CS 340 Project II

Insertion and Searches

Jordan Kramer

Algorithms and Data Structures 340

John Matta

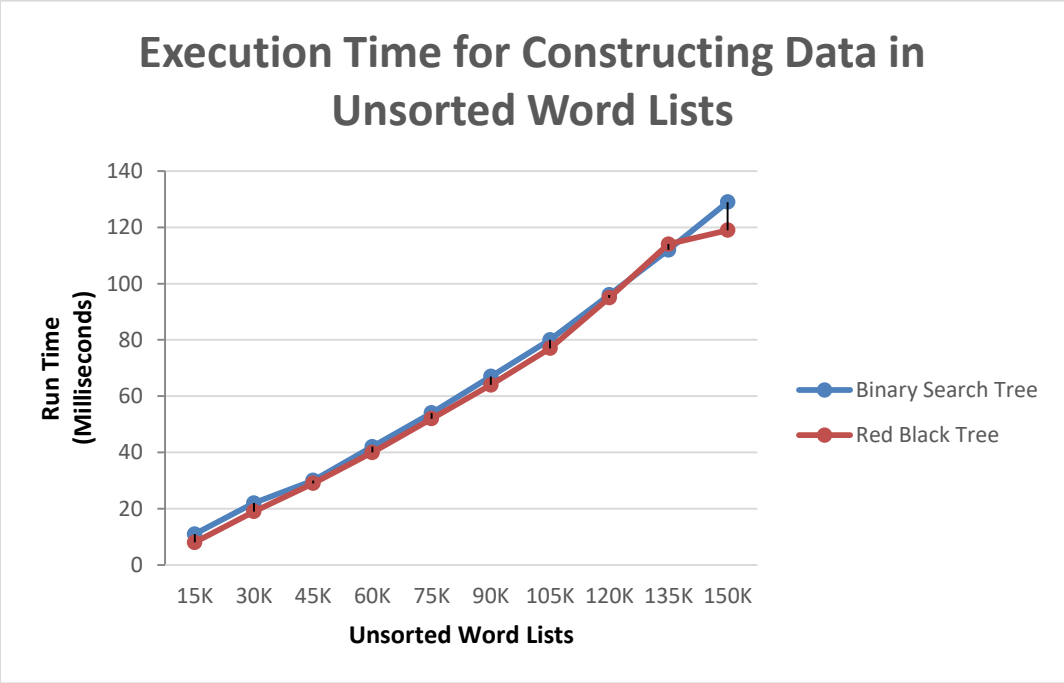
October 1st, 2018

Binary Search Trees and Red Black Trees may not be as different as one may think. I started the code by first working on the binary search tree. Since I wanted to see how different coding structures affect different trees, I went ahead and coded my insertion and search functions differently. I made my insert function recursive where as I made my search function iterative. My binary search tree saves the data into the tree “bst” which is a new binary search tree. This tree was implemented using an interface. The time complexity of my recursive binary search tree insert on average is O(log (n)) . Its worst case would end up being O(n). As for the iterative tree search, the average case would be O(log (n)). The worst case would be O(n). The iterative search is more efficient than using a recursive search because it shouldn’t use as much stack space. The reverse goes for the recursive insert. It is less efficient than the iterative because my recursive requires more stack space which in turn means you are using more memory. As for my red black tree, it is fairly similar in structure. I implemented the same exact iterative search for the red black tree. As for the insert I also did recursion, but the difference in this method was that we had to add an insert-fixup method. The red black tree saves the data into the tree “rbt” which is a new red black tree. This tree was implemented using an interface. The time complexity for the insert is still O(log(n)) and the worst case is O(n). The search is also O(log(n)) in average case and the worst case is still O(n). The advantage of the iterative search is still that it requires less memory space its recursive counterpart. The disadvantages of the iterative search are still that it does require more memory than the iterative insert.

The queries aren’t too far apart as well. An example of a fast query in the perm15k file would be searching for the word ATHROCYTES. An example of a fast query in the sorted15k file

would be searching for the word AA. Both of these run times are 0ms. These both run so fast that they run in less than a millisecond. These two queries are so fast because ATHROCYTES is the first word in perm15k and AA is the first word in sorted15k. Since the search is iterative, it starts at the beginning. This would in turn return the first word instantly. An example of a longer search time could be found if one was looking for the word SURGEONFISHES in the sorted150k file. This search takes at least 3ms. It takes so long because it is the last word in the sorted file. When searching in a binary search tree, one would have to traverse the whole list of the tree before finding the word.

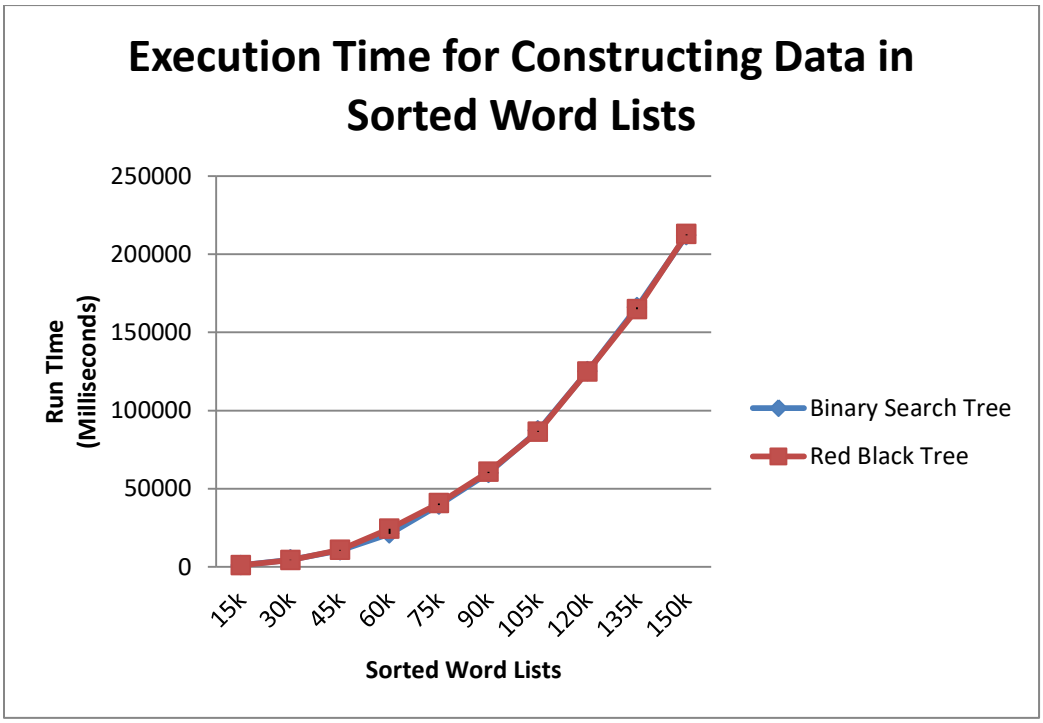
When moving on to graphing the execution time for constructing my data, I was actually surprised by the results. Below you will see the timings for the unsorted word lists.



Surprisingly enough, both lines seem to rest atop of one another. I did expect some similarities between the two because both the binary search tree and the red black tree are on O(log (n))

for their average case time complexities. What is unexpected to me is the last point on the graph. The binary search tree took 129ms whereas the red black tree only took 119ms. I figured that the red black tree would most likely take slightly longer than its counterpart because the red black tree also has to do the insert fix-up.

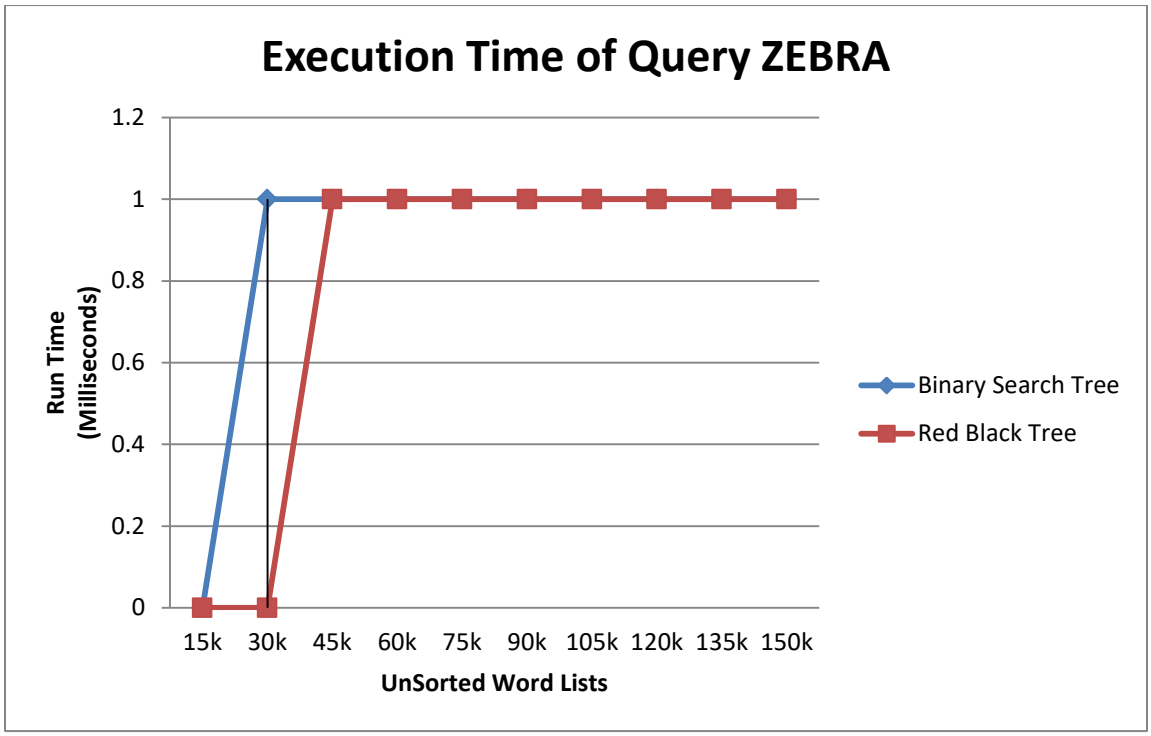
Both of these trees' time complexities actually shift when moving over to the sorted lists. Below you will see the timings for the sorted word lists.



This plot clearly shows how big of a difference there is between the sorted and unsorted word lists. Above, you will see that both the binary search tree and the red black tree are near identical on timings. This is the case because both trees run on the time complexity of O(n). They run on this time complexity because the listed is sorted. A sorted list makes a linked list in a binary search tree.

Looking at sorted and the unsorted lists as a whole, the unsorted list for perm15k ran at 11ms for a binary search tree and at 8ms for a red black tree. The sorted list for sorted15k ran at 1081ms for a binary search tree and at 1072ms for a red black tree. As you can see, there is a vast difference between an algorithm that runs on O(log (n)) verses one that runs on O(n).

Queries also can be affected by the differences in the sorted and unsorted lists. While looking at unsorted lists, I searched for the keyword “ZEBRA” below.



The query correctly reflects the time complexity of O(log (n)). You can tell by its shape, it appears to be curving off like the log (n) function does.

In conclusion binary search tree’s and red black tree’s average insert time complexity is O(log(n)) and their worst case insert time complexity is O(n). Binary search tree’s and red black tree’s search function both run on O(log(n)) and their worst case is O(n).