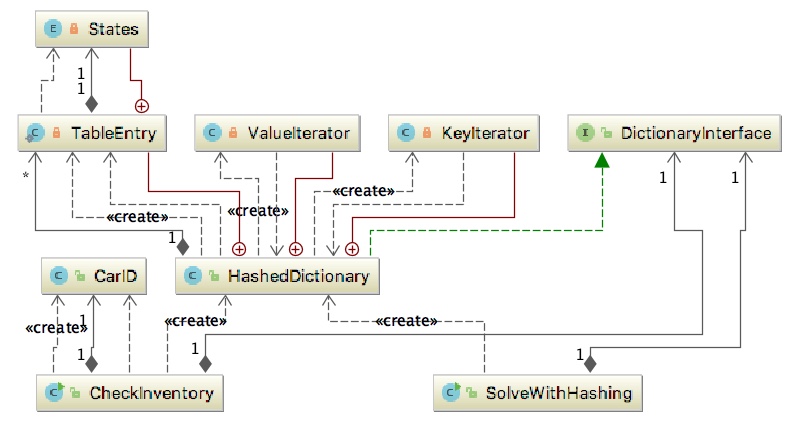
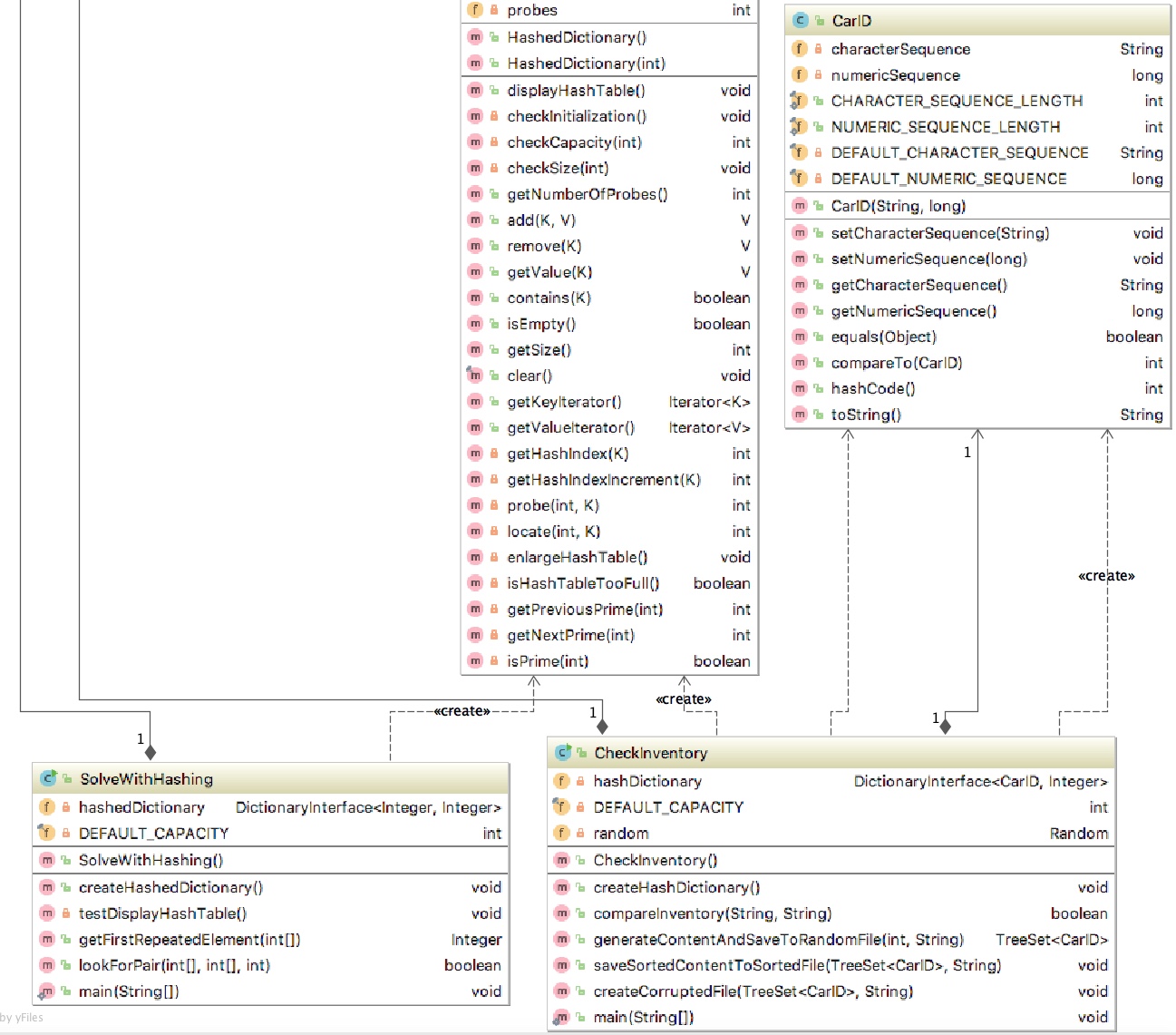
# Comp151 Lab12

The dictionary ADT is a set of associations between two items: the key and the value. Given a key you can find its value. There are a number of ways you could implement the dictionary ADT. In this lab a hash table is used to implement the dictionary. The details of how hash tables work are also considered in this lab. To start with, the important implication of that fact is that they allow fast access and that the items in the hash table are **not ordered** sequentially by their keys.

Your task is to implement two applications: SolveWithHashing (Problem #1 and Problem #2) and CheckInventory (Part1 and Part2) as described. See the UML diagram below:

****

****

# SolveWithHashing

### Problem #1

Given an array of n positive numbers, give an algorithm for finding the first element in the array which is repeated (may be repeated more than once). For example, in the array {0,3,2,1,2,2,3} the first repeated number is 3 (not 2). Utilize hashing where the **key** is the elementand the **value** is the position of the element in the array. Initially we store the position of the element in the array as the value for this key. If we get the same element again however, we just negate the current value in the hash table. The negative value in the hash table indicates that we have seen the same element more than once. After processing the complete input array, we scan the hash table with the values iterator and return the highest negative indexed value from it. The highest negative value indicates that we have seen that element first among repeated elements.

**Step1:** You will utilize HashedDictionary class that implements DictionaryInterface. Make yourself familiar with the class implementation. Next, implement displayHashTable() method in this class. SolveWithHashing class has testDisplayHashTable method you will use to test it.

**Step2:** Implement the algorithm in getFirstRepeatedElement method. Note: the algorithm assumes that recorded positions are 1-based, for example the element at index 0 has position 1.

### Problem #2

Given two arrays *a* and *b*, and a number *k*, consider the following algorithm for finding whether there exists a pair of elements, one of them from *a* and one from *b*, that add up to *k*.

* Select the set which has the smaller number of elements (swap the reference variables if needed)
* For the selected set create a hash table (you can use the element for the key and the value)
* Now scan the second array and check whether *k-selected element* exist in the hash table or not
  + HINT look for a key that is computed as a delta between k and the current element
* If it exists, then return the pair of elements
* Otherwise continue until we reach the end of the set.

Implement this algorithm in lookForPair method.

# Sample Run:

**\*\*\* Testing displayHashTable \*\*\***

**displaying empty dictionary**

**The size of hash table is: 5**

**0 null**

**1 null**

**2 null**

**3 null**

**4 null**

**displaying dictionary after 2 entries have been added**

**The size of hash table is: 5**

**0 null**

**1 KEY: 1 VALUE: 1**

**2 KEY: 7 VALUE: 7**

**3 null**

**4 null**

**displaying dictionary after 3 additional entries have been added**

**The size of hash table is: 11**

**0 null**

**1 KEY: 1 VALUE: 1**

**2 KEY: 13 VALUE: 13**

**3 null**

**4 null**

**5 null**

**6 KEY: 17 VALUE: 17**

**7 KEY: 7 VALUE: 7**

**8 KEY: 8 VALUE: 8**

**9 null**

**10 null**

**displaying dictionary after 2 entries have been removed**

**The size of hash table is: 11**

**0 null**

**1 notIn**

**2 KEY: 13 VALUE: 13**

**3 null**

**4 null**

**5 null**

**6 notIn**

**7 KEY: 7 VALUE: 7**

**8 KEY: 8 VALUE: 8**

**9 null**

**10 null**

**\*\*\* Find The First Element With Duplicate \*\*\***

**The content of the hash table for array: [9, 3, 5, 1, 2, 2, 5, 3]**

**The size of hash table is: 11**

**0 null**

**1 KEY: 1 VALUE: 4**

**2 KEY: 2 VALUE: -5**

**3 KEY: 3 VALUE: -2**

**4 null**

**5 KEY: 5 VALUE: -3**

**6 null**

**7 null**

**8 null**

**9 KEY: 9 VALUE: 1**

**10 null**

**--> the first element that is repeated is: 3**

**The content of the hash table for array: [6, 6, 3, 2, 1, 2, 2, 3]**

**The size of hash table is: 11**

**0 null**

**1 KEY: 1 VALUE: 5**

**2 KEY: 2 VALUE: -4**

**3 KEY: 3 VALUE: -3**

**4 null**

**5 null**

**6 KEY: 6 VALUE: -1**

**7 null**

**8 null**

**9 null**

**10 null**

**--> the first element that is repeated is: 6**

**The content of the hash table for array: [2, 1, 6, 2, 3, 2, 3, 6]**

**The size of hash table is: 11**

**0 null**

**1 KEY: 1 VALUE: 2**

**2 KEY: 2 VALUE: -1**

**3 KEY: 3 VALUE: -5**

**4 null**

**5 null**

**6 KEY: 6 VALUE: -3**

**7 null**

**8 null**

**9 null**

**10 null**

**--> the first element that is repeated is: 2**

**The content of the hash table for array: [3, 2, 1, 2, 2, 3, 6, 6]**

**The size of hash table is: 11**

**0 null**

**1 KEY: 1 VALUE: 3**

**2 KEY: 2 VALUE: -2**

**3 KEY: 3 VALUE: -1**

**4 null**

**5 null**

**6 KEY: 6 VALUE: -7**

**7 null**

**8 null**

**9 null**

**10 null**

**--> the first element that is repeated is: 3**

**The content of the hash table for array: [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The size of hash table is: 23**

**0 null**

**1 KEY: 1 VALUE: 6**

**2 KEY: 2 VALUE: -7**

**3 KEY: 3 VALUE: -2**

**4 KEY: 4 VALUE: -3**

**5 KEY: 5 VALUE: 9**

**6 null**

**7 null**

**8 null**

**9 KEY: 9 VALUE: 1**

**10 null**

**11 null**

**12 null**

**13 null**

**14 null**

**15 null**

**16 null**

**17 null**

**18 null**

**19 null**

**20 null**

**21 null**

**22 null**

**--> the first element that is repeated is: 3**

**The content of the hash table for array: [3, 3, 3, 3, 3, 3, 3]**

**The size of hash table is: 5**

**0 null**

**1 null**

**2 null**

**3 KEY: 3 VALUE: -1**

**4 null**

**--> the first element that is repeated is: 3**

**The content of the hash table for array: [1, 2, 3, 4, 5, 6, 7, 8]**

**The size of hash table is: 23**

**0 null**

**1 KEY: 1 VALUE: 1**

**2 KEY: 2 VALUE: 2**

**3 KEY: 3 VALUE: 3**

**4 KEY: 4 VALUE: 4**

**5 KEY: 5 VALUE: 5**

**6 KEY: 6 VALUE: 6**

**7 KEY: 7 VALUE: 7**

**8 KEY: 8 VALUE: 8**

**9 null**

**10 null**

**11 null**

**12 null**

**13 null**

**14 null**

**15 null**

**16 null**

**17 null**

**18 null**

**19 null**

**20 null**

**21 null**

**22 null**

**--> duplicates not found**

**The content of the hash table for array: [8, 1, 2, 3, 4, 5, 6, 7]**

**The size of hash table is: 23**

**0 null**

**1 KEY: 1 VALUE: 2**

**2 KEY: 2 VALUE: 3**

**3 KEY: 3 VALUE: 4**

**4 KEY: 4 VALUE: 5**

**5 KEY: 5 VALUE: 6**

**6 KEY: 6 VALUE: 7**

**7 KEY: 7 VALUE: 8**

**8 KEY: 8 VALUE: 1**

**9 null**

**10 null**

**11 null**

**12 null**

**13 null**

**14 null**

**15 null**

**16 null**

**17 null**

**18 null**

**19 null**

**20 null**

**21 null**

**22 null**

**--> duplicates not found**

**\*\*\* Check If There Exists A Pair Of Elements That Add Up To k \*\*\***

**k = 2**

**toPutInHashTable = [9, 3, 5, 1, 2, 2, 5, 3]**

**toCheck = [6, 6, 3, 2, 1, 2, 2, 3]**

**The pair {1,1} adds to 2**

**--> pair that add up to 2 found.**

**toPutInHashTable = [6, 6, 3, 2, 1, 2, 2, 3]**

**toCheck = [2, 1, 6, 2, 3, 2, 3, 6]**

**The pair {1,1} adds to 2**

**--> pair that add up to 2 found.**

**toPutInHashTable = [2, 1, 6, 2, 3, 2, 3, 6]**

**toCheck = [3, 2, 1, 2, 2, 3, 6, 6]**

**The pair {1,1} adds to 2**

**--> pair that add up to 2 found.**

**toPutInHashTable = [3, 2, 1, 2, 2, 3, 6, 6]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {1,1} adds to 2**

**--> pair that add up to 2 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**--> pair that add up to 2 NOT found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [1, 2, 3, 4, 5, 6, 7, 8]**

**--> pair that add up to 2 NOT found.**

**toPutInHashTable = [1, 2, 3, 4, 5, 6, 7, 8]**

**toCheck = [8, 1, 2, 3, 4, 5, 6, 7]**

**The pair {1,1} adds to 2**

**--> pair that add up to 2 found.**

**k = 3**

**toPutInHashTable = [9, 3, 5, 1, 2, 2, 5, 3]**

**toCheck = [6, 6, 3, 2, 1, 2, 2, 3]**

**The pair {1,2} adds to 3**

**--> pair that add up to 3 found.**

**toPutInHashTable = [6, 6, 3, 2, 1, 2, 2, 3]**

**toCheck = [2, 1, 6, 2, 3, 2, 3, 6]**

**The pair {1,2} adds to 3**

**--> pair that add up to 3 found.**

**toPutInHashTable = [2, 1, 6, 2, 3, 2, 3, 6]**

**toCheck = [3, 2, 1, 2, 2, 3, 6, 6]**

**The pair {1,2} adds to 3**

**--> pair that add up to 3 found.**

**toPutInHashTable = [3, 2, 1, 2, 2, 3, 6, 6]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {2,1} adds to 3**

**--> pair that add up to 3 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**--> pair that add up to 3 NOT found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [1, 2, 3, 4, 5, 6, 7, 8]**

**--> pair that add up to 3 NOT found.**

**toPutInHashTable = [1, 2, 3, 4, 5, 6, 7, 8]**

**toCheck = [8, 1, 2, 3, 4, 5, 6, 7]**

**The pair {2,1} adds to 3**

**--> pair that add up to 3 found.**

**k = 4**

**toPutInHashTable = [9, 3, 5, 1, 2, 2, 5, 3]**

**toCheck = [6, 6, 3, 2, 1, 2, 2, 3]**

**The pair {1,3} adds to 4**

**--> pair that add up to 4 found.**

**toPutInHashTable = [6, 6, 3, 2, 1, 2, 2, 3]**

**toCheck = [2, 1, 6, 2, 3, 2, 3, 6]**

**The pair {2,2} adds to 4**

**--> pair that add up to 4 found.**

**toPutInHashTable = [2, 1, 6, 2, 3, 2, 3, 6]**

**toCheck = [3, 2, 1, 2, 2, 3, 6, 6]**

**The pair {1,3} adds to 4**

**--> pair that add up to 4 found.**

**toPutInHashTable = [3, 2, 1, 2, 2, 3, 6, 6]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {1,3} adds to 4**

**--> pair that add up to 4 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {3,1} adds to 4**

**--> pair that add up to 4 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [1, 2, 3, 4, 5, 6, 7, 8]**

**The pair {3,1} adds to 4**

**--> pair that add up to 4 found.**

**toPutInHashTable = [1, 2, 3, 4, 5, 6, 7, 8]**

**toCheck = [8, 1, 2, 3, 4, 5, 6, 7]**

**The pair {3,1} adds to 4**

**--> pair that add up to 4 found.**

**k = 5**

**toPutInHashTable = [9, 3, 5, 1, 2, 2, 5, 3]**

**toCheck = [6, 6, 3, 2, 1, 2, 2, 3]**

**The pair {2,3} adds to 5**

**--> pair that add up to 5 found.**

**toPutInHashTable = [6, 6, 3, 2, 1, 2, 2, 3]**

**toCheck = [2, 1, 6, 2, 3, 2, 3, 6]**

**The pair {3,2} adds to 5**

**--> pair that add up to 5 found.**

**toPutInHashTable = [2, 1, 6, 2, 3, 2, 3, 6]**

**toCheck = [3, 2, 1, 2, 2, 3, 6, 6]**

**The pair {2,3} adds to 5**

**--> pair that add up to 5 found.**

**toPutInHashTable = [3, 2, 1, 2, 2, 3, 6, 6]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {2,3} adds to 5**

**--> pair that add up to 5 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {3,2} adds to 5**

**--> pair that add up to 5 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [1, 2, 3, 4, 5, 6, 7, 8]**

**The pair {3,2} adds to 5**

**--> pair that add up to 5 found.**

**toPutInHashTable = [1, 2, 3, 4, 5, 6, 7, 8]**

**toCheck = [8, 1, 2, 3, 4, 5, 6, 7]**

**The pair {4,1} adds to 5**

**--> pair that add up to 5 found.**

**k = 6**

**toPutInHashTable = [9, 3, 5, 1, 2, 2, 5, 3]**

**toCheck = [6, 6, 3, 2, 1, 2, 2, 3]**

**The pair {3,3} adds to 6**

**--> pair that add up to 6 found.**

**toPutInHashTable = [6, 6, 3, 2, 1, 2, 2, 3]**

**toCheck = [2, 1, 6, 2, 3, 2, 3, 6]**

**The pair {3,3} adds to 6**

**--> pair that add up to 6 found.**

**toPutInHashTable = [2, 1, 6, 2, 3, 2, 3, 6]**

**toCheck = [3, 2, 1, 2, 2, 3, 6, 6]**

**The pair {3,3} adds to 6**

**--> pair that add up to 6 found.**

**toPutInHashTable = [3, 2, 1, 2, 2, 3, 6, 6]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {3,3} adds to 6**

**--> pair that add up to 6 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {3,3} adds to 6**

**--> pair that add up to 6 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [1, 2, 3, 4, 5, 6, 7, 8]**

**The pair {3,3} adds to 6**

**--> pair that add up to 6 found.**

**toPutInHashTable = [1, 2, 3, 4, 5, 6, 7, 8]**

**toCheck = [8, 1, 2, 3, 4, 5, 6, 7]**

**The pair {5,1} adds to 6**

**--> pair that add up to 6 found.**

**k = 7**

**toPutInHashTable = [9, 3, 5, 1, 2, 2, 5, 3]**

**toCheck = [6, 6, 3, 2, 1, 2, 2, 3]**

**The pair {1,6} adds to 7**

**--> pair that add up to 7 found.**

**toPutInHashTable = [6, 6, 3, 2, 1, 2, 2, 3]**

**toCheck = [2, 1, 6, 2, 3, 2, 3, 6]**

**The pair {6,1} adds to 7**

**--> pair that add up to 7 found.**

**toPutInHashTable = [2, 1, 6, 2, 3, 2, 3, 6]**

**toCheck = [3, 2, 1, 2, 2, 3, 6, 6]**

**The pair {6,1} adds to 7**

**--> pair that add up to 7 found.**

**toPutInHashTable = [3, 2, 1, 2, 2, 3, 6, 6]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {3,4} adds to 7**

**--> pair that add up to 7 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {3,4} adds to 7**

**--> pair that add up to 7 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [1, 2, 3, 4, 5, 6, 7, 8]**

**The pair {3,4} adds to 7**

**--> pair that add up to 7 found.**

**toPutInHashTable = [1, 2, 3, 4, 5, 6, 7, 8]**

**toCheck = [8, 1, 2, 3, 4, 5, 6, 7]**

**The pair {6,1} adds to 7**

**--> pair that add up to 7 found.**

**k = 8**

**toPutInHashTable = [9, 3, 5, 1, 2, 2, 5, 3]**

**toCheck = [6, 6, 3, 2, 1, 2, 2, 3]**

**The pair {2,6} adds to 8**

**--> pair that add up to 8 found.**

**toPutInHashTable = [6, 6, 3, 2, 1, 2, 2, 3]**

**toCheck = [2, 1, 6, 2, 3, 2, 3, 6]**

**The pair {6,2} adds to 8**

**--> pair that add up to 8 found.**

**toPutInHashTable = [2, 1, 6, 2, 3, 2, 3, 6]**

**toCheck = [3, 2, 1, 2, 2, 3, 6, 6]**

**The pair {6,2} adds to 8**

**--> pair that add up to 8 found.**

**toPutInHashTable = [3, 2, 1, 2, 2, 3, 6, 6]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {6,2} adds to 8**

**--> pair that add up to 8 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {3,5} adds to 8**

**--> pair that add up to 8 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [1, 2, 3, 4, 5, 6, 7, 8]**

**The pair {3,5} adds to 8**

**--> pair that add up to 8 found.**

**toPutInHashTable = [1, 2, 3, 4, 5, 6, 7, 8]**

**toCheck = [8, 1, 2, 3, 4, 5, 6, 7]**

**The pair {7,1} adds to 8**

**--> pair that add up to 8 found.**

**k = 9**

**toPutInHashTable = [9, 3, 5, 1, 2, 2, 5, 3]**

**toCheck = [6, 6, 3, 2, 1, 2, 2, 3]**

**The pair {3,6} adds to 9**

**--> pair that add up to 9 found.**

**toPutInHashTable = [6, 6, 3, 2, 1, 2, 2, 3]**

**toCheck = [2, 1, 6, 2, 3, 2, 3, 6]**

**The pair {3,6} adds to 9**

**--> pair that add up to 9 found.**

**toPutInHashTable = [2, 1, 6, 2, 3, 2, 3, 6]**

**toCheck = [3, 2, 1, 2, 2, 3, 6, 6]**

**The pair {6,3} adds to 9**

**--> pair that add up to 9 found.**

**toPutInHashTable = [3, 2, 1, 2, 2, 3, 6, 6]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**The pair {6,3} adds to 9**

**--> pair that add up to 9 found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [9, 3, 4, 4, 4, 1, 2, 2, 5, 3]**

**--> pair that add up to 9 NOT found.**

**toPutInHashTable = [3, 3, 3, 3, 3, 3, 3]**

**toCheck = [1, 2, 3, 4, 5, 6, 7, 8]**

**The pair {3,6} adds to 9**

**--> pair that add up to 9 found.**

**toPutInHashTable = [1, 2, 3, 4, 5, 6, 7, 8]**

**toCheck = [8, 1, 2, 3, 4, 5, 6, 7]**

**The pair {1,8} adds to 9**

**--> pair that add up to 9 found.**

**Bye!**

**Process finished with exit code 0**

# CheckInventory Application

**Part 1 - Implement CheckInventory Application**

Port Hueneme receives a shipment of cars. Each car has a distinct id that consists of 3 letters in upper case (represented by a String), followed by a space, followed by 14 digits (represented by a long). As the cars are being unloaded from the ship their IDs are recorded and placed in sorted order in a file. The file is sent to the dealer. The cars are driven to the dealership by designated drivers and arrive at the destination in random order. The car IDs are recorded at the dealership. Your task is to compare the files and detect if they contain the same elements utilizing hashing technique with O(n) complexity.

For example, the following two arrays have the same set of elements (notice that duplicates are allowed):

A = {2,5,6,8,10,2,2}

B = {2,5,5,8,10,5,6}

To check if they have the same elements with hash table, consider the following algorithm:

* Construct the hash table with array A elements as keys
* While inserting the elements, keep track of frequency for each number
* After constructing the hash table for A’s elements, now scan the array B
* For each occurrence of B’s elements reduce the corresponding counter values
* At the end, check whether all counters are zero or not
* If all counters are zero, then both arrays are the same otherwise they are different.
* Think and implement also the scenarios where the checking process should stop earlier

Using the above algorithm as a guide, design an algorithm to check if two files contain the same elements (see UML diagram).

CarID class must implement Comparable interface in order to put CarID objects into a TreeSet:

* Please note that the equals method accepts Object as the input
* For the hashCode method use the following formula:

G \* (hash code for this.characterSequence) + (integer result of folding this.numericSequence)

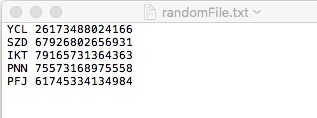
where G is 31

CheckInventory class has main method implemented. You need to implement the remaining methods:

* generateContentAndSaveToRandomFile method: generates the car IDs randomly and saves them in a TreeSet<CarID> to ensure that they are distinct. At the same time when it is known that they are distinct writes them into randomFile.txt.
* saveSortedContentToSortedFile method: writes the content of the TreeSet<CarID> returned by the above method to the sortedFile.txt.
* createCorruptedFile method: also writes the content of the TreeSet<CarID> to a file, but this time the name of the file is corruptedFile.txt. The method randomly decides which elements should be skipped in order to produce a file that will not have the same elements as the original file.
* compareInventory method: performs the comparison using **optimized** version of the above algorithm.
  + must throw InputMismatchException in case duplicates in the sent file are found

Utilize PrintWriter to write to a text file; utilize Scanner to read from a file.

The following shows the content of a sample randomFile.txt, I used Random generator with the seed of 101:



# Part 2 – Change Hashing Mechanism

Once the application works correctly move to part 2 that has to do with analyzing the hashing mechanism itself.

One of the faster dictionary implementations is the hash table. As long as the table does not become too full, the time for adding and finding an element will be O(1). This performance does not come without some cost. The obvious penalty is that there will be space in the table that is wasted. Another penalty is that the items in the hash table are not in any particular order. As more items are added to the hash table, the size of the table may be increased to maintain the performance. In this case, the items will be rehashed and will no longer be in the same locations or order. To observe this behavior, make changes in HashedDictionary class necessary to change the open addressing from *Linear Probing* mechanism to *Double Hashing*.

1. First implement getHashIndexIncrement method that calculates and returns the step increment for the given key to be used when probing for element inside the locate and probe methods. Use the following formula to calculate the initial value of the increment: step = (previousPrime - key.hashCode()) % previousPrime; If the calculated step value is zero set it to the provided DEFAULT\_PRIME. If the calculated step is negative, make it positive by adding the hashTable.length to it.
2. Change probe and locate to use *Double Hashing* instead of the *Linear Probing* currently in place.

# Sample Runs:

### For Part1 with HashedDictionary utilizing *Linear Probing*:

How many CarIDs to generate?

5

\*\*\* Checking if "sortedFile.txt" and "randomFile.txt" have the same elements \*\*\*

The content of the hash table after file randomFile.txt was processed:

The size of hash table is: 11

0 null

1 null

2 KEY: CarID{YCL 26173488024166} VALUE: 1

3 KEY: CarID{IKT 79165731364363} VALUE: 1

4 KEY: CarID{PNN 75573168975558} VALUE: 1

5 KEY: CarID{PFJ 61745334134984} VALUE: 1

6 null

7 null

8 null

9 KEY: CarID{SZD 67926802656931} VALUE: 1

10 null

The content of the hash table after file sortedFile.txt was processed:

The size of hash table is: 11

0 null

1 null

2 notIn

3 notIn

4 notIn

5 notIn

6 null

7 null

8 null

9 notIn

10 null

--> the elements in files "randomFile.txt" and "sortedFile.txt are the same

\*\*\* Checking if "sortedFile.txt" and "corruptedFile.txt" have the same elements \*\*\*

The content of the hash table after file sortedFile.txt was processed:

The size of hash table is: 11

0 null

1 null

2 KEY: CarID{YCL 26173488024166} VALUE: 1

3 KEY: CarID{PNN 75573168975558} VALUE: 1

4 KEY: CarID{IKT 79165731364363} VALUE: 1

5 KEY: CarID{PFJ 61745334134984} VALUE: 1

6 null

7 null

8 null

9 KEY: CarID{SZD 67926802656931} VALUE: 1

10 null

The content of the hash table after file corruptedFile.txt was processed:

The size of hash table is: 11

0 null

1 null

2 notIn

3 KEY: CarID{PNN 75573168975558} VALUE: 1

4 KEY: CarID{IKT 79165731364363} VALUE: 1

5 KEY: CarID{PFJ 61745334134984} VALUE: 1

6 null

7 null

8 null

9 KEY: CarID{SZD 67926802656931} VALUE: 1

10 null

--> the elements in files "corruptedFile.txt" and "sortedFile.txt are NOT the same

Bye!

Process finished with exit code 0

1. **For Part2 with HashedDictionary utilizing *Double Hashing*:**

How many CarIDs to generate?

5

\*\*\* Checking if "sortedFile.txt" and "randomFile.txt" have the same elements \*\*\*

The content of the hash table after file randomFile.txt was processed:

The size of hash table is: 11

0 KEY: CarID{PNN 75573168975558} VALUE: 1

1 null

2 KEY: CarID{YCL 26173488024166} VALUE: 1

3 KEY: CarID{IKT 79165731364363} VALUE: 1

4 null

5 null

6 null

7 null

8 KEY: CarID{PFJ 61745334134984} VALUE: 1

9 KEY: CarID{SZD 67926802656931} VALUE: 1

10 null

The content of the hash table after file sortedFile.txt was processed:

The size of hash table is: 11

0 notIn

1 null

2 notIn

3 notIn

4 null

5 null

6 null

7 null

8 notIn

9 notIn

10 null

--> the elements in files "randomFile.txt" and "sortedFile.txt are the same

\*\*\* Checking if "sortedFile.txt" and "corruptedFile.txt" have the same elements \*\*\*

The content of the hash table after file sortedFile.txt was processed:

The size of hash table is: 11

0 KEY: CarID{IKT 79165731364363} VALUE: 1

1 null

2 KEY: CarID{YCL 26173488024166} VALUE: 1

3 KEY: CarID{PNN 75573168975558} VALUE: 1

4 null

5 null

6 null

7 null

8 KEY: CarID{PFJ 61745334134984} VALUE: 1

9 KEY: CarID{SZD 67926802656931} VALUE: 1

10 null

The content of the hash table after file corruptedFile.txt was processed:

The size of hash table is: 11

0 KEY: CarID{IKT 79165731364363} VALUE: 1

1 null

2 notIn

3 KEY: CarID{PNN 75573168975558} VALUE: 1

4 null

5 null

6 null

7 null

8 KEY: CarID{PFJ 61745334134984} VALUE: 1

9 KEY: CarID{SZD 67926802656931} VALUE: 1

10 null

--> the elements in files "corruptedFile.txt" and "sortedFile.txt are NOT the same

Bye!

Process finished with exit code 0