Algorithms and Data Structures

Daniel Sleator Editor

Dynamic Hash Tables

Per-Åke Larson

ABSTRACT: Linear hashing and spiral storage are two dynamic hashing schemes originally designed for external files. This paper shows how to adapt these two methods for hash tables stored in main memory. The necessary data structures and algorithms are described, the expected performance is analyzed mathematically, and actual execution times are obtained and compared with alternative techniques. Linear hashing is found to be both faster and easier to implement than spiral storage. Two alternative techniques are considered: a simple unbalanced binary tree and double hashing with periodic rehashing into a larger table. The retrieval time of linear hashing is similar to double hashing and substantially faster than a binary tree, except for very small trees. The loading times of double hashing (with periodic reorganization), a binary tree, and linear hashing are similar. Overall, linear hashing is a simple and efficient technique for applications where the cardinality of the key set is not known in advance.

1. INTRODUCTION

Several dynamic hashing schemes for external files have been developed over the last few years [2, 4, 9, 10]. These schemes allow the file size to grow and shrink gracefully according to the number of records actually stored in the file. Any one of the schemes can be used for internal hash tables as well. However, the two methods best suited for internal tables seem to be linear hashing [9] and spiral storage [10]: they are easy to implement and use little extra storage. This paper shows how to adapt these two methods to internal hash tables, mathematically analyzes their expected performance, and reports experimental performance results. The performance is also compared with that of more traditional solutions for handling dynamic key sets. Both methods are found to be efficient techniques for applications where the cardinality of the key set is not known in advance. Of the two, linear hashing is faster and also easier to implement.

An inherent characteristic of hashing techniques is that a higher load on the table increases the cost of all basic operations: insertion, retrieval and deletion. If the performance of a hash table is to remain within acceptable limits when the number of records increases, additional storage must somehow be allocated to the table. The traditional solution is to create a new, larger hash table and rehash all the records into the new table. The details of how and when this is done can vary. Linear hashing and spiral storage allow a smooth growth. As the number of records increases, the table grows gradually, one bucket at a time. When a new bucket is added to the address space, a limited local reorganization is performed. There is never any total reorganization of the table.

2. LINEAR HASHING

Linear hashing was developed by W. Litwin in 1980 [9]. The original scheme is intended for external files. Several improved versions of linear hashing have been proposed [5, 7, 13, 14]. However, for internal hash tables their more complicated address calculation is likely to outweigh their benefits.

Consider a hash table consisting of N buckets with addresses $0, 1, \ldots, N-1$. Linear hashing increases the address space gradually by splitting the buckets in a predetermined order: first bucket 0, then bucket 1, and so on, up to and including bucket $N-1$. Splitting a bucket involves moving approximately half of the records from the bucket to a new bucket at the end of the table. The splitting process is illustrated in Figure 1 for an example file with $N = 5$. A pointer p keeps track of the next bucket to be split. When all N buckets have been split and the table size has doubled to $2N$, the pointer is reset to zero and the splitting process starts over again. This time the pointer travels from 0 to $2N-1$, doubling the table size to $4N$. This expansion process can continue as long as required.

Figure 2 illustrates the splitting of bucket 0 for an example table with $N = 5$. Each entry in the hash table contains a single pointer, which is the head of a linked list containing all the records hashing to that address. When the table is of size 5, all records are hashed by $h_0(K) = K \bmod 5$. When the table size has doubled to 10, all records will be addressed by $h_1(K) = K \bmod 10$. However, instead of doubling the table size immediately, we expand the table one bucket at a time as required. Consider the keys hashing to bucket 0 under $h_0(K) = K \bmod 5$. To hash to 0 under $h_0(K) = K \bmod 5$, the last digit of the key must be either 0 or 5. Under the hashing function $h_1(K) = K \bmod 10$, keys with a last digit of 0 still hash to bucket 0, while those with a last digit of 5 hash to bucket 5. Note that none of the keys hashing to buckets 1, 2, 3 or 4 under h_0 can possibly hash to bucket 5 under h_1. Hence, to expand the table, we allocate a new bucket (with address 5) at the end of the table, increase the pointer p by one, and scan through the records of bucket 0, relocating to the new bucket those hashing to 5 under $h_1(K) = K \bmod 10$.

The current address of a record can be found as follows. Given a key K, we first compute $h_0(K)$. If $h_0(K)$ is less than the current value of p, the corresponding bucket has already been split, otherwise not. If the bucket has been split, the correct address of the record is given by $h_1(K)$. Note that when all the original buckets (buckets 0–4) have been split and the table size has increased to 10, all records are addressed by h_1.

Returning to the general case, the address computation can be implemented in several ways. For internal hash tables the following solution appears to be the simplest. Let g be a normal hashing function producing addresses in some interval $[0, M]$. M should be sufficiently large, say $M > 2^{20}$. To compute the address of a record we use the following sequence of hashing functions:

$$h_j(K) = g(K) \bmod (N \times 2^j), \quad j = 0, 1, \ldots$$

where N is the minimum size of the hash table. If N is of the form 2^k, the modulo operation reduces to extracting the last $k + j$ bits of $g(K)$. The hashing function $g(K)$ can also be implemented in several ways. Functions of the type

$$g(K) = (cK) \bmod M,$$

where c is a constant and M is a large prime have experimentally been found to perform well [12]. Different hashing functions are easily obtained by choosing different values for c and M, thus providing a "tuning" capability. By choosing $c = 1$, the classical division-remainder function is obtained. There is also some theoretical justification for using this class of functions. As shown in [1], it comes within a factor of two of being *universal*$_2$.

We must also keep track of the current state of the hash table. This can be done by two variables:

L	number of times the table size has doubled (from its minimum size N), $L \geq 0$.
p	pointer to the next bucket to be split, $0 \leq p < N \times 2^L$.

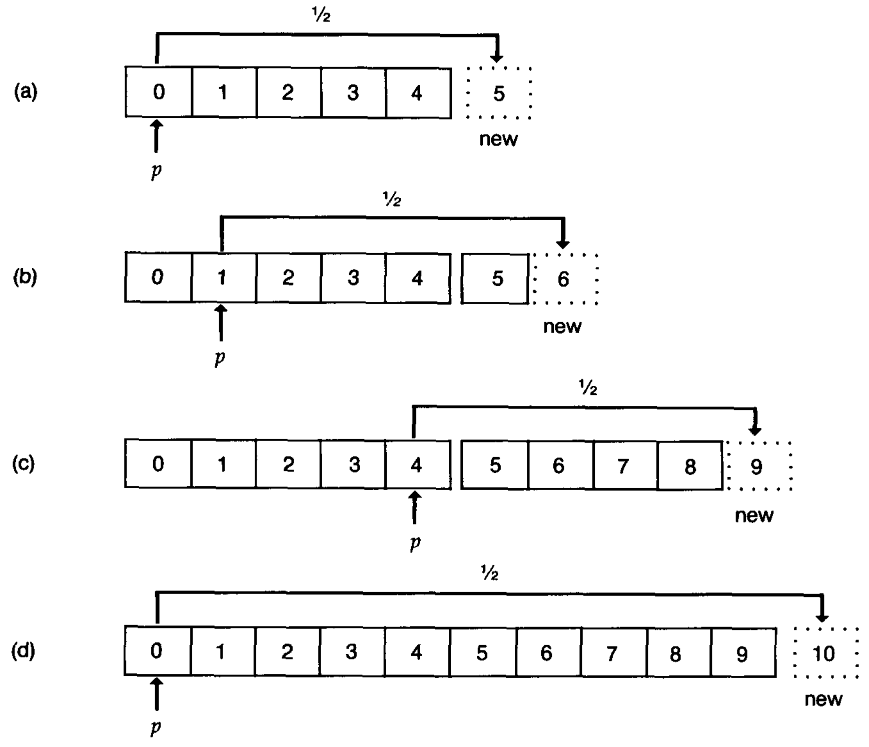

FIGURE 1. Illustration of the Expansion Process of Linear Hashing

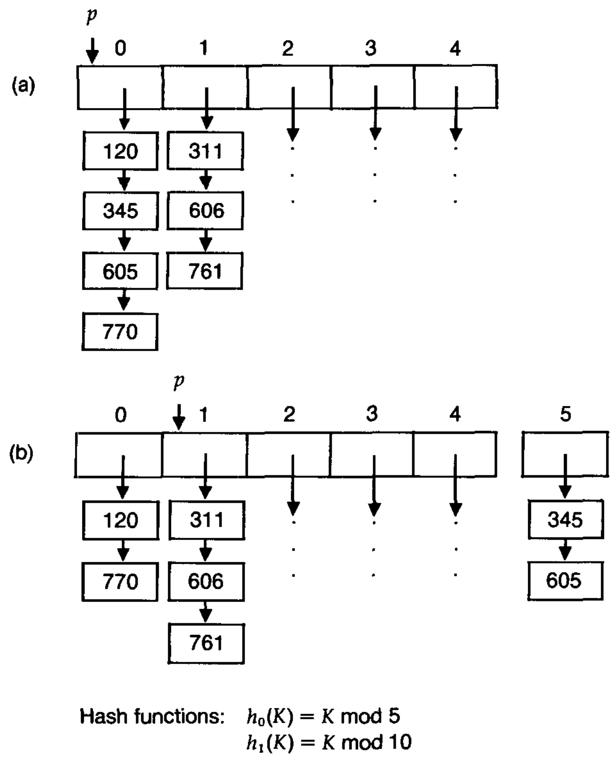

FIGURE 2. An Example of Splitting a Bucket

When the table is expanded by one bucket, these variables are updated as follows:

```
p := p + 1;
if p = $N \times 2^L$ then begin
  L := L + 1;
  p := 0;
end;
```

Given a key K, the current address of the corresponding record can be computed simply as

```
addr := $h_L(K)$;
if addr < p then addr := $h_{L+1}(K)$;
```

Contracting the table by one bucket is exactly the inverse of expanding it by one bucket. First the state variables are updated as follows:

```
p := p − 1;
if p < 0 then begin
  L := L − 1;
  p := $N \times 2^L - 1$;
end;
```

Then all the records of the last bucket are moved to the bucket pointed to by p, and the last bucket can be freed.

So far we have only discussed *how* to expand or contract the table but not *when* to do so. The key idea is to keep the overall load factor bounded. The overall load factor is defined as the number of records in the table divided by the (current) number of buckets. In our case, the overall load factor equals the average chain length. We fix a lower and an upper bound on the overall load factor and expand (contract) the table whenever the overall load factor goes above (below) the upper (lower) bound. This requires that we keep track of the current number of records in the table, in addition to the state variables L and p.

2.1 Data Structure and Algorithms

The basic data structure required is an expanding and contracting pointer array. However, few programming languages directly support dynamically growing arrays. The simplest way to implement such an array is to use a two-level data structure, as illustrated in Figure 3. The array is divided into segments of fixed size. When the array grows, new segments are allocated as needed. When the array shrinks and a segment becomes superfluous, it can be freed. A directory keeps track of the start address of each segment in use.

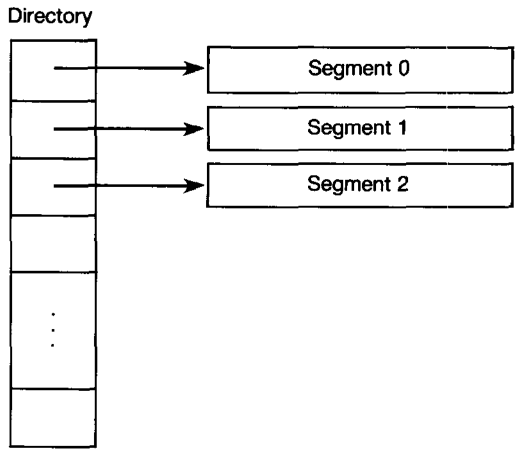

FIGURE 3. Data Structure Implementing a Dynamic Array

The only statically allocated structure is the directory. It is most convenient to let the minimum table size correspond to one segment. If the directory size and the segment size are both a power of two, the offset within the directory and the offset within a segment can be computed from the bucket address by masking and shifting. A directory size and segment size of 256 gives a maximum address space of $256 \times 256 = 64k$ buckets. As we shall see, an overall load factor of 5 is quite reasonable. This allows storage of over 300,000 records in the table, which seems adequate for most applications.

Pascal type declarations for a linear hash table using the proposed two-level data structure are given below. The directory is simply an array of pointers to segments and each segment is an array of pointers to a linked list of elements. Each element contains a record and a pointer to the next element. Some computation is saved by keeping track of the value $N \times 2^L$ instead of the value of L. The field *maxp* is used for this purpose. The use of other fields should be clear from the field names.

keylength, segmentsize, and *directorysize* are assumed to be globally declared constants.

```
type
  elementptr = ↑element;
  element =
    record
      key: keytype ;
      {Insert definitions of additional fields here}
      next: elementptr ;
    end ;
  segment =
    array [0 .. segmentsize − 1] of elementptr ;
  segmentptr = ↑segment ;
  hashtable =
    record
      p: integer ;
        {Next bucket to be split}
      maxp: integer ;
        {Upper bound on p during this expansion}
      keycount: integer ;
        {Number of records in the table}
      currentsize: integer ;
        {Current number of buckets}
      minloadfctr,
        {Lower and}
      maxloadfctr: real ;
        {upper bound on the load factor}
      directory:
        array [0 .. directorysize − 1] of segmentptr ;
    end;
```

A Pascal implementation of the hashing function is given below. If the keys are alphanumeric, the key value must first be converted into an integer. The function *convertkey* is assumed to perform this conversion. The hashing function g discussed above is implemented as $g(K) = K \bmod 1048583$. This implementation of $g(K)$ was used in the experiments reported in Section 4.

```
function hash
  (K: keytype ; T: hashtable): integer ;
const prime = 1048583 ;
var h, address: integer ;
begin
  h := convertkey(K) mod prime ;
  address := h mod T.maxp ;
  if address < T.p
    then address := h mod(2*T.maxp) ;
  hash := address ;
end;
```

Given the address computation algorithm, retrieval and insertion of an element is straightforward. The first element on the chain to which a record with key value K belongs can be located as follows:

```
address := hash (K, T);
currentsegment := T.directory[address div segmentsize];
segmentindex := address mod segmentsize;
firstonchain := currentsegment↑[segmentindex];
```

The procedure below expands the table by one bucket, creating a new segment when needed. It consists of three main parts. The first part locates the bucket to be split and the new bucket. The second part adjusts the state variables. The third part is a loop scanning down the chain of the "old" bucket, and moving records to the new bucket as necessary.

```
procedure expandtable (var T: hashtable) ;
var newaddress, oldsegmentindex, newsegmentindex: integer ;
  oldsegment, newsegment: segmentptr ;
  current, previous: elementptr ;
    {for scanning down the old chain}
  lastofnew: elementptr ;
    {points to the last element of the new chain}
begin
with T do
  {Reached maximum size of address space?}
  if maxp + p < directorysize * segmentsize then begin
    {Locate the bucket to be split}
    oldsegment := directory [p div segmentsize] ;
    oldsegmentindex := p mod segmentsize ;
    {Expand address space, if necessary create a new segment}
    newaddress := maxp + p ;
    if newaddress mod segmentsize = 0
      then new (directory [newaddress div segmentsize]) ;
    newsegment := directory[newaddress div segmentsize] ;
    newsegmentindex := newaddress mod segmentsize ;
    {Adjust the state variables}
    p := p + 1 ;
    if p = maxp then begin
      maxp := 2 * maxp ;
      p := 0 ;
    end ;
    currentsize := currentsize + 1 ;
    {Relocate records to the new bucket}
    current := oldsegment↑[oldsegmentindex] ;
    previous := nil ;
    lastofnew := nil ;
    newsegment↑[newsegmentindex] := nil ;
    while current <> nil do
      if hash(current↑.key, T) = newaddress
      then begin   {attach it to the end of the new chain}
        if lastofnew = nil
          then newsegment↑[newsegmentindex] := current
          else lastofnew↑.next := current ;
        if previous = nil
          then oldsegment↑[oldsegmentindex] := current↑.next
          else previous↑.next := current↑.next ;
        lastofnew := current ;
        current := current↑.next ;
        lastofnew↑.next := nil ;
      end
      else begin   {leave it on the old chain}
        previous := current ;
        current := current↑.next ;
      end ;
  end ;
end ;
```

2.2 Analysis
In this section we analyze the expected performance of a growing linear hash table under the assumption that the table is expanded as soon as the overall load factor exceeds α, $\alpha > 0$. For a large table, the overall load factor will be (almost) constant and equal to α. It is also assumed that there are no deletions. The analysis is asymptotic and similar to the analysis in [6].

The expected cost of retrieval and insertion depends on what fraction of the buckets has already been split during the current expansion. The performance is best at the end of an expansion because the load is uniform over the whole table. The performance varies cyclically where a cycle corresponds to a doubling of the table. Hence each cycle is twice as long as the previous cycle.

A linear hash table can be viewed as consisting of two traditional hash tables: (1) the buckets that have not yet been split during the current expansion, and (2) the buckets that have been split plus the new buckets created during the current expansion. Within each part, the expected load is the same for every bucket. During an expansion the relative sizes of the two parts change. Consider a traditional hash table with load factor y, where records are stored using chaining as explained in the previous section. The expected number of key comparisons for a successful search and an unsuccessful search in such a table are given by [3]

$$s(y) = 1 + y/2$$
$$u(y) = y.$$

Let x, $0 \le x \le 1$, denote the fraction of buckets that have been split during the current expansion and z, the expected number of records in an unsplit bucket. The expected number of records in a split or new bucket is then $z/2$. For the overall load factor to be equal to α, the following relationship between z and x must hold:

$$2xz/2 + (1 - x)z = \alpha(2x + 1 - x).$$

The left-hand side represents the expected number of records in a group, where a group consists of either one unsplit bucket (with load z) or one unsplit bucket plus one new bucket (each with load $z/2$). The expression $2x + 1 - x$ represents the number of bucket addresses allocated to a group. Solving for z gives the relationship

$$z = \alpha(1 + x).$$

In other words, the expected number of records in an unsplit bucket grows linearly from α to 2α. Let $S(\alpha, x)$ denote the expected number of key comparisons for a successful search when a fraction x of the buckets have been split. With probability x we hit a split group, in which case the expected search length is $s(\alpha(1 + x)/2)$. With probablity $1 - x$ we hit an unsplit group, in which case the expected search length is $s(\alpha(1 + x))$. Hence, we have

$$S(\alpha, x) = xs\left(\frac{\alpha(1 + x)}{2}\right) + (1 - x)s(\alpha(1 + x))$$
$$= 1 + \frac{\alpha}{4}(2 + x - x^2).$$

Similarly, the expected length of an unsuccessful search is obtained as

$$U(\alpha, x) = xu\left(\frac{\alpha(1 + x)}{2}\right) + (1 - x)u(\alpha(1 + x))$$
$$= \frac{\alpha}{2}(2 + x - x^2).$$

The minimum expected search lengths occur when the load is uniform over the whole table, that is, when $x = 0$ or $x = 1$. In this case we have

$$S(\alpha, 0) = 1 + \frac{\alpha}{2}$$
$$U(\alpha, 0) = \alpha.$$

The maximum expected search lengths occur when $x = ½$. For $x = ½$ we have

$$S\left(\alpha, \frac{1}{2}\right) = 1 + \frac{\alpha}{2}\frac{9}{8}$$
$$U\left(\alpha, \frac{1}{2}\right) = \alpha\frac{9}{8}.$$

The average search cost over a cycle can be computed by integrating the expected search length over an expansion cycle, which results in

$$\bar{S}(\alpha) = \int_0^1 S(\alpha, x)\,dx = 1 + \frac{\alpha}{2}\frac{13}{12}$$
$$\bar{U}(\alpha) = \int_0^1 U(\alpha, x)\,dx = \alpha\frac{13}{12}.$$

An insertion consists of two parts: the actual insertion of the new record at the end of a chain and, for some insertions, an expansion of the table. The cost of the actual insertion is the same as the cost of an unsuccessful search plus the cost of connecting the new record to the end of the chain. A fraction $1/\alpha$ of the insertions trigger an expansion. An expansion involves scanning down a chain of records belonging to an unsplit bucket. The expected number of records on the chain is $\alpha(1 + x)$, and for each record we must compute a hash address and update a few pointers. The expected number of extra hash address computations per record inserted is then given by

$$A(\alpha, x) = \frac{1}{\alpha}\alpha(1 + x) = 1 + x$$
$$\bar{A}(\alpha) = \int_0^1 A(\alpha, x)\,dx = 1.5.$$

3. SPIRAL STORAGE

When using linear hashing the expected cost of retrieving, inserting or deleting a record varies cyclically. Spiral storage [10] overcomes this undesirable feature and exhibits uniform performance regardless of the table size. In other words, the table may grow or shrink by any factor but the expected performance always remains the same. Spiral storage intentionally distributes the records unevenly over the table. The load is high at the beginning of the (active) address space and tapers off towards the end as illustrated in Figure 4. To expand the table additional space is allocated at the end of the address space, and at the same time a smaller amount of space is freed at the beginning. The records stored in the bucket that disappears are distributed over the new buckets.

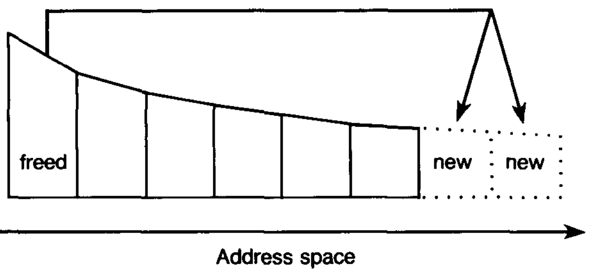

FIGURE 4. Illustrating the Load Distribution and the Expansion Process of Spiral Storage

The address computation of spiral storage is illustrated in Figure 5. The key value is first mapped into a real number x in the interval $[S, S + 1)$. The x-value is the mapped into an address y by the function $y = \lfloor d^x \rfloor$, where $d > 1$ is a constant called the growth factor. The function d^x is called the expansion (or growth) function. The currently active address space extends from $\lfloor d^S \rfloor$ to $\lceil d^{S+1} \rceil - 1$. This equals approximately $d^S(d - 1)$ active addresses. Spiral storage requires a hashing function that maps keys uniformly into $[0, 1)$, that is, $0 \leq h(K) < 1$. The value $h(K)$ is then mapped into a value x in $[S, S + 1)$. This value is uniquely determined by requiring that its fractional part must agree with $h(K)$. It is most easily computed as $x = \lceil S - h(K) \rceil + h(K)$. The final address is computed as $y = \lfloor d^x \rfloor$. Hence every key is mapped into an address in the active address space.

To increase the active address space we simply increase S to S', see Figure 5. The keys that previously mapped into the range $[S, S')$ now map into the range $[S + 1, S' + 1)$. The new address range corresponding to $[S + 1, S' + 1)$ is approximately d times the old address range corresponding to $[S, S')$. The records stored in the bucket(s) that disappears are relocated to the new buckets and the expansion is complete. The value S' is normally chosen so that exactly one bucket disappears.

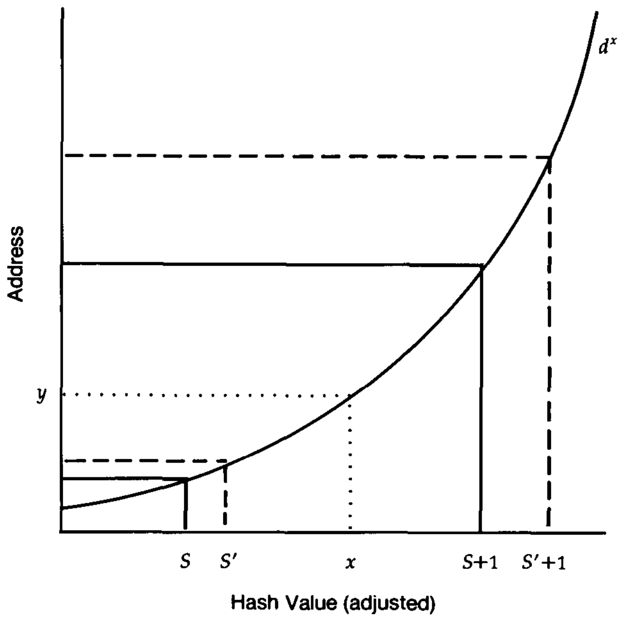

FIGURE 5. The Address Computation of Spiral Storage

Let us illustrate the above discussion by a small example, see Table I. We start from an active address space of 5 addresses. The growth factor is $d = 2$, which is the most convenient value for internal hash tables. The value of S required to give 5 active addresses can be determined from the equation $2^{S+1} - 2^S = 5$, which gives $S = \log_2 5 = 2.3219$. The first active address is then $y = \lfloor 2^{2.3219} \rfloor = 5$ and the last one, $y = \lceil 2^{3.3219} \rceil - 1 = 9$. In this situation all keys with $0.3219 \leq h(K) < 0.5849$ are stored in bucket 5, all those with $0.5849 \leq h(K) < 0.8074$ in bucket 6, . . . , and all those with $0.1699 \leq h(K) < 0.3219$ in bucket 9. For example, if $h(K) = 0.75$, the x-value is $x = \lceil 2.3219 - 0.75 \rceil + 0.75 = 2.75$, which gives the address $y = \lfloor 2^{2.75} \rfloor = 6$. The resulting load distribution is given in column three of Table I. Bucket 5 is expected to receive 26.3 percent of the records, bucket 6, 22.2 percent, and so on.

TABLE I. Address mapping for a small table, d = 2

Address	Hash interval	Relative load
5	[0.3219, 0.5849)	0.263
6	[0.5849, 0.8074)	0.222
7	[0.8074, 1.0000)	0.193
8	[0.0000, 0.1699)	0.170
9	[0.1699, 0.3219)	0.152
10	[0.3219, 0.4594)	0.137
11	[0.4594, 0.5849)	0.126

To increase the active address space by one, S is increased to $S' = \log_2 6 = 2.5849$. The last active address is now $\lceil 2^{3.5849} \rceil - 1 = 11$, that is, addresses 10 and 11 are now taken into use. All records in bucket 5 are relocated to the two new buckets. All records in bucket 5 with $h(K) < 0.4594$ are moved to bucket 10, and all those with $h(K) \geq 0.4594$ are moved to bucket 11.

Note that the active address space slides to the right as it increases. Addresses up to $\lfloor d^S \rfloor - 1$ are unused. There is a rather simple way of mapping the "logical" addresses of spiral storage into "physical" addresses so that the "physical" address space always begins from address zero [10, 11]. However, the cost of computing the "physical" address grows with the table size. If this mapping is used, the performance would therefore depend on the table size. By using the same two-level data structure as for linear hashing we can avoid this extra cost.

Choosing $d = 2$ gives the simplest scheme; every expansion creates two new buckets and deletes one bucket. From now on it is assumed that $d = 2$. The most expensive part of the address calculation is the computation of 2^x. A function of this type is normally computed by approximating it with a polynomial of a fairly high degree. Fortunately, most programming languages have built-in library routines for this type of computation, thus hiding the complexity from a "normal" user. However, it is still a fairly expensive operation. To reduce the cost we can approximate 2^x by some other function that is faster to compute. Let $f(x)$ be a function that approximates 2^x in the interval $0 \le x \le 1$. Then 2^x can be approximated in the interval $n \le x \le n+1$ (integer n) by the function $2^n f(x-n)$. Note that n is just the integer part of x and $x - n$ is the fractional part. Hence the problem is reduced to approximating 2^x for $0 \le x \le 1$. Martin [10] suggested using a function of the form

$$f(x) = \frac{a}{b-x} + c, \quad 0 \le x \le 1.$$

The values of the parameters a, b and c can be determined by fixing the value of $f(x)$ at three points. To guarantee that the function $2^n f(x-n)$ is continuous, the function values at the two end points must be exact. This gives two conditions: $f(0) = 1$ and $f(1) = 2$. The third point can be any point in [0, 1]. Some numerical experimentation showed that choosing $x = 0.5$ gives a good approximation. The maximum error is approximately 0.0023. This gives the condition $f(0.5) = 2^{0.5}$. From these three conditions we get the following parameter values:

$$b = (2-\sqrt{2})/(3-2\sqrt{2}) \approx 3.4142136$$

$$a = b(b-1) \approx 8.2426407$$

$$c = 2 - b \approx -1.4142136.$$

The inverse of the function $y = a/(b-x) + c$ is needed to compute the new value for S after an expansion. The inverse is

$$x = b - a/(y-c).$$

Instead of using the function above we could use a second degree polynomial. However, evaluation of a second degree polynomial requires four floating point operations while the function above requires only three.

It should be emphasized that there are many possible expansion functions. The expansion function 2^x has the property that the expansion rate is constant; one old bucket is always replaced by two new buckets. The expected performance is also constant. Other expansion functions do not necessarily have this property. The function $2^n f(x-n)$ where $f(x) = a/(b-x) + c$ should therefore be seen as a different expansion function, not as an approximation of 2^x. However, the resulting performance is very close to the performance obtained when using 2^x. The expansion rate is almost, but not exactly, constant. Most of the time an expansion creates two new buckets, but occasionally either one or three buckets are created.

3.1 Date Structure and Algorithms

The two-level data structure proposed for linear hash tables is also appropriate for spiral storage. Two minor modifications are required because the active address space moves to the right as the table size increases. When expanding the table, we can occasionally delete a segment and free a slot in the directory. When reaching the end of the directory, there will therefore be free slots in the beginning of the directory. To make use of these, we just "wrap around" and continue until all slots are in use.

The required type declarations are given below. The constants a, b, and c are assumed to be globally declared. Elements and segments are exactly the same as for linear hashing. There are two changes in the declaration of *hashtable*. The fields *lowaddr* and *highaddr* replace the field *currentsize*. The fields *y*0 and *x*0 replace *p* and *maxp*. Their use is explained in connection with the address computation algorithm.

```
type
  {Type definitions for elements and segments are the same as
    for linear hashing}
  hashtable =
    record
      y0: integer ;
        {See the text for an explanation}
      x0: real ;
        {of the use of these fields}
      lowaddr,
        {First address and}
      highaddr: integer ;
        {last address in current address space}
      keycount: integer
        {Number of records stored}
      minloadfctr,
        {Lower and}
      maxloadfctr: real ;
        {upper bound on the load factor}
      directory:
        array [0 .. directorysize − 1] of segmentptr ;
    end ;
```

Recall from the discussion of address computation that the beginning (and the size) of the active address space is determined by the state variable S. Let $f(x)$ denote the expansion function and l the desired (or current) lowest address. Then the following equality must hold: $l = f(S)$, that is, $S = f^{-1}(l)$. Using the proposed expansion function we obtain

$$l = 2^{\lfloor S \rfloor}(a/(b - \{S\}) + c)$$

where $\{S\}$ denotes the fractional part of S. From the way this function was constructed we also know that the following inequality must be satisfied at all times:

$$1 \le a/(b - \{S\}) + c \le 2.$$

This gives the following inequalities:

$$1 \le l/2^{\lfloor S \rfloor} \le 2$$

$$2^{\lfloor S \rfloor} \le l \le 2^{\lfloor S \rfloor + 1}.$$

In other words, the integer part of S, $\lfloor S \rfloor$, is just the highest integer value such that $2^{\lfloor S \rfloor} \le l$. Having determined $\lfloor S \rfloor$ we can then compute the fractional part from the equation.

$$\{S\} = b - a/(l/2^{\lfloor S \rfloor} - c).$$

The field $x0$ stores the fractional part of S, $\{S\}$. The field $y0$ is used for storing the value $2^{\lfloor S \rfloor}$. Storing $2^{\lfloor S \rfloor}$ directly, instead of storing $\lfloor S \rfloor$ and then computing $2^{\lfloor S \rfloor}$, speeds up the address calculation slightly.

The full address computation algorithm is given below. The statement "**if** h < T.x0 **then** . . ." requires some explanation. Recall that the x-value used in the address computation is computed as

$$x = \lceil S - h(K) \rceil + h(K).$$

Rewriting it in the form

$$x = \lceil \lfloor S \rfloor + \{S\} - h(K) \rceil + h(K)$$

it is easy to see that there are two cases:

$$\begin{array}{ll} \text{if } \{S\} - h(K) \leq 0 & \text{then } x = \lfloor S \rfloor + h(K) \\ \text{if } \{S\} - h(K) > 0 & \text{then } x = \lfloor S \rfloor + 1 + h(K). \end{array}$$

In both cases the fractional part is the same and equal to $h(K)$. Thus the value $a/(b - h(K)) + c$ is also the same in both cases. The only difference is whether this value is multiplied by $2^{\lfloor S \rfloor}$ or $2^{\lfloor S \rfloor + 1}$. If $h(K) < \{S\}$, the value is multiplied by $2^{\lfloor S \rfloor + 1}$, i.e., 2 * T.x0, otherwise by $2^{\lfloor S \rfloor}$, i.e., T.x0.

The logical address space of spiral storage starts from one. The mapping of addresses onto directory entries and segment entries is slightly easier if the address space starts from zero. Hence, the subtraction of one in the last statement. The fields *lowaddr* and *highaddr* are assumed to keep track of "physical" addresses, that is, addresses offset by one. This algorithm was used in the experiments reported in section 4, where the reasons for multiplying by a "scrambling" constant are also discussed.

```
function hash(K: keytype ; T: hashtable): integer ;
const
  cnst = 314159 ;   {scrambling constant}
  prime = 1048583 ;
var
  h, address: real ;
  temp: integer ;
begin
  temp := convertkey(K) ;
  temp := abs(temp*cnst)mod prime ;
  h := temp/prime ;
  address := T.y0*(a/(b − h) + c) ;
  if h < T.x0
    then address := address * 2.0 ;
  hash := trunc(address) − 1 ;
end ;
```

The algorithms for retrieving or inserting a record are the same as for linear hashing. However, because of the wrap-around, finding the correct segment requires a statement like the following:

```
currentsegment := directory[(address
                    div segmentsize)mod directorysize];
```

A full expansion algorithm can be found in [8]. The algorithm is quite straightforward and consists of four main parts: (1) locate and free the first bucket of the active address space, (2) update the state variables, (3) create the required new buckets at the end of the address space, and (4) redistribute the records from the deleted buckets. The expansion function explained above causes a slight complication: most expansions create two new buckets, but occasionally either one or three new buckets are created. This can be handled, for example, by first distributing the records over three auxiliary lists and then connecting each one to the chain of the appropriate bucket. A program fragment showing how to update the state variables is included below.

```
lowaddr := lowaddr + 1;
if lowaddr + 1 > 2*y0 then y0 := 2*y0;
x0 := b − a/((lowaddr+1)/y0 − c);
                              {Note that lowaddr is offset by one}
newhighaddr := ceil(2*y0*(a/(b−x0)+c) − 1) − 1;
```

3.2 Analysis

In this section we analyze the expected performance of spiral storage. In the same way as for linear hashing, it is assumed that the overall load factor is kept constant and equal to α, $\alpha > 0$, and that there are no deletions. The analysis is asymptotic. The expected load on a bucket varies over the active address space. We derive the load distribution assuming that the expansion function 2^x is used. The load distribution resulting from the expansion function $2^{\lfloor S \rfloor}(a/(b - h) + c)$ is, for all practical purposes, the same.

Without loss of generality we can consider only the normalized address range [1, 2). For any value of S, the active address range $[2^S, 2^{S+1})$ can be normalized to the range [1, 2) by multiplying all addresses by the factor 2^{-S}. Consider an infinitesimal interval $[y, y + dy) \subseteq [1, 2)$ and let $p(y)$ denote the probability that a key hashes to a (normalized) address in this interval. Under the assumption that the hashing function used distributes the keys uniformly over [0, 1), we obtain

$$p(y) = \log_2(y + dy) - \log_2(y)$$
$$= \log_2\left(1 + \frac{dy}{y}\right) = \frac{dy}{y \ln 2}.$$

Over the normalized address range [1, 2), the insertion probability density function is thus $1/(y \ln 2)$. The expected load factor of a bucket at address y is proportional to the insertion probability at y. The load factor, $\lambda(y)$, is therefore given by

$$\lambda(y) = c_1/(y \ln 2)$$

where c_1 is a normalizing constant. Because the average load factor must equal α, the value of c_1 can be determined from the equation

$$\int_1^2 \frac{c_1\, dy}{y \ln 2} = \alpha$$

from which we find that $c_1 = \alpha$. The highest load factor is at $y = 1$, $\lambda(1) = \alpha/\ln 2$, and the lowest at $y = 2$, $\lambda(2) = \alpha/(2 \ln 2)$.

Under the assumption that each record is equally likely to be retrieved, the probability of a successful search hitting a bucket is proportional to the load factor of the bucket. Hence, the probability of hitting a bucket with an address in $[y, y + dy]$ is $c_2 \alpha dy/(y \ln 2) = c_3 dy/y$. The constant c_3 is determined by the fact that the probability of a search hitting *some* bucket is one. Hence, the value of c_3 can be determined from the equation

$$\int_1^2 \frac{c_3\,dy}{y} = 1,$$

which gives $c_3 = 1/\ln 2$. If a successful search hits a bucket with a load factor of λ, the expected cost is $s(\lambda) = 1 + \lambda/2$. The expected cost of a successful search is then

$$\bar{S}(\alpha) = \int_1^2 s(\alpha/(y \ln 2))\frac{dy}{y \ln 2}$$
$$= \int_1^2 \left(1 + \frac{\alpha}{2y \ln 2}\right)\frac{dy}{y \ln 2}$$
$$= 1 + \frac{\alpha}{2}\frac{1}{2(\ln 2)} = 1 + \frac{\alpha}{2}\,1.0407.$$

Note that, not only is the cost of searching a bucket with a high load factor higher, we are also more likely to hit such a bucket. However, the net effect of the non-uniform distribution is rather small. The number of extra probes, beyond the one minimally required, is only 4 percent higher than for a uniform load distribution.

The probability that an unsuccessful search hits the (normalized) address interval $[y, y + dy]$ is $p(y)$. If the search hits a bucket in this address range, the search is done in a bucket whose expected load factor is $\alpha/(y \ln 2)$. Recall that the expected cost of an unsuccessful search in a bucket with load factor λ is $u(\lambda) = \lambda$. This gives the expected cost of an unsuccessful search as

$$\bar{U}(\alpha) = \int_1^2 u\left(\frac{\alpha}{y \ln 2}\right)\frac{dy}{y \ln 2}$$
$$= \int_1^2 \frac{\alpha}{(\ln 2)^2}\frac{dy}{y^2}$$
$$= \frac{\alpha}{2(\ln 2)^2} = \alpha\,1.0407.$$

The non-uniform load distribution hence increases the cost of unsuccessful searches by about 4 percent compared with a uniform load distribution.

The additional cost caused by expansions can be computed as follows. During an expansion all the records in the first bucket must be redistributed. The expected number of records in the first bucket is $\lambda(1) = \alpha/\ln 2$. A fraction $1/\alpha$ of the insertions trigger an expansion. Hence the additional hash address computations per record inserted is

$$\bar{A}(\alpha) = \frac{1}{\alpha}\left(\frac{\alpha}{\ln 2}\right) = \frac{1}{\ln 2} = 1.4427.$$

4. EXPERIMENTAL RESULTS

This section summarizes experimental results for linear hashing and spiral storage. The observed performance is also compared with that of a (unbalanced) binary tree and double hashing. All programs were written in C and run on a VAX 11/780 under UNIX 4.3BSD[1]. Test data were obtained from three real-life files:

File A: User names from a large time-sharing installation, 10000 keys, average key length of 7.1 characters.
File B: Dictionary of English words used by the UNIX spelling checker, first 20000 keys used, average key length of 7.2 characters.
File C: Library call numbers, 10000 keys, average key length of 13.3 characters.

Key conversion for linear hashing, spiral storage, and double hashing was done using the following algorithm:

```
convkey := 0;
for i := 1 to keylength do
  if K[i] <> ' '
  then convkey := 37*convkey + ord(K[i]);
convkey := abs(convkey);
```

For linear hashing and spiral storage the experiments were performed as follows. First all the keys were read into main memory. This was done to factor out I/O time from the experiments. Then the keys were inserted and the total insertion time was recorded. Each time a new key was inserted into the table, space for a new element was allocated by calling the standard memory allocation routine. The insertion process was halted every 1000 insertions and 1000 successful searches were performed. The search keys were selected from among the keys already in the table using a random number generator. Each experiment was repeated three times, each time using a different seed for the random number generator. The execution times reported are averages from the three experiments. The initial table size was 4 both for linear hashing and spiral storage. The directory size and segment size were both 256.

The goal of the first experiments was to test the performance of the hashing functions used. Tables II and III show the expected search lengths and the observed search lengths from one such experiment. The difference between the expected and observed values is less than 2 percent. Similar results were obtained for unsuccessful searches and for other values of α. These results confirmed that the hashing functions performed as expected.

In one experiment the "scrambling" constant in the hashing function for spiral storage was omitted. This significantly increased the observed search lengths, especially for File A. After some investigation, it was found that omitting the constant caused short keys

[1] The programs were first written in Pascal but found to be surprisingly slow. Switching to C (without changing the algorithms) speeded them up by a factor between 3 and 4.

TABLE II. Theoretically expected and observed average number of comparisons for a successful search in a linear hash table ($\alpha = 5$)

Number of records	Expected value	Observed average		
		File A	File B	File C
2000	3.81	3.84	3.84	3.84
4000	3.81	3.78	3.76	3.80
6000	3.68	3.67	3.66	3.72
8000	3.81	3.84	3.86	3.82
10000	3.56	3.60	3.56	3.52

TABLE III. Theoretically expected and observed average number of comparisons for a successful search using spiral storage ($\alpha = 5$)

Number of records	Expected value	Observed average		
		File A	File B	File C
2000	3.61	3.55	3.60	3.60
4000	3.61	3.57	3.56	3.59
6000	3.61	3.61	3.56	3.60
8000	3.61	3.58	3.59	3.59
10000	3.61	3.57	3.60	3.59

(3–4 characters) to cluster together near zero (after being hashed into the interval [0, 1)). Multiplication by a "scrambling" constant breaks up such clusters, and is therefore recommended whenever the key set contains a significant fraction of short keys.

The observed execution times for linear hashing and spiral storage are listed in Tables IV and V. The general trend is clear: spiral storage is consistently slower than linear hashing both for loading and searching. The main factor is the more expensive address calculation of spiral storage (approximately 0.16 ms/key for linear hashing and 0.24 ms/key for spiral storage). The expansion procedure of spiral storage is also more complex and somewhat slower. (For File A and $\alpha = 5$, the cost was approximately 1.41 ms per expansion for linear hashing and 2.19 ms for spiral storage.) The higher load and search times of File C are caused by the substantially longer keys. Longer keys increase the cost of key conversions and key comparisons.

For spiral storage, the loading cost per key is higher when $\alpha = 1$ than when $\alpha = 5$. This may appear surprising at first. The explanation is as follows. The cost of carrying out an expansion is of the form $C_1 + C_2\alpha$. The constant C_1 accounts for the overhead incurred in performing an expansion, that is, the cost incurred independently of the number of records in the bucket. The constant C_2 accounts for the cost of processing and relocating a record in the bucket being split. An expansion is triggered every α insertion. Hence, the cost of expansions *per record inserted* is $C_1/\alpha + C_2$. As α increases the effect of the first term (expansion overhead) decreases rapidly, but for small α it is a significant component of the cost. The cost of inserting a record (without the cost of expansions) is of the form $C_3 + C_4\alpha$. The total cost of an insertion is therefore $C_1/\alpha + C_2 + C_3 + C_4\alpha$. For small values of α the first term will dominate the cost and for large values of α the last term will dominate. This formula holds for both linear hashing and spiral storage, only the constants differ. (For File A and $\alpha = 0.5$ the loading cost of linear hashing was 0.99 ms per key.)

A simple, unbalanced binary search tree is a straightforward and generally efficient way of handling key sets of unknown cardinality. It was therefore decided to compare the performance of the new methods with that of a binary search tree. The experiments were organized as follows. First all records were read into main storage. The keys were then inserted into a binary tree. The insertion order was determined by a random number generator. The insertion process was again halted every 1000 insertions, the insertion time was recorded, 1000 random, successful searches were performed and the search time recorded. This loading-and-searching

TABLE IV. Average CPU-time in milliseconds/key for loading and searching in a linear hash table

	Loading			Searching		
Test data	$\alpha = 1$	$\alpha = 5$	$\alpha = 10$	$\alpha = 1$	$\alpha = 5$	$\alpha = 10$
File A	0.88	0.97	1.15	0.34	0.41	0.50
File B	0.94	1.02	1.20	0.36	0.44	0.53
File C	1.06	1.23	1.53	0.41	0.53	0.69

TABLE V. Average CPU-time in milliseconds/key for loading and searching in a hash table organized by spiral storage

	Loading			Searching		
Test data	$\alpha = 1$	$\alpha = 5$	$\alpha = 10$	$\alpha = 1$	$\alpha = 5$	$\alpha = 10$
File A	1.25	1.17	1.34	0.41	0.48	0.57
File B	1.26	1.20	1.37	0.42	0.49	0.59
File C	1.40	1.43	1.71	0.47	0.59	0.75

TABLE VI. Average CPU-time in milliseconds/key for loading and searching in an unbalanced binary tree

	Loading			Searching		
Tree size	File A	File B	File C	File A	File B	File C
1000	0.45	0.48	0.39	0.54	0.58	0.66
2000	0.69	0.70	0.73	0.66	0.72	0.86
3000	0.81	0.81	0.85	0.72	0.78	0.94
4000	0.88	0.87	0.95	0.76	0.82	0.99
5000	0.93	0.92	1.02	0.79	0.85	1.02
6000	0.97	0.96	1.07	0.82	0.87	1.06
7000	1.00	0.99	1.11	0.83	0.89	1.08
8000	1.03	1.01	1.15	0.86	0.91	1.12
9000	1.05	1.03	1.18	0.86	0.93	1.11
10000	1.07	1.05	1.21	0.87	0.95	1.13
20000		1.16			1.04	

process was repeated 10 times for each input file and averages computed.

Table VI shows the results obtained. The figures for loading give the total CPU-time required to build a tree of the indicated size. For example, to build a tree containing 5000 keys took, on average, 0.93 milliseconds/key for File A, 0.92 for File B, and 1.02 for File C. The corresponding figures for searching give the average CPU-time for searching in a tree of the indicated size. For example, locating a key in a tree of size 5000, required 0.79 milliseconds for File A, 0.85 milliseconds for File B, and 1.02 milliseconds for File C. The higher load and search costs for File C are again caused by the longer keys.

Comparing the results with those of Tables IV and V, we see that, for small key sets, building a binary tree is faster than loading a linear hash table. For key sets of 6000–10000 keys, the loading costs per key are approximately the same for a binary tree and a linear hash table with $\alpha = 5$. However, unless the key set is very small, searching in a binary tree is significantly slower than searching in a hash table.

TABLE VII. Average CPU-time in milliseconds/key for loading and searching using double hashing with a fixed-size table

	Loading			Searching		
α	File A	File B	File C	File A	File B	File C
0.60	0.37	0.36	0.42	0.31	0.31	0.38
0.70	0.38	0.37	0.42	0.32	0.33	0.39
0.80	0.38	0.37	0.44	0.34	0.34	0.41
0.90	0.40	0.40	0.47	0.37	0.37	0.44
0.95	0.43	0.43	0.51	0.40	0.40	0.49

The next comparison was with a traditional (fixed-size) hash table, where overflow records were handled by double hashing [3]. The results are summarized in Table VII. The search performance of a linear hash table with $\alpha = 1$ is approximately the same as that of double hashing with $\alpha = 0.8$. Even with the overall load factor as high as 10, the search performance of linear hashing is only 50 percent higher than that of double hashing with $\alpha = 0.8$. Not surprisingly, the cost of loading is substantially higher (2 to 3 times) for linear hashing and spiral storage than for double hashing. However, comparing the loading costs of a dynamic scheme and a traditional scheme with a fixed-size table is obviously not entirely fair.

As mentioned earlier, any traditional hashing scheme can be made pseudo-dynamic by rehashing all the keys into a larger table when the current table becomes too heavily loaded. The total loading cost of such a scheme depends on (1) the initial table size, (2) the maximal load factor, and (3) the expansion factor. The maximal load factor is the load factor at which the table is reorganized. The expansion factor is the relative size of the new table compared to the size of the old table. The double hashing implementation was modified to include this type of periodic reorganization. The resulting

TABLE VIII. Average CPU-time in milliseconds/key for loading a pseudo-dynamic hash table based on double hashing

Initial size	Maximum load factor	Expansion factor	File A	File B	File C
200	0.9	2	0.89	0.79	0.97
250	0.9	2	0.92	0.85	1.10
300	0.9	2	1.08	0.95	1.26
350	0.9	2	0.76	1.04	0.91
250	0.9	1.5	1.40	1.38	1.66
250	0.8	2	0.83	0.77	0.98

loading costs of the three test files are shown in Table VIII. The exact cost varies with the three parameters mentioned above but the main trend is clear: the loading cost of linear hashing is either lower or only slightly higher than that of double hashing with periodic reorganization. In addition, the insertion behavior of linear hashing and spiral storage is consistent; there are no long delays while the table is being reorganized. The delay can be substantial; total rehashing of 20000 records was observed to take over 11 seconds of CPU-time.

None of the methods above require an inordinate amount of overhead space. If the table for double hashing is also implemented as a pointer array, the overhead space consists solely of pointers for all methods. The space overhead typically ranges from one to two pointers per record. A binary tree always requires 2 pointers per record. For double hashing the number of pointers is $1/lf$ where lf is the load factor. This ranges from 2 for $lf = 0.5$ to 1.11 for $lf = 0.9$. For linear hashing and spiral storage the number of pointers per record is approximately $1 + 1/\alpha$, which ranges from 2 for $\alpha = 1$ to 1.1 for $\alpha = 10$. The exact formula for the total number of pointers is $n + \lceil n/(\alpha \times seg) \rceil seg + dir$, where n is the number of records, and seg and dir denote the segment size and directory size, respectively.

From the experimental results presented above the following overall conclusions can be drawn. For applications where the cardinality of the key set is known in advance, the best performance is obtained by a traditional fixed-size hash table. For applications where the cardinality of the key set is not known in advance, linear hashing gives the best overall performance. The average load factor can be set as high as 5 without seriously affecting the performance. Spiral storage is consistently slower than linear hashing. The expected loading cost of a binary tree is lower than that of linear hashing, but searching is slower (except for very small trees). Using a traditional hashing scheme with periodic reorganization does not seem to offer any advantages over using linear hashing.

REFERENCES

1. Carter, L.J. and Wegman, M.L. Universal Classes of Hash Functions, *Journal of Computer and System Sciences 18*, 1 (1979), 143–154.
2. Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3 (1979), 315–344.
3. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
4. Larson, P.-Å. Dynamic hashing. *BIT 18*, 2 (1978), 184–201.

5. Larson, P.-Å. Linear hashing with partial expansions. In *Proceedings of the 6th Conference on Very Large Databases*, (New York, 1980), 224–232.
6. Larson, P.-Å. Performance analysis of linear hashing with partial expansions. *ACM Trans. Database Syst. 7*, 4 (1982), 566–587.
7. Larson, P.-Å. Linear hashing with overflow-handling by linear probing. *ACM Trans. Database Syst. 10*, 1 (1985), 75–89.
8. Larson, P.-Å. Dynamic Hash Tables, Technical Report CS-86-21, University of Waterloo, 1986.
9. Litwin, W. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th Conference on Very Large Databases*, (New York, 1980), 212–223.
10. Martin, G. N. N. Spiral storage: Incrementally augmentable hash addressed storage. Theory of Computation Rep. 27, Univ. of Warwick, England, 1979.
11. Mullin, J. K. Spiral storage: Efficient dynamic hashing with constant performance. *Comput. J. 28*, 3 (1985), 330–334.
12. Ramakrishna, M.V. Perfect Hashing for External Files, Technical Report CS-86-25, University of Waterloo, 1986.
13. Ramamohanarao, K. and Lloyd, J. K. Dynamic hashing schemes. *Comput. J. 25*, 4 (1982), 478–485.
14. Ramamohanarao, K. and Sacks-Davis, R. Recursive linear hashing, *ACM Trans. Database Syst. 9*, 3 (1984), 369–391.

CR Categories and Subject Descriptors: E.1 Data Structures [Tables]; E.2 Data Storage Representations [**Hash Table Representations**]; F.2.2 Analysis of Algorithms and Problem Complexity [**Non-numerical Algorithms and Problems-sorting and searching**]

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Hashing, dynamic hashing, hashing functions, linear hashing, spiral storage

Received 6/86; accepted 4/87

Author's Present Address: Per-Åke Larson, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.