Mitchel Barton

CS-260: Final Portfolio

10/17/2020

In the realm of computer science and software development, data structures and algorithms are pivotal part of creating secure, efficient software. There are many unique problems a programmer might confront during the development process and knowing which algorithms and structures to deploy will greatly affect the useability of the final product. For this project, managers of eBid Nashville were looking for a program that allows users to access information on the items sold through the eBid website. As a programmer with SNHU software, I have contributed specific functional components for this larger piece of software.

The vector structure is a fundamental tool in storing and organizing data, often used in lieu of or alongside arrays and other structures. A vector is an ordered list of items of a given data type; each item in a vector is called an element. The ‘Structures’ folder in this portfolio perfectly introduces this fundamental tool. At the beginning of the program a structure is declared to hold the global variable definitions for the bid’s title, fund source, and amount. This structure will be built upon in later development stages. The bid information is read by a parser method from a CSV file and loaded into to a vector data structure. This method stores all the bids in an ordered list that allows the program to loop through each entry to display all of the bids in the CSV file.

Some more complex data structures introduced in later iterations are has tables and tree structures. Examples of these can be found in the ‘HashTable’ and ‘BinaryTreeSearch’ folders, respectively. A hash table is a structure that stores unordered items by mapping (or hashing) each item to a location in a vector (or array). This can drastically reduce operation time for finding a specific element being searched, since the program does not have linearly order and search through a list. A tree data structure is similar to a linked list, in that it’s an ordered structure. The tree is a collection of nodes, where each node is a data structure consisting of a value, linked together with a list of references to nodes, the “children”. Think of the biological tree structure, where there is a base root at the top, that branches off into parental and children nodes from that root. The order of nodes in a binary search tree means that each comparison skips about half of the remaining tree, so the whole lookup takes time proportional to the binary logarithm of the number of items stored in the tree. This method is much faster than the linear time required to find items by key in an unsorted vector.

A well-defined data structure is useless without the proper algorithms to traverse them. Each data structure has a best-case time complexity for traversal, denoted by Big-O notation. This is a mathematical way of describing how a function (running time of an algorithm) generally behaves in relation to the input size. In Big O notation, all functions that have the same growth rate (as determined by the highest order term of the function) are characterized using the same Big O notation. In essence, all functions that have the same growth rate are considered equivalent in Big O notation. Let us start with a relatively simple algorithm, the linked list search. Given a key, a search algorithm returns the first node whose data matches that key or returns 0 if a matching node is not found; this linear search on average takes *O(n)* time to return a value. The ‘LinkedList’ program demonstrates this well by storing the bids in a linked list, then using a constructor to set default head and tail nodes to be appended or prepended into from the dataset. When a search method is run, the program uses a *‘while’* loop to iterate through each node, looking for a match to a user specified bid ID.

Moving down the list we get to vector sorting, appropriately demonstrated in the ‘VectorSorting’ program. Two different sorting algorithms were used to compare speed and efficiency in returning the desired vector, Quicksort and Selection Sort. A Quicksort is a sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each of those parts. To partition the input, quicksort chooses a pivot to divide the data into low and high parts. The pivot can be any value within the array being sorted, commonly the value of the middle array element. In this program the bids a re loaded from the CSV file into a vector, then the structure is partitioned into low and high parts. The algorithm recursively calls the Quicksort method using the midpoint value until it reaches the beginning and end of the dataset. This algorithm on average takes *O(n log(n))* time to complete. A less efficient method used for comparison was the Selection Sort. This algorithm treats the input vector as two parts, a sorted part and an unsorted part, and repeatedly selects the proper next value to move from the unsorted part to the end of the sorted part. On average this algorithm takes *O(n^2))* time to complete, significantly slower if working with a large dataset.

Finally, we arrive at Hashing and Chaining, again found in the ‘HashTable’ program. When using a hash table, the programmer should make sure to account for collisions. A collision occurs when an item being inserted into a hash table maps to the same bucket as an existing item in the hash table. Chaining handles hash table collisions by using a list for each bucket, where each list may store multiple items that map to the same bucket. The insert operation first uses the item's key to determine the bucket, and then inserts the item in that bucket's list. Searching also first determines the bucket, and then searches the bucket's list. In this program the bids are loaded from the CSV file into a hash table structure, where the hash value is calculated by using a modulo calculation against an unsigned key size. The search method calculates a key for the requested bid, and tries to retrieve a node using that key. If a node is found that matches the key, the method walks the linked list (chaining) to find the match of that specific bid ID. This algorithm is one of the faster methods demonstrated, taking only *O(1)* time. This can greatly simplify the lookup process. When it comes to the impact of hashing on performance, it is and ideal method of security for resource constrained systems. Hash algorithms use far less resources than other cryptographic algorithms while still providing an acceptable level of security for applications (Stapko 2008).

Throughout the iterations of this program, one has stood out as excellent. The binary search tree data structure and algorithm allows for fast lookup, addition, and deletion of data items into nodes of the tree. The order of nodes in a binary search tree means that each comparison skips about half of the remaining tree, so the whole lookup takes time proportional to the binary logarithm of the number of items stored in the tree. This method is much faster than the linear time required to find items by key in an unsorted vector. This program is modular in the way methods and internal structures are laid out. A separate internal structure is made for the tree nodes to work alongside the bid structure already present in the other iterations. Two different constructors are present in this structure to account initialize with a given bid. Many of the tree’s algorithms are reusable for different application and data types, such as the node removal method. This method is separate from any influence of the bid data type by using an independent method for node removal. It recurses down the left subtree, and uses loops to account for one, two or no children (leaf nodes) to remove a specified piece of data from the structure. Annotations are used sparingly in concurrence with best programming practices; proper spacing and in scope with the correct functions they are describing. For instance, in the search method it starts with a comment describing the current node pointer as the root of the tree. Three other annotations are used to subdivide the loops in the method to describe the downwards traversal, node matching, and left traversal for when the bid is smaller than the current pointer.

Data structures are just another tool in the software developers’ belt. There is a best use case for a data structure based on the type of data being worked on, and the client’s specifications. If client’s program must be memory conscious based on hardware limitations, a hash table might be more appropriate for its’ simplicity in lookup processes while maintaining an acceptable level of security. The same can be said for algorithms, it all depends on what kind of data is being worked with. If the structure is handling a large data set, like in a data center, a binary search tree would be a good choice for its’ speed in searching, inserting, and deletion. The lookup takes time proportional to the binary logarithm of the number of items stored in the tree, so operating times are significantly lower when working with these large sets. The most important lesson learned was not memorizing the coding of these data structures and algorithms, but when and how to implement them. The ability to recognize the type of data being worked on and the challenges associated with it will lead a good developer to recognize what combination of data structure and algorithm would most efficiently solve those challenges. Specifically, I could see something like a hash table with chaining working well for my brother’s mechanic shop. He is typically stuck working on low end machines to track his employee’s spiffs for bonus payout, and this type of simplistic sorting algorithms could streamline the process of tracking his employee’s bonuses.

Work Cited:

Lysecky, Roman et. al. August 2017. *Data Structure Essentials*. zyBook ISBN: 978-1-394-02457-5.

Stapko, Timothy. 2008. “Practical Embedded Security”. https://www.sciencedirect.com/topics/computer-science/hash-algorithm