Implicit Data Structures for Weighted Elements*

GREG N. FREDERICKSON

Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907

Several new data structures are presented for dictionaries containing elements with different weights (access probabilities). The structures use just one location in addition to those required for the values of the elements. The first structure supports a worst-case search time that is within a constant multiplicative factor of optimal, in terms of the rank of the weight of the desired element with respect to the multiset of weights. If the values of the elements that comprise the dictionary have been drawn from a uniform distribution, then a variation of this structure achieves average search times that are asymptotically very good. Similar results are established for data structures which handle the case in which the intervals between consecutive dictionary values also have access probabilities. Lower bounds are presented for the worst-case search complexity. © 1985 Academic Press, Inc.

1. INTRODUCTION

Additional space, beyond the minimum required to store key values, appears useful in the implementation of various data structures. Consider a dictionary for a set of n elements, \( v_1 < v_2 < \cdots < v_n \), with weights (access probabilities) \( p_1, p_2, \ldots, p_n \), and with the \( n + 1 \) intervening intervals having weights \( q_0, q_1, \ldots, q_n \). If \( 2n + 1 \) pointers are available, then the elements can be stored in an optimal binary search tree (Knuth, 1971, 1973), or a nearly optimal binary search tree (Bayer, 1975; Fredman, 1975; Gotlieb and Walker, 1972; Hu and Tucker, 1971; Mehlhorn, 1975; 1977). But without that much additional space, how well can one do? In this paper we show that with no additional space, structures with very fast search times can be created. Not only successful but unsuccessful search can be handled very efficiently within our framework.

We consider data structures that have only one additional location, used to hold \( n \), the number of elements. We may assume that the values are stored in the first \( n \) locations of a one-dimensional array. Such data struc-

* This research was supported in part by the National Science Foundation under Grants MCS-7909259 and MCS-8201083.
atures have been termed *implicit data structures* (Munro and Suwanda, 1980; Frederickson, 1983). A simple example of an implicit data structure is a list in which the values have been arranged in increasing order. For this structure binary search will yield $\Theta(\log n)$ time for both successful and unsuccessful searches. However, these search times are not sensitive to the probabilities of access.

We shall consider implicit data structures for which we can characterize the search time of an element in terms of the rank of its weight. The *rank* $k$ of the weight $p_j$ of an element will be taken to be the number of elements of weight no smaller than $p_j$. An example of an implicit structure whose search times are sensitive to the rank $k$ is a list in which the values are ordered by decreasing weights. Sequential search (Knuth, 1973) may be used to search the list, and will require $O(k)$ time if the search is successful, and $O(n)$ time if it is not successful.

The implicit structures that we present in the paper are considerable improvements over these obvious choices. Our first structure will realize $O(\log k)$ time for a successful search, and $O(\log n)$ time for an unsuccessful search. The second structure will achieve $O(\log k)$ time for both types of search, where the rank of a weight is taken over the multiset of $\{ p_j \} \cup \{ q_j \}$. These search times are within a constant multiplicative factor of optimal, as we shall show. Thus, with no additional space, we can achieve very good performance. In fact, $O(\log k)$ is a better worst-case guarantee than $O(\log 1/p_j)$ or $O(\log 1/q_j)$, the worst case bounds on access times in an optimal binary search tree, as we shall discuss. Further, if values are assumed to be drawn from a uniform distribution, then the structure may be adapted to exhibit $O(\log \log k)$ average access times for both successful and unsuccessful search.

Our paper is organized in the following manner. In Section 2 we present a technique for generating structures that are sensitive to weights of elements, and give our basic structure. In Section 3 we show that no implicit structure based on partial orders can perform substantially better than our basic structure, in terms of the multiplicative constant. We also compare its performance with that of binary search trees. In Section 4 we generalize our basic structure to perform well for unsuccessful search. In Section 5 we show how to achieve very fast average search times for both successful and unsuccessful search.

Portions of this paper have appeared in preliminary form in (Frederickson, 1981). Self-organizing heuristics to approximate our structures are presented in (Frederickson, 1984).

---

1 All logarithms are to the base 2. When $O(\log x)$ is used, we shall mean $O(\max \{ 1, \log x \})$. 


2. Weight-Sensitive Structures

We first describe a technique for building structures that are sensitive to probabilities of access. We note that this technique may have application to a large number of data query problems. The idea is to group together a number of elements of largest weights. Within the group, the elements are arranged to facilitate fast query time irrespective of the weights. This process is repeated, with the remaining elements partitioned into groups on the basis of weights, and arranged within each group to facilitate fast query within the group. A query in the whole structure may be processed by querying the groups in turn until either the query is answered or the groups are exhausted. More recently, Alt and Mehlhorn have formalized the properties that a query problem must possess in order to be amenable to such an approach (Alt, 1982).

We now apply this technique to build an implicit structure with fast access times for successful search. The elements are partitioned into groups \( A_i, \ i=0, 1, ..., s, \) on the basis of their weights. For some appropriately chosen function \( h(\cdot), \) group \( A_i, \ i<s, \) will be of size \( h(i), \) and group \( A_s \) will be of size at most \( h(s). \) The elements will be partitioned among groups so that the weight of any element in group \( A_i \) is not smaller than the weight of any element in group \( A_{i+1}, \) for \( i=0, 1, ..., s-1. \) The elements will be arranged in sequential memory by increasing group number, and within group by increasing value. The structure satisfies two consistent partial orders, one based on the weights of the elements and the other based on the values of the elements. It may be seen to be a generalization of both the simpler sorted-by-value and sorted-by-weight structures. Exclusive of the choice of \( h(\cdot) \) and \( s, \) our structure is essentially the same as the hierarchical organization of order \( s \) presented by Gill (1980). However, these choices are crucial both for making the structure implicit and for realizing the time bounds we claim, which Gill’s structure does not achieve.

A set of values, along with their weights and the weights of the failure intervals, are shown in Fig. 1. An example of our structure is given in Fig. 2, using the set of elements presented in Fig. 1. The dotted lines indicate the boundary of groups, but are not physically present in the structure. Note that we have not made use of the failure probabilities in

\[
\begin{array}{cccccccccccccc}
 j & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
 v_j & 5 & 9 & 13 & 17 & 22 & 28 & 31 & 33 & 42 \\
p_j & .07 & .04 & .12 & .16 & .02 & .08 & .02 & .03 & .03 \\
q_j & .03 & .02 & .02 & .04 & .02 & .02 & .11 & .01 & .13 & .03 \\
\end{array}
\]

FIG. 1. A sample problem instance.
building our first structure. The search for a desired element proceeds in the fashion indicated above. Each group is examined in order, starting with group $A_0$, until the element is found, or the groups are exhausted. Within each group a binary search is performed. The strategy may thus be viewed as a careful interleaving of sequential search and binary search. Note that individual values of $h(i)$ need not be stored, since they may be computed as needed during the search.

If our technique for building structures is to be effective, then the size of groups must be chosen appropriately. Let $g(m)$ be the query time in a group of $m$ elements arranged to support fast query times. A good choice for the size of the $i$th group appears to be $h(i) = g^{-1}(2^i)$. The idea is to have the query time in the $i$th group be approximately equal to the sum of the query times in the previous groups. For our dictionary problem, search within a group of $m$ elements will require time $g(m) = \log(m + 1)$. Thus, for structure $I$, we choose $h(i) = 2^i - 1$. This will result in approximately $\log \log n$ groups. Note that the groups increase rather rapidly in size, with group $A_i$ containing approximately the square of the number of elements in group $A_{i-1}$, for $i = 1, 2, \ldots, s - 1$. With this choice of $h(\cdot)$, our structure achieves the $O(\log k)$ performance that we have claimed.

**Theorem 1.** Implicit structure $I$ requires no more than $4 \log k + 1$ element comparisons for a successful search and $3 \log n + 1$ comparisons for an unsuccessful search, where $k$ is the rank of the weight of the desired element.

**Proof.** If the desired element is in group $A_0$, then only one comparison is required. Suppose the desired element is in group $A_r$, for $r > 0$. Then the rank $k$ of this element's weight will be greater than $h(r - 1)$. Further, the number of element comparisons needed to find the element will be no greater than

$$\sum_{i=0}^{r} \lceil \log(h(i) + 1) \rceil = \sum_{i=0}^{r} 2^i = 2^{r+1} - 1$$

$$< 4 \cdot 2^{r-1} = 4 \log(h(r - 1) + 1) \leq 4 \log k.$$
Suppose the desired element is not in the dictionary. If \( n = 1 \), then only one comparison is required. If \( n > 1 \), then the number of element comparisons to determine that the element is not in the dictionary will be no greater than

\[
\sum_{i=0}^{s-1} \lceil \log(h(i) + 1) \rceil + \log(n - h(s - 1)) < \sum_{i=0}^{s-1} 2^i + \log n < 3 \log n.
\]

We next establish a worst-case search time of \( \Omega(\log k) \) for any algorithm.

**Lemma 1.** Consider any search algorithm in a set of \( n \) values with associated distinct weights. For any \( j, 1 \leq j \leq n/2 \), there is an element whose weight is of rank \( k, j \leq k < 2j \), and whose search time is at least \( \log k - 1 \).

**Proof.** Consider any such \( j \). Consider any decision tree which models a comparison-based search within the set. There must be some element whose weight is of rank \( k, j \leq k < 2j \), and whose depth in the tree is at least \( \lceil \log j \rceil \geq \log k - 1 \). This follows since not all of the \( j \) elements can be at shallower depths.

Thus the \( O(\log k) \) performance in structure \( I \) is optimal to within a constant multiplicative factor. Since \( k \ll 1/\log k \), the bound of \( O(\log k) \) can sometimes be more descriptive than the bound of \( O(\log 1/\log k) \) for optimal binary search trees (Mehlhorn, 1979). The bound \( O(\log 1/\log k) \) can be quite bad for individual elements. Let \( v_j = j \) for \( j = 1, \ldots, n \), \( p_j = 2^{-j} \) for \( j = 1, \ldots, n-1 \), and \( p_n = 2^{-(n-1)} \). Thus \( \log 1/p_n = n-1 \). In this case the optimal search tree would consist of a single long path.

If element values are drawn from a uniform distribution, and the weights are independently distributed, then it is possible to do considerably better on the average. Since the weights are independent from the values, the values within each group may be viewed as drawn from a uniform distribution. To improve average search times, we can perform interpolation search (Peterson, 1957) instead of binary search within each group. For interpolation search, the average search time is \( g(m) = c \log \log m \), for some constant \( c \). We thus organize structure \( I' \) in the same manner as structure \( I \), but choose \( h(i) = 2^{2^i} \). Of course, this is an extraordinarily fast growing function. Practical estimates of memory sizes limit this structure to groups of index \( i \leq 3 \).

**Theorem 2.** Let element values be drawn from a uniform distribution, and let the weights be drawn independently from the values. Implicit structure \( I' \) supports an average access time of \( O(\log \log k) \) for a successful search and \( O(\log \log n) \) for an unsuccessful search, where \( k \) is the rank of the weight of the desired element.
Proof. If the desired element is in group $A_0$, then the expected number of comparisons is no greater than some constant. Suppose the desired element is in group $A_r$, for $r > 0$. Since interpolation search uses $O(\log \log m)$ comparisons on the average in a table of size $m$ (see Perl, Itai, and Avni, 1978; Gonnet, Rogers, and George, 1980; Yao and Yao, 1976), the expected number of comparisons to find the element will be no larger than

$$\sum_{i=0}^{r} c \log \log h(i) = c \sum_{i=0}^{r} 2^i = c(2^{r+1} - 1) < 4c2^{r-1}$$

for some constant $c$. Since $k > h(r-1)$, the expected number of comparisons is $O(\log \log k)$. If the desired element is not in the structure, then the expected number of comparisons is no larger than

$$\sum_{i=0}^{s} c \log \log h(i) = c \sum_{i=0}^{s} 2^i < c2^{s+1} \leq 4c \log \log n.$$ 

Since $\log \log k \leq \log \log n$ and $\log \log k \leq \log 1/p_k$, the expected search performance in structure $I'$ for an element whose weight has rank $k$ is asymptotically never worse than that of either interpolation search or optimal binary search trees. For certain distributions of weights, the average performance, taken over all elements in structure $I'$, is asymptotically superior to either interpolation search, optimal binary search trees, or both. Consider a distribution of weights in which $p_k = c_{fn}/f(k)$, where $c_{fn}$ is a normalization factor that depends on $n$ and the function $f(\cdot)$. For $f(k) = k \log k \log \log k$, the factor is $c_{fn} = 1/(\sum_{k=1}^{n} 1/f(k)) = \Theta(1/\log \log \log n)$. For $f(k) = \Omega(k \log k (\log \log k)^2)$, the factor $c_{fn}$ is constant.

Search in structure $I'$ will have average cost proportional to $\sum_{k=1}^{n} p_k \log \log k$. For $f(k) = k \log k \log \log k$ this sum evaluates to $O(\log \log n/\log \log \log n)$. For faster growth rates of $f(k)$, the sum is no larger. Thus our search is asymptotically superior to interpolation search whenever $f(k)$ is $\Omega(k \log k \log \log k)$. Search in a binary search tree will have average cost proportional to $\sum_{k=1}^{n} p_k \log 1/p_k$. For $f(k) = k \log k (\log \log k)^2$, substituting in the above formula yields an average search time of $\Theta((\log \log n)^2)$. For $f(k) = k (\log k)^2$, the average search time is $\Theta(\log \log n)$. Based on arguments such as those above, it can be shown that our search performance is superior to that in an optimal binary search tree whenever $f(k)$ is $O(k (\log k)^2)$.

Our structures may be built in $O(n)$ time, given that the elements are presented in sorted order by value. Let $H(i) = \sum_{L=1}^{i} h(L)$, for $i < s$, and $H(s) = n$. For $i = s-1, s-2, \ldots, 0$, select the element $e_i$ of $H(i)$th largest
weight from among the set of $H(i + 1)$ elements of largest weights. Partition
the elements on the basis of the weights of elements $e_i$. The total time for
the selections will be $O(\sum_{i=1}^{n} H(i))$, which is $O(n)$.

3. Lower Bounds for the Multiplicative Constant

In the previous section we have shown that an implicit structure can be
built that requires no more than $4 \log k + 1$ element comparisons in the
worst case for a successful search. In this section we show that one cannot
do appreciably better, when the structure is arranged to satisfy
simultaneously two consistent partial orders, one on the values and the
other on the weights. We also compare this with the worst case perfor-
mance of binary search trees that are organized to guarantee $O(\log k)$ per-
formance.

We establish our lower bound for what we term a uniform structure.
Consider an infinite but countable set of elements. Let two partial orders
be defined on this set, with transitive relations $R_v$ and $R_p$. One partial
order is meant to reflect relations between values of elements, so that $j R_v j'$
means $v_j < v_{j'}$, and the other to reflect the relative weights of elements, so
that $j R_p j'$ means $w_j > w_{j'}$. We consider such partial orders defined by
relations $R_v$ and $R_p$ which satisfy the following additional requirements.
First, the two partial orders must be consistent. The orders defined by
relations $R_v$ and $R_p$ are said to be consistent if whenever $j R_v j'$ holds for
any elements $j$ and $j'$, then neither $j R_p j'$ nor $j' R_p j$ holds. Second, there
are a positive, finite number of maximal elements with respect to $R_p$. An
element $j^*$ is maximal with respect to $R_p$ if there is no element $j$ with
$j R_p j^*$. Given such partial orders with relations $R_v$ and $R_p$, there is at least
one total ordering that has a maximal element. We choose a particular
such total ordering and name a relation that defines it as $R$. Without loss
of generality, we index the elements from the set with the natural numbers
Corresponding to their order in $R$. A uniform structure consists of the set of
elements, the two consistent partial orders defined by $R_v$ and $R_p$, and the
total order defined by $R$.

A countable set of elements with associated values and weights is stored
in a uniform structure if the assignment of set elements to the elements of
the structure satisfies the relations $R_v$ and $R_p$. An example of a portion of a
uniform structure is illustrated with graphs showing the relations $R_v$ and
$R_p$ in Figs. 3a and b, respectively. Note that transitive edges have not been
shown. The relation $R$ has $j R j + 1$ for all $j$. The uniform structure model
appears to be a natural way to address the interplay between value and
weight partial orders. However, the model does not take into account
techniques such as the encoding of pointers. It is not clear that such techniques would be particularly helpful.

For any uniform structure, there is a procedure that correctly searches a set stored in the structure. That is, given a set of elements stored in the structure, and a value to be searched for, a search procedure is a procedure that outputs yes if an element with that value is contained in the set. Note that since the set is infinite, the procedure will not halt if no element has that value. Such a procedure is allowed to use comparisons of the search value with values of elements. Thus search time will be measured as the number of such comparisons performed. Since the structure stores an infinite number of elements, search times cannot be characterized meaningfully in terms of the total number of elements. However, a search procedure can make use of the information conveyed in the weight relation to perform searches that are efficient in the following sense. The search time for an element in the structure can be described as a function of \( k \), where \( k \) is the rank of the weight of the element.

We first discuss the strengthening of the partial orders while maintaining their consistency. Providing additional information on which to base a search cannot make it more expensive to search. This strengthening will render the orders into a rather special form. We shall strengthen the partial order on weights \( R_\rho \) in the following way. Suppose elements \( j \) and \( j' \) are
related in the value relation $R_v$, and suppose we have $j R_p j''$ in the weight relation. We augment the weight relation with $j' R_p j''$, and adjust the accompanying total order $R$ accordingly. We note that no inconsistency results, since if $j'' R_p j'$, then $j$ and $j'$ would be related in both the value and weight partial orders. Similarly, if $j'' R_p j$ in the weight relation, then include $j'' R_p j'$. Repeat this procedure until the weight relation cannot be augmented further.

We next strengthen the partial order on values $R_v$. Let $A_0$ be the set of elements that are maximal with respect to the weight relation. We augment the value relation on elements in $A_0$ to be a total order, and adjust the given total order accordingly. This augmentation cannot result in any inconsistency, since as a result of the previous augmentation, no element in $A_0$ is related in the value partial order to an element not in $A_0$. We may consider the set of elements and the partial orders that result by removing the elements in $A_0$. The same process may be repeated, generating $A_1, A_2, \ldots$, and augmenting the partial order on values. The final result is a structure similar to those in the previous section, with elements partitioned into groups on the basis of weights, and ordered within groups on the basis of values. We assume the existence of a function $h()$ which gives the size of each group. As in Section 2, let $H(i) = \sum_{l=1}^{i-1} h(L)$.

We consider the performance of a search procedure as a function of $k$, where $k$ is the rank of the weight of the desired value. For any $i$, an element whose weight is of rank $H(i) + 1$ can be in any one of $h(i + 1)$ positions in the uniform structure. Thus an adversary can make a search procedure examine at least $\lceil \log(h(i + 1) + 1) \rceil$ elements in group $i + 1$ before finding the desired one. Since the elements in group $i$ will have weights of rank smaller than those in group $i + 1$, the worst case performance of a search as a function of $k$ cannot be worsened if, for all $i$, comparisons with elements in group $i + 1$ are performed only after all comparisons with elements in group $i$ are performed. Thus the search procedure described in the previous section may be seen to be a best search procedure for this structure.

We will consider such structures for which there are search procedures that require no more than $a \log k + b$ element comparisons for successful search, where $a \leq 4$ and $b$ are constants. We will establish a lower bound for these structures as follows. Consider a structure $I^*$ that realizes $a^* \log k + b$ performance. We generate a structure $I^*_+$ from this that realizes no worse than $a^* \log k + b + 3$ performance. We then establish a lower bound for searches in $I^*$, which implies a similar lower bound for $I^*$.

**Lemma 2.** Let $I^*$ be a uniform structure. Let a best search procedure for $I^*$ have a search time of at most $a^* \log k + b$ for any element whose weight is of rank $k$, and for constants $a$ and $b$. There is a uniform structure $I^*_+$ whose best search procedure has a worst-case search time of at most
\[ a^* \log k + b + 3, \text{ and whose group sizes are such that } h_3(i) + 1 \text{ is a power of } 2, \text{ and } 2h(i) \leq h(i+1) \text{ for all } i \geq 0. \]

**Proof.** Given \( I^* \), we can generate \( I_1^* \) by setting \( h_1(i) + 1 \) to the smallest power of 2 no smaller than \( h(i) + 1 \). The performance bound for \( I_1^* \) can be seen to be no worse than that for \( I^* \).

We next generate \( I_2^* \) from \( I_1^* \) such that \( h_2(i) + 1 \) is a power of 2, and \( h_2(i) \leq h_2(i + 1) \), for all \( i \). Initialize \( I_2^* \) to be \( I_1^* \), and repeat the following. Let \( i \) be the smallest index such that \( h_2(i) > h_2(i + 1) \). Let \( i' \) be the smallest index such that \( h_3(i') = h_3(i) \). Let \( i'' \) be the smallest index such that \( h_3(i'') > h_3(i) \). Consider the groups \( i', i' + 1, \ldots, i'' - 1 \). Replace these groups with \( r \) groups of size \( 2h_3(i') + 1 \), where \( r \) is the smallest integer such that \( \sum_{L=i'}^{i'' - 1} h_3(L) \leq r(2h_3(i') + 1) \). The above is repeated until \( h_2(i) \leq h_2(i + 1) \) for all \( i \).

Each replacement will increase the worst-case search time for elements from group \( i' \) by one comparison, but not increase the worst-case search time for elements from succeeding groups. This is immediate if the groups are replaced by just one group. If there are at least two replacement groups, a case analysis shows that elements landing in the second group do not have their worst-case search time increased. Note that one crucial case is when \( h_2(i' + 2) = h_2(i') \) and \( h_2(i' + 1) = h_2(i' + 3) = 1 \). The result for third or higher groups follows by induction.

We next generate \( I_3^* \) from \( I_2^* \), such that \( h_3(i) + 1 \) is a power of 2, and \( 2h_3(i) \leq h_3(i + 1) \) for all \( i \). Initialize \( I_3^* \) to be \( I_2^* \), and give each group a label 0. Repeat the following. Let \( i' \) be the smallest index such that \( h_3(i') = h_3(i'+1) \), and let \( i'' \) be the smallest index such that \( h_3(i'') > h_3(i') \). If the label of group \( i' \) is 0, replace the groups \( i', i' + 1, \ldots, i'' - 1 \) with \( r \) groups of size \( 2h_3(i') + 1 \), where \( r \) is the smallest integer such that \( \sum_{L=i'}^{i'' - 1} h_3(L) \leq r(2h_3(i') + 1) \). Give the first resulting group the label 1. If instead the label of group \( i' \) is 1, and there are two or more groups in the sequence \( i' + 1, \ldots, i'' - 1 \), perform similar operations on this sequence. If the label of group \( i' \) is 1, and \( i' + 1 = i'' - 1 \), then increase the size of group \( i' + 1 \) to be \( 2h_3(i') + 1 \), and label it with a 1. The above is repeated until \( 2h_3(i) \leq h_3(i + 1) \) for all \( i \).

We claim that the worst-case search time for any element is increased by at most 2 comparisons by these operations. If the label of group \( i' \) is 0, then a replacement increases by 1 the worst-case search time of an element from group \( i' \), and does not increase the worst-case search time for elements from groups \( i > i' \). By giving the first resulting group a label of 1, the group will not be affected by any later replacements. If the label of \( i' \) was 1, and there are two or more groups in the sequence, then a similar reasoning holds. Otherwise, the worst-case search time for elements from groups \( i > i' \) are increased by 1 only if \( h_3(i') + 1 = 4 \). This follows by considering the
cases \( h_2(i') + 1 = 2, 4, \) and \( \geq 8 \). It follows that the worst-case search time in \( I^*_k \) is at most 2 more than the corresponding one in \( I^* \). The lemma then follows. 

**THEOREM 3.** Consider a uniform structure that stores elements according to two consistent fixed partial orders, one on the values and the other on the weights. Then for any constant \( \varepsilon > 0 \), there are an infinite number of elements such that the number of element comparisons needed to find such an element in the worst case is no fewer than

\[
\frac{4}{1 + \varepsilon} \log k
\]

where \( k \) is the rank of the weight of the desired element.

**Proof.** Given a uniform structure \( I^* \), we consider its associated structure \( I^*_k \). We derive a lower bound on the worst case search time in \( I^*_k \) in terms of \( k \). This will then yield a bound on the performance of \( I^* \). For the proof, all group sizes and indices refer to \( I^*_k \). Let \( S(i) \) be the worst-case search time for an element that is in group \( A_i \). Define \( a_i \) by \( S(i) = a_i \log(h(i) + 1) \). Since \( H(i-1) \leq h(i) \), \( \sup_{i \to \infty} a_i \) exists. Let \( \delta < \varepsilon \) be a constant to be specified subsequently. Consider an \( r > 0 \) such that \( a_r \leq (1 + \delta) a_{r-1} \). An infinite number of such \( r \) must exist, else \( \sup_{i \to \infty} a_i \) would not exist. Incidentally, since we must assume that \( 1 \leq a_i \leq 4 \), then such an \( r \) must exist in every sequence of \( 2/\log(1 + \delta) \) consecutive indices \( i \). Let \( c \) be such that

\[
h(r-1) + 1 = (h(r) + 1)^c.
\]

Since \( h(r-1) < h(r) \), we have \( 0 < c \leq 1 \). Then

\[
S(r) = \log(h(r) + 1) + S(r-1)
\]
\[
= \log(h(r) + 1) + a_{r-1} \log(h(r) + 1)^c
\]
\[
= (a_{r-1} c + 1) \log(h(r) + 1).
\]

Thus \( a_r = a_{r-1} c + 1 \). By choice of \( r \), we have

\[
a_r \geq \frac{1}{1 - c/(1 + \delta)}.
\]

Let \( x \) be an element of largest weight in group \( r \). Then its rank will be

\[
k = H(r-1) + 1 \leq 2(h(r - 1) + 1) = 2(h(r) + 1)^c.
\]


Thus the time to search for $x$ will be

$$S(r) = a, \log(h(r) + 1)$$

$$\geq \frac{1}{c(1 - c/(1 + \delta))} \log k - \frac{1}{1 - c/(1 + \delta)}.$$ 

The value of $c$ that minimizes the above expression over $0 < c \leq 1$ is

$$c = \sqrt{(\log k)^2 + (1 + \delta) \log k - \log k}$$

$$= \frac{1 + \delta}{2} \frac{(1 + \delta)^2}{8 \log k} + \frac{(1 + \delta)^3}{16(\log k)^2} - \cdots.$$ 

Substituting, we get

$$S(r) \geq \frac{4}{1 + \delta} \left( \frac{1}{1 + O((\log k)^{-2})} \right) \log k - \frac{2}{1 + O((\log k)^{-1})}.$$ 

Since there are an infinite number of such $r$, $r$ can be arbitrarily large, and hence the rank $k$ can be arbitrarily large. We now choose $\delta$ such that for sufficiently large $k$, $3$ less than the above expression is at least $(4/(1 + \epsilon)) \log k$. Using Lemma 2, and considering sufficiently large $k$ in the above, the result then follows.

We have shown that any uniform structure will require essentially $4 \log k$ element comparisons in the worst case. It is natural to compare this performance with that of binary search trees that are designed to have good performance in terms of $k$. We define a $b$-optimal binary search tree to be a binary search tree in which, for a given constant $b$, the value of the constant $a$ is minimized in the bound $a \log k + b$ on the worst case search time. The constant $b$ must be at least $1$, as must $a$ also.

Our characterization of binary search tree performance in terms of $k$, the rank of the weight of the desired element, is new. More usual characterizations are in terms of $p_j$, as in $\Theta(\log 1/p_j)$ (Mehlhorn, 1979). We identify problem instances that generate $b$-optimal binary search trees with comparatively poor performance. In this context we identify a trade-off between the values of $a$ and $b$. In order to have the multiplicative constant $a$ approach $1$, the additive constant $b$ must increase. We describe our bounds in terms of Riemann's "zeta function," $\zeta(r) = \sum_{i=1}^{\infty} i^{-r}$, for $r > 1$ (see Knuth, 1969).

**Theorem 4.** Let a $b$-optimal binary search tree require $a \log k + b$ element comparisons in the worst case, for positive constants $a$ and $b$. As $n$ becomes very large

$$\zeta^{-1}(2^b + 1) < a < \zeta^{-1}(2^b).$$
Simpler, but looser, bounds are

\[ 1 + \frac{1}{2^{b+1}} < a < 1 + \frac{1}{2^b - 1}. \]

Proof. Let the elements be indexed such that \( v_i < v_{i+1} \), for \( i = 1, \ldots, n-1 \). Let \( k_i \) be the rank of the weight of element \( i \). Consider a \( b \)-optimal binary search tree. Let the level of a node be the number of edges on the path from the root to that node. The leaf for the failure interval \( (v_{i-1}, v_i) \) can be at level no greater than \( \lfloor a \log \max\{k_{i-1}, k_i\} \rfloor + b + 1 \). (Otherwise, the node that is parent of the failure leaf would be too low in the tree.) If the level of the failure leaf is smaller than this value, then the tree can be restructured by moving either \( v_{i-1} \) or \( v_i \) down to replace the leaf, and rearranging accordingly. To describe boundary conditions, let \( k_0 = k_{n+1} = 0 \). Clearly, a worst case set of elements for a \( b \)-optimal search tree is one in which \( \sum_{i=1}^{n+1} \lfloor a \log \{k_{i-1}, k_i\} \rfloor \) is minimized. This certainly occurs when \( k_i = i \).

Let \( f(i) = \lfloor a \log i \rfloor + b \). Thus the leaf for the failure interval \( (v_{i-1}, v_i) \) will be at level \( f(i) + 1 \). Let the height \( h \) of the tree be the largest level number in the tree. We now count internal nodes in the tree as the number of internal nodes in a complete tree of height \( h \), minus the number of internal nodes missing by having leaves at levels less than \( h \):

\[ 2^h - 1 - \sum_{i=1}^{n+1} (2^h - (f(i) + 1) - 1) = n. \]

Simplifying, we get

\[ 1 = \sum_{i=1}^{n+1} 2^{-(f(i) + 1)}. \]

Thus

\[ \frac{1}{2} \sum_{i=1}^{n+1} i^{-a} \leq 2^b < \sum_{i=1}^{n+1} i^{-a}. \]

Using the definition of \( \zeta(\cdot) \), and noting that equality in the lower bound will not be achievable for sufficiently large \( n \), the result follows directly. For the simpler bounds, note that

\[ \frac{1}{a-1} < \sum_{i=1}^{\infty} i^{-a} < 1 + \frac{1}{a-1}. \]
Thus, for large enough $n$,

$$\frac{1}{2 \ a - 1} < 2^b < 1 + \frac{1}{a - 1}.$$ 

The simpler bounds then follow.

4. Structures Sensitive to Unsuccessful Search

In Section 2 we presented a structure whose time for an unsuccessful search is not sensitive to the weight of the failure interval encountered. Of course it is not immediately apparent that it is possible to build an implicit structure that is sensitive to failure probabilities. Such a structure must encode the presence of a failure interval without the use of any additional storage. For such a structure, we take $k$ to be the rank of a weight of an element or interval with respect to the set of all weights. In this section we present structure $H$, a static implicit structure that realizes $O(\log k)$ time for both successful and unsuccessful search.

The structure may be built in the following fashion. The elements are organized into groups on the basis of weights of both elements and intervals. Elements for group $A_i$ are selected in turn by the following rule. Consider the next largest weight that has not already been examined. If it is the weight of an element, include the element in the group. If it is the weight of an interval, then consider the set of elements that have not been assigned to groups $A_L$, for $L < i$. Identify the two elements in this set that most immediately enclose the interval, i.e., the largest element smaller than values in the interval, and the smallest element larger than values in the interval. If the set contains no element smaller than values in the interval, use the largest value in the set. A similar wrap-round choice is made if the set contains no element larger than values in the interval. If both elements are already in group $A_i$, then nothing need be done. If only one of these elements is in group $A_i$, include the other one, and mark its weight as examined. If neither of these elements are in the group, and there is room for both, include both, marking their weights as examined. Otherwise, defer the interval until the next group, and choose a remaining element of largest weight to complete the group.

When a group has been completed, arrange the elements in nondecreasing order. Let the interval between adjacent values in the group be called a group interval. Let the two open-ended intervals with values larger than the largest in the group and values smaller than the smallest in the group comprise the remaining group interval. There will be $h(i)$ group intervals in group $A_i$. Mark each group interval with a 0 or 1, depending on whether
any values not in $\bigcup_{l=1}^{L} A_l$ fall into it. If none do, mark the group interval with 0, and mark all weights of unexamined intervals that fall into it as examined. An example of this structure is given in Fig. 4, using the set of elements shown in Fig. 1.

The search will proceed in a fashion similar to that in Section 2. The groups will be examined in order, with a binary search performed in each group. However, if a value is not found in one group, a check will be made to determine whether or not to continue on to the next group. The search will be terminated if the appropriate group interval is marked with a 0. If an element has a weight whose rank is $k$, then it must be among one of the first $2k - 1$ elements included in groups. If an interval has a weight of rank $k$, then it will have enclosing elements chosen among the first $2k + 1$ elements. The search may be seen to use $O(\log k)$ time for either a successful or unsuccessful search.

The above structure is not completely implicit, since it requires $n$ additional bits. For a practical implementation, it might be reasonable to allow one additional bit per data value. However, it is possible to do without this additional storage. We now show how to avoid using these bits but still preserve fast search times. The elements in each group will not be kept in completely sorted order. Instead, the sorted elements are partitioned into subgroups, each subgroup consisting of four adjacent elements. Within a subgroup, the elements are stored in any of sixteen permutations, in order to encode the sixteen combinations of values of the four bits. A portion of one group is shown in Fig. 5a, with the bits shown and the group partitioned into subgroups. The encoding of the bits by the permutations of elements in the subgroups is shown in Fig. 5b.

A search will be conducted as before, except that the binary search is done on first elements of subgroups. Within any one group, the first elements of subgroups will be in increasing order, irrespective of the particular permutations. If the desired element is not found directly by binary search in a group, then the binary search will have determined two subgroups, into one of which the desired value must fall. These two subgroups are then searched sequentially. If the value is not found, the permutation of the appropriate subgroup is decoded, and it is determined whether to continue the search or not.

![Fig. 4. Implicit structure H before the bits are removed.](image-url)
Let the function $h'(\cdot)$ for structure $II$ be essentially the same as in structure $I$, except multiplied by four: $h'(i) = 4h(i)$, for $i = 0, 1, \ldots, s$. Bounds on the access time that are $O(\log k)$ can now be achieved for both successful and unsuccessful search.

**Theorem 5.** Implicit structure $II$ requires no more than $4 \log k + O(\log \log k)$ element comparisons for either a successful or unsuccessful search, where $k$ is the rank of the weight of the desired element or the interval in which it falls.

**Proof.** If the desired element or the encoded failure interval is in group $A_0$, then only a constant number of comparisons are required. Suppose that the desired element or the encoded failure interval is in group $A_r$, $r > 0$. If the rank of the weight of the desired item is $k$, then by a previous discussion the item will be among the first $2k + 1$ elements. Thus $2k + 1 \geq h'(r - 1) + 1$, or $k \geq \frac{1}{2}h'(r - 1)$. The number of comparisons used in the binary searches will be no greater than

$$\sum_{i=0}^{r} \log(\frac{1}{2}h'(i) + 1) = \sum_{i=0}^{r} 2^i$$

$$= 2^{r+1} - 2 - 1 < 4 \cdot 2^{r-1} < 4 \log k.$$

The additional time needed to sequentially search two subgroups and decode a permutation will be $\Theta(1)$ for each of $\Theta(\log \log k)$ groups searched.
An implicit structure with a simpler organization and $O(\log k)$ search time has recently been presented (Frederickson, 1984). However, the constant multiplicative factor on the number of element comparisons is larger for that structure than for our implicit structure $\Pi$.

Let $h(i) = 2^{2^i}$, as in structure $\Gamma$, and let elements of the last two groups be partitioned randomly. The interpolation search may again be used within the groups, and one would expect faster access times to result, if values are drawn from a uniform distribution. The intuition is that with very high probability the distribution of elements within any group would be rather close to a uniform distribution. For any interval handled before the last group, its two most nearly enclosing values in the original set will be used with high likelihood. This can be seen since if there are $m$ elements in a group, then there are only $O(2^{\sqrt{\log m}})$ in all preceding groups. We have been unable to make this intuition more substantial. However, we have been able to construct a similar structure and analyze the performance of a similar search procedure, which realizes the desired $O(\log \log k)$ performance. We present these results in the next section.

The time to construct structure $\Pi$ in the manner originally described is worse than linear. However, a structure with essentially the same $O(\log k)$ performance may be constructed in $O(n)$ time, if the elements are presented in sorted order by value. Since groups $A_i$, $i = 0, 1, \ldots, s-2$, will contain $O(\sqrt{n})$ elements, a heap on element weights and a heap on interval weights may be used to extract the elements for all groups except $A_{s-1}$ and $A_s$. These last groups may be constructed with elements partitioned arbitrarily between them, and access times will still be $O(\log k)$.

5. The Expected Performance of a Related Structure

In the preceding section, we presented an implicit structure that may be searched efficiently with respect to both successful and unsuccessful search. We claimed that if elements are drawn from a uniform distribution, weights are independent from values, and $h(i) = 2^{2^i}$, then there is reason to suspect that the average search times will be $O(\log \log k)$. We have not been able to substantiate this conjecture analytically. However, it is possible to show that a related structure may be searched in $O(\log \log k)$ time.

Our new structure $\Pi''$ is built in the following manner. The elements are partitioned among groups $\tilde{A}_i$, $i = 0, 2, \ldots, s_0$, so that each group $\tilde{A}_i$, with $i < s_0$, has $h(i)$ elements, and group $\tilde{A}_{s_0}$ has no more than $h(s_0)$ elements. The partition will be based on the weights of the elements, with the weight of any element in $\tilde{A}_i$ being no smaller than the weight of any element in $\tilde{A}_j$, for $i < j$. The same elements are partitioned a second time, among groups $\tilde{A}_i$, $i = 1, 3, \ldots, s_1$, with each group $\tilde{A}_i$, $i < s_1$, having $\frac{1}{2}h(i)$ elements, and
group $\tilde{A}_j$, having no more than $\frac{1}{2}h(s_i)$ elements. This partition will be based on weights of intervals, with an element assigned the weight of the interval immediately following it. If values in the set are drawn from a uniform distribution and weights are independent from values, then the elements in each group $\tilde{A}_i$ may be viewed as drawn from a uniform distribution.

Given the groups $\tilde{A}_i$, which together contain each element twice, we generate the groups $A_i$, $i = 0, 1, \ldots, s$, which will together contain each element exactly once. For $i < s$, group $A_i$ will contain $h(i)$ elements, and group $A_s$ will contain more than $h(s)$ and no more than $h(s + 1)$ elements. $A_0$ will be identical with $A_0$. To generate $A_i$, for $0 < j < s$ and $j \equiv 0 \mod 2$, given that $A_0, A_1, \ldots, A_{j-1}$ have been generated, we do the following: Initially set $A_j$ to $A_j$. For any element in $A_j$ that is in $\bigcup_{i=1}^{j-1} A_i$, replace it with an element of largest weight from the complement of $\bigcup_{i=1}^{j-1} A_i$. To generate $A_j$, $0 < j < s$ and $j \equiv 1 \mod 2$, given that $A_0, A_1, \ldots, A_{j-1}$ have been generated, we do the following. Initially set $A_j$ to $A_j$. For each element in $A_j$, insert the next largest element in the original set into $A_j$, eliminating duplicates in $A_j$. For any element in $A_j$ that is in $\bigcup_{i=1}^{j-1} A_i$, replace it with the most nearly enclosing element for that interval chosen from the complement of $\bigcup_{i=1}^{j-1} A_i$. If any additional elements must be inserted to fill out $A_j$ to its size of $h(j)$, choose these from $\tilde{A}_{j+1} \setminus \bigcup_{i=1}^{j} A_i$.

Once the groups are formed, bits may be used to mark the group intervals, and the bits may then be encoded by permutations, as in the preceding section. We claim that the resulting structure will support very fast average search times, if the elements have been drawn from a uniform distribution. Instead of interpolation search within the groups, we shall perform binary interpolation search (Perl and Reingold, 1977). We make this change to simplify the analysis.

We now quickly review binary interpolation search. As in interpolation search, an estimate is formed of the position at which the element is expected to be found, and a probe is made at that position. If the element is not at that position, then the second probe is made at a position $\lceil \sqrt{m} \rceil$ away from the initial probe in the direction toward the element, where $m$ is the number of elements in the subarray that is being searched. Probes will be made at distances of $\lceil \sqrt{m} \rceil$ further out until an interval of width $\lceil \sqrt{m} \rceil$ is found within which the desired element would lie. The procedure is then applied recursively to this interval. If elements are drawn from a uniform distribution, then the expected time for a search using binary interpolation search is $O(\log \log n)$ (Perl and Reingold, 1977). We now show that binary interpolation search works well for search in the groups of our structure II'.

**Theorem 6.** Let element values be drawn from a uniform distribution, and let weights be drawn independently from the values. Implicit structure II'
supports an average access time of \(O(\log \log k)\) for either a successful or unsuccessful search, where \(k\) is the rank of the weight of the desired element or the interval in which it falls.

**Proof.** Clearly the elements in group \(A_0\) will appear to be drawn from a uniform distribution, so that search in group \(A_0\) will require on average no more than \(c \log \log h(0)\) comparisons, for some constant \(c\). For group \(A_j\), where \(0 < j < s\), and \(j \equiv 0 \mod 2\), all but at most \(H(j-1)\) elements will be from \(\tilde{A}_j\). The expected position of a value \(x\) in \(\tilde{A}_j\), which will be used as the probe position, will be

\[
\hat{\mu} = \sum_{L=0}^{m} L \binom{m}{L} x^L (1-x)^{m-L} = xm
\]

where \(m = h(j)\), and element values range between 0 and 1. Let \(W\) be the position of the smallest value greater than or equal to \(x\) in \(A_j\). The expected value \(\mu\) of \(W\) will differ from \(\hat{\mu}\) by at most \(H(j-1)\). The variance of the position of \(x\) in \(\tilde{A}_j\) will be

\[
\sigma^2 = \sum_{L=0}^{m} (L - \hat{\mu})^2 \binom{m}{L} x^L (1-x)^{m-L} = x(1-x)m \leq \frac{1}{4}m
\]

The variance \(\sigma^2\) of \(W\) will differ from \(\sigma^2\) by at most \((H(j-1))^2\).

For \(i \geq 3\), the probability that at least \(i\) probes are necessary to find a search interval of size \(\sqrt{m}\) in \(A_j\) is less than

\[
\text{Prob}(|W - \hat{\mu}| > (i-2) \sqrt{m}) \leq \text{Prob}(|W - \mu| > (i-2) \sqrt{m - H(j-1)})
\]

which, by Chebyshev's inequality (Feller, 1968), is

\[
\leq \frac{\sigma^2}{((i-2) \sqrt{m-H(j-1)})^2} \leq \frac{\hat{\sigma}^2 + (H(j-1))^2}{((i-2) \sqrt{m-H(j-1)})^2}
\]

which, since \(H(j-1) < \sqrt{m}/10\), gives

\[
\leq \frac{(1/4)m + m/100}{(i-2.1)^2 m} = \frac{.26}{(i-2.1)^2}.
\]

By arguments similar to those in (Perl and Reingold, 1977), it can be shown that in any group \(A_j\), with \(0 < j < s\) and \(j \equiv 0 \mod 2\), the search will
require on the average no more than $c \log \log h(j)$ comparisons, for some constant $c$. Similar arguments will apply to group $A_j$.

The analysis of the search in $A_j$, $0 < j < s$ and $j \equiv 1 \mod 2$, is more complicated. Let $\mu$ and $\sigma^2$ be the expected value and variance of the position of a value $x$ in a set of $h(j)$ elements drawn from a uniform distribution. Construct set $\tilde{A}_j$, by inserting a value between every pair of values in $\tilde{A}_j$. Since elements in $\tilde{A}_j$ may be viewed as drawn from a uniform distribution, the expected position $\tilde{\mu}$ of a value $x$ in $\tilde{A}_j$ will differ from $\mu$ by at most one. Similarly, the variance $\tilde{\sigma}^2$ of the position of a value in $\tilde{A}_j$ will differ from $\sigma^2$ by at most one.

Suppose there are two elements in $\bigcup_{i=1}^{j} \hat{A}_i$, such that one immediately follows the other in the complete set. We claim that the probability of this occurrence is rather small. The probability that no two elements are side by side is

$$\frac{n-3}{n-1} \times \frac{n-6}{n-2} \times \cdots \times \frac{n-3(H(j)-1)}{n-(H(j)-1)}$$

$$= \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{4}{n-2}\right) \cdots \left(1 - \frac{2(H(j)-1)}{n-(H(j)-1)}\right)$$

$$> 1 - \frac{(H(j)-1)H(j)}{n-(H(j)-1)}.$$

For $j < s$, $H(j)$ will be $O(2^{\sqrt{\log n}})$. Thus the probability that there are two elements side by side will be $o(1/h(j))$. Therefore the expected contribution to the search time in this case will be $O(1)$, since the worst case cost of the search in $A_j$ cannot be larger than $h(j)$.

Suppose there are no two elements that are side by side. Then the expected position of a value in $\tilde{A}_j$ in this case can differ from the expected position in $\tilde{A}_j$ in every case by at most $h(j) \cdot c' / h(j) = c'$, for some constant $c'$. Furthermore, for every element in $\tilde{A}_j$, there is a next largest element available to be placed in $A_j$. Let these elements will be the ones placed between elements in $\tilde{A}_j$ to give $\tilde{A}_j$. Now all but at most $H(j-1)$ elements in $A_j$ will be supplied from $\tilde{A}_j$. Thus the search time may then be analyzed in a fashion similar to the case of $A_j$, for $0 < j < s$ and $j \equiv 0 \mod 2$.

From the preceding discussion, it can be seen that the expected search time in each groups $A_j$ will be no greater than $c \log \log h(j)$ for some constant $c$. If an element or failure interval has weight whose rank is $k$, then it must be in group $A_j$, where $j < j' + 2$ and $j'$ is the smallest index such that $H(j') \geq 2k$. From this it may be seen that the expected search time is $O(\log \log k)$. \[\square\]
6. DISCUSSION

We have identified implicit data structures that allow for fast access times in dictionaries where elements have probabilities of access. Only element values are stored in our structures, and the exclusion of weights from our structures appears crucial to obtaining interesting results. If weights were available and saved, then insertion would appear to be quite expensive. Upon insertion, the element of smallest weight in the receiving group would have to be identified and relocated. Maintaining a structure so that this element can be found quickly, while retaining fast search on element values, appears difficult. Dynamic search trees with explicit weights have been studied in (Mehlhorn, 1979; Bent, Sleator, and Tarjan, 1985).

A comment may be in order about the size of the groups in our data structures: they grow rather rapidly. This may in fact be taken as an indication that relatively few elements can have weights large enough to make it necessary to give them treatment more special than that of the overwhelming majority of the elements.

ACKNOWLEDGMENT

I would like to thank Jesse Barlow for several stimulating discussions.

RECEIVED March 1, 1985; ACCEPTED July 5, 1985

REFERENCES


