Cereal Killer
Programming Project 5
30 points

It is done. You, Cap'n Shaq Sorrow, have defeated the phantom menace of licorICE babies. With a *snap* of your fingers and some red-black tree magic, you eradicated half of the population types on the island immediately. The cold *crackles* in the air, and you *pop* the top off a perfectly chilled cream soda and take a long swig. The silence gives you time to reminisce, and you take a well-earned break from stacking the red and black corpses of your vanquished foes into a pretty bricked pattern surrounding your soon-to-be-opened strip mall, The *Dessert Oasis*.

From your water-logged pocket, you produce a *hash*brown[1] snack, one of the last from the wreckage that you found on the ship. You nibble idly on a corner as a flash of inspiration strikes you: your first establishment in the mall shall be an *Anglo-Saxon Castle*, a hallowed mecca of the best *hash*browns on the planet. You fondly remember the days where you met your friends Harold and Kumar at 2 AM and engaged in a glorious adventure to find the best *hash* there was to find.[2] Unfortunately, in your present situation, you have to *table* that idea.

You quickly scarf down three more hashbrowns and settle down to take a nap. Some time later, you hear the shriek of an asylum[3] of cuckoo birds bearing down FAST on your location carrying what look to be a large number of brown cardboard boxes. . . . of *Cocoa Puffs*? Do thine eyes deceive thee? Following them, is a toucan with a colorful beak sliding down a rainbow screaming about fruits, accompanied by a funny little man wearing green who seems to be obsessed with his lucky charms. We're going to leave *that* dude alone – sometimes leaving a wide berth is the most sensible thing to do. Frosty the Tiger and Winnie the Pooh are there too, carrying Honey Bunches of Oats by the boatload. Did that red-shirt bear just say *Cheerios* in a basic British accent? UGH. It's quite the Special group, 'K?

Finally, the **Bran Ambassador** himself arrives. . . . *Cap'n Crunch*. You must have summoned him by mistake with your snapping, crackling, and popping. Damn. Noob mistake. These Trix are supposed to be for kids. How did you fall for it? That's it. SINGLE COMBAT!!!! It briefly occurs to you that your hash-induced haze might be somewhat responsible for what is happening right now, but there's no way to know for sure, because right now, this experience is *Life*. It's time to be raisin hell.

The Synonym Toast Crunch is going to kill us all (or at the very least give us *irritable bowl syndrome*). The majorly Shredded (mini) Wheaties are marching on your location, and if you don't act soon, General Reese's peanut butter crisps may overwhelm us With-er-spoon. Bottom line: you need to *hash* all these cereals out of existence, because frankly, this whole protagonist/antagonist thing is getting old. There can only be one. This ain't no multigrain island.

Job Details. This project allows you to work in groups of 2 or 3, using the same Pair Programming mechanics as in earlier projects. In this project, you will be implementing several hash tables and resolving collisions using different methods. You will write a `virtual` class called `HashTable` from which you will inherit three different tables, each of which handles collisions in a different

[1] For the uninitiated, *hash* is 60s slang for weed, Mary J, the good stuff, jolly g, dope, grass, pot, broccoli, and roach. You're welcome.

[2] `https://www.youtube.com/clip/UgkxtlOST7meGYIuoXwn-YXwcdSHCuzoe_kN`

[3] And yes, a group of cuckoo birds is called an *asylum*.

way: `HashTableLinearProbing`, `HashTableChaining`, and `HashTableCuckoo`. The algorithms for each of these will be the same as the ones described in class. The description of all of the functions required (and their expected output) is included in the `hashtable.h` file that is available in the `P5` folder online.

For all of the hash tables, you may assume that values inserted into the tables are unique. In other words, you will never insert two copies of the same value in the table. The return values in `hashtable.h` reflect that behavior. Also notice that `hashtable.h` does not have many private data variables! If you think about this for a moment, you will realize that the reason is that each of the three collision types requires you to store different data structures (and possibly private helper functions). You will have to flesh out each of these classes for yourself based on your extensive knowledge of `C++`. And yes, you now have it. Believe.

Your task, ultimately, is to write each of these four classes, along with a menu to allow the user to choose select their probing method, and operation(s) to perform. The user will terminate when they are bored. Your program must dutifully serve their needs until that event occurs. Your program is allowed to be snarky and mean. No one said that good service is assumed.

You will need to provide any associated `.h` or `.cpp` files, a `makefile` to build your project, a clean `valgrindoutput.txt` file, and a `git` repository that contains well-documented code and commits for the entire process. For your reference, the valgrind command your program needs to come "clean" with is

```
valgrind --tool=memcheck --log-file=valgrindoutput.txt --leak-check=full ./blah
```

where `blah` is the obvious best name of the executable file. Detecting (and removing) memory leaks is a key component of your assessment on this project.

This project will proceed in phases, making sure that your code works at the conclusion of each phase. I strongly recommend that you make sure that each phase works correctly before moving on to the next phase. (That means both correct code and memory-leak free!) Although there are not many opportunities to lose memory, it is still something that you should be thinking about.

Phase 0 – Creating Load Files. As you execute this project, you're going to need to load your files. You may want to write either a short program or manually write some load files that match the format indicated below. For the sake of simplicity, I would recommend that your numbers not exceed 999. This will enable them to fit more easily on the screen. Beauty is a beast otherwise. Your load command in the menu (see Phase 1 for details) will ask for the file name and send that in to the appropriate data structure directly. Each `HashTable` will execute the `load` command. The format of the file is as follows, where in this project, each item is simply an integer in the range $[0, 1000)$:

```
<table_size>
<number of items in this file>
<item 1>
<item 2>
...
<item table_size>
```

Phase 1 – Menus. All this cereal talk has made you hungry. Create a tiered menu system for the Anglo-Saxon Castle to execute all of the algorithms mentioned above. At this phase, you won't be able to really use the menu areas, so you will want placeholder `cout` statements to indicate that your menu is working as it should. Here's what the first tier of the menu should look like:

```
Welcome to the Anglo-Saxon Castle
<insert pithy phrase here>

1) Linear Probing
2) Separate Chaining
3) Cuckoo Hashing
0) Quit

Please enter your choice:
**********************************
```

Once they make a choice, the second tier menu looks like this:

```
1) Load Table From File
2) Insert
3) Search
4) Delete
5) Print Hash Table
0) Quit

Please enter your choice: 3

Search (Enter Value): 69
**********************************
```

The second tier menu will repeat until they choose to `Quit` the program. While testing, it may be easier to make each menu item print the hash table after every operation to show the result. If you wish to later comment out that behavior so that it only prints when you tell it, I respect that decision. Frankly, users should learn to fend for themselves. Screw them.

After each `print` command, the table should be printed in a specialized format for each hash table. The details are in shown in each phase described below. Once you have written each command for each class, hook them back up to the menu as described above for true victory.

Phase 2 – Linear Probing. We, in an attempt to restrain ourselves, will not make a "that's what she said" joke at this juncture. Your task is to implement linear probing in a straightforward way in the class `HashTableLinearProbing`, inherited publically from `HashTable`. The hash function you will use is $h(x) = x \bmod n$, where n is the size of the hash table. For this project, $0 < n \leq 21$. The `print` command for linear probing (where $n = 11$) should look like this:

```
**********************************
[00]: 48
[01]: 5
[02]: 55
[03]:
[04]:
[05]: 41
[06]: 76
[07]:
[08]:
[09]: 40
[10]: 76
**********************************
```

For the sake of forward compatibility with the virtual class `HashTableCuckoo`, the private variable that stores the table size in `HashTable` is called m rather than n.

Phase 3 – Separate Chaining. In this phase, you will implement `HashTableChaining`, inherited publicly from `HashTable`. The hash function you will use is $h(x) = x \bmod n$, where n is the size of the hash table. For this project, $0 < n \leq 21$.

To write this class, you will need to use a linked list. Instead of using a modification of the list we built in the `CongaLine` class, we will instead use the *Standard Template Library (STL)*, a powerful template-based collection of reusable code that implements many common data structures and algorithms that are frequently used. The STL is the poster child for the value of accessible and reusable code, and uses many of the engineering tricks we have learned throughout the semester – inheritance and polymorphism, implementation-hiding, and templates.

Each data structure you typically think of (list, stack, queue, etc.) is called a *container* in the STL and it has associated member functions. Additionally, most containers have *iterators* (which are a lot like pointers) and allow you to manipulate STL container elements. Most commonly, you tend to think of them as variables that can access the "next" or "previous" element of a container. In this project, you will learn how to use an STL list. It's easy, so let's get to it!

The first thing you'll need to do is add `#include <list>` at the start of your header file. Then, you can create your table as an array of lists using the following declaration (and of course create the array in the constructor, just like any other data type).

```
list<int> * table;
```

If it is confusing you, just pretend that table is an array of `int` instead of an array of `list<int>`) and it will make sense again. You will also need to use an iterator to access a list. Given a list called `myFirstList`, here's how you would set up an iterator and print its contents. That complicated data type (`list<int>::iterator`) in front of `x` is the data type of the iterator `x` for an STL list.

```
for (list<int>::iterator x = myFirstList.begin(); x != myFirstList.end(); x++)
    cout << *x << " ";
cout << endl;
```

Many people use the `auto` keyword to automatically determine the data type instead of typing `list<int>::iterator`. *I do not recommend that you do it for this project, and since the project is relatively small, and the type is not that complex.* `auto` can be used in place of long and complicated types that are difficult (or in some cases) unmanageable to type by hand. They often save time, ease the code-writing process, and improve generic programming. There are some cons as well, namely that `auto` can make code hard to read because it hides explicit data types from the programmer and code reviewer.

Also, the compiler might assign a data type that is either more or less permissive than you would prefer, for example creating `double`s instead of `int`s when you need explicitly integer values. It might also fail to bind to a valid data type and require clarification at compile time. (In general, the question of whether to use `auto` isn't an automatic decision, and we'll let a later software engineering class teach you all the pros and cons!)

You may notice that the above code also had a few member functions that were useful, namely `begin()` and `end()`. Details of the full functionality of the STL list can be found here: `https://www.cplusplus.com/reference/list/list/`. The most likely functions you will need are `push_back`, `insert`, and `erase`. That's all there is to STL lists!

Finally, the `print` command for separate chaining (where $n = 11$) should look like this. Notice how all the arrows line up. `iomanip` baby!

```
**********************************
[00]:
[01]:
[02]:
[03]:
[04]:
[05]: 40 -> 5  -> 12
[06]: 76 -> 48 -> 55
[07]:
[08]:
[09]: 40
[10]: 76
**********************************
```

For the sake of forward compatibility with the virtual class `HashTableCuckoo`, the private variable that stores the table size in `HashTable` is called m rather than n.

Phase 4 – Cuckoo Hashing. In this phase, you will implement `HashTableCuckoo`, inherited publicly from `HashTable`. The hash functions you will use are $h_1(x) = x \bmod m$ and $h_2(x) = \lfloor \frac{x}{m} \rfloor \bmod m$, where $n = 2m$ is the size of the hash table. For this project, $0 < m \leq 21$. (Notice that said that $m \leq 21$, not n.) As a result, you may assume that n is even. Finally, you will also need to implement the function `rehash()`. Unlike what we did in class, we will rehash the data structure into a table where m is 3 larger than before, not to exceed 21. (If m would exceed 21, terminate the program instead with an appropriate message indicating what happened.) Here is an example of what the final `print` command for cuckoo hashing (where $m = 11$) should display:

```
**********************************
[00]: 48              [00]: 42
[01]: 5               [01]: 12
[02]: 55              [02]: 63
[03]:                 [03]: 4
[04]:                 [04]:
[05]: 41              [05]:
[06]: 76              [06]: 11
[07]:                 [07]: 7
[08]:                 [08]:
[09]: 40              [09]: 47
[10]: 76              [10]: 0
**********************************
```

1 Turn In Materials

You will need to share your GitHub repository for this code. You can submit a single repository for this project. Your repository should contain:

- All `.h` and `.cpp` code for your project, and NO `.o` or executable files.
- A makefile that correctly builds your project.
- A `valgrind` output file that shows the memory leaks that you have squashed.
- A printout of a hash table (for each collision method) after a successful `load` operation. Each load must have had collisions. You can pipe
- Submit the files that you used as part of your sample `load` operations.