CSC 1120 Lab 14 Assignment

Section 1: Analysis

Runtime Analysis of Ordered List for add()
With regards to the runtime analysis, we take into consideration as to whether the backing structure is an array list or a linked list.

Because the method uses a list iterator, it doesn't matter whether we use a linked list or an array list, because ultimately it becomes a list iterator.

With that in mind, we can say that the runtime is O(n) because we iterate through every element in the list iterator to find the right place to add the word. This is possible using a while loop. Hence, the runtime is dependent on the size of the list.

Runtime Analysis of Ordered List for exactMatch()
The runtime of this method is O(log n), and this is because we use binary search to look for the word in the items array. Binary search works by checking if an element is less than or greater than a certain element in the middle of the list.

If it is equal to the item in the middle of the list, then the element exists. Else we cut the list in half. If the element to be looked for is smaller than the element at the middle of the list, then we cut the list in half by removing all the elements that are greater than the element at the middle of the list. Else, we remove elements that are smaller than the element at the middle of the list.

And this is regardless of the backing structure, of whether it’s a Linked List or Array List, because ultimately, we’re searching for an exact match using binary search.

Runtime Analysis of Ordered List for allMatches()
With regards to the runtime analysis, we take into consideration as to whether the backing structure is an array list or a linked list. Because the method uses a list iterator, it doesn't matter whether we use a linked list or an array list, because ultimately it becomes a list iterator.

With that in mind, we can say that the runtime of the method is O(log n). And this is because we use binary search. Binary search works by checking if an element is less than or greater than a certain element in the middle of the list. If it is equal to the item in the middle of the list, then the element exists. Else we cut the list in half. If the element to be looked for is smaller than the element at the middle of the list, then we cut the list in half by removing all the elements that are greater than the element at the middle of the list. Else, we remove elements that are smaller than the element at the middle of the list.

We use the while loop to iterate through the list iterator to search for words that match the prefix. This would give a runtime of O(n), as the runtime is dependent on the size of the length of the iterator. However, log(n) is more dominant.

Furthermore, the number of elements, m, returned by allMatches() has an impact. Because we don't have to loop throughout the entire way thanks to a binary search tree, we can say there are two runtime operations, one being O (log n) and O(m), with O(m) being more dominant, hence it's O(m).

Runtime Analysis of Unordered List for add()
Given the way the method was implemented, the runtime for this method would be O(n). This is because we use a for loop to check whether the word, we're trying to add is existent within the array list. Hence if it is, then we don't add the word and return false. This is when the backing structure is an Array List.

If the backing structure for this method was a Linked List, the runtime would be O(n). This is because using a linked list implement would mean that we would have to use a walker to walk through the linked list and check each node to see if the node we're trying to add is already present in the linked list. If not, then we add the node to the linked list.

Runtime Analysis of Unordered List for exactMatch()
The runtime for this method is O(n) because we use the .contains(), we must check every element in the items array. And this is regardless of the backing structure, of whether it's a Linked List or Array List, because ultimately, we're having to iterate entirely.

Runtime Analysis of Unordered List for allMatches()
The runtime for this method is O(n), regardless of whether the backing structure is an array list or linked list.

If it was an array list, we go through the elements one by one until the end. The same with a linked list backing structure, where we use a walker to walk through the entire linked list until the end.

Furthermore, the number of elements, m, returned by allMatches() doesn't impact the runtime because we must iterate throughout the entire list Iterator as the elements are unordered. Hence, the runtime remains O(n), as n = m.

Runtime Analysis of Binary Search Tree for add()
The runtime of this method is O(n). This is because we use a for loop to iterate through elements in the items TreeSet to check if the word is present. Ideally, it should be O(log n), because we should be using .contains(), which will perform a binary search.

Runtime Analysis of Binary Search Tree for exactMatch()
The runtime of this method is O(log n). This is because we don't have to iterate throughout the entire list to check for the presence of a word. Hence, our runtime isn't dependent on the size of the list, as the contains() method performs a binary search on the binary tree.

Runtime Analysis of Binary Search Tree for allMatches()
The runtime for this method is O(n). This is because we convert the TreeSet Binary Tree into a Iterator, which is then used to iterate throughout to find elements that start with the prefix. Hence, the runtime is dependent on the size of the length of the iterator.

However, we must also take into consideration the m, which is the number of elements. For allMatches(), the runtime would be log n + m, because our starting point isn't from the beginning, but rather from the middle of the list. However, because m is more dominant, our runtime would be O(m).

Runtime Analysis of Trie for add()
The runtime for this method is O(1) because the number of edge cases is constant, which is the number of letters from a to z, being 26 in total, and this doesn't change throughout. This is in reference to the number of words.

Runtime Analysis of Trie for exactMatch()
The runtime for this method is O(1) because the number of edge cases is constant, which is the number of letters from a to z, being 26 in total, and this doesn't change throughout. This is in reference to the number of words.

Runtime Analysis of Trie for allMatches()
The runtime for this method is O(n), where n is the length of the prefix. Furthermore, we need to take into consideration about m, which is the number of elements returned.

Taking into consideration the edges of the trie, because the range of letters from a to z is 26, we raise the number to the power of the difference between depth of the trie (L) and the depth of the prefix (S). Hence, the runtime would be S + 26^L-S. Given that m here represents 26^L-S, we would say m is the more dominant runtime here, hence overall runtime is O(m).

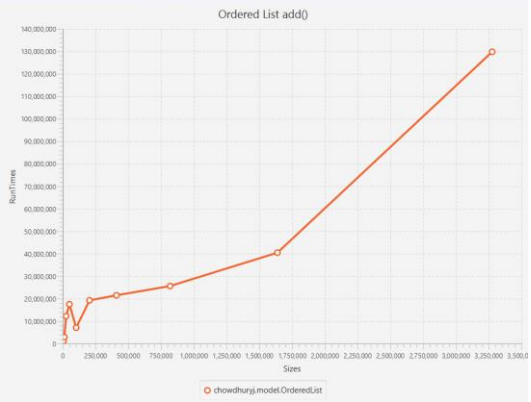
Runtime Analysis of Hash Table for add()
The runtime for this method is O(1). This is because given that we have a low load factor, it leads to a lower number of collisions, hence we get O(1). However, if there is a very high load factor, this can lead to several collisions, and assuming we do chaining, we will have to iterate through the long linked lists, and this will get us a runtime of O(n).

Runtime Analysis of Hash Table for exactMatch()
The runtime for this method is O(1). This is because given that we have a low load factor, it leads to a lower number of collisions, hence we get O(1) as we look for an exact match. However, if there is a very high load factor, this can lead to several collisions, and assuming we do chaining, we will have to iterate through s longer linked list, and this will get us a runtime of O(n).

Runtime Analysis of Hash Table for allMatches()
Ultimately, the runtime of the method is O(n), because we're using an iterator. The m doesn't impact anything here because ultimately we're having to iterate from the start to the end.

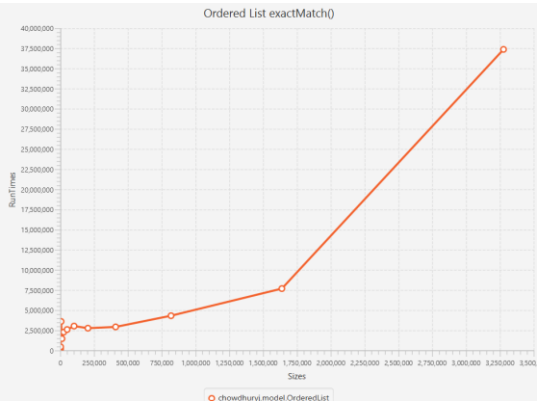
OrderedList add() Benchmarking

On the right side of this table, we have a benchmarking graph for the add() method of OrderedList. The OrderedList is backed by a LinkedList. The line graph appears to have a linear trend. This indicates that this method has a runtime of O(n). As a result, it lines up with my analysis that the add() method has a runtime of O(n).



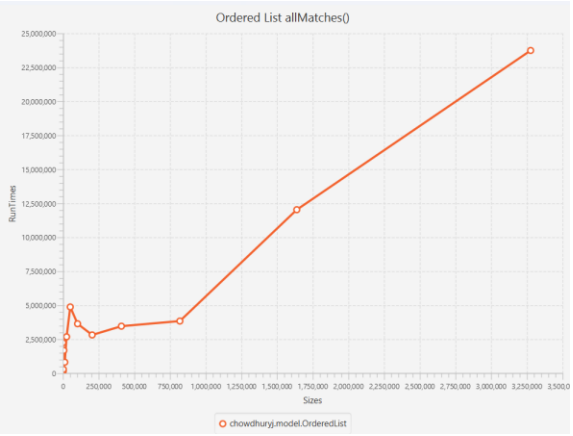
OrderedList exactMatch() Benchmarking

On the right side of this table, we have a benchmarking graph for the exactMatch() method of OrderedList. The OrderedList is backed by a LinkedList. The line graph appears to have a logarithmic trend. This indicates that this method has a runtime of O(n). As a result, it lines up with my analysis that the exactMatch() method has a runtime of O(log n).



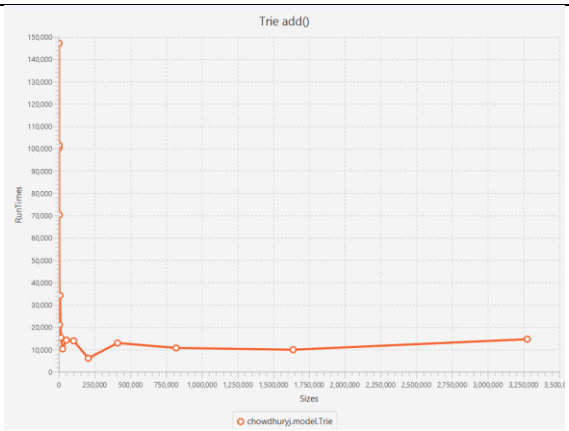
OrderedList allMatches() Benchmarking

On the right side of this table, we have a benchmarking graph for the allMatches() method of OrderedList. The OrderedList is backed by a LinkedList. The line graph appears to have a linear trend. This indicates that this method has a runtime of O(m). As a result, it lines up with my analysis that the allMatches() method has a runtime of O(m).



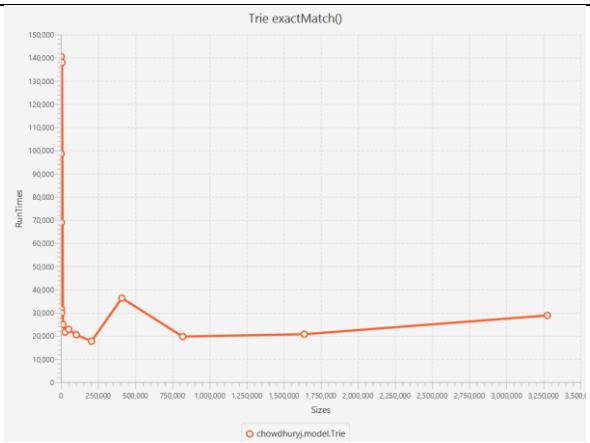
Trie add() Benchmarking

On the right side of the table, we have a benchmarking graph for the add() method of Trie. The line graph appears to have a constant straight line. This indicates that this method has a runtime of O(1). As a result, it lines up with my analysis that the add() method has a runtime of O(1).



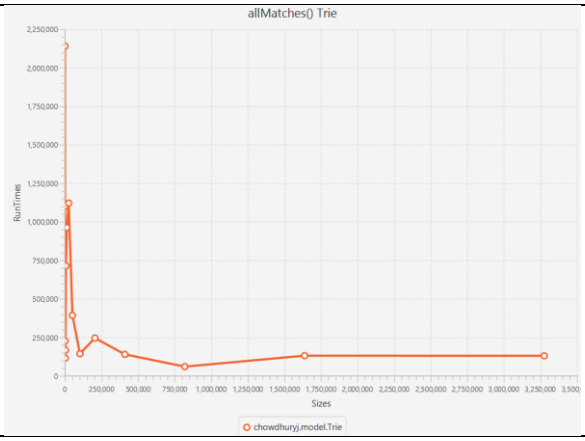
Trie exactMatch() Benchmarking

On the right side of the table, we have a benchmarking graph for the exactMatch() method of Trie. The line graph appears to have a constant straight line. This indicates that this method has a runtime of O(1). As a result, it lines up with my analysis that the exactMatch() method has a runtime of O(1).



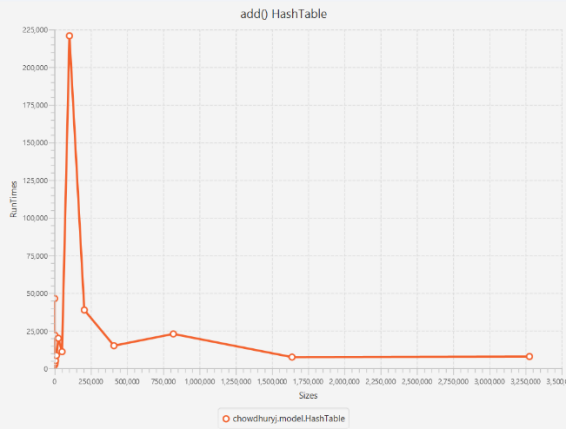
Trie allMatches() Benchmarking

On the right side of the table, we have a benchmarking graph for allMatches() method of Trie. The line graph appears to have a constant straight line. This indicates that the method has a runtime of O(1). As a result, it lines up with my analysis that allMatches() method has a runtime of O(n), where n is the length of the prefix, and since the length of the prefix is constant, this means we get O(1).



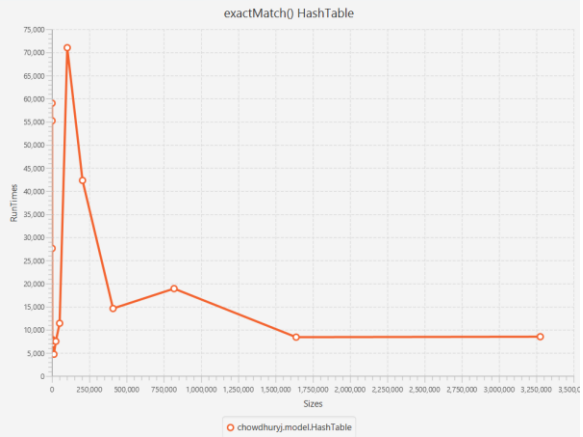
HashTable add() Benchmarking

On the right side of the table, we have a benchmarking graph for add() method of HashTable. The line graph appears to have a constant straight line. This indicates that the method has a runtime of O(1). As a result, it lines up with my analysis that add() method has a runtime of O(1).



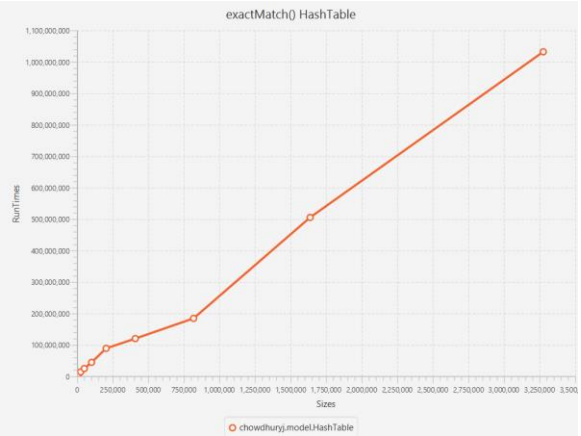
HashTable exactMatch() Benchmarking

On the right side of the table, we have a benchmarking graph for exactMatch() method of HashTable. The line graph appears to have a constant straight line. This indicates that the method has a runtime of O(1). As a result, it lines up with my analysis that exactMatch() method has a runtime of O(1).



HashTable allMatches() Benchmarking

On the right side of the table, we have a benchmarking graph for allMatches() method of HashTable. The line graph appears to have a linear trend. This indicates that the method has a runtime of O(n). As a result, it lines up with my analysis that allMatches() method has a runtime of O(n).



I decided on the number of the strings by ensuring I had enough to create a good benchmark that can clearly show the runtime of each method of every data structure. The data gathering process can be improved by having strings of different sizes within the list rather than having strings of a constant size.

Section 3: Reflection

In the final section of your report, address the following:

1. What questions would you try to answer to improve your understanding of the project?
2. Describe your greatest strength with respect to this course.
3. Describe an area related to this course where you believe you could improve.
4. Give an example of a software project you think would be valuable that you believe you could now implement.

What questions would you try to answer to improve your understanding of the project?

I would try to improve my understanding of the project by answering questions with regards to how each method within a data structure works, and by writing down the code for it, which would help improve my critical thinking skills. For example, understanding how the add() method works for any data structure, and then writing code on it.

Describe your greatest strength with respect to this course?

My greatest strength with respect to this course would be working on understanding how each data structure works, and then implementing each data structure, for example a stack, queue, circular queue, linked list etc.

Describe an area related to this course where you believe you could improve?

I could work on improving more on file handling, as I struggled with it the most during the first three weeks, as the file handling labs were a little difficult to do, mostly because the concepts were, to some extent, difficult to grasp and took some time getting used to.

Give an example of a software project you think would be valuable that you believe you could now implement.

During the summer, I am planning to build a website that can track a user's listening history as well as tell the user what their top songs were for the month, and I plan to implement that using OOP principles, as well as using data structures that can keep track of a user's listening history.