CS 300 8/6/2025

SNHU Clifton Brann

Project 1 Evaluation

Big O Analysis

Runtime analysis for each data structure:

| Vector Table Code | Line Cost | Number of times Executed | Total Cost |
| --- | --- | --- | --- |
| Create vector table structure | 1 | 1 | 1 |
| Open the file | 1 | 1 | 1 |
| Verify the file data | 1 | n | n |
| Parse the data | 1 | n | n |
| Insert the data into the table | 1 | n | n |
| Search through the entries (for loop) | 1 | n | n |
| Create new entry for file data | 1 | 1 | 1 |
| Add new line entry | 1 | 1 | 1 |
|  |  | Total Cost | 4n+4 |
|  |  | Big O = | O(n) |
|  |  |  |  |

| Hash Table Code | Line Cost | Number of times Executed | Total Cost |
| --- | --- | --- | --- |
| Create Hash Table | 1 | 1 | 1 |
| Read file Data | 1 | 1 | 1 |
| Verify data from file | 1 | n | n |
| Parse Data to Table | 1 | n | n |
| Inset data into table | 1 | n | n |
| Chain data in table | 1 | 1 | 1 |
| Search Table (for loop) | 1 | n | n |
| Create new table entry | 1 | n | n |
| Insert new entry | 1 | n | n |
| Create Hash value | 1 | n | n |
|  |  |  |  |
|  |  | Total cost = | 7n+3 |
|  |  |  |  |
|  |  | Big O = | O(n) |
|  |  |  |  |

| Binary Search Tree | Line Cost | Number of times Executed | Total Cost |
| --- | --- | --- | --- |
| Create data structure | 1 | 1 | 1 |
| Open the file | 1 | 1 | 1 |
| Verify data from file | 5 | 1 | 5 |
| Parse data to tree | 1 | n | n |
| Insert data in to tree | 1 | n | n |
| Search tree (for loop) | 6 | n | 6n |
| Create node for new data | 1 | 1 | 1 |
| Travers tree R or L | 4 | n | 4n |
| Move down 1 level | 1 | n | n |
| Locate node R or L | 3 | n | n |
|  |  |  |  |
|  |  | Total run cost = | 14n + 8 |
|  |  |  |  |
|  |  | Big O = | O(n) |
|  |  |  |  |

Performance Evaluation

After calculating the Runtime values for each data structure, I was surprised that that each structure shared the same Big O value of O(n) for the worst case of runtime. The commonality of this value shows that the complexity of the data structure will increase proportionally with the amount of data added into the structure. From reading about how to estimate the runtime for the different algorithms, it seems that the average Big O values will be less than O(n) for the Hash Table and probably close to O(log n) for the Binary Search Tree. These estimates are partially based on ideal conditions and a decent amount of experience in designing the structure and algorithm to handle the data most efficiently.

In reality, I think the Big O value for a Linked List will always be O(n), in the worst case, because to add a new entry, the program must traverse each entry of the list to find the last entry and add a new entry there. Or the algorithm has to traverse each entry until it finds where the new entry must be inserted and then has to adjust each subsequent entry to make room for the new entry. So with that in mind the Big O for a Linked List Structure can vary wildly from O(1) to O(n). For a Hash Table the Big O could be as low as O(log n) or as high as O(n) depending on the efficiency of the hash algorithm. Since the Hash table uses a derived value instead of the data value, the search runtime should be about O(log n) since the algorithm is checking a specific hash value instead of checking each value in a linear fashion. The same should be true of a Binary Search Tree, the Big O should be closer to O(log n) on average instead of O(n) which is the worst case value, if the search and insert algorithm are efficient and the tree is well balanced. The runtime savings are realized because, if all goes well, the search or insert algorithms only have to search half the nodes in the tree to find the desired entry or position of the new node.

Similarly the Big O values for the Space Complexity for two structures comes out to O(n). The reason for this is that as the number of entires increase, the amount of memory required to store the data also increases in a linear fashion in both the Hash Table and the Binary Search tree. The Linked List is a different story, as the size of the list must be determined when the code is written. And given the limits inherent in the programming language the list can only be so large, granted it is a large number, but it must be predetermined when the Linked List is created.

The relatively simple and straightforward forward nature of the Linked List has the advantage of being easy to code and maintain. However, as the size of the list grows the more time will be consumed by the program as it traverses the list from the first entry all the way to the end of the list to attach a new entry and pointer is its biggest disadvantage. The Hash Table has the advantage of not having to examine the table contents in a linear manner, but rather can examine one particular ‘bucket’ based on the hash value created by the algorithm. A well designed hashing algorithm will distribute the data evenly across the data structure, minimizing the number collisions or identical hash values that put multiple entries in each ‘bucket’. These collisions can be addressed by chaining the data, so that multiple data entries can exist in one ‘bucket”. The fewer items chained together in a particular ‘bucket’ means that fewer comparison will need to be made in the hunt for the particular data item or entry. A drawback of a Hash Table is a poor hashing algorithm will leave the data entries “clumped together” or concentrated in a few places instead of being every distributed. This concentration of data will result the search function needing to be run multiple times per ‘bucket’ to check the chained data entries under the hash value, and return the searched for value or a message that the search term was not found. The Binary Search Tree has the advantage of being the most efficient structure to search, ideally only half the nodes in the tree need to be compared to the search term. The most significant disadvantage to a Binary Search Tree is the complexity of the code required to make it work properly. If the root of the binary search tree is chosen well, and the algorithm is designed properly, then the tree will stay balanced. A balanced tree means that each side of the tree from the root will have roughly equal amounts of data entries. For a Binary Search Tree to work properly it must be constructed to iterate through each node and make decisions based on contents of the node relative to the search term or data to be inserted into the tree. During a search, the search term is compared to each the contents of each node starting at the root node. The search will then proceed downward and to the right or left depending on the relationship of the sear term to the node contents, if the search term is less than the node contents the sera will proceed to the left, if it is greater than it will move to the right. And this process will be repeated until the search term is found and returned or it is not found and an error message is returned to inform the user the search term was not found. And as data is added to the tree structure again the insert function starts at the root node and proceeds down the tree just like the search function, but will stop and create a new node when the insert function finds a place that is unoccupied and fits the data parameters. And due to this level complexity of logic and the need to be able to handle any kind of input error or problem there is a significant amount of error handing code that may or may not be utilized, but must be written and maintained just in case. .

Data Structure Recommendation

My recommendation for the use of a data structure for ABCU for a course catalog would be to use a Binary Search Tree structure. I think the advantages outweigh the drawbacks. With proper planning and design the algorithm used to construct the tree will keep it balanced, and efficient. The dynamic sizing of the data structure allows for a good level of efficient in memory usage, as it will only use the amount of resources needed to store the amount of data present, and will not have empty entries taking up memory while waiting to be filled at some point down the road. With a runtime Big O that should be closer to O(log n) than to O(n) on average, the search speed and efficiency are higher than the other structures that were candidates. Add to that the the built in ability to easily add and remove entires as needed to update the course catalog each year or semester, the Binary Search Tree is the most fitting data structure for this application.