Final Project

Portfolio Reflection

CS-260

Caroline Wilson

February 2020

**Data Structures**

Throughout the progression of this course, I was tasked with the augmentation of code in order to create a program that allows for the collection, sorting, and outputting of bid information in an auction setting. In order to do this, it was important to start with the groundwork that holds the data that was to be stored. This was done through the utilization of data structures. As with most things, there are a number of ways to be able to compartmentalize the data in the program. This class gave me the tools to do this in three different ways: vectors, hash tables and tree structures.

As shown in the attached file named “VectorSorting.cpp”, I utilized the vector method. As with linear algebra, you can think of vectors as a stacked list of inputs. This seems to me to be the most basic method of compilation. By creating compartments, or vectors, to store the information, known as elements, it gives the layout to be able to sort through using one of the algorithms that will be covered in the section titled “Algorithms”. The construction of a vector can be realized in the .cpp file where vector is created entitled “bid” referencing “Bid” and when the bids are loaded at “vector<Bid> loadBids”.

A hash table is another construct that can be used to hold data within a program for later reference. In the attached file, “HashTable.cpp”, the initialization of the structure can be seen with the lines of code that define the structure that will hold the data. This is initiated by the command “struct” which opens brackets to allow for strings and doubles to be held. The convenience of hash tables is that it allows for faster and more effective searching because it uses an index to search as opposed to searching through every element of the table.

The final structure is that of the tree structure. The tree structure is the most complex in its construction and editing. The basis of the binary tree structure starts with a root and then expands downward depending on the elements that follow. From the root, additional nodes are created that can have nodes that have additional children with the end of the line being called the leaf and the internal nodes have at least one child. There is a lot of logic that goes into the creation, using temporary nodes as place holders. These methods are demonstrated in the attached file, BinarySearchTree.cpp. There are multiple entries that represent the construction and logic there in. The initial insertion logic for a node can be seen at “void BinarySearchTree::Insert”. The initial remove structure can be seen at “void BinarySearchTree::Remove”. The most complicated logic is that of the full logic in adding and removing nodes. These functions can be seen at “Node\* BinarySearchTree::removeNode” and “void BinarySearchTree::addNode”. In these sections of code, the pointers are established and the order in which the tree must stand before and after the manipulation of the nodes in the tree sequence.

**Algorithms**

Once the structures are created, the functions to use them must be set in motion. This is done with a sequence of algorithms that dictate how the information is accessed and used. The program can dictate how and where to search for a particular element, it can sort the elements based on specific factors or it can compartmentalize the elements so that they can be pulled based on factors or stored so that there is no overlap in the space in which they are stored, known as chaining.

One of the most common algorithms you may think of is that of searching. With the search algorithms, you are able to go through one of the previously mentioned structures to find a specific element, or elements. There are multiple ways to do this, some resulting in different search times and levels of complexity, based on the structure and the language used to access information. A prime example of a search algorithm can be found in the attached file named, “BinarySearchTree.cpp”. With in this you can find a search algorithm that b is initiated with “Bid BinarySearchTree::Search”. This algorithm walks through a number of steps comparing an input, notated as current bid, and searches down the binary search tree, BST, looking for a specific node. With the BST the size of the bid id is compared to the root, then to its children nodes and continues down the tree until the bid that is being searched for is found. The way that the algorithm is structured, it checks to see if the node matches the input, then checks to see the comparison is larger or smaller than the element previously checked and directs the next pointer to either the right or left depending on the comparison. Once the bid id is matched to a node, the bid returned.

An example of sorting can be seen in the attached file, “VectorSorting.cpp”. This algorithm compares the bid’s title in intervals to sort them in order. This can be seen when the algorithm is initiated by “int partition(vector<Bid>&bids…”. It compares the title to an initiated pivot point, then determines if it is higher or lower. It then uses a swap function that moves the bids around until they are completely sorted, incrementing the high and low standing bids as it goes. After this is completed, a return of the bids will show the sorted bis list.

The final algorithm that is represented in my work from this term is that of chaining. This can be realized in the attached file, “HashTable.cpp”. Chaining is a function that allows for elements to be stored in what is referred to as “buckets”. This allows for multiple items to be stored in the same bucket based on a similarity, or specific key. Generically, the mathematical function known as a modulo operator, denoted as %. What this does is divide the key by the given amount, for example % 10 would divide the number my ten, then uses the remainder as the key that directs the element to the matching bucket. In the case of the attached .cpp file, chaining is demonstrated in the ability to add a node into its proper place using an unsigned key, in this case the bid id. This is at the point in the code that starts, “void HashTable::Insert(Bid bid)”. This structure creates an integer from the inputted string, looks for a node in which to place it, and if nothing corresponds then a new node is created. If a node is already in place, then the algorithm will search until an empty node is found.

**Student’s Choice: Binary Search Trees**

Over the course of the term, I was happy to see how my codes came together. In what could be a very convoluted process very often was made more streamline by the understanding of the processes I was using and the implementation of a pseudocode to direct me. While the creation of most of the structures is pretty straight forward, it can be very confusing to be able to take the structured data and put it to use. It is important in the writing of these codes that they would be modular; broken down into smaller components and less lines of code for each function. It is also important that they would be able to be reused should the opportunity present itself. With this, small elements of the code could be adapted to fit a number of uses. Equally as important is the annotations within the code. This helps not only myself stay on track and remember what I did, but it also give other programmers looking at my code to understand my thought process and see the reasoning behind the lines that I implemented. While most of my code fits into these parameters, the code in which I am most proud is that of the Binary Search Tree. This code required a lot of logic in its creation, and admittedly at times had me a bit stumped. I was happy to say that, after a long arduous process, I was able to submit a fully functional code that exhibited each of these three aforementioned factors.

With binary search trees, it is important that there is a very firm implementation of logic in place. With this, even more than other codes, I found it very helpful to create a pseudocode as road map to where I was trying to go. Without it, I can see how I could have easily gotten lost in the logic. The end result functionality of this program is what made me the proudest. Understanding how elements would be inserted, removed and searched for is paramount in the creation of the code. Without an understanding of loops, this code would not have been possible. I also had to use the .compare function quite a bit. While the creation of the tree in of itself was difficult in allowing it to sort as it was established, the algorithms that allowed for items to be removed was the most difficult. It would be so easy to just remove a node and call it a day, but in order for the BST to maintain its integrity, the remaining items needed to be sorted back into place. With the structure of the tree, the innate complexity of the structure, in place it made way for the ease of the search function. This is what makes this kind of code worthwhile, it makes the search process much easier in the fact that less elements need to be “touched” in order to return the desired result.

By nature, this kind of code allows for modularity. With its individual pieces that need to be picked apart, the modularity in the code presented itself. I would have been at a loss trying to construct a BST all in one shot. Piecing together each part of the code is what allowed it to function to best capacity while taking away some of the confusion of this kind of structure. By looking at this code as one big picture, it is a big pill to swallow. However, by breaking it down into smaller pieces it made it much easier to manage.

I can also see how easily this code could be reused for a number of practices. Granted, it would take more work than just renaming a single variable, with the proper data set provided and the proper variables established early on in the code, it would be very easy to implement this for a number of uses. Additionally, I learned along the way of a function within C++ programming that would allow for the renaming of a variable throughout the code without having to pick through every line of code to ensure that everything is changed. By using the refactor function, renaming one instance of a variable will change every corresponding carriable and pointer throughout the code. This, unfortunately, ended up being the demise of one of my source codes as the system froze and booted me out, but I was able to recover easily thanks to my pseudocode and can see how, when functioning properly, this could be paramount in the implementation of reusable code.

One thing that I learned early on in my discovery of code writing is that proper annotations can be the difference between a fluid or a convoluted code. This functions not only as reminders to myself as I am writing and reviewing my progress, but it also gives an onlooker a firmer understanding of my thought process and the reasoning behind the lines that I have written. Each step that I make in the creation of code are notated in this way. This in conjunction with my pseudocodes allow for me to keep track of where I am, what needs to be done, and what has been done along the way. Especially in the creation of the Binary Search Tree code, it helped me stay on track with my thought process as the many moving parts can get quite confusing. This can be seen throughout all of my code submissions, denoted by the standard “//” and signed at the end with my initials and the date that the changes were made.

**Conclusion**

Throughout this course I have learned not only how to implement structures and algorithms, but I have learned their functionality and overall their importance. Through this process, I have had to hone in on my problem-solving skills and the tools that I have been given to demystify the process in which these codes are written. It takes more than just an understanding of semantics, verbiage and syntax. It is important to understand not only the how and why, but also the best practices and opportunities for growth as a programmer.

Data structures, in this case, are the skeleton upon which the entirety of the programs rest. They give a vessel in which data can be stored and from which they can be accessed when implementing anything from complex algorithms to simple functions. With the different means by which data can be stored, the types of data structure used can aid in flexibility in functionality and reusability, simplicity or complexity, and overall readability of the code. From basic storage for smaller less complex data sets, such as vectors, to more versatile compartmentalization such as hash chains, to higher level of complexity for much larger data sets and their access, such as binary search trees, there is a time and place for each structure. It is important to understand which would be best based on the data set, and what you plan to do with the data there in. Just as with any trade, it is always best to use the right tool for the right job. This class gave me the ability to understand just how to do that when working with data sets.

Once the structure is built, or in some complex situations such as BST when they are used in the actual construct, it is important to understand what is going to be done with the data stored and how the program should react to user input commands. Afterall, what would be the point of having a seamlessly compiled data structure if there would be no way to access or manipulate the data? Throughout this course I learned how to access the data that I had stored, how to sort it into set parameters, and how to tell the program to react when changes are made to the data set. Algorithms are the directive that is set in place to allow these functions to happen. While the overall construction can get convoluted while looking at the big picture, when broken down into smaller solvable problems, simple reusable lines of code can make the overall functionality of the program more streamlined. By going into programming with the intent of the use of modular code, it is easy to see which algorithms will present the desired product and which will lead the program down the wrong path. Because of the flexibility of modular coding, if a portion of code presents errors, it is easy to pinpoint the issue, debug and try a different approach which may, ultimately, produce a more effective and efficient program.

It seems that there have been a few main things that have rung true throughout my programming experience on this degree path. In this course, one saved me a lot of headaches, while the ignorance toward another could have caused that headache to begin with. It had proven itself to be true, yet again, that a well written pseudocode can mean the difference between a well written code and a poorly written one. In creating a road map for the codes’ trajectory, it is easy to see where I need to be and how to get there. This term my pseudocode proved itself invaluable when, as I nodded towards the stupidity of this, I forgot to save and compile as I went along. What could have resulted in a total loss, I was able to reconstruct rather easily because of the map that I had made for myself. Also, having the pseudocode in front of me before actualization of the program made it easier to implement my problem-solving skills. This, to me, is the difference between evaluating an algebraic equation on scratch paper versus trying to do it in my head. Utilizing pseudocode in conjunction with modular programming gave me the ability to visualize the start point, the path, and the end point of the problem-solving experience, ultimately aiding in my success in writing code this term. This practice is something that I will carry with me throughout my career in cyber security. Though this lesson had seemingly been learned over my past experiences, this term allowed its importance to be driven in even further. With this tool, I undoubtedly will have more success and less hurdles in my journey to become a successful computer scientist.