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.

```
public class MyHashMap<K,V> extends HashMap<K,V>{
```

```
private class MapIterator implements MyMapIterator<K>{
```

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.

```
index = index % size();
```

```
if(index < 0) index = size() -1;
```

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.

2. 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.

3. RUNNING AND RESULTS

Part 1: MyHashMap Class

```
Adding keys between 0-10:

Key 0 added.
Key 1 added.
Key 2 added.
Key 3 added.
Key 4 added.
Key 5 added.
Key 6 added.
Key 7 added.
Key 8 added.
Key 9 added.

Iterating through all elements with next(). (zero-parameter iterator constructor):
0       1       2       3       4       5       6       7       8       9

prev:   9
next:   9
next:   0
next:   0
prev:   9
prev:   8
prev:   7
prev:   6
prev:   5
prev:   4
prev:   3
prev:   2
prev:   1
prev:   0
prev:   9
prev:   9
prev:   9
prev:   9
next:   0
```

repetitive firstKey

repetitive lastKey

HashMap contains 3, so it started iterating from 3.

```
Iterating through all elements with next() starting from 3 (one parameter iterator constructor):
3       4       5       6       7       8       9       0       1       2

prev:   2
next:   2
next:   3
next:   3
prev:   2
prev:   1
prev:   0
prev:   9
prev:   8
prev:   7
prev:   6
prev:   5
prev:   4
prev:   3
prev:   2
prev:   2
prev:   2
prev:   2
next:   3
```

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

```
Iterating through all elements with next() starting from 12 (non-existing key):
0       1       2       3       4       5       6       7       8       9

prev:   9
next:   9
next:   0
next:   0
prev:   9
prev:   8
prev:   7
prev:   6
prev:   5
prev:   4
prev:   3
prev:   2
prev:   1
prev:   0
prev:   9
prev:   9
prev:   9
prev:   9
next:   0
```

```
Iterating through all elements with next(). (zero-parameter iterator constructor):
128     1       193     6       138     203     12      13      141     208     18
82      146     83      147     20      89      26      90      219     221     158
30      228     38      103     169     170     236     237     48      176     242
114     244     181     54      184     121     187     124     60
```

```
prev:   60
next:   60
next:   128
next:   128
next:   128
next:   128
next:   128
next:   128
next:   128
next:   128
next:   128
prev:   60
prev:   124
prev:   187
prev:   121
prev:   184
prev:   54
prev:   181
prev:   244
prev:   114
prev:   242
prev:   176
prev:   48
prev:   237
prev:   236
prev:   170
prev:   169
prev:   103
```

```
prev:   38
prev:   228
prev:   30
prev:   158
prev:   221
prev:   219
prev:   90
prev:   26
prev:   89
prev:   20
prev:   147
prev:   83
prev:   146
prev:   82
prev:   18
prev:   208
prev:   141
prev:   13
prev:   12
prev:   203
prev:   138
prev:   6
prev:   193
prev:   1
```

```
prev:   128
prev:   60
prev:   60
prev:   60
prev:   60
prev:   60
next:   128
next:   1
next:   193
next:   6
next:   138
next:   203
next:   12
next:   13
next:   141
next:   208
next:   18
```

```
Iterating through all elements with next() starting from 18
18      82      146     83      147     20      89      26      90      219     221
158     30      228     38      103     169     170     236     237     48      176
242     114     244     181     54      184     121     187     124     60      128
1       193     6       138     203     12      13      141     208
```

```
prev:   208
next:   208
next:   18
next:   18
next:   18
next:   18
next:   18
next:   18
next:   18
next:   18
next:   18
prev:   208
prev:   141
prev:   13
prev:   12
prev:   203
prev:   138
prev:   6
prev:   193
prev:   1
prev:   128
prev:   60
prev:   124
prev:   187
```

```
prev:   121
prev:   184
prev:   54
prev:   181
prev:   244
prev:   114
prev:   242
prev:   176
prev:   48
prev:   237
prev:   236
prev:   170
prev:   169
prev:   103
prev:   38
prev:   228
prev:   30
prev:   158
prev:   221
prev:   219
prev:   90
prev:   26
prev:   89
```

```
next:   26
next:   90
next:   219
next:   221
```

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

```
Iterating through all elements with next() starting from 500.(non-existing key)
128     1       193     6       138     203     12      13      141     208     18
82      146     83      147     20      89      26      90      219     221     158
30      228     38      103     169     170     236     237     48      176     242
114     244     181     54      184     121     187     124     60
```

```
prev:   60
next:   60
next:   128
next:   128
next:   128
next:   128
next:   128
next:   128
next:   128
next:   128
next:   128
prev:   60
prev:   124
prev:   187
prev:   121
prev:   184
prev:   54
prev:   181
prev:   244
prev:   114
prev:   242
prev:   176
prev:   48
prev:   237
prev:   236
```

```
prev:   170
prev:   169
prev:   103
prev:   38
prev:   228
prev:   30
prev:   158
prev:   221
prev:   219
prev:   90
prev:   26
prev:   89
prev:   20
prev:   147
prev:   83
prev:   146
prev:   82
prev:   18
prev:   208
prev:   141
```

```
prev:   13
prev:   12
prev:   203
prev:   138
prev:   6
prev:   193
prev:   1
prev:   128
prev:   60
prev:   60
prev:   60
prev:   60
prev:   60
next:   128
next:   1
next:   193
next:   6
next:   138
next:   203
next:   12
next:   13
next:   141
next:   208
next:   18
```

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.

```
Demonstration of how CoalescedHashMap works:

Is empty?       true

Add 12
oldValue: null
1) Key: 12      Next: null

size: 1

Add 3
oldValue: null
1) Key: 12      Next: null
3) Key: 3       Next: null

size: 2

Add 2
oldValue: null
1) Key: 12      Next: null
2) Key: 2       Next: null
3) Key: 3       Next: null

size: 3

Add 13
oldValue: null
1) Key: 12      Next: null
2) Key: 2       Next: 6
3) Key: 3       Next: null
6) Key: 13      Next: null

size: 4

Add 24
oldValue: null
0) Key: 24      Next: null
1) Key: 12      Next: null
2) Key: 2       Next: 6
3) Key: 3       Next: null
6) Key: 13      Next: 0

size: 5
```

```
Add 2:
oldValue: value
0) Key: 24      Next: null
1) Key: 12      Next: null
2) Key: 2       Next: 6
3) Key: 3       Next: null
6) Key: 13      Next: 0

size: 5

Add null:
Key or value can't be null.

Remove 13:
1) Key: 12      Next: null
2) Key: 2       Next: 6
3) Key: 3       Next: null
6) Key: 24      Next: null

size: 4

Remove 12:
2) Key: 2       Next: 6
3) Key: 3       Next: null
6) Key: 24      Next: null

size: 3

Is empty?       false

Get value of key 12:    null
Get value of key 3:     value
Get value of key 2:     value
Get value of key 13:    null
Get value of key 24:    value
Get value of key 15:    null
Get value of key 7:     null
```

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.

```
RUNTIME COMPARISONS OF HASHMAPS WITH 1000 DATA


Is Empty?:

LinkedHashMap: true
TreeHashMap: true
CoalescedHashMap: true

Put method results(ms):

LinkedHashMap: 3
TreeHashMap: 4
CoalescedHashMap: 1


Size:
LinkedHashMap: 992
TreeHashMap: 995
CoalescedHashMap: 992


Is Empty?:
LinkedHashMap: false
TreeHashMap: false
CoalescedHashMap: false


Get method results with existing values(ms):

LinkedHashMap: 1
TreeHashMap: 1
CoalescedHashMap: 0
```

```
Get method results with non-existing values(ms):

LinkedHashMap: 0
TreeHashMap: 1
CoalescedHashMap: 0


Remove method results with existing values(ms):

LinkedHashMap: 0
TreeHashMap: 1
CoalescedHashMap: 0


Remove method results with non-existing values(ms):

LinkedHashMap: 0
TreeHashMap: 0
CoalescedHashMap: 0
```

```
RUNTIME COMPARISONS OF HASHMAPS WITH 10000 DATA


Is Empty?:

LinkedHashMap: true
TreeHashMap: true
CoalescedHashMap: true

Put method results(ms):

LinkedHashMap: 18
TreeHashMap: 21
CoalescedHashMap: 4


Size:
LinkedHashMap: 9896
TreeHashMap: 9888
CoalescedHashMap: 9894


Is Empty?:
LinkedHashMap: false
TreeHashMap: false
CoalescedHashMap: false


Get method results with existing values(ms):

LinkedHashMap: 5
TreeHashMap: 3
CoalescedHashMap: 1
```

```
Get method results with non-existing values(ms):

LinkedHashMap: 2
TreeHashMap: 2
CoalescedHashMap: 1


Remove method results with existing values(ms):

LinkedHashMap: 5
TreeHashMap: 5
CoalescedHashMap: 2


Remove method results with non-existing values(ms):

LinkedHashMap: 1
TreeHashMap: 0
CoalescedHashMap: 1
```

```
RUNTIME COMPARISONS OF HASHMAPS WITH 100000 DATA



Is Empty?:

LinkedHashMap: true
TreeHashMap: true
CoalescedHashMap: true

Put method results(ms):

LinkedHashMap: 79
TreeHashMap: 106
CoalescedHashMap: 18


Size:
LinkedHashMap: 99026
TreeHashMap: 99032
CoalescedHashMap: 98994


Is Empty?:
LinkedHashMap: false
TreeHashMap: false
CoalescedHashMap: false


Get method results with existing values(ms):

LinkedHashMap: 9
TreeHashMap: 16
CoalescedHashMap: 7
```

```
Get method results with non-existing values(ms):

LinkedHashMap: 5
TreeHashMap: 12
CoalescedHashMap: 2


Remove method results with existing values(ms):

LinkedHashMap: 12
TreeHashMap: 33
CoalescedHashMap: 16


Remove method results with non-existing values(ms):

LinkedHashMap: 4
TreeHashMap: 13
CoalescedHashMap: 4
```