Non-Adaptive Data Structure Bounds for Dynamic Predecessor Search

Joseph Boninger[1], Joshua Brody[1], and Owen Kephart[1]

1 **Swarthmore College, Swarthmore, PA, USA**
jboning1@swarthmore.edu, joshua.e.brody@gmail.com, okephar1@swarthmore.edu

Abstract

In this work, we continue the examination of the role *non-adaptivity* plays in maintaining dynamic data structures, initiated by Brody and Larsen. We consider non-adaptive data structures for predecessor search in the w-bit cell probe model. In this problem, the goal is to dynamically maintain a subset T of up to n elements from $\{1, \ldots, m\}$, while supporting insertions, deletions, and a predecessor query $\textsc{Pred}(x)$, which returns the largest element in T that is less than or equal to x. Predecessor search is one of the most well-studied data structure problems. For this problem, using non-adaptivity comes at a steep price. We provide exponential cell probe complexity separations between (i) adaptive and non-adaptive data structures and (ii) non-adaptive and memoryless data structures for predecessor search.

A classic data structure of van Emde Boas solves dynamic predecessor search in $O(\log \log m)$ probes; this data structure is adaptive. For dynamic data structures which make non-adaptive updates, we show the cell probe complexity is $O\left(\min\{\frac{\log m}{\log(w/\log m)}, \frac{n \log m}{w}\}\right)$. We also give a nearly-matching $\Omega\left(\min\{\frac{\log m}{\log w}, \frac{n \log m}{w \log w}\}\right)$ lower bound. We also give an $\Omega(m/w)$ lower bound for memoryless data structures.

Our lower bound technique is tailored to non-adaptive (as opposed to memoryless) updates and might be of independent interest.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases dynamic data structures, lower bounds, predecessor search, non-adaptivity

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2017.20

1 Introduction

The goal in a dynamic data structure problem is to maintain a database that changes over time, while supporting *queries* and *updates* to this data structure. A natural objective is to support both efficient queries and efficient updates. Often, either one is easily accomplished, but for many dynamic data structure problems, the optimal worst-case runtime on the maximal query/update time is polynomial. Nevertheless, proving such a data structure lower bound appears well beyond our current understanding. In fact, the highest lower bound for *any* dynamic data structure is currently $\Omega((\log n/\log \log n)^2)$ [8, 4, 17]. Identifying a dynamic data structure problem which supports a polynomial number of queries, which has a provably polynomial lower bound on either query or update time is one of the biggest open problems in data structures.

Given the current difficulty in showing large data structure lower bounds, it is natural to ask if one can prove large lower bounds for restricted classes of data structures. Brody and Larsen [2] initiated a study of data structure bounds for *non-adaptive* data structures. In a non-adaptive data structure, the memory cells probed during a query or update are chosen in advance, independent of the contents of those cells. For many data structure problems,

37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2017).
Editors: Satya Lokam and R. Ramanujam; Article No. 20; pp. 20:1–20:12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the optimal solution is indeed non-adaptive. Brody and Larsen [2] showed polynomial lower bounds for a large class of problems, both for when queries are non-adaptive and when updates are non-adaptive. It is worth noting that their lower bounds for non-adaptive updates were under an even more severe restriction called *memoryless updates*.

1.1 The Cell Probe Model.

In the cell probe model defined by Yao [18], a data structure consists of a series of memory *cells* or *words*, each storing a w-bit integer. We assume there are at most 2^w cells in the data structure, so that each cell can be addressed using a single w-bit integer. When a query or update is executed, a number of cells are *probed*. During a query or update, which cell is probed next may depend arbitrarily on previous computation, and similarly what is written to a cell during an update may also depend arbitrarily on previous computation in the update. We define the *query complexity* of a data structure, denoted t_q, as the maximum number of cells probed during a query. Similarly, the *update complexity*, denoted t_u, is the maximum number of cells probed during an update. As mentioned previously, it is often easy to design data structures that have either low t_q or low t_u; our goal is to minimize the *cell probe complexity* of a data structure, defined as $\max\{t_q, t_u\}$.

Cell probe complexity measures only the number of memory accesses used by the data structure. All other computation is not counted and essentially given for free. In particular, no assumption is made about what CPU computations are allowed. The generality of the cell probe model makes it ideally suited for studying lower bounds; these lower bounds will apply to other computation models which make more CPU assumptions.

1.2 Non-Adaptive Data Structures

Given the current barriers in proving high cell probe lower bounds for dynamic data structures, it makes sense to study restricted classes of data structures. Non-adaptivity is a natural restriction. We examine several different kinds of non-adaptivity, listed below.

- **Non-Adaptive Query Algorithm.** A cell probe data structure has a non-adaptive query algorithm if the cells probed when answering a query depend only on the query itself, and not on the contents of previously probed cells.
- **Non-Adaptive Update Algorithm.** A cell probe data structure has a non-adaptive update algorithm if the cells probed when answering an update depend only on the update itself, and not on the contents of previously probed cells.
- **Memoryless Update Algorithm.** A cell probe data structure has a memoryless update algorithm if the update algorithm is non-adaptive, and additionally the contents written to a cell depend only on the current contents of the cell and the update itself, and not on contents of other cells previously probed during the update operation.

A *non-adaptive data structure* is a cell probe data structure that has non-adaptive queries and updates. Similarly, a *memoryless data structure* is a cell probe data structure that has non-adaptive queries and memoryless updates.

The study of non-adaptivity in data structures is of both theoretical and practical interest. Our original motivation in examining non-adaptive data structures was to generalize the class of non-adaptive structures for which we can prove lower bounds, with an eye towards proving polynomial lower bounds on dynamic data structures. This work expands the range of non-adaptive data structures for which we can prove strong lower bounds.

From a more practical perspective, there are many real-world computational settings where the number of memory accesses does not capture the true performance of a data structure or algorithm. Instead, the computational bottleneck is the *rounds of communication* between disk and memory (e.g. the external memory or I/O model [1, 16, 15]) or between different processors (distributed/parallel computation [6, 7, 13]) This is directly related to how adaptive the algorithm or data structures are. We anticipate that a better understanding of the role non-adaptivity plays in data structures will shed further insight into data structures for other models of computation as well.

1.3 Predecessor Search and Our Results

In this work, we primarily focus on two dynamic data structure problems: Predecessor and Max. In the Predecessor data structure problem, we are to maintain a dynamic set of up to n elements $T \subseteq [m]$[1], initially empty, with updates $\text{Insert}(i), \text{Delete}(i)$ which insert/delete i from T. Each query $\text{Pred}(i)$ should return the largest $x \in T$ that is less than or equal to i. In the Max problem, we again maintain a dynamic set $T \subseteq [m]$ with the same insert and delete operations. Additionally, there is a single query Max(), which must return the largest element in T.

In either problem, we assume that each element of the universe $[m]$ can fit into a single cell of memory; i.e., we assume that $w \geqslant \log m$. We make a further reasonable assumption that $w \leqslant m^{0.25}$.

Predecessor search is one of the most well-studied data structure problems, and it deserves a complete study. Moreover, it is a common component in other data structure problems such as binary search trees. In this way, our lower bounds are likely to apply to other data structure problems as well.

As mentioned previously, the van Emde Boas tree [14] is an adaptive data structure for Predecessor with cell probe complexity $O(\log \log m)$. Our main result shows that adaptivity is crucial for such an upper bound.

▸ **Theorem 1.** *Let $\alpha := \min\{n, w/2\}$. Then, any non-adaptive data structure solving* Predecessor *with $t_u = O(\log m)$ must satisfy*

$$t_q \geqslant \frac{\alpha \log m}{2w \log(w \cdot t_u)} \ .$$

▸ **Corollary 2.** *Fix any non-adaptive data structure for the dynamic predecessor problem with query time t_q, and update time t_u. If $n \geqslant w/2$ then we have*

$$\max(t_q, t_u) = \Omega\left(\frac{\log m}{\log w}\right) \ .$$

If $n < w/2$, then we have

$$\max(t_q, t_u) = \Omega\left(\frac{n \log m}{w \log w}\right) \ .$$

Is our non-adaptive lower bound the best possible? Our next result shows that it is close to optimal.

▸ **Theorem 3.** *There exists a non-adaptive data structure for* Predecessor *with $t_q, t_u = O\left(\frac{\log m}{\log(w/\log m)}\right)$.*

[1] We use $[m]$ to denote the set $\{1, \ldots, m\}$, and $[y, z]$ to denote the set of integers $\{y, y+1, \ldots, z\}$.

When $w = \Theta(\log m)$ and n is large, our non-adaptive data structure bounds are loose by a factor of $\log \log m$. When n is large and $w \geqslant (\log m)^{1+\Omega(1)}$, our bounds are tight. For smaller values of n, our non-adaptive data structure is suboptimal, as the trivial data structure which stores an array of up to n values and probes the entire data structure on each operation uses $n \log(m)/w$ words. In this case, our lower bound is off by a factor of $\log w$.

Either way, Theorem 1 can be seen to interpolate between our non-adaptive data structure from Theorem 3 and the trivial upper bound, allowing for a smooth tradeoff between n and m.

Finally, we give a very strong lower bound for memoryless data structures solving MAX. Note that MAX is essentially PREDECESSOR with a further restriction that Pred(m) is the only query allowed, so this lower bound holds for PREDECESSOR as well.

▸ **Theorem 4.** *Any memoryless data structure for* MAX *must have* $\max\{t_q, t_u\} \geqslant m/w$.

This lower bound is much higher than the cell probe complexity of the first trivial solution and is a clean illustration of what is impossible to do with memoryless data structures.

1.4 Previous Results

The study of dynamic data structures, and data structures for predecessor search, has a long history; here we give a brief synopsis. Yao [18] introduced the cell probe model and gave cell probe bounds for the membership problem. Fredman and Saks [5] created the chronogram technique and showed $\Omega(\log(n)/\log\log(n))$ bounds for several dynamic data structure problems, including partial sums. This remained the highest lower bound for any dynamic data structures problem until Pătraşcu and Demaine gave an $\Omega(\log n)$ bound for dynamic connectivity. Pătraşcu and Thorup [10, 11] gave strong deterministic and randomized lower bounds for static predecessor search. Pătraşcu also developed an exciting line of attack for dynamic data structure lower bounds by connecting several conjectured hard problems to a communication problem called Multiphase [9]. Pătraşcu conjectured a strong communication lower bound for Multiphase and showed that this lower bound implies polynomial lower bounds for several dynamic data structures. Chattopadhyay et al. [3] disproved the strongest of Pătraşcu's conjectures for Multiphase, but not in a way which invalidates the implication for polynomial data structure lower bounds.

The highest lower bounds to date for any dynamic data structure problem is the bound $\Omega((\log(n)/\log\log(n))^2)$ of Larsen [8] for dynamic range counting. A similar lower bound was later given by Clifford et al. [4] for Matrix Vector Multiplication and Pătraşcu's Multiphase problem.

Brody and Larsen [2] initiated the study of non-adaptive bounds for dynamic data structures and showed polynomial lower bounds for a number of problems including Multiphase. The techniques in Chattopadhyay et al. [3], which preceded [2], give the same lower bounds. Neither of these works showed lower bounds for non-adaptive (but not memoryless) updates.

▸ Remark. Recently Ramamoorthy and Rao [12] independently proved non-adaptive data structure bounds for Predecessor Search. Ramamoorthy and Rao consider a similar set of secondary problems (median, minimum vs. maximum) and obtain similar bounds, showing for predecessor search that either $t_q \geqslant \frac{\log m}{\log w + \log\log m}$ or $t_u \geqslant \Omega\left(\frac{t_q m^{1/(2t_q+2)}}{\log m}\right)$. Our bounds are stronger when $t_q = \Omega(\frac{\log m}{\log\log m})$. Additionally, our analysis and bounds provide tradeoffs between the universe size m, word size w, and maximum number of items in the set n; Ramamoorthy and Rao consider only the first two parameters. In both works, the lower

bounds for predecessor search apply even when insertions are the only updates. Ramamoorthy and Rao only require queries to be non-adaptive; we require both queries and insertions to be non-adaptive.

Both papers have been developed independently and in parallel.

1.5 Our Technique

For the non-adaptive data structure lower bound (Theorem 1), we prove something stronger than is stated in the theorem. Specifically, we show that there must be a large set of cells C along with a set $A \subseteq [m]$ that is reasonably large such that for each $i \in A$, Pred(i) queries every cell in C. We build this set C iteratively by using the pigeonhole principle along with an encoding argument.

For complete details, see Section 2; we give a high-level sketch here. We begin with $C := \varnothing$ and $A = [m]$, and consider the first update Insert(1). This update has the potential to affect every query. Furthermore, for any operation, the set of cells probed are fixed and chosen in advance. Therefore, for each $i \in A$, the cells probed by Pred(i) and by Insert(1) must intersect. Insert(1) probes at most t_u cells, so by the pigeonhole principle, there must be some cell c that is probed by Insert(1) and by at least m/t_u queries. We fix that cell and set $A := \{i : \text{Pred}(i) \text{ probes } c\}$.

Next, we claim that if C is not too large and A is not too small, then there must be some $j \in A$ and a not-too-small fraction of A such that Insert(j) and Pred(i) intersect *outside of* C. This claim, formalized in Lemma 9, is nontrivial and is our main technical tool. We prove this using an encoding argument—essentially, we show that if there was no such update, then C would contain at least $\approx \log(m)^2$ bits of information, contradicting the assumption that $|C|$ is small. From here we apply another pigeonhole argument to show that there must be some cell outside of c that is probed by a large enough fraction of the remaining queries. Alternating applications of the pigeonhole argument and our main technical lemma allows us to grow C iteratively, up to a size of $|C| \geqslant \frac{\alpha \log m}{2w \log(w \cdot t_u)}$.

Theorems 3 and 4 use more standard techniques. Our $O(\log m)$ non-adaptive data structure uses range trees, and our $\Omega(m/w)$ lower bound on memoryless data structures uses a direct encoding argument—we're able to use a memoryless data structure for Max to encode an arbitrary subset of $[m]$, requiring m/w words.

Before getting into the lower bound proofs, we summarize the encoding arguments common to data structure lower bounds.[2]

1.5.1 The Coding Lower Bound.

In a typical encoding argument, an encoder is tasked with communicating information (say, an element $x \in \mathcal{S}$ from a finite set) to a decoder. The encoder can encode this information in any number of ways, as long as the decoder can unambiguously recover the information. In order for the decoder to recover the encoder's input, the encoder must at a minimum send a different message for each distinct input.

When applying encoding arguments to show data structure lower bounds, the encoder uses a data structure to encode an arbitrary element from $\mathcal{S}$. If the data structure was unreasonably efficient, then the length of the encoding would be too short for the decoder to

[2] Encoding arguments are often phrased in terms of input distributions and Shannon entropy. In this work, we focus on deterministic data structure bounds, and simplify the Coding Lower Bound accordingly.

recover the input without error. We conclude that the data structure cannot be *too efficient*. This intuition is formalized in the definition and fact below.

▸ **Definition 5** (Encoding Procedure)**.** An *encoding procedure* for a finite set $\mathcal{S}$ is a pair of functions $\text{ENC} : \mathcal{S} \to \{0,1\}^k, \text{DEC} : \{0,1\}^k \to \mathcal{S}$ such that for all $x \in \mathcal{S}$, we have

$$\text{DEC}(\text{ENC}(x)) = x \ .$$

The *length* of the encoding is k.

The key feature of an encoding procedure is that the encoder must send a different message for each element of $\mathcal{S}$. Otherwise, the decoder cannot decode without error.

▸ Fact 6. [Coding Lower Bound]

In any encoding procedure for a finite set $\mathcal{S}$, the length of the encoding must satisfy $k \geqslant \log(|\mathcal{S}|)$.

1.6 Roadmap

We prove Theorems 1, 3, and 4 in Sections 2, 3, and 4 respectively, and introduce notation relevant for each theorem in the relevant sections.

2 Non-Adaptive Lower Bound for Predecessor

For our non-adaptive lower bound, it is helpful to work with a more symmetric "wrap-around" variation of the standard Predecessor problem. In this variation, we define $\text{Pred}(i)$ to be equal to

1. the largest $x \leqslant i$ in T, if such an element exists,
2. the largest $x \in T$, if T is nonempty but contains no elements $\leqslant i$, or
3. $\perp$ if T is empty.

It is easy to see that this variation affects the cell probe complexity by at most a factor of 2. We resist notation expansion and in this section use Predecessor to denote this wrap-around variation. The symmetry of this version of Predecessor will be useful because each update has the potential to affect any query.

For this lower bound, we will need to compare the sets of cells probed by different updates and queries. It will be helpful to introduce some notation to make this argument easier to express.

For any $i, j \in [m]$, we let u_j and q_i denote $\text{Insert}(j)$ and $\text{Pred}(i)$ respectively. By convention, we use a subscript j to refer to updates and i to refer to queries. We use U_j and Q_i to denote the set of cells probed by u_j, q_i respectively. We'll also abuse notation a bit and use $A \subseteq [m]$ to denote both a subset of indices and the corresponding subset of queries or updates.

▸ **Theorem 7.** *[Restatement of Theorem 1] Let $\alpha := \min\{n, w/2\}$. Any non-adaptive data structure solving dynamic predecessor with update time $t_u = O(\log m)$, and query time t_q must satisfy*

$$t_q \geqslant \frac{\alpha \log m}{2w \log(w \cdot t_u)} \ .$$

▸ **Corollary 8.** *Fix any non-adaptive data structure for the dynamic predecessor problem with query time t_q, and update time t_u. If $n \geqslant w/2$ then we have*

$$\max(t_q, t_u) = \Omega\left(\frac{\log m}{\log w}\right) \ .$$

If $n < w/2$, then we have

$$\max(t_q, t_u) = \Omega\left(\frac{n \log m}{w \log w}\right) .$$

The proof of Theorem 1 depends on the following technical lemma, whose proof we defer to Subsection 2.1.

▸ **Lemma 9** (Main Technical Lemma). *Let C be a set of cells in the data structure, and let $A \subseteq [m]$. If*

1. *$|A| \geqslant \sqrt{m}$,*
2. *$|C| \leqslant \frac{\alpha \log m}{5w}$, and*
3. *for all $i \in A$, q_i probes all cells in C,*

then there exists $j \in A$ and a subset $A' \subseteq A$ such that $|A'| \geqslant \frac{|A|}{w^2}$ and for each $i \in A'$ there is a cell $c \notin C$ such that u_j and q_i both probe c.

At a high level, this lemma says that if we have a large enough set of queries A and a small enough set of cells C such that each query in A probes each cell in C, then there must be an update u_j that has a nontrivial intersection *outside of* C with a large subset of A.

Proof of Theorem 1. We prove this theorem by induction. Fix an arbitrary non-adaptive data structure for Predecessor. As mentioned in the introduction, we'll prove this theorem by iteratively growing a large set of cells C in the data structure and a not-too-small set of queries A such that each query in A probes each cell in C. If we can grow the set of cells until $|C| = \frac{\alpha \log m}{2w \log(w \cdot t_u)}$ while keeping the set of queries nonempty, the theorem will follow.

This intuition is captured by the following inductive claim.

▸ Claim 10. For all integers $1 \leqslant k \leqslant \frac{\alpha \log m}{2w \log(w \cdot t_u)}$, there is a set of k cells C and a set queries $A \subseteq [m]$ such that

1. $|A| \geqslant \frac{m}{w^{2(k-1)} t_u^k}$
2. $C \subseteq Q_i$ for all $i \in A$.

Setting $k = \frac{\alpha \log m}{2w \log(w \cdot t_u)}$ proves the theorem.

It remains to prove the claim. First, we prove the base case of $k = 1$. Fix an arbitrary update u_j, and note that U_j must intersect Q_i for each $i \in [m]$. Otherwise, the contents of the cells queried by q_i would be the same for the empty set and for $T = \{j\}$, but $\text{Pred}(i) = \perp$ when the set is empty, and $\text{Pred}(i) = j$ when the set is $\{j\}$. Note also that $|U_j| \leqslant t_u$, so by the pigeonhole principle, there must be a cell $c \in U_j$ probed by at least m/t_u queries $i \in [m]$. Fix this cell c, define $C := \{c\}$, and let A be the set of queries that probe c. This set of cells C and queries A fit the premise of Claim 10, completing the base case.

For the induction hypothesis, assume Claim 10 holds for some arbitrary $k < \frac{\alpha \log m}{2w \log(w \cdot t_u)}$.

In the induction step, we'll show that Claim 10 holds for $k+1$ as well. By the induction hypothesis, there is a set of k cells C_k and queries A_k such that $|A_k| \geqslant m/(w^{2(k-1)} t_u^k)$ and $C_k \subseteq Q_i$ for all $i \in A$. To invoke Lemma 9, $|A_k|$ must be at least $\sqrt{m}$. This holds as long as $k \lesssim \frac{\log(m)}{2 \log(w^2 t_u)}$, which is valid since $\alpha \leqslant w/2$.

By Lemma 9, there is an update $j \in A_k$ and subset $A'_k \subset A_k$ such that $|A'_k| \geqslant |A_k|/w^2$ and for each $i \in A'_k$ there is a cell $c \notin C_k$ such that u_j and q_i both probe c. Next, we again use the pigeonhole principle. Since $|U_j \backslash C| \leqslant |U_j| \leqslant t_u$, there must be a cell $c \in U_j \backslash C$ and a set $A''_k \subseteq A'_k$ such that $|A''_k| \geqslant |A'_k|/t_u$ and such that for each $i \in A''_k$, Q_i probes c. Set $C_{k+1} := C \cup \{c\}$ and $A_{k+1} := |A''_k|$. Note that $|A_{k+1}| \geqslant |A_k|/w^2 t_u$ and that $C_{k+1} \subseteq Q_i$ for all $i \in A_{k+1}$. The sets C_{k+1}, A_{k+1} fit the premise of Claim 10 for $k+1$, completing the induction step. ◂

2.1 Proof of Main Technical Lemma

We prove Lemma 9 using an encoding argument—we show that if the lemma is false, then we can use C to encode more than $|C| \cdot w$ bits of information, a contradiction.

Before delving into the technical details of the proof, we introduce some notation. Say that a set of cells C *satisfies* (u_j, q_i) if $U_j \cap Q_i \subseteq C$; that is, if C contains all cells probed by both u_j and q_i. Similarly, for a set $T \subseteq [m]$, say that C satisfies (T, q_i) if C satisfies (u_j, q_i) for all $j \in T$. Lemma 9 states that there is $j \in A$ and a large subset $A' \subseteq A$ (with $|A'| \geqslant |A|/w^2$) such that for all $i \in A'$, C *fails* to satisfy (u_j, q_i).

Proof of Lemma 9. Towards a contradiction, assume that for all $j \in A$, there are less than $|A|/w^2$ queries $i \in A$ such that the given set of cells C fails to satisfy (u_j, q_i). We'll then use the data structure and C to encode the following set:

$$\mathcal{S} := \{T \subseteq A : |T| = \alpha \text{ and } |j - j'| \geqslant \frac{|A|}{w} \text{ for all } j, j' \in T\} .$$

$\mathcal{S}$ is the set of all possible "spread-out" subsets of A with size α.

▸ Claim 11. $|\mathcal{S}| \geqslant 2^{\frac{\alpha \log(m)}{4}}$.

Proof. We construct a subset of $\mathcal{S}$ with the desired size. Let $x_1, \ldots, x_\alpha$ be arbitrary elements of $\{1, \ldots, |A|/w\}$. Set $y_i := \frac{(2i-1)|A|}{w} + x_i$, and set $T := \{y_i\}$. Note that $y_1 > \frac{|A|}{w}$, $y_\alpha \leqslant \frac{(2\alpha-1)|A|}{w} + \frac{|A|}{w} = \frac{2\alpha|A|}{w} \leqslant |A|$, and that by definition of T we have

$$\frac{2i-1}{w}|A| < y_i \leqslant \frac{2i}{w}|A| = \frac{2(i+1)-1}{w}|A| - \frac{|A|}{w} \leqslant y_{i+1} - \frac{|A|}{w} .$$

This means that $y_{i+1} - y_i \geqslant \frac{|A|}{w}$ for all i, hence T is a valid element of $\mathcal{S}$. There are $\frac{|A|}{w}$ choices for each x_i, and α elements of T, so there are $(|A|/w)^\alpha$ choices for T. Thus, we have

$$|\mathcal{S}| \geqslant \left(\frac{|A|}{w}\right)^\alpha = 2^{\alpha \log(|A|/w)} \geqslant 2^{\frac{\alpha}{4}\log(m)} ,$$

where the final inequality holds because $w \leqslant m^{1/4}$ and $|A| \geqslant \sqrt{m}$. ◂

Encoding Procedure. Given an arbitrary $T \in \mathcal{S}$, the encoder takes the non-adaptive data structure, initially storing an empty set. She then inserts each $j \in T$. After performing all insertions, the encoder sends the contents of each cell in C.

Decoding Procedure. The decoder first takes the non-adaptive data structure, initialized to store the empty set. Then, she overwrites the contents of each cell in C using the encoder's message. The decoder then executes q_i for each $i \in A$ and outputs the set of all elements that appear at least $\frac{|A|}{2w}$ times as answers; that is, the decoder returns the set $T' := \{j \in A :$ there are at least $\frac{|A|}{2w}$ elements i with Query(i) == $j\}$.

Analysis. It is easy to see that the length of the encoding is $w \cdot |C| \leqslant \frac{\alpha \log(m)}{5}$ bits, since the encoder sends the memory contents of each cell in C. Next, we claim that the decoder correctly recovers T. By assumption, we have that for all $j \in A$, the set of cells C satisfies (u_j, q_i) for all but at most $\frac{|A|}{w^2}$ queries. Therefore, for any $T \in \mathcal{S}$, C satisfies (T, q_i) for all but at most $\frac{|A|}{w^2}\alpha < \frac{|A|}{2w}$ queries $i \in A$.

Now, consider what happens when C satisfies (T, q_i). For any $j \in T$, C contains all cells probed by both u_j and q_i. Since this holds for all $j \in T$, C contains all cells that changed during insertions *that were probed by* q_i. Thus the decoder can correctly compute q_i when C satisfies (T, q_i).

When C does not satisfy (T, q_i), then the decoder is not guaranteed to correctly compute q_i; we assume without loss of generality that this is an error. The decoder executes query q_i for each $i \in A$, but computes this query incorrectly whenever C does not satisfy (T, q_i). Moreover, since the decoder does not know T in advance, she cannot know a priori which queries failed. We claim that because less than $\frac{|A|}{2w}$ queries are not satisfied, the decoder still has enough information to recover T.

To see this, take any $j \in T$. By construction, $|j - j'| \geqslant \frac{|A|}{w}$ for any $j, j' \in T$. Hence j is the correct answer to query q_i for all $i \in [j, j + \frac{|A|}{w} - 1]$. Even if all errors were in this range, there would still be more than $\frac{|A|}{2w}$ queries for which the decoder correctly computes j. Hence, the decoder will place $j \in T'$. Conversely, consider any $j \notin T$. Then, j is not a correct answer for any query. In the worst case, the decoder computes j for each possible query on which she errs. Since there are less than $\frac{|A|}{2w}$ such queries, the decoder will not place $j \in T'$. The decoder adds j to T' if and only if $j \in T$, hence the decoder correctly outputs T.

We've shown how to encode an arbitrary $T \in \mathcal{S}$ using $w \cdot |C|$ bits. By Fact 6 and Claim 11, we must have

$$w \cdot |C| \geqslant \log(|\mathcal{S}|) \geqslant \frac{\alpha \log m}{4} .$$

Therefore, we must have $|C| \geqslant \frac{\alpha \log(m)}{4w}$, contradicting our assumption that $|C| \leqslant \frac{\alpha \log m}{5w}$. ◂

3 Non-Adaptive Upper Bound

In this section, we give an $O(\frac{\log m}{\log(w/\log m)})$ non-adaptive upper bound for Predecessor by using a form of range tree.

Proof of Theorem 3. We first handle the case where $w = \lceil \log m \rceil$, so each cell stores a single element from the universe $[m]$. Then, we adjust the construction to handle $w \gg \log m$.

Let k be the least integer such that $2^k \geqslant m$. Our data structure consists of a complete binary tree with 2^k leaves, labeled $1, \ldots, 2^k$. At each node v in the tree, we store the largest i such that (i) $i \in T$ and (ii) i is a descendant of v. If no such i exists, we store $\perp$. Note that each leaf i stores either i or $\perp$, and that the root node stores the maximal element of T. Additionally, an interior node v with children l, r stores the maximum of what is contained in the cells of l, r, treating $\perp$ as 0. In other words, $\max(v) := \max\{\max(l), \max(r)\}$.

To execute Insert(i), for each node v on the path from i to the root (including leaf i), the data structure checks to see if i is now the largest element among descendants of v and updates appropriately if so. Note that the set of nodes probed corresponds to all nodes on the path from leaf i to the root. This is fixed in advance, so Insert(i) is indeed a non-adaptive update.[3]

Implementing Delete(i) is similar. Using the invariant that a the cell corresponding to node v maintains the max of whatever is stored in its children, the data structure must query both children of node v. This must happen for each node v on the path from leaf i up to the root, resulting in twice as many cell probes as an insert. However, as with insertions, *which* cells to probe are known in advance, so the data structure remains non-adaptive. Unlike insertions, these updates are not memoryless, since updating node v depends on the deletion, the current contents of the cell, and the contents of both children.

[3] In fact, insertions in this data structure are memoryless, since each cell update depends only on the insertion and the current contents of the cell.

To implement Pred(i), we traverse the path from the root down to leaf i. Each time we take the right child, the data structure queries the cell corresponding to the left child. We also query the node corresponding to leaf i, and return the maximal element found, or $\perp$ if all queried cells returned $\perp$. In this way, the range $\{1, \ldots, i\}$ is partitioned into a series of subranges, with at most one subrange per level of the binary tree. This sketch describes the query algorithm as walking down the tree, but the nodes that are queried depend only on i itself, and so they can be again chosen in advance.

Insertions, deletions, and queries can all be performed non-adaptively by querying at most two cells per level of the tree. The tree has size $2^k < 2 \cdot m$, so the height is at most $k = O(\log m)$. Since each operation probes at most two cells per level of the range tree, the query complexity is also $O(\log m)$.

Finally, suppose that $w \gg \log m$. In this case, we can pack $w/\log m$ elements into a single word. Fix h such that $2^h = w/\log m$, so that $h = \log(w/\log m)$. We modify the original range tree argument to pack subtrees of height h into a single cell. So, for example, one cell stores the root value and all values of nodes less than h away from the root. For each node at level h of the range tree, we store in a single cell all descendants at distance less than h from this node, and so on. Insertions, deletions, and queries happen as before, but the w-bit memory cell containing the value stored at each node is probed as opposed to the node itself. We still probe at most 2 cells per level, but this time, our "cells" consume h levels of the original range tree. As a result, our new query complexity is $O\left(\frac{\log m}{h}\right) = O\left(\frac{\log m}{\log(w/\log m)}\right)$. ◂

4 Memoryless Lower Bound for Predecessor

Our final result is a strong lower bound for the cell probe complexity of memoryless data structures that solve Predecessor. In fact, our result is a lower bound for a simpler problem Max. Max easily reduces to Predecessor, so the lower bound applies to both problems.

Proof of Theorem 4. The proof is a simple encoding argument. Let the encoder be given a set $T \subseteq [m]$, and let $\mathcal{D}$ be a memoryless data structure for Max.

The encoder encodes T by first preprocessing a copy of $\mathcal{D}$ and then inserting each element in T one at a time into $\mathcal{D}$. She then writes the contents of each cell probed by Max. Call these cells C_{Max}. Because the query algorithm is non-adaptive, C_{Max} can be determined without knowledge of their contents and will not change regardless of any updates that occur.

The decoding protocol is as follows. The decoder preprocesses her own copy of $\mathcal{D}$ and initializes $T' = \varnothing$. Then, she writes to the cells in C_{Max} using the encoding provided by the encoder. The decoder then performs the following until $k = 0$:

1. Run Max() on $\mathcal{D}$. Let k be the value returned by Max().
2. If $k = \perp$, the decoder ends the decoding algorithm and outputs T'.
3. If $k > 0$, the decoder adds k to T', emulates Delete(k) on $\mathcal{D}$, and repeats the process.

Note that the decoder cannot completely execute the Delete(k) operations, but does not need to. Since queries and updates are non-adaptive, she knows which cells get probed by Delete(k). Additionally, since the updates are memoryless, the contents of each cell written by Delete(k) are a function of Delete(k) and the current contents of the cell. The decoder only needs to maintain the cells probed by Max(). Since she is given the initial contents of these cells by the encoder, and since she knows which update operations to perform, she

can maintain the contents of the cells probed by Max(). The decoder might not be able to maintain cells outside of C_{Max} that are probed by updates, but she does not need to, since these cells are not queried by Max(). By repeating this process as long as Max returns nonzero elements, the decoder can recover all of T.

We now analyze the length of the encoding to determine a lower bound on t_q. The encoder sends w bits for each cell in C_{Max}. Since $|C_{\text{Max}}| = t_q$ by definition, the length of the encoding is wt_q. The encoding is for an arbitrary subset $S \subseteq [m]$, so by the coding lower bound, any encoding must be at least m bits long. Thus, we get $wt_q \geqslant m$, hence $t_q \geqslant m/w$. ◂

References

1. Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. Joshua Brody and Kasper Green Larsen. Adapt or die: Polynomial lower bounds for non-adaptive dynamic data structures. *Theory of Computing*, 11(19):471–489, 2015. URL: `http://www.theoryofcomputing.org/articles/v011a019`, `doi:10.4086/toc.2015.v011a019`.
3. Arkadev Chattopadhyay, Jeff Edmonds, Faith Ellen, and Toniann Pitassi. Upper and lower bounds on the power of advice. *SIAM Journal on Computing*, 45(4):1412–1432, 2016.
4. Raphael Clifford, Allan Grønlund, and Kasper Green Larsen. New unconditional hardness results for dynamic and online problems. In *Proc. 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1089–1107. IEEE, 2015.
5. Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st ACM ACM Symposium on Theory of Computing (STOC)*, pages 345–354. ACM, 1989.
6. Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 513–522. ACM, 2010.
7. Fabian Kuhn and Rotem Oshman. Dynamic networks: models and algorithms. *ACM SIGACT News*, 42(1):82–96, 2011.
8. Kasper Green Larsen. The cell probe complexity of dynamic range counting. In *Proc. 44th ACM Symposium on Theory of Computing (STOC)*, pages 85–94, 2012.
9. Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proc. 42nd ACM Symposium on Theory of Computing (STOC)*, pages 603–610, 2010.
10. Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.
11. Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *Proc. 18th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 555–564, 2007.
12. Sivaramakrishnan Natarajan Ramamoorthy and Anup Rao. Non-adaptive data structure lower bounds for median and predecessor search from sunflowers. https://eccc.weizmann.ac.il/report/2017/040/, 2017.
13. Nir Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, 2011.
14. Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. 16th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

15 Elad Verbin and Qin Zhang. The limits of buffering: a tight lower bound for dynamic membership in the external memory model. *SIAM Journal on Computing*, 42(1):212–229, 2013.

16 Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

17 Omri Weinstein and Huacheng Yu. Amortized dynamic cell-probe lower bounds from four-party communication. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 305–314. IEEE, 2016.

18 Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM (JACM)*, 28(3):615–628, 1981.