Project 2: Non-Linear Data Structures Documentation

Kathleen Tiley

Introduction

For this project, I implemented insert, delete, and traverse functions for an AVL Tree and a Min Heap and measured the time each data structure took to execute each function for input sizes of 500 (small), 5000 (medium), and 50000 (large). To allow for a more robust comparison of the data structure's performance, I restricted the operation of the AVL Tree delete to remove only the root node of the tree. This creates consistency with the Min Heap delete operation, which only removes the minimum element stored in the leftmost position in the heap array. I maintained consistency between the traverse operations by using a level order traversal for the AVL Tree and iterating through the heap array for the Min Heap.

Step 3: Tables

Data Structure 1: AVL Tree

Insert – AVL Tree	Ascending	Descending	Random
Small	1021	611	700
Medium	9359	7738	7990
Large	123101	99082	105914

Traverse – AVL Tree	Ascending	Descending	Random
Small	62057	625313	62369
Medium	955435	801072	718847
Large	11388122	9643799	8713253

Delete - AVL Tree	Ascending	Descending	Random
Small	367	323	357
Medium	7481	4458	5038
Large	66456	55886	60263

Data Structure 2: Min Heap

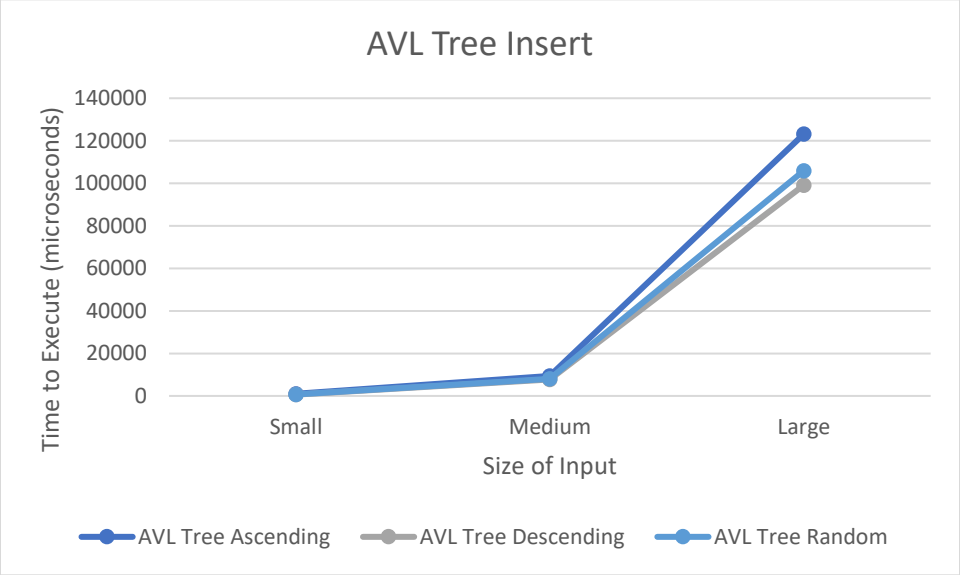
Insert - MinHeap	Ascending	Descending	Random
Small	250	506	250
Medium	2307	7068	2473
Large	22504	83917	26097

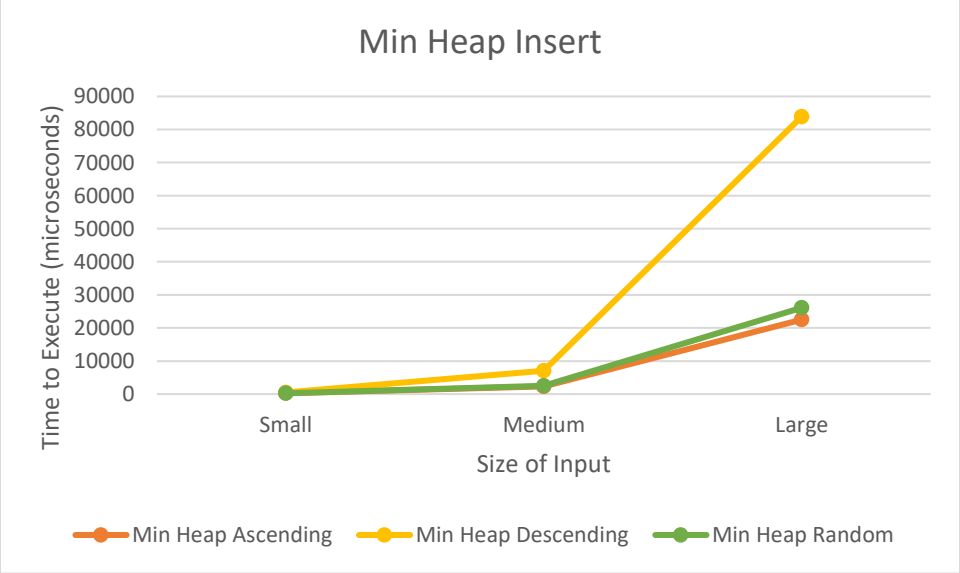
Traverse - MinHeap	Ascending	Descending	Random
Small	72900	61444	61302
Medium	954589	799262	674860
Large	11000592	8794700	8317554

Delete - MinHeap	Ascending	Descending	Random
Small	659	572	588
Medium	9441	7826	8064
Large	102737	100938	104695

Step 4: Practical Commentary and Reflection

Function 1: Insert





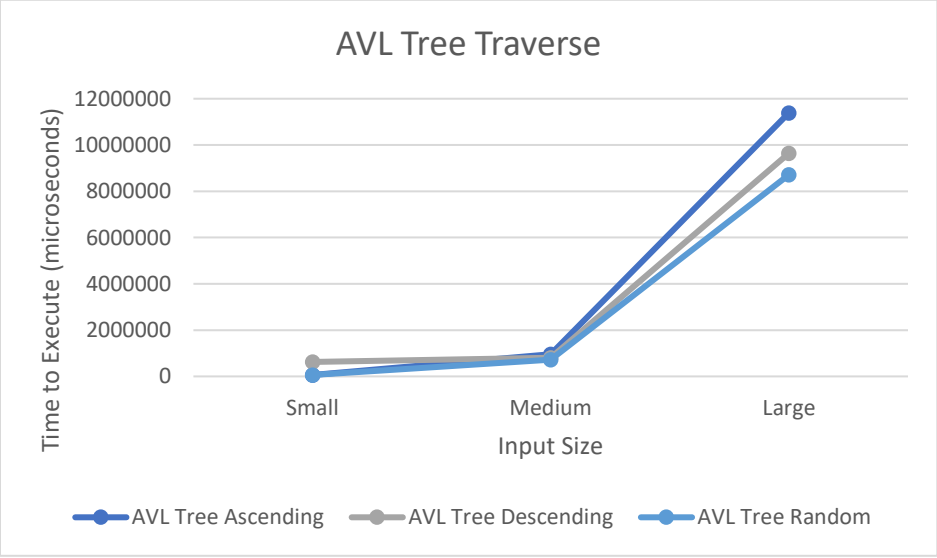
When inserting the large input, both the AVL Tree and the Min Heap required significantly more time to execute than they did when inserting the small and medium sized input sets. This jump is more pronounced in the AVL Tree, particularly when the input was sorted in ascending order.

The AVL Tree took longer to insert elements sorted in ascending order at all input sizes than either descending or random order, which took about the same time to execute for all input sizes. While I expected that elements sorted in ascending order would take longer to insert than randomly sorted elements, because the tree must perform more rotations, I also expected that the ascending and descending order insert times would remain similar. Instead, the descending insert time is closer to the time for the randomly sorted input.

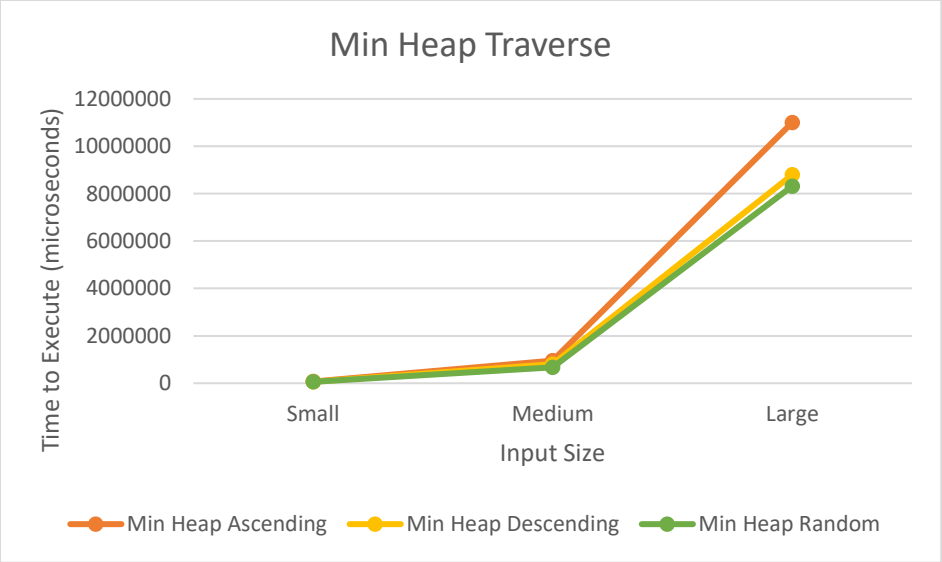
When inserting the descending input, the Min Heap took significantly more time to execute than the ascending or randomly sorted input. Although the difference is not visible in the graph for the small input, the data table in Step 3 shows that inserting the descending input took twice as long as ascending or random. I expected this behavior, because for the descending input order the Min Heap must move each new element all the way to the front of the array.

As input size increased by a factor of 10, execution time increased by a factor 12 for the AVL Tree and 11 for the Min Heap on average.[1] For both the AVL Tree and the Min Heap, a single insert is done in O(logn) time. Because we are calling insert m times, where m is the input size, we should expect the execution time to increase at a rate slightly above linear.

Function 2: Traverse



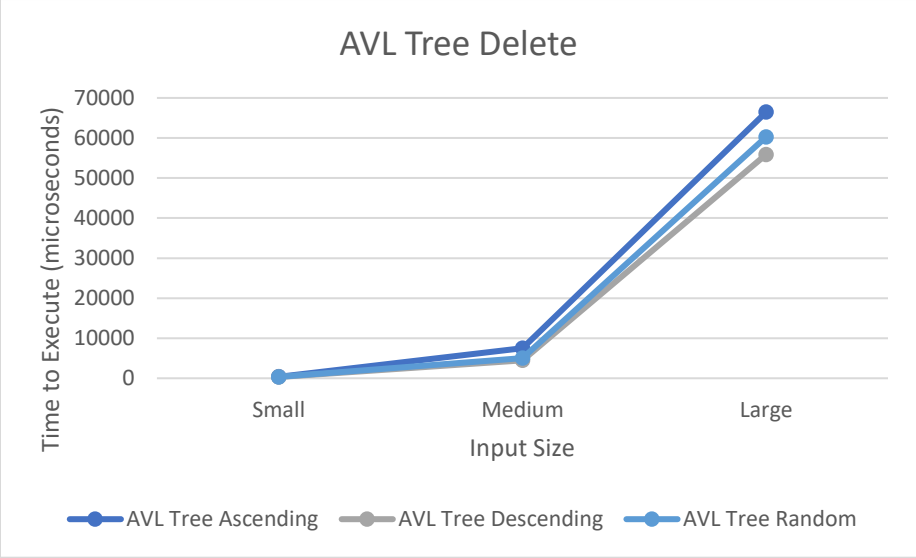
[1] The average increase factors for execution time were calculate by finding t_medium/t_small and t_large/t_medium for each input order for a given data structure and function and taking the average.

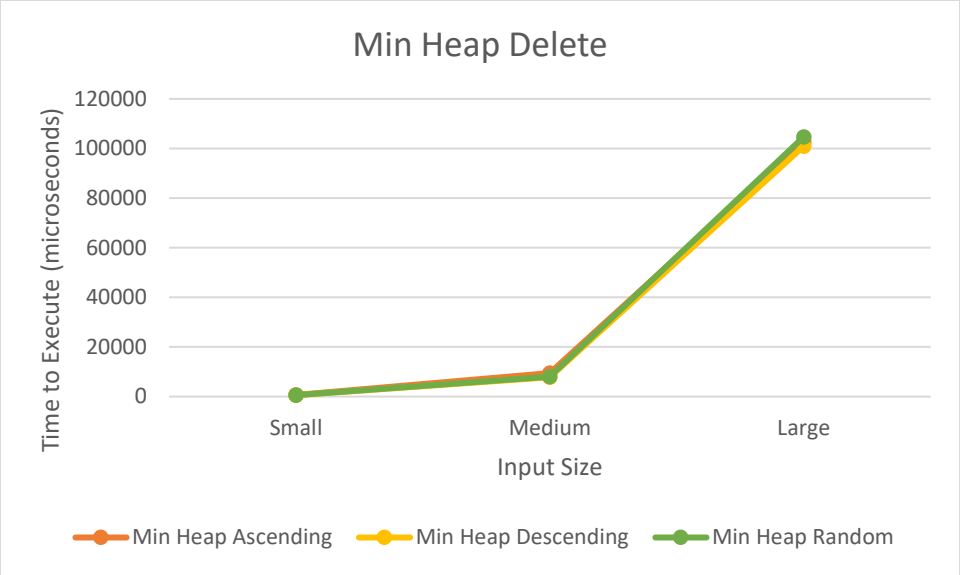


At all input sizes and for all orderings, the AVL Tree and the Min Heap executed the traversals in about the same execution time. Although the difference is not visible on these graphs due to the scale, the traverse operation took significantly longer for both data structures than insert and delete at all input sizes and orderings. This is expected, since the traversal operation must touch every node or element to complete and is thus O(n) time complexity, as opposed to insert and delete which enjoy O(logn) or O(nlogn) time complexity (n is the input size).

As input size increases by a factor of 10, traversal execution time increases by an average factor of 11 for the AVL Tree and 12 for the Min Heap. Because both traversals execute in O(n) time, I expected execution time to increase by a factor closer to 10, since a traversal of 10 times as many elements should take 10 times as long. However, many different factors can affect execution time.

<u>Function 3: Delete</u>





At all input sizes and for all orderings, the AVL Tree and the Min Heap deleted elements in about the same execution time. However, while the order of the input affected the execution for the AVL Tree, it did not significantly affect the execution time for the Min Heap. Because both delete functions only delete the root node, instead of deleting in ascending, descending, or a random order, the effect of the input order is felt only through the insert operation. The AVL Insert operation produces a different tree based on the input order, and likely produces a slightly better balanced tree when the input is random than when it is sorted. As a result, the delete operation for random input may conduct fewer operations than for sorted input.

While the Min Heap Insert operation for descending input took significantly longer than the ascending and random order equivalents, there is no equivalent slow down for the delete function. This is likely due to the nature of the Min Heap implementation. Because the Min Heap always stores the smallest element at the root or first array index and does not sort elements left and right as the AVL Tree does, the heap is probably less sensitive to slight changes in the order of elements resulting from different insert orders.

As input size increased by a factor of 10, delete execution time increased by an average factor of 14 for the AVL Tree and 13 for the Min Heap. For both data structures, the delete function operates in O(logn) time and is called n times, where n is the input size, so we should expect execution time to increase at a rate of nlogn. Although the delete functions have the same Big O time complexity as their insert equivalents, both require more operations overall, which may explain why these increase factors are higher than those found for insert.

Step 5: Theoretical Commentary and Reflection

<u>AVL Tree</u>

1. Insert: O(logn), n := number of nodes in tree

The insert function first finds the location at which to insert the new node, then conducts balancing rotations up the tree from the newly inserted node to the root. In the worst case, the path from the newly inserted node to the root will be as long as the height and each node will require a rotation. Because the height of a balanced tree is proportional to logn, insert will conduct rotations on a maximum of logn nodes, where n is the number of nodes in the tree. For each node visited, insert updates heights, calculates balance factors, and conducts rotations. By updating a height variable within the node class at each level of recursion up the tree, insert can complete all these balancing operations in constant time. As a result, the overall time complexity is O(logn) in the worst case.

2. Traverse: O(n), n := number of nodes in tree

   The traverse function conducts a level order traversal using a queue data structure. The traversal must visit each node in the tree, and at each node conducts constant time queue operations and prints the node. This means the overall time complexity is O(n) in the worst case, where n is the number of nodes in the tree.

3. Delete: O(logn), n := number of nodes in tree

   The delete function first locates the node to delete, which in this implementation is always the root node. If the root node is a leaf or has only one child, the root node can be deleted in constant time. However, in the worst case, the delete function must replace the root node with it's immediate successor by finding the minimum element in the root's right subtree. After this node is found and assigned to the root, it must then be removed from the right subtree. This requires visiting logn nodes, where n is the number of nodes in the tree. At each node visited, delete updates the node's height value, calculates balance factor, and conducts rotations as necessary. By using the height variable within the node class, delete can complete all rebalancing operations in constant time. As a result, the overall time complexity of delete is O(logn) in the worst case.

Min Heap

1. Insert: O(logn), n := number of nodes in tree

   The insert function first adds the new element to the back of the array holding the heap. This is a constant time operation as implemented with the C++ vector class. Then insert calls heapifyUp(), which moves up the array and swaps the newly inserted element with its parent if the newly inserted element is smaller. Although the heap is stored in an array, it can be visualized as a binary tree in which each parent has at most two children, but larger values are not necessarily in the right child or subtree. This means that, in the worst case when the inserted element is minimum element, heapifyUp() will conduct logn swaps where n is the number of nodes in the tree. Since all swap and array access operations are O(1), the overall time complexity for insert is O(logn) in the worst case.

2. Traverse: O(n), n := number of nodes

   The heap traversal function iterates through the array storing the heap using a basic for loop, because the traversal must visit each element in the heap. Each element is printed in constant time, so the overall time complexity is O(n) in the worst case.

3. Delete: O(logn), n := number of nodes

   The Min Heap delete function always deletes the minimum element of the heap, which is stored in first array index and is the root node in the binary tree visualization. To delete the minimum element, the function first copies the last element in the array into the first position. It then calls a function called HeapifyDown(), which swaps the newly copied root element with the smaller of its too children unless the root element is greater than both. Like heapifyUp, this function only conducts at most one swap per level of the tree, so it completes a maximum of logn swaps in the worst case, where n is the number of nodes in the tree. Thus, the overall time complexity of delete is O(logn) in the worst case.

Although the AVL Tree and Min Heap have different implementations, their overall time complexities are the same for each operation tested. The effects of these implementation differences on execution time is discussed in step 4. The data structures also have important differences in functionality. For example, in a fully implemented AVL Tree, it is possible to search for and delete any value. However, a Min Heap can only delete the root node, and a search function is not typically implemented.

Step 6: Learning Commentary and Reflection

This through project, I learned how to implement an AVL tree insert and delete in O(logn) time using a height data member within the Node class. I also implemented a true AVL delete and designed a heap data structure for the first time. I was glad to have the opportunity to improve on my AVL Tree implementation from Project 1 and develop my understanding of heap operations. Through the testing, I learned how to use the chrono library in C++ to time my functions and how to shuffle elements in a vector into a random order. Through the commentary sections, I gained experience analyzing executive time data, comparing it to theoretical time complexity, and hypothesizing about unexpected trends. I found this exercise very valuable and believe it deepened my understanding of time complexity and the functions implemented.

It was challenging for me to figure out how to update the height variables assigned to each node at each stage of the recursion and how to implement the rotations within the AVL Delete function. Conceptually, I also struggled to understand why some comparisons between functions were more helpful than others, although I am happy with the choices I made to alter the AVL delete function and use a level order traversal. Lastly, I spent some time experimenting to find the best graph visualizations that would help me draw insightful conclusions. The large

execution times for the large input sizes created a scale problem that masked the differences in execution times for the small input sizes. However, I decided against removing the large execution times into a separate graph in order to show the whole trend of the data I collected. I feel the graphs I chose are clear and reflect important aspects of the data, but I still resorted to the tables when differences in the execution times for the small input sizes were not visible.