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

Hasan Mutlu
1801042673

1. SYSTEM REQUIREMENTS

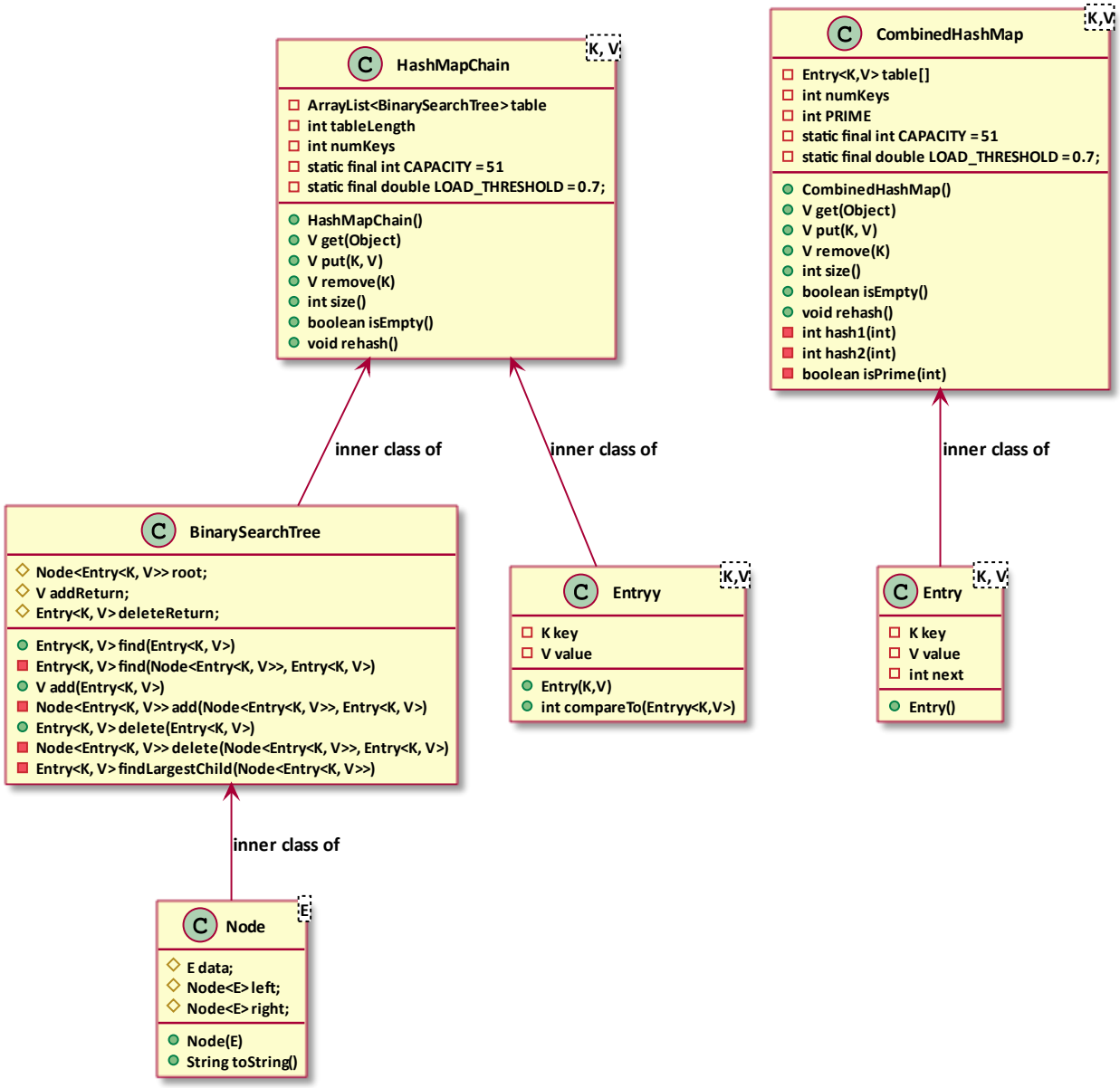
Functional Requirements:

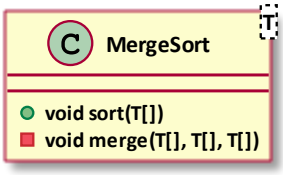
- ➔ User should be able to see the result of hard-coded test cases.

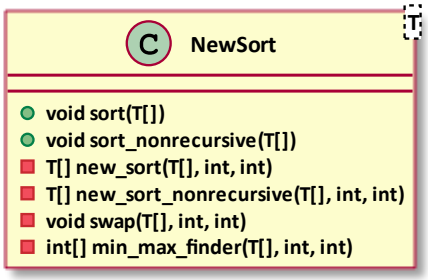
Non-Functional Requirements:

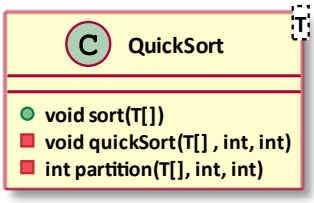
- ➔ Implementation: The programs shall be implemented using VSCode, Ubuntu 18.04 WSL and Java 11.
- ➔ Compiling and Running: The programs should be compiled and run with following commands:
- ➔ -$ javac *.java PartX.java
- ➔ -$ java PartX
- ➔ Efficiency: The algorithms in part 1 must run as efficiently as possible.
- ➔ Relaibility: The algorithms and the programs must run reliably, should handle every possible valid usage.

2. CLASS DIAGRAMS









PROBLEM SOLUTION APPROACH

In part 1.1, I named the chaining hash table with the binary search tree HashMapChain. I tried to use the BST in the book directly, but it got too complicated with the generics. So I decided to implement a new BST as inner class, specifically designed for my HashMapChain. After that, I implemented the HashMapChain's methods, compatible with BST. Creating a generic array of BST was causing some errors, so I used ArrayList to store the array of BSTs.

For part 1.2, I named my hash table CombinedHashMap. I introduced a new variable to Entry, an integer called "next" to indicate the position of the next collided item. Then I implemented the CombinedHashMap. When a collision occurs, the probe position of next element is calculated according to double hashing formula. This process is repeated until an empty place is found. The last checked occupied entry's next value is set to the position of the new item.

For part 1.3, I implemented a driver program to show that the methods of the two hash maps are working. After that, I ran a test as it stated in PDF and printed results.

For part 2, I implemented QuickSort and MergeSort classes from the book. I changed the usage of generics, only. Then, I implemented the NewSort method shown in PDF. After that, I tested the methods. It is terminated when NewSort was sorting the large array, after successfully sorting small and medium arrays. I made research on the internet and found out that it probably runs out of stack space due to large number of recursions. Then, I implemented the same algorithm without recursion, using while loop. But it ran soo poorly that even a single sort was taking seconds. That means it would take minutes to finish all 1000 sortings so I decided to exclude it from the final program.

TEST CASES

Test Case #	Test Case Description	Test Data	Expected Result	Actual Result	Pass/Fail
Part1.1 Test Put Chain	Put small, medium and large number of key/value pairs in HashMapChain	Randomly generated arrays with sizes 100, 1000 and 10000	No Errors	As Expected	Pass
Part1.1 Test Get Chain	Get 3 random values from the HashMapChain	3 random values from small array.	Returned value is correct	As Expected	Pass
Part1.1 Test Delete Chain	Delete 2 variables from the HashMapChain	2 random variables from small array.	They return null after deletion	As Expected	Pass
Part1.1 Test Size Chain	Print size of HashMapChain	The HashMapChain	Size is printed and below treshold rate	As Expected	Pass
Part1.2 Test Combined	Make all previous tests on CombinedHashMap as well	Same data as above	Same outputs printed except size. Size is 98.	As Expected	Pass
Part1.3	Part 1 Empirical Run	100 randomly generated arrays with sizes 100, 1000 and 10000	The time took for accessing existing/non-existing items or adding/removing items for all three sizes are printed	As Expected	Pass
Part 2	Part 2 Empirical Run	1000 randomly generated arrays with sizes 100, 1000 and 10000	The time took for sorting arrays for all three sizes are printed	As Expected	Pass

RUNNING AND RESULTS

Part1:

```
Part1.1 Test: HashMapChain Test:
100 random String/String key/value pairs are put in table.
Some random values from the map:

Key and Value should be: 4fxjAdFT17 - 3d4KPhBReE
Key: 4fxjAdFT17
Value in table: 3d4KPhBReE

Key and Value should be: qYtnpcm3hv - MiiNoCWtJp
Key: qYtnpcm3hv
Value in table: MiiNoCWtJp

Key and Value should be: kikOKvqz07 - 49MvK6jbRB
Key: kikOKvqz07
Value in table: 49MvK6jbRB

Test Delete:
Delete elements with key 4fxjAdFT17 and qYtnpcm3hv
Values after deletion:
Key: 4fxjAdFT17- Value in table: null
Key: qYtnpcm3hv- Value in table: null

Size of the table: 63
```

```
Part1.2 Test: CombinedHashMap Test:
Same 100 random String/String key/value pairs are put in table.
Some random values from the map:

Key and Value should be: 4fxjAdFT17 - 3d4KPhBReE
Key: 4fxjAdFT17
Value in table: 3d4KPhBReE

Key and Value should be: qYtnpcm3hv - MiiNoCWtJp
Key: qYtnpcm3hv
Value in table: MiiNoCWtJp

Key and Value should be: kikOKvqz07 - 49MvK6jbRB
Key: kikOKvqz07
Value in table: 49MvK6jbRB

Test Delete:
Delete elements with key 4fxjAdFT17 and qYtnpcm3hv
Values after deletion:
Key: 4fxjAdFT17- Value in table: null
Key: qYtnpcm3hv- Value in table: null

Size of the table: 98
```

```
Running Results:
Results of HashMapChain for small array:
Average time of putting all elements: 69 microseconds.
Average access time to existing element: 7452 nanoseconds
Average access time for non-existing element: 1664 nanoseconds
Average delete time: 6327 nanoseconds

Results of HashMapChain for medium array:
Average time of putting all elements: 1053 microseconds.
Average access time to existing element: 2311 nanoseconds
Average access time for non-existing element: 1363 nanoseconds
Average delete time: 1470 nanoseconds

Results of HashMapChain for large array:
Average time of putting all elements: 9603 microseconds.
Average access time to existing element: 2375 nanoseconds
Average access time for non-existing element: 1528 nanoseconds
Average delete time: 2177 nanoseconds
```

```
Results of CombinedHashMap for small array:
Average time of putting all elements: 65 microseconds.
Average access time to existing element: 5619 nanoseconds
Average access time for non-existing element: 2554 nanoseconds
Average delete time: 5450 nanoseconds

Results of CombinedHashMap for medium array:
Average time of putting all elements: 612 microseconds.
Average access time to existing element: 2155 nanoseconds
Average access time for non-existing element: 1798 nanoseconds
Average delete time: 1003 nanoseconds

Results of CombinedHashMap for large array:
Average time of putting all elements: 8628 microseconds.
Average access time to existing element: 1626 nanoseconds
Average access time for non-existing element: 1953 nanoseconds
Average delete time: 895 nanoseconds
```

Part2:

```
------------------------
MergeSort small array average: 23 microseconds
MergeSort medium array average: 134 microseconds
MergeSort large array average: 20676 microseconds
------------------------
Quicksort small array average: 3 microseconds
Quicksort medium array average: 9 microseconds
Quicksort large array average: 805 microseconds
------------------------
NewSort small array average: 11 microseconds
NewSort medium array average: 674 microseconds
Newsort was so bad for large array that even a single run took seconds.
------------------------
```