Garret Gherardini

CS-300

Module 6 – Project 1

Professor Ostrowski

Data Structure Analysis

Throughout our course, we have examined aspects of three primary data structures: vectors, hash tables, and binary trees. Each of these data structures are responsible for efficiently storing and organizing data of varying sizes. Throughout this evaluation, we will look at the advantages and disadvantages of each data structure from our course, utilize Big O analysis to determine the levels of efficiency, and provide a recommendation for the data structure type that is most suitable for our client, ABCU and their advising program.

First, we will examine the vector data structure. A vector, as defined by our textbook, is an ordered list of items of a given data type. Vectors are an easy way to store data when the allotted data size is not defined and may be subject to change. There is no fixed size for vectors so as data elements are added, the size of vector can be easily expanded. Data can simply be added or removed from a vector if the data type being modified is the last value in the vector. To modify data in the middle of a vector it can be accessed via an index, however the methods for removing or inserting data into the middle of a vector can be cumbersome. As an element is removed, for example, all other elements must fill the space from which a data has been taken away. This leads to inefficiencies when inserting or removing values. The nature of a vector is sequential, so any data values that may be easily inserted or removed only at the end of a grouping of elements. Key takeaways from vector data structures: it is an advantage to use them if the number of data elements are variable and do not quickly need to be sorted, it is a disadvantage to us a vector if accessing middle elements needs to be efficient. Vectors can be time-consuming when searching for data elements.

Next, we will look at the hash tables data structure. Hash tables are utilized by taking unsorted data and organizing them by assigning each data element to a specific location in an array. Unlike vectors, hash tables enable a higher degree of efficiency when inserting and removing data as these modifications do not require all other data elements to be shifted. To access these data elements, functions can be implemented to map keys to an array’s index making for a much more effective and faster method of accessing specific elements. The main problem that can arise with using hash tables is running the risk of having too many collisions. A collision occurs when there are multiple keys mapping to the same index, as a result of high collision volumes, the efficiency of data retrieval is diminished. Another disadvantage of this data structure is its limited capacity; regardless of the volume of data being stored, a hash table can only output based upon its designated size. In turn, this further complicates the main disadvantage mentioned previously of having high collision risk. Key takeaways from the hash table data structures: it is an advantage to use them if it is required that a system is able to pull data elements quickly and efficiently or needing to insert or omit data elements with ease. Hash tables run the risk of losing their efficiency advantage over vectors as increased collisions may occur as a set of input data increases in volume.

Finally, we will review binary trees. Binary tree data structures sort data by organizing elements by means of a node; each node in the binary tree has no more than two children and the structure sorts data elements by traversing from the root node and sorting at each incrementation until a value can be appropriately assigned to the left or right child. The binary tree data structure has many advantages, ease of searching for data elements being one of them. Each node within the binary tree will have a left subtree with a data element of a lesser value, therefore searching for a specific data element will result in a fairly fast runtime. Since the data is ordered based on its value size stemming from the root node, conducting searches is halved at every point in the traversal. These searches are conducted in one of three orders: in-order, pre-order, or post-order. Similar to hash tables, binary trees are limited in their capacity. With each node only being able to store data in two nodes, the left and right child, the type of data structure may need to be reconsidered based upon the data being inputted. As the size of a binary tree increases, the efficiency at which it can perform data retrieval or input/remove data is drastically impacted. Too many data values may cause some nodes to have only a single child, increasing the length of time to obtain the data. Key takeaways from the binary tree data structure: searching for elements within this structure can be efficient as long as the tree is balanced and not overloaded with data values. Managing data within the binary tree is generally efficient as long as capacity is not exceeded, and each node has a left and right child.

Assuming the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function: the Big O value for each to the data structures for reading the file and loading data amounts to O(n) except for binary trees which is O(log n), in which case n is dependent upon how many elements need to be read. To search for data, a vector and hash table’s Big O value is O(n) and the binary tree is at O(long n) – O(n). For sorting and printing the vector will run at O(n log n) while hash tables and binary trees will run at O(n). Assuming a binary tree is balanced, it might be the most ideal to use as its efficiency could be more consistent. However, a hash table could be another good option assuming it has the capacity to handle a large quantity of data since it does have limits on the amount of data that can be stored. You also run into the risk of collisions with hash tables that may impact the efficiency, so for this instance I think the use of a binary tree would be the best option.