AMERICAN RIVER COLLEGE

C++ HASH FUNCTIONS: AN EXPLORATION

FINAL PAPER
SUBMITTED
TO DAVID FOX

CISP 430 DATA STRUCTURES

BY

BRYAN GOODRICH

SACRAMENTO, CALIFORNIA

5 DECEMBER 2013

The aim of this project was to write a C++ hash table API that would dovetail with other class objects designed by earlier projects or by the professor. In particular, the hash function would map a *Record* object representing a 2-tuple of (key, value) pairs by the key value. Both the key and values belong to *String* objects coded at the beginning of this course. Collisions were handled by dropping duplicate mappings to a "bucket" utilizing a doubly-linked list structure, previously coded. The performance of the utilized hash function and data structure would be compared with that of a binary search tree, also previously coded.

The data for this assignment was provided by the professor consisting of 500,000 Records. The performance of the binary search tree was compared to the hash table implementation by timing both the complete population of the tree and table (push) and the complete extraction of the records from the tree and table (pull). Additionally, a simulation of 1,000 trials of 100,000 random extractions was conducted since extraction was the most time consuming process, as will be shown below.

The binary search tree had no parameters that could vary its performance; however, the hash table could be adjusted by altering the table (array) size or the hash function used. After producing an additive hash function that performed dismally, a little research proved useful in understanding how "real" hash functions are often designed. Arash Partow's Website[1] proved particularly useful in categorizing the variety of hash functions to consider, while also providing code examples of well-known hash functions across a number of languages.

For this project, the most basic additive hash function was included to show how bad performance can be with a poorly designed mapping. A simplified form of an adjusted additive hash function originally used will also contrast the vast improvement it provided.[2] Additionally a multiplicative and rotative (bit shifting) hash function was used. For the purposes of this project, two similar algorithms from Partow's files was included—specifically, the rotative hash functions by Justin

1 Partow, Arash. "General Purpose Hash Function Algorithms." Arash Partow's Website. http://www.partow.net/programming/hashfunctions/ (accessed November 24, 2013).

2 The basic hash was the sum of the product of a prime (5) to each ASCII character in the key. The adjusted form was to weigh each product by its position in the String (5+j).

Sobel (JS) and the function by Brian Kernighan and Dennis Richie (BKDR). Lastly, as the hash table can be adjusted, three prime values were used: 5,521, 55,051, and 104,729.

To measure the performance of the hash table to the binary search tree, time was used. However, to gauge performance among the hash functions for given hash table sizes, the size of the buckets was measured in terms of distribution. These included the number of empty buckets (non-hashed indices), the minimum, maximum, average, and standard deviation of the bucket sizes.

Table Size: 5,521	**Push (seconds)**	**Pull (seconds)**
Binary Search Tree	5.4	5.5
Hash Table – Additive	1.1	596
Hash Table – Additive (adj.)	1.3	56
Hash Table – Multiplicative	1.2	28
Hash Table –Rotative	1.1	28
Hash Table – JS	1.2	28
Hash Table – BKDR	1.2	28
Table Size: 55,021	**Push (seconds)**	**Pull (seconds)**
Hash Table – Additive	1.1	580
Hash Table – Additive (adj.)	1.3	57
Hash Table – Multiplicative	1.1	3.8
Hash Table –Rotative	1.4	3.9
Hash Table – JS	1.1	3.8
Hash Table – BKDR	1.2	3.8
Table Size: 104,729	**Push (seconds)**	**Pull (seconds)**
Hash Table – Additive	1.3	(ignored)
Hash Table – Additive (adj.)	1.2	56
Hash Table – Multiplicative	1.5	2.6
Hash Table –Rotative	1.2	2.6
Hash Table – JS	1.2	2.6
Hash Table – BKDR	1.1	2.6

Comparison of Hash Function Speed for pushing ins and pulling out 500,000 Records.

Regardless of the hash function algorithm, they all performed equally well at pushing data into the hash table, having to do with the O(1) performance of an array lookup. However, the choice of hash

table size did clearly alter the speed at which data was pulled out of the table, with the basic additive algorithm taking upwards of 10 minutes, regardless of table size. The adjusted additive algorithm also did not improve as the table size changed, even though it improved the performance considerably. It was still a very bad performer. It was surprising to find that the multiplicative and rotative (including JS and BKDR) algorithms all performed virtually the same to 2 significant digits.

The performance based on table size is clearly a product of how the hash function distributes keys across the table indices and the ultimate lookup through the bucket lists. We see that a ten-fold increase between the small and medium sizes produced an 86% decrease in time and a drop of 66% from doubling medium to large. This is too small a sample size to draw any definitive conclusions.

Table Size: 5,521	**Zeros**	**Mean**	**Std. Dev.**	**Min**	**Max**
Additive	4,980	90	417	0	2,895
Additive (Adj.)	442	90	29	0	302
Multiplicative	0	90	5	60	125
Rotative	0	90	5	59	136
JS	0	90	5	59	127
BKDR	0	90	5	58	123
Table Size: 55,021	**Zeros**	**Mean**	**Std. Dev.**	**Min**	**Max**
Additive	54,510	9	135	0	2,895
Additive (Adj.)	49,546	9	40	0	302
Multiplicative	8	9	3	0	25
Rotative	6	9	3	0	45
JS	12	9	3	0	22
BKDR	7	9	3	0	24
Table Size: 104,729	**Zeros**	**Mean**	**Std. Dev.**	**Min**	**Max**
Additive	104,188	4	98	0	2,895
Additive (Adj.)	99,493	4	92	0	302
Multiplicative	926	4	2	0	16
Rotative	981	4	2	0	34
JS	875	4	2	0	17
BKDR	897	4	2	0	17

Comparison of Hash Table Collision Bucket Distribution.

Except for the Additive algorithms, all the hash functions performed equally well, producing a trivial amount (less than 1 percent of table size) of non-used indices with similar maximum values. The rotative algorithm produced slightly larger (almost double) maximums. All of them, including additive, maintained the expected mean equal to the number of records divided by the table size for an equitable distribution. However, the additive algorithms produced extremely highly skewed bucket distributions, missing 90 to 99 percent of the indices. In fact, it almost appears that the mapping made no use of the additional space from the expanded table size, whereas the other algorithms continued to utilize the extra space well, requiring on average, in the case of a 104,729 table size, four list values to inspect on every extraction.

To return to the comparison between hash functions and binary search tree, the multiplicative algorithm was used. A vector of Record indices was randomly shuffled with a seed set to the current time that the call was executed. The first 100,000 indicated records were extracted from the tree and table, respectively. This was repeated 1,000 times. On average, the binary tree took 633 ms to extract 100,000 records whereas the hash table took 600 ms. Given the performance in the full data set times, it would suggest that for smaller sizes, the two approaches are comparable; at scale, the hash table approach will execute faster.