**Gebze Technical University**

**Computer Engineering**

**CSE 222 - 2018 Spring**

**HOMEWORK 5 REPORT**

**ZEYNEB AKDOĞAN**

**181041038**

Course Assistant:

# Double Hashing Map

This part about Question1 in HW5

Double hashing in aşağıdaki formülleri kullanılarak DoubleHasingMap classı yazılmıştır.

Key’in tablo üzeride yerleştirileceği index hashi(x) i. İterasyonda bulunan değerdir.

*hashi(x) = (hash(x) + f(i)) % ts  
 f(i) = i \* hash2(x)  
 hash2(x) = R - ( x % R)*

## Pseudocode and Explanation

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

<<interface>>

Map<K,V>

+size() : int

+isEmpty() : boolean

+containsKey(key: Object) : boolean

+containsValue(value: Object) : boolean

+get(key: Object) : V

+put(key : K , value : V ): V

+remove(key : Object) : V

+putAll(Map<? **extends** K, ? **extends** V> m)

+**void** clear()

+keySet() : Set<K>

+values() : Collection<V>

+entrySet() : Set<Entry<K, V>>

<<innerClass>>

HashNode<K,V>

**- entry :** Entry<K, V>  
**- state :** NodeState

+HashNode(Entry<K, V> entry)

+getEntry() : Entry<K, V>

+setEntry(Entry<K, V> entry)

+getState() : NodeState

+setState(NodeState state)

<<Class>>

DoubleHasingMap<K,V>

- table : HashNode<K, V>[]  
- count = 0 : int  
- R = 7 : int  
- INITIALSIZE = 23 : int  
- THRESHOLD = 0.75 : double

- getHashindex(code : int, i : int)

- getNewTableSize(currentsize : int)

-isPrime(number : int) : **boolean**

- getNewTableSize(currentsize **: int**) : **int**

**-** hash2(code : int) : **int**

+ printHashTable()

- ifNecessaryEnlargeHashTable();

<<innerClass>>

HashEntry<K,vV>

-**key : K**;  
**-value : V**

+HashEntry(K key, V value)

+getKey() : K

+getValue() : V

+setValue(V value)

## Test Cases

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

HashMap initial size : 23

HashMap Sequence 1

map.put(35, **"Izmir"**);  
map.put(34, **"İstanbul"**);  
map.put(06, **"Ankara"**);  
map.put(17, **"Çanakkale"**);  
map.put(10, **"Balıkesir"**);  
map.put(22, **"Edirne"**);  
map.put(20, **"Denizli"**);  
map.put(01, **"Adana"**);  
map.put(07, **"Antalya"**);  
map.put(41, **"Kocaeli"**);  
map.put(67, **"Zonguldak"**);  
map.put(43, **"Kütahya"**);  
map.put(55, **"Samsun"**);

HashMap initial size : 37

HashMap Sequence 2

map.put(34, **"İstanbul"**);  
map.put(35, **"Izmir"**);  
map.put(17, **"Çanakkale"**);  
map.put(06, **"Ankara"**);  
map.put(10, **"Balıkesir"**);  
map.put(20, **"Denizli"**);  
map.put(22, **"Edirne"**);  
map.put(43, **"Kütahya"**);  
map.put(55, **"Samsun"**);  
map.put(01, **"Adana"**);  
map.put(07, **"Antalya"**);  
map.put(41, **"Kocaeli"**);  
map.put(67, **"Zonguldak"**);

Following methods are tested

Size()

Put()

Get()

Remove()

putAll()

# Recursive Hashing Set

This part about Question2 in HW5

## Pseudocode and Explanation

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

## Test Cases

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

# Sorting Algortihms

SortTestAbstract

**-unorderednumbers :** Integer[]  
**- unorderednumberslist** : LinkedList<Integer>  
**- printnumbers** : **boolean**  
**- printstatus** : **boolean**

**+** *testandgetElapsedtime*(): long

+ getUnorderednumbers() : Integer[]

+ setUnorderednumbers(Integer[] unorderednumbers)

+ getUnorderednumberslist() : LinkedList<Integer>

+ setUnorderednumberslist(LinkedList<Integer> unorderednumberslist)

**+** isPrintnumbers() : **boolean**

+ setPrintnumbers(**boolean** printnumbers)

**+** isPrintstatus() : **boolean**

+ printArray(Integer[] numbers)

+ setPrintstatus(**boolean** printstatus)

+ printLinkedList(LinkedList<Integer> numbers)

TestRandom

+runTest()

TestWorstCase

+runTest()

SortTestManager

+runTest()

SortTestQuick

+SortTestInsertion(

Integer[] unorderednumbers,

**boolean** printnumbers,

**boolean** printstatus)

**+**testandgetElapsedtime() : long

SortTestInsertion

+SortTestInsertion(

Integer[] unorderednumbers,

**boolean** printnumbers,

**boolean** printstatus)

**+**testandgetElapsedtime() : long

SortTestMergeDoubleLinkedList

+SortTestInsertion(

Integer[] unorderednumbers,

**boolean** printnumbers,

**boolean** printstatus)

**+**testandgetElapsedtime() : long

SortTestMerge

+SortTestInsertion(

Integer[] unorderednumbers,

**boolean** printnumbers,

**boolean** printstatus)

**+**testandgetElapsedtime() : long

SortTestHeap

+SortTestInsertion(

Integer[] unorderednumbers,

**boolean** printnumbers,

**boolean** printstatus)

**+**testandgetElapsedtime() : long

Main class creates TestRandom and TestWordtCase objects.

They run the runTest() method.

Each method creates SortTestMAnager Object and calls runTest() method

İn runTest() method

* AverageTest Results Array are created.
* **arraysizes** ={100, 200, 500, 1000, 2000, 5000, 10000, 20000}

for each arraysizes

for i=1 to 10

create test array ( random or wrost case)

call SortTestInsertion and store test result

call SortTestMerge and store test result

call SortTestMergeDoubleLinkedList and store test result

call SortTestHeap and store test result

call SortTestQuick and store test result

end for

store average results of SortTestInsertion for current size

store average results of SortTestMerge for current size

store average results of SortTestMergeDoubleLinkedList for current size

store average results of SortTestHeap for current size

store average results of SortTestQuick for current size

end for each

runTest In testRandom

Print average results of each method for each size

## MergeSort with DoubleLinkedList

This part about Question3 in HW5

### Pseudocode and Explanation

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

MergeSortDoubleLinkedList

+ sort(LinkedList<T> table)

- merge(LinkedList<T> outputSequence,

LinkedList<T> leftSequence,

LinkedList<T> rightSequence)

### Average Run Time Analysis

|  |  |
| --- | --- |
| Size of Arrays | MergeDouble sort |
| 100 | 728968 |
| 200 | 299299 |
| 500 | 2168298 |
| 1000 | 2113503 |
| 2000 | 23203662 |
| 5000 | 31644086 |
| 10000 | 110989524 |
| 20000 | 384253413 |
| 50000 | 2430936175 |
| 100000 | 10026027052 |

### Wort-case Performance Analysis

This part about Question5 in HW5

|  |  |
| --- | --- |
| Size of Arrays | MergeDouble sort |
| 100 | 18212 |
| 1000 | 726045 |
| 5000 | 19504971 |
| 10000 | 82847070 |

## MergeSort

This part about code in course book.

MergeSort

+ sort(T[] table)

- merge(T[] outputSequence,

T[] leftSequence,

T[] rightSequence )

### Average Run Time Analysis

This part about Question4 in HW5

|  |  |
| --- | --- |
| Size of Arrays | Merge sort |
| 100 | 146607 |
| 200 | 68345 |
| 500 | 383131 |
| 1000 | 497975 |
| 2000 | 2274017 |
| 5000 | 3859283 |
| 10000 | 2897662 |
| 20000 | 4605239 |
| 50000 | 12766592 |
| 100000 | 28212734 |

### Wort-case Performance Analysis

This part about Question5 in HW5

|  |  |
| --- | --- |
| Size of Arrays | Merge sort |
| 100 | 9916 |
| 1000 | 91772 |
| 5000 | 453886 |
| 10000 | 1058332 |

## Insertion Sort

InsertionSort

+ sort(T[] table)

- insert (T[] table, **int** nextPos )

### Average Run Time Analysis

This part about Question4 in HW5

|  |  |
| --- | --- |
| Size of Arrays | Insertion sort |
| 100 | 174341 |
| 200 | 448237 |
| 500 | 275161 |
| 1000 | 907378 |
| 2000 | 4313012 |
| 5000 | 23360264 |
| 10000 | 99990874 |
| 20000 | 291710054 |
| 50000 | 1724479001 |
| 100000 | 7337549792 |

### Wort-case Performance Analysis

This part about Question5 in HW5

|  |  |
| --- | --- |
| Size of Arrays | Insertion sort |
| 100 | 13115 |
| 1000 | 945778 |
| 5000 | 23154713 |
| 10000 | 91352365 |

## Quick Sort

QuickSort

+ sort(T[] table)

- quickSort(T[] table, **int** first, **int** last)

- partition(T[] table, **int** first, **int** last) : int

### Average Run Time Analysis

This part about Question4 in HW5

|  |  |
| --- | --- |
| Size of Arrays | Quick sort |
| 100 | 155022 |
| 200 | 55466 |
| 500 | 152098 |
| 1000 | 4178533 |
| 2000 | 638104 |
| 5000 | 1108307 |
| 10000 | 1659814 |
| 20000 | 2740546 |
| 50000 | 6733834 |
| 100000 | 14289636 |

### Wort-case Performance Analysis

This part about Question5 in HW5

|  |  |
| --- | --- |
| Size of Arrays | Quick sort |
| 100 | 11733 |
| 1000 | 656672 |
| 5000 | 16608849 |
| 10000 | 67981907 |

## Heap Sort

HeapSort

+ sort(T[] table)

- buildHeap(T[] table)

- shrinkHeap(T[] table)

- swap(T[] table, **int** i, **int** j)

### Average Run Time Analysis

This part about Question4 in HW5

|  |  |
| --- | --- |
| Size of Arrays | Heap sort |
| 100 | 179358 |
| 200 | 262084 |
| 500 | 590025 |
| 1000 | 726875 |
| 2000 | 1029571 |
| 5000 | 1891636 |
| 10000 | 5295808 |
| 20000 | 4738968 |
| 50000 | 12520745 |
| 100000 | 28264488 |

### Wort-case Performance Analysis

This part about Question5 in HW5

|  |  |
| --- | --- |
| Size of Arrays | Heap sort |
| 100 | 6913 |
| 1000 | 89639 |
| 5000 | 619655 |
| 10000 | 1216080 |

# Comparison the Analysis Results

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

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