Project 2

Chained Hashing

Jose Eduardo Gamboa Barraza
Universidad Autonoma de Guadalajara

Guadalajara, Jalisco

Searching and accesing information inside a list sometimes can be a big problem if we are talking about big lists, where n might equal a million or more, as in the case of google, imagine sweeping through a billion results in order to get the get one, that would take a lot, to solve this problem, a common solution is the use of hash tables, but this creates a new problem, collisions, where two or more data share the same key, a good solution to do this is using a balanced binary tree, in this case, an AVL Tree.

I. Introduction

The problem we face is to create a small directory, with contacts containing different attributes, search, edition, deletion and addition of contacts should be allowed.

To solve this, we could use a simple list, as maybe this directory won't be too extense, it could work fine, but lets imagine we had a whole city yellow page, maybe finding a single contact wouldn't be that fast.

So we will use a hash table, for exercise purposes, this hash will contain only 4 spaces, in a real life case, our table would be bigger to avoid more collisions than the necessary, and to handle this collisions, an AVL tree will be used.

So we have an O(1) time complexity from the hash table, in the best case where only one contact fell under a single hash space, and O(log n), from the AVL tree.

II. Description

This problem consist of a list of at least 20 contacts, each contact contains a name, last name, phone number, address, email and social networks, you should be able to add a contact to the list, edit the fields from any existing contacts, except for name and last name, delete a contact, and search a contact by name, last name or phone.

This list should be contained in a hash table of four spaces, to make collisions more often, and this collisions will be handled using a balanced binary search tree.

III. Solution

First a hash class was created to handle the hash table, the hash function and a wrapper for searching, deleting and inserting data from the tree class.

As just stated, a class for the AVL tree was created, which handles search, deletion and insertion for each search parameter (name, last name and phone number), a Contact class was create with all the needed attributes, which also functions as the nodes, having left and right children. Finally a simple menu was designed to interact with the user.

The hash functions used, depends on either the key being a number(for phone) or a string(for name and last name), first a simple value % hash size was used for numbers and len(value) % hash size for strings.

To test a second hash function and to see which one resulted in better results, for numbers we used a 1/m % size with a 1e^9 fixed point and for strings, we got the value for each letter using ord(c), adding all the values and then following the same path used for numbers.

Results may not vary from functions because of actual list size being only 20, differences may be more noticeable with a bigger list.

For the Balanced Tree, the AVL tree was used, the idea of this tree is to keep the height of the left and right children of every node to differ at most +1 or -1, height of the children being the longest path to a leaf, therefore when you have a node where the difference is 2 or -2 you have to re-balance the tree doing a rotation, either left or right.

IV. Results

Following, is a table with the results obtained for searching.

Table. 1.Search Time comparison between hash functions

	First Hash Functions*	Second Hash Functions*
Search by name		
Highest	5.93E-06	7.90E-06
Lowest	2.77E-06	1.54E-05
Average	3.85E-06	5.95E-06
Search by last name		
Highest	4.35E-06	8.30E-06
Lowest	2.47E-06	5.14E-06
Average	2.83E-06	5.81E-06
Search by phone		
Highest	5.53E-06	1.26E-05
Lowest	2.37E-06	3.16E-06
Average	2.63E-06	3.87E-06

*Hash functions explained in *III. Solutions*

As we can see average in general was higher for the second hash functions, as they are more complicated to calculate, however in a situation where the contact list and the hash table were larger, times for second functions would improve against first functions.

Results for insertion times is showed next.

Table. 2.Insertion Time comparison between hash functions

	First Hash Functions*	Second Hash Functions*
insertion by name		
Highest	5.19E-04	5.71E-04
Lowest	1.03E-05	1.19E-05
Average	4.83E-05	5.07E-05
insertion by last name		
Highest	8.19E-04	5.22E-04
Lowest	1.86E-05	1.26E-05
Average	7.14E-05	5.06E-05
insertion by phone		
Highest	5.76E-04	5.26E-04
Lowest	1.19E-05	9.48E-06
Average	5.45E-05	4.81E-05

*See table 1.

Average for second functions in this case seem better than first functions, expect for insertion by name, but we can see how average for last name is reduced by 2.08e-6.

Finally results for deletion are showed next.

Table. 3.Deletion Time comparison between hash functions

	First Hash Functions*	Second Hash Functions*
Deletion		
Highest	8.22E-05	1.02E-04
Lowest	3.75E-05	3.79E-05
Average	5.42E-05	5.41E-05

*See Table 1

From the table we can notice that event though, the highest time is worse in the second functions, the lowest and average are very similar to each other.