JOURNAL OF COMPUTER AND SYSTEM SCIENCES 43, 406–424 (1991)

An Implicit Data Structure for Searching a Multikey Table in Logarithmic Time*

AMOS FIAT

Computer Science Division, University of California at Berkeley, Berkeley, California 94720

J. IAN MUNRO

Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

MONI NAOR

Computer Science Division, University of California at Berkeley, Berkeley, California 94720

ALEJANDRO A. SCHÄFFER

Computer Science Department, Stanford University, Stanford, California 94305

JEANETTE P. SCHMIDT AND ALAN SIEGEL

Computer Science Department, Courant Institute, New York, New York 10012

Received May 17, 1989

A data structure is *implicit* if it uses no extra strorage beyond the space needed for the data and a constant number of parameters. We describe an implicit data structure for storing n k-key records, which supports searching for a record, under any key, in the asymptotically optimal search time $O(\lg n)$. This is in sharp contrast to an $\Omega(n^{1-1/k})$ lower bound which holds if all comparisons must be against the sought key value. The theoretical tools we develop also yield a more practical way to halve the number of memory references used in obvious non-implicit solutions to the multikey table problem.

* Address for correspondence. A. A. Schäffer, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251. This work was supported by a Weizmann Postdoctoral Fellowship, a Fannie and John Hertz Foundation Fellowship, IBM Almaden Research Center, the Information Technology Research Centre of Ontario, NSF (under Grants DCR 83-20000, DCR 84-11954 and DCR 85-13926), NSERC (under Grant A8237), DARPA (under Grant F49620-87-C-0065), and ONR (under Grants N00014-84-K-0444 and N00014-85-K-0046).

0022-0000/91 $3.00

1. INTRODUCTION

One of the most basic problems in computing is searching a table under a specified key. If key comparisons are the basic operation being performed, then the well-known binary search of a sorted array leads to an optimal solution. The sorted array has the additional virtue that no storage is required other than that for the data and the single parameter specifying the size of the structure. In this paper, we address a very natural and common generalization of the problem, namely, "How can we arrange n k-key records in an n by k array so that searches under any of the keys can be performed quickly?" Our interest is in *implicit* data structures, which are tables that store only the data plus a constant number of parameters. Our model of computation is the usual comparison based model.

In addressing this problem, one naturally turns first to the two-key case and quickly thinks of partitioning the data into $\sqrt{n}$ blocks of contiguous rank under one key, each of which is sorted under the other. This leads to search times of $\Theta(\sqrt{n})$ and $\Theta(\sqrt{n}\lg n)$, respectively, under the two keys. The first paper to explicitly address the problem, [Mun79], used what is effectively an implicit representation of a *k-d* tree [Ben75]. Following Fig. 1, the median element under the first key is placed in the middle position of the table; all higher elements are placed in the top half, and all lower elements below. The two halves are ordered by putting the median element (of those in that half) under the second key in the (one or three) quarter position and partitioning by that value. The quarters of the table are arranged recursively. This organization leads to a search algorithm that involves, under either key, two comparisons and two subtable searches, each of quarter size. The search time is then $O(\sqrt{n})$. In the more general k-key setting, this approach leads to a structure supporting searches in $O(n^{1-j/k})$ time if j of k key values are known (and $O(\log n)$ if all are known). Here and throughout the paper k is taken to be an arbitrary constant.

1	S	3	A	2	E	2	E	2	E	2	E
2	E	5	C	(3)	A	5	C	3	A	3	A
3	A	2	E	5	C			5	C	5	C
4	R	(6)	H					6	H	6	H
5	C	7	I	1	S			1	S	1	S
6	H	4	R	(4)	R			4	R	4	R
7	I	1	S	7	I			7	I	7	I
(8)	N							8	N	8	N
9	G	11	A	11	A			11	A	11	A
10	T	12	B	(12)	B			12	B	12	B
11	A	14	E	14	E			14	E	14	E
12	B	(9)	G					9	G	9	G
13	L	13	L	10	T			10	T	10	T
14	E	15	S	(13)	L			13	L	13	L
15	S	10	T	15	S			15	S	15	S

FIG. 1. The construction of an implicit 2*D* tree.

In [Mun79] it was shown that this structure minimizes (to within a small constant factor) the number of comparisons required to perform a search, if we assume that the elements under each key are stored according to some fixed partial order. This model may seem reasonable, but it is known to be too weak for the related problem of maintaining, in a simple array, a single key structure that supports insertion, deletion and search. For that problem, one can prove the same $\Omega(\sqrt{n})$ lower bound under the partial order model [MS80]; yet an $O(\lg^2 n)$ scheme has been given [Mun86] that is not based on a simple partial order.

While partial orders such as those of the structures outlined above seem natural approaches to the problem, one can come up with other interesting schemes that really do make use of other ordering information. For example, recall the well-known theorem of Erdős and Szekeres [ES35] that every sequence of n distinct numbers contains either an increasing or decreasing subsequence of length at least $\sqrt{n}$. We can sort a two-key table under the first key, then pull out a monotonic subsequence of length $\lceil \sqrt{n} \rceil$ under the second. These $\lceil \sqrt{n} \rceil$ records are now placed in the first $\lceil \sqrt{n} \rceil$ locations of the array (sorted under both keys) and the ordering process continues with the remainder of the records. When completed, we have about $\sqrt{2n}$ lists. Each is sorted under both keys although we do not retain the information as to whether or not each list is in increasing or decreasing order. While there is a severe restriction on the ordering of the second key, it is not simply a partial order since the relative order of no two values is known until a comparison is performed. A search under the first key is just a sequence of $\sqrt{2n}$ binary searches, one per sublist. To search a sublist under the second key, however, one must first determine the list's order. This, of course, is easy to do, and search under either key is readily accomplished in $\sqrt{n/2} \lg n + O(\sqrt{n})$ comparisons. While this organization is not an improvement on the prior schemes, it does suggest that a lower bound based on a partial order model may be somewhet suspect in its robustness.

Much more credibility was given to the notion that the k-d tree approach is an optimal technique for the implicit representation of a multikey structure by [AMM84]. It was proven that a search for an element specified by one of its k keys in an implicit data structure requires $\Omega(n^{1-1/k})$ time, subject only to the assumption that all comparisons involve the specified key value. A number of researchers have tried to remove this apparently minor restriction from the model. This paper explains the difficulty of such a task: the result would be false. In that sense the contriution of this paper is not just "another cute algorithm," but insight into the importance of models of computation for data structures.

2. The Basis of a Faster Algorithm

It is instructive to examine the [AMM84] lower bound to see precisely how our stronger computational model circumvents its conclusion. The bound is proven in two steps. First, the values of a k-key table are shown to force the structure to

permit up to $(n!)^{1-1/k}$ orders among the values of at least one key. This count is a consequence of the independence of the orders of the records under the different keys, and there is nothing one can do about it. The second step shows that, if the list can be in any of p permutations under a given key, then $\Omega(p^{1/n})$ comparisons are necessary to perform a search, provided all comparisons involve the element being sought. The proof uses a computation tree argument that gives a bound on the number of possible permutations in terms of tree height. It depends crucially on the restriction that all comparisons involve the element sought. This is the soft part of the model, and we begin the development of our method with P. Feldman's observation [BFMUW86] that the $\Omega(p^{1/n})$ bound does not hold if other comparisons are permitted.

Consider the following construction of an arrangement of a single key table.

1. Sort the table.
2. Keep the elements of even rank in their current positions.
3. Consider the elements of odd rank. Pair each such element in the first half of the array with a (distinct) element in the second half. Swap the paired elements.

Observe that this construction admits $(n/4)!$ valid orderings. Hence, under the crucial assumption that all comparisons involve the value being sought, the above lower bound shows that search takes $\Omega(n^{1/4})$ time. But consider the following search algorithm:

1. Perform a binary search on the even locations; if the element is found, we are done.
2. Otherwise, we have found where the element "should be." Perform a binary search for the element in this odd numbered location, call it x, among the even locations, ending between consecutive even locations. The element, y, in the odd location between these two even locations must be the element with which x was interchanged in the construction phase. The element y is either the desired element or the desired element is not present.

The search algorithm requires at most $2 \lg n + O(1)$ comparisons.

The search is rapid because swap-based permutations, which are called *involutions*, compute their own inverses, and because sufficient structure remains to support binary search. Evidently, these involutions offer $(n/4)!$ orderings that can be used to encode whatever information we like. The crucial question now becomes just what to encode. Indeed, a substantial part of this paper is devoted to this issue.

The first observation one is inclined to make is that, in the two-key version of the problem, the involution can be used to permit $2 \lg n + O(1)$ comparison searches on one key while maintaining sorted order on $\frac{1}{4}$ of the elements under the other key. In an earlier report on this work [Mun87], an ordering scheme that recurses on this idea is shown to lead to an $O(\lg^2 n \lg \lg n)$ solution in the two-key case. The $O(\lg n)$ technique we shall describe in the remainder of this paper was reported in preliminary form in [FNSSS88]. In this method, the involution trick

is used only to permit an efficient encoding of what are effectively pointers. The recursive search scheme in [Mun87] does, however, contain the germ of several of the ideas behind the $O(\lg n)$ technique. Like the former scheme, our method involves a binary search on a subset of the records to find one whose rank, under the appropriate key, is very close (within one in the case above, within k in the case below) to that of the desired record. We solve this problem on each of the keys simultaneously and thus partition the elements to guide searches under the various keys. The next difficulty to address is that this initial search may not lead directly to the desired record. At this point we must use some knowledge of how the elements have been permuted from sorted order to complete the search. In the involution based scheme, a relocated record is found simply by moving one (binary search based) step along an implicitly specified 2-cycle. In the following algorithm, finding where the element stored in a given location "should be if the records were sorted under key i" remains an easy binary search. However, the task of finding the element that "should be in a given location, if the records were sorted under key i" requires following cycles of arbitrary length. This cycle chasing will require a more complex solution if $O(1)$ binary search based steps are to enable record recovery.

3. The Algorithm

Implicit data structures can be designed according to the following strategy: first, use some additional memory to design a "semi-implicit" data structure; second, show how to encode the contents of the additional memory, while preserving the complexity of the operations on the semi-implicit version. We follow this strategy and achieve the first task in two steps. Section 3.1 describes how to partition the n records into k subsets, so that each subset represents a nearly perfect sample of the sorted list of values under one key. Section 3.2 shows how to use this partition and $\varepsilon n \lg n$ additional bits of memory ($0 \leqslant \varepsilon \leqslant 1$ is a constant) to obtain a $O(\lg n)$ time search method for any key. Section 3.3 shows how to eliminate the assumption of extra memory, while maintaining the good search time. One surprising feature of our method is that reducing the number of bits of structural information from $kn \lg n$ to $\varepsilon n \lg n$ leads to a fully implicit structure.

3.1. *Getting Close*

As suggested above, our first problem is to partition the records into k subsets so that the ith subset consists of records that are "fairly evenly spaced" among the values under the ith key. These subsets will, in fact, be of size n/k (we will assume that $8k$ divides n to simplify the presentation). Because of their role in the search procedure, we call the elements of the ith subset the *i-guides*.

Let $A[1 \cdots n]$ be the array in which we store the records. The relation $\prec_i$ is the ordering on the records under the ith key. We require that any two entire records be distinct, but the values along individual keys may be equal. We avoid having to insist that key values be distinct by formally defining $\prec_i$ as the lexicographic

(sorted) order on the records induced by concatenating the key values in cyclic order beginning with key i. L_i is used to denote the sequence of records sorted under relation $\prec_i$, i.e.,

$$L_i[1] \prec_i L_i[2] \prec_i \cdots \prec_i L_i[n].$$

We take evaluation under $\prec_i$ to be a unit time operation, even when fully lexicographic comparisons are necessary.

The scheme below shows how to associate each key with its n/k i-guides so that at most $2k-2$ keys fall between two consecutive i-guides under $\prec_i$. (See Figs. 2 and 3.)

1. Write the lists $L_1, L_2, \ldots, L_k$ as columns of an $n \times k$ matrix of records so that each record appears in each column.

2. Divide each column into n/k sets of k consecutive items. The matrix now contains n sets, each of size k, and the sets within each column are pairwise disjoint.

3. Choose one element from each set in such a way that each of the n records is selected exactly once. P. Hall's classical theorem on "complete sets of distinct representatives" [Ha35] implies that such a selection is possible, since the union of any m out of the n/k sets contains at least m distinct records. The records chosen in column i are the i-guides.

To efficiently compute a set of guides in step 3, we can use the equivalence of Hall's theorem and the Perfect Marriage Theorem. Consider a bipartite graph with n nodes on each side; the nodes on one side represent distinct records, and the nodes on the other side represent subsets (of size k in our case). Each record is adjacent to all the sets that contain it. Whenever Hall's theorem applies to the collection of subsets, the Perfect Marriage Theorem can also be applied to show that the bipartite graph has a perfect matching. In our case, the bipartite graph is k-regular, so the matching problem can be solved very quickly: in linear time if k is of the form 2^j [Gab76] and $O(kn \lg n)$ time in general [CH82]. Given a perfect matching, a record is chosen as an i-guide if and only if it is matched to a set from L_i.

The records are stored in A, so that it is easy to search among the guides for any key. Our (arbitrary) choice is to place the i-guides in consecutive locations:

$$A[(i-1)(n/k)+1], \ldots, A[i(n/k)],$$

sorted by the $\prec_i$ order. We call these n/k locations the *i-cluster*.

RECORD	NAME(1)	BIRTHPLACE(2)	YEAR OF BIRTH(3)
i	Carol	Honolulu	1960
ii	Frank	Denver	1955
iii	Alice	Boston	1948
iv	Gus	Atlanta	1958
v	David	Chicago	1960
vi	Iris	Denver	1949
vii	Bob	Philadelphia	1968
viii	Henry	Detroit	1963
ix	Eve	Miami	1958

FIG. 2. Sample data for a subsequent figure; $k = 3$ and $n = 9$.

Our goal of having the i-guides "fairly evenly spaced" has been achieved.

Remark 3.1. The jth i-guide occurs in L_i somewhere between positions $k(j-1)+1$ and kj inclusive. In any list L_i, there are at most $2k-2$ records between consecutive i-guides.

3.2. *The Search*

The next stage in describing the data structure is to show how to search the array A with the records arranged as specified at the end of Section 3.1.

The reader ought to think of the i-cluster elements as being permanently sorted according to $\prec_i$. The final scheme, however, perturbs that order within each cluster for encoding purposes (see Section 3.3), but the guides for any particular key remain within their cluster in an order that supports a logarithmic search.

Remark 3.1 suggests the following partial search strategy for an item with ith key value v:

SEARCH-SKETCH(v)

1. Perform a binary search for v among the i-guides (in the i-cluster). If v is not found then following Remark 3.1, we have determined that the value v may be found in one of $2k-2$ known positions in the (imaginary) list L_i.

2. We can thus restrict our attention to a constant number of positions in L_i. Our goal is to find the locations in A that contain the items from these positions in L_i.

We present the method to find these items in two stages. In the first stage we use some extra pointers and counters in addition to the array A. In the second stage, deferred to Section 3.3, we show how to encode the $\Theta(n \lg n)$ additional bits of information into the array A, and modify the search algorithm accordingly. Perhaps the most striking feature of our algorithm is that at this stage, we are simply trying to reduce the number of pointers required to εn, for suitable $0 < \varepsilon < 1$, so that they may be encoded via involution.

Let $\pi_i: Z^+_{n+1} \to Z^+_{n+1}$ (where Z^+_{n+1} denotes $\{1, ..., n\}$) be the permutation that maps a record's index in A into the record's index when the records are sorted under the order $\prec_i$, i.e.,

$$A[\pi_i^{-1}(1)] \prec_i A[\pi_i^{-1}(2)] \prec_i \cdots \prec_i A[\pi_i^{-1}(n)].$$

(See Fig. 3.)

Step 1 of SEARCH-SKETCH restricts v to at most $2k-2$ known locations in L_i, say locations $q_1 \leqslant j \leqslant q_2$. Since $A[\pi_i^{-1}(j)] = L_i[j]$, v can, in principle, be found through a binary search among the $2k-2$ records,

$$A[\pi_i^{-1}(q_1)], A[\pi_i^{-1}(q_1+1)], ..., A[\pi_i^{-1}(q_2)],$$

provided we can compute π_i^{-1}. In fact, it turns out that the relative ease of computing π_i is the crucial starting point for computing its inverse. Computing $\pi_i(j)$ is

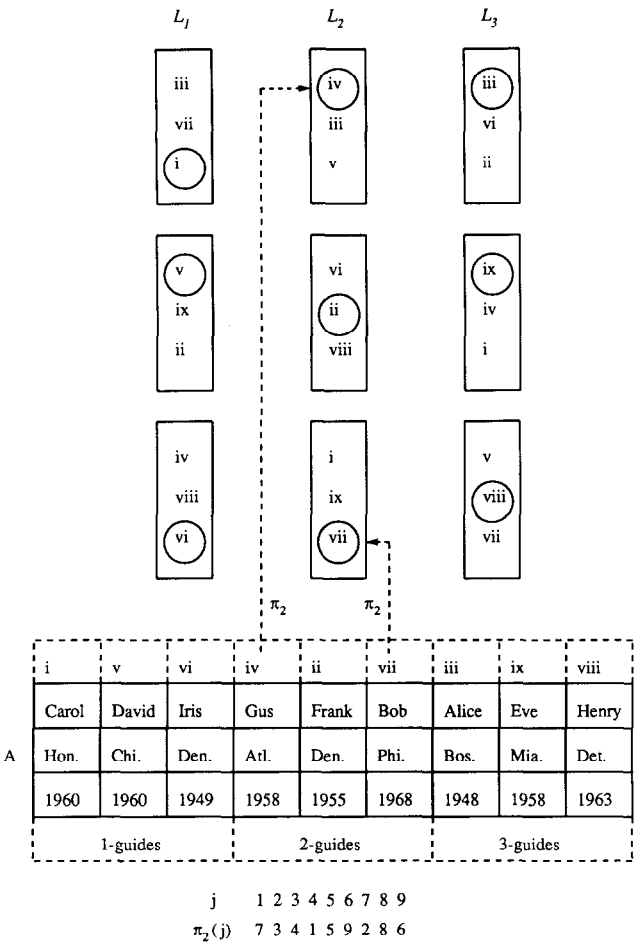

FIG. 3. The use of Hall's theorem; the records chosen as guides are circled. The initial placement of records is shown; Roman numerals are just record names and are not stored. The permutation π_2 is defined and its values at 4 and 6 are illustrated with arrows.

tantamount to finding the rank of $A[j]$ in the order $\prec_i$. To compute $\pi_i(j)$ approximately, we search for the record $A[j]$ in the i-cluster (under the order $\prec_i$). As in step 1 of SEARCH-SKETCH, this gives a range of $2k-2$ possible condidates for $\pi_i(j)$. To compute $\pi_i(j)$ exactly, we associate a vector $F_i[1 \cdots n]$ with every key $1 \leqslant i \leqslant k$, where $F_i[j] \in \{1, ..., 2k-2\}$ gives $\pi_i(j)$'s index within the range of $2k-2$ candidates. Note that each array F_i only requires $n \lg 2k$ bits.

In the involution scheme, we took advantage of the fact that the cycles of π_i were of length 2, i.e., $\pi_i = \pi_i^{-1}$. If cycles were of length h, we could, of course, evaluate π_i^{-1} by $h-1$ forward mappings. This would be fine if h were guaranteed to be bounded by a constant. Unfortunately we can make no such claim. The cycle length h can be $\Theta(n)$.

When cycles of π_i are long, we store strategic shortcuts that enable us to skip forward most of the way around the cycle in one step (see Fig. 4). We declare that a cycle is *long* if it is longer than some constant, l, to be estimated later. (In Section 3.3 we show that $l = O(k)$ and is independent of n). These shortcuts will require $\Theta(n)$ pointers, but the hidden constant factor can be made small, which is essential for the encoding scheme in Section 3.3. An arbitrary starting point p is chosen for each cycle, and starting from p, every lth element along the cycle is given a shortcut pointer to l places *back* along the cycle. Thus, $\pi_i^l(p)$ remembers the index p, $\pi_i^{2l}(p)$ remembers the value $\pi_i^l(p)$, and so on. Location p remembers the value of $\pi_i^{\hat{h}}(p)$ where $\hat{h}$ is the largest multiple of l strictly less than the cycle length h.

These special locations, which are l apart along the cycle, are called *gateways*. If $x \in Z_{n+1}^+$ is a gateway, we use $R_i(x)$ to denote the location that x remembers. These gateways provide convenient paths to speed up the search at an aggregate cost of only "εn" pointers.

Let g_i be the number of gateways for key i. Evidently, $g_i \leq \lfloor 2n/l \rfloor$. We number the gateways, x, associated with the permutation π_i (and our arbitrary choice of the p's above) with the values 1 through g_i, starting with the smallest gateway and in increasing order of x (regardless of which cycle they belong to).

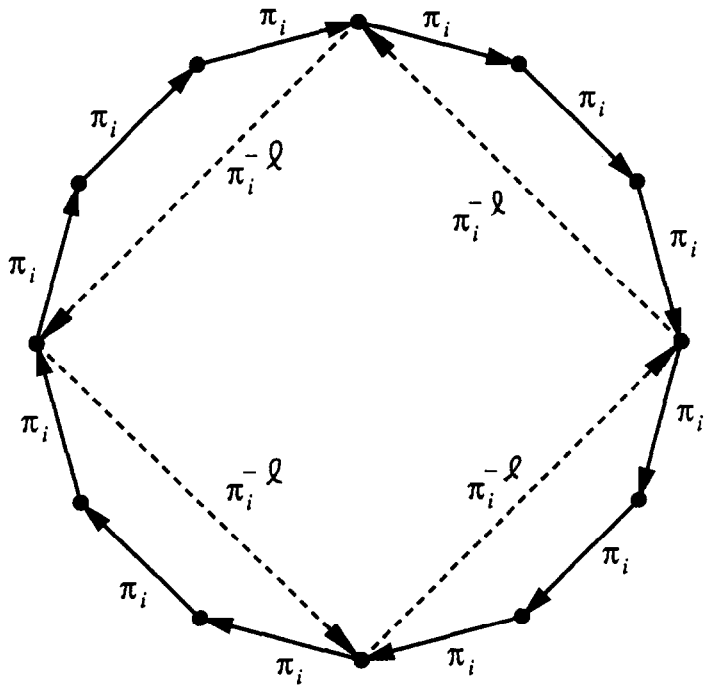

FIG. 4. One can chase around a cycle on π_i going forwards using a solid arrow or via a gateway using a dashed arrow.

A convenient way to store the values $R_i(x)$ is to use an array $Q_i[1 \cdots n]$, where $Q_i[x] = R_i(x)$ if x is a gateway and $Q_i[x] = 0$ otherwise. A more space-efficient way is to use a bitmap $B_i[1 \cdots n]$, where $B_i[x] = 1$ if and only if x is a gateway, and an array $G_i[1 \cdots g_i]$ compressed to contain only the nonzero value of $Q_i[x]$ (in order they would appear Q_i). To find where $R_i(x)$ is stored in G_i we use an array of counters, $C_i[0 \cdots \lfloor n/\lceil \lg n \rceil \rfloor]$; each entry of C_i is a number between 0 and g_i inclusive. The entry $C_i[j]$ is the partial sum

$$\sum_{1 \leq m \leq j\lceil \lg n \rceil} B_i[m].$$

It counts the number of gateways in the first $j\lceil \lg n \rceil$ locations (and is the index within G_i, where $R_i(x)$ is stored, for the largest gateway $x \leqslant j\lceil \lg n \rceil$). If $B_i[x] = 1$ then $R_i(x) = G_i[c]$ for some c. The index c can be computed as the sum of one C_i entry and at most $\lg n$ consecutive B_i entries.

To conclude the description of the search algorithm, we introduce PI-INVERSE, a function that computes $\pi_i^{-1}(j)$. As explained above, we find v by performing a binary search on the $2k-2$ records,

$$A[\pi_i^{-1}(q_1)], A[\pi_i^{-1}(q_1+1)], \ldots, A[\pi_i^{-1}(q_2)];$$

thus we call PI-INVERSE at most $\lceil \lg(k) \rceil + 1$ times.

PI-INVERSE(i, j)

Set nextindex := j;
Repeat

1. index := nextindex;
2. if B_i[index] = 1 (index is a gateway.) then
   set $d := \lfloor$index$/\lceil \lg n \rceil \rfloor$
   and set $c := C_i[d] + \sum_{d\lceil \lg n \rceil < m \leqslant \text{index}} B_i[m]$;
   index := $G_i[c]$.
3. Do a binary search for A[index] among the i-guides (according to $\prec_i$). (We have now identified a range $[q_1', q_2']$, $q_2' - q_1' \leqslant 2k-2$, such that π_i(index) lies in this range).
4. Set nextindex := $q_1' + F_i$[index] $- 1$;

Until nextindex = j;
Return(index);

Since the effective cycle length is bounded by l, the loop in PI-INVERSE is repeated l times at most. Each iteration requires $\lg n$ comparisons for the binary search at step 3. In Section 3.3, we convert from a semi-implicit structure, where the arrays B_i, C_i, F_i, and G_i are in auxiliary storage, to an implicit structure where these arrays are encoded into the internal order of A. We postpone the rest of the running time analysis until the end of Section 3.3.

3.3. *Encoding*

In this section, we show how to perturb A so that the structures B_i, C_i, F_i, and G_i, associated with each key $1 \leqslant i \leqslant k$, can implicitly encoded in A. Recall that B_i and C_i require $O(n)$ bits and F_i requires $O(n \log k)$ bits. Since l turns out to be independent of n, the storage requirements will be dominated by the G_i's, which might each contain $\Theta(n/l)\lceil \lg n \rceil$ bit words.

Previous work on implicit data structures has involved the encoding of pointers, flags, etc., into the relative order of the data [Mun86]. In particular, one can use the potential swapping of elements of contiguous odd–even ranks to encode $\Theta(n)$ bits. This method is simple and has the advantage of being able to change a bit with a simple data swap. While crucial to the solution of a dynamic problem such as that considered in [Mun86], this odd–even encoding falls short of the $\Theta(n \lg n)$ bits of information required under our premise that $l = O(k)$. We encode the arrays associated with each key in the appropriate cluster independently of the others. Our method of encoding B_i, C_i, F_i, and G_i addresses the following more abstract problem.

ENCODING PROBLEM. Find a constant $0 \leqslant c \leqslant 1$ and an algorithm that encodes $cm \lg m$ bits into a previously sorted array, M, of m distinct elements while attaining the following:

1. Searching for a value in the perturbed array takes time $O(\lg m)$.

2. Finding the record that would be in the jth location if the array were sorted takes time $O(\lg m)$.

3. The $\Theta(m \lg m)$ bits are divided into logical words of $(1 - o(1)) \lg m$ bits each in such a way that random access to and decoding of the jth logical words requires $O(\lg m)$ comparisons.

Our encoding will also allow the r most significant bits of a logical word to be decoded in r comparisons, for every r up to the word size. Our solution begins by exploiting the involution trick described in Section 2, but another idea is needed. We want to construct a permutation $\tau: Z^+_{t+1} \to Z^+_{t+1}$ that depends on the values we want to encode; t will be chosen to depend linearly on m. Since there are $t!$ such permutations, the choice of τ determines $\lg(t!)$ bits. It is not clear a priori if one can determine some of the bits encoded by the permutation by knowing its values on only some small subset of the domain, but this indeed turns out to be possible.

The next step is to use the ranks of $\Theta(m)$ elements, defined with respect to a sorted array of about $\Theta(m)$ other values, to define $\Theta(m \lg m)$ bit words.

Let $1 \leqslant \alpha, \beta \leqslant t$ such that $\alpha + \beta = t$. We describe how to design τ to encode β logical words, with value in the range $1, ..., \alpha$ so that random access to the jth word requires only one binary search one sequence of length α. (See Fig. 5 and 6). Let the data be the sequence: $x_1, x_2, ..., x_\beta$, where each x_j is a value between 1 and α. This data will be encoded by the permutation τ defined on a sorted sequence of t distinct encoding items, which are denoted by their ranks $1, ..., t$. For the sake of notational simplicity, we assume that τ is also defined on the value 0, and $\tau(0) = 0$.

Numbers to encode: 6,3,2,7,6,2,4,3
Key values: $1, 2, \ldots, 16$

Encoding:

$\tau(1)$	$\tau(2)$	$\tau(3)$	$\tau(4)$	$\tau(5)$	$\tau(6)$	$\tau(7)$	$\tau(8)$
1	4	7	9	10	13	15	16

$\tau(9)$	$\tau(10)$	$\tau(11)$	$\tau(12)$
11	5	2	14
encodes 6	encodes 3	encodes 2	encodes 7

$\tau(13)$	$\tau(14)$	$\tau(15)$	$\tau(16)$
12	3	8	6
encodes 6	encodes 2	encodes 4	encodes 3

ex. $\tau(9)$ encodes 6, since $\tau(5) < \tau(9) = 11 \leq \tau(6)$.

FIG. 5. An example of the function τ in Section 2 with $t = 16$, $\alpha = 8$, $\beta = 8$.

Define $s(v)$ as the number of occurrences of the value v in the sequence $x_1, \ldots, x_\beta$: $s(v) = |\{j \mid x_j = v\}|$ for $1 \leq v \leq \alpha$. Evidently, $\sum_{1 \leq v \leq \alpha} s(v) = \beta$.

Let

$$\tau(v) = \tau(v-1) + s(v) + 1 \qquad \text{for} \quad 1 \leq v \leq \alpha.$$

Note that $\tau(1), \tau(2), \ldots, \tau(\alpha) = t$ is a subsequence of $1, 2, \ldots, t$.

Suppose that $x_j = v$ and x_j is the pth occurrence of the value v in the sequence $x_1, \ldots, x_j$. Then define

$$\tau(\alpha + j) = \tau(v-1) + p \qquad \text{for} \quad 1 \leq j \leq \beta.$$

To decode x_j, find an index $1 \leq v \leq \alpha$ such that

$$\tau(v-1) < \tau(\alpha + j) < \tau(v),$$

Numbers to encode: 7,5,2,7,3,5
Key values: $1, 2, \ldots, 16$

Encoding:

$\tau(1)$	$\tau(2)$	$\tau(3)$	$\tau(4)$	$\tau(5)$	$\tau(6)$	$\tau(7)$	$\tau(8)$	$\tau(9)$	$\tau(10)$
1	3	5	6	9	10	13	14	15	16

$\tau(11)$	$\tau(12)$	$\tau(13)$	$\tau(14)$	$\tau(15)$	$\tau(16)$
11	7	2	12	4	8
encodes 7	encodes 5	encodes 2	encodes 7	encodes 3	encodes 5

ex. $\tau(11)$ encodes 7, since $\tau(6) < \tau(11) \leq \tau(7)$.

FIG. 6. Another example of the function τ in Section 2 with $t = 16$, $\alpha = 10$, $\beta = 6$.

in which case $x_j = v$. This means that x_j can be decoded by performing a binary search for $\tau(\alpha + j)$ in the sequence $\tau(1), ..., \tau(\alpha)$. For decoding, we only need to know the relative order among τ images, and not the actual τ values themselves.

To solve the encoding problem, we apply an involution to the m array elements, swapping odd-numbered locations in the first half of the array with odd-numbered locations in the second half of the array. Such an involution can define any permutation on $t = m/4$ elements. For all $1 \leqslant j \leqslant t$, if $\tau(j) = h$ then swap the jth odd element in the first half of the array with the hth odd element in the second half of the array (See Fig. 7). This means that for all $1 \leqslant i, j \leqslant t = m/4$,

$$M[2i-1] < M[2j-1] \qquad \text{iff} \quad \tau(i) - \tau(j).$$

Choose $\alpha = m/\lg(m)$; then $\beta = m/4 - m/\lg(m)$, and $\alpha + \beta = m/4 = t$. There are now $(\frac{1}{4} - o(1))m$ logical words, each of length $(1 - o(1))\lg(m)$ bits, encoded in the vector M.

We attain property 1 of the encoding problem by noting that we can still perform a binary search on the even locations. If the desired element is in an odd location, then the binary search leads us to the element that was swapped with our search value. By performing a second binary search on the swapped value, we find the desired element.

Property 2 its trivially obtained for even indices and requires one binary search on the even locations if the index is odd.

Property 3 is satisfied since decoding the jth logical word for, $1 \leqslant j \leqslant \beta = m/4 - M/\lg(m)$, can be done by searching for $M[2\alpha + 2j - 1]$ in the sorted α element subarray $M[1], M[3], ..., M[2\alpha - 1]$.

We apply the solution to the encoding problem to code the B_i, C_i, F_i, and G_i arrays implicitly within the i-cluster of $m := n/k$ guides. Within each such array, we can store $(\frac{1}{4} - o(1))m \lg m$ bits, in about $m/4 = n/(4k)$ words of length $(1 - o(1))\lg(n/k)$ bits.

The storage requirements for G_i are $2n/l$ integers, each $\lg n$ bits long. The storage needed for B_i and C_i is only $2n + O(1)$ bits, while F_i requires $n \lg k$ bits. We can therefore choose l to be approximately $8k$.

The arrays C_i and G_i are accessed only when the cycle chase in PI-INVERSE goes through a gateway, which occurs at most once per PI-INVERSE invocation. The arrays B_i and F_i, however, are used in *every* loop iteration. Therefore it is a good idea to store the B_i and F_i vectors in the most significant $4k \lg(2k)$ bits of the $n/(4k)$ available words, so that access to any entry of B_i or F_i costs at most $4k\lg(2k)$ comparisons.

33	2	37	4	41	6	43	8	49	10	51	12	57	14	59	16
61	18	63	20	53	22	45	24	35	26	55	28	39	30	47	32
1	34	25	36	3	38	29	40	5	42	7	44	23	46	31	48
9	50	11	52	21	54	27	56	13	58	15	60	17	62	19	64

FIG. 7. The involution of M induced by the τ of Fig. 6, when $m = 64$, and the keys are 1, 2, ..., 64. The permuted keys are in row-major order.

As a result of the encoding, we require $\lg n$ comparisons at step 3 in PI-INVERSE just to find $A[\text{index}]$ and $2 \lg n$ comparisons for the binary search. (This last count includes a binary search in the last comparison step to find the swapped key.)

An access to r consecutive entries of B_i can be done in $r + 4k \lg(2k)$ comparisons. When at a gateway (once per PI-INVERSE invocation), we may need to access as many as $\lg n$ consecutive entries of B_i, which may require $\lg n + 4k \lg k$ comparisons. Accessing C_i and G_i involves $2 \lg n$ comparisons for each, since each entry of C_i and G_i requires $\lg n$ bits and must, therefore, be spread over two encoded words.

Now we put together all the bounds. To do a search we require at most $\lg k$ calls to PI-INVERSE. In each call we require about $3 \lg n$ comparisons per iteration, l iterations, and an extra $5 \lg n$ comparisons in the (at most) one iteration in which we are at a gateway. This gives a total of roughly $(3l \lg k + 5) \lg n$ comparisons. Using our previous estimate on l we get a bound of $(24 + o(1))k \lg(k) \lg(n)$ comparisons per search. We briefly sketch one way to improve the multiplicative constant in Section 5. To summarize:

THEOREM 3.2. *Implicit k-key table search can be done in* $O(\lg n)$ *time. The constant is proportional to* $k \lg k$.

4. Application to Memory Efficient Data Structures

In this section, we sketch two potentially more practical schemes to organize a memory efficient data structure. First, we use the theoretical tools developed in Section 3.1 to suggest a scheme that uses $(k-1)n$ pointers, but can be searched with about half of the number of memory references that would be needed by the obvious kn-pointer solutions. Second, we take a different approach and suggest a solution that uses only an additional 6% of storage and supports search in about $(4k-3) \lg n$ memory references.

One obvious solution to the non-implicit multikey table problem is to keep k copies of the array, each sorted under a different key.

This is clearly wasteful in storage. A second solution is to keep sorted arrays of the form (key-value, pointer); this doubles the basic storage requirement and requires an additional $kn \lg n$ bits for pointers. To avoid duplicating the record values, one can store an array of pointers, sorted under the record value they address. This memory-efficient scheme requires $2 \lg n$ memory accesses per search.

To cut down the search time by $\frac{1}{2}$, we apply P. Hall's theorem and divide the records into i-guides. The basic organization is exactly as above (Section 3.1), i.e., sorted clusters of i-guides in one array. For each key, we keep an array of $n - n/k$ pointers, giving the sorted order of the non-guide records under that key. Since $k(n - n/k) = (k-1)n$, this arrangement uses no more storage than the scheme of the previous paragraph. To search under the ith key, we perform a binary search

among the i-guides. If no match is found, the search can nevertheless be limited to $2k-2$ pointers. This method requires at most $\lg(n/k)+2\lg(2k-2) \leqslant \lg n + \lg k + 2$ memory references per search.

In the remainder of this section we describe another non-implicit scheme that permits an efficient trade-off between the amount of extra space and the search time. This scheme assumes a more restrictive scenario in which a record consists of disjoint fields. This means that the records can be broken up with the different key values stored separately, provided that all the values that comprise any given record can be retrieved. In fact, breaking up the records makes the search much easier, and the principal difficulty is in reconstructing records. Another restriction is that no two records have the same value for any particular field.

The ith fields of all the records will be stored in an array V_i. In addition, two auxiliary arrays S_i and W_i are used to help search the arrays V_i and to reconstruct the records.

Choose a trade-off parameter p; we assume for simplicity that p divides n. For each field i, we sort the records under field i and produce the lists L_i as before.

The value of field i is extracted from evey pth record in the list L_i, and these n/p values are stored in an array S_i, which will guide searches along field i. Define $nxt(i) = (i \bmod k) + 1$. For each i, the field $nxt(i)$ is extracted from all records in list L_i, and these n values are stored in the array $V_{nxt(i)}$ according to their L_i order (i.e., sorted under the ith field of their record).

For each i, we use an array W_i of size n to help associate a value in V_i with its *companion* value in $V_{nxt(i)}$, which belongs to the same record. The vector $W_i[1, ..., n]$ takes values in the range $0, ..., p-1$.

Assume $V_i[h]$ and $V_{nxt(i)}[j]$ are companions (i.e., came from the same record). Then we search through S_i and determined the value $(j \operatorname{div} p)$. We therefore store $(j \bmod p)$ in $W_i[h]$, so that a search of S_i and one lookup in W_i jointly determines j. This concludes the data structure description. See Figs. 8 and 9 for an example.

For ease of notation, we extend the S_i, V_i, and W_i vectors to include the zero index, with the appropriately chosen boundary value.

To search for a record that has its ith field equal to v, we do the following:

1. Perform a binary search for v in the vector S_i, say $S_i[q] \leqslant v < S_i[q+1]$. This means that the $nxt(i)$ field companion for v is one of the p values:

$$V_{nxt(i)}[qp], ..., V_{nxt(i)}[(qp+p-1)].$$

RECORD	NAME(1)	BIRTHPLACE(2)	YEAR OF BIRTH(3)
i	Jane	Seattle	1950
ii	Ken	Memphis	1958
iii	Linda	Wichita	1957
iv	Mark	Phoenix	1952
v	Nancy	Cleveland	1961
vi	Oliver	Milwaukee	1949
vii	Pat	Dallas	1964
viii	Quincy	Baltimore	1953
ix	Ruth	Louisville	1955

FIG. 8. Sample data for a subsequent figure; $k=3$ and $n=9$.

V_1, W_1 :	Oliver,0	Jane,1	Mark,1	Quin.,2	Ruth,0	Linda,0	Ken,2	Nancy,2	Pat,1
V_2, W_2 :	Sea.,2	Mem.,2	Wich.,0	Phoe.,1	Cle.,2	Milw.,0	Dal.,0	Balt.,1	Lou.,1
V_3, W_3 :	1953,1	1961,2	1964,0	1955,2	1958,1	1949,1	1952,0	1950,2	1957,0
S_1 :			Linda			Oliver			Ruth
S_2 :			Dal.			Milw.			Wich.
S_3 :			1952			1957			1964

A sample search for Jane:

A binary search on S_1 shows that the (city) companion of Jane is in the first group of 3 in W_2, (Sea., Mem. or Wich.).

The (date) companion of Mem.,2 (col. 2 of V_2, W_2) is 1958, since a binary search on S_2 shows that Mem. belongs to the second group of 3, the W_1 value indicates that the companion is the second one in this group.

The (name) companion of 1958,1 is similarly identified as Ken.

Jane < Ken, therefore the (city) companion of Jane must be to the left of Mem., which is Sea.. This should be verified as above as Jane might not be in the table at all.

FIG. 9. The data structure contents for the example of Fig. 8 using the scheme of Section 4 with $p = 3$.

Reconstructing the records that contain these p values in field $nxt(i)$ will identify the record containing v in field i. Since the p values are sorted according to their ith field companion, binary serach will allow us to find the required record while reconstructing only $\lceil \lg p \rceil$ of the p candidate records.

2. Given an index s in $V_{nxt(i)}$, we can reconstruct the record containing $V_{nxt(i)}[s]$ by performing $(k-1)\lg(n/p)$ comparisons. Let $r = nxt(i)$ and repeat $(k-1)$ times:

(a) Search for $V_r[s]$ in S_r, say $s_r[t] \leqslant V_r[s] < S_r[t+1]$. The $nxt(r)$ companion to $V_r[s]$ is $V_{nxt(r)}[tp + W_r[s]]$.

(b) Set $s := tp + W_r[s]$, $r := nxt(r)$.

The extra memory required by this scheme is (S/p) bits to store the S_i vectors, where S is the space required to store the n records. In addition, we need $nk \lg p$ bits for the W_i vectors. The space required for the W_i vectors is low order, and the W_i vector can be interleaved with the V_i vector so that one disk access will suffice to read both $V_i[s]$ and $W_i[s]$.

The number of comparisons for step 1 is $\lg(n/p)$; step 2 requires $(k-1)\lg(n/p)$ comparisons and is repeated $\lceil \lg p \rceil$ times. Altogether, this gives an upper bound of $(\lg(p)(k-1)+1)\lg(n/p)$ comparisons. For example, choosing $p = 16$ gives us a 6% additional space requirement, plus another kn nibbles (4 bits) for the W_i vectors. The number of comparisons/disk accesses is $(4k-3)(\lg n - 4)$. The method of involution based encoding can be used to convert this non-implicit scheme into an implicit one.

5. Improvements, Remarks, Open Problems

A variant of the algorithm described in Section 3 achieves a better multiplicative constant in the worst-case search time. To get a better constant, we decrease the length of the cycles by increasing the number of "shortcuts"—this means that we must increase the "virtual storage" available.

Previously, in each set of n/k guides we fixed the even locations and involuted the $n/(2k)$ odd locations to encode $(\frac{1}{2}-o(1))n/(2k)$ words of $(1-o(1))\lg n$ bits in each set of guides.

For any constant $d \geqslant 2$, we can fix the locations $j=0 \pmod d$, and use our encoding involution $d-1$ times, by grouping locations congruent to $j \pmod d$, for $j=1, 2, ..., d-1$. This lets us encode $(1-1/d-o(1))n/2k$ words of $(1-o(1))\lg n$ bits in each set of guides.

To search for a value v, we perform an initial search on the guides at locations $j=0 \pmod d$. We then have to perform another $\lg d$ binary searches (inverting $\lg d$ involutions) until we find the two guides g_1 and g_2 such that g_1 is the predecessor to v and g_2 is the successor to v amongst the guides. This limits our search to $2k$ locations in the sorted list L_i. We now need to perform $\lg(2k)$ cycle chases.

Subsequently, as we chase through the cycle, we could emulate the previous scheme and determine the two guides g_1, g_2 that bound the serach value v. This would require $(\lg d)\lg n$ comparisons—in fact, it suffices to search the guides at locations $j=0 \pmod d$. Then we have a range of $(d+1)k-2$ possible locations for the next cycle chase element, rather than a range of $2k-2$. This larger range is handled by enlarging the F_i arrays, so that $F_i[j] \in \{1, ..., dk-2\}$, rather than having $F_i[j] \in \{1, ..., 2k-2\}$ as before. Each F_i array now consists of $n \lg(kd)$ bits.

As before, the B_i and F_i arrays are placed in the most significant bits of the encoded data so that the appropriate B_i and F_i entries can be decoded in a constant number of comparisons.

We view each step of the cycle chase as a traversal from one rank in L_i to another rank in L_i. Every iteration of the cycle chase, except the gateway iteration, requires $(2+o(1))\lg n$ comparsions: $\lg n$ comparisons to invert the involution, $\lg n$ comparisons to search among the appropriate guides in locations $j=0 \pmod d$, and a constant number of comparisons to decode the B_i and F_i arrays.

The gateway is traversed once per search, and requires some $5 \lg n$ comparisons to decode the appropriate B_i, C_i, and G_i entries. Let l denote the maximal distance between gateways, then the G_i array contains at most $2n/l$ pointers, thus $l \leqslant (1+1/(d-1)+o(1))4k$. Overall we have $2l \lg(2k) \lg n + 5 \lg n$ comparisons per search. Thus, we have

$$(8(1+1/(d-1))+o(1))k \lg(2k) \lg n + (\lg d + 5) \lg n$$

comparsons per search, for any $d \geqslant 2$.

It should be noted that there is a subtle space–time trade-off of our table organization, which is due to the limited encoding power of a fixed size table.

As written, our algorithm, for example, cannot be implemented even with zero shortcuts unless the F_i's can be encoded within the array. A straightforward counting argument shows that n must therefore be greater than $(dk)^{2k(d/(d-1))}$. More generally, the relationship between l, which governs the time complexity, and parameters n, k and d is: $n > (4\,dk)^{((2k/(1-1/d-4k/l))} > 1$.

In Section 3.2 we used a bitmap and counters to reduce the amount of storage for the array of gateways which is sparse. The problem of storing a sparse table of this type has been studied by Tarjan and Yao in a more general setting [TY79]. They point out that one can precompute the n possible partial sums of B values that might be needed in PI-INVERSE. Since each partial sum is at most $\lceil \lg n \rceil$, this second set of counters requires only $O(n \lg \lg n)$ bits and greatly reduces the search time for the sparse table. In our application, adding this second set of counters is not of great interest because the search time in PI-INVERSE is already dominated by the non-gateway iterations, and storing the counters would increase the lower order terms for how many words we have to encode.

The scheme described in Section 3 works for any duplicate key values, and returns one match. We could also request *all* records matching a given key value or even all records having a range of values for a given key tupe. A slight modification of our algorithm can output all matches in time $O(p \lg n)$, where p is the number of matches.

If values for all k keys are specified, then we can perform search in at most $2k \lg n$ comparisons by examining each cluster separately. It also makes sense to ask for a record matching some arbitrary subset of field values. We can do this type of search if the records are sorted in some lexicographic order where the search fields happen to be a prefix (in any internal order) of the lexicographic order. A simple generalization of our scheme makes this possible in $O(\lg n)$ time: pretend the record consists of 2^k fields, each representing one lexicographic order on the real keys. A further generalization might be to include searches on any constant number of predetermined functons on the record. We handle such queries in a similar manner by assigning a cluster of guides to each one. These extensions illustrate the difficulty of proving a lower bound.

Two open problems that we do not address are

1. Can one design a table that supports searching under one key in polylogarithmic time and also allows efficient multidimansional range searching?

2. Can one design the table to be dynamic, perhaps only to the extent of changing key values (without inserting or deleting records)?

The approaches developed here have already been applied to rather different problems. [FNSS88] utilizes several of these ideas to design a multikey hashing scheme in which a full hash table can be searched under any key in worst-case time $O(1)$. Subsequent to our work, X. J. Shen (personal communication) has studied variants of our methods.

ACKNOWLEDGMENTS

The authors thank Eli Upfal for bringing together the clusters of California and New York authors, who were originally working independently on the same problem. The authors thank Sandeep Bhatt, Manuel Blum, John Brzozowski, Richard Cole, Michael Fredman, Sampath Kannan, Dick Karp, Bud Mishra, Christos Papadimitriou, and Murray Sherk for helpful conversations.

REFERENCES

[AMM84] H. ALT, K. MEHLHORN, AND J. I. MUNRO, Partial match retrieval in implicit data structures, *Inform. Process. Lett.* **19** (1984), 61–65.

[Ben75] J. L. BENTLEY, Multidimensional binary search trees used for associative searching, *Comm. ACM* **18** (1975), 509–517.

[BFMUW86] A. BORODIN, F. E. FICH, F. MEYER AUF DER HEIDE, E. UPFAL, AND A. WIGDERSON, Tradeoff between search and update time for the implicit dictionary problem, *in* "Proceedings, 13th Internat. Colloq. Automata, Language, and Programming," Rennes, France, Lecture Notes in Computer Science, Vol. 226, pp. 50–59, Springer-Verlag, New York/Berlin, 1986; *Theoret. Comput. Sci.* **58** (1988), 57–68.

[CH82] R. COLE AND J. HOPCROFT, On edge coloring bipartite graphs, *SIAM J. Comput.* **11** (1982), 540–546.

[ES35] P. ERDŐS AND G. SZEKERES, A combinatorial problem in geometry, *Compositio Math.* **2** (1935), 463–470.

[FNSS88] A. FIAT, M. NAOR, A. A. SCHÄFFER, J. P. SCHMIDT, AND A. SIEGEL, Storing and searching a multikey table, extended abstract, *in* "Proceedings, 20th ACM Symp. on Theory of Computing," Chicago, 1988, pp. 344–353.

[FNSS88] A. FIAT, M. NAOR, J. P. SCHMIDT, AND A. SIEGEL, Non-oblivious hashing, extended abstract, *in* "Proceedings, 20th ACM Symp. on Theory of Computing," Chicago, 1988, pp. 367–376.

[Gab76] H. N. GABOW, Using Euler partitions to edge color bipartite multigraphs, *Int. J. Comput. Inform. Sci.* **5** (1976), 345–355.

[Hal35] P. HALL, On representations of subsets, *J. London Math. Soc.* **10** (1935), 26–30.

[Mun79] J. I. MUNRO, A multikey search problem, *in* "Proceedings, 17th Allerton Conference on Communication, Control, and Computing," Monticello, Illinois, 1979, 241–244.

[Mun86] J. I. MUNRO, An implicit data structure supporting insertion, deletion, and search in $O(\lg^2 n)$ time, *J. Comput. System Sci.* **33** (1986), 66–74.

[Mun87] J. I. MUNRO, Searching a two key table under a single key, *in* "Proceedings. 19th ACM Symp. on Theory of Computing," New York City, 1987, pp. 383–387.

[MS80] J. I. MUNRO AND H. SUWANDA, Implicit data structures for fast search and update, *J. Comput. System Sci.* **21** (1980), 236–250.

[TY79] R. E. TARJAN AND A. C.-C. YAO, Storing a sparse table, *Comm. ACM* **22** (1979), 606–611.