CSCI 1933 Project 5

Hash Table Implementation

Due Date: May 4, 2020 before 11:55pm

Introduction

You may work alone or with one other person on this project. If working with another person, turn in a single submission with both names and account IDs on it. The project is due on Monday, May 4, at 11:55 p.m. with late days of May 5 and 6 according to the deductions on the course syllabus. Submit your project on Canvas.

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

- The keys that we will use for this project will all be Strings which are "tokens."
- 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.

What To Do

1 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 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.

2 Build a Hash Table

Build a hash table of an "optimal" length that you choose (around 100 for now). The hash table implementation should use "chaining" (as discussed in lecture) to handle collided elements.

The hash table can be an array of NGen<T>. (Recall NGen<T> is the generic linked list node class.) To keep things simple, and since our purpose here is to develop good hash functions, the only two public methods for your Hash class are:

```
public void add(T item) // adds item to the hash table
public void display() // displays the hash table or stats about the hash
                      // table in order to determine how well the hash
                      // function works
```

3 Display a 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 lengths of the longest and shortest chains (to make it easier to check the performance of larger tables).

4 Finding a "Good" Hash Function

Write several hash functions:

```
private int hash(T key)
```

that attempt to distribute tokens "evenly" across the hash table. This function will be called in the add() method, and should return an index of the NGen<T> array where the item will be placed. Sample files have been provided for you to test your hash table and hash functions. (See proverbs.txt, canterbury.txt, gettysburg.txt, and that_bad.txt). You may also use files of your own. Note that the hash table does not 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. IMPORTANT NOTE: Do NOT attempt to "size" your table to the amount of data. In general, you will not know the amount of data. So, make your table relatively small (a prime number around 100). Again, a good hash function will evenly distribute the keys across the length of the table.

5 Performance

When you have Step 4 working, write some code to let you know how well your hash function distributed the keys over the table. This could be as simple as displaying the number of items chained at each location in the hash table. 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.

6 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" 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. Try to do this. Again, keep your hash function to yourself, but explain how it works in your program comments. What is the smallest your hash table can be without having any collisions?

What To Turn In

Implement all six sections above in Java. Show the output from each of the last two sections along with your Java code. Run tests using the text files provided. (You may also run additional tests using other files. But, additional tests are not required.) Include good comments within your code. Include a short README file with other comments and clarifications that may be helpful in grading.

Honors

Write a third Hash class method and test it:

```
public T remove(T item) // removes item from the hash table (if present)
                        // returns null, if item is not there
```