Assignment 3: Tree Report

Alvin Dang

Department of Computer Science, San Jose State University

CS 146: Data Structures and Algorithms

Vidya Rangasayee

November 5, 2023

1) Explain your choice of sorting algorithm. Its efficiency and disadvantages.
The sorting algorithm I chose was tree sort. Tree sort works by building a binary search tree (BST) using the elements that are to be sorted. From there, it performs an in-order traversal on the BST to get all the elements in a sorted order. BSTs are simple data structures that always maintain their sorted property whether or not they are undergoing an insertion or deletion operation.

I chose tree sort as it made sense to use given the requirements and instructions of Assignment 3. It may not be the most efficient or amazing sorting algorithm, but its simplicity made it an ideal choice for me.

1 A) Efficiency
In terms of complexity, its average case time is O(n log n). Where when one item is added to a BST, it takes on average O(log n) time and when n items are added, BST takes O(n log n) time.

However, in its worst case, it will run at O(n^2). The worst case occurs when the input is already sorted or almost completely sorted. The reason for issues to arise in this case is because the tree will become unbalanced from the insertion operation. During this case, the tree will look less like a tree and more like a diagonal linked-list.

1 B) Disadvantages
Tree sort utilizes recursion for its algorithm. In general, although a fun concept, recursion has many drawbacks with the main one being lots of overhead - especially for small lists. In cases such as small lists, recursion can lead to stack overflows and use up too much memory. Furthermore, as discussed previously tree sort becomes incredibly inefficient when the list to sort in nearly or completely sorted prior to tree sort being run on it.

2) How do you interpret the results in 5b-5e?

	2-3 Tree Search Time	Binary Tree Search Time	Linear Search Time	Binary Array Search Time
Min:	40750 ns	23917 ns	17323083 ns	16917 ns
Max:	346834 ns	208708 ns	23210167 ns	973958 ns
Avg:	48977 ns	82153 ns	17877606 ns	29984 ns

My interpretation of the results is that binary array search is the fastest of the four searches tested. I believe this is fairly in line with the general consensus that binary array search is a very efficient searching algorithm. Its average run time is O(log n) where n is the number of elements in a tree. What I find interesting about the results is that 2-3 Tree Search (uses a

depth-first-search or DFS) turned out to be faster than the Binary Tree Search. DFS's performance depends heavily on the size of the tree and its structure. And is usually regarded as running slower than Binary Tree's O(log n) run time. However, in this case DFS showed to be on average only half the time as Binary Tree search. Nonetheless, Binary Tree Search and Binary Array Search's minimum times were fairly close to each other and fairly far from DFS minimum. Indicating that they still have potential to be much faster than DFS depending on the scenario. As for linear search time, its results are what I expected for the most part. With a usual run time of O(n) and a large file to search, it's no surprise to me that it was significantly slower than the rest.

3) What are the reasons you may choose linear search over other approaches?
In scenarios where the data is not sorted, the binary search methods are immediately eliminated. And for small data sets, linear search becomes a reasonable choice as it is simple to implement and efficient enough for the data size. In the case of a small data set, the benefits of a DFS isn't completely realized. Thus, the implementation of a DFS becomes less reasonable to do when linear search would net similar run times for less work.

4) How do you interpret the results in 6a-6b?

	2-3 Tree Deletion Time (Top 10%)	Binary Tree Deletion Time (Top 10%)	2-3 Tree Deletion Time (Bottom 10%)	Binary Tree Deletion Time (Bottom 10%)
Min:	791 ns	125 ns	875 ns	166 ns
Max:	3385500 ns	1510500 ns	885250 ns	109541 ns
Avg:	208 ns	93 ns	54 ns	6 ns

The results here are entirely what I had expected. The overarching theme of the results for deleting the top 10% versus deleting the bottom 10% is that the bottom 10% will always take less time. This is what I had expected as deleting the bottom 10% should require less rebalancing. The need to rebalance less for deleting the bottom 10% comes from the fact that the nodes towards the bottom are usually leafs or have less children that would be affected by their removal. Because they have less children or no children, removing these nodes would be a simpler process than removing the top 10% of nodes since the bottom 10% have less nodes that they can affect. To elaborate, the top 10% would have the other 90% of the tree to check and valid for balancing everytime one of their nodes is removed. This process could prove to be lengthy and consuming compared to removing the bottom 10%. Which is what was reinforced in my results for 6a-6b. With that said, I think it was very interesting to see such a stark difference in average run times when comparing the two.