JOURNAL OF COMPUTER AND SYSTEM SCIENCES 47, 424–436 (1993)

Surpassing the Information Theoretic Bound with Fusion Trees

MICHAEL L. FREDMAN*

Rutgers University, New Brunswick, New Jersey, Bellcore, and University of Califonia, San Diego, California

AND

DAN E. WILLARD[†]

State University of New York, Albany, New York

Received October 23, 1990; revised September 2, 1991

This paper introduces a new sublogarithmic data structure for searching, the fusion tree. These trees lead to improved worst-case algorithms for sorting and searching, surpassing the limitations of the information theoretic lower bound. © 1993 Academic Press, Inc.

1. INTRODUCTION

This paper introduces a new sublogarithmic data structure for searching, the fusion tree. These trees lead to improved worst-case algorithms for sorting and searching, surpassing the limitations of the information theoretic lower bound. (We define search operations so that an unsuccessful search returns the predecessor of the search key.)

The information theoretic bound asserts that sorting N numbers requires $N \log N$ comparisons in the worst case. The degree of universality of this bound has not been fully resolved. Paul and Simon [4] have established that any unit cost random access machine algorithm whose operations include addition, subtraction, multiplication, and comparison with zero (but not division or bit-wise Boolean operations) requires $\Omega(N \log N)$ time to sort N numbers in the worst case. On the other hand, with the inclusion of division and Boolean operations, they obtain a linear time algorithm. This linear time algorithm, however, suffers from an abuse of the unit cost assumption. In particular, the algorithm generates operands during the course of its computation whose individual lengths are N^2 times the length of the largest of the N numbers to be sorted.

* Research supported partially by NSF Grant CCR-8504245.

[†] Research supported partially by NSF Grant CCR90-60509 and by Bellcore.

What seems to be needed is a computational model that avoids the potential abuses of the unit cost random access machine, but allows for unit cost operations among operands of "reasonable" size, i.e., operands of size commensurate with the sizes of the numbers to be sorted. Accordingly, we consider a reformulation of the sorting problem wherein our machine has a b-bit word size, and each of the N input numbers is assumed to be a non-negative integer less than 2^b (and hence each fits into one word). Moreover, it is desirable that a sorting algorithm use only $O(N)$ words of memory.

We assume throughout this paper that the number of data items that are present never exceeds 2^b, the universe size. (To cope with a larger number of items, we could proceed by storing bucket pointers in our data structure, with equal items placed in common buckets.)

Our "finite universe" reformulation of the sorting problem suggests the use of finite universe priority queues and similar types of data structures. For example, there is the priority queue of Van Emde Boas *et al* [5] which accommodates priority queue operations in time $\log\log U$ per operation, where $U=2^b$ is the universe size. This structure requires U words of memory, however. An alternative is Willard's Y-fast tries [6] in conjunction with the dynamic perfect hashing methods of Dietzfelbinger *et al.* [3], which brings the space down to linear in N, and accommodates priority queue operations in amortized randomized time $\log\log U$ per operation. Besides the fact that either too much space of randomization is required, these approaches improve upon conventional sorting only for sufficiently large values of N (i.e., when $\log N \gg \log\log U$).

Another finite universe data structure for implementing priority queue operations is described in Ajtai, Fredman, and Komlos [1]. This data structure is defined only within the cell probe model of computation and cannot be implemented with normal machine instructions. However, it is tantalizing in that it accommodates priority queue operations in constant time per operation for N sufficiently small compared with U. Ajtai *et al.* [1] state as an open problem whether some variant of their data structure can be implemented in a more realistic machine model. This paper contains a partial solution to this problem; one, however, which is good enough to be embedded in a new type of search tree data structure: the "fusion tree."

Fusion trees accommodate dynamic search operations in amortized time $O(\log N/\log\log N)$ per operation on a random access machine with b-bit word size, whose instruction set includes addition, subtraction, multiplication, comparison, and bit-wise AND operations. With the use of fusion trees, we can construct a linear space $O(N\log N/\log\log N)$ worst case time-sorting algorithm. We emphasize that the implicit constant inside the O-notation is independent of b. Moreover, if we allow randomization and integer division, we show that dynamic searching and sorting can be accomplished in times, respectively, $O(\sqrt{\log N})$ and $O(N\sqrt{\log N})$.

Our algorithms have theoretical interest only; the constant factors involved in the execution times preclude practicality.

2. Notation and Other Preliminaries

We let $\text{bin}(a_1, a_2, ...)$ denotes the sum $2^{a_1} + 2^{a_2} + \cdots$ for non-negative a_i. To refer to the bits of a binary number, we consecutively number its bit positions with position zero corresponding to its least significant bit. Thus, if $a_1 < a_2 < \cdots$, then the quantity $\text{bin}(a_1, a_2, ...)$ contains ones precisely in bit positions $a_1, a_2, ...$.

We assume that the multipliction instruction produces a two-word product, one consisting of the most significant b bits, the other containing the least significant b bits. This fact enables us to effect a right shift; upon multiplying x by $\text{bin}(b-r)$, we obtain the result of right shifting x by r bits (in the significant portion of the product). This procedure depends, however, on having available the quantity $\text{bin}(b-r)$, either as a program constant or as the result of a previous computation.

We occasionally require operands that slightly exceed the size of a single machine word. In such circumstances, we use double precision methods.

We observe that the bitwise Boolean operations, OR and XOR (exclusive OR), can be implemented on our machine by making use of the constant $2^b - 1 = \text{bin}(0, 1, ..., b-1)$, subtraction (so that we can complement a word), along with the AND operation.

Given a set of numbers S and a number x, we denote by $\text{rank}_S(x)$ the value $|\{t \mid t \in S,\ t \leqslant x\}|$.

3. Overview

The fusion tree can be roughly regarded as an implementation of a B-tree [2] such that (a) the approximate degree B of an internal node is an increasing, unbounded function of N, and (b) when performing a key search, the correct child of each node along the search path can be determined in constant time, independently of B. This suffices to achieve $o(\log N)$ search time. It will be the case, however, that updating an internal node (inserting or deleting a child, splitting the node, etc.) will require time B^4. To keep the amortized cost of update operations under control, we modify our structure so that it can be viewed as a B-tree whose leaves are identified as the roots of weight balanced binary search trees of size approximately B^4. For example, when the size of one of these binary search trees becomes too large, it is split into two trees with its associated root (B-tree leaf) likewise splitting. This reduces the amortized number of updates to the internal portion of the B-tree by a factor of B^4, while increasing the search times only by an additive amount of $\log B$. (We require that every key in the set represented by the Fusion tree reside in these binary trees; when a key gets deleted it is unnecessary to remove its presence from the B-tree nodes, where it serves only as a search discriminator.) Since we will be able to maintain $\log B = \Theta(\log \log N)$ using routine methods, we achieve a search tree whose operations can be performed in $O(\log N/\log \log N)$ amortized time.

Our algorithms require a fixed number of constants whose values depend on the word size b. We assume that these constants are given as part of the algorithms, and we are not charged for their computations, which could depend on the word size, b. This introduces a small amount of non-uniformity (in terms of b) into our algorithms.

The core of our data structure lies in our ability to represent a B-tree node so that (a) we can determine the correct child of a node in constant time when conducting a search, and (b) we can update a node in time B^4. This aspect of our data structure is motivated by an extends the work of Ajtai *et al.* [1]. We turn our attention to these considerations.

Consider a B-tree node consisting of k keys $(B/2 \leqslant k \leqslant B)$ given by $S = \{u_1, ..., u_k\}$. In order to determine the correct child of the node when conducting a search for u, it suffices to compute $\operatorname{rank}_S(u)$. This is to be accomplished in constant time. Central to the representation of our B-tree node is the notion of compressed key representation, which we proceed to describe.

4. Compressed Key Representation

Given a set $S = \{u_1, ..., u_k\}$ consisting of b-bit numbers, we proceed to construct a set $B(S)$ of distinguishing bit positions and a set $K(S) = \{\hat{u}_1, ..., \hat{u}_k\}$ of r-bit numbers, where $r = |B(S)| \leqslant k - 1$.

Our definition of $B(S)$ proceeds recursively. If $|S| = 1$, then $B(S)$ is the empty set. Otherwise, let p be the most significant bit position that distinguishes between two of the numbers in S. Let $S_0 = \{v_1, ..., v_g\}$ consist of those u_i's in S for which u_i has a zero in bit position p, and let $S_1 = \{w_1, ..., w_h\}$ consist of those u_i's for which u_i has a one in bit position p. Note that g, $h \geqslant 1$ and $g + h = k$. Then $B(S) = \{p\} \cup B(S_0) \cup B(S_1)$.

Now define $\hat{u}_i$ to be the result of deleting from u_i all bits except for those occupying positions contained in $B(S)$. More formally, let $c_1 < c_2 < \cdots < c_r$ denote the elements of $B(S)$ in sorted order. Then the bit in position $j - 1$ of $\hat{u}_i$ is the c_jth bit of u_i. We refer to $\hat{u}_i$ as the compressed key representative of u_i, and we define $K(S) = \{\hat{u}_1, ..., \hat{u}_k\}$. Observe that this compression operation is order preserving among the elements of S.

Given an arbitrary b-bit number u not necessarily in S, we define $\hat{u}(S)$ likewise to be the result of extracting just those bits of u occupying positions in $B(S)$. The following lemma presents an important property of the compressed key representation.

Lemma 1. *Let $c_1 < \cdots < c_r$ be the elements of $B(S)$ in sorted order and define $c_0 = -1$, $c_{r+1} = b$. Let $\hat{u}_i$ be an arbitrary compressed key in $K(S)$, and suppose that for an arbitrary b-bit number $u \neq u_i$, the most significant bit position in which $\hat{u}(S)$ and $\hat{u}_i$ differ is position $m - 1$. (In other words, the full keys u and u_i agree in bit positions $c_{m+1}, ..., c_r$, but disagree in position c_m. If $\hat{u}(S) = \hat{u}_i$, then we define $m = 0$.) Assume that the most significant bit position p in which u and u_i differ satisfies*

$p > c_m$. Then $\text{rank}_S(u)$ *is uniquely determined by the interval (c_{j-1}, c_j) containing p, together with the relative order between u and u_i.*

Proof. First, observe that the assumption $p > c_m$, together with the fact that u and u_i agree in bit positions, $c_{m+1}, ..., c_r$ imply the p cannot be any of the c_i's. We proceed by induction on $k = |S|$. If $k = 1$, then the result is trivial. Now suppose $p \in (c_r, c_{r+1})$. By definition of c_r, all elements in S agree in the bit positions more significant than position c_r, and therefore, $\text{rank}_S(u) = 0$ or k, depending on whether $u < u_i$ or $u > u_i$. Now suppose that $p < c_r$. Then u and u_i agree down to bit position c_r. Assume without loss of generality that u and u_i have a zero in bit position c_r. Then (using the notation preceding the statement of the lemma), $u_i \in S_0 = \{v_1, ..., v_g\}$, and $\text{rank}_S(u) \leqslant g$ since $u \leqslant w_1, ..., w_h$ (as each of the w_i's have a one in bit position c_r). It follows that $\text{rank}_S(u) = \text{rank}_{S_0}(u)$, and we proceed (below) to reduce to the case $S = S_0$ and apply the induction hypothesis. We observe that $B(S_0) \subset B(S)$, and therefore, the sorted sequence $c'_1 < c'_2 < \cdots$, which comprises the elements of $B(S_0)$, is a subsequence of $c_1 < c_2 < \cdots$. The most significant bit position c'_q in which u and u_i differ (from among just the positions $c'_1, c'_2, ...$) clearly satisfies $c'_q \leqslant c_m$. Therefore, $p > c'_q$. Applying the induction hypothesis for S_0 and observing that the intervals (c_{j-1}, c_j) refine the intervals (c'_{j-1}, c'_j), the lemma follows.

The above lemma enables us to reduce the computation of $\text{rank}_S(u)$ to computing $\text{rank}_{K(S)}(\hat{u}(S))$ as we now sketch. By computing $\text{rank}_{K(S)}(\hat{u}(S))$, we determine the predecessor $\hat{u}_j$ and the successor $\hat{u}_{j+1}$ of $\hat{u}(S)$ in $K(S)$ (one of the two quantities $\hat{u}_j$ or $\hat{u}_{j+1}$ may not exist). Next, we compare u with u_j and u_{j+1}; we are done if $u_j \leqslant u \leqslant u_{j+1}$. Otherwise choose $\hat{u}_i$ ($i = j$ or $j+1$) to be the value among $\hat{u}_j$ and $\hat{u}_{j+1}$ having the longest prefix of significant bits in common with $\hat{u}(S)$. Since $u < u_j$ or $u > u_{j+1}$ (by assumption), it must be the case that (using the notation of Lemma 1) $p > c_m$, and therefore, Lemma 1 can be applied to determine the rank of u once we know the interval (c_f, c_{f+1}) containing p.

A convenient alternative description for the set $B(S)$ that will be used below is given as follows. Assume that the u_i's are sorted, so that $u_1 < u_2 < \cdots$. Let d_i be the most significant bit position in which u_i and u_{i+1} differ. Then $B(S) = \{d_1, d_2, ..., d_{k-1}\}$.

A difficulty with using compressed keys as well as with computing the transformation that maps u into $\hat{u}(S)$ (in constant time) is that there is no obvious way to relocate the appropriate extracted bits so that they form a consecutive block of bits. The desired properties of the representation, however, are preserved by any similar transformation that relocates the extracted bits into (possibly nonconsecutive) positions without changing their relative order. We assume that the unused positions are filled with zeros. Because we ultimately need to concatenate the compressed representations of the keys in S into a single b-bit word, it is desirable that the extracted bits be relocated into a relatively narrow, if not consecutive, field of bits. This can be accomplished by an appropriate multiplication along with bitwise AND operations (to clear out the unwanted bits) as we proceed to describe.

Let $c_1 < \cdots < c_r$ be the bit positions (elements of $B(S)$) that need to be relocated, and let u be the quantity from which we wish to extract these bits. Let $C = \text{bin}(c_1, c_2, ..., c_r)$. Assuming that the quantity C is available to us, we compute $v = u$ AND C to zero out the irrelevant portion of u. Next, we multiply v by a suitable quantity M to relocate the appropriate bits of v into a narrow field, and once again use AND to zero out irrelevant bits. Our task is to prove the existence (and ultimately construct within our update algorithm) this quantity M. Writing $M = \text{bin}(m_1, m_2, ..., m_r)$, if we ignore for the moment the interfering effects of cross terms and carries that occur with multiplication, we can regard the effect of multiplying v and M as the relocation of the bits in positions $c_1, c_2, ..., c_r$ to the respective locations $c_1 + m_1, ..., c_r + m_r$. Our choosing to ignore the other cross terms will be justified if we can also arrange for the r^2 sums, $c_i + m_j$, $1 \leqslant i, j \leqslant r$, to be distinct (which, in particular, implies that no carries can take place). Our goal, therefore, is to construct a set of integers, $m_1, ..., m_r$ such that (a) $m_1 + c_1 < m_2 + c_2 < \cdots < m_r + c_r$, (b) the $m_i + c_j$ are distinct, and (c), the sums in (a) fall into a "small" interval. Lemma 2 (below) provides the desired result. We further require that $m_1 + c_1 = b$, so that our newly compressed key is right justified in the significant portion of the product. This requirement is easily satisfied by translating the m_i's of Lemma 2 by a suitable constant. We redefine the notation $\hat{u}(S)$ (and likewise the definition of the compressed keys $\hat{u}_1, ..., \hat{u}_k$), to reflect the computationally accessible variant offered by this approach, noting again that the desirable properties are preserved.

LEMMA 2. *Given a sequence of r integers, $c_1 < c_2 < \cdots < c_r$, there exists a sequence of r integers $m_1, m_2, ..., m_r$ such that*

(a) $m_1 + c_1 < m_2 + c_2 < \cdots < m_r + c_r$

(b) *the r^2 sums $c_i + m_j$, $1 \leqslant i, j \leqslant r$ are distinct,*

(c) $(m_r + c_r) - (m_1 + c_1) \leqslant r^4$.

Proof. First, we prove that there exist integers $m'_1, ..., m'_r$ such that $m'_i + c_g \not\equiv m'_j + c_h (\text{mod } r^3)$ whenever $i \neq j$. Assuming for the moment that such integers can be found, it follows that the r^2 (non-reduced) sums $m'_i + c_j$ are all distinct, and moreover, by adding suitable multiples of r^3 to the m'_i to obtain the final values m_i, we can further satisfy the requirement $m_i + c_i < m_{i+1} + c_{i+1} \leqslant m_i + c_i + r^3$, so that (a), (b), and (c) are satisfied.

To prove the existence of $m'_1, ..., m'_r$, suppose that $t < r$ and that we have succeeded in choosing $m'_1, ..., m'_t$ satisfying $m'_i + c_g \not\equiv m'_j + c_h (\text{mod } r^3)$ when $i \neq j$. We show how to construct a suitable m'_{t+1}. Observing that $m'_{t+1} + c_i \equiv m'_s + c_j$ implies $m'_{t+1} \equiv m'_s + c_j - c_i$, we can choose m'_{t+1} to be the least residue not represented among the fewer than r^3 residues of the form $m'_s + c_j - c_i$, $1 \leqslant s \leqslant t$, $1 \leqslant i, j \leqslant r$. In this way, we successively compute the quantities $m'_1, ..., m'_r$.

5. Fusion

We proceed to describe the representation of a B-tree node. An example is provided at the conclusion of this section. Assume that the keys stored in the node are given by $u_1 < \cdots < u_k$, $k \leqslant B$. Let $S = \{u_1, ..., u_k\}$ and let $B(S) = \{c_1, c_2, ..., c_r\}$. Our B-tree node contains the quantity $C = \text{bin}(c_1, c_2, ...)$ as well as the quantity $M = \text{bin}(m_1, m_2, ...)$ which provides the multiplier used in the computation of $\hat{u}(S)$. The final AND operation performed in the computation of $\hat{u}(S)$ requires the quantity, $D = \text{bin}(c_1 + m_1, c_2 + m_2, ...)$, which is likewise to be made available. Also present is an array containing the u_i's in sorted order and an array containing the $\hat{u}_i$'s in sorted order.

We will describe below a constant time method for computing $\text{rank}_{K(S)}(\hat{u}(S))$ and $\text{rank}_{B(S)}(m)$, where m is an integer in $[0, b]$. We will also describe a constant time method for computing $\text{msb}(u, v)$, which is defined to give the most significant bit position in which the two b-bit quantities u and v differ. For the present, we assume these capabilities.

As outlined in the previous section, after computing $j = \text{rank}_{K(S)}(\hat{u}(S))$, we compare u with u_j and u_{j+1}. Assuming that u does not fall in the interval $[u_j, u_{j+1}]$, we determine which of $\hat{u}_j$ and $\hat{u}_{j+1}$ has the longest prefix of significant bits in common with $\hat{u}(S)$ by comparing $\hat{u}(S)$ XOR $\hat{u}_j$ with $\hat{u}(S)$ XOR $\hat{u}_{j+1}$ (the smaller value corresponds to the longer prefix). Assuming that $\hat{u}_h$, $h = j$ or $j+1$, has the longer prefix in common with $\hat{u}(S)$, we compute $m = \text{msb}(u, u_h)$ and then identify the interval (c_i, c_{i+1}) containing m by computing $i = \text{rank}_{B(S)}(m)$. We finally obtain $\text{rank}_S(u)$ by executing a table look-up indexed on h, i, and whether (a) $u < u_h$ or (b) $u > u_h$. The space requirement for our B-tree node is dominated by the size of this table, which is $O(B^2)$. Since the number of B-tree nodes is $O(N/B^4)$, the total amount of space required to represent the fusion tree is $O(N)$.

The constant time computation of $\text{rank}_{K(S)}(\hat{u}(S))$ proceeds as follows. Contained within our B-tree node is a quantity K_S into which the compressed keys from $K(S)$ are "fused" together—hence the name of our structure. More specifically, the bits of K_S are partitioned into $E = \lfloor b^{1/6} \rfloor$ equal size fields. (If b is not divisible by E, then the rightmost $E\lfloor b/E \rfloor$ bits are partitioned into equal size fields.) Each compressed key will occupy a separate field. As a consequence, the maximum allowable degree B of a node is constrained to satisfy $B \leqslant E$. Since, as indicated earlier, we can assume that $N \leqslant 2^b$, we have sufficiently many fields to maintain $\log B = \Omega(\log \log N)$, which achieves the stated speed-up over conventional sorting. Our limitation on the size of E comes from the fact that our compressed keys require up to k^4 bits per key. The compressed keys are right justified in their respective fields, and the leading two bits in each such field are zeros. The fields which do not contain compressed keys are filled with all ones except for a leading zero. The structure of K_S is shown below:

$$K_S = \underbrace{0\,1\cdots1}_{\text{Field } E} \quad \cdots \quad \underbrace{0\,0\cdots\hat{u}_k}_{\text{Filed } k} \quad \underbrace{0\,0\cdots\hat{u}_{k-1}}_{\text{Field } k-1} \quad \cdots \quad \underbrace{0\,0\cdots\hat{u}_1}_{\text{Field } 1}.$$

Now by a executing a suitable multiplication and bitwise OR operation involving $\hat{u}(S)$, we obtain the number Y shown below (the fields of Y correspond to those of K_S):

$$Y = \underbrace{1\,0 \cdots 0\,\hat{u}(S)}_{\text{Field } E} \quad \cdots \quad \underbrace{1\,0 \cdots 0\,\hat{u}(S)}_{\text{Field } 1}.$$

Next, we subtract K_S from Y and zero out all but the lead bits of the fields (performing a bitwise AND). The result contains a one only in those positions that are leading bits of fields for which K_S contained a $\hat{u}_i$ with $\hat{u}_i \leqslant \hat{u}(S)$. A suitable multiplication will now sum these bits, so that a portion of the resulting product will contain the desired rank, which is then extracted by an AND operation.

We use the same method to compute $\text{rank}_{B(S)}(m)$ in constant time; instead of using the quantity K_S this computation utilizes the quantity B_S in which the elements of $B(S)$ are fused together.

Our next task is to describe the constant time computation of $\text{msb}(u, v)$, the most significant bit position in which the b-bit quantities u and v differ. For the additional purpose of performing update operations on B-tree nodes as described later, we wish to augment this computation so that as a by-product of computing $m = \text{msb}(u, v)$, we also obtain the quantities, $\text{bin}(m)$ and $\text{bin}(b - m)$. Our computation trivially reduces to determining the position of the leftmost one in u XOR v, which we can write as $\lfloor \lg(u \,\text{XOR}\, v) \rfloor$ (lg denotes the binary logarithm). We will need the following lemma.

LEMMA 3. *We say that a number x is d-sparse provided that the positions of all of its one bits belong to a set of the form, $Y = \{a + di \mid 0 \leqslant i < d\}$, which consists of d consecutive terms of an arithmetic progression with common difference d. (Not all of these positions have to be occupied by ones, however.) If x is d-sparse, then there exist constants y_1 and y_2 such that for $z = (y_1 x)$ AND y_2, the ith bit of the significant part of z equals the bit in the position $a + di$ of x, $0 \leqslant i < d$. In other words, z is the result of "perfectly compressing" x.*

Proof. Let $a_i = a + di$ and let $t_i = b - a + (1 - d)i$, $0 \leqslant i < d$. Then $a_i + t_i = b + i$ and all sums $a_i + t_j$, $0 \leqslant i, j < d$ are distinct. We choose $y_1 = \text{bin}(t_1, t_2, ...)$, and the lemma follows easily.

The computation for $\lfloor \lg x \rfloor$ is divided into two phases. We assume that $x \neq 0$. Let $s = \lceil \sqrt{b} + 1 \rceil$. Consider a partitioning of the b bits of our word x into right justified contiguous blocks of s bits. (The leftmost block has possibly fewer than s bits.) The number of blocks is given by $\lceil b/s \rceil$. The first phase of the computation determines the leftmost block in which x contains a one, extracts, and then right justifies this block. The second phase of the computation finds the leftmost one in this extracted block.

Define C_1 to have ones precisely in the leftmost bit position of each block: i.e., $C_1 = \text{bin}(s - 1, 2s - 1, ...)$. Define C_2 to be the bit-wise complement of C_1. We first compute the function, lead(x), which enjoys the property that for each of its blocks, the leftmost bit within the block equals one if and only if the corresponding block in x contains a one. (All other bits in lead(x) equal zero.) lead(x) is given by $(C_1 - [(C_1 - x \text{ AND } C_2) \text{ AND } C_1])$ OR (x AND C_1). Because lead(x) is s-sparse, we can apply Lemma 3 to obtain (in constant time) a compression of the leftmost bits from each of the blocks of lead(x). Let compress(x) denote the result of this compression. Now let $P = \{\text{bin}(0), ..., \text{bin}(\lceil b/s \rceil - 1)\}$ denote the first $\lceil b/s \rceil$ non-negative powers of two. We set $b_1 = \text{rank}_P(\text{compress}(x))$. This rank computation proceeds in the same manner in which we computed $\text{rank}_{K(S)}$, except that we use $\lceil b/s \rceil$ blocks each consisting of s bits in which the relevant quantities are right justified. In place of the quantity K_S, a constant consisting of the powers of two fused together in right-to-left ascending order is used. (It is also necessary to use double precision arithmetic since the quantities involved slightly fill up more than single machine words.) The quantity b_1 identifies the block number (counting from the right) of the leftmost block of x containing a one.

In addition to obtaining the block number of the leftmost block containing a one, we can also obtain the quantity bin(r), where r is the position of the leftmost one of lead(x). Reviewing the computation for $\text{rank}_P(\text{compress}(x))$, after performing the subtraction and zeroing of all but the lead bits from each of the blocks of s bits (the moment just before we multiply), we obtain a quantity which we denote as L. Among the lead bits of the blocks of L, the rightmost b_1 of them will be ones, and the others will be zeros. Because the ones of L appear periodically with period s, we can extract the leftmost such one (the other ones being replaced by zeros) by subtracting from L the result of right shifting L by s bits. The position of this extracted one, however, exactly coincides with the position of the leftmost one of lead(x) since the blocks of lead(x) and L have a common size of s bits. In a similar manner, we can also obtain bin($b - r$) by redoing our rank computation, but using instead the powers of two fused together in left-to-right ascending order. Now if $b_1 = 1$ then we proceed directly to the next phase (only the rightmost block contains a one). Otherwise, we multiply x by $\text{bin}(b - r + s - 1) = \text{bin}(b - r) \cdot \text{bin}(s - 1)$ to extract (and right justify in the significant portion of the product) the leftmost block of x containing a one. This completes the description of the first phase of the computation for $\lfloor \lg x \rfloor$.

The second phase of the computation computes the position j of the leftmost one in our extracted block of x consisting of s bits. As before, we perform a rank computation of these s bits relative to the set consisting of the first s powers of two. To obtain bin(j), we proceed as in the first phase, and then employ Lemma 3 to perfectly compress the leading bits of the blocks. We obtain bin($s - j$) by repeating this computation, but reversing the order in which the powers of two are concatenated. The results, b_1, bin(r), bin($b - r$), j, bin(j), and bin($s - j$), of these two phases can now be combined to yield the result $m = \lfloor \lg x \rfloor$ of the msb(u, v) computation, augmented to include bin(m) and bin($b - m$).

To complete our discussion of the fusion tree, we need to indicate how a B-tree node can be built in time B^4. The various update operations on B-tree nodes can be regarded as special instances of the node building operation. We assume that we are given the set of discriminator keys, $S=\{u_1, u_2, ..., u_k\}$. First, we sort the keys obtaining (say) $u_1 < \cdots < u_k$. Next, we compute the set $B(S)$ which is given by $\{\text{msb}(u_i, u_{i+1}) \mid 1 \leqslant i < k\}$. The augmented computation of the $c_i = \text{msb}(u_i, u_{i+1})$ also yields the $\text{bin}(c_i)$ values. These values are sorted to obtain distinct copies and then summed to obtain the constant $C=\text{bin}(c_1, ..., c_r)$. The proof of Lemma 2 provides an algorithm to compute the m_i positions which define the multiplier M. However, we actually need the values $\text{bin}(m_i)$. Since the m_i's of Lemma 2 satisfy $b \leqslant c_i + m_i \leqslant b + r^4$, by starting with the $\text{bin}(b-c_i)$ values (provided by the augmented msb computations), we can obtain each $\text{bin}(m_i)$ value by performing $O(\lg r)$ multiplications (in effect constructing addition chains). All of the required computations for the construction of a B-tree node, including the construction of the look-up table stored in such a node, involve straightforward algorithms that can be performed in time polynomial in k. (k^4 time is easily seen to be sufficient.)

The inclusion of a fixed number of constants in our algorithms (whose values depend only on b) is necessitated by considerations typified by the following. The quantity K_S (in which the compressed keys are fused together) contains $b^{1/6}$ fields, all but the right-most k of which have the form (*) 011...1. Let w denote the constant obtained by concatenating together fields having the form (*), and let $w' = \text{bin}(f)$, where f is the common size of these fields. Using w and w' we can compute K_S in $O(k)$ time. (This assumes we have already computed the compressed keys which make up K_S.) Similarly, given $\hat{u}(S)$, the constant time computation of the quantity Y used in the $\text{rank}_{K(S)}(\hat{u}(S))$ computation utilizes certain constants.

Example. The important aspects of our B-tree node representation are illustrated by the following detailed example. Our node contains the key set $S=\{x, y, z\}$. The values of x, y, and z are given below along with the values of the various constructs associated with the node as discussed above. Again, bit positions are numbered from right to left starting with zero:

$x = 001001001110$

$y = 001001010110$

$z = 001100000111.$

The distinguishing bit positions are $c_1 = 4$ and $c_2 = 8$, so that

$B(S) = \{4, 8\}.$

$C = 000100010000$, and we choose our multiplier to be

$M = 000100100000.$

Observe that our multiplier M shifts the distinguishing bits into positions 12 and 13, or equivalently, the two rightmost positions in the significant portion of the product. Thus, to extract these bits, we use the mask

$D = 000000000011.$

Accordingly our compressed keys are given by (rightmost bits):

$\hat{x}=00$, $\hat{y}=01$, $\hat{z}=10$,

and the concatenation K_S of these compressed keys is given by

$$K_S=\underbrace{0010}_{\hat{z}}\ \underbrace{0001}_{\hat{y}}\ \underbrace{0000}_{\hat{x}}.$$

The constant B_S, the concatenation of the distinguishing bit positions, is given by

$$B_S=\underbrace{001000}_{c_2}\ \underbrace{000100}_{c_1}.$$

The look-up table giving $\text{rank}_S(u)$ as a function of h, i, and whether (a) $u<u_h$ or (b) $u>u_h$ (as described in the third paragraph of Section 5) is

	$u<u_h$			$u>u_h$		
h	i			i		
	0	1	2	0	1	2
1	0	0	0	1	2	3
2	1	0	0	2	2	3
3	2	2	0	3	3	3

Now suppose we wish to compute $\text{rank}_S(u)$, where $u=000100010111$. Utilizing the constants C, M, and D, we obtain $\hat{u}(S)=11$. Utilizing K_S we determine that $\text{rank}_{K(S)}(\hat{u}(S))=3$. Finding that $u<z$ (z being the third ranking element of S) and seeing that S does not have an element of rank 4, we conclude $h=3$. Next, computing $m=\text{msb}(u, z)$, we find that $m=9$. Computing $i=\text{rank}_{B(S)}(m)$, we obtain $i=2$. Indexing into our table with the values $h=3$, $i=2$, and $u<z$ ($z=u_h$), we conclude that $\text{rank}_S(u)=0$.

6. Extensions

We remark first that by suitably refining our fusion tree structure, we obtain the worst case as opposed to amortized time bounds.

If we are willing to either (a) relax the linear space restriction or (b) preserve the linear space restriction but use randomization and integer division, then we can achieve $\sqrt{\lg N}$ amortized time bounds for dynamic search operations. This is accomplished by using the data structure of van Emde Boas *et al.* [5] in the case of (a) (or, in the case of (b), using dynamic Y-fast tries [3.6]) when N exceeds

$$2^{(\log b)^2/36}. \qquad (*)$$

For such N, these data structures perform search operations in time $O(\log\log U) = O(\log b) = O(\sqrt{\log N})$. For smaller N, we use the fusion tree, but it is modified to maintain $B = \Theta(2^{\sqrt{\log N}})$. With this choice for B, the amortized complexities of our operations are given by $O(\log B + \log N/\log B) = O(\sqrt{\log N})$. (We also need to check that the constraint $B \leqslant E$ is satisfied when N does not exceed (*).) Thus, with the use of randomization and integer division, sorting can be accomplished in time $O(N\sqrt{\log N})$ and linear space.

7. Applications

Sorting and searching frequently occur as algorithmic components. Moreover, the commonly occurring priority queue data structure can be implemented as a special case of searching. The field of computational geometry offers some immediate applications of the fusion tree or simple modifications of the fusion tree. For instance, we have improved algorithms for planar convex hull construction, priority search trees and rectangle intersections, fractional cascading, and certain orthogonal range query problems.

Another application leads to an improved version of Willard's Q-fast tries [7]. As described above, the Y-fast trie in conjunction with dynamic perfect hashing [3, 6] provides a dynamic linear space data structure for searching with randomized amortized $O(\log\log U)$ time operations. This data structure provides the best known performance for large subset sizes. The Q-fast trie, on the other hand, provides the best known performance among linear space *deterministic* data structures, again for large subset sizes. Balanced search trees constitute a component part of the Q-fast trie. We can improve the performance of Q-fast tries by using fusion trees to implement these search tree components and by changing the parameters of the trie so that its height is given by $\sqrt{\log U/\log\log U}$, and its node degree is given by $2^{\sqrt{\log U \cdot \log\log U}}$. The resulting data structure provides $\sqrt{\log U/\log\log U}$ operation times, improving upon the original Q-fast trie by a $\sqrt{\log\log U}$ factor.

8. Open Questions

Two immediate open questions are: *How fast can we sort*? and *How fast can we search*? Some interesting variations of these questions are as follows. First, we ask if there exist possible exotic machine instructions that can lead to yet faster sorting and searching algorithms. Put another way, *Is RISC Risky*? We note that the instructions used in our algorithms belong to the class NC^1. This can be considered a reasonable restriction to impose on possible exotic instructions, although the question remains interesting even without this restriction. Several people have pointed out to the authors that while comparison is an AC^0 operation, our algorithms use multiplication, which does not belong to AC^0. Thus, it would be interesting to know if there exist fast sorting and searching algorithms that can be implemented with AC^0 instructions. Put another way, *Can fusion trees be implemented with brisk RISC*?

9. Concluding Remark

The decision tree model of computation enjoys the properties of being reasonably general, tractable, and oftentimes quite challenging. The beguiling successes of this appealing model of computation have led to many claims that various $N \log N$ algorithms are optimal, conclusions that should not be taken too seriously.

References

1. M. Ajtai, M. Fredman, and J. Komlos, Hash functions for priority queues, *Inform. and Comput.* **63** (1984), 217–225.
2. R. Bayer and E. McCreight, Organization and maintenance of large ordered Indices, *Acta Inform.* **1** (1972), 173–189.
3. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Robert, and R. E. Tarjan, Dynamic perfect hashing: Upper and lower bounds, *in* "Proceedings, 29th IEEE Symposium on Foundations of Computer Science, 1988," pp. 524–533.
4. W. Paul and J. Simon, Decision trees and random access machines, *in* "Symposium uber Logik und Algolrithmik, Zurich 1980"; K. Mehlhorn, "Sorting and Searching," pp. 85–97, Springer-Verlag, New York/Berlin, 1984.
5. P. Van Emde Boas, R. Kaas, and E. Zijlstra, Design and implementation of an efficient priority queue, *Math. Systems Theory* **10** (1977), 99–127.
6. D. Willard, Log-logarithmic worst case range queries are possible in space $O(N)$, *Inform. Process. Lett.* (1983), 81–84.
7. D. Willard, New trie data structures which support very fast search operations, *J. Comput. System Sci.* **28** (1984), 379–394.