The Cell Probe Complexity of Dynamic Data Structures

Michael L. Fredman [1]

Bellcore and
U.C. San Diego

Michael E. Saks [2]

U.C. San Diego,
Bellcore and
Rutgers University

1. Summary of Results

Dynamic data structure problems involve the representation of data in memory in such a way as to permit certain types of modifications of the data (**updates**) and certain types of questions about the data (**queries**). This paradigm encompasses many fundamental problems in computer science.

The purpose of this paper is to prove new lower and upper bounds on the time per operation to implement solutions to some familiar dynamic data structure problems including list representation, subset ranking, partial sums, and the set union problem . The main features of our lower bounds are:

(1) They hold in the *cell probe* model of computation (A. Yao [18]) in which the time complexity of a sequential computation is defined to be the number of words of memory that are accessed. (The number of bits b in a single word of memory is a parameter of the model). All other computations are free. This model is at least as powerful as a random access machine and allows for unusual representation of data, indirect addressing etc. This contrasts with most previous lower bounds which are proved in models (e.g., algebraic, comparison, pointer manipulation) which require restrictions on the way data is represented and manipulated.

(2) The lower bound method presented here can be used to derive amortized complexities, worst case per operation complexities, and randomized complexities.

(3) The results occasionally provide (nearly tight) tradeoffs between the number R of words of memory that are read per operation, the number W of memory words rewritten per operation and the size b of each word. For the problems considered here there is a parameter n that represents the size of the data set being manipulated and for these problems $b = \log n$ is a natural register size to consider. By letting b vary, our results illustrate the effect of register size on time complexity. For instance, one consequence of the results is that for some of the problems considered here, increasing the register size from $\log n$ to polylog(n) only reduces the time complexity by a constant factor. On the other hand, decreasing the register size from $\log n$ to 1 increases time complexity by a $\log n$ factor for one of the problems we consider and only a $\log\log n$ factor for some other problems.

The first two specific data structure problems for which we obtain bounds are:

List Representation. This problem concerns the represention of an ordered list of at most n (not necessarily distinct) elements from the universe $U = \{1, 2, ..., n\}$. The operations to be supported are **report**(k), which returns the k^{th} element of the list, **insert**(k, u) which inserts element u into the list between the elements in positions $k-1$ and k, **delete**(k), which deletes the k^{th} item.

Subset Rank. This problem concerns the representation of a subset S of $U = \{1, 2, ..., n\}$. The operations that must be supported are the updates "**insert** item j into the set" and "**delete** item j from the set" and the queries **rank**(j), which returns the number of elements in S that are less than or equal to j.

The natural word size for these problems is $b = \log n$, which allows an item of U or an index into the list to be stored in one register. One simple solution to the list representation problem is to maintain a vector v, whose k^{th} entry contains the k^{th} item of the list. The **report** operation can be done in constant time, but the **insert** and **delete** operations may take time linear in the length of the list. Alternatively, one could store the items of the list with each element having a pointer to its predecessor and successor in the list. This allows for constant time updates (given a pointer to the appropriate location), but requires linear cost for queries.

This problem can be solved much more efficiently by use of balanced trees (such as AVL trees). When $b = \log n$, the worst case cost per operation using AVL trees is $O(\log n)$. If instead $b = 1$, so that each bit access costs 1, then the AVL tree solution requires $O(\log^2 n)$ per operation.

It is not hard to find similar upper bounds for the subset rank problem (the algorithms for this problem are actually simpler than AVL trees).

The question is: are these upper bounds best possible? Our results show that the upper bounds for the case of $\log n$ bit registers are within a $\log\log n$ factor of optimal. On the other hand, somewhat surprisingly, for the case of single bit registers there are implementations for both of these problems that run in time significantly faster than $O(\log^2 n)$ per operation.

Let CPROBE(b) denote the cell probe computational model with register size b.

1 Supported in part by NSF grant DCR8504245

2 Supported in part by NSF grant DMS87-03541 and Air Force Office of Scientific Research grant AFOSR-0271.

Theorem 1. If $b \leq (\log n)^t$ for some t, then any CPROBE(b) implementation of either list representation or the subset rank requires $\Omega(\log n/\log\log n)$ amortized time per operation.

Theorem 2. Subset rank and list representation have CPROBE(1) implementations with respective complexities $O((\log n)(\log\log n))$ and $O((\log n)(\log\log n)^2)$ per operation.

Paul Dietz (personal communication) has found an implementation of list representation with $\log n$ bit registers that requires only $O(\log n/\log\log n)$ time per operation, and thus the result of theorem 1 is best possible.

The lower bounds of theorem 1 are derived from lower bounds for a third problem:

Partial sums mod k. An array $A[1],\ldots,A[N]$ of integers mod k is to be represented. Updates are **add**(i, δ) which implements $A[i] \leftarrow A[i] + \delta$; and queries are **sum**(j) which returns $\sum_{i \leq j} A[i] \pmod k$.

This problem is denoted PS(n, k). Our main lower bound theorems provide tradeoffs between the number of register rewrites and register reads as a function of n, k and b. Two corollaries of these results are:

Theorem 3. Any CPROBE(b) implementation of **PS**(n, 2) (partial sums mod 2) requires $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation, and for $b \geq \log n$, there is an implementation that achieves this. In particular, if $b = \Theta((\log n)^c)$ for some constant c, then the optimal time complexity of **PS**(n, 2) is $\theta(\log n/\log\log n)$.

Theorem 4. Any CPROBE(1) implementation of **PS**(n, n) with single bit registers requires $\Omega((\log n/\log\log n)^2)$ amortized time per operation, and there is an implementation that achieves $O(\log^2 n)$ time per operation.

It can be shown that a lower bound on PS(n, 2) is also a lower bound for both list representation and subset rank (the details, which are not difficult, are omitted from this report), and thus theorem 1 follows from theorem 3. The results of theorem 4 make an interesting contrast with those of theorem 2. For the three problems, list representation, subset rank and **PS**(n, k), there are standard algorithms that can be implemented on a CPROBE($\log n$) that use time $O(\log n)$ per operation, and their implementations on CPROBE(1) require $O(\log^2 n)$ time. Theorem 4 says that for the problem **PS**(n, n) this algorithm is essentially best possible, while theorem 2 says that for list representation and rank, the algorithm can be significantly improved. In fact, the rank problem an be viewed as a special case of **PS**(n, n) where the variables take on values only {0, 1}, and apparently this specialization is enough to reduce the complexity on a CPROBE(1) by a factor of $\log n/\log\log n$, even though on a CPROBE($\log n$) the complexities of the two problems differ by no more than a $\log\log n$ factor.

The third problem we consider is the set union problem. This problem concerns the design of a data structure for the on-line manipulation of sets in the following setting. Initially, there are n singleton sets $\{1\},\{2\},\ldots,\{n\}$ with i chosen as the name of the set $\{i\}$. Our data structure is required to implement two operations, **Find**(j), and **Union**(A,B,C). The operation **Find**(j) returns the name of the set containing j. The operation **Union**(A,B,C) combines the sets with names A and B into a new set named C, destroying the sets A and B. The names of the existing sets at any moment must be unique and chosen to be integers in the range from 1 to $2n$. The sets existing at any time are disjoint and define a partition of the elements into equivalence classes.

A well known data structure for the set union problem represents the sets as trees and stores the name of a set in the root of its corresponding tree. A Union operation is performed by attaching the root of the smaller set as a child of the root of the larger set (weight rule). A Find operation is implemented by following the path from the appropriate node to the root of the tree containing it, and then redirecting to the root the parent pointers of the nodes encountered along this path (path compression). From now on we consider sequences of Union and Find operations consisting of $n-1$ Union operations and m Find operations, with $m \geq n$. Tarjan [14] demonstrated that the above algorithm requires time $\theta(m\,\alpha(m,n))$, where $\alpha(m,n)$ is an inverse to Ackermann's function, to execute $n-1$ Union and m Find operations. In particular, if $m = \theta(n)$, then the running time is almost, but not quite, linear. Tarjan conjectured [14] that no linear time algorithm exists for the set union problem, and provided significant evidence in favor of this conjecture (which we discuss in the following section). We affirm Tarjan's conjecture in the CPROBE($\log n$) model.

Theorem 5. Any CPROBE($\log n$) implementation of the set union problem requires $\Omega(m\,\alpha(m,n))$ time to execute m Find's and $n-1$ Union's, beginning with n singleton sets.

N. Blum [2] has given a $\log n/\log\log n$ algorithm (worst case time per operation) for the set union problem. This algorithm is also optimal in the CPROBE(polylog$n$) model.

Theorem 6. If $b \leq (\log n)^t$ for some t, then any implementation of the set union problem with CPROBE(b), starting with n singleton sets, requires worst case time $\Omega(\log n/\log\log n)$ per operation.

The following Section provides further discussion of these results, Section 3 outlines our lower bound method, and Section 4 contains some proofs.

2. Discussion

This Section is devoted to a general discussion of our results and their relationship to previous results. The upper bounds of Theorem 2 will not be discussed further in this report. We mention, however, that the algorithms are "practical" in that they do not exploit the power of the cell probe model in any unnatural way. For the sake of this discussion, we introduce two additional problems for which our lower bounds do not apply.

Dynamic Predecessor. The data is a subset of $S = \{1, 2, \ldots, n\}$. The operations that must be supported are the updates "**insert** item j into the set" and "**delete** item j from the set" and the queries **predecessor**(j), which returns the largest element of S that is less than j.

Dynamic Dictionary. The data is a subset of $S = \{1, 2, \ldots, n\}$. Each element of S must be assigned its own location in memory and the queries are of the form "is element x in S and if so, to what location is it assigned?" The updates are "**insert** item j into the set" and "**delete** item j from the set".

Both of these problems have very fast RAM implementations. Van Emde Boas, et al. described an implementation of the dynamic predecessor problem that uses only $O(\log\log n)$ time per operation. Recently, Dietzfelbinger, et al. found a randomized hashing scheme that implements the dynamic dictionary with constant expected time per operation. On the other hand, they also show that for a class of generalized hashing algorithms there is a $\Omega(\log n)$ lower bound on the worst case time per operation.

All of the above dynamic data structure problems have a static version, in which there are no updates to the data and the problem is to find the representation of the given data in memory that minimizes the time required to answer queries. If the only goal is

to minimize the cost per query then the solution is usually trivial: precompute the answers to all of the queries, and store the answer to each query in a memory location indexed by that query. Of course, such a solution is usually undesirable for two reasons, it requires a large amount of preprocessing and a large amount of memory. This suggests that for a given static data structure problem it is important to understand the tradeoffs between query time, preprocessing time and space required to implement a solution. Tradeoffs between preprocessing time and query time for the dictionary problem were analyzed in worst case by Borodin, et al. [3] and in average case by Mairson [10], but their results assume that the algorithms used are "comparison based". Time space trade-offs for the static predecessor problem were obtained in the cell probe model by Ajtai [1].

For dynamic data structure problems, the problem of minimizing time per operation often becomes nontrivial, even without considering the cost of preprocessing or space. If we try to minimize the costs of queries as in the static case by maintaining the answer to each query in memory, each update may change the answer to many queries and thus require many memory locations to be rewritten. On the other hand, the cost of updates can be minimized by simply recording each update in a list without doing any processing. Answering a query is typically very expensive, this entire history must be read. These two trivial solutions suggest that the intrinsic difficulty of dynamic problems may lie in the tradeoff between the number of memory locations that must be rewritten to record updates and the number of memory locations that must be read to perform queries. It is this tradeoff that we investigate in this paper.

This tradeoff is not present for all dynamic problems. If, for instance, each update affects the answers to a bounded number of queries, then the data structure that stores the answers to each query can be updated quickly. This is the case, for instance, with the dynamic dictionary problem. Our lower bound techniques apply in situations where "typical" updates may effect the answer to many queries.

The results of Dietzfelbinger et al. mentioned above leave open the following important question: what is the worst complexity of the dynamic dictionary problem on a RAM whose space is bounded by, say, a polynomial in the size of the subset S?

Our results point out an interesting contrast between the static and dynamic cases of the predecessor and subset rank problems. For both of these problems a subset S from a universe U ($|U| = n$). is maintained. In the static case, various researchers have considered the problem of implementing these problems using space that is linear in the size of S. The best known upper bound for the static predecessor problem was obtained by Willard [17] who gave a linear space implementation requiring $O(\log\log n)$ time per query. Ajtai's lower bound shows that this is best possible for a CPROBE($\log n$) implementation, even if the space used is polynomial in $|S|$. In fact, the same results hold for the static version of the rank problem because the static versions of these problems are computationally equivalent. For example a data structure for predecessor queries can be used to compute rank queries by precomputing the ranks of each element in S and storing them in a linear space perfect hash table [6]; to compute the rank of an element it suffices to determine its predecessor and look up the rank of its predecessor in the hash table.

In the dynamic case (with no space bound), however, the rank and predecessor problems have very different complexities. The lower bound of theorem 1 shows that, even relative to amortized complexity, the dynamic subset rank problem requires $\Omega(\log n/\log\log n)$ time per operation, yet the algorithm in [16] solves the predecessor problem in time $\log\log n$ per operation.

We return now to the partial sum problem. This problem was previously considered by Fredman [5] and Yao [19] in two different algebraic models of computation. Fredman's results assume that the values stored in the array A belong to an abstract commutative group S, and that a memory register can store a single value from S. Complexity is measured only in terms of the number of addition/subtraction operations performed, and no charge is assessed for computing memory addresses etc. With the imposition of an additional ***obliviousness*** constraint, algorithms having complexity $\log_{5.83} n$ per operation exist, and this complexity is optimal. Roughly speaking, the obliviousness constraint requires that the registers accessed by each operation depend only on the operation instance, and not on the prior history of operations that have been performed. Yao establishes a lower bound of $\Omega(\log n/\log\log n)$ per operation (amortized worst case) without the obliviousness assumption, but his result requires that S be a commutative semigroup, and that the subtraction operation not be available. Yao [19] raised the question of whether the same lower bound holds in Fredman's group model sans obliviousness. In one sense, this model is weaker than the cell probe model because of the restrictions in the way information can be represented and manipulated. In another sense, the model is more powerful since there is no charge for memory address computations. The techniques developed in this paper can be adapted to answer Yao's question affirmatively.

One other intriguing observation can be made. The oblivious $\log n$ upper bound for PS(n, 2), which can be implemented in CPROBE(1), and our lower bound in Theorem 3, together, show that the parallelism provided by $\log n$ bit registers does not significantly reduce time complexity. However, in a machine model which permits $\log n$ bit words that overlap (as opposed to words which are comprised of disjoint sets of bits), the oblivious algorithm can be implemented in constant time. This provides a separation in power between a RAM with disjoint bytes and a RAM with byte overlap (RAMBO [12]).

Next, we turn to the set union problem. Tarjan showed that any algorithm belonging to a class of pointer machine algorithms, which he referred to as separable algorithms [15], requires $\Omega(m\,\alpha(m,n))$ time [13]. Pointer machine algorithms encompass all data structures that represent the relevant objects as nodes of a directed graph (allowing at most one item or set name to be stored in any given node) with edges serving as pointers. The separability condition imposes the further requirement that no edge may join two nodes associated with different sets, so that distinct sets correspond to separate connected components of the graph.

Because the class of separable algorithms do not utilize the full power of random access memory, Tarjan's lower bound argument is not formally conclusive in the context of random access machines. Indeed, Tarjan has stated [15] that the existence of a linear time algorithm remains an open question. There are further reasons to speculate about the existence of a linear time algorithm, even within the confines of pointer machine algorithms (without the separability constraint). First, Mehlhorn, Naher and Alt [11] show that for a related data structure problem (union-split-find) there exists a pointer machine algorithm with an $O(m\log\log n)$ running time, and yet for which any separable algorithm requires $\Omega(m\log n)$ time. Secondly, regarding the lowest common ancestor problem for a static tree with n nodes, Harel and Tarjan [8] show that pointer machines require $\Omega(m\log\log n)$ time to process m queries. But in addition, they describe a random access machine algorithm that, together with preprocessing, can accomplish the same in linear time. Thirdly,

and most tantalizing, Gabow and Tarjan [7] give a random access machine algorithm which runs in linear time and solves all instances of the set union problem for which knowledge of the Union operations (but not of the occurrences of the Find operations) is provided in advance. Interestingly, Tarjan's lower bound analysis for separable algorithms applies even to this restricted type of set union problem. Thus, separable algorithms can be superceded for a restricted set union problem, suggesting the possibility that this may be the case for the general set union problem as well. However, Theorem 5 refutes this possibility. (All of the above mentioned random access machine algorithms fall within the CPROBE($\log n$) model.)

N. Blum [2] demonstrated that his $\log n/\log\log n$ upper bound on the worst case time per operation for the set union problem is likewise optimal for the class of separable algorithms. While, as Theorem 6 shows, Blum's result is optimal in the CPROBE(polylog$n$) model, we do not know whether the $\log n$ register size in Theorem 5 can be extended to polylog$n$. However, the class of "hard" union-find sequences considered in the proof of Theorem 5 can be executed in linear time with a CPROBE($\log^2 n$) algorithm.

All of our lower bound results also hold for the class of randomized algorithms of Dietzfelbinger etal. To summarize briefly, for our purposes a randomized algorithm can be regarded as a probability distribution defined on a set of deterministic cell probe algorithms; an adversary only gets to know the distribution and not the actual choice of algorithm in any instance. This notion has been shown to be useful for the dynamic dictionary problem, but we are claiming that it cannot be used to advantage for the problems considered here. The lower bound proofs given below require only small modification to establish this strengthening.

3. Framework of Our Lower Bound Method

A central notion used in our lower bound proofs is the labeling of memory registers with chronograms (time stamps). (This labeling is a tool for analysis, and does not actually take place by the class of algorithms under consideration.) During the execution of a hypothetical sequence of operations, each time a register r is modified by a write instruction, we relabel r with the chronogram d, where d is the number of update operations that have been requested prior to the occurrence of this write instruction. In particular, a register which gets changed during the execution of the d^{th} update operation will be relabeled with d at that time. All registers have the chronogram O initially. We use these chronograms to bound from below the expected number of registers which must be read during the execution of a query operation. This expectation is defined relative to a certain distribution defined on the possible values of the input parameter provided to the query.

More precisely, starting at the point just before a given query operation and proceeding backwards in time, we partition the axis of time into intervals of increasing lengths referred to as epochs. Typically, a query will need to access from each epoch a certain expected number of registers with chronograms falling in that epoch. That this should be the case gets argued as follows. Focusing our attention on a particular epoch E, the update events taking place following the beginning of this epoch will be too rich to be suitably reflected by the effects of the (usually) comparatively small number of write instructions that take place during the subsequent epochs. Of course, the write instructions taking place during the prior epochs cannot reflect these events. Thus, in order to properly respond to our query, a certain number of registers with chronograms belonging to the E^{th} epoch require access. Finally, the expected cost of a query must exceed the total number of epochs multiplied by the expected required number of register accesses from each epoch.

4. Some Proofs

We prove first the following theorem which gives, in essence, the lower bound portion of Theorem 3. This will be followed by a proof of Theorem 5.

Theorem 3′. Any CPROBE(b) implementation of PS(n, 2) (partial sums mod 2) requires $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation. More precisely, given an algorithm $\hat{A}$ and $m \geq 2\sqrt{n}$, there exists a sequence of m operations for which $\hat{A}$ requires total time $\Omega(m \log n/(\log\log n + \log b))$.

The proof uses the following lemmas.

Lemma 1. Let C be a sequence of m numbers $a_1, \ldots, a_m$, and let $A = \Sigma a_i$. For each t, at least half of the $m - t + 1$ contiguous subsequences, $a_j, a_{j+1}, \ldots, a_{j+t-1}$, of C satisfy

$$\sum_{i=j}^{j+t-1} a_i \leq 2A\ t/(m - t + 1)$$

(Proof left to the reader)

Lemma 2. Let $\phi = (1 + \sqrt{5})/2$ $(\phi^{-1} = \phi - 1)$. The numbers $x_i = i\ \phi^{-1} \bmod 1, i \geq 1$, have the property that $|x_i - x_j| \geq 1/(3|i - j|), i \neq j$.

Proof. $x_i - x_j \equiv (i - j)\ \phi^{-1} \bmod 1 = x_{i-j}$. Thus, $|x_i - x_j| \geq \min(x_{i-j}, 1 - x_{i-j})$. The Lemma now follows from the 3-distances theorem of Swierczkowski (see [9], pp. 511 and Exercise 9, pp. 543).

Proof of Theorem 3′. Let $\hat{A}$ be an algorithm solving our problem in the cell probe model with b bit words. To prove the lower bound, we focus our attention on sequences of the form

$$(1) \qquad u_1 Q_1 u_2 Q_2 \ldots u_m Q_m\ ,\ m \geq 2\sqrt{n}\ ,$$

where each Q_i is a partial sum query with arbitrary input, and u_i is an update of the form $A[\lfloor i\ \phi^{-1} \rfloor \bmod n] \leftarrow (A[\lfloor i\ \phi^{-1} \rfloor \bmod n] + x_i) \bmod 2$, $x_i = 0$ or 1. (Note, if $x_i = 0$, then the update has no effect on subsequent query responses). Let W be the maximum number of write instructions performed by $\hat{A}$ while executing any sequence of the form (1), and let $w = W/m$. Similarly, let R denote the maximum number of memory probes performed by $\hat{A}$ for queries during the execution of any sequence of the form (1), and let $r = R/m$. Our goal is to establish the inequality

$$(2) \qquad r = \Omega(\log n/\log(bw\ \log n))$$

of which our Theorem is an immediate consequence.

We use probabilistic reasoning to establish (2), and accordingly, we assume a uniform distribution on the sequences of the form (1). As described in the above framework discussion, for each query operation

$Q_j, j \geq \sqrt{n}$, we define a series of epochs proceeding backwards from Q_j, so that the i most recent epochs contain $L_i = (\alpha \cdot w)^i$ update operations, where α is a parameter to be chosen later. The number q of epochs within such a series is chosen so that the total number of update operations among the q epochs does not exceed $\sqrt{n}$. Accordingly, we choose

(3) $$q = \lfloor \log \sqrt{n} / \log(\alpha w) \rfloor$$

Now for a sequence σ of the form (1), let $w(\sigma, j, i)$ denote the number of write instructions executed during the $i-1$ most recent epochs preceding $Q_j, j \geq \sqrt{n}$. Applying Lemma 1 (and using the fact the $\sqrt{n} \leq m/2$), we conclude that

(4) $$w(\sigma, j, i) \leq 4\, w\, L_{i-1} \leq 4\, L_i/\alpha$$
for at least one half of the $j's, j \geq \sqrt{n}$.

Now fix $i \leq q$ and $j \geq \sqrt{n}$. Let S denote the set of sequences σ of the form (1) such that

(5) $$w(\sigma, j, i) \leq 4\, L_i/\alpha$$

For $\sigma \in S$, let $\hat{\sigma}$ denote the prefix of σ preceding the j^{th} query Q_j. We may write $\hat{\sigma} = \tau\, \mu$, where μ denotes the operations which comprise the i most recent epochs preceding Q_j. Condition (5) implies that the execution of that portion of $\hat{\sigma}$ subsequent to the i^{th} epoch entails no more than $4\, L/\alpha_i$ write instructions. We now wish to fix τ, allowing only μ to vary. Accordingly, we define $S(i, \tau)$ to be those μ of length L_i for which $\tau\, \mu \in S$. As discussed in our framework description, we wish to show that the write instructions taking place *subsequent* to epoch i are typically not sufficiently numerous to suitably encode the update events from the beginning of epoch i. Towards this end, we define equivalence classes on the various μ of $S(i, \tau)$ to reflect the encoding induced by these write instructions.

We say that a given memory register r is q-accessible provided that there exists a memory state ω and an input x for which the execution of sum(x), from memory state ω, encounters register r among its first q probes. Let γ denote the number of q-accessible registers. Then

(6) $$\gamma \leq n\, 2^{qb}$$

This follows from the fact that the possible probe sequences for each of the sum(k) operations, $1 \leq k \leq n$, can be modeled in terms of a 2^b-ary tree; each internal node is labeled with the address of the associated register being probed, and its 2^b children correspond to the possible values of the contents of that register.

At the moment immediately preceding the execution of Q_j each register has a chronogram as described in our framework discussion. A register whose chronogram lies in epoch i is referred to as an epoch i register. With i fixed as above, we refer to the registers with chronograms subsequent to epoch i as *recent* registers. It is important to realize that the set of recent registers may vary with μ. We say that $\mu_1, \mu_2 \in S(i, \tau)$ are equivalent provided that for $\hat{\sigma}_1 = \tau\, \mu_1$ and $\hat{\sigma}_2 = \tau\, \mu_2$ the respective sets of recent q-accessible registers and their contents are identical. Using (5) and (6), the number $\varepsilon_{i,\tau}$ of these equivalence classes is bounded by

(7) $$\varepsilon_i = \sum_{j \leq 4L_i/\alpha} \binom{\gamma}{j} 2^{bj} \leq 2^{8L_i(\log n + qb)/\alpha}$$

Now we consider the execution of Q_j = sum(x) following the operations $\tau\, \mu$ with $\mu \in S(i, \tau)$. Without loss of generality, we may assume that whenever a query is executed, a sequence of register probes is performed to determine the appropriate answer, followed by a sequence of register writes to record new information. Now roughly speaking, if no register from epoch i is probed during this execution, then the outcome depends primarily on the equivalence class to which μ belongs. More precisely, consider a modified computation in which each register probed with an epoch i chronogram reports its contents to be the value that was stored at the conclusion of the previous epoch (namely, the value present upon completion of the operations in τ). If this computation proceeds for at most q probes, then we define $J(\tau,\mu,x)$ to be the resulting output. Otherwise, we define $J(\tau,\mu,x) = \infty$. We observe that if sum(x) $\neq J(\tau,\mu,x)$ then either the computation for sum(x) involves more than q probes, or a register with an epoch i chronogram gets probed.

Now let $J(\tau,\mu)$ denote the n dimensional vector of $J(\tau,\mu,x)$ values. We claim that if μ_1 and μ_2 belong to the same equivalence class, then $J(\tau,\mu_1) = J(\tau,\mu_2)$ since the respective computations proceed identically for the first q probes. Accordingly, with each equivalence class C_j, $1 \leq j \leq \varepsilon_{i,\tau}$, of $S(i,\tau)$ we associate an n dimensional vector $V_j = J(\tau,\mu)$ for $\mu \in C_j$.

Given $\mu \in C_j$ let $F(\tau,\mu)$ denote the vector of correct query responses (sum(1) ,..., sum(n)). The Hamming distance $\delta(\tau,\mu)$ between V_j and $F(\tau,\mu)$ represents the contribution of epoch i to the average complexity of sum(x) following $\hat{\sigma} = \tau\mu$. More precisely, consider the computation of sum(x) following σ. We claim that the number of probes performed is at least the number of epochs i for which the conditions
(**) $\sigma = \tau\mu$ with $\mu \in S(i,\tau)$ and $J(\tau,\mu,x) \neq$ sum(x)
are satisfied. First, if the computation for sum(x) proceeds for more than q probes, then the claim is immediate since there are only q epochs. So we may assume that for each epoch i satisfying (**), there is a register with an epoch i chronogram that gets probed by the sum(x) computation, thereby implying our claim. We conclude that the expected complexity E_j of Q_j satisfies (z_j denotes the number of distinct prefixes of sequences of the form (1), preceding Q_j)

(8) $$E_j n \geq \frac{1}{z_j} \sum_{i \leq q} \sum_{\tau} \sum_{\mu \in S(i,\tau)} \delta(\tau, \mu)$$

We now proceed to bound $\sum_{\mu \in S(i,\tau)} \delta(\tau, \mu)$. Intuitively, for each equivalence class C_j, we show that there cannot be very many query response vectors $F(\tau, \mu)$ that are close to V_j. By Lemma 2, the j at which updates $A[j] \leftarrow A[j] + x$ take place during the i most recent epochs are separated by gaps of at least $n/(3L_i)$.

Lemma 3. The query response vectors $F(\tau, \mu)$ with μ chosen in all possible ways consistent with (1), form an affine space spanned by L_i mutually orthogonal (orthogonal in the sense of R^n) 0, 1 basis vectors, where each basis vector (except possibly one) has Hamming weight at least $n/(3L_i)$. The number of such vectors within distance $n/30$ of any fixed vector V does not exceed $2 \cdot (2^{.95L_i})$. (Proof left to the reader.)

Now let T_i denote the set of all possible choices for μ consistent with (1), and let $\theta_i = |T_i|$. Applying Lemma 3, the number of μ satisfying $\delta(\tau, \mu) \leq n/30$ is bounded by $2\,\varepsilon_i\,\theta_i\,2^{-.05L_i}$ (ε_i bounds the number of V_j). Accordingly, we obtain

(9) $$\sum_{\mu \in S(i,\tau)} \delta(\tau, \mu) \geq \frac{n}{30} \left[|S(i,\tau)| - 2\,\varepsilon_i\,\theta_i\,2^{-.05L_i} \right]$$

Since $\sum_{\tau} \theta_i = z_j$, we conclude (upon combining (8) and (9))

(10) $$E_j \geq \frac{1}{30} \sum_{i \leq q} \left[\left[\sum_{\tau} |S(i,\tau)| / z_j \right] - 2\,\varepsilon_i\,2^{-.05L_i} \right]$$

Now let $p_j^{(i)} = \text{prob}(w(\sigma, j, i) \leq 4L_i/\alpha)$. From (4) we have that

(11) $$\sum_{j \geq \sqrt{n}} p_j^{(i)} \geq \frac{1}{2}(m + 1 - \sqrt{n})$$

Now (10) becomes $E_j \geq \frac{1}{30} \sum_{i \leq q} (p_j^{(i)} - 2\,\varepsilon_i\,2^{-.05L_i})$.

Summing over j and applying (7) and (11), we obtain

(12) $$\sum_{j \geq \sqrt{n}} E_j \geq \frac{1}{60} q(m + 1 - \sqrt{n})$$

$$- \frac{1}{15}(m + 1 - \sqrt{n}) \sum_{i \leq q} 2^{L_i(8(\log n + qb)/\alpha - .05)}$$

Referring to the right hand side of (12), we can choose α so that the second term is less than half the first term. In particular, if we choose $\alpha = 400 \cdot b \cdot \log n$ we obtain

$$r = \Omega\left(\frac{\log n}{\log(bw \log n)} \right).$$

This completes our proof.

We briefly indicate how the above argument is modified to obtain Theorem 4. This is accomplished by showing that a typical query requires access to (almost) logarithmically many bits from each epoch (as opposed to just a constant number). Observing that the space of query responses is n-ary, we show that even when the operation sequences are subdivided into equivalence classes on the basis of recent write instructions, the entropy of the query responses remains, on average, $\Omega(\log n)$. The number of bit probes (from the appropriate epoch) is then related to this entropy. To both estimate query response entropy and relate the number of bit probes to the entropy of a query response, we use the well known fact that the entropy of a joint distribution is bounded by the sum of the individual entropies.

Proof of Theorem 5. We consider the case of n Find's and $n-1$ Union's, beginning with n singleton sets. (See the remark following the Proof for a discussion of the general case with $m \geq n$ Find operations.) We concentrate our attention on Union-Find sequences that build up the sets evenly; $n/2^k$ sets of size 2^k are paired to form $n/2^{k+1}$ sets of size 2^{k+1} during what is referred to as union round k, $k \geq 0$. We use the notation $x(k,j)$, $1 \leq j \leq n/2^k$, to name the sets that exist at the beginning of union round k. The set $x(k+1,j)$ is a union of sets $x(k,p)$ and $x(k,p')$, where the p, p' choices constitute arbitrary pairings performed during round k. The Union-Find sequences we consider do not create sets of size larger than $\sqrt{n}$, so that no more than $1/2 \cdot \log_2 n$ union rounds take place. Moreover, all Find operations take place between rounds or after the last round. Our proof uses an adversary which, given a proposed algorithm A, constructs a sequence over the letters U and F consisting of at most $1/2 \cdot \log_2 n$ U's and at most n F's. Let S denote the sequence obtained. In terms of S we define the family R(S) of all Union-Find operation sequences obtained by substituting for each U in S an arbitrary round of unions (the kth U corresponds to union round $k-1$), and substituting for each F in S an arbitrary Find operation. For the sake of definiteness, assume that the Union operations taking place in round $k-1$ create sets in the order, $x(k,1), x(k,2)$, etc. It will be the case that, when averaging over the sequences in R(S), the expected execution cost required by algorithm A to perform a sequence in R(S) will be large.

Our adversary uses a parameter $q \geq 2$, which will be appropriately chosen later (and will be proportional to the average cost of a Union or Find operation in the constructed sequences). The sequence S is constructed by the adversary one letter at a time, the first letter being a U. Let S′ denote the prefix of S constructed so far by our adversary. According to the rule given below, the adversary maintains a partition of S′ into at most q contiguous subsequences referred to as epochs. The first letter of each epoch is always a U. For the initial case

when S′ = U, there is one epoch consisting of U. In general, if S′ has q epochs, then the next letter added by the adversary to S′ is an F. On the other hand, if S′ has fewer than q epochs, then the next letter is chosen to be a U. Now if the next letter chosen happens to be an F, then this letter is tentatively absorbed into the rightmost epoch. If the next letter chosen is a U, then it tentatively constitutes a new epoch consisting of one letter. We now describe a rule by which the adversary adjusts the tentative subdivision into epochs existing at this point.

Given S′, let R(S′) denote the family of all union-find operation sequences obtained from S′ by performing substitutions as described above in the definition of R(S). The tentative subdivision of S′ into epochs induces, in the obvious manner, a subdivision of the operation sequences in R(S′) into epochs. Now suppose that a given epoch e (other than the rightmost) begins with the kth U, corresponding to union round $k-1$. We say that this epoch e is compressed provided that, upon averaging over all sequences in R(S′), the average number of register writes performed by algorithm A strictly after the completion of epoch e is at least $n/(100 \cdot q \cdot 2^{k-1})$. Observe that more than one epoch may become compressed. The tentative epoch subdivision is then adjusted by extending the right boundary of the leftmost compressed epoch to the end of S′, absorbing into this epoch all of the epochs to its right. Note that this adjustment leaves no compressed epochs. If there are no compressed epochs in the tentative subdivision, then no adjustment takes place. The process of extending S′ then begins anew. This continues until S′ can no longer be extended according to our rules without violating the limitations on the numbers of U's and F's permitted ($1/2 \cdot \log_2 n$ and n respectively), yielding S.

Our use of the term "compressed" is deliberately chosen to evoke consideration of the use of path compression within Union-Find algorithms. Intuitively, until an epoch is compressed, a Find operation typically needs to follow a membership chain which touches upon sets created during that epoch. If sufficiently many register writes have taken place subsequent to that epoch, then they may encode sufficiently many membership links that transcend the sets created during that epoch, so that we may consider the epoch to be compressed.

Lemma 4. Assume that $n \geq 10^5$. For each F in the sequence S, algorithm A performs, upon averaging over the sequences in R(S), at least $q/10$ probes for the corresponding Find operation in these sequences. (Proof given below.)

Thus, if our adversary constructs a sequence S having n F's, then upon averaging over R(S) we conclude that the expected time required by A to execute these sequences is $\Omega(qn)$. Now suppose that, instead, the sequence S constructed has $1/2 \cdot \log_2 n$ U's. Then we argue that
(*) A performs on average $ng_q(n)$ register writes when executing these sequences, where $g_q(n)$ is an unbounded function of n.

Observe that during the construction of S, if the leftmost epoch (for which $k-1=0$) gets compressed m times, then A performs on average at least $mn/(100q)$ register writes when executing the sequences in R(S). This follows from the fact that each sequence within R(S′) uniformly appears as a prefix among the sequences in R(S), so that the average number of register writes taking place during a particular interval of operations, when averaging over the sequences in R(S′), is the same as when averaging over the sequences in R(S). Thus, we may sum the cost $n/(100q)$ associated with each of the m compressions. The assertion (*) is therefore a consequence of the following Lemma.

Lemma 5. There exists a function $h_q(m)$ such that if our adversary generates at least $h_q(m)$ U's, then before the next U is generated, the leftmost epoch will have been compressed in total at least m times. (Proof given below.)

Combining Lemmas 4 and 5, we conclude that if $h_q(q^2) \leq 1/2 \cdot \log_2 n$ then A performs either an average of $\Omega(qn)$ register probes or an average of $\Omega(qn)$ register writes when executing the sequences in R(S). Our lower bound for the set union problem becomes $\Omega(zn)$ where $z = \max\{q \mid h_q(q^2) \leq 1/2 \cdot \log_2 n\}$.

Comment: The above Lemmas show, in fact, that the following simpler adversary would suffice to prove our lower bound. Pick q as above. If the average complexity of the next operation, if chosen to be a Find, exceeds $q/10$, then perform a Find; otherwise, perform another round of Unions.

Proofs of the Lemmas.

First we prove Lemma 5, followed by the proof of Lemma 4.

Proof of Lemma 5: Let $h_r(m)$ be given by

$$h_r(m) = \begin{cases} 1, \text{ if } m = 0, \\ m+1, \text{ if } r = 2, \\ h_r(m-1) + h_{r-1}(2^{h_r(m-1)}), \text{ if } r > 2 \text{ and } m > 0. \end{cases}$$

Let r satisfy $2 \leq r \leq q$. Assume at a given moment that epoch $q+1-r$ (from the left) contains exactly one U and only $q+1-r$ epochs are currently present. We show that before the adversary generates $h_r(m)$ more U's, either a compression of one of the first $q-r$ epochs will take place or at least m subsequent compressions of epoch $q+1-r$ will take place. Our Lemma then follows by choosing $r = q$. We proceed by induction on m and r. When $m = 0$, the assertion is immediate. Now assume that the assertion holds when $r = 2$ and $m = k$. Unless a compression of one of the $q-2$ leftmost epochs has already occurred, when the $(k+1)$st subsequent U is generated, it

will tentatively constitute epoch q. Before the next ($(k+2)$nd) U is generated, either epoch $q-1$ or an epoch to its left will be compressed one more time. This justifies our assertion when $r = 2$ and $m = k+1$.

Now assume that the assertion holds whenever $r < r'$, and when $r = r'$ and $m = k$, where $r' > 2$. This assertion implies that unless a compression of one of the leftmost $q-r'$ epochs has already occurred, when the pth subsequent U is generated (for some $p \le h_{r'}(k)$), it will tentatively constitute epoch $q+2-r' = q+1-(r'-1)$, and epoch $q+1-r'$ will have been compressed k times and will contain p U's. Applying the case $r = r'-1$ and $m = 2^p$ of our assertion, we conclude that before another $h_{r'-1}(2^p)$ U's are generated, either (a) epoch $q+1-(r'-1)$ will have been compressed another 2^p times, or (b) an epoch to the left of epoch $q+1-(r'-1)$ will have been compressed. Considering case (b) first, if this compressed epoch is also to the left of epoch $q+1-r'$, then we conclude that before a total of $p + h_{r'-1}(2^p) \le h_{r'}(k) + h_{r'-1}(2^{h_{r'}(k)}) = h_{r'}(k+1)$ subsequent U's have been generated, an epoch to the left of epoch $q+1-r'$ will have been compressed. Otherwise, we conclude that before a total of $h_{r'}(k+1)$ subsequent U's have been generated, epoch $q+1-r'$ will have been compressed for the $(k+1)$st time. Now consider case (a). Summing the compression condition for epoch $q+1-(r'-1)$ existing prior to each of its 2^p compressions, we find that the average number of register writes taking place, after epoch $q+1-(r'-1)$ was established (when the pth U was generated), is sufficient to satisfy the condition for another ($(k+1)$st) compression of epoch $q+1-r'$ (since epoch $q+1-r'$ contains p U's). In other words, case (a) reduces to case (b). This justifies our assertion when $r = r'$ and $m = k+1$. By double induction our assertion follows generally, completing the proof.

Proof of Lemma 4: This is very similar to the proof of Theorem 3′. Again, we assign to each register a chronogram which get updated whenever a write instruction changes the contents of that register. In this instance a chronogram designates the corresponding position in S of the Union or Find operation in progress when the write instruction takes place. We demostrate below that, upon averaging over all sequences in R(S′) and all n input choices for a Find operation, the average cost required by A to execute a Find operation, after having executed the operations corresponding to S′, exceeds $q/10$ probes. From the structure of R(S), the corresponding average cost of this Find operation among the sequences in R(S) must also exceed $q/10$.

Because S′ precedes an F in S, the adversary has S′ partitioned into q epochs. As in the proof of Theorem 3′ we argue that each epoch contributes at least 1/10 to the expected cost of a Find operation at this point. Given one of these epochs, say epoch e, because epoch e is not compressed, the number of register writes taking place subsequent to epoch e is, on averaging over R(S′), less than $n/(100 \cdot q \cdot 2^{k-1})$ where 2^{k-1} is the size of the sets existing as epoch e commences. Thus, for at least one half of the sequences in R(S′), the actual number of register writes taking place subsequent to epoch e is less than $n/(50 \cdot q \cdot 2^{k-1})$. Let θ_e denote the set of these sequences, so that

$$|\theta_e| \ge \frac{1}{2}|R(S')|. \tag{13}$$

For $\sigma \in \theta_e$ we may write $\sigma = \tau\mu$, where μ denotes the operations proceeding from the beginning of epoch e. Now fix τ, allowing only μ to vary, and let $S(e,\tau)$ denote the set of μ such that $\tau\mu \in \theta_e$. We now proceed to subdivide $S(e,\tau)$ into equivalence classes, reflecting the information encoded by the write instructions taking place subsequent to epoch e.

We say that $\mu_1, \mu_2 \in S(e,\tau)$ are equivalent provided that for $\sigma_1 = \tau\mu_1$ and $\sigma_2 = \tau\mu_2$, the respective sets of q-accessible memory registers with chronograms subsequent to epoch e, and their respective contents, are identical. We have that γ, the number q-accessible registers, is bounded by

$$\gamma \le n^{q+1}. \tag{14}$$

From the definition of θ_e and using (14), we conclude that the number $\varepsilon_{e,\tau}$ of these equivalence classes is bounded by

(15)

$$\varepsilon_e = \sum_{j \le n/(50q2^{k-1})} \binom{\gamma}{j} n^j \le n^{(q+2)n/(50q2^{k-1})} \le n^{n/(25\cdot 2^{k-1})}$$

Now reasoning as in the proof of Theorem 3′, and using analogous notation, we conclude that the expected complexity E of our Find operation satisfies

$$E \cdot n \ge \frac{1}{|R(S')|} \sum_e \sum_\tau \sum_{\mu \in S(e,\tau)} \delta(\tau,\mu) \tag{16}$$

We now proceed to bound $\sum_{\mu \in S(e,\tau)} \delta(\tau,\mu)$. We substitute the following Lemma for Lemma 3.

Lemma 6. The query response vectors $F(\tau,\mu)$ with μ chosen in all possible ways consistent with the suffix of S′ beginning with the eth epoch, correspond to arbitrary groupings of the sets of size 2^{k-1} existing at the conclusion of τ into sets of the size existing at the conclusion of S′. As μ varies, these groupings occur with uniform frequency. The fraction of these groupings for which $F(\tau,\mu)$ is within distance $n/4$ of any fixed assignment vector V_j does not exceed $8^{n(e)}\hat{n}^{-n(e)/2}$ where $n(e) = n/2^{k-1}$ denotes the number of sets existing as epoch e commences, and $\hat{n}(\ge \sqrt{n})$ denotes the number of sets existing at the conclusion of S′.

Proof: Because of the convention we have chosen above for naming sets, the descriptions of the union

operations taking place in the various rounds can chosen independently of one another. Thus, we can meaningfully discuss the the effect of varying the union operations in a given round while fixing the union operations taking place in other rounds, which we now proceed to do.

Upon fixing the unions taking place in the rounds subsequent to round $k-1$, if any, we find that the unions of round $k-1$, done in all possible ways, induce an equidistribution of the possible groupings. The uniform frequency statement of Lemma 3 is an immediate consequence. The number of distinct groupings under consideration is given by

$$(17) \qquad \binom{n(e)}{n(e)/\hat{n}, \ldots, n(e)/\hat{n}} \geq \frac{\hat{n}^{n(e)}}{4^{n(e)}}$$

Now given V_j let $\hat{V}_j$ be an $n(e)$ dimensional vector obtained by assigning to each of the $n(e)$ sets existing when epoch e commences, a value obtained with greatest frequency by the set's members under the assignment V_j. Observe that if the Hamming distance between any assignment α of the $n(e)$ sets and $\hat{V}_j$ is $\hat{\delta}$, then the corresponding distance δ between V_j and and the n dimensional vector of set memberships induced by α satisfies

$$(18) \qquad \delta \geq \hat{\delta} n/(2n(e))$$

Now the number of assignments α of the $n(e)$ sets within distance $n(e)/2$ of $\hat{V}_j$ does not exceed

$$(19) \qquad \sum_{i \leq n(e)/2} \binom{n(e)}{i} \hat{n}^i \leq 2^{n(e)} \hat{n}^{n(e)/2}$$

Combining (17), (18), and (19), the fraction of the groupings of the $n(e)$ sets for which $F(\tau,\mu)$ is within distance $n/4$ of V_j does not exceed

$$8^{n(e)} \hat{n}^{-n(e)/2}$$

completing the proof of Lemma 6.

Now let Z_e denote the number of choices for μ corresponding to the appropriate suffix of S'. Applying Lemma 6, the number of μ satisfying $\delta(\tau,\mu) \leq n/4$ is bounded by $\varepsilon_e Z_e 8^{n(e)} \hat{n}^{-n(e)/2}$. (Recall ε_e bounds the number of equivalence classes.) Accordingly, we obtain from (15) and the fact that $\hat{n} \geq \sqrt{n}$

(20)

$$\sum_{\mu \in S(e,\tau)} \delta(\tau,\mu) \geq \frac{n}{4} \left[|S(e,\tau)| - Z_e n^{n(e)(\frac{1}{25}-\frac{1}{4})} 8^{n(e)} \right]$$

Since $\sum_\tau Z_e = |R(S')|$ and $\sum_\tau |S(e,\tau)| = |\theta_e| \geq |R(S')|/2$, we obtain (upon combining (16) and (20))

$$E \geq \frac{1}{4R(S')} \sum_e \sum_\tau |S(e,\tau)| - \frac{1}{4} \sum_e n^{n(e)(\frac{1}{25}-\frac{1}{4})} 8^{n(e)}$$

$$\geq \frac{q}{8} - \frac{1}{4} \sum_e n^{n(e)(\frac{1}{25}-\frac{1}{4})} 8^{n(e)} \geq \frac{q}{10}$$

for $n \geq 10^5$. This completes the proof of Lemma 4.

Remark 1: The above argument is easily modified to show that for $m \geq n$, the set union problem requires time $\Omega(z \cdot m)$ where $z = \max\{q \mid h_q(\lceil q^2 m/n \rceil) \leq 1/2 \cdot \log_2 n\}$ to execute m Find's and $n-1$ Unions. This is accomplished by limiting the number of F's permitted in S to m instead of n, and by noting that $\lceil q^2 m/n \rceil$ compressions of the leftmost epoch account for an average of $\Omega(q \cdot m)$ register writes. A routine argument shows that $z = \Omega(\alpha(m,n))$, where $\alpha(m,n)$ is the inverse to Ackermann's function defined by Tarjan [14].

References

[1] M. Ajtai, A lower bound for finding predecessors in Yao's cell probe model, Combinatorica, to appear.

[2] N. Blum, On the single operation worst-case time complexity of the disjoint set union problem, SIAM J. on Computing, 15(1986), 1021-1024.

[3] A. Borodin, L. Guibas, N. Lynch, and A. Yao, Efficient searching using partial ordering, Inf. Proc. Letters 12 (1981), 71-75.

[4] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnhert and R. Tarjan, Dynamic perfect hashing: upper and lower bounds, Proc. 29th IEEE, Symp. on Foundations of Computer Science (1988).

[5] M.L. Fredman, The complexity of maintaining an array and computing its partial sums, JACM 29 (1982), 250-260.

[6] M. Fredman, J. Komlos, and E. Szemeredi, Storing a sparse table with O(1) worst case access time, J. ACM, 31, 3(1984), 538-544.

[7] H.N. Gabow and R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union. J. Comput. Sys. Sci. 30(1985), 209-221.

[8] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(1984), 338-355.

[9] D. E. Knuth, Sorting and Searching, Addison Wesley (1973).

[10] H. Mairson, Average case lower bounds on the construction and searching of partial orders, Proc. 26th IEEE Symp. on Foundations of Computer Science (1988), 303-311.

[11] K. Mehlhorn, S. Naher and H. Alt, A lower bound on the complexity of the union-split-find problem. SIAM J. Comput. 17,6(1988), 1093-1102.

[12] S. Stallone, First blood, United Artists (1982).

[13] R.E. Tarjan, A class of algorithms which requre nonlinear time to maintain disjoint sets. J. Comput. Sys. Sci. 18(1979), 110-127.

[14] R.E. Tarjan, Efficiency of a good but not linear set union algorithm. J. ACM 22, 2(1975), 215-225.

[15] R.E. Tarjan and J. Van Leeuwen, Worst-case analysis of set union algorithms. J. ACM, 31,2(1984), 245-281.

[16] P. Van Emde Boas, R. Kaas and E. Zijlstra, Design and implementation of an efficient priority queue, Math. Systems Theory 10 (1977), 99-127.

[17] D.E. Willard, Logarithmic worst case range queries are possible in space O(n), Inf. Proc. Letters 17 (1983), 81-89.

[18] A. C. Yao, Should tables be sorted?, JACM 28 (1981), 615-628.

[19] A. C. Yao, On the complexity of maintaining partial sums, SIAM J. on Computing 14 (1985), 277-289.