**Collection in Java**

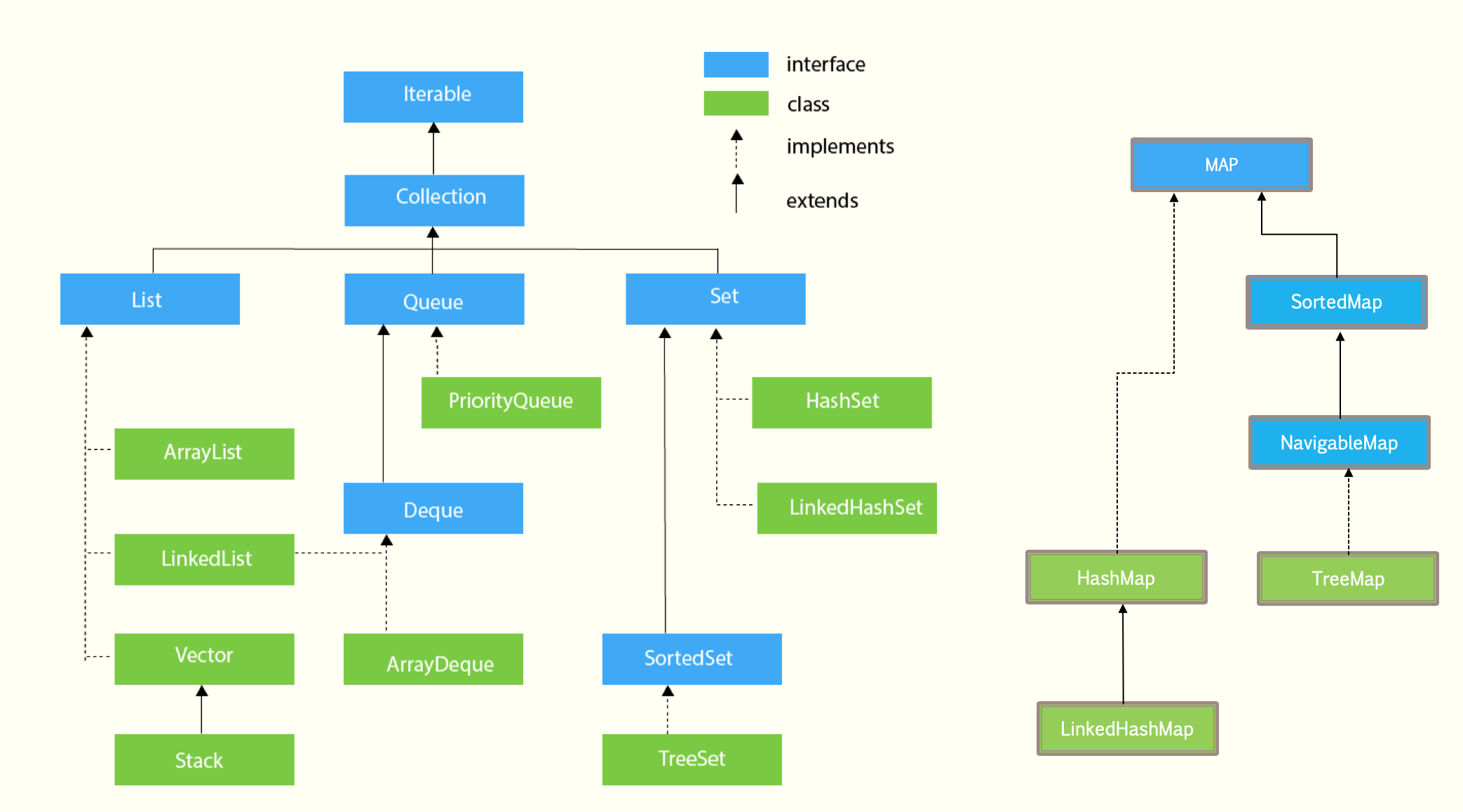
***Collection in Java is a framework that provides an architecture to store and manipulate the group of objects***

* The Collection framework in Java has: - Interfaces and its implementations, i.e., classes – Algorithm
* **java.util** package contains all the classes and interfaces for the Collection framework

**“Collection” and “Collections” look similar, but are different.**

- Collection is a top-level interface of java collection framework -- gives List, Queue etc.

- Collections is a utility class. - like sort(), reverse(), binarySearch()



**LIST Interface**

**ArrayList Class**

ArrayList<String> names = new ArrayList<>({intial\_capacity});

* ArrayList **internally use an ARRAY**object to add (or store) the elements.
* The array of ArrayList is resizable (or dynamic).By default, the **capacity of the array is 10**

**LinkedList Class**

LinkedList<String> linkedList = new LinkedList<>();

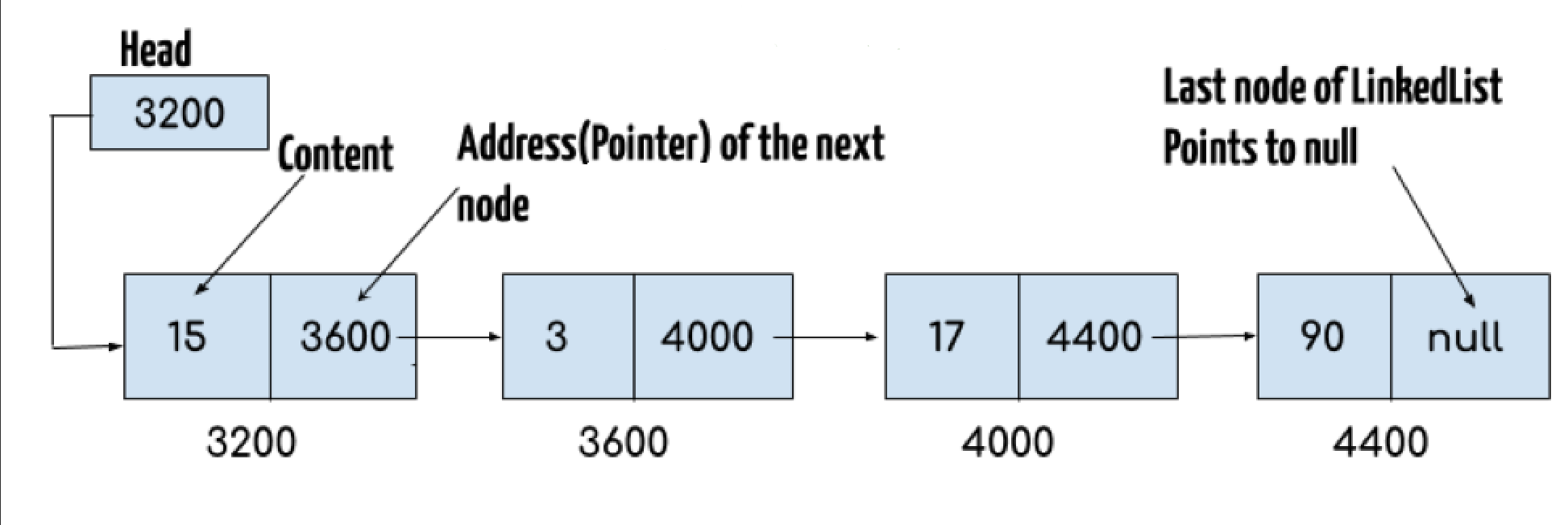
* Similar to arrays in Java, **LinkedList is a linear data structure**. (and ordered)
* However, LinkedList elements are **not stored in contiguous locations** like arrays, they are linked with each other using **POINTERS**.
* Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.

## **Representation:**

* Each element in the LinkedList is called the **Node**.
* Each Node of the LinkedList contains two items:

1. Content of the element
2. Pointer/Address/Reference to the Next Node in the LinkedList.

**This is how a LinkedList looks:**



**Note:**

* **Head** of the LinkedList only contains the Address of the **First element** of the List.
* The Last element of the LinkedList contains **null** in the pointer part of the node because it is the end of the List so it doesn’t point to anything as shown in the above diagram.
* The diagram shown above represents a **singly linked list**.
* There is another complex type variation of LinkedList which is called **doubly linked list**, node of a doubly linked list contains three parts: 1) Pointer to the previous node of the linked list 2) content of the element 3) pointer to the next node of the linked list.

## **Why do we need a Linked List?**

**Arrays have certain limitations such as:**

1. **Size of the array is fixed** – large size means wastage of memory; short size means cannot expand it more *(Linked list allows****dynamic memory allocation****)*
2. Array elements **need contiguous memory locations** to store their values. (*whereas Linked list elements don’t need contiguous memory locations)*
3. **Inserting an element in an array is performance wise expensive** as we have to shift several elements to make a space for the new element. Similarly **deleting an element** from the array is also a performance wise expensive operation because all the elements after the deleted element have to be shifted left.

**Arraylists are also resizable array** i.e. with Arraylists there is a risk of resizing array and copying content to new array if array gets full which makes adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList in Java.

# STACK Class

Stack<String> stack = new Stack<>();

1. Stack is the subclass of Vector (and Vector implements the List interface).
2. It implements the last-in-first-out data structure (LIFO), i.e., Stack.
3. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean pop(), which defines its properties.

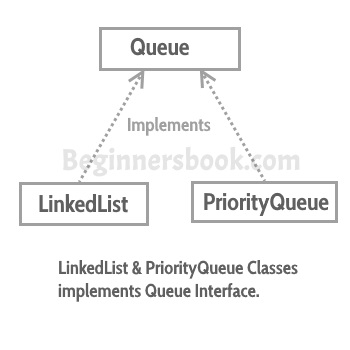
push() - adds an element in the Stack

pop()– pulls out the topmost element of the stack

**Queue Interface**

***A Queue is designed in such a way so that the elements added to it are placed at the end of Queue and removed from the beginning of Queue.  (FIFO)***

Ex. when a new iPhone launches we stand in a queue outside the apple store, whoever is added to the queue has to stand at the end of it and persons are served on the basis of FIFO (**First In First Out**), The one who gets the iPhone is removed from the beginning of the queue.



* **LinkedList, PriorityQueue, Deque, and ArrayDeque** implements Queue interface.
* **Queue is an interface** so we cannot instantiate it, rather we create instance of LinkedList or PriorityQueue and assign it to the Queue like this:

Queue<String> q1 = new ***LinkedList<> ();***

Queue<Integer> q2 = new ***PriorityQueue<> ();***

# PriorityQueue Class

What if we want to serve the request based on the priority rather than FIFO.This can be done with the help of PriorityQueue**, which serves the request based on the priority that we set using**[**Comparator**](https://beginnersbook.com/2017/08/comparator-interface-in-java/)**.**

* **By default,** a PriorityQueue is ordered **according to the natural ordering.** (i.e. sorted in ascending order lexicographically). But,
* A [PriorityQueue](http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html) is what is called a [binary heap](http://en.wikipedia.org/wiki/Binary_heap) (priority heap). It is only ordered/sorted in the sense that the **first element is the least**. ***In other word, it only cares about what is in the front of the queue, the rest are "ordered" when needed***.
* The elements are only ordered as they are dequeued, i.e. removed from the queue using poll(). This is the reason why a PriorityQueue manages to have such good performance, as it is not doing any more sorting than it needs at any time.
* Uses siftUpComparable() internally to compare items when passed without comparator

# Deque Interface

* The Deque is related to the **double-ended queue** that supports addition or removal of elements from either end of the data structure
* Used as a **queue (first-in-first-out/FIFO) or as a stack** (last-in-first-out/**LIFO**).
* These are faster than Stack and LinkedList.

Deque is an interface and has two implementations: **LinkedList and ArrayDeque**. So, we can create the instance of these and assign it to the Deque like this:

Deque<> dq = new ***LinkedList<>();***

Deque<> dq = new ***ArrayDeque<>();***

**Few important features of Deque are:**

* It provides the support of resizable array
* ArrayDeque prohibits the use of Null elements and do not accept any such elements.
* In the absence of external synchronization, Deque is not thread-safe.

### **When to use ArrayList and when to use ArrayDeque?**

ArrayDeque has the ability to add or remove the elements from both ends (head or tail), on the other hand removing last element from ArrayList takes O(n) time as it traverses the whole list to reach the end.

So, if you want to add or remove elements from both ends choose ArrayDeque over ArrayList, however if you only want to perform the operation on the tail (at the end) then you should choose ArrayList.

**SET Interface**

**[ADD MORE]**

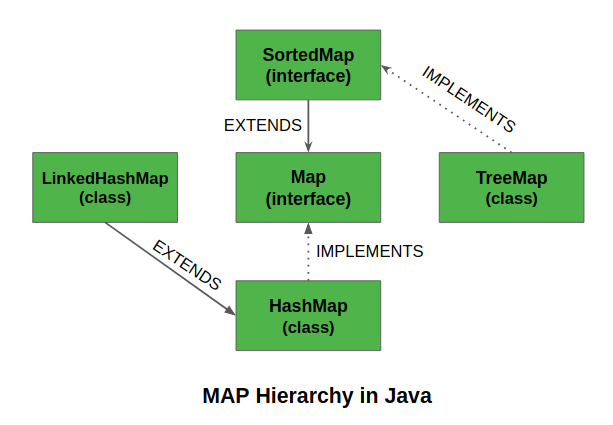
**MAP Interface**

The Map interface is not a subtype of the [Collection](https://www.geeksforgeeks.org/collections-in-java-2/)interface. Therefore, it behaves a bit different from the rest of the collection types.

Few characteristics of the Map Interface are:

1. A Map ***cannot contain duplicate keys*** and each key can map to at most one value.
2. Some implementations allow null key and null value like the [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/)and [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples), but some do not like the [TreeMap](https://www.geeksforgeeks.org/treemap-in-java/).
3. The order of a map depends on specific implementations, e.g [TreeMap](https://www.geeksforgeeks.org/treemap-in-java/)and [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples)have predictable order, while [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/)does not.
4. There are two interfaces for implementing a map in java: ***Map and***[***SortedMap***](https://www.geeksforgeeks.org/sortedmap-java-examples/),

and three classes: [***HashMap***](https://www.geeksforgeeks.org/java-util-hashmap-in-java/)***,***[***TreeMap***](https://www.geeksforgeeks.org/treemap-in-java/)***and***[***LinkedHashMap***](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/)***.***



# SortedMap Interface

* SortedMap is an interface in [collection framework](https://www.geeksforgeeks.org/collections-in-java-2/).
* This interface extends [Map inrerface](https://www.geeksforgeeks.org/map-interface-java-examples/) and provides a total ordering of its elements (elements can be traversed in sorted order of keys).
* [TreeMap](https://www.geeksforgeeks.org/hashmap-treemap-java/) implements this interface

The main characteristic of a SortedMap is that, it orders the keys by their **natural ordering**, **or by a specified comparator**. So, consider using a [TreeMap](https://www.geeksforgeeks.org/hashmap-treemap-java/) when you want a map that satisfies the following criteria:

* null key or null value are not permitted.
* The keys are sorted either by natural ordering or by a specified comparator.

**HashMap Class**

* HashMap uses a technique called Hashing.
* Hashing is a technique of converting a large String to small String that represents the same String. ***A shorter value helps in indexing and faster searches.***
* HashSet also uses HashMap internally.
* The JAVA HashMap class implements the interface Map<K,V>.

The main methods of this interface are:

* put(K key, V value)
* get(Object key)
* remove(Object key)
* Boolean containsKey(Object key)

# Internal Working of HashMap in Java:

# Internally HashMap contains an array of Node and a Node is represented as an inner class which contains 4 fields:

1. int hash // represents the hash value of the key
2. K key
3. V value
4. Node next //reference to another Node, to store entries like a **singly Linked List\***

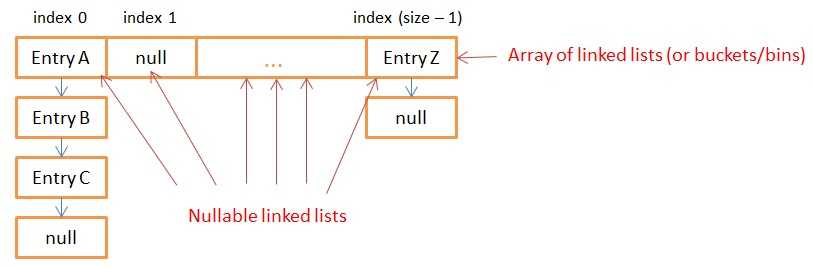
\* Prior to Java 8, Nodes could be extended to Linked Lists but since Java 8, Nodes are extended to **TreeNodes**. *A TreeNode is a red-black tree structure that stores really more information so that it can add, delete or get an element* ***in O(log(n)).***

**Red black trees are self-balancing binary search trees**. Their inner mechanisms ensure that their length is always in log(n) despite new adds or removes of nodes. The main advantage to use those trees is in a case where many data are in the same index (bucket) of the inner table, the search in a tree will cost **O(log(n))** whereas it would have cost **O(n)** with a linked list.

**WORKING:**

* A HashMap stores data into multiple singly linked lists of entries (also called **buckets** or **bins**).
* All the lists are registered in an array of Entry (Entry<K,V>[] array) and the default capacity of this inner array is 16.

The following picture shows the inner storage of a HashMap instance with an array of nullable entries. Each Entry can link to another Entry to form a linked list.



* All the keys with the same hash value are put in the same linked list (bucket
* When a user calls put(K key, V value) or get(Object key), the function computes the index of the bucket in which the Entry should be. Then, the function iterates through the list to look for the Entry that has the same key (using the equals() function of the key).
* In the case of the get(), the function returns the value associated with the entry (if the entry exists).
* In the case of the put(K key, V value), if the entry exists the function replaces it with the new value otherwise it creates a new entry (from the key and value in arguments) at the head of the singly linked list.

**HASHING**

Hashing is a process of converting an object into integer form by using the method hashCode(). Its necessary to write hashCode() method properly for better performance of HashMap. As HashMap also allows null key, so hash code of null will always be 0.

This index of the bucket (linked list) is generated in 3 steps by the map:

* It first gets the **hashcode** of the key.
* It **rehashes** the hashcode to prevent against a bad hashing function from the key that would put all data in the same index (bucket) of the inner array
* It takes the rehashed hash hashcode and **bit-masks** it with the length (minus 1) of the array. This operation assures that the index can’t be greater than the size of the array. You can see it as a very computationally optimized modulo function.

***In order to work efficiently, the size of the inner array needs to be a power of 2***

# AUTO RESIZING

Imagine that the size of the inner array is the default value (16) and you need to store 2 million values. In the best-case scenario, each linked list will have a size of 125 000 entries (2/16 millions). So, each get(), remove() and put() will lead to 125 000 iterations/operations.

To avoid this case, the **HashMap has the ability to increase its inner array** in order to keep very short linked lists.

When you create a HashMap, you can specify an initial size and a loadFactor with the following constructor:

public HashMap(int initialCapacity, float loadFactor)

**The default initialCapacity is 16 and the default loadFactor is 0.75.**

**Capacity**: is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created.

**Load factor:** It is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

The resizing of the array creates twice more buckets (i.e. linked lists) and **redistributes all the existing entries** into the buckets (the old ones and the newly created).

**HASH COLLISION**

***The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.***

In very simple terms, Java Hash table implementations uses following logic for the get and put operations.

1. First identify the “Bucket” to use using the “key” hash code.
2. If there are no objects present in the bucket with same hash code, then add the object for put operation and return null for get operation.
3. If there are other objects in the bucket with same hash code, then “key” equals method comes into play.
   * If equals() return true and it’s a put operation, 🡪 object value is overridden.
   * If equals() return false and it’s a put operation, 🡪 new entry is added to the bucket.
   * If equals() return true and it’s a get operation, 🡪 object value is returned.
   * If equals() return false and it’s a get operation, 🡪null is returned.

# THREAD SAFETY

HashMaps are not threads safe, but why?

Because during the auto-resizing mechanism, if a thread tries to put or get an object, the map might use the old index value and won’t find the new bucket in which the entry is.

The worst case scenario is when 2 threads put a data at the same time and the 2 put() calls resize the Map at the same time. Since both threads modify the linked lists at the same time, the Map might end up with an inner-loop in one of its linked lists. If you try to get a data in the list with an inner loop, the get() will never end.

# TreeMap Class

The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

**[ADD MORE]**

**DATA STRUCTURES**

***A data structure is a particular way of organizing data in a computer so that it can be used effectively.***

* Linear Data Structures) – Array, LinkedList, Stack, Queue
* Binary Tree, BST, Heap and Hash)
* Graph, Trie, Segment Tree and Suffix Tree)

**BINARY TREE Data Structure**

***A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.***



A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child  
     
   **REPRESENTATION IN JAVA:**

/\* Class containing left and right child of current node and key value\*/

class Node

{

int key;

Node left;

Node right;

public Node(int item)

{

key = item;

left = right = null;

}

}

Unlike arrays, linked lists, stack and queues, which are **linear data structures**, **trees are *hierarchical data structures*.**

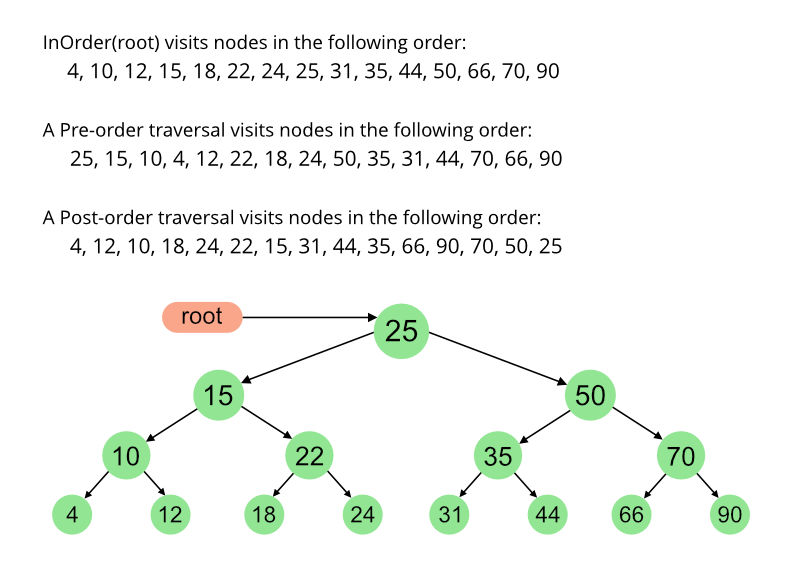
* **Root:** The topmost node is called root of the tree
* **Children**: elements that are directly under an element
* **Parent**: The element directly above something
* **Leaves**: elements with no children (like the bottom most line)
* **Height:** Height of a node is the number of edges from the node to the deepest leaf.

**Why Trees?**

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer
2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists)
4. Trees don’t have an upper limit on number of nodes as nodes are linked using pointers.

**BINARY TREE PROPERTIES:**A Binary Tree can be traversed in two ways:

1. **Depth First Traversal:** Inorder (Left-Root-Right), Pre-order (Root-Left-Right) and Post-order (Left-Right-Root)



1. **Breadth First Traversal:** Level Order Traversal

The maximum number of nodes at level ‘l’ = 2l-1.

Maximum number of nodes = 2h + 1 – 1.

Here ‘*h’* is height of a tree. Height is considered as the maximum number of edges on a path from root to leaf.

Minimum possible height =  ceil(Log2(n+1)) - 1

In Binary tree, number of leaf nodes is always one

more than nodes with two children.

Time Complexity of Tree Traversal: O(n)

# [BFS vs DFS for Binary Tree]

**Binary Search Tree**

**Binary Search Tree** is a ***node-based binary tree data structure*** which has the following properties:  
1. The left subtree of a node contains only nodes with keys less than the node’s key.  
2. The right subtree of a node contains only nodes with keys greater than the node’s key.  
3. The left and right subtree each must also be a binary search tree.

200px-Binary_search_tree.svg

Search: O(h)

Insertion: O(h)

Deletion: O(h)

Extra Space: O(n) for pointers

**h:** Height of BST

**n:** Number of nodes in BST

If Binary Search Tree is Height Balanced,

then h = O(Log n)

Self-Balancing BSTs such as AVL Tree, Red-Black tree and Splay Tree make sure that height of BST

remains O(Log n)

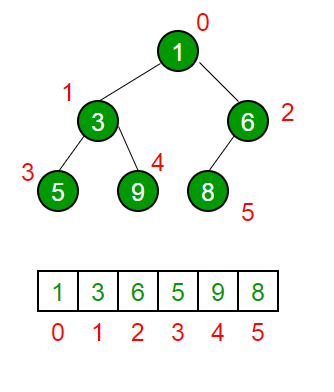
**Binary HEAP**

A Binary Heap is a Binary Tree with following properties.

* 1. It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
  2. A Binary Heap is either **Min Heap or Max Heap**. In a Min Binary Heap, the ***key at root must be minimum among all keys present in Binary Heap***. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

**Applications of Binary Heap:**  
**Priority Queues**: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Heap Implemented priority queues are used in Graph algorithms like [Prim’s Algorithm](http://en.wikipedia.org/wiki/Prim%27s_algorithm) and [Dijkstra’s algorithm](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm).

**Order statistics**: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.



[RED-BLACK TREE, AVL TREE, SPLAY TREE, SKEWED TREE]

**GRAPH Data Structure**

***A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.***

* A Graph is a non-linear data structure.
* **Edges** are lines or arcs that connect any two nodes in the graph.
* Graphs are used to **solve many real-life problems**. Graphs are used to ***represent networks.*** The networks may include paths in a city or telephone network or circuit network.
* Graphs are also used in **social networks like LinkedIn, Facebook.** For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.



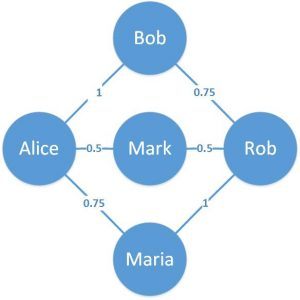
In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

### https://www.baeldung.com/wp-content/uploads/2018/11/graph2.jpg**DIRECTED GRAPH**

The graph we've defined so far has edges without any direction. If these **edges feature a direction in them**, the resulting graph is known as a directed graph.

An example of this can be representing who send the friend request in a friendship on the online portal as shown in the figure

Here, we can see that the edges have a fixed direction. ***The edges can be bi-directional as well.*WEIGHTED GRAPH**

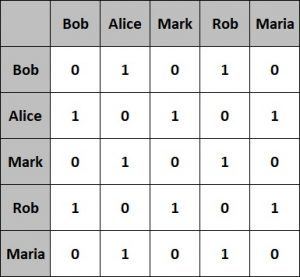
If **edges carry relative weight**, this graph is known as a weighted graph.

An example of a practical application of this can be representing how relatively old is a friendship on the online porta

Here, we can see that the edges have weights associated with them. This provides a relative meaning to these edges.

## **GRAPH REPRESENTATION:**

A graph can be represented in different forms like adjacency matrix and adjacency list. Each one has their pros and cons in a different set-up.

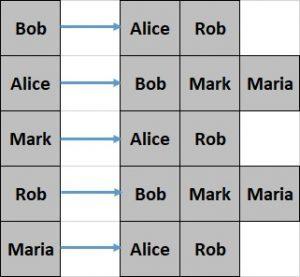
**1. ADJACENCY MATRIX**

An adjacency matrix is **a square matrix with dimensions equivalent to the number of vertices** in the graph.

The elements of the matrix typically have values ‘0' or ‘1'. A value of ‘1' indicates adjacency between the vertices in the row and column and a value of ‘0' otherwise.

The adjacency matrix for the graph from previous section:

This representation is fairly **easier to implement and efficient to query** as well. However, it's **less efficient with respect to space occupied**.

**2. ADJACENCY LIST**

An adjacency list is nothing but**an array of lists**. The size of the array is equivalent to the number of vertices in the graph.

***The list at a specific index of the array represents the adjacent vertices of the vertex represented by that array index.***

This representation is **comparatively difficult to create and less efficient to query**. However, it offers **better space efficiency**.

# Graph Representations

Following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific.

**ADJACENCY MATRIX:**  
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:  
Adjacency Matrix Representation

*Pros:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

*Cons:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

