**Most Popular Data Structures:**

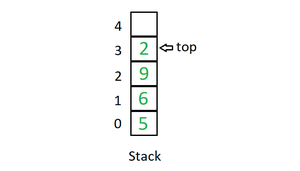
**1.** [**Array**](https://www.geeksforgeeks.org/array-data-structure/)**:** An array is a collection of data items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



**2.** [**Linked Lists**](https://www.geeksforgeeks.org/data-structures/linked-list/)**:** Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



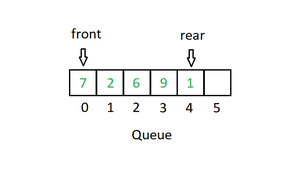
**3.** [**Stack**](https://www.geeksforgeeks.org/stack-data-structure/)**:** Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). In stack, all insertion and deletion are permitted at only one end of the list.



**Mainly the following three basic operations are performed in the stack:**

* **Initialize**: Make a stack empty.
* **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
* **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
* **Peek or Top:** Returns top element of the stack.
* **isEmpty:** Returns true if the stack is empty, else false.

**4.** [**Queue**](https://www.geeksforgeeks.org/queue-data-structure/)**:** Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). In the queue, items are inserted at one end and deleted from the other end. A good example of the queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



**Mainly the following four basic operations are performed on queue:**

* **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
* **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
* **Front:** Get the front item from the queue.
* **Rear:** Get the last item from the queue.

**5.** [**Binary Tree**](https://www.geeksforgeeks.org/binary-tree-data-structure/)**:** Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures. A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

A Binary Tree is represented by a pointer to the topmost node in the tree. If the tree is empty, then the value of root is NULL. A Binary Tree node contains the following parts.

1. Data

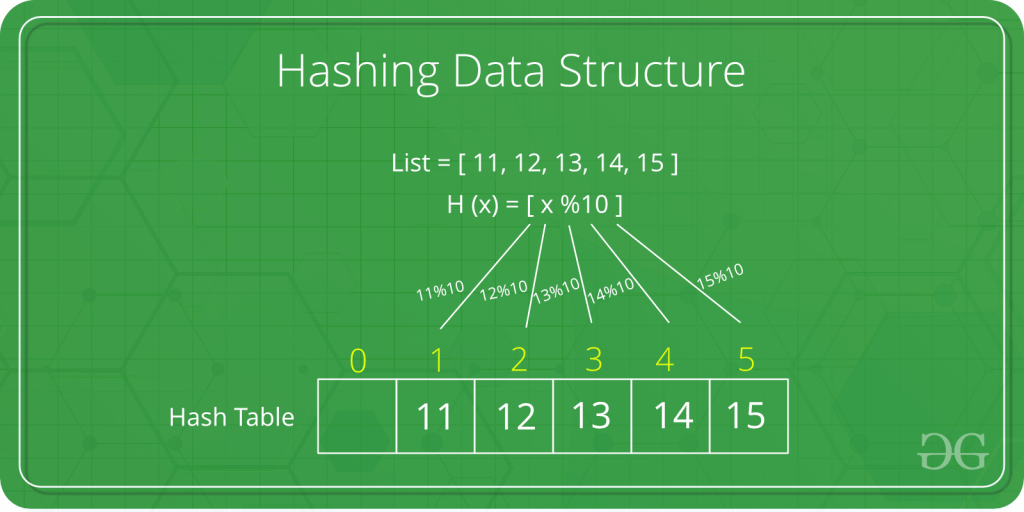
2. Pointer to left child

3. Pointer to the right child

<https://www.geeksforgeeks.org/hashing-data-structure/?ref=lbp>

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function H(x) maps the value at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



[Recent Articles on Hashing](https://www.geeksforgeeks.org/category/Hash/) **Why to use Hashing?**

If you observe carefully, in a balanced binary search tree, if we try to search , insert or delete any element then the time complexity for the same is O(logn). Now there might be a situation when our applications want to do the same operations in a faster way i.e. in a more optimized way and here hashing comes into play. In hashing, all the above operations can be performed in O(1) i.e. constant time. It is important to understand that the worst case time complexity for hashing remains O(n) but the average case time complexity is O(1).  
Now let us understand a few basic operations of hashing.

HashSet is a **set**, e.g. **{1,2,3,4,5}**

HashMap is a **key -> value** (key to value) map, e.g. **{a -> 1, b -> 2, c -> 2, d -> 1}**

Notice in my example above that in the HashMap there must not be duplicate keys, but it may have duplicate values.

In the HashSet, there must be no duplicate elements

## ****What is Collision?****

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

**How to handle Collisions?**

There are mainly two methods to handle collision:

* Separate Chaining
* Open Addressing

**Basic Operations:**

* **HashTable:** This operation is used in order to create a new hash table.
* **Delete:** This operation is used in order to delete a particular key-value pair from the hash table.
* **Get:** This operation is used in order to search a key inside the hash table and return the value that is associated with that key.
* **Put:** This operation is used in order to insert a new key-value pair inside the hash table.
* **DeleteHashTable:** This operation is used in order to delete the hash table

**Hashing Components:**

**1)** [**Hash Table**](https://en.wikipedia.org/wiki/Hash_table)**:** An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.  In simple terms, we can say that hash table is a generalization of array. Hash table gives the functionality in which a collection of data is stored in such a way that it is easy to find those items later if required. This makes searching of an element very efficient.

**2)** [**Hash Function**](https://en.wikipedia.org/wiki/Hash_function)**:** A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. So, in simple terms we can say that a hash function is used to transform a given key into a specific slot index. Its main job is to map each and every possible key into a unique slot index. If every key is mapped into a unique slot index, then the hash function is known as a perfect hash function. It is very difficult to create a perfect hash function but our job as a programmer is to create such a hash function with the help of which the number of collisions are as few as possible. Collision is discussed ahead.

A **good hash function** should have following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each).
3. Should minimize collisions.
4. Should have a low load factor(number of items in table divided by size of the table).

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

**3) Collision Handling**: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

* **Chaining:**The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
* **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

# Difference between HashMap and HashSet

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 20 Apr, 2022

 Read

 Discuss

[HashSet](https://www.geeksforgeeks.org/hashset-in-java/) is an implementation of [Set Interface](https://www.geeksforgeeks.org/set-in-java/) which does not allow duplicate value. The main thing is, objects that are stored in HashSet must override equals() for check for equality, and hashCode() methods for no duplicate value are stored in our set. [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) is an implementation of [Map Interface](https://www.geeksforgeeks.org/map-interface-java-examples/), which maps a key to value. Duplicate keys are not allowed in a Map. Basically, Map Interface has two implementation classes HashMap and [TreeMap](https://www.geeksforgeeks.org/treemap-in-java/) the main difference is TreeMap maintains an order of the objects but HashMap will not.HashMap allows null values and null keys. Both HashSet and HashMap are not synchronized.

Now let us formulate the difference between HashMap and HashSet as provided in a tabular manner below as follows:

| **Basic** | **HashSet** | **HashMap** |
| --- | --- | --- |
| Implements | Set interface | Map interface |
| Duplicates | No | Yes duplicates values are allowed but no duplicate key is allowed |
| Dummy values | Yes | No |
| Objects required during an add operation | 1 | 2 |
| Adding and storing mechanism | HashMap object | Hashing technique |
| Speed | It is comparatively slower than HashMap | It is comparatively faster than HashSet because of hashing technique has been used here. |
| Null | Have a single null value | Single null key and any number of null values |
| Insertion Method | Add() | Put() |

Let us grasp understanding by peeking into internal working with help of clean java programs.

**Example 1:** HashSet

|  |
| --- |
| // Java program to demonstrate working of HashSet    // Importing HashSet class from java.util package  import java.util.HashSet;    // Mai class  public class GFG {        // Main driver method      public static void main(String[] args)      {            HashSet<String> hs = new HashSet<String>();          // Adding elements to the HashSet          hs.add("geeks");          hs.add("practice");          hs.add("contribute");          ;            System.out.println(              "Before adding duplicate values \n\n" + hs);            // Addition of duplicate elements          hs.add("geeks");          hs.add("practice");            System.out.println(              "\nAfter adding duplicate values \n\n" + hs);            // Addition of null values          hs.add(null);          hs.add(null);            // Displaying HashSet elements          System.out.println("\nAfter adding null values \n\n"                             + hs);      }  } |

**Output**

Before adding duplicate values

[practice, geeks, contribute]

After adding duplicate values

[practice, geeks, contribute]

After adding null values

[null, practice, geeks, contribute]

**Example 2:** HashMap 

|  |
| --- |
| import java.util.HashMap;    public class HashMapExample {        public static void main(String[] args)      {            // This is how to declare HashMap          HashMap<Integer, String> hm = new HashMap<Integer, String>();            // Adding elements to HashMap\*/          hm.put(12, "geeks");          hm.put(2, "practice");          hm.put(7, "contribute");            System.out.println("\nHashMap object output :\n\n" + hm);            // store data with duplicate key          hm.put(7, "geeks");          hm.put(12, "contribute");            System.out.println("\nAfter inserting duplicate key :\n\n" + hm);      }  } |

**Output:**

HashMap object output :

{2=practice, 7=contribute, 12=geeks}

After inserting duplicate key :

{2=practice, 7=geeks, 12=contribute}

From the above two outputs after going through an understanding of their internal working, now we can talk about conceptual differences which are as follows:

1. **Implementation:** HashMap implements Map interface and HashSet implements Set interface.
2. **Duplicates:** HashSet doesn’t allow duplicate values. HashMap stores key, value pairs and it does not allow duplicate keys. If the key is duplicate then the old key is replaced with the new value.
3. **Number of objects during storing objects:** HashMap requires two objects put(K key, V Value) to add an element to HashMap object, while HashSet requires only one object add(Object o)
4. **Dummy value:** In HashMap no concept of dummy value,   
   HashSet internally uses HashMap to add elements. In HashSet, the argument passed in **add(Object)** method serves as key K. Java internally associates dummy value for each value passed in add(Object) method.
5. **Storing or Adding mechanism:** HashMap internally uses hashing to store or add objects, HashSet internally uses HashMap object to store or add the objects.
6. **Speed:** HashSet is slower than HashMap.
7. **Insertion** HashMap uses the put() method for storing data, While in HashSet use add() method for add or storing data.

Let us wrap up with an example

HashSet is a set, e.g. {1, 2, 3, 4, 5, 6, 7},

HashMap is a key -> value pair(key to value) map, e.g. {a -> 1, b -> 2, c -> 2, d -> 1}

Here, in the example of the HashMap there must not be duplicate keys, but it may have duplicate values. In the HashSet, there must be no duplicate elements