JOURNAL OF COMPUTER AND SYSTEM SCIENCES 48, 533–551 (1994)

Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths

MICHAEL L. FREDMAN*

University of California at San Diego, La Jolla, California 92093, and Rutgers University, New Brunswick, New Jersey 08903

AND

DAN E. WILLARD†

SUNY at Albany, Albany, New York 12203

Received February 5, 1991; revised October 20, 1992

Two algorithms are presented: a linear time algorithm for the minimum spanning tree problem and an $O(m + n \log n/\log\log n)$ implementation of Dijkstra's shortest-path algorithm for a graph with n vertices and m edges. The second algorithm surpasses information theoretic limitations applicable to comparison-based algorithms. Both algorithms utilize new data structures that extend the fusion tree method. © 1994 Academic Press, Inc.

1. INTRODUCTION

We extend the fusion tree method [7] to develop a linear-time algorithm for the minimum spanning tree problem and an $O(m + n \log n/\log\log n)$ implementation of Dijkstra's shortest-path algorithm for a graph with n vertices and m edges. The implementation of Dijkstra's algorithm surpasses information theoretic limitations applicable to comparison-based algorithms. Our extension of the fusion tree method involves the development of a new data structure, the *atomic heap*. The atomic heap accommodates heap (priority queue) operations in constant amortized time under suitable polylog restrictions on the heap size. Our linear-time minimum spanning tree algorithm results from a direct application of the atomic heap. To obtain the shortest-path algorithm, we first use the atomic heat as a building block to construct a new data structure, the AF-heap, which has no explicit size restriction and surpasses information theoretic limitations applicable to comparison-based algorithms. The AF-heap, which belongs to the Fibonacci heap family [5, 6, 10], provides constant amortized costs for findmin, insertion, and decrease key

* Research supported partially by NSF Grant CCR-9008072.
† Research supported partially by NSF Grant CCR-9060509.

0022-0000/94 $6.00

operations, and $O(\log n/\log\log n)$ amortized cost for deletion (when there are n items in the heap). Our new implementation of Dijkstra's shortest-path algorithm is an immediate application of the AF-heap.

To relate these results to previous results, it is necessary to discuss the models of computation involved. The traditional algorithms for both the minimum spanning tree and shortest-path problems are comparison based. All ordering information is ultimately inferred on the basis of binary comparisons. The minimum spanning tree problem can be regarded purely as a comparison problem, since the minimum spanning tree of any graph is uniquely determined by the relative ordering of its edge weights. The traditional shortest-path algorithms more appropriately fall within the algebraic computation tree framework [3] since they perform arithmetic involving edge weights. The point of view taken in this paper is to allow ourselves more flexibility than afforded by comparison-based models, but nonetheless to conform to the assumptions implicit in the analysis of a typical shortest-path algorithm. Examining the criteria for evaluating such an algorithm reveals in particular the assumption that arithmetic operations can be performed in constant time for operands of size commensurate with that of the individual input values (e.g., edge weights). Not allowed, however, are computations that achieve hidden parallelism by doing operations on "long words," conforming to the reality that computers have fixed, bounded word length. We take as our model of computation the random access machine with word size b, where b is assumed to be only large enough to hold the edge weights (assumed to be integers) and also the number of vertices n of the input graph. (Thus we assume that $b \geqslant \log n$.) We allow the normal arithmetic operations as well as bitwise Boolean operations. We assume that when computing the product of two b-bit numbers the result consists of two words, one word consisting of the least significant b bits and a second word consisting of the most significant b bits. We refer to the second word as the *significant portion* of the product. We measure space in terms of words of memory required, and time in terms of machine operations on words.

We regard algorithms that utilize the addressing and perhaps other capabilities of the random access machine, falling outside the framework of comparison based algorithms, as being *trans-dichotomous.* A typical example is radix-sort. If, in addition, the time and space requirements of the algorithm are independent of the machine word size b, then we say that the algorithm is *strongly trans-dichotomous.* Radix sort is not strongly trans-dichotomous, as contrasted with the algorithms presented in this paper.

Our minimum spanning tree algorithm, the first such algorithm to run in linear time, slightly improves upon the previously fastest known algorithm, which has a running time of $O(m \log \beta(m, n))$, where $\beta(m, n) = \min\{i \mid \log^{(i)} n \leqslant m/n\}$ [8]. Our shortest-path algorithm, which runs in time $O(m + n \log n/\log\log n)$, slightly improves upon the $O(m + n \log n)$ implementation of Dijkstra's algorithm that uses Fibonacci heaps [6]. Expressed in terms of the word size b, the implementation of Dijkstra's shortest-path algorithm given by Ahuja, Mehlhorn, Orlin, and Tarjan [1] has a running time of $O(m + n\sqrt{b})$ and is faster or slower than our algorithm,

depending on the relative sizes of b and n. Using our terminology, the shortest-path algorithm of Ahuja *et al.* [1] is trans-dichotomous, but not strongly trans-dichotomous. Another implementation of Dijkstra's algorithm, with a running time $O(m \log b)$, uses the priority queue of Van Emde Boas *et al.* [12].

Our shortest-path algorithm is the first such algorithm to uniformly improve upon comparison-based algorithms over all ranges of word size. It remains unresolved whether a linear-time comparison-based algorithm exists for the minimum spanning problem, although we note that such an algorithm for minimum spanning tree verification has recently been discovered [4, 9].

We make no claims concerning the practical viability of these algorithms. To the contrary, the constant factors implicit in the big-O terms are too large.

We assume that the reader is familiar with the definitions of the standard heap operations, insert(x), delete(x), findmin, and decrease(x, decrement). We also assume familiarity with the notions of amortized complexity. We refer the reader to [6] for the relevant material.

2. Algorithms and Data Structures

Section 2.1 describes our minimum spanning tree algorithm. Section 2.2 gives an overview of the AF-heap. Our shortest-path algorithm is an immediate application of the AF-heap. Section 2.3 describes the construction of the atomic heap, which is used in Sections 2.1 and 2.2. Before proceeding to Section 2.1, we summarize some properties of the atomic heap.

The atomic heap is constructed by combining layers of an even smaller heap, the Q-heap. The Q-heap is structured in part as a finite state machine whose size grows rapidly with the size limitation of the Q-heap. This finite state machine requires explicit representation, the construction of which takes place during a preprocessing phase. The amount of time and space available for this preprocessing ultimately determines the limit on the maximum number of items that the heap can accommodate. The limitation on the size of the Q-heap likewise leads to a limitation on the size of the atomic heap.

The Q-heap accommodates findmin, insertion, and deletion operations in constant worst-case time and has a $(\log n)^{1/4}$ size limitation, where n is the amount of preprocessing time and space available. The atomic heap accommodates findmin, insertion, and deletion operations in constant amortized time and has a $\log^2 n$ size limitation, where n is the amount of preprocessing time and space available.

A further constraint for both of these heaps is that the word size b must satisfy $b \geqslant \log n$. This assumption is satisfied for our intended graph-theoretic applications. As explained in the Introduction, for the purpose of constructing graph algorithms we assume that $b \geqslant \log n$, where n is the number of vertices of the input graph. Under these circumstances we are prepared to invest $O(n)$ time and space for preprocessing, and the word size b is sufficient to satisfy the requirements for our heaps.

2.1. *Minimum Spanning Tree Algorithm*

Our algorithm is patterned after the algorithm presented in [6]. First we summarize that algorithm, quoting from [6].

> The idea is to grow a single tree [starting from an arbitrary vertex, as in Prim's algorithm] only until its heap of neighboring vertices exceeds a certain critical size. Then we start from a new vertex and grow another tree, again stopping when the heap gets too large. We continue in this way until every vertex is in a tree. Then we condense every tree into a single supervertex and begin a new pass of the same kind over the condensed graph. After a sufficient number of passes, only one supervertex will remain, and by expanding the supervertices, we can extract a minimum spanning tree.

We refer the reader to [6] for details on how a pass is implemented. In particular, if the critical heap size chosen for the first pass is k, then the number of vertices existing as the second pass begins is bounded by $2m/k$.

Our algorithm is a two-pass variant of the above. In the first pass, we use the atomic heap, which implements the required heap operations in constant amortized time per operation. The critical heap size for the first pass is given by $\log n$. The total time required for the first pass is $O(m)$, and the number of vertices in the resulting condensed graph is given by $n' = O(m/\log n)$. (Prior to the first pass, a preprocessing phase of time $O(n)$ is required to set up the look-up tables utilized by the atomic heap.) The second pass is implemented using a Fibonacci heap [6] and grows a complete spanning tree in time $O(m + n' \log n') = O(m)$. This discussion establishes the following theorem.

THEOREM. *Assuming the existence of the atomic heap as described at the beginning of Section* 2, *there exists a linear-time minimum spanning tree algorithm.*

2.2. *AF-Heaps*

We proceed to describe the AF-heap data structure. We assume the availability of atomic heaps as described at the beginning of Section 2. Throughout the remainder of this paper, we treat n as though it is a fixed upper bound, to within a factor of two, on the current size of the AF-heap. By suitably rebuilding our various structures when this size assumption becomes violated, resetting n in the process, we can maintain this relationship, incurring only a constant factor increase in the various amortized complexities. We are using here the fact that $O(n)$ time is required for rebuilding, including the preprocessing time required by the atomic heap so that it can accommodate as many as $\log^2 n$ elements.

We begin with an overview. The AF-heap consists of a forest of trees. Values are stored in the nodes of these trees, one value per node, so as to satisfy the usual heap order condition. The individual trees are structured so that each internal node has between $B/2$ and B children, where $B = \log n$, and all paths from the root to any of the leaves have common length h, where h is the height of the tree. In addition to its data value, stored in each node is the height of the subtree of that node and

also the number of children it has. A tree with t nodes has height $O(\log t/\log\log n)$, and since t does not exceed n, the maximum tree height is $O(\log n/\log\log n)$. The individual trees that constitute the forest of an AF-heap are restricted in that there can be at most $B-1$ trees of a given height. Consequently, there are fewer than $\log^2 n$ trees in one of these forests. The trees of a given height are placed in a common bucket and the number of trees in each bucket is maintained.

Attached to each internal node is an atomic heap containing the children of that node with keys corresponding to the key values stored in the children. Similarly, the roots of the forest are maintained in an atomic heap. A basic operation performed on these trees is the *ripple operation*; a ripple operation removes the value stored in a designated node. That node then has its value replaced with the smallest of the values stored among its children, which in turn has its value replaced etc. The leaf at which this process culminates is then removed from the tree. We defer for the moment our handling of the situation that arises when a node no longer has sufficiently many $(B/2)$ children. Since operations on atomic heaps take constant amortized time, the amortized cost of a ripple operation is given by the height of the node in the tree at which the operation is initiated. The ripple operation preserves heap order.

The findmin operation is accomplished by accessing the atomic heap containing the tree roots of the forest. An insertion operation is performed by creating a single node tree containing the inserted item and then inserting this tree into the forest. In general, whenever one of the buckets exceeds its capacity of $B-1$ tree roots, *consolidation* is performed as follows. A new node is created which becomes the root node of a new tree; the subtrees of this node consist of the trees from the overflowing bucket (of which there are B) and the value to be stored in this node is obtained by performing a ripple operation. The new tree is then inserted into the forest, which may trigger further consolidation. A deletion operation is performed by executing a ripple at the appropriate node. A decrease operation is performed by pruning away the subtree of the node containing the valued being decreased and then inserting this tree into the forest (having decreased the value stored in its root).

Next, we discuss the processing required when a node no longer has sufficiently many children. In this case we say that the node is *light*. The remaining subtrees of the node and the node itself are pruned from the tree and inserted into the forest as new trees. This may in turn trigger further pruning, further consolidation of tree roots, which in turn can lead to more pruning etc., possibly creating a shower of pruned tree fragments. These tree fragments are simply placed in a queue while awaiting their insertion into the forest.

To add precision to the above overview, there are issued of control that need to be addressed. As alluded to above, our heap operations require that we maintain a queue consisting of trees whose insertion into the forest is pending. Let Ex-Q denote this queue. Ex-Q is empty upon completion of any given heap operation. Our algorithms use an auxiliary procedure called *Pop-queue* that transfers the trees from Ex-Q and inserts them into the forest, simultaneously taking care of needed

consolidation. We now give a more precise description of the control aspects of the operations.

Insert(x). Create a single node tree containing x and enter this tree into Ex-Q. Then call Pop-queue.

Delete(x). Call Ripple(x). Then call Pop-queue.

Decrease(x, δ). Change the value stored in x, decrementing its value by δ. If x is the root of a tree in the forest, then we are done. Otherwise remove the subtree of x from the parent of x and enter it into Ex-Q. If the parent y of x is now light, then perform Prune(y). Finally, call Pop-queue.

Ripple(x). Perform the atomic heap operations described in the preceding overview to restore the value for the node x. Let y be the parent of the leaf that finally gets removed. If y is light, then call Prune(y).

Pop-queue. If Ex-Q is empty, then we are done. Otherwise, remove a tree T from Ex-Q. Let j be the height of T. Place T into the forest bucket consisting of trees of height j. If this bucket now has B trees, then execute Consolidate(j). Finally, call Pop-queue.

Consolidate(j). Create a new root node x whose children consist of the trees in the forest bucket containing the trees of height j. (There are B trees in this bucket.) Simultaneously remove from the forest the trees in this bucket. Next, perform Ripple(x) and enter the tree rooted at x into Ex-Q.

Prune(x). Remove each child of x and enter its subtree into Ex-Q. Remove x from its parent and enter x into Ex-Q (as a single node tree). If the parent y of x is now light, then perform Prune(y).

The three primary operations that modify the heap, namely, insertion, decrease, and deletion, each call Pop-queue. While Pop-queue is being executed, the auxiliary procedures Ripple, Consolidate, and Prune are invoked as necessary to guarantee that the heap order and node degree invariants described in the above overview are maintained. (The primary operations also directly call the auxiliary procedures where necessary.) Thus the algorithms are correct. Next, we consider their complexities.

THEOREM. *The amortized operation costs for the AF-heap are constant for findmin, insertion, decrease, and* $O(\log n/\log\log n)$ *for deletion.*

Proof. We use a potential function argument. Let τ_{for} denote the number of trees belonging to the forest of the heap, let τ_Q denote the number of trees in Ex-Q, and let Γ denote the sum, over all internal nodes in the heap, of $(B-d)$, where d is the number of children of the node. We define the potential of the heap to be $3\tau_{\text{for}}+4\tau_Q+6\Gamma$. This potential function is always nonnegative, and it is zero for an empty AF-heap. The amortized cost of an operation is obtained by adding the change in potential to the other incurred costs. These other incurred costs are referred to as the *direct costs* and may themselves be amortized costs.

First, we consider the Prune(x) procedure. Excluding the recursive subcall and considering just its immediate body, we observe that an execution of Prune(x) changes Γ by $-\lceil B/2 \rceil$ and changes τ_Q by $\lfloor B/2 \rfloor$. Observing that the direct cost of this execution is B, we conclude that its amortized cost, taking into account the change in potential, is zero. It follows that the total amortized cost of Prune(x), including the recursive subcalls, is zero.

Next, we consider Ripple(x). The cost of the atomic heap operations is given by the height of x in the tree and Γ is increased by one when executing Ripple(x). The total amortized cost, therefore, is bounded by $O(\log n/\log\log n)$ (since Prune has zero cost). We consider this cost to be at most B.

Turning next to Consolidate(x), aside from the call to Ripple, its direct cost is given by B. The quantity τ_{for} changes by $-B$, and τ_Q changes by one. Γ does not change since the new root contributes zero to Γ. The amortized cost of the Ripple does not exceed B. We conclude that the total amortized cost, taking into account the change in potential, does not exceed $4 - B$. Without loss of generality, we may assume that $B \geqslant 4$ and conclude that the cost of Consolidate is bounded by zero.

Now consider Pop-queue. The cost of the Consolidate call is zero. Hence, apart from the recursive subcall the direct cost is one. If Ex-Q is non-empty, then τ_Q decreases by one, τ_{for} increases by one, and except for the recursive subcall, the amortized cost is zero. If Ex-Q is empty, then the amortized cost is one since there is no change in potential. We conclude that the total amortized cost, including recursive subcalls, is one.

Having analyzed the auxiliary operations, Ripple, Pop-queue, Consolidate, and Prune, the claims of the theorem follow immediately. ∎

The AF-heap does not accommodate meld operations [6] in constant amortized time, a capability that Fibonacci heaps enjoy. Whether this is possible without degrading the efficiencies of the other operations is an open question.

2.3. *Atomic Heaps and Q-Heaps*

We proceed to describe the atomic heap and the Q-heap. First we assume the existence of the Q-heap as described at the beginning of Section 2.

The atomic heap is built in the same manner that AF-heaps are built, except that the role played by atomic heaps in the AF-heap data structure is now played by Q-heaps. We use tree structures of height at most 11 and with branching factors (of internal nodes) varying between $B/2$ and B, where $B = (\log n)^{1/5}$. The amortized cost of the ripple operation is now constant, and the (at most) $12 \cdot (\log n)^{1/5}$ roots belonging to the trees of the forest are stored in a single Q-heap whose capacity, $(\log n)^{1/4}$, is sufficient for this purpose. (This presumes that $n > 2^{12^{20}}$. An alternative but less readable method circumvents this requirement. However, as already noted we are foregoing any pretense of practicality.) All operations have constant amortized cost, and the maximum allowed height of our trees is sufficient for the storage of up to $\log^2 n$ items.

Next, we turn to the Q-heap, the subject matter for the remainder of this paper.

The Q-heap is actually a data structure for searching, accommodating insertion, deletion, and search operations in constant worst-case time. (We define search operations so that an unsuccessful search returns the successor of the search key.) We note that the findmin operation is a special case of searching, so that the structure can serve as a heap. The Q-heap is based on fusion tree techniques [7] but requires non-trivial modifications.

Like the fusion tree, the Q-heap embellishes the priority queue data structure in Ajtai *et al.* [2], which functions in the cell probe model of computation. We begin by summarizing the results from [2, 7] that we will be using. We let L denote the limitation on the allowed size of the Q-heap. The quantity L grows as a function of n and will be determined later.

Given a set $S = \{u_1, u_2, ..., u_k\}$ of b-bit numbers with $u_1 < u_2 < \cdots < u_k$, we define the set $B(S)$ of distinguishing bit positions as follows. Consecutively number the b bit positions so that position zero corresponds to the least significant bit position. Then for $1 \leqslant i \leqslant k-1$, let $c_i = \text{msb}(u_i, u_{i+1})$, where $\text{msb}(x, y)$ denotes the most significant bit position in which the two b-bit numbers x and y differ. Then $B(S)$ denotes the set $\{c_1, ..., c_{k-1}\}$, and τ_S denotes the sequence $c_1, ..., c_{k-1}$. (We emphasize that $B(S)$ is not a multi-set and may have fewer than $k-1$ elements.) Next, we define a binary tree $\text{Tree}(\tau_S)$ as follows. Let c_j be the (unique) maximum of the terms in τ_S. Then the root of $\text{Tree}(\tau_S)$ is a node labeled with the integer c_j, the left subtree of $\text{Tree}(\tau_S)$ is recursively defined to be $\text{Tree}(c_1, c_2, ..., c_{j-1})$, and the righ subtree is recursively defined to be $\text{Tree}(c_{j+1}, ..., c_{k-1})$. We augment $\text{Tree}(\tau_S)$ with k leaves (external nodes) numbered from left to right, starting with one. Given a b-bit number u, u defines a path through Tree (τ_S) in the usual way; if u has a zero in the bit position labeling the root, then the path proceeds through the left subtree; otherwise the path proceeds through the right subtree, etc. We define $\text{Leaf}(\tau_S, u)$ to be the number of the leaf terminating the path that u defines through $\text{Tree}(\tau_S)$. Given a finite set of integers, Q, and a number x, we let $\text{rank}_Q(x)$ denote the value $|\{t \mid t \in Q,\ t \leqslant x\}|$. Computing the quantity $\text{rank}_S(x)$ is central to executing a search for the key x.

LEMMA A. *The quantity,* $\text{rank}_S(u)$, *is uniquely determined by the objects,* $\text{Tree}(\tau_S)$, $i = \text{Leaf}(\tau_S, u)$, *and* $\text{rank}_{B(S)}(\text{msb}(u, u_i))$, *along with the relative order between* u *and* u_i *(greater than, equal, or less than).*

Remark. This lemma is very similar in both content and proof to Lemma 1 from [7].

Proof. Our proof proceeds by induction on $r-m$, where $r = |B(S)|$ and $m = \text{rank}_{B(S)}(\text{msb}(u, u_i))$. First, we consider the base case in which $r-m=0$. If $u = u_i$, then we have nothing to prove. Otherwise, since $u \neq u_i$ and $m = r$, we conclude that the most significant bit position in which u and u_i differ is more significant than those bit positions in which any two elements of S differ. In other words, $\text{rank}_S(u)$ is zero or $|S|$, depending on whether $u < u_i$ or $u > u_i$.

Now suppose that $r-m>0$. As above, we can assume that $u \neq u_i$. The largest

element in $B(S)$ designates the most significant bit position c_j in which any two elements of S can differ. Let S_0 denote the subset of those elements of S which have a zero in position c_j, and let S_1 denote the subset of those elements of S having a one in this position. The elements in S_0 are less than those in S_1. Moreover, these subsets, S_0 and S_1, determine the left and right subtrees of τ_S, respectively. We say that u *goes with* S_0 if u is less than the elements of S_1, and similarly, that u *goes with* S_1 if u is greater than the elements of S_0. Because $m < r$, u and u_i agree in the most significant bit positions, at least down to position c_j. It follows that u goes with the same subset, S_0 or S_1, to which u_i belongs. The values $\text{Tree}(\tau_S)$ and i, therefore, uniquely determine which of the two subsets u goes with. The value, $\text{rank}_S(u)$, can then be deduced by the determining the rank of u within the appropriate subset, S_0 or S_1.

To determine the rank of u within the subset (say) S_0, this being the subset containing u_i, let r_1 and m_1 denote the values corresponding to r and m, but relative to the set S_0. (Both r_1 and m_1 can be determined by having available m, i, and $\text{Tree}(\tau_S)$). Moreover, since c_j does not belong to $B(S_0)$, we conclude that $r_1 - m_1 < r - m$. (Observe that $r - m$ counts the number of values in $B(S)$ strictly greater than $\text{msb}(u, u_i)$.) We now apply the induction hypothesis relative to the set S_0 to determine the required rank. This completes the proof of the lemma. ∎

Remark. Instead of using $\text{Tree}(\tau_S)$, the construction of fusion tree nodes [7] uses the notion of compressed key and gains the advantage of being able to represent larger sets S (than will be the case here) at the expense of being able to efficiently update the sets. The priority queue described in [2] directly makes use of the object, $\text{Tree}(\tau_S)$.

Our plan is to exploit Lemma A as the basis for a table look-up scheme for computing $\text{rank}_S(u)$. As described in [7], the $\text{msb}(x, y)$ function can be computed in constant time. Furthermore, if we let $d = \text{msb}(x, y)$, then this computation for $\text{msb}(x, y)$ also returns the two additional quantities, 2^d and 2^{b-d}. We maintain a b-bit quantity B_S containing the (small) binary representations of the numbers of $B(S)$, packed into uniformly spaced fields each containing $\log b + 4$ bits. Using B_S, the computation of $\text{rank}_{B(S)}(\text{msb}(u, u_i))$ can be accomplished in constant time as described in [7].

Remark. The two rank computations, $\text{rank}_S(u)$, where u is an arbitrary b-bit number, and $\text{rank}_{B(S)}(a)$, where $0 \leqslant a \leqslant b$ should not be confused. The latter computation, described in [7], does not involve table look-up. The $\log b + 4$ specification for the size of the fields into which B_S is subdivided places a limitation of $b/(\log b + 4)$ on the size of $B(S)$. With the assumption that $b \geqslant \log n$, we can accommodate a set $B(S)$ of size up to $\theta(\log n / \log\log n)$ as concerns this aspect of the computation. Thus, one constraint on $L = |S|$ is that $L = O(\log n / \log\log n)$.

The two remaining hurdles to computing $\text{rank}_S(u)$ are maintaining a representation of $\text{Tree}(\tau_S)$ (as we perform insertions and deletions) and being able to compute the function $\text{Leaf}(\tau_S, u)$. A plausible approach involves the construction, during a

preprocessing phase, of a finite state machine whose states correspond to the τ_S. An immediate obstacle, however, concerns the fact that the number of possibilities for the τ_S depends on the word size b since the individual terms in the τ_S designate bit positions. The space resources required to explicitly represent this many states are not available within the framework of strongly trans-dichotomous algorithms. Our solution is to combine states into equivalence classes to achieve size reduction. Given $B(S) = \{c_1, ..., c_{k-1}\}$ and $\tau_S = c_1, ..., c_{k-1}$, we define the canonical representative for τ_S to be $\sigma_S = d_1, ..., d_{k-1}$, where $d_i = \text{rank}_{B(S)}(c_i)$. The number of equivalence classes is strictly a function of $|S|$. Moreover, Lemma A can be adapted as follows.

LEMMA A′. *The quantity,* $\text{rank}_S(u)$, *is uniquely determined by the objects,* $\text{Tree}(\sigma_S)$, $i = \text{Leaf}(\tau_S, u)$, *and* $\text{rank}_{B(S)}(\text{msb}(u, u_i))$, *along with the relative order between* u *and* u_i (*greater than, equals, or less than*).

Proof. Observe that only the relative ordering of the various bit positions plays a role in the proof of the Lemma A. Thus, the same proof establishes the current lemma. ▮

The objects described in the statement of Lemma A′ jointly range over a space of possible values having size $|S|^{O(|S|)}$. In particular, the size of this space is independent of the word size b. With appropriate restrictions on the size of S, these objects, jointly taken together, constitute a plausible index for our table look-up scheme to compute $\text{rank}_S(u)$. However, there are a multitude of technical considerations to overcome. One consideration concerns the computation of the quantity $\text{Leaf}(\tau_S, u)$, which is required to take place in constant time. Once again, the argument τ_S is not explicitly provided.

Clarification. We will make liberal use of the following convention and principle, stated in the interest of avoiding potential confusion. When we say that a particular combinatorial object constitutes an index to a table, we have in mind any natural and reasonably efficient binary encoding of the object. Indexes are restricted to have at most $\log n$ bits, restricting the range of values of the objects involved. Now if we can show that the value of a given function can be inferred in terms of certain combinatorial objects, then we potentially have the basis for a table look-up scheme to compute the function. Lemma A′ provides an example as we now explain. Assume we are given the value $i = \text{Leaf}(\tau_S, u)$, the outcome α of the comparison between u and u_i, the value $d = \text{rank}_{B(S)}(\text{msb}(u, u_i))$, and the object σ_S. Provided that our limitation L on $|S|$ is sufficiently small, $L < \frac{1}{10} \log n / \log \log n$ (say), these objects can jointly be encoded as an index with at most $\log n$ bits to access a table that returns the value $\text{rank}_S(u)$. (We require that such tables be built in linear time and space.) Assuming that the table has been built and that the index described above is provided, the rank value can be obtained in constant time. Now getting to the main point: In the sequel we occasionally need to argue that the value of a certain function can be inferred from the values of certain objects,

thereby enabling the construction of a table look-up scheme. Such arguments may proceed by describing an algorithm that actually computes the function in terms of the objects provided. However, once the look-up table has been constructed, *making use of this table only takes constant time even though the algorithm which justifies its existence could not be executed in constant time.*

We first take up the issue of computing Leaf(τ_S, u). Again, we intend to perform a table look-up, using σ_S in place of τ_S when indexing the table. This can work provided that we are able (implicitly) to map the appropriate bit values from the input u to the appropriate nodes of Tree(σ_S), thereby allowing Leaf(τ_S, u) to be inferred. We now focus upon this mapping problem. Write $B(S) = \{a_1, ..., a_r\}$ with $a_1 < \cdots < a_r$. Let $u(p)$ denote the value of the bit in position p of u. Solving the mapping problem requires being able to infer the values $b_1, ..., b_r$, where $b_j = u(a_j)$. We remark that the problem of computing compressed keys, addressed in [7], bears analogy to our mapping problem.

Our solution to the mapping problem requires that we maintain the quantity $C = \text{bin}(a_1, ..., a_r) = 2^{a_1} + \cdots + 2^{a_r}$. (*Note.* This defines the notation, bin($a_1, ..., a_r$.) In using this notation, we require that the a_i's be distinct.) The Boolean product $v = C$ AND u zeroes out the irrelevant bit positions of u. We then multiply v by a suitable value M to relocate the remaining relevant bits into a small field. A similar approach was used in [7] for the purpose of computing compressed keys. In [7] the multiplier M was chosen to satisfy the resuirement that the product $v \cdot M$ actually relocates the appropriate bits of v to distinct positions without involving any carries, the purpose being that it would then be possible to directly recover the required bit values. The construction of M was readily accomplished, but in time that was polynomial in $k = |S|$. In the current situation, however, we have only constant time to maintain M as the set S changes as a result of insertions and deletions. In particular, we are constrained to change only a constant number of bits of M during the execution of an update. The requirement imposed on M in [7], described above, must be relaxed given these circumstances we face. M will now be maintained to satisfy the less stringent requirement that the appropriate mapping of bit values from u to the terms of σ_S can be inferred by appropriately decoding a small specified field z from the product $v \cdot M$. (The size of the field z must be bounded by a suitable function of L, the limit on the permissible size of S, as we intend to incorporate z into a table index to access the value Leaf(τ_S, u).) We will show that it is possible to satisfy this less stringent requirement by changing only one bit of M during the course of an update. This can be accomplished provided that we augment the states of our finite state machine to include, in addition to σ_S, information to be used for decoding the field z as well as some information to guide the process of changing M during updates. The augmented states will no longer depend only on S, but they will also depend in part on the sequence of update operations leading to S.

A *quick summary* of where we stand:

1. We have mentioned variables C (providing a mask for extracting the distinguishing bits whose positions are specified by $B(S)$), M (the multiplier),

and B_S (containing the binary representations of the numbers in $B(S)$, packed into equally spaced fields).

2. In addition to the state field σ_S, a state field referred to as *decoder* will also contribute to the definition of the states of our finite state machine. To compute Leaf(τ_S, u), we first extract a particular f-bit field z from the quantity (C AND u) $\cdot M$. Next, we perform the table access Leaf-table(σ_S, *decoder*, z) to obtain the desired result. In essence, *decoder* serves to provide decoding information to extract from z the required information to navigate a path through Tree(σ_S).

3. The above entities need to be maintained as updates take place.

We will need the following fact, the proof of which is immediate and left to the reader.

Fact. Suppose S consists of $u_1 < \cdots < u_k$ and S' is obtained from S by deleting u_j. Then $\tau_{S'}$ is obtained from τ_S by deleting the smaller of the terms msb(u_{j-1}, u_j) and msb(u_j, u_{j+1}).

This implies that a deletion operation either results in no change to $B(S)$ or results in one element being deleted from $B(S)$. Likewise, an insertion operation either results in no change to $B(S)$ or results in one new element being inserted into $B(S)$.

We now proceed with a detailed discussion of the multiplier M. The quantity M is defined implicitly by the manner in which it is maintained. M is maintained to satisfy a number of conditions that will be revealed as the discussion proceeds. The number of ones among the bits of M is exactly $r = |B(S)|$. Thus, if our Q-heap contains fewer than two items, $M = 0$. Now write $M = \text{bin}(m_1, ..., m_r)$. According to the fact stated above, an insertion or deletion operation can cause $B(S)$ to change by at most a single element. When a new a_i is added to $B(S)$, a new m_i will contribute to M. When an a_i gets deleted from $B(S)$, an m_i gets deleted from M. In fact, the a_i's and m_i's are paired with one another as M and $B(S)$ undergo change; the m_i deleted from M when an a_i is deleted from $B(S)$ is the same m_i that was added when that a_i was added to $B(S)$. For the sake of notational convenience, we index the m_i's so that m_i is paired with a_i. (Note that the indexing of the m_i's need not coincide with their sorted order, although this is the case with the a_i's.)

We define an ordering on the a_i's in $B(S)$ which we refer to as *insertion ordering*. Relative to insertion ordering, those a_i's that appear earlier in $B(S)$ (as a consequence of updates being performed) are considered to precede those appearing later. The corresponding ordering on the m_i's paired with the a_i's is also referred to as insertion ordering.

The computation of Leaf(τ_S, u) proceeds by computing the quantity (u AND C) $\cdot M$ and then extracting a field z consisting of the rightmost $f = 5L^3$ bits from the significant portion of the product. From z we must be able to infer the values, $u(a_1), ..., u(a_r)$. Our state information will include the entity *decoder* which will specify the direct contribution of these r bits from u to the field z. In particular,

decoder contains an r-tuple of sets, where the ith set contains the bit positions (within z) that $u(a_i)$ contributes to, namely, $\{m_j + a_i - b \mid 1 \leqslant j \leqslant r$ and $b \leqslant m_j + a_i < b + f\}$. We let Ω denote the union of these sets. Observe that $|\Omega| \leqslant L^2$. Let $t_i = m_i + a_i - b$; decoder also contains the r-tuple $(t_1, ..., t_r)$, and the r-tuple $(m_1 \bmod f, ..., m_r \bmod f)$. Observe that $u(a_i)$ contributes to position t_i of z.

Four conditions will be maintained as a consequence of the manner in which updates are implemented. First, we introduce some notation.

Notation. For any given a' in $B(S)$, let $B(S)'$ denote the set of those a_i's that precede a' (relative to insertion ordering) and let m' denote the m_i value paired with a'. Let Ω' be defined as Ω was defined, but relative to the set $B(S)'$. Ω' is given by $\{m_j + a_i - b \mid m_j$ precedes m', a_i precedes a', and $b \leqslant m_j + a_i < b + f\}$.

Conditions (These conditions hold for each a' in $B(S)$).

1. The residues (mod f) of the various m_j's including m' are distinct.
2. For $t' = m' + a' - b$ we have that $2L \leqslant t' < f$.
3. The interval $[t' - 2L, t']$ avoids all members of Ω'.

Our conditions thus far suffice to guarantee that if a' is the last element to join $B(S)$, then the value $u(a')$ can be read directly from position t' of z; no carries or conflicts can affect this bit. More precisely, carries generated by the (at most L^2) cross terms to the right of position $t' - 2L$ (of z) cannot be propagated into position t' (of z) even with the presence of those cross terms $m' + a_j$ that (relative to z) fall into the interval $[t' - 2L, t']$ (since there are at most L cross terms $m' + a_j$, all of whose values are distinct). Furthermore, it is unnecessary to know in advance that t' is associated with the last a' to join $B(S)$ in order to recover $u(a')$. By examining *decoder* we can identify any position t_j of z which is free of interference from other cross terms, and the rank within $B(S)$ of the corresponding a_j is revealed by the position j of t_j within the r-tuple $(t_1, ..., t_r)$. One more condition is imposed on m'.

4. For no i and h, with a_i and a_h in $B(S)'$, does $m' + a_i - b$ fall into the protected interval $[t_h - 2L, t_h]$.

This last condition has the effect of partly extending Condition 3 "forward" in time: Upon applying Conditions 3 and 4 for each a' in $B(S)$ we conclude that

(*) for each triple (h, i, j) such that a_i precedes a_h and $j \neq h$, the sum $m_j + a_i - b$ avoids the interval $[t_h - 2L, t_h]$.

We remark that the first condition serves to guarantee that all of the m_j's will be distinct. The width $f = 5L^3$ of our field z is chosen to satisfy the requirement that all of the above conditions can be maintained. This will be demonstrated below.

LEMMA B. *Assuming that Conditions* 1 *through* 4 *are maintained as updates take place, the values* $u(a_1), ..., u(a_r)$ *are uniquely determined by* z *and* decoder.

Proof. The proof proceeds by induction on the positions of the a_i's relative to reverse insertion order. The base case of the induction is given by the paragraph following the statement of Condition 3. Now suppose that q of the $u(a_i)$'s have been deduced. Using *decoder* we can subtract off the contribution of these values to z. Let z' denote the result. Now suppose a_h is the next most recent of the a_i's (to join $B(S)$) whose value $u(a_h)$ is yet to be recovered. By virtue of (*), we can now read $u(a_h)$ from position t_h of z' (the justification being essentially that of the base case). This constitutes the induction step, completing the proof. ▮

Lemma B and the prior discussion justify the construction of our look-up table,

(1) Leaf-table(σ_S, *decoder*, z),

referred to in the *quick summary*. (For ease of reference, we are assigning reference numbers to various instances where table look-up is invoked.)

Once we have completed the computation for $i = \text{Leaf}(\tau_S, u)$, we proceed to obtain $\text{rank}_S(u)$ as follows. We assume that the values in S are stored in contiguous positions of an array A, not necessarily in sorted order. (This condition of contiguous positioning is maintained during updates as explained below.) We maintain, in addition to σ_S and *decoder*, a state field ρ, where ρ is the permutation such that $\rho(j)$ designates the position in A of the jth smallest element of S (i.e., ρ gives the permutation that sorts A). To compute rank, first access u_i, computing its location in the array A by executing the table look-up,

(2) Loc-A(ρ, i).

(Loc-A is a table for this purpose that is set up during the preprocessing phase.) Second, we compare u with u_i. Next, assuming that $u \neq u_i$, we compute $d = \text{rank}_{B(S)}(\text{msb}(u, u_i))$. Last, based on Lemma A', we obtain $\text{rank}_S(u)$ by executing the table look-up,

(3) rank(d, i, compare(u, u_i), σ_S).

(The rank table is constructed during preprocessing. The function compare(u, u_i) returns the appropriate value, $<$, $=$, or $>$.) To perform a search operation for u, we first compute $g = \text{rank}_S(u)$. Then Loc-A(ρ, g) and Loc-A(ρ, $g+1$) provide access, via A, to the gth and $(g+1)$th ranking elements of S.

Next, we discuss the processing of insertions and deletions.

Insertion

When inserting a new item u, the first step is to update σ_S in the following manner. Suppose that $j = \text{rank}_S(u)$, so that $u_{j-1} < u < u_j$. Let a' be the smaller of the two values, $\text{msb}(u, u_{j-1})$ and $\text{msb}(u, u_j)$. The quantity a' contributes a term to τ_S, being inserted into position $j-1$ or j, depending on which of the two quantities $\text{msb}(u, u_{j-1})$ and $\text{msb}(u, u_j)$ gives the value for a'. The change in τ_S induces a corresponding change in σ_S, but since the terms in σ_S reflect ranks relative to $B(S)$, it is necessary to determine whether a' already belongs to $B(S)$. Let

$d = \text{rank}_{B(S)}(a')$. If a' is not in $B(S)$ then those terms in σ_S greater than or equal to d must each be incremented. Observe that the changes required of σ_S can be determined without recourse to τ_S.

Based upon the above discussion, we update σ_S as follows. First, we execute a search for u to determine its rank j and the items u_{j-1} and u_j in S such that $u_{j-1} < u < u_j$. Second, we compute and then compare $\text{msb}(u, u_{j-1})$ and $\text{msb}(u, u_j)$. We set a' to be the smaller of these two values. Next, we determine whether a' appears in $B(S)$. (We defer for the moment how this is accomplished.) Finally, we update σ_S by executing a table look-up.

(4) The index used for this table includes $j = \text{rank}_S(u)$, the current σ_S, $d = \text{rank}_{B(S)}(a')$, compare($\text{msb}(u, u_{j-1})$, $\text{msb}(u, u_j)$), and (the Boolean quantity) Is-member(a', $B(S)$).

To determine whether a' already belongs to $B(S)$, we maintain an array V for the elements of $B(S)$ and another state field β, where β is the permutation that sorts V (analogous to ρ). We compute $d = \text{rank}_{B(S)}(a')$ and then use β and V (as we have used ρ and A) to access the element a_d of $B(S)$ of rank d. We have that a' belongs to $B(S)$ if and only if $a' = a_d$.

The next two entities that require updating when performing insertion are the array A and the permutation ρ. The array A is updated by placing u in the next empty position, and

(5) ρ is updated by executing a table look-up indexed by the current ρ and j.

We are finished with the insertion if a' already belongs to $B(S)$. If a' does not belong to $B(S)$, then we must further update the quantity C, the multiplier M, the quantity B_S, the object *decoder*, the array V, and the permutation β. Each of these updates is considered in turn.

The array V and β are updated in the same way that A and ρ are updated. As is the case with the array A, the non-empty locations of V are positioned contiguously. To update $C = \text{bin}(a_1, ..., a_r)$, we add to it the value $\text{bin}(a')$, (obtained from the same msb computation that generated a').

To update M, we must determine the bit position $m' = b - a' + t'$ of M which gets set to one; M gets updated by adding to it the value $\text{bin}(m')$. We proceed to describe the computation for t' and $\text{bin}(m')$. The value m' must comply with the four conditions preceding the statement of Lemma B. (Conversely, any m' satisfying these conditions will suffice.)

LEMMA C. *Let window(C, a') denote the bits in positions b through $b + 2f$ of $C \cdot 2^{b-a'} \cdot 2^f$. (The quantity window$(C, a')$ is obtained by extracting the rightmost $2f + 1$ bits from the significant portion of the preceding product. As noted earlier, the quantity $2^{b-a'}$ is obtained from the msb computation that produces a'.) Then the set of possible values t' such that Conditions 1 through 4 are satisfied by $m' = b - a' + t'$ is uniquely determined by the objects, $(b - a') \bmod f$, window(C, a'), and* decoder.

Proof. The set A_1 consisting of the t' compatible with Condition 1 can be deduced given $(b-a') \bmod f$ and *decoder* since the latter contains the residues of all the m_i's (mod f). Let A_2 denote the interval $[2L, f)$. The set A_3 consisting of the t' compatible with Condition 3 can be deduced given *decoder*. Now let A_4 denote the set consisting of the t_i compatible with Condition 4. We show next that $A_2 \cap A_4$ is uniquely determined. Observe that the intersection of the four sets A_i precisely defines the set of t' values referred to in the statement of the lemma.

Number the positions of window(C, a') consecutively starting with zero (to obtain the relative position numbers). The set of positions within window(C, a') occupied by ones is given by $\{a_i - a' + f \mid a_i \in B(S)\} \cap [0, 2f]$. This set enables us to deduce the set $E = \{a_i - a' \mid a_i \in B(C)\} \cap [-f, f]$. Now for any value t' in $[0, f]$, the set E uniquely determines the set $E_{t'} = \{a_i - a' + t' \mid a_i \in B(S)\} \cap [0, f]$. Substituting $t' = m' + a' - b$, this latter set can be rewritten as $E_{m'} = \{m' + a_i - b \mid a_i \in B(S)\} \cap [0, f]$. Now Condition 4 asserts that the set $E_{m'}$ avoids each of the protected intervals $[t_h - 2L, t_h]$ (since each of these intervals lies in $[0, f]$). These protected intervals are uniquely determined by *decoder*. Thus, $A_2 \cap A_4$ is uniquely determined by window(C, a') and *decoder*, completing the proof of the lemma. ∎

Lemma C justifies performing a table look-up,

(6) indexed by *decoder*, $(b-a') \bmod f$, and window(C, a'),

to obtain a suitable value for t', assuming one exists. Finally, bin(m') is given by the product $2^{b-a'} \cdot 2^{t'}$, where $2^{t'}$ is obtained by indexing a precomputed table (of size f). This completes the discussion for updating M.

LEMMA D. *The updated value for* decoder *is uniquely determined by the quantities* t', $m' \bmod f$, $d = \operatorname{rank}_{B(S)}(a')$, *the bits in positions* b *through* $b+f-1$ *of* $2^{a'} \cdot M$, *the bits in positions* b *through* $b+f-1$ *of* $C \cdot 2^{m'}$, *and the current value of* decoder.

Proof. The tuple $(t_1, t_2, \ldots)$ is easily updated given its prior value, t', and d. The tuple $(m_1 \bmod f, m_2 \bmod f, \ldots)$ is easily updated given its prior value, $m' \bmod f$, and d. Next, consider the tuple of sets. The dth tuple entry, corresponding to a', is determined by the positions occupied by ones among the bit positions b through $b+f-1$ of $2^{a'} \cdot M$. What was previously the jth tuple entry, $j \geq d$, now becomes the $(j+1)$th. For each $j \neq d$, one element may have to be added to the set representing the jth tuple entry by virtue of m', determined as follows. Consider the positions occupied by ones among the bit positions b through $b+f-1$ of $C \cdot 2^{m'}$. Label these positions consecutively, assigning the label d to the one in the (relative) location t' (of the f-bit field). A position labeled j reflects the contribution of a_j, the element in $B(S)$ of rank j, to the f-bit field as a consequence of m'. Its relative location p within the f-bit field reflects the need to include p in the set comprising the jth tuple entry. The reader can readily verify that this updating conforms to the definition of *decoder*, completing the proof. ∎

(7) Thus, we update *decoder* by executing a table look-up indexed by the objects listed in Lemma D.

To update B_S, we write a' into the first vacant field of B_S. (As with the arrays A and V, the fields of B_S are maintained so that the non-empty fields are positioned contiguously. In fact, the ordering of the a_i's in B_S coincides with their ordering in V.)

In order to facilitate the processing of deletions, there is one further step involved in the processing of insertions for the case in which C and M are modified. We maintain an array P which stores the pairs, $(\text{bin}(a_i), \text{bin}(m_i))$ in the same order that the corresponding a_i's are stored in the array V.

To justify the correctness of our proposed insertion scheme, the single outstanding issue is whether there exist choices for t' satisfying the required conditions. Specifically, we need to demonstrate that the width f of z is sufficiently large to guarantee that a t' can always be found.

LEMMA E. *Given the choice $f = 5L^3$, there exists a value t' satisfying the four given conditions.*

Proof. For each of the four conditions which must be satisfied by m', we bound the number of values for t' that are disallowed by that condition. Condition 1 disallows at most L values. Condition 2 disallows at most $2L$ values. Conditions 3 and 4 each disallow at most $(2L+1) \cdot L^2$ values. Summing these amounts, we conclude that the four conditions jointly disallow at most $4L^3 + 2L^2 + 3L$ choices for t'. Our choice for f exceeds this latter amount, demonstrating the existence of t'. ∎

Deletion

Deletions are handled in a manner similar to insertions. The one significant difference centers upon the fact that when an item is deleted, a bit (in position) a_i from the quantity C (and its associated bit m_i from the multiplier M) may need to be set to zero, based on whether $\text{rank}_{B(S)}(a_i)$ uniquely appears in σ_S (determined by using table look-up). The quantities C and M are updated by subtracting off the appropriate pair of values obtained from the array P discussed above. (The required index into P is obtained by table look-up, based on the values β and $\text{rank}_{B(S)}(a_i)$.) The fields of B_S also need to be appropriately managed. In general, whenever a field (or array) element needs to be deleted it is replaced by the last (non-empty) field (or array) element so as to maintain contiguous positioning of the elements. This last field (or array) element is removed from that last position, so that only one copy is kept. Also, the permutation for indexing these field (or array) elements gets updated using table look-up. (We are referring here to the permutations ρ and β, the arrays A, V, and P, and the quantity B_S). Still considering the case under which the quantities C and M require updating, it also is necessary to modify *decoder*. This is handled in essentially the same way as when doing insertion, except that the process is reversed. Observe that the four conditions

required of the a_j's and m_j's remain satisfied upon deletion of the appropriate a_i and m_i.

THEOREM. *The Q-heap performs insertion, deletion, and search operations in constant time and accommodates as many as* $(\log n)^{1/4}$ *items given the availability of* $O(n)$ *time and space for preprocessing and word size* $b \geqslant \log n$.

Proof. We remark first that the correctness of the algorithms was established in the above discussion. Each of the algorithms involves a constant number of table look-ups and other operations. What remains to be established is the claimed relationship between capacity and preprocessing effort. The limitation $L = (\log n)^{1/4}$ on the size of the Q-heap is determined by taking into consideration the index sizes of the various look-up tables that have to be constructed. We proceed to demonstrate that the largest table index has $O(f) = O(L^3)$ bits. For this purpose we focus upon the insertion and search operations; deletion can be similarly treated. There are seven characteristic types of table look-up that we have encountered, (1)–(7). Referring to (1), *decoder* requires $O(L^2 \log L)$ bits for its encoding, σ_S requires $O(L \log L)$ bits, and z requires $O(L^3)$ bits. The total size of the index, therefore, is $O(L^3)$. The indexes in (2)–(5) each require $O(L \log L)$ bits. The indexes in (6) and (7) each require $O(L^3)$ bits. Our imposed bound of $(\log n)^{1/4}$ on L implies that at most $O((\log n)^{3/4})$ bits are required for any index. With this limitation, the necessary tables require only $O(n^\varepsilon)$ space. There is nothing inherently difficult about the table constructions. Each table entry requires only polynomial time (in the size of its index, measured in bits) for its computation. Thus each table entry can be computed in polylog n time, and therefore, $O(n)$ time suffices for the table constructions. Our bound for L also satisfies the limitation imposed by the $\text{rank}_{B(S)}(a)$ computations since we are assuming that $b \geqslant \log n$.

Also in connection with preprocessing we mention that, as is the case with [7], there are a certain fixed number of program constants whose values are a function of the word size b. As a typical example, to assist with manipulating the fields of the quantity B_S we fill during the preprocessing phase an array of masks, one mask for each of the L fields of B_S. This requires that we have available as a program constant the value, $\text{bin}(w)$, where w is the width of a single field (and depends on the word size b). The array of masks is filled iteratively; each iteration involves a multiplication by $\text{bin}(w)$. This completes the proof of the theorem. ∎

3. CONCLUDING REMARK

As mentioned above, it remains unresolved whether there exists a linear time algorithm for the minimum spanning tree problem in the decision tree model of computation. However, if the ordering of the edges of the graph by edge weight is given in advance, then our methods allow the minimum spanning tree to be constructed in linear-time, improving upon the $O(m \cdot \alpha(m, n))$ bound [11] based upon a fast implementation of Kruskal's algorithm.

ACKNOWLEDGMENTS

We thank Bob Tarjan for stimulating discussions, and in particular for focusing our attention upon the minimum spanning tree problem. We also thank the referees for their conscientious efforts to improve our presentation.

REFERENCES

1. R. AHUJA, K. MEHLHORN, J. ORLIN, AND R. TARJAN, Faster algorithms for the shortest path problem, *J. Assoc. Comput. Mach.* **37** (1990), 213–223.
2. M. AJTAI, M. FREDMAN, AND J. KOMLOS, Hash functions for priority queues, *Inform. and Comput.* **63** (1984), 217–225.
3. M. BEN-OR, Lower bounds for algebraic computation trees, *in* "Proceedings, 15th Annual ACM Symposium on Theory of Computing, 1983," pp. 80–86.
4. B. DIXON, M. RAUCH, AND R. E. TARJAN, Verification and sensitivity analysis of minimum spanning trees in linear time, unpublished manuscript.
5. J. DRISCOLL, H. GABOW, R. SHRAIRMAN, AND R. TARJAN, An alternative to Fibonacci heaps with applications to parallel computation, *Comm. ACM* **31** (1988), 1343–1354.
6. M. FREDMAN AND R. TARJAN, Fibonacci heaps and their uses in improved network optimization algorithms, *J. Assoc. Comput. Mach.* **34** (1987), 596–615.
7. M. FREDMAN AND D. WILLARD, Surpassing the information theoretic bound with fusion trees, *J. Comput. System Sci.* **47** (1993).
8. H. GABOW, Z. GALIL, T. SPENCER, AND R. TARJAN, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, *Combinatorica* **6** (1986), 109–122.
9. J. KOMLOS, Linear verification for spanning trees, *in* "Proceedings, 25th Annual IEEE Symposium on Foundations of Computer Science, 1984," pp. 201–206.
10. G. PETERSON, "A Balanced Tree Scheme for Meldable Heaps with Updates," Tech. Rep. GIT-ICS-87-23, School of Information and Comput. Sci., Georgia Institute of Technology, Atlanta, GA 1987.
11. R. TARJAN, "Data Structures and Network Algorithms," Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
12. P. VAN EMDE BOAS, R. KAAS, AND E. ZIJLSTRA, Design and implementation of an efficient priority queue, *Math. Systems Theory* **10** (1977), 99–127.