**Final Project**

**Portfolio of Work and Reflection Paper**

T. M. Lang

Southern New Hampshire University

CS260: Data Structures and Algorithms

Professor Giles

20 June 2021

As a programmer at SNHU Software, data from the city of Nashville was provided consisting of a listing of the monthly sales of surplus property items[[1]](#footnote-1). Tasked with building a program that allows users to access information about the items in the listing various data structures and algorithms were explored. The data structures of vectors, hash tables, and trees were developed and the algorithms of search, sort, and hash/chaining.

The vector data structure is exemplified in the project VectorSorting.cpp from Module 4. In this program, a vector was used to hold the collection of bids. Vectors are ideal when the size is not known from the beginning as the vector size can increase as needed and is not fixed (BitDegree, 2019). In the program, the bids were loaded from a CSV file into the vector: vector<Bid> loadBids(string csvPath). Without knowledge of how many rows were in the CSV file when it was loaded, a vector was the preferred option to as it would be able to add all elements.

Hash tables are seen in the project file of the same name, HashTable.cpp, from Module 5. This data structure provides a specific location for each element in the array or vector identified by a key. In this coding project, the keys were created using a hash function to calculate the bid ID mod hash table size:

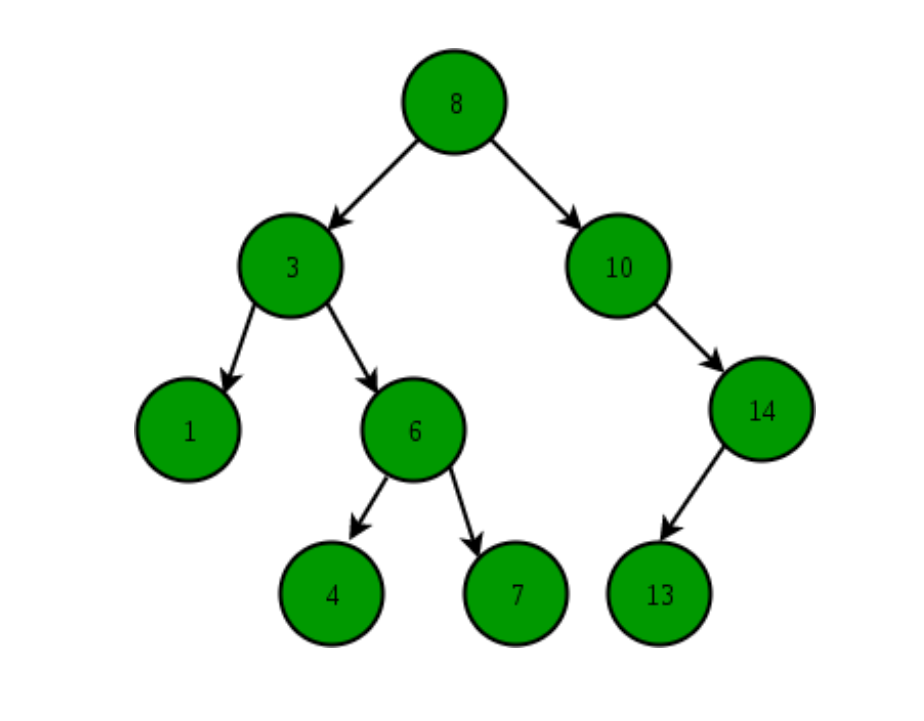
unsigned int HashTable::hash(int key) {

return key % tableSize;

}

This allows the number of keys to change as the tableSize changes, reducing the likelihood of collisions, where multiple elements have the same key.

The Module 6 project worked with tree structures and is seen in BinarySearchTree.cpp. In creating a binary search tree, as each data point is passed in, a decision is made to assign it to the left if it is smaller than or to the right if it is greater than the current node. This continues down the tree until this new node is added. The outcome is what appears to be an upside-down tree, as can be seen with an example by GeeksforGeeks (2020), where each node follows the same pattern of left if smaller, right if larger:



The code to accomplish the creation of a binary search tree in this project follows the same pattern:

if (node->bid.bidId.compare(bid.bidId) > 0) {

if (node->left == nullptr) {

node->left = new Node(bid);

}

else {

this->addNode(node->left, bid);

}

}

else {

if (node->right == nullptr) {

node->right = new Node(bid);

}

else {

this->addNode(node->right, bid);

Values of the node are compared to the value of the bid ID, continuing to the left if it is smaller and to the right if it is larger. At the point where there is no longer a node to compare, the new node is added to that location. Since the data is being sorted as it is added, this data structure creates an organized, sorted, easily searched set.

The algorithms used throughout this semester included search, sort, and hashing. The search function was utilized in several of the modules, but best executed in the Module 6 program, BinarySearchTree.cpp. As the data was loaded into a binary search tree data structure, the data was sorted and prepared to search. Utilizing a similar recursing of the tree as when adding a node, the comparison and matching of nodes was a fast task. The code below demonstrates how comparisons lead to the right and left of each node based on whether values are less than or greater than the node of comparison:

while (current != nullptr) {

if (current->bid.bidId.compare(bidId) == 0) {

return current->bid;

}

if (bidId.compare(current->bid.bidId) < 0) {

current = current->left;

}

else {

current = current->right;

}

}

As the tree is recursed, should the node not be matched it would quickly be returned as not being found since there would be no further locations that data point could be.

In the project VectorSorting.cpp from Module 4, two sorting algorithms were used: selection sort and quick sort, with quick sort being the more efficient algorithm. The algorithm partitioned the vector into two parts with the code calling a partition function: mid = partition(bids, begin, end);. This created a pivot point at the midpoint and sorted the elements into less than or greater than the pivot point, moving them to the left or right side of the pivot point, respectively. The sort function was then executed recursively on the section to the left of the partition using quickSort(bids, begin, mid); and on the section to the right of the partition- quickSort(bids, mid + 1, end); , with the result being a fully sorted vector. Visual representations of this sort occurring can be seen in videos, such as one by Sapientia University [AlgoRythmics] (2011), or graphics in educational materials, as shown in the Appendix (Techie Delight, n.d.). The graphics and videos identify the comparison of elements within each partition section as greater than or less than the identified element and moving to the left or right of it accordingly, sorting the vector into order.

In Module 5, HashTable.cpp is a program written utilizing the hash/chaining algorithm. The node where the data is stored is retrieved using the key as an address location for the node as a pointer: Node\* oldNode = &(nodes.at(key));. Using a search function that seeks a match between the key and the bid ID will find the location of the ID that is sought: if (node != nullptr && node->key != UINT\_MAX && node->bid.bidId.compare(bidId) == 0) and return the node for that match: return node->bid;. In large, unordered data sets, hashing is helpful to look up a specific key to find the data at that location.

The BinarySearchTree.cpp program was highly effective. It was able to search for a bid ID nearly instantaneously. The time calculated to find a specific bid using this program was 0. Since the tree is sorted as the file is loaded, recursing through the tree is significantly easier than matching a key, as would need to occur with a hash table and hash algorithm, as the hash table is not sorted.

This program was modularly composed in that each menu operation was separated and independent of others. The individual methods of Insert(), Remove(), and Search() were not overlapping and able to perform without the other. Making changes to any of these individual methods would not impact the functioning of the program.

The code is also reusable. According to Harvie (2018), modularity, in addition to factors such as cohesion and loose coupling, increases reusability. By virtue of it being modularly composed with different methods, each of these methods could be swapped out or replaced or used for other programs. This program also exhibits high cohesion since the individual methods are responsible for their specific functions, but all the functions are related to each other, such as searching, removing, and inserting. As well, the loose coupling is exhibited by the functioning of the different methods independently of each other.

The code is thoroughly annotated, though not excessively so. Should an outside person look at the code, they should have an idea of what the code was written to do and how it is expected to perform. An example of this is the removeNode method:

Node\* BinarySearchTree::removeNode(Node\* node, string bidId) {

// if tree is empty

if (node == nullptr) {

…

// recurse down left tree

if (bidId.compare(node->bid.bidId) < 0) {

…

// recurse down right tree

else if (bidId.compare(node->bid.bidId) > 0) {

…

else {

// no children - this is leaf node

if (node->left == nullptr && node->right == nullptr) {

…

// one child to the left

else if (node->left != nullptr && node->right == nullptr) {

…

// one child to the right

else if (node->right != nullptr && node->left == nullptr) {

…

// 2 children

else {

Each of the comments identifies the condition that is expected as with following along this recursion it may get confusion. With the annotation, it is clearer to see what the code is expected to do in those circumstances.

Data structures are an important consideration at the outset of program development. Understanding what the needs are for the data that will be handled can impact what structures would best manage the data and for the program to perform as necessary. As an example of this, having data that is growing unpredictably may benefit from being stored in a vector rather than an array. A tree may not handle data that can be unexpectedly unbalanced as it could become a linked list rather than a tree. A hash table with many collisions may be ineffective, or if keys do not need to be looked up it may not be the preferred data structure. Knowing what will be done with the data will help shape the data structure that is a best fit in that program.

Pairing the algorithm with the data structure is an important consideration as some are much more effective when working together. Certain data structures combined with some algorithms would be unnecessary, such as a sort algorithm with a tree structure. Knowing what the program needs to be capable of doing will help identify which algorithms are important to incorporate. In some cases, a linked list may suffice as a data structure with simple search functions that traverse the list. For other data sets, it may be more helpful to have a hash table with a hashing function that is able to hold data in labeled containers that can be accessed directly. The algorithmic needs will be dependent on the data structure and what will be the best use of the data structure. As well, the data structure will be chosen based on the functioning needs of the program, which may dictate a certain algorithm that would best be applied.

I frequently use different sorting algorithms in my daily life, from my household budget to prioritizing tasks that I need to accomplish. Creating efficiencies in daily life using data structures and algorithms is possible. A grocery list as a linked list sorted by aisles would be one way to use a sorted linear data structure.

For professional ambitions, in thinking of creating a database of patients, or using an existing database of patients, a hash table would likely be the structure I would choose. With a hashing algorithm, the creation of a key based on the patient ID would be able to be made and that key searched for rapidly. It would seem this is the most appropriate data structure of the ones we have studied to manage a large, unsorted data set.

References

BitDegree. (2019, September 3). *Creation and Use of C++ Vector*. https://www.bitdegree.org/learn/c-plus-plus-vector

GeeksforGeeks. (2018, January 6). *Advantages of BST over Hash Table*. https://www.geeksforgeeks.org/advantages-of-bst-over-hash-table/

GeeksforGeeks. (2020, February 14). *Binary Search Tree*. https://www.geeksforgeeks.org/binary-search-tree-data-structure/

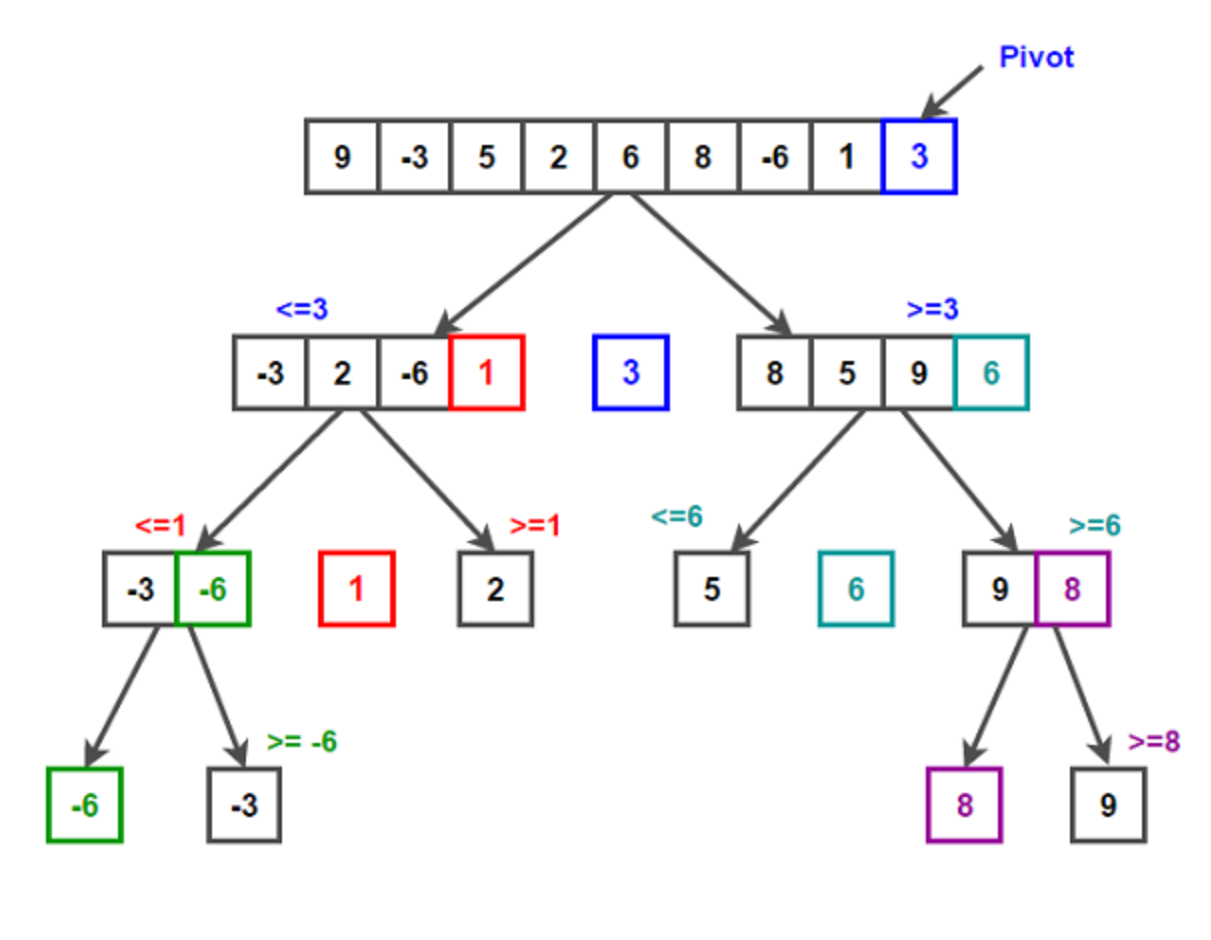
Harvie, L. (2018, October 14). *4 Ways to Make Your Code More Reusable - Lance Harvie*. Medium. https://medium.com/@lanceharvieruntime/4-ways-to-make-your-code-more-reusable-bc20889c1e4

Sapientia University [AlgoRythmics]. (2011, May 2). *Quick-sort with Hungarian (Küküllőmenti legényes) folk dance* [Video]. YouTube. https://www.youtube.com/watch?v=ywWBy6J5gz8&feature=youtu.be

Techie Delight. (n.d.). *Quicksort Algorithm – C++, Java, and Python Implementation*. Retrieved June 21, 2021, from https://www.techiedelight.com/quicksort/

**Appendix**

Quick Sort



Visualization of quick sort algorithm. (Techie Delight, n.d.)

1. https://data.nashville.gov/Business-Development-Housing/eBid-Monthly-Sales/n54t-t7gg/data [↑](#footnote-ref-1)