HASH TABLE METHODS

W. D. MAURER

Department of Electrical Engineering and Computer Science, George Washington University, Washington, D. C. 20052

T. G. LEWIS

Computer Science Department, University of Southwestern Louisiana, Lafayette, Louisiana 70501

This is a tutorial survey of hash table methods, chiefly intended for programmers and students of programming who are encountering the subject for the first time. The better-known methods of calculating hash addresses and of handling collisions and bucket overflow are presented and compared. It is shown that under certain conditions we can guarantee that no two items belonging to a certain class will have the same hash code, thus providing an improvement over the usual notion of a hash code as a randomizing technique. Several alternatives to hashing are discussed, and suggestions are made for further research and further development.

Keywords and Phrases: hash code, hashing function, randomizing technique, key-to-address transform, hash table, scatter storage, searching, symbol table, file systems, file search, bucket, collision, synonyms

CR Categories: 3.51, 4.33, 4 34

INTRODUCTION

One of the most actively pursued areas of computer science over the years has been the development of new and better algorithms for the performance of common tasks. Hash table methods are a prime example of successful activity of this kind. With a hash table, we can search a file of n records in a time which is, for all practical purposes, independent of n. Thus, when n is large, a hash table method is faster than a linear search method, for which the search time is proportional to n, or a binary search method, whose timing is proportional to $\log_2 n$. This paper gives a tutorial survey of various hash table methods; it is chiefly intended for those who are encountering the subject for the first time.

Not every linear or binary search can be replaced by a hash table search. If we have a file of customers, and we wish to print out, for each month in the past year, the name of that customer which generated the most activity during that month, we must use a linear search (although it would not be necessary to make twelve separate linear searches). Nevertheless, hashing methods may always be used for the most common type of search, namely that in which we have n records $R_1, R_2, \cdots, R_n$, each of which (say R_i) consists of data items $R_{i1}, R_{i2}, \cdots, R_{ij}$ for some j, and we are trying to find that record R_i for which R_{i1} is equal to some pre-

CONTENTS

determined value. The data item R_{i1} is called the *key*.

As an example, consider a symbol table in an assembler, compiler, interpreter, or more general translator. There is one record R_i corresponding to each identifier in the source program. The identifier itself is R_{i1}; the other information which must be associated with that identifier (an address, type, rank, dimensions, etc.) is R_{i2}, R_{i3}, and so forth. Each time an identifier is encountered as the source program is processed, we must find the record R_i which contains that identifier as R_{i1}. If there is no such record, we must create a new one. All of the records are normally kept in main memory.

As another example, consider a data base. Here the records are normally kept on disk or some other form of addressable auxiliary memory. The key, R_{i1}, in each record R_i may be a part number, an airline flight number and date, an employee number, an automobile license number, or the like. Whenever a transaction takes place, we must find, on disk, that record which contains a given key value.

All hashing methods involve a *hash code* or *hashing function* or *mapping function* or *randomizing technique* or *key-to-address transform*, which we shall denote by h. If K is an arbitrary key, then h(K) is an address. Specifically, h(K) is the address of some position in a table, known as a *hash table* or *scatter storage table*, at which we intend to store the record whose key is K. If we can do this, then if at some later time we want to search for the record whose key is K, all we have to do is to calculate h(K) again. This is what makes hashing methods so inviting; *most* of the time, we can find the record we want immediately, without any repeated comparison with other items.

The only time we cannot store a record at its *home address*—that is, at the address h(K), where K is the key in that record—is when two or more records have the same home address. In practice, there is no way we can prevent this from happening, because there are normally several orders of magnitude more possible keys than there are possible home addresses. For example, with six bits per character, six characters per word, and eighteen bits per address, there are 2^{18} possible addresses, while the number of possible six-character alphabetic identifiers is 26^6, which is over 1000 times greater than 2^{18}.

The phenomenon of two records having the same home address is called *collision*, and the records involved are often called *synonyms*. The possibility of collision, although slight, is the chief problem with hash table methods. It may be circumvented in a number of ways, as follows:

1) If the home addresses are in main memory, we can make a list of all the synonyms of each record. To find a record in the table, we would first find its home address, and then search the list which is associated with that particular home address.

2) Alternatively, we can determine some address, other than the home address, in which to store a record. We must then make sure that we can retrieve such a record. This subject is taken up further in the section below on "Collision and Bucket Overflow."

3) There are some situations in which more than one record can be stored at the same address. This is particularly common

when the records are to be stored on disk. A disk address is often the address of an area (such as a track) which is large enough to hold several records. Similarly, a drum channel is an addressable area which may hold several records. For the purposes of hashing, such an area is called a *bucket.* If n records can be stored in a bucket, then any record which is kept in that bucket can have n-1 synonyms, with no problem. Of course, it might conceivably have more synonyms than that, in which case we have to find another bucket. This is called *overflow*, and will also be taken up in the section on "Collision and Bucket Overflow."

4) Finally, we can seek to minimize the probability of collisions—or, for records in buckets as above, the probability of bucket overflow—by a suitable choice of the function h(K). Intuitively, we can see that this should be more likely to happen the more thoroughly the hashing function "mixes up" the bits in the key (in fact, this is how "hashing" got its name). Ideally, for a hash table of size n, the probability that two randomly chosen identifiers have the same home address should be $1/n$. However, there are hashing methods which are actually "better than random" in that it is possible to prove that all members of a certain class of identifiers *must* have distinct home addresses. This is considered in the following section, "Hashing Functions"; further "Theoretical Analyses" are taken up in the section so titled.

A hash table has fixed size. If it is necessary to increase the size of a hash table during a run, all quantities currently in the table must have their hash codes recalculated, and they must be stored at new locations. Also, hashing is applicable only to certain rather simple (although quite commonly occurring) searching situations. The section titled "Alternatives to Hashing" offers solutions for these problems, together with guidelines as to whether hashing is or is not applicable in various situations.

Hashing was first developed by a number of IBM personnel, who did not publish their work. (See [14], pp. 540–541, for a discussion of this early work.) Hashing methods were first published by Dumey [9], and independently by Peterson [27]. The term "synonym" seems to be due to Schay and Raver [30]. Several surveys of hash table methods exist, of which the best (although also the oldest, and restricted to file addressing methods) is that of Buchholz [6]; also see Morris [24], Knuth [14], and Severance [32]. An interesting method of recalculating hash codes when the hash table size is increased is given by Bays [2].

HASHING FUNCTIONS

As an example of a hashing function, consider the following. Take the key and divide it by some number n. Consider the *remainder.* This is an integer less than n and greater than or equal to zero. Hence we may use it as an index in a table of size n. In other words, if T is the starting address of a hash table and K is an arbitrary key, then h(K) = T + MOD(K, n) (as it is referred to in FORTRAN) is the home address of K.

This is the *division method* of calculating h(K). At least six other suggestions have been made, over the years, as to what hashing function h(K) should be used. The choice of a hashing function is influenced by the following considerations:

1) Are we allowed to choose our hashing function to fit the particular records being searched? In an assembler or compiler, the answer is clearly no; we do not know what identifiers we will encounter in the source program until we actually encounter them. In a data base environment, however, the answer might be yes, particularly if we have a data base which remains constant, or nearly so, over a period of time. This is discussed further when we consider the digit analysis method.

2) Does it matter to us how long it takes to calculate h(K)? If our records are on disk or drum, the answer might very well be no, particularly if access time is large. In such a case, we should use the method which minimizes the number of disk or drum accesses, regardless of how long it takes to calculate h(K). We shall see in the discussion of "Theoretical Analyses" that this often means we want to use the *division* method as

outlined above, even on a computer which does not have a divide instruction. In an assembler or compiler, on the other hand, it would not make much sense to minimize the number of accesses to the hash table, when a single calculation of h(K) could, for some hashing functions h, take as long as several such (main memory) accesses.

3) Are we allowed to design hardware to implement the hashing function? Most computers do not have instructions which perform hashing directly; we must use some combination of existing instructions. This is quite unfortunate, since hashing is a common operation in a variety of programs. One hashing function (*algebraic coding*, described below) is constructed specifically for implementation in hardware. Another possibility would be to design a hashing instruction which incorporates one or another of the collision algorithms discussed in the next section.

4) How long are the keys? If we are concerned about calculation time, we might not want to use the division method for keys which are longer than one computer word. For longer keys, we may combine one of the other methods with the method of *folding*, which is discussed further on.

5) Are the addresses binary or decimal? Some of the methods described below rely heavily on specific instructions which are normally available only in binary or only in decimal form.

6) Are we free to choose which addresses or indices will be used in our table? If so, we can choose the table size to be a power of 2 (or a power of 10, if the addresses are in decimal form). This is required by certain of the methods discussed in this paper. It may be, however, that our hashing routines are to be used in an environment in which the addresses (presumably on drum or disk) have already been chosen in some other way.

7) Are the keys more or less randomly chosen? Usually there will be some degree of uniformity in the keys. It is very often possible to show that, for a certain choice of hashing function, a wide class of plausible keys will all have the same hash code (or home address). Also, it may be that the keys are *so* nonrandom that we may not want to use hashing at all. This is discussed further in the section, "Alternatives to Hashing."

Under certain conditions, we may show that keys constructed in special ways *do not* have the same home address. In particular, suppose that K is a key and suppose that K+1, K+2, etc., are also keys. If the division method above is used, and h(K) = α, then h(K+1) = α+1, h(K+2) = α+2, and so on (modulo the size of the table). Thus, under these conditions, all of these keys must have distinct home addresses. This subject is considered further in the discussion of "Theoretical Analyses."

Let us now look at various other methods which have been proposed for performing hashing.

a) The random method. Almost all computers have subroutines which are used in statistical work when a random sequence of numbers is desired; they are called "random number generators," although the numbers they produce are actually not quite random (in fact such subroutines are more properly referred to as *pseudo-random number generators*). Typically such a subroutine is provided with a starting number, called the *seed*, and proceeds to produce, when called repeatedly, a random-looking sequence of numbers. The idea here is to use the key K as the seed, choose the first of the random numbers for the home address, and use the others, if necessary, for collision handling (see the following section). The output of a random number generator is usually a floating-point number between 0 and 1, and this must be normalized by first multiplying by n and then converting from real to integer, giving a random number between 0 and n-1 inclusive. We remark for future reference that, if this method produces a truly random result, the probability of any two keys having the same home address is strictly positive, even if the keys are such that, as noted before, that probability would be zero if we used the division method. Thus, even perfect randomization gives less than optimal results in many cases.

b) Midsquare and other multiplication methods. Take the key and multiply it either by itself or by some constant. This presumably "hashes up" the bits rather thoroughly (this assumption is often false, and must be thought through carefully in every hashing situation). Now pick out a certain *field*, say 10 bits long, from the *middle* (approximately) of the result. Mask out everything but this 10-bit field, and consider it as a more or less random integer between 0 and $2^{10}-1$, inclusive. This means that the hash table will be $2^{10} = 1024$ words long. If we need a larger or a smaller hash table, we can use a different power of 2.

c) The radix method. Take the key and consider it as a string of octal digits. Now consider this same string of digits as if they were digits, not in base 8, but in some other base (say 11). Convert the resulting number, in base 11, to base 10 (if you want a decimal address). This process "hashes up" the bits as before, and we can now take some field as in the midsquare method. This presumes, of course, that the table has size 10^n, for some n; for a table of size 10,000, a four-digit field is required. A general discussion of this method, with parameters p, q, and m (we have here taken $p = 11$, $q = 10$, and $m = 4$) is given by Lin [16].

d) The algebraic coding method. Suppose the key K is n bits long. Consider a polynomial of degree n-1, each of whose coefficients is one of these n bits. (Each coefficient is therefore either 0 or 1.) Divide this polynomial by some constant kth degree polynomial; all arithmetic operations in this polynomial division are to be carried out modulo 2. (Mathematically, this amounts to performing the polynomial division over GF(2), the Galois field of two elements.) Consider the *remainder*, just as in the division method; this is a polynomial of degree k-1, and may be considered as a string of k bits. In algebraic coding theory, we would append these k bits as check bits to the original n bits; if the constant polynomial by which we divide is chosen carefully, we obtain an error-correcting code. In the present application, however, we simply use the resulting k bits as the home address of K. As in the midsquare method, this requires that the table size be a power of 2 (in this case, 2^k).

e) Folding and its generalizations. A very fast method of obtaining a k-bit hash code from an n-bit key, for $k < n$, is by picking out several k-bit fields from the n-bit key and adding them up, or, alternatively, taking the exclusive OR. This is sometimes called "fold-shifting," since it is related to the original idea of folding, which is as follows. Consider the key as a string of digits written down on a slip of paper which is then folded as in Figure 1. All the digits which are "next to each other" as a result of the folding process are now added up. It will be seen that this is the same as fold-shifting except that some of the fields have reversed bit (or digit) strings, making the method considerably slower in that case. Folding is often combined with other methods when the keys are long; for example, for 16-byte (128-bit) keys on the IBM 370, we may consider these as four single words, add these up (or exclusive-OR them), and apply division or some other method to the single-word result.

f) Digit analysis. Of all the methods presented here, this is the only one which de-

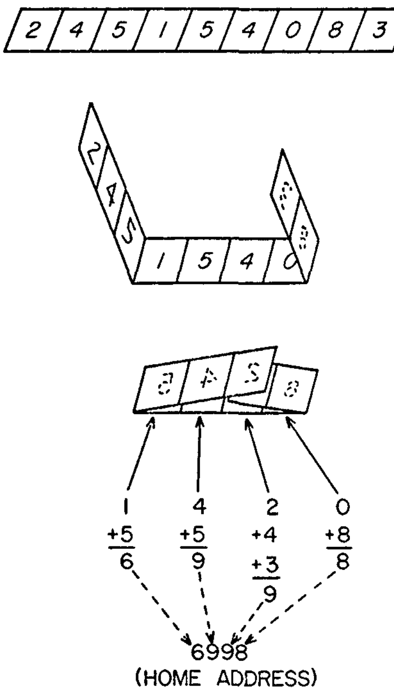

FIG. 1. Folding (sometimes, in the form illustrated, known as the fold-boundary method).

pends on the specific data to be hashed. It is usually used with decimal-number keys and decimal-number addresses, and in this context it works as follows. Look at the first digit of each key. If there are N different keys, then $N/10$ of these, on the average, should have the first digit zero, another $N/10$ should have the first digit 1, and so on. Suppose that there are actually N_i keys whose first digit is i, $0 \leq i \leq 9$. Sums such as

$$\sum_{i=0}^{9} | N_i - N/10 | \quad \text{or} \quad \sum_{i=0}^{9} (N_i - N/10)^2$$

represent the "skewness" of the distribution of the first digit. Repeat this analysis for *each* digit, and find the k digits which are the best—that is, which have the least amount of skewness, using one or the other measure or both—where k is the number of digits in an address. The home address of an arbitrary key is now found by "crossing out" all but those particular digits from that key.

The division method is mentioned explicitly by Dumey [9]. We have already mentioned Lin's work on the radix method. The algebraic coding method seems to have been first presented by Hanan and Palermo [11] and Schay and Raver [30]. The origin of the other methods we have mentioned is lost at the present time. Buchholz [6] surveys all of these methods; an introductory account of key-to-address transformation by Price [28] mentions folding and digit analysis, while a random search method is mentioned in a "pracnique" by McIlroy [23]. Several books ([8], [12], [14], [22], [38]) also survey hashing methods.

COLLISION AND BUCKET OVERFLOW

We draw in this survey a sharp distinction between the problem of *collision* and the problem of *overflow* of buckets. These problems arise in similar ways, but they are not identical. A collision problem arises when we are assuming that, most of the time, all of our keys have distinct home addresses. When two keys happen to have the same home address, then we invoke a collision handling method. An overflow problem arises when we are assuming that, most of the time, no more than n keys, for some fixed value of n, have the same home address. In this case each bucket will contain n records. We invoke an overflow handling method in the few cases where there is one home address corresponding to more than n keys.

There are quite a number of problems which have to be solved in connection with collision; and, in order to give the reader a feel for these problems, we now describe one particular collision handling method in detail. Suppose that the home address of key K is α; we store at α the record whose key is K. Now suppose that another key, K$'$, also has home address α. When we try to store the new record at α, we find that the old record is already there; so we store the new record at $\alpha+1$ instead. This raises the following further problems:

1) There may be a record at $\alpha+1$ already, as well. In this case we try $\alpha+2$, $\alpha+3$, and so on.

2) We have to be able to tell whether there is, in fact, a record currently at any given position. This means that we have to have a code which means "nothing is at the given location"; the entire table is initialized at the start of our program in such a way that every position contains this special code.

3) If α is the *last* location in the table, and there is something there already, we cannot store the new record at $\alpha+1$; so we cycle back to the beginning of the table (that is, we try to store the new record as the *first* record in the table).

4) If we ever have to take any records *out* of the table, there is a rather subtle difficulty which will be discussed below under *deletions*.

5) The next position in our table after the one with address α might have address (say) $\alpha+8$, rather than $\alpha+1$. This would happen, for example, if the records were double words on the IBM 370, each of which is eight bytes long. The adjustments in our algorithms which are necessary in order to take care of this phenomenon are so trivial that we will ignore them hereafter.

This is the *linear* method of handling collisions. If $\alpha = h(K)$ is the home address, then we shall refer to $\alpha+1$, $\alpha+2$, and so on (possibly adjusted as in (5) preceding) as the

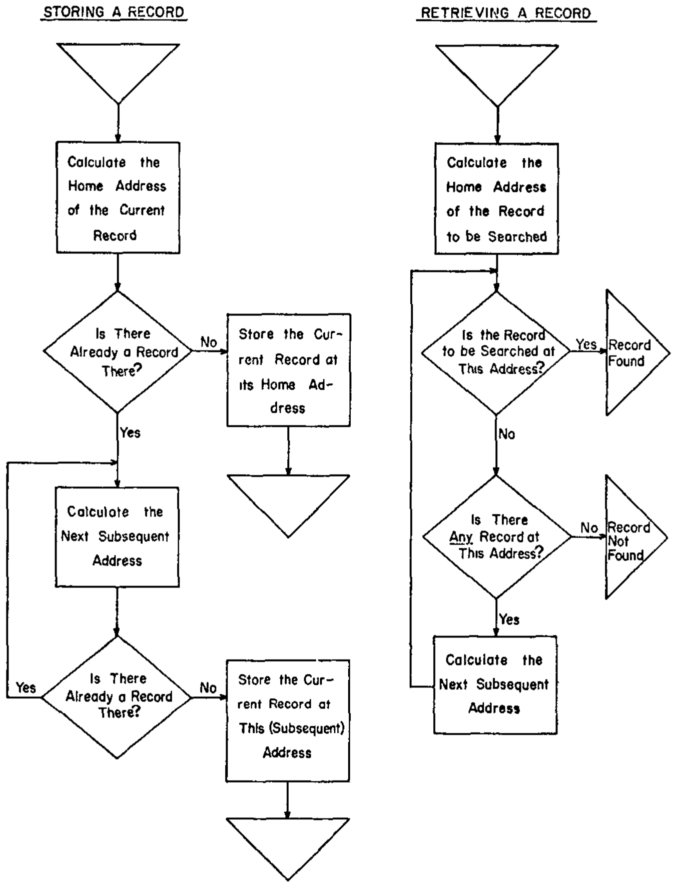
STORING A RECORD
Calculate the Home Address of the Current Record
Is There Already a Record There?
No
Store the Current Record at its Home Address
Yes
Calculate the Next Subsequent Address
Is There Already a Record There?
Yes
No
Store the Current Record at This (Subsequent) Address
RETRIEVING A RECORD
Calculate the Home Address of the Record to be Searched
Is the Record to be Searched at This Address?
Yes
Record Found
No
Is There Any Record at This Address?
No
Record Not Found
Yes
Calculate the Next Subsequent Address

FIG. 2. Storing and retrieving a record in a hash table in main memory.

subsequent addresses for the record whose key is K. In general, any hashing method involves a home address h(K) and a sequence of subsequent addresses, which we shall denote by $h_1(K)$, $h_2(K)$, and so on. Simplified flowcharts for storing and retrieving a record, given an arbitrary hashing function and collision handling method, are shown in Figure 2.

There is also a linear method of handling bucket overflow, known as *open addressing* (or *progressive overflow*), which works much the same way. If we are putting a new record in the table, and the home address for that record refers to a bucket which is entirely full of records, we try the next bucket. We will refer to the address of the next bucket, the next one after that, and so on, as the subsequent addresses $h_1(K)$, $h_2(K)$, and so on, just as before. Simplified flowcharts for storing and retrieving records in buckets, given arbitrary hashing and bucket overflow methods, are given in Figure 3 (page 12).

Just as there are several methods of constructing hashing functions, so there are various ways of handling collisions and handling overflow. The choice of method is influenced by the following:

1) Suppose we have just accessed a record with address α. How long does it take to access another record with address β; and, more importantly, in what ways does this timing depend on the relation between α and β? For example, if α and β are disk addresses, the time will be considerably shorter if they are on the same cylinder (for a movable-head disk), because then the access arm does not have to move. Similarly, if β immediately

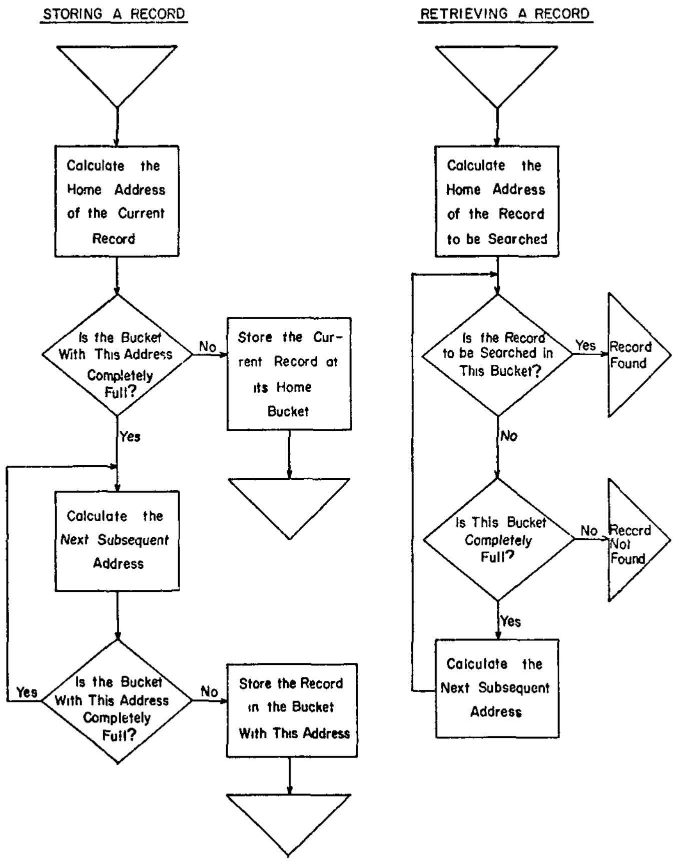
STORING A RECORD
Calculate the Home Address of the Current Record
Is the Bucket With This Address Completely Full?
No
Store the Current Record at its Home Bucket
Yes
Calculate the Next Subsequent Address
Is the Bucket With This Address Completely Full?
Yes
No
Store the Record in the Bucket With This Address
RETRIEVING A RECORD
Calculate the Home Address of the Record to be Searched
Is the Record to be Searched in This Bucket?
Yes
Record Found
No
Is This Bucket Completely Full?
No
Record Not Found
Yes
Calculate the Next Subsequent Address

FIG. 3. Storing and retrieving records in buckets.

follows α on a disk track, then it is much faster to get from α to β than it is, say, to get from β all the way around the track to α again.

2) Does it matter how long it takes to calculate the subsequent addresses? If our records are in auxiliary memory, the answer is almost certainly no. If they are in main memory, however, there arises the possibility that the improvement of one method over another in reducing the number of accesses is more than cancelled out by the increased calculation time for each of the subsequent addresses.

3) How well does our basic hashing method perform? We can improve the performance of a hashing method by using a good collision-handling method. In particular, if there are a number of synonyms of a particular record, certain methods allow us to find most of these synonyms by calculating only one subsequent address in each case. A similar phenomenon is *clustering*, in which certain records are not synonyms, but have hash codes in sequence, so that, if the linear method is used, they occupy the same space in the hash table as if they were synonyms.

We will now present a number of methods of handling collision and bucket overflow that have been used as alternatives to the linear and open addressing methods described above.

a) The overflow-area method. This is extremely simple, and is used when records are to be kept on disk or drum and the expected probability of overflow is low. It consists simply in having a general overflow bucket for *all* overflow records, or, sometimes, a separate such area for each cylinder on the disk.

b) The list method. In main memory, this consists of making a list (in Knuth's notation [14], a list with linked allocation) of all the synonyms of each record, and placing the head of this list at the home address of the given record. Items on such lists reside in a free storage area which is separate from the hash table and which, unlike the hash table, may be expanded in length without any necessity for reorganization. As in the case of the linear method, there is a bucket overflow handling method similar to this one, known as the *closed chaining method*, in which each bucket contains a pointer to that bucket which contains its overflow records.

c) The random method. We recall from the previous section that the random method of obtaining a home address also supplies us with subsequent addresses to be used if collision occurs. This has one immediate advantage over the list method and over the linear method, in that there is no need to make a linear search of a list of synonyms.

d) The quadratic method and its extensions. Suppose that our data are left-justified, blank-filled identifiers, and suppose that we have several of these in sequence and of maximum length (such as GENER1, GENER2, GENER3, and so on, for 6 bits per character and 36 bits per word). If we use the division method of hashing, the home addresses of these will also be in sequence, and, as we have noted, they will, in particular, all be different. However, if we use the linear method of handling collisions, we have to search through this cluster of records whenever we have a subsequent key whose home address falls *anywhere* within the cluster. The original quadratic method, devised by one of the authors of this paper [22], consists of calculating the subsequent address $h_i(K)$ as $h(K) + mi + ni^2$, for some fixed values of m and n. This device circumvents the difficulty mentioned above, although it does nothing about the problem of a large number of actual *synonyms* of a record. The quadratic method has been subject to numerous extensions ([1], [3], [4], [5], [7], [17], [29]), some of which are formulated to attack the synonym problem as well. Many of these extensions are included in the survey by Severance [32].

Whenever there are *deletions* from a hash table, the positions at which these occur cannot merely be reset to the same value they had originally (meaning "no record at this position"). The reason is that, when a search for another item encounters this record, the search will terminate, and the search routine will conclude, perhaps erroneously, that the given item is not to be found in the table. Instead, a second special code, signifying a deleted item, should be used.

The open method, as presented here, was mentioned by Peterson [27] and modified by Schay and Spruth [31]; an analysis of the modified method is carried out by Tainiter [33]. The overflow-area method is compared with the open method by Olsen [26]. Closed chaining is mentioned first by Johnson [13] and is compared to open chaining in a comparative study by Lum, Yuen, and Dodd [18] of a number of hashing methods in a file system context. The observation made here concerning deletions was noted by Morris [24].

THEORETICAL ANALYSES

A certain amount of common sense should be applied before any hashing method is used. For example, we have noticed that quite a number of hashing methods require that the size of the table (for binary addresses) be a power of 2. It is not hard to see, however, that this would be disastrous if we used the division method. The remainder upon dividing an arbitrary binary quantity by 2^k is simply the last k bits of that quantity. An extreme case happens when $k \leq 12$ and the keys are left-justified, blank-filled identifiers, with 6 bits per character and c characters per word; *all* identifiers of length less than or equal to c-2 would be given the same home address.

Similar difficulties can occur with other hashing methods, for slightly more involved reasons. The CDC 6000 series of computers do not have fixed-point divide instructions, but do have floating-point multiplication. If we use the midsquare method on such a computer, however, all one- and two-character identifiers (again left-justified and blank-filled) have the same last 48 bits, namely eight blanks. These 48 bits are the mantissa of a floating-point number on these computers—and the mantissa of the product of two floating-point quantities depends only on the *mantissas* of those quantities. The result is that all one- and two-character identifiers have the same hash code if this method is used. Much the same problem arises with any of the multiplication methods, although it may be circumvented by shifting the keys before multiplication.

In comparing the various hashing methods, there are two factors to be taken into account. One is the time taken to calculate the value of the hashing function, and the other is the expected number of subsequent addresses that are to be calculated for each record. For a record with no synonyms, of course, or with fewer synonyms than the number of records in a bucket, this expected number is zero. We have already mentioned the fact that, if our records are on disk, the first of these factors can often be ignored, because it is negligible in comparison with the second. That is, the calculation of even the most complex hashing function takes less time than a disk access.

Under these conditions, a thorough analysis of basic hashing methods has been made in a series of papers by Lum, Yuen, and Ghosh ([19], [20], [10]). The main results here are that, of all the commonly used methods, the division method and the random method minimize disk accesses the best, and furthermore that the division method is better than the random method even if the random method gives perfect randomization (which of course it never does). This last, rather startling, conclusion may be explained as follows. Key sets often contain runs of keys such as PRA, PRB, PRC, etc., or STR1, STR2, STR3, etc., which are in sequence. We have already seen that, if such a character string is of maximum length (that is, if it contains exactly as many characters as will fit into one word), the resulting keys will be in sequence, and will therefore, using the division method, have different hash codes. But now suppose that these strings do not have maximum length. If they are u bits long, and there are v bits per word, they will still be in sequence by multiples of $k = 2^{v-u}$; that is, they will be of the form K, K+k, K+$2k$, K+$3k$, etc. So long as k is relatively prime to the size of the table, all these keys will still have different hash codes, if the division method is used. This can always be brought about by choosing a *prime* number for the table size (this is also required, for a different reason, by the original quadratic search method of collision handling).

The most widely usable consequence of these results is the following. Suppose we are choosing a basic hashing method for records stored on a movable-head disk. Suppose our computer is an IBM 360 or 370 which does *not* have the optional DP (decimal divide) instruction. We have the binary divide (D) instruction, but recall that disk addresses on this machine are decimal quantities. The results of Ghosh and Lum now tell us that we should use the division method anyway—even if it means going over to a macro.

One important basic property of hash table methods is that they start working very badly when the hash table becomes almost full (unless the list method of handling collisions is used). In the extreme case in which the table is completely full except for *one* space, and the linear method of handling collisions is used, a search for this space takes, on the average, $N/2$ steps, for a table of size N. In practice, a hash table should never be allowed to get that full. The ratio of the number of entries currently in a hash table to the number of spaces for such entries is the *load(ing) factor* of the hash table; it ranges between 0 and 1. When the load factor is about 0.7 or 0.8—or, in other words, when the table is about 70% or 80% full—the size of the hash table should be increased, and the records in the table should be rehashed. The expected number of subsequent addresses that must be calculated in a hash table of size m, when n positions in the table are currently filled, and when the linear method of handling collisions is used, is given by

$$d(m, n) = \frac{1}{2}\left(2\frac{n-1}{m} + 3\frac{n-1}{m}\frac{n-2}{m} + 4\frac{n-1}{m}\frac{n-2}{m}\frac{n-3}{m} + \cdots\right)$$

(see Morris [24] or Knuth [15]). The load factor here is n/m; if this is held to the constant value α while m and n themselves go to infinity, the above quantity has the limiting value $\frac{1}{2}\alpha/(1 - \alpha)$.

When we use the list method of handling collisions, the performance of our hashing algorithm deteriorates slowly, rather than quickly, as the table fills. The list method, of course, puts no restrictions on the total number of records to be stored, which might even be greater than the size of the table, since synonyms of a record are kept in a separate free storage area. One method which was used on early UNIVAC 1107 software (in particular, the SLEUTH II assembler, under the EXEC II operating system), in fact, involved a *very* small hash table (64 positions), together with a much larger free storage space for lists. (Fold-shifting was used as the hashing function.) Under these conditions the average time taken to retrieve an item is $N/128$, where there are N items in the table (provided that N is large). For $N < 1000$, this compares favorably to binary searching (see the following section); larger values of N, in this context, were held to be too infrequent to matter much.

Comparison of the time of calculation of various hashing methods, in a context where this matters, is unfortunately completely machine-dependent. On many machines, it is actually impossible, because instruction timings are not published and may, in fact, vary from one execution to another, particularly if pipeline hardware is used. Fold-shifting is probably the fastest, followed by division (on machines that have divide instructions). The radix method and the algebraic coding method both suffer much more, in such a comparison, than they would if they were implemented in special-purpose hardware, and the same is true of conventional folding (as opposed to fold-shifting) and, to a lesser extent, of digit analysis.

Recent work of van der Pool ([35], [36]) and Webb [37] incorporates other parameters into the evaluation process to determine performance. Van der Pool includes in his analysis storage cost for the whole file, cost of storage for one record per unit time, fraction of the key set accessed per unit time, cost of access to the prime area, and cost of additional accesses. He derives formulas for calculating the total cost, with and without additions and deletions, but always limited to the case of separate overflow areas. His results show that loading factors higher than 1 may give better results than loading factors less than 1, particularly if wasted space is taken into account. (A loading factor higher than 1 is obtained if we divide the number of records in the table, *including those in overflow areas*, into the space available, *excluding space in overflow areas*.) Webb includes the computer CPU time in his evaluation of hash coding systems. He combines the various hashing functions with various collision handling methods to determine an overall cost. Basically, his results agree with those of Lum, Yuen, and Dodd [18].

ALTERNATIVES TO HASHING

There are many searching situations in which we either cannot or should not use hashing. We will now discuss some alternative methods of searching. It is not our purpose here to compare various alternative methods with each other (this is done in the surveys by Severance [32] and by Nievergelt [25]), but only to compare them with hashing.

First of all, *there is no order in a hash table.* The entries in the table are scattered around, seemingly at random (this is why we sometimes use the term "scatter storage" for a hash table). There is no way that we can go through a hash table and immediately print out all keys in order (without using a separate sorting process). If we want that capability, there are two methods which will still allow us to search our table relatively fast. The first is good for tables which do not change, or which change only infrequently, over time. We simply keep a *sorted array* and use a *binary search*, which repeatedly divides the table in half and determines in which half the given key is. A binary search of a table of N records, with two-way comparisons at each stage, takes $1 + \log_2 N$ steps [22] (and is therefore sometimes called a *logarithmic search*). The steps are short, so this is comparable with many of the hashing functions—though not with the fastest of them—even for a table which is quite large, as long as it remains in main memory. Insertion of a new item in such a table, however, takes $N/2$ steps on the average (since the table must remain sorted), although a group of new items which are already sorted can be inserted by a merging process which takes $N + N'$ steps, where there are N' new items.

The second method gives faster insertion, but also takes up more space. It is the *binary tree search*. A binary tree is constructed as in Figure 4. Each item contains a *left pointer* to a *left subtree* and a *right pointer* to a *right subtree*. Every item in the left subtree is less than every item in the right subtree, and the item which points to these two subtrees is between the two. Searching proceeds from top to bottom in the figure, and takes $\log_2 N$ steps, for a tree of N items which is *balanced* (that is, in which each left subtree is as large as the corresponding right subtree). Insertion of a new item, after we have searched the table unsuccessfully for it, only takes one step; we simply "hook it on the bottom" as shown in Figure 4. The main disadvantage of this method is the $2N$ additional pointers required.

Binary methods, in this form, do not work well when the records are in auxiliary memory. For a table of size $2^{10} = 1024$, for example, we would need to make 10 accesses to drum or disk, and this process is much too slow. A variation of the binary tree search, however, in which there is some larger number, k, of pointers at each level, rather than two, is widely used. For $1 \leq i < j \leq k$, each item in the ith subtree at any given level is less than each item in the jth subtree at that level. This, in fact, is the basic idea behind the *indexed sequential* (sometimes *index-sequential*) file organization method. Indexed sequential files are easy to process sequentially, in order, if this is desired, at the expense of a few more disk accesses for each individual record (almost never more than four, however).

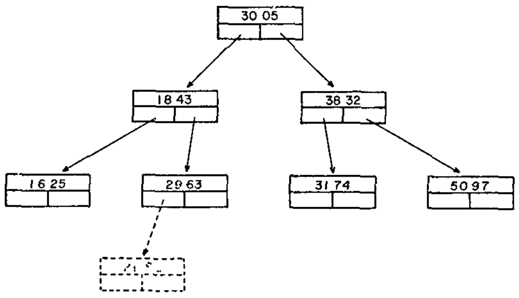

FIG. 4. Binary tree search and insertion.

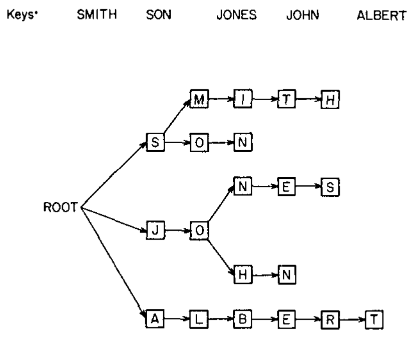

FIG. 5. The distributed key method.

Another method of searching, allied to the binary search, is the *distributed key method*. It is commonly used with keys which are strings of characters; a typical tree structure, as required for this method, is shown in Figure 5. At each level, there is a *variable* number of pointers, each of which corresponds to a character at that level. Searching proceeds by starting at the root and taking the branch corresponding to the first character in the key, then from there the branch corresponding to the second character in the key, and so on.

Linear searching, in which *every* record in the table is retrieved and checked, is still required for any type of searching which is not specifically provided for by the structure of the table. It often happens that we need to know which record R is such that some function $f(R)$ has its maximum value over all records. In a data base environment, for example, we might want to know which customer ordered the most of product X last year, which department spent the most on stationery, which supplier had the greatest total slippage, or which salesman has the best record on some specific item. This type of search always takes as many steps as there are records in the table, unless we know beforehand that we need certain specific information of this kind and build facilities for this into the program.

An important class of searching methods, sometimes called *direct methods*, may be used to advantage when the keys are highly nonrandom. It is generally advantageous, whenever the choice of values for the keys is within our control, to choose them in such a way that hashing becomes unnecessary. As an example, consider a personnel file with records stored by employee number. If the social security number of an employee is used as the key, some form of hashing is generally necessary. However, some organizations have their own employee numbering systems. In this case, one commonly encountered device is to keep an employee's record at a disk address which is itself the employee number. This, of course, allows us to make access to the disk *directly* (without hashing), although if such an organization gets a new computer it will probably have to renumber all its employees (which might not necessarily be bad).

Even when the choice of keys is not under our control, the keys might still be so nonrandom that a direct access method gives better results than a hashing method. A common example of this has to do with zip codes. If a mailing list is kept in zip-code order, and all the zip codes in the list have the same first three digits, it is not necessary to use hashing; we simply take the last two digits and use them directly as an index into a 100-position table of disk addresses. This superficially resembles hashing by digit analysis, but has the additional property that, since the first three digits are *always the same*, we have a file which can be accessed sequentially, as well as by the direct access method described above.

Usually, the situation is not this simple, but direct methods can still be used in many cases. Suppose that zip codes in a certain city range from a_1 to $a_1 + k_1$, and those in the suburbs lie in two other ranges, a_2 to $a_2 + k_2$, and a_3 to $a_3 + k_3$. In this case we can test a zip code to see which range it belongs in, subtract an appropriate constant, and directly obtain the index in a $(k_1 + k_2 + k_3 + 3)$-position table.

Finally, we must mention the all-important fact that these are *basic* techniques only. An improved solution to a specific problem may almost always be achieved by using a *combination* of techniques. We have already mentioned folding combined with division as a hashing method; it is also possible to combine hashing with binary tree search methods, or with direct methods. In particular, it is quite often true that a large proportion of a table will be accessible directly, with hashing used only for the exceptional cases. This would be true, for example, if (say) 80% of the zip codes appearing in a system were in a single city and its suburbs, as in the preceding example, with the remainder scattered around the country.

FURTHER AREAS OF STUDY

By far the greatest need for research in hashing at the time of this writing is for further empirical studies of the effects of varying hashing methods, collision and overflow handling methods, and alternatives to hashing in a wide variety of programming situations. One interesting paper along these lines is due to Ullman [34]. We should never forget the observational aspect of computer science, particularly since mathematical results in this area are always open to questions concerning the validity of the mathematical assumptions which must be made.

This is not to downplay the need for further mathematical studies. Lower bounds on search time as a function of other factors in a computational process would be greatly appreciated. Also, distributions of key sets, other than a uniform distribution, should be studied in order to determine, for each such distribution, a best hashing function. This would extend the work of Lum [20] which concerned itself solely with the uniform distribution.

Work of this kind, however, should always be subjected to a critical eye. One of the authors of this paper can remember vividly the attention that was showered recently on a new algorithm which performed a certain operation in one fewer multiplication than was previously thought necessary—at the expense of *ten* more addition and subtraction operations. Of course, it is perfectly true that, as the number of bits in the quantities being multiplied goes to infinity, one multiplication must always take longer than ten additions; in most practical cases, however, this will not be the case.

Another area of importance, which has more to do with development than with research, is in the making of plans for providing the next generation of computers with instructions which perform hashing directly. At the present time, we have instructions on many disk and drum units which will *find* a record in a bucket, given the bucket address and the key, and which will automatically read that record into main memory. It should be just as easy to develop instructions which will *calculate* the home address and the subsequent addresses as well. This is particularly true of the digit analysis method; one can easily imagine a hardware operation of removing certain digits from a key, as indicated by a digit analysis pattern which would be analogous to the IBM 360 and 370 editing patterns.

ACKNOWLEDGMENTS

The first author received partial support from the National Science Foundation under Grants GJ-41160 and DCR73-03431-A01. The authors would like to express their appreciation to V. Y. Lum for his thoughtful comments on an initial version of the paper, and to Stephen Chen for a careful reading of the final version.

REFERENCES

[1] ACKERMAN, A. F. "Quadratic search for hash tables of size p^n," *Comm. ACM* **17**, 3 (March 1974), p. 164.

[2] BAYS, C "The reallocation of hash-coded tables," *Comm. ACM* **16**, 1 (Jan. 1973), pp. 11–14.

[3] BELL, J. R. "The quadratic quotient method: a hash code eliminating secondary clustering," *Comm. ACM* **13**, 2 (Feb. 1970), pp. 107–109.

[4] BELL, J. R.; AND KAMAN, C. H. "The linear quotient hash code," *Comm ACM* **13**, 11 (Nov. 1970), pp. 675–677.

[5] BRENT, R. P "Reducing the retrieval time of scatter storage techniques," *Comm. ACM* **16**, 2 (Feb 1973), pp. 105–109

[6] BUCHHOLZ, W. "File organization and addressing," *IBM Systems J.* 2 (June 1963), pp. 86–111.

[7] DAY, A. C. "Full table quadratic searching for scatter storage," *Comm. ACM* **13**, 8 (August 1970), pp. 481–482.

[8] DIPPEL, G.; AND HOUSE, W. C. *Information systems*, Scott, Foresman & Co , Glenview, Ill , 1969.

[9] DUMEY, A. I. "Indexing for rapid random-access memory systems," *Computers and Automation* **5**, 12 (Dec. 1956), pp. 6–9

[10] GHOSH, S. P.; AND LUM, V Y. "An analysis of collisions when hashing by division," Tech. Report RJ-1218, IBM, May 1973.

[11] HANAN, M.; AND PALERMO, F. P. "An application of coding theory to a file address problem," *IBM J. Res. & Development* **7**, 2 (April 1963), pp. 127–129.

[12] HELLERMAN, H. *Digital computer system principles*, McGraw-Hill, New York, 1967.

[13] JOHNSON, L. R. "An indirect chaining method for addressing on secondary keys," *Comm ACM* **4**, 5 (May 1961), pp. 218–222.

[14] KNUTH, D. E. *The art of computer programming*, Vol. III: *Sorting and searching*, Addison-Wesley, Reading, Mass , 1973.

[15] KNUTH, D. E. "Computer science and its relation to mathematics," *Amer. Math. Monthly* **81**, 4 (April 1974), pp 323–343.

[16] LIN, A. D. "Key addressing of random access memories by radix transformation," *Proc 1963 Spring Joint Computer Conf.* AFIPS Vol. 23, Spartan Books, Baltimore, 1963, pp. 355–366.

[17] LUCCIO, F. "Weighted increment linear search for scatter tables," *Comm. ACM* **15**, 12 (Dec. 1972), pp. 1045–1047.

[18] LUM, V. Y.; YUEN, P. S. T.; AND DODD, M. "Key-to-address transform techniques. a fundamental performance study on large existing formatted files," *Comm. ACM* **14**, 4 (April 1971), pp. 228–239.

[19] LUM, V Y.; AND YUEN, P. S. T. "Additional results on key-to-address transform techniques," *Comm. ACM* **15**, 11 (Nov 1972), pp. 996–997.

[20] LUM, V. Y. "General performance analysis of key-to-address transformation methods using an abstract file concept," *Comm. ACM* **16**, 10 (Oct 1973), pp. 603–612.

[21] MAURER, W. D. "An improved hash code for scatter storage," *Comm. ACM* **11**, 1 (Jan. 1968), pp 35–38.

[22] MAURER, W D. *Programming*, Holden-Day, San Francisco, Calif., 1968.

[23] MCILROY, M. D. "A variant method of file searching," *Comm. ACM* **6**, 3 (March 1963), p. 101.

[24] MORRIS, R. "Scatter storage techniques," *Comm. ACM* **11**, 1 (Jan. 1968), pp. 38–43.

[25] NIEVERGELT, J. "Binary search trees and file organization," *Computing Surveys* **6**, 3 (Sept. 1974), pp. 195–207.

[26] OLSON, C. A. "Random access file organization for indirectly accessed records," *Proc. ACM 24th National Conf.*, 1969, pp. 539–549.

[27] PETERSON, W. W. "Addressing for random-access storage," *IBM J. Res. & Development* **1**, 2 (April 1957), pp. 130–146.

[28] PRICE, C. E. "Table lookup techniques," *Computing Surveys* **3**, 2 (June 1971), pp. 49–65.

[29] RADKE, C. E. "The use of quadratic residue research," *Comm. ACM* **13**, 2 (Feb. 1970), pp. 103–105.

[30] SCHAY, G.; AND RAVER, N. "A method for key-to-address transformation," *IBM J. Res. & Development* **7**, 2 (April 1963), pp 121–126.

[31] SCHAY, G.; AND SPRUTH, W. G. "Analysis of a file addressing method," *Comm. ACM* **5**, 8 (August 1962), pp. 459–462.

[32] SEVERANCE, D. G. "Identifier search mechanisms: a survey and generalized model," *Computing Surveys* **6**, 3 (Sept. 1974), pp. 175–194.

[33] TAINITER, M. "Addressing for random-access storage with multiple bucket capacities," *J. ACM* **10**, 3 (July 1963), pp. 307–315.

[34] ULLMAN, J. D. "A note on the efficiency of hashing functions," *J. ACM* **19**, 3 (July 1972), pp. 569–575.

[35] VAN DER POOL, J. A. "Optimum storage allocation for initial loading of a file," *IBM J. Res. & Development* **16**, 6 (Nov. 1972), pp. 579–586.

[36] VAN DER POOL, J. A. "Optimum storage allocation for a file in steady state," *IBM J. Res. & Development* **17**, 1 (Jan. 1973), pp. 27–38.

[37] WEBB, D. A. "The development and application of an evaluation model for hash coding systems," PhD Thesis, Syracuse Univ., Syracuse, N. Y., August 1972.

[38] WEGNER, P. *Programming languages, information structures, and machine organization*, McGraw-Hill, New York, 1968.