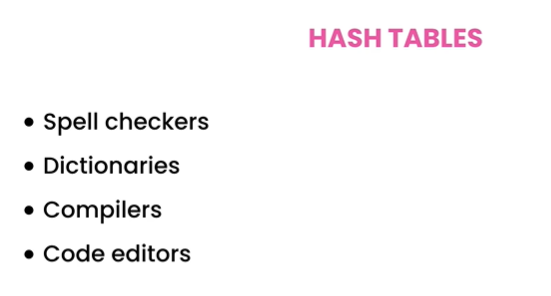
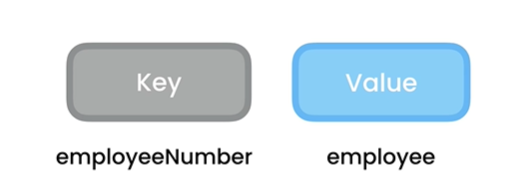
Hash Tables/Dictionaries-(Super-Fast Lookup)

V-(1+2)-HashTables?

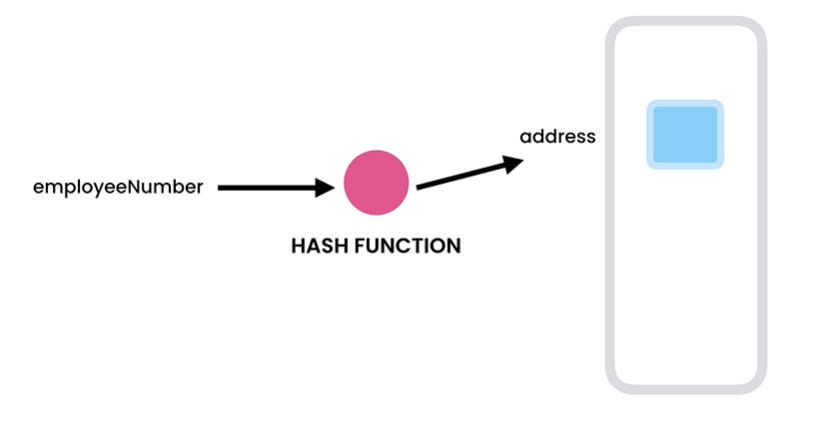






**Hash Table**

**Attributes**

Functionality:-

Its super-fast and it return same value for same input that’s why we use hash tables for storing and retrieving data’s.[Internally Hash Table use array’s]

**Same Input**

**Same Value**

**Always**



As Hash Table use arrays for implementation that’s why we don’t have to iterate the whole array. Many people had contradiction but most of them came to this conclusion of time complexities to be O(1).

(V-3)-(Working with Hash Tables):-

Hash Table is actually a implementation of Map Interface-(<https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>).

Uses in

multithread programming.



Older Implementation.

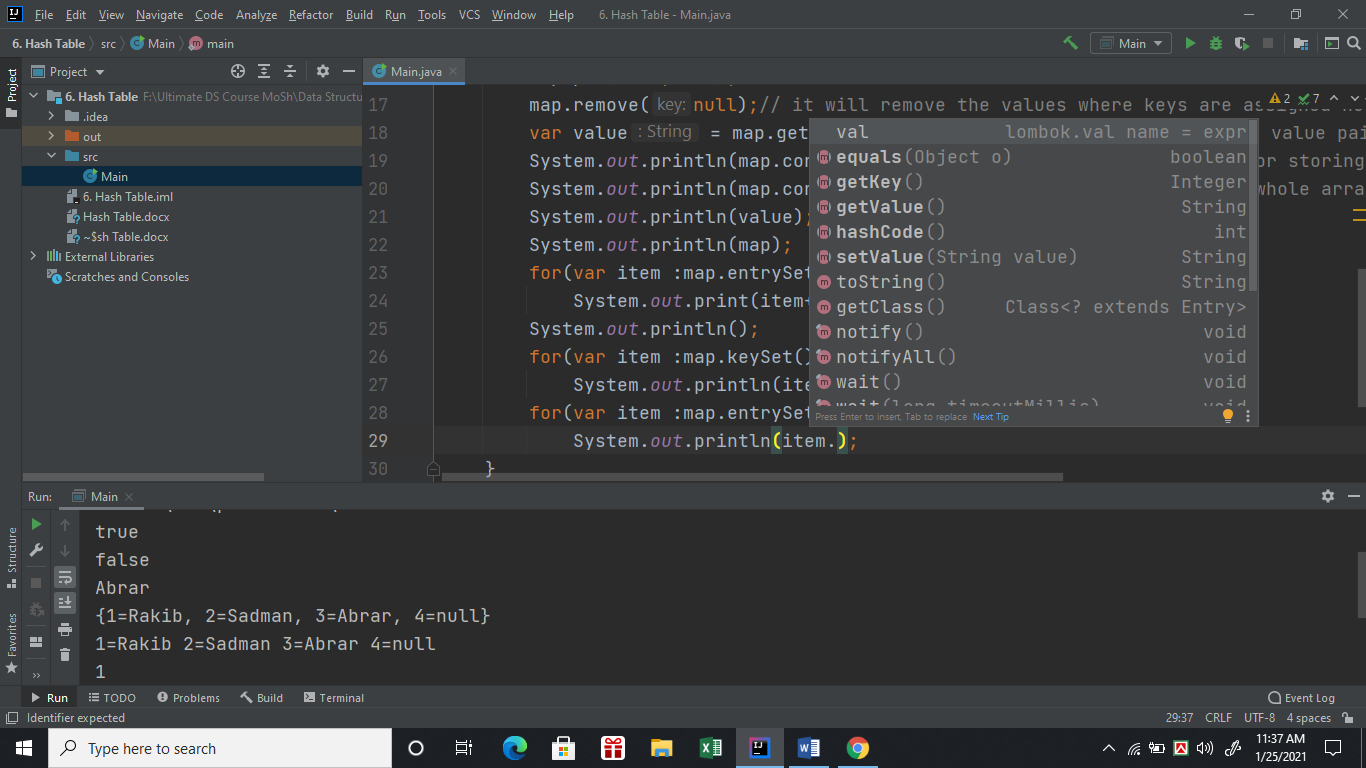
People don’t use this now-a-days.

Maximum Time we use this class

For implementing Hash Tables.

Basic methods(must) of HashMaps:-

import java.util.HashMap;  
import java.util.Map;  
// This main class has very basic implementation using built in methods in Hash Map  
public class Main {  
 public static void main(String[] args) {  
 // Key : employee Number (Integer)  
 // Value: Name (String)  
 Map<Integer, String> map = new HashMap<>();  
 map.put(1,"Rakib");// adding items method  
 map.put(2,"Sadman");  
 map.put(3,"Nabik");  
 map.put(3,"Abrar");// this will override value of key ==> 3  
 // in interview they ask if map can store both key and value as null  
 // So, it can store both null value in map  
 map.put(4,null);  
 map.put(null,null);  
 map.remove(null);// it will remove the values where keys are assigned null  
 var value = map.get(3);// it will return a String as our key value pair is <Integer,String>  
 System.*out*.println(map.containsKey(3));// O(1) ==> fixed address for storing and retriving the value  
 System.*out*.println(map.containsValue("Nabik"));//O(n) ==> iterate whole array and compare value with each keys  
 System.*out*.println(value);// it will give Abrar  
 System.*out*.println(map);  
 for(var item :map.entrySet())// print key value pair  
 System.*out*.print(item+" ");  
 System.*out*.println();  
 for(var item :map.keySet())// print the keys only  
 System.*out*.println(item+" ");  
 }  
}



**Hash Table**

**Methods(extra)**

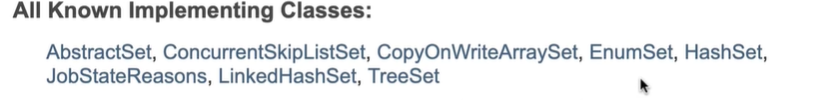
V-(4 + 5):-[First Non\_Repeated Character]:-[Most Popular Interview question]

import java.util.HashMap;  
import java.util.Map;  
  
public class CharFinder {  
 //popular interview question  
 // Find first non repeated character from a given String.  
 // String = "a green apple"  
 // So, for the given example it is 'g'  
 // This is a good exercise for hashmap  
 // a green apple from this we build hashmap  
 // a = 2  
 // = 2  
 // g = 1 etc for the whole string and return the first character with an occurance of 1  
 public char findFirstNonRepeatingChar(String str){  
 Map<Character,Integer> map = new HashMap<>();  
 var chars = str.toCharArray();  
 for(char ch:chars){//full string iterated using chararray  
 //if character is got multiple times then count value will be incremented by 1 otherwise it will set to 0  
 var count = map.containsKey(ch)? map.get(ch):0;  
 map.put(ch,count+1);  
  
 }  
 // System.out.println(map);  
 // now check if it had only one occourance  
 for (char ch :chars){  
 if(map.get(ch)==1)// occurance 1 first return  
 return ch;  
 }  
 return Character.*MIN\_VALUE*;// if all characters are repeated.  
 }  
}

V-6): Sets

HashTable : Key 🡺 Value but Set: Unique Values.

Details of Set interface: ( <https://docs.oracle.com/javase/7/docs/api/java/util/Set.html>)



Among all the classes implemented with sets we just have to use HashSet in maximum real life implementation.

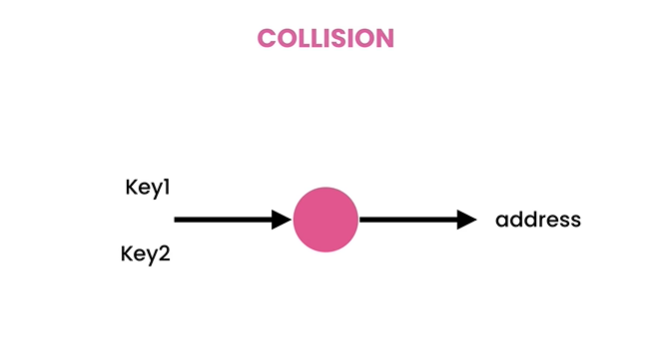
V-(7+8)First Repeated Character:-

public char findfirstRepeatedCharacter(String str){  
// We can solve this using set, cause set willnot contain any repeated value  
 java.util.Set<Character> set = new HashSet<>();  
 for(char ch:str.toCharArray()){//covert string to chararray  
 if(set.contains(ch))//check if it is already in the set if it already exist then immidiately return it and thats our first repeated character  
 return ch;  
 set.add(ch);  
 }  
 return Character.*MIN\_VALUE*;  
}

V-(9)-Hash Function:-

import java.util.HashMap;  
import java.util.HashSet;  
import java.util.Map;  
  
public class Hashing {  
 public static void main(String[] args) {  
 Map<Integer,String> map = new HashMap<>();  
 map.put(167546,"Rakib");  
 //Now we have a limited array of 100 items  
 // so we can't declare a an array of 167546 (not needed)  
 // We can use hashing  
 // though the key value is represented using a big int value we can simplify it using mod  
  
 System.*out*.println(*hash*(167546));  
 //output for 167546 is 46 for hash method that means  
 // if we want to save the key value pair in an array of 100 it should be in the 46th index  
 Map<String,String> map1 = new HashMap<>();  
 map1.put("1234567-A","Rakib");  
 System.*out*.println(map1);  
 System.*out*.println(*hashString*("1234567-A"));  
 Map<Integer,String> rakibul = new HashMap<>();  
 rakibul.put(1,"Ra\*765kib");  
 //items[1]=Ra\*765kib;  
 System.*out*.println(*hashString*("Ra\*765kib"));  
 //{1,Ra\*765kib}  
 }  
 public static int hash(int number){  
 return number%100;  
 }  
 public static int hashString(String key){  
 int hash = 0;  
 for(var ch:key.toCharArray())  
 hash+=ch;// this is an implicit casting  
 //as all the characters are represented by a number  
 // though we add an integer with an character but character is coverted  
 // to an integer and added with hash which is also an integer  
 return hash%100;  
 }  
}

V-(10)Collision:-

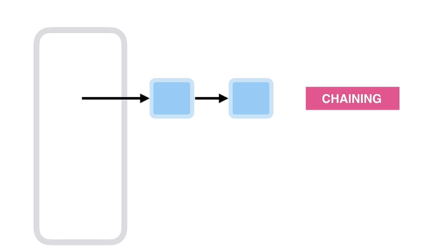


**Two separate keys**

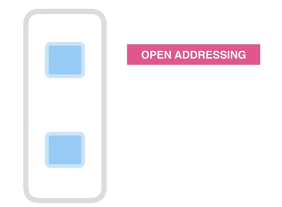
**want to store Same Value**

**That generate Collision.**

Two ways to solve this collision problem.

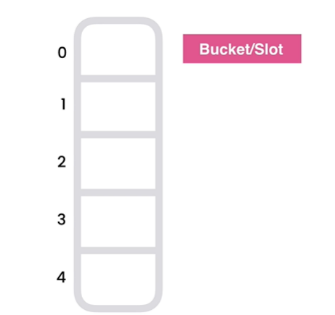
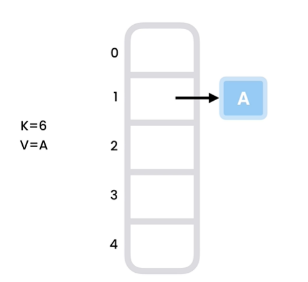
1. Chaining:-(Use Linked List instead of array and if collision happened then add the value next to the value (Collision with))

2. Open Addressing :-( Here we have to generate new address for storing the value(Collision with)



V-(11)-Chaining:-

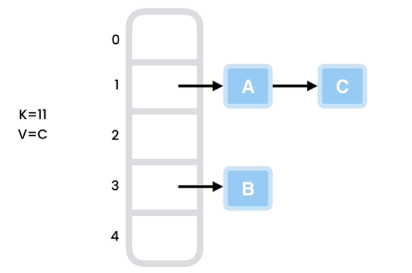
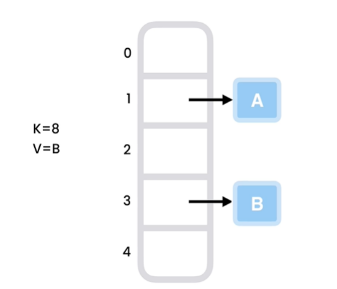
We don’t directly inserted to the remainder (index) instead we point that index and built Linked List.



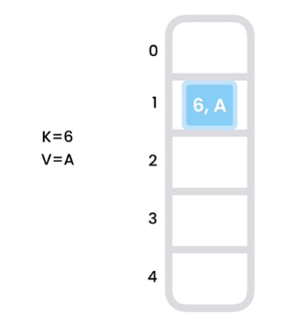
**6%5 = 1 . So we point to index 1**

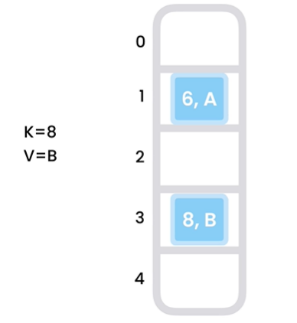
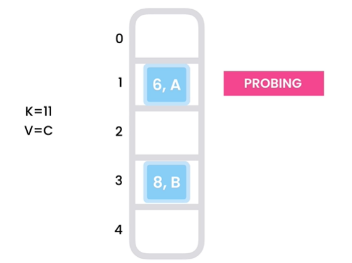
**Formula:- (Key % array.length)**

**8%5=3**

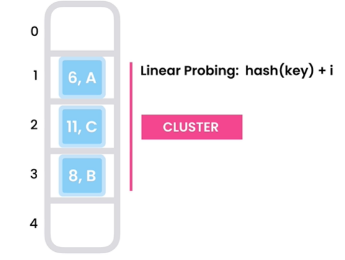


**11%5=1**

V-(12)-Open Addressing-(method1-Linear Probing):-



**6%5=1**



**If index out of bound then mod the value(Hash). But if we found any index for 1 then it have to search for next available position . As time passes cluster will increases and it make the program slow.**

**It will search**

**For next available**

**Position to insert key value pair which is**

**also called searching/PROBING.**

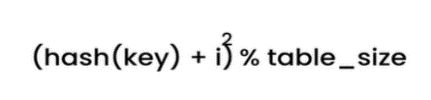
**1 index is already**

**booked**

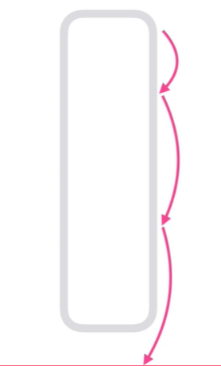
**11%5=1**

**8%5=3**

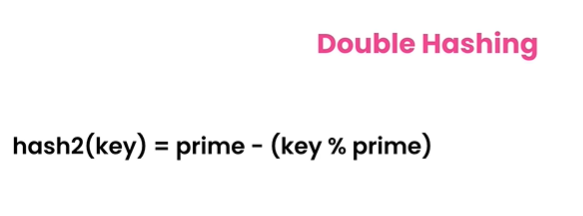
V-(13)-Open Addressing-(method2-Quadratic Probing):-

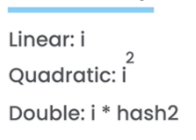
Quadratic Mean Square.

**We just change the formula for finding the address where data will be allocated it is faster than linear probing but it has a problem also. As we take big jumps for finding next position we may ended up in infinite loop for repeating same steps.**

V-(14)-Open Addressing-(method3-Double Hashing):-

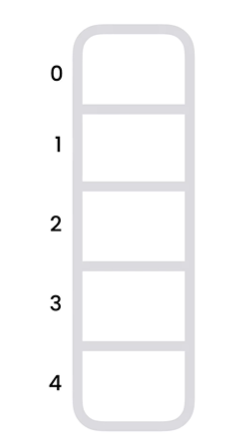
As Quadratic has infinite loop problem we can solve it using double hashing .





**This is the formula in double hashing for finding the index where data should be inserted.**

**This prime number should be less than the size of the array. For ex:- if we have an array of size 5 then prime number can be 1,2,3**.

Now , Let’s see double hash in **Action.**

***Before starting:-***

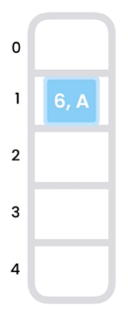
**Double Hashing Formulas:-**

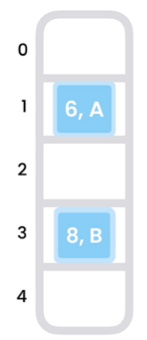
**1. hash1(key) = key % table\_size.**

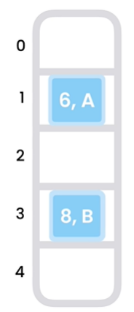
**2. hash2(key) = prime – (key % prime). In our example it should be 3 cause array size is 5 and indices are [0-4]. Between this range we have to select prime numbers. Its ideal to select the largest but less than the range of array could be okay . So we choose 3 as our prime number for double hashing. So our 1 no. formula is hash2(key) = 3 – (key % 3).**

**3. (hash1(key)+i\*hash2(key)) % table\_size.**

**We will use the following two formulas if collison occur.**

Now lets start to insert key value pairs in the given array. If this is the first key-value pair {K=6,V=A} then key % array\_size = 6 % 5 = 1. Then we should insert it in the 1 number index like below:-

Secondly, if key-value pair {K=8,V=B} then key % array\_size = 8 % 5 = 3. Then we should insert it in the 3 number index like below:-

Thirdly, if key-value pair {K=11,V=C} then key % array\_size = 11 % 5 = 1 [using no. 1 formula of double hashing]. As number 1 index has already store a key-value pair {6,A}. That means collision occur. Now we will use double hashing.

Double hashing formulas:-

1. **hash1(key) = key % table\_size.**

2. **hash2(key) = prime – (key % prime). [we already select 3 as prime number for this problem.]**

3. **(hash1(key)+i\*hash2(key)) % table\_size.**

key-value pair {K=11,V=C}

Calculations (after finding collsion):-

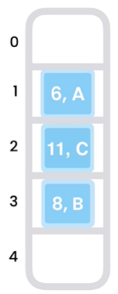
**hash2(11) = 3 – (11 % 3)**

**=> hash2(11) = 3 – (2)**

**=> hash2(11) = 1 [using no. 2 formula]**

**for index finding we will use the second fromula of double hashing.**

**(hash1(11)+i\*hash2(11)) % table\_size**

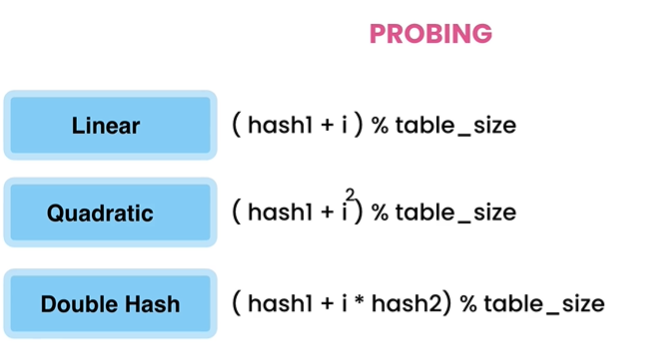
**= (1+(1\*1)) % 5**

**= 2**

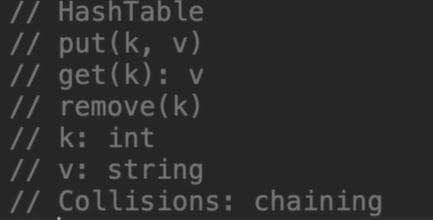
**So, {K=11,V=C} inserted at index 2. It seems that there is a chance of clustering but it will not always like linear probing (search next index for availability).**

Probing Summary of video 12, 13 , 14:-

**Here i is the index where collision occur.**



V-(15)-Build a HashTable-(Popular Interview questions):-



V-(16)-Build a HashTable-(put method):-

import java.util.LinkedList;  
  
public class HashTable {  
 private class Entry{  
 // Entry object will be added in the specified index's LinkedList  
 private int key;  
 private String value;  
 public Entry(int key, String value) {  
 this.key = key;  
 this.value = value;  
 }  
 }  
 private LinkedList<Entry>[] entries = new LinkedList[5];

public void put(int key,String value){  
 var index = hash(key);//get the index  
 if(entries[index]==null){// if no entries happend before in that //specified index  
 entries[index]= new LinkedList<>();// make new LinkedList there  
 }  
 var bucket = entries[index];// as this is use many times (for refactor)  
 for(var entry:bucket){// This loop is only for updating value if same //key found  
 if(entry.key==key){  
 entry.value=value;  
 return;  
 }  
 }  
 // below 2 lines are for always entries with new keys to be added at //the end of the list (list at the specified index)  
 var entry = new Entry(key, value);  
 bucket.addLast(entry);  
 }

private int hash(int key){  
 return Math.*abs*(key% entries.length);// it will //return positive value  
 }  
}

Put the values

1. {K,V}={1,”B”}

2.{K,V}={11,”C”}

3.{K,V}={1,”D”}

0



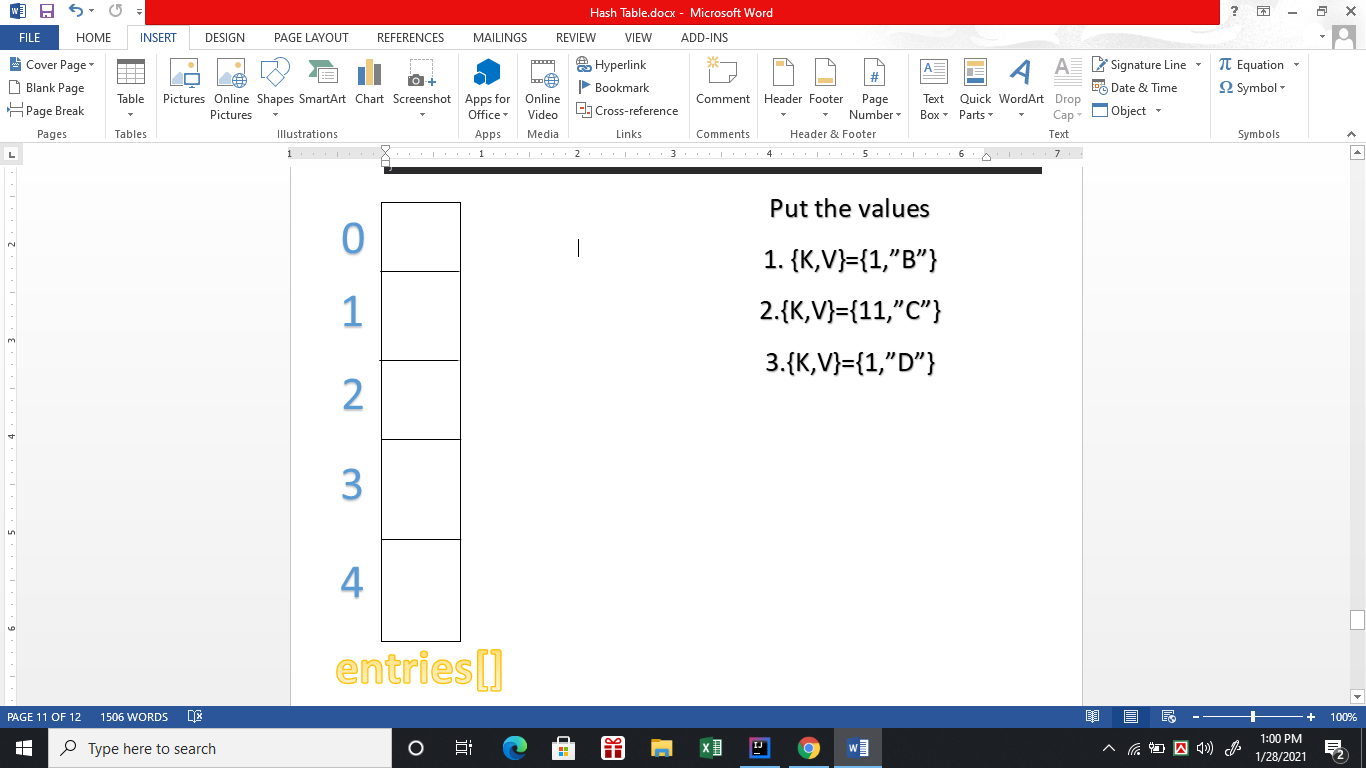
**entries[]**

4

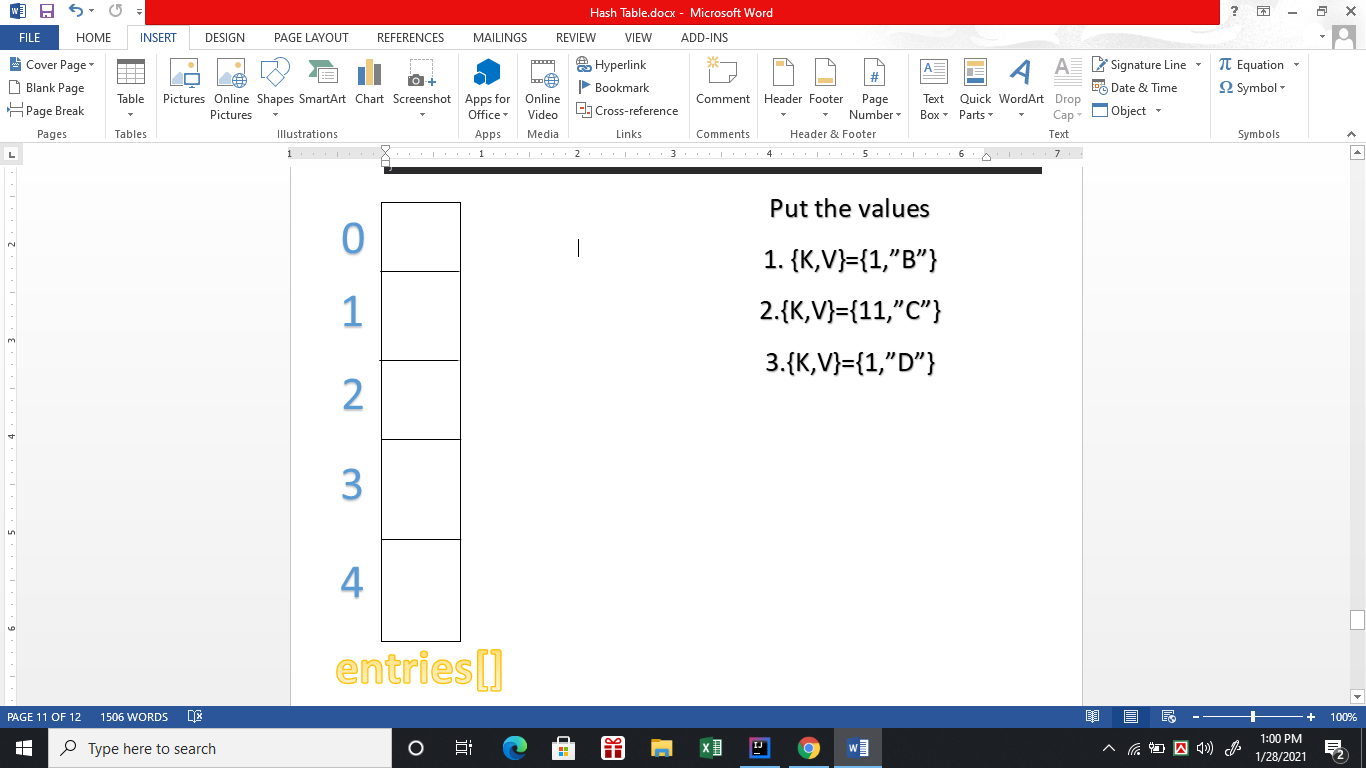
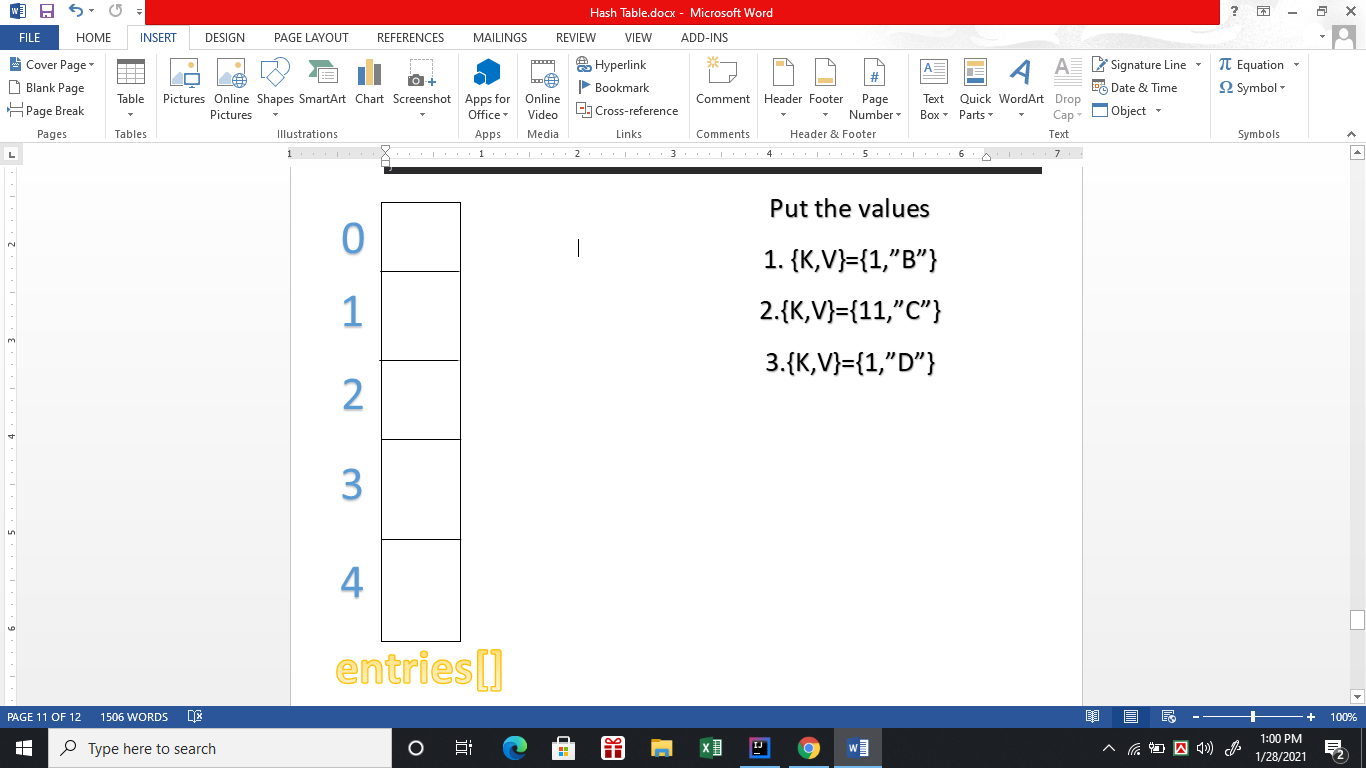
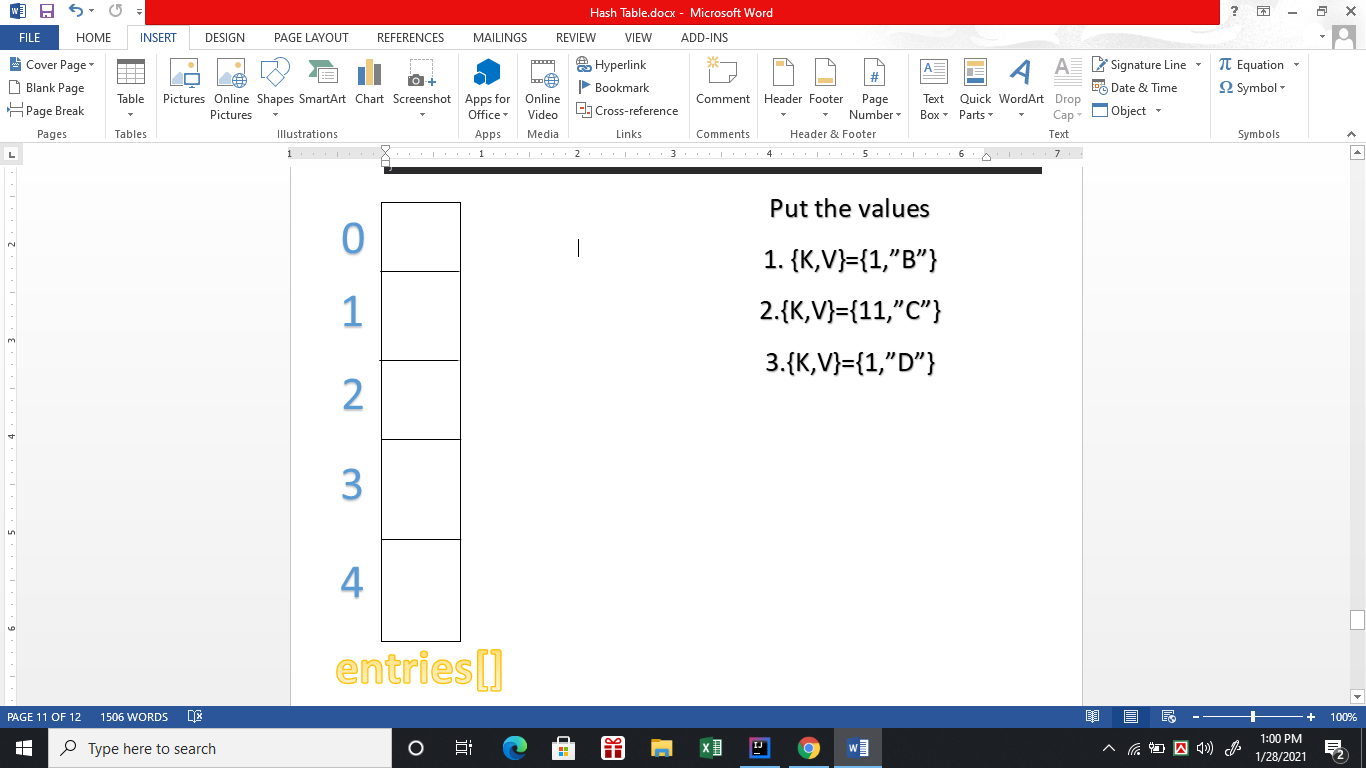
3

2

1





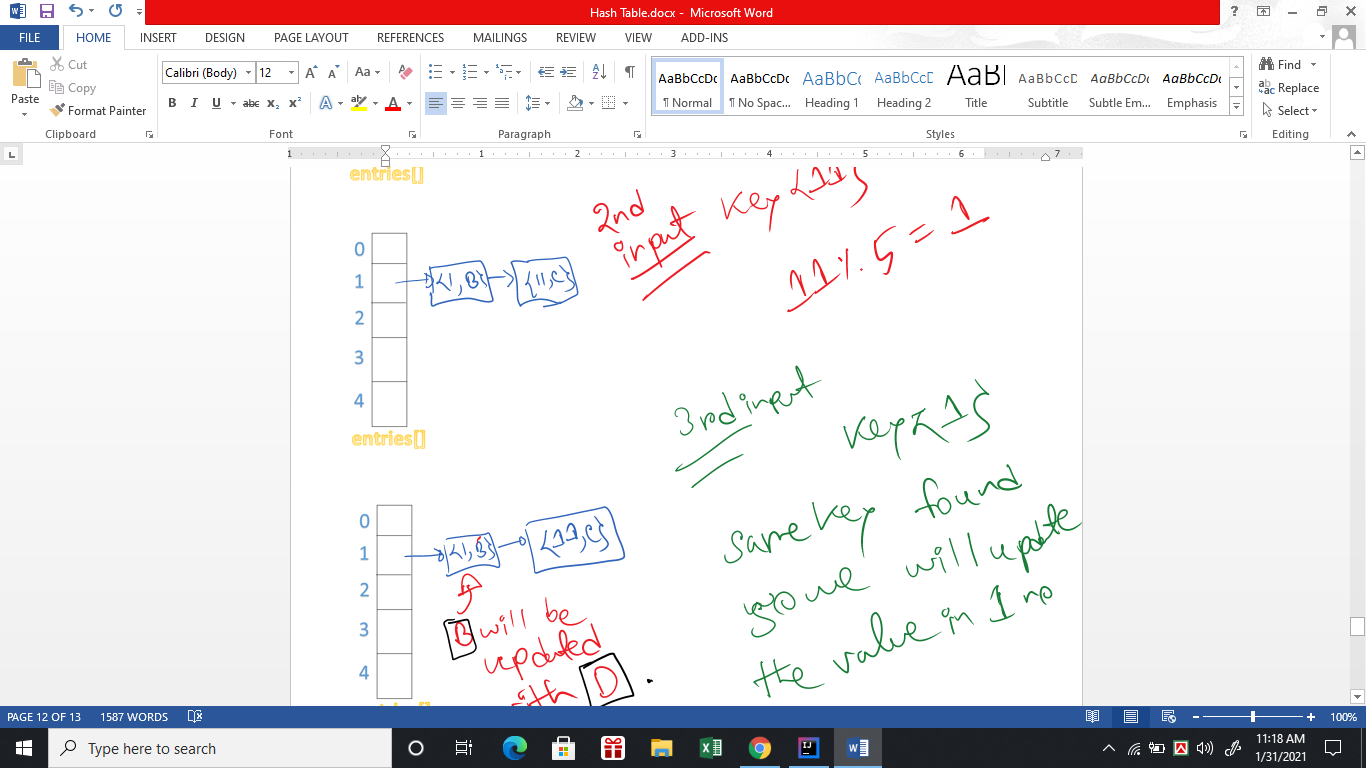




V-(17)-Build a HashTable-(get method):-

public String get(int key){  
var index = hash(key);//Find the specific index for the given key where data may be stored  
var bucket = entries[index];// easy to realise when we //renamed it bucket (also in put)  
if(bucket!=null){//if no Linkedlist is created in the //specific index then we don't need to iterate there  
 for (var entry:bucket){  
 if(entry.key==key)  
 return entry.value;  
 }  
}  
return null;  
}

[N.B- First we have to find the index where keys maybe found in a LinkedList (in the specific index)]



***Suppose we have to to find the value of a key{11}. So , we have to find the index where key may be stored in a linkedlist. So, we found the index using hash method and then iterate the linkedlist to find the given key if found then return the value of that key.***

V-(18)-Build a HashTable-(remove method):-

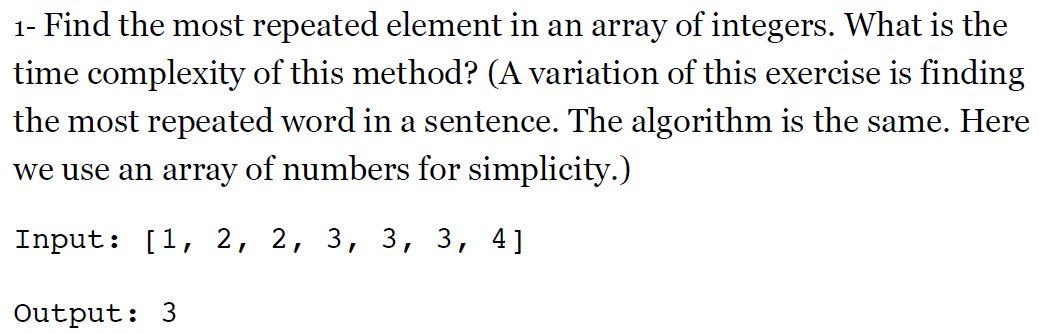
public void remove(int key){  
 var index = hash(key);//get the index  
 var bucket = entries[index];//renamed as bucket so we can iterate and remove the key value pair  
 if(bucket == null)  
 throw new IllegalStateException();  
 for(var entry: bucket){  
 if(entry.key == key){  
 bucket.remove(entry);  
 }  
 }  
 throw new IllegalStateException();// If key is not found anywhere   
}

V-(19)-Build a HashTable-(Refactoring):-

import java.util.LinkedList;  
  
public class HashTable {  
 private class Entry{  
 // Entry object will be added in the specified index's LinkedList  
 private int key;  
 private String value;  
 public Entry(int key, String value) {  
 this.key = key;  
 this.value = value;  
 }  
 }  
 private LinkedList<Entry>[] entries = new LinkedList[5];  
 public void put(int key,String value){  
// var entry = getEntry(key);  
// if(entry!=null){  
// entry.value = value;//updating the value if update needed  
// return;  
// }  
// // below 2 lines will check if bucket need to create or not  
// // if bucket needs to create it will be created  
// // then it will add the entry by calling constructor  
// getOrCreateBucket(key).add(new Entry(key, value));  
  
 var index = hash(key);//get the index  
 if(entries[index]==null){// if no entries happened before in that specified index  
 entries[index]= new LinkedList<>();// make new LinkedList there  
 }  
 var bucket = entries[index];// as this is use many times (for refactor)  
 for(var entry:bucket){// This loop is only for updating value if same key found  
 if(entry.key==key){  
 entry.value=value;  
 return;  
 }  
 }  
 // below line are for always entries with new keys to be added at the end of the list (list at the specified index)  
 bucket.addLast(new Entry(key, value));  
  
 }

public String get(int key){  
 // This below 2 line is very simple  
 // check the entry found or not.  
 // if found , then return the value of the entry , otherwise return null.  
 var entry = getEntry(key);  
 return (entry == null) ? null : entry.value;  
 /\*  
 var index = hash(key);//Find the specific index for the given key where data may be stored  
 var bucket = entries[index];// easy to realize when we renamed it bucket (also in put)  
 if(bucket!=null){//if no Linkedlist is created in the specific index then we don't need to iterate there  
 for (var entry:bucket){  
 if(entry.key==key)  
 return entry.value;  
 }  
 }  
 return null;  
 \*/  
 }  
 private int hash(int key){  
 return Math.*abs*(key% entries.length);// it will return positive value  
 }  
 public void remove(int key){  
 // below 3 lines logic is very simple  
 // 1. find the entry,  
 // 2. return null if entry not found  
 // 3. but if found , find the bucket where the entry exists, then remove the entry from the bucket.  
 var entry = getEntry(key);  
 if(entry == null)  
 throw new IllegalStateException();  
 getBucket(key).remove();  
 /\*  
 // as we refactor and make our code simple , we don't have to write code like below  
 var index = hash(key);//get the index  
 var bucket = entries[index];//renamed as bucket so we can iterate and remove the key value pair  
 if(bucket == null)  
 throw new IllegalStateException();  
 for(var entry: bucket){  
 if(entry.key == key){  
 bucket.remove(entry);  
 return;  
 }  
 }  
 throw new IllegalStateException();// If key is not found anywhere  
 \*/  
 }  
 private LinkedList<Entry> getOrCreateBucket(int key){  
 // find the index using hash key  
 // if no linkedlist is created in the index then create a linkedlist and return.  
 // otherwise it just return the linkedlist that created before.  
 var index = hash(key);  
 var bucket = entries[index];  
 if(bucket==null)  
 entries[index] = new LinkedList<>();  
  
 return bucket;  
 }  
 private LinkedList<Entry> getBucket(int key){  
 /\*var index = hash(key);// storing the index  
 var bucket = entries[index];  
 return bucket;  
 \*/  
 // for upper logic below one line is enough  
 return entries[hash(key)];  
 //method return type is LinkedList<Entry> cause in the index data stored in linkedlist and class type is Entry.  
 }  
 private Entry getEntry(int key){  
 /\*  
 var index = hash(key);// storing the index  
 var bucket = entries[index];// easier to understand  
 // as we developed getBucket method we do not have to write code like this  
 \*/  
 var bucket = getBucket(key);//find the linkedlist where key,value pair is stored.  
 if(bucket !=null){// if bucket is not null then we have to iterate through the bucket (list) , if found as same key then return the entry  
 for(var entry:bucket){  
 if (entry.key == key)  
 return entry;  
 }  
 }  
 return null;// if no such entry with the give key found, then only return null  
 }  
}

Mosh Exercises:-

Ex-1)

Ans:-

// O(n)

public int mostFrequent(int[] numbers) {

// To find the most frequent item in an array, we have to count the

// number of times each item has been repeated. We can use a hash

// table to store the items and their frequencies.

Map<Integer, Integer> map = new HashMap<>();

for (var number : numbers) {

var count = map.getOrDefault(number, 0);

map.put(number, count + 1);

}

// Once we've populated our hash table, we need to iterate over all

// key/value pairs and find the one with the highest frequency.

int max = -1;

int result = numbers[0];

for (var item : map.entrySet()) {

if (item.getValue() > max) {

max = item.getValue();

result = item.getKey();

}

}

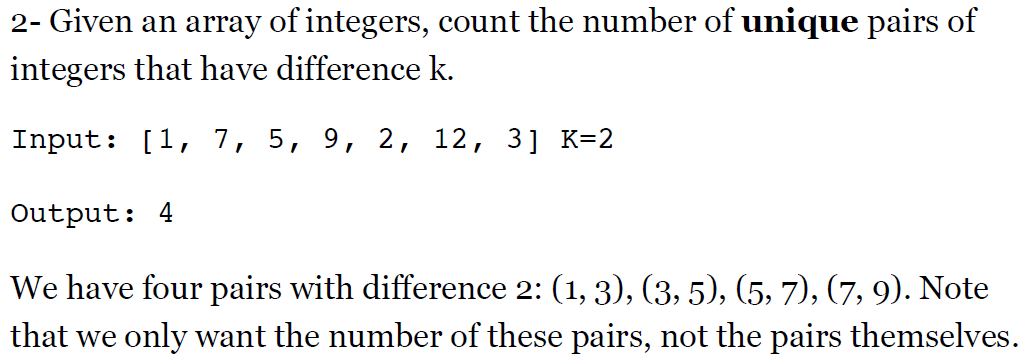
// Runtime complexity of this method is O(n) because we have to

// iterate the entire array to populate our hash table.

return result;

}

Ex-2)



Answer:-

// O(n)  
public static int countPairsWithDiff(int[] numbers, int difference) {  
 // For a given number (a) and difference (diff), number (b) can be:  
 //  
 // b = a + diff  
 // b = a - diff  
 //  
 // We can iterate over our array of numbers, and for each number,  
 // check to see if we have (current + diff) or (current - diff).  
 // But looking up items in an array is an O(n) operation. With this  
 // algorithm, we need two nested loops (one to pick a,  
 // and the other to find b). This will be an O(n^2) operation.  
 //  
 // We can optimize this by using a set. Sets are like hash tables  
 // but they only store keys. We can look up a number in constant time.  
 // No need to iterate the array to find it.  
  
 // So, we start by adding all the numbers to a set for quick look up.  
 Set<Integer> set = new HashSet<>();  
 for (var number : numbers)  
 set.add(number);  
  
 // Now, we iterate over the array of numbers one more time,  
 // and for each number check to see if we have (a + diff) or  
 // (a - diff) in our set.  
 //  
 // Once we're done, we should remove this number from our set  
 // so we don't double count it.  
 var count = 0;

for (var number : numbers) {  
 if (set.contains(number + difference))  
 count++;  
 if (set.contains(number - difference))  
 count++;  
 set.remove(number);  
 }  
  
 // Time complexity of this method is O(n).  
  
 return count;  
}

Ex-3)

Given an array of integers, return **indices** of the two numbers such

that they add up to a specific target.

Input: [2, 7, 11, 15] - target = 9

Output: [0, 1] (because 2 + 7 = 9)

Assume that each input has **exactly** one solution, and you may not use

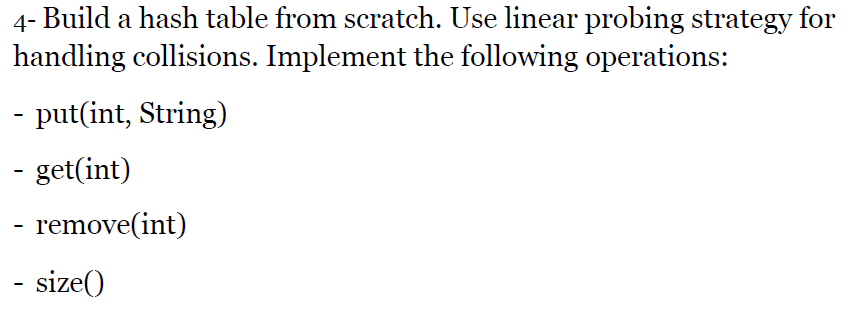
the *same* element twice.

Answer:-

// O(n)  
public static int[] twoSum(int[] numbers, int target) {  
 // This problem is a variation of the previous problem  
 // (countPairsWithDiff).  
 //  
 // If a + b = target, then b = target - a.  
 //  
 // So we iterate our array, and pick (a). Then,  
 // we check to see if we have (b) in our array.  
 // Similar to the last problem, this would be an O(n^2)  
 // operation, because we'll need two nested loops for  
 // looking up (b).  
 //  
 // We can optimize this by using a hash table. In this  
 // hash table, we store numbers and their indexes.  
 //  
 // There is no need to store all the numbers in the hash table  
 // first. If we find two numbers that add up to the target,  
 // we simply return their indexes.  
  
 Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < numbers.length; i++) {  
 int complement = target - numbers[i];  
 if (map.containsKey(complement)) {  
 return new int[] { map.get(complement), i };  
 }  
 map.put(numbers[i], i);  
 }  
  
 // Time complexity of this method is O(n) because we need to iterate  
 // the array only once.  
  
 return null;  
}

Ex-4)



Answer:-

import java.util.Arrays;

public class HashMap {

private class Entry {

private int key;

private String value;

public Entry(int key, String value) {

this.key = key;

this.value = value;

}

@Override

public String toString() {

return value;

}

}

private Entry[] entries = new Entry[5];

private int count;

public void put(int key, String value) {

var entry = getEntry(key);

if (entry != null) {

entry.value = value;

return;

}

if (isFull())

throw new IllegalStateException();

entries[getIndex(key)] = new Entry(key, value);

count++;

}

public String get(int key) {

var entry = getEntry(key);

return entry != null ? entry.value : null;

}

public void remove(int key) {

var index = getIndex(key);

if (index == -1 || entries[index] == null)

return;

entries[index] = null;

count--;

}

public int size() {

return count;

}

private Entry getEntry(int key) {

var index = getIndex(key);

return index >= 0 ? entries[index] : null;

}

private int getIndex(int key) {

int steps = 0;

// Linear probing algorithm: we keep looking until we find an empty

// slot or a slot with the same key.

// We use this loop conditional to prevent an infinite loop that

// will happen if the array is full and we keep probing with no

// success. So, the number of steps (or probing attempts) should

// be less than the size of our table.

while (steps < entries.length) {

int index = index(key, steps++);

var entry = entries[index];

if (entry == null || entry.key == key)

return index;

}

// This will happen if we looked at every slot in the array

// and couldn't find a place for this key. That basically means

// the table is full.

return -1;

}

private boolean isFull() {

return count == entries.length;

}

private int index(int key, int i) {

return (hash(key) + i) % entries.length;

}

private int hash(int key) {

return key % entries.length;

}

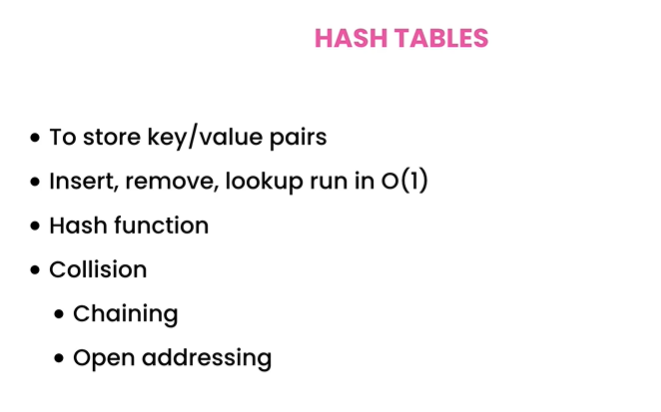
@Override

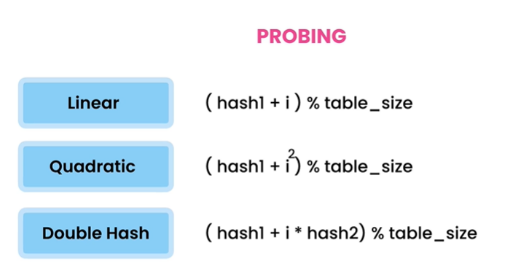
public String toString() {

return Arrays.toString(entries);

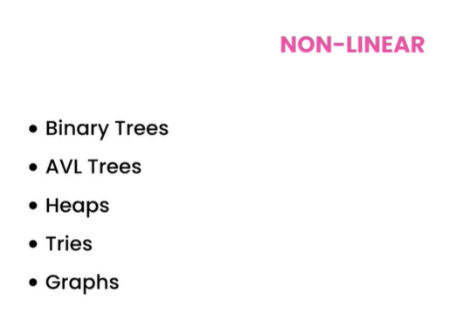
}

}

**Summary:-**







**Part 2**