GTU Department of Computer Engineering
CSE 222/505 - Spring 2021
Homework 5 Report

Mustafa Karakaş
1801042627

Part1

1. SYSTEM REQUIREMENTS

HashMapIterable():

This functions constructs the HashMap which is iterable. You can use it as it is a HashMap because this class is extended from HashMap class.

The special part is this function has a iterator. If you call:

```
mapIterator()
```

This function returns you mapIterator which starts from the first element in the map.

```
mapIterator(K key)
```

Also, if you call this function , the iterator starts from the given key if it exists, otherwise it works life default constructer.

After calling this function, you can use these functions :

hasNext() , next() , prev()

2. PROBLEM SOLUTION APPROACH

To be able to go through all elements, I keep the number of elements traveled. So that when all elements are visited hasNext returns false. Next and prev functions can go as much as you want. They move like they are traveling on circular array.

3. TEST CASES

```
HashMapIterable<Integer ,Integer> hash = new HashMapIterable<Integer ,Integer>();
for(int i=0;i<45;i++) {
    hash.put(i, i*2);
}
```

Hash is constructed and it is filled.

```
System.out.println("Hash itself:\n"+hash);
System.out.println("\nEnhanced for loop");
for(int x : hash) {
    System.out.print(x+" ");
}
System.out.println();
```

```
Hash itself:
{0=0, 1=2, 2=4, 3=6, 4=8, 5=10, 6=12, 7=14, 8=16, 9=18, 10=20, 11=22, 12=24, 13=26, 14=28, 15=30, 16=32, 17=34, 18=36, 19
=38, 20=40, 21=42, 22=44, 23=46, 24=48, 25=50, 26=52, 27=54, 28=56, 29=58, 30=60, 31=62, 32=64, 33=66, 34=68, 35=70, 36=7
2, 37=74, 38=76, 39=78, 40=80, 41=82, 42=84, 43=86, 44=88}
```

It showed all elements.

```
HashMapIterable<Integer,Integer>.MapIterator <Integer> iter = hash.mapIterator();
HashMapIterable<Integer,Integer>.MapIterator <Integer> iter2 = hash.mapIterator(10);
HashMapIterable<Integer,Integer>.MapIterator <Integer> iter3 = hash.mapIterator(-32412);
```

3 different iterator is constructed.

Iter = iterator starts from the head

Iter2 = iterator starts from the element whose key is 10

Iter3 = iterator starts from the head because –32412 does not belong to hash

```
System.out.println("\nNO PARAMETER ITERATOR TEST");
IteratorTest(iter);
```

```
static void IteratorTest(HashMapIterable<Integer,Integer>.MapIterator <Integer> iter2 ) {

    System.out.println();

    System.out.println("\nhasNext");
    while(iter2.hasNext()) {
        System.out.print(iter2.next()+" ");
    }
    System.out.println();
    System.out.println("\n50 step backward");
    for(int i=0;i<50;i++)
        System.out.print(iter2.prev()+" ");

    System.out.println();
    System.out.println("\n50 step forward");
    for(int i=0;i<50;i++)
        System.out.print(iter2.next()+" ");
    System.out.println();
}
```

```
NO PARAMETER ITERATOR TEST


hasNext
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

50 step backward
44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 44 43 42 41 40

50 step forward
40 41 42 43 44 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
```

It starts from the head and goes to the end.

50 step backward starts from the last key and it goes 50 step back.it goes back to the last element when it past the first element.

50 step forward starts from where the iterator was in. And it goes 50 element forward. When it reaches the last element it goes back to the first element.

```
System.out.println("\nIterator with parameter test");
IteratorTest(iter2);
```

Iter2 starts from the element whose key is 10.

```
Iterator with parameter test


hasNext
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 0 1 2 3 4 5 6 7 8 9

50 step backward
9 8 7 6 5 4 3 2 1 0 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5

50 step forward
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 0 1 2 3 4 5 6 7 8 9
```

It works expectedly.

```
System.out.println("\nIterator with non exist parameter test");
IteratorTest(iter3);
```

```
Iterator with non exist parameter test


hasNext
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

50 step backward
44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 44 43 42 41 40

50 step forward
40 41 42 43 44 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
```

It works in the same way as first example.

Part2

1.SYSTEM REQUIREMENTS

```
HashTableChainLinked()
HashTableChainTree()
HashTableOpen()
```

These functions creates hashmaps.

```
V get(Object key);
```

This function returns the value associated with the given key.

```
boolean isEmpty();
```

This function shows if the hashmap is empty or not.

```
V put(K key, V value);
```

This function adds the given key and value to map. If the key already exist it changes the associated value. And returns the old value. Else it returns null.

```
V remove(Object key);
```

This function removes the element which has given key and returns the value of it. If it does not exist, removes null.

```
int size();
```

This function returns the size of the map.

2.PROBLEM SOLUTION APPROACH

First 2 implementations are available in the book. I will explain only the third part.

Put Method:

To be able to add new element, I checked first the hashCode%table.length th index. If it is filled with another variable, it will check index+1^2,index+2^2,index+3^2... and so on. Every element points to the next element whose first possible index is same with its first possible index. Put methods handled it. The last element's next is –1. If the number of element is more than wanted, rehash function is called.

Remove method:

Remove method checks the first possible index and it goes where the next index shows. If the given key exists it removes it and shifts the elements which is in the next of the deleted element. It puts special element DELETED at the end of the sequence.

Get method:

Get method checks indexes quadraticly. If the element is found, it returns its value, othewise returns null.

3.TEST CASES

```
TEST OF PART 2

Duration for adding 100 elements in HashTableChain:: 2.1539 ms
Duration for adding 100 elements in HashTableTreeSetChain:: 2.8852 ms
Duration for adding 100 elements in HashTableCoalesced:: 0.1703 ms

Duration for Get() 100 elements in HashTableChain:: 0.1612 ms
Duration for Get() 100 elements in HashTableTreeSetChain:: 0.2888 ms
Duration for Get() 100 elements in HashTableCoalesced:: 0.0914 ms

Duration for Removing 100 elements in HashTableChain:: 0.1976 ms
Duration for Removing 100 elements in HashTableTreeSetChain:: 0.1831 ms
Duration for Removing 100 elements in HashTableCoalesced:: 0.1297 ms

Duration for Removing NOT EXIST 100 elements in HashTableChain:: 0.08 ms
Duration for Removing NOT EXIST 100 elements in HashTableTreeSetChain:: 0.0903 ms
Duration for Removing NOT EXIST 100 elements in HashTableCoalesced:: 0.0229 ms
```

```
Duration for adding 1000 elements in HashTableChain:: 3.0281 ms
Duration for adding 1000 elements in HashTableTreeSetChain:: 2.3903 ms
Duration for adding 1000 elements in HashTableCoalesced:: 1.1548 ms

Duration for Get() 1000 elements in HashTableChain:: 0.6302 ms
Duration for Get() 1000 elements in HashTableTreeSetChain:: 0.6503 ms
Duration for Get() 1000 elements in HashTableCoalesced:: 0.2524 ms

Duration for Removing 1000 elements in HashTableChain:: 0.7167 ms
Duration for Removing 1000 elements in HashTableTreeSetChain:: 1.0672 ms
Duration for Removing 1000 elements in HashTableCoalesced:: 0.6868 ms

Duration for Removing NOT EXIST 1000 elements in HashTableChain:: 0.1553 ms
Duration for Removing NOT EXIST 1000 elements in HashTableTreeSetChain:: 0.1608 ms
Duration for Removing NOT EXIST 1000 elements in HashTableCoalesced:: 0.104 ms
```

```
Duration for adding 10000 elements in HashTableChain:: 7.5748 ms
Duration for adding 10000 elements in HashTableTreeSetChain:: 11.32 ms
Duration for adding 10000 elements in HashTableCoalesced:: 3.6965 ms

Duration for Get() 10000 elements in HashTableChain:: 1.7948 ms
Duration for Get() 10000 elements in HashTableTreeSetChain:: 3.0588 ms
Duration for Get() 10000 elements in HashTableCoalesced:: 1.0504 ms

Duration for Removing 10000 elements in HashTableChain:: 4.7938 ms
Duration for Removing 10000 elements in HashTableTreeSetChain:: 5.4686 ms
Duration for Removing 10000 elements in HashTableCoalesced:: 1.3556 ms

Duration for Removing NOT EXIST 10000 elements in HashTableChain:: 2.1336 ms
Duration for Removing NOT EXIST 10000 elements in HashTableTreeSetChain:: 3.0747 ms
Duration for Removing NOT EXIST 10000 elements in HashTableCoalesced:: 1.2307 ms
```

```java
for (int i = N; i > 1 ; i--){

    hashtableChain.put(2*i,2*i);
}

for (int i = N; i > 1 ; i--){

    hashtableTreeSetChain.put(2*i,2*i);
}


for (int i = N; i > 1 ; i--){

    hashtableCoalesced.put(2*i,2*i);
}
```

Elements are placed.

```java
int test;
for (int i = N; i > 1 ; i--){
    test = 2*i;
    //element which exists searched
    if(hashtableChain.get(2*i) != test || hashtableCoalesced.get(2*i) != test || hashtableTreeSetChain.get(2*i) != test) {
        System.out.println("ERROR OCCURED");
        return false;
    }
    // element which does not exist searched
    if(hashtableChain.get(-2*i) != null || hashtableCoalesced.get(-2*i) != null || hashtableTreeSetChain.get(-2*i) != null) {
        System.out.println("ERROR OCCURED");
        return false;
    }

}
```

```java
System.out.println("");
//TEST OF REMOVING ELEMENT NOT IN THE MAP
for (int i = 1; i < N ; i++){
     if(hashtableChain.remove(-5*i) != null) {
        System.out.println("ERROR OCCURED");
        return false;
     }
}
for (int i = 1; i < N ; i++){
     if(hashtableTreeSetChain.remove(-5*i) != null) {
        System.out.println("ERROR OCCURED");
        return false;
     }
 }
 for (int i = 1; i < N ; i++){

     if(hashtableCoalesced.remove(-5*i) != null) {
        System.out.println("ERROR OCCURED");
        return false;
     }
 }
```

```java
// TEST OF REMOVING ELEMENT IN THE MAP
for (int i = 2; i < N ; i++){
    Integer a = hashtableChain.remove(2*i);
    if(!a.equals(2*i)) {
        System.out.println("ERROR OCCURED");
        return false;
    }
}

for (int i = 2; i < N ; i++){
    Integer a = hashtableTreeSetChain.remove(2*i);
    if(!a.equals(2*i)) {
        System.out.println("ERROR OCCURED");
        return false;
    }
}
for (int i = 2; i < N ; i++){
    Integer a = hashtableCoalesced.remove(2*i);
    if(!a.equals(2*i)) {
        System.out.println("ERROR OCCURED");
        return false;
    }
}
```

It never shows a message so it works expectedly.

```java
HashTableOpen<Integer,Integer> hashtableCoalesced = new HashTableOpen<Integer,Integer>();

hashtableCoalesced.put(13, 13);
hashtableCoalesced.put(24, 24);
hashtableCoalesced.put(35, 35);
hashtableCoalesced.put(51, 51);
hashtableCoalesced.put(46, 46);
System.out.println("Before remove :\n"+hashtableCoalesced);
hashtableCoalesced.remove(24);
System.out.println("After remove :\n"+hashtableCoalesced);
```

```
Before remove :
index    value   next
0        46      -1
1        null
2        13      3
3        24      6
4        null
5        null
6        35      0
7        51      -1
8        null
9        null
10       null

After remove :
index    value   next
0        deleted -1
1        null
2        13      3
3        35      6
4        null
5        null
6        46      -1
7        51      -1
8        null
9        null
10       null
```