**7-1 Final Project: Portfolio of Work and Reflection**

Marc A. Aradillas

Southern New Hampshire University

CS-260: Data Structures and Algorithms

Professor Nathan Lebel

February 19, 2023

**7-1 Final Project: Portfolio of Work and Reflection**

**Data Structures report**

Data structures and algorithms are fundamental concepts in computer science that play a crucial role in solving complex problems efficiently. Data structures refer to the way data is organized and stored in a computer's memory, whereas algorithms are a set of step-by-step instructions that are used to perform a particular task. By choosing the appropriate data structure and algorithm, we can optimize our code to improve performance and reduce the amount of time and resources required to complete a task. Whether you're building software, designing systems, or working on a data-driven project, a deep understanding of data structures and algorithms is essential to write efficient and scalable code. In this context, it is vital to explore the different types of data structures and algorithms to be able to make informed decisions when designing and developing software systems.

A vector is a dynamic data structure that allows you to store and manipulate a sequence of elements. It is a container that can hold elements of the same data type and can be resized at runtime to accommodate new elements. The elements in a vector are stored in contiguous memory locations, which makes it easy to access and manipulate them. Vectors are commonly used for implementing dynamic arrays, and they offer several advantages over traditional arrays, including the ability to resize dynamically, efficient element access, and the ability to insert and remove elements at any position. When using a vector, it is important to consider the memory requirements and the complexity of operations such as resizing, insertion, and deletion, as they can impact the overall performance of the program. The program I built in Lab 3-2, a vector was used as a container to hold a collection of bids. A for loop implemented to add a collection of bids after reading rows of the CSV file after initializing the CSV parser. After the vector compiled to load the bids the program then follows algorithms to partition and quicksort or selection sort the data.

A hash table is a data structure that allows you to store and retrieve data quickly using a key-value pair. It is a type of associative array that uses a hash function to map keys to indices in an array. Hash tables provide constant-time complexity for insertions, deletions, and lookups in the average case, making them one of the most efficient data structures for implementing symbol tables. Hash tables can suffer from collisions, which occur when two keys map to the same index in the array, resulting in slower performance. To address this issue, different collision resolution techniques can be used, such as separate chaining or open addressing. Hash tables have efficiency and versatility which I have displayed in Lab 4-2. The “HashTable” class in the program contained struct and a default struct to have a bid, key, and Node pointer. A second constructor added to initialize the functions. The functions insert and search provided bids to be insertable and searchable. The use of a key and node pointers by using if, else, and while loops to iterate through the hash table data structure aided in creating an algorithm to assign keys (looping until null) through the bids.

A tree is a non-linear data structure that represents a hierarchical structure of elements, with a root node at the top and child nodes branching out below. It consists of a set of nodes, which are connected by edges to form branches, and each node can have zero or more child nodes. Trees are commonly used for organizing and representing hierarchical data, such as file systems, organization charts, and decision trees. They can also be used for searching and sorting data, as well as for implementing abstract data types such as sets, maps, and priority queues. Trees come in various types, including binary trees, AVL trees, B-trees, and red-black trees, each with its unique properties and applications. When working with tree structures, it is important to consider the balance of the tree, as unbalanced trees can lead to poor performance and degraded functionality. In my Lab 6-2 program file a class named “Binary Search Tree” was used with private members (addNode, inOrder, and removeNode) and public members (inOrder, Insert, Remove, Search). The program implements if, elseif, else, and while loops constructively to be able to expand the tree from root to nodes. The use of pointers and operations to set right or left nodes makes this data structure search quickly to the matching node or a node close to matching it. The use of a compare method recursively loops down a right subtree or a left subtree and when conditions are met a temporary node pointer is used in further loops to remove or add nodes (leafs) in the binary search tree.

**Algorithms report and Personal Choice**

Search algorithms are used to find a specific item or location within a dataset. Search algorithms are typically used to retrieve information from a large dataset, and they can be classified as either sequential or non-sequential. Sequential search algorithms check each item in the dataset in turn until the target item is found, while non-sequential algorithms use a more efficient approach based on the data structure. The performance of a search algorithm is typically measured by its time complexity, or how long it takes to complete the search, and its space complexity, or the amount of memory required to perform the search. The choice of search algorithm depends on the type and size of the data being searched, as well as the desired performance characteristics. In Lab 3-2 I used a linked list to search for bid ids. The linked list was a while loop that checked to see if a bid was null and compare the bid with the csv file bidID. If it did find it in the linked list it would return that current bid and loop through each node on the list.

Sorting algorithms are a fundamental concept used to rearrange data into a specific order. The most common use of sorting algorithms is to sort data in ascending or descending order, but they can also be used to sort data based on other criteria such as alphabetical order, date, and time. Some popular sorting algorithms include bubble sort, selection sort, insertion sort, merge sort, quicksort, and heapsort. The performance of a sorting algorithm is typically measured by its time complexity, or how long it takes to sort the data, and its space complexity, or the amount of memory required to perform the sort. The choice of sorting algorithm depends on the size and type of data being sorted, as well as the desired performance characteristics. In Lab 4-2 sorting algorithms implemented were the selection and quick sort algorithm. The quick sort recursively partitions the data until that it is ordered appropriately. Selection sort in the program looped over the data then swapping each value until data is organized in an ascending order after compilation.

Hash chaining, also known as separate chaining, is a popular collision resolution technique used in hash table data structures. Hash chaining is an efficient and straightforward collision resolution technique, and it can handle many collisions without compromising the overall performance of the hash table. However, the performance of hash chaining can degrade when the linked lists become very long, and it can require more memory to store the linked lists than other collision resolution techniques. The hashing algorithm implementations in my Lab 5-2 program chaining was used in the insert function and I described it in detail earlier in my data structures report. Essentially when the data organizes, and calculated key bid that information collided then logic implementation was made to print bids with key and those not matching without key in the returned display.

In my programs I feel my best worked is exemplified when I was working on the Binary Search Tree. A combination of utilizing a tree structure and search algorithm worked very efficiently and time was not a issue when compilation gave fair to exceeding results after testing. The program is modularly composed due to the detailed nature of the lines of code. The functions were designed to communicate with one another. This code could be reused for a different project and can be modified or replaced to support new requirements within the structure of the tree and the algorithms for searching through the nodes from root to leafs. The annotations provided exist through the program and are concentrated at the complex portions of the structure and algorithmic functions of the program.

**Conclusion**

The role and importance of data structures in developing computer programs is explained through my reporting paragraphs, but to further touch on how they are impactful or can constrain development it depends on the level of effect the program needs to have and the datasets that are being handled. Algorithms are the other half to a successful and effective program. The program can be constrained in the example of Lab 5-2 the hash table with the chaining algorithm made it a bit constrained due to the collisions, but the algorithm made the program effective by looping and assigning to the next pointer for the nodes based off of the key value. I believe having this knowledge of data structures and algorithms in my skillset will set me apart when developing on projects that require well unorganized data and being able to clean datasets for service or product purposes. It has also ultimately taught me to be cognizant of how working through different processes to understand how they are effective in different programming settings which can depend on