Jordan Davis

Professor Shaykian

CS-260-T2616

December 12, 2021

Project Three: Data Structures and Algorithms

In module 2 of our Zybooks textbook, we were given a refresher on vectors and how they held data. A vector is like a list, but this type of list has another list of variables attached to each element in it. You can have a list of students and have that student’s personal and academic information attached to that student’s element in the vector. In the attached “VectorSorting.cpp” file attached to this submission, which was submitted for week 4, we can see a vector in action. The implementation of relevant actions for the vector are created in main through separate functions with no other class involved. In the code, there is a loadBids function that takes advantage of the table format that the information is provided in and uses that to add the correct column to the right variable for that information. Each bit of information pertaining to the bid is easy to access and iterate through inside the vector due to the use of a Bid structure inside each vector element.

In the fifth week of this course, the class looked at hash tables and chaining. Hash tables map elements using a hash key to an array or vector. Because of this use of locating by hash key, the time for inserting, removing, and searching can be reduced. The attached file of “HashTable.cpp” demonstrates how to construct a class for a hash table that can insert, remove, and search through the table. Rather than searching by the bidId like other methods, we look through the hash table by the key associated with the elements. Since there is a class for our hash table, we have the structure separate and have both public and private functions and variables set. We have a Node struct private so that other classes we create that might need nodes do not access this struct for the hash table. Our key variable is also set private so that no one can go in and change the key. This would disrupt the whole table if the key was altered midway through the table.

In module 6, we learned about tree structures. This tree structure makes it to where each element has two successors rather than just one. This more compact form of a list makes it to where comparisons can be made, and the tree is halved in its search of an element at each level. If the tree was structured in a way where each level is full, it could take up to 16 comparisons to find the correct element. This structure is also implemented through a class in the “BinarySearchTree.cpp” file. Like the HashTable class in the “HashTable.cpp” file, there are private functions and variables. While the Node struct itself is not private, the node representing the root is. The functions that implement the actions of adding, removing, and displaying the elements of the tree are private while there are public functions that call these private functions. The user has no need to see or access these because they are set and do not need to be changed.

Regarding the search algorithm introduced in module 3 of our textbook with respect to linked lists, this algorithm is not optimal for our scenario. Let’s look at the code completed in the file of “LinkedList.cpp”. Our search algorithm starts at the beginning of the list and moves down the list until a match is found. If the bid that we are looking for is towards the back of the linked list, our search would go through over 40,000 bids until it found the bid that we need. This would eat up a lot of time and is not efficient.

In the case of the hash table and binary search tree structures, the elements are sorted as they are entered into the structure by the hash key and node key respectively. This is proven in the files “HashTable.cpp” and “BinarySearchTree.cpp” files. We do not use any function to sort the structures, only the loadBids and Insert function in “HashTable.cpp” and Insert function in “BinarySearchTree.cpp”, and yet, when we display all the bids, they are in sorted order by their keys. In the case of sorting the vector in the “VectorSorting.cpp” file, we must manually sort the vector after adding the bids into the vector in the order they are presented on the given file. In “VectorSorting.cpp”, we implement the partition, selection sort, and quicksort algorithms.

Hash and chaining have a lot of room for errors and inefficiencies. Hash is only a good method if a proper hash key is instituted and may take some trial and error to implement it in an efficient way. Inserting a list of 41,000 bids into a hash table could potentially have a lot of collisions that would lead to having long chains. Depending on how many possible collisions that accrue in a single location, chaining could be a slower process than other search methods because it still might have to look through hundreds of elements. In the case of the file “HashTable.cpp” using this type of algorithm for the small list of bids, this was a good use of the hash table and made searching way faster than the search algorithm implemented in “LinkedList.cpp”. In the case of searching the file “eBid\_Monthly\_Sales\_Dec\_2016.csv”, there was at most 3 elements at a specific bucket, so a search only had to compare 3 items at the most. In the case for the larger file “eBid\_Mobthly\_Sales.csv,” when the tableSize variable is changed to the number of bids in the file, there was at most only 2 items at a specific bucket.

My favorite program to create and implement was the program that ran “HashTable.cpp”. In terms of searching for the default bidId in the larger “eBid\_Monthly\_Sales.csv", it was by far the fastest, taking 0.000041 seconds to find the right entry as compared to 0.001581 seconds with “LinkedList.cpp” and 0.00021 seconds with “BinarySearchTree.cpp”. This is because, as stated previously, it only must compare at most two entries in the hash table since we can implement the table size with the exact number of bids since we have knowledge of the number. It was also on par with “LinkedList.cpp” and “VectorSorting.cpp” in terms of loading the bids.

This program also creates a class with many functions that are versatile. If an excel has a different order for their information, you need to only change 4 numbers in the loadBids function. If you need to use a different file, then just change the file in the command line. If the output format needs to be changed, then go to displayBid and change the cout statement. A vast majority of the program is modularized inside the class and its functions to where changes can easily be made. A majority of the HashTable class could be reusable at its core but the bid structs and its specific variables would need to be adjusted to a new program. All the aspects regarding the Node structures involved could be reused. Each function is also properly annotated before it’s beginning with a clear explanation of its purpose, parameters and what it returns. Amongst the blocks of code, it is clear what that block is meant to be doing. For example, explaining in the Insert function what block of code represents when the node is found and what to do if it is not.

Data structures are important in programming because programs are always storing a lot of information and we need some way to store this data in an efficient way that also allows us to easily access the data or a variable of that data. Vectors, lists, hashes, and trees are structures that help us to accomplish these tasks. Without these structures, it would be very difficult to access and iterate through data, especially is we need to access hundreds or thousands of elements. For example, in our “VectorSorting.cpp” file which utilizes a vector structure, we can easily add each aspect of the bid information to a Bid struct and then put that Bid struct into the vector. Because of this we can easily access specific information about each bid by iterating through the vector with a simple For loop.

Algorithms are procedures that help to solve a problem, like the selectionSort function in “VectorSorting.cpp” or the Search functions in “HashTable.cpp” and “BinarySearchTree.cpp”. Algorithms help us to solve a problem, but only if we know how to use them effectively. For example, let us consider the Search function is “BinarySearchTree.cpp”. If we did not implement the algorithm properly to look for bids smaller and larger to navigate through the nodes since that is how the tree is structured, then it wouldn’t properly iterate through the tree and find the right bid.

In conclusion, data structures and algorithms are what many programs are built upon. You need structures in place to hold your data and you need the proper algorithms to do the needed tasks on that data. Without these concepts in place in your program, you would have thousands of lines of code trying to look through unsorted data that is hard to keep up with and hard to access and change. Structures like hashes can store large numbers like the way a stem plot would hold the values. Algorithms like quickSort and selectionSort, as seen in “VectorSorting.cpp,” can be used to keep a list of names in alphabetical order. Algorithms and data structures are foundational concepts that can be applies to any program.