Final Project

Jake Catlett

Southern New Hampshire University

CS-260 Data Structures and Algorithms

John Watson

June 21, 2017

**Data Structures**

**Vectors**

One key, base data type that is often used in C++ (and other programming languages) is the vector. As opposed to data types such as integers or strings, vectors can be understood as indexed lists composed of elements of other data types. All elements within a vector must be of the same data type; however, a vector can be far more flexible in practice than this stricture might imply. While the elements must all be of the same data type, user defined data structures are valid as data types within these vectors. For example, a data structure which includes integers, strings, and Boolean values qualifies as a valid data type for the vector elements. In fact, a vector can contain vectors as elements!

Vectors are therefore important for consideration for use within advanced applications. Without getting into details, there are also important security issues which make vectors a superior choice over similar data types such as arrays (Lysecky, Vahid, Lysecky & Givargis, 2015). As one example, we choose to reference the use of vectors within a program that was created in an attempt to research the loading and searching of auction bids as required by the current project.

Included in the portfolio submitted with this report is a .cpp file titled Lab\_2.cpp. Please reference line 79 in this code file, at which point begins the construction of a function titled loadBids, which performs the function of reading a list of bids from a .csv file and loading them into a vector. The individual bids in this case are stored as a user-defined data structure called “Bid”, which consists of the title of an individual bid, the fund which the bid is associated with, and the amount of the bid – of data types, string, string, and double respectively. This illustrates the point previously made about the capability of vectors to contain unique data types which in and of themselves contain multiple elements of disparate data types.

The vector is therefore capable of storing all the requisite data for each element, stored in an indexed format. This allows for searching and for other looping algorithms which can access the individual bids. The individual fields within those bids can then be accessed and used as necessary, proving that the vector was an appropriate data type to use for this application.

**Hash Tables**

A hash table is a data structure that stores unordered elements in a way which lends itself to extremely fast searching. A hash table has a pre-defined amount of “buckets” which are indexed by number and which each can contain a single data element, or a vector or elements if necessary. A modulus function (or another more complex function) transforms an identifier such as an integer ID number into a smaller key which matches the index number of a “bucket” within the hash table. By reducing the ID into a unique key which matches the index of a hash table bucket, a search capability which potentially requires only O(1) steps for a properly implemented hash table is enabled (Lysecky, et al, 2015). This makes hash tables an extremely powerful and desirable data structure within applications where searches are made on large numbers of data elements.

Within the portfolio provided a hash table was implemented in the file Lab\_5.cpp. Please reference line 48 in this code file, where the implementation of a class title HashTable begins. Within this hash table a set number of “buckets” are created. In our case, this happens to be 179 buckets, based on the fact that 179 bids are included in the .csv file that is loaded. As the ID numbers which are used in the hash function to create hash keys to not fall within the ranges of 1 – 179, a vector structure is used to contain bids within each bucket, thereby allowing for cases where multiple records within the bid database have ID numbers that translate to identical keys. The vectors contain the same Bid structures discussed in the section about vectors, meaning that all the requisite data can be contained within each data element in each vector contained in each hash table vector.

The elegant design of a hash table is particularly suited for our purposes. While the December, 2016 set of bids used within this program contains only 179 records, there is a significantly larger number of bids stored within the file provided for the entire year of 2016. If bids from several years, and/or several were included, this number could quickly grow to a size which would be otherwise unwieldy if stored within a simple vector. Clock tick counters were included in the implementation of this code which proved that searches resulted in locating the correct bid based on its ID number within just a few clock ticks. As such it is highly recommended that hash tables be considered for use within our application design.

**Tree Structures**

Search trees are yet another data structure that was explored during our research into building a successful application for bid storage and searching. Search trees are used to store an unsorted set of data elements in a fashion which lends itself to rapid search capabilities. The roots node in the tree can have up to two “children”, which in turn can each have two “children”. The data elements stored in these nodes are identified by keys, and the key being searched on can be used to determine whether or not to search down the left or right hand branch coming off each node. This allows a search to be executed which can result with the element being searched for found in as little as O(log N) clock ticks.

Within the portfolio provided a search tree has been implemented in the file Lab\_6.cpp. Please reference line 52, where the construction of a class called BinarySearchTree begins. The class includes several methods which manage the creation of a search tree, creation and insertion of nodes, removal and deletion of nodes, and searching and displaying of specified nodes. It should be noted that the construction of a binary search tree such as this is rather complex and will likely result in more time testing and debugging. The structures required for implementing a binary search tree are also somewhat larger than what is necessary for a structure such as a hash table. This needs to be taken into consideration when choosing to use a binary search tree in our application.

If the number of bids loaded into memory is relatively small, then the benefits don’t really justify the time and effort necessary to create and maintain code that incorporates binary search trees. On the contrary, if there is a very large number of bids to be loaded then the amount of memory required to hold them may prove unmanageable. There is, therefore, a “sweet spot” where adding a relatively large number of new bids to the structure will not result in any significant amount of resources being necessary to perform a search, but which will not overload the system’s resources. With the simplicity of a hash table taken into consideration, along with a hash table’s benefit of only requiring O(1) clock ticks to execute the search (if implemented well) then the benefits of using a binary search tree must be weighed very carefully.

**Algorithms**

**Search Algorithms**

While choosing appropriate data structures is certainly a key component of our auction bid search project, choosing which algorithms to implement in order to execute searches and sorting and other functions is just as important. First of all we will take a look at search algorithms that have been implemented, discuss their benefits, and see how the coding that was implemented applies to other areas of importance.

Please reference the file Lab\_3.cpp at line 183. At this point we will find a search algorithm that was created within a class that implements a linked list. This search method traverses a linked list to find a bid identified by a user-specified ID number. If the bid is found then it is returned by the algorithm, and if the bid is not found then an empty bid is returned. While using a simple linked list for storing bids is not feasible, this function in particular is highlighted due to its simplicity, elegance, and possible application to many other data types.

The first thing to note is that the ability to traverse a linked list is essential in any search algorithm that might be used with a hash table or a binary search tree. Hash tables and binary search trees both use linked list methodologies to “join” the separate nodes within the structures, and understanding how to use pointer variables both to implement these structures and to traverse them is a basic principle that must be understood in order to code them effectively. Within this simply-coded method we can see the basic concepts needed to correctly implement search methods in these more complex data structures.

The algorithm must take into account three scenarios: a search that is executed on a linked list which does not yet contain any nodes, a linked list which contains a node holding the bid being searched for, and a linked list which contains nodes, yet does not contain a node which holds the bid being searched for. The algorithm begins by scanning the memory location indicated by the linked list’s head pointer. If the head pointer is null it outputs a message to the console informing the user that there are no nodes in the list and returns an empty bid structure. If the head pointer is not null it traverses the list to find the ID number specified. If at any time the specified bid is found it immediately returns this bid. If the end of the linked list is reached without the specified bid being found the algorithm simply returns an empty bid structure. While it could be argued that the algorithm should also return a console message at this point, it is also true that the algorithm accounts for all three possible scenarios, and deals with them all in a fashion which causes no errors and which allows the code to be compiled without any warnings.

**Sort Algorithms**

Sort algorithms are another type of algorithm that can be used to help organize data and in order to enable the use of certain types of search algorithms. One simple example is that a user may wish to output the bids that exist in a data structure to the console, or (more plausibly) to a file. In this case it is likely that the user would like to have the data sorted in one fashion or another in order to make it more convenient to view or scan through the output – especially if their goal was to have a visible reference to aid in manual searches. Another example is when an application has implemented a binary search algorithm. While a binary search algorithm that uses a bid ID number enables very fast searching capabilities, it is dependent upon the data it is searching through being sorted on the ID that it is using. As such, any application that uses binary search algorithms on data that is not sorted before being entered into a data structure must also use a sort algorithm to sort that data before searching through it.

A fine example of a sort algorithm can be found in the file Lab\_4.cpp, beginning at line 125 and continuing through line 194. Note that there are two functions necessary to implement this search function: one which creates partitions within which to store “chunks” of sorted data, and a simple recursive function that takes the vector of bids as an input and a middle and end number, and calls the partitioning function in order to create partitions which partially sort the data and then merge them back together until the data is fully sorted.

One of the greatest benefits of using a sort method such as this as opposed to a more simply implemented search algorithm such as a selection sort is that while it may be more complex to code correctly, it performs the sort much more quickly. While using a small file containing only 179 records, such as the one used in this case, results in a search being executed so quickly that the human brain cannot perceive the difference in time, this is not the case when dealing with a data set that contains hundreds of thousands of records. In one instance recorded recently the very algorithm submitted within the portfolio referenced above executed the search on 179 records in only 5 clock ticks. In general, a quicksort algorithm can be expected to perform the search in O(N log N) steps or less as compared to O(N^2) when using a selection sort algorithm (Lysecky, et al, 2015).

A very important point to make about executing sorts for end user convenience as opposed to use within a search algorithm is that it is possible that the users may find using ID numbers unhelpful. Suppose an end user wishes to search based on a product name and is unconcerned with (or unable to) identification of bids by ID number. In the file that we have used, a bid title show the product that was bid upon. In this case the sort algorithm would need to sort alphabetically rather than numerically.

The quicksort algorithm detailed in the submitted code does exactly this. At this point, we encourage the readers of this document to compile the code and place the requisite data file in the location where the .exe file is located. Run the .exe file, load the bids, display them, then run the quicksort algorithm and display the bids again. It can be seen at this point that the bids have been sorted alphabetically, and not numerically. This proves that the algorithm can be configured to handle text sorting as well as numerical sorting, and therefore may be very useful in our application.

While a selection sort algorithm is certainly less complex to code, it should be noted that the quicksort algorithm itself has still been implemented with a relatively low level of complexity. The partition function simply sets a low and high number which are user defined, creates a pivot point at the middle location in the data set, compares alphabetic values of the titles of bids in the upper and lower partitions to the middle value to determine if it has worked through the dataset, and swaps the values when the lower partition value is greater than the higher partition value. The quicksort function itself is very simple, as it simply calls the partition function and recursively calls itself twice. The results, in terms of accuracy and number of clock ticks required for execution, speak for themselves. It is therefore highly recommended that this specific algorithm be considered for inclusion in our bid search application.

**Hash/Chaining**

Another very important algorithm to understand and consider for our project is chaining within a hash table. While a “perfectly” implemented hash table will have only 1 record per hash key, there good reasons why this is not feasible for our project. Take, for example, the December 2016 bids list that we have been loading into our applications for testing. This file contains only 179 records. In a “perfect” hash table we would be able to create a has table with only 179 buckets and use the modulus function to return keys that have the value between 0 and 178, with just one bid inserted into each table. We can easily see, though, that our bid ID numbers have 5 digits, and if we were to use the modulus function on these ID numbers we would discover that some keys are never used, and others are used multiple times. Furthermore, we can assume that this small number of bids is not consistent with what will actually be loaded into our programs. If we were to create a hash table with 100,000 buckets (as an ID number with 5 digits would imply) we would use far more memory than necessary, wasting valuable resources. If we were to use only 179 buckets with the ID numbers supplied we would create collisions where we were attempting to insert bids into memory locations that were already occupied.

This is where chaining comes into its own. By allowing the use of nodes which point to other memory locations containing bids that have IDs which result in identical hash keys we solve the problem elegantly, and without resulting in an unacceptable addition of processor resources to continue the search. I would ask the readers of this report to refer to the file Lab\_5.cpp. Within this file please find the Insert method within the class HashTable, located at line 130.

This method very simply and elegantly implements the insertion of nodes containing bids, both in cases where the hash bucket referenced by the resulting hash key is empty as well in cases where the bucket is already occupied by a node. First, the algorithm creates a key by reading the bid ID of the bid being entered and taking the remainder of the ID divided by 179. Note that this number is chosen specifically because of the size of our data set – this can be reconfigured to divide by actual number of bids present in any file containing bids. The algorithm then searches for a node at the location indicated by the hash key in the hash table. If it discovered that this bucket is empty then a new node is created in this location which contains the bid that is being inserted. If the bucket is occupied it then determines whether or not the node’s pointer to the next node is null or not. If it is null, then it creates a new node containing the bid, and “connects” the two nodes by the first node’s next pointer, and the new node’s previous pointer. If it is not null then it repeats this process until it reaches a node with no node located next to it.

The results, again, speak for themselves. If one creates a new column within the .csv file provided which divides the ID by 179, then sorts the columns to look for multiple bids with the same key, one can then reconfigure the code to search for any of the bids which have an identical key. The fact that any of these bids will then be found shows that the chaining algorithm has been implemented correctly, and the resulting output showing the number of clock ticks and total time necessary to find these bids is consistent with the speed that one would expect from using a hash table.

I believe that this chaining algorithm is one of the best candidates we have for inclusion in our project. The fact that our ID numbers begin at about 18000 or so, and can be expected to increase, means that using a 5 digit hash key is not recommended. Using a smaller hash key will most certainly result in collisions when identical hash keys are inevitably produced. As long as the bid structure which is implemented within this specific program is used, the code proves to be extremely modular and portable, and very little refactoring necessary to implement it within another program.

**Student’s Choice**

**Linked List Implementation**

It has been requested that a program be put forth as an example of an effective, and well-executed application. For multiple reasons an implementation of a linked list has been chosen. In the portfolio provided, in the file Lab\_3.cpp, a linked list has been implemented. This section will reference that program and several reasons will be given why this program and its attending code are good examples of both the requisite knowledge and coding skills which are necessary for the project being undertaken.

Within this report both data structures and algorithms have been explored. As both are necessary for any properly implemented bid searching program, we will detail here how both elements are represented within the linked list program. First the data structures will be detailed.

Within the linked list there are three distinct data structures which are used, all of which, in this case, are user-defined. The first user-defined data structure, or ‘struct’ as it is called in C++, is located at line 25 in file Lab\_3.cpp. The structure defined here, called ‘Bid’, is used to store the bid ID, the bid title, the fund which bid on the item, and the amount of the winning bid. This is a simple user defined structure simply composed of 4 data elements. While these four data elements could be stored in individual variables, the creation of a unique data structure allows for convenient individual instances to be created which contain all the information needed to reference and understand the key elements within a single winning bid.

The next user-defined data structure or ‘struct’ can be found on line 36 of the code file, and is titled ‘Node’. This data structure, while simple on the surface, is more advanced in and of the fact that it is a user-defined data structure that contains another user-defined data structure. The first element in each Node instance is an instance of the user-defined data structure Bid. Following this is a pointer variable which is used to contain the memory location of a subsequent Node – a key element to implementing a linked list.

Finally, the third user-defined data structure is actually implemented within a class, and begins on line 49 of the code file and continues through line 208. The linked list class is used to create a series of Node instances, all of which contain Bid instances, and which are all ‘linked’ through the pointer variables contained in the Node instances. By implementing both the Node and Bid data structures, the class creates a “list” of data elements which are all linked by pointer variables, which makes it possible to add new data elements to the list, remove data elements from the list, search through the data elements, sort the data elements, and print/display the data elements.

These capabilities are also implemented within the LinkedList class. Aside from the creation of the base linked list structure itself, algorithms are implemented which are used to append a Node instance to the end of the list, prepend a Node instance to the beginning of the list or create the initial Node instance within the list, remove a Node instance from within an existing list, search for a specific bid within the Nodes that exist in the list, and print out the information contained within the Bid instances implemented within the individual Nodes in the list.

All of these capabilities are necessary for the project at hand. It could be argued that a ‘simple’ linked list such as the one detailed here is not the best example to be used for this report. This developer begs to differ. The program detailed here contains key elements which simply must be understood in order to create the application needed. The user-defined Node and Bid structures are necessary to both contain the data which must be referenced for each bid, as well as enabled the capability to create more complex structures such as hash tables and binary search trees. The concepts executed within the LinkedList class also are essential for any developer to master if he or she hopes to implement the use of pointer variables that is necessary again in hash tables and binary search trees.

The implementation of displaying/printing bids, searching for bids, and adding and removing bids also relies heavily on the concepts which are implemented in the methods that are implemented in the LinkedList class. Any aspiring developer that wishes to master these concept would do well to study and master the methodologies that are used in the methods in this application.

In defense of the code written, it must be noted that the code is highly modular. The two data structures used within the LinkedList class are implemented outside of, and therefore independently of the LinkedList class itself. Furthermore, each of the methods used for appending, prepending, removing, searching, and printing/displaying bids are independent of one another. Nowhere within the code will you find multiple “verbs” being executed within a single method or function. This aids both in isolating problems within the code as well as debugging those problems once identified.

Furthermore, the codes is highly reusable. Take for example the Bid and Node structures. The best thing that can be said in regards to the reusability of the code used to implement these structures is that it has been used in several of code files provided in this portfolio! To be fair, however, the true test lies in analyzing the methods that lie within the LinkedList class.

Take, for example, the Append method. Let’s imagine a situation where the bids are stored in a user-defined structure that differs from the Bid struct implemented in this program. Loosely stated – no problem! The append function does not require a specific architecture within the Bid struct, it simply takes the Bid struct – in any form – and appends it to a Node instance. Say that the Bid struct is called a different name, and the instance of it created is no longer called ‘bid’. This also is no problem, and simply requires that the code be refactored so that ‘Bid’ and ‘bid’ are altered to match the new struct name and instance name. As far as the pointer variables are concerned, it is simply necessary that either the pointer variables created within any new program this code is ported into are named identically, or, in cases where this proves to be infeasible or illogical, the pointer variable names can be refactored to represent the existing names for pointer variables within the new program.

As far as annotations are concerned, the program referenced here has clear annotations throughout which explain the purpose and methodology behind the code. Even within the methods/algorithms themselves there is clear indication of the purpose of the code, and what the individual components achieve. This can help greatly with debugging and in cases where new developers must access the code an understand it in order to maintain and add to it.

**Conclusions**

The research undertaken so far has made it clear that an understanding of data structures and algorithms, and an appreciation of their capabilities and limitations is key for the success of our developers in this project. Speaking to data structures, there are a wide array of both native and user-defined data structures to choose from. The choice of which data structures to use, or how to use them in conjunction with one another, will have a great impact on the performance of our application. For one example, take vectors. Vectors are very useful, and in fact are used in the applications in the submitted portfolio for the initial loading of bids. Vectors are also more secure than other structures such as arrays, adding even more benefit to their use in applications (Lysecky, et al, 2015). However, searching through vectors can tax resources and be time consuming, especially when dealing with large data sets. Building something like a hash table would be much more effective and enable fast searching (Lysecky, et al, 2015). Even using hash tables, though, requires an understanding of how to wisely choose a size of the hash table. Simply using an integer to divide into in a modulus function that matches the number of digits in an ID number is likely to use up an inordinate amount of memory, simply wasting resources and amplifying the possibility of problems or crashes in the process.

Algorithms, as well, need careful consideration. First of all, complex data structures such as hash tables and binary search trees require the use of algorithms to add and remove data from the structure – they cannot be added to as simply and easily as native structures such as arrays and vectors. Secondly, algorithms are necessary for functions such as searching, printing, and sorting. Understanding the implications of using one methodology compared to another, though, is necessary in order to ensure acceptable performance. Especially in today’s world, where the size of data sets being used is growing very quickly, understanding how use of one algorithm or another will impact performance and use of resources is vital. One simple example, visited upon earlier, is the difference between selection sort and quick sort algorithms. While a selection sort algorithm may be more easily implemented, a quick sort algorithm would provide the same results, but with great savings in terms of time and processing resources (Lysecky, et al, 2015). While much has been learned by comparing the performance stats of different algorithms within the same application, it is strongly advised that resources be allocated to allow developers to do further analysis of methodologies for writing algorithms in order to ensure that the best possible performance is being achieved in our application.

Outside of the application at hand, I have personally benefited greatly from this research project. I am nearing completion of a BS in Data Analytics, and have found several examples of how understanding of data structures and algorithms will help me in my future endeavors. One simple example is learning how to create and check timers within programs. The simple act of reporting on time and clock ticks necessary to perform operations can be very useful in developing data science applications, as the heavy reliance upon large data sets means that performance is always a very key consideration. In situations where, for example, I may need to choose from algorithms already implemented in open-source packages such as those I often use in the R programming language, it can sometimes be challenging to determine which “pre-canned” algorithms are the best to use. By checking performance this way a determination can be made on which algorithms are using the least resources to provide similar results. Furthermore, an understanding of more complex data structures can be extremely helpful in situations where it is necessary to search and sort through data tables in order to analyze interesting instances or compare similarities or differences in populations. By learning to understand the architecture and uses for different data structures it ensures that going forward I will be able to make more educated choices about which data structures to be included in my projects, as well as to understand what processes and applications those data structures best lend themselves to.

**References**

Lysecky, R., Vahid, F., Lysecky, S., & Givargis, T. (2015). *Southern New Hampshire University CS 260 Academic Year 2016-2017: Data Structures Essentials and Programming in C*. Retrieved from https://zybooks.zyante.com/#/zybook/SNHUCS260AY16-17