**Contents**

[Time Complexities 2](#_Toc41151859)

[Arrays Methods 2](#_Toc41151860)

[Primitive Types 4](#_Toc41151861)

[Conversion Table 5](#_Toc41151862)

[Important in-built Methods 6](#_Toc41151863)

[Bit Manipulation 7](#_Toc41151864)

[Important Tips 8](#_Toc41151865)

[Sorting Algorithms 8](#_Toc41151866)

[Why Quick Sort is preferred over MergeSort for sorting Arrays 9](#_Toc41151867)

[Why MergeSort is preferred over QuickSort for Linked Lists? 9](#_Toc41151868)

[Java Iterator, ListIterator 9](#_Toc41151869)

[Iterator Methods 10](#_Toc41151870)

[Result 10](#_Toc41151871)

[Method 10](#_Toc41151872)

[Description 10](#_Toc41151873)

[**Important points:** 10](#_Toc41151874)

[Difference between Iterator and Enumeration interfaces 11](#_Toc41151875)

[**ListIterator methods** 11](#_Toc41151876)

[Result 11](#_Toc41151877)

[Method 11](#_Toc41151878)

[Description 11](#_Toc41151879)

[ConcurrentModificationException Basic 12](#_Toc41151880)

[Java Constructors 12](#_Toc41151881)

[Java Initializers 13](#_Toc41151882)

[Java Modifiers 13](#_Toc41151883)

[Java Features 14](#_Toc41151884)

[Encapsulation 14](#_Toc41151885)

[Inheritance 14](#_Toc41151886)

[**Polymorphism** 15](#_Toc41151887)

[Abstraction 15](#_Toc41151888)

[Overriding and Overloading 15](#_Toc41151889)

[Hashing 16](#_Toc41151890)

[**Hash table:** 16](#_Toc41151891)

[Hash Collision 16](#_Toc41151892)

[Hash Load Factors 16](#_Toc41151893)

# Time and Space Complexities

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Access by Index** | **Search** | **Add** | | | | | | **Deletion** | | |
|  |  |  | Before first Node | | After given Node | | | After Last Node | The First Node | A given Node | The last Node |
| **Array** | O (1) | O (n) | O(n) | | O(n) | | | O(1) | O(n) | O(n) | O(1) |
| **Singly LinkedList** | O(N) | O(n) | O(1) | | O(1) | | | O(1) | O(1) | O(n) | O(n) |
| **Doubly LinkedList** | O(N) | O(N) | O(1) | | O(1) | | | O(1) | O(1) | O(1) | O(1) |
| **Time and Space Complexity for Maps** | | | | | | | | | | | |
|  |  | Get | put | | containsKey | | | remove |  |  |  |
| **HashMap** | Time | O(1) | O(1) | | O(1) | | | O(1) |  |  |  |
|  | Space | O(n) | | | | | | |  |  |  |
| **TreeMap** | Time | O(logN) | | O(logN) | | O(logN) | O(logN) | |  |  |  |
|  | Space | O(n) | | | | | | |  |  |  |
| **Time and Space Complexity for Queue,Stacks,Binary Tree, Binary Search Tree** | | | | | | | | | | | |
|  | **Access** | **Search** | | **Insertion** | | **Deletion** | Contains | |  |  |  |
| **Stack** | O(n) | O(n) | | O(1) | | O(1) |  | |  |  |  |
| **Queues** | O(n) | O(n) | | O(1) | | O(1) |  | |  |  |  |
| **PriorityQueue** | O(1) |  | | O(logN) | | O(logN) | O(n) | |  |  |  |
| **Binary Tree** |  | O(n) | | O(n) | | O(n) |  | |  |  |  |
| **Binary search tree** |  | O(h) where h is height of BST | | O(h) | | O(h) |  | |  |  |  |
|  |  |  | |  | |  |  | |  |  |  |

* If you need to add or delete a node frequently, a linked list could be a good choice.
* If you need to access an element by index often, an array might be a better choice than a linked list.

# Arrays Methods

1. Print Array as String – Arrays.toString(arr);
2. Create an ArrayList from Array

Integer [] arr = new int[7];

List<Integer> lst = new ArrayList<>(Arrays.asList(arr));

1. Convert Arraylist to Array

List<Integer> lst = new ArrayList<>();

Integer[] arr = new Integer[lst.size()];

lst.toArray(arr);

1. Convert Array to a Set

String[] arr = new String[];

Set<String> s = new HashSet<>(Arrays.asList(arr));

1. Reverse elements of an Array

String[] arr = new String[];

ArrayUtils.reverse(arr);

1. Iterating through array

For(int item:nums){

System.out.println(item);

}

For(int i=0;i<nums.length;i++){

System.out.println(nums[i]);

}

1. Concatenate 2 arrays

Int[] arr1 = new int[7];

Int[] arr2= new int[8];

Int[] arr3 = ArrayUtils.addAll(arr1,arr2);

1. Java.util.Arrays methods

Arrays.sort(object[] a) 🡪 Sorts Array in ascending order

Arrays.fill(arr,-1) 🡪 Fills all elements of array with specific value

Arrays.binarySearch(int[] arr,int key) 🡪 returns index of the element If its present else returns –(insertion point +1)

Arrays.equals(int[] arr1,int[] arr2) 🡪 Returns true if both arr1 and arr2 have same elements and same length

1. Arrays Comparator , comparing 2D Arrays

**import** java.util.Arrays;

**import** java.util.Comparator;

Int[][] arr = new int[][]{};

Arrays.sort(arr,new Comparator<int[]>(){

public int compare(int[] a, int[] b){

return a[0] – b[0];

}

});

Return value :

<0 indicates accending order

0 indicates same value

>0 indicates descending value

1. Arrays.CopyOfRange(nums,0,K)

Copies the specified range of the specified array into a new array. The initial index of the range (from) must lie between zero and original.length, inclusive.

# Primitive Types

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Description** |
| int | 4 bytes | Stores value between -2^32 to +2^31-1  **Default value 0** |
| char | 2 bytes | Stores single character ASCII Character/Letter  0 to 65535(Unsigned)  A – Z is 65 to 90  a – z is 97 to 122  **Default value ‘\u0000’ equivalent to null** |
| byte | 1 byte | -128 to 127  **Default value 0** |
| short | 2 bytes | -2^16 to +2^16-1(-32768 to + 32767)  **Default value 0** |
| long | 8 bytes | Stores whole numbers -2^64 to +2^64-1  **Default value 0L** |
| float | 4 bytes | Stores fractional numbers . Decimals sufficient for storing 6 to 7 decimal digits  **Default value 0.0f** |
| double | 8 bytes | Stores fractional numbers. Decimals sufficient for storing upto 15 digits  **Default value 0.0d** |
| boolean | 1 byte | Stores true or false values  **Default Value false** |

Declaring primate types

char ch =’a’;

int n = 1;

long lg = 1500000000L;

float myNum = 1.234f

double myNum = 19.999d

# Conversion Table

|  |  |  |
| --- | --- | --- |
| **From** | **To** |  |
| Integer | String | Integer.valueOf(str) |
| Integer | Character | Character.forDigit(n);  //Different behavior for int to char  Int n = 1;char ch = (char)(n+’0’); //prints 1  Int n = 1;char ch = (char)n; //it will store ASCII character of given number which is start of heading which is not printable  Int n =’1’;char ch = (char)n; //prints 1 |
| Character | Integer | Character.getNumericValue(ch)  Char ch = ‘6’;  int n = ch – ‘0’; |
| Character | String | String.valueOf(ch);  Character.toString(ch); |
| String | Integer | Integer.valueOf(str)  Integer.parseInt(str); |
| String | Character | s.charAt(i);  str.toCharArray() |
| Display Integer in Binary Format |  | Integer.toBinaryString(n) |

# Important in-built Methods

|  |  |
| --- | --- |
| **Function** | **Description** |
| Math.sqrt(dbl) | Sqrt(x)   * public static double sqrt(double a)   Returns the correctly rounded positive square root of a double value. Special cases:   * + If the argument is NaN or less than zero, then the result is NaN.   + If the argument is positive infinity, then the result is positive infinity.   + If the argument is positive zero or negative zero, then the result is the same as the argument.   Otherwise, the result is the double value closest to the true mathematical square root of the argument value. |
| Math.pow(x,y) | X^y   * public static double pow(double a,double b)   Returns the value of the first argument raised to the power of the second argument. Special cases:   * + If the second argument is positive or negative zero, then the result is 1.0.   + If the second argument is 1.0, then the result is the same as the first argument.   + If the second argument is NaN, then the result is NaN. |
| Math.floor(dbl) | * public static double floor(double a)   Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer. |
| Math.ceil(dbl) | * public static double ceil(double a)   Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer. |
| Random | Import java.util.Random  Random rand = new Random();  rand.nextInt() 🡪 returns a number between 0 and 2^32 inclusive  rand.nextInt(16) 🡪 returns a number between 0(included) and 16 (excluded) |
| Math.sin(x) | sinxsin⁡x |
| Math.cos(x) | cosxcos⁡x |
| Math.tan(x) | tanxtan⁡x |
| Math.asin(x) | sin−1x |
| Math.acos(x) | cos−1x |
| Math.atan(x) | tan−1x |
| Math.exp(x) | ex |
| Math.log(x) | lnx |
| Math.log10(x) | log10x |
| Math.round(float x) | the closest integer to x, as the closest integer to x, as a long |
| Math.abs(x) | |x| |
| Math.max(x,y) | The maximum of the two values |
| Math.min(x,y) | The minimum of the two values |
| Math.PI | Double  3.14159265358979323846 |
| Math.E | Double  2.7182818284590452354 |

# Bit Manipulation

|  |  |
| --- | --- |
| **Operation to Check** | **How to Check** |
| Check whether lowest significant bit is 1 or 0 | X & 1 |
| Reset Lowest bit set to 0 | X & (X -1) // drops lowest set bit of x  Example – 7 & 6 makes  111 & 110 🡪 110 which causes lowest bit to 0 |
| Retain Only Lowest bit and reset all other bits | X & ~(X -1) |
|  |  |

# Important Tips

* + 1. 2D matrix with Rows and columns can be remembered with “code”

Example – r = 2 rows , c – 3 columns

Code = i \* c + j 🡪 where i is a row being processed and j is a column being processed

Fetching back row and column

i = code /c

j = code % c

Refer Leetcode question – 200 – Maximum number of Islands

# Sorting Algorithms

| **Algorithm** | **Time Complexity** | | |
| --- | --- | --- | --- |
|  | **Best** | **Average** | **Worst** |
| Selection Sort | Ω(n^2) | θ(n^2) | O(n^2) |
| Bubble Sort | Ω(n) | θ(n^2) | O(n^2) |
| Insertion Sort | Ω(n) | θ(n^2) | O(n^2) |
| Heap Sort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| Quick Sort | Ω(n log(n)) | θ(n log(n)) | O(n^2) |
| Merge Sort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| Bucket Sort | Ω(n+k) | θ(n+k) | O(n^2) |
| Radix Sort | Ω(nk) | θ(nk) | O(nk) |

## Why Quick Sort is preferred over MergeSort for sorting Arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

## Why MergeSort is preferred over QuickSort for Linked Lists?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

# Java Iterator, ListIterator

Introduced in the Java JDK 1.2 release, the java.util.Iterator interface allows the iteration of Collection classes. Each Iterator provides a next() and hasNext() method, and may optionally support a remove() method. Iterators are created by the corresponding container class, typically by a method named iterator(). It is an improvement over Enumeration interface. The order in which the elements contained in a Java Iterator are traversed depends on the object that supplies the Iterator. For instance, an iterator obtained from a List will iterate through the elements of that List in the same order the elements are stored internally in the List. An Iterator obtained from a Set, on the other hand, does not make any guarantees about the exact sequence the elements in the Set are iterated in. The Iterator interface allows us to modify a collection while traversing it, which is more difficult with a simple for/while statement. This, in turn, gives us a good pattern we can use in many methods that only requires collections processing while maintaining good cohesion and low coupling.

## Iterator Methods

Iterator defines three methods, one of which is [optional](https://crunchify.com/missing-maven-settings-xml-file-for-your-eclipse-what-if-you-need-two-settings-xml-file-for-work-personal-workspace/).

|  |  |  |
| --- | --- | --- |
| Result | Method | Description |
| b = | it.hasNext() | true if there are more elements for the iterator. |
| obj = | it.next() | Returns the next object. If a [generic](https://crunchify.com/java-8-temporaladjusters-and-stream-flatmap-tutorial/) list is being accessed, the iterator will return something of the list’s type. Pre-generic Java iterators always returned type [Object](https://crunchify.com/in-java-how-to-flatten-or-unflatten-complex-json-objects-into-flat-map-like-structure/), so a downcast was usually required. |
|  | it.remove() | Removes the most recent element that was returned by next. Not all collections support delete. UnsupportedOperationException - if the remove operation is not supported by this iterator. IllegalStateException - if the next method has not yet been called, or the remove method has already been called after the last call to the next method |

### **Important points:**

* Iterators are unidirectional. We can iterate only in one direction.
* Iteration can be done only once. If you reach the end of series its done. If we need to iterate again we should get a new Iterator.

### **Difference between Iterator and Enumeration interfaces**

1. remove() method is introduced in iterator. Using this method we can remove element from the underlying [collection](https://crunchify.com/how-to-convert-hashmap-to-arraylist-in-java/) which we are iterating.
2. Enumeration has two methods and both are available in iterator. Method names for both of them are shortened.

## **ListIterator methods**

ListIterator is implemented only by the classes that implement the List interface ([ArrayList](https://crunchify.com/java-how-to-find-unique-values-in-arraylist-using-treeset-hashset/), LinkedList, and Vector). ListIterator provides the following.

|  |  |  |
| --- | --- | --- |
| Result | Method | Description |
| Forward iteration | | |
| b = | it.hasNext() | true if there is a next element in the collection. |
| obj = | it.next() | Returns the next object. |
| Backward iteration | | |
| b = | it.hasPrevious() | true if there is a previous element. |
| obj = | it.previous() | Returns the previous element. |
| Getting the index of an element | | |
| i = | it.nextIndex() | Returns index of element that would be returned by subsequent call to next(). |
| i = | it.previousIndex() | Returns index of element that would be returned by subsequent call to previous(). |
| Optional modification methods. UnsupportedOperationException thrown if unsupported. | | |
|  | it.add(obj) | Inserts obj in [collection](https://crunchify.com/what-is-java-collections-framework-benefits-of-collections-framework/) before the next element to be returned by next() and after an element that would be returned by previous(). |
|  | it.set() | Replaces the most recent element that was returned by next or previous(). |
|  | it.remove() | Removes the most recent element that was returned by next() or previous(). |

### ConcurrentModificationException Basic

**We cannot add or remove elements to the underlying collection when we are using an iterator.**

# Java Constructors

* Constructors are invoked while instantiating a new Object of a class using new ClassName(). It can be used to set initial values for object attributes. A Constructor must have no explicit return type. Constructor name must be the same as its class name. A Java constructor cannot be abstract, static, final, and synchronized.
* **Abstract classes** have constructors, and those constructors are always called when a concrete subclass is instantiated.
* **Default Constructor** 🡪 It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
* **Types of Constructors 🡪** Default constructors and Parameterised Constructors. A constructor is called "Default Constructor" when it doesn't have any parameter. A constructor which has a specific number of parameters is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.
* **Constructor Overloading** 🡪 In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.
* **Calling a Constructor From a Constructor -** In Java it is possible to call a constructor from inside another constructor. When you call a constructor from inside another constructor, you use the **this** keyword to refer to the constructor.
* **Calling super() vs this() –** the very first statement in a constructor has to be either a super() or this(). Every constructor invokes the constructor of its superclass with an (implicit) call to super(), unless the constructor invokes an overloaded constructor of the same class (more on that in a minute). **Only static variables and methods can be accessed as part of the call to super() or this().**
* **Java Constructor Access** **Modifiers -** The access modifier of a constructor determines what classes in your application that are allowed to call that constructor. For instance, if a constructor is declared protected then only classes in the same package, or subclasses of that class can call that constructor. A class can have multiple constructors, and each constructor can have its own access modifier.
* **Throwing Exceptions From a Constructor** - It is possible to throw an exception from a Java constructor. In case an exception is thrown from the Class constructor, the Object variable will never be assigned a reference to the class object you are trying to create. Making a constructor throw an exception can be a good idea if you want to prevent an object of the given class to be created in an invalid state. Typically it is the input parameters to the constructor that may cause the object to be invalid. If the constructor detects an invalid input parameter, it can throw an exception and prevent the assignment of the object to any variable.

# Java Initializers

* **Java instance initializer blocks** - Java instance initializers are code blocks which are executed before the constructor code is executed. These initializers run everytime we create a new object.
* **Java Static initializer blocks** - A *static initialization block* is a normal block of code enclosed in braces, { }, and preceded by the static keyword.
* **Java Initializers features**

1. We can define multiple initializers in a class.
2. All initializers execute in sequence in order they appear in class body.
3. Initializers run after the parent class constructor has been invoked and before executing child class constructor. Please note that Java inserts the default constructor of parent class super(), if we do not explicitly provide the constructor as the first statement in child class’s constructor.
4. After all the initializers have executed, constructor’s statements are executed.
5. We can use call the constructors of this class and parent classes inside initializers.

* **Java instance initialization sequence flow**

1. Child class constructor is invoked.
2. Child class constructor has first statement as super() (or provided explicit constructor) so parent class constructor is invoked.
3. Parent class’s initializers are executed in sequence of their appearance.
4. Parent class constructor statements are executed.
5. Child class’s initializers are executed in sequence of their appearance.
6. Child class constructor statements are executed.

# Java Modifiers

**class Modifiers**

|  |  |
| --- | --- |
| **Access Modifiers** | |
| public | The class is accessible by any other class |
| default | The class is only accessible by classes in the same package. This is used when you don't specify a modifier. |
| **Non-Access Modifiers** | |
| final | The class cannot be inherited by other classes(or another class cant extend it) |
| abstract | The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. |

**Attributes, methods and constructors**

|  |  |
| --- | --- |
| **Access Modifiers** | |
| public | The code is accessible for all classes |
| private | The code is only accessible within the declared class |
| default | The code is only accessible in the same package. This is used when you don't specify a modifier. |
| protected | The code is accessible in the same package and **subclasses**. |
| **Non-Access Modifiers** | |
| final | Attributes and methods cannot be overridden/modified |
| static | Attributes and methods belongs to the class, rather than an object |
| abstract | Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). |
| transient | Attributes and methods are skipped when serializing the object containing them |
| synchronized | Methods can only be accessed by one thread at a time |
| volatile | The value of an attribute is not cached thread-locally, and is always read from the "main memory" |

# Java Features

## Encapsulation

For maintainability, flexibility, and extensibility your design must include encapsulation. The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. Keep instance variables protected (with an access modifier, often private). Make public accessor methods, and force calling code to use those methods rather than directly accessing the instance variable. For the methods, use the JavaBeans naming convention of set<someProperty> and get<someProperty>.

## Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system). The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also. Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

**subclass** (child) - the class that inherits from another class

**superclass** (parent) - the class being inherited from

## **Polymorphism**

Java Polymorphism - Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways. For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.)

There are two types of polymorphism in Java: **compile-time polymorphism and runtime polymorphism**. We can perform polymorphism in java by method overloading and method overriding. If you overload a static method in Java, it is the example of compile time polymorphism. Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

## Abstraction

**Java Abstract Classes and Methods**

Data abstraction is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either abstract classes or interfaces

The abstract keyword is a non-access modifier, used for classes and methods:

**Abstract class**: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class). An abstract class can have both abstract and regular methods

**Abstract method**: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

## Overriding and Overloading

**Overriding** - In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

**Method overriding is** one of the way by which java achieve [Run Time Polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/).The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

# Hashing

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. The values are used to index a fixed-size table called a hash table. Hash functions are used in conjunction with Hash table to store and retrieve data items or data records.

## **Hash table:**

Data structure of computer science that returns search result in constant time or O(1). It is based on the concept of hashing. So, it has better access time than linkedlist, binary search trees etc. Why nearly O(1): It uses an array as its base structure internally to store the objects and as arrays have constant access time hence, the Hash table does too.

[Basic internal]: So, it uses an array of fixed size internally and when you insert an (Key, Value) pair, it calculates key's hash and uses this hash value as index to store the (Key,Value) pair in the array. Next, when you search for the object using the same key, it uses the hash of the key again as index to search for the key in the array

# Hash Collision

When an item is to be added to the table, the hash code may index an empty slot (also called a bucket), in which case the item is added to the table there. If the hash code indexes a full slot, some kind of collision resolution is required: the new item may be omitted (not added to the table), or replace the old item, or it can be added to the table in some other location by a specified procedure. That procedure depends on the structure of the hash table: In chained hashing, each slot is the head of a linked list or chain, and items that collide at the slot are added to the chain. Chains may be kept in random order and searched linearly, or in serial order, or as a self-ordering list by frequency to speed up access.

# Hash Load Factors

An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The 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. The load factor 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.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.