Max Heap

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

    The number of nodes in the max heaps created by my Assignment 2 algorithm is a constant: 505. This is due to the fact that the number of SOC elements to be put into an array to make the max heap is 505.

2. Print the total height of the max heap, as well as the height of it's left and right subtrees

    Since a heap has to be a balanced binary tree of sorts (although the nodes are linked through indexes rather than references), the height of this tree can be found using $\log_2(505)$, which is ~8.98, meaning that the height of the tree must be 9. The left subtree will always have a height equal to the height of the tree. However, the right subtree can sometimes be 1 less than the height of the tree. In this case, the height of the right subtree is 8, since 505 != $2^n - 1$ for any integer value of n.

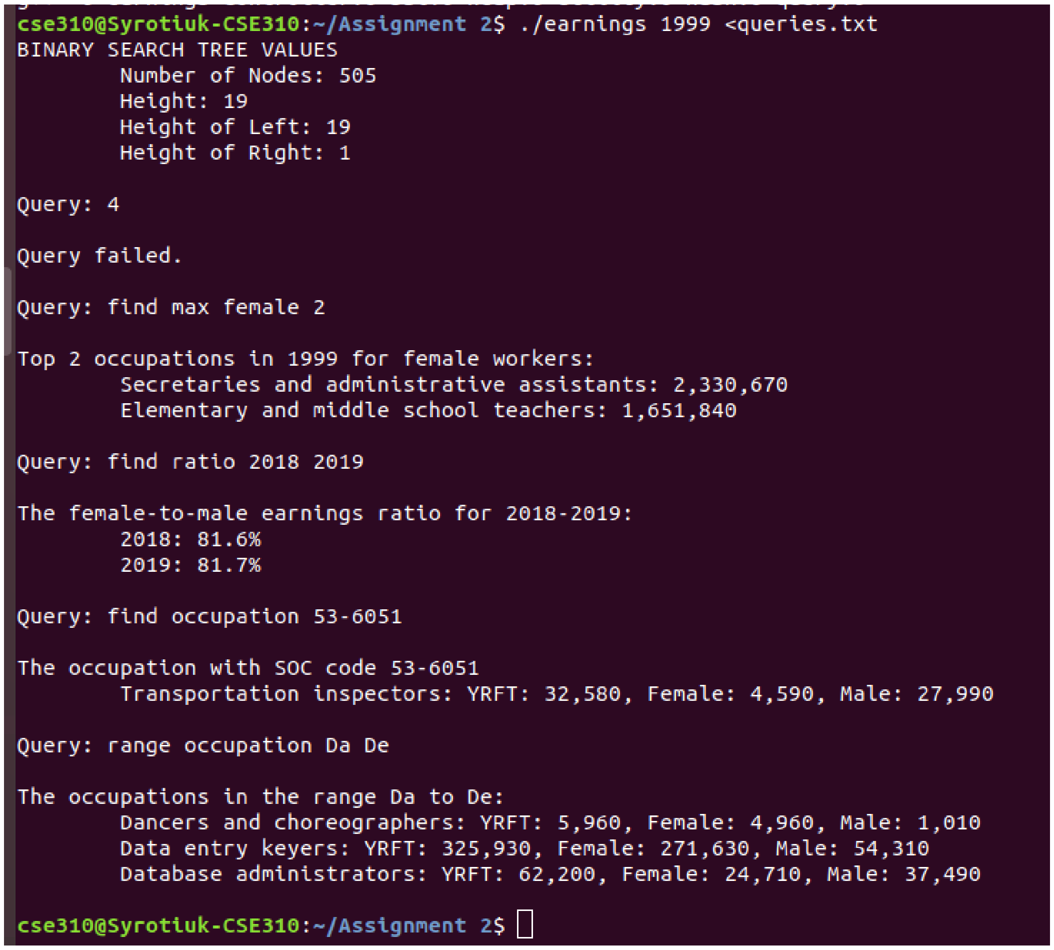
3. Is the max heap an efficient way to perform the find max query? What's another implementation for find max? Which is more efficient?

    A max heap is an efficient way to perform the find max query, due to its average runtime being O(log(n)). Another way to implement this would be an iterative solution, where the program creates an array of the size to print, and then finds the largest items to fit into that array, keeping the items sorted within the array so that it stays largest-smallest. Between the max heap and this iterative implementation, the max heap is more efficient due to its O(log(n)) runtime being faster than an iterative algorithm's O(n) runtime.

Binary Search Tree

1. Find the total number of nodes in the binary search tree.
2. Find the height of the search tree, and its left and right subtrees.

Image for (1) and (2):



```
cse310@Syrotiuk-CSE310:~/Assignment 2$ ./earnings 1999 <queries.txt
BINARY SEARCH TREE VALUES
        Number of Nodes: 505
        Height: 19
        Height of Left: 19
        Height of Right: 1

Query: 4

Query failed.

Query: find max female 2

Top 2 occupations in 1999 for female workers:
        Secretaries and administrative assistants: 2,330,670
        Elementary and middle school teachers: 1,651,840

Query: find ratio 2018 2019

The female-to-male earnings ratio for 2018-2019:
        2018: 81.6%
        2019: 81.7%

Query: find occupation 53-6051

The occupation with SOC code 53-6051
        Transportation inspectors: YRFT: 32,580, Female: 4,590, Male: 27,990

Query: range occupation Da De

The occupations in the range Da to De:
        Dancers and choreographers: YRFT: 5,960, Female: 4,960, Male: 1,010
        Data entry keyers: YRFT: 325,930, Female: 271,630, Male: 54,310
        Database administrators: YRFT: 62,200, Female: 24,710, Male: 37,490

cse310@Syrotiuk-CSE310:~/Assignment 2$ 
```

3. Is the binary search tree an efficient implementation of the range occupation query? What is another implementation? Which is more efficient?

The binary search tree is created in O(nlog(n)) time, the fastest a comparative algorithm can sort a data set. This data set can then be used in O(log(n)) time to find the beginning of data to print, and O(n) time to print all the data in the desired range, making the query itself an efficient algorithm.

Another possible implementation of the query would depend on pre-sorted data, much like the binary search tree of this assignment. This implementation would simply parse through a linear set of data in order to print out occupations in a given range, making for an O(n) run time. However, pre-sorting the data could, at fastest, be done in O(nlog(n)) time, such as could be done with a merge sort algorithm.

Between these two algorithms, the binary search tree is more efficient. While both data sets are created in O(nlog(n)) time, and the printing is done in O(n) time, finding the values to print are different, with the bst’s time being O(log(n)) and the iterative’s time being O(n). However, the iterative’s O(n) runtime could be acceptable or even preferred given a small n.

Hash Tables

1. Chain information table:

In Chain is the number of elements in a chain.
Repeated is how many times “In Chain” elements happened in the program.

In Chain:	Repeated:
1	442
2	49
3	36
4	0
5	0

2. What is the load factor? Is it a good hash table?

Since the total number of elements in the hash table is 527, and the size of the hash table is 1523, the load factor is 0.346. This is a good hash table since not only does it have a small load factor, but the vast majority of its elements are independent in their respective chains, limiting the run time of the hash searches.

3. Is the hash table an efficient way to implement the find occupation method? What is another way to implement the query? Which is more efficient?

This hash table is an efficient way to implement the find occupation method since it runs in near constant time. Another way to implement this query is to use the binary search tree used in the range occupation query, which was created alongside the hash table. However, even though the binary search tree would run in O(log(n)) time after creation, the hash table would still be more efficient with its O(c) run time. During creation as well, the hash table has efficiency O(n) while the binary search tree has O(nlog(n)).