GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 6 Report

Emircan Demirel
1901042674

1. SYSTEM REQUIREMENTS

Functional Requirements:

Both two hashmap implementation should implement KWHashMap interface and it's methods. These methods are put, remove, get, size and isEmpty.

User can add new element by passing key and value parameters to put method.

User can search an element with the help of get method.

User can remove an element from the table by passing key parameter to remove method.

Colliding items should initialize to BST for Q1.1.

Colliding items should initialize another index on table for Q1.2.

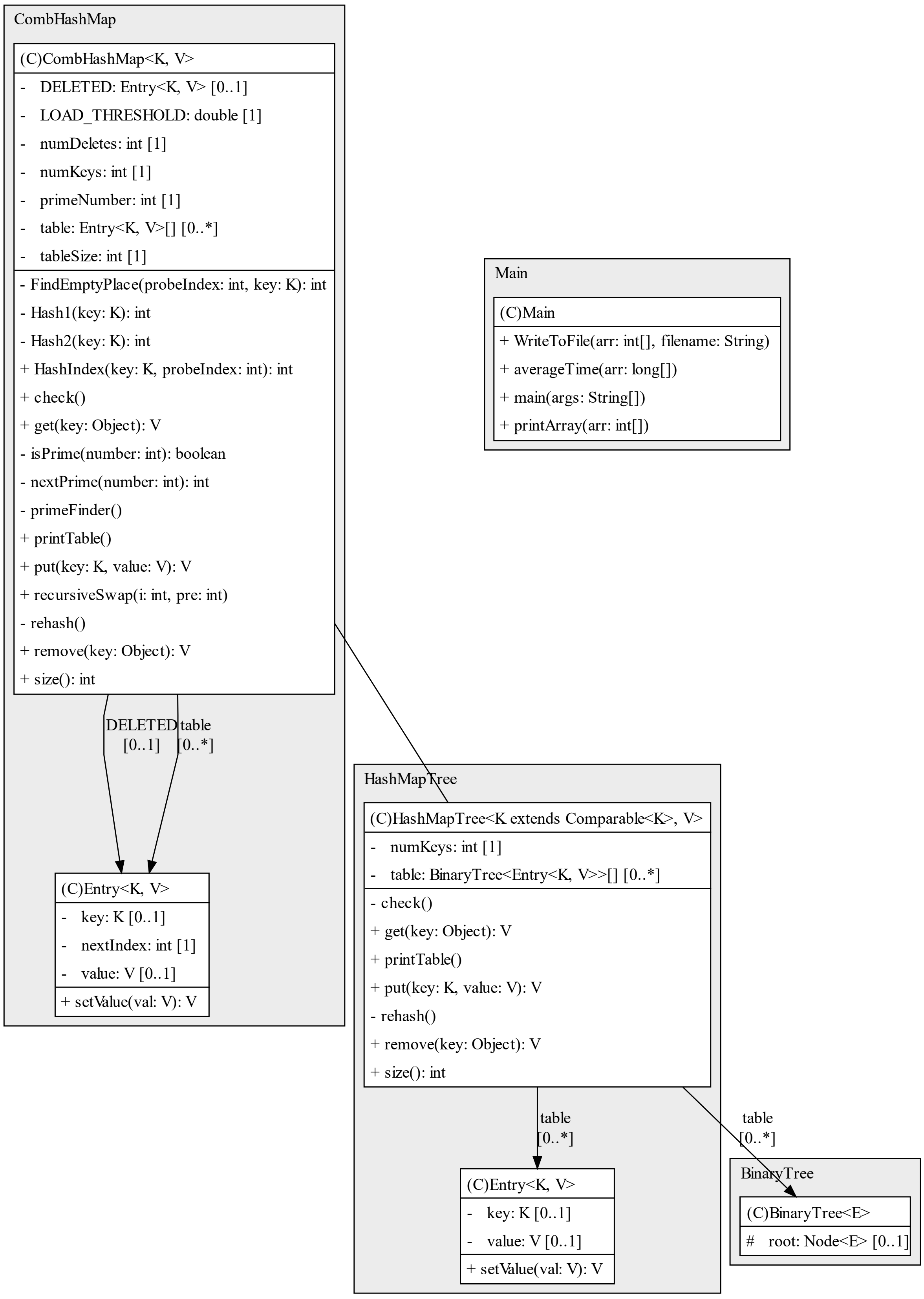
Non-Functional Requirements:

Program must handle errors and colliding items.

Min_max_finder method has to return two values at once.

Java-SE13 or higher should be installed and running properly.

2. USE CASE AND CLASS DIAGRAMS



3. PROBLEM SOLUTION APPROACH

Q1.1)

In this part of the assignment, We have to implement hashmap chain by using Binary Search Tree structure instead of List structure. So, to maintain this I create a BST array which named as table in HashMapTree class. My program initialize new elements to table according to their hash codes and create new BST at this location. If two elements are colliding on same location, new element added as child to BST.

To solve problem about removal method, I use lazy deletion technique so deleted object will leave with null data fields till rehash method calls. To find elements via get method I use CompareTo method, according to return value program will traverse in BST and try to find wanted Entry.

Q1.2)

In second part I have to combine coalesced hashing and double hashing techniques. To add new elements program will get an index value with the help of double hashing technic. If index position on the table is non-empty then program will increment probeIndex value and get new index value again. Next data field in every Entry object link this elements each other. We can easily find and remove any element on table, following and evaluating this link structure.

<u>COALESCED HASHING</u> is a collision avoidance technique when there is a fixed sized data. It is a combination of both Separate chaining and Open addressing. It uses the concept of Open Addressing (linear probing) to find first empty place for colliding element from the bottom of the hash table and the concept of Separate Chaining to link the colliding elements to each other through pointers. The hash function used is h=(key)% (total number of keys). Inside the hash table, each node has three fields:

- h(key): The value of hash function for a key.
- Data: The key itself.
- Next: The link to the next colliding elements.

The best-case complexity of all these operations is O (1) and the worst-case complexity is O(n) where n is the total number of keys. It is better than separate chaining because it inserts the colliding element in the memory of hash table only instead of creating a new linked list as in separate chaining.

<u>DOUBLE HASHING</u> is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Advantages of Double hashing

- The advantage of Double hashing is that it is one of the best form of probing, producing a uniform distribution of records throughout a hash table.
- This technique does not yield any clusters.
- It is one of effective method for resolving collisions.

Q2)

To return a pair I create an inner class in NewSort class. This Pair class includes two private data field MIN and MAX. When min_max_finder called by sort method, it affect and make changes on a Pair object’s data fields. Then based on these data field values, program will make swap operations to sort array.

Other two class are implementations of Merge and Quick Sort algorithms.

4. Analysis of Sorting Algorithms

Quick Sort Algorithm:

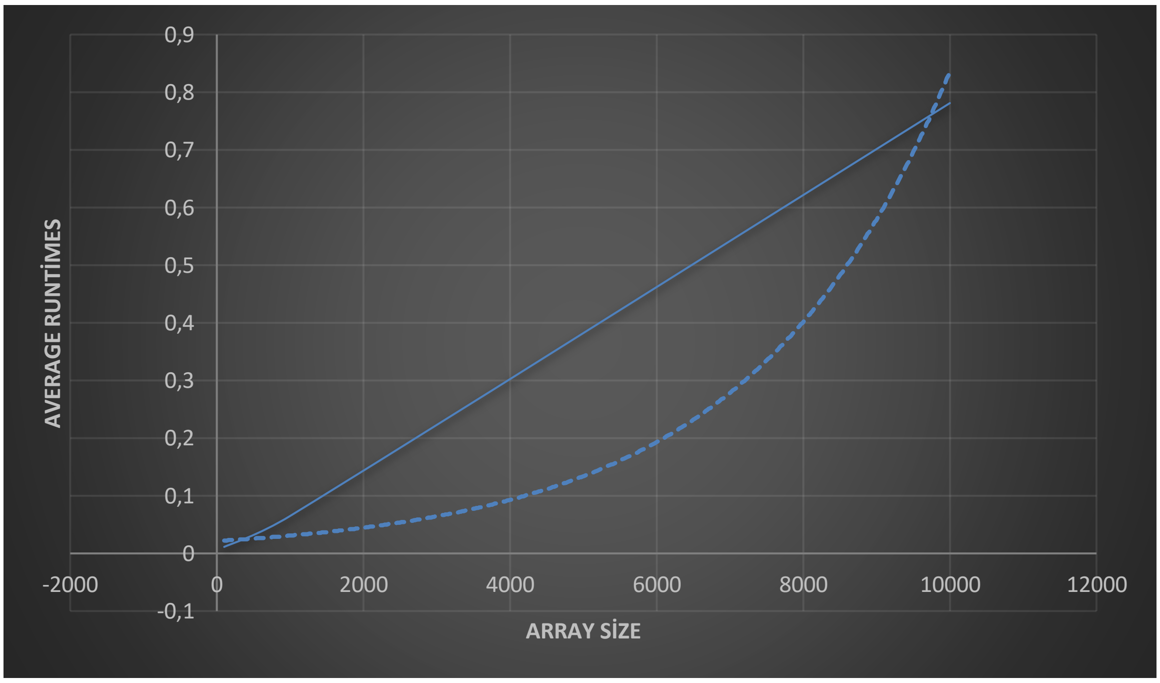
Time Complexity: O(n log n)

Runtime Results for different size array

Average sorting time for large sized (10000) arrays based on runtime of 1000 different arrays: *781086500 ns*

Average sorting time for medium sized (1000) arrays based on runtime of 1000 different arrays: *64993200 ns*

Average sorting time for small sized (100) arrays based on runtime of 1000 different arrays: *11345200 ns*



Runtime, exponentially increasing based on array size as expected in theoretical results.

Merge Sort Algorithm:

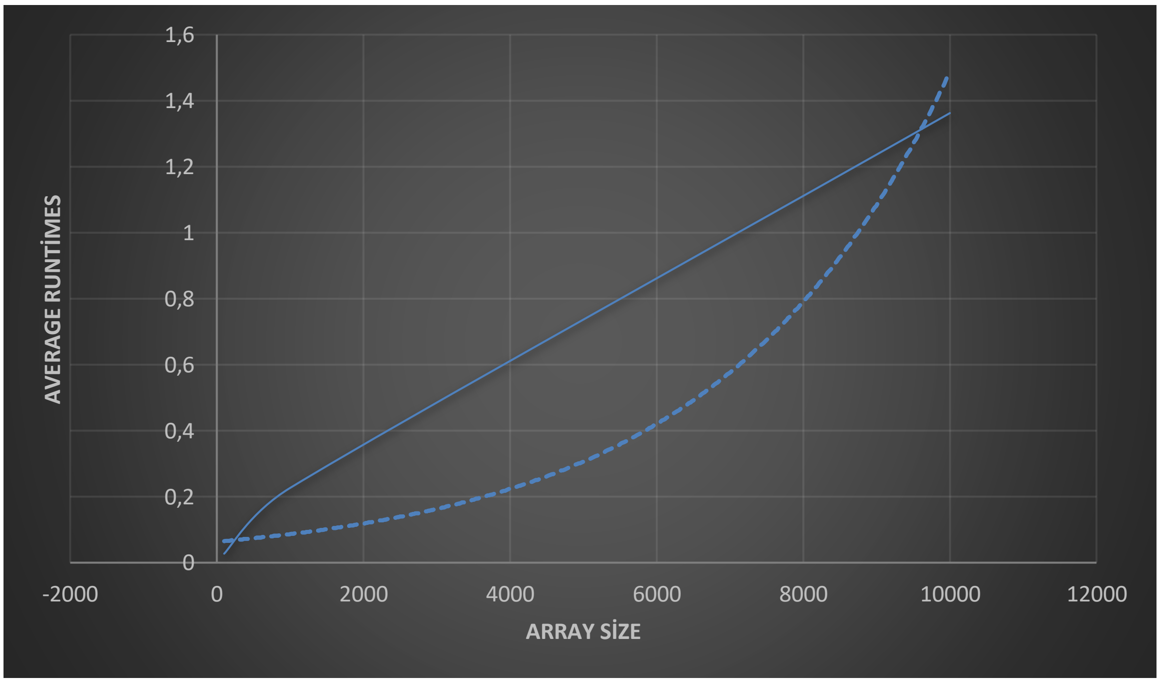
Time Complexity: O(n log n)

Runtime Results for different size array

Average sorting time for large sized (10000) arrays based on runtime of 1000 different arrays: *1362341300 ns*

Average sorting time for medium sized (1000) arrays based on runtime of 1000 different arrays: *226527000 ns*

Average sorting time for small sized (100) arrays based on runtime of 1000 different arrays: *27198800 ns*



Runtime, exponentially increasing based on array size as expected in theoretical results.

New Sort Algorithm:

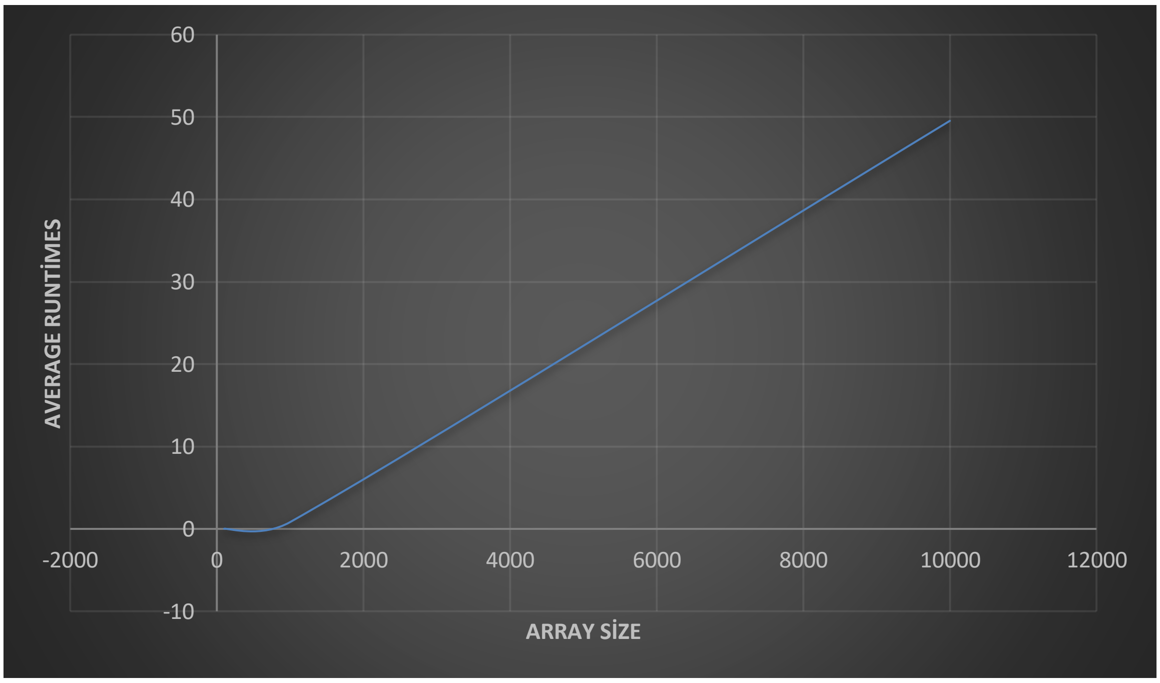
Time Complexity: θ(n^2)

Runtime Results for different size array

Average sorting time for large sized (10000) arrays based on runtime of 1000 different arrays: *49544064600 ns*

Average sorting time for medium sized (1000) arrays based on runtime of 1000 different arrays: *825810800 ns*

Average sorting time for small sized (100) arrays based on runtime of 1000 different arrays: *18384900 ns*



Runtime, quadratically increasing based on array size as expected in theoretical results.

5. TEST CASES

Q1.1)

1. Compile -> Test Put Method -> Add new element
2. Compile -> Test Put Method -> Change an existing element
3. Compile -> Test Get Method -> Search an Existing element on the table
4. Compile -> Test Get Method -> Search a non-existing element on the table
5. Compile -> Test Remove Method -> Delete an element from the table
6. Compile -> Test Remove Method -> try to Delete a non-existing element from the table

Q1.2)

1. Compile -> Test Put Method -> Add new element
2. Compile -> Test Put Method -> Change an existing element
3. Compile -> Test Get Method -> Search an Existing element on the table
4. Compile -> Test Get Method -> Search a non-existing element on the table
5. Compile -> Test Remove Method -> Delete an element from the table
6. Compile -> Test Remove Method -> Delete a linked element from the table
7. Compile -> Test Remove Method -> try to Delete a non-existing element from the table

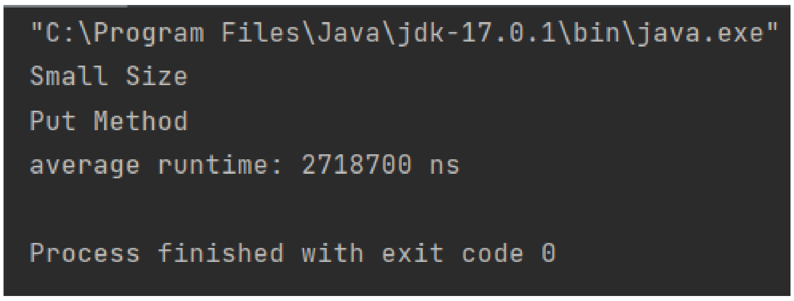
Q2)

1. Compile -> Display Unsorted Array -> Test QuickSort Algorithm
2. Compile -> Display Unsorted Array -> Test MergeSort Algorithm
3. Compile -> Display Unsorted Array -> Test NewSort Algorithm

6. RUNNING AND RESULTS

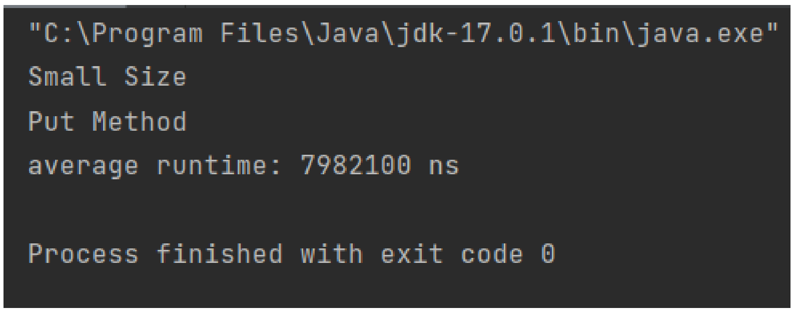
Q1.1)

Average runtime of put method for small sized arrays:



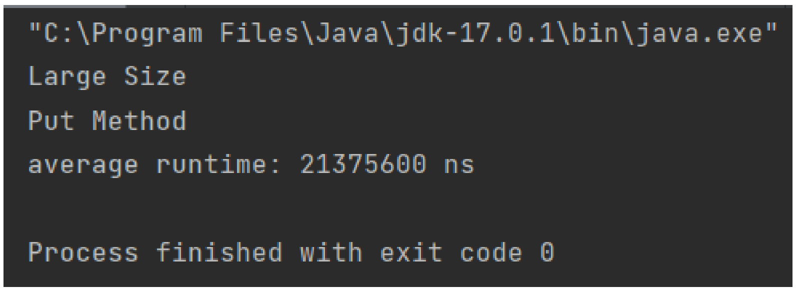
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
Small Size
Put Method
average runtime: 2718700 ns

Process finished with exit code 0
```

Average runtime of put method for medium sized arrays:



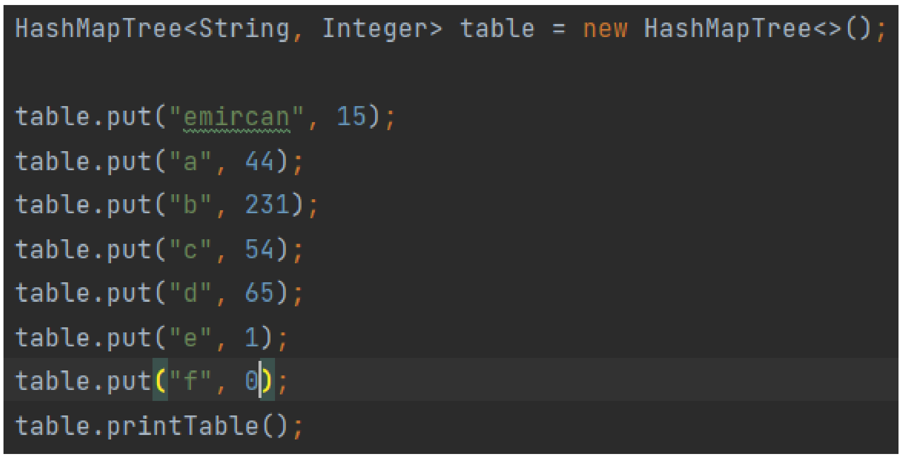
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
Small Size
Put Method
average runtime: 7982100 ns

Process finished with exit code 0
```

Average runtime of put method for large sized arrays:

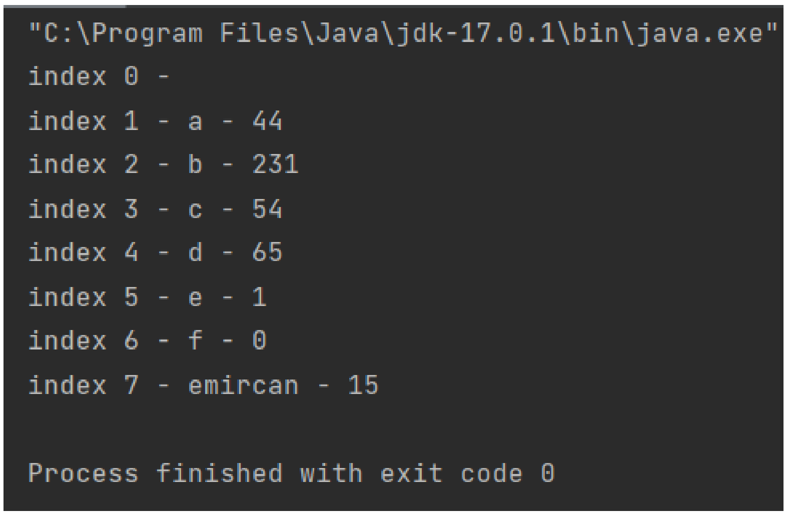


```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
Large Size
Put Method
average runtime: 21375600 ns

Process finished with exit code 0
```

1. Compile -> Test Put Method -> Add new element

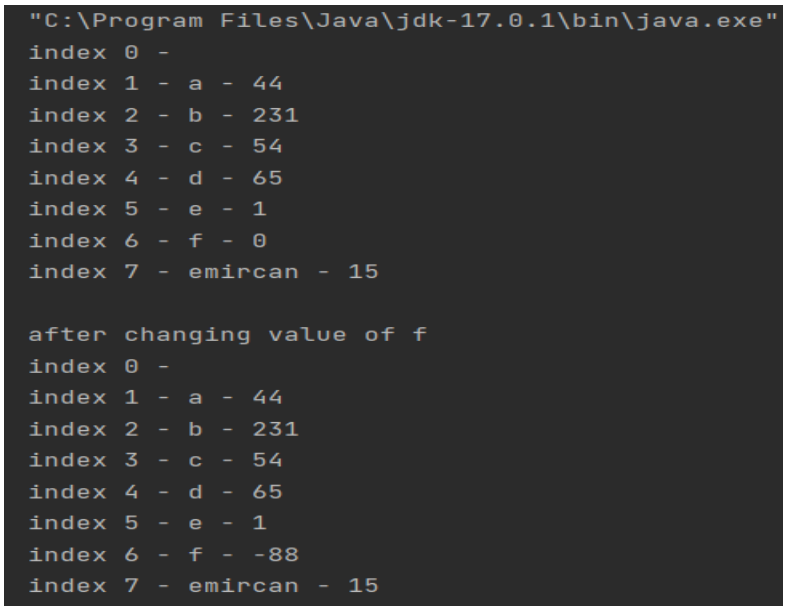


```java
HashMapTree<String, Integer> table = new HashMapTree<>();

table.put("emircan", 15);
table.put("a", 44);
table.put("b", 231);
table.put("c", 54);
table.put("d", 65);
table.put("e", 1);
table.put("f", 0);
table.printTable();
```



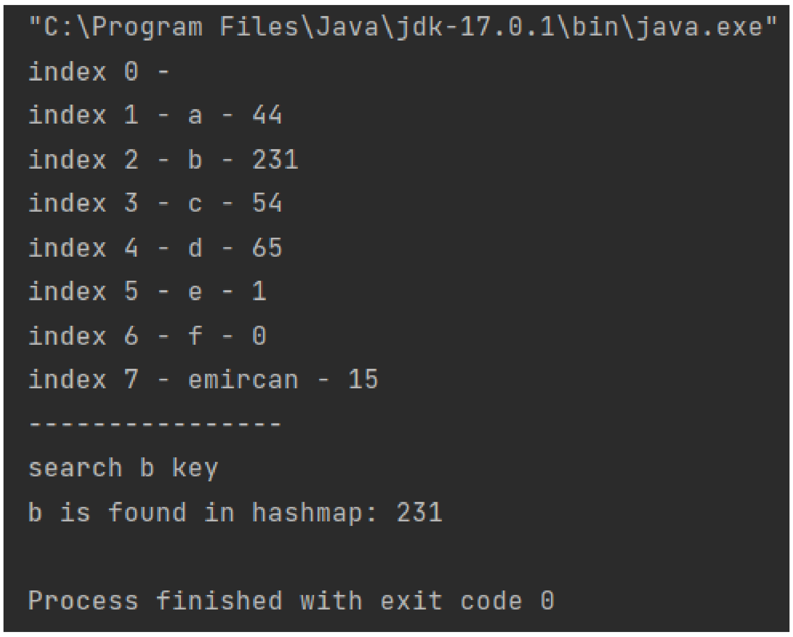
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
index 0 -
index 1 - a - 44
index 2 - b - 231
index 3 - c - 54
index 4 - d - 65
index 5 - e - 1
index 6 - f - 0
index 7 - emircan - 15

Process finished with exit code 0
```

2. Compile -> Test Put Method -> Change an existing element



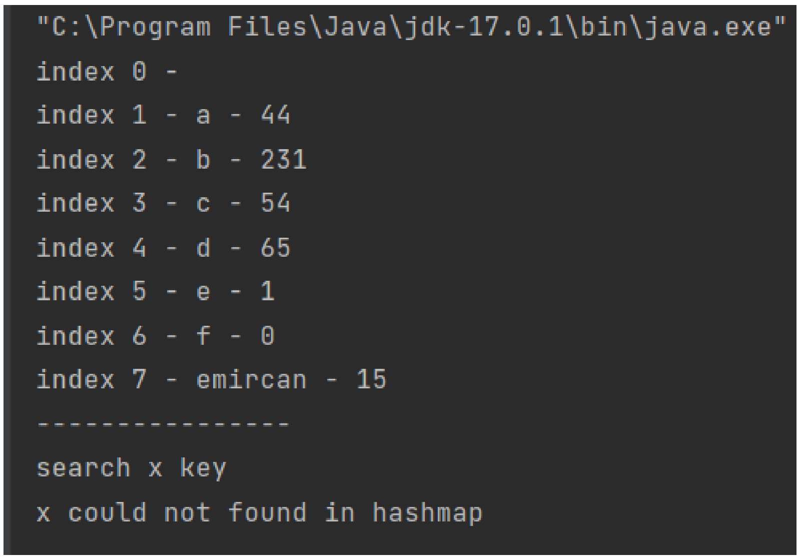
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
index 0 -
index 1 - a - 44
index 2 - b - 231
index 3 - c - 54
index 4 - d - 65
index 5 - e - 1
index 6 - f - 0
index 7 - emircan - 15

after changing value of f
index 0 -
index 1 - a - 44
index 2 - b - 231
index 3 - c - 54
index 4 - d - 65
index 5 - e - 1
index 6 - f - -88
index 7 - emircan - 15
```

3. Compile -> Test Get Method -> Search an Existing element on the table



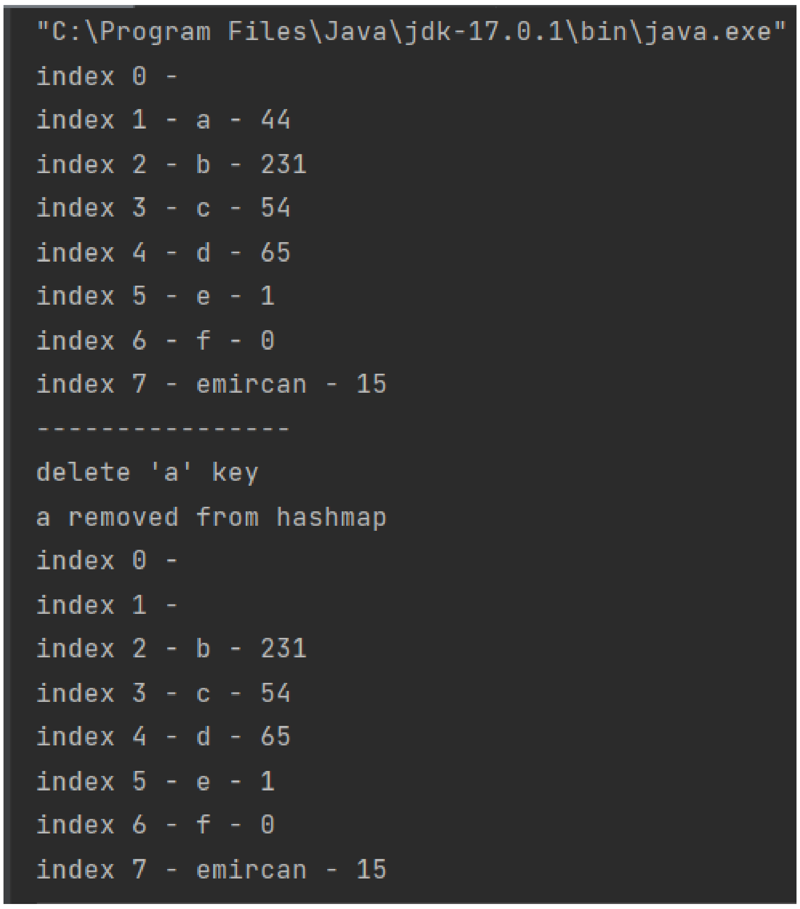
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
index 0 -
index 1 - a - 44
index 2 - b - 231
index 3 - c - 54
index 4 - d - 65
index 5 - e - 1
index 6 - f - 0
index 7 - emircan - 15
----------------
search b key
b is found in hashmap: 231

Process finished with exit code 0
```

4. Compile -> Test Get Method -> Search a non-existing element on the table



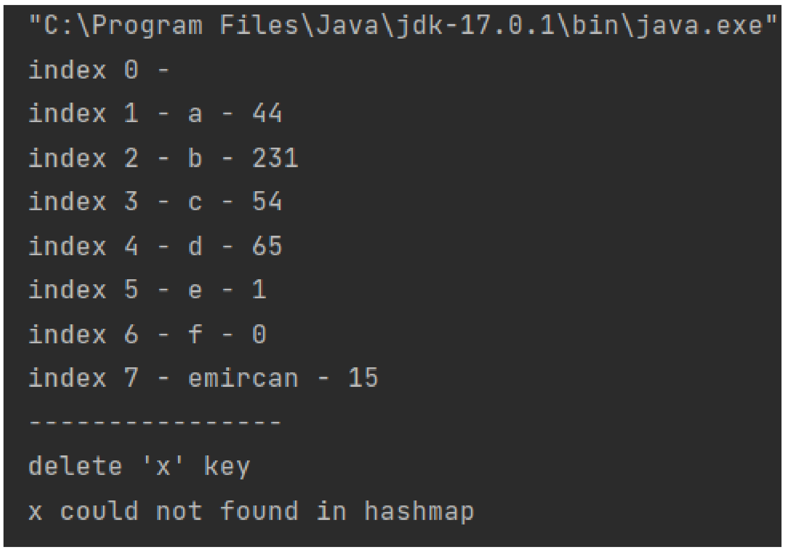
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
index 0 -
index 1 - a - 44
index 2 - b - 231
index 3 - c - 54
index 4 - d - 65
index 5 - e - 1
index 6 - f - 0
index 7 - emircan - 15
----------------
search x key
x could not found in hashmap
```

5. Compile -> Test Remove Method -> Delete an element from the table



```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
index 0 -
index 1 - a - 44
index 2 - b - 231
index 3 - c - 54
index 4 - d - 65
index 5 - e - 1
index 6 - f - 0
index 7 - emircan - 15
----------------
delete 'a' key
a removed from hashmap
index 0 -
index 1 -
index 2 - b - 231
index 3 - c - 54
index 4 - d - 65
index 5 - e - 1
index 6 - f - 0
index 7 - emircan - 15
```

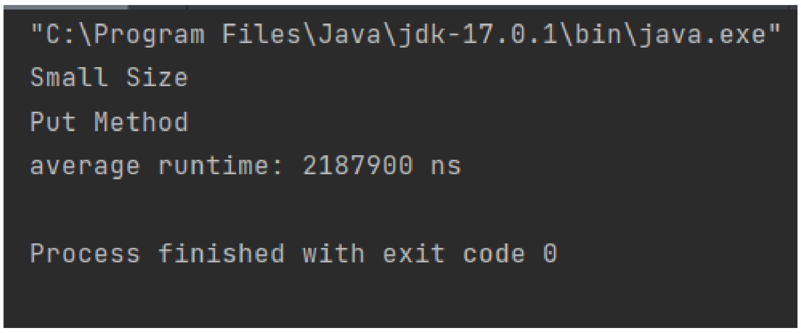
6. Compile -> Test Remove Method -> try to Delete a non-existing element from the table



```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
index 0 -
index 1 - a - 44
index 2 - b - 231
index 3 - c - 54
index 4 - d - 65
index 5 - e - 1
index 6 - f - 0
index 7 - emircan - 15
----------------
delete 'x' key
x could not found in hashmap
```

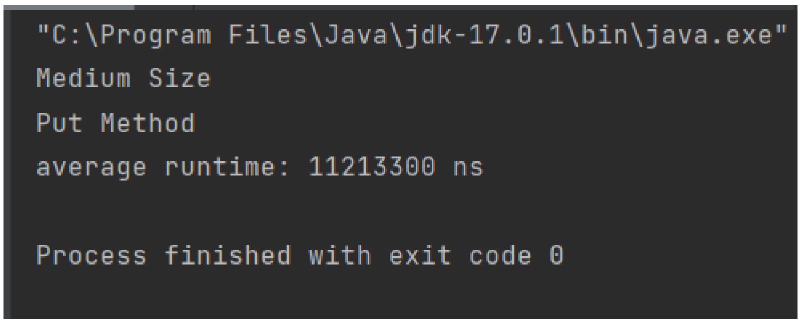
Q1.2)

Average runtime of put method for small sized arrays:



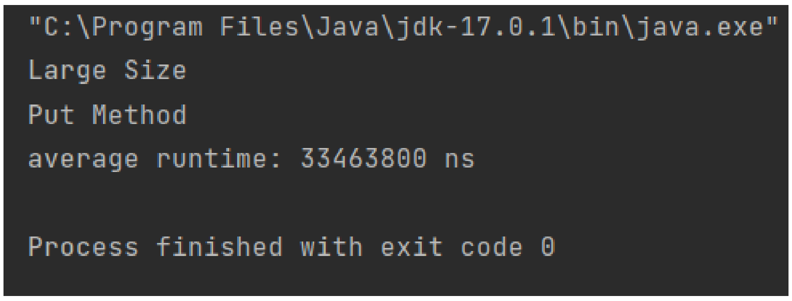
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
Small Size
Put Method
average runtime: 2187900 ns

Process finished with exit code 0
```

Average runtime of put method for medium sized arrays:



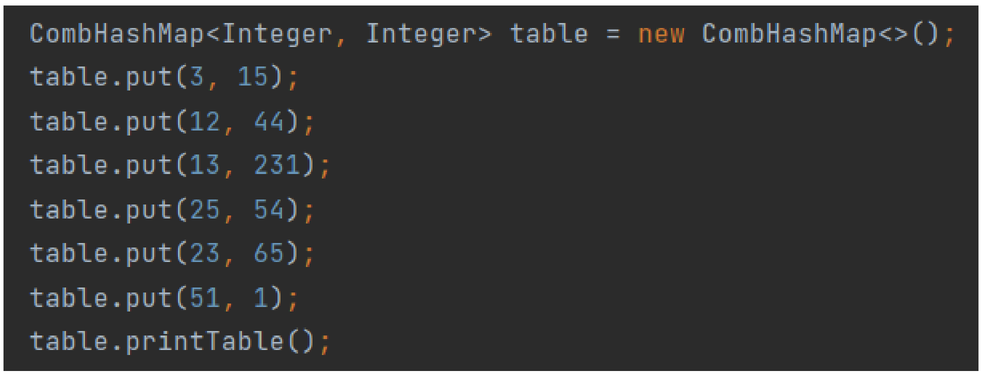
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
Medium Size
Put Method
average runtime: 11213300 ns

Process finished with exit code 0
```

Average runtime of put method for large sized arrays:

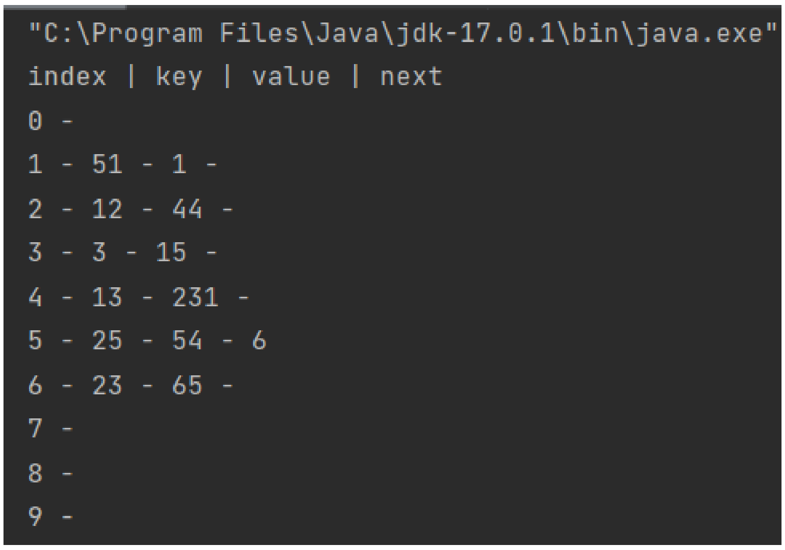


```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
Large Size
Put Method
average runtime: 33463800 ns

Process finished with exit code 0
```

1. Compile -> Test Put Method -> Add new element

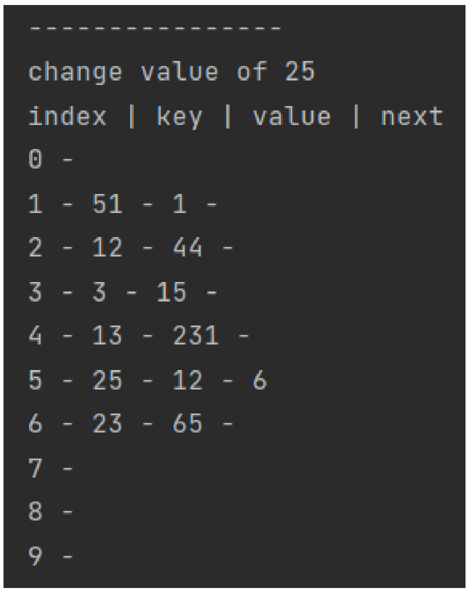


```
CombHashMap<Integer, Integer> table = new CombHashMap<>();
table.put(3, 15);
table.put(12, 44);
table.put(13, 231);
table.put(25, 54);
table.put(23, 65);
table.put(51, 1);
table.printTable();
```



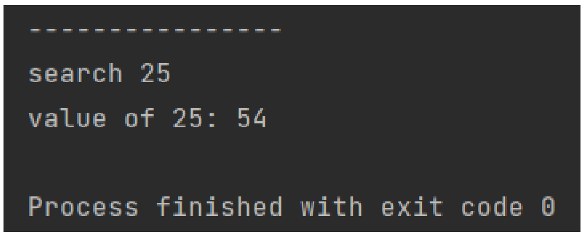
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
index | key | value | next
0 -
1 - 51 - 1 -
2 - 12 - 44 -
3 - 3 - 15 -
4 - 13 - 231 -
5 - 25 - 54 - 6
6 - 23 - 65 -
7 -
8 -
9 -
```

2. Compile -> Test Put Method -> Change an existing element



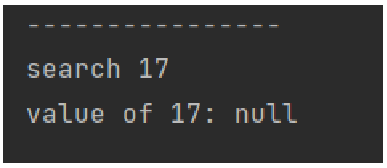
```
----------------
change value of 25
index | key | value | next
0 -
1 - 51 - 1 -
2 - 12 - 44 -
3 - 3 - 15 -
4 - 13 - 231 -
5 - 25 - 12 - 6
6 - 23 - 65 -
7 -
8 -
9 -
```

3. Compile -> Test Get Method -> Search an Existing element on the table



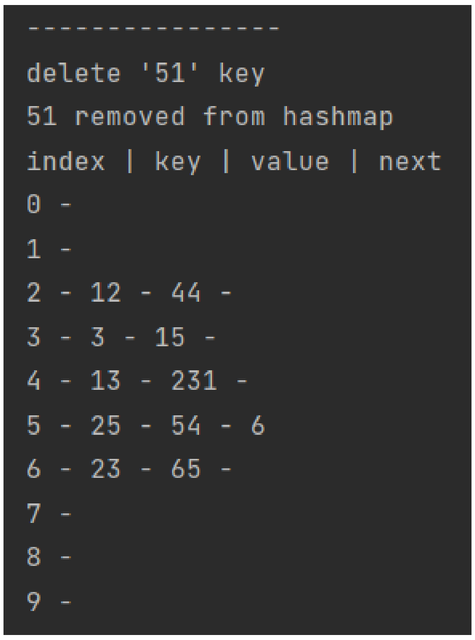
```
----------------
search 25
value of 25: 54

Process finished with exit code 0
```

4. Compile -> Test Get Method -> Search a non-existing element on the table



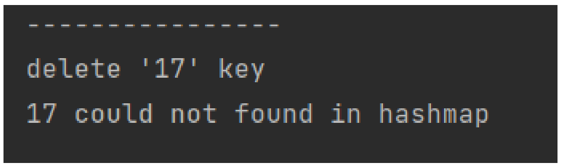
```
----------------
search 17
value of 17: null
```

5. Compile -> Test Remove Method -> Delete an element from the table



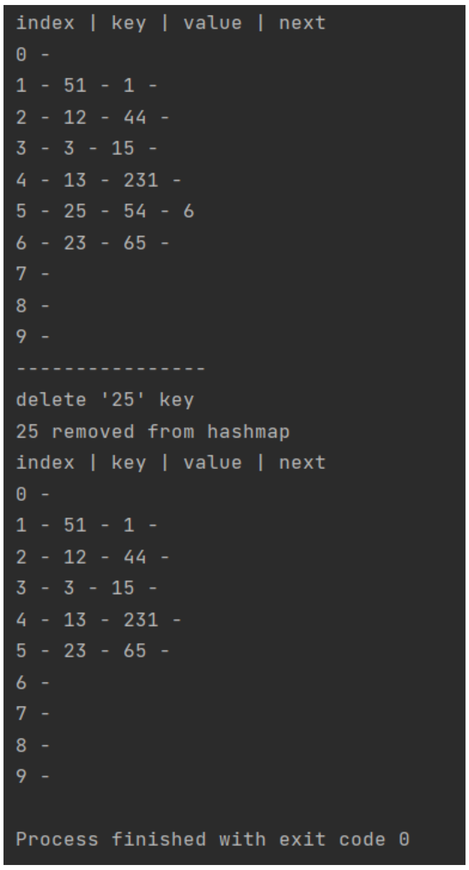
```
----------------
delete '51' key
51 removed from hashmap
index | key | value | next
0 -
1 -
2 - 12 - 44 -
3 - 3 - 15 -
4 - 13 - 231 -
5 - 25 - 54 - 6
6 - 23 - 65 -
7 -
8 -
9 -
```

6. Compile -> Test Remove Method -> Delete a non-existing element



```
----------------
delete '17' key
17 could not found in hashmap
```

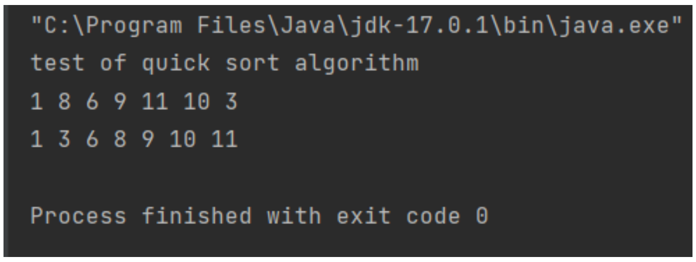
7. Compile -> Test Remove Method -> Delete a linked element from the table



```
index | key | value | next
0 -
1 - 51 - 1 -
2 - 12 - 44 -
3 - 3 - 15 -
4 - 13 - 231 -
5 - 25 - 54 - 6
6 - 23 - 65 -
7 -
8 -
9 -
----------------
delete '25' key
25 removed from hashmap
index | key | value | next
0 -
1 - 51 - 1 -
2 - 12 - 44 -
3 - 3 - 15 -
4 - 13 - 231 -
5 - 23 - 65 -
6 -
7 -
8 -
9 -

Process finished with exit code 0
```

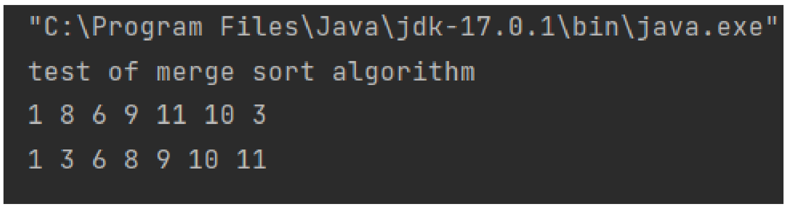
Q2)

1. Compile -> Display Unsorted Array -> Test QuickSort Algorithm



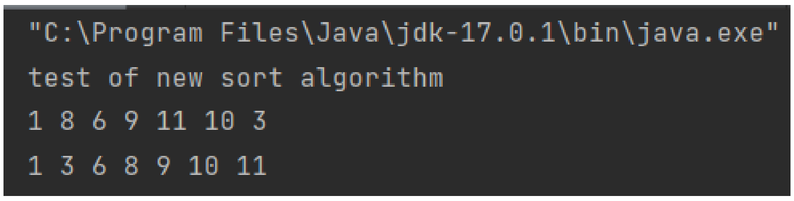
```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
test of quick sort algorithm
1 8 6 9 11 10 3
1 3 6 8 9 10 11

Process finished with exit code 0
```

2. Compile -> Display Unsorted Array -> Test MergeSort Algorithm



```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
test of merge sort algorithm
1 8 6 9 11 10 3
1 3 6 8 9 10 11
```

3. Compile -> Display Unsorted Array -> Test NewSort Algorithm



```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe"
test of new sort algorithm
1 8 6 9 11 10 3
1 3 6 8 9 10 11
```