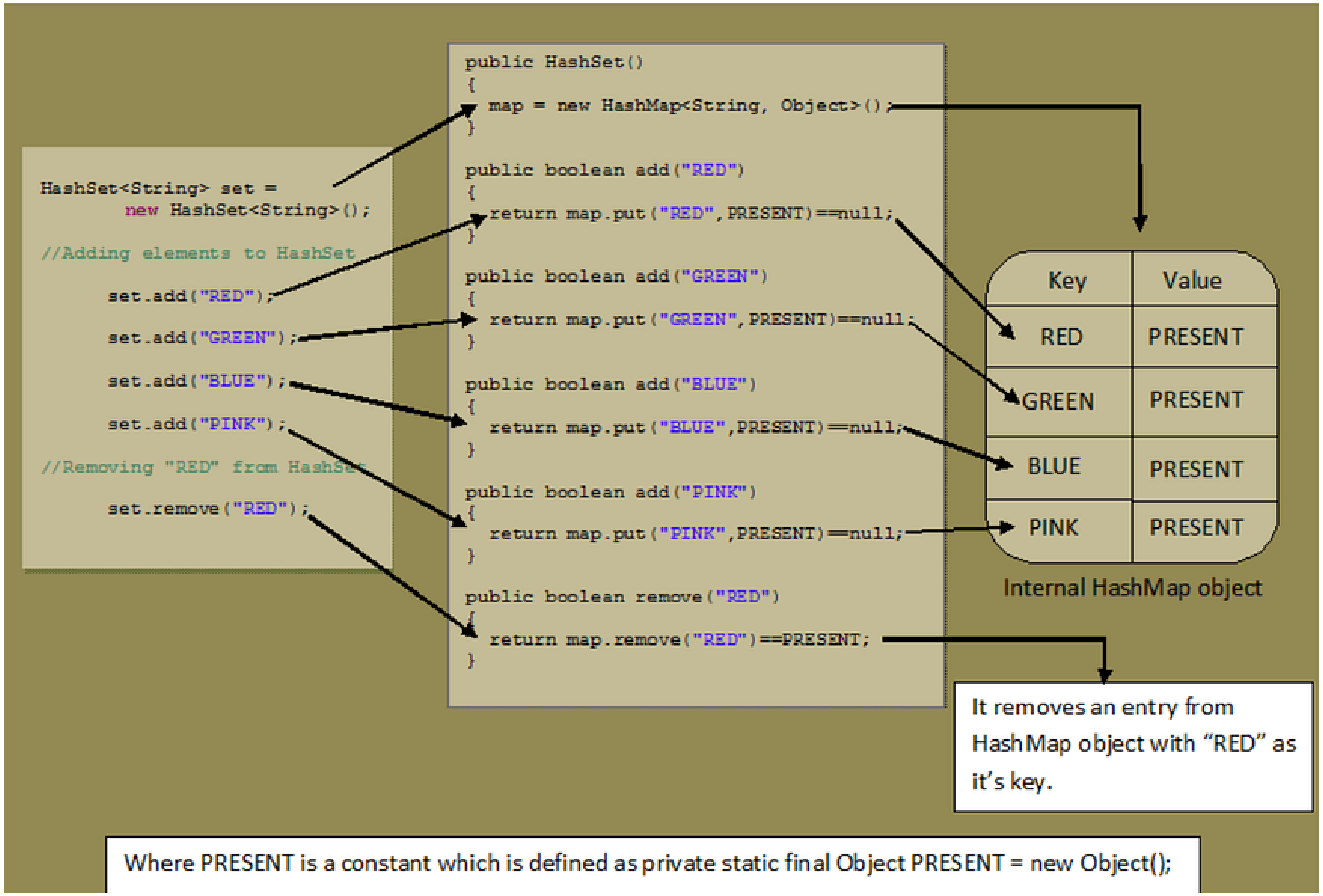
- Ex: add("Java")

- In background "Java" will behave like a "key" for the map and PRESENT will be it's value. So, add("Java") is similar to map.put("Java",PRESENT).

- More formally, adds the specified element e to this set if this set contains no element e2 such that (e==null ? e2==null : e.equals(e2)). If this set already contains the element, the call leaves the set unchanged and returns false

- To make two User defined objects equals based on values we must override equals and hashcode both methods

Hashset Internal Working



```
HashSet<String> set =
        new HashSet<String>();

//Adding elements to HashSet

      set.add("RED");

      set.add("GREEN");

      set.add("BLUE");

      set.add("PINK");

//Removing "RED" from HashSet

      set.remove("RED");
```

```
public HashSet()
{
  map = new HashMap<String, Object>();
}

public boolean add("RED")
{
  return map.put("RED",PRESENT)==null;
}

public boolean add("GREEN")
{
  return map.put("GREEN",PRESENT)==null;
}

public boolean add("BLUE")
{
  return map.put("BLUE",PRESENT)==null;
}

public boolean add("PINK")
{
  return map.put("PINK",PRESENT)==null;
}

public boolean remove("RED")
{
  return map.remove("RED")==PRESENT;
}
```

Key	Value
RED	PRESENT
GREEN	PRESENT
BLUE	PRESENT
PINK	PRESENT

Internal HashMap object

It removes an entry from HashMap object with "RED" as it's key.

Where PRESENT is a constant which is defined as private static final Object PRESENT = new Object();

LinkedHashSet

- LinkedHashSet implements a set interface and extends HashSet (refer collection hierarchy).

- LinkedHashSet is the linked list representation of the interface Set. LinkedHashSet contains unique elements but allows null values.

  - The LinkedHashSet class extends the HashSet class.
  - The basic data structure is a combination of LinkedList and Hashtable.
  - Insertion order is preserved.
  - Duplicates are not allowed.
  - LinkedHashSet is non synchronized.
  - LinkedHashSet is the same as HashSet except the above two differences are present.

LinkedHashset Internal Working

- LinkedHashSet is an extended version of HashSet.
- HashSet doesn’t follow any order where as LinkedHashSet maintains insertion order.
- HashSet uses HashMap object internally to store its elements where as LinkedHashSet uses LinkedHashMap object internally to store and process its element

```
HashSet(int initialCapacity, float loadFactor, boolean dummy)
{
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```

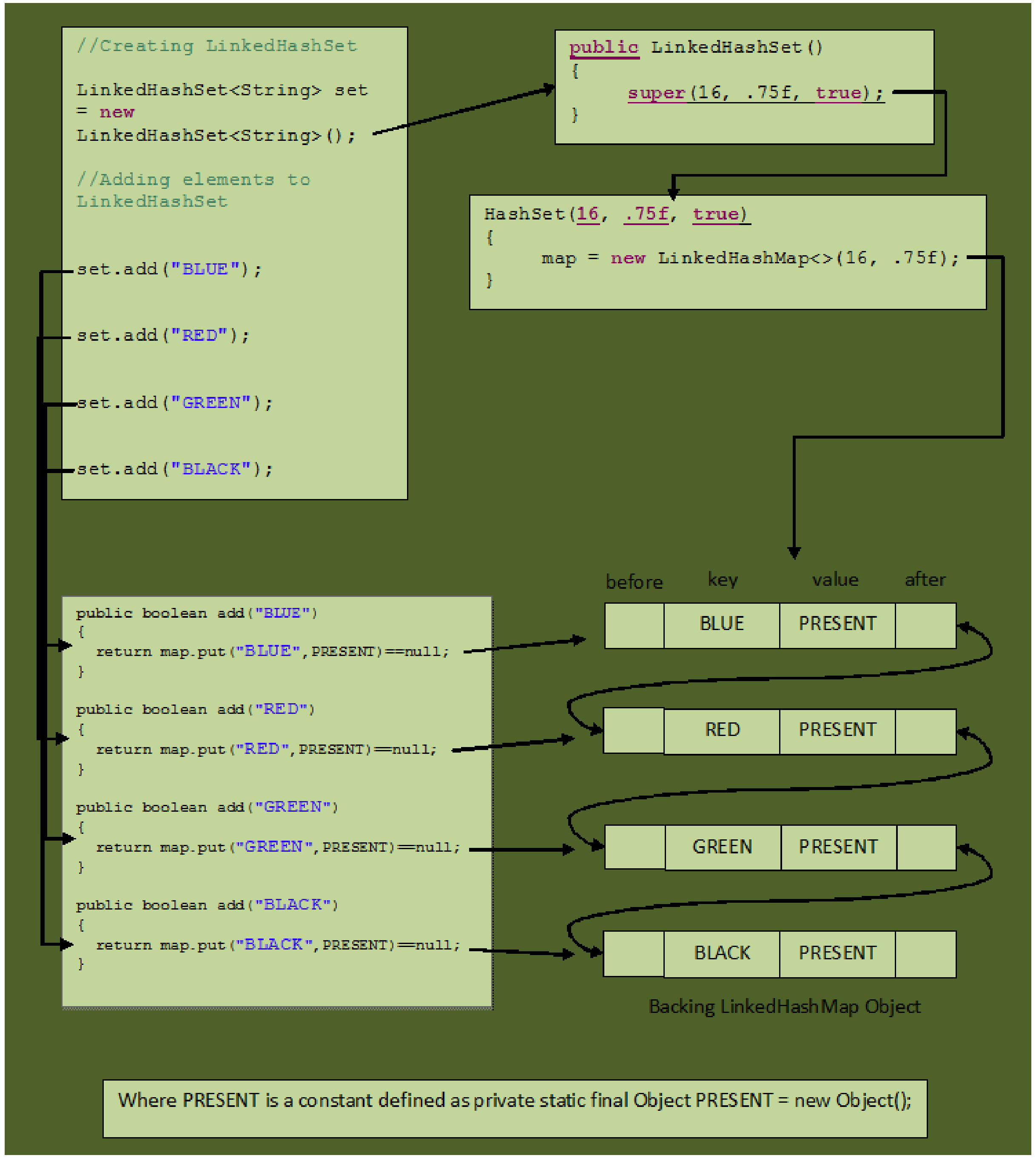
- LinkedHashSet doesn’t have its own methods. All methods are inherited from its super class i.e. HashSet. So. all operations on LinkedHashSet work in the same manner as that of HashSet.

- The only change is the internal object used to store the elements. In HashSet, elements you insert are stored as keys of HashMap object. Where as in LinkedHashSet, elements you insert are stored as keys of LinkedHashMap object. The values of these keys will be the same constant i.e. “PRESENT“.

How LinkedHashset Maintains Order ?

- LinkedHashSet uses LinkedHashMap object to store its elements. The elements you insert in the LinkedHashSet are stored as keys of this LinkedHashMap object. Each key, value pair in the LinkedHashMap are instances of its static inner class called Entry<K, V>. This Entry<K, V> class extends HashMap.Entry class.

- The insertion order of elements into LinkedHashMap are maintained by adding two new fields to this class. They are before and after. These two fields hold the references to previous and next elements. These two fields make LinkedHashMap to function as a doubly linked list.



```
//Creating LinkedHashSet

LinkedHashSet<String> set
= new
LinkedHashSet<String>();

//Adding elements to
LinkedHashSet


set.add("BLUE");


set.add("RED");


set.add("GREEN");


set.add("BLACK");
```

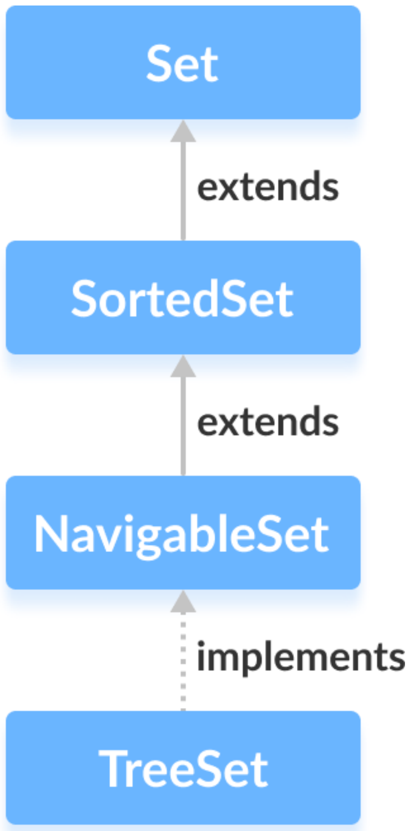
```
public LinkedHashSet()
{
      super(16, .75f, true);
}
```

```
HashSet(16, .75f, true)
{
      map = new LinkedHashMap<>(16, .75f);
}
```

```
public boolean add("BLUE")
{
   return map.put("BLUE",PRESENT)==null;
}

public boolean add("RED")
{
   return map.put("RED",PRESENT)==null;
}

public boolean add("GREEN")
{
   return map.put("GREEN",PRESENT)==null;
}

public boolean add("BLACK")
{
   return map.put("BLACK",PRESENT)==null;
}
```

Where PRESENT is a constant defined as private static final Object PRESENT = new Object();

TreeSet



- Java TreeSet class implements the Set interface that uses a tree structure to store elements.
- It contains Unique Elements.
- TreeSet class access and retrieval time are very fast
- It does not allow null elements.
- It maintains Ascending Order.

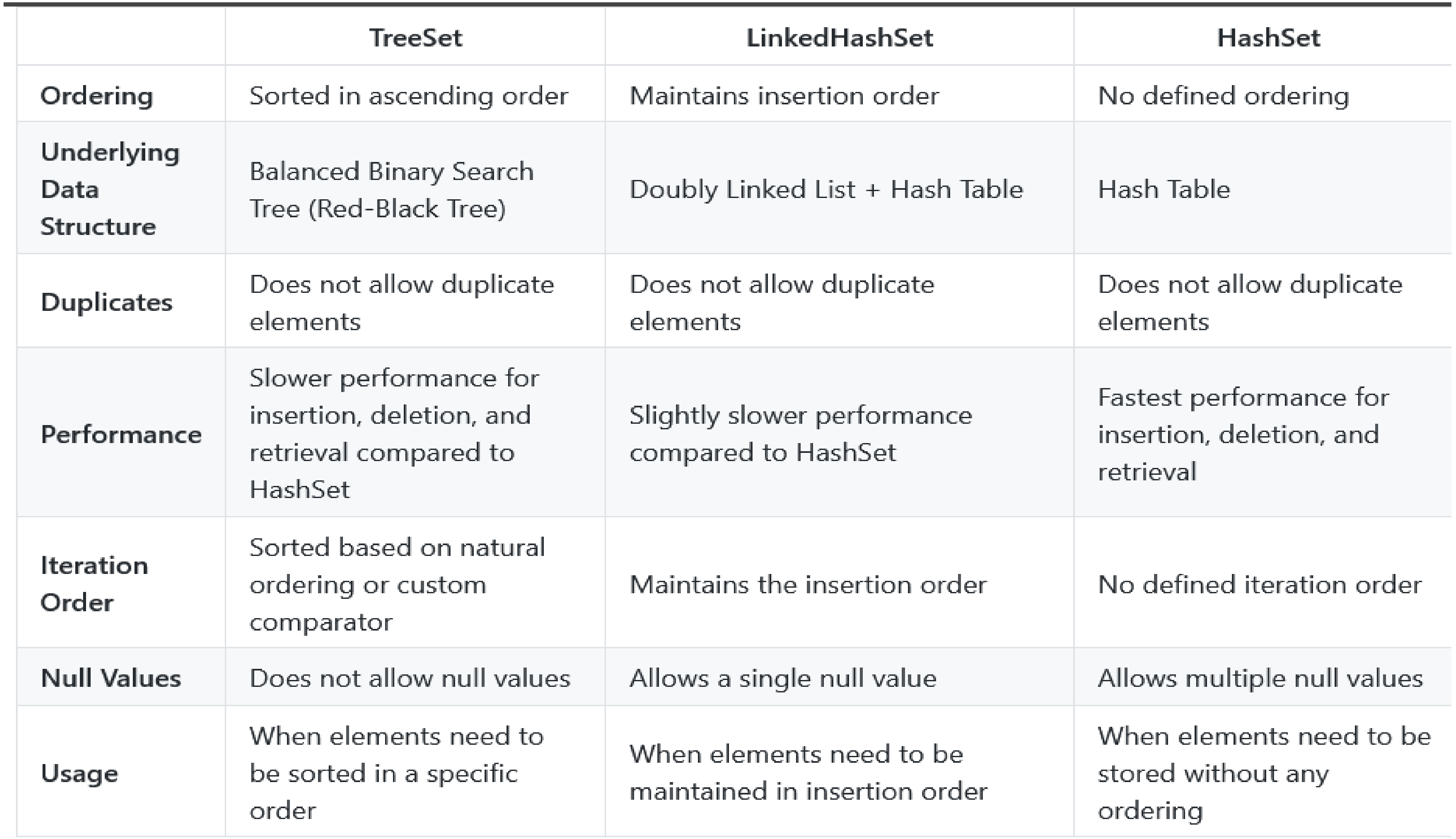
TreeSet Internal Working

- A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

- This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains)

Set Operations

- The Java Set interface allows us to perform basic mathematical set operations like union, intersection, and subset.
- Union - to get the union of two sets x and y, we can use **x.addAll(y)**
- Intersection - to get the intersection of two sets x and y, we can use **x.retainAll(y)**
- Subset - to check if x is a subset of y, we can use **y.containsAll(x)**



	TreeSet	LinkedHashSet	HashSet
Ordering	Sorted in ascending order	Maintains insertion order	No defined ordering
Underlying Data Structure	Balanced Binary Search Tree (Red-Black Tree)	Doubly Linked List + Hash Table	Hash Table
Duplicates	Does not allow duplicate elements	Does not allow duplicate elements	Does not allow duplicate elements
Performance	Slower performance for insertion, deletion, and retrieval compared to HashSet	Slightly slower performance compared to HashSet	Fastest performance for insertion, deletion, and retrieval
Iteration Order	Sorted based on natural ordering or custom comparator	Maintains the insertion order	No defined iteration order
Null Values	Does not allow null values	Allows a single null value	Allows multiple null values
Usage	When elements need to be sorted in a specific order	When elements need to be maintained in insertion order	When elements need to be stored without any ordering

Maps in Java

- An object that maps keys to values.

- In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual Values.

- A map cannot contain duplicate keys. and each key is associated with a single value.

- We can access and modify values using the keys associated with them.

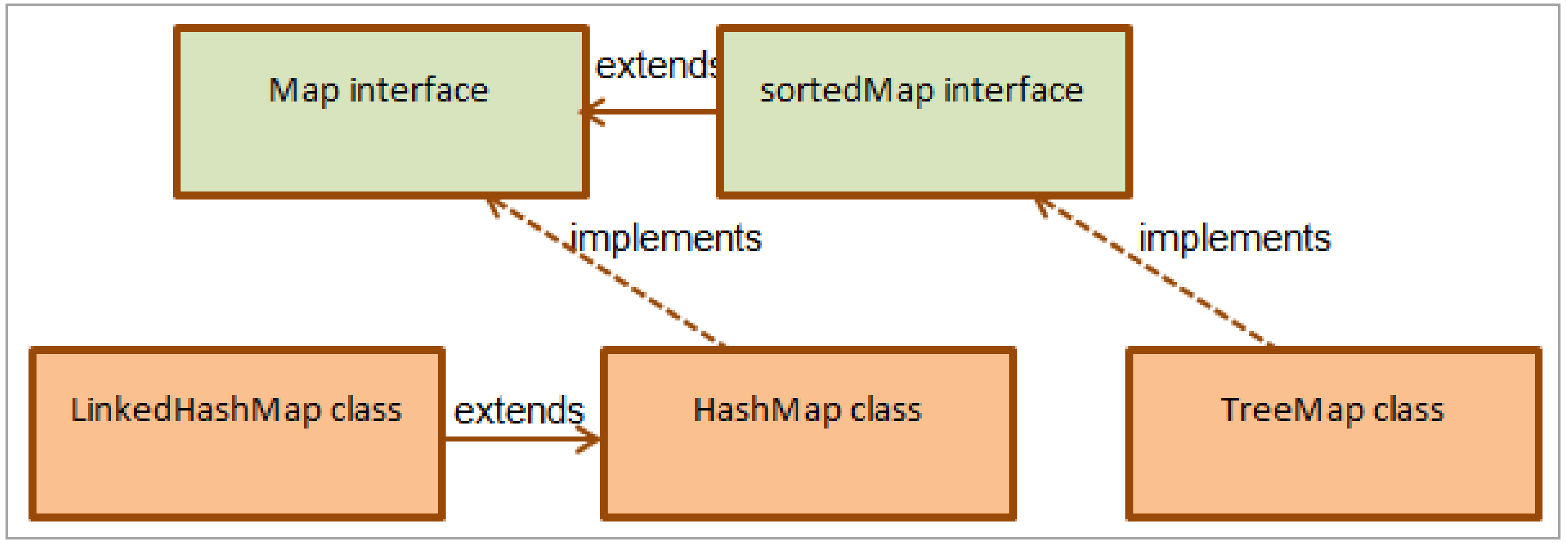
- The Map interface maintains 3 different sets:

  - the set of keys
  - the set of values
  - the set of key/value associations (mapping).

    Hence, we can access keys, values, and associations individually.

A Map is useful if you have to search, update or delete elements on the basis of a key.

Maps in Java



Class	Description
LinkedHashMap	Extends from HashMap class. This map maintains the insertion order
HashMap	Implement a map interface. No order is maintained by HashMap.
TreeMap	Implements both map and sortedMap interface. TreeMap maintains an ascending order.

HashMap

- A HashMap is class which implements the Map interface
- It stores values based on key
- It is unordered, which means that the key must be unique
- It may have null key-null value
- For adding elements in HashMap we use the put method
- Return type of put method is Object
- It returns the previous value associated with key or null if there was no mapping for key

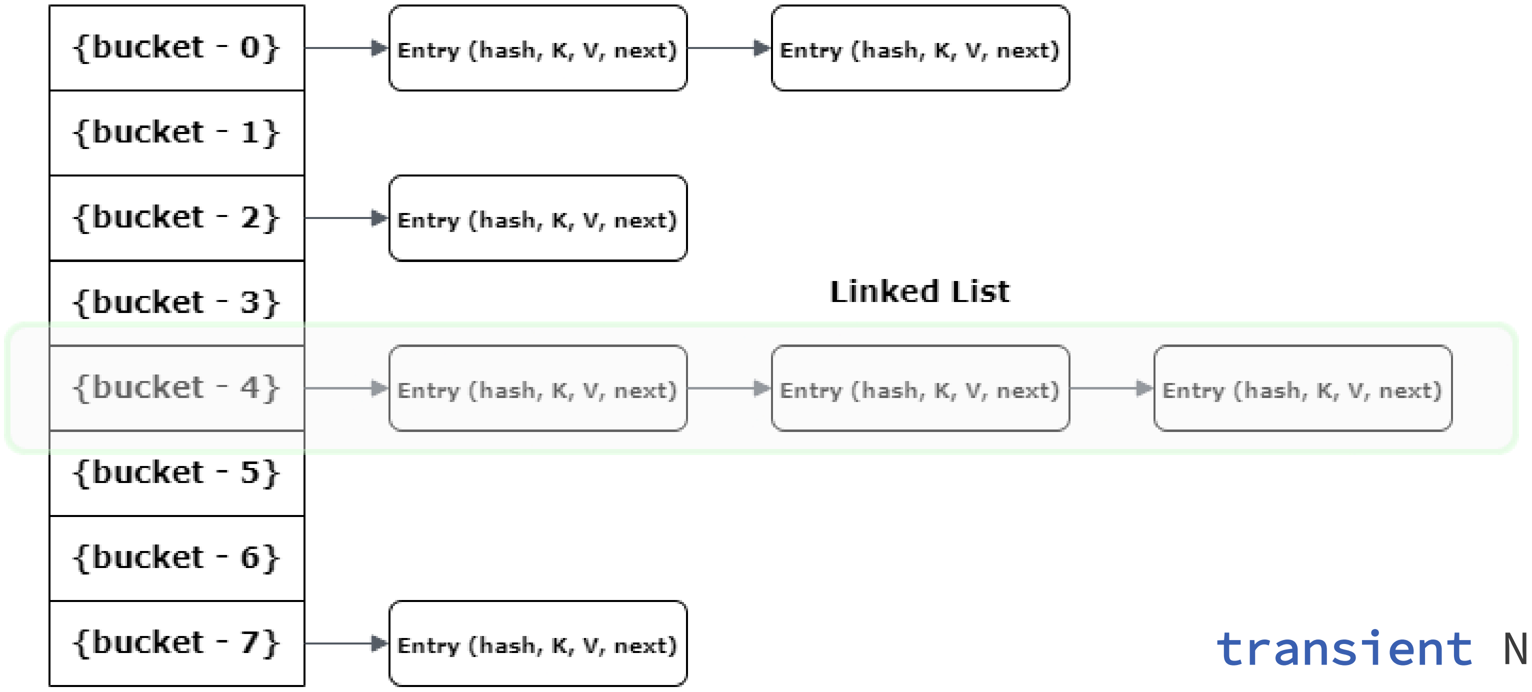
Map Entry

- Each key-value pair in a map is called an entry. An entry is represented by an instance of the Map.Entry<K,V> interface.
- Map.Entry is an inner static interface of the Map interface.
- Map.Entry has three methods called getKey(), getValue(), and setValue(), which return the key to the entry, the value of the entry, and sets a new value in the entry, respectively.
- A typical iteration over an entry set of a Map is written as follows:

```java
// Get the entry Set
Set<Map.Entry<String, String>> entries = map.entrySet();

entries.forEach((Map.Entry<String, String> entry) -> {
  String key = entry.getKey();
  String value = entry.getValue();
  System.out.println("key=" + key + ",  value=" + value);
});
```

Internal Implementation of HashMap

- The HashMap is a HashTable based implementation of the Map interface. It internally maintains an array, also called a "bucket array". The size of the bucket array is determined by the initial capacity of the HashMap, the default is 16.



```
transient Node<K,V>[] table;
```

- Each index position in the array is a bucket that can hold multiple Node objects using a LinkedList.

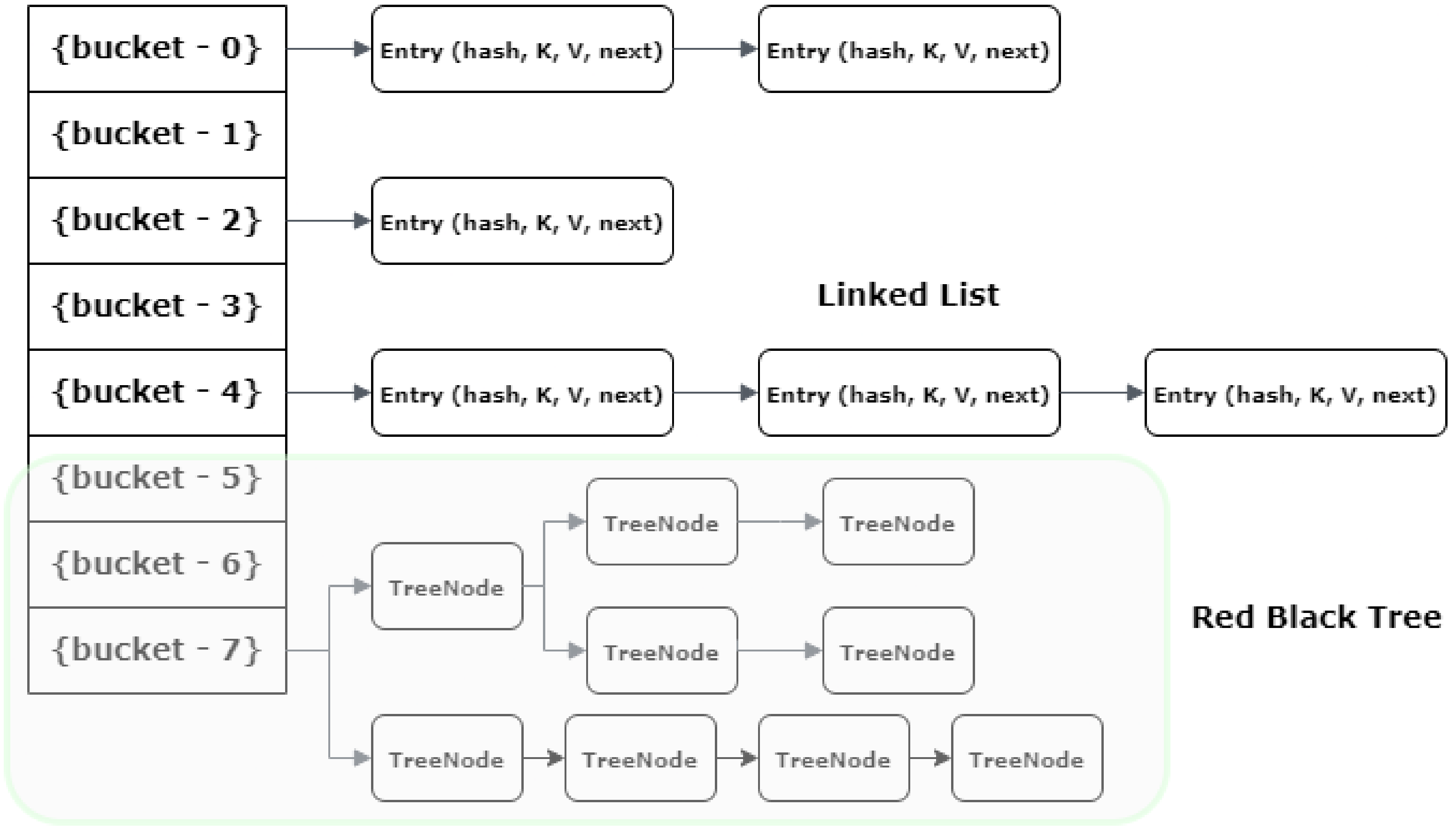
Internal Implementation of HashMap

- It is possible that multiple keys may produce the hash that maps them into a single bucket. This is why, the Map entries are stored as LinkedList.

- But when entries in a single bucket reach a threshold (TREEIFY_THRESHOLD, default value 8) then Map converts the bucket's internal structure from the linked list to a RedBlackTree (JEP 180). All Entry instances are converted to TreeNode instances.

- Basically, when a bucket becomes too big, HashMap dynamically replaces it with an ad-hoc implementation of TreeMap. This way, rather than having a pessimistic O(n) performance, we get a much better O(log n).

```
static class Node<K,V> implements Map.Entry<K,V> {

  final int hash;
  final K key;
  V value;
  Node<K,V> next;

  //...
}
```



Note that when nodes in a bucket reduce less than UNTREEIFY_THRESHOLD the Tree again converts to LinkedList. This helps balance performance with memory usage because TreeNodes takes more memory than Map.Entry instances.

So, Map uses Tree only when there is a considerable performance gain in exchange for memory wastage.

How Hashing is used to Locate Buckets?

- In Java, all objects inherit a default implementation of hashCode() function defined in Object class. It produces the hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.

- Java designers understood that end-user-created objects may not produce evenly distributed hash codes, so Map class internally applies another round of hashing function on the key’s hashcode() to make them reasonably distributed.

- This is the final hash, generated from the initial hash of the Key object, that is the index of the array where Node should be searched.

```
static final int hash(Object key)
{
        int h; return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
 }
```

- This is the final hash, generated from the initial hash of the Key object, that is the index of the array where Node should be searched.

How are Collisions Resolved in the Same Bucket?

- When a Node object needs to be stored at a particular index, HashMap checks whether there is already a Node present in the array index?? If there is no Node already present, the current Node object is stored in this location.

- If an object is already sitting on the calculated index, its next attribute is checked. If it is null, and current Node object becomes next node in LinkedList. If next variable is not null, the process is followed until next is evaluated as null.

- Note that while iterating over nodes, if the keys of the existing node and new node are equal then the value of the existing node is replaced by the value of the new node.

```
if ((e = p.next) == null) {
    p.next = new Node(hash, key, value, null);
    break;
}
```

How are Collisions Resolved in the Same Bucket?

- Note that if the first node in the bucket is of type TreeNode then TreeNode.putTreeVal() is used to insert a new node in the red-black tree.

```
else if (p instanceof TreeNode)
    e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
```

LinkedHashMap

- Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface.

- LinkedHashMap is like HashMap with an additional feature of maintaining an order of elements inserted into it.

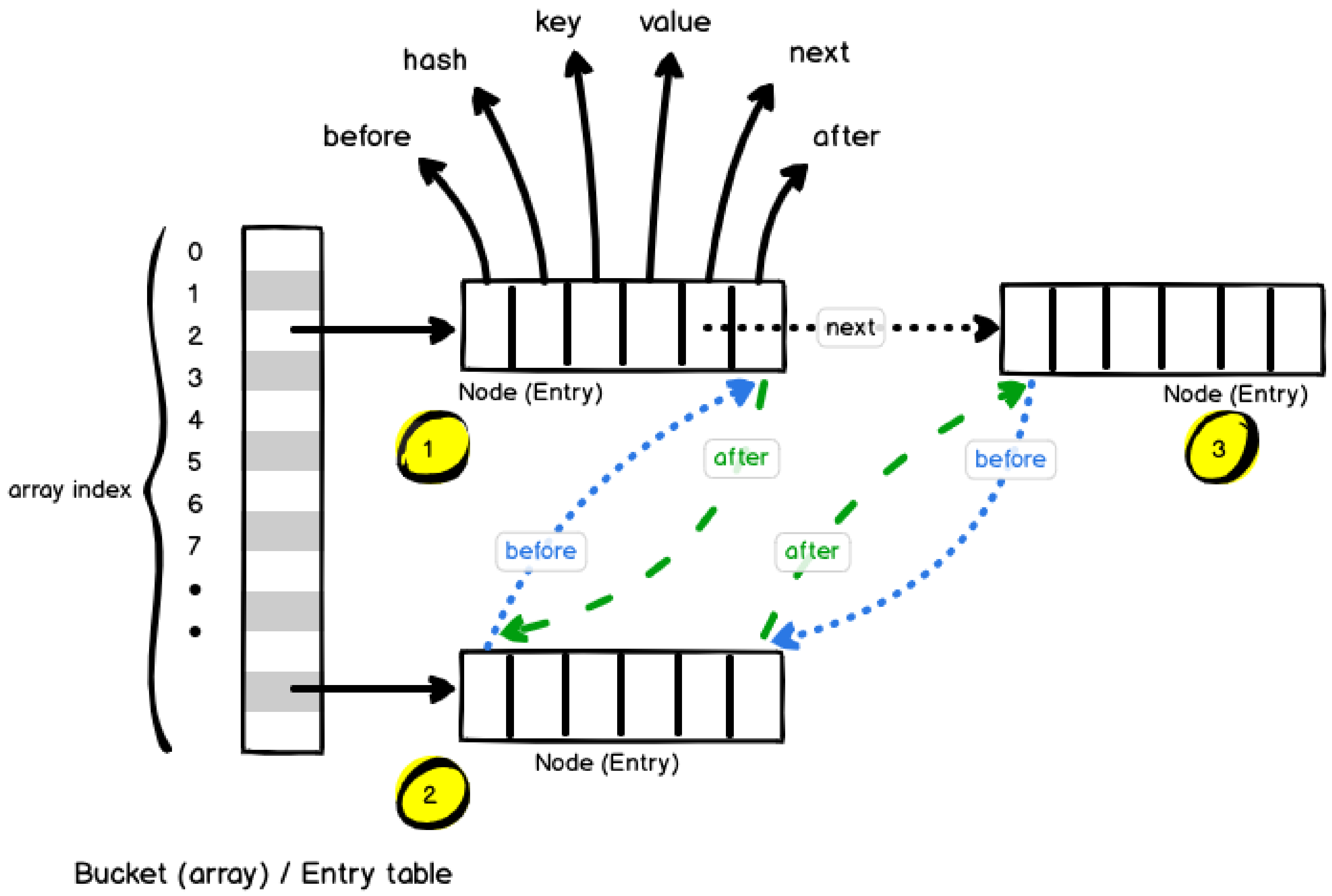
- HashMap provided the advantage of quick insertion, search, and deletion but it does not track and order the insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

  - Java LinkedHashMap contains values based on the key.
  - Java LinkedHashMap contains unique elements.
  - Java LinkedHashMap may have one null key and multiple null values.
  - Java LinkedHashMap is non synchronized.
  - Java LinkedHashMap maintains insertion order.
  - The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

LinkedHashMap

- We have total of 6 fields now:-
  - before: points to the node inserted before this node
  - after points to the node inserted after this node
  - key: key as provided
  - value: value as provided
  - next: points to the next node in the same bucket of array table(like in HashMap)
  - hash: hashcode to calculate the index of this node and check for equality.

  Also, there is head and tail fields in the LinkedHashMap, which points to the head and tail of our doubly LinkedList.



LinkedHashMap

TreeMap

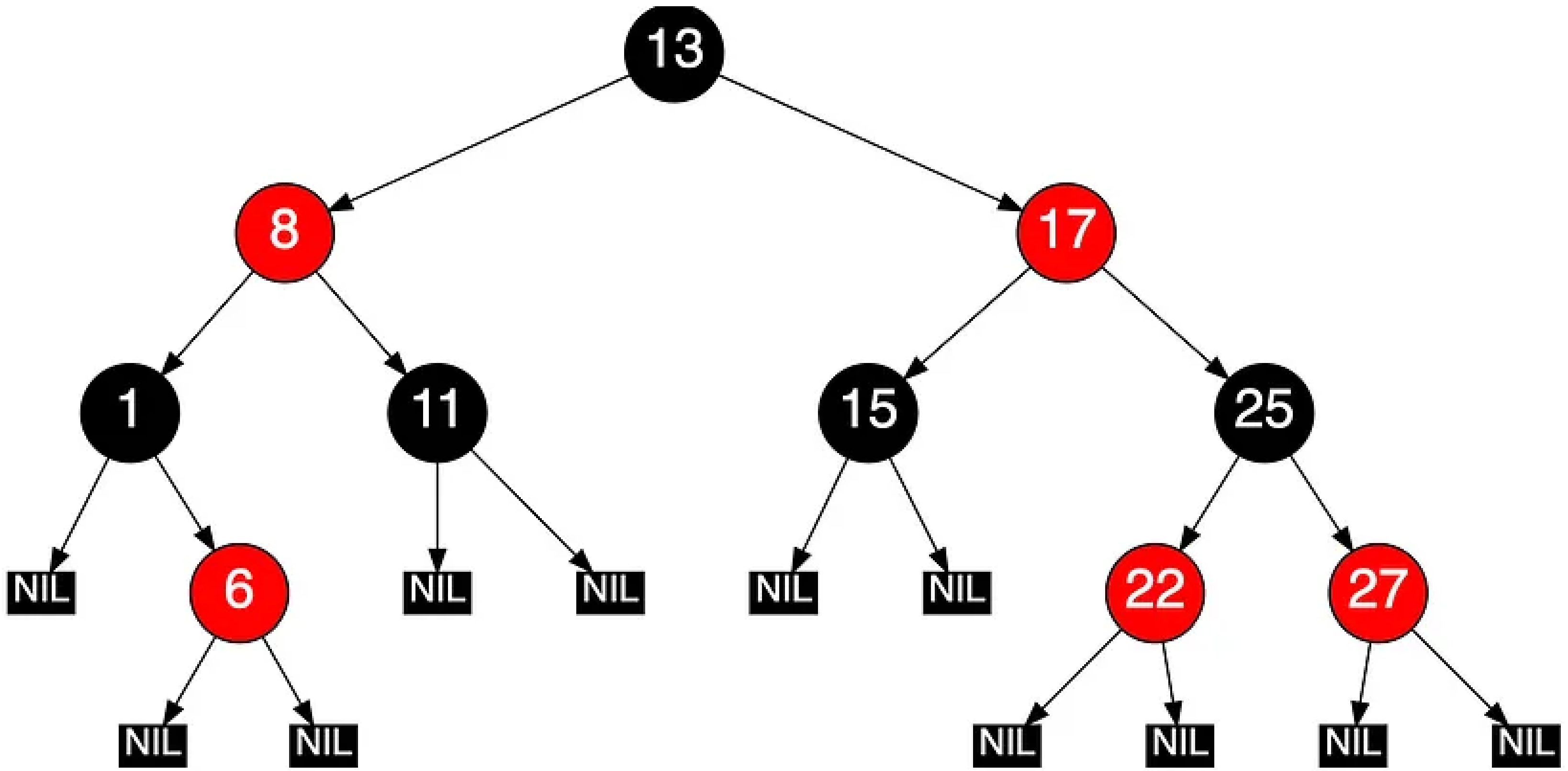
- TreeMap is a map implementation that keeps its entries sorted according to the natural ordering of its keys or better still using a comparator if provided by the user at construction time.

- Tree Map does not use hashing for storing keys. It's use Red-Black tree(self-balancing binary search tree.).

- The TreeMap is sorted according to the natural ordering of its keys. Tree Map will always have all elements sorted.

    - Working of Tree Map is based on tree data structure.
    - Tree Map is not Synchronized and hence not thread safe.
    - Tree Map in java doesn't allow null key but allow multiple null values.

- This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.


13
8
17
1
11
15
25
NIL
6
NIL
NIL
NIL
NIL
22
27
NIL
NIL
NIL
NIL
NIL
NIL

- A red-black tree is a self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black).

- These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. It must be noted that as each node requires only 1 bit of space to store the color information, these types of trees show identical memory footprint to the classic (uncolored) binary search tree.

- As the name of the algorithm suggests color of every node in the tree is either black or red.

- The root node must be Black in color.

- The red node can not have a red color neighbors' node.

- All paths from the root node to the null should consist of the same number of black nodes.

HashTable

- A Hashtable is an array of lists. Each list is known as a bucket.
- A hashtable contains values based on key-value pair.
- It contains unique elements only.
- Hashtable class does not allow null key as well as value otherwise it will throw NullPointerException.
- **Every method is synchronized**. i.e At a time only one thread is allowed, and the other threads are on a wait.
- **Performance is poor as compared to HashMap**.

Parameters	**TreeMap**	**HashMap**	**LinkedHashMap**
Insertion and Hookup	For insertion and hookup, it has O(logN) complexity.	For insertion and hookup, it has O(1) complexity.	For insertion and hookup, the complexity of LinkedHashMap is O(1).
Null Keys	It does not allow any null key.	It only allows a single null key.	It only allows a single null key.
Null Values	It allows null values.	It allows multiple numbers of null values.	It allows multiple numbers of null values.
Maintaining Order	The primary function of TreeMap is to maintain order. It helps us in the storage of keys in a sorted manner in ascending order.	The HashMap does not function to maintain any order.	The primary function of the LinkedHashMap is to maintain an order in which we would insert the key

A HashMap is implemented as a Hash table, a TreeMap is implemented as a Red-Black Tree, and LinkedHashMap is implemented as a doubly-linked list buckets in Java.

HashMap	Hashtable
1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
2) HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
3) HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
4) HashMap is **fast**.	Hashtable is **slow**.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
7) Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
8) HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

Fail fast and Fail Safe Iterators

- The iterators can be either fail-safe or fail-fast.
- Fail-safe iterators means they will not throw any exception even if the collection is modified while iterating over it.
- Whereas Fail-fast iterators throw an exception(ConcurrentModificationException) if the collection is modified while iterating over it

Fail Fast Internal Working

- Every fail fast collection has a modCount field, to represent how many times the collection has changed/modified.

- So, at every modification of this collection, we increment the modCount value.

- So, every time there is some change in the collection structure, the mod count is incremented.

- Now the iterator stores the modCount value in the initialization as below:

```
int expectedModCount = modCount;
```

- Now while the iteration is going on, expectedModCount will have old value of modCount.
If there is any change made in the collection, the modCount will change and then an exception is thrown using:

```
if (modCount != expectedModCount)
          throw new ConcurrentModificationException();
```

- This code is used in most of the iterator methods e.g.   next , remove , add

So, if we make any changes to the collection, the modCount will change, and expectedModCount will not be hence equal to the modCount. Then if we use any of the above methods of iterator, the ConcurrentModificationException will be thrown.

Note: If we remove/add the element using the remove() or add() of iterator instead of collection, then in that case no exception will occur. It is because the remove/add methods of iterators call the remove/add method of collection internally, and it reassigns the expectedModCount to new modCount value.

Fail-Safe Iterators internal working:

Unlike the fail-fast iterators, these iterators traverse over the clone of the collection. So even if the original collection gets structurally modified, no exception will be thrown.

Collections class

- The Java Collections Class is a utility class in java.util package that consists of several static methods that operate on or return collections. It is a member of the Java Collections Framework.

Important java.util.Collections methods

•sort(List list)
•sort(List list, Comparator<? super Project> c)
•shuffle(List<?> list)
•reverse(List<?> list)
•rotate(List<?> list, int distance)
•swap(List<?> list, int i, int j)
•replaceAll(List list, String oldVal, String newVal)
•copy(List<? super String> dest, List<? extends String> src)
•Collections.binarySearch(list, "element 4")
•frequency(Collection<?> c, Object o)
•disjoint(Collection> c1, Collection> c2)
•min(Collection extends ?> coll)
•max(Collection extends ?> coll)

Arrays class

- Arrays Class has built in functions to achieve the frequently used array operations.

- These include :

- – asList() for creating a List from the array
- – sort() for sorting arrays
- – binarySearch() for searching a sorted array
- – equals() for comparing arrays
- – fill() for filling values into an array

Comparable Vs. Comparator

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Conversions – List to Array , Array to List

The toArray() method of the ArrayList class is used to convert the ArrayList to an array and return the new array.

```
public Object[] toArray()

public <T> T[] toArray(T[] a)

public static <T> List<T> asList(T... a)
```