7-1 Portfolio of Work and Reflection

Christian Kesler

On behalf of SNHU Software, I have compiled this report summarizing different programs and the data structure/algorithm pairings they implement that exemplify the varying solutions we have created for you at eBid Nashville. The goal of this report to is properly explain the design and function of each programming solution provided, along with an evaluation of their strengths and weaknesses so you might make an informed decision regarding future implementation. The programs we created were designed to work with a sample data set for the purposes of illustration, and are not ready to implement on the website itself currently. While small changes will be needed in order for the programs we've developed to communicate effectively with your website, the data structures and algorithms presented are complete and reusable in their modular design. Development on integration will begin after decisions about the approach moving forward have been made, which is the purpose of this report.

Data Structures

Before we can discuss the solutions put in place, we need to understand the data we are working with, as well as different ways we can manipulate it to our benefit. We were given a sample CSV file in the form of a spreadsheet that follows the model in place on data.nashville.gov. It contained rows of bids with varying information in each column, such as the bid's ID or the value of the winning bid. While the spreadsheet is a tried and true format for readability and manual manipulation, in order to effectively perform functions on the data set, we needed to rearrange the bids for efficiency's sake. The act of rearrangement puts the bids into what we call data structures, a defined and organized method of storing data with specific manipulations in mind. There are several types of data structures we implemented for your review, the first of which being a Vector.

1. Vectors

A Vector is akin to a list of entries, which behaves much like a spreadsheet when manipulated. It stores each value or bid in a sequential memory slot on the list, much like bids take rows on a spreadsheet. We can insert and remove bids anywhere in the list, and the Vector is regarded as one of the most versatile built-in data structures in the C++ programming language. An example of its simplicity is apparent in VectorSorting.cpp at lines 83-112. All we need to do to add a bid to the Vector called "bids" is one command called push_back, passing the bid as the argument. While the Vector is quite easy to implement and understand, it does have its shortcomings, which is where the Linked List comes into the picture.

2. Linked Lists

As I mentioned before, bids can be inserted anywhere into a Vector much like inserting a row in a spreadsheet. However, when we are using a spreadsheet, we usually insert all of our values at once, creating twenty new rows and pasting twenty new entries in a matter of two commands. With a Vector, each bid is inserted one by one, and each bid following the inserted bid has to shift by one memory slot to make room. When inserting a large number of bids, the efficiency of the program slows dramatically since it has to constantly rearrange every bid in the list located after the bid being inserted with each insert. This can lead to thousands of operations even with a data set that doesn't exceed a few hundred in total. A Linked List is designed to counteract this specific weakness. With a Linked List, the bids are not stored in sequential memory spaces, but instead each item in the list holds a bid and the address of the next bid in the list, which we call a pointer. At a glance this difference seems negligible, but when inserting a large number of items, the number of operations that the system has to perform decreases exponentially. An example of the moderately simple design is apparent in LinkedList.cpp at line 97-107. Adding a bid to the Linked List only requires making sure the list is not empty and setting the address or pointer of the last item to the new bid. The Linked List is mildly more complicated to implement, but the efficiency of data set manipulation will go a long way in regards to the performance of your website.

3. Hash Tables

A Linked List offers more efficient manipulation of the data set, but manipulation isn't the only operation that you requested we design for this data set. We also need to be able to search and return a specific bid based on entered criteria. With a Linked List or Vector, we can simply read each bid in the set one by one until we find the bid or bids that match the criteria. However, if we have a large set of bids, say ten thousand, and the bid we want is at the very end, the system will have to compare our criteria to all ten thousand bids before it finds the correct one. This issue resulted in our inclusion of a data structure called a Hash Table. A Hash Table can be envisioned much like a list with a fixed number of entries. Rather than loading the bids into the Hash Table in whatever order they are presented, we perform a mathematical operation called modulo on a unique key value like bid ID in order to determine which line in the Hash Table the bid belongs to. This can be seen in HashTable.cpp at lines 129-156. The key is calculated based on the value of the bid ID, and the bid is used to either initialize a new list at that location if empty, or add to the end of the list if it exists already. Each line in the Hash Table can have multiple entries, and as such, each line is stored in the Hash Table as a Vector or Linked List. This might seem a bit multidimensional in that we took a spreadsheet, turned it into a list, and then made a grid out of those lists, but I assure you that there are many benefits to this data structure that will be covered in the “algorithms” section of this report.

4. Binary Search Trees

The last function that we needed to consider was sorting the data set, which the Hash Table doesn’t exactly do any better than a Linked List or Vector. With those data structures, we have to read every value in the set and find the smallest one, then repeat using the previous value as the benchmark. Effectively, the number of operations required if the size of the data set is N is equal to N!, which is remarkably inefficient as the size increases. The last data structure we decided to demonstrate was the Binary Search Tree, or BST. A BST is similar to a Linked List in that we store the value and an address to the next value in each memory slot, but differs in that there are two memory addresses listed; one smaller than the actual value being stored, and one larger. If we were to visualize a BST that holds the numbers from 1 to 16, ideally the top or root of the tree would be the value 8. Branching down to the left would be the value 4, and to the right would be 12. From the value 4 would be 2 and 6, and from the value 2 would be 1 and 3, etc. It may seem confusing, but the fact that the BST uses consistent logic in that the right address stores a greater value and the left address stores a lesser value gives us a great deal of computational power. The applications may not be obvious, but if we want to find the smallest value in the set, we simply start at the root or top of the tree and keep to the left until there is nowhere else to go. Since lesser values are stored to the left, we would arrive at the smallest value in the set rather quickly. This is the intent behind the BST, and more on the BST's strength when sorting will be covered in the "algorithms" section of this report.

The specifics of the logic are somewhat difficult to grasp without visual aid, but in essence a BST allows a data set to be sorted without any comparisons being made during the operation since the sorting logic is implemented when each bid is stored in the first place. You might be wondering why couldn't we do that with a Linked List in the first place, since sorting as we add data sounds easy enough. There are two advantages a BST has over a Linked List in this regard. Firstly, a BST is more efficient at sorting when adding an item to a data set than a Linked List would be since there are on average fewer comparisons needed to find the correct insert location. This can be seen in BinarySearchTree.cpp at lines 155-174. When adding a node to the tree, we make two comparisons before either adding the node as a child or recursively call the same method until we have traversed down the tree to the point where there are spots open for children to be added. Secondly, a BST is designed to minimize the number of comparisons needed when performing a search, which is a notable weakness of the Linked List.

With proper exception handling and height restrictions, a BST that has 64 items will only require a maximum of 6 comparisons in order to find any one value in the data set. It may not be quite as efficient at searching as a Hash Table, but given that the BST can sort the data set almost without effort once the data structure is in place, the BST has a clear advantage over the other data structures presented thus far. The major weakness of the BST is the level of complexity involved in writing the programs that create and maintain the data structure, but its power and efficiency once in place quickly offset those difficulties.

Algorithms

Now that we have an idea of how we can arrange the data we are given, we can begin to look at how we can manipulate and perform functions on the data structures listed above. For our purposes here, an algorithm will be defined as a series of operations that the system performs when we want a specific function. In essence, we want the computer to do something, and how it does that thing varies greatly depending on the data structure. We’ll evaluate each data structure within the scope of each function or algorithm, noting which data structures perform best or worst.

1. Search

With large data sets, searching to find a specific item is often required. For this data set, we were tasked with entering a bid ID and finding the bid that has a matching bid ID. This may seem extremely simple on the surface, but the data structures we implemented handle this function very differently. In a Vector or Linked List, the first item is read and compared to the criteria entered. If the first item does not match, it moves on to the second, then the third, and so on so forth. We call this a loop, and the computer will continue comparing values until it finds a match or reaches the end of the data set. Depending on the size of the data set, this can mean thousands of comparisons. If the size of the data set is N, then the average number of operations needed to complete a search is with a Vector or Linked List is equal to N/2.

With a Hash Table, the search algorithm is much more efficient. If we search for a bid within a Hash Table, we can perform the same mathematical (modulo) operation on the entered bid ID that we used when initializing the Hash Table, which directs us to the line that holds bids that returned identical values. This reduces the number of operations needed to find a searched item by a staggering amount. With a sufficiently large denominator and limits on the bid ID key value, the number of comparisons needed in order to find a specific bid regardless of the size of the data set would be equal to 1. While this data structure does sacrifice a small amount of memory, it has drastic positive effects on performance, which is crucial when hosting a server being accessed by any number of individuals at the same time.

A BST that has an ideal structure with 10,000 items would only require a maximum of 14 comparisons since the tree is perfectly balanced on both sides. This can be seen in BinarySearchTree.cpp at lines 125-147. Searching the BST only requires a small while loop that navigates the tree based on the value entered. It determines the correct path by comparing the entered value to the values stemming from the root. However, the Hash Table and BST scenarios are both ideal situations, and while they are both more effective than Linked Lists or Vectors, incorrect design of the data structure can slow the algorithm greatly. In summary, the Hash Table performs best when searching, followed closely by the BST. The Linked List and Vector perform moderately when searching.

2. Sort

Sorting the items in a data set is also a common function needed for many reasons, especially as items are added or removed from the data set. With a Linked List, Vector, and even Hash Table, we encounter a problem similar to the shortcomings of a Linked List and Vector when searching. If we want to sort a Linked List or Vector, we have to read the first value in the data set and compare it to the subsequent value, storing the lesser of the two until we have gone through the entire data set. We then have to repeat this as we increase the threshold until we have found the least item, second least item, third least item, and so on. Effectively, the number of operations required if the size of the data set is N is equal to N!, which is remarkably inefficient as the size increases.

We encounter the same problem when using a Hash Table as well, with certain exceptions. In an unusual and ideal circumstance for this function, there would only be one item in each line of the Hash Table, meaning the lines are already in order. However, that would require sorting based on the Hash Table key value, and would still require at least N comparisons to ensure that an item even exists in each line.

That brings us to our last data structure, the BST, who excels in this area. The design of the BST mandates that each item stored to the left must be lesser than the item itself, meaning that the data set is already sorted. We just have to know how to navigate the tree structure. By traversing down and to the left (lesser) until there is nowhere else to go, we have arrived at the smallest item in the data set very quickly. By going left, reading that item, going right, then going back up in that looped priority order, we are able to generate a sorted version of our data set with as few operations as needed to traverse the entire data set. This can be seen in BinarySearchTree.cpp at lines 175-184. Printing the entire data set using the inOrder method only requires that we ensure the node exists, recursively call inOrder using the left child, print the data of the current node, and recursively call inOrder using the right child. This logic will properly navigate the entire tree more efficiently than any other sorting algorithm on display here. A properly managed BST is clearly the superior option when efficient sorting is considered, and it excels at searching when compared to a Linked List or Vector. Only the Hash Table is more efficient when searching under certain circumstances, and even then the difference between the two is negligible. For these reasons I strongly suggest moving forward with a properly managed BST that accounts for height limits and balance handling in order to maintain the data structure's integrity.

3. Hash/Chaining

Hash/Chaining involves exception handling within a Hash Table. If we have 100 items that each have a unique numerical key, and we use the modulo operation to divide that key by 100, we would be left with the remainder. This would be used as the index in the Hash Table, which would ideally have one entry per index location. However, that is assuming that the key ranges from 1 to 100 without gaps or overlap. If the keys did have values such as 31, 78, 131, 157, . . . and so on, we would have multiple values attempting to overwrite each other at specific index locations. The bid with a key of 31 and 131 would try to take the same spot, since modulo 31/100 and modulo 131/100 both return the Hash index of 31. This exception is what we need to account for, and that kind of exception handling is known as Hash/Chaining.

Instead of storing one single value at each of the Hash index locations, we instead add a Vector or Linked List at each Hash index location. When we add a bid to the Hash Table, we determine the Hash index based off of bid ID. We then are directed to the Vector or Linked List located at that Hash index location. The bid is then added to the end of the Vector or Linked List, becoming the first entry if empty. This can be seen in HashTable.cpp at lines 129-156. The key is calculated based on the value of the bid ID, and the bid is used to either initialize a new list at that index if empty, or added to the end of the list if it exists already. All the changes to the algorithms that need to be handled are rather simple, mostly involving a comparison or two to ensure we have the right value.

Student’s Choice

My selection for the Student’s Choice data structure/algorithm pairing is the Hash Table/Search. While the BST is certainly powerful when searching and superior when sorting, I find the concept of a Hash Table to have the perfect balance of efficient design and simplistic implementation. The Hash Table takes a unique key and uses that as an index, creating lists at each location with only a few items in each. The efficiency of the search is apparent in that when given search criteria, we already know exactly what location in the Hash Table to check. The number of comparisons needed is minimal when compared to other data structures, only becoming inefficient if the Hash Table is poorly designed.

The Hash Table that we demonstrated in HashTable.cpp was effectively implemented, using a modulo denominator equal to the size of the data set. This balanced the need for a small memory impact via index size with minimizing the number of items in each bucket. By using proper exception handling and comparisons to ensure we did not overwrite any items when adding them to the buckets, we were able to create a Hash Table that performed exactly as the data structure was intended.

The Hash Table that we designed used modular programming in that the methods for adding items, removing items, searching, and Hash/Chaining are self contained, meaning they can be included and called in variant programs moving forward with minimal changes. The change that will likely require the most attention when converting our Hash Table will be the denominator of the modulo operation, since we will want one sufficiently large to keep fewer items in each bucket. A safe number would be the projected maximum of items in the data set at any given time. Other than that, a grasp of the relevant file paths and input needed for the methods are all that is necessary to reuse the pairing we created. One method that requires no modification at all is the PrintAll method found in HashTable.cpp at lines 162-188, looping through each bucket one by one until it reaches the end of the data set. This is a prime example of modularity in our programming; we design algorithms that work with the data structure architecture instead of a specific iteration of that data structure.

Modularity is an incredibly powerful concept since it minimizes the number of changes that need to be made to a program when using it for different scenarios. However, there will always be scenarios that do require a change in programming, and no amount of modularity can prepare for that. The next best thing is proper annotation so that any developer, including the original author of the program, can quickly determine what each portion of the program does. At a minimum the developer making changes needs to understand the purpose of the lines or method they are changing, but a firm grasp of overall program functionality will yield the best results. It can often be confusing when trying to read a program, and walking through the logic line by line to determine its function can be tedious. With proper annotation, the developer can simply read the comments to determine where they need to make changes and what those changes might look like.

For example, let’s say we wanted to include a search method that returns every item in the same bucket as the searched item instead of just the searched item. A developer would likely start by copying and renaming the existing Search method in HashTable.cpp. Without proper annotation, the developer would have to go through the method and read the code in order to find out what the method does and where the changes need to be made. With proper annotation, the developer would be able to read the comments and find the portion of the method that needs adjusting.   In this case, the developer would likely merge lines 223-234 to walk through the entire linked list and return all the bids found there.

Conclusions

In conclusion, the data structures that we have implemented have varying strengths and weaknesses depending on the functionality desired. Each data structure is defined and interacted with in entirely different ways, and as such, the algorithms present in a program depend heavily on the data structure chosen. If halfway through a project we decided to convert the data structure from a Hash Table to a BST, nearly all of our algorithms would need a complete redesign. We discussed modularity in the programs earlier, but effective modularity for an algorithm means that it can interact with nearly any iteration of a specified data structure, not multiple kinds of data structures. With all that in mind, we encourage you to make a decision regarding the data structure before any further steps regarding implementation or development are taken in order to avoid the development of unusable code. A properly implemented data structure is extremely powerful, but it does constrain our development and prevent us from switching to other kinds of data structures with ease.

After you have decided on a data structure based on the strengths needed and weaknesses tolerated, we can then move forward with the specific algorithms requested. The importance of powerful and well defined algorithms is apparent when working with complex data structures in that once an interaction is defined, it can be reused with great ease. For example, if you want to allow users with certain credentials to enter new bids on the website, we would be able to read the information presented and pass it into the Hash Table with the Insert method. Algorithms do constrain development in a small way however. If we wanted to incorporate an "Edit" function that read a specific bid and changed its values, we would likely have to use a combination of the Search, Remove, and Insert methods in order to replicate an edit function. If we weren't using algorithms, we would have to code the edit function from the ground up, potentially making something slightly more direct and efficient. However, the benefits of modular algorithms once a data structure has been decided far outweigh the limitations, and any situation like the one above can be solved by creating a new algorithm if any issues do arise.

Several things that were learned by our development team while creating these pairings for your review include large and small scale problem solving, along with sequential logic and development etiquette. We developed the data structure/algorithm pairings in order, starting with the least complex first and the most complex last. This built upon the knowledge we had with each successful program, allowing us to better prepare for the complex programs and easily revisit the simpler ones. Problem solving skills can be seen in HashTable.cpp in the Search method on lines 206-237. The value of node->key strangely initialized as a massive value beyond the scope of a standard integer. By incorporating a comparison with the value of UINT_MAX, which is the maximum value that an integer can have, and an assignment to UINT_MAX when the node->key was defined, we were able to implement proper exception handling while accounting for a consistent initialization anomaly.

Sequential logic skills are on display throughout most of BinarySearchTree.cpp, but a notable example is within the removeNode method on lines 186-226. After checking to ensure the tree exists and finding the node we are searching for, we then had to implement checks for four different cases regarding the positioning of the node's children. This alone is makes this method impressive, but the truly fascinating step is in the last case; if the node has two children. In that scenario, the method checks for the left child of the node's right child, reads the bid from the current node, and passes that as an argument along with the right child into the removeNode method.

It recursively calls itself as long as the node in question has two children, working its way down the tree until it finds the bottom. Even with diagrams, the moving parts involved in that algorithm are difficult to grasp.

Our development team has shown their competency and proficiency when working with complex data structures, as many of our members pursue their skills outside of the professional environment. For example, I will be taking the skills I refined while working on these programs and applying them to a video game I am developing. It is a text based interaction for now, and it allows you to take on the role of a wealthy tycoon in a medieval/fantasy setting. You purchase materials, invent arcane devices, send mercenaries on monster hunts, buy property, and sell your inventions for a profit. While the functionality of the game is a passionate topic for me, the proper management of the data will be key to ensuring modular code and efficient performance. The statistics for the monsters will likely be held in a large Hash Table, and the various inventories that the player owns will be held in Binary Search Trees for fast ordered display.

My personal side projects are not particularly relevant to our collaboration; I simply wanted to illustrate the point that data structures and algorithms are more than concepts we have to work on for our development team here at SNHU. Programming is something that we genuinely enjoy doing. We take pride in our work, and we actively sharpen our own skills for our own sake. This is our passion.