**Comparable Interface in TreeSet Example**

import java.util.\*;  
  
class TreeSetCustomSortingDemo  
{  
    public static void main(String arg[])  
    {  
        TreeSet<Student> students = new TreeSet<Student>(); // LINE B  
        students.add(new Student("Sreeram", 10, 'A'));  
        students.add(new Student("Karthik", 12, 'A'));  
        students.add(new Student("Ram", 14, 'B'));  
        students.add(new Student("Yeshwanthi", 5, 'C'));  
        students.add(new Student("Bhavya", 6, 'D'));  
        System.out.println(students); // LINE C    }  
}  
  
class Student implements Comparable<Student>  
{  
    String name;  
    int rollNumber;  
    char section;  
      
    public Student(String name, int rollNumber, char section)  
    {  
        this.name = name;  
        this.rollNumber = rollNumber;  
        this.section = section;  
    }  
      
    public String toString()  
    {  
        return name + " " + rollNumber + " " + section;  
    }  
      
    public int compareTo(Student o)  
    {  
        return rollNumber - o.rollNumber;  
    }  
}

In the above program we have demonstrated how to change the natural ordering of the TreeSet depending on the requirements. Firstly we have created a Student class which implements Comparable interface with the fields name, rollNumber, section and implemented toString and compareTo method of Comparable interface. At LINE B in the other class we have created a TreeSet and added Student objects. Now Set will consider the compareTo method in the Student class and sorts the objects accordingly.

**compareTo()**

Comparable interface has exactly one method compareTo() which accepts a single object as parameter and returns an integer value. Java determines the sorting order based on the interger value returned from the compareTo() method.

Syntax

int compareTo(T o1);

Consider an object x is compared with y via the statement x.compareTo(y), then the value return from the method should be

0, if values in x and y are equal.

Negative, if the value in x is less than in y.

Positive, if the value in x is greater than in y.

**Sort using section**

**Comparator:**

import java.util.\*;

class Student {

   String name;

   int age, roll;

   Student(String name, int age, int roll) {

      this.name = name;

      this.age = age;

      this.roll = roll;

   }

}

class AgeComparator implements Comparator {

   @Override

   public int compare(Object o1, Object o2) {

      return ((Student) o1).age - ((Student) o2).age;

   }

}

class RollComparator implements Comparator {

   @Override

   public int compare(Object o1, Object o2) {

      return ((Student) o1).roll - ((Student) o2).roll;

   }

}

public class ComparatorDemo {

   public static void main(String[] args) {

      List list = new ArrayList<>();

      list.add(new Student("Ramesh", 30, 20));

      list.add(new Student("Adithya", 7, 10));

      list.add(new Student("Krishna", 25, 5));

      list.add(new Student("Vineet", 24, 15));

      System.out.println("BEFORE SORTING");

      for (Student e : list) {

         System.out.println("[ STU : name = " + e.name + " age = " + e.age + " roll = " + e.roll + "]");

      }

      Collections.sort(list,new AgeComparator());

      System.out.println("AFTER SORTING WITH AGE");

      for (Student e : list) {

         System.out.println("[ STU : name = " + e.name + " age = " + e.age + " ]");

      }

      Collections.sort(list,new RollComparator());

      System.out.println("AFTER SORTING WITH ROLL");

      for (Student e : list) {

         System.out.println("[ STU : name = " + e.name + " roll = " + e.roll + " ]");

      }

   }

}

| **Comparable** | **Comparator** |
| --- | --- |
| Comparable provides compareTo() method to sort elements in Java. | Comparator provides compare() method to sort elements in Java. |
| Comparable interface is present in java.lang package. | Comparator interface is present in java.util package. |
| The logic of sorting must be in the same class whose object you are going to sort. | The logic of sorting should be in separate class to write different sorting based on different attributes of objects. |
| The class whose objects you want to sort must implement comparable interface. | Class, whose objects you want to sort, do not need to implement a comparator interface. |
| It provides single sorting sequences. | It provides multiple sorting sequences. |

**Iterator**

The Iterator helps you to move through all the elements in a collection.

The methods used by Iterator are:

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean hasNext() | Returns true if the collection has next element, else it returns false. |
| E Next() | Returns the next element. If there is no next element, it throws anexception. |
| void remove() | Removes the current element to which the iterator is pointing. |

import java.util.\*;  
  
class IteratorDemo  
{  
    public static void main(String arg[])  
    {  
        ArrayList<Integer> numbers = new ArrayList<Integer>();  
        numbers.add(10);  
        numbers.add(20);  
        numbers.add(30);  
        numbers.add(40);  
        numbers.add(50);  
        Iterator<Integer> itr = numbers.iterator();  
        while (itr.hasNext())   
        {  
            int number = itr.next();  
            System.out.print(number + " ");  
            if (number == 30)  
                itr.remove();  
        }  
        System.out.println("\n..................");  
        itr = numbers.iterator();//LINE A  
        while (itr.hasNext())   
        {  
            int number = itr.next();  
            System.out.print(number + " ");  
        }          
    }  
}

**ListIterator**

ListIterator is used to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

ListIterator is available to only those collections that implement the **Java List Interface** . A listIterator has no current element, it's cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().  
  
Advantages of using listIterator over iterator:

* With iterator you can move only forward, but with listIterator you can also move reverse while reading the elements.
* With listIterator you can obtain the index at any point while traversing, which is not possible with iterator.
* With iterator you can only check whether the next element is available or not, but with listIterator you can check the previous and next elements.
* With listIterator you can add or modify an element at any point while traversing, which is not possible with iterator.
* import java.util.\*;  
    
  class ListIteratorDemo  
  {  
      public static void main(String arg[])  
      {  
          ArrayList ar = new ArrayList();  
          ar.add("Black");  
          ar.add("Red");  
          ar.add("Blue");  
          ListIterator litr = ar.listIterator();  
          while (litr.hasNext()) // In forward direction  
          {  
              System.out.print(litr.next() + " ");  
          }  
          System.out.println();  
          while (litr.hasPrevious()) // In reverse direction  
          {  
              System.out.print(litr.previous() + " ");  
          }  
          System.out.println();  
          litr = ar.listIterator(2); // LINE A - Set iterator at specified index  
          System.out.println(litr.previousIndex() + " " + litr.nextIndex()); // Indices  
          litr.add("Orange"); // LINE B  
          System.out.println("After adding Orange : " + litr.previous());  
          litr.remove(); // LINE C  
          System.out.println("After removing : " + litr.previous());  
          litr.set("Yellow"); // LINE D  
          System.out.println("After setting : " + litr.next());      
      }  
  }

**Java Map**

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

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

**Java Map Hierarchy**

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:

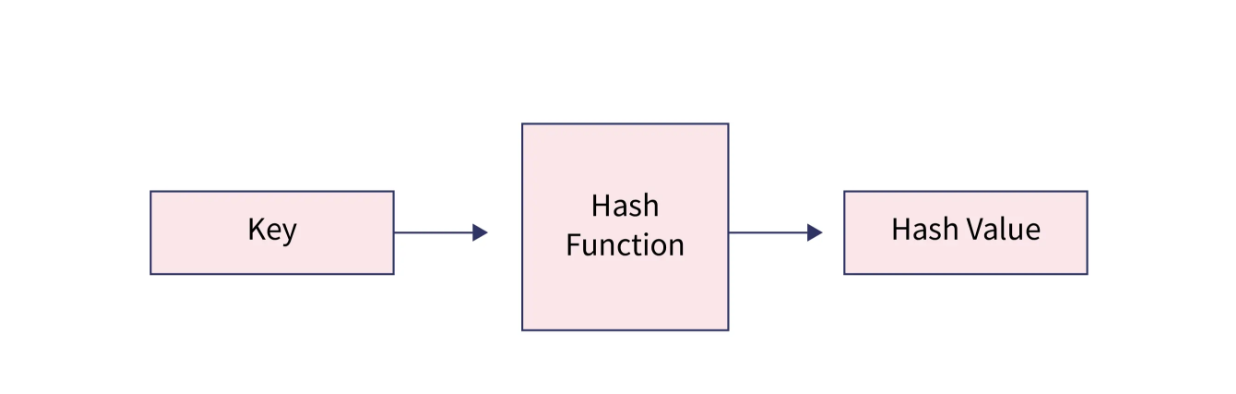
Java Map Hierarchy

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value

**Hashing**

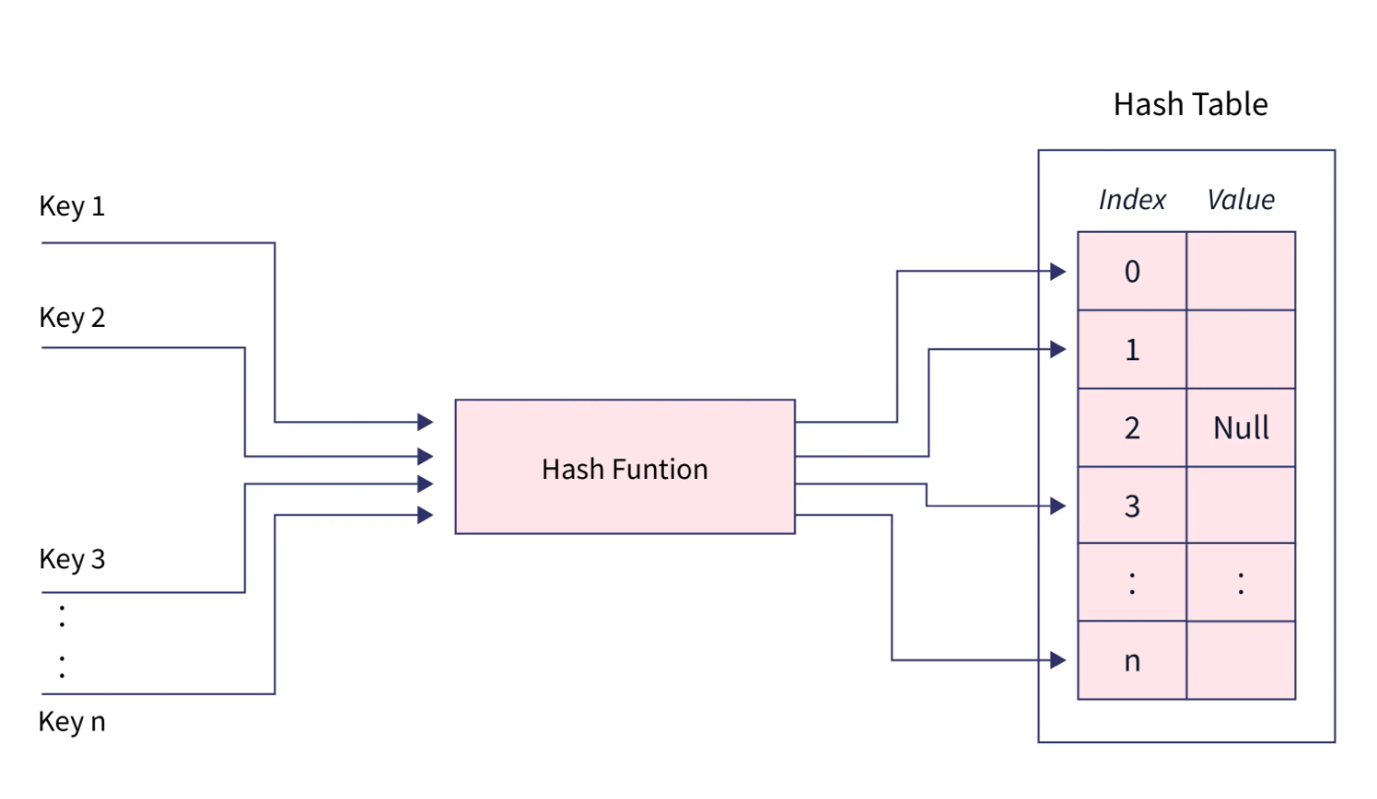
Hashing in Java is the technique that enables us to store the data in the form of key-value pairs, by modifying the original key using the hash function so that we can use these modified keys as the index of an array and store the associated data at that index location in the Hash table for each key.

**Hash Function**

While implementing hashing in java it uses a function called hash function, it is the most important part of hashing; it transforms supplied keys into another fixed-size value (hash-value). The value returned by a hash function is called hash value, hash code, or simply hashes.  
  


**Hash table**

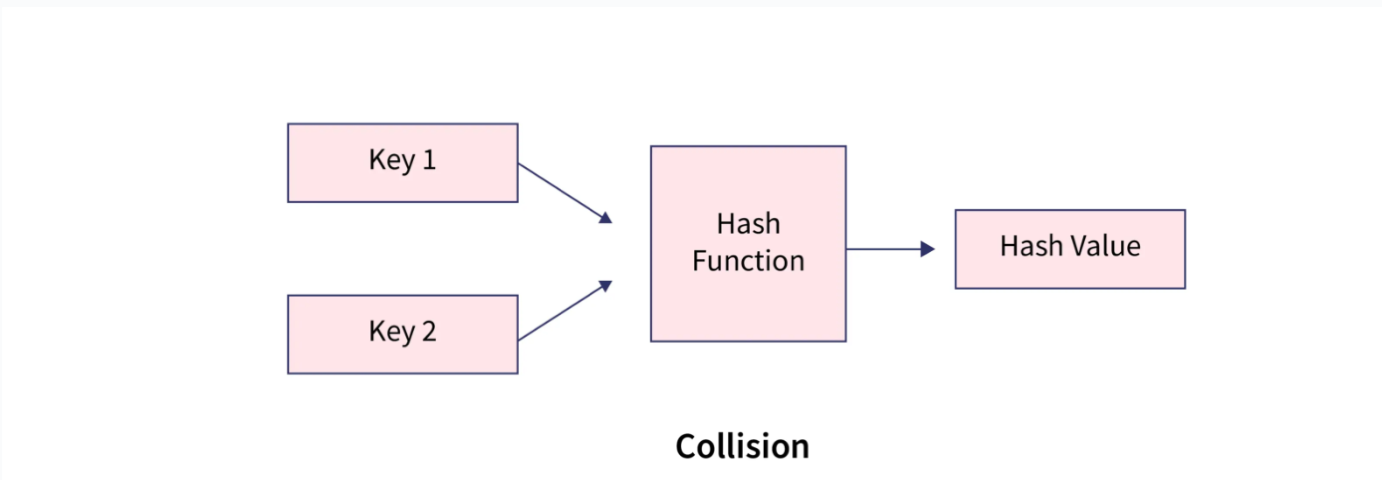
A hash table is an array that holds pointers to data that corresponds to a hashed key. Hash table uses hash values as the location index to store the associated data in the array.



Hashing in java can be termed as the entire process of storing data in a hash table in the form of key-value pairs, with the key computed using a hash function.

**Characteristic of a Hashing Algorithm**

* It should be very fast on its computation and should convert the given key into a hash value quickly.
* Hashing algorithm must avoid the collision. Actually, a collision occurs when two differed inputs to the hash function are converted to the same hash value.



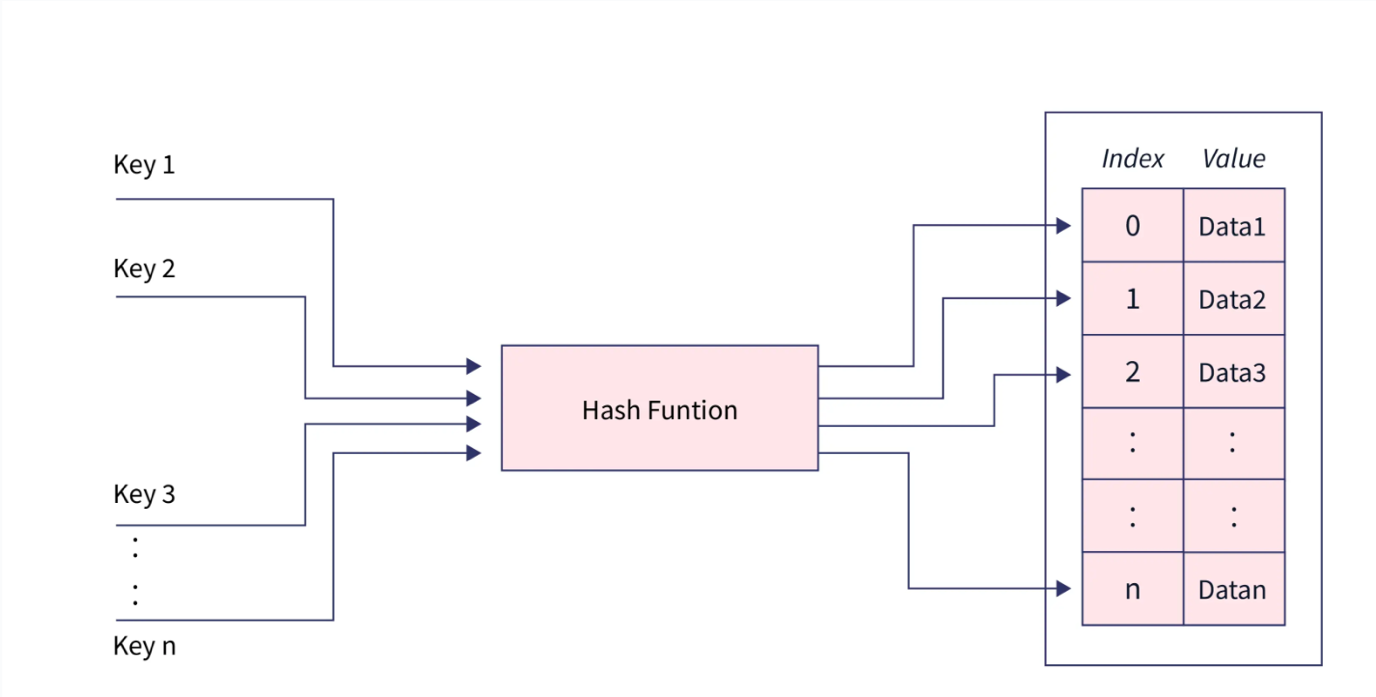
* When the message's hash value changes even slightly, the message's hash value must change as well. The avalanche effect is what it's called.

**How does Hashing Work?**

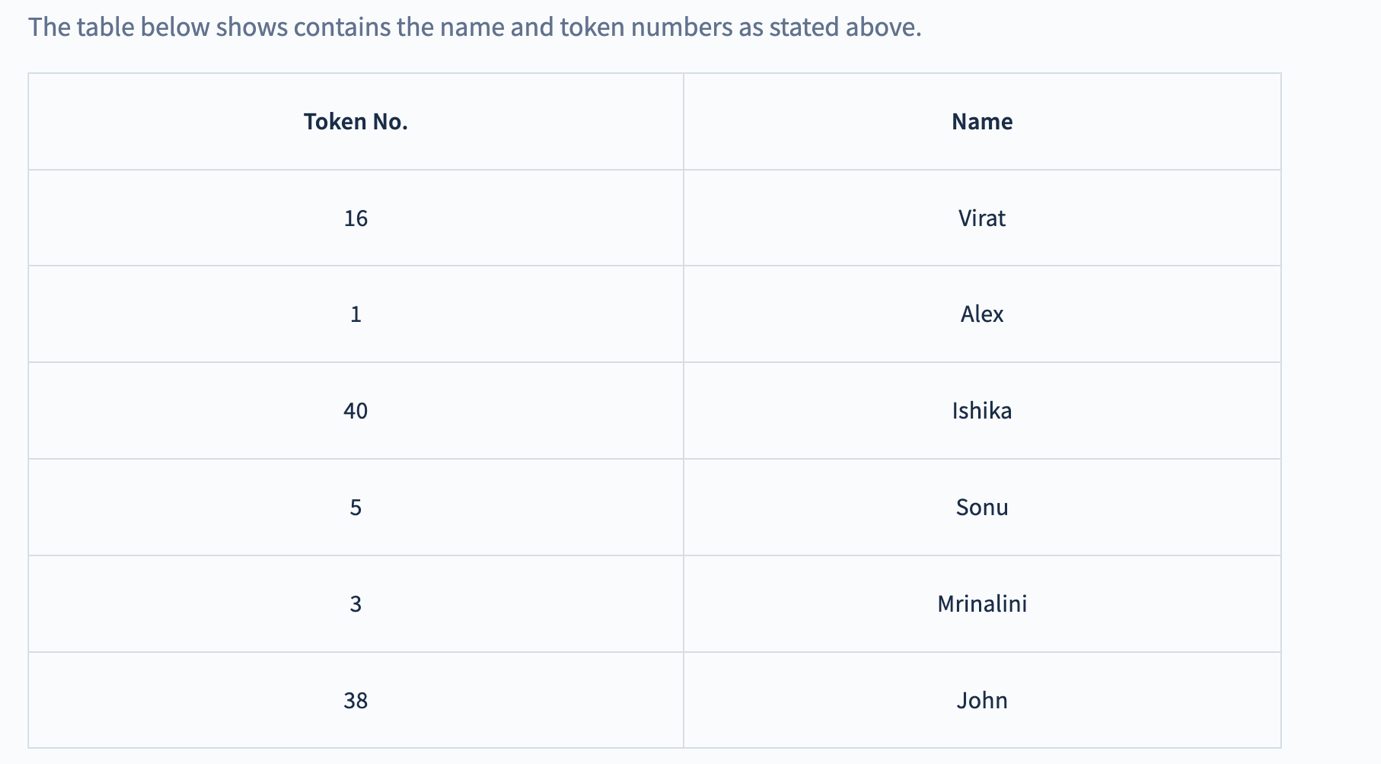
Hashing in java is a two-step process:

firstly, A hash function is used to turn an input key into hash values.

This hash-value is used as an index in the hash table and corresponding data is stored at that location in the table.

****

**Example:**

****

Our goal is to create a quick and space-efficient hash table for data storage and retrieval.

* To solve this problem, one nieve solution we can think of is to use an array of size 41 so that we can be able to use each key(token no) as the index to the array and store the data at those index locations.
* This works but it is inefficient and we will be wasting the majority of the space we have used because we will be having data stored at only six (1,5,3,16,38 and 40) locations out of 41. we should think of a method to narrow down the search space for us.
* In such cases, we can use hashing, We can use think of a function that is able to convert the given keys (token no) into less spread hashed keys and follows all the characteristics of the hash function discussed in the section above.

**Choosing hash function**

* If we look closely at the keys, we can see that they can easily be converted to numbers from 0 to 10 if we use

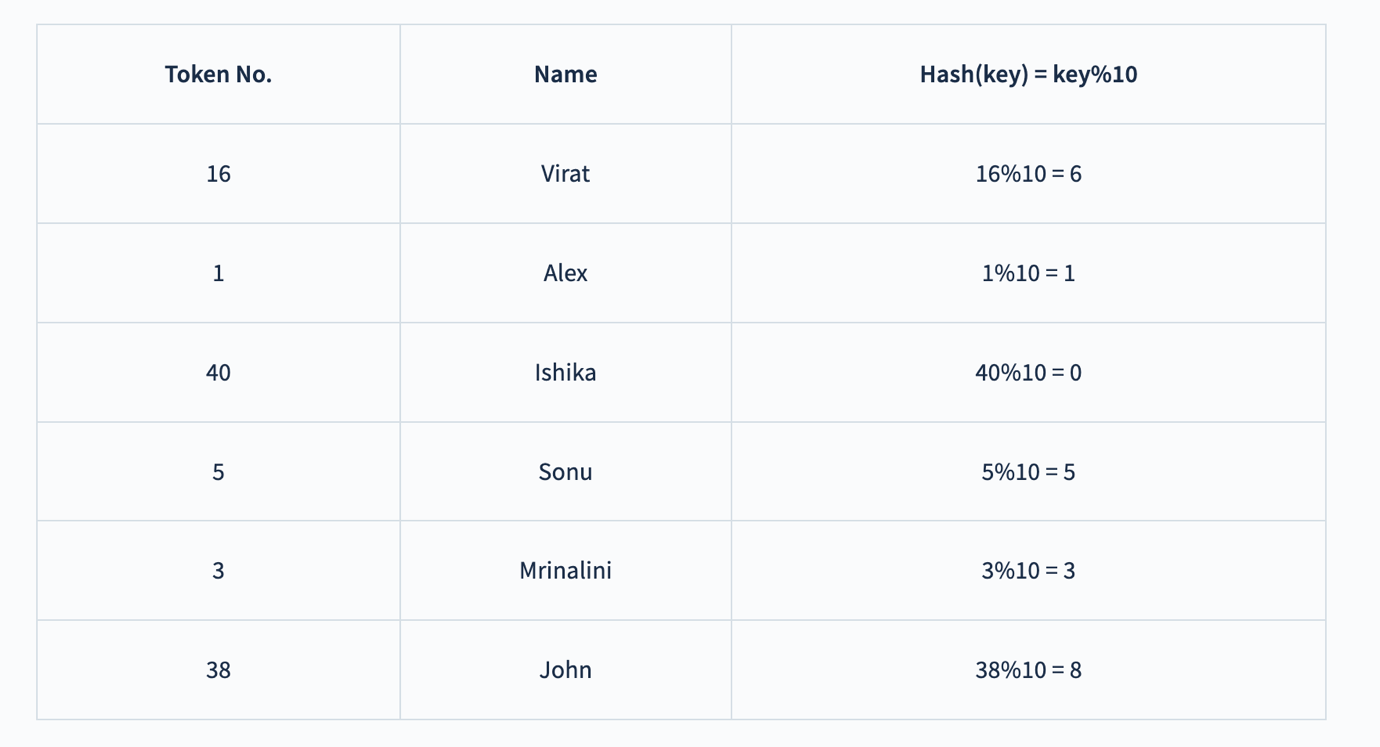
**Hash(key) = key%10.**

Using this hash function we can observe that

Hash(16) = 16%10 = 6, indicating that the value corresponding to key 16 i.e. (virat)will be stored in the array at index 6.

Similarly other keys can be hashed in the same way to find a suitable location in the array.

Our hash table should be of size 10 because the hash function can be able to give hash values from 0 to 9.

****

**So our hash-table will look like:**

****

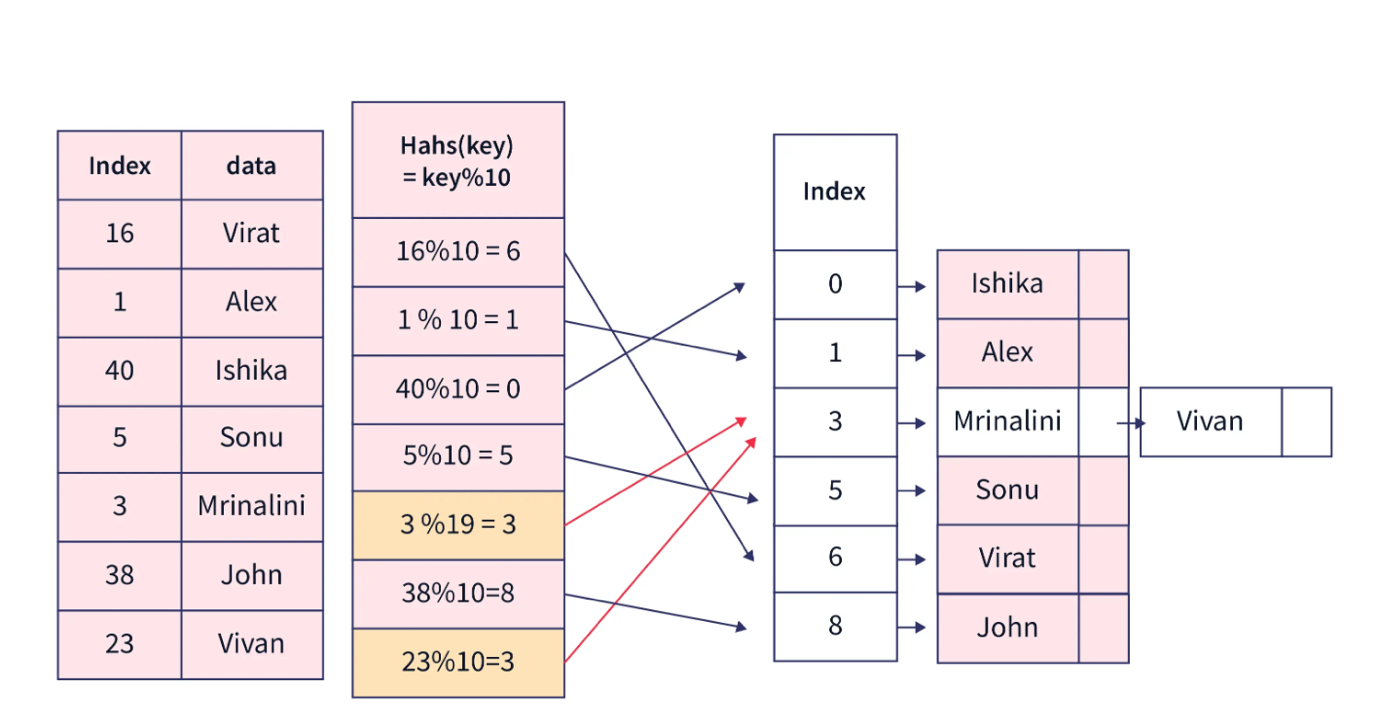
Now if we want to be able to get the name of the person with token no 38, then we can generate an index using the hash function we used above.

**index = hash(38) = 38%10 = 8.**

and we can get the value from the array now, table[8] will return John as the name of that person.

Now suppose we have one more person's data with token no 23, in that case, the hash function we used above will generate the same hash values for both 23 and 3. **This is what we call collision when two different keys are converted to the same hash value by the hash function**. We have no. of techniques to resolve collision.

**Chaining**: Although the hash function should minimize the collision, if it still happens then we can use an **array of LinkedList** as a hash table to store data, the fundamental idea is for each hash table slot to point to a linked list of records with the same hash value. This technique is called chaining type of hashing in java.



**HashMap**

HashMap, also known as HashMap<Key, Value> or HashMap<K, V>, is an easy way to implement hashing in java, it is a Map-based collection class that is used to store key-value pairs. It uses an array and LinkedList data structure internally for storing Key and Value.

* **It uses hashCode() method of object class, which implements hashing and returns the index of the object called hash-value or hash**. This hash is used as bucket no that is the address of element inside the map.
* The order of the map is not guaranteed by this class.
* HashMap does not allow duplicate keys to be stored. If you try to store a duplicate key with a different value, the value will be replaced. Which indicates that we can at max have one NULL key in the map.
* It has an initial capacity of 16 and load factor of 0.75.

import java.util.HashMap;  
import java.util.Map;  
  
class MapTest  
{  
    public static void main(String arg[])  
    {  
        Map<String, String> m = new HashMap<String, String>();  
        Map<String, String> m1 = new HashMap<String, String>();  
        System.out.println("Map objects equal : " + m.equals(m1)); // Equals method  
          
        m.put("8", "Prakash"); // Put method  
        m.put("31", "Shabaz");  
        m.put("12", "Raj ");  
        m.put("14", "Praveen");  
        m.put("5", "Gopi");  
          
        System.out.println("Map Elements : " + m); // Display map elements  
        System.out.println("Size of Map : " + m.size());  
        System.out.println("Key contains : " + m.containsKey("31")); // LINE A  
        System.out.println("Value contains : " + m.containsValue("Raj")); // LINE B  
        System.out.println("Gets the value of key : " + m.get("14")); // LINE C  
        System.out.println("Hash code for map : " + m.hashCode());  
        System.out.println("Is map empty : " + m.isEmpty());  
        System.out.println("Removes the key value : " + m.remove("12")); // LINE D  
        System.out.println("Key set : " + m.keySet()); // LINE E  
        System.out.println("Collection values : " + m.values()); // LINE F  
          
        m1.putAll(m); // LINE G  
        m.clear();  
        System.out.println("Map after clear : " + m);  
        System.out.println("All data of m is put into m1 map : " + m1);  
        System.out.println("Entry set : " + m1.entrySet());      
    }  
}

**Sorted Map**

The SortedMap interface extends Map. It ensures that the entries are maintained in ascending key order.

import java.util.Iterator;  
import java.util.Map;  
import java.util.Set;  
import java.util.TreeMap;  
  
class SortedMapTest  
{  
    public static void main(String arg[])  
    {  
        TreeMap<String, Double> tm = new TreeMap<String, Double>();  
        tm.put("Santosh", new Double(3020.55));  
        tm.put("Ram", new Double(2550.22));  
        tm.put("Nishan", new Double(2060.66));  
        tm.put("Amar", new Double(1890.88));  
        tm.put("Om", new Double(1650.11));  
        System.out.println("Map after initialization : " + tm);  
          
        Set set = tm.entrySet(); // Setting entry set  
        Iterator i = set.iterator(); // Iterating to set  
        while (i.hasNext()) {  
            // Assigning iterator to map entry  
            Map.Entry m = (Map.Entry) i.next();  
            if (m.getKey().equals("Ram")) {  
                m.setValue(3550.33); // Set value for Ram key  
            }  
            // Getting key and value from map entry  
            System.out.println(m.getKey() + " : " + m.getValue());  
        }  
        System.out.println("Map after changing Ram value : " + tm);  
          
        System.out.println("First key : " + tm.firstKey());  
        System.out.println("Last key : " + tm.lastKey());  
        System.out.println("Keys set : " + tm.keySet());  
        System.out.println("Values set : " + tm.values());  
        System.out.println("Head map : " + tm.headMap("Om"));  
        System.out.println("Sub map : " + tm.subMap("Nishan", "Santosh"));  
        System.out.println("Tail map : " + tm.tailMap("Om"));  
      
    }  
}

**Map.Entry Interface**

The Map.Entry interface enables you to work with a map entry.

The entrySet() method declared by the Map returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

import java.util.\*;  
  
class MapEntryTest  
{  
    public static void main(String arg[])  
    {  
        HashMap<String, Double> hm = new HashMap<String, Double>();  
        hm.put("Santosh", new Double(3020.55));  
        hm.put("Ram", new Double(2550.22));  
        hm.put("Nishan", new Double(2060.66));  
        hm.put("Amar", new Double(1890.88));  
        hm.put("Om", new Double(1650.11));  
        System.out.println("After initializing : " + hm);  
          
        Set set = hm.entrySet(); // Setting entry set  
        Iterator i = set.iterator(); // Iterating to set  
        while (i.hasNext()) {  
            // Assigning iterator to map entry  
            Map.Entry m = (Map.Entry) i.next();  
            if (m.getKey().equals("Ram")) {  
                m.setValue(3550.33); // Set value for Ram key  
            }  
            // Getting key and value from map entry  
            System.out.println(m.getKey() + " : " + m.getValue());  
        }  
        System.out.println("After changing : " + hm);      
    }  
}

**Weak HashMap**

WeakHashMap is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbagecollected when its key is no longer referenced outside of the WeakHashMap.

import java.util.\*;  
  
class WeakHashMapTest  
{  
        public static void main(String args[])  
        {      
          Map weakmap=  (Map) new WeakHashMap();  
          String one=new String("one");  
          String two=new String("two");  
          weakmap.put(one, "Lahari");  
          weakmap.put(two, "Latha");  
          System.gc();  
          System.out.println("Before: "+weakmap.get("one")+" "+weakmap.get("two"));  
          one=null;  
          two=null;  
           System.gc();  
          System.out.println("After: "+weakmap.get("one")+" "+weakmap.get("two"));     
        }  
}

**LinkedHash Map**

The Class LinkedHashMap is an extension of Java HashMap Implementation with specific feature of retaining the insertion order in the order, in which they were inserted. Also if one inserts the key again into the LinkedHashMap the original orders is retained.

This allows insertion-order iteration over the Java Map Interfaces - HashMap, TreeMap, LinkedHashMap. That is, when iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted. You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.

import java.util.\*;  
  
class LinkedHashMapTest  
{  
    public static void main(String arg[])  
    {  
        LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();  
        lhm.put("Ramesh", "Intermediate");  
        lhm.put("Shiva", "B-Tech");  
        lhm.put("Santosh", "B-Com");  
        lhm.put("Asha", "Msc");  
        lhm.put("Raghu", "M-Tech");  
          
        Set set = lhm.entrySet();  
        Iterator i = set.iterator();  
        while (i.hasNext()) {  
            Map.Entry me = (Map.Entry) i.next();  
            System.out.println(me.getKey() + " : " + me.getValue());  
        }  
                  
        System.out.println("The Key Contains : " + lhm.containsKey("Shiva"));  
        System.out.println("The value to the corresponding to key : " + lhm.get("Asha"));      
    }  
}

**Identity HashMap**

IdentityHashMap is similar to HashMap except for the following differences:

1) In case of HashMap, JVM will always use equals() method to identify duplicate keys.

2) In case of IdentityHashMap, as its name suggests, JVM will always use == operator to identify duplicate keys.

IdentityHashMap was added in Java 1.4. It implements Map interface. But it doesn’t use equals() and hashCode() methods for comparing objects unlike other implementations of Map (e.g., HashMap).

Which is faster HashMap or IdentityHashMap?

Since IdentityHashMap uses == operator for comparing objects, it is faster compared to HashMap.

When to use IdentityHashMap?

It is suitable where we need reference equality check instead of logical equality.

NOTE:

Any implementation of Map interface is supposed to use equals() to compare objects, but IdentityHashMap doesn’t do that. **So, it violates Map’s general contract.** Hence it is easy to see that it is a special kind of Map which is rarely used.

import java.util.\*;  
  
class IdentityHashMapDemo  
{  
    IdentityHashMap hm = new IdentityHashMap();  
    Integer i1 = new Integer(10);  
    Integer i2 = new Integer(10);  
    hm.put(i1, "Sachin");  
    hm.put(i2, "Dravid");  
    System.out.println(hm);  
}