

Data Structures and Algorithms

**Lab 5: Dictionary.h**

**The Scenario**

Having slain the dragon, you look out over the land as you emerge from the cave where the dragon once stood. Over the past weeks, you’ve trained and learned the ways of the list, the array, and associated data structures, and the fruits of your labor have finally paid off. The world enters a period of relative peace all thanks to you. You decide to turn to a quiet life of art after such an adventure—to become a master painter. The same tools that were your sword and shield become your new paintbrushes. Visual Studio is the canvas upon which your new life begins post-dragon.

As you put your brush to paper to begin, you decide that your first masterpiece must be the culmination of everything you’ve learned—it must combine lists, arrays, and use them in one large data structure—the **Dictionary**, or **Map.** Moreover, the **Hash Map** is what we will be dealing with in our lab today. It is a special type of Dictionary/Map that uses a **Hash Function** to determine how to access and store entries in the Dictionary.

Dictionaries and Hash Maps have some technical differences in the programming world which we won’t explore in too much detail here. It is safe to say, however, that a Dictionary is a data structure comprised of Key, Value pairs. If we are to compare the Dictionary data structure to a physical Dictionary book, the key would be the word and the definition would be the value. So a word, definition pair would be analogous to a key, value pair.

A Hash Map is a Dictionary where the key is put through a **Hash Function** to determine the storage location of the pair. Continuing with our Dictionary book example, we may have a Dictionary comprised of 3 volumes. A-E, F-K, and L-Z. The Hash Function would be the act of looking at the first letter of each word and determining which volume to store the word, definition pair in.

In a perfect world, each key, when put through the hash function, would give a unique location. In practice, however, it is rare to find a perfect hash function like this. As a result, most of the time, multiple keys can be stored in one bucket. i.e. if the hash function is “Add up the digits in the key”, 9, 45, 54, and 18 would all be in bucket number 9, while 47, 11, and 8 would be in buckets 11, 2, and 8 respectively. This is why the list is there—to maintain multiple entries for each location.

**What To Do…**

Open Dictionary.h. There will be instructions written in the comments on what is expected. Below is the gist of each function and variable.

***Variables:***

**key** Part of the Pair struct. Used in conjunction with mHashFunc to locate data.

**value** Also part of the Pair struct. The data intended to be accessed using the key.

**mTable** mTable is one of the more abstract variables you’ll encounter in this class. It is an array of lists. Each list is full of Pairs. So you have an array of lists of Pairs. Let’s break that down further.

mTable being an array of lists means that within each position (**bucket**) of the array, a list will be stored. Within each position of the list, a Pair will be stored. The list can have zero pairs or many pairs. The syntax of this may be difficult, so here is an example of a valid statement:

**ex. mTable[indexOfBucketYouWant].front( ).key** is valid.

**mNumBuckets** The number of buckets needed in mTable.

**mHashFunc** A pointer to the hash function. The hash function determines the position within mTable to store and retrieve data.

***Functions:***

**Pair Overloaded** ==You don’t have to worry about this one—it’s already done for you! :) Take a 5 second breather!

**Dictionary Constructor** Initializes everything for Dictionary using the parameters.

**Dictionary Destructor** Destroys the dynamically allocated memory of our program.

**Copy Constructor** Takes in a Dictionary and makes a deep copy of it in the current object.

**Dict. Overloaded** =Sets the invoking Dictionary equal to \_dict without any memory leaks and ensures it is a deep copy.

**Clear** Clears all internal data—every bucket should be empty. The array should still be the same size after this is done and the hash function should still be usable.

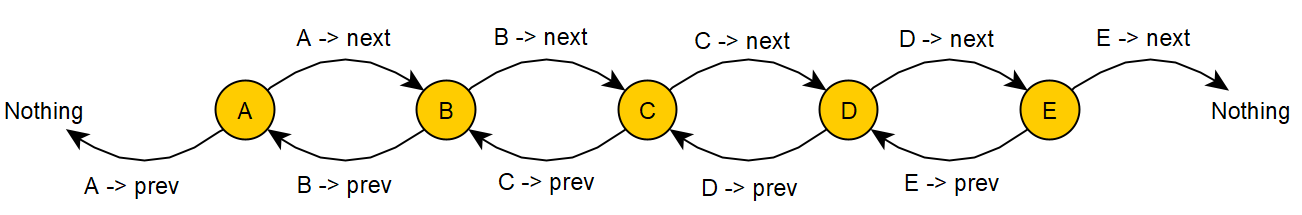
**Insert** Insert the key-value pair into the appropriate bucket. Remember how the hash function plays into this.

**Find** Accepts \_key, the key to search for. Using the key, finds an associated value if one is present and returns it. Otherwise returns a null pointer.

**Remove** Removes a value at a specified key. i.e. takes \_key, finds the value associated with it, and removes the pair. Returns True if an item was removed.

**Tips, Tricks, and Resources**

* Functions/Data Members available in the List class ( i.e. myList.pop\_back( ) ) can be found on the Cplusplus.com documentation for lists and arrays:
  + <http://www.cplusplus.com/reference/list/list/>
  + <http://www.cplusplus.com/reference/array/array/>
* In case you want to use it to visualize what’s going on under-the-hood in C++’s list class, here is the illustration of DLists from a previous lab’s handout:



* Searching on Google Images for Hash Map may help you visualize it if things get too muddled.

**Plagiarism**

Plagiarism and Academic Dishonesty are considered a **very** serious offense in this class and can have a range of consequences including suspension, and in very serious cases, expulsion. If you either share your code or copy someone else’s code, you will be given a **0** on your lab and can face further disciplinary action.

In other words, don’t cheat please!