CS3460: Data Structures

Lab 03: Hash Tables

Total Points: 20

Problems

Notes on Grading: Unless otherwise stated, all programs will receive input via `System.in` and will output solutions via `System.out`.

To simplify the grading process, all grading will be automated. When applicable, you will be provided with sample input/output files for testing. You can ensure that your program will receive full marks by testing it with these provided files.

```
$ java YourProgram < input.txt > output.txt
$ diff output.txt correct.txt
```

The first line executes the Java program, redirecting input from a file `input.txt` and writing the output to a file `output.txt`. The second line compares your program's output (now stored in `output.txt`) with the correct answer (stored in `correct.txt`). If these files match exactly, the `diff` program will print nothing. Otherwise, it will list the differences.

Important Note: You are not allowed to use any classes or code from the Java Collections library. While the classes defined in that library would not be an ideal fit for most of our tasks, the purpose of these assignments is to build these data structures from first principles. Programs which import any of these libraries will receive zero points.

Submission: Please submit the files `SpellChecker.java`, `StringSet.java`, and `Closest.java`. Please do not include any other files.

1. **SpellChecker (10 points):** In this assignment, we will build a rudimentary spell checker, one that will make simple recommendations when a word isn't found in our dictionary. We will build a set data structure using a hash table implementation, capable of storing all words from a provided dictionary. We can then query this dictionary for words.

   You have been provided files `StringSet.java` and `SpellChecker.java`, as well as a dictionary (`dictionary.txt`). The `StringSet` class contains an inner class, `Node`, that shouldn't need to be modified, but feel free to make any changes you think might help you.

   Step 1: Complete `StringSet.java`, so that it fully supports operations `insert(key)`, `find(key)`, and `print()`. You will also need to provide an implementation for `hash(key)`, which will hash an input String to a valid table index using a polynomial hash function like we discussed in class. You should evaluate that polynomial at any prime number. Look up a list of prime numbers and choose your favorite (mine is 7919 – please do not steal it, it is very special to me).

   Note that `StringSet.java` provides a constructor that allocates a table of size 100. If the number of elements inserted into this table reaches this size, we will start to get a lot of collisions, so your `insert(key)` function should expand the table to twice its original size and re-hash each of the elements into this new table. This can be implemented after the other functions are working, but if you do not implement the resize operation, your code might not be very efficient.

   Step 2: Complete `SpellChecker.java`. This contains the `main(...)` method which has already been provided for you. An incomplete `getSuggestions(word)` method is provided, which should recommend any word that is only one letter different[1] from the queried word. An example of this code running correctly can be seen below:

[1] Investigate the `StringBuilder` class in the Java standard library, including its `setCharAt()` method for doing this.

```
Loaded dictionary.txt
Enter word: coee
Could not find coee
Consider the following alternatives...
code
coke
cole
come
cone
cope
core
cove
```

```
Loaded dictionary.txt
Enter word: algorithm
algorithm is a valid word.
```

2. **Closest Pair of Points (10 points):** Computing the closest pair of points from a collection of n points takes $\Theta(n^2)$ naively, by checking each point against every other point and keeping track of the minimum distance seen. We would like to improve upon that solution.

   For this problem, you are not provided any helper code. You will be provided 1 million points via standard input, each described by an x and y value on the same line, each in the range (0, 0) to (1 000 000, 1 000 000). The distance can be computed via

   $$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

   Comparing all pairs of points is infeasible with 1 million points. Instead, we will divide the grid into $b \times b$ grids, each of size $\frac{1000000}{b} \times \frac{1000000}{b}$. Each point is then hashed into its specific grid based on its location, and need only to be compared to points in its grid and in the neighboring **8** grids. This reduces the total number of comparisons needed, provided we choose a good grid size. You can experiment yourself to find the best possible value of b.

   Each grid cell should contain a linked list of the points that hashed to that location. If you create a `Node` class to store the points, the two dimensional grid could be declared by writing:

```
1 Node[][] grid = new Node[b][b];
```

   Your program should follow the following steps:

   (a) Allocate a 2D array of grid cells, based on your particular choice of b.

   (b) Read the input file, hash each point to the corresponding grid cell.

   (c) For each point, compare it against all points in the 3x3 block of grid cells centered on that point's grid cell. Maintain the smallest distance you've seen.

   (d) Print minimum distance.

   Your program should be named `Closest.java`

   A file containing 1 million points has been provided to you in `points.txt`, which you can feed into your program using input redirection. The correct answer has been provided in `closest.txt`

```
$ java Closest < points.txt
```