**[CS250-Data-Structures](https://github.com/Rachels-Courses/CS250-Data-Structures)**/[Assignments](https://github.com/Rachels-Courses/CS250-Data-Structures/tree/2017-01-Spring/Assignments)/[Projects](https://github.com/Rachels-Courses/CS250-Data-Structures/tree/2017-01-Spring/Assignments/Projects)/**Project 03 / CL07 - Student Dictionary**/

**CL07-Project 3: Dictionary**

**Introduction**

For this lab, we will implement a Dictionary object. A dictionary is a **key-value pair** data-type, where you identify your value based on some key, such as...

student["r123m789"] = "Bob";

student["a887b321"] = "Notbob";

Dictionaries are implemented with arrays, and use a special **hash function** to generate an index for an item.

Because it's an array, and we have a function to get the index, we don't have to traverse the array to find an item. That is the perk of Dictionaries.

**Starter files**

* main.cpp
* StudentTable.hpp
* StudentTable.cpp
* students.csv

The program will load in the .csv file of students into the table, and after it is done, it will output the files:

* out\_double.txt
* out\_linear.txt
* out\_quadratic.txt

so you can see where the students get placed in the array using each collision method.

**Turn-In**

* Turn in your **StudentTable.cpp** and **StudentTable.hpp** files!

**Group work policy**

* This project is a **solo effort**.
* You can brainstorm with others, but you cannot code together, share code, etc.

**Dictionary basics**

If you're familiar with other programming languages 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 the **key** be something other than an integer index from *0* to *size-1*.

A key of a dictionary can be any data-type, and the value of a dictionary can be any data-type. This is the layer that other programmers see...

class Entry

{

public:

string key;

string value;

};

But within the dictionary, we store a simple array of Entry elements.

class Dictionary

{

/// ...

private:

Entry entries[100];

// ...

}

And our dictionary should have a **hashing function** that gives us an index for this simple array, given the **key** of our Entry.

**Hash Functions**

The Dictionary **key** can be any data type, but we need a way to *map* the key to an integer index. This is where the Hash function comes in.

If two keys generate the same **hash key**, then we have a collision and have to decide on a method to solve the problem. More on that later.

There are [different Hash function algorithms](https://en.wikipedia.org/wiki/Hash_function#Hash_function_algorithms) that can be used, with their own pros/cons.

**Integer keys**

If our keys are simple integers, then we could potentially just take *key modulus tableSize* to get an index. However, to reduce the likelihood of collisions, this works best if the table size is a *prime number*.

Let's say we have a table whose size is 13, and we are hashing student IDs, so to find the index in the table (the array), we will do simple modulus:

Student # 1068777

1068777 % 13 = 8

Student # 1013582

1068777 % 13 = 11

Student # 1087654

1087654 % 13 = 9

Student # 1079255

1079255 % 13 = 8 COLLISION!

This works fine but we can get collisions. How to resolve these is below.

**Collisions and solutions**

When two keys generate the same index via the hash function, we need a way to resolve the **collision**. A few common ways are...

**Linear probing**

With Linear probing, you just keep moving forward one index until you find a free slot in the array, and place the item there.

If a lot of items end up being stored in the array contiguously, one-after-another, then this could be slow. However, if all the items in the array are spread out, this might not be too bad.

Before collision:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| **Value** | - | - | - | - | - | - | - | - | 1068777 | 1087654 | - | 1068777 | - |

At this point, we have the three student IDs inserted. But when we want to insert *1079255*, it gives us an index of 8, and we have a collision.

Using linear probing, we would look at the next index, *9*, and see that it is also taken. So we move forward once more.

Index *10* is free, so student #1079255 would go into index 10 under the Linear probing strategy.

Of course, if we hit the end of the array, we wrap back around to the beginning!

**Quadratic probing**

With Quadratic probing, if there is a collision we change the amount we look forward each time...:

* First collision: Check *hashIndex + 2*
* Second collision: Check *hashIndex + 22*
* Third collision: Chekc *hashIndex + 23*
* etc.

So once again, before collision:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| **Value** | - | - | - | - | - | - | - | - | 1068777 | 1087654 | - | 1068777 | - |

* Hash of 1079255 is 8 - COLLISION
* Try 8+12, which is 9 - COLLISION
* Try 8+22, which is 12 - No collision

So with Quadratic probing, *1079255* would be inserted at index *12*.

**Double Hashing**

Another option is to have two hash functions: The first hash function, as usual, determines the *index*, which is used as a *starting location*. If there is a collision, then a second hash funtion generates the *amount of steps to take* past the *starting location*, to find a place to put the new data.

Let's say we have the following hash functions:

* Primary hash function: h1( key ) = key % 11
* Secondary hash function: h2( key ) = 7 - ( key % 7 )

So if our primary hash function gives us an index that results in a collision, we plug the key into the secondary hash function, and add the two results together.

If there continues being collisions, we add the result of h2 again, until we find an empty spot.

**Clustering**

One of the challenges of Dictionaries is the problem of too many items being grouped nearby together. If we're using linear probing, and hit a collision, and many items are back-to-back, we will end up moving forward one step at a time, checking the value at that position, and repeating until we find an empty space.

When elements are close together, this is known as **clustering**. As clusters form, the Dictionary gets less efficient and more time is spent finding an index for items.

**Project specifications**

For this project, you need to implement:

* void StudentTable::Insert( int studentId, const string& name )
* int StudentTable::HashFunction( int key )

as well as functions for:

* Linear probe
* Quadratic probe
* 2nd hash function

I have commented out prototypes that I used to implement these, and you can either write your own or use these.

It would probably be easiest to just start off and use Linear probe and make sure it works (no segfaults, no infinite loops) before working on the others.

**Grading rubric**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Assignment: |  | CS 250 Project 3: Student Dictionary |  | Grading breakdown | |
| Student name: |  |  |  |  | Spring 2017 |
|  |  |  |  |  |  |
| Grade key (each item out of 5 points) | | |  |  |  |
| 0 | Nothing attempted | 0.00% |  |  |  |
| 1 | Something attempted, but wrong | 20.00% |  |  |  |
| 2 | Incorrect, with a few good items | 40.00% |  |  |  |
| 3 | Almost correct, a few errors | 60.00% |  |  |  |
| 4 | Mostly correct, 1 or 2 small errors | 80.00% |  |  |  |
| 5 | Completely perfect | 100.00% |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Breakdown |  |  |  |  |  |
|  | Item | Score | Item weight % | Result | Notes |
| Basics | No syntax errors (it builds) | 0 | 50.00% | 0.00% |  |
|  | No logic errors (works as specified) | 0 | 3.00% | 0.00% |  |
|  | No run-time errors (doesn’t crash) | 0 | 5.00% | 0.00% |  |
|  | Clean code - consistent indentation, good variable/function/class names | 0 | 2.00% | 0.00% |  |
|  | No memory errors | 0 | 2.00% | 0.00% |  |
| Project | StudentTable::Insert function | 0 | 3.00% | 0.00% |  |
| specific | StudentTable::HashFunction function | 0 | 5.00% | 0.00% |  |
|  | Linear probing solution | 0 | 10.00% | 0.00% |  |
|  | Quadratic probing solution | 0 | 10.00% | 0.00% |  |
|  | Double hashing solution | 0 | 10.00% | 0.00% |  |
|  |  |  |  |  |  |
| Totals |  |  | 100.00% | 0.00% | |
|  |  |  |  |  |  |
| Additional notes |  |  |  |  |  |