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

Mert Gürşimşir
1901042646

PART 1

COALESCED HASHING

Coalesced hashing is another strategy of collision resolution in a hash table. With this coalesced hashing, we hold our key-value pairs in the hash table and also we link elements to the next element. In coalesced hashing, if we find the index of our element but that index is not empty, than we look the table starting from its end and go through up (through beginning of the array) until we find an empty slot. At each probe, we link table elements.

Advantages - Disadvantages

Coalesced hashing has advantage on searching the elements thanks to the links that connects elements which create collision. Also this hashing strategy avoid primary clustering as in open addressing. If the chains, which are created by links, are short then coalesced hashing is very efficient.

On the other hand, removal is very expensive because we have to create links again. Also rehashing is very expensive again because of the same reason. In rehashing, we have to put every item at the table linearly and also we have to create links.

All in all, coalesced hashing is good for searching over standard open addressing and chaining methods but bad for rehashing and removal.

DOUBLE HASHING

Double hashing is another strategy of collision resolution in a hash table. This strategy includes 2 hashing methods to key and add them and lastly takes mod with table size for the proper index in the table. If a collision occurs, then 2^{nd} hash will be multiplied by 2, and then 3, and then 4 etc.
First hash → hash1 = key % table_size
Second hash → hash2 = prime - (key % prime)

Advantages - Disadvantages

Double hashing can find empty slot for the given key-value pair effectively (it requires less comparisons) thanks to hashing algorithms. Also this double hashing algorithm prevents clustering because probes are different for most of the elements thanks to 2 separate hashing methods.

On the other hand, if the table is big, double hashing has big intervals between collided elements. So cache performance can be considered as bad. Also this strategy is more complicated because of 2 hashing methods.

All in all, double hashing is good for prevention of clusterings but bad for complication.

MERGE SORT

At each part of the split the array, we have to move n elements from small subarray to large array. So time requirement for this operation is θ(n).

Also the number of lines at merge operation is logn because at each recursive step, we split the array in half.

All in all, total time requirement for sorting the array with merge sort is θ(nlogn).

QUICK SORT

If the array is already sorted (each split yields one empty subarray), then quick sort is $\theta(n^2)$ which is worst case.

If the pivot is random, then roughly half of the items in the array is less than the pivot and half of the array is larger than the pivot. If both subarrays always have same number of elements, then there is logn recursions. Also at each level we move every element to correct partition (n). So best casae for quick sort is θ(nlogn).

This 2 makes quick sort $O(n^2)$.

For average case, quick sort is θ(nlogn). Constant for this running time is less than merge sort so quick sort is actually "quick"er than merge sort.

NEW_SORT

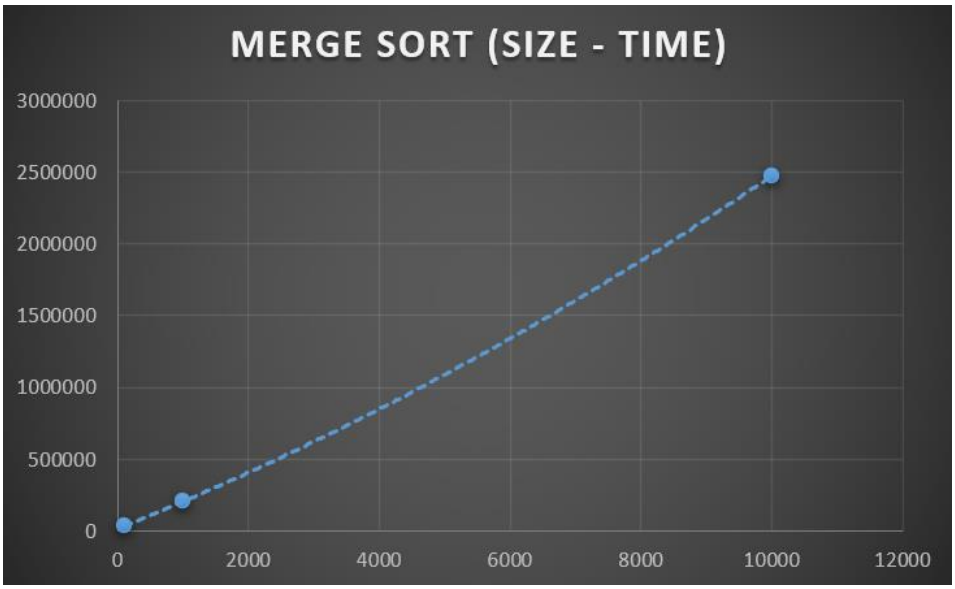
In min_max_finder method, problem is divided into 2 half parts. Remainings (other than recursive call) in the method is constant time:

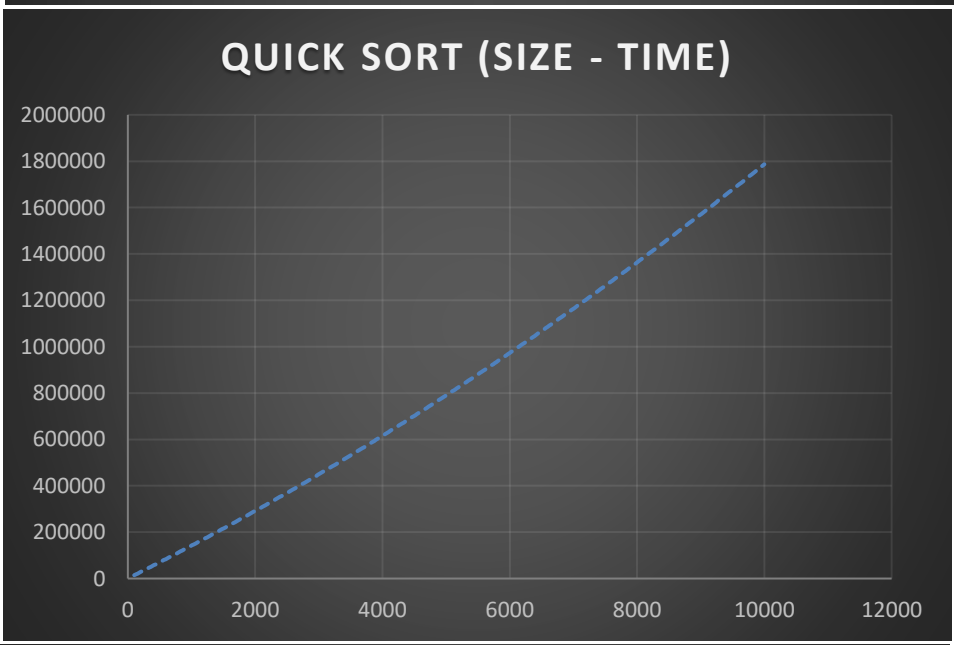
T(n) = 2T(n/2) + θ(1)
T(n) = 2(2T(n/4) + θ(1)) + θ(1) = 2^2T(n/4) + 3θ(1)
T(n) = 2^2[2T(n/8) + θ(1)] + 3θ(1) = 2^3T(n/8) + 7θ(1)
T(n) = 2^3[2T(n/16) + θ(1)] + 7θ(1) = 2^4T(n/2^4) + 2^3θ(1) +
2^2θ(1) + 2^1θ(1) + 2^0θ(1)
T(n) = 2^5T(n/2^5) + 2^4θ(1) + 2^3θ(1) + 2^2θ(1) + 2^1θ(1) + 2^0θ(1)
…
T(n) = 2^kT(n/2^k) + 2^{k-1}θ(1) + 2^{k-2}θ(1) + … + 2^0θ(1)
2^k = n , k = $log_2 n$:
T(n) = nT(1) + (2^k - 1)θ(1) = nT(1) + (n - 1)θ(1) (T(1) = θ(1))
T(n) = n + n - 1 = 2n - 1
T(n) = θ(n)

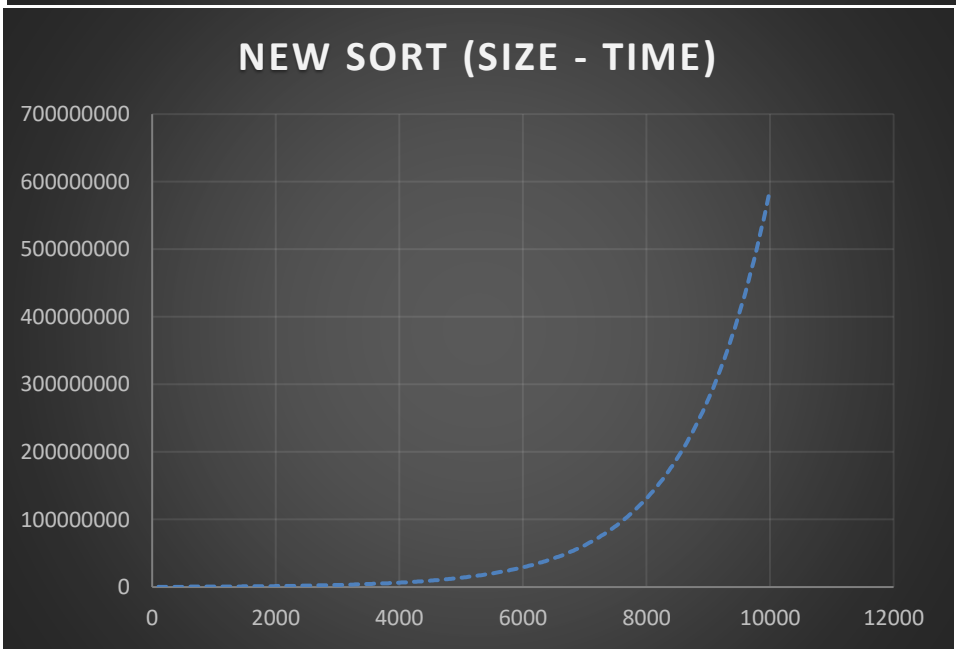
In new_sort method, we have 1 recursive call + n (n comes from method min_max_finder):

T(n) = T(n-2) + n
T(n) = T(n-4) + (n-1) + n - 1
T(n) = T(n-6) + (n-2) + (n-1) + n - 3
…
T(n) = $\theta(n^2)$

SIZE	TIME - MERGE SORT	TIME - QUICK SORT	TIME - NEW SORT
100	33569 ns	14348 ns	60090 ns
1000	205112 ns	141950 ns	4743809 ns
10000	2475147 ns	1786712 ns	493621262 ns







PART 2

1. SYSTEM REQUIREMENTS

To use hash table which uses binary search trees to chain items mapped on the same table slot, first you need to create hash table as :

BSTHashing<firstType, secondType> tableName = new BSTHashing<firstType,secondType>(capacity);

Example:

```
BSTHashing<Integer, String> tester = new BSTHashing<Integer, String>(7);
```

To use hash table which uses combination of the double hashing and coalesced hashing techniques, first you need to create hash table as :

HybridHash<firstType, secondType> tableName = new HybridHash<firstType,secondType>(capacity);

```
HybridHash<Integer, String> tester = new HybridHash<Integer, String>(10);
```

If no capacity value is used both for BSTHashing and HybridHash, 11 (prime) will be used as default capacity.

Below methods can be used in the same way both with BSTHashing and HybridHash

You can insert entries to table with put command:

```
tester.put(key, value);
```

You can remove entries from table with remove command:

```
tester.remove(13);
```

You can get an entry with the key:

```
tester.get(16)
```

You can check if the table is empty with isEmpty method:

```
tester.isEmpty();
```

You can learn the size of the table with size method:

```
tester.size();
```

To use sorting algorithms, you need to have an array. As an example:

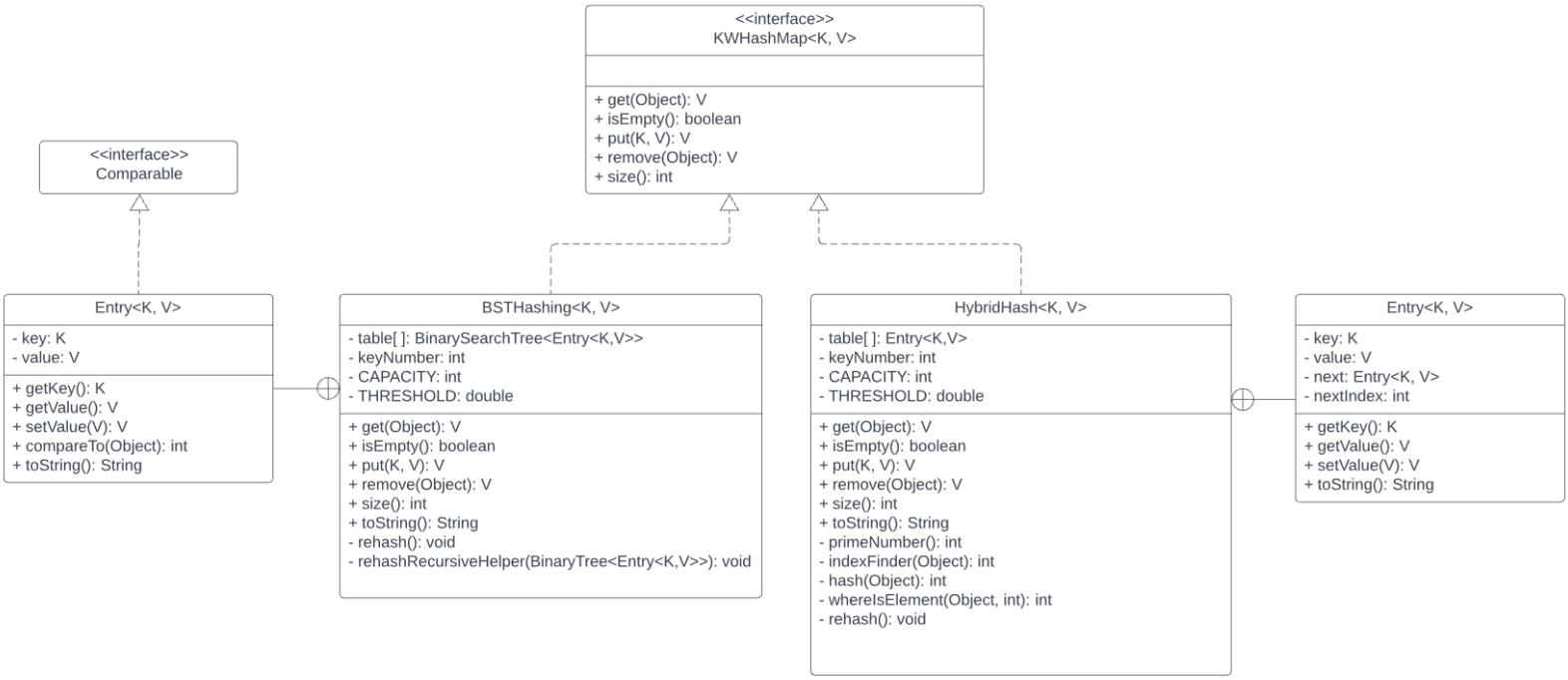
```
Integer arrayTest1[] = {-123, 1234, 1, 140, 4, 5, 3, 0, -1, 32, 4, 13, -12312, 12};
```

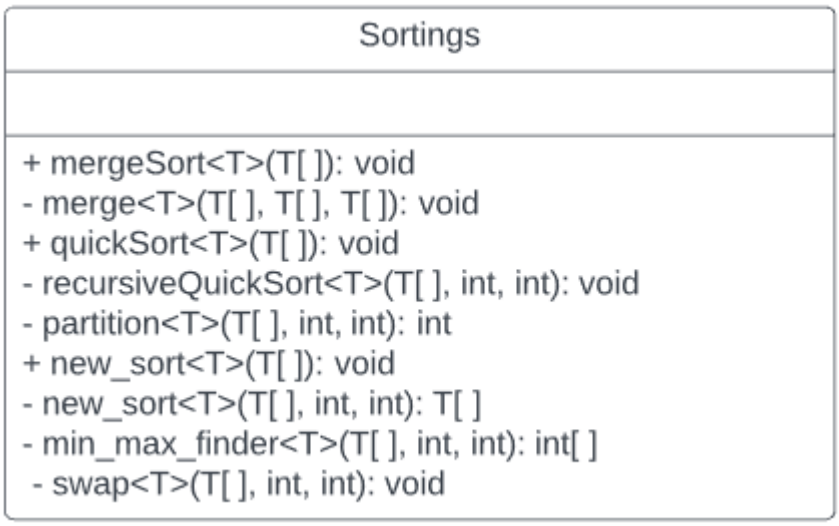
*****Primitive type arrays are not allowed!**

Then you can use whichever sorting method you want as shown below:

```
Sortings.mergeSort(arrayTest1);
Sortings.quickSort(arrayTest1);
Sortings.new_sort(arrayTest1);
```

2. CLASS DIAGRAMS





3. PROBLEM SOLUTION APPROACH

In the first part of the first question, I have started with implementing the binary search tree with the help of the notes I have taken in the lessons. We have implemented that tree in the lesson so it was not hard for me. Then I have implemented the methods in the KWHashMap interface in my BSTHashing class with using a binary search tree array as a field. I have decided to make inner Entry class comparable so that I can easily put elements in the binary search tree with comparing Entries according to key.

In the second part of the first question, I have started with searching coalesced and double hashing techniques deeply. After I had understood the concept, I have started to implement hybrid hashing class. Firstly, I have decided how inner Entry class will be. I have added link to next element in Entry class and also use index of next entry to make things easier. Then I have implemented hashing methods first and then put method. Remove and rehash methods were the hardest parts to implement for me because all links have to be reconsidered and it took so much time to decide the algorithm.

In the second question, I have started with implementing the merge and quick sort with the help of the notes I have taken in the lessons. We have implemented these methods in the class so implementations were not hard. Then I have implemented the new_sort method using the pseudocode in the pdf. Hardest part for me in this question is to implement min_max_finder because I couldn't decide the efficient algorithm easily. After I had managed to divide the problem into two almost equal size subproblems, it was easy to implement this algorithm.

4. TEST CASES

TEST FOR HASHING WITH BINARY SEARCH TREE

```
System.out.printf("TEST OF THE HASHING WITH BST\n");
System.out.printf("---------------------------\n");
System.out.printf("VALIDATION TEST WITH FILE \"pairs.txt\"\n\n");
System.out.printf("After adding elements in the file to the hash table with size 7:\n\n");
BSTHashing<Integer, String> tester = new BSTHashing<Integer, String>(7);
readFile(tester);
System.out.printf("%s", tester);

System.out.printf("Let's get value of the key 16: %s\n", tester.get(16));
System.out.printf("Let's get value of the key 13: %s\n", tester.get(13));

String temp = tester.remove(13);
System.out.printf("\nLet's remove non-existing key (13).\nReturned element of remove(13): %s\n", temp);
temp = tester.remove(44);
System.out.printf("\nLet's remove existing key (44).\nReturned element of remove(44): %s\n", temp);
tester.remove(12); tester.remove(17); tester.remove(6); tester.remove(15);
tester.put(23, "changedValue");
System.out.printf("\nLet's remove some other keys (12, 17, 6, 15) and change value of 23.\n");
System.out.printf("\nTable after these removals:\n\n%s", tester);

for (int i = 100; i < 111; ++i) tester.put(i, "toExceedThreshold");
System.out.printf("\nLet's try to exceed threshold with adding more dummy elements. Table after rehash:\n\n%s", tester);
```

readFile reads the key-value pairs in the pairs.txt and insert them to file:

```
public static void readFile(BSTHashing<Integer, String> tester) throws FileNotFoundException{
    File pairs = new File("pairs.txt");
    Scanner reader = new Scanner(pairs);
    while (reader.hasNext()){
        Integer key = Integer.parseInt(reader.next());
        String value = reader.next();
        tester.put(key, value);
    }
}
```

pairs.txt:

```
0 computer
15 cup
7 book
23 dart
30 vertigo
19 ship
1 yatch
12 bird
17 dance
44 secret
38 cactus
16 insomnia
6 level
20 again
9 cosmos
10 chess
44 redesign
```

```java
System.out.printf("\n\nPERFORMANCE TEST\n\n");
BSTHashing<Integer, Integer> tester2 = new BSTHashing<Integer, Integer>();

long totalTime = 0;
for (int i = 0; i < 100; ++i){
    long start = System.nanoTime();
    for (int j = 0; j < 100; ++j){
        Integer nextInt = (int)(32000*Math.random());
        tester2.put(nextInt, nextInt);
    }
    long end = System.nanoTime();
    totalTime += (end-start);
    tester2 = new BSTHashing<Integer, Integer>();
}
System.out.println("Average time for 100 (size = 100)   tables: " + (totalTime / 100) + " ns\n");

totalTime = 0;
for (int i = 0; i < 100; ++i){
    long start = System.nanoTime();
    for (int j = 0; j < 1000; ++j){
        Integer nextInt = (int)(32000*Math.random());
        tester2.put(nextInt, nextInt);
    }
    long end = System.nanoTime();
    totalTime += (end-start);
    tester2 = new BSTHashing<Integer, Integer>();
}
System.out.println("Average time for 100 (size = 1000)  tables: " + (totalTime / 100) + " ns\n");

totalTime = 0;
for (int i = 0; i < 100; ++i){
    long start = System.nanoTime();
    for (int j = 0; j < 10000; ++j){
        Integer nextInt = (int)(32000*Math.random());
        tester2.put(nextInt, nextInt);
    }
    long end = System.nanoTime();
    totalTime += (end-start);
    tester2 = new BSTHashing<Integer, Integer>();
}
System.out.println("Average time for 100 (size = 10000) tables: " + (totalTime / 100) + " ns\n");
```

TEST FOR HASHING WITH COALESCED AND DOUBLE HASHING

```
System.out.printf("TEST OF THE HYBRID HASHING\n");
System.out.printf("-------------------------\n");
System.out.printf("VALIDATION TEST WITH ELEMENTS IN THE PDF\n\n");
HybridHash<Integer, String> tester = new HybridHash<Integer, String>(10);
tester.put(3, "valueOf3");
tester.put(12, "valueOf12");
tester.put(13, "valueOf13");
tester.put(25, "valueOf25");
tester.put(23, "valueOf23");
tester.put(51, "valueOf51");

System.out.printf("After adding elements to the hash table with size 10 (index --> {key - value} | NEXT : {next entry}):\n\n");
System.out.printf("%s", tester);

System.out.printf("\nLet's get value of the key 12: %s\n", tester.get(12));
System.out.printf("Let's get value of the key 40: %s\n", tester.get(40));

String temp = tester.remove(40);
System.out.printf("\nLet's remove non-existing key (40).\nReturned element of remove(40): %s\n", temp);
temp = tester.remove(25);
System.out.printf("\nLet's remove existing key (25).\nReturned element of remove(25): %s\n", temp);
System.out.printf("\nTable after these removals:\n\n%s", tester);
temp = tester.remove(23);
System.out.printf("\nLet's remove another existing key (23).\nReturned element of remove(23): %s\n", temp);
System.out.printf("\nTable after this removal:\n\n%s", tester);

tester.put(25, "toExceedThreshold"); tester.put(23, "toExceedThreshold"); tester.put(31, "toExceedThreshold"); tester.put(32, "toExceedThreshold");
System.out.printf("\nLet's try to exceed threshold with adding more dummy elements. Table after rehash:\n\n%s", tester);

tester.remove(51);
System.out.printf("\nLet's remove key 51.\n");
System.out.printf("\nTable after this removal:\n\n%s", tester);
tester.remove(13);
System.out.printf("\nLet's remove key 13.\n");
System.out.printf("\nTable after this removal:\n\n%s", tester);
tester.remove(32);
System.out.printf("\nLet's remove key 32.\n");
System.out.printf("\nTable after this removal:\n\n%s", tester);
tester.remove(25);
tester.put(12, "changedValue");
System.out.printf("\nLet's remove key 25 and change value of key 12.\n");
System.out.printf("\nTable:\n\n%s", tester);
```

```
System.out.printf("\n\nPERFORMANCE TEST\n\n");
HybridHash<Integer, Integer> tester2 = new HybridHash<Integer, Integer>();

long totalTime = 0;
for (int i = 0; i < 100; ++i){
    long start = System.nanoTime();
    for (int j = 0; j < 100; ++j){
        Integer nextInt = (int)(32000*Math.random());
        tester2.put(nextInt, nextInt);
    }
    long end = System.nanoTime();
    totalTime += (end-start);
    tester2 = new HybridHash<Integer, Integer>();
}
System.out.println("Average time for 100 (size = 100)   tables: " + (totalTime / 100) + " ns\n");

totalTime = 0;
for (int i = 0; i < 100; ++i){
    long start = System.nanoTime();
    for (int j = 0; j < 1000; ++j){
        Integer nextInt = (int)(32000*Math.random());
        tester2.put(nextInt, nextInt);
    }
    long end = System.nanoTime();
    totalTime += (end-start);
    tester2 = new HybridHash<Integer, Integer>();
}
System.out.println("Average time for 100 (size = 1000)  tables: " + (totalTime / 100) + " ns\n");

totalTime = 0;
for (int i = 0; i < 100; ++i){
    long start = System.nanoTime();
    for (int j = 0; j < 10000; ++j){
        Integer nextInt = (int)(32000*Math.random());
        tester2.put(nextInt, nextInt);
    }
    long end = System.nanoTime();
    totalTime += (end-start);
    tester2 = new HybridHash<Integer, Integer>();
}
System.out.println("Average time for 100 (size = 10000) tables: " + (totalTime / 100) + " ns\n");
```

TEST FOR SORTINGS

```
System.out.printf("VALIDATION TEST FOR MERGE SORT, QUICK SORT AND NEW_SORT\n-----------------------------------------------------\n");
validationTest("merge sort");
validationTest("quick sort");
validationTest("new_sort");

System.out.printf("PERFORMANCE TEST FOR MERGE SORT, QUICK SORT AND NEW_SORT\n-----------------------------------------------------------\n");
long totalSM = 0, totalMM = 0, totalLM = 0;
long totalSQ = 0, totalMQ = 0, totalLQ = 0;
long totalSN = 0, totalMN = 0, totalLN = 0;

for (int i = 0; i < 1000; ++i){
    Integer[] smallArrayMerge = randomArrayGenerator(100);
    Integer[] mediumArrayMerge = randomArrayGenerator(1000);
    Integer[] largeArrayMerge = randomArrayGenerator(10000);

    Integer[] smallArrayQuick = new Integer[100];
    System.arraycopy(smallArrayMerge, 0, smallArrayQuick, 0, 100);
    Integer[] mediumArrayQuick = new Integer[1000];
    System.arraycopy(mediumArrayMerge, 0, mediumArrayQuick, 0, 1000);
    Integer[] largeArrayQuick = new Integer[10000];
    System.arraycopy(largeArrayMerge, 0, largeArrayQuick, 0, 10000);

    Integer[] smallArrayNew = new Integer[100];
    System.arraycopy(smallArrayMerge, 0, smallArrayNew, 0, 100);
    Integer[] mediumArrayNew = new Integer[1000];
    System.arraycopy(mediumArrayMerge, 0, mediumArrayNew, 0, 1000);
    Integer[] largeArrayNew = new Integer[10000];
    System.arraycopy(largeArrayMerge, 0, largeArrayNew, 0, 10000);

    long startSM = System.nanoTime();
    Sortings.mergeSort(smallArrayMerge);
    long endSM = System.nanoTime();
    totalSM+=endSM-startSM;

    long startMM = System.nanoTime();
    Sortings.mergeSort(mediumArrayMerge);
    long endMM = System.nanoTime();
    totalMM+=endMM-startMM;

    long startLM = System.nanoTime();
    Sortings.mergeSort(largeArrayMerge);
    long endLM = System.nanoTime();
    totalLM+=endLM-startLM;
```

```
    long startSQ = System.nanoTime();
    Sortings.quickSort(smallArrayQuick);
    long endSQ = System.nanoTime();
    totalSQ+=endSQ-startSQ;

    long startMQ = System.nanoTime();
    Sortings.quickSort(mediumArrayQuick);
    long endMQ = System.nanoTime();
    totalMQ+=endMQ-startMQ;

    long startLQ = System.nanoTime();
    Sortings.quickSort(largeArrayQuick);
    long endLQ = System.nanoTime();
    totalLQ+=endLQ-startLQ;

    long startSN = System.nanoTime();
    Sortings.new_sort(smallArrayNew);
    long endSN = System.nanoTime();
    totalSN+=endSN-startSN;

    long startMN = System.nanoTime();
    Sortings.new_sort(mediumArrayNew);
    long endMN = System.nanoTime();
    totalMN+=endMN-startMN;

    long startLN = System.nanoTime();
    Sortings.new_sort(largeArrayNew);
    long endLN = System.nanoTime();
    totalLN+=endLN-startLN;
}

System.out.printf("\nAverage time for 1000 small  size (100)   array sorting with merge sort: %d ns\n", totalSM/1000);
System.out.printf("Average time for 1000 medium size (1000)  array sorting with merge sort: %d ns\n", totalMM/1000);
System.out.printf("Average time for 1000 large  size (10000) array sorting with merge sort: %d ns\n\n", totalLM/1000);

System.out.printf("Average time for 1000 small  size (100)   array sorting with quick sort: %d ns\n", totalSQ/1000);
System.out.printf("Average time for 1000 medium size (1000)  array sorting with quick sort: %d ns\n", totalMQ/1000);
System.out.printf("Average time for 1000 large  size (10000) array sorting with quick sort: %d ns\n\n", totalLQ/1000);

System.out.printf("Average time for 1000 small  size (100)   array sorting with new_sort: %d ns\n", totalSN/1000);
System.out.printf("Average time for 1000 medium size (1000)  array sorting with new_sort: %d ns\n", totalMN/1000);
System.out.printf("Average time for 1000 large  size (10000) array sorting with new_sort: %d ns\n\n", totalLN/1000);
```

validationTest takes name of the sorting algorithm, creates 4 different arrays and test given sorting algorithm with these 4 arrays:

```
private static <T> void validationTest(String s){
    Integer arrayTest1[] = {-123, 1234, 1, 140, 4, 5, 3, 0, -1, 32, 4, 13, -12312, 12};
    Integer arrayTest2[] = {-123, 1234, 1, 140, 4, 5, 3, 0, -1, 32, 4, 13, -12312};
    Integer arrayTest3[] = {};
    Integer arrayTest4[] = {12};

    System.out.printf("Array with even number (14) of elements ---> %s", stringArray(arrayTest1));
    if (s.equals("merge sort")) Sortings.mergeSort(arrayTest1);
    else if (s.equals("quick sort")) Sortings.quickSort(arrayTest1);
    else if (s.equals("new_sort")) Sortings.new_sort(arrayTest1);
    System.out.printf("Array after being sorted by %-10s  ---> %s", s, stringArray(arrayTest1));

    System.out.printf("\nArray with odd number (13) of elements  ---> %s", stringArray(arrayTest2));
    if (s.equals("merge sort")) Sortings.mergeSort(arrayTest2);
    else if (s.equals("quick sort")) Sortings.quickSort(arrayTest2);
    else if (s.equals("new_sort")) Sortings.new_sort(arrayTest2);
    System.out.printf("Array after being sorted by %-10s  ---> %s", s, stringArray(arrayTest2));

    System.out.printf("\nEmpty array                              ---> %s", stringArray(arrayTest3));
    if (s.equals("merge sort")) Sortings.mergeSort(arrayTest3);
    else if (s.equals("quick sort")) Sortings.quickSort(arrayTest3);
    else if (s.equals("new_sort")) Sortings.new_sort(arrayTest3);
    System.out.printf("Array after being sorted by %-10s  ---> %s", s, stringArray(arrayTest3));

    System.out.printf("\nArray with 1 element                     ---> %s", stringArray(arrayTest4));
    if (s.equals("merge sort")) Sortings.mergeSort(arrayTest4);
    else if (s.equals("quick sort")) Sortings.quickSort(arrayTest4);
    else if (s.equals("new_sort")) Sortings.new_sort(arrayTest4);
    System.out.printf("Array after being sorted by %-10s  ---> %s", s, stringArray(arrayTest4));

    System.out.printf("\n\n==========================================================================================\n\n");
}
```

randomArrayGenerator creates randomly filled array with given size and returns it:

```
private static Integer[] randomArrayGenerator(int size){
    Integer[] array;
    if (size == 100) array = new Integer[100];
    else if (size == 1000) array = new Integer[1000];
    else array = new Integer[10000];

    for (int i = 0; i < size; ++i){
        Random generator = new Random();
        array[i] = generator.nextInt(12345);
    }

    return array;
}
```

5. RUNNING AND RESULTS

RESULTS FOR HASHING WITH BINARY SEARCH TREE

```
TEST OF THE HASHING WITH BST
----------------------------
VALIDATION TEST WITH FILE "pairs.txt"

After adding elements in the file to the hash table with size 7:

0-computer
 null
 7-book
  null
  null


=======================================

15-cup
 1-yatch
  null
  null
 null


=======================================

23-dart
 16-insomnia
  9-cosmos
   null
   null
  null
 30-vertigo
  null
  44-redesign
   null
   null


=======================================

17-dance
 10-chess
  null
  null
 38-cactus
  null
  null
=======================================

null

=======================================

19-ship
 12-bird
  null
  null
 null


=======================================

6-level
 null
 20-again
  null
  null


=======================================
```

```
Let's get value of the key 16: insomnia
Let's get value of the key 13: null

Let's remove non-existing key (13).
Returned element of remove(13): null

Let's remove existing key (44).
Returned element of remove(44): redesign

Let's remove some other keys (12, 17, 6, 15) and change value of 23.

Table after these removals:

0-computer
 null
 7-book
  null
  null


=======================================

1-yatch
 null
 null


=======================================

23-changedValue
 16-insomnia
  9-cosmos
   null
   null
  null
 30-vertigo
  null
  null


=======================================

10-chess
 null
 38-cactus
  null
  null


=======================================

null

=======================================

19-ship
 null
 null


=======================================

20-again
 null
 null


=======================================
```

```
Let's try to exceed threshold with adding more dummy elements. Table after rehash:

0-computer
 null
 105-toExceedThreshold
  30-vertigo
   null
   null
  null


========================================

1-yatch
 null
 106-toExceedThreshold
  16-insomnia
   null
   null
  null


========================================

107-toExceedThreshold
 null
 null


========================================

108-toExceedThreshold
 null
 null


========================================

109-toExceedThreshold
 19-ship
  null
  null
 null


========================================
```

```
110-toExceedThreshold
 20-again
  null
  null
 null


========================================

null

========================================

7-book
 null
 null


========================================

23-changedValue
 null
 38-cactus
  null
  null


========================================

9-cosmos
 null
 null


========================================

100-toExceedThreshold
 10-chess
  null
  null
 null


========================================

101-toExceedThreshold
 null
 null


========================================

102-toExceedThreshold
 null
 null


========================================

103-toExceedThreshold
 null
 null


========================================

104-toExceedThreshold
 null
 null


========================================



PERFORMANCE TEST

Average time for 100 (size = 100)   tables: 329252 ns

Average time for 100 (size = 1000)  tables: 1112304 ns

Average time for 100 (size = 10000) tables: 3804298 ns
```

RESULTS FOR HASHING WITH COALESCED AND DOUBLE HASHING

```
TEST OF THE HYBRID HASHING
--------------------------
VALIDATION TEST WITH ELEMENTS IN THE PDF

After adding elements to the hash table with size 10 (index --> {key - value} | NEXT : {next entry}):

0 --> null | NEXT : null
1 --> null | NEXT : null
2 --> null | NEXT : null
3 --> 23 - valueOf23 | NEXT : null
4 --> 12 - valueOf12 | NEXT : 13 - valueOf13
5 --> 13 - valueOf13 | NEXT : null
6 --> 51 - valueOf51 | NEXT : null
7 --> 3 - valueOf3 | NEXT : null
8 --> 25 - valueOf25 | NEXT : 23 - valueOf23
9 --> null | NEXT : null

Let's get value of the key 12: valueOf12
Let's get value of the key 40: null

Let's remove non-existing key (40).
Returned element of remove(40): null

Let's remove existing key (25).
Returned element of remove(25): valueOf25

Table after these removals:

0 --> null | NEXT : null
1 --> null | NEXT : null
2 --> null | NEXT : null
3 --> null | NEXT : null
4 --> 12 - valueOf12 | NEXT : 13 - valueOf13
5 --> 13 - valueOf13 | NEXT : null
6 --> 51 - valueOf51 | NEXT : null
7 --> 3 - valueOf3 | NEXT : null
8 --> 23 - valueOf23 | NEXT : null
9 --> null | NEXT : null

Let's remove another existing key (23).
Returned element of remove(23): valueOf23

Table after this removal:

0 --> null | NEXT : null
1 --> null | NEXT : null
2 --> null | NEXT : null
3 --> null | NEXT : null
4 --> 12 - valueOf12 | NEXT : 13 - valueOf13
5 --> 13 - valueOf13 | NEXT : null
6 --> 51 - valueOf51 | NEXT : null
7 --> 3 - valueOf3 | NEXT : null
8 --> null | NEXT : null
9 --> null | NEXT : null

Let's try to exceed threshold with adding more dummy elements. Table after rehash:

0 --> 31 - toExceedThreshold | NEXT : null
1 --> null | NEXT : null
2 --> 3 - valueOf3 | NEXT : null
3 --> null | NEXT : null
4 --> null | NEXT : null
5 --> 23 - toExceedThreshold | NEXT : 13 - valueOf13
6 --> 25 - toExceedThreshold | NEXT : null
7 --> null | NEXT : null
8 --> null | NEXT : null
9 --> null | NEXT : null
10 --> 13 - valueOf13 | NEXT : 51 - valueOf51
11 --> 51 - valueOf51 | NEXT : 25 - toExceedThreshold
12 --> null | NEXT : null
13 --> 12 - valueOf12 | NEXT : 3 - valueOf3
14 --> null | NEXT : null
15 --> null | NEXT : null
16 --> null | NEXT : null
17 --> null | NEXT : null
18 --> 32 - toExceedThreshold | NEXT : 31 - toExceedThreshold
19 --> null | NEXT : null
20 --> null | NEXT : null
```

```
Let's remove key 51.

Table after this removal:

0 --> 31 - toExceedThreshold | NEXT : null
1 --> null | NEXT : null
2 --> 3 - valueOf3 | NEXT : null
3 --> null | NEXT : null
4 --> null | NEXT : null
5 --> 23 - toExceedThreshold | NEXT : 13 - valueOf13
6 --> null | NEXT : null
7 --> null | NEXT : null
8 --> null | NEXT : null
9 --> null | NEXT : null
10 --> 13 - valueOf13 | NEXT : 25 - toExceedThreshold
11 --> 25 - toExceedThreshold | NEXT : null
12 --> null | NEXT : null
13 --> 12 - valueOf12 | NEXT : 3 - valueOf3
14 --> null | NEXT : null
15 --> null | NEXT : null
16 --> null | NEXT : null
17 --> null | NEXT : null
18 --> 32 - toExceedThreshold | NEXT : 31 - toExceedThreshold
19 --> null | NEXT : null
20 --> null | NEXT : null
```

```
Let's remove key 13.

Table after this removal:

0 --> 31 - toExceedThreshold | NEXT : null
1 --> null | NEXT : null
2 --> 3 - valueOf3 | NEXT : null
3 --> null | NEXT : null
4 --> null | NEXT : null
5 --> 23 - toExceedThreshold | NEXT : 25 - toExceedThreshold
6 --> null | NEXT : null
7 --> null | NEXT : null
8 --> null | NEXT : null
9 --> null | NEXT : null
10 --> 25 - toExceedThreshold | NEXT : null
11 --> null | NEXT : null
12 --> null | NEXT : null
13 --> 12 - valueOf12 | NEXT : 3 - valueOf3
14 --> null | NEXT : null
15 --> null | NEXT : null
16 --> null | NEXT : null
17 --> null | NEXT : null
18 --> 32 - toExceedThreshold | NEXT : 31 - toExceedThreshold
19 --> null | NEXT : null
20 --> null | NEXT : null
```

```
Let's remove key 32.

Table after this removal:

0 --> null | NEXT : null
1 --> null | NEXT : null
2 --> 3 - valueOf3 | NEXT : null
3 --> null | NEXT : null
4 --> null | NEXT : null
5 --> 23 - toExceedThreshold | NEXT : 25 - toExceedThreshold
6 --> null | NEXT : null
7 --> null | NEXT : null
8 --> null | NEXT : null
9 --> null | NEXT : null
10 --> 25 - toExceedThreshold | NEXT : null
11 --> null | NEXT : null
12 --> null | NEXT : null
13 --> 12 - valueOf12 | NEXT : 3 - valueOf3
14 --> null | NEXT : null
15 --> null | NEXT : null
16 --> null | NEXT : null
17 --> null | NEXT : null
18 --> 31 - toExceedThreshold | NEXT : null
19 --> null | NEXT : null
20 --> null | NEXT : null
```

```
Let's remove key 25 and change value of key 12.

Table:

0 --> null | NEXT : null
1 --> null | NEXT : null
2 --> 3 - valueOf3 | NEXT : null
3 --> null | NEXT : null
4 --> null | NEXT : null
5 --> 23 - toExceedThreshold | NEXT : null
6 --> null | NEXT : null
7 --> null | NEXT : null
8 --> null | NEXT : null
9 --> null | NEXT : null
10 --> null | NEXT : null
11 --> null | NEXT : null
12 --> null | NEXT : null
13 --> 12 - changedValue | NEXT : 3 - valueOf3
14 --> null | NEXT : null
15 --> null | NEXT : null
16 --> null | NEXT : null
17 --> null | NEXT : null
18 --> 31 - toExceedThreshold | NEXT : null
19 --> null | NEXT : null
20 --> null | NEXT : null


PERFORMANCE TEST

Average time for 100 (size = 100)   tables: 55733770 ns

Average time for 100 (size = 1000)  tables: 38059355 ns

Average time for 100 (size = 10000) tables: 1133124554 ns
```

RESULTS FOR SORTINGS

```
VALIDATION TEST FOR MERGE SORT, QUICK SORT AND NEW_SORT
-------------------------------------------------------
Array with even number (14) of elements ---> -123 | 1234 | 1 | 140 | 4 | 5 | 3 | 0 | -1 | 32 | 4 | 13 | -12312 | 12
Array after being sorted by merge sort  ---> -12312 | -123 | -1 | 0 | 1 | 3 | 4 | 4 | 5 | 12 | 13 | 32 | 140 | 1234

Array with odd number (13) of elements  ---> -123 | 1234 | 1 | 140 | 4 | 5 | 3 | 0 | -1 | 32 | 4 | 13 | -12312
Array after being sorted by merge sort  ---> -12312 | -123 | -1 | 0 | 1 | 3 | 4 | 4 | 5 | 13 | 32 | 140 | 1234

Empty array                             --->
Array after being sorted by merge sort  --->

Array with 1 element                    ---> 12
Array after being sorted by merge sort  ---> 12


================================================================================

Array with even number (14) of elements ---> -123 | 1234 | 1 | 140 | 4 | 5 | 3 | 0 | -1 | 32 | 4 | 13 | -12312 | 12
Array after being sorted by quick sort  ---> -12312 | -123 | -1 | 0 | 1 | 3 | 4 | 4 | 5 | 12 | 13 | 32 | 140 | 1234

Array with odd number (13) of elements  ---> -123 | 1234 | 1 | 140 | 4 | 5 | 3 | 0 | -1 | 32 | 4 | 13 | -12312
Array after being sorted by quick sort  ---> -12312 | -123 | -1 | 0 | 1 | 3 | 4 | 4 | 5 | 13 | 32 | 140 | 1234

Empty array                             --->
Array after being sorted by quick sort  --->

Array with 1 element                    ---> 12
Array after being sorted by quick sort  ---> 12


================================================================================

Array with even number (14) of elements ---> -123 | 1234 | 1 | 140 | 4 | 5 | 3 | 0 | -1 | 32 | 4 | 13 | -12312 | 12
Array after being sorted by new_sort    ---> -12312 | -123 | -1 | 0 | 1 | 3 | 4 | 4 | 5 | 12 | 13 | 32 | 140 | 1234

Array with odd number (13) of elements  ---> -123 | 1234 | 1 | 140 | 4 | 5 | 3 | 0 | -1 | 32 | 4 | 13 | -12312
Array after being sorted by new_sort    ---> -12312 | -123 | -1 | 0 | 1 | 3 | 4 | 4 | 5 | 13 | 32 | 140 | 1234

Empty array                             --->
Array after being sorted by new_sort    --->

Array with 1 element                    ---> 12
Array after being sorted by new_sort    ---> 12


================================================================================

PERFORMANCE TEST FOR MERGE SORT, QUICK SORT AND NEW_SORT
--------------------------------------------------------

Average time for 1000 small  size (100)   array sorting with merge sort: 33569 ns
Average time for 1000 medium size (1000)  array sorting with merge sort: 205112 ns
Average time for 1000 large  size (10000) array sorting with merge sort: 2475147 ns

Average time for 1000 small  size (100)   array sorting with quick sort: 14348 ns
Average time for 1000 medium size (1000)  array sorting with quick sort: 141950 ns
Average time for 1000 large  size (10000) array sorting with quick sort: 1786712 ns

Average time for 1000 small  size (100)   array sorting with new_sort: 60090 ns
Average time for 1000 medium size (1000)  array sorting with new_sort: 4743809 ns
Average time for 1000 large  size (10000) array sorting with new_sort: 493621262 ns
```