Data Structures

[Definition](https://en.wikipedia.org/wiki/Data_structure)

A *data structure* is a specialized format/technique/container for *storing and structuring (or organizing)* the *data (large);* so that *data* can be used (retrieve, delete, search etc.) efficiently.

Here, *data* may be *single or multiple collection of items (mostly homogeneous)* and stored on a computer memory using either *contiguously* *or* *non-contiguously.*

[Abstract Data Type(s](https://en.wikipedia.org/wiki/Abstract_data_type)) vs Data Structures

*Abstract Data Type* provides the *logical description* of new data types i.e. describes its properties, operations, etc.

A *data structure* is a concrete implementation of the specification provided by an [***abstract data types***](https://en.wikipedia.org/wiki/Abstract_data_type)(ADT).

Data Structures in Java

In Java platform; the data structures and algorithms are implemented (and defined) in a framework called *Java Collections Framework*.

Java Collections Framework

The *Java Collections Framework* is a unified architecture; which defines and implements the data structures and algorithms.

*A****collection****— sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.*

The primary advantages of a Java collections framework are that it:

1. **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
2. **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
3. **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
4. **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
5. **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
6. **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

The Java collections framework consists of:

All collections frameworks contain the following:

* **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

More on Java collections framework:

<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html>

<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>

<https://docs.oracle.com/javase/tutorial/collections/>

<https://docs.oracle.com/javase/tutorial/collections/intro/>

**Data Structure Terms/Concepts**

1. *Fail-Fast & Fail-Safe* Iterators
   * ***Fail-Fast Iterator*** – Iterator throws ***ConcurrentModificationException*** exception (fails immediately) if collection is modified *structurally* while iterating over it.
     1. Typically this happens in multi-threading environment; one thread modifies the structure of the collection while other thread is iterating over it.
     2. Here *Structural* change means, change in collection’s size due to use of collections add/remove methods (but not due Iterators own add/remove methods) after Iterator is created.
     3. This is because Iterator doesn’t make clone of the collection
   * ***Fail-Safe Iterator*** – Iterator doesn’t throw ***ConcurrentModificationException*** exception (or any exception) even if collection is modified *structurally*
     1. This is because Iterator makes clone of the collection
2. How to synchronize non-synchronized collections?

List list = Collections.synchronizedList(new LinkedList(...));

List list = Collections.synchronizedList(new ArrayList(...));

1. Difference between [**java.util.Iterator<E>**](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html) **&** [**java.util.ListIterator<E>**](https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html) **Interfaces**
   * **Iterator<E> interface** supports *only forward iteration/traversal* and has limited methods
     1. *hasNext(), next(), remove()*
   * Whereas **ListIterator<E> interface** supports *both forward and backward* iteration/traversal has additional methods for easy traversal and access
     1. set(E e), add(E e), previous(), hasPrevious(), nextIndex(), previousIndex()
     2. Extends **Iterator<E> interface**
2. What is [**Spliterator<T> Interface**](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)
3. Why Set doesn’t allow duplicates?
   * Internally SET store element using HASHTABLE ...HASHTABLE is a structure of Key value pairs..Here what the values passed by the SET is treated as Keys of HASHTABLE Internally. Keys are unique cannot be duplicated. That is the reason if you pass any duplicate value it return false and does not added to the SET

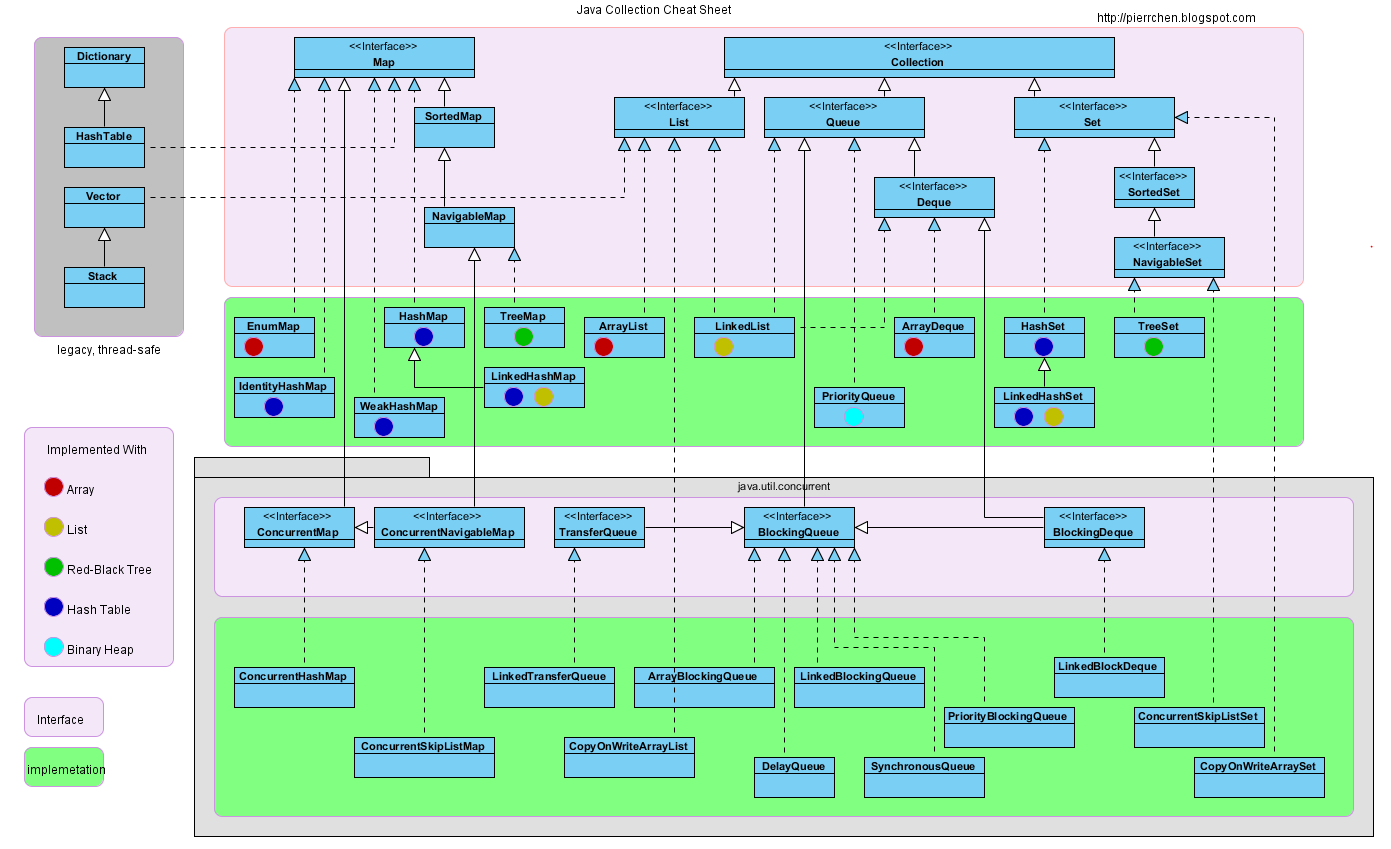
**Data Structures Visualization:**

<https://www.cs.usfca.edu/~galles/visualization/>

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

<http://interactivepython.org/runestone/static/pythonds/index.html>

**List of Data Structures in Java Collections Framework:**



1. **List Data Structure:**  List is an ***ordered*** *data structure*.

The characteristics of List data structure are,

* Elements are stored/arranged ***contiguously*** **&*****orderly***
* **Duplicate Elements** – Yes
* **NULL Elements** – Yes (Multiple)
* **Order of Insertion** – Ordered
* **Insertion** – Positional (indexed)
* **Accessibility** – Positional (indexed)
* **Iteration/Traversal Order** – Sequential

The types of *List* data structure implementations are,

* + Array List (Dynamic Array)
  + Linked List
  + Concurrent Array List (CopyOnWriteArrayList)

[Array List (Dynamic Array):](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

Array List is a *dynamic array* in which Array size grows/shrinks dynamically.

The characteristics of Array List data structure are,

* **Linear Data Structure**
* **Duplicate Elements** – Yes
* **NULL Elements** – Yes (Multiple)
* **Order of Insertion** – Ordered
* **Insertion** – Positional (indexed)
* **Access** – Positional (indexed)
* **Iteration/Traversal Order** – Sequential
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Amortized Constant Time **O (1)** | Amortized Constant Time **O (1)** |
| **Access** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Searching** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Size** | **O (1)** | **O (1)** |

* + Here ***n*** is total number of elements in a ArrayList
* **Big O** – **Space Complexity**:
  + Linear Time O (n)
* **Thread Safe** – No (Non-synchronized)
* **Implementation Base** - Static Array
* **APIs**
  + boolean add(T element) – Inserts the element at the end of the list
    - Always returns *True*
  + boolean add(int index, T element) – Inserts the element at the specified position
    - Always returns *True*
  + T set(int index, T element) - Replaces the element at the specified position in this list with the specified element
  + get(int index) – Retrieves the element from the specified position
  + T remove(int index) – Removes the element from the specified position
  + boolean remove(Object o) – Removes the specified element from the list (first occurrence)
  + Iterator<E> iterator() or listIterator(int index) - Returns the *Fail-Fast* iterator to iterate through the list elements
    - hasNext()
    - next()
    - remove

The implementation details of [Array List](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/ArrayList.java#ArrayList) data structure,

* Resizable-array implementation of the List interface
* Implemented using *static array* ([]) for storing elements
* If list is full, increases the size by copying all elements from existing array to the new static array
* Full implementation details can be found [here](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/ArrayList.java#ArrayList)

[Linked List (Doubly Linked List):](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html)

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

The characteristics of Linked List data structure are,

* Can be used as,
  + **Doubly Linked List**
  + **Queue – add(), poll(), peek()**
  + **Stack – push(), pop(), peek()** methods from Deque interface
* **Duplicate Elements** – Yes
* **NULL Elements** – Yes (Multiple)
* **Order of Insertion** – Ordered
* **Insertion** – Positional (indexed)
* **Access** – Positional (indexed)
* **Iteration/Traversal Order** – Sequential
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion –** at start/end of the list | Constant Time **O (1)** | Constant Time **O (1)** |
| **Insertion –** in between the nodes | Linear Time **O (n)** | Linear Time **O (n)** |
| **Access** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Searching** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion** | Linear Time **O (n)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + Linear Time O (n)
* **Thread Safe** – No (Non-synchronized)
* **Implementation Base** - Linked List
* **APIs** 
  + [boolean add(T element)](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#add-E-) – Inserts the element at the end of the list
    - Always returns *True*
  + boolean add(int index, T element) – Inserts the element at the specified position
  + boolean addFirst(T element) – Inserts the element at the start of the list
  + boolean addLast(T element) – Inserts the element at the end of the list
  + T set(int index, T element) - Replaces the element at the specified position in this list with the specified element
  + T get(int index) – Retrieves the element from the specified position
  + T remove(int index) – Removes the element from the specified position
  + boolean remove(Object o) – Removes the specified element from the list (first occurrence)
  + T remove() or T removeFirst() – Removes head first element in the list
  + T removeLast() – Removes last element in the list
  + boolean removeFirstOccurrence(Object o) or removeLastOccurrence(Object o)
  + Use ListIterator<E> listIterator()/listIterator(int index) instead of Iterator<E> iterator() - Returns the *Fail-Fast* iterator to iterate through the list elements
    - hasNext(),hasPrevious()
    - next(), previous()
    - remove, add()
    - nextIndex(), previousIndex()
  + From Queue Interface - Doesn’t throw exception if collection is empty/non-empty
    - boolean offer(T element) – same as add()
    - T peek(int index) – same as get()
    - T poll(int index) – same as remove()

Synchronized Array List [CopyOnWriteArrayList:](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html)

A thread-safe variant of ArrayList in which all mutative operations (add, remove, and so on) are implemented by making a fresh copy of the underlying array.

The characteristics of CopyOnWriteArrayList data structure are,

* **Duplicate Elements** – Yes
* **NULL Elements** – Yes (Multiple)
* **Order of Insertion** – Ordered
* **Insertion** – Positional (indexed)
* **Access** – Positional (indexed)
* **Iteration/Traversal Order** – Sequential
* **Iterator** – Fail-Safe
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Access** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Searching** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion** | Linear Time **O (n)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + Linear Time O (n)
* **Thread Safe** – Yes (Synchronized)
* **Implementation Base** - Static Array
* **APIs**

The implementation details of [CopyOnWritteArrayList](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/concurrent/CopyOnWriteArrayList.java#CopyOnWriteArrayList) data structure,

* Resizable-array, thread safe implementation of the List interface
* All mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array
* Implemented using *static array* ([]) for storing elements
* If list is full, increases the size by copying all elements from existing array to the new static array
* Full implementation details can be found [here](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/concurrent/CopyOnWriteArrayList.java#CopyOnWriteArrayList)

[Skip Lists:](http://www.geeksforgeeks.org/skip-list/) *A skip lists* is a [*randomized*](https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf) data structure that allows ***fast search*** *O (log (n))* within an ***ordered/sorted sequence*** of elements.

The characteristics of perfect *Skip Lists* data structure is,

* Elements are stored in a *sorted order*
* O(log n) higher levels -- because you cut the # of items in each level
* Each higher level contains 1/2 the elements of the level below it.
* Called *skip lists* because higher level lists let you skip over many items
* Skip Lists are a randomized data structure: the same sequence of inserts / deletes may produce different structures depending on the outcome of ***random coin flips***
* Fast search is made possible by maintaining a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one
* **Big O** – **Time Complexity**:
  + ***Insertion*** –O(log(n)) [Why O (log (n))?](https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf)
    - First search for the position to insert new element
    - Also, Insertion/deletion might need to rearrange the entire list to put it in a ordered sequence
  + ***Access*** – O(log(n))
  + ***Search*** – O(log(n))
  + ***Deletion*** – O(log(n))
    - Why O (log (n))? Insertion/deletion might need to rearrange the entire list to put it in a ordered sequence

[Self-Organizing Lists:](http://www.geeksforgeeks.org/self-organizing-list-set-1-introduction/) In a *Self-Organizing List* more frequently accessed items are placed closer to the head. Following are different strategies used by *Self-Organizing Lists*.

* **Move-to-Front Method:** Any node searched is moved to the front. This strategy is easy to implement, but it may over-reward infrequently accessed items as it always move the item to front.
* **Count Method:** Each node stores count of the number of times it was searched. Nodes are ordered by decreasing count. This strategy requires extra space for storing count.
* **Transpose Method:** Any node searched is swapped with the preceding node. Unlike Move-to-front, this method does not adapt quickly to changing access patterns.

The characteristics of perfect *Self-Organizing Lists* data structure is,

* **Big O** – **Time Complexity**:
  + ***Insertion*** –?
  + ***Access*** –?
  + ***Search*** –?
  + ***Deletion*** –?

[Unrolled Linked List:](http://www.geeksforgeeks.org/unrolled-linked-list-set-1-introduction/) An *Unrolled Linked List* is a *specialized Linked List* inwhich, at each node multiple (array of) elements are stored; instead of just one as in case of standard linked list. The characteristics of *Unrolled Linked Lists* data structure is,

* Instead of storing single element at a node, *unrolled linked lists* store an array of elements at a node.
* Unrolled linked list covers advantages of both array and linked list as it reduces the memory overhead in comparison to simple linked lists by storing multiple elements at each node and it also has the advantage of fast insertion and deletion as that of a linked list.
* **Advantages:**
  + Because of the Cache behavior, linear search is much faster in unrolled linked lists.
  + In comparison to ordinary linked list, it requires less storage space for pointers/references.
  + It performs operations like insertion, deletion and traversal more quickly than ordinary linked lists (because search is faster).
* **Disadvantages:**
  + The overhead per node is comparatively high than singly linked lists. Refer an example node in below code.
* **Big O** – **Time Complexity**:
  + ***Insertion*** –?
  + ***Access*** –?
  + ***Search*** –?
  + ***Deletion*** –?

1. [**Set Data Structure:**](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)A collection that contains ***no duplicate elements***.

The characteristics of *Set* data structure are,

* Elements are stored/arranged ***unorderly***
* **Duplicate Elements** – No, why?
* **NULL Elements** – Yes (at most one)
* **Order of Insertion** – Ordered or Unordered (doesn’t guarantee’s order of insertion)
* **Access** – Only through Iterator (no get() method)
* **Iteration/Traversal Order** – Ordered or Unordered
* **Iterator** – Fail-Fast

**[HashSet:](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html)**

This class implements the Set interface, backed by a hash table (actually a HashMap instance)

The characteristics of HashSet data structure are,

* **Duplicate Elements** – No, why?
* **NULL Elements** – Yes (at most one)
* **Order of Insertion** – Unordered (doesn’t guarantee’s order of insertion)
* **Insertion** – Random (no index)
* **Access** – Only through Iterator (no get() method)
* **Iteration/Traversal Order** – Random
* **Iterator** – Fail-Fast
* **Implementation Base** - Hash Table (actually a HashMap instance)
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Access** | N/A - no get() method | N/A - no get() method |
| **Searching** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Deletion** | Constant Time **O (1)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No

The implementation details of [HashSet](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/HashSet.java#HashSet) data structure,

* Implemented using *HashMap class* for storing elements. Uses elements as “Key” for HashMap
* Full implementation details can be found [here](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/HashSet.java#HashSet)

**Linked[HashSet:](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html)**

Hash table and linked list implementation of the *Set* interface, with **predictable iteration order**

The characteristics of *LinkedHashSet* data structure is,

* **Duplicate Elements** – No
* **NULL Elements** – Yes (at most one)
* **Order of Insertion** – Ordered
* **Insertion** – Random (no index)
* **Access** – Only through Iterator (no get() method)
* **Iteration/Traversal Order** – Ordered (Order of Insertion)
* **Iterator** – Fail-Fast
* **Implementation Base** - Hash Table (actually a HashMap instance)
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Access** | N/A - no get() method | N/A - no get() method |
| **Searching** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Deletion** | Constant Time **O (1)** | Linear Time **O (n)** |

* + In **worst case** scenario why O(n)?
    - If hash function is not implemented efficiently
    - If same key maps to same bucket index in a hash table
* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No

The implementation details of [LinkedHashSet](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/HashSet.java#HashSet) data structure,

* Implemented using Linked*HashMap class* for storing elements. Uses elements as “Key” for HashMap
* Full implementation details can be found [here](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/LinkedHashSet.java#LinkedHashSet)

**[TreeSet](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html)**[:](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html)

*TreeSet* is a *TreeMap* (actually ***Red-Black tree****)* based implementation of NavigableSet interface (or Map data structure).

The characteristics of TreeSet data structure is,

* The set is ***sorted*** (*in* *ascending order)*according to the [***natural ordering***](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) *of its elements, or by a*[***Comparator***](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) provided at set creation time, depending on which constructor is used.
* Here *natural ordering* is achieved using Compararable<T> interface i.e. each *element* class must implement Comparable<> interface in order to compare against other key by using CompareTo() method
* Elements are sorted in ***ascending order***
* **Duplicate Elements** – No
* **NULL Elements** – No
* **Order of Insertion** – Natural ordering
* **Access** – No get method
* **Iteration/Traversal Order** – Sorted Order
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Access** | N/A | N/A |
| **Searching** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Deletion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No (Non-Synchronized)
* **Implementation Base** - Red Black Tree
* **APIs**
  + Practice API’s properly

The implementation details of [TreeSet](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/TreeSet.java) data structure is,

* Implemented using ***Red-Black Tree***

[**CopyOnWriteArraySet**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArraySet.html)**:**

Synchronized *Set* based on *CopyOnWriteArrayList*

The characteristics of *CopyOnWriteArraySet* data structure is,

* **Duplicate Elements** – No
* **NULL Elements** – Yes (at most one)
* **Order of Insertion** – Ordered
* **Insertion** – Random (no index)
* **Access** – Only through Iterator (no get() method)
* **Iteration/Traversal Order** – Ordered (Order of Insertion)
* **Iterator** – Fail-Safe
* **Implementation Base** - *CopyOnWriteArrayList*
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Access** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Searching** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion** | Linear Time **O (n)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No

**[ConcurrentSkipListSet](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html)**[:](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html)

*ConcurrentSkipListSet* is a *ConcurrentSkipListMap* (actually [***Skip-List***](https://en.wikipedia.org/wiki/Skip_list)*)* based implementation of NavigableSet interface.

The characteristics of *ConcurrentSkipListSet* data structure is,

* The set is ***sorted*** (*in* *ascending order)*according to the [***natural ordering***](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) *of its elements, or by a*[***Comparator***](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) provided at set creation time, depending on which constructor is used.
* Here *natural ordering* is achieved using Compararable<T> interface i.e. each *element* class must implement Comparable<> interface in order to compare against other key by using CompareTo() method
* Elements are sorted in ***ascending order***
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Natural ordering
* **Access** – By *Key*
* **Iteration/Traversal Order** – Sorted Order
* **Iterator** – Fail-Safe
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Logarithmic Time **O (log(n))** | Linear Time **O (n)** |
| **Access** | Logarithmic Time **O (log(n))** | Linear Time **O (n)** |
| **Searching** | Logarithmic Time **O (log(n))** | Linear Time **O (n)** |
| **Deletion** | Logarithmic Time **O (log(n))** | Linear Time **O (n)** |
| **Size** | O (n) | O (n) |

* + The size method is not a constant-time operation. Because of the asynchronous nature of these sets, determining the current number of elements requires a traversal of the elements.
* **Big O** – **Space Complexity**:
  + **Worst Case** – O (n log(n))
* **Thread Safe** – Yes (Synchronized)
* **Implementation Base** - Red Black Tree
* **APIs**
  + Practice API’s properly

The implementation details of [TreeSet](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/TreeSet.java) data structure is,

* Implemented using ***Red-Black Tree***

1. [Map Data Structure:](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)*Map* is a *data structure* in which data is stored in the form of *key-value* pairs where every *key* is unique and maps to value hence the name *Map*.

The characteristics of *Map* data structure are,

* Easy to *search* O(1)
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Ordered or Unordered (based on implementation class)
* **Access** – By *Key*
* **Iteration/Traversal Order** – Ordered or Unordered
* **Iterator** – Fail-Fast

Implementations of *Map* data structure are,

[HashMap:](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)

HashMap is a *Hash table* based implementation of the *Map* interface which maps keys to values.

The characteristics of [HashMap](http://javabypatel.blogspot.in/2015/09/hashmap-data-structure-and-hashcode.html) data structure are,

* Easy to *Search/Access O(1)*
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Unordered
* **Access** – By *Key*
* **Iteration/Traversal Order** – Unordered
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Access** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Key** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Value** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion Key** | Constant Time **O (1)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No (Non-Synchronized)
* **Implementation Base** - Hash Table
* **APIs**
  + put(k, v), get(k)

The implementation details of [HashMap](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/HashMap.java) data structure is,

* Implemented using [*Hash Table*](http://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/)

[LinkedHashMap:](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html)

LinkedHashMap is a *Hash table* based implementation of the *Map* interface which maps keys to values with **predictable iteration order**.

The characteristics of LinkedHashMap data structure is,

* Easy to *Search/Access O(1)*
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Ordered
* **Access** – By *Key*
* **Iteration/Traversal Order** – Ordered
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Access** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Key** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Value** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion Key** | Constant Time **O (1)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No (Non-Synchronized)
* **Implementation Base** - Hash Table
* **APIs**
  + put(k, v), get(k)

The implementation details of [LinkedHashMap](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/LinkedHashMap.java) data structure is,

* Implemented using [*Hash Table*](http://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/)

[WeakHashMap:](https://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html)

*Hash table* based implementation of the *Map interface,* with *weak keys*.

The characteristics of *WeakHashMap* data structure is,

* An entry in a *WeakHashMap* will automatically be removed when its key is no longer in ordinary use
* [When to use](http://www.baeldung.com/java-weakhashmap)? For efficient memory cache
* How to use?
  + weakHashMap.put(new String("Maine"), "Augusta"); // No reference to key i.e. put and forget. GC will reclaim the memory in next GC cycle.
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Unordered
* **Access** – By *Key*
* **Iteration/Traversal Order** – Unordered
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Access** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Key** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Value** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion Key** | Constant Time **O (1)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No (Non-Synchronized)
* **Implementation Base** - Weak Hash Table

The implementation details of *WeakHashMap* data structure is,

* Implemented using ***Weak Hash Table***

[IdentityHashMap:](https://docs.oracle.com/javase/7/docs/api/java/util/IdentityHashMap.html)

*Hash table* based implementation of the *Map interface,* which uses reference-equality in place of object-equality when comparing keys (and values) i.e.

The characteristics of *WeakHashMap* data structure is,

* Two keys k1 and k2 are considered equal if and only if (k1==k2).
* In normal Map implementations (like HashMap) two keys k1 and k2 are considered equal if and only if k1.equals(k2)
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Unordered
* **Access** – By *Key*
* **Iteration/Traversal Order** – Unordered
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Access** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Key** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Value** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion Key** | Constant Time **O (1)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No (Non-Synchronized)
* **Implementation Base** - Identity Hash Table

The implementation details of *IdentityHashMap* data structure is,

* Implemented using ***Identity Hash Table***

[EnumMap:](https://docs.oracle.com/javase/7/docs/api/java/util/EnumMap.html)

A specialized Map implementation for use with *enum type keys.*

The characteristics of *EnumMap* data structure is,

* Keys are always ***Enum***
* All of the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created.
* Enum maps are represented internally as arrays. This representation is extremely compact and efficient.
* **Enum maps are maintained in the natural order of their keys (the order in which the enum constants are declared).**
* Iterators returned by the collection views are weakly consistent: they will never throw ConcurrentModificationException and they may or may not show the effects of any modifications to the map that occur while the iteration is in progress.
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – No (throws NPE)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Ordered (enum ordinal order)
* **Access** – By *Key*
* **Iteration/Traversal Order** – Ordered (enum ordinal order)
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Access** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Searching Key** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Searching Value** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion Key** | Constant Time **O (1)** | Constant Time **O (1)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No (Non-Synchronized)
* **Implementation Base** - Dynamic Array

The implementation details of *EnumMap* data structure is,

* Implemented using ***Dynamic Array***

[TreeMap:](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html)

*TreeMap* is a ***Red-Black tree*** based implementation of NavigableMap interface (or Map data structure).

The characteristics of TreeMap data structure is,

* *TreeMap* provides *both* [***sorting***](https://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)*and*[***navigable***](https://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html) (methods are designed for locating) functionalities.
* The map is ***sorted*** (*in* *ascending order)*according to the [***natural ordering***](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) ***of its keys, or by a*** [***Comparator***](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) provided at map creation time, depending on which constructor is used.
* Here *natural ordering* is achieved using Compararable<T> interface i.e. each *key* class must implement Comparable<> interface in order to compare against other key by using CompareTo() method
* Keys are sorted in ***ascending order***
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Natural ordering
* **Access** – By *Key*
* **Iteration/Traversal Order** – Sorted Order
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Access** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Searching Key** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Searching Value** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion Key** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – No (Non-Synchronized)
* **Implementation Base** - Red-Black Tree
* **APIs**
  + Practice API’s properly

The implementation details of [TreeMap](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/TreeMap.java) data structure is,

* Implemented using ***Red-Black Tree***

[ConcurrentHashMap:](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html)

*Synchronized/thread safe* version of *HashMap*

The characteristics of ConcurrentHashMap data structure is,

* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Unordered
* **Access** – By *Key*
* **Iteration/Traversal Order** – Unordered
* **Iterator** – Fail-Safe
* **Big O** – **Time Complexity**:
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Access** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Key** | Constant Time **O (1)** | Linear Time **O (n)** |
| **Searching Value** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion Key** | Constant Time **O (1)** | Linear Time **O (n)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** - O(n)
* **Thread Safe** – Yes (Synchronized)
* **Implementation Base** - Hash Table
* **APIs**
  + put(k, v), get(k)

[ConcurrentSkipListMap:](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html)

*ConcurrentSkipListMap* is a thread-safe/synchronized ***Skip-List*** based implementation of *ConcurrentNavigableMap* interface.

The characteristics of TreeMap data structure is,

* *ConcurrentSkipListMap* provides *both* [***sorting***](https://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)*and*[***navigable***](https://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html) (methods are designed for locating) functionalities.
* The map is ***sorted*** (*in* *ascending order)*according to the [***natural ordering***](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) ***of its keys, or by a*** [***Comparator***](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) provided at map creation time, depending on which constructor is used.
* Here *natural ordering* is achieved using Compararable<T> interface i.e. each *key* class must implement Comparable<> interface in order to compare against other key by using CompareTo() method
* Keys are sorted in ***ascending order***
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Natural ordering
* **Access** – By *Key*
* **Iteration/Traversal Order** – Sorted Order
* **Iterator** – Fail-Safe
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Logarithmic Time **O (log(n))** | Linear Time **O (n)** |
| **Access** | Logarithmic Time **O (log(n))** | Linear Time **O (n)** |
| **Searching Key** | Logarithmic Time **O (log(n))** | Linear Time **O (n)** |
| **Searching Value** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Deletion Key** | Logarithmic Time **O (log(n))** | Linear Time **O (n)** |
| **Size** | O (n) | O (n) |

* + The size method is not a constant-time operation. Because of the asynchronous nature of these sets, determining the current number of elements requires a traversal of the elements.
* **Big O** – **Space Complexity**:
  + **Worst Case** – O (n log(n))
* **Thread Safe** – Yes (Synchronized)
* **Implementation Base** - Skip-List

1. [Queue Data Structure:](https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm)*Queue* is abstract data type in which one end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows F-I-F-O methodology i.e., the data item stored first will be accessed first.

**Types of Queues**

* **Unbounded Queue** – No restriction on size, queue grows dynamically
* **Bounded Queue** – Restriction on size, queue doesn’t grow. Most of the blocking queues are bounded queues
* **Single Ended Queue** –Insertion or deletion occurs only one on side
* **Doubled Ended Queue** **(Dequeue)**–Insertion or deletion occurs on both the side
* **Blocking Queue** – enqueue or dequeue operation blocks if
  + Queue is empty (take() method blocks) or full (put() method blocks) respectively [OR]
  + Producer blocks if there are no waiting consumer or consumer blocks if queue is empty
* **Priority Queue**

**API’s** – There are three variants of APIs for each basic operations insertion, deletion,

* Throws Exception (If queue is empty/full) – add(), remove(), element()
* Returns Special Value (If queue is empty/full) – offer(), poll(), peek()
* Blocks (If queue is empty/full) – put(), take(), not applicable
* **Insert** 
  + add() - If queue is full, Bounded queue - Throws exception, Unbounded Queue – Doesn’t throw exception, grows queue’s size
  + peek() - If queue is full, Returns special value i.e. always returns false for both Bounded & Unbounded queue
* **Remove** (Retrieves & removes head of the queue)
  + remove() - If queue is empty, Bounded queue - Throws exception, Unbounded Queue – Doesn’t throw any exception instead returns false
  + poll() - If queue is empty, Bounded queue – Returns special value i.e. always false, Unbounded Queue – Returns special value i.e. always false
* **Access/Examine** (Retrieves head of the queue but doesn’t remove)
  + element() - If queue is empty, Bounded queue - Throws exception, Unbounded Queue – Doesn’t throw any exception instead returns NULL always
  + peek() - If queue is empty, Bounded queue – Returns special value i.e. always NULL, Unbounded Queue – Returns special value i.e. always NULL

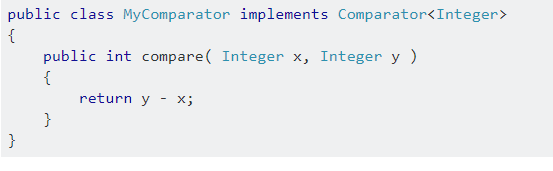
The implementations of *Queue* data structure are,

[Priority Queue:](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html)

An unbounded (no size/capacity restriction), ***Binary Heap*** *(actually* ***Min-Binary Heap****)* based queue in which the elements are ***ordered*** according to their **natural ordering.**

The characteristics of *Priority Queue* data structure are,

* PQ has 2 variants of constructor
  + Default Constructor 🡪 Uses natural ordering of the elements (comparable.compareTo())🡪 ***Min-Binary Heap***
  + Specialized Constructor 🡪 Accepts comparator (comparator.compare ())🡪 ***Min/Max-Binary Heap***
* PQ is implanted as Min-Binary Heap but can be used as Max-Binary Heap by using specialized constructor and modifying comparoto.compare() method



* + Instead of x - y use y - x
* Does not permit insertion of non-comparable objects
* A priority queue is **unbounded**, but has an internal capacity governing the size of an array used to store the elements on the queue
* **NULL Element** – No
* **Iterator** – Fail-Fast
* **Order of Insertion** – Natural ordering
* **Access** – By *Key*
* **Iteration/Traversal Order** – No order
  + The Iterator provided in method iterator() is not guaranteed to traverse the elements of the priority queue in any particular/priority order.
  + If you need ordered traversal, consider using Arrays.sort(pq.toArray()).
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Enqueueing** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Access – getMin()/getMax()** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Searching** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Dequeuing – extractMin()/extractMax()** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Deletion** | Linear Time O (n) | Linear Time O (n) |
| **Size** | Constant Time **O (1)** | Constant Time **O (1)** |

* **Big O** – **Space Complexity**:
  + **Worst Case** – O (n)
* **Implementation Base** *–* [Implemented](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/PriorityQueue.java) using[Binary Heap](http://www.geeksforgeeks.org/binary-heap/)
* **API’s** – Doesn’t support updating element (no decreaseKey())

[Linked Blocking Queue:](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedTransferQueue.html)

Linked Blocking Queue is a **blocking queue** based on a ***Linked List*** which implements the BlockingQueue interface.

The characteristics of *Linked Blocking Queue* data structure are,

* Linked Blocking Queue is a bounded queue
* put(), take() API’s blocks if queue is full, empty respectively
* **NULL Element** – Yes
* **Iterator** – Fail-Safe
* **Order of Insertion** –Ordered
* **Iteration/Traversal Order** – Insertion Order
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Enqueueing** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Searching** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Dequeuing – poll()** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Deletion** | Linear Time O (n) | Linear Time O (n) |
| **Size** | Constant Time **O (1)** | Constant Time **O (1)** |

* + ***Enqueue*** –O(1)
  + ***Deueue*** –O(1)
  + ***Contains*** - O(n)
* **Implementation Base** *–* [Implemented](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/PriorityQueue.java) using *Linked List*

[Synchronous Queue:](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html)

A specialized blocking queue in which each (only one) insert operation must wait for a corresponding remove operation by another thread, and vice versa.

A synchronous queue does not have any internal capacity, not even a capacity of one.

The characteristics of *Synchronous Queue* data structure are,

* Only put(), take() method is supported and all other method returns false or null
* **NULL Element** – No
* **Iterator** – No iterator as there are no elements in queue
* **Order of Insertion** –No insertion
* **Iteration/Traversal Order** – No traversal as there are no elements in queue
* **Big O** – **Time Complexity**:
  + ***Enqueue*** –O(1)
  + ***Deueue*** –O(1)
* **Implementation Base** *–*?

[Linked Transfer Queue:](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedTransferQueue.html)

Linked Transfer Queue is an **unbounded queue** based on a ***Linked List*** which implements the TransferQueue interface.

The characteristics of *Linked Transfer Queue* data structure are,

* Transfer Queue is a unbounded queue
* Provides special blocking methods (along with standard methods) transfer(), tryTransfer() used in a producer-consumer model. If there are no waiting consumers then
  + transfer() method blocks until consumers takes it
  + tryTransfer() method simply returns it (doesn’t put in to queue)
* **NULL Element** – Yes
* **Iterator** – Fail-Safe
* **Order of Insertion** –Ordered
* **Iteration/Traversal Order** – Insertion Order
* **Big O** – **Time Complexity**:
  + ***Enqueue*** –O(1)
  + ***Deueue*** –O(1)
  + ***Contains*** - O(n)
* **Implementation Base** *–* [Implemented](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/PriorityQueue.java) using *Linked List*

[Delay Queue:](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html)

An unbounded blocking queue of ***Delayed*** elements, in which an element can only be taken when its delay has expired.

The characteristics of *Delay Queue* data structure are,

* Delay Queue is a unbounded queue
* All the elements must implement *Delayed* interface

1. [Tree Data Structure:](https://en.wikipedia.org/wiki/Tree_(data_structure))*Tree* is an abstract data type which represents the ***'hierarchical'*** tree structure, consisting of set of ***linked nodes***.

*For more info* –

https://www.youtube.com/watch?v=oSWTXtMglKE&index=3&list=PLEMvscCKqIzdWMBSZC2-D8DKNKTARxq

https://www.youtube.com/watch?v=qH6yxkw0u78&index=25&list=PL2aWCzGMAwI3WJlcBbtYTwiQSsOTa6P

https://en.wikipedia.org/wiki/Tree(datastructure)

**Why/when to Use Tree Data Structure?**

* **Non-Linear Data Structure – Hierarchical Data Structure**
* For Quick Search, Insertion, Deletion
* Hierarchical Representation
  + File System
  + Organizational Structure
  + Networking

**Terminologies**

<http://btechsmartclass.com/DS/U3_T1.html>

<http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/>

<https://en.wikipedia.org/wiki/Tree(datastructure)#Terminology>

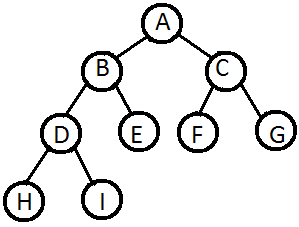
* **Depth** - Depth of a "Node X" (always referred to Node not to Tree) is the total no. of edges from "Root Node" to the given Node "X" - Depth of a "Root Node" is always "0"
* **Height** - Height of "Node X" is a total no. of edges from given "Node X" to "Leaf" Node in a *"Longest Path"* 
  + ***Height of "Tree"*** is a total no. of edges from "Root Node" to "Leaf" Node in a "Longest Path"
  + ***Height of "Root Node" = Height of "Tree"*** - Height of Tree with only "Root Node" is always "0"
* **Level** - Level 1 (Root Node), Level 2, Level 3 and so on...
  + **Level = Depth + 1**
* **Leaf Node** - Node with no children
* **Left/Right Subtree** - Tree from Left/Right child node of the root node to all leaf nodes
* Total no. of "Edges" = Total no. of nodes – 1 (Here 1 is root node)
* **Degree –** Degreerepresentsthemaximum number of children each node can have in a tree. For example, the order of binary tree is 2.

**Binary Tree**

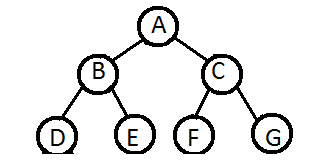
*Binary Tree* is a specialized tree structure, in which each node has no more than two children referred as *left and right child nodes*. *Binary Tree* is a most commonly used tree data structure.

**Types of Binary Trees**

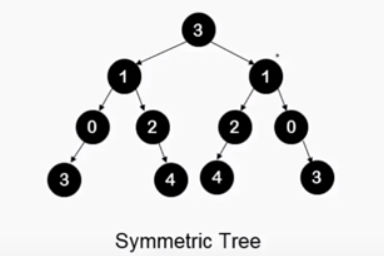
1. **Balanced/Height-Balanced Binary Tree -** In Balanced tree the height difference between the left and right subtree's of each node is no more than one.
2. **Perfect Balanced/Perfect Height-Balanced Tree -** In *Perfect Balanced* tree the height of left and right subtree's of each node is same
   * It’s very difficult to achieve perfect balanced tree
3. [**Self-Balancing Tree**](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) **-** In *Self-Balancing* tree in the event of insertion and deletion; tree is automatically *balanced.*
   * *Red-Black Tree*
   * *AVL Tree*
4. **Full/Proper/Strict Binary Tree -** In *Strict Binary* tree; each node contains the two children except for the leaves node i.e. no single/zero child.
5. **Complete Binary Tree -** In *Complete Binary* tree; each *level* is completely filled except for the possibly last level, and all the nodes in the last level are as far left as possible.
   * Here level is completely *filled* means, in a given level each node’s parent has two children.
   * In the below diagram, at level 3; all the node’s ( D, E, F, G) parent (B, C) has two children i.e. filled except for the last level 4



1. **Perfect Binary Tree –** In*Perfect Binary* tree all the nodes are completely filled (even including last level unlike *Complete Binary Tree*) and all the ***leaves*** have the same depth or level
   * *Perfect Binary* tree is also a *Complete Binary Tree*

******

1. [**Ordered Binary Tree**](https://en.wikipedia.org/wiki/Tree_(graph_theory)#Plane_tree) **–** In*Ordered Binary* tree, children of a node is “*ordered”* as per the ordering rule. For example, *Binary Search Tree* is an *ordered tree* with following *order* is applied for children
   * The left subtree of a node contains only nodes with keys less than the node’s key.
   * The right subtree of a node contains only nodes with keys greater than the node’s key
   * References
     + <http://cs.lmu.edu/~ray/notes/orderedtrees/>
2. [**Symmetric Binary Tree**](https://www.youtube.com/watch?v=8t4KCaDB108) **–** In *Symmetric Binary Tree,* theleft sub tree is mirror image of the right sub tree or vice-versa to its root node.



**Binary Tree Traversal**

<https://www.youtube.com/watch?v=9RHO6jU--GU&index=32&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P>

<https://www.youtube.com/watch?v=gm8DUJJhmY4&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=34>

<https://www.youtube.com/watch?v=86g8jAQug04&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=33>

**Types of Binary Tree Traversal**

1. [**Depth-First Traversal**](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)is based on [*Depth-First Search*](https://en.wikipedia.org/wiki/Depth-first_search)algorithm.
   * [**In-Order Traversal**](http://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/)Order of the traversal is **Left Child Node 🡪 Parent Node 🡪Right Child Node**
   * [**Pre-Order Traversal**](http://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) **-** Order of the traversal is **Parent** **Node 🡪Left Child Node 🡪 Right Child Node**
   * [**Post-Order Traversal**](http://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) **-** Order of the traversal is **Left Child Node 🡪Right Child Node 🡪 Parent Node**
2. [**Breadth-First Traversal**](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)is based on [*Breadth-First Search*](https://en.wikipedia.org/wiki/Breadth-first_search)algorithm. In this case, nodes are visited/traversed by at each level.

* [**Level-Order Traversal**](http://www.geeksforgeeks.org/level-order-tree-traversal/) **-** All the nodes are visited at each level (left to right)

**[Binary Search Tree](http://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)**

[*Binary Search Tree (BST)*](https://www.cs.swarthmore.edu/~newhall/unixhelp/Java_bst.pdf), is an ***ordered binary tree*** data svtructure which has the following properties,

* The left subtree of a node contains only nodes with keys less than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.
* There must be no duplicate nodes.
* Easy to *Search* hence called as Binary Search Tree
* BST is a ***Unbalanced Binary Tree***
* **Big O** – **Time Complexity**:
  + **Insertion** – O(log(n)) - O(n) (worst case)
  + **Deletion** – O(log(n)) - O(n) (worst case)
  + **Search** – O(log(n)) - O(n) (worst case)
* **Advantages**
  + Easy to Insert, Delete and Search with O(log(n)) in average case
* **Disadvantages**
  + Since BST is an ***Unbalanced Binary Tree;*** if the items to be inserted are *sorted,*then the tree degenerates to a list. For example if you insert numbers between 1-100 in the order then it BST looks like a tree (items are always added to right)
  + How to overcome above problem? By implementing BST as a *Balanced Tree* like Red-Black Tree, AVL Tree
* **Implementation Details**
  + Create node class BSTNode with following member variables
    - Left, right child node, parent node
    - data
  + **Remove**
    - Search the node to be deleted
    - Check if node to be deleted is,
      * **Leaf node (no children)** – Remove the reference from parent node i.e. set left/right child to NULL
      * **Has only one child** – Replace node to be deleted value (data) with the child node value and remove the reference of child node from parent node i.e. set left/right child to NULL
      * **Has two children**
        + Find the smallest/largest node from either right/left sub tree respectively (root node is node to be deleted)
        + Replace node to be deleted value (data) with the smallest/largest node value and remove the smallest/largest node (root node is node to be deleted)
* References
  + <http://pages.cs.wisc.edu/~siff/CS367/Notes/bsts.html>
  + <https://www.cs.swarthmore.edu/~newhall/unixhelp/Java_bst.pdf>
  + Visualization 🡪 <https://www.cs.usfca.edu/~galles/visualization/BST.html>

**Types of [Binary Search Tree](http://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)**

1. **Red-Black Tree** – *Red–Black Tree (RBT)* is a ***Self-Balanced BST*** in which tree's height is automatically "balanced" in the event of insertion, deletion.

**The Properties of RBT**

* Each node is either Red or Black
* Root node and NULL descendants (leaf nodes) are always Black
* Red parent node can't have Red child node
* **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Access** | N/A | N/A |
| **Searching** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Deletion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Height** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |

* **Big O** – **Space Complexity**:
  + Worst Case – **O (n)**

[**How to Fix RBT?**](https://www.youtube.com/watch?v=tJ7niBAhDQI&index=8&t=86s&list=PLE_MvscCKqIzdWMBSZC2-D8DKNK_TARxq)

* If tree violates either one of the above property then tree rotation will be done to fix it.
* Rotate tree either left or right or both
* Visualization 🡪<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>
* **Scenario#1: If Uncle is Red – Push Blackness down from grandparent (Re-Coloring)**
  + Push blackness down from grandparent to parent and uncle 🡪 now, both parent & uncle nodes are Black and GP node color is turned to Red. If GP is root node then change GP color to Black
* **Scenario#2: If Uncle is Black – Tree Rotation (Left or Right)**
  + **Left-Left-Right Rule->** If inserted node is **LEFT** child and parent node is also **LEFT** child node then,
    - Rotate the tree in **RIGHT** direction from its parent node (single rotation) and
    - Pass the color (red) of parent node to its child nodes
  + **Right-Right-Left Rule ->**If inserted node is **RIGHT** child and parent node is also **RIGHT** child node then,
    - Rotate the tree in **LEFT** direction from its parent node (single rotation) and
    - Pass the color (red) of parent node to its child nodes
  + **If inserted node is LEFT/RIGHT child and parent node is RIGHT/LEFT node then rotate the tree two times**
    - **1st Rotation** – Make inserted node and parent node relation to be either LEFT-LEFT or RIGHT-RIGHT.
      * Rotate the tree either by left or right to its inserted node if inserted node is right or left child node respectively.
      * In this case only inserted node and parent position will be altered (grandparent won’t change) i.e. child node becomes parent node and parent node becomes child node
    - **2nd Rotation –** Rotate thetree as per the **LEFT-LEFT-RIGHT or RIGHT-RIGHT-LEFT Rule**

1. [**AVL Tree**](https://www.hackerrank.com/challenges/self-balancing-tree) – AVL tree is similar to Red-Black tree but with height property.
   * Check for the height property violation after each insertion or deletion
     + Height property – The height difference between left & right subtree shouldn’t be greater than 1 or lesser than -1
   * If height property violates then fix the tree by tree rotation (same as RBT tree rotation algorithm)
   * Visualization 🡪 <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>
   * **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Access** | N/A | N/A |
| **Searching** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Deletion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Height** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |

* + **Big O** – **Space Complexity**:
    - Worst Case – **O (n)**

1. [**Splay Tree**](http://www.geeksforgeeks.org/splay-tree-set-1-insert/) – In [Splay Tree](http://digital.cs.usu.edu/~allan/DS/Notes/Ch22.pdf) recently/frequently searched item to be accessible in Constant Time O (1) time if accessed again.
   * Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.
   * All splay tree operations take O(log(n)) time on average. Splay trees can be rigorously shown to run in O (log (n)) average time per operation, over any sequence of operations (assuming we start from an empty tree)
   * Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node.
   * Unlike AVL tree, a splay tree can change even with read-only operations like search.
   * All the operations are similar to RB/AVL Tree except for the additional steps
   * **Tree Rotation/Splaying**
     + Zig Rotation (Right Rotation)
     + Zag Rotation (Left Rotation)
     + Zig-Zag Rotation (Right-Left Rotation)
     + Zag- Zig Rotation (Left- Right Rotation)
     + Zig- Zig Rotation (Right- Right Rotation)
     + Zag- Zag Rotation (Left- Left Rotation)
   * **Insert Operation –**
     + Insert operation is similar to RB tree with additional steps to make sure that the newly inserted key becomes the new root.
   * **Search Operation -** 
     + The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). Hence last accessed key is always at root.
     + If the search is successful, then the node that is found is splayed and becomes the new root.
     + Else the last node accessed prior to reaching the NULL is splayed and becomes the new root
   * Visualization 🡪 <https://www.cs.usfca.edu/~galles/visualization/SplayTree.html>
   * **Big O** – **Time Complexity**:

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Access** | N/A | N/A |
| **Searching** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Deletion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Height** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |

* + **Big O** – **Space Complexity**:
    - Worst Case – **O (n)**
  + **Application/Uses of Splay Tree**
    - To implement garbage collection & caches algorithms

[**Radix/Prefix/Trie Tree**](http://www.geeksforgeeks.org/splay-tree-set-1-insert/)

**Suffix Tree**

**Binomial Tree**

**B-Tree**

1. [Heap Data Structure:](http://www.geeksforgeeks.org/unrolled-linked-list-set-1-introduction/) Heap is a specialized "Tree" based data structure which satisfies the **"Heap" property** i.e. the value stored in the parent node is either greater than or equal to or less than or equal to the values stored in the children's node.

Heap data structure is classified in to,

* **Max-Heap** - Each child of a node has a value less than or equal to its parent node's value
* **Min-Heap** - Each child of a node has a value greater than or equal to its parent node's value

Heap data structure implementations,

* [**Binary Heap**](http://www.geeksforgeeks.org/binary-heap/) **(Max or Min) –** *Binary Heap*isa *heap*baseddata structure which is implemented as ***Complete Binary Tree.*** 
  + ***Priority Queue*** is implemented using *Binary Heap*
  + Visualization 🡪 <https://www.cs.usfca.edu/~galles/visualization/Heap.html>
  + Common operations
    - getMin()/getMax() – Returns the root element value
    - extractMin()/extractMax() – Extracts and returns the root element value. Heapify’s the tree for any violation
    - decreaseKey() -
  + **Big-O Complexities**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Access – getMin()/getMax()** | Constant Time **O (1)** | Constant Time **O (1)** |
| **Searching** | Linear Time **O (n)** | Linear Time **O (n)** |
| **Dequeuing – extractMin()/extractMax()** | Logarithmic Time **O (log(n))** | Logarithmic Time **O (log(n))** |
| **Deletion – remove (int data)** | Linear Time O (n) | Linear Time O (n) |
| **Size** | Constant Time **O (1)** | Constant Time **O (1)** |

* + *Binary Heap* can be [implemented](http://www.geeksforgeeks.org/binary-heap/) using an [Array](http://www.geeksforgeeks.org/array-representation-of-binary-heap/)
    - It’s easy and efficient to represent tree as an array
    - Parent node of ith node is 🡪 (i-1)/2
    - Left child node of ith node is 🡪 (i\*2) + 1
    - Right child node of ith node is 🡪 (i\*2) + 2
    - **extractMin() 🡪array[0]**
      * Remove array[0] and replace root value array[0] with array[size-1] leaf node value and remove leaf node array[size-1]
      * Heapify Down from 0th node
        + Fix the heap by comparing new root value with its child values by swapping
        + Repeat recursively until end of the list
    - **delete (int data)**
      * Find index of the element
      * Replace array[index] value with leaf node
      * Heapify Down from ith node
    - **decreaseKey(int data)**
  + **Application/Uses of Binary Heap Tree**
    - To implement Priority Queue
* [Binomial Heap](http://www.geeksforgeeks.org/binomial-heap-2/)
* [Fibonacci Heap](http://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/)

1. [Hash Table Data Structure:](http://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/)*Hash Table* is a *data structure* in which data is stored/retrieved in the form of *key-value* pairs in a specialized table called ***Hash Table*** using ***Hashing Technique*** *to* ***perform faster look up Constant Time O (1).***

The characteristics of *Hash Table* data structure is,

* Each *key* is unique and maps to its *value*
* Easy to *search* O(1)
* **Duplicate Key(s)** – No
* **Duplicate Value(s)** – Yes
* **NULL Key** – Yes (at most one, allows multiple but replaces NULL key’s value with new *value*)
* **NULL Value** – Yes (Multiple)
* **Order of Insertion** – Ordered or Unordered (based on implementation class)
* **Access** – By *Key*
* **Iteration/Traversal Order** – Ordered or Unordered
* **Iterator** – Fail-Fast
* **Big O** – **Time Complexity**:
  + ***Insertion*** –O(1)
  + ***Access*** – O(1)
  + ***Search*** **Key** (containsKey(K))– O(1) (In worst case scenario if hash code function is not implemented correctly then O(n))
  + ***Search*** **Value** (containsValue(V)) – O(n)
  + ***Delete Key*** – O(1)
* **Thread Safe** – Yes (Synchronized)

The generic implementation details of *Hash Table* data structure is,

* ***Hashing*** is a technique in which items are stored in a data structure (like array) such a way that it always provides constant time search operation - O(1).
* Hashing makes use of special function called ***"Hash Function"*** which generates the integer value (called 'Hash code/key/index') from the given input (called 'key' )which determines the position/location of the given item to be stored in a table called "Hash Table" (array).
* A specialized table is constructed using *static array* HashNode<K, V> *[]* / ArrayList<HashNode<K, V>> / LinkedList<HashNode<K, V>>
* Each entry in a table is referred as ***“Bucket”***
* Each element’s (HashNode<K, V>) entry position (i.e. *bucket position/index*) in a table is uniquely identified by using specialized function called **“Hash Function”**
* ***Hash Function*** computes the ***bucket index*** of the *element* by its *key’s* ***Hash Code*.** EveryHash Function has two parts,
  + ***Hash code***is an Integer number (random or nonrandom). In Java every Object has its own unique hash code
  + ***Compressor*** compresses the hash code to get *bucket index* by using modulo (%) operator and *size of the table* i.e. hash code % by size of the hash table. So here modulo operator (%) is a ***Compressor****.*
  + Good ***Hash Function*** is one that would try to minimize the number of ***collisions***
* **Table S*ize/Capacity*** represents the *total no. of* ***buckets***in the table
  + *Table size/initial table size* plays important role in determining the efficiency of the hash table.
  + [It’s recommended that table size should be always “Prime number](https://stackoverflow.com/questions/3980117/hash-table-why-size-should-be-prime)” to avoid clustering problems and for the even distribution.
  + How to calculate the correct table size / when to increase table size?
    - Recommended initial table size is 17
    - Always maintain ***Load Factor*** of ***0.75F*** to make sure that elements are distributed evenly in the table and offers a good tradeoff between time and space costs
    - [***Load Factor***](http://stackoverflow.com/questions/10901752/what-is-the-significance-of-load-factor-in-hashmap)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, then hash table is ***rehashed*** so that the hash table has approximately twice the number of buckets
    - ***Rehash/Rehashing*** is a technique to rebuild the table with larger table size
* ***Collision:*** If *hash function* returns the same bucket/hash table index for two or more different keys then it causes ***collision*** of bucket position*.*
* *In the event of collisions; collisions* can be handled/resolved using ***Collision Resolution Techniques*** using below formula **hi (x) = (h(x) + f(i))% HT size** and *Hash tables* which are implemented using *collision resolution techniques* are categorized as,
  + ***Chaining Hash Table***
    - Forms chain (linked list) from the point of collision and implemented using linkedlist
    - Disadvantage - Remaining free slots(if any) are not utilized instead creates new node (linked list node)
    - **f(i) = 0**
    - **LF = 0.75f**
  + ***Linked Chaining Hash Table***
    - **LF = 0.75f**
  + ***Double Hashing Hash Table***
    - Double hashing uses two hash functions, one to find the initial location *to place the key and a second to determine the size of the jumps in the probe sequence in case of collision*
    - *Second hash function h2(x) is used to calculate free spot after collision*
    - ***h2(x) = R - (x % R) where is 'R' is 'prime number' which is less than the HT sizes***
    - ***f(i) = i\*h2(x)***
    - **LF = 0.5f**
    - *Disadvantage - ??*
  + ***Linear Probing Hash Table***
    - Finds/probes free slot in a HT by moving 'one'spot from the point of collision hence it's called as "Linear"
    - **f(i) = i** where 'i' is no.of attempts
    - **LF = 0.5f**
    - Disadvantages - Forms "Primary Clustering" which reduces the performance (takes longer time to search for free spot)
    - How to fix Primary Clustering? by using "Quadratic Probing" technique
  + ***Quadratic Probing Hash Table***
    - Finds/probes for a free slot in a HT by moving **i\*i spot(s)** from the point of collision so that elements are distributed evenly (not clustered single place)
    - Probes more widely separated cells, instead of those adjacent to the primary hash site
    - HT size must be always Prime number and LF = 0.5f else probe sequences do not probe all locations in the table
    - ***f(i) = i\*i*** where 'i' is no.of attempts (1, 4, 9, 16, 25 so on)
    - **LF = 0.5f**
    - Disadvantages
      * Forms "Secondary Clustering"
      * Probe sequences do not probe all locations in the table
* ***Primary Clustering*** is caused by Linear Probing
  + Items are not distributed evenly & fairly in HT, at some point items are densely accumulated at some portion of HT (forming clusters)
* ***Secondary Clustering*** is *caused by Quadratic Probing*
  + *Items are placed/distributed far away from the point of collision x, x+1, x+4, x+9, x+16 so on this causes not probing of all free cells*
  + *Because elements that hash to the same hash key will always probe the same alternative cells*
* ***References***

<http://stackoverflow.com/questions/730620/how-does-a-hash-table-work>

<http://www.cs.cmu.edu/afs/cs/academic/class/15210-f13/www/lectures/lecture24.pdf>

<http://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/>

<http://stackoverflow.com/questions/40471/differences-between-hashmap-and-hashtable>

<http://faculty.cs.niu.edu/~freedman/340/340notes/340hash.htm>

<http://stackoverflow.com/questions/9124331/meaning-of-open-hashing-and-closed-hashing>

<http://stackoverflow.com/questions/3980117/hash-table-why-size-should-be-prime>

<https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/content/tables/quadratic_probing_and_double_hashing.html>

<http://stackoverflow.com/questions/10901752/what-is-the-significance-of-load-factor-in-hashmap>

<http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html>

<http://stackoverflow.com/questions/27742285/what-is-primary-and-secondary-clustering-in-hash>

<http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html>

<https://www.youtube.com/watch?v=nEcUS90C4fo>

<https://www.youtube.com/watch?v=CwM-Cxilk4g>

1. [Graph Data Structure:](https://www.youtube.com/watch?v=gXgEDyodOJU&index=38&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P)*Graph* is a collection of nodes/vertices where each node/vertex is connected other nodes/vertices in a graph. Graph consists of following two components,
   * A finite set of ***vertices / nodes***.
   * A finite set of ***ordered*** pair of the form (u, v) called as **edges**.
     + - The pair is called ***ordered*** because **(u, v)** is not same as **(v, u)** in case of directed graph(di-graph) i.e.
       - The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

**Types of graphs & graph terminologies**

* + **Directed Graph (Di-graph)** 🡪Edge direction is one side
  + **Undirected Graph** 🡪Edge direction is both the side
  + **Sparse Graph** – Each node/vertex is not connected to all other nodes/vertices
  + **Dense Graph** – Each node/vertex is connected to almost all other nodes/vertices
  + [Cycle/Circular Graph](https://en.wikipedia.org/wiki/Cycle_graph)– In a *Directed graph*some number of nodes are connected in a closed chain. 
    - * All *Undirected graphs* are Cycle graph
      * ***How to detect cycle in a directed graph?***
  + **Acyclic Graph** – No path to Start Node
  + **Normal Graph** *(simplest graph)*
  + **Multi Edge Graph** (complex graph)
  + **Weight 🡪** Each node/vertex or edge is associated with some value/property. Usually weight property is associated with edge. E.g. Distance between two cities
  + **Weighted Graph** – Usually a*directed* graph which has weight property associated with node or edge.
    - * **Node** – Binary Search Tree
      * **Edge** – Distance between two cities

[Representation of Graph](http://www.geeksforgeeks.org/graph-data-structure-and-algorithms/)

* + **Adjacency Matrix** 🡪Graph is represented in 2D matrix form
    - * **Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion – Node/Vertex** | O (|V| square of 2) | O (|V| square of 2) |
| **Insertion – Edge** | O (1) | O (1) |
| **Deletion – Node/Vertex** | O (|V| square of 2) | O (|V| square of 2) |
| **Deletion – Edge** | O (1) | O (1) |
| **Query – A is adjacent of B?** | O (1) | O (1) |

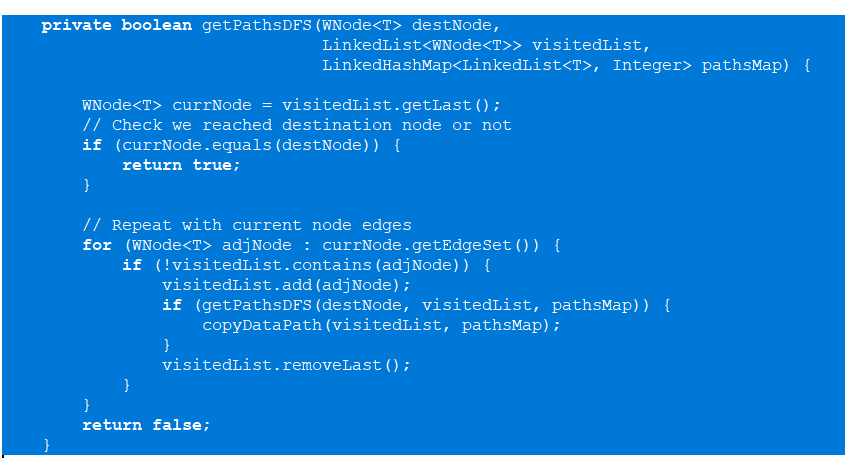
* + - * Here *V* is total no. of vertices in a given graph
      * **Space Complexity:** O (|V| square of 2)
  + **Adjacency List** 🡪Graph is represented in list (Linked list & array list) form
    - * **Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average Case** | **Worst Case** |
| **Insertion – Node/Vertex** | O (1) | O (1) |
| **Insertion – Edge** | O (1) | O (1) |
| **Deletion – Node/Vertex** | O (|E|) | O (|E|) |
| **Deletion – Edge** | O (|V|) | O (|V|) |
| **Query – A is adjacent of B?** | O (|V|) | O (|V|) |

* + - * Here *V* is total no. of vertices in a given graph
      * Here *V* is total no. of edges in a given graph
      * **Space Complexity:** O (|V| + |E|)

[Graph Traversal](https://www.youtube.com/watch?v=zaBhtODEL0w) **-** <https://en.wikipedia.org/wiki/Graph_traversal>

* + **Depth-First** **Traversal** 🡪All nodes in the left/right subtree is visited first then right/left subtree
    - * Always recursion
      * VisitedNodeList (LinkedList) – To avoid visiting again visited nodes
      * For loop – Recursion – Recursive for all the edges
      * **Applications** - Find all the possible paths



* + **Breadth-First** **Traversal**🡪At each level, all nodes are visited first before moving to the next level
    - * Always **Queue** (Linked List)–Initialize queue with start node
      * **Loop** queue until queue is non-empty using **while loop**
        + Poll queue - node
        + Check we already visited this node or not using visitedSet
        + If already visited then continue with next node in queue else
        + Add the current node in to visitedSet
        + Get the list of edges associated with the current node and perform the required operation
        + **hasPath** 🡪

Check edgeSet contains destNode, if not add all the edges to queue and repeat

* + - * + **Shortest Path 🡪**

Loop through all the edges using **for loop**

Add the edge node in to the queue

[Shortest Path Problem](https://en.wikipedia.org/wiki/Shortest_path_problem)

* + - The **single-source shortest path problem**, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
    - The **single-destination shortest path problem**, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
    - The **all-pairs shortest path problem**, in which we have to find shortest paths between every pair of vertices v, v' in the graph.
    - **Algorithms**
      * **General Algorithm**
        + Using **DFS** find all the possible paths – **O (|V| + |E|)** along with calculate weight for all the possible paths
        + Check path with less weight
        + Not efficient
      * [Dijkstra Algorithm](https://www.youtube.com/watch?v=pVfj6mxhdMw) – Solves the **single-source shortest path problem** 
        + Greedy algorithm, used only for non-negative weights.
        + Finds the shortest path from source node to all other nodes (not to just the destination node) in a given graph using **BFS or Priority Queue**
        + <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>
      * [Bellman–Ford Algorithm](https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm) - Solves the **single-source shortest path problem** evenif edge weights may be negative.
      * [Floyd–Warshall Algorithm](https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) - Solves **all -pairs shortest path problem**.

**Dijkstra Algorithm Using BFS**

* + - Find the shortest path from source node to all other nodes (not to just the destination node) in a given graph using **BFS**
    - [Time Complexity: O (|V| + |E|)](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)
    - From the below table we will get the shortest path from the source node to the destination node.

|  |  |  |
| --- | --- | --- |
| Node | Shortest distance from Node “A” | Previous Node |
| A | 0 |  |
| B |  |  |
| C |  |  |
| D |  |  |

* + - Implementation details
      * **Queue** – Construct the queue with all the nodes
      * **Distances [], Paths []**
      * Visit each node from the queue using while loop
        + currNode = q.poll()
        + edgeSet = currNode.getEdgeSet();
        + Using for loop calculate distance from the startNode to adjNode by using formula,

newDist = weight (currNode, adjNode) + distances[currNodeIndex]🡪 Gives distance from start node to current node



[Dijkstra Algorithm Using Binary Heap/Priority Queue](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/)

* + - Find the shortest path from source node to all other nodes (not to just the destination node) in a given graph using **Binary Heap/Priority Queue**
    - [Time Complexity: O (|E| Log |V|)](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

Dijkstra Algorithm Using Fibonacci Heap (Fastest running time)

* + - [Time Complexity: O (|E| + |V| Log |V|)](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)