Anton Ha

CSCI - 335

April 7th, 2024

Project 2 Report

Project 2 was to implement an ADT that has only 2 operations, insertion and pop median. The four ways given to implement this project was using a sorted list, sorted vector, heaps, and AVL trees. In regards to implementation using AVL trees and Heaps, I created two of the structures, where one will contain all the elements less than or equal to the median, and the other containing all the elements greater than the median.

For the List implementation of the project, in order to keep the list maintained all throughout every insertion operation, I decided to find the proper index position of the element being inserted in reference to the list. I broke down this thought process into 3 cases, one case is if the proper position is between one and end of the list minus one, the list is empty, and if the element belongs at the end of the list. To find the median of the list, I used the formula ceiling of the list size divided by two, all minus one. I had an iteration pointer that advanced that by the formula to be given the proper median, and then pushed it to the median vector, and then erased the iteration pointer. In terms of big O notation, inserting into this list would be the worst case of O(n), since we have to iterate through the list to find the proper position of the inserted element, and since updating the list would be O(1) because it is just updating the pointers of the nodes in insertion. For big O notation of the popping of the median, it would take ceil(size of list / 2) - 1, which is the same as O(n), and since updating the pointers to delete the median would take O(1), overall popping the median will be O(n).

For the vector implementation, it is very similar to my list implementation, however rather than traversing the list to find the element's position in that specific case, I could use binary search to find the proper position instead and removing the median is the same as the list implementation. In terms of big O notation, insertion would take log(n) as the worst case to find the proper position of the inserted element, and added O(logn) is O(n) to shift the elements down in insertion to retain order, making insertion O(n). Since it takes O(n) to find the median position, and O(n) to shift all the elements down, making it O(n) to pop the median of a vector.

For the heap implementation, I created a heap called 'small' that will hold all the elements smaller or equal to the median, and another heap called 'large' that will hold all the elements greater than the median. I broke down the insertion operation into 2 cases, when the small heap is empty, and when the small heap is not empty. For the small heap being empty, I would just need to insert the value into the small heap since the small heap has to be equal to the size of the large heap or equal to the size of the large heap plus one. Else, I would compare the inserted value to the max element in small, if it's bigger than it, I will insert it to large, else insert it into the small, then I balance to keep the size constraint. Analyzing the big O notation of this implementation, my insertion is O(log n) as inserting into a heap is O(logn) and balancing it would involve two operations of inserting into one and removing from the other, making it O(logn) for insertion operation. Popping the median will take O(logn), since in order to get the median will be O(1) since the median is stored in as the largest element in the small heap, and since the small heap will be implemented as a min heap, it will be accessible at the top, and removing an element of the heap, in our case removing the median wil take O(logn), overall removing the median as O(logn).

For the AVL implementation, I used the same concept as the heap implementation, using an AVL tree to store all the elements that are smaller than or equal to the median called ‘small’, and an AVL tree to store all the elements that are greater than the median. In order to alter the AVL tree implementation for this median tree and contain duplicates, I added an amount data member to both the avl tree class and avl node struct. This allowed me to keep track of the amount of nodes that have the same value for the avl node’s amount, and the tree’s amount allowed me to find out how many nodes, including duplicates, are in a tree, to keep the size constraint of small tree being the size of large or large + 1. Analyzing the big O notation of the AVL tree implementation, insertion operation of this structure would take O(logn) because it takes O(logn) to find the proper position and insert the element, and if necessary to balance the two trees based on the size constraint will take O(logn). For removing the median, it takes O(logn) to find the median, which is the greatest element in the small tree, and removing that element would take O(logn), in total making the removal of the median operation O(logn).

Testing these implementations using the given three input files shows us how fast the certain data structures are compared to each other in the following picture. It surprises me to see that with a smaller amount of instructions, that comparing a vector, which has O(n) operation times, is faster than the tree and heaps, which has O(logn) operation times. However, after testing with large amounts of instructions it is apparent that the heaps and trees complete the data set much faster than the vector and lists, which lines up with our big O complexity analysis as O(n) grows much faster than O(logn). Another aspect that surprised me is the list runtime, as it takes significantly more time to complete the data set compared to the other three, and I believe this occurs due to memory, caching, etc.

```
Testing File: input1.txt
List Median ran in 23485 microseconds.
Vector Median ran in 1270 microseconds.
Heap Median ran in 1809 microseconds.
AVL Tree Median ran in 1472 microseconds.

Testing File: input2.txt
List Median ran in 379317 microseconds.
Vector Median ran in 6460 microseconds.
Heap Median ran in 7930 microseconds.
AVL Tree Median ran in 7371 microseconds.

Testing File: input3.txt
List Median ran in 10297696 microseconds.
Vector Median ran in 53164 microseconds.
Heap Median ran in 33742 microseconds.
AVL Tree Median ran in 31484 microseconds.
```

```
Testing File: input1.txt
List Median ran in 24546 microseconds.
Vector Median ran in 1130 microseconds.
Heap Median ran in 1825 microseconds.
AVL Tree Median ran in 1652 microseconds.

Testing File: input2.txt
List Median ran in 396706 microseconds.
Vector Median ran in 6258 microseconds.
Heap Median ran in 7874 microseconds.
AVL Tree Median ran in 7371 microseconds.

Testing File: input3.txt
List Median ran in 10022850 microseconds.
Vector Median ran in 53246 microseconds.
Heap Median ran in 34020 microseconds.
AVL Tree Median ran in 31971 microseconds.
```

```
Testing File: input1.txt
List Median ran in 24711 microseconds.
Vector Median ran in 1141 microseconds.
Heap Median ran in 1787 microseconds.
AVL Tree Median ran in 1481 microseconds.

Testing File: input2.txt
List Median ran in 396684 microseconds.
Vector Median ran in 6594 microseconds.
Heap Median ran in 7543 microseconds.
AVL Tree Median ran in 7195 microseconds.

Testing File: input3.txt
List Median ran in 9658433 microseconds.
Vector Median ran in 55277 microseconds.
Heap Median ran in 33517 microseconds.
AVL Tree Median ran in 34176 microseconds.
```

Average Time of the Implementations

	Input1.txt (4000)	Input2.txt (16000)	Input3.txt (64000)
List Imp	**24247.33**µs	**390902.33**µs	**9992993**µs
Vector Imp	**1180.33**µs	**6437.33**µs	**53895.67**µs
Heap Imp	**1749.33**µs	**7782.33**µs	**33759.67**µs
AVL Tree Imp	**1535**µs	**7312.33**µs	**32543.67**µs