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

Emre YILMAZ
1901042606

1. **SYSTEM REQUIREMENTS**

First Part:

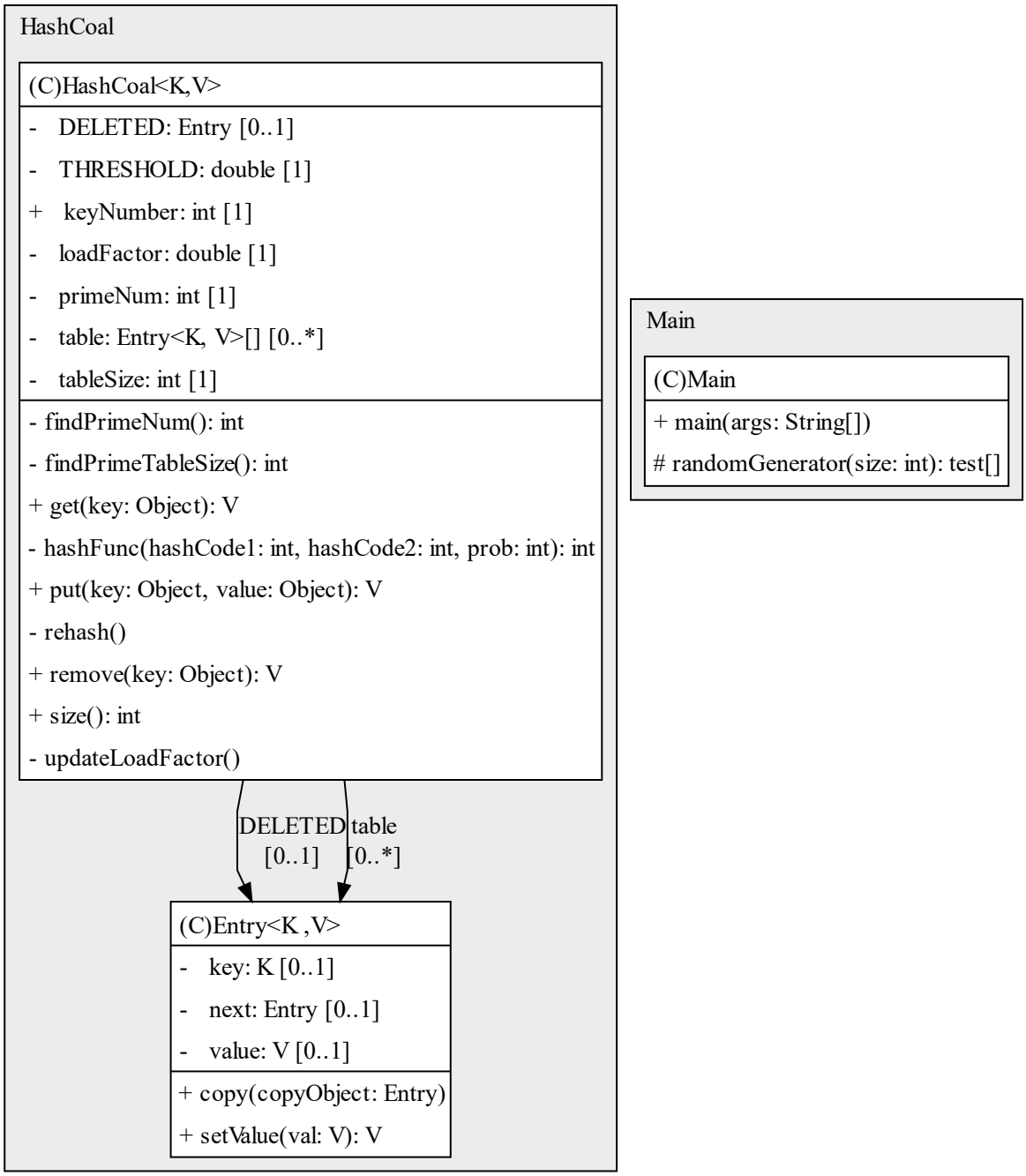
- We have to implement the chaining technique for hashing. But, unlike usual, elements with the same hash code will be kept in Binary Search Tree, not Linked List. We will have a binary search tree array
- In the second question of the first part, we must implement a hashing technique that is a combination of the double hashing and coalesced hashing techniques (which is a hybrid of open addressing and chaining). In our hybrid method, during an insertion operation, the probe positions for the colliding item are calculated by using the double hashing function. Moreover, the colliding items are linked to each other through the pointers as in the coalesced hashing technique.
- We must test these both implementations with 100 randomly generated arrays in different sizes(100, 1000 and 10.000 elements)

Second Part:

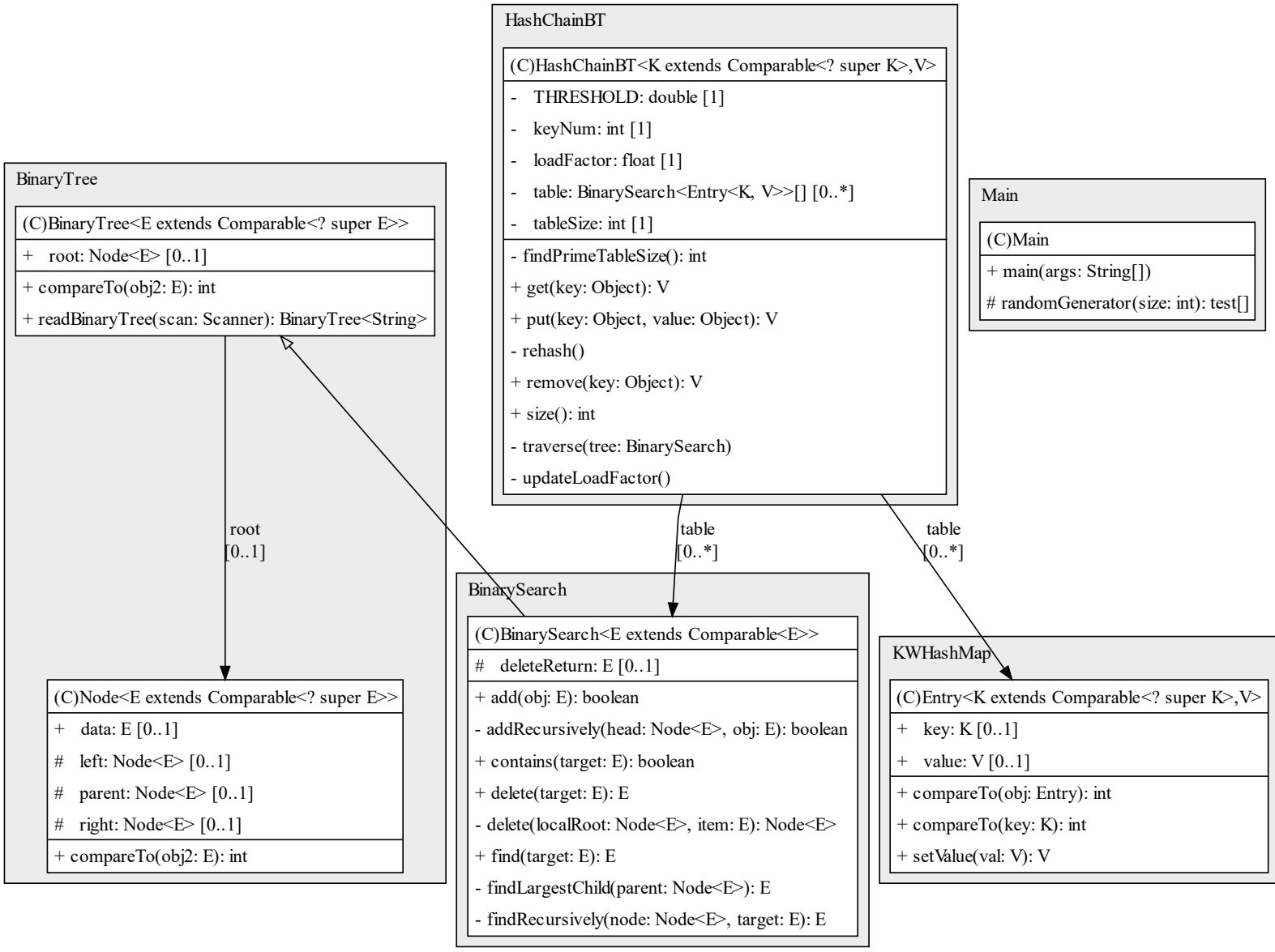
- We must implement the pseudo code is given in the PDF. After that, we must implement Merge Sort and Quick Sort algorithms from TextBook for testing and comparing.
- Firstly we evalulate theoritical analysis results of both 3 algorithms.
- Then, we will have 1000 randomly generated arrays with different sizes. We will sort them with 3 different algorithms (Merge, Quick and New Sort) and we record the run-time of algortithms. Then we will compare them.
- Finally, we will compare the theoritical and empirical results.

2. **CLASS DIAGRAMS**

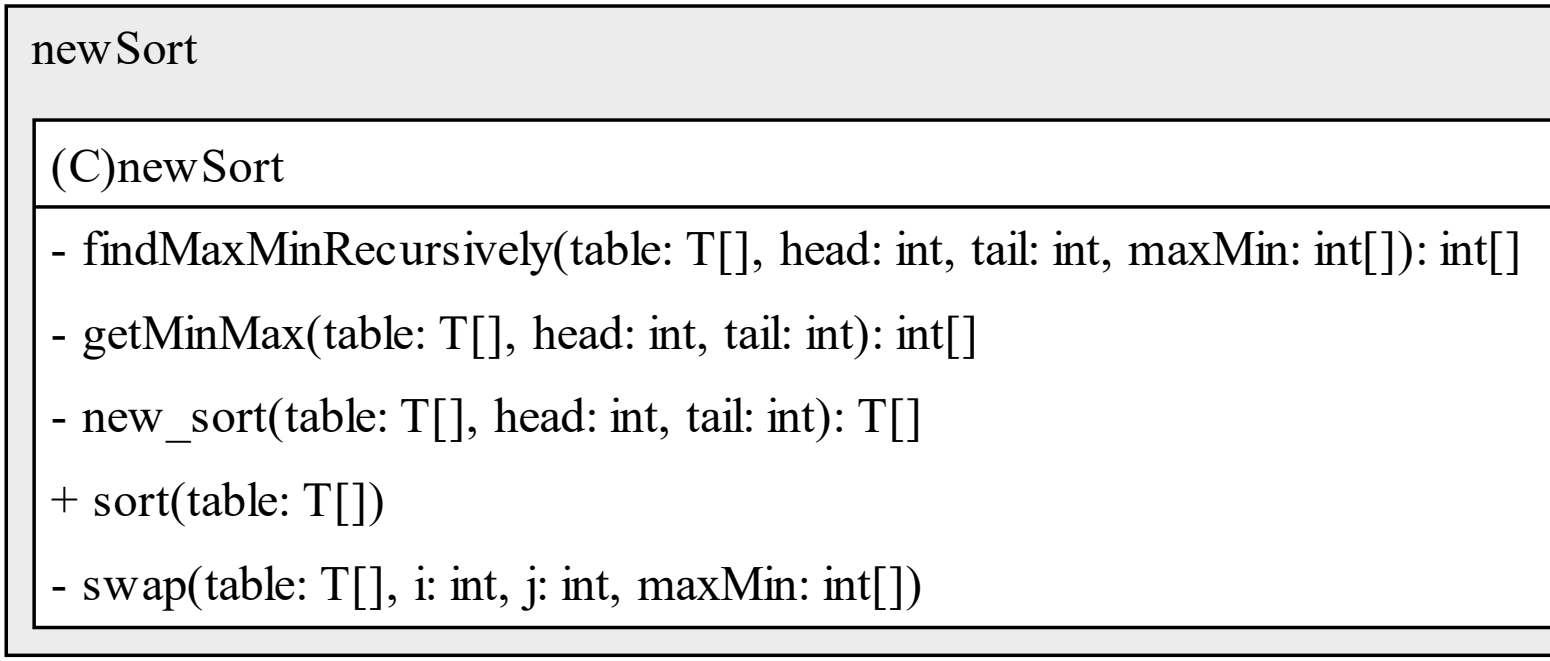
Coalesced Hashing class:



Binary Search Tree Hashing:



Sort Program Diagram:



3. PROBLEM SOLUTION APPROACH

First Part:

1. For first question of this part, we have to implement a new chaining technique for hashing. The new thing is we will use binary search trees to chain items mapped on the same table slot. For this we will have an array. And each element of this array references a Binary Search Tree. I use the Binary Search Implementation from the TextBook. But I had to make some small differences. I have to access the root, so, I add a getRoot() function to Binary Search Tree. I will mention about other main differences such as the function traverse(BinarySearch tree). This function is added to HashChain class (not interface). This function makes inorder traverse in the given Binary Search Tree and puts it to HashChain object. It is used in rehash() function. It adds all the elements to new object. Printing the HashChain is handled by using ToString method of Binary Search Tree. Add/Remove/Get functions are simple. No need to mention but I am going to mention briefly. These functions calculate the index using hash function, then goes the index. Checks whether it is the right place, if it is not, it starts to make move in the tree. If the key is smaller than the target key, it goes towards left in the tree. It is the summary.

2. This part is the main part of the homework. In the HashCoal class, we have an inner class named Entry as usual. In addition, this class has a copy function to copy an entry's data to itself. In this implementation, we have an Entry array. While putting an entry to array, we firstly calculate the index. If we reach the right place, we put the entry; if we are in the wrong place, we will calculate a new index with our hashing functions. Then, (if the new index is ok to put a new entry) this new index is going to be pointed by the previous index we calculate. This is the coalesced hashing. The elements that have the same hash code, point each other. While using get function, we calculate the index and go there. After that, if the key is not our key, we will go to 'next' field of the array[index]. Removal part was hard. There are 2 possibilities. The first of them is the index that is calculated using the key is pointed by another entry. In this case, we will change that *another entry*'s next field to *our index*'s next field. Then we will make *our index* DELETED. The table size is always set to be prime even though the initial size in pdf is 10. Nevertheless, the size 10 is showed as example and the algorithm is proved. The example in the PDF and our test is the same.

Second Part:

1. The only hard thing is the Pseudo-Code is the where to keep the max and min terms. It is solved by using arrays. I declared a 2-element array. The first term of this array keeps the min term, and the second term keeps the max term. While swapping, we must have been careful since when we change the head or tail, we may accidently change the max and min values. So, I add a parameter to private swap function to protect the max and min terms. In this way, I could be guarantee correctness of the max and min terms.

4. TEST CASES

- In the first question of the first part, it is constructed a hash table to print. It is filled with the random numbers. After the printing process, the removing process is done. After that the final view of the hash table is printed.
- Then, the hash table is tested with randomly generated different input size(100, 1000 and 10000) arrays. The run-time is measured and printed end of the program

- In the second question of the first part, it is constructed a coalesced hash table with the data according to example in the PDF. One extra removing process is done while the instance table was being constructing to show that removing function runs successfully.
- IMPORTANT NOTE: The table size is set 10 for the test above since the case in the PDF, the table size was 10. Except for this test, 11 was set as the first size in all other cases. (So I must have to make comment block the part of the code that I showed the example in the pdf.) Then, the coalesced hash table is tested with different input size(100, 1000 and 10000). The run-time is measured and printed end of the program. Also, after the 100 size test, the table is printed.
- NOTE: The hash tables are generated randomly. Numbers were randomly determined from a range of 1 and 50,000. While getting and removing process, all the numbers between 1 and 50.000 passes the get and remove functions.

- In the second part, firstly, 2 random array is constructed. First, it is printed as not-ordered. After that the new_sort algorithm is called and the arrays are ordered and printed.
- Then, the Merge Sort, Quick Sort and New Sort algorithms is tested with randomly generated different input size (100, 1000 and 10000). The run-time is measured and printed end of the program.

5. RUNNING AND RESULTS

Q-1.A.)

ADDING PROCESS ->

```
0. Index Tree:
NULL

1. Index Tree:
95
 null
 null

2. Index Tree:
NULL

3. Index Tree:
NULL

4. Index Tree:
NULL

5. Index Tree:
1979
 1415
  null
  null
 null
```

```
6. Index Tree:
1040
 null
 null

7. Index Tree:
1417
 null
 null

8. Index Tree:
NULL

9. Index Tree:
NULL

10. Index Tree:
NULL

11. Index Tree:
NULL

12. Index Tree:
952
 null
 null
```

```
13. Index Tree:
NULL

14. Index Tree:
NULL

15. Index Tree:
1284
 null
 null

16. Index Tree:
NULL

17. Index Tree:
NULL

18. Index Tree:
NULL

19. Index Tree:
NULL

20. Index Tree:
NULL

21. Index Tree:
NULL
```

```
22. Index Tree:
821
 null
 null

23. Index Tree:
NULL

24. Index Tree:
1857
 null
 null

25. Index Tree:
1905
 null
 null

26. Index Tree:
NULL

27. Index Tree:
1296
 null
 null
```

```
28. Index Tree:
NULL

29. Index Tree:
NULL

30. Index Tree:
NULL

31. Index Tree:
NULL

32. Index Tree:
NULL

33. Index Tree:
268
 null
 456
  null
  null

34. Index Tree:
NULL
```

```
35. Index Tree:
129
 null
 null

36. Index Tree:
NULL

37. Index Tree:
NULL

38. Index Tree:
NULL

39. Index Tree:
1637
 null
 null

40. Index Tree:
NULL

41. Index Tree:
1451
 null
 null

42. Index Tree:
NULL
```

```
43. Index Tree:
NULL

44. Index Tree:
NULL

45. Index Tree:
1455
 null
 null

46. Index Tree:
1691
 null
 null
```

AFTER REMOVING PROCESS

```
0. Index Tree:
NULL

1. Index Tree:
null

2. Index Tree:
NULL

3. Index Tree:
NULL

4. Index Tree:
NULL

5. Index Tree:
1415
 null
 null

6. Index Tree:
null

7. Index Tree:
null

8. Index Tree:
NULL
```

```
9. Index Tree:
NULL

10. Index Tree:
NULL

11. Index Tree:
NULL

12. Index Tree:
null

13. Index Tree:
NULL

14. Index Tree:
NULL

15. Index Tree:
null

16. Index Tree:
NULL

17. Index Tree:
NULL

18. Index Tree:
NULL
```

```
20. Index Tree:
NULL

21. Index Tree:
NULL

22. Index Tree:
null

23. Index Tree:
NULL

24. Index Tree:
null

25. Index Tree:
null

26. Index Tree:
NULL

27. Index Tree:
null

28. Index Tree:
NULL

29. Index Tree:
NULL
```

```
30. Index Tree:
NULL

31. Index Tree:
NULL

32. Index Tree:
NULL

33. Index Tree:
null

34. Index Tree:
NULL

35. Index Tree:
null

36. Index Tree:
NULL

37. Index Tree:
NULL

38. Index Tree:
NULL

39. Index Tree:
null
```

```
40. Index Tree:
NULL

41. Index Tree:
null

42. Index Tree:
NULL

43. Index Tree:
NULL

44. Index Tree:
NULL

45. Index Tree:
null

46. Index Tree:
null
```

```
Testing put function with 100-size array:
100-size array Putting proces: the run-time is: 0.13

Testing get function with 100-size array:
 100-size array Getting process: the run-time is: 0.66

Testing remove function with 100-size array:
100-size array Removing process: the run-time is: 0.05
```

```
Testing put function with 1000-size array:
1000-size array Putting proces: the run-time is: 0.37

Testing get function with 1000-size array:
 1000-size array Getting process: the run-time is: 1.17

Testing remove function with 1000-size array:
1000-size array Removing process: the run-time is: 0.03
```

```
Testing put function with 10000-size array:
10000-size array Putting proces: the run-time is: 1.63

Testing get function with 10000-size array:
 10000-size array Getting process: the run-time is: 1.12

Testing remove function with 10000-size array:
10000-size array Removing process: the run-time is: 0.03
```

Q-1.B.)

Constructing the same hash table in the PDF. We can observe that the functions run correctly.

```
After adding 3 ->
0.) NULL
1.) NULL
2.) NULL
3.) NULL
4.) NULL
5.) NULL
6.) NULL
7.) 3 -> NULL
8.) NULL
9.) NULL

After adding 12 ->
0.) NULL
1.) NULL
2.) NULL
3.) NULL
4.) 12 -> NULL
5.) NULL
6.) NULL
7.) 3 -> NULL
8.) NULL
9.) NULL
```

```
After adding 13 ->
0.) NULL
1.) NULL
2.) NULL
3.) NULL
4.) 12 -> 5
5.) 13 -> NULL
6.) NULL
7.) 3 -> NULL
8.) NULL
9.) NULL

After adding 25 ->
0.) NULL
1.) NULL
2.) NULL
3.) NULL
4.) 12 -> 5
5.) 13 -> NULL
6.) NULL
7.) 3 -> NULL
8.) 25 -> NULL
9.) NULL
```

```
After adding 23 ->
0.) NULL
1.) NULL
2.) NULL
3.) 23 -> NULL
4.) 12 -> 5
5.) 13 -> NULL
6.) NULL
7.) 3 -> NULL
8.) 25 -> 3
9.) NULL

After adding 51 ->
0.) NULL
1.) NULL
2.) NULL
3.) 23 -> NULL
4.) 12 -> 5
5.) 13 -> NULL
6.) 51 -> NULL
7.) 3 -> NULL
8.) 25 -> 3
9.) NULL
```

```
After removing 24 ->
0.) NULL
1.) NULL
2.) NULL
3.) NULL
4.) 12 -> 5
5.) 13 -> NULL
6.) 51 -> NULL
7.) 3 -> NULL
8.) 23 -> NULL
9.) NULL

After removing 13 ->
0.) NULL
1.) NULL
2.) NULL
3.) NULL
4.) 12 -> NULL
5.) NULL
6.) 51 -> NULL
7.) 3 -> NULL
8.) 23 -> NULL
9.) NULL

Process finished with exit code 0
```

Run-Time observations:

```
Testing put function with 100-size array:
100-size array Putting proces: the run-time is: 0.08

Testing get function with 100-size array:
 100-size array Getting process: the run-time is: 0.89

Testing remove function with 100-size array:
100-size array Removing process: the run-time is: 0.06
```

```
Testing get function with 1000-size array:
 1000-size array Getting process: the run-time is: 1.53

Testing remove function with 1000-size array:
1000-size array Removing process: the run-time is: 0.06

Testing put function with 10000-size array:
10000-size array Putting proces: the run-time is: 2.71
```

```
Testing put function with 10000-size array:
10000-size array Putting proces: the run-time is: 2.71

Testing get function with 10000-size array:
 10000-size array Getting process: the run-time is: 0.6

Testing remove function with 10000-size array:
10000-size array Removing process: the run-time is: 0.03
```

Q-2.)

New Sort algorithm proof:

```
Firstly, let's show that the New Sort algorithm runs successfuly:

The array, before sorting:
4 2 16 17 11 2 9

The array, after sorting:
2 2 4 9 11 16 17

Another example:

The array, before sorting:
-19 -10 15 3 -8 -16 3

The array, after sorting:
-19 -16 -10 -8 3 3 15
```

Comparing the sorting algorithms:

```
******************** Testing 100-size arrays sorting ********************

Quick -> 0,031000
Merge -> 0,049000
New -> 0,023000

******************** Testing 1000-size arrays sorting ********************

Quick -> 0,104000
Merge -> 0,235000
New -> 0,931000

******************** Testing 10000-size arrays sorting ********************

Quick -> 1,088000
Merge -> 1,854000
New -> 87,121000
```

Note: The results are written in miliseconds. 1000ms = 1s

Q-1.a)

Coalesced hashing is a collision avoidance technique. 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 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. As in open addressing, deletion from a coalesced hash table is awkward and potentially expensive, and resizing the table is terribly expensive and should be done rarely, if ever.

In addition, coalesced chaining avoids the effects of primary and secondary clustering, and as a result can take advantage of the efficient search algorithm for seperate chaining. If the chains are short, this strategy is very efficient and can be highly condensed, memory-wise.

The longer the average length of the chains, the longer the expected search time. For this reason coalescing is bad for performance. The chains do however still help the lookup algorithm skip over irrelevant slots and mitigate the effects of primary and secondary clustering as I told above. The downside is the size overhead due to next-pointers and the extra level of indirection which affects cache performance.

Also, we can say that in the test of the algorithms in the homework, coalesced hashing was the bad one. Binary Search Tree Hashing is more efficient.

Q-1.b)

Double Hashing uses one hash value generated by the hash function as the starting point and then increments the position by an interval which is decided using a second, independent hash function. It is a popular in open-addressed hash tables.

Advantages:

1. It reduces clustering.

2. It requires fewer comparisons.

3. Smaller hash tables can be used.

Disadvantage:

1. As the table fills up the performance degrades.

Q-2.)

Complexity of the Quick Sort:

- "If the pivot value is a random value selected from the current subarray, then statistically it is expected that half of the items in the subarray will be less than the pivot and half will be greater than the pivot. If both subarrays always have the same number of elements (the best case), there will be log n levels of recursion. At each level, the partitioning process involves moving every element into its correct partition, so quicksort is ***O(n log n)***, just like merge sort."
- This is from book. But, there is something we should pay attention to. If one of subarrays is empty, the other subarray will have one less element than the one just split (only the pivot value will be removed). Therefore, we will have n levels of recursive calls (instead of log n), and the algorithm will be $O(n^2)$. It is solved for the implementation in the my code.

Complexity of the Merge Sort:

- Because each of these lines involves a movement of n elements from smaller-size arrays to larger arrays, the effort to do each merge is *O(n).* The number of lines that require merging (three in this case) is log n because each recursive step splits the array in half. So the total effort to reconstruct the sorted array through merging is ***O(n log n).***
- We need to be able to store both initial sequences and the output sequence. So the array cannot be merged in place, and the additional space usage is ***O(n).***

Complexity of the New Sort:

- Swap() function is $\Theta(1)$
- Recursive getMinMax() function is $O(n)$
- recursive new_Sort method is $\Theta(n)$
- General complexity = $O(n^2)$
- *Note*: the new_Sort method is not exactly O(n), approximately so

Comparing the sorting algorithms ->

```
******************** Testing 100-size arrays sorting ********************

Quick -> 0,031000
Merge -> 0,049000
New -> 0,023000

******************** Testing 1000-size arrays sorting ********************

Quick -> 0,104000
Merge -> 0,235000
New -> 0,931000

******************** Testing 10000-size arrays sorting ********************

Quick -> 1,088000
Merge -> 1,854000
New -> 87,121000
```

Quick merge was the best. In general, Quick Sort and Merge Sort has the same time complexity ***O(n logn)***. But in my test cases, Quick Sort was the best.

As we expected, newSort algorithm was poor. Especially it was observed to be very inefficient, especially when using large data.