Electronic Colloquium on Computational Complexity, Report No. 85 (2004)

Selection from Structured Data Sets

Erez Petrank[*] Guy N. Rothblum[†]

Abstract

A large body of work studies the complexity of selecting the j-th largest element in an arbitrary set of n elements (a.k.a. the SELECT(j) operation). In this work, we study the complexity of SELECT in data that is partially structured by an initial *preprocessing* stage and in a data structure that is *dynamically maintained.* We provide lower and upper bounds in the comparison based model. For preprocessing, we show that making at most $\alpha(n) \cdot n$ comparisons during preprocessing (before the rank j is provided) implies that SELECT(j) must make at least $(2 + \epsilon)(n/e2^{\alpha(n)})$ comparisons in the worst case, where $\epsilon > 2^{-40}$.

For *dynamically maintained* data structures, we show that if the amortized number of comparisons executed with each INSERT operation is bounded by $i(n)$, then SELECT(j) must make at least $(2 + \epsilon)(n/e2^{i(n)})$ comparisons in the worst case, no matter how costly the other data structure operations are. When only INSERT is used, we provide a lower bound on the complexity of FINDMEDIAN. This lower bound is much higher than the complexity of maintaining the minimum, thus formalizing the intuitive difference between FINDMIN and FINDMEDIAN.

Finally, we present a new explicit adversary for comparison based algorithms and use it to show adversary lower bounds for selection problems. We demonstrate the power of this adversary by improving the best known lower bound for the FINDANY operation in a data structure and by slightly improving the best adversary lower bound for sorting.

1 Introduction

The *selection problem*—finding the i-th largest element from a set of n distinct values, where $1 \leq i \leq n$, has received extensive attention over the past three decades. This is a fundamental problem in algorithms and data structures. Selection is used as a tool in the solution of several important problems—sorting, optimizations and finding convex hulls, among others. The best known instances of the selection problem are selecting the minimum, the maximum and the median from a set. In 1973 Blum et al. [4] showed that SELECT(i) (the operation that finds the i-th largest element in a set) can be executed by a comparison based algorithm making $\Theta(n)$ comparisons. The gap between the upper and lower bounds for this problem, especially for finding the median, has been the focus of several research

[*]Computer Science Department, Technion – Israel Institute of Technology, Haifa 32000, Israel. E-mail: erez@cs.technion.ac.il. This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant no. 36/03)

[†]Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. E-mail: guy.rothblum@weizmann.ac.il. Part of this work was done while the author was at the Technion – Israel Institute of Technology.

ISSN 1433-8092

papers over the past years, culminating in the results of Dor and Zwick [9, 10] which give an upper bound of $\approx 2.95n + o(n)$ and a lower bound of $(2+\epsilon)n$ for some small constant $\epsilon > 2^{-40}$. For the more general problem of selecting the i-th, or αn-th, largest element, an upper bound of $3n + o(n)$ comparisons is implied by Schönhage et al. [21]. For small values of α Dor and Zwick [8] give an improved upper bound of $\{1 + [1 + o(1)]H(\alpha)\} \cdot n$, where H is the binary entropy function. This is almost tight, as a lower bound of $[1 + H(\alpha)] \cdot n$ is given by Bent and John [3].

While the problem of finding the i-th largest element in an arbitrary set has received much attention, far less attention has been given to selection from more structured data sets—data sets that have been previously maintained or preprocessed. One natural approach is applying *preprocessing* to the data in order to reduce the subsequent work needed to find the i-th largest element. The question that naturally arises, is how much work can be saved by employing preprocessing. Note the two phases in this framework: in the preprocessing phase i is unknown—it is provided only in the subsequent selection phase. The two extremes of this approach are pre-sorting the entire set using $\Theta(n \log n)$ comparisons and then selecting the i-th largest element using 0 comparisons, and in the other extreme, conducting no preprocessing and selecting with $\Theta(n)$ comparisons. Previous works (e.g. [4, 9, 10]) are restricted to this second case. This model has been explored for the SEARCH operation. The SEARCH operation receives a key and returns its address in the data set or that the given key is not in the data set. Borodin et al. [5] and Mairson [16] show that if $\alpha(n) \cdot n$ comparisons are made during preprocessing then SEARCH requires at least $n/e2^{\alpha(n)}$ comparisons in the worst case. The model is analyzed for the average-case metric in [16] and by McDiarmid in [17]. We note that a lower bound of $f(n)$ comparisons for the SEARCH operation immediately yields a lower bound of $\frac{f(n)}{\log n}$ comparisons for the SELECT operation, since selection can be used to implement binary search for a given key with logarithmic slowdown. It seems interesting to ask whether this logarithmic denominator in the lower bound can be avoided.

Another natural setting involving structured data sets is that of dynamically maintained data structures. In this framework a data structure supports several operations, including the maintenance operations INSERT and DELETE, and some selection operation (FINDMIN for example, which finds the minimal value in the data structure). A natural question here is the tradeoff between the cost of each maintenance operation and the cost of executing the selection operation.

Brodal et al. [7] examined the complexity of maintaining element ranks in a comparison based data structure under the operations INSERT, DELETE, FINDMIN and the operation FINDANY, which is only required to return some key in the data structure and its rank. They show that if the number of keys in the data structure is n and INSERT and DELETE both have worst case amortized complexity of $t(n)$ comparisons per operation then FINDANY makes at least $n/2^{4t(n)+3} - 1$ comparisons in the worst case, and the *expected* number of comparisons made by FINDMIN is at least $n/e2^{2t(n)}$. It is natural to ask what can be said about other selection operations in a dynamically maintained data structure. Intuitively, it seems that SELECT and FINDMEDIAN are hard to maintain even under INSERT operations only (without DELETE operations), while the minimum is easy to maintain under INSERT operations—requiring only a single comparison per key inserted into the data structure. This observation seems to shed new light on the difference between the difficulty of finding the median compared with the difficulty of finding the minimum. Previous work does not show a substantial separation between the two operations.

1.1 Main Results

We begin with the use of preprocessing to reduce the running time of SELECT on a set of distinct values. As mentioned, if no preprocessing is executed, SELECT requires $\Omega(n)$ comparisons, whereas, if the set is sorted during preprocessing (using $\Theta(n \log n)$ comparisons) then the comparison complexity of SELECT is reduced to 0. We extend this observation to a tradeoff (with lower and upper bounds) between the amount of work spent during preprocessing and the amount of work required then to run the selection itself. Theorem 1 asserts that if only $\alpha(n) \cdot n$ comparisons are made during preprocessing SELECT must make at least $(2+\epsilon)(n/e2^{\alpha(n)})$ comparisons in the worst case, where $\epsilon > 2^{-40}$. The lower bound is proved using the methods of [5, 16]. Note the strength of this lower bound. It means, for example, that a linear time preprocessing algorithm can only save a constant factor of the (linear time) execution of SELECT. We also demonstrate a matching upper bound for this problem by presenting a preprocessing algorithm which is allowed to make $\alpha(n) \cdot n$ comparisons and a corresponding selection algorithm that makes at most $\lceil 6n/(2^{\alpha(n)-7}) \rceil$ comparisons (after preprocessing).

We continue by considering dynamically maintained data structures with the SELECT operation. Brodal et al [7] showed an *expected time* lower bound of $n/e2^{2i(n)}$ comparisons for FINDMIN when both update operations (INSERT and DELETE) have a worst-case amortized cost of at most $i(n)$ comparisons. Can this lower bound be asymptotically improved when considering the more general SELECT operation? We answer this question negatively by presenting (in Section 5) a data structure with amortized comparison complexities of $i(n)$ comparisons for INSERT, 12 comparisons for DELETE and $\lceil 6n/(2^{i(n)-7}) \rceil$ comparisons for SELECT. As FINDMIN is a special case of SELECT, this demonstrates, somewhat surprisingly, that FINDMIN and SELECT are asymptotically equivalent in data structures with INSERT and DELETE update operations.

Next, we present a trade-off between the comparison complexity of SELECT and INSERT only (nothing is assumed about the DELETE operation). We start with Corollary 1 (which follows from Theorem 1). This corollary asserts that if the amortized comparison complexity of the INSERT operation is at most $i(n)$ then SELECT must make at least $(2+\epsilon)(n/e2^{i(n)})$ comparisons in the worst case, where $\epsilon > 2^{-40}$. The same upper bound (of Section 5) matches this lower bound as well. Note that a lower bound based only on INSERT is significant. The complexity of INSERT is often much lower than that of DELETE. The Priority queues of [11], for example, support INSERT in time $O(1)$ but DELETE requires time $O(\log n)$. For data structures such as this our lower bound for SELECT is far higher than the best known (e.g. $\Omega(n)$ instead of $\Omega(1)$ for the given priority queues).

We have established the fact that in a comparison based dynamically maintained data structure with an INSERT operation SELECT is hard (and even for many indices). But the hard indices are determined dynamically as the data structure makes comparisons. The question is: are there predetermined provably hard indices? For some indices the answer is negative—maintaining the minimum requires only one comparison per INSERT operation. The "hardest" fixed index selection problem seems to be the problem of selecting the median.

In [4] it was shown, somewhat surprisingly, that selecting the median from an (unprocessed) input set is asymptotically equivalent to selecting the minimum. This seminal result defies the intuition that median selection is much harder than minimum selection. Intuition also suggests that maintaining the median should be much harder than maintaining the minimum, and in this case the intuition is correct. In Theorem 2 and Corollary 2 it is shown that if the amortized comparison complexity of the INSERT operation is at most $i(n)$ then FINDMEDIAN must make at least $(2+\epsilon)[n/(4e \cdot 2^{3i(n)})]$ comparisons in the worst case, where $\epsilon > 2^{-40}$. This means, for example, that if only $O(1)$ comparisons are made during insertion then FINDMEDIAN must make $\Omega(n)$ comparisons in the worst case (compared with 0 comparisons in the worst case for FINDMIN). Alternatively, Corollary 2 also implies that if FINDMEDIAN always makes $O(1)$ comparisons, then the amortized comparison complexity of INSERT must be $\Omega(\log n)$ (compared with 1 for FINDMIN). We note that previous proof techniques used for the SEARCH operation (e.g. [5]) fail when directly applied to this problem.

Corollary 2 follows from the more general assertion of Theorem 2 which generalizes the result beyond the FINDMEDIAN operation to the operation FIND-αnLARGEST. This operation returns the αn-th largest key in the data structure, where $\alpha = \frac{1}{m}$ and $m \in Z_{\geq 2}$. Theorem 2 shows that if the amortized comparison complexity of the INSERT operation is at most $i(n)$ then FIND-αnLARGEST must make at least $(2+\epsilon)\{[(\alpha - \alpha^2)n/(2^{\frac{1+\alpha}{\alpha}i(n)})]\}$ comparisons in the worst case, for the same $\epsilon > 2^{-40}$. A symmetric result can be similarly proven for the operation FIND-αnSMALLEST.

Finally, we address the question of finding explicit adversaries for these comparison based algorithms. The lower bounds described so far are all derived from an information theoretic argument. The problem of finding explicit adversaries for comparison based algorithms was addressed by Atallah and Kosaraju in [2], and adversaries were presented in [5, 7]. As mentioned in [2], adversaries are important when it is required to efficiently produce a counterexample to an impossibly efficient comparison based algorithm (say, an algorithm that sorts in $o(n \log n)$). As they note, the information theoretic lower bound assures us that there exists an input sequence for which the algorithm will make many comparisons (or err), but provides no method for efficiently finding this counterexample. Another application is using such adversaries to slow down an algorithm as much as possible. We present a new generic adversary for comparison based algorithms. Our adversary is similar to the adversary of [2], but slightly superior. We proceed to show adversarial lower bounds for several problems. For the problem of selection after preprocessing using at most $\alpha(n) \cdot n$ comparisons, the adversary forces that at least $[n/(2^{2\alpha(n)}) - 3]$ comparisons be made. We then revisit the lower bound of Brodal et al. [7] for maintaining the FINDANY operation. We simplify their proof and improve the lower bound by a multiplicative factor of 8, showing in Theorem 4 that if the amortized comparison complexities of INSERT and DELETE are at most $i(n)$ and $d(n)$ respectively then FINDANY must make at least $[n/(2^{2(i(n)+d(n))}) - 3]$ comparisons in the worst case. Note that in this case the new adversary improves the *best known* lower bound for the problem. The new adversary can also force slower sorting than the adversary of [2], though this improvement is only in the second-order term of the lower bound.

1.2 Related Work

As previously mentioned, the problem of selecting the i-th largest element from a set of n distinct values has received much attention. A survey of progress on the selection problem is given by Paterson [19] and the references therein. In the early 70's Blum et. al. [4] were the first to show that selection could be done in $O(n)$ (specifically in $5.43n$) comparisons. Their technique has been the basis of all subsequent improvements. They also gave a lower bound of $n + min\{k-1, n-k\} - 1$ for selecting the k-th largest element, in particular $1.5n$ comparisons for selecting the median.

The upper bound for SELECT and median selection was improved by Schönhage et. al. [21] who showed a selection algorithm that makes at most $3n + o(n)$ comparisons. This results was improved by Dor and Zwick [9] who presented an algorithm for median selection that makes at most $2.9423n$ comparisons. They also extended their techniques to the general selection problem, improving the existing upper bounds for selecting the αn-th largest element.

The lower bound for selecting the median was continually improved using progressively more elaborate arguments by Pratt and Yao [20], Kirkpatrick [15], Yap [22] and Munro and Poblete [18] to about $1.837n$. In 1979 Fussenegger and Gabow [12] used a new technique to prove a $1.5n + o(n)$ lower bound for finding the median. Bent and John [3] used this technique to further improve the lower bound for selecting the median to $2n + o(n)$ in 1985. In 1995 Dor and Zwick [10] were able to improve this lower bound to $(2 + \epsilon)n + o(n)$ where $\epsilon > 2^{-40}$. This is currently the highest known lower bound for median selection.

The problem of maintaining data structures with search operations has also been explored by Borodin et al. [6], where a tradeoff is shown between the comparison complexities of update and search operations in implicit dictionaries. Alt and Mehlhorn [1] studied the complexity of searching data that has been "semisorted"—data for which only part of the $n!$ order permutations are possible.

1.3 Organization

In **Section 2** we define the comparison based model and some notations used throughout the paper. In **Section 3** lower bounds are shown for the SELECT operation with preprocessing. In **Section 4** we show lower bounds for selection and for selecting the median in data structures with an INSERT operation. In **Section 5** we present an upper bound for selection in data structures. In **Section 6** we present a new explicit adversary for comparison based algorithms and adversarial lower bounds for preprocessing algorithms run before a SELECT operation. **Appendix A** is devoted to lower bounds for the FINDANY operation on dynamically maintained data structures. In **Appendix B** we present efficient algorithms for preprocessing and selection.

2 Preliminaries

Notations We will use $[a \dots b]$ where a and b are integers and $a \leq b$ to denote the set $\{a, a+1, \dots, b-1, b\}$. $[n]$ for an integer n denotes $[1 \dots n]$. For two sets A and B that are both subsets of some well ordered set U we denote by $A > B$ the fact that $\forall a \in A, \forall b \in B\ a > b$. $Z_{\geq n}$ will refer to the set of integers greater or equal than n.

The Model In the *comparison model* an algorithm may only access input keys by comparing them. A *comparison based algorithm* is an algorithm that may only access input keys by comparing them and storing them in memory. The algorithm is only charged for comparisons, all other operations are free. A *comparison based data structure* is a data structure whose operations (e.g. INSERT, DELETE etc.) are all comparison based. Many generic data structure implementations are comparison based. In this paper we deal only with comparison based algorithms and data structures. Lower bounds are stated for *dynamically maintained* data structures—data structures which receive no a-priori knowledge of the sequence of operations that will be run on them.

Lower Bounds Throughout the paper we use the best known lower bound for selection from a set of n elements (without preprocessing). Lower bounds are stated in terms of this best known lower bound for selection—the lower bound of $(2+\epsilon)n$ for selecting the median of [10]. It is important to note that if the lower bound of [10] is improved, our lower bounds will also improve. Throughout the paper ϵ will refer to the small constant $\epsilon > 2^{-40}$ of the work of [10].

3 Selection with Preprocessing

In this section we present a lower bound for the SELECT operation with preprocessing. This is a consequence of the proofs methods of [5, 16]. We begin with a brief review of the model and relevant past results, following the notation and exposition of [16].

A preprocessing algorithm, P, is run on a set of distinct and ordered keys $U = \{k_1, \ldots, k_n\}$, with an *order permutation* π, where $k_{\pi(1)} < \ldots < k_{\pi(n)}$. The preprocessing algorithm can be represented as a binary tree (the preprocessing tree), where each internal node makes a comparison between two keys in the input set. The two children of each node correspond to the two possible outcomes of each such comparison. Every leaf v of the preprocessing tree has a partial order imposed by the comparisons made to reach it. This partial order imposes a set of possible order permutations. An *independent set* of U under a partial order A (also known as an anti-chain) is a set of elements which are completely incomparable under A. $W \subseteq U$ is an independent set if any order permutation of the elements of W is possible under A. Lemma 3.1.1 of [16] states that if a partial order A of the n elements of U is consistent with p permutations then U contains an independent set of size at least $p^{1/n}$. We are now ready to prove a simple theorem regarding selection after preprocessing.

Theorem 1 *Let U be a data set containing n distinct values drawn from an ordered set. Let P be a comparison based preprocessing algorithm that receives U, and S a comparison based selection algorithm that receives P's output and an index i and outputs the i-th largest element in U. If P makes at most $\alpha(n) \cdot n$ comparisons then S must make at least $(2+\epsilon)(n/e2^{\alpha(n)})$ comparisons in the worst case.*

Proof: After P is run, making $\alpha(n) \cdot n$ comparisons, there exists some leaf v of the preprocessing tree which is reached by at least $\frac{n!}{2^{\alpha(n)n}}$ permutations of the inputs. Let A be the partial order imposed by the comparisons the preprocessing algorithm made to reach v. By Lemma 3.1.1 of [16], under A there exists a maximal independent set W of size at least

$\frac{n}{e2^{\alpha(n)}}$. By the results of [10], if a selection algorithm is run after P terminates in the leaf v, selecting the median from the independent set requires that at least $(2+\epsilon)|W|$ comparisons be made in the worst case. To show this rigorously, we assume towards a contradiction that after P runs, S can always select the median with less than $(2+\epsilon)|W|$ comparisons. We now use P and S to construct an impossibly efficient algorithm M for median selection from a set of $|W|$ input keys. The selection algorithm M will simulate P on a large set of dummy keys to arrive at the memory configuration of leaf v of the preprocessing tree. Note that W is a maximal independent set, and thus every dummy key not in the independent set is either smaller or larger than some key in W. Let l be the number of dummy keys larger than some key in W, and i the index of the median in a set of size $|W|$. M will now use P's output on the dummy keys and S to select the median of its input set. It will place the input keys in the addresses of the items of the independent set W and run S to find the $l+i$-th largest key in P's output. Any comparisons S makes between items not in the independent set will be determined by the partial order of the leaf v or arbitrarily. If a key (outside W) smaller or larger than some key in W is compared with some key in W, the result will be that the item is smaller or larger respectively than the key in W. Comparisons made between keys in W are answered according to the input set. Clearly the new algorithm makes at most as many comparisons on the input set as S (i.e. less than $(2+\epsilon)|W|$) and the i-th largest key in the input set is the $l+i$-th largest item in the input of S's simulated run. In conclusion, we get an algorithm for selecting the median from an independent set W that makes less than $(2+\epsilon)|W|$ comparisons—a contradiction. $\square$

An upper bound of $\lceil 6n/(2^{\alpha(n)-7})\rceil$ is shown in Section 5. We have shown that SELECT is “hard” for at least one (dynamically chosen) index. It is not hard to see, by the same argument used in the proof of Theorem 1, that if $\alpha(n)\cdot n$ comparisons are made during preprocessing then the selection of any of the $\frac{n}{e2^{\alpha(n)}}$ indices in the resulting independent set requires that at least $\frac{n}{e2^{\alpha(n)}}-1$ comparisons be made in the worst case. These indices, however, are different for every preprocessing algorithm. It may seem interesting to ask whether there are any fixed indices for which selection is hard after preprocessing. We note, however, that the approach of using preprocessing before a SELECT operation is only interesting when the index to be selected is not known in advance—if it is known which rank is to be selected (e.g. the minimum, the median etc.) there is no use in a separate preprocessing stage. This is part of our motivation for exploring selection operations in dynamically changing data structures.

4 Selection in (Dynamic) Data Structures

In this section we give a lower bound for the operations SELECT and SELECT-MEDIAN in a dynamic data structure. We base our lower bounds only on the time spent in the INSERT operation. As previously stated, this refinement is significant as many widely used data structure implementations have INSERT operation which are much cheaper than their DELETE operations (see [11]). In such cases, our lower bound is much stronger. The following corollary is an easy consequence of Theorem 1.

Corollary 1 *Let D be a comparison based data structure with* INSERT *and* SELECT *operations. If the amortized comparison complexity of the* INSERT *operation is $i(n)$, where n*

is the number of keys in the data structure, then the SELECT *operation must make at least* $(2+\epsilon)(n/e2^{i(n)})$ *comparisons in the worst case.*

Proof: Assume towards a contradiction that there exists a comparison based data structure with amortized INSERT complexity $i(n)$ where SELECT always makes less than $(2+\epsilon)(n/e2^{i(n)})$ comparisons. A special case of using this data structure is to partition the operations into a preprocessing and a selection algorithm. During preprocessing, the n keys will be inserted into the data structure, making at most $i(n)\cdot n$ comparisons. The data structure's SELECT operation can now be used for selection. Thus, by Theorem 1 the data structure's SELECT operation must make at least $(2+\epsilon)(n/e2^{i(n)})$ comparisons in the worst case—a contradiction. □

It is again natural to ask whether the SELECT operation is hard only for one index or for many. As in the case of selection after preprocessing, it is not hard to see that at least $n/e2^{i(n)}$ indices require that at least $n/e2^{i(n)}-1$ comparisons be made in the worst case. These "hard" indices, however, are determined on-the-fly as keys are inserted into the database and comparisons are made. We now ask whether there are fixed indices for which selection from a data structure is always hard. There exist indices for which selection may be very easy even if the INSERT operation makes very few comparisons. For example, if we are only concerned with INSERT operations, the minimum can be maintained by making only a single comparison in each insertion, keeping the minimum value in a special location. Finding the minimum then requires zero comparisons. However, we show that there are (many) predetermined indices for which SELECT is hard. The following theorem states a lower bound for the number of comparisons needed to perform the FIND-αnLARGEST operation—selecting the αn-th largest key in the data structure, where $\alpha=\frac{1}{m}$ and $m\in Z_{\geq 2}$. This lower bound shows a large gap between the complexity of maintaining the minimum and the complexity of maintaining the αn-th largest key. (A symmetric result can be similarly proven for the FIND-αnSMALLEST operation.)

Theorem 2 *Let D be a comparison based data structure with* INSERT *and* FIND-αnLARGEST *operations, where $\alpha=\frac{1}{m}$ and $m\in Z_{\geq 2}$. If the amortized complexity of the* INSERT *operation is at most $i(n)$, where n is the number of keys in the data structure, then* FIND-αnLARGEST *must make at least $(2+\epsilon)[(\alpha-\alpha^2)n/(e2^{\frac{1+\alpha}{\alpha}i(n)})]$ comparisons in the worst case.*

Proof: Assume $\alpha(1-\alpha)n$ is an integer. We insert keys from the set $\{k_1,\ldots,k_n\}$ into the data structure. The set K of $(\alpha-\alpha^2)n$ keys $\{k_{\alpha^2 n+1},\ldots,k_{\alpha n}\}$ is initially an independent set and all $((\alpha-\alpha^2)n)!$ order permutations are possible for it. The $\alpha^2 n$ keys $k_1,\ldots,k_{\alpha^2 n}$ will get "large" values. $\forall i\in[\alpha^2 n], \forall a\in K, k_i>a$ and $k_1>\ldots>k_{\alpha^2 n}$. The $(1-\alpha)n$ keys $k_{\alpha n+1},\ldots,k_n$ will get "small" values. $\forall i\in[\alpha n+1\ldots n], \forall a\in K, k_i<a$ and $k_{\alpha n+1}>\ldots>k_n$. In conclusion, all keys are fixed except for the keys $k_{\alpha^2 n+1},\ldots,k_{\alpha n}$.
Consider $(\alpha-\alpha^2)n$ operation sequences, where $\forall i\in[1\ldots(\alpha-\alpha^2)n]$:

$$S_i = \text{INSERT}(k_1)\ldots\text{INSERT}(k_{\alpha n+i\frac{1}{\alpha}})\text{FIND-}\alpha n\text{LARGEST}$$

The keys returned by all of these operation sequences are from $k_{\alpha^2 n+1},\ldots,k_{\alpha n}$. In S_1 the maximum of $k_{\alpha^2 n+1},\ldots,k_{\alpha n}$ is returned, and more generally, in S_i the i-th largest key of $k_{\alpha^2 n+1},\ldots,k_{\alpha n}$ is returned.

Before the FIND-αnLARGEST operation is run in the last operation sequence, $S_{(\alpha-\alpha^2)n}$, the total number of comparisons that have been made on the $(\alpha-\alpha^2)n$ keys $k_{\alpha^2 n+1}, \ldots, k_{\alpha n}$ is at most $(1-\alpha^2) \cdot i(n) \cdot n$ (recall that in the first $\alpha^2 n$ insertions no comparisons involving these keys are made). Thus there is an independent set $W \subseteq K$ such that $|W| \geq [(\alpha-\alpha^2)n/(e2^{\frac{1+\alpha}{\alpha}i(n)}) - 2]$. Note that $\forall i \in [1 \ldots (\alpha-\alpha^2)n]$ in operation sequence S_i, before the FIND-αnLARGEST call, W is an independent set. By the argument given in the proof of Theorem 1, selecting the median from W requires at least $(2+\epsilon)(|W|)$ comparisons, and thus, since there exists a sequence S_j in which the median is selected, in S_j the FIND-αnLARGEST operation must make at least $(2+\epsilon)(|W|)$ comparisons in the worst case. □

The lower bound of Theorem 2 is a function of α, where $\alpha = \frac{1}{m}$ and $m \in Z_{\geq 2}$. This lower bound is highest for $\alpha = \frac{1}{2}$. The operation FIND-αnLARGEST with $\alpha = \frac{1}{2}$ is simply the operation FINDMEDIAN. This is expected, as the median is considered the "hardest" index for selection. The lower bound for maintaining the median is stated in the following corollary:

Corollary 2 *Let D be a comparison based data structure with* INSERT *and* FINDMEDIAN *operations. If the amortized comparison complexity of the* INSERT *operation is $i(n)$, then* FINDMEDIAN *must make at least $(2+\epsilon)[n/(4e \cdot 2^{3i(n)})]$ comparisons in the worst case.*

5 Upper Bounds for Selection in Data Structures

In this section we show upper bounds for the problem of maintaining a data structure holding n keys with INSERT, DELETE and SELECT operations. We note that the data structure presented is efficient in terms of its *comparison complexity*, but its actual complexity is higher. Our main purpose in presenting this data structure is to show that SELECT is asymptotically equivalent to FINDMIN in data structures with INSERT and DELETE operations. This surprising result highlights the motivation behind our (higher) lower bound for SELECT and FINDMEDIAN under only INSERT operations. We also use this data structure to present upper bounds. We will use the following two well known algorithms as subroutines:

- SELECT(A, i) on an arbitrary (unordered) array A of size n returns the address of the i-th largest value in the array. By [21] SELECT can be done with $3n$ comparisons for any i.

- PARTITION(A, i, j, k) on an array $A[i \ldots j]$ of size $n = j - i + 1$ shuffles the array keys so that the lowest k values are in the leftmost k array addresses, and the highest $n-k$ values are in the rightmost $n-k$ array addresses. PARTITION can be done in $3n$ comparisons by the results of [21] (their algorithm determines for every key whether it is larger or smaller than the selected key, no further comparisons are needed for shuffling).

The amortized comparison complexity of the data structure's INSERT operation will be $i(n) + c$, where n is the number of keys in the data structure and c is a constant. We assume that $i(n)$ is a "well-behaved" function. By "well behaved" we mean that for all large enough n, n' if $n' \leq n$ then $n'/2^{i(n')} \leq n/2^{i(n)}$. Note that this implies in particular that $i(n) = O(\log n)$, which is reasonable since insertion with more than $\log n$ comparisons is not interesting. It also implies $i(n)$ is monotone. These "well-behaved" functions cover most interesting cases (functions such as $\log n$, $\sqrt{logn}$ and $\log\log n$ are all well behaved).

The data structure will hold $m \leq 2^{i(n)+1}$ lists of keys $L_1 \dots L_m$, where the size of each list L_i is at most $2n/2^{i(n)}$ and $L_1 > L_2 > \dots > L_m$. For each list L_i the data structure will maintain the size of L_i, s_i, and a "pivot" of L_i, p_i. The pivot p_i satisfies $L_i \geq \{p_i\} > L_{i+1}$. As long as all pivots satisfy this inequality it is clear that $\forall i \in [m-1], L_i > L_{i+1}$. Note that pivots are not keys in the data structure—the value of a pivot may or may not be a key in the data structure. We now describe the data structure operations:

INSERT(k): The insertion algorithm will first find the list in which k belongs, by finding the smallest j such that $k > p_j$. This is done using binary search on the list of pivots with at most $\log m$ comparisons. k will be inserted into L_j (updating its size s_j). If the inserted key is smaller than all keys in the data structure, it is inserted into L_m and p_m is updated to be k.

The algorithm now makes sure that none of the lists are too large. This is done using the SPLIT-LISTS procedure, which requires an extra amortized cost of at most 6 comparisons per INSERT. From this we get that the *amortized* comparison complexity of INSERT is at most $\log m + 6$. After making sure none of the lists is too large, the algorithm verifies the number of lists isn't too large either. If the combined size of lists L_i and L_{i+1} is at most $n/2^{i(n)}$, the two lists are merged (this requires no comparisons). The new pivot of the merged list is p_{i+1}. Thus there are indeed at most $2^{i(n)+1}$ lists in the data structure after an INSERT operation. The total amortized comparison complexity of the INSERT operation is $\log m + 6 \leq i(n) + 7$.

DELETE(k): The delete operation receives a key to be deleted and its address. The key is simply deleted from its list. The lists are then checked to make sure they are not too small or large. A call to SPLIT-LISTS ensures the lists aren't too large. For every $i \in [m-1]$, if the combined size of lists L_i and L_{i+1} is at most than $n/2^{i(n)}$, they are merged as in the INSERT operation. The DELETE operation has an amortized cost of at most 12 comparisons, used by the SPLIT-LISTS call.

SELECT(i): To select the i-th largest element from the data structure we locate the smallest j such that $\sum_{l=1}^{j} s_l > i$. This does not require any comparisons *on the keys stored in the data structure*. We then select the $(i - \sum_{l=1}^{j-1} s_l)$-th largest element from L_j, using at most $3|L_j| \leq 6n/2^{i(n)}$ comparisons. It is unfortunate that the true complexity of selection is much higher—in a naive implementation, to find the index j, the selection algorithm needs to sequentially scan all list sizes and sum them all, requiring $O(2^{i(n)})$ operations, for a total complexity of $O(2^{i(n)} + n/2^{i(n)})$, which is at least $O(\sqrt{n})$.

SPLIT-LISTS: This procedure is used by INSERT and DELETE to ensure that no list is too large. The procedure examines each list in the data structure and if it has become of size $2n/2^{i(n)}$ it is split around its median into two lists of equal sizes. A list L_j can become too large either via keys inserted into it or via keys deleted from other lists. Splitting the list requires $6n/2^{i(n)}$ comparisons. We will "invest" a constant number of comparisons per INSERT and DELETE operation, and the analysis will show that we always have enough comparisons "invested" to split the list. In conclusion, L_j is split into two lists, L_j and L_{j+1}, each of size $\frac{n}{2^{i(n)}}$. The pivot of the smaller list remains p_j, the pivot of the larger list is the median (around which the list was split).

For the list L_j we will refer to n_j, the size of the data structure at the last point in time when the list was "balanced"—of size $n_j/2^{i(n_j)}$ when the data structure was of size n_j. Note that after it is split a list is always balanced, and thus any operations run since the last time the list was balanced were certainly run after the last split. If $n \geq n_j$ the data structure has grown since the list was last balanced, and it suffices to consider INSERT operations since that time. Since $n_j \leq n$ we know that $n_j/2^{i(n_j)} \leq n/2^{i(n)}$. Thus at least $n/2^{i(n)}$ keys were inserted into L_j since it was last balanced. Splitting L_j around the median takes $6n/2^{i(n)}$ comparisons, which means that "investing" 6 comparisons in each key inserted into the data structure will suffice.

If $n < n_j$ then the list has grown too large via keys deleted from other lists, and possibly via insertions into the list. Let a be the number of keys inserted into L_j since the last time it was balanced. We get that $\frac{n_j}{2^{i(n_j)}} + a \geq \frac{2n}{2^{i(n)}} \geq \frac{n}{2^{i(n_j)}} + \frac{n}{2^{i(n)}}$. Thus $\frac{n_j - n}{2^{i(n_j)}} + a \geq \frac{n}{2^{i(n)}}$. Investing 6 comparisons per INSERT operation and 12 comparisons per DELETE operation will suffice. To see this, consider that in each "delete" operation 12 comparisons are "invested", and these comparisons are divided equally between all lists in the data structure. Now note that at least $n_j - n$ items have been deleted from the data structure since L_j was last balanced. The comparisons invested by these DELETE operations are split between at most $2^{i(n_j)+1}$ lists. Thus a total of $6\frac{n_j - n}{2^{i(n_j)}}$ comparisons are invested in L_j via DELETE operations. The total number of comparisons invested in L_j since it was last split is at least $6(\frac{n_j - n}{2^{i(n_j)}} + a) \geq \frac{6n}{2^{i(n)}}$ and thus enough comparisons were invested to "split" L_j.

Conclusion: We have presented a data structure with amortized INSERT complexity of $i(n)+7$ comparisons and amortized DELETE complexity of 12 comparisons, in which SELECT requires at most $6n/2^{i(n)}$ comparisons. This is asymptotically equivalent to the lower bound of $n/e2^{2(i(n)+7)}$ given in [7] for FINDMIN. Thus, SELECT is asymptotically equivalent to FINDMIN (and FINDANY) under INSERT and DELETE operations. This data structure also demonstrates tightness of our lower bounds for maintaining SELECT and FINDMEDIAN in a data structure and of the lower bound for selection with preprocessing, as it can be used as a preprocessing algorithm.

6 An explicit Adversary

In this section we present an adversary for comparison based algorithms. The adversary is based on a new approach for analyzing comparison based algorithms, examining *potential value intervals.* We assume that the comparison based algorithm receives the input keys in some data structure (possibly a simple array). The analysis is based on looking at each key in the data structure as being drawn from a potentially large set of values. For example, initially (with no previous processing) one may think of every one of the n keys in the data structure as getting any value in the set $[n]$. As comparisons are made, the number of possible values for each key involved in comparisons drops. We will show that comparisons reduce the number of possibilities slowly.

Consider a data structure holding n keys $k_1, k_2 \dots k_n$. Before any work has been invested in structuring the data, each of the keys potentially holds any possible value[1]. We distill this disorder by thinking of each such key as a set of potential values. Initially, each key k_i is thought of as drawn from a set A_i of n potential values. The sets $A_1 \dots A_n$ consist of n values each, $A_i = \{a_i^1, a_i^2, ...a_i^n\}$, and n^2 values all together. These n^2 values, $\{a_j^i\}_{j\in[n]}^{i\in[n]}$, are distinct and ordered and satisfy:

$$\begin{gathered}\{a_1^1, a_2^1 \dots a_n^1\} < \\ \{a_1^2, a_2^2 \dots a_n^2\} < \\ < \cdots < \\ \{a_1^n, a_2^n \dots a_n^n\}.\end{gathered}$$

Initially the ordering of the values within each set $\{a_1^j, a_2^j \cdots a_n^j\}$ is undetermined.

When thinking of the potential values of a key we sometimes consider the *indices* of possible values and not the values themselves. Initially, the set of potential indices for each key is the interval $[1..n]$, meaning, that any value in the set A_i is possible. In our analysis, the set of potential indices will always form an interval of indices, shortening to sub-intervals of $[1..n]$ as more comparisons are made. Thus, if I_i is the interval of possible indices of the key k_i, then the set of values k_i may get is $\{a_i^j\}_{j\in I_i}$.

As initially $I_i = [1...n]$ for all $1 \leq i \leq n$, the algorithm uses comparisons to narrow down intervals of possible values to obtain a better idea of the ordering of values seen so far. Comparing pairs of keys (or intervals) is the only tool the comparison-based algorithm can use to narrow down intervals. For example, if intervals I_i and I_j are compared, and the result of the comparison is that $I_i > I_j$ then it must be that $\forall k \in I_i, l \in I_j : a_i^k > a_j^l$. Our construction of the values $\{a_t^s\}$ is such that $a_i^k > a_j^l$ implies $k \geq l$. Thus, after two intervals are compared they cannot overlap—if they overlapped before the comparison, at least one of them must decrease in size. The only exception is that the smallest index k in the larger interval I_i may also be the largest index (also, k) in the smaller interval I_j. This can happen if $a_i^k > a_j^k$. Note that intervals of potential indices remain contiguous after comparisons are made. This is because when an interval shrinks after a comparison it "loses" only its largest elements or its smallest elements, depending on the result of the comparison.

6.1 The Adversary

We now explicitly describe the adversary. Whenever an algorithm makes a comparison between two keys, and if the two possible answers are compatible with the history so far, then our adversary will determine an outcome for the comparison. Key intervals shrink as comparisons are made. The adversary will attempt to ensure the sum of interval sizes does not decrease too much, and will maintain at all times a set of intervals consistent with its past answers. The adversary will also maintain ordered subsets of every set $\{a_1^j, a_2^j \cdots a_n^j\}$ for each $j \in [n]$. Recall that initially no order has been determined between all the j-th elements of all key intervals. Such an ordered subset will be marked as O_j. Even though there may

[1]We restrict ourselves to integral keys for simplicity, but keys drawn from any ordered set will do.

be many possible sets of intervals and ordered subsets consistent with the adversary's past answers, the adversary maintains only one such set. For any comparison requested by the algorithm we will explicitly define the adversary's behavior. All intervals I_i are initialized to be $[1..n]$ and ordered subsets are all initially empty.

Maintaining ordered subsets: an update to the ordered subset is only applied when an interval I_i shrinks to size 1. In this case, the value a_i^j remaining in this interval is determined to be the smallest between the j-th values determined so far. Namely, if $I_i = \{j\}$ and O_j is $a_{l_1}^j < a_{l_2}^j < \ldots < a_{l_m}^j$ then O_j is updated to be $a_i^j < a_{l_1}^j < \ldots < a_{l_m}^j$. Intervals only shrink and never expand, thus an interval only shrinks to size 1 once and each of the n intervals adds at most one item to some ordered subset.

Maintaining intervals. We now specify how the adversary acts when keys k_i and k_j with corresponding intervals I_i and I_j are compared. There are several possible cases:

1. The two intervals are disjoint—$I_i \cap I_j = \emptyset$. The result of the comparison has already been determined. The intervals are both contiguous and thus all values in one of the intervals are larger than all values in the other. The adversary simply replies that the interval of larger values is the larger of the two intervals. I_i and I_j remain unchanged.

2. The two intervals are not disjoint and $|I_i| = |I_j| = 1$ ($I_i = I_j = \{k\}$). The adversary returns an answer according to the ordered subset O_k. a_i^k and a_j^k must both be in O_k, and their order is already determined. I_i and I_j remain unchanged. This is the only case in which I_i and I_j are not disjoint after a comparison has been made between k_i and k_j.

3. The two sets are not disjoint and $I_i \not\subseteq I_j$ **and** $I_j \not\subseteq I_i$**.** Let B be the intersection of the two intervals—$B = I_i \cap I_j$. B, like I_i and I_j, is a contiguous interval and $B \subseteq [1..n]$. We split B evenly (more or less) between the two intervals. Suppose w.l.o.g $(I_i \setminus B) > (I_j \setminus B)$. Let B_{high} be the largest $\frac{|B|}{2}$ values in B, and B_{low} be $B \setminus B_{high}$. The adversary updates I_i to be $I_i \setminus B_{low}$, I_j to be $I_j \setminus B_{high}$, and returns the answer $k_i > k_j$.

4. The two sets are not disjoint, I_j **is of size greater than** 1**, and** $I_i \subseteq I_j$ ($I_j \subseteq I_i$ **is a symmetric case).** I_i "splits" I_j into two unconnected blocks—I_j^{high} which is the set of indices in I_j larger than any index in I_i, and I_j^{low}, the set of indices in I_j smaller than any index in I_i. Suppose, first, that $|I_j^{low}| > |I_j^{high}|$, and define B, B_{high}, B_{low} as in case 3. The adversary updates I_j to be $B_{low} \cup I_j^{low}$, I_i to be B_{high} and returns the answer $k_i > k_j$. If $|I_j^{low}| < |I_j^{high}|$ then the adversary updates I_j to be $B_{high} \cup I_j^{high}$, I_i to be B_{low} and returns the answer $k_i < k_j$.

Remark: In cases 3,4 if B is not of even size, suppose w.l.o.g $|I_j| > |I_i|$ (and in case 4 $|I_j^{low}| < |I_j^{high}|$). B_{high} will be the largest $\frac{|B|-1}{2}$ values in B, B_{low} will, as usual, be $B \setminus B_{high}$. In other words, the interval of smaller size gets the larger chunk of B.

The adversary we have just described maintains a few important invariants:

Invariant 1 *An interval never becomes empty.*

Invariant 2 *The adversary leaks no information on which of the values whose indices are in interval I_i is indeed the value of k_i, and no information about the order of $\{a_1^j, a_2^j \cdots a_n^j\}$ except the information in the ordered subset O_j.*

Invariant 3 *Intervals maintained by the adversary always remain contiguous.*

6.2 Sum of Interval Sizes

We are now ready to prove that if the adversary is employed versus a comparison based algorithm then the sum of interval sizes shrinks slowly as comparisons are made:

Lemma 1 *If a data set contains n intervals of size n then after making at most $\alpha(n) \cdot n$ comparisons, if the adversary is employed, the sum of the sizes of these intervals is at least $[n^2/(2^{2\alpha(n)}) - n]$.*

Proof: When interval I_i of size s_0^i is involved in a comparison operation with interval I_j its size decreases by at most a factor of 2—of the indices in $I_i \setminus I_j$ at most half are removed (this only happens in case 4). Of the elements in $I_i \cap I_j$ at most $(|I_i \cap I_j|+1)/2$ are removed (this happens in cases 3 and 4). Thus, I_i's new size, s_1^i, satisfies $s_1^i \geq \frac{s_0^i}{2} - \frac{1}{2}$. It follows that after k comparisons I_i's size, s_k^i, satisfies $s_k^i \geq \frac{s_0^i}{2^k} - 1$.

We denote by sum_n the sum of interval sizes after all of the comparisons have been made. If k_i is the number of comparisons involving interval I_i we get:

$$sum_n \geq \sum_{i=1}^{n} (\frac{n}{2^{k_i}} - 1)$$

Since there are at most $\alpha(n) \cdot n$ comparisons during the sequence of operations, and each comparison involves two intervals, we get $\sum_{i=1}^{n} k_i \leq 2\alpha(n) \cdot n$. And thus $sum_n \geq [n^2/(2^{2\alpha(n)}) - n]$. □

We have established the fact that the accumulated size of the intervals deteriorates slowly. We now deduce that there is a large overlap of intervals on some index.

Lemma 2 *If there are n intervals whose combined size is at least $[n^2/(2^{2\alpha(n)}) - n]$, then there exists some index $j \in [1..n]$ such that for at least $[n/(2^{2\alpha(n)}) - 2]$ intervals I_i , $|I_i| > 1$ and $j \in I_i$.*

Proof: There are n intervals of accumulated size at least $[n^2/(2^{2\alpha(n)}) - n]$. If we consider only intervals of size larger than 1, then the sum of interval sizes must be at least $[n^2/(2^{2\alpha(n)}) - 2n]$. From a simple counting argument we get that there exists some $j \in [1..n]$ such that j is in at least $[n/(2^{2\alpha(n)}) - 2]$ such intervals. □

We now use Lemmas 1, 2 to show that if an algorithm does not make too many comparisons (in a preprocessing phase or during data structure maintenance), then employing the adversary ensures that there is a large subset of keys for which nothing is known about their order.

Lemma 3 *Let D be a data set whose n keys may get values in n intervals of size n as above. Let M be a comparison based algorithm. Then after M makes at most $\alpha(n) \cdot n$ comparisons on D, if the adversary is employed, there is a set K of at least $[n/(2^{2\alpha(n)}) - 2]$ different keys and an index $j \in [n]$ such that $\forall k_i \in K$, k_i may get the value a_i^j and nothing is known about the order of the values $\{a_i^j\}_{i|k_i \in K}$ (and thus K is an independent set).*

Proof: By Lemma 1, after M runs there must be n intervals of combined size at least $[n^2/(2^{2\alpha(n)}) - n]$. By Lemma 2 there exists some index j such that there are at least $[n/(2^{2\alpha(n)}) - 2]$ intervals I_i for which $|I_i| > 1$ and $j \in I_i$. Let K be the set of keys k_i such that $j \in I_i$ and $|I_i| > 1$. By Invariant 2, $\forall k_i \in K$, k_i may indeed get the value a_i^j and nothing is known about the order of the values $\{a_i^j\}_{i|k_i \in K}$. □

It is important to note these lemmas also hold for dynamically maintained data structures making INSERT and DELETE operations. In this case, the algorithm is defined to include all (comparison) operations executed when INSERT or DELETE (or any other data structure) operations were run. The number of elements in the input set is the number of keys inserted into the data structure. Note that lower bounds are easier and upper bounds (algorithms) are harder to construct with dynamically maintained data structures. The reason is that preprocessing may access all the keys from the beginning. Dynamically maintained data structures run only a small number of comparisons per operation, do not know in advance what keys will be inserted and can only access keys that are currently in the data structure. They are thus more limited than a preprocessing algorithm. We will now show that the adversary can be used to ensure that after preprocessing there is at least one index for which a SELECT operation is especially hard. We give an information theoretic lower bound for the selection operation, though any known adversary for selection (e.g. that of [4]) can be used without any modifications to ensure the selection operation also runs slowly.

Theorem 3 *Let D be a data set containing n distinct values drawn from an ordered set. Let P be a comparison based preprocessing algorithm that receives D, and S a comparison based selection algorithm that receives P's output and an index i and outputs the i-th largest element in D. If P makes at most $\alpha(n) \cdot n$ comparisons then the adversary can be used to ensure that S must make at least $(2+\epsilon)[n/(2^{2\alpha(n)}) - 2]$ comparisons in the worst case.*

Proof: S is a SELECT operation performed after preprocessing with at most $\alpha(n) \cdot n$ comparisons. By Lemma 3 we know that if the adversary is employed while S runs then there exists an independent set W of the input keys of size at least $[n/(2^{2\alpha(n)}) - 2]$. By an argument similar to that in the proof of Theorem 1, this implies that selecting the median from W requires that at least $(2+\epsilon)|W|$ comparisons be made in the worst case. □

7 Acknowledgements

The second author wishes to thank the Technion's Graduate School and Department of Computer Science for generous support.

References

[1] Helmut Alt, Kurt Mehlhorn. "Searching Semisorted Tables". *SIAM Journal on Computing.* 14(4): 840-848 (1985)

[2] Mikhail J. Atallah, S. Rao Kosaraju. "An Adversary-Based Lower Bound for Sorting". *Information Processing Letters.* 13(2): 55-57 (1981)

[3] S. W. Bent, J. W. John. "Finding the Median Requires 2n Comparisons". *STOC 1985*: 213-216, 1985.

[4] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, R. E. Tarjan. “Time bounds for selection”. *Journal of Computer and System Sciences*, 7(4):448-461, 1973.

[5] Allan Borodin, Leonidas J. Guibas, Nancy A. Lynch, Andrew Chi-Chih Yao. “Efficient Searching Using Partial Ordering”. *Information Processing Letters.* 12(2): 71-75 (1981).

[6] Allan Borodin, Faith E. Fich, Friedhelm Meyer auf der Heide, Eli Upfal, Avi Wigderson. “A Tradeoff Between Search and Update Timefor the Implicit Dictionary Problem”. *Theoretical Computer Science.* 58: 57-68 (1988)

[7] G. S. Brodal, S. Chaudhuri, J. Radhakrishnan. “The Randomized Complexity of Maintaining the Minimum”. *Nordic Journal of Computing.* 3(4):337-351, 1996.

[8] D. Dor, U. Zwick. “Finding The alpha n-Th Largest Element”. *Combinatorica* 16(1): 41-58, 1996.

[9] D. Dor, U. Zwick. “Selecting the Median”. *SIAM Journal of Computing.* 28(5): 1722-1758, 1999.

[10] D. Dor, U. Zwick. “Median Selection Requires $(2+\epsilon)n$ Comparisons”. *SIAM Journal of Discrete Math.* 14(3):312-325, 2001.

[11] J. R. Driscoll, H. N. Gabow, R. Shrairman, R. E. Tarjan. “Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation”. *Communications of the ACM.* 31(11): 1343-1354, 1988.

[12] F. Fussenegger, H. N. Gabow. “A Counting Approach to Lower Bounds for Selection Problems”. *Journal of The ACM.* 26(2): 227-238, 1979.

[13] C. A. R. Hoare. “Algorithm 63: partition”. *Communications of the ACM.* 4(7): 321, 1961.

[14] C. A. R. Hoare. “Algorithm 64: Quicksort”. *Communications of the ACM.* 4(7): 321, 1961.

[15] D. G. Kirkpatrick. “A Unified Lower Bound for Selection and Set Partitioning Problems”. *Journal of the ACM.* 28(1): 150-165, 1981.

[16] Harry G. Mairson. ”Average Case Lower Bounds on the Construction and Searching of Partial Orders”. *FOCS 1985*: 303-311.

[17] Colin McDiarmid. ”Average-Case Lower Bounds for Searching“. *SIAM Journal on Computing.* 17(5): 1044-1060 (1988)

[18] I. Munro and P.V. Poblete. “A lower bound for determining the median”. Technical Report Research Report CS-82-21, University of Waterloo, 1982.

[19] M. Paterson. “Progress in Selection”. *SWAT 1996*: 368-379, 1996.

[20] V. R. Pratt, F. F. Yao. “On Lower Bounds for Computing the i-th Largest Element”. *FOCS 1973*: 70-81, 1973.

[21] A. Schönhage, M. Paterson, N. Pippenger. “Finding the Median”. *Journal of Computer and System Sciences.* 13(2): 184-199 (1976)

[22] C.K. Yap. “New lower bounds for medians and related problems”. Computer Science Report 79, Yale University, 1976. Abstract in *Symposium on Algorithms and Complexity: New Results and Directions*, (J. F. Traub, ed.) Carnegie-Mellon University, 1976.

A FindAny in Data Structures

The approach of using preprocessing before a SELECT operation is only interesting when the index to be selected is not known in advance—if it is known which rank is to be selected (e.g. the minimum, the median etc.) there is no use in a separate preprocessing stage. This is also the case for the FINDANY operation, which returns some key in the input set and its rank. All information is available once the input set is revealed, and there is no need for preprocessing. It is, however, interesting to consider the complexity of the FINDANY operation in a dynamically maintained data structure with INSERT and DELETE operations. This question was first considered by Brodal et al. in [7]. They have shown that if the worst case amortized complexity of the INSERT and DELETE operations on the data structure is $t(n)$, where n is the number of keys in the data structure, then the FINDANY operation requires at least $[n/(2^{4t(n)+3}) - 1]$ comparisons in the worst case. In this section we reformulate the proof of [7] using the potential value intervals of section 6. We simplify the proof and improve the lower bound by a multiplicative factor of 8. We begin with a proposition used in the subsequent proof of adversarial lower bounds for data structures:

Proposition 1 *Let D be a comparison based data structure and S an operation sequence. Let K and j be the set of keys and index, whose existence after S is run is guaranteed by Lemma 3. For any operation sequence S' which is a prefix of S, after S' is run, $\forall k_i \in K$, k_i may get the value a_i^j and nothing is known about the order of the values $\{a_i^j\}_{i|k_i \in K}$ (If K is an independent set after the operation sequence S it is also an independent set after any prefix of S).*

Proof: D has no knowledge about future operations and thus the view of operation sequence S' is identical to the view of S when running the operations of S'. Thus after S' is run all intervals are at least as large as they are after S is run and from here we can proceed as in the proof of Lemma 3. $\square$

Theorem 4 *Let D be a comparison based data structure supporting* INSERT, DELETE *and* FINDANY *operations. If the amortized comparison complexities of D's* INSERT *and* DELETE *operations are $i(n)$ and $d(n)$ respectively, where n is the number of keys in the data structure, then D's* FINDANY *operation must make at least $[n/(2^{2(i(n)+d(n))}) - 3]$ comparisons in the worst case.*

Proof: We begin similarly to the proof of [7] and define a series of operation sequences $S_1 \ldots S_{n+1}$:

$$S_1 = \text{INSERT}(k_1) \ldots \text{INSERT}(k_n)\text{FINDANY}$$

b_1 is the key returned by the FINDANY operation in S_1. We similarly define S_i for $2 \leq i \leq n$:

$$S_i = \text{INSERT}(k_1) \ldots \text{INSERT}(k_n)\text{DELETE}(b_1) \ldots \text{DELETE}(b_{i-1})\text{FINDANY}$$

Where b_i is the key returned by the FINDANY operation at the end of sequence S_i and

$$S_{n+1} = \text{INSERT}(k_1) \ldots \text{INSERT}(k_n)\text{DELETE}(b_1) \ldots \text{DELETE}(b_n)$$

We look at deleted elements as intervals that are simply no longer accessible by the data structure. Examining the sequences, we note that the FINDANY operation is always preceded by n INSERT operations and at most n DELETE operations. Thus, there are at most $(i(n) + d(n)) \cdot n$ comparisons made before any FINDANY operation. By Lemma 3, in sequence S_{n+1} there is a set A of size at least $n/(2^{2(i(n)+d(n))}) - 2$ of (deleted) keys about whose order nothing is known. Let b_{min} be the first key in A that is deleted during S_{n+1}. In the sequence S_{min}, when the FINDANY operation is run, all keys in A are still in the data structure D (by b_{min}'s minimality) and nothing is known about their order (by Proposition 1). The FINDANY operation of sequence S_{min} returns the rank of b_{min}. To know the rank of b_{min} with certainty, at least $|A| - 1$ comparisons must be made in the worst case. $\square$

B A Preprocessing Algorithm for Selection

In this appendix we provide efficient algorithms for preprocessing and selection. These algorithms will be efficient not only in terms of their comparison complexity and will also be in-place. For convenience we will assume n is a power of 2.

Preprocessing Algorithm: We now turn to describing a preprocessing algorithm using at most $3\alpha(n) \cdot n$ comparisons. The preprocessing algorithm of this section is similar to the sorting algorithm Quicksort [14]. The algorithm will split the array A into $2^{\alpha(n)}$ sub-arrays of size $n/2^{\alpha(n)}$: $A_1, \ldots, A_{2^{\alpha(n)}}$, where $A_1 > A_2 > \ldots > A_{2^{\alpha(n)}}$. The array will be split using PARTITION. The Preprocessing algorithm will then be used recursively to split both the left and right parts of the array.

```
PREPROCESS($A, i, j$):
    1. if $j - i + 1 \leq 2^{\alpha(n)}$ then *return*.
    2. PARTITION($A, i, j, \lfloor \frac{j-i+1}{2} \rfloor$).
    3. PREPROCESS($A, i, i + \lfloor \frac{j-i+1}{2} \rfloor$).
    4. PREPROCESS($A, i + \lfloor \frac{j-i+1}{2} \rfloor + 1, j$).
```

The preprocessing algorithm splits the array into $2^{\alpha(n)}$ different sub-arrays $A_1 \ldots A_{2^{\alpha(n)}}$ where $A_1 > \ldots > A_{2^{\alpha(n)}}$ and A_i contains the values located at addresses $\frac{(i-1)\cdot n}{2^{\alpha(n)}} \ldots \frac{i \cdot n}{2^{\alpha(n)}} - 1$ of the array. The comparison complexity of the preprocessing algorithm with any pair (i, j) satisfying $n = j - i + 1$, $C_p(n)$, satisfies $C_p(n) \leq 2 \cdot C_p(\frac{n}{2}) + 3n$ and $C_p(\frac{n}{2^{\alpha(n)}}) = 0$ (note that we count only comparisons of array keys). Thus, we get that $C_p(n) \leq 3\alpha(n) \cdot n$.

Selection After Preprocessing After the array has been preprocessed using at most $3\alpha(n) \cdot n$ comparisons, it is split into $2^{\alpha(n)}$ sub-arrays. Our selection algorithm will find the sub-array in which the i-th largest key is located—the index of this sub-array is $j = \lfloor \frac{i}{2^{\alpha(n)}} \rfloor + 1$. It will then run SELECT($A[\frac{(j-1)n}{2^{\alpha(n)}} \ldots \frac{jn}{2^{\alpha(n)}} - 1], i - \frac{(j-1)n}{2^{\alpha(n)}} + 1$), using at most $\frac{3n}{2^{\alpha(n)}}$ comparisons.

Conclusion: We show that if $\alpha(n) \cdot n$ time is invested in preprocessing, then SELECT(i) can done using at most $3n/(2^{\frac{\alpha(n)}{3}}) = n/(2^{\Omega(\alpha(n))})$ comparisons. A nice property of this preprocessing algorithm is that it is *efficiently progressive*—it can be run intermittently while selection queries are made, and almost nothing is lost by splitting up the algorithm's run into phases separated by selection queries. This property is very useful when the amount of time for preprocessing is not known in advance, or if it is expected that there will be stretches of time when selection queries are relatively rare, when the algorithm can continue to preprocess the data, finally arriving at a sorted array. While this algorithm has higher comparison complexity than the algorithm of section 5, it is efficient not only in terms of its comparison complexity, but also in terms of its true complexity and is in-place and efficiently progressive.