Module 4-2: Hash Tables

Reflection

Chris Richards

[Christopher.richards4@snhu.edu](mailto:Christopher.richards4@snhu.edu)

In week four, we learned about Hash Tables. As we learned in the material this week, hash tables can be implemented using either arrays or vectors, though with ability to dynamically resize vectors, I image this would be the most common approach to a hash table implementation. A hash table can come in a variety of ways to handle collisions. A collision is the case in which a bucket (key) is already occupied in a hash table. The method of we used this week in handling collisions in our hash table is chaining. Chaining is a method of handling collisions, by which each bucket contains a linked list of nodes. If a bucket is occupied, the node is inserted in the next position in that bucket’s linked list.

In my implementation of the hash table, we have two constructors. There is a default constructor that takes no arguments and resizes the vector used to store nodes to the default table size specified in the hash table’s properties. The overridden constructor takes an int that specifies the size of the table. It then sets the hash table’s table size and resizes the vector to the size of the table. The hash table defines a hash function that takes an integer as a key and returns a hashed key. The hashing function performs a modulo operation to create the hashed key.

The insert function takes the bidId of a bid, hashes it, and looks to see if there is an entry in the hash table at this bucket. If there is already an entry, it appends this entry to the end of the list at that bucket. The search function takes a key as its argument and will create the hashed key from this argument. It then looks at the bucket associated with this hashed key. If the bid at that entry is the bid we are looking for, it returns that bid. Otherwise, it will keep looking through the list for that bid. If it gets to the end of the list at that bucket, the search function will return an empty bid, indicating that no bid was found for that key. The remove function similarly looks for a specified bid with the bidId passed as an argument, and if it finds it, the remove function will do one of two operations. If the bid found is the first bid in the bucket, it will set the current bid to the next bid in the bucket, delete the found bid, then set the node at that bucket to the new current node. If the current node was set to a null pointer, this effectively marks the bucket as empty. If the bid found is not the first bid found in the bucket, it will iterate through the list in the bucket. When it finds the node, it sets the current node to the next node’s next node and deletes the found node. If the node is not found, it simply does not delete any node and returns from the function.

Finally, the hash table implements a node to search for a bid within the hash table. This iterates over the nodes vector. For each bucket visited in the node vector, the search function will then iterate through the list at that bucket, printing each node.

One of the challenges of this assignment was taking careful consideration to not break a list within a bucket when removing the first bid in the list. I needed to make sure that the next node is set to the current bucket after removing the first bid. Otherwise, the bucket would show as empty, even if it was supposed to contain additional bids. One thing I would like to comment on regarding the hash table is the speed at which you can access items in the hash table. Worst case scenario, every item is in the same bucket, making the time complexity of finding a node O(N). The best case scenario is that the item you are looking for is the first item in a bucket, making the time complexity for finding that item O(1).