Tyler Hunt

113401590

Design Document

For my collision resolution strategy, I implemented double hashing. I had to encapsulate the templated types in another class, HashTableBucket to keep track of whether the bucket had been empty since the last re-hash. That is necessary for double-hashing since it can be hard to tell if it should keep offsetting by the secondary hash. The data I store in my buckets I also chose to encapsulate in another object called HashTableItem. It stored the key and value pairs for the table. By doing this, I could easily keep track of what key went to what value. I chose the load factor of .75 meaning that the size of the table will be 1.75 time the amount of data expected to be read in. To improve this, I could also add a minimum load factor, like .5, which would resize and re-hash the table any time the number of open spots in the table dropped below .5 times the size of the table. This would allow there to be a large buffer in the table so that collision would be less, and resolved quicker.

My merge and purge first updated the internal linked list, then re-hashed the table based on the list. The re-hashing resizes the table based on the load factor and size of internal linked list and hashes each key, value pair and places them in the buckets with double hashing. This is done for all merges, purges, and table resizes.

My primary hash is nothing special, and my secondary hash really isn't either. However, in order to try to fight the possibility of an infinite loop when offsetting, I made sure that the secondary hash did not divide, or was a multiple, of the hash table size. If it was, I incremented the hash until it was no longer a divisor or multiple of the table size. This was done so that in the event of an almost full table, it would be able to probe every bucket in the table instead of some pattern like every 2, or every 3, etc.