Dylan W. Sanderson

2/19/2020

CS-260-J3682 Data Structures and Algorithms 20EW3

Final Project: Reflections

We are tasked with addressing a particular data set and interacting with it in specific ways. Attached you will find some code segments that can be implemented in a larger database program. These individual code segments each fulfill a different purpose which shall be discussed later in this document. Each code segment has strengths and weaknesses, and, therefore, are suited to different purposes. Fortunately, the data set we are drawing from is uniformly constructed, which made the data structuring coding capable of being reused across other similar data sets with a matching structure, and each code segment can effectively be implemented across the current data set.

**I. Data Structures**

There are a number of data structures with a number of advantages and disadvantages, in speed of various functions, storage size, and program complexity. The fascinating part about most of the data structures is how radically different the same set of data can be sorted and interacted with. We also have run into massive shifts in processing times, as well as the capabilities of many structures.

First, we will discuss vectors. As can be seen in the attached files, there is a file titled VectorSorting.cpp which directly handles the data as a vector. The idea is relatively simple. We start by creating a bid object, with values like title, fund, and amount stored within the object. Then, each of those is placed within a vector, in which they are referenced by the name of the vector and an index number. In our case, we named the vector bids, and used the push\_back() vector function as we were pulling values from the original data set to add each bid item to the end of the vector.

While this is all relatively easy to program, the disadvantage of the vector data structure is the processing time. Additionally, we also are not necessarily dealing with an already sorted set, and may need to sort before we attempt anything else. Depending on the sorting algorithm used, the processing time can be significant. If selection sort is used, each item is compared to each other item countless times, with each iteration only having one less comparison until the items are ordered. In a ten item vector, this wouldn’t be a problem, but one this large suffers with the number of comparisons needed.

The quicksort (or partition sort might be a more accurate name) algorithm can be relatively quick, but neither sorting algorithm outweighs the data retrieval time. As with all vectors, items are referenced by their index within the vector. If you are trying to find a particular bit of data, the only way to find it is to start at the beginning (or the end) of the vector, and compare every single item to the value you are trying to find. Worst case scenario, the item is the last item in the vector, you have unfortunately done a comparison on each and every item in that vector.

The file labeled HashTable.cpp is our next file for consideration, with it’s own particular set of solutions for the shortcomings of vectors while still maintaining similar functionality. Rather than storing the data sequentially and requiring it to be resorted every time new data is added, a hash table inserts the data into a pre-constructed table based on a sorting key. In our case, we used the modulo operand on the set size and a key value, which we pulled as the bidId using atoi. This allowed us to create a (hopefully) distinct key value for each node, making finding them again incredibly easy, as we would only need to recalculate the key value based off the ID, then do a singular comparison at that particular spot. However, if that key value was already taken on insert, we used the nextNode pointers until we found an empty spot to place it.

Overall, this means we are able to create a fast, efficient database. The search algorithm in an optimal case (no repeated keys) would only need one comparison. Even if there are a few collision instances and we had to do a few extra comparisons, it is still notably faster than the arduous task of drilling through a vector a spot at a time until the desired value is found. With a has table, we are able to jump ahead of the line.

The last file I would direct your attention to in this section of the examination is BinarySearchTree.cpp. Here, we were able to build a binary search tree from the data provided. The binary search tree structure is inherently efficient and building a naturally sorted database, using basic less and greater than comparison operators. In our case, since we checked the right with a greater than first, we directed any potentially equal to values down the left branch. With each node only holding a bid, left pointer, and right pointer, the storage method is relatively data efficient, depending on the order the data points are read in.

The inherent inefficiency of a binary search tree is that it only remains balanced if the values being added are midpoint values, meaning if we have nodes 1-9, first we insert 5, then 3, then 4, and so on. If the nodes are entered already ordered, and are entered as such (1, 2, 3…) then we wind up with a tree that resembles a vector with each node having a seemingly pointless null pointer attached. However, if the data entered is balanced, (or, after entry, the tree is run through a balancing algorithm) then we have the natural ability with the data structure’s search algorithm to halve the number of potential candidates each comparison.

**II. Algorithms**

A wide variety of search algorithms were build in these programs, all with a variety of different search speeds based on the data structure. The two most efficient were in the file HashTable.cpp and BinarySearchTree.cpp. We will begin by discussing a basic vector search algorithm, and then the superiority of the two algorithms included in these programs.

When you are dealing with a search across a vector, the search is naturally innefficient. This is due to the fact that a vector is structured only based on the number of items in it. It does not matter if it is a simple numbered vector or a linked list, you are trapped starting with the root or position 0 item, performing a comparison, and then sequentially proceeding down the list, comparing every single item. In the worst case scenario, if the item is the last item in the structure, then you have to compare every single item in the data structure, taking huge amounts of processing time. This would mean a data structure of, say, 4 million items would require 4 million comparisons.

In our binary search tree algorithm, the data structure naturally sorts itself, and lends itself to efficient searches. If the tree is balanced, even dealing with an item that is a bottom leaf requires minimum comparisons, as each comparison effectively cuts half of the remaining items off of the comparison. As such, we are able to efficiently maneuver to exactly the desired value, or determine if the item does not exist in the tree. In the case of our code, we use a default constructor to make a nullified object to return to the main method, indicating that it does not exist, if we reach a comparison that leads us to a null pointer.

The hash table algorithm is possibly even more efficient, as each item’s location is keyed based on the item’s ID. As such, when we are searching for an item by said ID, we do a simple calculation based on the hash value, and get the exact key, meaning only one comparison is needed in most cases. If the spot we check is null, we know said item does not exist. If something is there and it matches ID numbers, then we have located and returned the desired item in only one comparison. If something is there and the ID does not match, then we have more than one item with matching keys, and do a quick sequential comparison to find the matching item. In a table hashed as efficiently and dynamically as the one we coded, the chances of collision are relatively low, and it is likely that, even in the case of a repeated key, it would only be a few comparisons to find the desired item.

Sorting algorithms come in more varieties than the data structures they are sorting. In some cases, like a binary search tree, the data structure is naturally sorted. However, vectors are not naturally sorted, and our VectorSorting.cpp file required it’s own sorting algorithm. Two sorting algorithms were installed to compare efficiency, so we will discuss both, and why ‘quicksort’ is the obvious choice.

First, we made a rudimentary selection sort algorithm to be able to see how effectively it worked, and in the end, it was functional yet slow. Selection sort requires a large number of comparisons, as it starts by grabbing the first value in the vector, and comparing it to ever other item, looking for the smallest value. After comparing every single item in the vector, the lowest item is moved to the first position (indicated by 0), and then the process begins anew on every item excluding the first. This means we compare every value in the vector excluding the first, then the next iteration compares ever item excluding the first two, then the first three, and so on. This results in a massive number of comparisons in order to create a sorted list, and, while functional, it is not effective. As such, we decided to create a quick sort algorithm.

The quick sort algorithm relies on partitioning and recursive function calls to succeed. Basically, it takes the entire vector and chops it apart into single-item vectors. It then pieces it back together into consecutively larger vectors by inserting based on comparing the values within each partition sequentially, through the recursive calls to quicksort based on the value returned from the partition function call. As such, as the partitions rebuild, each item getting placed into the next level’s vector is done with an individual comparison, rather than being repetitively compared with a selection sort algorithm. The result is a much faster yet equally capable sorting algorithm.

Our hashing algorithm was an effective one, as can be seen in HashTable.cpp. Given that we were able to enact a hashing algorithm to create quite distinct keys, we ended up with a well-hashed table. We created the hash function to initially be called with a value that would be run through the operation modulo setSize, which, if we were dealing with a unique value, would inherently create unique keys. We then ran the value of the bidId, which is distinct for every item, through the calculation, thereby creating a unique key you assign to each node in the table.

In the absolute worst-case scenario of a collision, we implemented a chaining algorithm to handle that. We verified that, upon inserting a node with a specific key value, that the spot we’re trying to put it in the hash table is a null pointer. If it is not, we implemented a nextNode\* pointer on each node, so then we iterate to the node’s next pointer until we find a null spot. As soon as we find a null spot, we inserted the algorithm.

**III. Student’s Choice**

Out of all of the submitted code sections, my favored one is the Binary Search Tree algorithm, in BinarySearchTree.cpp. I think the functionality of this particular code is surprisingly efficient given the data structure we are dealing with. I will discuss the benefits of the code in the following segments.

Overall, the code was effective, in that we were able to build a tree in which each node was capable of holding the bid data, as well as pointers to two other nodes. It also held within it all of the necessary CRUD functions we were asked for at this point, though I would add more algorithms in the future that I will discuss later. Given that each node has a ‘bid’ struct at it, and the comparisons for the structure were uniform, the constructed tree is a naturally sorted data structure, which even remains sorted as more items are added. Since each item’s spot within the tree is based on comparisons within the code (in this case, greater than goes right, all else left), it does not matter how many data points are entered, the structure remains sorted, as the insertions algorithm mirrors a sorting algorithm in it’s design.

The code is written modularly to increase simplicity, readability, comprehension, and recursive calls. We have two separate functions that call themselves recursively, and some functions call other functions within the code. The insertion function in particular is an interesting example of this. First, we call the insert function with just the bid as an argument. This performed a simple null pointer check (and inserted at the root if null) and, if the tree is already populated, called add node with the root node and the bid as the arguments.

The add node function began by checking if the bid should be put to the left or right of the current node. Whatever it decided, if the next pointer was null, the bid was inserted. However, if it was not null, addNode was called again, but this time with the next child and the bid as the arguments, so the function could recursively traverse the tree until it found a proper null spot to insert the bid. This modular structure to the code was both efficient and effective in it’s implementation.

This code could be reused quite effectively on any data set structured the same as the one it is written for, and could be utilized for other structures with minor modifications. The majority of the code is written in such a way that, were some other item rather than a bid object put into each node, as long as there is a value to pull out and compare, it will add, search, and remove the same way. The structure of the code is not based inherently on the type of data it is handling, rather, it simply needs the data being handled to have a comparable integer value in order to build a sorted data structure. It uses the atoi function to pull the needed value out, which, if we were dealing with a different data structure, would only require a few lines of code to be altered to work just as effectively.

The annotating within the code for the binary search tree is effective and direct, as can be seen while trying to trace the code. It is written in plain language, and, in many cases, line by line. A good example of this is within the search function. We begin with an annotation describing exactly what each initially declared item is and exactly what it is going to do. Then, we state that we will be using a while statement, both describing why we are using it, and how we will know when the loop is done and should close. The annotations describe exactly what is happening within the function concisely without over-describing and cluttering the code, such that anyone with a basic understanding of programming could easily follow exactly what it is doing.

If we were to continue with this function, I would include a few other methods for ease-of-use. First, I would right a function for printing the items in order, by descending as far left as possible, then printing the parent, then the right side of the parent, and working our way back up with recursive calls. Additionally, I would implement a function to be regularly implemented at low traffic times that would disassemble and reassemble the binary search tree as balanced as possible, ensuring minimum processing time and maximum data efficiency during high volume times.

**IV. Conclusions**

Data structures are of vital importance when designing a program. The data structure used can greatly affect the efficacy of the program it is being written for. For instance, let’s say you are writing an ‘undo’ function for a particular program, and each action within the program can be stored as an object. If you were to use a Binary Search Tree to store each occurrence, the undo function would be clunky, as the tree would have to be searched constantly for the most recently time-stamped event, depending on how it was sorted. However, if the events were stored in a vector, with each new occurrence appended, one would simply have to use a getSize function, then pull the last item out of the vector. It would be fast and efficient. Alternatively, if you need a sorted information set that is constantly added to and removed from, a binary search tree would be a superior option to a vector, as it is naturally sorted, and would not require resorting on a regular basis while items are being appended.

The algorithm choice is equally as important, as can be easily demonstrated with the vector sorting code. While dealing with a smaller data set, the processing time difference between selection-based and partition-based sorting algorithms may be negligible, but when dealing with thousands or millions of data points, the processing time would be easily noticeable. The important part of the algorithm is that it is efficient, meaning it gets the job done in a resource-effective way. A bad algorithm choice is the difference between a functional program and a good program.

At this point, it is difficult to say exactly what effect the lessons of this course will have on me. I guarantee that the increased knowledge base will aid me professionally, as some of my friends who are willing to give me a foot in the door at their companies work specifically in database management. Even just having an understanding of how the database styles work and what algorithms could be implemented to speed access to them up is endlessly valuable. At some point I hope to dabble in my own video game design, and forms of binary trees have been used in 3d rendering for decades now, starting with games like the original Doom. Ultimately, there isn’t a negative to having more knowledge, and I hope that I will not only walk away from this course with a deeper data structure understanding, but with functional skills that I use in the future.