

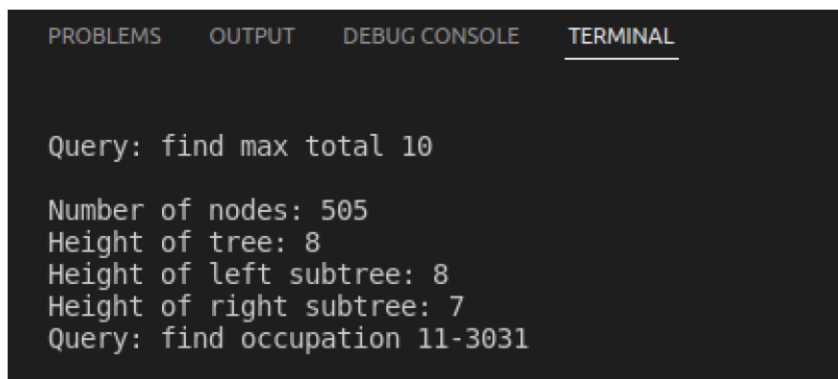
Experimentation

Max-Heap:

Experimentation Run

The experimentation run for this was pretty simple, as the heap is a complete binary tree that means the maximum height it can attain will be $\log(n)$ where “n” is the number of nodes. And since it’s a complete binary tree the height of left tree at max will be similar to the height of the tree and the height of right tree will be at least total height – 1.

1. Print a count of the total number of nodes in the max-heap.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Query: find max total 10

Number of nodes: 505
Height of tree: 8
Height of left subtree: 8
Height of right subtree: 7
Query: find occupation 11-3031
```

2. Print the total height of the max-heap, and the height of its left and right sub trees. Is it “height balanced”?

Ans. Yes, since heap is a complete binary tree it’s balance as right and left sub tree do not differ by more than 1.

3. Is the max-heap an efficient implementation of the find max query? What is another implementation to compute the answer to a find max query? Would it be more or less efficient (does it depend on the value of n)?

Ans. I don’t believe there would be a better implementation for find max query as most of the efficient sorting algorithms are $O(n \log(n))$ in worst case, like merge sort. Just looping through the array and finding max will take $O(n)$ in worst case. Though it does depend on n as Heap sort is better for larger data sets while insertion sort is better for smaller ones.

BST:

Experimentation Run

The way I ran this experiment was I made recursive function that successively traverses the tree and on each level increases the height of tree. The way this function works is that it recursively calls the left and right sub-trees to get their height and at the end return the height of the greater sub tree. The total number of nodes was simple as they were equal to number of occupation in the SOC array.

1. Print a count of the total number of nodes in the BST.

```
Total Height of BST: 19
Height of left subtree: 15
Height of right subtree: 18
Total Number of Nodes: 505
```

2. Print the height of the root of the BST, and the height of its left and its right sub tree. Is it “height balanced”?

Ans. NO, the binary tree is not balanced, as the difference between leaf-node of left sub tree and leaf-node of right sub tree differ by more than 1.

3. Is the BST an efficient implementation of the range occupation query? What is another implementation to compute the answer to a range occupation query? Would it be more or less efficient (does it depend on the value of n)?

Ans. BST is a good data structure for range queries, but in worst case that is a skewed binary tree it becomes $O(n)$ in complexity for searching. So the better implementation would be a self-balancing binary tree, like AVL tree or Red-Black tree, which will have $O(\log(n))$ time complexity for searching. Alternatively since we had static data that is we didn't need to change the array in every query we could have use a sorting algorithm of time complexity of $O(n \log(n))$ and then used binary search to find value in range. The alternative solutions will depend on n as mentioned above with their individual time complexities.

Hash-Table:

Experimentation Run

I found this experiment a little tricky, in this experiment I calculated the max-length chain where 0 is when there is no entry at index of hash table; 1 is the hash table entry is not NULL; 2 is when there is one collision and 3 is when there are one or more collisions. Note that the total number of entries is more than the Number of occupations in the file this is because there certain entries where they have more than one SOC code. And all the different SOC codes hashed at different entries.

1. Print a table that lists for each chain length l , $0 \leq l \leq l_{\max}$, the number of chains of length l , up to the maximum chain length l_{\max} that your hash table contains.

Length	Number
0	0
1	423
2	49
3	2
Load Factor: 0.35	

2. Compute and print the load factor for the hash table. Do you consider the hash function to be a “good” one for the SOC codes?

Ans. Yes, the hash function is good for SOC codes as the load factor is quite low. As evident from the above table there 429 entries that don't have further chains that is they exist direct at index. While only 49 have linked list of size 2, and only 2 have size 3.

3. Is the hash-table an efficient implementation of the find occupation query? What is another implementation to compute the answer to a find occupation query? Do you think it would be more or less efficient? Why or why not?

Ans. Hash-table is a very efficient implementation of find occupation as it has constant look up time. The other implementation would be simple sorting of Array and then doing a binary search in $O(\log(n))$ time. It will be less efficient as first we'll need to sort the array $O(n \log(n))$ then do binary search $O(\log n)$. Although binary search is an efficient algorithm it not as efficient as hash-table. While in hash-table we are just going through the array once and inserting value in Hash table $O(n)$ and then searching in hash table $O(1)$. That's why hash-table is a better approach.

Complete Results:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Query: find max total 10
Number of nodes: 505
Height of tree: 8
Height of left subtree: 8
Height of right subtree: 7
Query: find occupation 11-3031
Total Height of BST: 19
Height of left subtree: 15
Height of right subtree: 18
Total Number of Nodes: 505

Length | Number
0      | 0
1      | 423
2      | 49
3      | 2
Load Factor: 0.35

[1] + Done
sidraw@sidharthPC:~/Desktop/Project2$ /usr/bin/gdb --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-5wx1c5v1.ifz" 1>"/tmp/Microsoft-MIEngine-Out-241hueub.dfj"

```