Lab 12: Dictionary

1.1 Information

Topics: Dictionary (aka Hash Table / Associative Array)

Turn in: All source files (.cpp and .hpp).

Starter files: Download on GitHub or D2L.

```
Lab 12 - Dictionary/
 ├── lab12_main.cpp
 ├── lab12_StudentTable.hpp
 ├── lab12_StudentTable.cpp
 └── students.csv ...  Data input file
```

Contents

1.2 About: Dictionaries

1.2.1 Key-value pairs

If you're familiar with other programming lanugages like Python, Lua, PHP, or JavaScript, or have used the map STL structure in C++, you might be familiar with the idea of storing data in an array, but having a **key** map to some value, rather than just an **index**.

Because of the way a key is associated with a value in the Dictionary, these structures are often called **associative arrays**. They are sometimes also called **hash tables**, because they use a hashing function to achieve the key-value pair functionality, or even a **map**, because they map a key to a value.

Key vs. Index?
The **index** of an element in an array refers to its position in the array. If the array is of length n, then valid indices are 0 through $n - 1$.

A **key** is similar to an index, in that you use the key to find an element of the array, but the key can be *any data-type*. For example, if you had an alphanumeric Employee ID, you could use that as a key to point to some Employee element in an array.

Since the key of a dictionary can be of any data-type, and the value of a dictionary can also be of any data-type, it can be a useful structure to give some more meaningful data (than just *index/position*) as a way to map some value.

1.2.2 Dictionary basics

A Dictionary uses an array as its underlying structure. This allows us to access elements of the array with a $O(1)$ access time.

But if we have a key that points to a value (not an index), how do we still achieve the $O(1)$ access time?

We actually achieve this by creating a **Hash function**, which converts the key into an index via some sort of mathematical function.

As a simple example, let's say we have the following array:

0	1	2	3	...	26

And let's say the keys we're using are just letters of the alphabet. We could easily map these values to each number. The letter 'A' has an ASCII code of 65, 'B' is 66, and so on, so we could essentially create a Hash function like this:

```
int Hash( char key )
{
    return int( key ) - 65;
}
```

Then if we called `Hash('A')`, we would get a value of 0. For `Hash('D')`, we would get a value of 3.

1.2.3 Hash functions

Usually, we will want something a bit more sophisticated than just single letters as the key for our Dictionaries. With a hash function, there is some probability that two or more different keys will generate the same index. This is called a **collision**. There are several ways we can deal with collisions in a Dictionary.

Simple Hash function

For now we will think about integer keys, to make it simpler to understand the basic hashing concepts. Let's say a key can be any integer value of any length, and we have an array to fill up - let's say it's only length 10 for now.

0	1	2	3	4	5	6	7	8	9

And for our hash function, we start with a simple one where we make the key the index.

```
int HashA( int key )
{
    return key;
}
```

This would be OK as long as the keys were 0 through 9, but if the key is 10 or more, we will run into a problem. OK - we can handle this, let's just wrap around if we go past the bounds of the array. For example, if we have 10, we wrap around to 0. If we have 11, we wrap around to 1, and so on.

```
int HashB( int key )
{
    // SIZE = 10
    return key % SIZE;
}
```

This will work alright... now our keys can be any length, because the modulus operation will restrict the index to being between 0 and $SIZE$.

However, let's say all our keys end up being multiples of 5...

- `Table size: 10. Hash(key) { return key % 10;}`
- `Insert item at key 5: Index is 5`
- `Insert item at key 10: Index is 0`
- `Insert item at key 15: Index is 5      COLLISION!`
- `Insert item at key 20: Index is 0      COLLISION!`
- `Insert item at key 25: Index is 5      COLLISION!`
- `Insert item at key 30: Index is 0      COLLISION!`

We will talk about how to handle collisions in a little bit, but for now it is good to note that a general good design rule for Dictionaries is to make sure the size of your array is a **prime number**.

- `Table size: 13. Hash(key) { return key % 13;}`
- `Insert item at key 5: Index is 5`
- `Insert item at key 10: Index is 10`
- `Insert item at key 15: 15 % 13 = 2;      Index is 2`
- `Insert item at key 20: 20 % 13 = 7;      Index is 7`
- `Insert item at key 25: 25 % 13 = 12;      Index is 12`
- `Insert item at key 30: 30 % 13 = 4;      Index is 4`

See? Without doing any extra work, we can avoid collisions much easier by simply making sure our table (array) size is a prime number.

1.2.4 Collisions

With enough data, collisions are bound to happen at some point. A collision is when the Hash function generates an index for a key, but that index is already taken by some other data.

Linear probing

With linear probing, the solution to a collision is simply to go forward by one index and see if *that* position is free. If not, continue stepping forward by 1 until an available space is found.

Quadratic probing

With quadratic probing, we will keep steping forward until we find an available spot, just like with linear probing. However, the amount we step forward *by* changes each time we hit a collision on an insert.

- For the first collision, the index 1^2 position over is tested. If that causes a collision, then we try a second time.
- For the second collision, we try 2^2 positions over (from the original index we generated). If this causes a collision, then we try a third time.
- For the third collision, we try 3^2 positions over (from the original index we generated). And so on...

So, we square the amount of times we've hit a collision, and use that to offset the original index.

Double hashing

With double hashing, we have *two* hash functions. If the first Hash function returns an index that is already taken, then we use the **second Hash function** on the key to generate an offset. Then, our new index will be $Hash(key) + Hash2(key)\%$ TABLE_SIZE.

Clustering

Clustering is a problem that arises, particularly when using linear probing. If we're only stepping forward by 1 index every time there's a collision, then we tend to get a Dictionary table where all the elements are clustered together in groups...

0	1	2	3	4	5	6	7	8	9
	A	B	C	D			X	Y	Z

Because of this, the more full the Dictionary gets, the less efficient it becomes when using linear probing. Quadratic probing helps mitigate this, because it jumps forward by a quadratic amount when searching for an available index.

1.3 Lab specifications

1.3.1 About the program