Seth Cram

CS121

* I began by copying and pasting code found in the lecture notes.
  + I later discovered we were supposed to get code from the sample programs folder, but found it to be the exact same.
  + My first problem arose because of this mishap. When compiled, the copied code kept throwing a stray ‘some number’ error.
  + After researching some, I discovered that the single quotes that I copied in were the wrong single quotes and wouldn’t allow the program to compile.
  + After deleting and rewriting the single quotes, it compiled without issue.
* I began by writing a function called void storeDictionary() without any argument.
  + Its purpose was to fulfill activity 3 and store the contents of both specified dictionaries in the hash table.
    - The first problem I ran into was an infinite loop.
      * Using the fstream library, I had read-in each word from a dictionary, but when it reached the end of the file it kept reading-in the last word.
      * Then I found the int eof() public member function of the fstream library that I used to close the file when the end of the file was reached.
    - The second problem was using the already established Linked List type Insert() function.
      * After reading a value, at the end of the while loop, the insert function was supposed to put each read-in word into the table.
      * The Insert() function kept throwing an error when I attempted to pass a string as the first argument, when it was supposed to be a char pointer.
      * I fixed this problem by changing the string that read in the words from the file from a string type to an array of characters that designated enough space for any word to be entered.
      * This furthered my understanding of how to code properly in base c
      * Also with the Insert function, I placed which dictionary a word came from as its definition as more of a placeholder for the actual definitions, since we didn’t have them.
* After storing the dictionary words in the hash table, I wrote the void numOfWordsPerBucket() function to output how many words were stored in each bucket.
  + I took code from the pre-established PrintHashTable() function to traverse through the hash table.
  + Modifying this code, instead of outputting each node’s contents, I incremented by one every time a word was passed by.
  + The problem I ran into here was that the function counted the total number of words in the entire hash table.
    - To combat this, I reset the counting variable to zero everytime a new bucket started, outputting the number of counted words beforehand.
* In accordance with activity 5, I wrote void minMaxInaBucket() to display the min and max number of values in the hash table a single bucket held.
  + I grabbed the same code from numOfWordPerBucket(), and added two new variables, int min and int max, to compare the number of words across buckets.
  + Everytime the while loop that counted the number of words in a bucket was less than min, that bucket now had the record for the minimum number of words in a bucket.
  + It was practically the same, except backwards for the max variable.
  + I learned that most code can just be adapted, without the need for rewriting entire segments of original code, to save time.
* Next up, int numOfProbesLookup() with a char pointer argument, followed a lot of the same logic that the original Lookup() function had.
  + Originally, I just attempted to increment the Lookup() function’s code to count and keep track of everytime a probe was used in the search for a word.
    - This resulted in a large chunk of unnecessary code being output, since the Lookup function was also called by the Insert() function.
    - To fix this problem, I wrote a new function altogether, copying over much of the original Lookup() function’s code.
  + This new function did away with returning the node at which the searched word was found, but rather, returned the number of nodes used to search for that word.
    - In accordance with activity 6, to search for at least ten words in a hash table and document how many probes were needed.
  + In the test file, to exemplify how this fulfilled activity 6, I simply called the function 10 times, searching for a different word each time, and added the results.
  + This way, any amount of words could be searched for and the number of nodes required would be outputted.
* Moving onto activity 7, in order to examine the effects the size of the hash table had for the varying number of hash table sizes, I needed to uncomment three different hash table sizes, one at a time, and look at the output.
  + I found no way to change the hash table size without editing and recompiling the code, since the hash table size was a global, constant value.
    - When attempting to change the hash table size with user input, the static linked list type hashTable[] array wouldn’t compile since it had to take a constant in its designation for size of the array.
  + Through this discovery, I learned that constants are fickle and often difficult to deal with in altering conditions.
* To account for how many empty buckets there are when the size of the hash table changed, I wrote the type int numOfBuckets() function.
  + At first, I just attempted to look through the output from numOfWordsPerBucket() and see how many buckets had zero, but with a larger hash table size this was impossible since the output code was too large and went offscreen.
  + This function is largely copied from the numOfBuckets() function.
  + I changed the counter variable’s name to probeCount and added another counter variable, bucketCount, to keep track of the number of empty buckets.
  + The bucketCount would only increment when the probeCount was equal to zero, which meant that the current bucket was empty.
  + A problem I ran into with this is that I initially wrote the if loop with a single equals, instead of a double equals.
    - This made me slow down and code more carefully, so I wouldn’t have to go back and search for more simple mistakes like this.
* Overall, I learned that a larger hash table makes for empty buckets that aren’t even filled, or more buckets with less items in them.
  + The larger a hash table gets, the fewer number of probes are required to search for an item in the buckets.
  + The max number of items in a bucket also decreases by a lot.
* Open hashing seems to be a faster strategy than linear collision resolution, when the hash table size is big enough for the appropriate number of items.
* All in all, it took me around 10 hours to design and implement my program.
  + Much of this time wasn’t spent actually coding, but understanding the pre-written hash table code that Mr. Bolden gave us.
  + In order to properly alter and implement new functions, I had to first understand what the original program did and why it worked.
  + I could’ve shortened this time by, ironically, slowing down, since a couple mistakes that took me awhile to discover and fix were very minor and simple errors.