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

Can Duyar
171044075

Explanation of the PDF Requirements:

* Each of the requirements are done.(OOP design,using interface,method overriding,error handling,inheritance and polymorphism)

* 1-PROBLEM SOLUTIONS APPROACH:

Part-1(MapIterator):
-The problem was that writing a custom iterator class MapIterator to iterate through the keys in a HashMap data structure in Java. I used logic of inner class for this part. My main methods are inside the inner class and I used outer class to reach my iterator. I can reach the iterator without using any parameter, if I do this then my iterator will start from the 0. index. Other case is that if I give an index as a parameter for my iterator's constructor then it will start from the index which is given as parameter. I used a inner class to solve this difference.

Part-2 (Implement KWHashMap interface with three ways)

2.1(Using the chaining technique for hashing by using linked lists):
- the problem was the keeping key-value pairs in linkedlist if they have same index value so it was not so challenging part. I solved it with creating a HashEntry class to keep information(key-value) of the pairs and keep a linkedlist in my LinkedListChaining class. I also kept my collision value in a variable and showed it my test cases.I increased the number of this variable, if I have more than one key that have same index value.

2.2(Using the chaining technique for hashing by using TreeSet(instead of linkedlist):

- the problem was the keeping key-value pairs in TreeSet if they have same index value so it was not so challenging part(because it was so similar with my previous linkedlist solution). I solved it with creating a HashEntry class to keep information(key-value) of the pairs and kept a TreeSet in my TreeSetChaining class. I also overrode comparetTo and equals methods in my HashEntry to solve the problem of TreeSet implementation. I also changed my previous linkedlist solution with adding iterator instead of forwarding with index.

2.3(Using the chaining technique for hashing by using TreeSet(instead of linkedlist):

I used the Coalesced hashing technique. This technique uses the concept of Open Addressing to find first empty place for colliding element by using the quadratic probing and the concept of Separate Chaining to link the colliding elements to each other through pointers (indices in the table). It was the most challenging part for me because I couldn’t understand the logic of “next” part on the hash table at the first stage. After that I understood and solved it with keeping a next variable in addition to key-value variables in my HashEntry class. Other problem was related with implementing quadratic probing problem. I kept a value for this operation and increased its value with also multiplying with itself and added with index value.

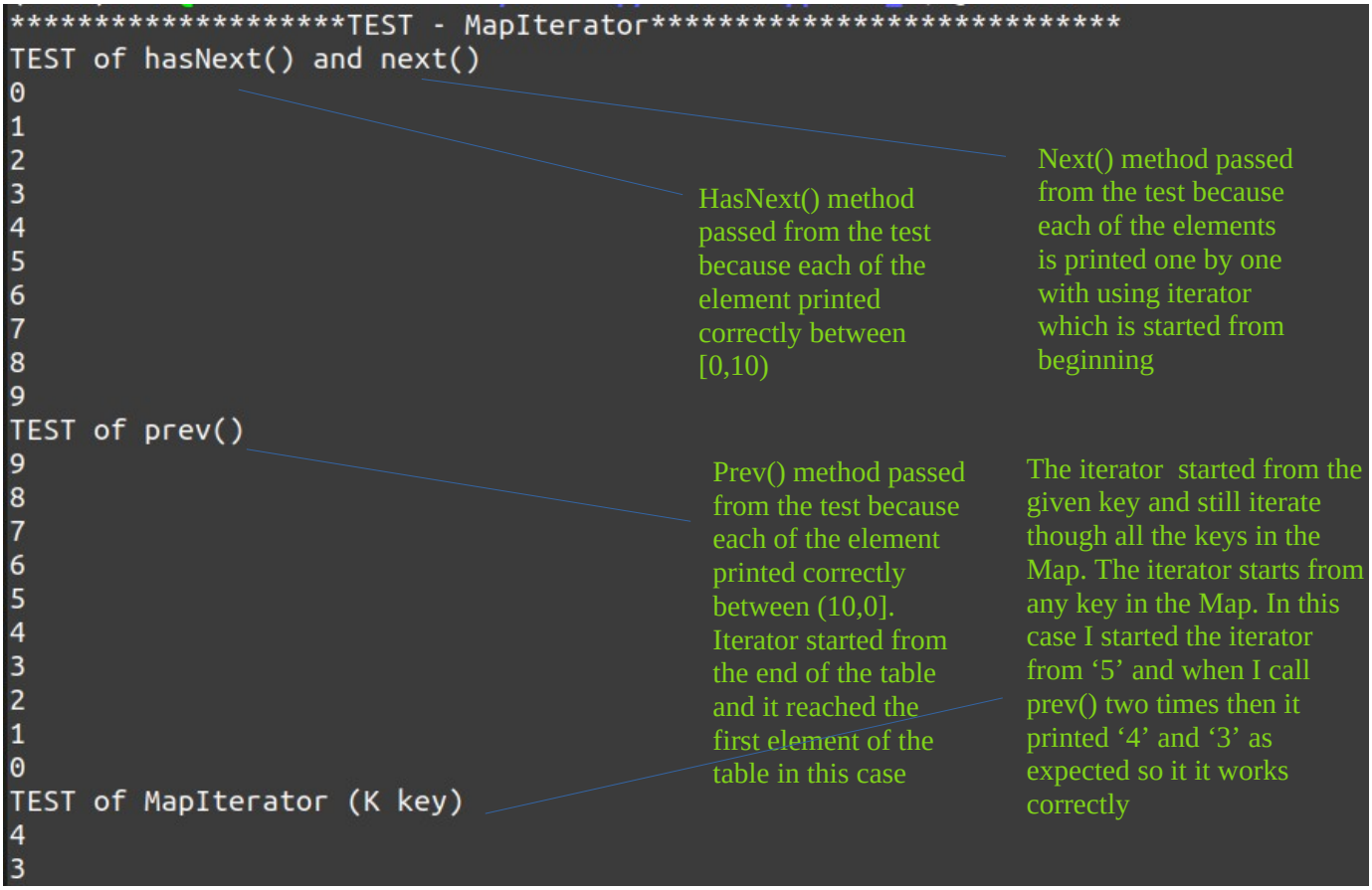
2-TEST CASES / RUNNING COMMAND AND RESULTS:

Part-1: Test of MapIterator

```
System.out.println("********************TEST - MapIterator**************************");

MapIterator<Integer, Character> testIter = new MapIterator<>();

    for(ch='a',in = 0; ch<='z' && in < 10 ; ch++,in++){
        testIter.put(in,ch);
    }

    /*  KEY VALUE
         0    a
         1    b
         2    c
         3    d
        .
        .
        .

    */
MapIterator.IteratorMap it = testIter.MapIterator();

System.out.println("TEST of hasNext() and next()");
while(it.hasNext())
    System.out.println(it.next());

System.out.println("TEST of prev()");
for(int t = 0; t < testIter.size();t++)
    System.out.println(it.prev());


System.out.println("TEST of MapIterator (K key)");
MapIterator.IteratorMap it2 = testIter.MapIterator(5);
System.out.println(it2.prev());
System.out.println(it2.prev());
```

HasNext() helps the iterator to iterate elements one by one and it stops when iterator reaches the end then it stops because hasNext() returns false

I tested all methods of the MapIterator class. I added 10 keys and values for my iterator and test its methods in the following part. The result is below with explanations.



Part-2.1: Test of Hashing by using linkedlist

I tested all methods of the LinkedListChaining class. I added 10 keys and values for small-sized, 100 for medium-sized and 1000 for larged-sized. I also calculate the execution time with using large-sized data to compare the performance of hashing techniques.

Key-value pairs are like:

1 -1

2 -2

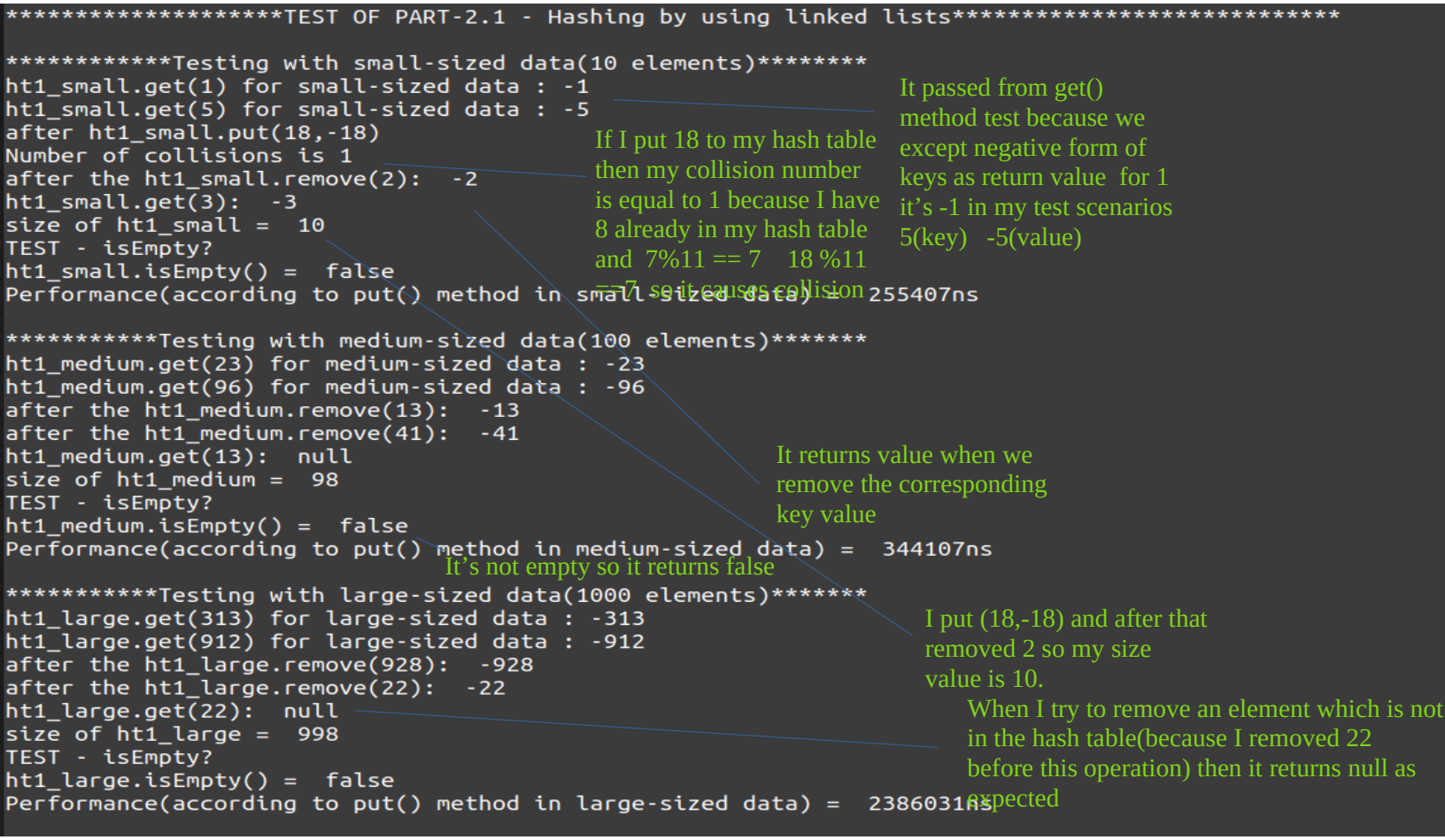
.

.

I assigned the negatives of the keys as the value for my hashing tests.

```
System.out.println("\n***********Testing with small-sized data(10 elements)********");
start = System.nanoTime();
for(t = 0,u = 0; t < 10 && u > -10; t++,u--)
    ht1_small.put(t,u);
end = System.nanoTime();

System.out.println("ht1_small.get(1) for small-sized data : " + ht1_small.get(1));
System.out.println("ht1_small.get(5) for small-sized data : " + ht1_small.get(5));
System.out.println("after ht1_small.put(18,-18)");
ht1_small.put(18,-18);
ht1_small.getNumberOfCollisions();

System.out.println("after the ht1_small.remove(2):  " + ht1_small.remove(2));
System.out.println("ht1_small.get(3):  " + ht1_small.get(3));
System.out.println("size of ht1_small =  " + ht1_small.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht1_small.isEmpty() =  " + ht1_small.isEmpty());
System.out.println("Performance(according to put() method in small-sized data) =  " + (end-start) + "ns");

System.out.println("\n***********Testing with medium-sized data(100 elements)*******");
start = System.nanoTime();
for(t = 0,u = 0; t < 100 && u > -100; t++,u--)
    ht1_medium.put(t,u);
end = System.nanoTime();

System.out.println("ht1_medium.get(23) for medium-sized data : " + ht1_medium.get(23));
System.out.println("ht1_medium.get(96) for medium-sized data : " + ht1_medium.get(96));

System.out.println("after the ht1_medium.remove(13):  " + ht1_medium.remove(13));
System.out.println("after the ht1_medium.remove(41):  " + ht1_medium.remove(41));
System.out.println("ht1_medium.get(13):  " + ht1_medium.get(13));
System.out.println("size of ht1_medium =  " + ht1_medium.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht1_medium.isEmpty() =  " + ht1_medium.isEmpty());
System.out.println("Performance(according to put() method in medium-sized data) =  " + (end-start) + "ns");

System.out.println("\n***********Testing with large-sized data(1000 elements)*******");
start = System.nanoTime();
for(t = 0,u = 0; t < 1000 && u > -1000; t++,u--)
    ht1_large.put(t,u);

end = System.nanoTime();

System.out.println("ht1_large.get(313) for large-sized data : " + ht1_large.get(313));
System.out.println("ht1_large.get(912) for large-sized data : " + ht1_large.get(912));

System.out.println("after the ht1_large.remove(928):  " + ht1_large.remove(928));
System.out.println("after the ht1_large.remove(22):  " + ht1_large.remove(22));
System.out.println("ht1_large.get(22):  " + ht1_large.get(22));
System.out.println("size of ht1_large =  " + ht1_large.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht1_large.isEmpty() =  " + ht1_large.isEmpty());


System.out.println("Performance(according to put() method in large-sized data) =  " + (end-start) + "ns");
```

Results: I used 11 as table size because it's a prime number and prime numbers are more efficient for hash-tables



*execution time of Hashing by using linkedlist:
-small sized data: 255407 ns for put operation
-medium sized data: 344107 ns for put operation
- large sized data: 2386031 ns for put operation

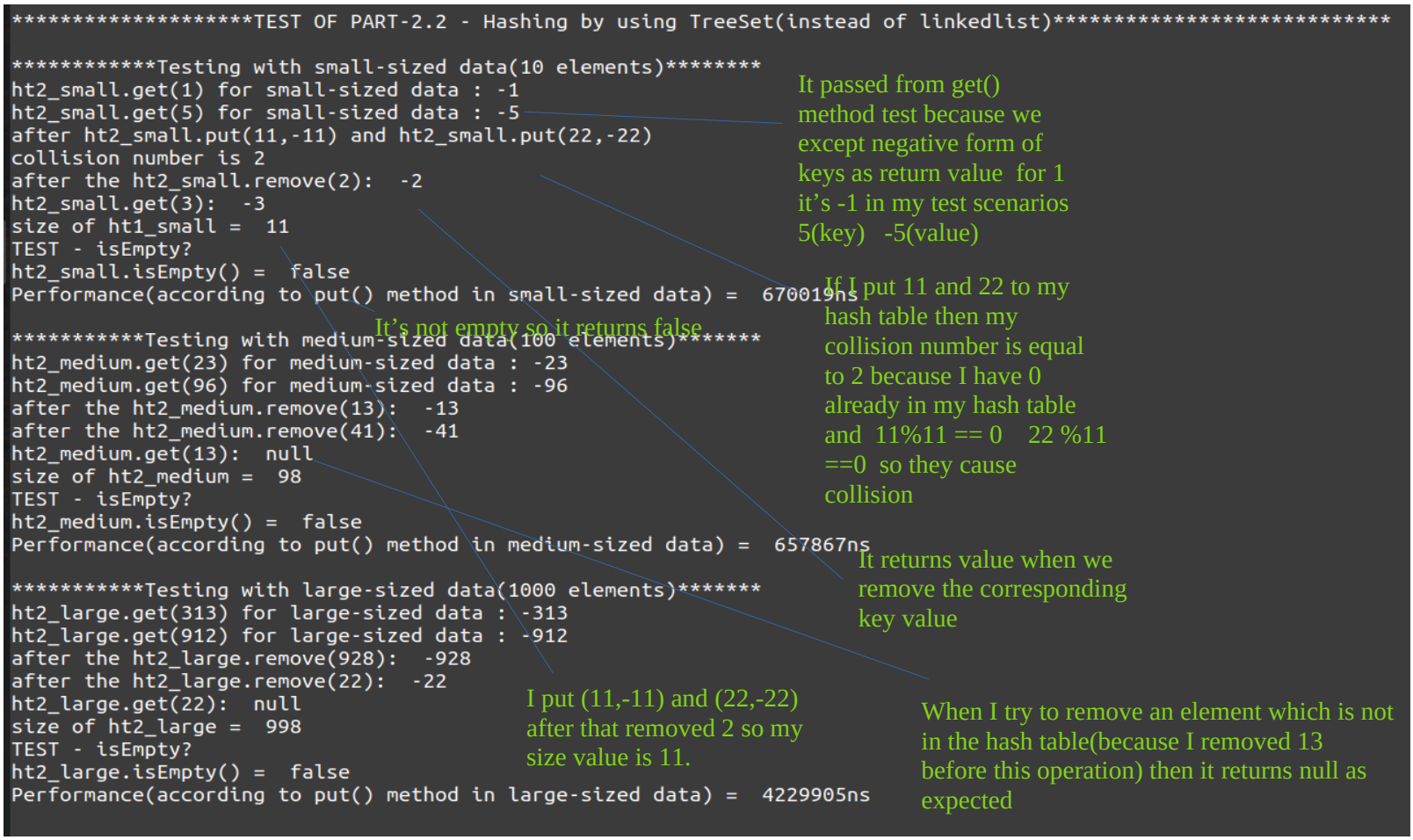
Part-2.2: Test of Hashing by using TreeSet(instead of linkedlist)

I tested all methods of the TreeSetChaining class. I added 10 keys and values for small-sized, 100 for medium-sized and 1000 for larged-sized. I also calculate the execution time with using large-sized data to compare the performance of hashing techniques.

Key-value pairs are like:
1 -1
2 -2
.
.
I assigned the negatives of the keys as the value for my hashing tests.

```
System.out.println("\n***********Testing with small-sized data(10 elements)*******");
start = System.nanoTime();
for(t = 0,u = 0; t < 10 && u > -10; t++,u--)
    ht2_small.put(t,u);
end = System.nanoTime();


System.out.println("ht2_small.get(1) for small-sized data : " + ht2_small.get(1));
System.out.println("ht2_small.get(5) for small-sized data : " + ht2_small.get(5));
System.out.println("after ht2_small.put(11,-11) and ht2_small.put(22,-22)");
ht2_small.put(11,-11);
ht2_small.put(22,-22);
ht2_small.getNumberOfCollisions();

System.out.println("after the ht2_small.remove(2):  " + ht2_small.remove(2));
System.out.println("ht2_small.get(3):  " + ht2_small.get(3));
System.out.println("size of ht1_small =  " + ht2_small.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht2_small.isEmpty() =  " + ht2_small.isEmpty());
System.out.println("Performance(according to put() method in small-sized data) =  " + (end-start) + "ns");

System.out.println("\n***********Testing with medium-sized data(100 elements)*******");
start = System.nanoTime();
for(t = 0,u = 0; t < 100 && u > -100; t++,u--)
    ht2_medium.put(t,u);
end = System.nanoTime();

System.out.println("ht2_medium.get(23) for medium-sized data : " + ht2_medium.get(23));
System.out.println("ht2_medium.get(96) for medium-sized data : " + ht2_medium.get(96));

System.out.println("after the ht2_medium.remove(13):  " + ht2_medium.remove(13));
System.out.println("after the ht2_medium.remove(41):  " + ht2_medium.remove(41));
System.out.println("ht2_medium.get(13):  " + ht2_medium.get(13));
System.out.println("size of ht2_medium =  " + ht2_medium.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht2_medium.isEmpty() =  " + ht2_medium.isEmpty());
System.out.println("Performance(according to put() method in medium-sized data) =  " + (end-start) + "ns");

System.out.println("\n***********Testing with large-sized data(1000 elements)*******");
start = System.nanoTime();
for(t = 0,u = 0; t < 1000 && u > -1000; t++,u--)
    ht2_large.put(t,u);

end = System.nanoTime();

System.out.println("ht2_large.get(313) for large-sized data : " + ht2_large.get(313));
System.out.println("ht2_large.get(912) for large-sized data : " + ht2_large.get(912));

System.out.println("after the ht2_large.remove(928):  " + ht2_large.remove(928));
System.out.println("after the ht2_large.remove(22):  " + ht2_large.remove(22));
System.out.println("ht2_large.get(22):  " + ht2_large.get(22));
System.out.println("size of ht2_large =  " + ht2_large.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht2_large.isEmpty() =  " + ht2_large.isEmpty());

System.out.println("Performance(according to put() method in large-sized data) =  " + (end-start) + "ns");
```

Results: I used 11 as table size because it's a prime number and prime numbers are more efficient for hash-tables



*execution time of Hashing by using TreeSet(instead of linkedlist)
-small sized data: 670019 ns for put operation
-medium sized data: 657867 ns for put operation
- large sized data: 4229905 ns for put operation

Part-2.3: Test of Hashing by using Open Addressing with Quadratic Probing

I tested all methods of the OpenAddressing class. I added 10 keys and values for small-sized, 100 for medium-sized and 1000 for larged-sized. I also calculate the execution time with using large-sized data to compare the performance of hashing techniques.

Key-value pairs are like:

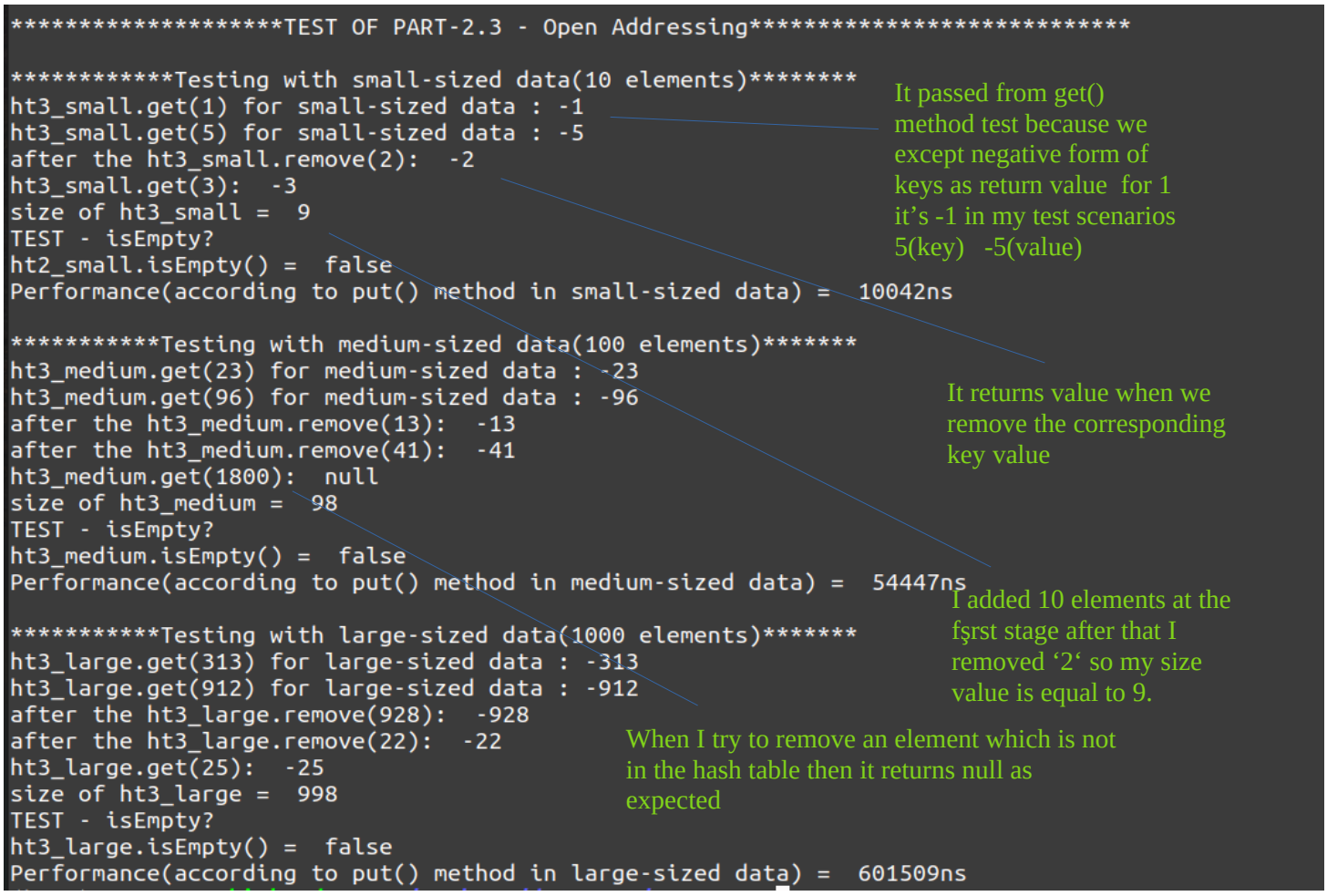
1 -1

2 -2

.

.

```
System.out.println("\n***********Testing with small-sized data(10 elements)********");
start = System.nanoTime();
for(t = 0,u = 0; t < 10 && u > -10; t++,u--)
    ht3_small.put(t,u);
end = System.nanoTime();

System.out.println("ht3_small.get(1) for small-sized data : " + ht3_small.get(1));
System.out.println("ht3_small.get(5) for small-sized data : " + ht3_small.get(5));

System.out.println("after the ht3_small.remove(2):  " + ht3_small.remove(2));
System.out.println("ht3_small.get(3):  " + ht3_small.get(3));
System.out.println("size of ht3_small =  " + ht3_small.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht2_small.isEmpty() =  " + ht3_small.isEmpty());
System.out.println("Performance(according to put() method in small-sized data) =  " + (end-start) + "ns");


System.out.println("\n***********Testing with medium-sized data(100 elements)*******");
start = System.nanoTime();
for(t = 0,u = 0; t < 100 && u > -100; t++,u--)
    ht3_medium.put(t,u);
end = System.nanoTime();

System.out.println("ht3_medium.get(23) for medium-sized data : " + ht3_medium.get(23));
System.out.println("ht3_medium.get(96) for medium-sized data : " + ht3_medium.get(96));

System.out.println("after the ht3_medium.remove(13):  " + ht3_medium.remove(13));
System.out.println("after the ht3_medium.remove(41):  " + ht3_medium.remove(41));
System.out.println("ht3_medium.get(1800):  " + ht3_medium.get(1800));
System.out.println("size of ht3_medium =  " + ht3_medium.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht3_medium.isEmpty() =  " + ht3_medium.isEmpty());
System.out.println("Performance(according to put() method in medium-sized data) =  " + (end-start) + "ns");

System.out.println("\n***********Testing with large-sized data(1000 elements)*******");
start = System.nanoTime();
for(t = 0,u = 0; t < 1000 && u > -1000; t++,u--)
    ht3_large.put(t,u);

end = System.nanoTime();

System.out.println("ht3_large.get(313) for large-sized data : " + ht3_large.get(313));
System.out.println("ht3_large.get(912) for large-sized data : " + ht3_large.get(912));

System.out.println("after the ht3_large.remove(928):  " + ht3_large.remove(928));
System.out.println("after the ht3_large.remove(22):  " + ht3_large.remove(22));
System.out.println("ht3_large.get(25):  " + ht3_large.get(25));
System.out.println("size of ht3_large =  " + ht3_large.size());
System.out.println("TEST - isEmpty?");
System.out.println("ht3_large.isEmpty() =  " + ht3_large.isEmpty());

System.out.println("Performance(according to put() method in large-sized data) =  " + (end-start) + "ns");
```



*execution time of Open addressing with quadratic probing
-small sized data: 10042 ns for put operation
-medium sized data: 54447ns for put operation
- large sized data: 601509 ns for put operation

*My program passed from all of the possible tests

According to performance results(based on put() method): (from best to worst)

OpenAddressing with quadratic probing> Hashing with LinkedList >Hashing with TreeSet