COMP202: Exercise 2 - Binary Trees

ODYSSEAS STAVROU 2018030199
MAY 2020

Technical University of Crete - ECE

Results and Statistics:

Pros and Cons for each implementation of a Binary Tree:

Dynamically Allocated Binary Tree:

- Pros:
  - Can work with any size of elements
  - Easier for the OS to build. Nodes are allocated wherever the kernel chooses instead of a big chunk of memory needed for a matrix
  - More functionality available. The abstraction of Node and Tree Classes provide increased functionality and enables more features to be added to each class. ie: methods for printing the tree with current node as root, or removing a subtree beginning with a specific Node
  - Easier to understand its functionality and easier to follow the nodes. The tree diagram helps to understand the flow of data rather than traverse a matrix using indexes.
  - More intuitive to the mind.
  - Garbage Collector. When a Node is deleted the reference (path) to every sub-tree below it it's purged automatically by the GC.
- Cons:
  - Slower to create than the pre-allocated because of the need to allocate memory and create a new Object.
  - Slower to search. Jumping around memory is more time hungry than accessing a matrix at a specific index

Pre-Allocated (Array Based Binary Tree):

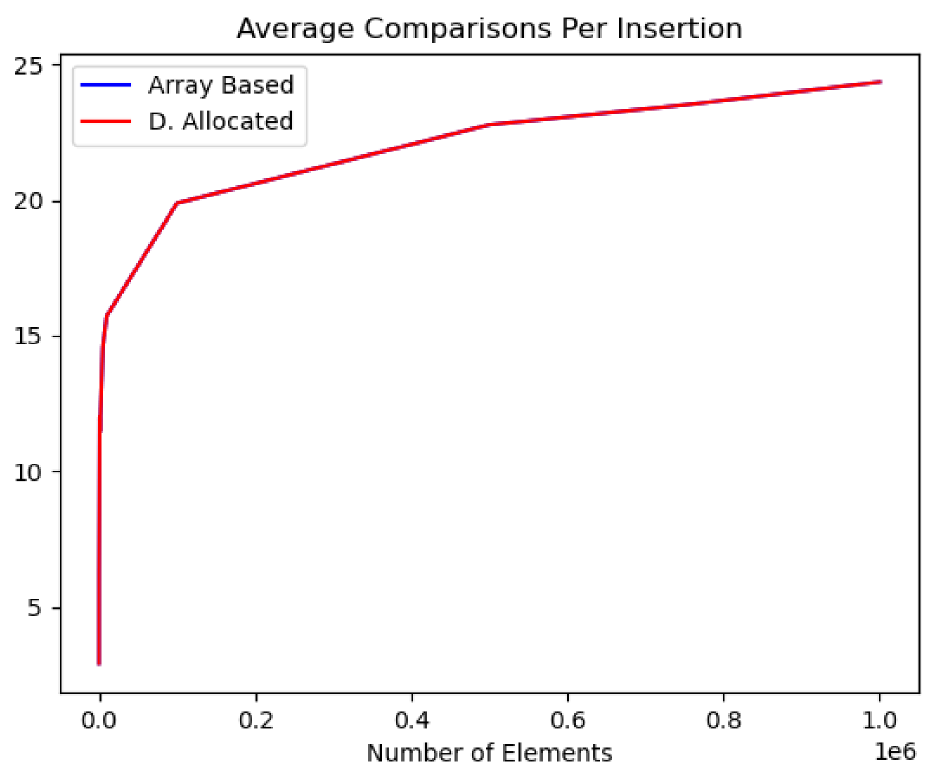
- Pros:
  - Faster to build and faster to search. Having pre-allocated all needed memory space if much faster.
  - Easier on the CPU. No need for creating a new object and assigning a value to its attributes each time we need to append something
- Cons:
  - Heavy on RAM. Having the need to pre-allocate all your data makes it more ram demanding regardless if you fill the tree or not
  - Less features than the original BST. Although the same functionality can me implemented in both types of trees much more code is needed to achieve it on the Array Based, due to it's not objective design.
  - Much more difficult to grasp its working principle than a basic tree diagram

Statistics:

We tested the performance of each tree in 6 different categories along with a sorted Array in 4 of those categories (*). The categories are as follow:

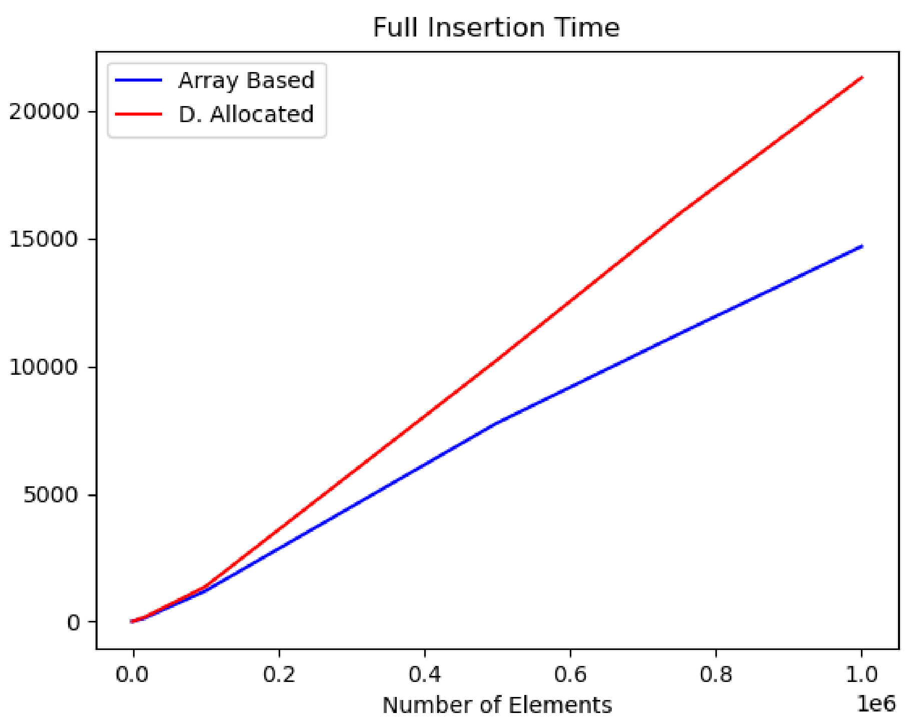
- Average Comparisons per Insertion
- Time Spent for Insertion
- Average Comparisons per Random Search (*)
- Time Spent for 100 Random Searches (*)
- Average Comparisons per Ranged Search (K=100) (*)
- Average Comparisons per Ranged Search (K=1000) (*)

The following Graphs present the performance of each of the 3 data structures.
(All time values are in ms)

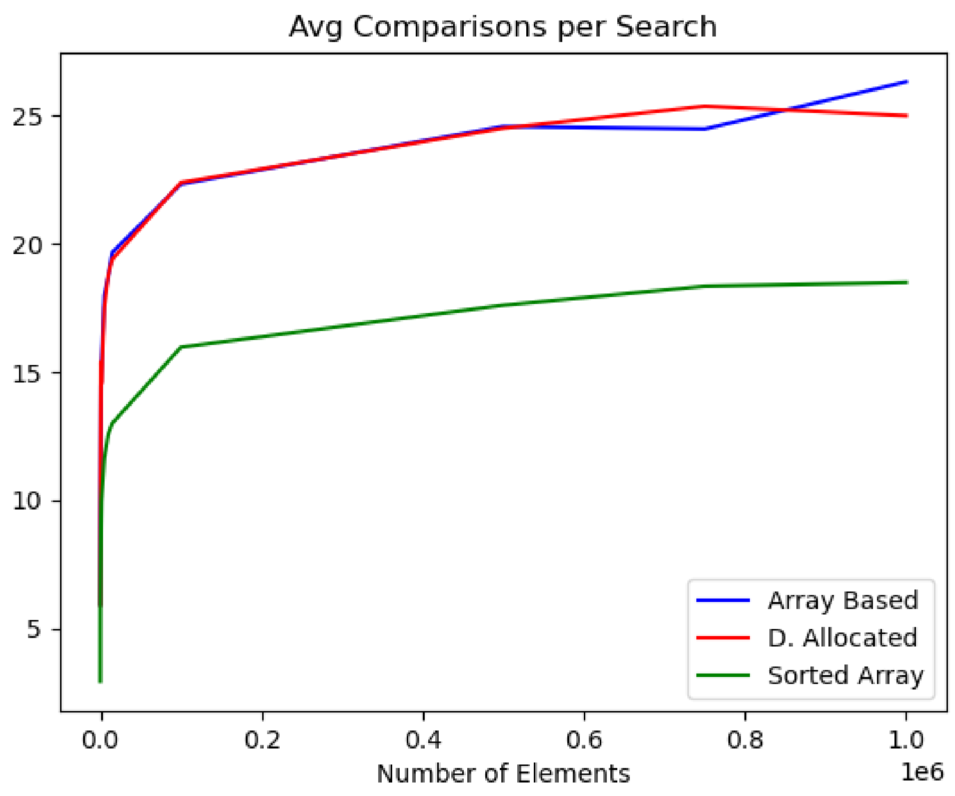


As expected, the comparisons done are exactly the same for both implementations. The order of the numbers should never change. The only thing changing is the time of insertions.

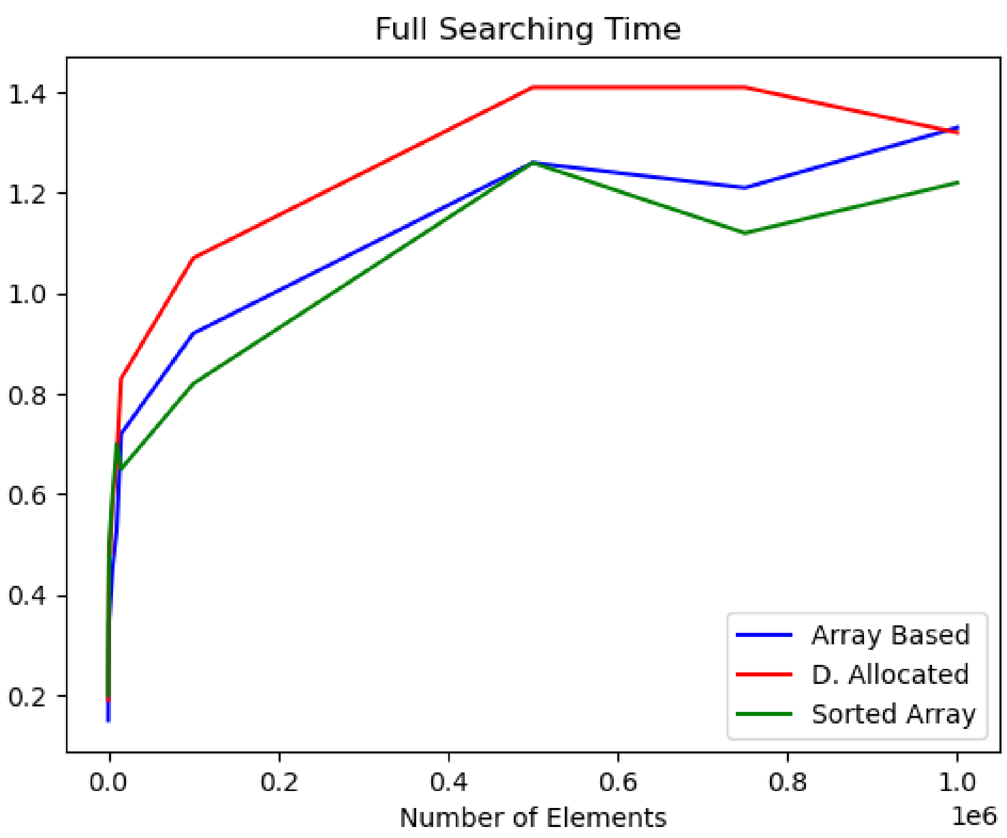
We can clearly see the type of the graph to be a logarithmic type of graph. Thus, confirming the Complexity of the Binary Tree being O (log n). The amount of comparisons needed to insert a new element to the data structure is logarithmically proportional to the number of elements and not Linearly (O (n))



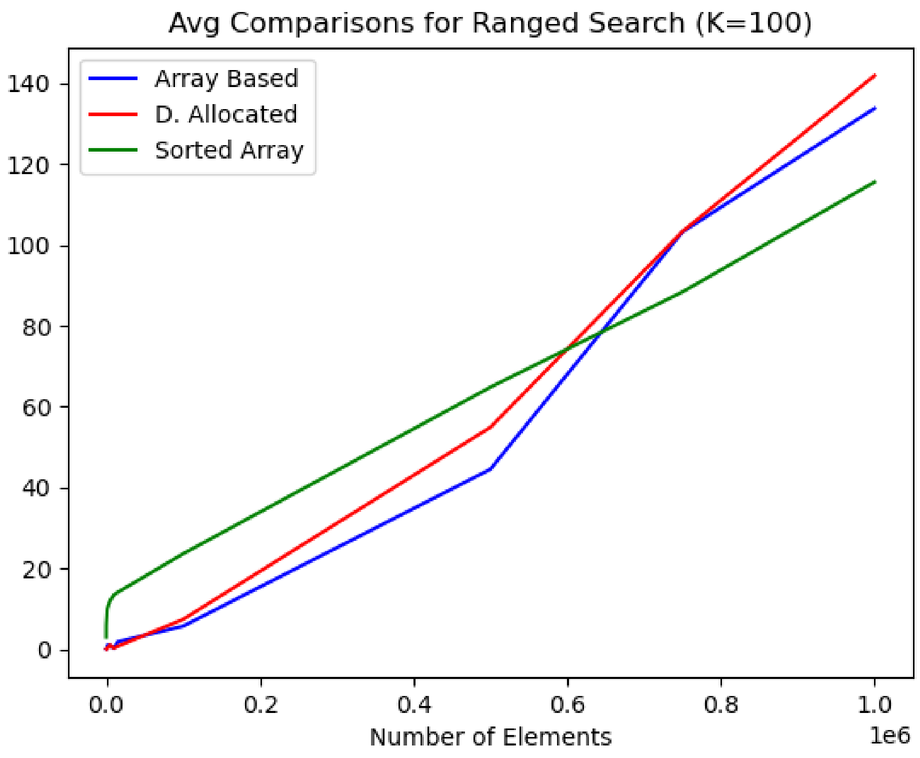
As suspected the Pre-Allocated Bst is faster for Inserting data into it, because, as mentioned above, there is no need to allocate memory for the Node and assing the values to the Object's Attributes.



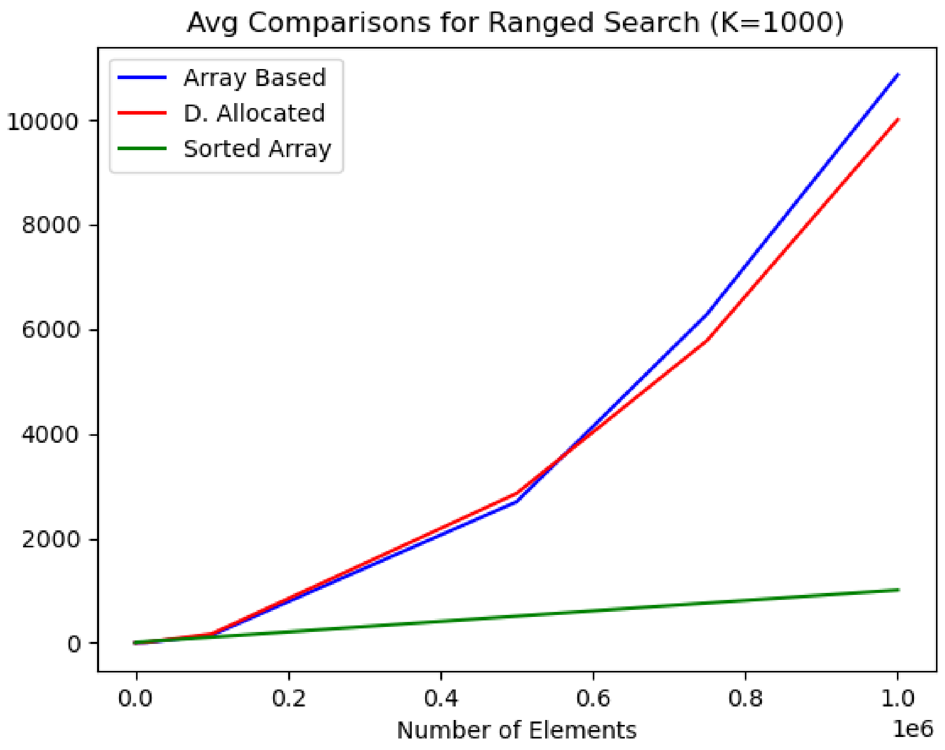
The same principle regarding complexity applies to the search function as well. Looking for a specific key inside the structure in all 3 cases is O (log n) with the difference in the Sorted array which is due to the fact that there are no branches that need checking. Having subtrees still increase the comparisons that need to be performed, however using a binary search function in a single dimension array requires scientifically less comparisons because of the ever-shrinking length of the array.



Naturally the single dimension array with the binary search finishes the quickest of them all. With the array based BST coming in 2nd due to it's flexible "moving" in the ram.



We can see that the 2 types of BSTs are almost the same (just like the random Searches). All three of them follow a linear fashion because of the small range thus, fewer comparisons are performed.



The difference between this graph and the previous one is the exponential fashion of the comparisons performed. A bigger range means that more comparisons will be performed. The sorted Array still follows a shallow linear fashion due to its single dimension design.

Note: While testing under Linux, in the 2 previous graphs the graphs of the BSTs where identical with each other. Maybe an issue in the windows version of the interpreter and the round() function. (had issues with it before).

For 10^6 elements	Avg Comparisons / Insertion	Full Insertion Time (ms)	Avg Comparisons / Search	Full Search Time (ms)	Avg Comparisons / Ranged Search (K=100)	Avg Comparisons / Ranged Search (K=1000)
Dynamically Allocated BST	24.36	22256.49	25.1	1.48	165.48	10684.18
Array Based BST	24.36	16450.91	25.73	2.16	110.21	11056.54
Sorted Array	N/A	N/A	18.36	3.31	113	1011.38