Project 3

100 points

Due April 20th, 9:00pm via CSE Handin

This is not a team work, do not copy somebody else's work.

Assignment Overview

This assignment focuses on implementing a Hash Table using hashing with chaining and table doubling. You will design and implement the C++ program described below.

Assignment Deliverables

The deliverables for this assignment are the following files:

> **HashTable.h -** The source code of your solution.
> **problem2.pdf** - The PDF containing your answers to **Problem 2**.

Be sure to use the specified file names and to submit your files for grading **via the CSE handin system** before the project deadline.

Assignment Specifications

Your task will be to complete the methods list under **Problem 1 and Problem 2** below.

Assignment Notes

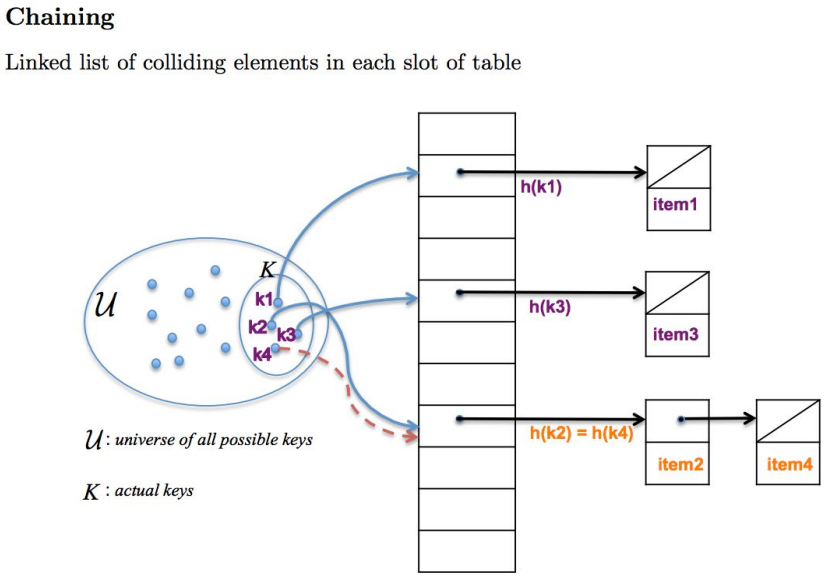
Points will be deducted if your solution has any warnings of type:

```
comparison between signed and unsigned integer expressions [-Wsign-compare]
```

The types specified in the functions are very important because of the size of the numbers used in this ADT. With integers/floating-points in C++, if the destination of a value is not large enough to hold the data in the value in the source, (`int f = uint64_t d`). If you ignore the warnings from the compiler because you used the incorrect types, **consider those warnings errors and fix them without losing precision.**

Hash Tables

Hash Tables are one of the most important ADT in Computer Science because of their promise of **constant time** operations and "constant" space complexity. We say "constant" space complexity because in reality most efficient implementations Hash Tables may actually be *slightly,* but not *overly,* wasteful when it comes to space.



https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf

The implementation used in this project uses *Chaining* and *Table Doubling/Halving*. Visti lecture notes for Chaining and please see the above links for a clear understanding of the resizing that will enable the desire performance.

When we **insert** a **key** into the Hash Table a few things happen.

1. The **key** is **pre-hashed**—converted to an integer representation for mapping.
2. Using the **hash function**, the **pre-hash** is mapped to one of the *m* slots (chains).
    a. Hashing functions like division method, multiplication method, etc.
3. If the Hash Table has as many items as it has chains, the table doubles in size.
    a. If doubled, all items in the Hash Table are put into the **hash function** once more in order to find which chain the item should be placed in.
4. Once the chain is found, the node is placed at the **tail** of the chain.

When we **remove** a **key** something from the Hash Table a few things happen.

1. The **key** is **pre-hashed**—converted to an integer representation for mapping.
2. Using the **hash function**, the **pre-hash** is mapped to one of the *m* slots (chains).
    a. Hashing functions like division method, multiplication method, etc.
3. Once the chain is found, it is traversed (linked list traversal) until a node is found with the matching **key** (the **hash** of the key, that is).
4. Once the node is found, the removal is made in-place.

Problem 1

Your job is to implement a Hash Table that uses table doubling and chaining to amortize most standard operations to **Θ(1)**. You have to implement the following public methods in your Hash Table:

1) `HashTable(size_t m)`
    a) Builds the empty Hash Table with a total `m` chains. All chains will be empty. `m` must be positive.
2) `~HashTable()`
    a) Destructor for the Hash Table ensures all nodes in every chain is deleted properly.
3) `void insert(string key, string value):`
    a) Adds the key value pair to the Hash Table.
    b) If the key already exists, the value is overwritten with `value`.
    c) The table is required to grow using the `Grow` function when $n > m$. where `n` is the number of elements in the Hash Table and `m` is the number of chains in the Hash Table. For example, a Hash Table of size 4 would need to grow after the 5th `insert` operation without any interweaving `remove` operations. The size will grow to $2m$ where m is the number of chains in the Hash Table currently. Only double after an `add` operation. A table of size **0** should double to size **1**.
4) `const string* get(string key) const:`
    a) Returns a pointer to the value of the key in the Hash Table if found, otherwise returns `nullptr`.
5) `void remove(string key):`
    a) Removes key-value pair (represented by the `ChainNode` class) from the Hash Table.
    b) The table is required to shrink using the `Shrink` function when the table is at least ¾ empty. This is true when $n \leq \frac{m}{4}$. The check is always done at the **end** of a `remove` operation. Only shrink after a `remove` operation. In your code, take the *ceiling* of $\frac{m}{4}$ and check if it is $\geq n$ if so, shrink**.**
6) `uint64_t HashFunction(string key):`
    a) Returns the index of the chain `key` should be mapped to according to the *division* method.
7) `uint64_t Grow():`
    a) Grows the Hash Table such that $m = 2m$
8) `uint64_t Shrink():`
    a) Shrinks the Hash Table such that $m = m/2$
9) `void Rehash(size_t originalSize):`
    a) Rehashes the items in the Hash Table to align with the current `HashFunction`.he See the secondary resources for intuition on why we must re-hash all our elements (and what rehash actually entails. *Hint:* the pre-hash of the values will never change.

Problem 2

Answer the following questions in a digital document and submit your answers in **a single** PDF, named **problem2.pdf**:

1. What changes would we need to make in order to decrease the probability of a collision (in our implementation)?
2. Why must we re-hash all of the items in the table when growing/shrinking?
3. Is this data structure constant time per operation? Why or why not? (hint: amortization)

Using the makefile and testing your work

Run the project on the CSE server *black (black.cse.msu.edu)* using the commands below.

Makefile commands:

`make`

This will compile the code in main.cpp and generate an executable name `ProjectHashTable`. Run this executable with the command `./ProjectHashTable`

`make test`

This will run the tests for the project. These tests are only a subset of all tests and only cover a few of the required methods. Passing all tests does not guarantee you will get all the points. These are only public tests. Your code will be tested against these tests and extra private tests.

Tracing segfaults:

Is the executable generated from main.cpp crashing?

1. In the terminal run the following (type the command and press return), notice the **or** which means either command will work:
   a. make
   b. gdb (or lldb) ./ProjectHashTable
   c. r
   d. where (if gdb)
   e. bt (if lldb)
2. The line your code crashed on should be indicated in the terminal.

Are the tests crashing?:

1. All steps are == except:
   a. make test
   b. gdb (or lldb) ./tests/tests

Please make sure to have your *tests* subfolder in your project folder in order to run the make test command successfully. If not, extract the folder from the project zip.

More information

For more information, see the following online video lecture or the following lecture notes:

CLRS(11.2, 11.3)

6.006 MIT Online Video Lecture / Lecture Notes

***Reference*:** This project was created by the following incredible students. *Ibrahim Ahmed* author of the project and multiple test cases. *Fatema Alsaleh and Scott Swarthout* authors of multiple test cases.