Assignment 4: School of Hash Knocks

Pre-Assignment Notes

The **late penalty** is -10% per day late. After two days, late submissions won't be accepted.

This is an **individual assignment**, meaning you're not supposed to share or copy your solution/code from other students or the internet. There's still the class's Piazza if you want to ask questions to the group, or you can ask during lecture, send an email to the class's help account, or attend an office hour on Discord.

Note you can use any code shared by the instructor or anything from class or the tutorial videos linked on the course website. You *can* get outside help from guides and other people, just try to make sure they're teaching you how to solve the assignment, not writing the code for you. You won't learn anything that way, and they can't write your exam or midterm for you. If in doubt about whether something is plagiarism or not, don't be afraid to reach out and ask!

1. In Pursuit of the Perfect Hash

Hash tables are an important data structure, because they let us capture many of the benefits conferred by arrays without relying on the array's strict structure or having to remember an entry's location within the array to look it up again later. We're free to define anything as our key, so long as it uniquely identifies the entry we're looking for, which in the best case is like being able to search even a huge database in constant time.

The downside to hash tables is that this best case isn't guaranteed. Configuring the table requires a lot of work, matching the right hash function and compression function to the problem, figuring out the right number of buckets and load factor, deciding how to handle collisions, and more. Even if you make sound, justified decisions on all of these components, it's hard to be entirely sure – if any of your assumptions about the sort of data you'll be dealing with are off, it's easy for an unfavorable pattern to emerge that results in a load of collisions that drives down performance.

While becoming a true hash-master, capable of generating the optimal hash table setup for any set of constraints, is beyond the scope of this course, we should at least get in some practice in making better decisions. Part of that is learning how to measure the impact of different decisions on performance, which will be a key (no pun intended) part of today's assignment.

2. Assignment Outline

You're part of a team trying to build a system for managing student records at a school. Each student's file has a few basics, like their first name and their birth year. You've been tasked with developing a Hash Table to store these student records, hoping to keep the time for adding and removing close to constant by minimizing collisions through smart design decisions.

While your team's providing you with an Entry class for a student's record, they haven't chosen what part should be the key. That, along with other implementation questions, will have a significant impact on the performance of the system, but it also comes with some practical constraints. After all, whatever key you choose must be able to uniquely identify each student, while also being meaningful and practical – looking up someone's name is easy, but doesn't guarantee uniqueness, while looking up a key of everything stored in the entry would be very unique, but not very practical!

2.1 Basic Implementation

The first step is simply setting up your own Hash Table class. This should be your own implementation, rather than using the standard Java class Hashtable. It doesn't have to extend that one nor implement Maps (though if you think that would help, you may).

As for your design choices:

1. The Entry class you should use is provided at the end of this sub-section. You can't add variables, but you may add any helper methods or tweak any accessors that you need.
2. You may choose whatever variables in whatever combination you wish from the Entry to be your key. Be ready to try more than one!
3. Your choice of Hash Function will depend on your choice of key, and is one of the major factors of optimization. You might be able to use a key directly as a Hash Code, or else need to sum its components, or else devise a Polynomial Hash Code. If you're considering just using the hashCode() function, be sure to look closely at the requirements laid out at the start of this section.
4. For the Compression Function, use the Division method (which means there's nothing to optimize here).
5. For Collision-Handling, use Separate Chaining. While this could theoretically be a site of clever optimization, you will not be measuring the performance of whatever structure you choose to put in the buckets, so anything that simply and reliably produces the Entry from the requested key is fine. We'll be focusing more on reducing the number of collisions overall.
6. The Load Factor should be set to 0.9, though you'll also want to keep track of just how many collisions have happened and how full the structures in your buckets are getting.

The StudentEntry class (and a related ContactInfo class) is as follows:

```
class StudentEntry{
	String firstName;
	String lastName;
	String major;
	int birthYear;
	ContactInfo personalInfo;
	public StudentEntry(String firstNameIn, String lastNameIn, String majorIn, int birthYearIn, ContactInfo
contactInfoIn)
	{
		firstName = firstNameIn;
		lastName = lastNameIn;
		major = majorIn;
		birthYear = birthYearIn;
		personalInfo = contactInfoIn;
	}
}
class ContactInfo{
	int phoneNumber;
	String email;
```

```
    public ContactInfo(int phoneNumberIn, String emailIn)
    {
        phoneNumber = phoneNumberIn;
        email = emailIn;
    }
}
```

2.2 Experimentation

Once your Hash Table is up and running, it's time to measure its performance. Our main interest will be reducing the number of collisions through your choice of key and hash function, which you'll be testing experimentally.

Write a main method that creates one of your hash tables and loads it with some initial student data. Your test method should print how many collisions were triggered, how many times the table's capacity was increased, how many buckets are free vs. how many are occupied, and what the highest number of entries that ended up in the same bucket is.

Run this test for at least three different keys. Each key must also run the test for at least two different hash functions (assuming at least two different hash functions are possible given the choice of key). Be sure to collect all the data printed to the screen, you'll need this output in the next sub-section.

To get you started, here is an initial sample of student data your team wants you to use for your tests:

1. John Smith, Computer Science, 1988, 5555555, jsmith@email.com
2. Howard Calico, Chemical Engineering, 1991, 8675309, hcalico@email.com
3. Yarah Bahri, Psychology, 1976, 6647665, ybahri3@email.com
4. Aveline Macon, Arts, 1991, 7181234, its_aveline@avelineworld.com
5. Pratik Gera, Psychology, 1996, 3232368, pgera@email.com
6. Usui Aimi, Psychology, 1996, 4623457, x_shadow_hunter_x03@gaming.com
7. Jane Smith, Computer Science, 1990, 5555555, j2smith@email.com
8. Vreni Spitz, Law, 1992, 5554444, vspitz@email.com
9. Pratik Comar, Business, 1991, 6645622, pcomar@email.com
10. Padma Gera, Arts, 1994, 5554544, pgera@realemail.org

The sample represents “real data” from the student body (well, within the narrative of our assignment), but they cannot say that it's truly representative, or that it would stay representative year over year.

A brief note about the bucket array, a crafty coder might conclude that we could drastically cut down on likely collisions by massively increasing the bucket array. This is true, but the tradeoff between improved runtime and wasted memory space should not be ignored. There's a reason the 0.9 load factor is a popular target for Hash Tables that implement Separate Chaining, so we're interested in a bucket array that's both evenly-distributed and close to full, without having to Rehash the whole table every time a new entry is added.

2.3 Discuss your Findings

Take your findings from the tests you ran in the previous sub-sections and provide a write-up on how different choices of key and Hash Function resulted in more or fewer collisions and higher or lower load factor. Include the results for each of your tests (a table or two might help you here) and highlight what gave you the best result in terms of fewest collisions and best-distributed load. You do not need to discover the optimal hash table configuration or rigorously, statistically prove your speculation, but you will be graded in part based on the strength of your analysis.

Include in your analysis for your chosen key a set of five unique entries who would all end up in the same bucket, and discuss what sort of patterns could emerge in the real world that could create this sort of undesirable result (like, say, a year where every student's last name starts with J, or most of the students are born in the same year and take the same Major).

This writeup only needs to be a few paragraphs long, and should definitely not be longer than a page. Be sure to conclude with your belief about what will make the best choice of key for this hash table and why.

3. Deliverables

All assignment submissions will be done through CourSys at https://courses.cs.sfu.ca/. For this assignment, that should include:

1. A .zip archive of your code, organized into a single Java project. To ensure compatibility, it's recommended to use IntelliJ while developing your code, then using the export function under File -> Export -> Project To .zip File. **Remember to include the test file required by 2.2!**
2. A .pdf of your written analysis for part 2.3. Please submit this as a single document in the separate available field on CourSys, rather than bundled within the code .zip file.

Be sure you tidy up your code before submitting! Check the Java code style guide posted on the course website, and do your best to provide helpful comments.