**overview**

Double hashing is a computer programming technique used in conjunction with open addressing in hash tables to resolve hash collisions, by using a secondary hash of the key as an offset when a collision occurs.

* Complexity of Double hashing algorithm
  + Time complexity - O(n)

**Introduction to Double Hashing**

**Hashing** is a powerful technique used for storing and retrieving data in average constant time. In this technique, we store data or some keys in a fixed-size array structure known as a hash-table.

Each key is mapped to a specific location on the hash-table. This mapping is known as a hash function. An ideal hash function must map each unique key to a unique location on the hash table.

In most practical scenarios, the number of keys is way too large than the number of locations in the hash table, as a result, no matter how good the hash function is, collisions are bound to happen.

A collision is said to occur when two or more distinct keys are hashed to the same location of the hash table. To resolve these collisions, we need some kind of collision resolution techniques. One such effective collision resolution technique is **Double Hashing**.

**What is Double Hashing?**

In double hashing, we make use of two hash functions. The first hash function is ℎ1*h*1​(k), this function takes in our key and gives out a location on the hash-table. If the new location is empty, we can easily place our key in there without ever using the secondary hash function.

However, in case of a collision, we need to use secondary hash-function *h*2​(k) in combination with the first hash-function *h*1​(k) to find a new location on the hash-table. The combined hash-function used is of the form h(k,i) = (*h*1​(k) + i \* *h*2​(k))%m. Here, i is an non-negative integer which signifies the collision number, k = element/key which is being hashed and m= hash table size.

The use of secondary hash-function *h*2​(k) after the collision, helps us to reach new locations on the hash-table, each new location is at a distance of *h*2​(k), 2\**h*2​(k), 3\**h*2​(k)….., this results in a non-linear fashion of addressing hash-table which reduces the number of collisions.

A very important point to be considered is that both the hash functions should be computed in order of O(1) time.

1. In the diagram, we have a hash-table of size 5 and two keys **10, 15** which need to be inserted in the hash-table. We are using two hash-function here, **first hash-function = *h*1​(key) = key % 5** and **secondary hash-function = *h*2​(key) = key % 7**
2. We start with the first key **10** and apply our first hash-function to get the location for **10** on the hash-table.  
   Location of **key 10 = *h*1​(10) = 10 % 5 = 0 0*th*** location. Now the 0th location is empty and hence we can place our **10** there. No need to use the secondary hash function here.
3. Now, we take our second key **15** and apply our first hash function again.  
   Location of **key 15 = *h*1​(15) = 15 % 5 = 0 0*th*** location. Oops! This time the 0 0*th* location is not empty, hence it is a collision. We need to find a new location for key **15**, for this we use a secondary hash-function.  
   New Location of **key 15 = *h*1​(15) + i \* *h*2​(15) = 0 + (1 \* 15 % 7) = 1 1*st*** location. The first location of the hash-table is empty and we can place key 15 here.

**Example of Double Hashing in Data Structure**

The idea behind double hashing is fairly simple,

1. Take the key you want to store on the hash-table.
2. Apply the first hash function *h*1​(key) over your key to get the location to store the key.
3. If the location is empty, place the key on that location.
4. If the location is already filled, apply the second hash function *h2​(key)* in combination with the first hash function *h1​(key)* to get the new location for the key.

Let’s explore this idea in more detail, with a hands-on problem,

**Problem Statement:**

Insert the keys 79, 69, 98, 72, 14, 50 into the **Hash Table of size 13**. Resolve all collisions using Double Hashing where first hash-function is ***h*1​(k) = k mod 13** and second hash-function is ***h*2​(k) = 1 + (k mod 11)**

**Solution:**

Initially, all the hash table locations are empty. We pick our **first key = 79** and apply *h*1​(k) over it,

***h*1​(79) = 79 % 13 = 1**, hence key 79 hashes to 1 1*st* location of the hash table. But before placing the key, we need to make sure the 1 1*st* location of the hash table is empty. In our case it is empty and we can easily place key 79 in there.

**Second key = 69**, we again apply *h*1​(k) over it, ***h*1​(69) = 69 % 13 = 4**, since the 4 4*th* location is empty we can place 69 there.

**Third key = 98**, we apply *h*1​(k) over it, ***h*1​(98) = 98 % 13 = 7**, 7 7*th* location is empty so 98 placed there.

**Fourth key = 72**, ***h*1​(72) = 72 % 13 = 7**, now this is a collision because the **7 7*th*** location is already occupied, we need to resolve this collision using double hashing.

**ℎ *hnew*​ = [ *h*1​(72) + i \* *h*2​(72) ] % 13**

**= [ 7 + 1 \* ( 1 + 72 % 11) ] % 13**

**= 1**

Location **1 1*st*** is already occupied in the hash-table, hence we again had a collision. Now we again recalculate with i = 2

**ℎ *hnew*​ = [ ℎ1*h*1​(72) + i \* ℎ2*h*2​(72) ] % 13**

**= [ 7 + 2 \* ( 1 + 72 % 11) ] % 13**

**= 8**

Location 8 8*th* is empty in hash-table and now we can place key 72 in there.

**Fifth key = 14, *h*1​(14) = 14%13 = 1**, now this is again a collision because 1st location is already occupied, we now need to resolve this collision using double hashing.

**ℎ *hnew*​ = [ *h*1​(14) + i \* *h*2​(14) ] % 13**

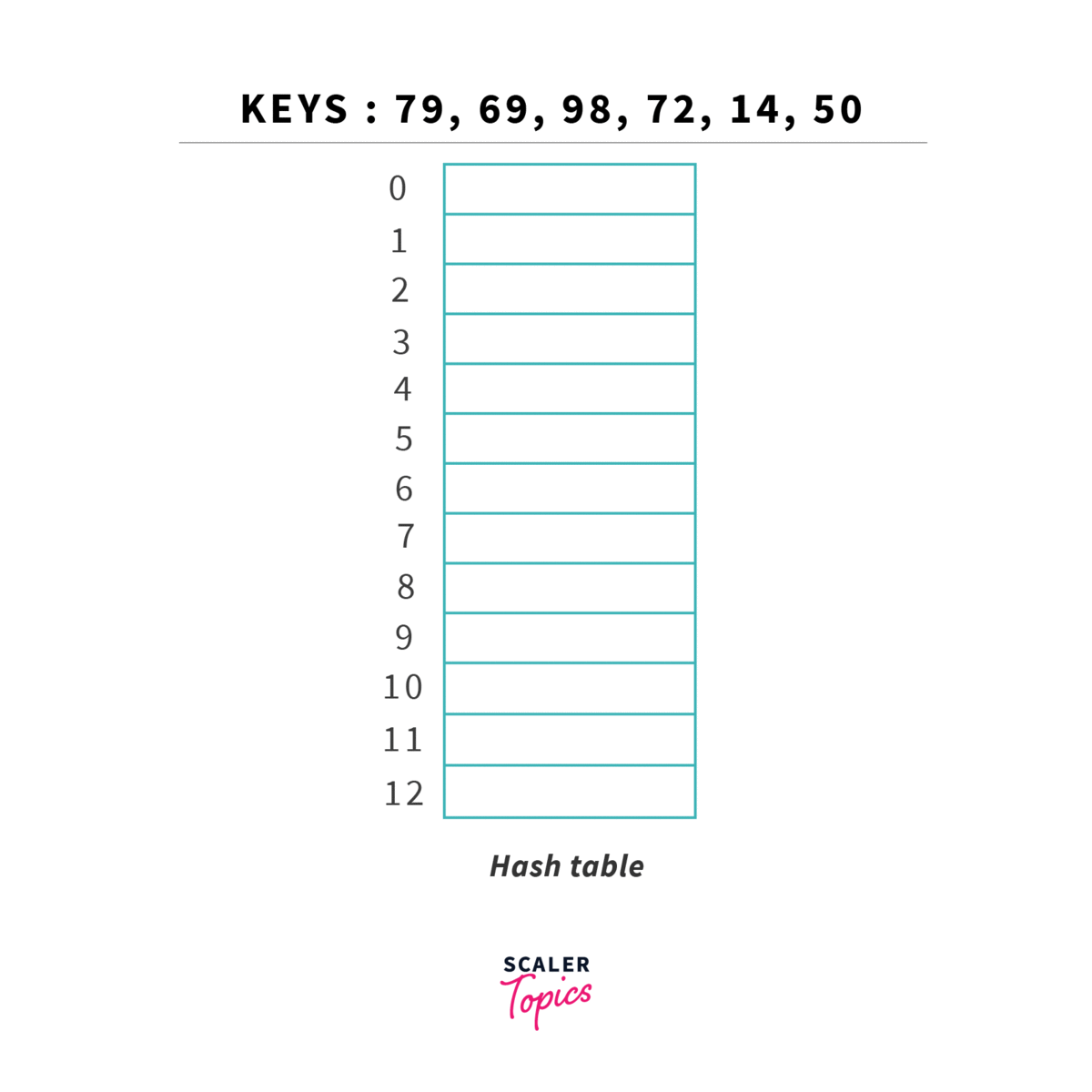
**= [ 1 + 1 \* ( 1 + 14 % 11) ] % 13**

**= 5**

Location 5 5*th* is empty and we can easily place key 14 there.

**Sixth key = 50**, ***h*1​(50) = 50%13 = 11**, 11 11*th* location is already empty and we can place our key 50 there.

Since all the keys are placed in our hash table the double hashing procedure is completed. Finally, our hash table looks like the following,



**Why Use Double Hashing?**

Double hashing is one of the popular collision resolution techniques. The other popular variants which serve the same purpose are **Linear Probing** and **Quadratic Probing**. But if other techniques are available, then why do we need double hashing in the first place?

Double hashing offers better resistance against clustering. A major reason for this is the use of dual functions. Dual functions enable double hashing to perform a more uniform distribution of keys, resulting in lower collisions.

**Advantages of Double Hashing in Data Structure**

1. After each collision, we recompute the new location for the element in the hash-table. For successive collisions, this generates a sequence of locations. If the sequences are unique then the number of collisions are less.
2. Double hashing creates most unique sequences, providing a more uniform distribution of keys within the hash-table.
3. If the hash function is not good enough, the elements tend to form grouping in the hash-table. This problem is known as clustering. Double hashing is least prone to clustering.

**Practice Problem Based on Double Hashing**

**Problem Statement 1:**

Given the two hash functions, ***h*1​(k) = k mod 23** and ***h*2​(k) = 1 + k mod 19**. Assume the table size is 23. Find the address returned by double hashing after 2nd collision for the key = 90.

**Solution:**

We will use the formula for double hashing-

**h(k,i) = ( *h*1​(k) + i \* *h*2​(k) )%m**

As it is given, k = 90, m = 23

Since the 2nd collision has already occurred, **i = 2**. Substituting the values in the above formula we get,

h(90,2) = [ ( 90 % 23 ) + 2 \* ( 1 + 90 % 19 ) ] % 23

= [ 21 + 2 \* 15 ] % 23

= 5

Hence after the second collision, the address returned by double hashing for Key = 90 is 5.

**Problem Statement 2:**

Given an Array[] of positive integers and a number **n**, find if there exists a pair of numbers in the array whose sum equals **n**. If such a pair exists, return true else false.

**Example:**

1. **Input:-** A = [ 6, 4, 3, 1 ] and n = 7  
   **Output:- True,** the pairs are {4,3} or {6,1}  
   **Explanation:-** 4 + 3 = 7 or 6 + 1 = 7
2. **Input:-** A = [ 6, 4, 3, 1 ] and n = 8  
   **Output:- False**, no such pair exist  
   **Explanation:-** There is no pair of numbers in array, whose addition would sum up to 8

**Solution:**

boolean isPairSumPresent(int[] arr, int n) {

HashSet<Integer> hSet = new HashSet<>();

for (int num : arr) {

int diff = n - num;

if (hSet.contains(diff)) {

return true;

} else {

hSet.add(num);

}

}

return false;

}

1. We maintain a hash-set of integers which would store the complement of array numbers.
2. The hash-set ensures that each look-up can be completed in average O(1) time. It internally maintain’s so-called buckets and then tries to distribute our elements among these buckets depending upon the hash-function.
3. The core logic is to iteratively check our hash-set whether it contains the desired complement number or not. For ex: If the sum was 8 and the first element of the array is 3 then in the hash-set we check if **5** *(8 - 3)* exists or not.
4. If such a complement is not present in the hash-set, then we simply put the array number in the hash-set and move forward.
5. If such a complement exists, then we return **true** and stop.

**Time Complexity of Double Hashing:**

1. Each lookup in the hash-set costs O(1) constant time.
2. In the worst case we might not find such a compliment and iterate through the entire array, this would cost us O(n) linear time.
3. Rest operations like subtraction are mere constant-time operations.
4. Hence total time complexity = O(n) time.

**Space Complexity of Double Hashing:**

1. We need to maintain an extra hash-set of size upto **n elements** which costs us extra O(n) space.
2. Hence total space complexity = O(n) time.