A Cell Probe Lower Bound for the Predecessor Search Problem in PRAM

Peyman Afshani* Nodari Sitchinava†

Abstract

We study the predecessor search problem in the classical PRAM model of computation. In this problem, the input is a set of n ℓ-bit integers and the goal is to store the input in a data structure of size $S(n)$ such that given a query value q, the predecessor of q can be found efficiently. This is a very classical problem with an extensive history.

We prove a lower bound for this problem in the strongest CRCW PRAM model. A simplified version of the lower bound states that in a K-processor PRAM model with $O(\log n)$-bit registers, the query requires $\Omega(\log_K \log n)$ worst-case time under the realistic setting where the space is near-linear.

1 Introduction

We study the problem of *predecessor search* in the classical parallel RAM (PRAM) model. The problem asks to construct a data structure for an ordered set X from a universe U, such that for any query $q \in U$ we can find efficiently the $\texttt{pred}_X(q) = \max\{x \in X | x \leq q\}$. This is one of the fundamental and well-studied data structuring problems (e.g., see [1, 12]). In this paper, we are interested in the parallel version of this problem, i.e., the fastest way to answer a query when using multiple processors.

Our research is motivated by at least three different other lines of research. To start with, let us consider the very classical problem of merging two sorted lists. Observe that if the predecessor query can be answered in (parallel) time $t(n)$ using K processors, then we can merge two arrays A and B, each of size n, in time $O(t(n))$ using $K \cdot n$ processors (assuming the data structure can be built quickly). The merging problem is settled in the comparison-based setting: in the concurrent-read exclusive-write (CREW) PRAM model, merging can be completed in $O(\log \log n)$ time and $O(n)$ work, i.e, using $\frac{n}{\log \log n}$ processors [9, 15] (which is also optimal [4]). However, what is perhaps even more interesting is that integers consisting of $O(\log n)$ bits can be merged in $O(\log \log \log n)$ time [2]. Ignoring some technicalities, this essentially relies on the observation that for n integers from a polynomial universe, it is possible to perform predecessor search in constant time using $O(\log n)$ processors. Contrast this with the sequential predecessor search solution, which takes $O(\log \log n)$ time, e.g., using the van Emde Boas tree data structure [16]. In other words, it is possible to improve the sequential query time by a $\log \log n$ factor by using exponentially more ($\log n$) processors. If it were possible to achieve the same improvement by using only $\log \log n$ processors, then merging could be potentially done in $O(\log \log \log \log n)$ time.

This observation that the query time of predecessor search queries can be improved by a factor f at the cost of spending 2^f computational resources, has been used in at least three different research directions. We will mention two more examples of it shortly but it should be clear that there is motivation to explore whether such "wasteful" trade-off is the best possible.

Yet, despite its classical nature, to the best of our knowledge, there are no known lower bounds for the predecessor search problem in any version of the PRAM model with concurrent read capabilities. In this paper, we show that any data structure for the predecessor search problem on n integers, each consisting of at most $\ell = c \log n$ bits (for a constant $c > 1$), that uses at most $S(n) = \alpha n$ words of space requires $\Omega\left(\log_K\left(\frac{(\hat{c}-1)\log n}{\log \alpha + \log K + \log w}\right)\right)$ time on a K-processor PRAM model with word size of w bits and $\hat{c} = \min(2, c)$. This bound holds even for the strongest concurrent-read, concurrent-write (CRCW) PRAM model. In what follows, we briefly review the prior work and also mention two additional motivations for studying this problem.

*Supported by DFF (Danmarks Frie Forskningsfond) of Danish Council for Independent Research under grant ID 10.46540/3103-00334B

†Supported by National Science Foundation grants 1911245 and 2432018.

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

1.1 The Motivations and the Prior Work The predecessor search problem has a very long history. Here we only review the most relevant results and refer the reader to a recent survey [12]. In the classical comparison-based model, the problem holds no mysteries as it can be solved optimally with any of the balanced binary search data structures in $\Theta(\log n)$ time, even dynamically.

The problem is certainly more interesting in the word-RAM model, which is also the setting that we consider here. In this model, the machine is equipped with registers of size w bits, the input integers have ℓ bits and they are from a universe of size U (meaning, $\ell \leq \log U$). In the static version of the problem, Pătraşcu and Thorup [13] completely solved the problem by giving four different possible trade-offs for the problem, depending on the range of the various parameters involved. The two most often used ones are $\Theta(\log_w n)$ and $\Theta\left(\log\left(\frac{\ell-\log n}{\log(\alpha)+\log w}\right)\right)$. The latter bound simplifies to $\Theta(\log\log n)$ under the natural assumptions that space is near-linear and $\ell = c\log n$ for some constant c. Thus, the static version of the problem is considered completely settled. There are still some gaps in the bounds for the dynamic settings but here we only focus on the static version of the problem and refer the reader to a recent survey [12] for more details on the dynamic problem.

The results by Pătraşcu and Thorup cover the lower bound side (with minor tweaks to the existing data structures to fit them for some corner cases of the parameters). On the upper bound side, the classical solutions include van Emde Boas trees [16, 17], x-fast and y-fast tries [18], and fusion trees [8].

In the PRAM model, even in the basic comparison-based model, the solutions to the predecessor search problem are more complicated than in the sequential setting. In the EREW PRAM model, comparison-based predecessor problem can be solved in the optimal $\Theta(\log(n/K))$ time. In the CREW PRAM model, this improves significantly to $O(\log_K n)$ time [14]. These results can be combined with the x-fast and y-fast tries, in a relatively straightforward way to obtain data structures. In particular, by using x-fast tries, it is possible to obtain a data structure with $O(\log_K(\log n))$ query time for CREW PRAM and with $O\left(\log\left(\frac{\log n}{K}\right)\right)$ query time in EREW PRAM. By using y-fast tries and hashing (assuming a static data structures), the space can be reduced to linear and the query time can be made deterministic.

To the best of our knowledge, outside the comparison-based model, and the aforementioned straightforward combination of known techniques, the predecessor search problem has not received the same attention in the PRAM model as in its sequential counterpart. The notable exception is the related $O(\log\log\log n)$-time integer merging solution [2] mentioned earlier. While the solution does not cite the x-fast or y-fast tries, the techniques are remarkably similar.

When using x-fast tries (or y-fast tries) one simultaneously probes all the ℓ nodes in the root-to-leaf path in the data structure to answer the predecessor query. This exponentially increases the number of probed memory cells at the cost of making parallel probes or "non-adaptive" probes. This phenomenon has received additional attention in the following two other lines of work.

Non-adaptive data structures. Non-adaptive data structures have received some attention recently, e.g., see [10] for motivations and some discussion of the prior work in the area. A data structure is called *non-adaptive* if the memory locations that the querying algorithm needs to probe only depend on the given query value and not on the contents of those memory locations. On the other hand, in an *adaptive* one, the query algorithm might have to read a different memory cell depending on the contents of the previously read ones.

The non-adaptive variant of the predecessor search problem has been studied in the (sequential) Word-RAM model [3, 10, 11]. The current best lower bound for the predecessor problem in this setting is $\Omega\left(\frac{\ell-\log n}{\log(S(n)/n)}\right)$ [10].

The power of non-adaptability has also been discovered independently in the context of RAM with Byte Overlap (RAMBO) model. More specifically, we refer to RAMBO with Yggdrasil memory layout [5, 7], where all the values along a root-to-leaf path in a tree are assumed to be accessible in constant time due to the byte overlap (presumably implemented in the hardware). It has been observed that under this model, a lot of query problems can be answered in constant time [5].

PRAM: Bridging adaptability and non-adaptability. Both ideas of non-adaptivity and byte-overlap (RAMBO) have strong connections to the PRAM model. They are also motivated by the practical consideration (e.g., RAMBO) that parallel (non-adaptive) memory accesses are faster than the adaptive ones. PRAM model generalizes both. In other words, it is natural to ask what happens with $K \leq \ell$ processors, i.e., what happens if we allow the query to be answered in t adaptive rounds where in each adaptive round we can do K simultaneous (non-adaptive) memory probes? For the concrete case of answering predecessor queries, this brings up the natural question of whether there can be a better trade-off than "exponential" between computational power and

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

increasing non-adaptability.

1.2 Our Results In this paper, we consider the predecessor search problem in the CRCW PRAM model of computation. First, we observe that getting the lower bound of $\Omega(\log_{wK} n)$ is almost trivial: Replace the space of the data structure with $\binom{|S(n)|}{K}$ cells of size Kw by including every choice of K cells out of $|S(n)|$ available cells. Now, the PRAM query algorithm can be simulated by a single processor with word-size Kw and thus we can obtain a lower bound of $\Omega(\log_{Kw} n)$ via previous results [13].

This trick cannot work at all for polynomial universes since blowing up the space by a polynomial factor will make the problem trivial. Due to this, we focus on the case when $\log U = c \log n$ for a constant c. We show that using K processors and $S(n) = \alpha n$ words of space, answering predecessor queries requires $\Omega\left(\log_K\left(\frac{(c-1)\log n}{\log \alpha + \log K + \log w}\right)\right)$ worst-case time.

On the technical side, we first simplify the previous approach [13] by applying a more direct attack and second, we use new ideas to overcome the challenges of proving lower bounds for K processors. To do this, we need to pay attention to the pattern of memory accesses across different processors in the same time unit. This is critical, since we essentially want to prove that with K processors, the query time can only improve by a factor of $\log K$, meaning, it is very important that we exploit that fact that the K memory accesses are done at the same time. This is done by employing new strategies when it comes to "publishing memory cells".

2 Preliminaries

Our lower bound works in the general cell probe model [19]. We assume that we have an algorithm $\mathcal{A}$ such that given any input of n values, it produces a data structure $\mathcal{D}$ that uses $S(n)$ space. The data structure can answer any predecessor query in $Q(n)$ time and we would like to lower bound $Q(n)$.

2.1 Extension to the PRAM model To prove our lower bound the model also needs to take into account the existence of multiple processors. We assume a machine with K processors that are synchronized, with a constant number of registers of size w bits each. At each unit of time, the processors can access K memory cells (we allow concurrent reads and thus some of the reads could overlap). Each memory cell also stores w bits. Additionally, we allow the processors infinite computational power as well as infinite communication power; this in turn implies that the contents of the memory cell read by a processor are immediately available to all the other processors for free. Note that this is a very strong assumption and as we mentioned in the Introduction, a running time of t in this model implies t rounds of adaptive probing where at each round we probe K cells non-adaptively.

2.2 Notations We assume we have a universe of size U where $U = n^c$ for a constant $c > 1$. The input values are n integers from the universe and we assume they are $\ell \leq \log U$ bits long. We use the term *input value* to refer to an (ℓ bit long) integer in the input. We use the term *query* to refer to any integer $q \in [U]$. Thus, queries are $\log U$ bits long. We assume we have a CRCW PRAM with K processors. These processors are synchronized, meaning, at each time step each processes can select a different memory cell to read or write independent of all the other processors. We assume the processors are equipped with a constant number of integer registers of $w \geq \log U$ bits long. We will prove a lower bound for data structures that use $S(n)$ space. Let $\alpha = \frac{S(n)}{n}$.

A *bit range* is an interval $I \subset [\ell]$ which corresponds to a contiguous range of bit positions in an input integer. We will need some terminology to refer to the various bit positions within these ℓ-bit values. The bit positions are labelled left to right from 1 to ℓ, meaning, the bit position 1 is the most significant bit value and the bit position ℓ is the least significant value. A bit position i is said to be to the *left* (resp. *right*) of I if i is smaller (resp. larger) than all the values in I. The *prefix of* I (resp. *suffix of* I) corresponds to all the bit positions to the left (resp. right) of I. The *inclusive prefix of* I includes the prefix of I as well as all the bit positions in I.

Finally, we will use the following basic combinatorial result.

LEMMA 2.1. *Let A be a universe of size N and let $K \geq 1$ be a fixed integer. Let B be the set of all subsets of size at most K of A (i.e., B includes every S such that $S \subset A$ and $|S| \leq K$), and let $\mathscr{X} \subset 2^B$. If every $S \in \mathscr{X}$ contains at least T mutually disjoint subsets, for some parameter T, then there exists a subset $H \subset A$ of size $|H| = O\left(N\left(\frac{\log|\mathscr{X}|}{T}\right)^{1/K}\right)$, such that for every $S \in \mathscr{X}$ there exists a subset $\tau \in S$ such that $\tau \subset H$.*

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

Proof. Let H be a random sample of A, chosen by sampling each element of A with probability $p = (C \log |\mathscr{X}|/T)^{1/K}$, where $C > 0$ is a constant of our choice. For $S \in \mathscr{X}$ and a subset $\tau \in S$, we say τ is fully sampled if $\tau \subset H$. This happens with probability $p^{-|\tau|} \geq p^{-K}$. If S contains at least T mutually disjoint subsets, then the expected number of subsets that get fully sampled is at least $Tp^K = C \log |\mathscr{X}|$. Then, by choosing large enough C, the Chernoff bound implies that we will be hitting every subset S with probability at least $1 - 1/|\mathscr{X}|^2$. The claimed number of sampled elements follows from the application of the union bound. □

2.3 Basic Concepts in Information Theory We will also use the following basic concepts from information theory. These can be found in any introductory course on the subject, e.g., [6].

Given a random variable $\boldsymbol{X}$ with distribution $\mu_{\boldsymbol{X}},$, the *entropy* of $\boldsymbol{X}$, denoted by $H(\boldsymbol{X})$ is defined as

$$H(\boldsymbol{X}) = -\sum_v \mu_{\boldsymbol{X}}(v) \log_2 \mu_{\boldsymbol{X}}(v)$$

where $\mu_{\boldsymbol{X}}(v)$ is the probability of $\boldsymbol{X} = v$. For two random variables $\boldsymbol{X}$ and $\boldsymbol{Y}$, the joint entropy $H(\boldsymbol{X}, \boldsymbol{Y})$ is defined similarly with respect to the joint distribution. The conditional entropy $H(\boldsymbol{X}|\boldsymbol{Y})$ is defined as $H(\boldsymbol{X}|\boldsymbol{Y}) = -\sum_v \mu_{\boldsymbol{Y}}(v) H(\boldsymbol{X}|\boldsymbol{Y} = v)$. Finally, the mutual information between $\boldsymbol{X}$ and $\boldsymbol{Y}$, denoted by $I(\boldsymbol{X}; \boldsymbol{Y})$ is defined as $I(\boldsymbol{X}; \boldsymbol{Y}) = H(\boldsymbol{X}) + H(\boldsymbol{Y}) - H(\boldsymbol{X}, \boldsymbol{Y})$. The *relative entropy* or *Kullback-Leibler divergence* between two probability distributions μ_1 and μ_2 is denoted by $\mathrm{D}_{\mathrm{KL}}(\mu_1 \,||\, \mu_2)$ and it is defined as

$$\mathrm{D}_{\mathrm{KL}}(\mu_1 \,||\, \mu_2) = \sum_v \mu_1(v) \log(\mu_1(v)/\mu_2(v)).$$

Note that this requires $\mu_2(v)$ to be non-zero.

LEMMA 2.2. *The entropy has the following properties.*

- $I(\boldsymbol{X}; \boldsymbol{Y}) = H(\boldsymbol{X}) - H(\boldsymbol{X}|\boldsymbol{Y}) = H(\boldsymbol{Y}) - H(\boldsymbol{Y}|\boldsymbol{X})$.
- *If* $\boldsymbol{X}$ *is a function of* $\boldsymbol{Y}$ *when* $H(\boldsymbol{X}, \boldsymbol{Y}) = H(\boldsymbol{Y})$ *and* $H(\boldsymbol{X}|\boldsymbol{Y}) = 0$ *and* $I(\boldsymbol{X}; \boldsymbol{Y}) = H(\boldsymbol{X})$.
- *If* $\boldsymbol{X}$ *and* $\boldsymbol{Y}$ *are independent then* $H(\boldsymbol{X}, \boldsymbol{Y}) = H(\boldsymbol{X}) + H(\boldsymbol{Y})$ *and* $I(\boldsymbol{X}; \boldsymbol{Y}) = 0$.
- $I(\boldsymbol{X}; \boldsymbol{Y}) = \sum_v \mu_{\boldsymbol{X}}(v) D_{KL}(\mu_{\boldsymbol{Y}|\boldsymbol{X}=v} \,||\, \mu_{\boldsymbol{Y}})$.
- $H(\boldsymbol{X}, \boldsymbol{Y}|\boldsymbol{Z}) \leq H(\boldsymbol{X}|\boldsymbol{Z}) + H(\boldsymbol{Y}|\boldsymbol{Z})$.

3 A Predecessor Lower Bound

In this section we prove our main result. Before going forward, let us quickly summarize the main challenges in adapting the techniques by Pătraşcu and Thorup [13] to the PRAM model. They use a probe elimination approach that shows that by publishing a certain subset of the data structure, one can simulate a (small) subset of queries in such a way that it causes a contradiction if the query time is too fast. They use different strategies to publish memory cells that involve randomly sampling as well as publishing cells that are most often used. In the PRAM model, we have K processors and thus *all* the probes made at a time step must be eliminated at the same time. However, the different processors can have very different memory access patterns (i.e., they can be very "entangled") and this requires new ideas as the original strategies for publishing memory cells no longer work. An additional challenge is that the probe elimination of Pătraşcu and Thorup is indirect.

We use a more direct approach that involves first building a random distribution of difficult inputs. Then, we follow a round-elimination approach, in the spirit of the previous proofs. Another modification that we make is to allow for different queries to have their probes eliminated at different stages. This approach enables us to generalize the lower bound argument to the PRAM setting, although, the overall line of attack is similar: we publish some cells (using some new ideas) and published bits can be used by the data structure for free. This allows us to eliminate probes in the data structure. Eventually, we will arrive at a situation where "few" cells have been published but a lot of probes have been skipped. Thus, if the query time was too fast, then most queries can be answered by only looking at the published bits. But since there will be too few such bits it'll be impossible to differentiate some of the remaining queries using only the published bits, with this contradiction implying a lower bound on the number of probes for some of the queries.

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

3.1 Input Construction We need to construct n bit strings of size ℓ. The bits in these bit strings are not created at the same time; they are created in a number of *steps*. The details are as follows.

At the beginning of the i-th step, we have the following situation. The n input values are distributed into a number, t_i, of sub-problems. In each sub-problem, we have to construct $n_i = \frac{n}{t_i}$ values. Each sub-problem S is associated with a bit range R_S that we call *defining bit range* of S and a fixed value v assigned to the prefix of R_S. This sub-problem S is tasked with creating n_i bit strings such that they all have the same value v assigned to the prefix of R_S, and that all the n_i bit strings are distinct within the bit range R_S. For this latter point to be possible, the construction needs to maintain an invariant that

$$|R_S| \geq \log n_i. \tag{3.1}$$

We will verify that this invariant is maintained throughout the construction. During each step, we apply some of the following operations. The construction terminates when we reach a certain base case.

Operation one: *Splitting.* Consider a sub-problem S with its defining bit range R_S. Given a parameter x, the splitting operation is the following. S is replaced by 2^x sub-problems each tasked with creating $\frac{n_i}{2^x}$ values. The x most significant bits of the j-th sub-problem are used to encode j and this encoding is placed at the x rightmost bit positions of the bit range R_S. The defining bit range of the j-th sub-problem is obtained from R_S after removing the x rightmost bit positions. It is easy to see that this operation preserves the construction invariant.

Operation two: *branching.* Consider a sub-problem S with n_i values and a defining bit range R_S. Given a parameter b, the interval R_S is divided into b equal-sized sub-intervals, $R_1, \cdots, R_b$. Then, one sub-interval R_j is chosen uniformly at random to be the new defining interval and the prefix of R_j within R_S is filled with a random bit string. The new sub-problem is tasked with creating n_i values, same as S. Thus, this operation can be seen as reducing the size of R_S to its random sub-interval. To preserve our invariant, we need to ensure that $\log n_i \leq \frac{|R_S|}{b}$.

The base case and the queries. The construction terminates when there is not enough bits left in the defining range of S, i.e., R_S has too few bits. In this case, we simply create n_i distinct random values within R_S. For each created value q, we set the bits in the suffix of R_S to zero. If R_S has T bits of suffix, then we create 2^T queries, one for each possible choice of the T bits and we call these *matching* queries of q. We consider the worst-case query among all these queries.

In particular, given a fixed input I, which results in a data structure D, we will be publishing some of the cells in D in a step-by-step fashion. We use P_i to denote the set of published words, with p_i being their number. We use $\boldsymbol{I}$ and $\boldsymbol{P}_i$ to denote random variables that correspond to the random input generated and the set of published words at the i-th step, respectively.

Parameters of the construction. The parameters of the construction are set up to make the later probe elimination work. Here, we describe the details of the construction but also mention a few things about the future arguments where we establish our publishing strategy. We assume $\ell = c \log n$ where c is a constant. Initially, we start with one sub-problem, with a defining interval of size ℓ that will create n values and we have $p_0 = 0$, meaning, no published words. For simplicity, we assume $1 < c \leq 2$ (if c is a larger constant, we simply ignore the larger universe and work with a universe of size n^2).

To initialize, we apply the splitting operation with parameter $(2-c)\log n$; if $c = 2$, then we skip this initialization step. Otherwise, this creates $t_0 = n^{2-c}$ sub-problems of size $n_0 = n^{c-1}$, each with a defining interval of size $\ell_0 = c\log n - (2-c)\log n = (2c-2)\log n$. The point of this initialization is to make sure that $\ell_0 = 2\log n_0$. Then, we begin our step-by-step construction and at each step, we first apply branching and then splitting. Generally speaking, at the beginning of the i-th step, we have t_{i-1} sub-problems where each sub-problem is to create $n_{i-1} = \frac{n}{t_{i-1}}$ input values and each sub-problem S has some defining range of size ℓ_{i-1}; the defining ranges of the sub-problems have the same length but they could be at different bit positions.

Assume that we would like to show a lower bound for a data structure that uses $S(n)$ words. Define $\delta_i = (\alpha n_i)^{\frac{K}{K+1}}$ and $d_i = \delta_i t_i$. At step i, first we apply branching with parameter $4K$ and then we apply splitting with a parameter such that the number of sub-problems increases to $\delta_i^{1+\lambda} t_i w^5 K$, where $\lambda > 0$ is some fixed constant to make the construction work.

Observation 3.1. *There exist constants $\lambda > 0$ and $\varepsilon > 0$ such that we can iterate the construction for $\gamma = \varepsilon \log_K(\frac{(c-1)\log n}{\log \alpha + \log K + \log w})$ steps. In addition, once the recursion stops, each sub-problem has at least*

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

$n_\gamma = n^{(\log n)^\varepsilon}$ *points.*

Proof. Observe that initially, we have t_0 sub-problems of size n_0 each with a defining range of length ℓ_0 and we have $n_0 = 2^{\ell_0/2}$. We also have $\delta_0 = (\alpha n_0)^{K/(k+1)}$ and $d_0 = \delta_0 t_0$.

In the next step (after applying branching one and then splitting once), the number of sub-problem increases to $t_1 = \delta_0^{1+\lambda} t_0 w^5 K$ and in general, we have $t_{i+1} = \delta_i^{1+\lambda} t_i w^5 K$ and $\delta_i = (\alpha n_i)^{K/(K+1)}$, $n_i = \frac{n}{t_i}$. This gives a recursion for n_{i+1}:

$$\frac{n}{n_{i+1}} = (\alpha n_i)^{(1+\lambda)\frac{K}{K+1}} \frac{n}{n_i} w^5 K.$$

We can set λ such that $(1+\lambda)\frac{K}{K+1} = 1-\beta$ for some constant $\frac{1}{2} > \beta > 0$ and this solves to the following:

$$n_{i+1} = \frac{n_i^\beta}{\alpha^{1-\beta} K w^5}.$$

We get a similar recursion for t_i. The only subtlety is that there should be enough bits in the defining ranges to allow the branching operations to work. For this, observe that it is sufficient to set λ to a constant such that $\beta \leq \frac{1}{8K}$.

To bound the depth of the recursion, observe that the recursive definition of n_i solves to

$$n_i = \frac{n_0^{\beta^i}}{\alpha^{O(1)} K^{O(1)} w^{O(1)}},$$

implying the claimed bound for a choice of $\varepsilon > 0$. ∎

3.2 Publishing Strategy Our publishing strategy is a deterministic strategy that given an input I will produce a list of words from the data structure. However, this will be done in a step-by-step fashion and also ultimately, we will look at the whole picture, taking our random choices during the input generation into account.

Let $\mu_{\boldsymbol{I}}$ be the distribution of inputs defined by our construction; note that the only source of randomness is the branching operation. We assume that we have a deterministic algorithm $\mathcal{A}$ and we would like to lower bound its query time. This is done as follows: first, we generate an input $I \in \boldsymbol{I}$ according to our distribution. Then, this input is given to $\mathcal{A}$ which produces a data structure D. To determine the published bits, we follow the step-by-step procedure by which I was generated and at step i, we will publish a set P_i of the words of the data structure. P_i will be a collection of pairs of the form (a, v) where a is a memory address and v is the value of the memory location; this means that a has $O(\log n)$ bits and v has w bits. The effect of this operation will be that on average (over the random choices of the construction), a lot of the queries will now require one fewer probe (across all of the processors). We use random variable $\boldsymbol{P}_i$ to denote the published words, which is a function of the random variable $\boldsymbol{I}$.

Consider the i-th step just before the branching operation. By assumption, P_{i-1} is already determined in the previous step ($P_0 = \emptyset$). Consider a query q and its simulation. Observe that as long as all K simultaneous memory accesses belong to P_{i-1}, we can simulate the query using the published words only. Let us assume that for some query q, for the first τ units of time, all memory accesses belong to P_{i-1} but at the next time unit, at least one of the memory cells to be accessed lies outside P_{i-1}. In this case, we say that q has *simulation time* τ and we call the tuple of memory cells that lie outside P_{i-1} the *first unpublished probe (FUP)* of q; FUP is represented as the tuple of all the cells that lie outside P_{i-1}.

To describe our publishing strategy, we need one more concept. Let R be a bit range and v be a fixed value assigned to the bit positions in the inclusive prefix of R. Then, a *segment* $s(R, v)$ is the set of all ℓ-bit values that can be obtained by assigning values to the suffix of R (i.e., filling in the remaining bit positions). For a sub-problem S with the defining bit range R_S, a segment $s(R_S, v)$ is called a *basic segment*. For a fixed S, the prefix of R_S has already assigned a fixed value and thus S has $2^{|R_S|}$ many basic segments. The next definitions are with respect to our branching operation done on S which divides R_S into b sub-intervals, $R_1, \cdots, R_b$. Consider two segments $s = s(R_j, v)$ and $s' = s(R_{j'}, v')$, for $1 \leq j' < j$. If s is contained in s', meaning v' is a prefix of v then s' is a *super-segment of* s and the latter is a *sub-segment*. If $j' = j - 1$, then s' is the *parent* segment of s and the latter is a *child* segment.

Consider a segment $s = s(R, v)$. We define a hypergraph $G(s)$ based on the pattern of memory accesses of the K cores. The vertex set of $G(s)$ is the set of memory cells in the data structure. Fix a simulation time τ. We

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

will build different hypergraphs for different values of τ. For every query q of simulation time τ, add its FUP as a hyperedge to $G(s)$. Each query adds at most one edge and also the edges have cardinality at most K. Given a set of vertices H, we define $G_H(s)$ as the subhypergraph of G that is obtained after removing H from every edge in $G(s)$. Also, recall that a matching in a hypergraph is a set of disjoint edges. For a given set H of vertices, let M be a maximum matching in $G_H(s)$. If M contains more than δ_i edges, then we call s a *dense segment (wrt H)*. If there are at most δ_i unique memory cells among all the memory accesses in $G(s)$, then we call s *sparse*. Otherwise, s is *unsuitable*.

Next, we more precisely establish the nature of the set H in the definition of dense graphs. For this, we will assign a *frequent* set to each segment $s(R_j, v)$ in a top-down fashion. Consider a segment $s(R_1, v)$. If the segment is dense or sparse (wrt to the empty set), then its frequent set is the empty set, otherwise, let M be a maximum matching in $G(s)$. Then, we assign M as the frequent set to $s(R_1, v)$. This assigns frequent sets to all the top-level segments. Now consider a segment $s = s(R_j, v)$ with $j > 1$. Let $s' = s(R_{j-1}, v')$ be the parent of s and let H' be the frequent set assigned to s'. We consider the graph $G_{H'}(s)$ and a maximum matching M in it. If M contains more than δ_i edges, then s is said to be dense and we set the frequent set of s to be H'. If there are at most δ_i different memory cells in $G_{H'}(s)$, then s is sparse. Otherwise, s is unsuitable and we define the frequent set of s to be $H' \cup M$.

OBSERVATION 3.2. *Consider the notation in the above paragraph. Let K' be the maximum cardinality of the edges in $G_{H'}(s')$. Then, the edges in $G_{H' \cup M}(s)$ have at most $K' - 1$ vertices.*

Proof. Any maximal matching in $G_{H'}(s)$ intersects all of its edges as otherwise we can add one more edge to the matching. Thus, removing M from $G_{H'}(s)$ will reduce the cardinality of all the edges. The lemma then follows from the observation that s is a child of s' and, therefore, every edge in $G(s)$ also exists in $G(s')$. ∎

COROLLARY 3.1. *Consider a sequence of b segments $s_1 = s(R_1, v_1), \cdots, s_b = s(R_b, v_b)$ such that s_i is a child of s_{i-1}. Then, there can be at most $K - 1$ unsuitable segments in this sequence.*

Proof. After encountering an unsuitable segment, we remove a maximum matching from the hypergraph of the memory accesses which by Observation 3.2 causes the cardinality of edges of the hypergraphs defined by the subsequent segments to decrease by at least one. After $K - 1$ such steps, we will have edges with cardinality one in which case we have essentially vertices, i.e., the unsuitable case cannot happen. ∎

What we publish depends on what the branching operation does at step i. For every value of τ, we consider the aforementioned hypergraphs. Now assume that the operation for a sub-problem S has chosen the segment $s = (R_j, v)$. In this case, we publish all frequent cells assigned to s. Next, if s is sparse, we publish all possible memory accesses in FUP of s. Since there are $t_i = \frac{n}{n_i}$ sub-problems, in total we publish at most $\delta_i t_i bK \log n$ words, over all choices of τ, and taking into account that each frequent set has size at most bK memory cells. This can be upper bounded by $\delta_i t_i w^4 K$. If s is dense, then we use Lemma 2.1 with $N = S(n)$, and all the FUPs of all the queries of simulation time τ over all the sub-problems, with $T = \delta_i$. This means we publish at most $O\left(S(n)\delta_i^{-1/K}\right)$ words which is also upper bounded by $\delta_i t_i w^4 K$, given the choice of our parameters.

3.3 Probe Elimination The following observation gives an alternative view of the branching operation.

OBSERVATION 3.3. *Consider a sub-problem R_S in which R_S is divided into b sub-intervals $R_1, \cdots, R_b$. The distribution created by the branching operation is equivalent to the following: Choose a basic segment s uniformly at random among all the basic segments of S. Then consider the sequence of b segments $s_1 = (R_1, v_1), \ldots, s_b = (R_b, v_b)$ which are the unique ancestors of s. Then, choose a random segment s_j uniformly at random among them.*

We now briefly explain the main intuition behind the probe elimination. We claim that combining Corollary 3.1 with the above observation, we should be able to eliminate a probe. Consider the view in Observation 3.3. The main intuition is that a random matching query can also be selected by first sampling the base segment s. If the chosen segment s_j is sparse, then it follows that the FUP of q is published. The same happens if s_{j+1} is dense. So the only case when q doesn't get its FUP published is when s_j is dense and s_{j+1} is

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

sparse or s_j is unsuitable. But this event happens with probability 1/4. Thus, on average each probe will have its FUP published with some constant probability. We will make this intuition precise, however, to be able to define the FUPs and, consequently, the hypergraphs above, we need to fix a set of published words P_i. But they depend on the entire input I! To fix this issue, we will use the fact that the number of sub-problems is far greater than the number of published bits before the branching operation. We formalize this below.

Consider the beginning of the i-th step where we have t_i sub-problems. Let $\boldsymbol{S}_1, \cdots, \boldsymbol{S}_{t_i}$ be the random variables that represent the final values generated in each sub-problem. In our construction, each $\boldsymbol{S}_i$ is built using a distribution $\mu_{\boldsymbol{S}_i}$ (which are all identical) as outlined by the original statement or an equivalent view of it by Observation 3.3. However, what we are interested in is $\mu_{\boldsymbol{S}_i|\boldsymbol{P}_i=P_i}$ for a fixed choice of published words P_i, i.e., we are looking at a conditional distribution.

To prove the probe elimination lemma, we will first assume that FUPs are *given* (or in fact, they are fixed to be some choices) and that we are dealing with the original distribution $\mu_{\boldsymbol{S}_i}$. Next, we will show that as the number of sub-problems vastly exceeds the number of published bits, the distributions $\mu_{\boldsymbol{S}_i|\boldsymbol{P}_i=P_i}$ and $\mu_{\boldsymbol{S}_i}$ are often very close (they have very small Kullback-Leibler divergence) and, thus, we will show that the analysis goes through with small changes in the constants involved.

Consider a fixed input I and let v be an input value. Among all the matching queries of v, pick one uniformly randomly and let the random variable $\boldsymbol{r}(v)$ denote this randomly chosen query. Let $\boldsymbol{r}(\boldsymbol{v})$ be the corresponding random variable for the random variable $\boldsymbol{v}$. We would like to show that our publishing strategy makes progress in eliminating probes with respect to $\boldsymbol{r}(\boldsymbol{v})$, i.e., with constant probability, at every step we eliminate one probe. Expressing this requires dealing with some technicalities, so we need additional definitions and observations.

Let $\mathbf{C}_{i-1}$ be the random variable that represents the choices of all the branching operations from step 1 to $i-1$. Let $\mathcal{C}_{i-1}$ denote one particular outcome of $\mathbf{C}_{i-1}$. Consider the step i of our publishing strategy. Observe that segment $s = (R_S, v)$ that is a base segment at the start of the step i of the construction might not be a base segment at the end of this step because the defining interval will shrink and we will have different sub-problems. Nonetheless, s will be contained in some other base segment s'. To capture the progress at step i, we define the random variable $X(\boldsymbol{r}(\boldsymbol{v}))$ as follows. Consider the event that $\boldsymbol{r}(\boldsymbol{v}) = q$ for some fixed value q and $\mathbf{C}_{i-1} = \mathcal{C}_{i-1}$. The latter event, determines the base segment s that contains q. Let τ be the designated maximum simulation time of a query in s. Define $X(q) = 1$ if the there exists a query q' in s' such that q' has designated simulation time of τ and its designated FUP is included in P_i. Otherwise, $X(q) = 0$.

Our main lemma is the following.

LEMMA 3.1. *Conditioned on the event that* $\mathbf{C}_{i-1} = \mathcal{C}_{i-1}$, $\sum_{\boldsymbol{v}} \mathbb{E}[X(\boldsymbol{r}(\boldsymbol{v}))] = \Omega(n)$ *where the summation is over all the* n *(random) values* $\boldsymbol{v}$ *in the input.*

Proof. We simply need to show that $\mathbb{E}[X(\boldsymbol{r}(\boldsymbol{v}))] = \Omega(1)$ and the claim will follow by the linearity of expectation. Consider the event that $\boldsymbol{r}(\boldsymbol{v}) = q$ for a fixed value q. We first claim that $\mathbb{E}(X(q)) = \Omega(1)$.

To see this, consider the sub-problem S with bit range R_S that contains q. R_S is divided into smaller intervals $R_1, \cdots, R_b$, according to the branching operation. Consider a base segment s of R_S that contains q and consider the sequence of b segments $s_1 = s(R_1, v_1), \cdots, s_b = s(R_b, v_b)$ such that s_i is a child of s_{i-1} and they are all super-segments of s. By Observation 3.2, the branching operation can be viewed as a process that makes two random choices: the first choice involves selecting a base segment uniformly at random, followed by the second choice that involves selecting its ancestor segment uniformly at random. Crucially, we claim that the conditional distribution of the second choice on the event that $\boldsymbol{r}(\boldsymbol{v}) = q$ is still uniformly random among all the indices 1 to b. To see this, observe that all the bit positions in q are either filled deterministically (via the splitting operation), or filled uniformly at random via either the branching operation or the selection of a random matching query; in other words, all the big positions in R_S that are not determined by the splitting operation are filled uniformly at random and the second choice of branching operation is independent from the event $\boldsymbol{r}(\boldsymbol{v}) = q$.

Let $\boldsymbol{j}$ be the random variable that represents the second choice of the branching operation. The event $\boldsymbol{j} = j$ is a good event in two cases: (i) s_j is sparse, or (ii) $j < b$ and s_{j+1} is dense. Under this good event, we can see that the lemma follows: If s_j is sparse, then by our publishing strategy, we will be publishing all the designated FUPs of all the queries with simulation τ in s_j, which includes all the queries in s and thus $X(\boldsymbol{r}(\boldsymbol{v})) = 1$. For the second case, observe that s_{j+1} will be the base segment s' (in the definition of X) that contains s but also in this case, due to Lemma 2.1 we have published the FUP of at least one query with simulation time τ from s_{j+1} which shows in this case $X(\boldsymbol{r}(\boldsymbol{v})) = 1$.

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

Thus, it remains to calculate the probability of the above good event. Observe that if a segment s_j is sparse, then all the subsequent segments $s_{j+1}, \ldots, s_b$ are also sparse. This means there can be at most one index j' such that $s_{j'}$ is dense but $s_{j'+1}$ is sparse. For all the other non-sparse indices, we can have at most $K-1$ unsuitable segments by Corollary 3.1. This implies that the probability that a good event does not happen is at most K/b which proves the lemma. ∎

The above lemma is essentially the probe elimination lemma. However, we need to get rid of a technical problem where the published bits give the data structure information about the future operations. At the beginning of step i, we have a set P_{i-1} of published words and the FUPs and the simulation times are defined with respect to those. In other words, we do not have "designated" FUPs and simulation times and they in fact depend on the random variable $\boldsymbol{P}_{i-1}$, the published words; the event $\boldsymbol{P}_{i-1} = P_{i-1}$ for a fixed choice of P_{i-1} is what enables us to define these terms. However, we show that if the number of sub-problems is significantly more than the number of published words, then the data structure has essentially no useful information about most of the individual sub-problems, most of the time.

Consider the event $\mathbf{C}_{i-1} = \mathcal{C}_{i-1}$ for a fixed $\mathcal{C}_{i-1}$ and observe that this determines the t_i sub-problems and their defining ranges. Let $\boldsymbol{I} = (\boldsymbol{S}_1, \cdots, \boldsymbol{S}_{t_i})$ be the random variable that describes the final input generated, i.e., $\boldsymbol{S}_i$ represents n_i input values in the i-th sub-problem. These depend on the choices in steps i and beyond. Here, we refer to the properties of the entropy function in Lemma 2.2. The independence of the sub-problems implies that $H(\boldsymbol{I}) = \sum_j H(\boldsymbol{S}_j)$ and since $\boldsymbol{P}_{i-1}$ is a function of $\boldsymbol{I}$ (as our publication strategy is deterministic),

$$H(\boldsymbol{I}, \boldsymbol{P}_{i-1}) = H(\boldsymbol{I}) = \sum_j H(\boldsymbol{S}_j).$$

We now observe that

$$\begin{aligned}\sum_j I(\boldsymbol{S}_j; \boldsymbol{P}_{i-1}) &= \sum_j H(\boldsymbol{S}_j) - H(\boldsymbol{S}_j|\boldsymbol{P}_{i-1}) = H(\boldsymbol{I}) - \sum_j H(\boldsymbol{S}_j|\boldsymbol{P}_{i-1}) \leq H(\boldsymbol{I}) - H(\boldsymbol{S}_1, \cdots, \boldsymbol{S}_{t_i}|\boldsymbol{P}_{i-1}) \\ &= H(\boldsymbol{I}) - H(\boldsymbol{I}|\boldsymbol{P}_{i-1}) = H(\boldsymbol{P}_{i-1}) \leq (w + \log S(n))|P_{i-1}| = O(w|P_{i-1}|).\end{aligned}$$

We can observe that according to our parameters, $t_i = \omega(w|P_{i-1}|)$. Thus, for at least $0.9t_i$ indices j we have that $I(\boldsymbol{S}_j; \boldsymbol{P}_{i-1}) = o(1)$. Then it follows that

$$\sum_{P_{i-1}} \Pr[\boldsymbol{P}_{i-1} = P_{i-1}] \mathrm{D}_{\mathrm{KL}}(\mu_{\boldsymbol{S}_j|\boldsymbol{P}_{i-1}=P_{i-1}} \,||\, \mu_{\boldsymbol{S}_j}) = o(1). \tag{3.2}$$

Consequently, with probability at least 0.9, we have that $\mathrm{D}_{\mathrm{KL}}(\mu_{\boldsymbol{S}_j|\boldsymbol{P}_{i-1}=P} \,||\, \mu_{\boldsymbol{S}_j}) = o(1)$, i.e, most often the distribution $\mu_{\boldsymbol{S}_j|\boldsymbol{P}_{i-1}=P_{i-1}}$ is very close to $\mu_{\boldsymbol{S}_j}$. The proof of our main result relies on the following technical lemma.

LEMMA 3.2. *Let μ_1 and μ_2 be two distributions such that $D_{KL}(\mu_1 \,||\, \mu_2) = o(1)$. Let G be a series of "good" events such that $\mu_2(G) = 0.5$. Let $\mu_1(G) = c_1$. Then, $c_1 \geq c$ for some constant $c > 0$.*

Proof. Let $\lambda = \mathrm{D}_{\mathrm{KL}}(\mu_1 \,||\, \mu_2)$. By the definition of the Kullback-Leibler divergence we can write:

$$\lambda = \sum_x \mu_1(x) \log\left(\frac{\mu_1(x)}{\mu_2(x)}\right) = \sum_x \mu_1(x) \log(\mu_1(x)) - \sum_{x \in G} \mu_1(x) \log(\mu_2(x)) - \sum_{x \notin G} \mu_1(x) \log(\mu_2(x)).$$

Observe that given any values $0 < a_1, a_2, b < 1$, the function $f(x) = a_1 \log(b-x) + a_2 \log(x)$ is maximized when $\frac{a_1}{a_2} = \frac{b-x}{x}$ which in turn implies that $a_1 \log(b_1) + a_2 \log(b_2)$ when $b_1 + b_2$ is restricted to be fixed is maximized when $\frac{a_1}{a_2} = \frac{b_1}{b_2}$. Let $c_2 = 0.5$, $c_2 = \beta c_1$ and $1 - c_2 = \gamma(1 - c_1)$ which implies $\gamma = \frac{1-c_2}{1-c_1}$. Since $\Pr_{\mu_2}[G] = c_2$, by repeatedly applying the above observation it follows that the two latter sums $\sum_{x \in G} \mu_1(x) \log(\mu_2(x)) + \sum_{x \notin G} \mu_1(x) \log(\mu_2(x))$ is maximized which in turn implies λ is minimized when for each $x \in G$, we have $\mu_2(x) = \beta \mu_1(x)$ and for $x \notin G$, we have $\mu_2(x) = \gamma \mu_1(x)$. With these values, we can obtain that

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

$$\lambda \geq \sum_x \mu_1(x)\log(\mu_1(x)) - \sum_{x\in G}\mu_1(x)\log(\mu_1(x)) - \sum_{x\in G}\mu_1(x)\log(\mu_1(x)) - \sum_{x\in G}\mu_1(x)\log(\beta) - \sum_{x\in G}\mu_1(x)\log(\gamma)$$
$$= -c_1\log(\beta) - (1-c_1)\log(\gamma) = -\frac{0.5}{\beta}\log(\beta) - (1-\frac{0.5}{\beta})\log(\gamma).$$

Observe that as β increases, the first term $-\frac{0.5}{\beta}\log(\beta)$ converges to zero, γ converges to 0.5 and thus the term $(1-\frac{0.5}{\beta})\log(\gamma)$ converges to two. Thus, $c_1 \geq c$ for some constant $c > 0$. ∎

We are now ready to prove our main result:

THEOREM 3.1. *The predecessor search problem on an input of n integers of length $\ell = c\log n$ bits, for a constant $c > 1$, has a worst-case query lower bound of $\Omega\left(\log_K(\frac{(\hat{c}-1)\log n}{\log\alpha+\log K+\log w})\right)$ for data structures that use $S(n) \leq \alpha n$ words of space in the K-processor CRCW PRAM model with word size of w bits and $\hat{c} = \min(2, c)$.*

Proof. Before proceeding with the proof, let us quickly recap the arguments so far. Let us assume that we are trying to produce a worst-case query lower bound for data structures that use $S(n) = \alpha n$ space.

We outlined a construction which defines a distribution $\mu(\boldsymbol{I})$ on the inputs. It is a step-by-step construction, which defines the least significant $\hat{c}\log n$ bits of the input integers. At step i, some parts of the bit range of the input integers are filled in and the construction is run for γ steps.

We generate one random input I from this distribution which is given to the algorithm which leads to a data structure $\mathcal{D}$.

Given $\mathcal{D}$, our publishing strategy follows a similar step-by-step approach used in the construction of the input and produces a sequence $P_1, P_2, \cdots, P_\gamma$ where each P_i is a list of published memory cells.

Then, for every input value $v \in I$, we select a random query, $q = \boldsymbol{r}(v)$, among all the matching queries of v. We follow the base segments that contain q and find the query with the maximum simulation time, with respect to P_{i-1}, i.e., the steps of the computation that fully lie within P_{i-1}. In particular, at step i, we find the base segment s_i that contains q and then within that base segment, we find the query q_i with the largest simulation time, say τ_i, in s_i. Let $X_i(q) = 1$ denote the event that the base segment containing q in step $i+1$ contains query with a larger simulation time than τ_i; note that at step $i+1$ the simulation time is considered with respect to P_{i+1}.

To analyze this, we consider not just a fixed input I but the entire random process. In particular, we consider the random variable $X_i(\boldsymbol{r}(\boldsymbol{v}))$ over the random choices of the construction, a random query $\boldsymbol{v}$ and selection of the match query of $\boldsymbol{v}$, $\boldsymbol{r}(\boldsymbol{v})$. Let $\boldsymbol{Y}_i = \sum_{\boldsymbol{v}} X_i(\boldsymbol{r}(\boldsymbol{v}))$. $\boldsymbol{Y}_i$ represents for how many input values we advance during the simulation.

Now consider the choices for steps 1 to $i-1$ represented by $\mathbf{C}_{i-1}$. By the law of total probability, we can write

$$\mathbb{E}[\boldsymbol{Y}_i] = \sum_{\mathcal{C}_{i-1}} \mathbb{E}[\boldsymbol{Y}_i|\mathbf{C}_{i-1} = \mathcal{C}_{i-1}]\Pr[\mathbf{C}_{i-1} = \mathcal{C}_{i-1}]$$

where the summation is over all possible values $\mathcal{C}_{i-1}$. Here, we would like to use Lemma 3.1 and argue that this summation is bounded by $\Omega(n)$ but Lemma 3.1 assumes that we are dealing with designated FUP and simulation times and that the branching operation makes fully random choices, which is clearly not the case here. Observe that the event $\mathbf{C}_{i-1} = \mathcal{C}_{i-1}$ determines the sub-problems in the step i. As before, let $\boldsymbol{S}_1, \cdots, \boldsymbol{S}_{t_i}$ denote the values in these sub-problems. We decompose the sum based on the sub-problems. Let $\boldsymbol{W}_{i,j} = \sum_{\boldsymbol{v}\in\boldsymbol{S}_j} X_i(\boldsymbol{r}(\boldsymbol{v}))$. Observe that $\boldsymbol{Y}_i = \sum_j \boldsymbol{W}_{i,j}$. We once again use the law of total probability but this time on all possible published words in step $i-1$.

$$\sum_{j=1}^{t_i}\mathbb{E}[\boldsymbol{W}_{i,j}|\mathbf{C}_{i-1} = \mathcal{C}_{i-1}] = \sum_{P_{i-1}}\sum_{j=1}^{t_i}\mathbb{E}[\boldsymbol{W}_{i,j}|\mathbf{C}_{i-1} = \mathcal{C}_{i-1}, \boldsymbol{P}_{i-1} = P_{i-1}]\Pr[\boldsymbol{P}_{i-1} = P_{i-1}].$$

Previously, we have proven that for $0.9t_i$ indices j, Eq. (3.2) holds. Call these indices *good*. Also, assuming $\boldsymbol{P}_{i-1} = P_{i-1}$, allows us to define FUP and simulation times for all queries. These are now used as the designated

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited

simulation times and FUP in Lemma 3.1. By Lemma 3.1, given the designated FUPs and simulation times and assuming that we are allowed to use the distributions $\mu_{\boldsymbol{S}_j}$, the maximum simulation time of each query is about to increase by probability at least $1 - b/K = 3/4$, or $\Omega(n)$ queries in expectation over all the n queries in all the sub-problems. By Markov's inequality, there exists a constant ε such that with probability $1/2$, at least εn queries increase their maximum simulation time. Take this as the "good" event in Lemma 3.2. As $\mu_{\boldsymbol{S}_j|\boldsymbol{P}_{i-1}=P_{i-1}}$ is very close to $\mu_{\boldsymbol{S}_j}$, it follows by Lemma 3.2 that this good event has probability $\Omega(1)$ under the distribution $\mu_{\boldsymbol{S}_j|\boldsymbol{P}_{i-1}=P_{i-1}}$. This shows that

$$\sum_{j=1}^{t_i} \mathbb{E}[\boldsymbol{W}_{i,j}|\mathbf{C}_{i-1} = \mathcal{C}_{i-1}] \geq \sum_{P_{i-1}} \sum_{\text{good } j} \mathbb{E}[\boldsymbol{W}_{i,j}|\mathbf{C}_{i-1} = \mathcal{C}_{i-1}, \boldsymbol{P}_{i-1} = P_{i-1}] \Pr[\boldsymbol{P}_{i-1} = P_{i-1}] = \Omega(n).$$

Over all γ steps, we will be simulating the n queries for $\Omega(n\gamma)$ probes, in expectation. If the worst-case query time of the data structure is at most $\varepsilon'\gamma$, by a fixed positive constant probability, we will be able to simulate a fraction of the queries. However, by the parameters of our construction, Observation 3.1, n_γ is at least polylogarithmic, meaning, $t_\gamma \leq \frac{n}{\log^{O(1)} n}$ and also by the construction, t_γ is larger than the number of published bits. But as we can simulate $\Omega(n)$ using only the published bits, this leads to a contradiction. ∎

References

[1] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences (JCSS)*, 65:38–72, August 2002.

[2] O. Berkman and U. Vishkin. On parallel integer merging. *Information and Computation*, 106(2):266–285, 1993.

[3] J. Boninger, J. Brody, and O. Kephart. Non-adaptive data structure bounds for dynamic predecessor. In *Proceedings of the 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[4] A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC)*, pages 338–344, 1982.

[5] A. Brodnik, S. Carlsson, M. L. Fredman, J. Karlsson, and J. I. Munro. Worst case constant time priority queue. *Journal of Systems and Software*, 78(3):249–256, 2005.

[6] T. M. Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[7] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

[8] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the 22nd ACM Symposium on Theory of Computing (STOC)*, pages 1–7, 1990.

[9] Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, 100(10):942–946, 1983.

[10] K. G. Larsen, R. Pagh, G. Persiano, T. Pitassi, K. Yeo, and O. Zamir. Optimal non-adaptive cell probe dictionaries and hashing. In *Proceedings of the 51st International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 297 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 104:1–104:12, 2024.

[11] S. Natarajan Ramamoorthy and A. Rao. Lower bounds on non-adaptive data structures maintaining sets of numbers, from sunflowers. In *Proceedings of the 33rd Computational Complexity Conference (CCC)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2018.

[12] G. Navarro and J. Rojas-Ledesma. Predecessor search. *ACM Computing Surveys (CSUR)*, 53(5):1–35, 2020.

[13] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[14] M. Snir. On parallel searching. In *Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of distributed computing*, pages 242–253, 1982.

[15] L. G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348–355, 1975.

[16] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters (IPL)*, 6:80–82, 1977.

[17] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

[18] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters (IPL)*, 17(2):81–84, 1983.

[19] A. C. C. Yao. Should tables be sorted? *Journal of the ACM (JACM)*, 28(3):615–628, 1981.

Copyright © 2025 by SIAM
Unauthorized reproduction of this article is prohibited