Project 1 hash table design

This project consists of two parts, one to simulate a hashtable implemented with the chaining technique, the other simulating hash table with double hashing technique. Here are all the classes and a list of files I have implemented.


**Chaining:**

For the ordered hash table test, I reused my linkedlist.cpp implementation from the previous project. The driver code resides in orderedhttest.cpp

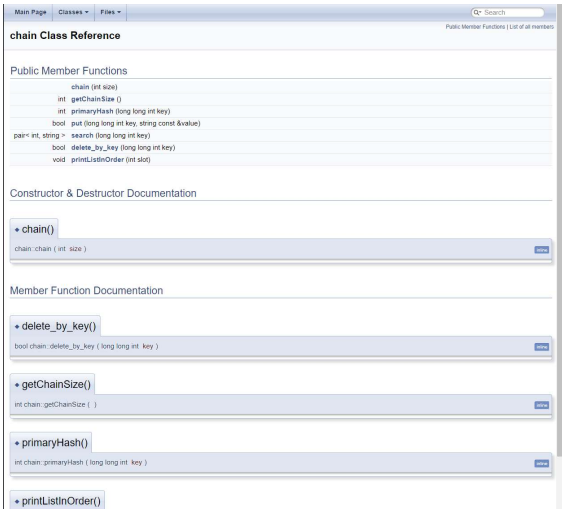
When that is run, an instance of chain is made and used to finish the rest of the test. The chain allocates a vector array of empty linkedlist head nodes in its constructor. The chain.cpp file also contains a couple of relevant methods that conducts operation on the hashtable and individual linkedlists referenced by each hash slot.

On insertion into the hashtable, we are storing the phone number (key) as a long long int type, mapping that to the customer's name as a string value. Before insertion, we must do a quick look up whether this key is already claimed. The look up is a fast operation since if the key we are attempting to enter already exists, it would reside in the same hash table slot. So a quick look of existing keys is O(n), where n is the number of elements in the linkedlist at the current slot. However, this linear time does not seem to matter as long as we don't have too many collisions, in other words, as long as our hash function is doing its job, we won't end up with all entries in the same linkedlist. Thus, the average time to iterate thru linkedlists at each slot is O(1). For deletion and search, this quick look up process is used as well, we need to check if the key we are searching / deleting exists in the hashtable. Thus, search and deletion are O(1) operations. Now to ensure that our individual linkedlists are sorted when being queried, we would need to implement a doublylinked list for quick comparisons and assignments. But since we only have a singly linkedlist to work with, I have implemented a *sortList(node* head)* method in the *linkedlist* class. When the print command is issued, we are expecting to see the linkedlist elements displayed in the order of the natural ordering of the keys and not the insertion order. The *sortList(node* head)* method uses the divide&conquer merge sort method. The runtime is O(nlogn) and space complexity being O(1) if addition to the call stack doesn't count, again n being the number of list elements at this particular slot. So instead of sorting the entire list every time on every insertion and deletion, the sort method is only called when we need to display all elements in order, thus not slowing down the other hash table operations.

**Double Hashing:**

Operations related to double hashing are initiated similarly from *openhttest.cpp* where all the input parsing and allocations are made. However, for double hashing, I have made class called *openAddress*. When initiated, a vector of type <entry> is allocated. This is the vector that will hold values of the hash table. For insert, find and delete operations, we compute the primary hash, and look to see if a collision occurs. If no collisions, we simply perform the operation on the slot, otherwise, we need to compute h(k, 2) and so on until no collisions are occurring. A detailed implementation can be seen in the *put()* method from *openAddress.cpp*

Lastly, I have compiled the UML diagram with a tool called “doxygen”.




It automatically scanned my project and created documentation of each class and method along with their interactions. I have placed the HTML files including the UML diagrams generated in the project folder. It can be opened from any .html file such as *ECE250/P1/html/index.html*