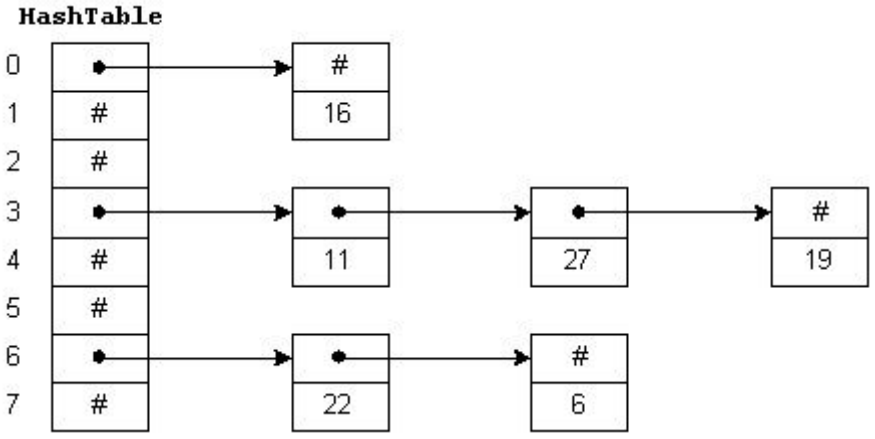
HashTable :

Java Hashtable class is an implementation of hash table data structure. It is very much similar to HashMap in Java, with most significant difference that Hashtable is synchronized while HashMap is not.

1. How Hashtable Works?

Hashtable internally contains buckets in which it stores the key/value pairs. The Hashtable uses the key's hashcode to determine to which bucket the key/value pair should map.



The function to get bucket location from Key's hashcode is called hash function. In theory, a hash function is a function which when given a key, generates an address in the table. A hash function always returns a number for an object.

Two equal objects will always have the same number while two unequal objects might not always have different numbers.

When we put objects into a hashtable, it is possible that different objects (by the equals() method) might have the same hashcode. This is called a collision. To resolve collisions, hashtable uses an array of lists.

The pairs mapped to a single bucket (array index) are stored in a list and list reference is stored in array index.

```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable
{
    //implementation
}
```

The important things to learn about Java Hashtable class are:

1. It is similar to HashMap, but it is synchronized while HashMap is not synchronized.
2. It does not accept `null` key or value.
3. It does not accept duplicate keys.
4. It stores key-value pairs in hash table data structure which internally maintains an array of list. Each list may be referred as a bucket. In case of collisions, pairs are stored in this list.
5. Enumerator in Hashtable is not fail-fast.

Hashtable(): It is the default constructor. It constructs a new, empty hashtable with a default initial capacity (11) and load factor (0.75).

Please note that initial capacity refers to number of buckets in hashtable. An optimal number of buckets is required to store key-value pairs with minimum collisions (to improve performance) and efficient memory utilization.

The fill ratio determines how full hashtable can be before it's capacity is increased. It's Value lie between 0.0 to 1.0.

Performance wise HashMap performs in O(log(n)) in comparion to O(n) in Hashtable for most common operations such as get(), put(), contains() etc.

The naive approach to thread-safety in Hashtable ("synchronizing every method") makes it very much worse for threaded applications. We are better off externally synchronizing a HashMap. A well thought design will perform much better than Hashtable.

Hashtable is obsolete. Best is to use ConcurrentHashMap class which provide much higher degree of concurrency.

Performance wise HashMap performs in O(log(n)) in comparion to O(n) in Hashtable for most common operations such as get(), put(), contains() etc.

The naive approach to thread-safety in Hashtable (“synchronizing every method”) makes it very much worse for threaded applications. We are better off externally synchronizing a HashMap. A well thought design will perform much better than Hashtable.

Hashtable is obsolete. Best is to use ConcurrentHashMap class which provide much higher degree of concurrency.