assignment 2

Due: March 13, 2020

Assignment Objectives:

- Analyze existing code for efficiency
- Implement A hash table using Chaining for collision resolution
- Analyze and compare efficiencies

Overview

The Table class is a templated abstract base class. That is, it describes the functionalities of a Table but does not implement any part of it. It is up to classes derived from the Table to implement the Table itself. This section describes the interfaces (ie the operations) of a Table. The specifics of the implementation (ie is it a hash table or a sorted array? what collision resolution method is used etc.) is all done in the derived classes.

The records for this table consist of key-value pairs. The key is a c++ string. The value can be any data type that supports the following:

- assignment (=)
- instantiation without arguments (you can declare variables of TYPE, the value of which is undefined)

In the description below the word record refer to a key-value pair. The way you store it (whether you make a template struct called Record with a data member for key and value, or you store them separately in parallel structures is entirely up to you.

Here are the operations on Table:

```
Table();
```

Initializes a Table to an empty table

```
void update(const string& key, const TYPE& value);
```

This function is passed a key-value pair. If your table already has a record with a matching key, the record's value is replaced by the value passed to this function. If no record exists, a record with key-value pair is added to the table.

```
bool find(const string& key,TYPE& value );
```

This function is passed a key and a reference for passing back a found value. If your table contains a record with a matching key, the function returns true, and passes back the value from the record. If it does not find a record with a matching key, function returns false.

```
bool remove(const string& key);
```

This function is passed a key. If your table contains a record with a matching key, the record (both the key and the value) is removed from the table

---

```
int numRecords() const
```

This function returns the number of records in the table.

```
bool isEmpty() const
```

This function returns true if the table is empty, false otherwise

---

Aside from the above functions, all tables must have proper destructors, copy constructors/assignment operators, move constructor/operator that allow for resources to be properly managed.

Part 1: Analysis (14 marks)

Starter Files

- table.h

Description

Study the implementation of the SimpleTable class without making any modifications. A SimpleTable, uses a sorted array as its underlying data structure. Thus, all key/value pairs are stored in sorted order (according to their key). Using this implementation, determine the run time of the following functions/situation as they are currently implemented, where n is the amount of data (number of key/value pairs) currently in the table.:

- **update()** - if item does not exists so you need to add it as a new record and you don't need to grow the array()

- **update()** - if item does not exists so you need to add it as a new record and you need to grow the array()
- **update()** - if item does exists and you are just modifying what is there
- **find()** - if item is there
- **find()** - if item is not there
- **remove()** - if item is there
- **remove()** - if item is not there
- **numRecords()**
- **isEmpty()**
- copy constructor
- assignment operator
- move assignment operator
- move constructor
- destructor

NOTE: do not assume any of the functions are written well. They work in the sense that the table meets specs. They are not necessarily efficient.

Part 2: Discussion (6 marks)

Description

Suggest 3 improvements you could make to the code that will improve its efficiency. Write up your result in a wiki page.
What counts and what doesn't:

- You can't turn it into a different data structure all together... for example, "make it a hash table" would make it something different so that won't count. Fundamentally it must use a sorted array as the underlying data structure
- A process only counts once: "Do a selection sort instead of what is written in the copy constructor" and "Do this selection sort instead of what is written in the assignment operator" is just one suggestion not two. (note this suggestion is silly, and just used as an example)

Part 3: Implementation of hash table with chaining for collision resolution (25 marks)

Starter Files

- table.h
- mylist.h

Description

A hash table organizes the records as an array, where records are placed according to a hash value that you get from a hash function. Please use the hash function from the C++ STL. You can find more info about how to do it here:

http://en.cppreference.com/w/cpp/utility/hash

Here is a simple example of how to use the hash function with string keys:

```
#include <iostream>
#include <string>

//you need to include functional to use hash functions
#include <functional>

const int cap = 100;
int main(void)
{
    //this is just a string... any string can be a key
    std::string key = "this is my key";

    //declare an hash function object.  hashFunction is a variable but it is also a
    function
    std::hash<std::string> hashFunction;

    //you can now call hashFunction and it will return an unsigned whole number
    size_t hash = hashFunction(s);
    std::cout << hash << std::endl;

    //if you need to change it so that it is a proper index for a table with capacity
    of cap
    size_t idx = hash % cap;
    std::cout << idx << std::endl;
}
```

Aside from the hash function, you are **NOT** allowed to use anything else from the C++ Standard Library. However, you are allowed to use the linked lists you wrote for your labs 3/labs 4. Simply duplicate whichever list you wish to use into the file called mylist.h . The submitter will submit it as part of the submission process. I will want to see it as part of your a2 submission. You are also welcome add functionality to support what you need to the list that you use. However, you must place the list into the mylist.h file

For this derived class use the class name: ChainingTable

ChainingTable is a hash table that uses chaining as the collision resolution method. The chaining table has the following functionality:

```
ChainingTable(int maxExpected, int maxLoadFactor);
```

Your ChainingTable constructor should accept the following parameters:

- ***maxExpected*** is the maximum number of records your table is expected to hold. This number indicates how many "slots" the table can have.
- ***maxLoadFactor*** is the maximum load factor allowed before the table gets reconfigured to have twice as many slots. load factor is calculated as: (number of records stored)/(number of slots)

```
bool update(const string& key, const TYPE& value);
```

This function is passed a key-value pair. If your table already has a record with a matching key, the record's value is replaced by the value passed to this function. If no record exists, a record with key-value pair is added.

If the update() causes the load factor to go above the ***maxLoadFactor*** for the table, this function must reallocate the number of slots in the table to be double the current number of slots before the new key-value pair gets passed in.

```
bool find(const string& key,TYPE& value );
```

This function is passed a key and a reference for passing back a found value. If your table contains a record with a matching key, the function returns true, and passes back the value from the record. If it does not find a record with a matching key, function returns false.

```
bool remove(const string& key);
```

This function is passed a key. If your table contains a record with a matching key, the record (both the key and the value) is removed from the table

---

Aside from the above functions, all tables must have proper **destructors, copy constructors/assignment operators, move constructor/operator** that allow for resources to be properly managed.

Assignment submission

Submit a test verification

A2 Part 3 submitter command:

```
~catherine.lelung/submit dsa555-A2Part3
```

Submit your code

- push all your code into your team github repository for a2 (just into the master branch, do not use folders as only a2 files will be in the repo).
- Each member of the team must submit their own code under their own account (ie if you wrote it, and we look at the code we should see that it was you who submitted it not your

partner...can be done by looking hitting the blame button on a source file). Your teammates should not be pushing code into the repo on your behalf. Learning to use enough git to do this is good for you. If you don't love the command line use github desktop.

Late penalties

- up to 1 week late (and yes, break week does count), 10%
- 10% per day thereafter to a maximum of 50%

Resubmissions

With the test verification there is very rarely a need to have you resubmit your program. However, there are instances where the work may pass testing but still miss the point of the assignment entirely. In those cases, your work may get a grade of "resubmit" (usually indicated with a grade of -1). Here are some examples:

- doing a recursive function iteratively
- using a totally different data structure than specified (using linear probing for collision resolution instead of chaining for example)
- writing a data structure in a manner that clearly demonstrates a lack of understanding of how the data structure works

Any work that is resubmitted, will receive a 50% penalty. NOTE: Any work with a grade of resubmit is considered to be not completed.

Coding Grading Breaking

- Documentation - 20%
- Coding Style - 10%
- Memory Management - 20%
- Efficiency - 25%
- Completion and correctness of functions/functionality - 25% - Some aspects of some functions may not be tested by the tester. Sometimes this is because it simply isn't possible (like destructors for example). Thus, we reserve the right to read your program and take off marks for anything that does not work to spec.