# CSCI 1933 Project 5

## Hash Table Implementation

### Due Date: December 14, 2020 before 11:55pm

## Instructions

Please read and understand these expectations thoroughly. Failure to follow these instructions could negatively impact your grade. Rules detailed in the course syllabus also apply but will not necessarily be repeated here.

- **Due:** The project is due on **Monday, December 14** by **11:55 PM**.
- **Identification:** Place you and your partner's x500 in a comment in all files you submit. For example, `//Written by shino012 and hoang159`.
- **Submission:** Submit a `zip` archive on Canvas containing all your `java` files. You are allowed to change or modify your submission, so submit early and often, and *verify that all your `.java` files are in the submission.*

  Failure to submit the correct files will result in a score of zero for all missing parts. Late submissions and submissions in an abnormal format (such as `.rar` or `.java`) will be penalized. Only submissions made via Canvas are acceptable.
- **Partners:** You may work alone or with *one* partner. **Failure to tell us who is your partner is indistinguishable from cheating and you will both receive a zero**. Ensure all code shared with your partner is private.
- **Code:** You must use the *EXACT* class and method signatures we ask for. This is because we use a program to evaluate your code (more on this later). Code that doesn't compile will receive a significant penalty. Code should be compatible with Java 11, which is installed on the CSE Labs computers.
- **README:** You must include a README.txt in your submission that contains the following information:
  - Group member's names and x500s
  - Contributions of each partner (if applicable)
  - How to compile and run your program
  - Any assumptions
  - Additional features that your project had (if applicable)
  - Any known bugs or defects in the program

  There is a 5-point deduction if no README is included as one of your project files.
- **Questions:** Questions related to the project can be discussed on Piazza in abstract. This relates to programming in Java, understanding the writeup, and topics covered in lecture and labs. **Do not post any code or solutions on the forum**. Do not e-mail the TAs your questions when they can be asked on Piazza.

- **Grading:** Grading will be done by the TAs, so please address grading problems to them *privately.*

> **IMPORTANT:** You are NOT permitted to use ANY built-in libraries, classes, etc. Double check that you have NO import statements in your code, except for those explicitly permitted.

## Code Style

Part of your grade will be decided based on the "code style" demonstrated by your programming. In general, all projects will involve a style component. This should not be intimidating, but it is fundamentally important. The following items represent "good" coding style:

- Use effective comments to document what important variables, functions, and sections of the code are for. In general, the TA should be able to understand your logic through the comments left in the code.

  Try to leave comments as you program, rather than adding them all in at the end. Comments should not feel like arbitrary busy work - they should be written assuming the reader is fluent in Java, yet has no idea how your program works or why you chose certain solutions.
- Use effective and standard indentation.
- Use descriptive names for variables. Use standard Java style for your names: `ClassName`, `functionName`, `variableName` for structures in your code, and `ClassName.java` for the file names.

Try to avoid the following stylistic problems:

- Missing or highly redundant, useless comments. `int a = 5; //Set a to be 5` is not helpful.
- Disorganized and messy files. Poor indentation of braces (`{` and `}`).
- Incoherent variable names. Names such as `m` and `numberOfIndicesToCount` are not useful. The former is too short to be descriptive, while the latter is much too descriptive and redundant.
- Slow functions. While some algorithms are more efficient than others, functions that are aggressively inefficient could be penalized even if they are otherwise correct. In general, functions ought to terminate in under 5 seconds for any reasonable input.

The programming exercises detailed in the following pages will both be evaluated for code style. This will not be strict – for example, one bad indent or one subjective variable name are hardly a problem. However, if your code seems careless or confusing, or if no significant effort was made to document the code, then points will be deducted.

In further projects we will continue to expect a reasonable attempt at documentation and style as detailed in this section. If you are confused about the style guide, please talk with a TA.

## Introduction

This project focuses specifically on hash tables and should not take much time to implement. You will implement hash table solutions to two common situations (general - unknown data, and specific - known data) with the goal of minimizing the number of key collisions. While Java and other languages all have implementations of some kind of hash table, in this project, you are required to write your own hash table and hash functions.

## Assumptions and Rules

- The keys that we will use for this project will all be `Strings` which are "tokens" (described next)
- For this project, a "token" will be defined as any sequence of characters (other than the white space characters: tab, space, newline) delimited by one or more white space characters. For example, the following are all tokens:
  -123.34
  computer
  and
  result.
  *hello
  abc)
  {}
  (
  x
  x,y
- Do NOT maintain duplicate entries of a key in your hash table. In other words, if a token occurs multiple times, only put it in the hash table once.
- Keep in mind: the idea behind hash functions is that they are O(1), so do not "hand" map keys to indices in order to reduce collisions (i.e. do not manually assign indexes for each key).

## What To Do

## 1 Build a Hash Table

Create a hash table class

```
public class HashTable<T>
```

and build a hash table of an "optimal" length that you choose (around 100, but not more than 150). The hash table should be an array of `NGen<T>` (recall `NGen<T>` is the generic linked list node class; very similar to the `Node` class used for linked lists in Project 3), and should use "chaining" (as discussed in lecture) to handle collided elements. You can use the provided `NGen.java` file.

For the general case, do not attempt to build a table that is large enough to contain all the data. Rather, it will be the goal of your hash function to evenly distribute the chains of collided elements over the length of your table.

For the specific hash table, you will want to minimize the length of the table without creating large numbers of collisions – but a few evenly distributed collisions are just fine.

To keep things simple, and since our purpose here is to develop good hash functions, the only public methods required for your Hash class are:

```
public void add(T item) // adds item to the hash table
public void display() // displays the hash table along with stats about
                      // the hash
public static void main() // main method. detailed below
```

You are free to write the constructor however you see fit. You will also write a `main()` method to run your general and specific case hash. You can add your own helper methods if you feel the need.

## 2 Reading Tokens

To make testing of your hash table quick and simple, it is best to start with a way to easily read symbols (or tokens) from a file. The file `TextScan.java` has been provided for you to use (or modify) in order to read tokens from an arbitrary file. The main() driver method in TextScan.java reads (and displays) all tokens found in the file specified on the command line when running TextScan. Try it on some arbitrary file–for example, itself. You may borrow TextScan.java and modify it to meet the requirements of this project, but if you do, be sure to properly credit the source within your program.

## 3 Display the Hash Table

It will be necessary to print out the contents of your hash table so that we can tell how well your hash functions are working. Write a method:

```
public void display()
```

that will display each location in your hash table along with the number of keys that "hashed" to that location. You should also report the length of the longest chain and the average length of the chains.

Example output with hash table of length 5 and 6 tokens:

```
0: 1
1: 0
2: 2
3: 3
4: 0
average collision length: 2
longest chain: 3
```

Average collision length is calculated as $\frac{\text{number of unique tokens}}{\text{number of non-empty indices}}$ assuming all tokens (without duplicates) are successfuly added to the hash table.

## 4 A General Hash Table

Write several hash functions:

```
private int hash1(T key)
private int hash2(T key)
private int hash3(T key)
...
```

that attempt to distribute tokens "evenly" across the hash table. Sample files have been provided for you to test your hash table and hash functions. (See proverbs, canterbury, gettysburg, that_bad.) You may also use files of your own. Note that the hash table does not always need to grow to handle larger files, but the hash function should distribute the collided values evenly across the hash table. Once you have a good hash function, keep it to yourself! A good hash function is the key to an effective hash table, and it can give you a competitive performance edge. However, in your program comments you should **explain how your hash function works.** There is no requirement to the number of hash functions you need to write, but if you have difficulty coming up with a good hash function, showing you tried different approaches to writing the hash function will earn partial credit.

## 5 A Specific Hash Table

In some cases, the data that will go into the hash table is known in advance. One such example of this is the reserved words for a programming language. In the file `keywords.txt` are the 50 reserved keywords in the Java programming language. With some effort, it should be possible to map all the keywords (without collision) to a hash table. But, the table ends up having to be very long in order to achieve nearly 0 collisions. Instead, find a hash function that does a good job of minimizing and evenly distributing the few collisions that will occur in a table that is not overly huge (Note for the 50 Java reserved words, you will need a table length that is probably well over 150 - even up to 500). Keep your hash function to yourself, but **explain how it works in your program comments.**

What is the smallest your hash table can be before the collisions start to collect on a single index? What happens when you size the table very near to the number of kewords? Can you keep the collisions somewhat evenly distriuted?

## 6 Performance

You will need to provide some metrics to determine how well your hash function(s) distribute keys over the table.

In the `display()` method, you will be printing out the length of the longest collision and the average length of collisions in the hash table.

For the general case, the average collision length should be <= 6 with our provided .txt files. An example of how to calculate average collision length is in Section 3.

For the specific case, the length of the longest collision should be 3

Do not expect a perfect hash function, but a good one will have very few unused locations, and no locations will have a chain of collided elements that is much longer than the others. We would like to see most chains of collided elements at about the same length.

## What To Turn In

Implement all six sections above in Java.

**You must have a main method that creates and displays the general case hash table for `gettysburg.txt` and the specific case hash table for `keywords.txt`.**

Run tests using the text files provided on different hash functions. (You may also run additional tests using other files. But, additional tests are not required.) Be sure to show the different hash functions you tried (together with their success at reducing collisions) as you arrived at your best one. We are NOT looking for perfection here. Rather, we are looking at finding out what makes a difference and what does not. Demonstrate the effect of different hash functions and table length. Does even, odd or prime number table lengths make a difference with the same hash function? Include good comments within your code. Include in your README file other comments and clarifications that may be helpful in grading.