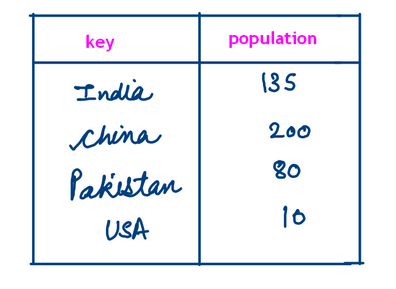
**1. Understanding Problem**

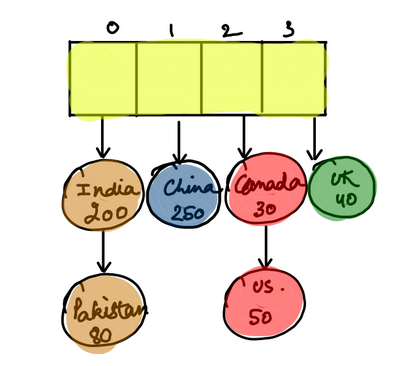
You are required to complete the code of our Hashmap class. Please watch the video for this question too. The theoretical details of the required functionality is explained in detail there.

We have already discussed the functions, get( ), put( ), containsKey( ), remove( ), keyset( ) and size( ).

To understand them better, we consider the hashmap given in figure 1 which depicts an example of a "country vs. its population" chart.

****

Now we see how the hashmap is implemented. A hashmap contains an array of linked lists as shown in figure 2.

****

The yellow part is the array and each block/bucket of the array stores a linked list.

**2. Approach**

What, How & Why

We have already been given the following code

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static class HashMap< K, V> { //1

private class HMNode {

K key;

V value;

HMNode(K key, V value) {

this.key = key;

this.value = value;

}

}

private int size; // n

private LinkedList< HMNode>[] buckets; // N = buckets.length

public HashMap() { //2

initbuckets(4);

size = 0;

}

private void initbuckets(int N) { //3

buckets = new LinkedList[N];

for (int bi = 0; bi < buckets.length; bi++) {

buckets[bi] = new LinkedList<>();

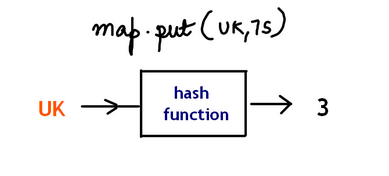
}

}

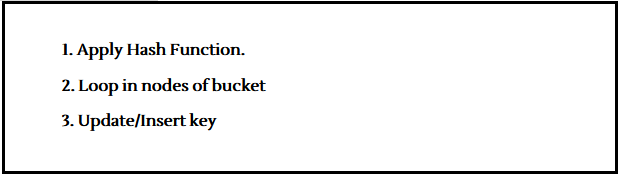
● We are given a class HashMap of generic support < K,V > where K=key and V=value. This contains a private class HMNode. This class contains a key and its value as the data members and a constructor for initializing these values. ● A constructor for class HashMap is formed where the size of the array is set to 4 using "initbuckets()" function and the size is initialized to 0. ● The function "initbuckets" creates a new linked list of the parameter passed to it and each bucket in that linked list stores a new linked list. ● Let's discuss each function one at a time and simultaneously write the code for them.

put()

If we call the put function as "put(UK, 75)" and if the key of "UK" already exists in the hashmap then its value of 40 will get updated to 75. Else if the key "UK" doesn't exist in the hashmap then a new key named "UK" is inserted in the hashmap against a value of 75. We see how this process works in figure 3.

****

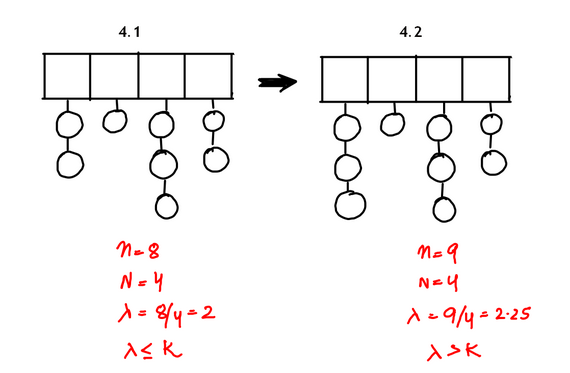
A "hash function", takes the key as the input and gives us the index of the bucket of the array at which that key is stored (according to figure 2, an index 3 is returned for UK). Now we compare the "UK" key with the rest of the keys in the ArrayList of that bucket. If our required key is found, then we make the necessary changes and if not found, we add that key to the last of the arraylist. Hence, the steps are:

****

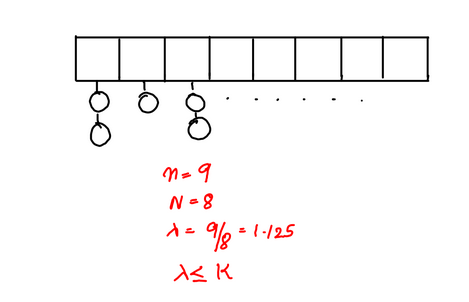
We say that n= total number of elements= 6 and N= number of blocks/buckets of array=4 (both according to figure 2). Hence, LF=n/N = loading factor. This loading factor denotes the "average number of elements per bucket". Now we say that the time complexity of the put( ) function is O(LF). Can you think why? It is because the put( ) function compares only the keys of a bucket and the other steps are formed in constant time. You also need to know that LF<= K, where K is a constant threshold value. Since LF should always be less than or equal to K, therefore we perform a process called "Rehashing" if LF crosses the value of K. After calling the put() function, the key gets updated/inserted. If the key is updated then LF remains the same because there is no change in either n or N. But if the key gets inserted, then, n increases and hence LF increases. When LF grows, we immediately check whether it has crossed the value of K. If it has, then we apply Rehashing.

REHASHING

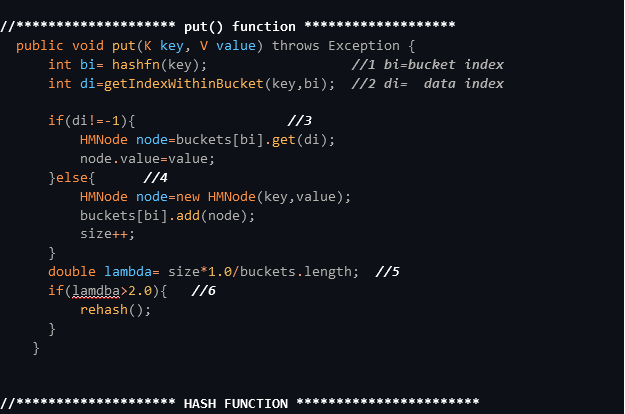
We can understand rehashing using the example given below where K is taken as 2.

****

In figure , observe that the value of LF is less than/equal to K. If we insert a key to the first bucket as shown in figure, then the LF increases and crosses the value of K. Now we apply Rehashing. A new array of linked lists is made of size 8 (twice the size of initial array=4).

****

Here, the keys from the original array get distributed throughout the new array of size 8. By using an array of bigger size, N increases and hence we are able to reduce the value of LF such that it is under the threshold limit. Refer to the code below for put() to understand it better.

****

1• We store the value of the bucket index in "bi" by passing the key through the hash function. 2• The data index of the key in that bucket is found by using the "getIndexWithinBucket" function for the parameters "key" and "bi". 3• If the "di" value is not -1 i.e. the key is found in the bucket, we simply update that key. We get the index of the key from the bucket "bi" and change its value to the given value. 4• Else if the "di" value is -1, i.e. the key is not found, we insert the key in the bucket "bi". We make a new HMNode "node" and assign it the given values of "key" and "value". This node is added to the bucket "bi" and the size is increased by 1. 5• A variable "lambda" is assigned the value n/N or (size\*1.0/buckets.length). Here size is multiplied by 1.0 to convert it into a "double" value. 6• If this lambda comes out to be greater than the threshold value, K (Here taken as 2.0), we call the rehash() function as already discussed before.

getIndexWithinBucket() Function

1• We initialize the value of "di" with 0. 2• We iterate through every node of the linked list at the "bi" index. If the "key" of the node matches the key we passed then that data index "di" is returned. 3• If that key is not found in the bucket then we increase "di" by 1. 4• If we come out of the loop, i.e. the key is not found then we simply return -1.

rehash() Function

1• We make a linked list of type HMNode and store our original array of buckets in it. 2• We call the "initbuckets" function which renews the array "buckets" with a length twice of the original array. 3• Now we initialize "size" with 0. 4• Next, we iterate through the original bucket array and then for each bucket of the array, go through all the nodes 5• Then the put function is called for each of these nodes so that they can get distributed in the renewed buckets array.

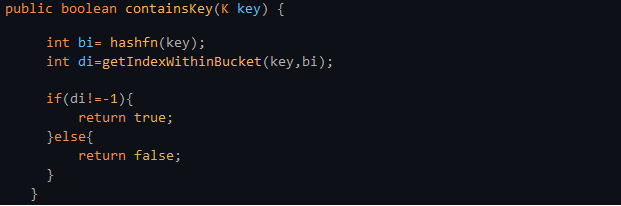
get( ), containsKey( ) & remove( )

Reader, these functions also run similar to the put( ) function. The following steps are involved:

1• Apply Hash Function and obtain bucket index. 2• Loop in block/bucket 3• Get the value/ return "true"or "false" for containsKey()/ remove the key

In these functions again, the time complexity is O(LF) as long as LF<=K. If LF crosses the value of K. Rehashing is applied. We study these functions one by one.

containsKey Function

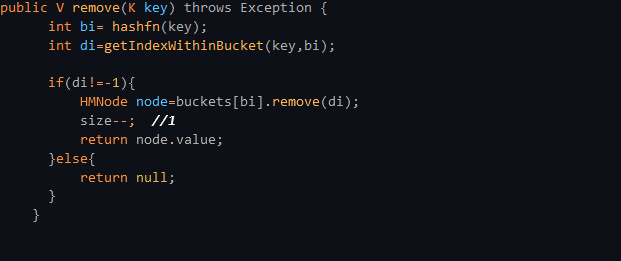
****

We copy most of our code from the put() function. Here, instead of updating/inserting the code, we simply return "true" if the key is found and "false" if it isn't.

get() Function

Again, the get() code is similar to the put() code. We have to return the value of the key if it is found. Else we just return "null".

remove() Function

****

The remove() function is also the same as the get() function. If the required node is found, we just decrease the size by 1 and return the "value" of that node as seen in 1st point.

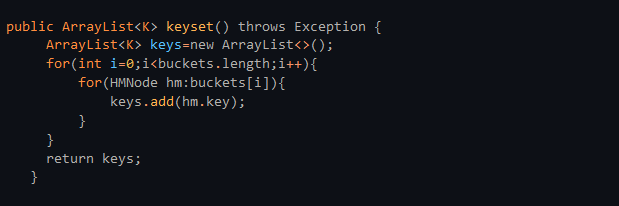
Now let's also discuss the remaining 2 functions: remove() and keyset().

size() Function

****

The size is simply returned.

keyset() Function

****

On using the "keyset()" we put all the keys of all the buckets in an ArrayList called "keys". The code for the same is pretty straightforward. With this we conclude our article. It might appear a bit difficult at first because it contains a lot of functions. But we promise, if you go through the article line by line it'll become easier with every attempt. We highly suggest you check out its solution video too. It will give you great insights in this concept.