# Assignment 5
# Hashing
# 4/23/19

Submit ZIP file with code (with directories for Java packages as needed).

Reference: “Part C: Tree Data Structures – Hash Tables”

Overview:

- Data – 25 Points
- Prototype – 80 Points
- HashLookup – 35 Points
- Analysis – 35 Points

  **175 Points Total**

## Assignment Structure

Please put all of your classes in a package named “hash”. The two provided **interfaces** should also be put in this package. To submit, please ZIP up the hash directory.

## Materials Provided

There are two **interfaces** provided for you:

- LookupPrototype
- Lookup<K, V>

## General

- Do **not** use any Java library data structures
- Implementing “toString” on all your classes is helpful, esp. for debugging

## Problem 1 of 4 – Data

The purpose of this problem is to prep for the hashing structures by building **two or more classes** that contain different types of keys.

Example Data Classes (and dream up your own as well)

| Class | Possible Key |
|---|---|
| Employee | id (int) |
| Contact | phoneNumber (String or long) |
| Address | zip + street (String) |
| Citizen | ssn (int or long) |

**Key Patterns** – we'll want to generate "keys" (for the constructed objects) in different ways:

- Simple Sequential, e.g. 1, 2, 3, … , 9, 10
- Sequential with uniform steps, e.g. 10, 20, 30, … , 90, 100
- Sequential with uneven spacing e.g. 1, 9, 11, 19
- Random

Note that the keys must also be unique within a given data set.

Most of these keys (and thus objects) can be generated automatically (we probably would not want to manually code 10M object constructs). The "uneven spacing" pattern is probably a manual effort (i.e. and very small data set).

**Set Sizes** – We should be able to generate any sized set. Example set sizes:

- 1
- 10
- 100,000
- 1,000,000
- 10,000,000

At the end of this doc, there are some tips and suggestions available.

## Problem 2 of 4 – Prototype

This is a prototype that will help get us familiar with the hashing mechanisms.

This prototype-only will handle only very simplistic (naive) data. The prototype will not handle hash function collisions.

Specs:

- Class should be named "HashLookupPrototype".
- It should implement the "**LookupPrototype"** interface (provided for you)
- Core methods are described in the interface JAVA file
- It should provide a constructor that takes one parameter (an integer) that is the initial size of the data structure. (e.g. dictates how many "slots" are available).
- Assume keys ~~(type "K")~~ are unique.
- You may use Java's "hashCode" method (defined on Object) to compute the hashCode from the "key". The key would typically be a simple object like String, Integer, Long, etc but it need not be. If desired, you can also devise your own hash function.

## Problem 3 of 4 – HashLookup<K, V>

Generics:

- K: Lookup Key
- V: Lookup Value for a Given Key

Specs:

- Class should be named "HashLookup".
- It should implement the "**Lookup<K, V>"** interface (provided for you). Note that it uses generics. If you use any of your own data structures in the solution, they do not need to use generics.
- Core methods are described in the interface JAVA file
- It should be a rigorous solution – e.g. it should be able to handle any key pattern (as described for Problem #1). In other words, it should be able to handle hash function collisions (e.g. where the hash function on two keys results in the same hashCode).
- Assume keys (type "K") are unique.
- You may use Java's "hashCode" method (defined on Object) to compute the hashCode from the "key". The key would typically be a simple object like String, Integer, Long, etc but it need not be. If desired, you can also devise your own hash function.

## Problem 4 of 4 – Analysis

Prepare a short write-up

Submit TXT as PDF. Please add analysis doc to ZIP file when you submit.

Please copy these questions into analysis doc, and under each write out a short answer.

prototype (problem #2)

1. Which key patterns did the prototype handle reasonably well? Which key patterns caused it to fail?

HashLookup – problem #3

1. How does bucket size impact performance? At what average bucket size do you start to see a slowdown in performance?

2. Complete the following Table

Note that the keys should be unique and random for these performance measurements.

Performance of HashLookup Using Unique Random Keys

| Data Size | Lookup Speed<br>(Lookups Per Second) |
|---|---|
| 100,000 | |
| 1,000,000 | |
| 10,000,000 | |
| One Billion (may extrapolate) | |
| One Trillian (may extrapolate) | |

3. Based on your table in the previous question, what is Big-O for your HashLookup?

4. What was your solution for handling hash code collisions? Describe another possible solution (dream one up).

5. When looking for a key and you find a bucket of size > 1, how did you find the value corresponding to the key? Any pros/cons of your approach.