A SPARSE TABLE IMPLEMENTATION OF PRIORITY QUEUES

by

Alon Itai [1], Alan G. Konheim [2] and Michael Rodeh [3]

1. INTRODUCTION

Priority queues have been defined in several ways. In this paper a *priority queue* is a data structure on which the following operations can be executed:

Search(x), *Insert*(x), *Delete*(x), *Min*, *Next*(x), *Scan*.

There are numerous implementations of priority queues, in which each of the first five operations requires no more than O(log n) time (n is the number of keys currently in the queue). All of these implementations use trees: 2-3 trees [AHU], AVL-trees, weight balanced trees [RND], binomial trees [V] and require large overhead, both in time and in space. In many cases the algorithms devote much of their running time and storage manipulating the priority queue, often rendering a theoretically efficient algorithm to be infeasible or inefficient for all practical purposes.

An implementation of priority queues by means of *sparse tables* is presented in this paper. Data is stored in a linear array and requires only a single pointer. Insertion requires three operations:

- *searching* – to locate the table address at which a record will be inserted,
- *moving* – shifting of records in the table to free space for the record to be inserted, and

---

[1] Computer Science Department, Technion – Israel Institute of Technology, Haifa, Israel.

[2] Mathematical Sciences Department, IBM Thomas J. Watson Research Center.

[3] IBM Research, San Jose Laboratory. On sabbatical from IBM Israel Scientific Center, Technion City, Haifa, Israel.

- *reconfiguring* – increasing the size of the table and distributing the keys.

Searching a table of size m requires O(log m) time using binary search and O(log log m) time using interpolation search [PIA]. An exact analysis of the complexity of the move operation is given in [IKR]. It shows that although the number of moves on the worst case is O(m), the expected number of moves is bounded by a constant provided the density is bounded away from 1. In [IKR] an extensive use of generating function theory and calculus is made. In Section 3 an O(1) bound on the expected number of moves is derived for a special case which is similar to a structure proposed by Melville and Gries [MG] and Franklin [Fr]. Both Melville and Gries and Franklin relate the insertion of keys into a sparse table to the length of a probe in hashing with linear probing [KN2, KW, BK]. While such a relationship exists, the probabilistic models underlying these two processes differ so that the analyses offered in [Fr] and [MG] are incorrect. We discuss this briefly in Section 4.

To improve the worst case behavior, a more complicated sparse table scheme is introduced in Section 5 which requires no more than O(n log ^2n) time to build up a table with n elements.

Deletion may be implemented by marking the records that have been deleted. This idea was also used by Bentley et. al. [BDGS] and by Guibas et. al. [GMPR] and is discussed in Section 6.

2. SPARSE TABLE SCHEMATA

Associate a *key* $K = K(R)$ with each record R such that the correspondence between records and keys $R \Leftrightarrow K(R)$ is biunique. When records are stored in a table with their keys in sorted order, *Search*(K) is efficient. If records are stored contiguously, *Insert*(K) is carried out by moving records to free space for the record to be inserted. The efficiency of insertion will be improved by introducing gaps in the table thereby storing n records in a

table of capacity m for some $m>n$. A key is assigned to each address in the table by introducing fictitious or *dummy* keys. We describe the state of the table by the vector

$$y = (y_0, y_1, \ldots, y_{m-1}) \qquad y_0 \le y_1 \le \ldots \le y_{m-1}$$

indicating with this notation that the (genuine or dummy) key y_i is stored at address i for $0 \le i < m$. If the genuine keys in the table are

$$y_{i_0} < y_{i_1} < \ldots < y_{i_{n-1}} \qquad 0 \le i_0 < i_1 < \ldots i_{n-1} = m - 1$$

then dummy keys, each with the value y_{i_t}, are stored at each address in the contiguous block of $i_t - i_{t-1} - 1$ addresses

$$i_{t-1} + 1, i_{t-1} + 2, \ldots, i_t - 1 \qquad (0 \le t < n;\ i_{-1} = -1)$$

so that

$$y_0 = \ldots = y_{i_0} < y_{i_0+1} = \ldots = y_{i_1} < \ldots < y_{i_{n-1}+1} = \ldots = y_{i_n - 1}$$

For example, genuine keys are located at addresses 2, 5, 6 and 8 in the table

$$y = (2,2,2,3,3,3,4,5,5) \qquad n = 4, m = 9.$$

It will be useful to consider the table as a circular array: Address 0 follows address $m - 1$, address calculations are made modulo m, and a pointer is used to point to the origin of the table.

The building of a table by means of the insertion of keys is determined by two sequences of real numbers:

$$\{n_k : 0 \le k < \infty\} \quad \text{and} \quad \{m_k : 1 \le k < \infty\}$$

$$1 = n_0 < n_1 < \ldots < n_k < \ldots \qquad n_k \le n_{k-1} m_k \qquad 1 \le k < \infty$$

which have the following interpretation:

(i) The size of the table m can only take the values $n_{k-1} m_k$ for $k = 1,2, \ldots$

(ii) The size of the table is $m = n_{k-1} m_k$ when the number of distinct keys in the table n satisfies $n_{k-1} < n \le n_k$.

(iii) To insert the key x into a table of size $m = n_{k-1} m_k$ containing $n_{k-1} < n < n_k$ genuine keys, the address s satisfying

$$y_{s-1}<x<y_s \qquad 0\le s<m$$

is determined by a binary search. $s = 0$ means that x is either smaller or larger than any key presently in the table. If y_s is a dummy key, it is replaced by x yielding the table

$$(y_0, \ldots, y_{s-1}, x, y_{s+1}, \ldots, y_{m-1}).$$

If y_s is a genuine key, the block of t (say) consecutive genuine keys

$$y_s \neq y_{n+1} \neq, \ldots, y_{s+t-1} \neq y_{s+t} = y_{s+t+1}$$

is *moved* circularly to the right one position, the pointer to the origin of the table is adjusted if necessary and x is inserted at address s yielding the table

$$(y_0, \ldots, y_{s-1}, x, y_s, \ldots, y_{s+t-1}, y_{s+t+1}, \ldots, y_{m-1})$$

(iv) A *reconfiguration* of the table takes place when the key x is to be inserted into a table of size $m = n_{k-1}m_k$ containing n_k genuine keys. The size of the table is first increased to $n_k m_{k+1}$ and the n_k genuine keys

$$y_{i_0}<y_{i_1}<, \ldots, <y_{i_{n_k-1}}$$

are uniformly distributed in it. Then a binary search for x in the expanded table is carried out and the key x is inserted as in step (iii).

The numbers $\{m_k\}$ are the (multiplicative) *expansion factors*; the ratio $\rho = n/n_{k-1}m_k$ with $n_{k-1}<n\le n_k$ is the *density* of genuine keys in the table. Note that $1/m_k<\rho\le 1$. The expansion factor m_k adds approximately $\log m_k$ steps to the binary search but reduces the numbers of keys which must be *moved* to insert a new key. The cost of reconfiguration is $O(n_k m_{k+1})$ so that if the number of keys inserted since the last reconfiguration, $n_k - n_{k-1}$, is proportional to the size of the expanded table, the cost of reconfiguration per key is constant. As for the worst case, $O(n)$ keys may be moved to insert the n^{th} key. We will show in Section 3 that the expected number of moves remains bounded as n_k increases provided ρ is bounded away from one.

3. AVERAGE BEHAVIOR OF SPARSE TABLES

Let us consider a special case of the sparse tables schemata by assuming that the table size is of the form 2^k and by setting a threshold $\alpha < 1$ on the density of the table such that when the threshold is exceeded, the table is reconfigured. These assumptions conform with the definition of sparse tables if α is of the for m $a/2^b$ (for integral a and b) and $k > b$. If these reasonable conditions hold then $m_k = \alpha/2$.

Let $(K_0, \ldots, K_n)$ be the keys to be inserted into the table. We shall investigate inserting K_n into the table containing $\{K_0, \ldots, K_{n-1}\}$. Since only the relative order of the keys affects the algorithm, we assume that the keys are a permutation of the integers $\{0, \ldots, n\}$. Furthermore, because of the circular symmetry we assume that $K_n = 0$. Since the expected number of move operation increases with the density, the expected number of move operations is maximal just before a reconfiguration, i.e. $n = n_k - 1$. We shall, therefore, confine our analysis to this case. Let $N = n_{k-1}$, then $n = 2N - 1$.

Suppose a move of length j was required to insert K_{2N-1}, then

$$(y_0, \ldots, y_j) = (1,2, \ldots, j-1, j, j).$$

Let a_j be the number of keys in this segment of the table which also belong to $\{K_0, \ldots, K_{N-1}\}$. As a result of the most recent reconfiguration, at least $\lfloor j/m_k \rfloor$ and at most $\lceil j/m_k \rceil$ elements were put there and thus

$$\lfloor j/m_k \rfloor \leq a_j \leq \lceil j/m_k \rceil.$$

Therefore the number of ways to choose $\{K_0, \ldots, K_{N-1}\}$ is at most

$$\binom{j}{a_j} \binom{2N-1-j}{N-a_j}$$

and the probablility to make such a choice is

$$p_j \leq \binom{j}{a_j} \binom{2N-1-j}{N-a_j} / \binom{2N-1}{N}.$$

Therefore the expected number of moves is bounded by:

$$E = \sum_{j=1}^{\infty} j p_j.$$

If $a_j = \lceil j\alpha/2 \rceil$ then it is easy to show that

$$\binom{j}{a_j} \leq C_0 \binom{j}{a_j - 1}$$

where C_0 depends only on α. Therefore, by Stirling's formula [Fe],

$$\binom{j}{a_j} \leq C_1 (\beta\gamma j)^{-1/2} (\beta^{\beta}\gamma^{\gamma})^{-j}$$

where $\beta = a_j/j$, $\gamma = 1-\beta$ and C_1 depends only on α.

Other approximations which follow from Stirling's formula are (C_2 and C_3 are constants):

$$\binom{2N-1}{N} > C_2 N^{-1/2} 2^{2N}$$

$$\binom{2N-1-j}{N-a_j} < \binom{2N-j}{N-\lfloor j/2 \rfloor} < C_3 (2N-j)^{-1/2} 2^{2N-j}.$$

Therefore, there exists a constant C_4 which depends only on α such that for sufficiently large N:

$$\begin{aligned}
E &< C_4 \sum_{j=1}^{N/(1-\alpha/2)} j(\beta\gamma j)^{-1/2} (\beta^{\beta}\gamma^{\gamma})^{-j} N^{1/2} 2^{-2N} (2N-j)^{-1/2} 2^{2N-j} \\
&= C_4 (\beta\gamma)^{-1/2} \sum_{j=1}^{N/(1-\alpha/2)} j^{1/2} 2^{-j} (\beta^{\beta}\gamma^{\gamma})^{-j} (N/(2N-j))^{1/2} \\
&= C_4 (\beta\gamma)^{-1/2} \sum_{j=1}^{N/(1-\alpha/2)} j^{1/2} (N/(2N-j))^{1/2} (2\beta^{\beta}\gamma^{\gamma})^{-j} \\
&< C_4 (\beta\gamma)^{-1/2} ((2-\alpha)/(2-2\alpha))^{1/2} \sum_{j=1}^{N/(1-\alpha/2)} j^{1/2} (2\beta^{\beta}\gamma^{\gamma})^{-j} \\
&< C_4 ((2-\alpha)/(2\beta\gamma(1-\alpha)))^{1/2} \sum_{j=1}^{\infty} j^{1/2} (2\beta^{\beta}\gamma^{\gamma})^{-j}.
\end{aligned}$$

Since $\alpha < 1$ and $\beta < 1/2$ we have:

$$(2\beta^{\beta}\gamma^{\gamma})^{-1} < 1.$$

Therefore, E is bounded by a function of α independent of N.

To summarize we have the following:

Theorem 1: On insertion (no deletions allowed) the average number of move operations is bounded by a constant. □

A corollary of Theorem 1 is the following:

Theorem 2: On the average, insertion requires O(log n) operations.

Proof: The binary search conducted to find the place where to insert the new key takes O(log m) = O(log n) time. The expected number of moves is constant. Reconfiguration

adds only constant time per each insertion. □

4. THE RELATIONSHIP OF KEY INSERTION TO HASHING WITH LINEAR PROBING

Both Franklin [Fr] and Melville and Gries [MG] apply the analysis of hashing with linear probing [KN2, KW] to sparse tables. However, in hashing, the address space is fixed (and is equal to the size of the table) while in sparse tables the address space varies, and is equal to the number of genuine keys in the table. Also, the order in which the keys are stored in a sparse table influences the total number of move operations. When hash tables are used, the total probe length does not depend on the order of key arrival. Therefore, the model used to analyze sparse tables must be different from the one used to analyze hash tables. For a more formal discussion see [IKR].

5. IMPROVING THE WORST CASE OF INSERTION

As indicated in Section 2, the worst case of insertion may be quite bad. For example, if $m_k = 2$ and $n = 1.5n_{k-1}$ then inserting a new key may cause n_{k-1} keys to move while the expected value tends to 5 as the table size increases (see [IKR]). To improve the worst case performance an additional structure is imposed on the sparse table, yielding a more complicated scheme which we refer to as the *hierarchical sparse table.* The basic idea is to *redistribute* the keys *locally* when the local density becomes high. The insertion algorithm to be described below is somewhat different than the one described in Section 2.

Let m be the size of the table, $h = \lfloor \log m - \log \log m \rfloor$ and $b = m/2^h$. Note that $\log m \leq b < 2 \log m$. Divide the table into $m/b = 2^h$ *blocks* $B_0, B_2, \ldots, B_{2^h-1}$; the first $2^h\lceil b \rceil - m$ blocks are of size $\lfloor b \rfloor$ and the others are of size $\lceil b \rceil$.

Now consider a full binary tree of height h with *leaves* $L_0, L_1, \ldots, L_{2^h-1}$ (scanned from left to right) and associate with each of its nodes v a segment $s(v)$ of the table as follows:

(i) To the leaf L_i associate the block B_i $\quad (0 \leq i < 2^h)$.

(ii) For an internal node with children u and w, $s(v) = s(u) \cup s(w)$.

For every node v let $m(v)$ be the size of $s(v)$. Thus if r is the root, then $m(r)=m$, the size of the table. The *density* $\rho(v)$ of v is the number of genuine keys in $s(v)$ divided by $m(v)$.

The nodes of the tree are divided into levels. The root r is at level 0, and the level of any other node is greater by one than that of its parent. The level of the leaves is obviously h. A distinct maximum density is associated with each of the levels. Let $0 \leq \tau_L < \tau_U \leq 1$ and define the sequence $\tau_0, \tau_1, \ldots, \tau_h$ of *threshold densities* of nodes in levels 0, 1, . . . ,h by:

$$\tau_q = \tau_L + q(\tau_U - \tau_L)/h \qquad 0 \leq q \leq h.$$

Thus $\tau_L = \tau_0 < \tau_1 < \ldots < \tau_h = \tau_U$ and $\tau_{q+1} - \tau_q = (\tau_U - \tau_L)/h$.

During the process of insertion into a hierarchical sparse table, the density $\rho(L_i)$ of each leaf L_i satisfies $\rho(L_i) \leq \tau_h = \tau_U$. However, it may happen that for an interval node v of level q that $\rho(v) > \tau_q$.

An insertion is performed as follows:

(i) Conduct a binary search and insert the new key as in Section 2.

(ii) Assume that the block to which a key was added is B_i. If the density of B_i is less than or equal to τ_h, then the insertion process has been completed. Otherwise, consider the ancestors $r = v_0, v_1, \ldots, v_{h-1}$ of L_i and find the maximal value of q for which $\rho(v_q) < \tau_q$. If such a q is found, then the genuine keys of $s(v_q)$ are redistributed locally; the size of $s(v_q)$ is not changed – only its genuine keys are evenly distributed. However, if no such q exists, then the density of the entire table is greated than or equal to $\tau_0 = \tau_L$ and the table size is increased.

One way to increase the size of the table is by expanding to a table of size nm_{k+1} where n is the number of genuine keys currently in the table. Note that n may be different that n_k so that the sequence $\{n_k : 0 \leq k < \infty\}$ no longer plays its former role. Another possibility is to reconfigure when $n = n_k$ even if there is no need to do so according to the local densities criterion. In this case, one can use $\tau_L = n_k/n_{k-1}m_k$ which conforms with the (regular)

sparse table scheme in the sense that in both schemata table expansion occurs for the same table contents.

The advantage of hierarchical sparse tables is the improvement of their *worst case* performance over the original schemata.

Theorem 3: Performing $n-n_{k-1}$ insertions into a hierarchical sparse table of size $m = n_{k-1}m_k$ requires at most $O((n - n_{k-1}) \log^2 m/(\tau_U-\tau_L))$ operations.

Proof: The density of a block is bounded by $\tau_h = \tau_U<1$. Therefore, each block contains some dummy keys and the length of a move is less than the size of two blocks $2\lceil b\rceil \leq 2 + 4 \log m$.

Some insertions trigger a redistribution of the entire table which costs m operations. However others are not immediately followed by redistribution. We wish to bound the entire time spent on redistribution while inserting $n-n_{k-1}$ keys. To this end we first estimate the number of insertions into $s(v)$ between two successive redistributions.

After redistributing $s(v)$, the density of $s(v)$ and therefore the densities of both of its children is at most τ_q where q is the level of v. At the next redistribution of $s(v)$, v must have at least one child u with density τ_{q+1} or higher. Thus, the density of $s(u)$ has increased by at least $\tau_{k+1}-\tau_k = (\tau_L-\tau_U)/h$. Hence at least $(\tau_U-\tau_L)m(u)/h$ dummy keys have been replaced by genuine keys in $s(v)$ between two redistributions of $s(v)$. The cost of a single redistribution of $s(v)$ is $m(v)$. Therefore, the cost per insertion is at most

$$m(v)/((\tau_U-\tau_L)m(u)/h) = (m(v)/m(u))h/(\tau_U-\tau_L).$$

However,

$$m(v)/m(u) \leq 2 + 1/b$$

and thus the cost per insertion for $s(v)$ is at most

$$(2 + 1/b)h/(\tau_U-\tau_L)$$

Each block has one ancestor at each level, and therefore each insertion contributes to at most h redistributions. Thus the cost of inserting $n-n_{k-1}$ keys is at most

$$(n - n_{k-1})(2 + 1/b)h^2/(\tau_U - \tau_L).$$

Since b and h are both of the order log m, the theorem is proved. □

As for the implementation of this tree, several possibilities exist:

(i) *Explicit Representation:* The tree is stored by using nodes to contain the current density ($\rho(v)$), and pointers to the two children. The pointers can be eliminated if we use an array where locations $2i$ and $2i+1$ are the children of location i. The number of leaves is $2^h = m/b$ and the number of nodes is less than twice as much. Thus the storage requirements are $o(m)$. On each insertion the densities of the ancestors must be updated. This can be done within $O(h) = O(\log m)$ time.

(ii) *Implicit Representation:* No tree structure is maintained. On insertion we first calculate the boundaries of the block which has received an additional key. Then the entire block is scanned to calculate its density. If the density is found to be too high then the sibling block is scanned to calculate the density of the common parent. This process is continued until arriving at the node v to be redistributed. The scan time is $O(m(v))$ which is proportional to the redistribution time, and therefore the worst case time bound does not change. However, a scan of length b must be conducted even if no redistribution takes place, thus increasing the cost of insertion somewhat.

(iii) *Compromise Representation:* The insertion cost using the implicit representation can be reduced if a vector containing the densities of the blocks is maintained. If no redistribution is required then we must only update the density of a single block (in $O(1)$ time). In case of redistribution of $s(v)$, the densities of all of the offsprings of v must be updated, but the time required for the update is negligible compared to the redistribution time. As for storage, the extra space is equal to the number of blocks $m/b = o(m)$.

Note that these schemata are equivalent in the sense that redistribution occurs for the same table states and affects the same blocks.

(iv) *Alternative Scheme:* As in the implicit representation, no tree structure is maintained. The boundaries of the blocks and their densities are calculated when needed. Redistribution occurs whenever a scan of length $2b$ or more is conducted. This implies that at least one block is full and requires redistribution. Note that in this scheme redistribution does not occur at the same time and does not occur for the same table state and does not have the same scope as in the other three schemata. An analysis similar to that used in proving Theorem 3 may be carried out.

5. DELETIONS

Deletions, though easy to implement, are difficult to analyze statistically. We propose two deletion strategies:

(i) *Physical removal* – to delete the key k from the table

$$y = (y_0, y_1, \ldots, y_{n_{k-1}m_k-1})$$

conduct a search to find s such that

$$y_{s-1} < k = y_s.$$

Suppose L satisfies

$$y_s = y_{s+1} = \cdots = y_{s+L-1} \neq y_{s+L}$$

where the subscripts are taken modulo $n_{k-1}m_k$. Then replace the block

$$(y_s, y_{s+1}, \ldots, y_{s+L-1}) \text{ by } (y_{s+L}, y_{s+L}, \ldots, y_{s+L})$$

obtaining the table

$$y' = (y_0, y_1, \ldots, y_{s-1}, y_{s+L}, y_{s+L}, \ldots, y_{s+L}, y_{s+L+1}, \ldots, y_{n_{k-1}m_k-1})$$

In addition to the reconfiguration, which occurs whenever we attempt to insert a key into a table $y = (y_0, y_1, \ldots, y_{n_{k-1}m_k-1})$ currently containing n_k keys, a reconfiguration will also occur whenever deletion reduces the number of genuine keys to some threshold. There are many ways to specify these contraction thresholds; the simplest is to reconfigure (after deletion) when the number of genuine keys remaining is n_{k-2}.

We are not able to provide an analysis of sparse tables under a sequence of insertions/deletions. To begin with, the set of possible table states attainable by a sequence of insertions/deletions is larger than the set of possible table states attainable by only insertions. (For example, delete the key 4 from the table (0, 0, 0, 1, 2, 2, 4, 4, 4, 6, 6, 6, 8, 8, 8).) The analysis of the pure insertion process is simplified by the existence of *renewal points* – the epochs of reconfiguration. The insertion/deletion process might be compared with a birth and and death process and the analysis given in Section 3 has determined a probability distribution on the state space of the pure birth (= insertion) process.

(ii) *Tagged deletions* – Like indexed sequential files (ISAM) this scheme requires an additional Boolean vector of length m to distiguish between genuine and dummy keys. A key is deleted by setting the appropriate entry to *false*. The physical removal of keys is postponed until reconfiguration time; until then, at least one copy of each key must remain. The execution time of deletion consists principally of the search time O(log m). Additional O(m_k) time is required to mark all the copies of the key to be deleted as dummy. By marking only the rightmost copy of a key as deleted, the additional operation requires only O(1) time.

6. FINGERS

Guibas et. al. [GMPR] introduced the idea of *fingers* (see also Brown and Tarjan [BT]): Assume that many search operations accumulate near some prespecified keys, called fingers. Given a key k which is close to some finger f, it is required to design an algorithm which searches for k in time O(log d) where d is the distance between the location of f and the location of k. This feature can be incorporated into the sparse table scheme by maintaining a sorted list for their fingers (by way of balanced trees ,say) through which the sparse table may be accessed and adding a Boolean field to each table entry to indicate

whether the entry is currently being pointed by a finger. Searching for a key k is done by first finding the appropriate finger, using it to access the table and then searching for k by the unbounded search technique (Bentley and Yao [BY]). As for insertion, the proper location is found and move operations takes place. When table elements are moved, the fingers which point to them are updated.

7. LINEAR SPARSE TABLES

Replace the circular table by a linear one, with additional space on the "right end". This extra space is used for storing keys which would otherwise shift the origin of the table. The additional amount of storage depends on the density. It is conjectured that for density ρ, bounded away from unity, $o(m)$ extra space is sufficient to preserve the O(1) bound on the expected number of moves.

8. CONCLUSION

The sparse table scheme is an extremely simple data structure. As indicated by Melville and Gries [MG], it can be used for sorting. Another application is to B-trees, where all nodes have the same prespecified size m, and the number of keys may be as low as $m/2$. Implementing each node as a sparse table trades a reduced search time within a node (from $O(m)$ to $O(\log m)$) for a minor increase in storage (one bit per node, for deletions). Even though many memory management systems (such as Buddy systems [KN1]) allocate space in predefined quantities, not many data structures take advantage of this. (The exceptions are hash tables, sparse tables and some list processing system with garbage collection.)

For constant m_k, average behavior of sparse tables is optimal (up to a constant). However, the worse case behavior is $O(n)$. To effectively control the worst case, a hierarchical scheme has been introduced, and an upper bound of $O(\log^2 n)$ has been proved. Whether this bound is tight remains an open question. Another open question is the average number of moves in a hierarchical scheme. We conjecture the bound is O(1) for constant m_k.

9. REFERENCES

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "The Design And Analysis Of Computer Algorithms", Addison-Wesley, 1974.

[BDGS] J. L. Bentley, D. Detig, L. Guibas and J. Saxe, "An Optimal Data Structure For Minimal-Storage Dynamic Searching", Computer Science Department, Carnegie-Mellon University, 1978.

[BT] M. R. Brown and R. E. Tarjan, "A Representation For Linear Lists With Moveable Fingers", Tenth Annual Symposium on the Theory of Computing, San Diego, California, 1978.

[BY] J. L. Bently and A. C. Yao, "An Almost Optimal Algorithm For Searching", *Information Processing Letters*, ***5***, 3, 1976, pp. 82-87.

[Fe] W. Feller, "An Introduction To Probability Theory And Its Applications", Volume 1, John Wiley, 1950.

[Fr] W. R. Franklin, "Padded Lists: Set Operations In Expected O(log log N) Time", *Information Processing Letters*, ***9***, 4, November 1979, pp. 161-166.

[GMPR] L. J. Guibas, E. M. McCreight, M. F. Plass and J. R. Roberts, "A New Representation For Linear Lists", *9th Annual Symposium Theory Of Complexity*, pp. 49-60, 1977.

[IKR] A.Itai, A. G. Konheim and M. Rodeh, "A Sparse Table Implementation Of Priority Queues", RC-8550, IBM Thomas J. Watson Research Center, November 1980.

[KN1] D. E. Knuth, "The Art Of Computer Programming : Fundmanetal Algorithms", Addison-Wesley, 1969.

[KN2] _____, "The Art Of Computer Programming : Searching And Sorting", Addison-Wesley, 1973.

[KW] A. G. Konheim and B. Weiss, "An Occupancy Discipline and Applications", *SIAM Journal Of Applied Mathematics*, ***14***, 6, November 1966, pp. 1266-1274.

[MG] R. Melville and D. Gries, "Sorting And Searching Using Controlled Density Arrays", ***TR 78-362***, Cornell University, Ithaca, New York, 1978. (See also *IPL*, July 1980, pp. 169-172.)

[PIA] Y. Perl, A. Itai and H. Avni, "Interpolation Search a LogLog N Search", *CACM*, ***21***, 1978, pp. 550-553.

[RND] E. M. Reingold, J. Nievergelt and N. Deo, "Combinatorial Algorithms: Theory And Practice", Prentice Hall, 1977.

[V] V. Vuillemin, "A Data Structure For Manipulating Priority Queues", *CACM*, ***21***, pp. 309-315, 1978.