Hashing Revisited

John Hamer
Department of Computer Science
University of Auckland
Auckland, New Zealand
J.Hamer@cs.auckland.ac.nz

ABSTRACT

Hashing is a singularly important technique for building efficient data structures. Unfortunately, the topic has been poorly understood historically, and recent developments in the practice of hashing have not yet found their way into textbooks.

This paper revisits the theory and practice of hashing in a modern light, relates our teaching experiences, and presents some suggestions for student exercises.

Categories and Subject Descriptors

E.2 [**Data Storage Representations**]: Hash table representations

General Terms

Algorithms, Measurement, Performance, Design, Theory

Keywords

General hashing

1. INTRODUCTION

Pick up just about any introductory data structure textbook and you will find the standard presentation of hashing. Expect to read about "chained hashing," "open-addressing," "primary clustering," "double hashing," and learn a variety of formula involving "load factor." A number of hash functions with sundry caveats may be included, and the chapter will conclude with remarks to the effect the hashing is usually very fast but has an unreliable worst case.

A radically different presentation is possible if a very, very good hash function is postulated. In such circumstances, performance is maintained even when the load factor is increased far beyond the conventional value of "somewhat less than 1." This change immediately obviates open addressing; the technique becomes an anachronism. Likewise, no place remains for "double hashing" or "universal hash functions."

ITiCSE'02, June 24–26, 2002, Aarhus, Denmark.

The core theory of hashing is reduced to the analysis of a single "general" hash function [3], and formulas that predict the worst-case and average performance of a hash table. The analysis draws directly on concepts from theoretical statistics, and provides an excellent opportunity for experimental laboratory work.

2. PRESENTATION

Hashing can be taught at any stage during a CS2-level data structures course. We have tended to cover the list data model first, since lists appear in the implementation of a chained hash table. However, the chains used in hashing are simple enough that they could be easily introduced at the same time.

2.1 The Associative Array

The approach we have taken in motivating the key ideas behind hashing is to first consider the privileged status of the array. Out of all the data structures, the array has one unbeatable property: it allows constant time access and update. Unfortunately, arrays only work with small integers as indices. Wouldn't it be great, we suggest to the students, to index an array with strings, like this:

```
office[ "John" ] = "S209";
```

The objective is thus set on a method for implementing associative arrays while retaining constant time operations.

2.2 An Extreme Space-Time Tradeoff

To introduce students to the idea of manipulating characters as numbers, an implausible implementation of an associative array is proposed at this point. The suggestion is made that a string can in fact be treated as a number, by considering each ASCII character to be a number in base 256. For example, the string `"John"` is viewed as the number $1,248,815,214$ (being $74 \times 256^3 + 111 \times 256^2 + 104 \times 256^1 + 110 \times 256^0$). All that remains is to declare an array with at least 1.2 billion entries, and our objective of an associative array is achieved!

This "strawman" solution is an extreme example of the perennial space-time tradeoff. As well as introducing the concept of characters as numbers, it serves to refine our objective. In order to exclude this solution, we must stipulate that space be used in proportion to the number of elements held in the structure, rather than the number of elements in the "universe" (in this case, the "universe" is all four-character strings). The Big-Oh notation can be used to concisely state this requirement. For example, "we seek a

technique for storing a small subset K from a universe U in $O(|K|)$ space, while retaining $O(1)$ access and update time."

2.3 Towards a Hash Function

With the problem now accurately characterized, we introduce the concept of a hash function. To keep the topic concrete, we talk about storing 1000 strings in an array with 1000 entries. Since the strings cannot be used directly (i.e., as base-256 numbers) to index into the array, we suggest computing an index from a string; i.e., we seek a function `int hash(String s)` that returns a suitable index for a given string `s`.

Given such a function, we could then write code like this:

```
office[ hash("John") ] = "S209";
```

This is called "perfect hashing." It is also impossible! The argument against the existence of such a function is a variation of the "pigeon hole principle," and we take this opportunity to introduce the students to this form of reasoning.

Theorem Such a function cannot be written.

PROOF. Pretend hash has successfully allocated a distinct index to 999 of the 1000 entries in the array. There is only one index left, and this index must be returned for the next string. If the hash function gets lucky, we can try again with a different string. One way or the other, hash loses. □

A perfect hash function can only succeed if it is written with explicit knowledge of the strings that are to be used, in which case it becomes trivial to write as a sequence of conditional string comparisons, with each successful match returning a consecutive integer. Perfect hashing is not really hashing at all, and turns out to offer only a minimal performance advantage over real hashing at best.

2.4 A Statistical Solution

Having shown than perfect hashing cannot work, we lower our expectations and look for to a solution that works within $O(1)$ bounds. This theme of working to within constant bounds is emphasized throughout the course as a standard design strategy. For example, it arises again when considering balanced binary tree algorithms.

Given the absence of any prior knowledge of the strings that will be passed to the hash function, we propose investigating what happens if the strings we pass to hash are from a "uniform random distribution."

The definition of hash(s) is to compute

```
s mod arraySize
```

This satisfies our space requirement, but there is no longer any guarantee that hash will return a unique index for each string.

We solve this problem by making each array entry (or "bin") hold a list of values (or "chain"). A put operation can insert the new value into the list, and get can search the list.

This works just fine provided none of the lists get too long. The statistical analysis provides some reassurance. We are interested in the probability that k out of n strings hash into the same bin. That will happen for a given bin if k strings hash into that bin (with probability $1/n$), and the remainder hash into a different bin (with probability $(1 - 1/n)$). The probability is given by the formula

$$\left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}$$

When n is reasonably large, this formula is approximated by $1/(ek!)$.

k	prob. k collisions
0	37%
1	37%
2	18%
3	6%
4	1.5%
5	0.3%

The analysis suggests that if we place 1000 random numbers into 1000 bins, about 37% of the bins will be empty, 37% will have exactly one number, 18% will contain two numbers, and we would consider ourselves unlucky if there was a single bin with as many as 6 numbers.

Students are usually assigned a laboratory exercise at this juncture, in which they perform this experiment using random numbers obtained from, e.g., `www.random.org`. They are asked to collect 1000 random numbers in the range 1–1000, and count (with the help of a program!) how many numbers are missed (empty slots), how many numbers occur once, etc. The results are then compared to the theoretical analysis.

2.5 Load factor

An inefficient use of space is apparent with this analysis, and we use this observation to introduce the concept of a "load factor."

With 37% empty space and a maximum chain length of 5, we can afford to use rather fewer bins. The average chain length will grow in proportion, but it turns out the maximum chain length grows much more slowly. This maximum chain length determines the worst-case performance of hashing, and is emphasized as a key property.

The formula for collisions can be generalized to handle a different number of elements (m) from slots (n). The first thing to go as the load factor ($\alpha = m/n$) increases is the empty bins. The probability that any particular bin will be empty after m elements have been added to a table of n bins is

$$\left(1 - \frac{1}{m}\right)^n = \left(1 - \frac{\alpha}{n}\right)^n \approx e^{-\alpha}$$

(e is the base of the natural logarithms).

When $\alpha = 2$ (a normal load factor), about 13% of the bins will be empty. When $\alpha = 5$ (still not a high factor), around 0.7% of the bins will be left empty. This is shown in figure 1.

The probability that k out of m strings fall into a particular slot in a table of n elements is

$$\left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{m-k} \binom{m}{k} \approx \frac{\alpha^k}{e^\alpha k!}$$

For load factor $\alpha = 2$ (e.g., 1000 bins and 2000 strings), we have

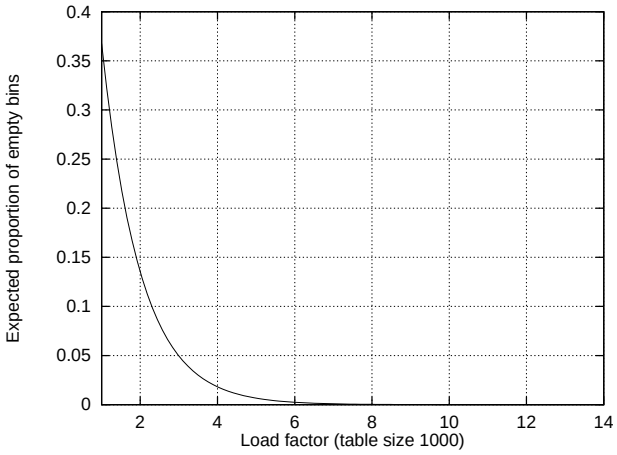

Figure 1: Empty bins by load factor

k	prob. k collisions
0	13%
1	27%
2	27%
3	18%
4	9%
5	3%
6	1%
7	0.3%

Now, about 13% of the bins will be empty, 95% have 4 or fewer numbers, and we would consider ourselves unlucky if there was a single bin with as many as 8 numbers.

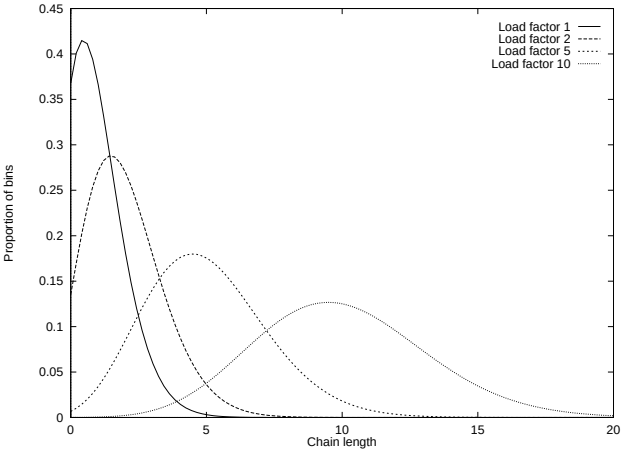

Figure 2: Chain length distribution by load factor

Figure 2 shows how the expected distribution of chain lengths changes as the load factor increases. The graph shows that, while the average chain length is equal to the load factor, tail of the distribution moves more slowly, growing from around 5 to around 20 as the load factor increases ten-fold.

2.6 Average probe length

For a given load factor, α, the expected number of nodes examined in a successful search is approximately $1 + \alpha/2$. For an unsuccessful search, the number is $\alpha/2$.

Students can use these formulas to design a hash table with certain performance characteristics. For example, if they are told to expect about two thousand keys, and allocate a hash table with 400 bins, then they load factor will be around 5. The formulas give an expectation of 2.5 probes (for successful or unsuccessful search) and around 3 unused hash table spaces (less than 1%).

Figure 3 shows the average search performance for successful and unsuccessful search by load factor, along with the "minimal" successful search that would be achieved with "perfect hashing." The graph show how "perfect" hashing (the middle curve) would not make a very large improvement—half a probe at best.

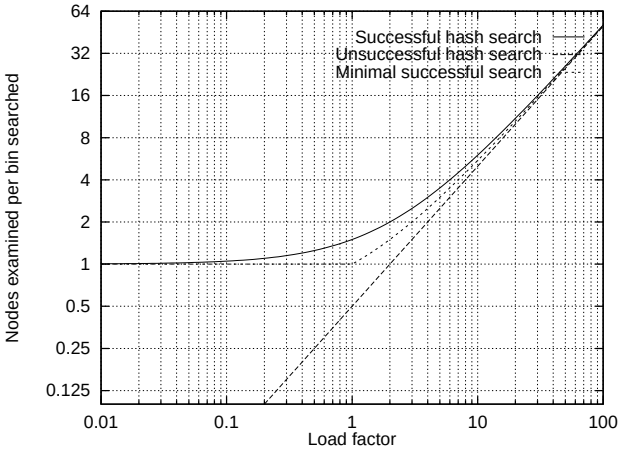

Figure 3: Average probe length by load factor

3. GENERAL HASH FUNCTIONS

The whole analysis of hashing presented here is predicated on input strings coming from a uniform random source. The expected maximal chain lengths increase dramatically if the input becomes skewed in any way, giving rise to the slippery slope of reducing the load factor and recovering the wasted space using open addressing.

Fortunately, there is no cause to worry. The "general hash functions" developed in the early 1990s, first by Pearson [2] and then Uzgalis [3], demonstrate that for all practical purposes a perfectly uniform random source can be produced from even highly skewed inputs.

To convince students of this counter-intuitive result, they are assigned a task of comparing the performance of a selection of hash functions against the theoretical performance of a uniform random source.

The experiment requires them to first assemble a variety of input sources, such as program identifiers, words from a dictionary, words from a work of prose, and a highly skewed source (e.g., strings consisting of 1s and 0s only). They then feed these input sets through each of the hash functions, and compare the resulting chain length distributions.

4. CONCLUSIONS

Nearly thirty years ago, a classic textbook from a prominent author made a claim that gave rise to a persistent myth:

"It is theoretically impossible to define a hash

function that creates random data from the non-random data in actual files." [1].

This myth is still alive and well today, despite compelling practical evidence to the contrary. One consequence of this myth is that, until recently, little work was done in developing good hash functions. Practitioners were content to use poor hash functions, and patch up the result with ad-hoc techniques like open addressing and double-hashing.

The existence of efficient general hash functions requires educators to radically change the way this important topic is taught. There are currently no textbooks in common use that do justice to the topic. We hope this paper will serve to stimulate educators to revisit hashing.

5. REFERENCES

[1] D. Knuth. *Sorting and Searching, The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[2] P. Pearson. Fast hashing of variable length text strings. *Communications of the ACM*, 33(6):677–680, June 1990.

[3] R. Uzgalis and M. Tong. Hashing myths. Technical Report 97, Department of Computer Science University of Auckland, July 1994.

APPENDIX

A. THE BUZHASH FUNCTION

```
public class BuzHash {
 static int[] buztbl = {
//- the table contains 256 ''random''
//- numbers, with the property that,
//- when viewed as 32 columns of bits
//- each column has 128 0s and 128 1s
  };

 public int hash( String s ) {
   int h = 1783936964; //-  randomly chosen
   for( int i=0; i < s.length( ); i++ )
      h = (h >>> 1)
        ^ buztbl[ s.charAt(i)%256 ];
   return h & 0x7fffffff; //- clears the sign bit
 }
}
```