Optimal Bounds for the Predecessor Problem and Related Problems

Paul Beame*
Computer Science and Engineering
University of Washington
Seattle, WA, USA 98195-2350
beame@cs.washington.edu

Faith E. Fich†
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4
fich@cs.utoronto.ca

December 11, 2001

Abstract

We obtain matching upper and lower bounds for the amount of time to find the predecessor of a given element among the elements of a fixed compactly stored set. Our algorithms are for the unit-cost word RAM with multiplication and are extended to give dynamic algorithms. The lower bounds are proved for a large class of problems, including both static and dynamic predecessor problems, in a much stronger communication game model, but they apply to the cell probe and RAM models.

1 Introduction

Many problems in computer science involve storing a set S of integers and performing queries on that set. The most basic query is the *membership query*, which determines whether a given integer x is in the set. A *predecessor query* returns the predecessor $pred(x,S)$ of x in S, that is, the largest element of the set S that is less than x. If there is no predecessor (which is the case when x is smaller than or equal to the minimum element of S), then a default value, for example, 0, is returned.

Predecessor queries can be used to efficiently perform nearest neighbour queries (i.e. find the element of S that is closest to the given integer x). They can also be used to determine the rank of an element x with respect to S (i.e. the number of elements in S less than or equal to x), even though this information is stored only for elements of S.

The *static dictionary* problem is to store a fixed set and perform membership queries on it; the *static predecessor* problem allows predecessor queries. If insertions and deletions may also be performed on the set, we have the *dynamic dictionary* problem and the *dynamic predecessor* problem, respectively.

The complexities of searching (for example, performing membership or predecessor queries) and sorting are well understood under the assumption that elements are abstract objects which may only be compared. But many efficient algorithms, including hashing, bucket sort, and radix sort, perform word-level operations, such as indirect addressing using the elements themselves or values derived from them.

*Research supported by the National Science Foundation under grants CCR-9303017 and CCR-9800124

†Research supported by grants from the Natural Sciences and Engineering Research Council of Canada and Communications and Information Technology Ontario. Part of the research was conducted while visiting Laboratoire de Recherche en Informatique, Université de Paris–Sud, Orsay, France.

Often, such algorithms are applied only when the number of bits to represent individual elements is very small in comparison with the number of elements in the set. Otherwise, those algorithms may consume huge amounts of space. For example, van Emde Boas trees [43, 42] can be used to perform predecessor queries on any set of integers from a universe of size N in $O(\log\log N)$ time, but they require $\Omega(N)$ space.

However, there have been important algorithmic breakthroughs showing that such techniques have more general applicability. For example, with two level perfect hashing [25], any n element set can be stored in $O(n)$ space and constant time membership queries can be performed. Fusion trees fully exploit unit-cost word-level operations and the fact that data elements need to fit in words of memory to store static sets of size n in $O(n)$ space and perform predecessor queries in $O(\sqrt{\log n})$ time [27].

For the static predecessor problem, it has been widely conjectured that the time complexities achieved by van Emde Boas trees and fusion trees are optimal for any data structure using a reasonable amount of space [28]. We prove that this is NOT the case. Specifically, we construct a new data structure that stores n element sets of integers from a universe of size N in $n^{O(1)}$ space and performs predecessor queries in time

$$O\left(\min\left\{\frac{\log\log N}{\log\log\log N}, \sqrt{\frac{\log n}{\log\log n}}\right\}\right).$$

Using recent generic transformations of Andersson and Thorup [7, 10], the algorithm can be made dynamic and the space improved to $O(n)$, although the time increases to $O\left(\min\left\{\frac{\log\log N}{\log\log\log N}\cdot\log\log n, \sqrt{\frac{\log n}{\log\log n}}\right\}\right)$

We also obtain matching lower bounds for a class of problems that includes the static predecessor problem. These lower bounds are proved in the powerful communication game model and improve Miltersen's $\Omega(\sqrt{\log\log N})$ lower bound [35] and Miltersen, Nisan, Safra, and Wigderson's $\Omega((\log n)^{1/3})$ lower bound [35, 37]. The key to our lower bounds is to use a better distribution of inputs on which to consider the behaviour of the algorithm. However, this requires a more complicated analysis. The same approach has also been used to obtain an $\Omega(\log\log d/\log\log\log d)$ lower bound for the approximate nearest neighbour problem over the universe $\{0,1\}^d$ [15].

A description of related work is given in Section 2, followed by our lower bounds in Section 3.4, and our new algorithms in Section 4. Throughout the paper log will denote a logarithm to the base 2 and $[i,j]$ will denote the set of $j-i+1$ integers $\{i, i+1, \ldots, j\}$.

2 Related Work

The simplest model in which the dictionary and predecessor problems have been considered is the *comparison model*. In this model, the only operations that involve the query key x are comparisons between x and elements of the set S. Using binary search, predecessor and membership queries can be performed in $O(\log n)$ time on a static set of size n stored in a sorted array. Balanced binary trees (for example, Red-Black trees, AVL trees, or 2-3 trees) can be used in this model to solve the dynamic dictionary and predecessor problems in $O(\log n)$ time. A standard information theory argument proves that $\lceil\log(n+1)\rceil$ comparisons are needed in the worst case for performing membership queries on a static set.

Faster algorithms are possible when x or some function computed from x can be used for indirect addressing. If S is a static set from a universe of size N, one can trivially perform predecessor queries using N words of memory in constant time: Simply store the answer to each possible query x in a separate location. This also works for more general queries. If the number of bits, k, needed to represent each answer is smaller than the number of bits b in a memory word, the answers can be packed b/k to a word, for a total of

$O(Nk/b)$ words. For example, for membership queries, $k = 1$. In this special case, updates to the table can also be performed in constant time.

If the universe size N is significantly less than 2^b, where b is the number of bits in a word, then packed B-trees [27, 6, 12, 40] are time and space efficient. Specifically, using branching factor $B \leq b/(1+\log N)$, insertions, deletions, and membership and predecessor queries can be performed in $O(\log n/\log B)$ steps using $O(n/B)$ words.

The most interesting data structures are those that work for an arbitrary universe whose elements can fit in a single word of memory (i.e. $N \leq 2^b$) and use a number of words that is polynomial in n, or ideally $O(n)$. The static dictionary problem has optimal constant-time data structures with these properties: Constant time membership queries can be obtained for any set of size n using an $O(n^2)$ word hash table and a hash function randomly chosen from a suitable universal family [13]. Fredman, Komlòs, and Szeméredi [25] improved the space to $O(n)$ using two level perfect hashing. Their data structure can be constructed in $O(n)$ expected time. To evaluate the hash functions, multiplication and division of $\log N$ bit words are used. Recently it was shown that hash functions of the form

$$h(x) = ax \mod 2^{\lceil \log N \rceil} \operatorname{div} 2^{\lceil \log N \rceil - r},$$

for $r \leq \lceil \log n \rceil$, suffice for implementing the two level perfect hashing data structure [20, 21]. Notice that the evaluation of such functions does not depend on constant time division instructions; a right shift suffices. Raman [40] proves that it is possible to choose such hash functions deterministically in $O(n^2 \log N)$ time. using a different type of hash function, Pagh [38] has recently shown how a linear space static dictionary with constant query time can be constructed deterministically in $n(\log n)^{O(1)}$ time.

Dynamic dictionaries can be built using two level perfect hashing, rehashing when necessary (with hash functions randomly chosen from an appropriate universal family), to perform membership queries in constant time and perform updates in expected constant time [21, 22, 19]. Miltersen [36] has constructed a deterministic dynamic dictionary using error correcting codes and clustering. It performs membership queries in constant time and performs updates in time $O(n^\varepsilon)$ for any constant $\varepsilon > 0$. All of these data structures use $O(n)$ space.

Although hashing provides an optimal solution to the static dictionary problem, it is not directly applicable to the predecessor problem. Another data structure, van Emde Boas (or stratified) trees [43, 42], is useful for both static and dynamic versions of the predecessor problem. These trees support membership queries, predecessor queries, and updates in $O(\log \log N)$ time. The set is stored in a binary trie and binary search is performed on the $\log N$ bits used to represent individual elements. The major drawback is the use of an extremely large amount of space: $\Omega(N)$ words.

Willard's x-fast trie data structure [44] uses the same approach, but reduces the space to $O(n \log N)$ by using perfect hashing to represent the nodes at each level of the trie. In y-fast tries, the space is further reduced to $O(n)$ by only storing $\Theta(n/\log N)$ approximately evenly spaced elements of the set S in the x-fast trie. A binary search tree is used to represent the subset of $O(\log N)$ elements of S lying between each pair of consecutive elements in the trie. Both of Willard's data structures perform membership and predecessor queries in worst-case $O(\log \log N)$ time, but use randomization (for rehashing) to achieve expected $O(\log \log N)$ update time.

Fusion trees, introduced by Fredman and Willard [27] use packed B-trees with branching factor $\Theta(\log n)$ to store approximately one out of every $(\log n)^4$ elements of the set S. The effective universe size is reduced at each node of the B-tree by using carefully selected bit positions to obtain compressed keys representing the elements stored at the node. As above, binary search trees are used to store the roughly equal-sized sets of intermediate elements and a total of $O(n)$ space is used. Membership and predecessor queries take

$O(\log n/\log\log n)$ time. Updates take $O(\log n/\log\log n)$ amortized time. (The fact that this bound is amortized arises from the worst case $\Theta((\log n)^4)$ time bound to update a node in their B-tree.) This data structure forms the basis of their ingenious $O(n\log n/\log\log n)$ sorting algorithm. Fusion trees actually allow one to implement B-trees with any branching factor that is $O((\log N)^{1/6})$, so for $n \leq (\log N)^{(\log\log N)/36}$, the time bounds for fusion trees can be improved to $O(\sqrt{\log n})$ while retaining $O(n)$ space. This is done by using branching factor $\Theta(2^{\sqrt{\log n}})$ in the B-tree and storing $2^{\Theta(\sqrt{\log n})}$ elements in each of the binary search trees. For the remaining range, $n > (\log N)^{(\log\log N)/36}$, Willard's y-fast tries have query time and expected update time $O(\log\log N) = O(\sqrt{\log n})$.

Andersson [6] uses similar ideas to construct another data structure with $O(\sqrt{\log n})$ time membership and predecessor queries, expected $O(\sqrt{\log n})$ update time, and $O(n)$ space. As above, only $\Theta(n/2^{\sqrt{\log n}})$ elements are stored in the main data structure; the rest are in binary search trees of height $\Theta(\sqrt{\log n})$. His idea is to reduce the length of the representation of elements by a factor of $2^{\sqrt{\log n}}$ using $\sqrt{\log n}$ recursive calls each of which halves the length (as in van Emde Boas trees and y-fast tries). Essentially, $2^{\sqrt{\log n}}$ equally spaced levels of the y-fast trie data structure are stored. At this point, packed B-trees with branching factor $2^{\sqrt{\log n}}$ and height $O(\sqrt{\log n})$ are used.

Also using B-trees with compressed keys, Hagerup [28] can construct a static ranking dictionary in $O(n)$ time, given the elements in sorted order. His data structure uses $O(n)$ space and performs rank, selection, and, hence, predecessor queries, in $O(1+(\log n)/\log b)$ time.

Brodal [11] constructs a data structure that is similar to Andersson's but uses buffers to delay insertions and deletions to the packed B-tree. In the worst case, it uses $O(f(n))$ time to perform updates and $O(\log n/f(n))$ time to perform predecessor queries, for any nice function f such that $\log\log n \leq f(n) \leq \sqrt{\log n}$. However, it uses $O(nN^{\varepsilon})$ space, for some constant $\varepsilon > 0$.

Thorup [41] shows that $O(n\log\log n)$ time is sufficient to perform a batch of n predecessor queries. He uses a trie of height $\log n\log\log n$ over an $N^{1/\log n\log\log n}$ letter alphabet. For each query, binary search is used to find the deepest trie node which is a prefix of the query word, as in van Emde Boas trees or x-fast tries, in time $O(\log(\log n\log\log n)) = O(\log\log n)$. This reduces the original problem to a collection of subproblems of combined size n over a universe of size $N^{1/\log n\log\log n}$. In this smaller universe, linear time sorting can be used to solve all the subproblems in a total of $O(n)$ time.

Interpolation search can be used to perform membership and predecessor queries in a static set of size n in expected time $O(\log\log n)$ under the assumption that the elements of the set are independently chosen from some distribution [39, 45].

Andersson's exponential search trees [7] give a general method for transforming any static data structure that performs membership and predecessor queries in time $T(n)$ into a linear space dynamic data structure with query and amortized update time $T'(n)$, where $T'(n) \leq T(n^{k/(k+1)}) + O(T(n))$, provided the static data structure can be constructed in n^k time and space, for some constant $k \geq 1$. The root of the tree has degree $\Theta(n^{1/(k+1)})$ and the degrees of other nodes decrease geometrically with depth. Instances of the static data structure are used to implement the search at each node. Global and partial rebuilding are used to update the data structure. Combined with fusion trees, packed B-trees, and x-fast tries, exponential search trees give a solution to the dynamic predecessor problem that uses worst case search time and amortized update time

$$O\left(\min\left\{\begin{array}{l}\sqrt{\log n}\\ \log\log N\cdot\log\log n\\ \frac{\log n}{\log b}+\log\log n\end{array}\right\}\right)$$

and $O(n)$ space. Andersson and Thorup [10] have combined a variant of exponential search trees with eager

partial rebuilding to improve the resulting dynamic data structure, achieving worst-case instead of amortized bounds for update time.

Although priority queues can be implemented from dynamic predecessor, there are more efficient data structures. Thorup [41] gives a randomized priority queue with expected time $O(\log\log n)$ for insert and extractmin and a deterministic priority queue with amortized time $O((\log\log n)^2)$. Using exponential search trees, Andersson and Thorup [10] improve this bound to $O((\log\log n)^2)$ in the worst case.

Another problem that can be solved using predecessor queries is one dimensional range search (i.e. find some element of S contained within a query interval, if one exists). However, Alstrup, Brodal, and Rauhe [3] have recently obtained a static data structure of size $O(n)$ for storing a set S of n integers that supports range search in constant time. Using this data structure, they can approximately count (i.e. to within a factor of $1+\varepsilon$, for any constant $\varepsilon > 0$) the number of elements of S that that lie within a query interval in constant time. They can also return all elements of S contained within a query interval in time linear in the number of elements of S reported.

One of the most natural and general models for proving lower bounds for data structures problems, and one that is ideally suited for representing word-level operations, is the *cell probe* model, introduced by Yao [48]. In this model, there is a memory consisting of *cells*, each of which is capable of storing some fixed number of bits. A cell probe algorithm is a decision tree with one memory cell accessed at each node. The decision tree branches according to the contents of the cell accessed. We only count the number of memory cells accessed in the data structure; all computation is free. This means that no restrictions are imposed on the way data is represented or manipulated, except for the bound on the size of values that each memory cell can hold. Thus, lower bounds obtained in this model apply to all reasonable models of computation and give us insight into why certain problems are hard.

In this model, Fredman and Saks [26] showed that any dynamic data structure supporting rank queries for sets of integers from a universe of size N require $\Omega(\log N/\log\log N)$ amortized time per operation. A rank query asks how many elements in a given set of integers are less than or equal to a given integer. It is interesting that, for static sets, predecessor queries and rank queries are equally difficult [28].

Alstrup, Husfeldt, and Rauhe [4] considered a generalization of the dynamic predecessor problem in a universe of size N: the marked ancestor problem in a tree of N nodes. They proved a tradeoff,

$$t \in \Omega\left(\frac{\log N}{\log(ub\log N)}\right),$$

between the update time u and the query time t in the cell probe model with word size b. If the word size and update time are both in $O(\log N)^{O(1)}$, this gives a lower bound of $\Omega(\log N/\log\log N)$ for query time. They also construct a RAM algorithm which matches this query time while using only $O(\log\log N)$ time per update and $O(N)$ words, each containing $O(\log N)$ bits.

Ajtai, Fredman, and Komlos [1] showed that, if the word length is sufficiently large (i.e. $n^{\Omega(1)}$ bits), then any set of size n can be stored, using a trie, in $O(n)$ words so that predecessor queries can be performed in constant time in the cell probe model. On the other hand, Ajtai [2] proved that, if the word length is sufficiently small (i.e. $O(\log n)$ bits), and only $n^{O(1)}$ words of memory are used to represent any set of n elements, then worst-case constant time for predecessor queries is impossible.

Miltersen [35] observed that a cell probe algorithm can be viewed as a two-party communication protocol [47] between a Querier who holds the input to a query and a Responder who holds the data structure. In each round of communication, the Querier sends the name of a memory cell to access and the Responder answers with the contents of that memory cell. The communication game model is more general, since

the response at a given round can depend on the entire history of the computation so far. In the cell probe model, the response can depend only on which memory cell is being probed, so different probes to the same memory cell must always receive the same response. In fact, for many problems, the cell probe complexity is significantly larger than the communication complexity [34].

Miltersen [35] generalized Ajtai's proof to obtain an $\Omega(\sqrt{\log\log N})$ lower bound on time in this model for the problem of finding predecessors in a static set from a universe of size N. In [37], it was shown that for certain universe sizes, Ajtai's proof and its generalization in [35] also gives an $\Omega((\log n)^{1/3})$ lower bound on time. These lower bounds (and the lower bounds in our paper) actually apply to a large natural class of data structure problems introduced in [24]. Furthermore, Miltersen [35] provides a general technique for translating time complexity lower bounds (under restrictions on memory size) for static data structure problems into time complexity lower bounds for dynamic data structure problems. In particular, he shows that the time to perform predecessor queries is $\Omega(\sqrt{\log\log N})$ if the time to perform updates is at most $2^{(\log N)^{1-\varepsilon}}$ for some constant $\varepsilon > 0$.

Although the cell probe model is useful for proving the most generally applicable data structure lower bounds, it does not permit one to analyze the particular instructions necessary for these algorithms.

Fich and Miltersen [23] have shown that, for the standard RAM model (which includes addition, multiplication, conditional jumps, and indirect addressing instructions, but not shifts, bitwise Boolean operations, or division), the complexity of performing membership queries in a set of size n stored using at most $N/n^{\Omega(1)}$ words (of unbounded size) requires $\Omega(\log n)$ time. Thus, for this model, binary search is optimal.

AC^0 RAMs allow conditional jumps and indirect addressing, as well as any finite set of AC^0 instructions (such as addition and shifts, but not multiplication or division). Hagerup [28] gave an AC^0 RAM algorithm for the dynamic predecessor problem that uses $O(n)$ words and performs queries and updates in $O(1+\log n/\log b)$ time. Andersson, Miltersen, and Thorup [9] showed how to efficiently implement fusion trees on an AC^0 RAM. In the same model, Andersson, Miltersen, Riis, and Thorup [8] proved that the time complexity of the static dictionary problem is $\Theta(\sqrt{\log n/\log\log n})$. Their algorithm uses $O(n)$ words and their lower bound holds even if $2^{(\log n)^{O(1)}}$ words are allowed. It is intriguing that the somewhat unusual function describing the time complexity in this case is the same as the one that we derive in a different context.

On the pointer machine model, the time complexity of the dynamic predecessor problem is $\Theta(\log\log N)$ [32, 33, 31].

3 Lower Bounds

Our lower bounds are proved in the general language-theoretic framework introduced in [35]. We begin with a brief description of the class of problems and then show how certain problems in this class can be reduced to various data structure problems. Next, we present some technical combinatorial lemmas. This is followed by a lower bound proof in the communication game model for any static problem in our class. Finally, a number of corollaries are given, including tradeoffs between update and query time for dynamic data structure problems.

3.1 Indecisive Languages and the Static Prefix Problem

Let Σ denote a finite alphabet that does not contain the symbol $\perp$ and let ε denote the empty word. We focus attention on a special class of regular languages.

DEFINITION 3.1: A regular language $L \subseteq \Sigma^*$ is *indecisive* if and only if for all $x \in \Sigma^*$ there exist $z, z' \in \Sigma^*$ such that $xz \in L$ and $xz' \notin L$.

Thus, for an indecisive language, knowing that a particular word is a prefix of the input does not determine the answer to the membership query for that input.

Suppose that L is an indecisive language that is accepted by a deterministic finite automaton with q states. Then $q \geq 2$, since the automaton accepts some words and reject others. Furthermore, the strings z and z' in Definition 3.1 can be chosen to be of length at most $q-1$.

For any string $y = y_1 \cdots y_N \in (\Sigma \cup \{\perp\})^*$ and any nonnegative integer $j \leq N$, let $\mathrm{PRE}_j(y) \in \Sigma^*$ denote the string of length at most j obtained by deleting all occurrences of $\perp$ from the length j prefix, $y_1 \cdots y_j$, of y. For example, if $y = 01\perp\perp01$, then $\mathrm{PRE}_2(y) = 01$ and $\mathrm{PRE}_5(y) = 010$. Let $Z(N,n)$ denote the set of strings in $(\Sigma \cup \{\perp\})^N$ containing at most n non-$\perp$ characters.

DEFINITION 3.2: Let $L \subseteq \Sigma^*$ and $x \in \Sigma^*$. The *static* (L,N,n,x)*-prefix problem* is to store an input string $y \in Z(N,n)$ so that, for any $j \in [0,N]$, the query "Is $x \cdot \mathrm{PRE}_j(y) \in L$?" may be answered. When $x = \varepsilon$, the static (L,N,n,x)-prefix problem will also be called the static (L,N,n)-prefix problem.

If $N' \geq N$ and $n' \geq n$, then any string in $Z(N,n)$ can be viewed as a string in $Z(N',n')$ by appending $\perp^{N'-N}$ to it. Thus the static (L,N',n',x)-prefix problem is at least as hard as the static (L,N,n,x)-prefix problem.

The static (L,N,n,x)-prefix problem for any language L and any string $x \notin L$ can be reduced to the static $(\{0,1\}^*1,N,n)$-prefix problem as follows. Given an instance $y \in (\Sigma \cup \{\perp\})^N$, let $y' \in \{0,1,\perp\}^N$ be defined so that

$$y'_i = \begin{cases} \perp & \text{if } y_i = \perp \\ 1 & \text{if } y_i \neq \perp \text{ and } x \cdot PRE_i(y) \in L \\ 0 & \text{if } y_i \neq \perp \text{ and } x \cdot PRE_i(y) \notin L. \end{cases}$$

Then, by construction, $x \cdot PRE_i(y) \in L$ if and only if $PRE_i(y') \in \{0,1\}^*1$. Thus, the static $(\{0,1\}^*1,N,n)$-prefix problem is the hardest such problem. Similarly, when $x \in L$, the static (L,N,n,x)-prefix problem can be reduced to the $(\{0,1\}^*1,N,n,1)$-prefix problem.

3.2 Related Problems

DEFINITION 3.3: The *static* (N,n)*-predecessor-parity problem* is to store a set $S \subseteq [1,N]$ of size at most n so that for any value $x \in [1,N]$, the parity of the predecessor of x in the set S, $parity(pred(x,S))$, can be determined.

The static (N,n)-predecessor-parity problem is no harder than the static predecessor problem. It can also be reduced to the static $(\{0,1\}^*1,N,n)$-prefix problem, as follows. Given a set $S \subseteq [1,N]$ of size at most n, let $y \in \{0,1,\perp\}^N$ be defined so that

$$y_i = \begin{cases} \perp & \text{if } i \notin S \\ 1 & \text{if } i \in S \text{ and } i \text{ is odd} \\ 0 & \text{if } i \in S \text{ and } i \text{ is even.} \end{cases}$$

Then $pred(j,S)$ is odd if and only if $PRE_{j-1}(y) \in \{0,1\}^*1$.

Conversely, the static $(\{0,1\}^*1,N,n)$-prefix problem can be reduced to the static $(2N,n)$-predecessor-parity problem, as follows. Given a string $y \in \{0,1,\perp\}^N$, let $S = \{2i-1 \mid y_i = 1\} \cup \{2i \mid y_i = 0\} \subseteq [1,N]$. Then $PRE_j(y) \in \{0,1\}^*1$ if and only if $parity(pred(2j+1,S)) = 1$.

DEFINITION 3.4: The *point separation problem* is to store a set of points in the plane and decide whether they all lie on the same side of a query line.

The static $(\{0,1\}^*1,N,n)$-prefix problem can also be reduced to the point separation problem, as follows. Given an instance $z \in \{0,1,\perp\}^N$ of the prefix problem, an instance of the point separation problem can be constructed as follows, using ideas in [16, 17]. Suppose $i_1 < \cdots < i_{s-1}$ denote the indices of the non-$\perp$ characters of z. Let $i_s = N+1$ and $P = \{p_{i_1}, x_{i_1 i_2}, p_{i_2}, \ldots, x_{i_{s-1} i_s}, p_{i_s}\}$, where $p_{i_j} = (2i_j, 4i_j^2)$ for $j = 1,\ldots,s$ and, for $j = 1,\ldots,s-1$,

$$x_{i_j i_{j+1}} = \begin{cases} (i_j + i_{j+1}, 2i_j^2 + 2i_{j+1}^2) & \text{if } z_{i_j} = 0 \\ (i_j + i_{j+1}, 4i_j i_{j+1}) & \text{if } z_{i_j} = 1. \end{cases}$$

Then, for $k \in [1,N]$, $PRE_k(z) \in \{0,1\}^*1$ if and only if there are points in P on both sides of the query line $y = 2(2k+1)x - (2k+1)^2$ (which is tangent to the parabola $y = x^2$ at the point $(2k+1,(2k+1)^2)$).

The *rank* of an element x in a set S is the number of elements in S that are less than or equal to x.

DEFINITION 3.5: The *rank problem* is to store a set from an ordered universe so that, for any element in the universe, the rank of the element in the set can be determined. The *rank parity problem* is to store a set from an ordered universe so that, for any element in the universe, the parity of the rank of the element in the set can be determined.

The rank parity problem is no harder than the rank problem. The static $((11)^*1,N,n)$-prefix problem is equivalent to the rank parity problem for subsets of $[1,N]$ of size at most n. Let $y \in \{1,\perp\}^N$ be a string and let $S \subseteq [1,N]$ be a set such that $y_i = 1$ if and only if $i \in S$. Then $PRE_j(y) \in (11)^*1$ if and only if $parity(rank(j+1,S)) = 1$.

DEFINITION 3.6: The *(exact) range counting problem* is to store a set from an ordered universe so that, for any two elements $x \le x'$ in the universe, the number of elements in the set that are in the range $[x,x']$ can be determined.

The range counting problem is no harder than the rank problem.

3.3 Combinatorial Preliminaries

In this section, we state two combinatorial results which are important for the lower bound proofs given in the next subsection.

The following form of the Chernoff-Hoeffding bound follows easily from the presentation in [18].

Proposition 3.1: Fix $H \subseteq U$ with $|H| \ge \rho|U|$ and let $S \subseteq U$ with $|S| = s$ be chosen uniformly at random. Then

$$\Pr[|H \cap S| \le \rho s/4] \le (\sqrt{2}/e^{3/4})^{\rho s} < 2^{-\rho s/2}.$$

The next result is a small modification and rephrasing of a combinatorial lemma that formed the basis of Ajtai's lower bound argument in [2].

Suppose we have a tree T of depth d such that all nodes on the same level have the same number of children. For $\ell = 0, \ldots, d$ let V_ℓ be the set of nodes of T on level ℓ (i.e. at depth ℓ) and for $\ell < d$ let f_ℓ be the fan-out of each node on level ℓ. Thus $|V_{\ell+1}| = f_\ell |V_\ell|$ for $\ell = 0, \ldots, d-1$.

For any node $v \in T$, let leaves(v) denote the set of nodes in V_d that are descendants of v and, if v is not the root of T, let parent(v) denote the parent of v. Let $A(0), \ldots, A(m-1)$ be disjoint sets of leaves of T and let $A = \bigcup_{c=0}^{m-1} A(c)$. The leaves in $A(c)$ are said to have *colour* c. A nonleaf node v has *colour* c if leaves(v) contains a node in $A(c)$. For $c = 0, \ldots, m-1$, let $A'(c) = \{v \mid \text{leaves}(v) \cap A(c) \neq \phi\}$ denote the set of nodes with colour c. Note that the sets $A'(0), \ldots A'(m-1)$ are not necessarily disjoint, since a nonleaf node may have more than one colour.

The density of a nonleaf node is the maximum, over all colours c, of the fraction of its children that have colour c. A nonleaf node v is δ-*dense* if it has density at least δ.

Let $R_\ell^\delta(c)$ be the set of those nodes on level ℓ that are coloured c and do not have a δ-dense ancestor at levels $1, \ldots, \ell-1$. In particular, $R_1^\delta(c) = A'(c) \cap V_1$. The fraction of nodes on level ℓ that are in $R_\ell^\delta(c)$ decreases exponentially with ℓ.

Proposition 3.2: For $1 \leq \ell \leq d$, $|R_\ell^\delta(c)| \leq \delta^{\ell-1}|V_\ell|$.

Proof By induction on ℓ. The base case, $\ell = 1$, is trivial since $R_1^\delta(c) \subseteq V_1$.

Now let $1 \leq \ell < d$ and assume that $|R_\ell^\delta(c)| \leq \delta^{\ell-1}|V_\ell|$. If $v \in R_{\ell+1}^\delta(c)$, then, by definition, v has colour c and no ancestor of v at levels $1, \ldots, \ell$ is δ-dense. Since v has colour c, parent(v) also has colour c and, thus, parent$(v) \in R_\ell^\delta(c)$. Furthermore, parent(v) is not δ-dense, so fewer than $\delta \cdot f_\ell$ of its children are in $R_{\ell+1}^\delta(c)$. Hence,

$$|R_{\ell+1}^\delta(c)| < \delta \cdot f_\ell |R_\ell^\delta(c)| \leq \delta \cdot f_\ell \cdot \delta^{\ell-1}|V_\ell| = \delta^\ell |V_{\ell+1}|,$$

as required. □

We now prove Ajtai's Lemma:

Proposition 3.3: (Ajtai's Lemma) Let T be a tree of depth $d \geq 2$ such that all nodes on the same level of T have the same number of children. Suppose that at least a fraction α of all the leaves in T are coloured (each with one of m colours). Then there exists a level ℓ, $1 \leq \ell \leq d-1$, such that the fraction of nodes on level ℓ of T that are δ-dense is at least

$$\frac{\alpha - m\delta^{d-1}}{d-1}.$$

Proof By Proposition 3.2, $|R_d^\delta(c)| \leq \delta^{d-1}|V_d|$ for all colours c. Let A be the set of all coloured leaves in T and let $R^\delta = \bigcup_{c=0}^{m-1} R_d^\delta(c) \subseteq A$. There are m colours; therefore $|R^\delta| \leq m\delta^{d-1}|V_d|$.

If $w \in A \subseteq V_d$ and none of its ancestors at levels $1, \ldots, d-1$ are δ-dense, then $w \in R^\delta$. Thus $w \in A - R^\delta$ implies that some ancestor of w at some level $1, \ldots, d-1$ is δ-dense.

For $\ell = 1, \ldots, d-1$, let δ_ℓ denote the fraction of nodes in V_ℓ that are δ-dense. Observe that because the fan-out at each level of T is constant, for any $v \in V_\ell$, $|\text{leaves}(v)| = |V_d|/|V_\ell|$. Therefore, for each ℓ, $1 \leq \ell \leq d-1$, the number of leaf nodes of T that lie below δ-dense nodes in V_ℓ is $\delta_\ell |V_d|$. It follows that

$$|A - R^\delta| \leq \sum_{\ell=1}^{d-1} \delta_\ell |V_d|.$$

But $|A - R^{\delta}| = |A| - |R^{\delta}| \geq \alpha\,|V_d| - m\delta^{d-1}|V_d|$, so

$$\sum_{\ell=1}^{d-1} \delta_\ell \geq \alpha - m\delta^{d-1}.$$

Thus there is some ℓ, $1 \leq \ell \leq d-1$, such that $\delta_\ell \geq (\alpha - m\delta^{d-1})/(d-1)$, as required. □

The number of δ-dense nodes can change significantly with a small change in δ. Consider a tree T of depth $d \geq 2$, where all nodes on level l have $f_l = 2^{e_l}$ children, for $0 \leq l \leq d-1$. Then T has $N = f_0 \cdots f_{d-1} = 2^E$ leaves, where $E = e_0 + \cdots + e_{d-1}$. Let $a \geq 0$ and $e \geq 1$ be integers such that $a \leq e \leq \min\{e_1, \ldots, e_{d-2}, e_{d-1}\}$ and let $m = 2^{e(d-1)-a}$. Colour a fraction $\alpha = 2^{-a}$ of the leaves of T as follows: The $x+1$'st leaf is coloured if and only if the a least significant bits of the binary representation of x are all 0. The colour of this leaf is a number in $[0, m-1]$ whose binary representation is formed by the concatenation of selected bits from x's binary representation. Specifically, divide the binary representation of x into d blocks of length $e_0, \ldots, e_{d-1}$. Then concatenate together the e most significant bits in each of the middle $d-2$ blocks plus the $e-a$ most significant bits in the least significant block. The root of the tree T has density 1 and all other internal nodes have density 2^{-e}. Thus when $\delta \leq 2^{-e}$, all internal nodes are δ-dense; whereas, when $\delta > 2^{-e}$, only the root of T is δ-dense. An example with $d = 4$, $e_0 = e_3 = 1$, $e_1 = e_2 = 2$, $e = 1$, and $a = 1$ is depicted in Figure 1. The positions of the selected bits in the binary representation (i.e. those that determine the colour of the leaves) are shaded.

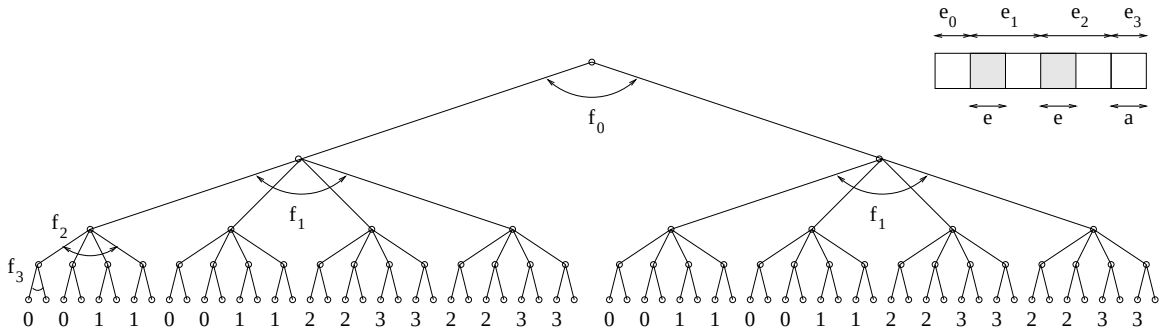

Figure 1: An Example Related to Ajtai's Lemma

3.4 Lower Bounds for Static Problems

In this section, we present an adversary argument that proves a lower bound on the time to perform prefix queries for any indecisive language, provided strings are stored using only a polynomial amount of memory.

As discussed in the introduction, Miltersen [35] observed that one can phrase a static data structure algorithm in the cell-probe model in terms of a communication protocol between between two players: the Querier, who holds the input to a query, and the Responder, who holds the data structure. Each probe that the Querier makes to the data structure, a cell name, consists of $\log m$ bits of communication, where m is the number of memory cells, and each response by the Responder, the contents of that named cell, consists of exactly b bits of communication. The number of rounds of alternation of the players is the time t of the cell-probe communication protocol.

For technical reasons, we require that, at the end of the protocol, both the Querier and the Responder know the answer to the problem instance. Since we are considering decision problems, the answer to each problem instance is a single bit. In this case, once one of the players knows the answer, the other player

can be told the answer in the next round. Therefore, if there is an algorithm for the static (L,N,n,x)-prefix problem that uses at most $t-1$ probes, then there is a communication protocol that solves this problem in at most t rounds.

The lower bound, in the style of [30], works 'top down', maintaining, for each player, a relatively large set of inputs on which the communication is fixed. Unlike [30], we actually have non-uniform distributions on the Responder's inputs, so our notion of 'large' is with respect to these distributions. The distributions get simpler as the rounds of the communication proceed.

If $\mathcal{Z}$ is a probability distribution on a set Z and $B \subseteq Z$, we use $\mu_{\mathcal{Z}}(B)$ to denote the probability an element randomly chosen from the distribution Z is in B. Let $\mathcal{U}(N,n)$ be the distribution which chooses an element $y = y_1 \ldots y_N$ from $Z(N,n)$ by first choosing $S \subseteq [1,N]$ of size n uniformly at random and then, independently for each $j \in S$, choosing $y_j \in \Sigma \cup \{\perp\}$ uniformly at random, and finally setting $y_j = \perp$ for $j \notin S$. That is, each of the $\binom{N}{i}|\Sigma|^i$ strings in $Z(N,n)$ with exactly i non-$\perp$ characters has probability $\binom{N-i}{n-i}/[\binom{N}{n}(|\Sigma|+1)^n]$.

The base case of the lower bound considers the situation in which no communication is performed. Specifically, we argue that there do not exist a large set of positions $A \subseteq [1,N]$ and a large set of strings $B \subseteq Z(N,n)$ for which the static (L,N,n,x)-prefix problem can be solved, for all $j \in A$ and $y \in B$, without communication.

Lemma 3.4: Suppose that $L \subseteq \Sigma^*$ is an indecisive regular language accepted by a deterministic finite automaton with q states. Let $N \geq n > 0$ and suppose that $b \geq 2(|\Sigma|+1)^q$, $\alpha n \geq \max(8qb^2, 12b^3)$, and $\beta \geq 2^{-2b+1}$. Consider any set of positions $A \subseteq [1,N]$, with $|A| \geq \alpha N$, and any set of strings $B \subseteq Z(N,n)$, with $\mu_{\mathcal{U}(N,n)}(B) \geq \beta$. Then, for any $x \in \Sigma^*$, there exist integers $a, a' \in A$ and a string $y \in B$ such that $x \cdot \mathrm{PRE}_a(y) \in L$ and $x \cdot \mathrm{PRE}_{a'}(y) \notin L$.

Proof Let $x \in \Sigma^*$. Consider the event that a string y randomly chosen from the distribution $\mathcal{U}(N,n)$ has $x \cdot \mathrm{PRE}_a(y) \in L$ for all $a \in A$ or $x \cdot \mathrm{PRE}_a(y) \notin L$ for all $a \in A$. We will show that the probability of this event is less than β. Since $\mu_{\mathcal{U}(N,n)}(B) \geq \beta$, it will follow that there exist integers $a, a' \in A$ and a string $y \in B$ such that $x \cdot \mathrm{PRE}_a(y) \in L$ and $x \cdot \mathrm{PRE}_{a'}(y) \notin L$.

It is convenient to restrict attention to a well spaced subset of A. Specifically, it is possible to choose $a_0 < a_1 < \cdots < a_{b^2} \in A$ such that $|[a_{i-1}+1, a_i]| = a_i - a_{i-1} \geq \lfloor (|A|-1)/b^2 \rfloor \geq |A|/b^2 - 1$. Since $|A| \geq \alpha N \geq \alpha n \geq 8b^2$, it follows that $|[a_{i-1}+1, a_i]| \geq |A|/b^2 - 1 \geq \alpha N/b^2 - 1 \geq 7\alpha N/(8b^2) > \alpha N/(2b^2)$ for $i = 1, \ldots, b^2$.

Let $S \subseteq [1,N]$ with $|S| = n$ be chosen uniformly at random. Then, since $q \leq \alpha n/(8b^2)$ and $\alpha n/(4b^2) \geq 3b$, applying Proposition 3.1 with $H = [a_{i-1}+1, a_i]$ and $\rho = \alpha/(2b^2)$,

$$\mathrm{Prob}[|[a_{i-1}+1, a_i] \cap S| < q] \leq \mathrm{Prob}[|[a_{i-1}+1, a_i] \cap S| \leq \alpha n/(8b^2)] < 2^{-\alpha n/(4b^2)} \leq 2^{-3b}.$$

Since $b \geq 4$, there are $b^2 \leq 2^b$ intervals. Therefore, the probability that at least one of them contains fewer than q elements of S is less than $b^2 2^{-3b} \leq 2^{-2b} \leq \beta/2$.

Now consider any fixed choice for S that has at least q elements in each of these b^2 intervals. For each interval $[a_{i-1}+1, a_i]$, consider the set Q_i of the last q elements of S in the interval. L is an indecisive regular language. Therefore, for each fixed choice w for the symbols of y that occur before the first element of Q_i, there are strings $z, z' \in (\Sigma \cup \{\perp\})^q$ such that $x \cdot \mathrm{PRE}_{a_i}(wz) \in L$ and $x \cdot \mathrm{PRE}_{a_i}(wz') \notin L$. There are $(|\Sigma|+1)^q$ equally likely ways that the characters of y in positions indexed by Q_i will be assigned values. Thus, with probability at least $(|\Sigma|+1)^{-q}$, either $x \cdot \mathrm{PRE}_{a_{i-1}}(y) \in L$ and $x \cdot \mathrm{PRE}_{a_i}(y) \notin L$ or $x \cdot \mathrm{PRE}_{a_{i-1}}(y) \notin L$ and $x \cdot \mathrm{PRE}_{a_i}(y) \in L$. Therefore, the probability that this event does not occur is at most $1 - (|\Sigma|+1)^{-q}$.

Since the choices of the portions of string y in each of the b^2 intervals $[a_{i-1}+1, a_i]$ are independent, the probability that either $x \cdot \mathrm{PRE}_{a_i}(y) \in L$ for all $i = 0, \ldots, b^2$ or $x \cdot \mathrm{PRE}_{a_i}(y) \notin L$ for all $i = 0, \ldots, b^2$ is at most

$$(1 - (|\Sigma|+1)^{-q})^{b^2} \leq e^{-(|\Sigma|+1)^{-q} b^2} < 2^{-2b} \leq \beta/2 < \beta$$

since $b \geq 2(|\Sigma|+1)^q$.

Because $\mu_{\mathcal{U}(N,n)}(B) \geq \beta$, it follows that there exists a string $y \in B$ such that neither $x \cdot \mathrm{PRE}_a(y) \in L$ for all $a \in A$ nor $x \cdot \mathrm{PRE}_a(y) \notin L$ for all $a \in A$. □

Let $L \subseteq \Sigma^*$ be an indecisive regular language. We define a sequence of distributions on $Z(N,n)$ and use it to demonstrate that no cell-probe communication protocol using n^k memory cells of b bits can solve the static (L,N,n,x)-prefix problem in t rounds. Given integers b, k, t, N, and n we will define two sequences of integers N_i and n_i for $i = 0, \ldots, t-1$ with $N_0 = N$, and $n_0 = n$. The general idea of the lower bound argument is to find, after each round, a portion of the Querier's and Responder's inputs on which the cell-probe communication protocol has made little progress. After i rounds, the possible values of the Querier's input will lie in an interval of length N_i and, within this interval, the Responder's input y will have at most n_i non-$\perp$ elements. Thus, the Responder's input can be viewed as an element of $Z(N_i, n_i)$ together with a modified prefix x' consisting of x together with all characters of y preceding this interval.

More precisely, let b, k, t, N, and n be positive integers and define

- $\alpha = n^{-1/(4t)}$
- $u = 8kt$
- $r = 16bu/\alpha$
- $f = 8ru/\alpha = 128bu^2/\alpha^2$
- $N_0 = N$
- $n_0 = n$
- and, for $i = 0, \ldots, t-1$, define $N_{i+1} = (N_i/f)^{1/u}$ and $n_{i+1} = n_i/(ru)$.

We say that the tuple of parameters (b,k,t,N,n) satisfies the *integrality condition* if $1/\alpha$ is an integer greater than 1 and, for every integer $i \in [0,t]$, N_i and n_i are integers and $N_i \geq n_i$.

If n is the $4t$'th power of an integer larger than 1, then $1/\alpha$ is an integer greater than 1 and f and r are also integers. Since $f \geq ru$ and $u \geq 1$, the condition $N_t \geq n_t$ is sufficient to imply that $N_i \geq n_i$ for $i \in [0,t]$. Furthermore, if N_t and n_t are both integers, then $n_i = (ru)^{t-i} n_t$ and $N_i = f^{(u^{t-i}-1)/(u-1)} N_t^{u^{t-i}}$ are integers for $i \in [0,t]$. In particular, the integrality condition will hold for (b,k,t,N,n) if n is the $4t$'th power of an integer larger than 1 and there are integers $N_t \geq n_t$ such that $n = (ru)^t n_t$ and $N = f^{(u^t-1)/(u-1)} N_t^{u^t}$.

Suppose that the integrality condition holds for (b,k,t,N,n). We define a probability distribution $\mathcal{Z}_i$ on $Z(N_i,n_i)$ inductively, for $i = t, \ldots, 0$. The basis, $\mathcal{Z}_t$, is the distribution $\mathcal{U}(N_t,n_t)$. For every $i < t$, each string in $Z(N_i,n_i)$ can be thought of as labelling the leaves of a tree T_i with depth $u+1$, having fan-out f at the root and a complete N_{i+1}-ary tree of depth u at each child of the root. We choose a random element of $\mathcal{Z}_i$ as the sequence of leaf labels of the tree T_i, which we label using the distribution $\mathcal{Z}_{i+1}$ as follows: First, choose r nodes uniformly from among all the children of the root. For each successively deeper level, excluding the leaves, choose r nodes uniformly among the nodes at that level that are not descendants of nodes chosen at higher levels. (Notice that, since the root of T_i has $f \geq ru$ children, it is always possible to

choose enough nodes with this property at each level.) Independently, for each of these ru nodes, v, choose a string $w_v \in Z(N_{i+1}, n_{i+1})$ from $\mathcal{Z}_{i+1}$ and label the leftmost leaf in the h'th subtree of v with the h'th symbol of w_v, for $h = 1, \ldots, N_{i+1}$. Label all other leaves of T_i with $\perp$.

Lemma 3.5: Suppose that $L \subseteq \Sigma^*$ is a regular language accepted by a deterministic finite automaton M with q states. Suppose (b, k, t, N, n) satisfies the integrality condition, $b \geq 16$, and $2^b \geq 4q$. Let $x \in \Sigma^*$, $A \subseteq [1, N_i]$ with $|A| \geq \alpha N_i$, and $B \subseteq Z(N_i, n_i)$ with $\mu_{\mathcal{Z}_i}(B) \geq \beta = 2^{-2b+1}$. Suppose there is a $t - i$ round cell-probe communication protocol, using $m \leq n^k$ memory cells of b bits, that correctly determines whether $x \cdot PRE_j(y) \in L$ for all $j \in A$ and $y \in B$. Then there exist $x' \in \Sigma^*$, $A' \subseteq [1, N_{i+1}]$ with $|A'| \geq \alpha N_{i+1}$, $B' \subseteq Z(N_{i+1}, n_{i+1})$ with $\mu_{\mathcal{Z}_{i+1}}(B') \geq \beta$ and a $t - i - 1$ round cell-probe communication protocol, using m cells of b bits, that correctly determines whether $x' \cdot PRE_{j'}(y') \in L$ for all $j' \in A'$ and $y' \in B'$.

Proof We begin with an overview of the argument. Our goal is to identify a node v in T_i and fix one round of communication in the original $t - i$ round cell-probe communication protocol to obtain a new $t - i - 1$ round communication protocol that still works well in the subtree rooted at v.

We first focus attention on the message the Querier sends during the first round of communication for each of its possible input queries (i.e. leaves in T_i). A node is a good candidate for v if there is some message such that a large fraction of the node's children have a leaf among their descendants for which this message is sent. Using Proposition 3.3 (Ajtai's Lemma), we show that there is always a level with many nodes that are good candidates for v.

Next, we select v from among these nodes and fix the values of those leaves which are not descendants of v so that the following property holds. If the Responder's string y is chosen according to the distribution $\mathcal{Z}_i$, the probability that v is one of the r nodes chosen on its level, $y \in B$, and y is consistent with the fixed values of the leaves not in v's subtree is not too much smaller than the probability that $y \in B$. Moreover, we can fix the response so that the probability that y is also one on which the Responder gives this response is not too much smaller.

If v is one of the nodes chosen on its level, then only the leftmost descendants of its children may have values other than $\perp$. This implies that the answer to the prefix problem is the same if any of the leaves in the subtree rooted at a given child of v is the input query. Thus we can identify the portion of the string y below v with an element in $Z(N_{i+1}, n_{i+1})$, and the input query below v with the position of its ancestor among the children of v.

Now, we proceed to give the details of our proof.

Finding the node v

We examine the behaviour of the Querier during the first round of the original cell-probe communication protocol to find a set of candidates for the node v. For each value of $j \in A$, the Querier sends one of m messages indicating which of the m memory cells it wishes to to probe. Colour the jth leaf of T_i with this message.

To find the node v, we first find a level ℓ of T_i with many α-dense nodes. Then we fix the $r(\ell - 1)$ nodes chosen at higher levels and show that, with high probability, many of the r nodes selected at level ℓ are α-dense. We fix the choice of a set V of $\lceil b^2/8 \rceil$ α-dense nodes at level ℓ. Next we fix the levels of leaves that are not descendants of V. All other leaves, except for the leftmost descendant of each of the N_{i+1} children of each node in V are set to $\perp$. Finally, for each node in V, we consider the restriction of B projected onto

the remaining unfixed leaves of the subtree rooted at the node and choose the node that contains the highest density of elements from $\mathcal{Z}_{i+1}$.

Since $|A| \geq \alpha N_i$, it follows from Proposition 3.3 (Ajtai's Lemma) that there exists a level ℓ such that $1 \leq \ell \leq u$ and the fraction of α-dense nodes in level ℓ of T_i is at least $(\alpha - m\alpha^u)/u$. By the integrality condition, $\alpha \leq 1/2$. Furthermore, $u > 6$ and $m \leq n^k = \alpha^{-4kt} = \alpha^{-\frac{u}{2}}$. Therefore

$$(\alpha - m\alpha^u)/u \geq \alpha(1 - \alpha^{\frac{u}{2}-1})/u > 3\alpha/(4u).$$

We now argue that there is a sufficiently large set of candidates for v among the α-dense nodes at level ℓ and a way of labelling all leaves of T_i that are not descendants of these candidates so that the probability of choosing a string in B remains sufficiently large.

Note that in the construction of $\mathcal{Z}_i$ from $\mathcal{Z}_{i+1}$, the r nodes chosen on level ℓ are not uniformly chosen from among all nodes on level ℓ. The constraint that these nodes not be descendants of any of the $r(\ell-1)$ nodes chosen at higher levels skews this distribution somewhat and necessitates a slightly more complicated argument.

Consider the different possible choices for the $r(\ell-1)$ nodes at levels $1, \ldots, \ell-1$ of T_i in the construction of $\mathcal{Z}_i$ from $\mathcal{Z}_{i+1}$. By simple averaging, there is some such choice with $\mu_{\mathcal{Z}_i'}(B) \geq \beta$, where $\mathcal{Z}_i'$ is the probability distribution obtained from $\mathcal{Z}_i$ conditioned on the fact that this particular choice occurred. Fix this choice.

Let R be the random variable denoting the set of r nodes chosen at level ℓ. Since the choice of nodes at higher levels has been fixed, there are certain nodes at level ℓ that are no longer eligible to be in R. Specifically, each of the r nodes chosen at level $h < \ell$ eliminates its $N_{i+1}^{\ell-h}$ descendants at level ℓ from consideration. In total, there are

$$\sum_{h=1}^{\ell-1} r \cdot N_{i+1}^{\ell-h} < 2r \cdot N_{i+1}^{\ell-1}$$

nodes eliminated from consideration at level ℓ. There are $fN_{i+1}^{\ell-1}$ nodes at level ℓ, so the fraction of nodes at level ℓ that are eliminated is less than $2r/f = \alpha/(4u)$. Thus, of the nodes at level ℓ that have not been eliminated, the subset D of nodes which are α-dense constitutes more than a fraction $3\alpha/(4u) - \alpha/(4u) = \alpha/(2u)$.

We may view the random choice R of the r nodes at level ℓ as being obtained by choosing r nodes randomly, without replacement, from the set of nodes at level ℓ that were not eliminated. Applying Proposition 3.1 with $\rho = \alpha/(2u)$ and $|R| = r$,

$$Pr[|D \cap R| \leq r\alpha/(8u)] < 2^{-r\alpha/(4u)} = 2^{-4b}.$$

Since $b \geq 1$, this probability is smaller than $2^{-2b} = \beta/2$. Let E be the event that at least $r\alpha/(8u) = 2b$ of the r elements of R are α-dense. Then $\mu_{\mathcal{Z}_i''}(B) \geq \beta - \beta/2 = \beta/2$, where $\mathcal{Z}_i''$ is the probability distribution obtained from $\mathcal{Z}_i'$ conditioned on the fact that event E occurred.

Assume that event E has occurred. Then $|D \cap R| \geq 2b$. Let V be the random variable denoting the first $2b$ nodes chosen for R that are also in D. By simple averaging, there is some choice for V with $\mu_{\mathcal{Z}_i'''}(B) \geq \beta/2$, where $\mathcal{Z}_i'''$ is the probability distribution obtained from $\mathcal{Z}_i''$ conditioned on the fact that this particular choice for V occurred. Fix this choice.

Finally, consider the different possible choices σ for the sequence of labels on those leaves which are not descendants of nodes in V. By simple averaging, there is some choice for σ with $\mu_{\mathcal{Z}_i^*}(B) \geq \beta/2$, where $\mathcal{Z}_i^*$ is the probability distribution obtained from $\mathcal{Z}_i'''$ conditioned on the fact that this particular choice for σ occurred. Fix this choice.

By construction, the distribution $\mathcal{Z}_i^*$ is isomorphic to a cross-product of $2b$ independent distributions, $\mathcal{Z}_{i+1}$, one for each of the nodes in V. Specifically, for each $v \in V$, the string consisting of the concatenation of the labels of the leftmost descendants of v's children is chosen from $\mathcal{Z}_{i+1}$. (All other descendants of v are labelled by $\perp$.) For $v \in V$ and y chosen from $\mathcal{Z}_i^*$, let $\pi_v(y)$ denote the string consisting of the N_{i+1} characters of y labelling the leftmost descendants of the N_{i+1} children of v. Let $B_v = \{\pi_v(y) \mid y \in B \text{ is consistent with } \sigma\}$. Then

$$\beta/2 \leq \mu_{\mathcal{Z}_i^*}(B) \leq \prod_{v \in V} \mu_{\mathcal{Z}_{i+1}}(B_v).$$

Hence, there is some $v \in V$ such that

$$\mu_{\mathcal{Z}_{i+1}}(B_v) \geq (\mu_{\mathcal{Z}_i^*}(B))^{1/|V|} \geq (\beta/2)^{1/2b} = 1/2.$$

Choose that node v.

Fixing a round of communication for each player

Since v is α-dense, there is some message c that the Querier may send in the first round such that $|A'|/N_{i+1} \geq \alpha$, where

$$A' = \{j' \in [1, N_{i+1}] \mid \text{the } j'\text{-th child of } v \text{ is coloured } c\};$$

i.e., there is some input j corresponding to a descendant of the j'-th child of v on which the Querier sends message c in the first round. We fix the message sent by the Querier in the first round to be c.

For each node $v \in V$ and string $y \in B$, let $\lambda_v(y) \in \Sigma^*$ denote the string consisting of the non-$\perp$ characters of y labelling the leaves of T_i that occur to the left of the subtree rooted at v. For each state p of the deterministic finite automaton M, let $B_{v,p}$ denote the set of strings $y' \in B_v$ for which there exists $y \in B$ consistent with σ such that $\pi_v(y) = y'$ and $x \cdot \lambda_v(y)$ takes M from its initial state to state p. Since B_v is the (not necessarily disjoint) union of the q sets $B_{v,p}$, there is a state p' such that

$$\mu_{\mathcal{Z}_{i+1}}(B_{v,p'}) \geq \mu_{\mathcal{Z}_{i+1}}(B_v)/q \geq 1/(2q).$$

Fix any function $\iota : B_{v,p'} \to B$ so that, for each string $y' \in B_{v,p'}$, $\iota(y')$ is consistent with σ, $\pi_v(\iota(y')) = y'$, and $x \cdot \lambda_v(\iota(y'))$ takes M from its initial state to state p'. In other words, $\iota(y')$ witnesses the fact that $y' \in B_{v,p'}$.

There are only 2^b different messages the Responder can now send. Therefore, there is some fixed message c' for which

$$\mu_{\mathcal{Z}_{i+1}}(B') \geq 1/(2q2^b) \geq 2^{-2b+1} = \beta,$$

since $2^b \geq 4q$, where B' is the set of strings $y' \in B_{v,p'}$ such that, in round one, given the input $\iota(y')$ and the query c, the Responder sends c'. We fix the message sent by the Responder in the first round to be c'.

Constructing the $t-i-1$ round protocol

Choose $x' \in \Sigma^*$ to be any fixed word that takes M from its initial state to state p'.

Consider the following new $t-i-1$ round protocol: Given inputs $j' \in A'$ and $y' \in B'$, the Querier and the Responder simulate the last $t-i-1$ rounds of the original $t-i$ round protocol, using inputs $j \in A$ and $y = \iota(y') \in B$, respectively, where j is the index of some leaf in T_i with colour c that is a descendant of the j'-th child of node v. Note that it doesn't matter which leaf of colour c in the subtree rooted at the j'-th child of v is chosen. This is because every leaf in this subtree, except the leftmost leaf, is labelled by $\perp$, so $\text{PRE}_j(y)$ is the same no matter which leaf in the subtree is indexed by j.

It follows from the definitions of A' and B' that, for inputs j and y, the original protocol will send the fixed messages c and c' during round one. By construction, the new protocol determines whether $x \cdot PRE_j(y) \in L$.

Since $\pi_v(y) = y'$ and j is the index of a leaf in T_i that is a descendant of the j'th child of node v, $\mathrm{PRE}_j(y) = \lambda_v(y) \cdot \mathrm{PRE}_{j'}(y')$. Furthermore $x \cdot \lambda_v(y)$ and x' both lead to the same state p' of M, so $x \cdot PRE_j(y) = x \cdot \lambda_v(y) \cdot \mathrm{PRE}_{j'}(y') \in L$ if and only if $x' \cdot \mathrm{PRE}_{j'}(y') \in L$. Thus the new protocol determines whether $x' \cdot \mathrm{PRE}_{j'}(y') \in L$. □

We now combine Lemma 3.4 and Lemma 3.5 to prove the main technical result.

Theorem 3.6: Let $L \subseteq \Sigma^*$ be an indecisive regular language accepted by a deterministic finite automaton with q states and let $x \in \Sigma^*$. There is a constant $c > 0$ such that if $b \geq \max(16, 2(|\Sigma|+1)^q)$, $(ckb^2t)^{4t} \leq n \leq (ckb^2t)^{8t}$, and $N \geq n^{(ckt)^t}$, then there is no t round cell-probe communication protocol for the static (L,N,n,x)-prefix problem using n^k memory cells of b bits each.

Proof Let c=256. Suppose that $b \geq \max(16, 2(|\Sigma|+1)^q)$, $(ckb^2t)^{4t} \leq n \leq (ckb^2t)^{8t}$, and $N \geq n^{(ckt)^t}$. To obtain a contradiction, suppose that there is a t round cell-probe communication protocol for the static (L,N,n,x)-prefix problem using $m \leq n^k$ memory cells of b bits. Let $n' = (ckb^2t)^{4t} \leq n$ and $k' = 2k$. Then $n' \leq n$ and $m \leq (n')^{k'}$.

Let $u = 8k't$, $\alpha = (n')^{-1/(4t)}$, $r = 16bu/\alpha$, $f = 128bu^2/\alpha^2$, $N_0 = N$, $n'_0 = n'$, and let $N_{i+1} = (N_i/f)^{1/u}$ and $n'_{i+1} = n'_i/(ru)$ for $i = 0, \ldots, t-1$.

Note that $f = 128b \times 64(k')^2t^2(n')^{1/2t} \leq n$. One can now easily check that since $N \geq n^{(256kt)^t}$, $N_t \geq n \geq n_t$. As noted above, $N_t \geq N/(f^{u^t})$.

Therefore (b, k', t, N, n') satisfies the integrality condition and the algorithm works correctly for all inputs $j \in A = [1,N]$ and $S \in B = Z(N,n')$. Since $2^b \geq b \geq 2^{q+1} \geq 4q$, $b \geq 2$, Lemma 3.5 can be applied t times starting with $A = [1,N]$ and $B = Z(N,n)$, to obtain $x' \in \Sigma^*$, $A' \subseteq [1,N_t]$ with $|A'| \geq \alpha N_t$, $B' \subseteq Z(N_t,n_t)$ with $\mu_{Z_t}(B') \geq \beta = 2^{-2b+1}$, and a 0 round cell-probe communication protocol that correctly determines whether $x' \cdot \mathrm{PRE}_j(y) \in L$ for all $j \in A'$ and $y \in B'$. This implies that $x \cdot PRE_j(y) \in L$ for all $j \in A'$ and $y \in B'$ or $x' \cdot PRE_j(y) \notin L$ for all $j \in A'$ and $y \in B'$.

Since $\alpha = (n')^{-1/(4t)}$ and $t \geq 1$, $\alpha^{1+t} \geq (n')^{-1/2}$. Also, $n' \geq (256kb^2t)^{4t}$ and $b \geq 4q$, so

$$\alpha n'_t = \frac{\alpha n'}{(ru)^t} = \frac{n'\alpha^{1+t}}{(16bu^2)^t} \geq \frac{(n')^{1/2}}{16b(8k't^2)^t} \geq \frac{(256b^2kt)^{2t}}{b^t(32k't)^{2t}} = 4^{2t}b^{3t} \geq 16b^3 > 8qb^2$$

since $t \geq 1$. But $N_t \geq n_t > 0$, $b \geq 2(|\Sigma|+1)^q$, and $\beta = 2^{-2b+1}$. Therefore, by Lemma 3.4, there exist integers $a, a' \in A'$ and a string $y \in B'$ such that $x' \cdot PRE_a(y) \in L$ and $x' \cdot PRE_{a'}(y) \notin L$. This is a contradiction. □

The following two results, which are direct consequences of the preceding result, give us the desired lower bounds for the static prefix problem.

Theorem 3.7: For any indecisive regular language L, any string x, any positive integer k, and any positive constant ε, there exists a function $n(N) \leq N$ such that any cell probe data structure for the static $(L,N,n(N),x)$-prefix problem using $(n(N))^k$ memory cells of $2^{(\log N)^{1-\varepsilon}}$ bits requires time $\Omega(\log\log N/\log\log\log N)$ per query.

Proof Fix $k, \varepsilon > 0$ and choose the largest integer t such that $(\log N)^\varepsilon \geq (ckt)^{4t}$ where c is the constant from Theorem 3.6. Clearly $c' \log\log N/\log\log\log N \geq t \geq c'' \log\log N/\log\log\log N$ for some constants $c', c'' > 0$

depending only on k and ε (and the constant c). Let $b = 2^{(\log N)^{1-\varepsilon}}$ and set $n = (cb^2kt)^{4t}$. Then, for N sufficiently large, $b \geq \max(16, 2(|\Sigma|+1)^q)$ and

$$n^{(ckt)^t} \leq n^{(\log N)^{\varepsilon/4}} \leq \left[(\log N)^{\varepsilon} 2^{8c'(\log N)^{1-\varepsilon}\frac{\log\log N}{\log\log\log N}}\right]^{(\log N)^{\varepsilon/4}} < N.$$

Therefore, by Theorem 3.6, any cell-probe data structure for the static (L, N, n, x)-prefix problem requires time at least $t+1$ using n^k memory cells of b bits each. □

Theorem 3.8: For any indecisive regular language L, any string x, and any positive integers k, k', there is a function $N(n)$ such that any cell probe data structure for the static $(L, N(n), n, x)$-prefix problem using n^k memory cells of $(\log N(n))^{k'}$ bits requires time $\Omega(\sqrt{\log n / \log\log n})$ per query.

Proof Fix k and k' and consider the largest integer t for which $n \geq (ckt)^{8k't^2+4t}(\log n)^{8k't}$ where c is the constant from Theorem 3.6. Then $t \geq c'\sqrt{\log n/\log\log n}$ where $c' > 0$ is a constant depending only on k and k' (and the constant c). Set $N = n^{(ckt)^t}$ and $b = (\log N)^{k'} = [(ckt)^t \log n]^{k'}$, so $n \geq (ckt)^{8k't^2+4t}(\log n)^{8k't} = (cb^2kt)^{4t}$. For all $t \geq 3$, $(t+1)^{t+1} \leq t^{2t}$ and, hence by the choice of t, it follows that $n \leq (cb^2kt)^{8t}$. For N sufficiently large, $b \geq \max(16, 2(|\Sigma|+1)^q)$ Thus, by Theorem 3.6, any cell-probe data structure for the static (L, N, n, x)-prefix problem requires time at least $t+1$ using n^k memory cells of b bits each. □

It follows from the reductions described in Sections 3.1 and 3.2 that the static predecessor problem is at least as hard as any static prefix problem. Thus, we obtain the following corollaries.

Corollary 3.9: Consider any cell probe data structure for the static predecessor problem that stores each set S from a universe of size N using $|S|^{O(1)}$ memory cells of $2^{(\log N)^{1-\Omega(1)}}$ bits. Then, in the worst case, queries take $\Omega(\log\log N/\log\log\log N)$ time.

A result similar to Corollary 3.9 was independently shown by Xiao [46].

As noted in Section 2, the static predecessor problem for a set S from a universe of size N can be solved in constant time in the cell probe model using $O(|S|)$ memory cells if $|S| \in (\log N)^{O(1)}$ [1]. With indirect addressing, it can also be solved in constant time using $|S|^{O(1)}$ memory cells if $N \in |S|^{O(1)}$. Thus, in Corollary 3.9, the worst case occurs for a set S of size $(\log N)^{\omega(1)}$ and $N^{o(1)}$.

Corollary 3.10: Consider any cell probe data structure for the static predecessor problem. If each set of size n from a universe of size N is stored using $n^{O(1)}$ memory cells of $(\log N)^{O(1)}$ bits. Then, in the worst case, queries take $\Omega(\sqrt{\log n/\log\log n})$ time.

The reductions in Sections 3.1 and 3.2 also imply that the same lower bounds apply to the static versions of the predecessor parity problem, the point separation problem, the rank problem, and the exact range counting problem.

3.5 Lower Bounds for Dynamic Problems

One can apply the results of Section 3.4 to obtain lower bounds for dynamic data structures, using a translation argument given by Miltersen [35].

For $a \in \Sigma \cup \{\perp\}$ and $i \in [1, N]$, the operation update(i, a) applied to a string $y \in Z(N, n)$ updates the i'th letter of the string to be a. It can only be applied if the resulting string is also in $Z(N, n)$.

DEFINITION 3.7: Let $L \subseteq \Sigma^*$ and let $x \in \Sigma^*$. The *dynamic* (L,N,n,x)*-prefix problem* is to maintain a string $y \in Z(N,n)$ under update operations and support the queries "Is $x \cdot \mathrm{PRE}_j(y) \in L$?", for all $j \in [1,N]$. The *semi-dynamic* (L,N,n,x)*-prefix problem* is a restricted version of the dynamic (L,N,n,x)-prefix problem in which the operation update(i,a) can only be applied when the i'th letter of the string is $\perp$ and $a \neq \perp$.

Because the semi-dynamic prefix problem is a restricted version of the dynamic prefix problem, lower bounds for the former also apply to the latter.

Theorem 3.11: Consider any cell-probe data structure for the semi-dynamic (L,N,n,x)-prefix problem. Suppose there are $b = (\log N)^{O(1)}$ bits per memory cell and the data structure uses $2^{O(b)}$ memory cells. If the amortized time for updates is $n^{O(1)}$, then the worst-case query time is not $o\left(\sqrt{\log n / \log\log n}\right)$.

Proof Suppose there is a data structure such that the amortized time for updates is $n^{O(1)}$. Using static dictionary techniques from [26], we obtain a space efficient solution to the static (L,N,n,x)-prefix problem.

Let $y \in Z(N,n)$ and consider the configuration of the data structure that results from updating the string $\perp^N$ to become y, one character at a time. At each time step, at most one memory cell of the data structure is changed, so in n updates, $n^{O(1)}$ memory cells are changed. Suppose that the indices of all these changed memory cells, together with their new values, are stored in a perfect hash table that uses $n^{O(1)}$ memory cells, each containing b bits.

To determine whether $x \cdot \mathrm{PRE}_j(y) \in L$, given $j \in [1,N]$, it suffices to simulate the query algorithm used by the dynamic data structure. Each probe to a location i in the dynamic data structure is simulated by searching the hash table for the key i. If it is present, the associated value is used; otherwise, the value of location i in the initial (empty) state of the dynamic data structure is used. Since the simulation of a single probe can be done in constant time, the worst case query time in the static data structure is $O(t)$, where t is the worst case query time in the semi-dynamic data structure.

It follows from Theorem 3.8 that t is not $o\left(\sqrt{\log n / \log\log n}\right)$. □

The restriction of $2^{O(b)}$ on the number of memory cells is reasonable, since it is the number of different cells that can be accessed when performing indirect addressing.

Exactly the same proofs apply to the dynamic predecessor problem with inserts instead of updates.

Theorem 3.12: Consider any cell-probe data structure for the semi-dynamic predecessor problem (i.e. no delete operations are performed) on $[1,N]$ restricted to sets of size at most n. Suppose there are $b = (\log N)^{O(1)}$ bits per memory cell and the data structure uses $2^{O(b)}$ memory cells. If the amortized time for inserts is $n^{O(1)}$, then the worst-case query time is not $o(\sqrt{\log n / \log\log n})$.

Similar lower bounds, but for even larger word size, can be obtained in terms of the universe size.

Theorem 3.13: Consider any cell-probe data structure for the semi-dynamic (L,N,n,x)-prefix problem. Suppose there are $b = 2^{(\log N)^{1-\Omega(1)}}$ bits per memory cell and the data structure uses $2^{O(b)}$ memory cells. If the amortized time for updates is $2^{(\log N)^{1-\Omega(1)}}$, then the worst-case query time is $\Omega(\log\log N / \log\log\log N)$.

Proof The proof is the same as for Theorem 3.11, except that we need to track the parameters slightly differently and use Theorem 3.7 instead of Theorem 3.8.

Fix some $\varepsilon > 0$ and suppose that $b = 2^{(\log N)^{1-\varepsilon}}$. If the amortized time per update is at most $2^{(\log N)^{1-\varepsilon}}$, then the time to perform n updates is $O(nb)$. As in the proof of Theorem 3.11, a hash table containing $O(nb)$ entries of b bits each can be constructed. Now, let $k = 2$ and set n to the value chosen in the proof of Theorem 3.7 as a function of b and N and k. Since $n \geq b$, the hash table has size $O(n^2)$ and we obtain a static data structure that meets the conditions of Theorem 3.7. Therefore it has worst-case query time $\Omega(\log\log N / \log\log\log N)$. □

Theorem 3.14: Consider any cell-probe data structure for the dynamic predecessor problem on $[1, N]$ restricted to sets of size at most n on which no delete operations are performed. Suppose there are $b = 2^{(\log N)^{1-\Omega(1)}}$ bits per memory cell and the data structure uses $2^{O(b)}$ memory cells. If the amortized time for inserts is $2^{(\log N)^{1-\Omega(1)}}$, then the worst-case query time is $\Omega(\log\log N / \log\log\log N)$.

When there is no a priori restriction on the number of memory cells used by the dynamic data structure, we can obtain lower bounds for worst-case query time given an upper bound on the worst-case, rather than the amortized, update time.

Theorem 3.15: Consider any cell-probe data structure for the semi-dynamic (L, N, n, x)-prefix problem. Suppose there are $b = 2^{(\log N)^{1-\Omega(1)}}$ bits per memory cell and the worst-case time for updates is $2^{(\log N)^{1-\Omega(1)}}$. Then the worst-case query time is $\Omega(\log\log N / \log\log\log N)$.

Proof Let $\varepsilon > 0$ and suppose there is a data structure for the semi-dynamic (L, N, n, x)-prefix problem such that the worst-case time T per update and the number of bits b per memory cell are both bounded above by $2^{(\log N)^{1-\varepsilon}}$. If $L \subseteq \Sigma^*$, there are only $(|\Sigma| + 1)N$ possible different updates. At each of the at most T steps in the update algorithm for a given update, there are at most 2^b ways to branch (depending on the value controlling the branch). Hence, the total number of different memory cells that can be accessed during any update is at most $(|\Sigma| + 1)N2^{bT}$, which is bounded above by $2^{2^{(\log N)^{1-\varepsilon/2}}}$ for N sufficiently large. Therefore it takes at most $b' = 2^{(\log N)^{1-\varepsilon/2}}$ bits to describe which cell is updated at each step.

We now apply the same hash table construction as in Theorems 3.11 and 3.13 to solve the static (L, N, n, x)-prefix problem. We get a table with at most $nT = O(nb')$ entries each of b' bits. The remainder of the proof proceeds as in the proof of Theorem 3.13 with b' replacing b. □

Theorem 3.16: Consider any cell-probe data structure for the semi-dynamic (L, N, n, x)-prefix problem. Suppose there are $b = (\log N)^{O(1)}$ bits per memory cell and the worst-case time for updates is $2^{O(\sqrt{\log n})}$. Then the worst-case query time is not $o\left(\sqrt{\log n / \log\log n}\right)$.

Proof Consider a cell-probe data structure for the semi-dynamic (L, N, n, x)-prefix problem with $b = (\log N)^{k'}$ bits per memory cell and worst-case update time $T = 2^{O(\sqrt{\log n})}$.

As in the proof of Theorem 3.15, the number of different possible memory cells accessed during an update is at most $(|\Sigma| + 1)N2^{bT}$. Thus $(\log N)^{O(1)}2^{O(\sqrt{\log n})}$ bits are needed to represent which memory cell is updated.

Let $k = 2$. Following the proof of Theorem 3.8, choose $N = n^{(ckt)^t}$, where c is the constant from the proof of Theorem 3.6 and t is $\Theta(\sqrt{\log n / \log\log n})$. Then $\log N = 2^{\Omega(\sqrt{\log n})}$.

Therefore, for this value of N, only $(\log N)^{O(1)}$ bits are needed to describe which memory cell is updated at each step of the dynamic algorithm. The hash table construction results in a data structure for the static

(L, N, n, x) prefix problem using $nT = O(n^2)$ memory cells of $(\log N)^{k'}$ bits each, for some constant $k' > 0$. A query can be performed by simulating each step of the query algorithm for the semi-dynamic data structure in constant time. From Theorem 3.8, it follows that the worst-case query time for the resulting static data and, hence, for the semi-dynamic data structure is at least $c'\sqrt{\log n / \log\log n}$ for some positive constant c'. □

There are analogous lower bounds for the semi-dynamic predecessor problem.

4 An Optimal Static Predecessor Data Structure

This section presents a new data structure for the static predecessor problem that matches the lower bounds for query time in Section 3.4 to within a constant factor. The previous best static predecessor data structures all use one of Williard's small space variants of van Emde Boas trees to do substantial range reduction. The essential idea of this range reduction is that one is able to perform a binary search on the binary representation of the query element to find the portion that is most relevant for determining its predecessor.

Our key contribution is to replace most of the uses of this binary search by a new, faster, multi-way search that either reduces the number of bits in the relevant portion of the binary representation by a large factor or significantly reduces the number of elements under consideration. This technique was motivated by our lower bound work: Our first algorithm was for the restricted class of inputs used in our lower bound proof and these inputs provided us with key intuition.

At each round of the binary search procedure, Willard uses a perfect hash table. This table stores the nodes at the middle level of the binary trie representing the relevant bits of the subset still under consideration. To implement our multi-way search, we replace the binary search with a method for parallel hashing that examines several different levels of the binary trie at once. Because of limits on the sizes of the tables we can accommodate, we are only be able to represent some of the nodes at these different levels of the binary trie, namely, those with many leaves in their subtrees. The missing nodes mean that we sometimes get a substantial reduction in the size of the set under consideration, instead of always obtaining a substantial range reduction.

We begin by describing our method for parallel hashing. Next, we explain how to implement one round of multi-way search. Then we build a data structure for the predecessor problem when the set size, universe size, and word size are in certain ranges. Finally, we incorporate previous techniques and optimize parameters to obtain a data structure for any range of parameters.

Throughout this section,

- the symbol $\langle c \rangle$ denotes the binary string of length k representing $c \in [0, 2^k - 1]$,
- the symbol $\langle\langle c \rangle\rangle$ denotes the binary string of length $2k$ representing $c \in [0, 2^{2k} - 1]$, and
- the symbol $\prec c \succ$ denotes the binary string of length $r+1$ representing $c \in [0, 2^{r+1} - 1]$.

A string is stored right justified in a word (or a constant number of words), padded with 0's on the left, as necessary. For example, if there are $b \geq lk$ bits per word, a string of length l over the alphabet $[0, 2^k - 1]$ is stored as $b - lk$ zero bits followed by the concatenation of the k-bit binary representations of each of the l letters.

We describe the space requirements of some of our data structures in terms of the total number of bits used, B, and a lower bound on the number of bits, b, in a single word of memory. For any b that is at least the lower bound, the data structure can be configured to use $O(B/b)$ words.

Parallel Hashing

We begin by showing that, if the word length b is sufficiently large, it is possible to evaluate q independent linear space perfect hash functions in parallel in constant time and, hence, perform membership queries in q dictionaries in parallel in constant time.

Lemma 4.1: Let $S_1, \ldots, S_q \subseteq [0, 2^k - 1]$ be sets of size at most 2^r, where $r < k$. If memory words contain $b \in \Omega(kq^2)$ bits, then there is an $O(kq2^{(r+1)q})$-bit data structure that can be constructed in $O(kq2^{(r+1)q})$ time and supports q parallel membership queries $z_1 \in S_1?, \ldots, z_q \in S_q?$ in constant time.

Proof Let $i \in [1, q]$. Since S_i contains at most 2^r elements, it is possible, in $O(k2^{2r})$ time, to (deterministically) construct a two-level hash function $h_i : [0, 2^k - 1] \longrightarrow [0, 2^{r+1} - 1]$ which is one-to-one on S_i [40, 20, 25]. More specifically, it is possible to find constants $a_i, a_{i,0}, \ldots, a_{i,2^{r+1}-1} \in [0, 2^k - 1]$, $p_{i,0}, \ldots, p_{i,2^{r+1}-1} \in [0, 2^{r+1} - 1]$, and $r_{i,0}, \ldots, r_{i,2^{r+1}-1} \in [0, r]$ and functions $f_i : [0, 2^k - 1] \longrightarrow [0, 2^{r+1} - 1]$ and $g_{i,j} : [0, 2^k - 1] \longrightarrow [0, 2^{r_{i,j}} - 1]$, for $j = 0, \ldots, 2^{r+1} - 1$, such that

$$\begin{aligned} f_i(x) &= a_i x \bmod 2^k \operatorname{div} 2^{k-1-r}, \\ g_{i,j}(x) &= a_{i,j} x \bmod 2^k \operatorname{div} 2^{k-r_{i,j}} \text{ for } j = 0, \ldots, 2^{r+1} - 1, \text{ and} \\ h_i(x) &= p_{i,f_i(x)} + g_{i,f(x)}(x). \end{aligned}$$

The data structure consists of the $(2q-1)k$-bit string $a = \langle a_q \rangle \langle 0 \rangle \langle a_{q-1} \rangle \langle 0 \rangle \cdots \langle 0 \rangle \langle a_1 \rangle$ and four $2^{(r+1)q}$ element arrays A, R, P, and M, where, for $j_1, \ldots, j_q \in \{0,1\}^{r+1}$,

- $A[\prec j_1 \succ \cdots \prec j_q \succ] = \langle a_{q,j_q} \rangle \langle 0 \rangle \cdots \langle 0 \rangle \langle a_{1,j_1} \rangle \in \{0,1\}^{k(2q-1)}$,
- $R[\prec j_1 \succ \cdots \prec j_q \succ] = \langle 2^{r_{q,j_q}} \rangle \langle 0 \rangle \cdots \langle 0 \rangle \langle 2^{r_{1,j_1}} \rangle \in \{0,1\}^{k(2q-1)}$,
- $P[\prec j_1 \succ \cdots \prec j_q \succ] = \prec p_{1,j_q} \succ \cdots \prec p_{q,j_q} \succ \in \{0,1\}^{(r+1)q}$, and
- $M[\prec j_1 \succ \cdots \prec j_q \succ] = \langle x_1 \rangle \cdots \langle x_q \rangle \in \{0,1\}^{kq}$, such that, for $i = 1, \ldots, q$, either $h_i(x_i) = j_i$ and $x_i \in S_i$ or $h_i(x_i) \neq j_i$ and $j_i \notin h_i(S_i)$.

This data structure can be stored using $(2q-1)k + 2k(2q-1)2^{(r+1)q} + (r+1)q2^{(r+1)q} + kq2^{(r+1)q} \in O(kq2^{(r+1)q})$ bits. The time to construct the data structure is $O(kq2^{2r})$ for finding the q two-level hash functions and $O(2^{(r+1)q})$ for constructing the arrays.

Given a word Z containing the string $\langle z_1 \rangle \cdots \langle z_q \rangle \in (\{0,1\}^k)^q$ the function $h(z_1, \ldots, z_q) = (h_1(z_1), \ldots, h_q(z_q))$ can be computed in constant time as follows.

- Expand Z so that there are $(2q-1)k$ bits with value 0 between each k-bit character of Z. To do this, multiply Z by the string $(\langle 0 \rangle^{2q-2} \langle 1 \rangle)^q$ to give the string $(\langle 0 \rangle^{q-1} \langle z_1 \rangle \cdots \langle z_q \rangle)^q$. Then perform a bitwise AND with the string $(\langle 0 \rangle^{2q-1} 1^k)^q$ to obtain the string $W = \langle z_1 \rangle \langle 0 \rangle^{2q-1} \langle z_2 \rangle \langle 0 \rangle^{2q-1} \cdots \langle 0 \rangle^{2q-1} \langle z_q \rangle$.

- Multiply $z_1,\ldots,z_q$ by the constants $a_1,\ldots,a_q$, respectively. This is done by multiplying W by the string a to obtain the string

$$\langle\langle a_q \times z_1\rangle\rangle \cdots \langle\langle a_1 \times z_1\rangle\rangle \langle\langle a_q \times z_2\rangle\rangle \cdots \langle\langle a_1 \times z_{q-1}\rangle\rangle \langle\langle a_q \times z_q\rangle\rangle \cdots \langle\langle a_1 \times z_q\rangle\rangle \in \{0,1\}^{2kq^2}.$$

  Then, perform a bitwise AND with the string $(0^k1^{r+1}0^{k-r-1}\langle\langle 0\rangle\rangle^{q-2})^q\langle\langle 0\rangle\rangle$, to obtain $F' = \prec f_1(z_1) \succ 0^{2k(q-1)-r-1} \prec f_2(z_2) \succ 0^{2k(q-1)-r-1} \cdots \prec f_{q-1}(z_{q-1}) \succ 0^{2k(q-1)-r-1} \prec f_q(z_q) \succ 0^{2k(q-1)+k-r-1}$.

- Compress F' to get $F = \prec f_1(z_1) \succ \prec f_2(z_2) \succ \cdots \prec f_q(z_q) \succ \in \{0,1\}^{(r+1)q}$. To do this, multiply F' by the bit string $(0^{2k(q-1)-r-2}1)^q$ and then AND the result with the bit string $1^{q(r+1)}0^{(2q(q-1)+1)k-q(r+1)}$ to obtain the bit string $\prec f_1(z_1) \succ \cdots \prec f_q(z_q) \succ 0^{(2q(q-1)+1)k-q(r+1)}$. Finally, shift the result right $(2q(q-1)+1)k-q(r+1)$ bit positions.

- Multiply $z_1,\ldots,z_q$ by the values $a_{1,f_1(z_1)},\ldots,a_{q,f_q(z_q)}$, respectively. This is done by multiplying W by $A[F]$, which gives

$$\langle\langle a_{q,f_q(z_q)} \times z_1\rangle\rangle \cdots \langle\langle a_{1,f_1(z_1)} \times z_1\rangle\rangle \cdots \langle\langle a_{q,f_q(z_q)} \times z_q\rangle\rangle \cdots \langle\langle a_{1,f_1(z_1)} \times z_q\rangle\rangle.$$

  Then, perform a bitwise AND with the string $(0^k1^k\langle\langle 0\rangle\rangle^{q-2})^q\langle\langle 0\rangle\rangle$, to get

$$\langle 0\rangle\langle z'_1\rangle\langle\langle 0\rangle\rangle^{q-2}\langle 0\rangle\langle z'_2\rangle\langle\langle 0\rangle\rangle^{q-2}\cdots\langle 0\rangle\langle z'_{q-1}\rangle\langle\langle 0\rangle\rangle^{q-2}\langle 0\rangle\langle z'_q\rangle\langle\langle 0\rangle\rangle^{q-1},$$

  where $z'_i = a_{i,f_i(z_i)}z_i \bmod 2^k$. Compress this string by multiplying it by $(\langle\langle 0\rangle\rangle^{q-2}\langle 1\rangle)^q$, taking the AND with $1^{qk}0^{(2q-1)(q-1)k}$, and shifting right $(2q-1)(q-1)k$ bit positions to obtain $Z' = \langle z'_1\rangle\langle 0\rangle\cdots\langle 0\rangle\langle z'_q\rangle$,

- Multiply $z'_1,\ldots,z'_q$ by the values $2^{r_{1,f_1(z_1)}},\ldots,2^{r_{q,f_q(z_q)}}$, respectively. As in the first two steps, this is done by multiplying Z' by $(\langle 0\rangle^{2(q-1)}\langle 1\rangle)^q$, performing a bitwise AND with the string $(\langle 0\rangle^{2q-1}1^k)^q$, and multiplying by $R[F]$, which gives

$$\langle\langle 2^{r_{q,f_q(z_q)}} \times z'_1\rangle\rangle \cdots \langle\langle 2^{r_{1,f_1(z_1)}} \times z'_1\rangle\rangle \cdots \langle\langle 2^{r_{q,f_q(z_q)}} \times z'_q\rangle\rangle \cdots \langle\langle 2^{r_{1,f_1(z_1)}} \times z'_q\rangle\rangle.$$

  Then, perform a bitwise AND with the string $(1^k0^k\langle\langle 0\rangle\rangle^{q-2})^q\langle\langle 0\rangle\rangle$, to obtain

$$G' = \langle g'_1\rangle\langle 0\rangle\langle\langle 0\rangle\rangle^{q-2}\langle g'_2\rangle\langle 0\rangle\langle\langle 0\rangle\rangle^{q-2}\cdots\langle g'_{q-1}\rangle\langle 0\rangle\langle\langle 0\rangle\rangle^{q-2}\langle g_q\rangle\langle 0\rangle\langle\langle 0\rangle\rangle^{q-1},$$

  where $g'_i = g_{i,f_i(z_i)}(z_i) = z'_i \operatorname{div} 2^{k-r_{i,f_i(z_i)}} \in \{0,1\}^{r_{i,f_i(z_i)}}$. Note that since $r_{i,f_i(z_i)} < r < k$, it follows that $\langle g'_i\rangle = 0^{k-r-1} \prec g'_i \succ$.

- Compress G' to get $G = \prec g'_1 \succ \cdots \prec g_q \succ \in \{0,1\}^{(r+1)q}$. As above, to do this, multiply G by the bit string $(0^{2k(q-1)-r-2}1)^q$, AND the result with the bit string $1^{q(r+1)}0^{2kq(q-1)+k-(r+1)(q-1)}$ and shift right $2kq(q-1)+k-(r+1)(q-1)$ bit positions.

- Add $P[F] = \prec p_{1,f_1(z_1)} \succ \cdots \prec p_{q,f_q(z_q)} \succ$ and G to obtain $H = \prec h_1(z_1) \succ \cdots \prec h_q(z_q) \succ$.

To complete the membership queries, compare the corresponding k-bit fields of $M[H]$ and Z. Since h_i is one-to-one on S_i, it follows that $z_i \in S_i$ if and only if z_i is in the $(q+1-i)$'th least significant k-bit field of $M[H]$.

To compare the corresponding fields in parallel, compute E, the exclusive OR of $M[H]$ and Z. Next compute the AND of E and $(01^{k-1})^q$ and subtract the result from $(10^{k-1})^q$. Finally, AND the result with $(10^{k-1})^q$ and the complement of E to get the string whose $k(q+1-i)$th least significant bit is 1 if and only if $z_i \in S_i$ for $i = 1,\ldots,q$ and all of whose other bits are 0. □

Multi-way Search in a Trie

Next, we explain the fundamental primitive for our new technique which is based on parallel hashing. Let S denote a set of s strings, $1 < s \leq n$, of length L over the alphabet $[0, 2^k - 1]$ and let T denote the trie of branching factor 2^k and depth L representing this set. Each node at depth d of T corresponds to the length d prefix of some element of S. Thus T contains at most $Ls + 1$ nodes. A node v in T is said to be *n-heavy* (which we will simply refer to as *heavy* since the value of n will always be understood) if the subtrie rooted at v has at least $\max(s/n^{1/L}, 2)$ leaves. Any ancestor of a heavy node is a heavy node. The root of T is always heavy and no leaf is heavy. For $0 < d < L$, there are at most $n^{1/L}$ heavy nodes at depth d.

Lemma 4.2: Let T be a trie of depth L over the alphabet $[0, 2^k - 1]$ with at most n leaves. If $2L(L-1) \leq \log n$ and memory words contain $b \in \Omega(kL^2)$ bits, then there is an $O(kLn)$-bit data structure that can be constructed in $O(kLn)$ time such that, given a string $x = x_1 \cdots x_L$ of length L over $[0, 2^k - 1]$, the longest proper prefix of x that is a heavy node in T, and the length of this prefix, can be determined in constant time.

Proof If $L = 1$, the empty word is the only proper prefix of x, so we may assume that $L > 1$. Let $S_d = \{z \in [0, 2^k - 1] \mid yz \text{ is a heavy node at depth } d \text{ for some heavy node } y \text{ at depth } d - 1\}$, for $0 < d < L$. Since T has at most $n^{1/L}$ heavy nodes at depth d, it follows that S_d has size at most $n^{1/L}$.

We use a slight variant of the data structure in Lemma 4.1, with $q = L - 1$ and $r = \lceil (\log n)/L \rceil$. Specifically, if $M[\prec j_1 \succ \cdots \prec j_q \succ] = \langle y_1 \rangle \cdots \langle y_q \rangle$, then there exists $0 \leq e \leq q$ such that $y_1 \cdots y_e$ is a heavy node. $h_i(y_i) = j_i$ for $1 \leq i \leq e$, and $h_i(y_i) \neq j_i$ for $e < i \leq q$. Furthermore, if $e < q$, then $y_1 \cdots y_e z$ is not a heavy node for all $z \in \{0,1\}^k$ with $h_{e+1}(z) = j_{e+1}$. Note that $y_i \in S_i$ and h_i is one-to-one on S_i for $i = 1, \ldots, e$. Hence, if $h(x_1, \ldots, x_q) = (j_1, \ldots, j_q)$ and $x_1 \cdots x_d$ is the longest proper prefix of x which is a heavy node, then $d \leq e$ and $x_i = y_i$ for $i = 1, \ldots, d$.

Thus, to find the length, d, of the longest proper prefix of x which is a heavy node, it suffices to compute $H = \prec h_1(x_1) \succ \cdots \prec h_q(x_q) \succ$ and find the length of the longest common prefix of $M[H]$ and $X = \langle x_1 \rangle \cdots \langle x_q \rangle$. This can be computed by computing E, the exclusive OR of $M[H]$ and X, computing the prefix OR of E, taking the AND with $\langle 1 \rangle^q$, then computing the Hamming weight of the resulting string (which can be done by multiplying the string by $\langle 1 \rangle^q$, shifting right $k(q-1)$ bits, and taking the AND with the string 1^k), and finally subtracting the result from q. Alternatively, look up the prefix OR of E in a hash table of size $O(kq)$ containing the strings $\{0^{kq-i}1^i \mid i = 0, \ldots, kq\}$ stored together with the corresponding values of d.

The prefix of length d of x can be obtained by shifting x right $k(L-d)$ bit positions.

The data structure uses $O(kLn)$ bits and can be constructed in $O(kLn)$ time. This follows from Lemma 4.1 using the fact that $(r+1)q < ((\log n)/L + 2)(L-1) = [(L-1)(\log n) + 2L(L-1)]/L \leq [(L-1)(\log n) + \log n]/L = \log n$. □

The Static Predecessor Data Structure

Now, we present our new data structure for the static predecessor problem. It is constructed recursively, using multi-way search in a tree, described above.

Lemma 4.3: If n, u, and c are positive integers such that $n \geq u^u$, $c \leq u$, and memory words contain $b \geq 2u^{c+2}$ bits, then there is a static data structure for representing a set of at most n integers from the universe $[0, 2^{u^c} - 1]$ that supports predecessor queries in $O(u)$ time, uses $O(n^2/u^2)$ words, and can be constructed in $O(u^c n^2)$ time.

Lemma 4.3 follows immediately from the following technical lemma, by setting $L = 1$, $a = u$, and $s = n$. This technical lemma will be in a convenient form for the recursive construction. At each stage in the recursion, either the size s of the set being represented will decrease by at least a factor of $n^{1/u}$ or the number of bits required to represent each element will decrease by a substantial amount. The parameters a and c are upper bounds on the number of additional rounds of set-size reduction and range-size reduction, respectively, that can be performed.

As a result of a round of range-size reduction, the number of bits required to represent an element will decrease to the largest power of u less than the current value. If another range reduction is performed next, then the power of u will decrease by one. When set-size reduction is performed, the range also gets smaller, but the size of the resulting range may vary. Although the number of bits will not necessarily be a power of u, it will be a small integer multiple L of a power of u, where $L \leq u$.

Lemma 4.4: If a, c, L, n, s, and u are integers such that $a, c \in [0,u]$, $n \geq u^u$, $L \in [1,u]$, $s \leq n^{a/u}$, and memory words contain $b \geq [2(u-1)^2 - 1]Lu^c$ bits, then there is a static data structure for representing a set of s integers from the universe $[0, 2^{Lu^c} - 1]$ that supports predecessor queries in $O(a+c)$ time, uses $O(sn/u^2)$ words, and can be constructed in $O(sLu^c n)$ time.

Proof If $a = 0$ then $s = 1$ and it suffices to store the element. Therefore, we assume that $a > 0$.

If $c = 0$ and $L = 1$, then the universe is $[0,1]$ and a 2-bit characteristic vector suffices. If $c \geq 1$ and $L = 1$, then $Lu^c = L'u^{c'}$, where $L' = u$ and $c' = c - 1$. Therefore, we may assume that $L > 1$.

Let S be any set of $s \leq n^{a/u}$ integers from the universe $[0, 2^{Lu^c} - 1]$ and let T_0 denote the binary trie of depth Lu^c representing S.

For $j \in [1,c]$, let T_j be the trie of depth Lu^{c-j} and branching factor 2^{u^j} that consists of all nodes in T_0 at levels divisible by u^j, where node v is the parent of node w in T_j if and only if v is the ancestor at distance u^j from w in T_0. Then T_j represents S, when viewed as a set of strings of length Lu^{c-j} over the alphabet $[0, 2^{u^j} - 1]$.

Note that the set of nodes of T_j is exactly the set of nodes of T_{j-1} at depths divisible by u. Furthermore, the children of each non-leaf node v in T_j are the leaves of the subtrie of depth u rooted at v in T_{j-1}.

For every node v in T_c, let $\min_S(v)$ denote the smallest element of S with prefix v and $\max_S(v)$ denote the largest element of S with prefix v.

The data structure is built recursively. For the purposes of analysis, we will associate each recursive instance of the data structure with a node in one of the trees $T_c, \ldots, T_0$. The entire data structure will be associated with the root of T_c. The subproblems for sets of strings of length $1 < L' \leq u$ over the alphabet $[0, 2^{u^j} - 1]$ will be associated with distinct non-leaf nodes of T_j. The data structure consists of the following parts:

1. the data structure described in Lemma 4.2, with $T = T_c$ and $k = u^c$,
2. for each non-heavy node v in T_c that has a heavy parent and for each heavy node v in T_c:
   (a) $\max_S(v)$, $\min_S(v)$, and $pred(\min_S(v), S)$,
3. for each heavy node v in T_c with at least two children:
   (a) a linear size perfect hash table containing the letters (in $[0, 2^{u^c} - 1]$) labelling edges from v to non-heavy children of v,

(b) a recursive instance of the data structure for the set $S'_v = \{v' \in [0, 2^{u^c} - 1] \mid v \cdot v' \text{ is a child of } v\}$ of size at most s (which we associate with node v in T_{c-1}), and

4. for each non-heavy node w at depth $0 < d < L$ in T_c with a heavy parent and at least two leaves in its subtree:

   (a) a recursive instance of the data structure for the set $S'_w = \{v' \in [0, 2^{u^c} - 1]^{L-d} \mid w \cdot v' \in S\}$ of size at most $s/n^{1/L} \leq s/n^{1/u} \leq n^{(a-1)/u}$ (which we associate with node w in T_c, if $d < L-1$, or node w in T_{c-1}, if $d = L-1$).

To find the predecessor of $x \in [0, 2^k - 1]$ in the set S, first determine the longest prefix v of x that is a heavy node in T_c, as described in Lemma 4.2. Suppose that v is at depth d.

If v has at most one child, then

$$pred(x, S) = \begin{cases} pred(\min_S(v), S) & \text{if } x \leq \min_S(v) \\ \min_S(v) & \text{if } x > \min_S(v). \end{cases}$$

Now consider the case when v has at least two children. Determine whether some child of v is a prefix of x, using the hash table containing all of v's non-heavy children. By definition of v, if there is such a child, then it is not heavy.

If no child of v is a prefix of x, then

$$pred(x, S) = \begin{cases} pred(\min_S(v), S) & \text{if } x \leq \min_S(v) \\ \max_S(v \cdot pred(x_{d+1}, S'_v)) & \text{if } x > \min_S(v). \end{cases}$$

Find $pred(x_{d+1}, S'_v)$ using the recursive instance of the data structure for the set S'_v. Note that $v \cdot pred(x_{d+1}, S'_v)$ is either a non-heavy child of the heavy node v or is itself heavy, so the largest element of S in the subtree rooted at this node is stored explicitly.

Finally, consider the case when some child w of v is a prefix of x. If w has exactly one leaf in its subtree, then

$$pred(x, S) = \begin{cases} pred(\min_S(w), S) & \text{if } x \leq \min_S(w) \\ \min_S(w) & \text{if } x > \min_S(w). \end{cases}$$

Otherwise,

$$pred(x, S) = \begin{cases} pred(\min_S(w), S) & \text{if } x \leq \min_S(w) \\ w \cdot pred(x_{d+2} \cdots x_L, S'_w) & \text{if } x > \min_S(w). \end{cases}$$

Find $pred(x_{d+2} \cdots x_L, S'_w)$ using the data structure for the set S'_w.

It follows by induction that predecessor queries are supported in $O(a + c)$ time.

Next, we analyze the storage requirements of the entire data structure (including all recursive instances of the data structure for the subproblems).

First, we count the total space used by part 2(a) of the construction. There are at most $sLu^{c-j} + 1$ nodes in T_j some of which contribute a constant number of words. If $j = c$, these words are at most $Lu^c \leq u^{c+1}$ bits long. If $j < c$, these words are at most u^{j+1} bits long. This gives a total of $O(csLu^{c+1})$ bits.

Now, we count the total space used by part 3(a) of the construction. Since each tree T_j has s leaves, there are at most $2(s-1)$ nodes in T_j with siblings (i.e. that are children of nodes with at least two children). Each of these nodes can be viewed as contributing at most a constant number of u^j-bit entries to its parent's perfect hash table. Thus, over the entire recursive data structure, the hash tables use $O\left(\sum_{j=0}^{c} su^j\right) = O(su^c)$ bits.

Finally, we count the total space used by part 1 of the construction. The only association of nodes of T_c with recursive instances of the data structure occur as a result of part 4(a) of the construction. Since these nodes are non-heavy children of heavy nodes, they have siblings in T_c. Therefore, at most $2(s-1)$ nodes in addition to the root are associated with an instance of the data structure. For each such node, part 1 of the construction is an instance of the data structure of Lemma 4.2 with T the subtree (of depth at most L) rooted at this node and $k = u^c$, which uses $O(Lu^c n)$ bits. In total, $O(sLu^c n)$ bits are used.

Similarly, for $j < c$, each node in T_j that is associated with a recursive instance of the data structure either has a sibling (in case the instance arose from part 4(a) with $d < L-1$), or is also a node in T_{j+1} with at least two children (in case the instance arose from part 4(a) with $d = L-1$ or from part 3(b)). Since each tree has s leaves, it contains at most $s-1$ nodes with two or more children and at most $2(s-1)$ nodes with siblings. For each such node, part 1 of the construction is an instance of the data structure of Lemma 4.2 with T the subtree (of depth at most u) rooted at this node and $k = u^j$, which uses $O(u^{j+1}n)$ bits. In total, over all trees T_j, with $j < c$, the number of bits used is $O(\sum_{j=0}^{c-1} su^{j+1}n) = O(su^c n)$.

Altogether, $O(csLu^{c+1}) + O(su^c) + O(su^c n) + O(sLu^c n) = O(sLu^c n)$ bits are used. Since $b \in \Omega(Lu^{c+2})$, only $O(sn/u^2)$ words are needed.

It remains to determine the construction time of the data structure. The only two time-consuming parts of the construction are 1 and 3(a). Since the time needed to construct an instance of the data structure from Lemma 4.2 is bounded above by the same quantity as the upper bound on the number of bits used, it follows that $O(sLu^c n)$ time suffices to construct all the instances that occur in some part 1 during the recursive construction. A perfect hash table with $2(s-1)$ or fewer u^j-bit elements (or a collection of perfect hash tables with a combined total of $2(s-1)$ elements) can be constructed in $O(s^2u^j)$ time [40]. Thus $O(s^2u^c) \subseteq O(sLu^c n)$ time suffices to construct all the instances that occur in some part 3(a) during the recursive construction. □

The data structure of Lemma 4.3 is not quite enough on its own to achieve our desired result, since it requires a particular relationship between the universe size and the set size. Also, Lemma 4.3 requires word size greater than the logarithm of the universe size. However, combining this with packed B-trees and a small number of rounds of range reduction using Willard's x-fast tries, we obtain an upper bound that matches our lower bound for the static predecessor problem in Section 3.4.

Theorem 4.5: There is a static data structure for representing n integers from the universe $[1,N]$ that uses $O(n^2 \log n/\log\log n)$ words of $\log N$ bits and answers predecessor queries in time $O\left(\min\left\{\frac{\log\log N}{\log\log\log N}, \sqrt{\frac{\log n}{\log\log n}}\right\}\right)$. Moreover, this data structure can be constructed in $O(n^{2+\varepsilon})$ time for any $\varepsilon > 0$.

Proof Let $b = \log N$. If $n < 2^{4(\log\log N)^2/(\log\log\log N)}$, then

$$(\log n)/\log b \le (\log n)/\log\log N < \sqrt{8\log n/\log\log n} \le 4\sqrt{2}\log\log N/\log\log\log N,$$

for N sufficiently large. In this case, we use Fredman and Willard's fusion trees [27] which, as shown by Hagerup [28], can be parameterized to use $O(n)$ words, support predecessor queries in $O(1+(\log n)/\log b)$ time and can be constructed in $O(n)$ time.

Now assume that $n \ge 2^{4(\log\log N)^2/(\log\log\log N)}$. Then $\sqrt{\log n/\log\log n} \ge \sqrt{2}\log\log N/\log\log\log N$ for N sufficiently large. Therefore, it suffices to give an $O(\log\log N/\log\log\log N)$ query time in this case. Let u be the smallest integer such that $u^u \ge \log N$. Then, for N sufficiently large, $(\log\log N)/\log\log\log N \le u \le 2(\log\log N)/\log\log\log N \le \sqrt{(2\log n)/\log\log n}$ and $u^u \le (\log N)^2 \le n^{1/u}$. Observe that since $(u-1)^{u-1} <$

$\log N$, $u^u/\log N \leq u^u/(u-1)^{u-1} \leq u(1+1/(u-1))^{u-1} \leq eu$. Therefore $2u^{u-2} \leq (2e\log N)/u \leq \log N$, for $u \geq 2e$, which holds for N sufficiently large.

In this case, the data structure has two parts. The first part consists of the top $\lceil 4\log u \rceil$ levels of Willard's x-fast trie [44]. This reduces the problem of finding the predecessor in a set of size n from a universe of size N to finding the predecessor in a set of size at most n from a universe of size 2^k, where $k = (\log N)/2^{\lceil 4\log u \rceil} \leq (\log N)/u^4 \leq u^{u-4}$.

For each resulting subproblem, the set size is at most n, $b = \log N \geq 2u^{u-2}$, and the universe size is at most $2^k \leq 2^{u^{u-4}}$. Therefore, the data structure of Lemma 4.3, with $c = u-4$, can be used. The time spent by the query in this portion of the data structure is $O(u)$ and the query takes only $O(\log u)$ steps in the x-fast trie portion of the data structure. Thus, the total query time is $O(\log\log N/\log\log\log N)$.

There are $O(u^4)$ subproblems. By Lemma 4.3, the data structures for each subproblem uses $O(n^2/u^2)$ space and the truncated x-fast trie uses $O(n\log N) \subseteq O(n^2)$ space. Thus the total space used is $O(n^2u^2) \subseteq O(n^2\log n/\log\log n)$.

For each subproblem, the data structure can be constructed in $O(n^2u^{u-4})$ time, for a total of $O(n^2u^u)$ time over all subproblems. Each level of the x-fast trie can be constructed in time $O(n^2\log N) = O(n^2u^u)$, using the time bounds for hash table construction [40], and there are $O(u^4)$ levels. Thus, the total construction time is $O(n^2u^{u+4})$, which is $O(n^{2+\varepsilon})$ for any constant $\varepsilon > 0$. □

Our algorithm also works for sets of positive floating point numbers that use a fixed number of bits for the exponent followed by a fixed number of bits for the mantissa, where the exponent is expressed using biased notation. This is because lexicographic ordering of the binary strings representing numbers in this way is consistent with the numerical order of these numbers [29].

Similarly, our algorithm can be used for sets of rational numbers with numerators and denominators at most $k \in O(b)$ bits long, if each rational number x/y, where $x \in [0, 2^k-1]$ and $y \in [1, 2^k-1]$ is represented by the $3k$-bit binary representation of the integer $\lfloor x4^k/y \rfloor$. To see why, note that if $\lfloor x4^k/y \rfloor < \lfloor x'4^k/y' \rfloor$, then $x4^k/y < x'4^k/y'$ and if $\lfloor x4^k/y \rfloor = \lfloor x'4^k/y' \rfloor$, then $(x4^k/y) - 1 < x'4^k/y' < (x4^k/y) + 1$, so $-1 < -yy'/4^k < x'y - y'x < yy'/4^k < 1$ and, hence, $x/y = x'/y'$, since $x'y - y'x$ is an integer.

Algorithms for Related Problems

Observe that any data structure for the static predecessor problem for sets of size n from the universe $[1,N]$ can be augmented (using $O(n)$ additional space) to solve the static (L,N,n,x)-prefix problem for any language $L \subseteq \Sigma^*$ and any string $x \in \Sigma^*$ using only a constant amount of additional time. Specifically, given a string $y \in (\Sigma \cup \{\perp\})^N$, consider the set $S = \{i \mid y_i \in \Sigma\}$. Add a hash table of size $O(n)$ storing each element of $S \cup \{0\}$ in a different location. Together with each element $j \in S \cup \{0\}$, also store the answer to the query "Is $x \cdot \mathrm{PRE}_j(y) \in L$?". To determine whether $x \cdot PRE_j(y) \in L$ for an arbitrary element $j \in [0,N]$, first use the hash table to determine whether $j \in S \cup \{0\}$ and, if so, return the precomputed answer. Otherwise, find the predecessor j' of j in S. Then use the answer to the question "Is $x \cdot PRE_{j'} \in L$?" stored in the hash table.

Similarly, the static-predecessor-parity problem, the static rank-parity problem, and the static rank problem can be solved within the same time bounds.

The point separation problem for a set of points in the plane, with coordinates that are rational numbers expressed as the quotient of two b-bit words is considered by Chazelle [16]. He shows that asking whether a query (do all points in the set lie on the same side of a given line?) reduces to finding the predecessor among the slopes of the lines comprising the convex hull of the points in the set. Thus, this problem can be solved

in time $O\left(\min\left\{(\log b)/\log\log b, \sqrt{(\log n)/\log\log n}\right\}\right)$.

Exponential search trees [7, 10] can be combined with our static data structure to give a linear size dynamic data structure.

Corollary 4.6: The dynamic predecessor problem for a set of up to n integers from the universe $[1,N]$ can be solved on a RAM with $O(n)$ words of $\log N$ bits, in time $O\left(\min\left\{\frac{(\log\log n)\log\log N}{\log\log\log N}, \sqrt{\frac{\log n}{\log\log n}}\right\}\right)$.

5 Conclusions

In this paper, we achieve asymptotically matching upper and lower bounds for the static predecessor problem and the static prefix problem for indecisive languages. We obtained the lower bound first and then worked for a long time trying to improve it. Eventually, in an attempt to understand the reasons for our difficulties, we started looking for a communication game algorithm that would be efficient for inputs drawn from the hard distribution used in our lower bound. The algorithm we obtained for this problem directly led to the development of our algorithm in Section 4. This is an example where the insight obtained from a lower bound can lead to a substantially improved algorithm.

Further connections have been made between the approximate nearest neighbour problem for inputs in $[1,N]^d$, for constant d, and our bounds for the predecessor problem. Amir, Efrat, Indyk, and Samet [5] give a reduction from the predecessor problem to the approximate nearest neighbour problem in one dimension to derive an $\Omega(\log\log N/\log\log\log N)$ lower bound for the approximate nearest neighbour problem. Using an extension of our data structure, Cary [14] gives a data structure that matches this lower bound for any constant number of dimensions.

It would be nice to remove the $\log\log n$ factor in the numerator in the first term of the minimum in Corollary 4.6 so that our upper and lower bounds would match for the dynamic versions of these problems.

For the static version of the predecessor data structure, could an algorithm be obtained that uses linear or nearly-linear space, rather than quadratic space without using the full power of exponential search trees? A key place where there is room for improvement in our arguments is in Lemma 4.3 when the set size is reduced. Our construction and analyses do not take advantage of the fact that the set may be drastically reduced in size, rather than simply reduced periodically by $n^{1/u}$ factors. The ability to adjust to different set sizes is one of the main advantages of exponential search trees and, in order to do this, one may need a similar structure.

Acknowledgements

We are grateful to Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup for helpful discussions.

References

[1] M. Ajtai, M. Fredman, and J. Komlós. Hash functions for priority queues. *Information and Control*, 63:217–225, 1984.

[2] Miklós Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8:235–247, 1988.

[3] S. Alstrup, G. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, pages 476–482, Hersonissos, Crete, Greece, July 2001.

[4] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proceedings 39th Annual Symposium on Foundations of Computer Science*, pages 534–543, Palo Alto, CA, November 1998. IEEE.

[5] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient regular data structures and algorithms for location and proximity problems. In *Proceedings 40th Annual Symposium on Foundations of Computer Science*, pages 160–170, New York,NY, October 1999.

[6] A. Andersson. Sublogarithmic seaching without multiplications. In *Proceedings 36th Annual Symposium on Foundations of Computer Science*, pages 655–665, Milwaukee, WI, October 1995. IEEE.

[7] A. Andersson. Faster deterministic sorting and seaching in linear space. In *Proceedings 37th Annual Symposium on Foundations of Computer Science*, pages 135–141, Burlington, VT, October 1996. IEEE.

[8] A. Andersson, P. B. Miltersen, S. Riis, and M. Thorup. Static dictionaries on AC^0 RAMs: query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient. In *Proceedings 37th Annual Symposium on Foundations of Computer Science*, pages 441–450, Burlington, VT, October 1996. IEEE.

[9] A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215:337–344, 1999.

[10] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, pages 335–342, Portland,OR, May 2000.

[11] G. Brodal. Predecessor queries in dynamic integer sets. In *(STACS) 97: 14th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1200 of *Lecture Notes in Computer Science*, pages 21–32. Springer-Verlag, February 1997.

[12] A. Brodnik, P.B. Miltersen, and I. Munro. Trans-dichotomous algorithms without multiplications–some upper and lower bounds. In *Proceedings of the 5th Workshop on Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 426–439, Halifax, NS, Canada, 1997. Springer-Verlag.

[13] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[14] Matthew Cary. Towards optimal ε-approximate nearest neighbor algorithms in constant dimensions. *Journal of Algorithms*. To appear.

[15] A. Chakrabarti, B. Chazelle, B. Gum, and A. Lvov. A good neighbor is hard to find. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 305–311, Atlanta, GA, May 1999.

[16] B. Chazelle. Geometric searching over the rationals. In *Proceedings of the 7th European Symposium on Algorithms*, Lecture Notes in Computer Science, pages 354–365. Springer-Verlag, 1999.

[17] B. Chazelle. *The Discrepancy Method: Randomness and Complexity*. Cambridge University Press, 2000.

[18] V. Chvátal. Probabilistic methods in graph theory. *Annals of Operations Research*, 1:171–182, 1984.

[19] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Automata, Languages, and Programming: 19th International Colloquium*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, July 1992.

[20] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[21] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R Tarjan. Dynamic pertect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[22] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In Michael S. Paterson, editor, *Automata, Languages, and Programming: 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 6–17, Warwick University, England, July 1990. Springer-Verlag. Informatik-Festschrift, zum 60, Geburtstag von Gunter Hotz, J. Buchmann, H. Ganzinger, and W.J. Paul, eds., B.G. Teubner, 1992, pages 95–119.

[23] F. Fich and P. B. Miltersen. Tables should be sorted (on random access machines). In *Proceedings of the 4th Workshop on Algorithms and Data Structures*, volume 995 of *Lecture Notes in Computer Science*, pages 163–174. Springer-Verlag, 1995.

[24] Frandsen, P. B. Miltersen, and Skyum. Dynamic word problems. In *Proceedings 34th Annual Symposium on Foundations of Computer Science*, pages 470–479, Palo Alto, CA, November 1993. IEEE. Also, J. ACM. 44 (2), 1997, pages 257–71.

[25] M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[26] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 345–354, Seattle, WA, May 1989.

[27] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[28] T. Hagerup. Sorting and searching on a word RAM. In *(STACS) 98: 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398, Paris, France, February 1998. Springer-Verlag.

[29] J. L. Hennessey and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufman, San Mateo, CA, 1994.

[30] Mauricio Karchmer and Avi Wigderson. Monotone circuits for connectivity require super-logarithmic depth. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 539–550, Chicago, IL, May 1988.

[31] J. A. La Poutré. Lower bounds for the Union-Find and the Split-Find problem on pointer machines. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, pages 34–44, Baltimore, MD, May 1990.

[32] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5, 1990.

[33] K. Mehlhorn, S. Näher, and H. Alt. A lower bound on the complexity of the Union-Split-Find problem. *SIAM Journal on Computing*, 17:1093–1102, 1988.

[34] P. B. Miltersen. The bit probe complexity measure revisited. In Alain Finkel, Patrice Enjalbert, and Klaus W. Wagner, editors, *STACS 93: 10th Annual Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 662–671, Wurzburg, Germany, February 1993. Springer-Verlag.

[35] P. B. Miltersen. Lower bounds for Union-Split-Find related problems on random access machines. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, pages 625–634, Montréal, Québec, Canada, May 1994.

[36] P. B. Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 556-563. 1998.

[37] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. *Journal of Computer and System Sciences*, 57(1):37–49, 1998.

[38] Rasmus Pagh. Faster deterministic dictionaries. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 487–493, 2000.

[39] Y. Perl and E. M. Reingold. Understanding the complexity of interpolation search. *Information Processing Letters*, 6(6):219–222, 1977.

[40] R. Raman. Priority queues: Small, monotone, and trans-dichotomous. In *Proceedings of the 4th European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer-Verlag, 1996.

[41] M. Thorup. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555. 1998.

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

[43] P. Van Emde Boas, R Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

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

[45] D. E. Willard. Searching unindexed and nonuniformly generated files in $\log\log N$ time. *SIAM Journal on Computing*, 14(4):1013–1029, 1985.

[46] Bing Xiao. *New bounds in cell probe model*. PhD thesis, University of California, San Diego, 1992.

[47] A. C. Yao. Some complexity questions related to distributive computing. In *Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing*, pages 209–213, Atlanta, GA, April-May 1979.

[48] A. C. Yao. Should tables be sorted? *Journal of the ACM*, 28:615–628, July 1981.