**GTU Department of Computer Engineering**

**CSE 222/505 - Spring 2021**

**Homework 5# Report**

**Sena Nur Ulukaya**

**1901042622**

1. **PROBLEM SOLUTION APPROACH**

**Part 1: MyHashMap Class**

First, I extended Java’s HashMap class and write a private inner iterator class to implement part 1 methods. I also write an interface for my MapIterator class to get and use my iterator in other classes because it is a private inner class itself.





In my MapIterator class, I used keySet() method to get all the keys and make that set an array with the toArray() method. By this way, I can reach the keys of the HashMap. I kept count as a counter of how many times I move forward and index for my array’s index.

In the zero-parameter constructor, I convert my HashMap to an array and get first element as a firstKey and last element as a lastKey to use later at the next() and prev() methods. Starting index is 0.

In the constructor with a key parameter, I first called my zero-parameter constructor and than checked if the given key is in the HashMap. If it is not, It will be same as zero-parameter constructor. Otherwise, firstKey will be given key and lastKey will be key before that and starting index will be firstKey’s index at the array.

Design of this class is dependent on the array. Basically, it is working like a circular array. If starting index is 0, it doesn’t need to but when it starts from other than zero it should be. Because it has to iterate through all the elements. For that I used % operation for my index.

In next(), it is iterating forward, count and index increases; in prev(), it is iterating backward, count and index decreases. When count is greater than size, next() returns the firstKey because there’s no not-iterated key and when count is smaller than 1, prev() returns the lastKey because it is in the first key. As I mentioned in the previous paragraph, next() takes mod of index with size and prev() makes index size -1 when index is smaller than 0 to make them act like a circular array.

hasNext() returns true if count is smaller than size, otherwise it returns false. Because count is the counter of moving forward.

I didn’t throw any exception because when there’s no next it should return firstKey and when there’s no prev(), it should return lastKey. So no exception needed.

**Part 2:**

I used the KWHashMap interface in our book and HashtableChain implementation. I added remove and rehash method for it. HashMap with TreeSet is very similar with LinkedList except Entry class and K should be comparable for TreeSet.

Coalesced HashMap is different from them. I added nextItem field for Entry class to keep the next key’s index.

When putting an entry to the hashMap, I checked the hashCode(), if that index is empty I put my entry there. If it is not empty, I’m finding an empty place by using the quadratic probing and make the first index that I checked’s nextItem the one that I put my entry. Also if key is already in the map, I replace its value with the new value and returned the old value.

When removing an entry, if there’s a next of the entry that I want to remove than I put the next to the its place and make the other one null. If there’s no next of the entry, I checked if there’s a next to the entry that I want to remove and if there’s I make it null, because there will be no such entry anymore.

To get an entry, I checked the first place it can be with hashCode again and then checked the next of them. If it’s exists it will return the value, otherwise it will return null.

1. **TEST CASES**

**Part 1: MyHashMap Class**

I used 2 MyHashMap object, hashMap with the numbers 0-10 in order and hashMap2 with 50 random numbers in range 250.

First, I iterate through them until hasNext() is false. After that I called my next() and prev() methods enough to show repetitive keys. (in the end and beginning of the HashMap.)

I repeated these steps with a zero-parameter constructor, a existing key as a parameter and a non-existing key as a parameter.

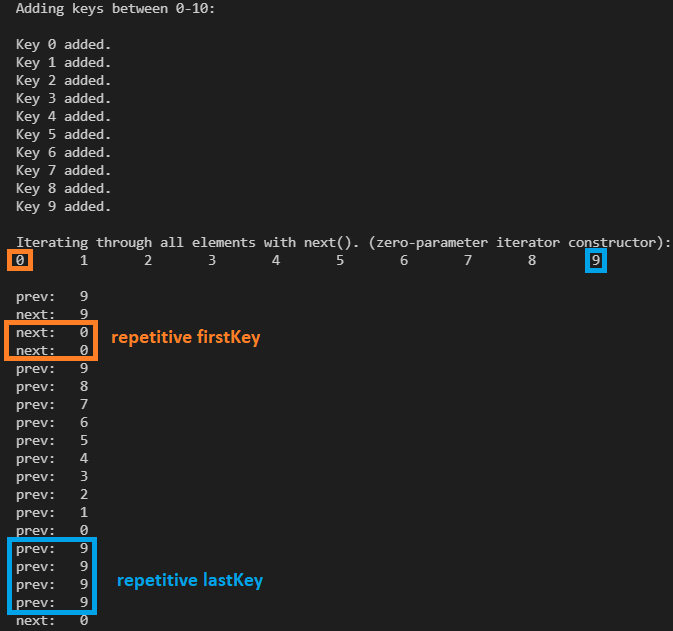
**Part 2:**

First, I made a little demonstration of how Coalesced HashMap works with a 4-5 key and try to show every feature of it. This is why I didn’t use random numbers and selected the myself.

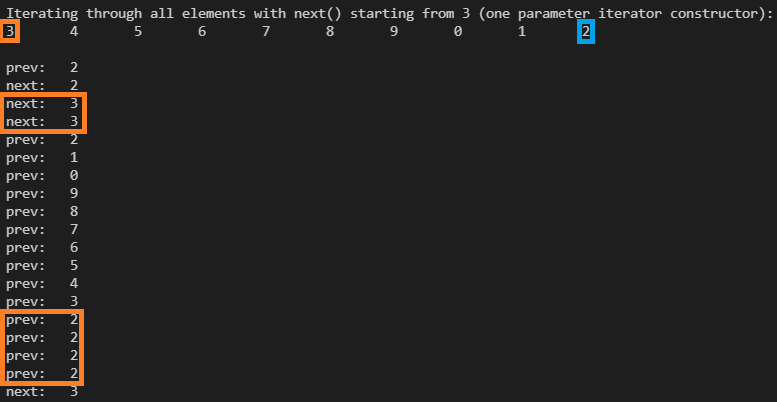
Then, I made runtime tests with 1000-10000-100000 datas with the 3 of the HashMap. I showed the all cases.

1. **RUNNING AND RESULTS**

**Part 1: MyHashMap Class**

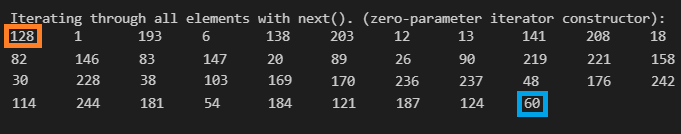
****

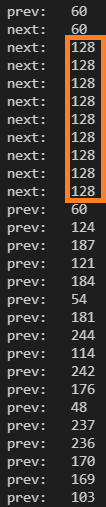
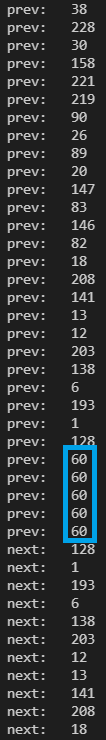
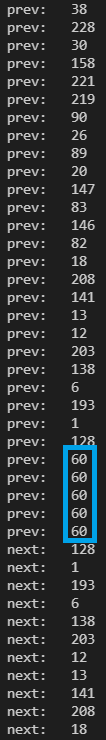
HashMap contains 3, so it started iterating from 3.

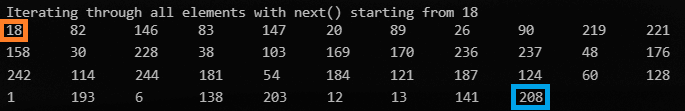
****

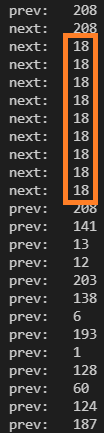
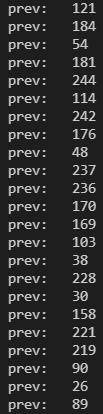
HashMap does not contain 12, so it is started iterating from beginning, same result with the first test.



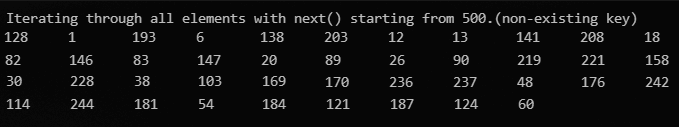


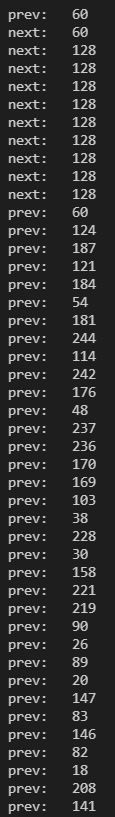
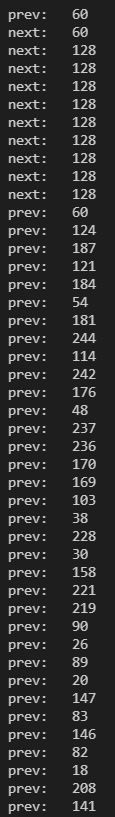
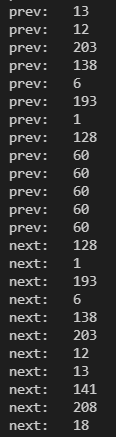
  



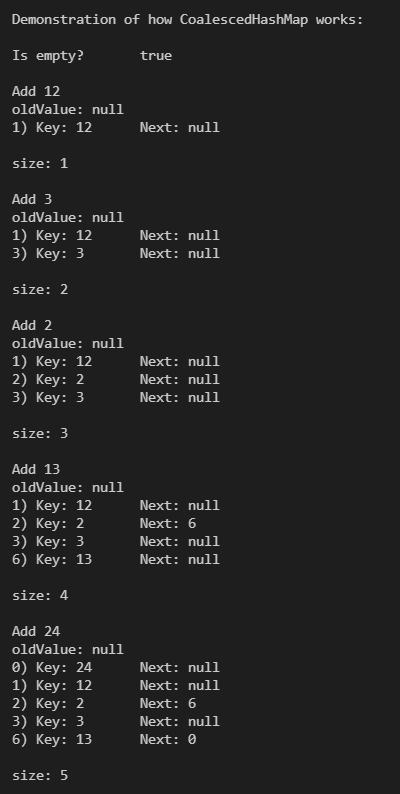
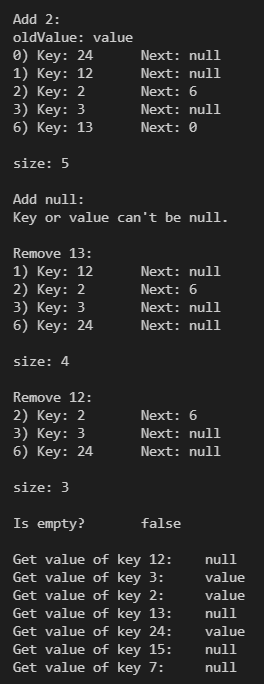
HashMap does not contain 500, so it is started iterating from beginning, same result with the first test.



**Part 2:**

\*oldValue is put method’s return value. It is null when element is adding first time and “value” otherwise because I make all the values, “ value” for the test.

**** 

When 13 is added, 2’s next changed and become 13’s index 6. 13%11 = 2.

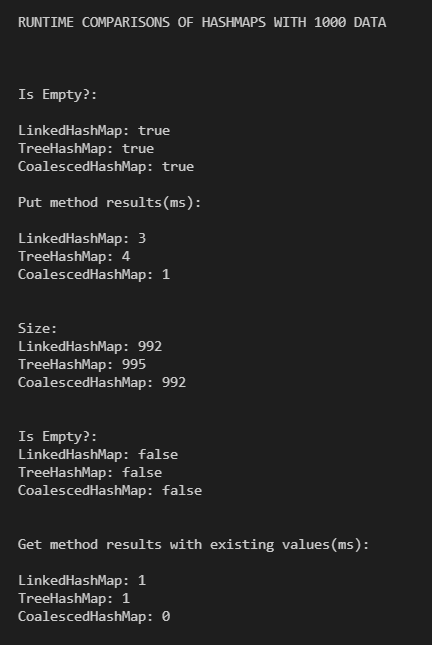
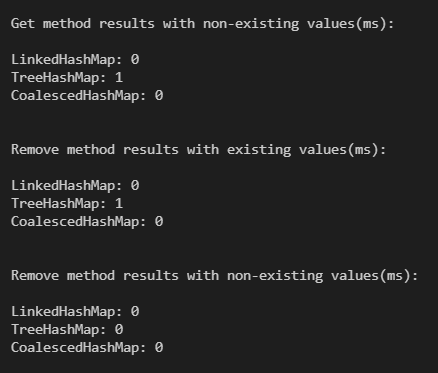
When 24 is added, 13’s next changed and become 24’s index 0. 24%11 = 2.

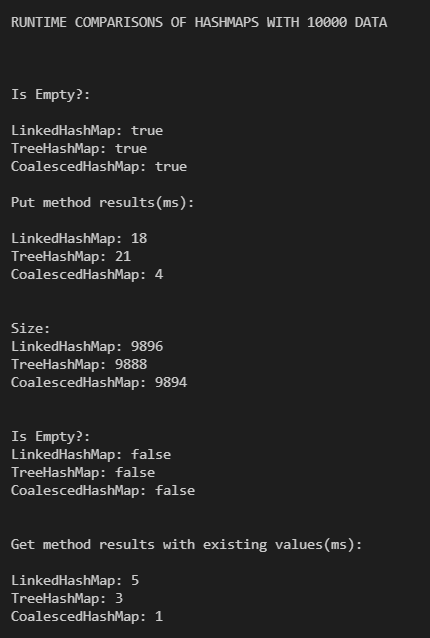
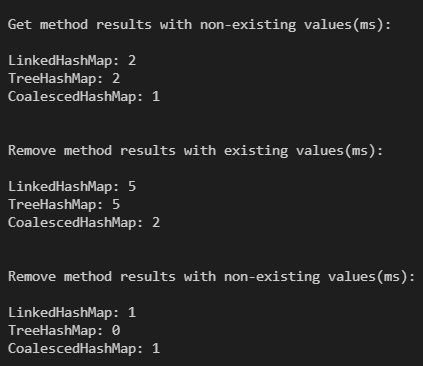
When 13 is removed 24 took its place because it was next of the 13.

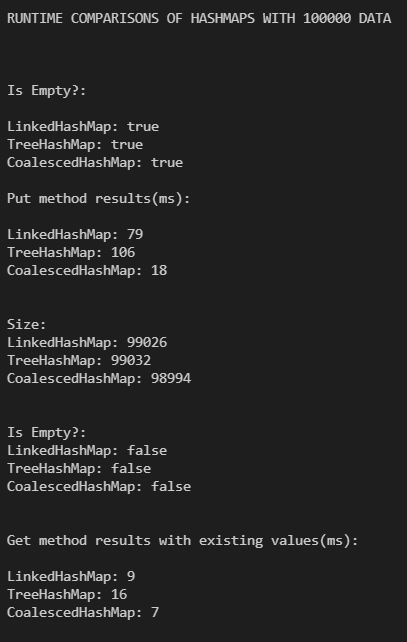
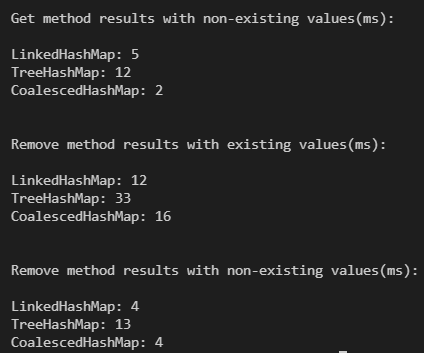
When I added 2, oldValue is value because other 2 is replaced. When I added “null” it throwed an exception.

**Runtime Tests:**

Sizes are not 1000-10000-10000 everytime because random can add same numbers and HashMap doesn’t allow this.

**** ****

**** ****

**** ****