OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Hashing And Its Applications

T. G. Lewis
Curtis R. Cook
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

87-60-8

Hashing And Its Applications

T. G. Lewis and Curtis R. Cook
Computer Science Department
Oregon State University
Corvallis, OR. 97331
(503)-754-3273

ABSTRACT

This tutorial discusses one of the oldest problems in computing: how to search and retrieve keyed information from a list in the least amount of time. Hashing -- a technique that mathematically converts a key into a storage address -- is one of the best methods of finding and retrieving information associated with a unique identifying key. We briefly survey techniques which have evolved over the past 25 years and then introduce more recent research results for extremely compact and fast methods based on perfect and minimal perfect hashing. Perfect and minimal perfect hashing is useful for rapid lookup of keywords in a compiler, spelling checkers, and database management systems. The results presented here show techniques for constructing long lists which can be searched in one memory reference.

KEYWORDS AND PHRASES

Key-to-address transformation, hash coding, hash table, scatter table, associative retrieval, associative memory, bucket hashing, perfect hashing, minimal perfect hashing, lookup, indexed retrieval.

INTRODUCTION

A classic problem in computer science is how to store information so that it can be searched efficiently. That is, given a set of records R,

$$R = \{ r_1, r_2, \ldots r_k \}$$

where $r_i = (K_i, D_i)$,

D_i is a data part, and

K_i is an identification part called the key.

We want to retrieve arbitrary records from R with the fewest possible number of comparisons of keys. R is typically stored as an array or array-like structure. For example, R might be stored as a direct access file when used in a database management system, or as a simple array when used to hold tabular data in a running program.

An Example: The Calendar Problem

It will be helpful to illustrate the techniques described in this paper by example. The *calendar problem* typifies the class of problems well suited for hashing. Suppose R is a calendar containing 3-letter abbreviations for the months of the year and their number of days:

R = { (JAN, 31), (FEB, 28), (MAR, 31), (APR, 30), (MAY, 31), (JUN, 30),
(JUL, 31), (AUG, 31), (SEP, 30), (OCT, 31), (NOV, 30), (DEC, 31) }

Obviously, r_i is the i^{th} month of the year and D_i is the number of days in the i^{th} month. Each record contains both key and data fields -- K_0= JAN, and D_0= 31, for example. To find how many days are in the month of August, R must be searched for matching key K_i=AUG and the corresponding value i=7 returned. The value D_7=31 tells how many days are in August. The problem, then, is to find ways to store R such that retrieval of the number of days in a specific month is fast and storage efficient.

Solutions to the Calendar Problem

The calendar problem can be solved in a number of ways. First, we can store R in an array and the array can be searched sequentially from top-to-bottom using a linear search algorithm. An average search would require (12 + 1)/2 = 6.5 comparisons to find an arbitrary month. This solution is memory efficient, but the search time grows linearly with the size of R -- doubling the size of the list also doubles the average time to locate an item. In many applications, linear growth in search time with increase in list size is not acceptable performance.

A second way to solve the calendar problem is to sort the list of months and use an ordered table searching algorithm, such as a binary search. Binary search requires an average of $\log_2(12) = 3.5$ comparisons to locate a record. While the binary search solution is a dramatic improvement over linear searching, it has two major drawbacks. First, the set R must be sorted, and sorting is in general very time-consuming even if it is done only once when R is entered into computer memory. Second, even though the average lookup time grows more slowly than does linear search, search time is still a function of the size of R.

A third way to solve the calendar problem is to use hashing. The records in R are stored in an array indexed from zero to some upper limit, and a mathematical function is used to convert each key into a unique array index. When searching the array for a certain month, the mathematical function transforms the search key into the corresponding array index so the desired month and day are retrieved in one comparison. The success of such a scheme depends on finding the appropriate mathematical function.

A *hash function* maps each key to a unique array index. There may be many hash functions which solve the calendar problem. One such hash function is given below.

h(KEY) = (first_letter(KEY) + second_letter(KEY)) **mod** 13

This hashing function maps each key of R into a corresponding number between 0 and 12 by summing together the ASCII code equivalent of the first and second letters in each key. The remainder of the sum is used to "scale" the index value to a number between 0 and 12 -- correponding to the index of each array element. Assuming the keys arrive in chronological order, the following mapping is obtained for all 12 months in R.

	KEY	h(KEY)	Comparisons
0	MAY	12	2
1	NOV	1	1
2	APR	2	1
3	JUN	3	1
4	JUL	3	2
5	OCT	3	3
6			
7	AUG	7	1
8	DEC	7	2
9	JAN	9	1
10	FEB	9	2
11	SEP	9	3
12	MAR	12	1

Figure 1

The hashing function must produce unique correspondences between key and index, but as shown in the calendar problem, uniqueness is not guaranteed by the mathematical function. This occurs because conversion of alphanumeric strings into relatively small integers often results in a *collision* - two or more different keys map into an identical integer. Keys that collide in this manner are called *synonyms*. For example, (JUN, JUL, OCT), (AUG, DEC), (JAN, FEB, SEP), and (MAR, MAY) collide in the hash table shown in Figure 1.

Various *offset rules* have been proposed to "correct" the problem caused by collisions. The simplest rule is called linear offset, because it finds unique index correspondences by searching forward in the table using a linear search.

We resolved collisons in Figure 1 by searching forward for the next empty slot in the array. Each forward search increases the number of comparisons needed to map the key into its index. Thus, imperfect mapping leads to an inefficiency in subsequent searches. The comparison column shown in Figure 1 gives the number of comparisons to locate each key in the hash table. The average number of comparisons is 1.67 -- a number which is an improvement over either linear or binary search.

The calendar problem illustrates how hashing can be used to advantage, but it also illustrates some of the difficulties inherent in the general problem of hashing. The purpose of this paper is to show how to apply hashing to various applications, indicate the problems to be overcome, and to introduce new techniques for solving some of the problems which have previously restricted hashing to a narrow class of problems.

In the next section we survey the classical literature on Hashing. The subsequent sections review traditional hashing techniques, including sections on selection of hashing functions, collision resolution, chaining, open addressing, overflow, and deletion. The final section introduces the more recent research on perfect hashing and gives a minimal perfect hashing algorithm which can be used for extremely fast searches on minimal storage tables.

KEY-TO-ADDRESS TRANSFORMATIONS

Hashing is a class of techniques for performing rapid associative searches of random or direct access memory. The term *hashing* first appeared in the mid-1960's in reference to a class of rapid table search techniques found useful in compiler construction, but hashing techniques have been used by programmers since the early days of the direct access disk drive(circa 1957) [PRIC 71, MAUL 75]. When used to quickly retreive a record from a disk drive, hashing is known as a *key-to-address transformation*, and when used to search a table, hashing is often called a *scatter-table* technique.

Hashing techniques are an alternative to "classical" search techniques because they,

(1). Permit arbitrarily fast retrievals independent of the size of the table, i.e. number of records in R,

(2). Rely on simple data structures, and therefore, are easy to implement and maintain, and

(3). Do not place constraints on the data, e.g. no ordering or special placement in the data structure is required.

Problems With Hashing

Hashing poses several problems. First, a suitable hashing function that distributes the keys in a uniform manner must be found, and if the hashing function permits collisions, an acceptable method must be devised to deal with them. Other problems are:

1. How to select the size of the hashing table and handle problems such as overflow of arrays when R becomes larger than the initial table size,

2. How to handle difficulties associated with removing records from R.

One way to avoid many of the problems created by hashing functions is to prevent collisions. A hash function is *perfect* if no two keys hash to the same table address. Recent advances in the theory of hashing have made it possible to devise a perfect hashing function when R is known in advance. R is said to be a *static set* if all the elements of R are initially known, otherwise, R is said to be a *dynamic set* of records.

While restrictive, the class of perfect hashing functions is most useful for applications in compiling (keywords in a programming language are known in advance of writing the compiler), database retrieval(we often manage known tables of data in a database system, e.g. the calendar), and text processing (electronic dictionaries contain static sets of words).

Perfect hash tables are very fast because only one probe is needed to retreive an arbitrary record. This is due to the simplicity of retrieval as follows:

Lookup

Step 1. Compute the Location from the perfect hash function, say Location = i,

Step 2. Compare the key at location i with the search key.

Step 3. If the keys match, the desired record has been found; if the keys do not match, we know the desired record does not exist without searching the entire table.

It is the ability to return "not-found" from a failed lookup without searching the entire table that makes perfect hashing extremely attractive for many applications.

In a perfect hashing function the keys must map into unique integers corresponding to table locations, but the mapping need not be compact, i.e. the table may contain empty slots. A perfect hash function is *minimal* if the set to be hashed and the hash table are the same size, e,g, the hash table has no empty slots. Besides the highly efficient comparison property, minimal perfect hash functions have the additional property that the hash table has no unused space.

While fast and simple, perfect hashing is limited by the need to know the never-changing keys in advance. This restriction can be overcome by other techniques, but not without paying a price in terms of decreased performance, increased demands for memory space, and increased complexity in the design and coding of search algorithms.

If both the number and the nature of the keys are unknown then we must approximate these attributes. For example, we might assume all records will fit into an array of size N and use a key-to-address transformation that sometimes yields collisions. Then we can devise various strategies for dealing with imminent overflow and collision. This leads to the traditional approach to hashing which we briefly survey before developing the more recent perfect hashing function techniques in greater depth.

TRADITIONAL HASHING

In traditional hashing a dynamic set R = { r_1, r_2, ...r_k, ...} is mapped into an array of length N. The size of R and the value of the keys in R are not known in advance. The table must cope with new records being added to R as well as deletions of records from R.

The three major steps in developing a hash table are:

Scatter Table Design

Step 1. Find a hashing function that distributes as evenly as possible the various possible sets of keys in R.

Step 2. Select a scheme for resolving collisions.

Step 3. Select schemes for expanding the table if it overflows and for handling deletions of records.

These steps involve a certain amount of ingenuity. The literature of the past 25 years has been concerned with overcoming the problems inherent in these steps. The first problem is to find a hashing function.

Hash Functions

Mapping a key to a table address is normally a two stage process. First, we must convert the alphanumeric key into an integer, then map this integer into a table address. In some instances, such as when the key is short, the alphanumeric key can be interpreted as an integer and the first stage bypassed.

Folding is a common method of converting a long alphanumeric key into an integer. Two common folding schemes are: (1) adding, shifting and exclusive-or'ing the characters in the key; and (2) converting the key into a base K number by treating each character in the key as a digit in a base K number; where K is the number of possible distinct characters, and then take this number modulo a large prime. Ramakrishna [RAMA 86] found less clustering (keys that map to same integer) for the latter folding algorithm.

Extensive tests have shown that once the key is converted into an integer, functions based on multiplication and division work well [KNUT73]. For example, the division scheme,

h(numeric_key) = numeric_key **mod** N

produces an integer in the range [0, (N-1)], corresponding to index numbers of an array with elements [0, (N-1)]. In the division scheme the table size, N, should be a prime number to assure full-table searching.

Collison Resolution

Because of the large number of possible sets of keys, collisions are inevitable. A good hashing function minimizes the number of collisions but does not eliminate them altogether. Thus, some scheme for resolving collisions is necessary. The two general classes of collision resolution schemes are:

1. Chaining: Instead of a key, each table entry is the head of a linked list of records with keys that map to that table location.

2. Open Addressing: Specification of an offset rule for the sequence of table locations to be searched until one of the following conditions occur:

   a. The record containing the key is found (success)
   b. An empty location is found (failure),
   c. The complete table is searched without success (failure).

These two collision resolution techniques can be illustrated by revisiting the calendar problem. What happens when two or more keys map into the same table location?

Chaining

To illustrate chaining let h be the hash function given earlier for the calendar problem. Using chaining, we re-compute the mapping of Figure 1 (First_letter(KEY) + Second_letter(KEY) Mod 13) with the exception that synonyms are placed in a linked list rooted in the hash table. The resulting chained hash table for the calendar problem is shown in Figure 2.

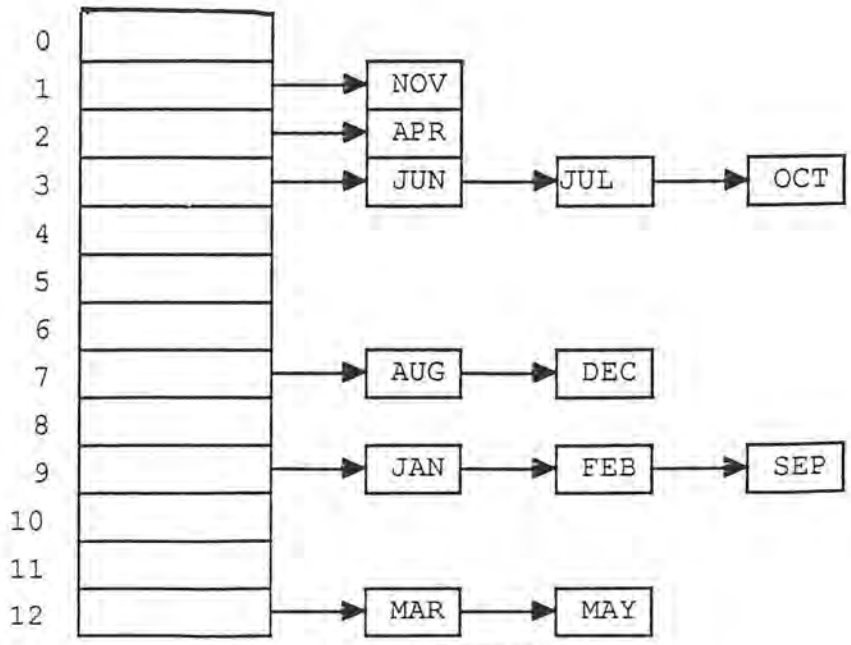

FIGURE 2

This type of chaining is called *separate chaining*. The average number of comparisons to find a key is 1.67. Note that there is an obvious space-time trade-off: the larger the table, the smaller the average chain size -- hence fewer comparisons are needed to find a key.

Another chaining technique is to have the synonym chain pointers point to locations in the table itself. This is called *coalesced hashing*. [VIT83]. For the calendar problem, the next available location for colliding keys is found at location 12 -- and working back up the table. The coalesced table shown in Figure 3 illustrates this. Note that h(MAR) = 12, but since FEB collided with JAN and was placed at location 12, MAR is placed in the next available location -- working back up the table from 12 yields 11. The average number of comparisons for the coalesced table is 1.75.

	KEY	POINTER
0		
1	NOV	
2	APR	
3	JUN	8
4	DEC	
5	OCT	
6	SEP	
7	AUG	4
8	JUL	5
9	JAN	12
10	MAY	
11	MAR	10
12	FEB	6

FIGURE 3

Open Addressing

Open addressing is illustrated by the mapping in Figure 1. The offset rule is given by:

location(KEY) = (h(KEY) + j) mod 13; where j = probe number.

This collision resolution function is called a *linear offset function* because it scans the array linearly, looking for an "empty" element. That is, the j^{th} probe is made at the j^{th} element beyond the initial mapping. For example, when SEP was encountered, h(SEP) = 9, but this table location was already filled so location10 was examined and found to be empty.

Notice that colliding keys tend to cluster around the synonym's initial location. This is called *primary clustering* and may cause the hash function to degenerate into a linear search. We could have used any increment (such as a number relatively prime to the table size) as long as the offset function guarantees that all possible table locations will be examined. However, synonyms still tend to pile-up into *secondary clusters*.

A variety of solutions to clustering have been proposed [MAUL 75, PRIC 71, MORR 68]. For example, *open hashing* suggests that an overflow area be used to hold synonyms. The synonyms are linked together in a chain via pointers and stored outside the array. This approach may have some value when the array is implemented as a direct access disk file and more than one record fits into a single disk "bucket" -- a track, cylinder or sector. But the maintenance and management of such a system makes it inferior to alternate approaches [LUMY 71].

One way to remedy both primary and secondary clustering is to let the increment depend on the value of the key. *Double-hashing*, or *quotient-offset hashing* is the best known hash function for dynamic sets [MAUL 75]. Cleverly, the quotient-offset function uses the fact that every integer is uniquely characterized by its remainder and quotient.

w(KEY) = First_Letter(KEY) + Second_Letter(KEY)
h(KEY) = w(KEY) **mod** N;
q = w(KEY) **div** N;
location(KEY) = (h(KEY) + q*j) **mod** N; for the j^{th} probe

This function uses the (unique) quotient of transformed key w as the offset value whenever a collision occurs. Thus, for the example, SEP collides with JAN and FEB, but w(SEP) = 152 and q(SEP) = 11 while w(JAN) = w(FEB) = 139 and q(JAN) = q(FEB) = 10. The important point is that the quotient-offset function spreads the keys across the table by searching out different sequences of elements even though colliding keys are synonyms.

Overflow

One of the most damaging limitations of dynamic hashing into fixed-size scatter storage tables is the problem of predicting the best size for the array. In fact, we stated that the performance of hashing was invariant with respect to the size of R, and depends entirely on the *density* of the storage array. Let density be measured by the loading factor of the storage array:

f = (number elements of the array that are occupied)/ N

Thus, if only one-half of the array contains hashed keys, then the loading factor is 0.5

and we say the density is 50%.

Performance can be roughly estimated by ignoring clustering effects and assuming N is very large.

average number of probes to lookup an existing key @ $1 / (1 - f)$

For example, a scatter table with 50% of its elements occupied will require approximately 2 probes per lookup. The formula does not consider all effects of hashing, however, and should be used only as an estimate of performance. It fails, for instance, to accurately estimate the number of probes expected when f approaches unity, e.g. an infinite number of probes is estimated when the table becomes full and $f = 1$.

The point of this analysis is to suggest that arrays not be allowed to fill up before we become concerned with overflow. Rather, suppose an overflow condition is defined, based on performance requirements:

Overflow = $(f > \max f)$; where $\max f$ is given

For example, overflow might be defined as the condition arising when $f > 0.8$, which corresponds to a performance estimate of 5 probes per lookup.

Overflow can be accommodated in one of two ways:

(1) allocate a larger array and move the contents of the old array into the new array by hashing all records a second time, or

(2) incrementally extending the old array to accomodate growth.

The first method is simple and slow while the second method is clever and fast.

An *extendible hash table* is a hashing function and scatter table data structure for accommodating growth in the table [FAGN 79]. The idea is to divide the storage array into fixed-length blocks and allocate one block at a time, as needed. The blocks are managed by a directory table containing pointers to each block, and a number telling how many bits of each key to examine during a lookup.

Suppose, for purposes of illustration, the calendar information is stored in an extendible hash table. Each block of the table is 5 elements long, and we define overflow to arise whenever $\max f = 3/5$ is exceeded. This means a maximum of 3 of the 5 elements of each block are occupied -- never more.

Initially, the directory table contains 2 elements and a flag set to 1, indicating that one least significant bit is used to decide which of the two blocks to search. The first

block contains all keys ending in a 0 bit, and the second block contains all keys ending in a binary 1. We use the quotient offset hashing function to place each key in its appropriate block.

When an overflow condition arises and a new block allocated to accomodate the overflow,

1. The directory table is doubled in size and its flag is incremented, e.g. the flag is bumped up to 2,

2. A new block is allocated and the directory pointer made to point to it. The new block will be used to hold the overflow from one of the existing blocks,

3. The overflowed block is "split" -- all keys whose least significant bits are the same are placed in the same block, e.g. the 2 least significant bits are used to determine which block to relocate each key to.

The extendible hash scheme is clever because it preserves a measure of performance while at the same time using a relatively simple extension technique to overcome the limitation of the fixed-length array structure. Performance studies suggest that this method is very competitive with other methods of storing and retreiving records from disk files.

Deletion

A final problem must be overcome -- deletion. For hash tables using chaining to resolve collisions, deletion is straightforward. However for open addressing, collision resolution schemes are complicated by the fact that a deleted record leaves a "gap" in the probe sequence. This gap can cause failure in subsequent lookups because a search is (prematurely) terminated as soon as an "empty" cell is encountered. If the "empty" cell previously held a colliding key, the search path leading to the correct key is destroyed by removing the colliding key.

A simple solution is to add an additional field to each record. This new field can take on one of three values: Empty, Occupied, or Deleted. Then when searching for a particular key, treat deleted records the same as occupied ones, but when inserting a record treat deleted records the same as empty ones.

The presence of Deleted elements increases the effective loading factor, which increases the number of probes during a lookup operation. In fact, an overflow condition can arise even when most of the array contains Deleted elements!

This solution to the deletion problem is not exceptional, and points to the need for more research into deletion techniques. If a storage table is predominately used for lookups then the high cost of deletion may not be a major concern, but if inserts and

deletes are the predominate operations on the structure, then other techniques should be considered.

PERFECT HASHING

Now we turn our attention to perfect hashing functions, a class of hashing functions without collision resolution problems since each key maps to a unique table location. Perfect hashing is also defined for external files. A hash table for an external file consists of buckets or pages each with a capacity of, say b records. An external hashing function is *perfect* if no bucket receives more than b records. In this paper we will be concerned with internal hash tables only.

Perfect hashing functions are rare. Knuth [KNUT 73] showed that in the set of all possible mappings of a set with 31 records to a table with 41 locations, only about one in ten million of the mappings is perfect.

The first scheme for finding perfect hashing functions appeared in 1963 [GREN63]. However, it was not general and the resulting function was complicated. Spugnoli [SPUG 77] developed two systematic schemes, Quotient Reduction and Remainder Reduction, for generating perfect hashing functions. Assuming the KEY has been converted into an integer w, the quotient reduction scheme involves finding integers s and N so that

$$h(w) = (w + s) / N$$

is perfect while the remainder reduction scheme involves find integers d, q, M, and N so that

$$h(w) = ((d + qw) \text{ Mod } M) / N$$

is perfect.

The table below illustrates the remainder reduction scheme for the calendar problem. The function E2+3(KEY) is the sum of the EBCDIC codings of the second and third letters in the KEY.

KEY	v = E2+3(KEY)	((3v+4) Mod 23)/2
JAN	49621	5
FEB	50626	6
MAR	49625	0
APR	55257	7
MAY	49640	11
JUN	58581	2
JUL	58579	10
AUG	58567	4
SEP	50647	3
OCT	50147	1
NOV	55013	9
DEC	50627	8

Figure 4.

Neither of Sprugnoli's schemes guarantee that the needed integers can be found nor are they practical for sets with more than 15 elements. However, for larger sets, Sprugnoli suggested a two stage hashing scheme called *segmentation*.

In segmentation, the first stage hash function is applied to the entire set R to hash all records into synonym buckets -- each synonym bucket contains all records with the same hash value. In the second stage, a different perfect hashing function for each bucket maps the keys in the bucket to unique locations in the segment of the hash table for that bucket.

By adjusting the first hashing function, the bucket size can be made small enough so that it is not too difficult to find a perfect hashing function for each bucket. The table below illustrates this two stage process for the calendar set. The FIRST(KEY) function is simply the ASCII coding of the first letter of the KEY **Mod** 2 and the function A2+3(KEY) is the sum of the ASCII codings of the second and third letters in KEY.

KEY	FIRST(KEY)	(A2+3(KEY)) **MOD** 12
JAN	0	11
FEB	0	3
JUN	0	7
JUL	0	5
NOV	0	9
DEC	0	6

KEY	FIRST(KEY)	12 + (A2+3(KEY)) **MOD** 10
MAR	1	19
APR	1	14
MAY	1	16
AUG	1	18
SEP	1	21
OCT	1	13

Figure 5.

The segmentation scheme requires two probes -- one to determine the bucket and obtain the information for the second hashing function for that bucket, and a second probe to access the table location that contains the KEY. Perfect hashing usually requires that the records in R be known in advance, but the segmentation scheme can be modified to handle updates (insertions and deletions) in a reasonable amount of time.

Deletions are simple and for a limited number of insertions the effort is localized to the perfect hashing function for that bucket. If a bucket overflows then the entire table must be rehashed or a special overflow area used. In the latter case the number of probes may greatly exceed two.

Minimal Perfect Hashing

Recall that a hashing function is *minimal* if there are no empty table locations. The hashing function in Figure 4 is both minimal and perfect. Minimal perfect hash functions have two desirable properties:

(1) One can, in exactly one probe, determine the table address of the item with the search key or conclude that the item is not present, and

(2) the hash table has no unused space.

Minimal perfect hash functions are ideally suited for compact storage and fast

retrieval of frequently searched static sets such as reserved or predefined words in programming languages, operation codes in assembly languages, or frequently used words in a natural language.

There has been a recent flurry of theoretical and heuristic algorithms for minimal perfect hash functions. The reciprocal hashing scheme of Jaeschke [JAES 81] is an example of a theoretical algorithm. The hash function for key w in R (set of distinct positive integers) is

$$h(w) = (C/(Dw + E)) \textbf{ mod } N$$

where C, D, and E are integer constants computed by special algorithms and N is the size of the set R.

Although the existence of the function h is guaranteed, there are several practical problems with this scheme. First, it is assumed that R is a set of distinct positive integers. Second, the reciprocal hashing algorithm has exponential time complexity and hence is not practical for sets with more than twenty elements. Finally, the constant C may become very large.

Chang [CHAN 84] developed a theoretical scheme similar to Jaeschke's based on the Chinese Remainder Theorem. His hash function is of the form

$$h(w) = C \textbf{ mod } p(w)$$

where C is an integer constant computed by his algorithm and p(w) is a prime number function, e.g, p(w) is a different prime for each integer w. Unfortunately the number of bits to represent C is proportional to $m(\log_2 m)$ where m is the size of R and Chang gives no general method of finding the prime number function.

Cichelli [CICH 80] presented a simple heuristic hash function for small static sets. His hash function for a word w is an extension of the First/Last function introduced earlier:

$$h(w) = \text{Value}(\text{First\_Letter}(w)) + \text{Value}(\text{Last\_Letter}(w)) + \text{Length}(w)$$

Figure 6 gives the letter values and resulting hash table for R = { JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER }

LETTER	VALUE(LETTER)	KEY	h(KEY)
A	0	JANUARY	7
D	-7	FEBRUARY	2
E	5	MARCH	10
F	-6	APRIL	11
H	5	MAY	3
J	0	JUNE	9
L	6	JULY	4
M	0	AUGUST	12
N	-3	SEPTEMBER	8
O	-1	OCTOBER	6
R	0	NOVEMBER	5
S	-1	DECEMBER	1
T	6		
Y	0		

Figure 6

Building the hash table in Figure 6 involves finding the Value function that makes h(w) a perfect minimal hash function, i.e. finding a set of character values that map each word in the set to a unique table address such that the table contains no internal blank slots. To find the character value assignments, Cichelli's algorithm uses an intelligent, exhaustive search with backtracking that considers one key at a time.

Cichelli successfully applied his algorithm to several sets -- the 31 most common English words and the 36 Pascal reserved words.

Cook and Oldehoeft [COOO 85] improved the performance of Cichelli's backtracking algorithm by developing a letter-oriented algorithm that considers groups of key words rather than a single word in finding letter value assignments. Also when the letter value assignment algorithm reaches an impasse, Cichelli's algorithm backtracks one word at a time while the Cook and Oldehoeft algorithm immediately backs up to the prior letter value assignment that caused the impasse.

Cichelli's hash function is simple, fast, and machine independent. The latter property is especially important because once a hash function has been found for a word set, the same hash function (assignment of character values) can be used for the word set on any computer.

However, Cichelli's algorithm has several shortcomings. Because the letter value assignment process uses an exhaustive search with backtracking its time complexity is exponential. Hence it is only practical for small static sets with less than 45 words. In addition, the algorithm is not guaranteed to work for all sets. For example it cannot

work for sets with conflicting word pairs, two words with the same length and same pair of characters at their ends (e.g. JAN and JUN, or TYPE and EXIT). (This is why we used the full calendar month names for Figure 6 rather than the three letter abbreviations used in previous examples).

Conflicting word pairs are common in moderate-sized sets of English words as shown by a Monte Carlo study [BELF 83] in which random word sets with English-language probabilities of first letters, last letters, and word length were generated. About half of the sets with 30 words were found to contain conflicting pairs of letters. Choosing other letter positions is not a solution either, for example, every pair of letters conflict in the seven word set R ={CASE, ELSE, PAGE, READ, REAL, TRUE, TYPE } [HA 86]. Jaeschke and Osterburg [JAEO 80] point out many other cases for which Cichelli's heuristic does not work.

Chang and Lee [CHAL 86] developed a letter-oriented minimal perfect algorithm based on ideas from Chang's previous algorithm and Cichelli's algorithm. The hashing function for a set of words is

$$h(L_i, L_j) = d(L_i) + C(L_i) \text{ Mod } p(L_j))$$

where L_i and L_j are the i^{th} and j^{th} letters in the KEY; d and C are integer valued functions, and p is a prime number function. They give algorithms for finding the functions d, C, and p. The hashing function for the full word calendar set is given below. Second and third letters were selected for the letter pairs because they are all unique.

LETTER	d(LETTER)	C(LETTER)	p(LETTER)
A	0	119	
B			5
C	3	1	7
E	4		
G			11
L			13
N			2
O	7	1	
P	8	1	17
R			3
T			19
U	9	211	
V			23
Y			29

KEY	LETTER PAIR	h(KEY)
JANUARY	(A,N)	1
FEBRUARY	(E,B)	5
MARCH	(A,R)	2
APRIL	(P,R)	9
MAY	(A,Y)	3
JUNE	(U,N)	10
JULY	(U,L)	12
AUGUST	(U,G)	11
SEPTEMBER	(E,P)	7
OCTOBER	(,C,T)	4
NOVEMBER	(E,V)	8
DECEMBER	(E,C)	6

Figure 7.

Thus h(SEPTEMBER) = d(E) + (C(E) **mod** p(P)) = 4 + (156 **mod** 17) = 7. However, they not give a method for selecting the pair of letter positions nor the general prime number function.

Extensions To Cichelli's Hash Function

Cichelli's hash function is attractive because of its simplicity and machine independence. There have been several attempts at extensions of his function that

preserve its attractiveness. Several of these extensions handle large static sets with up to several thousand words while preserving both its simplicity and machine independence and at the same time overcome its word pairs conflict problem. The extensions are significant because they extend minimal perfect hashing to practical applications in electronic dictionaries, natural language processing, and textual database processing.

The two general extension schemes are:

1. Modify Cichelli's hash function by adding more terms, changing the character evaluation function, and considering different character positions in a word.

2. Segment or partition the large set into small sets (buckets) and find minimal perfect hashing functions for each bucket.

1. MODIFICATIONS OF CICHELLI'S HASH FUNCTION

Many modifications of Cichelli's hash function have involved adding terms or changing the letter positions used. For example the 63 reserved words of Ada™ contain three pairs of conflicting words (EXIT-TYPE, RAISE-RANGE, PRIVATE-PACKAGE) and the 40 reserved words of Modula-2 contain one conflicting pair (TYPE-EXIT). Sebesta and Taylor [SEBT 85, SEBT 86] found a minimal perfect hash function for the Modula-2 words by adding the term

alphabetic position of second to last character in word

to Cichelli's hash function, and a minimal perfect hash function for the Ada™ reserved words by adding twice the same term to Cichelli's hash function.

Cercone, Boates, and Krause [CEBK 85] developed an interactive system that allows the user to specify the set of character positions and whether or not to include the key length in the hash function. Their algorithm uses a non-backtracking, intelligent, enumerative search to find the character value assignments. If the user's selection of character positions results in a conflict, the system invites the user to make another selection. The user may also specify the hash table loading factor (1 for a minimal perfect hash and less than 1 for a non-minimal perfect hash). Their algorithm found minimal perfect hash functions for sets with up to 64 elements and almost minimal solution for sets with up to 500 elements.

Haggard and Karplus [HAGK 86] generalized Cichelli's hash function by considering every character position in a key. They search for functions of the form

$$h(w) = \text{length}(w) + g_1(\text{first\_char}(w)) + g_2(\text{second\_char}(w)) + \ldots + g_k(\text{last\_char}(w))$$

where the g_i's are called selector functions. Their algorithm finds a set of selector

functions that uniquely identify each word and then uses backtracking heuristics to find the character value assignments for each selector function. They found minimal perfect hash functions for sets with up to 181 words and non-minimal perfect hash functions for sets with up to 667 words.

Sager [SAGE 85] developed a hash function of the form

$$F(w) = (h_0(w) + g(h_1(w)) + g(h_2(w)) + g(h_3(w))) \textbf{ mod } N$$

where h_0, h_1, h_2 are pseudorandom functions, g is the function found by his mincycle algorithm, and N is the size of the set of words. Cichelli's hash function is a special case of Sager's hash function where h_0 (w) is the length of w, h_1(w) is the first character in w, h_2(w) is the last character in w, and g is the character value assignments. Sager gives several choices of pseudorandom functions that "seem to work well", but he did not give a general scheme for choosing the pseudorandom functions. He claims his algorithm is practical for sets with 512 or more words.

These modification schemes seem to be aimed primarily at overcoming the conflicting word pair problem and, even though they handle larger sets, they do not represent a general solution for very large sets with several thousand words. In addition many of the schemes generate perfect, but not minimal hash functions. Hence it appears that these schemes are limited to minimal perfect hash functions for sets with at most 500 words.

2. SEGMENTATION SCHEMES

Segmentation, the second type of extension scheme seems to be more promising for very large sets. Several papers [SPRU 77, COHK 85, FRKS 82, LARR 85] suggest buildin perfect hash tables for large sets of words by finding one hash function to partition the large set into synonym buckets (all words with same hash value are in the same bucket) and then finding individual (minimal) perfect hash functions for each of the buckets.

Sprugnoli called the first hash function a grouping function [SPRU 77]. The task of the grouping function is to partition the large set into buckets with at most maxsize words, where maxsize is the practicl limit for the minimal perfect hashing function. Buckets are stored in unique segments (consecutive locations) in the hash table. Thus the hash function is

$$h(w) = b_i + h_i(w)$$

where i is the grouping hash function value for w; b_i is the base address for bucket i in the hash table, and h_i is the perfect hash function words in bucket i.

In the segmentation schemes of Cormack [COHK 85], Fredman [FRKS 82] , and Larson ar Ramakrishna [LARR 85] the hash table consists of two parts: a table D containing the actual data elements and their keys and a header table H with an entry for each bucket containing the information needed to compute the exact address of an element in that bucket. Buckets are stored in unique segments in table D. The first hash function computes the location in the header of the bucket information -- base address for segment and constants in the perfect hash function for the bucket. A trial-and-error approach is used to select the perfect hash function from a family of functions.

Cormack [COHK 85] obtained a family of hash functions for buckets of size r, of the form:

$$h_i(w) = (w \textbf{ mod } c) \textbf{ mod } r$$

where c is a randomly generated integer.

Larson and Ramakrishna were primarily concerned with finding a perfect hash function for large external files. Their hash table (hash file) consisted of m pages each with a capacity of b records. It maps each record onto a page. A hash function is said to be perfect if no page receives more than b records. The family of hash functions that worked well were of the form

$$h(w) = ((c * w + d) \textbf{ mod } p) \textbf{ mod } m$$

where c and d are integers, p is a prime number greater than the largest key, and m is the size of a page. They successfully hashed files ranging from 3,000 to 24,000 records. Significantly, they showed it was practical to find a perfect hash function from the family by randomly generating (c,d)-integer pairs. Depending on the load factor of the tables, only a small number of (c,d) pairs need be generated to find a perfect hash function. For example, for a load factor less than 70%, the probability of a randomly chosen (c,d) pair yielding a perfect hash function is greater than 0.5.

We investigated a minimal perfect extension of Cichelli's hash function based on Larson and Ramakrishna's scheme. They were interested in finding a hash function that partitioned the set of records into pages with at most b records. We were interested in finding a hash function that partitions the set of words into small buckets with at most 40 words such that no bucket contains a conflicting word pair (two words with the same length and same pair of characters at their ends). We chose the same family of hash functions for the grouping function to compute the bucket number of a word, where c, d and p are the same and m is the number of buckets. Hence our minimal perfect hash function is

$$h(w) = \text{offset}(b\#(w)) + \text{value}(b\#(w), \text{first\_char}(w)) + \text{value}(b\#(w), \text{last\_char}(w)) + \text{length}(w)$$

where b#(w) is the bucket number computed by the grouping function, value(n,ch) is the character values for characters in bucket number n, and offset(n) is where the segment of words in bucket number n begins in the hash table. Thus the algorithm to determine the location of a word in the table or to conclude that it is not present is a simple two step process: first compute the bucket number and length of the word; then perform a table look-up to find the values of first and last characters of the word and to find the offset of the bucket.

A perfect minimal hash table for the calendar problem (3 letter abbreviations) constructed via the segmentation scheme is given in Figure 8.

BUCKET#1 = { JAN, MAR, MAY, JUL, SEP, NOV, DEC }

BUCKET#2 = { FEB, APR, JUN, AUG, OCT }

LETTER	BUCKET#1	BUCKET#2
A		0
B		2
C	0	
D	3	
F		0
G		-1
J	0	2
L	1	
M	0	
N	-2	-1
O		0
P	1	
R	2	-2
S	3	
T		0
V	2	
Y	1	

	BUCKET#1	BUCKET#2
OFFSET	0	7

KEY	h(KEY)
JAN	1
MAR	5
MAY	4
JUL	2
SEP	7
NOV	3
DEC	6
FEB	12
APR	8
JUN	11

AUG	9
OCT	10

Figure 8.

Thus for SEP, the bucket number is 1 so

h(SEP) = OFFSET(1) + VALUE(1,S) + VALUE(1,P) + LENGTH = 0 + 3 + 1 + 3 = 7.

Larson and Ramakrishna [LARR 85] and Ramakrishna [RAMA 86] found their perfect hash function through a trial-and-error process by fixing p and generating random (c,d) pairs. We wanted a more systematic method. We found that selecting c, d, and p from different small sets of prime numbers partitioned the word sets into relatively uniform size buckets. A (c, d, p) triple is successful if no bucket contains a conflicting word pair.

Our scheme worked successfully for the 447 FORTH words, the 288 FORTH Assembler words, and sets of 500 and 1000 random words. However, for the 1000 most common English words and 2000 random words the same exhaustive generation scheme was unsuccessful unless we reduced the bucket size to near 10 because of the skewed distribution of the word lengths and the letters at the ends of the words. We did notice that very few of the buckets contained word pair conflicts. This suggested modifying our scheme to move the second word of a word pair conflict to a special extra bucket. After this process none of the original buckets contained word pair conflicts and only the extra bucket needed to be checked for word pair conflicts. Using the extra bucket decreased the effort and increased the bucket sizes for the solutions to the 1000 English words and 2000 random words. The additional cost for this scheme was slight since the number of words in the extra table is small relative to the size of the set.

This points out the two major difficulties with the segmentation approach:

1. Finding a general segmentation scheme that partitions the words into buckets such that the words in each bucket have the required properties for a minimal perfect hash function.

2. There is no systematic and practical scheme for generating minimal perfect hashing functions. We only considered Cichelli's minimal perfect hashing function. Additional research needs to be done which considers different variations of Cichelli's scheme or other types of minimal perfect hash functions for the buckets.

CONCLUSIONS

A large class of problems can be solved with hashing functions. For dynamic sets the best techniques are based on simple prime division hashing functions, and where

[JAES 81] Jaeschke G., "Reciprocal hashing: A method for generating minimal perfect hashing functions." ***Comm. ACM***, 24,12, (Dec 1981), 829-833.

[JAEO 80] Jaeschke G. and G. Osterburg, "On Cichelli's minimal perfect hash functions method." ***Comm. ACM*** 23, 12, (Dec. 1980), 728-729.

[KNUT 73] Knuth D. E., ***The Art of Computer Programming: Vol. 3.*** Addison-Wesley, Reading, 1973.

[LARR 85] Larson P. and M. V. Ramakrishna, "External perfect hashing." ***Proc. ACM-AIGMOD Internation Conference on Management of Data*** (Austin Texas 1985), 190-200.

[LUMY 71] Lum, V. Y., P.S.T. Yuen, and M. Dodd, "Key-to-Address Transformation Techniques: A Fundamental Performance Study On Large Existing Formatted Files." ***Comm ACM*** 14, 4, (Apr 1971), 228-239.

[MAUL 75] Maurer, W. D., and T. G. Lewis, "Hash Table Methods." ***Computing Surveys*** 7, 1 (Mar 1975), 5-19.

[MORR 68] Morris, R., "Scatter Storage Techniques." ***Comm ACM***, 11, 1, (Jan 1968), 38-44.

[PRIC 71] Price, C. E., "Table Lookup Techniques." ***Computing Surveys*** 3, 2, (June 1971), 49-65.

[RAMA 86] Ramakrishna M. V., "Perfect hashing for external files." ***Technical Report CS-86-25,*** Computer Science Department, University of Waterloo, June 1986.

[SAGE 85] Sager T., "A Polynomial time generator for minimal perfect hash functions." ***Comm. ACM*** 28, 5, (May 1985), 523-532.

[SEBT 85] Sebesta R. and M. Taylor, "Minimal perfect hash functions for reserved word lists." ***ACM SIGPLAN Notices*** 20, 12, (Dec. 1985), 47-53.

[SEBT 86] Sebesta R. and M. Taylor, "Fast identification of Ada and Modula-2 reserved words." ***J. Pascal, Ada, and Modula-2*** March/April 1986, pp. 36-39.

[SPRU 77] Sprugnoli, R. "Perfect Hashing Functions: A Single Probe Retrieving Method For Static Sets, ***Comm. ACM***, 20, 11, (Nov 1977), 841-850.

[VITT 87] Vitter, J.S. and W. Chen, ***Design and Analysis of Coalesced hashing.*** Oxford University Press, New York, 1987.