Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 5 REPORT

Azmi Utku Sezgin
131044048

Course Assistant: fesirci@gtu.edu.tr

1 Double Hashing Map

This part about Question1 in HW5

1.1 Pseudocode and Explanation

DoubleHashingMap implements **MapInterface** which has the following methods;

```
V get(K key);
boolean isEmpty();
V put(K key, V value);
V remove(K key);
int size();
```

Load treshold for this HashMap is **0.75** and collisions are handled using double hashing. Second hashing calculated **Math.abs(3-index%3)** using this function which gives us the number of buckets. Index is the first collision point here. There's a method called getIndex which returns the value of table at given index. Although there shouldn't be such method in this particular data structure, it was only used for testing purposes as i will be explaining how i used it on **Test Cases** section.

1.2 Test Cases

```
First double hashing map tests
Get by key: CSE222
Get by key: CSE312
Get by key: CSE344
Get by index: CSE222
Get by index: CSE312
Get by index: CSE344

Second double hashing map tests
Get by key: 5.5
Get by key: 9.5
Get by key: 6.5
Get by key: 2.5
Get by key: 15.5
Get by key: 88.5
Get by index: 5.5
Get by index: 9.5
Get by index: 6.5
Get by index: 2.5
Get by index: 5.5
Get by index: 88.5
```

```
DoubleHashingMap<Integer, String> doubleHashingMap1 = new DoubleHashingMap<>( size: 10);

doubleHashingMap1.put(5, "CSE222");
doubleHashingMap1.put(15, "CSE312");
doubleHashingMap1.put(13, "CSE344");

System.out.println("First double hashing map tests");
System.out.println("Get by key: " + doubleHashingMap1.get(5));
System.out.println("Get by key: " + doubleHashingMap1.get(15));
System.out.println("Get by key: " + doubleHashingMap1.get(13));
System.out.println("Get by index: " + doubleHashingMap1.getIndex(5));
System.out.println("Get by index: " + doubleHashingMap1.getIndex(6));
System.out.println("Get by index: " + doubleHashingMap1.getIndex(3));

System.out.println("\nSecond double hashing map tests");
DoubleHashingMap<Integer, Double> doubleHashingMap2 = new DoubleHashingMap<>( size: 5);

doubleHashingMap2.put(5, 5.5);
doubleHashingMap2.put(9, 9.5);
doubleHashingMap2.put(6, 6.5);
doubleHashingMap2.put(2, 2.5);
doubleHashingMap2.put(15, 15.5);
doubleHashingMap2.put(88, 88.5);

System.out.println("Get by key: " + doubleHashingMap2.get(5));
System.out.println("Get by key: " + doubleHashingMap2.get(9));
System.out.println("Get by key: " + doubleHashingMap2.get(6));
System.out.println("Get by key: " + doubleHashingMap2.get(2));
System.out.println("Get by key: " + doubleHashingMap2.get(15));
System.out.println("Get by key: " + doubleHashingMap2.get(88));
//Since there's a rehashing, indexes will be changed.
//Node: Normally there shouldn't be a getIndex method, these are just for testing purposes.
System.out.println("Get by index: " + doubleHashingMap2.getIndex(5));
System.out.println("Get by index: " + doubleHashingMap2.getIndex(9));
System.out.println("Get by index: " + doubleHashingMap2.getIndex(6));
System.out.println("Get by index: " + doubleHashingMap2.getIndex(2));
System.out.println("Get by index: " + doubleHashingMap2.getIndex(5));
System.out.println("Get by index: " + doubleHashingMap2.getIndex(8));
```

There are 2 different DoubleHashingMaps, first one is to check if the hashing is done correct. After declaring first dll with size of 10, there are 3 put calls. First put call is 5, “CSE222” which should be placed at index of 5. Then, there’s 15 there’s a collision here and it’s handled by double hashing **Math.abs(3-index%3)** according to this. 3-5%3 = 1 so the number of buckets is 1 which makes our new index 6. For the 3rd one 13 is gonna be placed at index of 3. You can normally use get method to find your value by giving your key also to test it getIndex() will also give you the correct value.

Second test is for rehashing, to see what happens if table exceeds load treshold.Size is 5, treshold is 0.75 so hehashing happens after 4th insertion as 4/5 is 0.8. After 4th insertion whole table changes as rehashing needs to be done with new table size. For example 88 on a table sized 5 should be on 3 whereas after rehashing table size doubles therefore 88’s new index on table becomes 8 instead of 5.

2 Recursive Hashing Set

This part about Question2 in HW5

2.1 Pseudocode and Explanation

Write pseudocode and explanation about code design. Indicate what you are using that interfaces, classes, structures, etc.

2.2 Test Cases

Try this code least 2 different hash table size and 2 different sequence of keys. Report all of situations.

3 Sorting Algortihms

3.1 MergeSort with DoubleLinkedList

Double linked list **KWLinkedList** is from class textbook. MergeSort was implemented for Generic arrays and is changed for double linked list to work. Implemented set method to replace it with = operator, used get method for [] operator.

```
Run Time : 4833999
Run Time : 115613
Run Time : 124065
Run Time : 233942
Run Time : 131611
Run Time : 132819
Run Time : 125273
Run Time : 123461
Run Time : 127385
Run Time : 164514
Running time of merge sorting for Double Linked List with size of 10 is 611268 nanoseconds.

Process finished with exit code 0
```

Avarage time is calculated by running the program 10 times. As you can see the first run time is a lot higher than the others. I could not find the reasoning behind this and i would like to know why does that happen.

3.1.1 Pseudocode and Explanation

Write pseudocode and explanation about code design. Indicate what you are using that interfaces, classes, structures, etc.

3.1.2 Average Run Time Analysis

5 sorting algorithms ran 10 times for 10 different size of arrays.

```
5, 10, 25, 50, 100, 250, 500, 1000, 2500, 5000
```

Average time complexities for sortings:
Heap Sort: θ(n log(n))
Insertion Sort: θ(n^2)
Merge Sort θ(n log(n))
Quick Sort θ(n log(n))

Heap Sort, Merge Sort and Quick Sort are roughly the same as are their time complexities and Insertion is a lot slower than them. As for Merge Sort for DoubleLinkedList, MergeSorting is θ(n log(n)) but in the code, it uses get(int index) method which runs in θ(n) time. Which causes this sorting to be extremely slow at least for my implementation. As you can see on test result, it is 100 times slower than merge sort for arrays.

```
Average runtimes for 5 sorting algorithms calculated for 10 different sizes in 10 different runs.

Heap Sort Average runtime: 222069
Insertion Sort Average runtime: 5548853
Merge Sort Average runtime: 196037
Merge Sort for DLL Average runtime: 27383968
quick Sort Average runtime: 151247

Process finished with exit code 0
```

3.1.3 Wort-case Performance Analysis

This part about Question5 in HW5

3.2 MergeSort

This part about code in course book.

3.2.1 Average Run Time Analysis

This part about Question4 in HW5

3.2.2 Wort-case Performance Analysis

This part about Question5 in HW5

3.3 Insertion Sort

3.3.1 Average Run Time Analysis

This part about Question4 in HW5

3.3.2 Wort-case Performance Analysis

This part about Question5 in HW5

3.4 Quick Sort

3.4.1 Average Run Time Analysis

This part about Question4 in HW5

3.4.2 Wort-case Performance Analysis

This part about Question5 in HW5

3.5 Heap Sort

3.5.1 Average Run Time Analysis

This part about Question4 in HW5

3.5.2 Wort-case Performance Analysis

This part about Question5 in HW5

4 Comparison the Analysis Results

This part about Question5 in HW5. Using before analysis results in show that section 3. Show that one graphic (like Figure 4.1) include 5 sorting algorithm worst-case analysis cases.

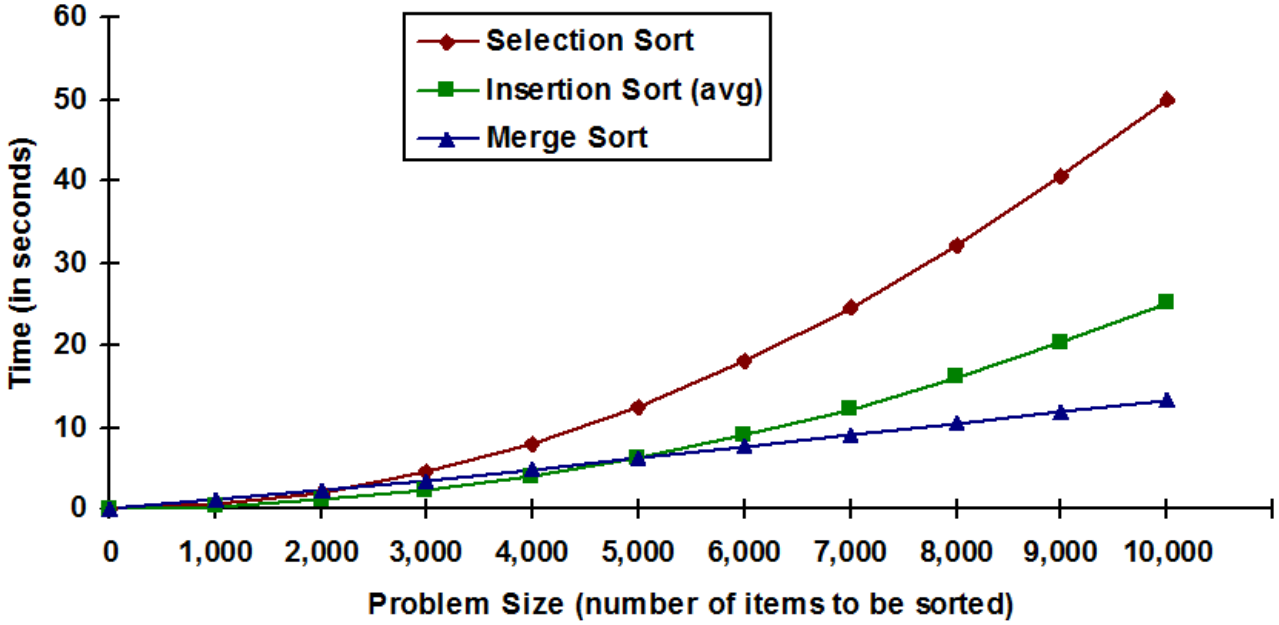

Figure 4.1. Comparison of sorting algorithms (this figure just a example)