Marked ancestor problems *
(Extended abstract)

Stephen Alstrup
DIKU, Comp. Sc. Dept.,
University of Copenhagen, Denmark

Thore Husfeldt
DNA, Comp. Sc. Dept.,
Lund University, Sweden,

Theis Rauhe
BRICS, Comp. Sc. Dept.,
University of Aarhus, Denmark

Abstract

Consider a rooted tree whose nodes can be in two states: marked or unmarked. The marked ancestor problem is to maintain a data structure with the following operations: mark(v) *marks node v;* unmark(v) *removes any marks from node v;* firstmarked(v) *returns the first marked node on the path from v to the root.*

We show tight upper and lower bounds for the marked ancestor problem. The lower bounds are proved in the cell probe model, the algorithms run on a unit-cost RAM.

As easy corollaries we prove (often optimal) lower bounds on a number of problems. These include planar range searching, including the existential or emptiness problem, priority search trees, static tree union–find, and several problems from dynamic computational geometry, including segment intersection, interval maintenance, and ray shooting in the plane. Our upper bounds improve algorithms from various fields, including coloured ancestor problems and maintenance of balanced parentheses.

1. Introduction

Let T be a rooted tree with n nodes V, each of which can be in two states: *marked* or *unmarked*. The nodes on the unique path from v to the root are denoted $\pi(v)$, which includes v and the root. The *marked ancestor problem* is to maintain a data structure with the following operations:

mark(v): mark node v,

unmark(v): unmark node v,

firstmarked(v): return v's first marked ancestor, i.e., the marked node on $\pi(v)$ closest to v (this may be v itself).

This generalises the well-known predecessor problem where T is a path.

We present a new lower bound for the marked ancestor problem in the cell probe model with word size b between the update time t_u and the query time t_q,

$$t_q = \Omega\Big(\frac{\log n}{\log(t_u b \log n)}\Big). \qquad (1)$$

For logarithmic word size this means a lower bound of $\Omega(\log n / \log\log n)$ per operation.

The *incremental* version of this problem does not support *unmark*, the *decremental* problem does not support *mark*, and the *fully dynamic* problem supports both update operations. The bound (1) holds for the worst case complexity of both the incremental and the decremental problem, as well as for the amortised complexity of the fully dynamic problem. In all cases this matches the upper bounds to within a constant factor, so the complexity of the marked ancestor problem is now fully understood; Table 1 provides an overview.

We complement the tradeoff (1) with an algorithm for the RAM with logarithmic word size achieving the following bounds:

Table 1: Marked ancestor problems

updates		complexity pr. operation	
mark	*unmark*	amortised	worst case
•		} $O(1)$	} $\Theta(\log n / \log\log n)$
	•		
•	•	$\Theta(\log n / \log\log n)$	

Bounds given for logarithmic word size

*Part of this work was done while the first author visited BRICS and Lund University, and while the last author visited the Fields Institute of Toronto. This work was partially supported by the ESPRIT Long Term Research Programme of the EU, project number 20244 (ALCOM-IT). The second author was partially supported by a grant from the Swedish TFR.

1. worst case update time $O(\log\log n)$ for both *mark* and *unmark*,
2. worst case query time $O(\log n/\log\log n)$,
3. linear space and preprocessing time.

To achieve these results we present a new micro–macro division of trees. In contrast to standard tree divisions [15, 20], our approach does not limit the number of nodes or the number of boundary nodes in a micro tree. This leads to exponentially better update times than Amir *et al.* [5], who achieved $O(\log n/\log\log n)$ per operation.

1.1. Variants and extensions

Existential queries. In the *existential* marked ancestor problem, the query is modified to

exists(v): return 'yes' if and only if $\pi(v)$ contains a marked node.

The lower bound (1) holds even for this, potentially easier problem, which is our main tool for proving lower bounds in Section 2. For this query we can improve our fully dynamic algorithms to mark nodes in worst case constant time.

Aggregate queries. We slightly modify the problem as follows. Each node v of T now contains a value $\mathrm{val}(v) \in \{0,\ldots,m\}$, and we change the operations accordingly,

update(v,k): change $\mathrm{val}(v)$ to k,

sum(v): return $\sum_{u\in\pi(v)} \mathrm{val}(u) \mod m$

For m sufficiently large ($m = 2^b$, the largest value that can be stored in a register) we can show the stronger trade-off

$$t_q \log t_u = \Omega(\log n). \tag{2}$$

Note that this bound does not depend on the word size.

If the update operation is somewhat restricted (the change to $\mathrm{val}(v)$ can be no larger than polylogarithmic in n), we can give an algorithm with $O(\log n/\log\log n)$ time per operation, which was known previously only for paths [11]. See [3].

Reporting queries. Our algorithms can be extended to support *reporting* queries of the form

find(v,k): return the first k marked nodes on $\pi(v)$.

The worst case query time becomes $O(s+\log n/\log\log n)$, where $s \le k$ is the size of the output.

Adding leaves. The variants of the marked ancestor problem studied in [5, 20] provide additional operations that modify the topology of the tree:

add leaf(v,w): insert a new leaf v with parent $w \in V$.

delete leaf(v): remove the leaf v.

Our data structure can support *add leaf* in amortised constant time and *delete leaf* in worst case constant time while maintaining the worst case bounds on the other operations.

Summary. Table 2 shows time bounds per operation for a number of variants of the marked ancestor problem supporting various sets of update and query operations.

In summary, we can provide optimal worst case algorithms for *mark* and *exists*. For other combinations of operations, we are left with a small gap of size $O(\log\log n)$ between upper and lower bounds for the updates, while our query algorithm is optimal even for much slower updates and even if we change to amortised analysis. These bounds survive the addition of harder updates like *add leaf*.

1.2. Models of Computation

Our algorithms run on a random access machine with logarithmic word size and standard unit-cost operations. The lower bounds are proved in the cell probe model, which allows arbitrary operations on registers and can be regarded as a strong nonuniform version of the RAM, the cell size is denoted $b = b(n)$. The model makes no assumptions on the preprocessing time or the size of the data structure.

2. Applications

We state the results in this section in terms of logarithmic word size for concreteness.

2.1. The Emptiness Problem

The *emptiness* (or, existential) problem lies at the heart of all range searching problems: maintain a set $S \subseteq [n]^2$ of points in the plane under insertions and deletions, and given rectangle R determine whether $S \cap R$ is empty. Finding a lower bound for this problem is Open Problem 1 in a recent handbook chapter on range queries by Agarwal [1].

Proposition 1 *The planar emptiness problem requires time* $\Omega(\log n/\log\log n)$ *per operation. This is true even for* dominance queries, *where all query rectangles have their lower left corner in the origin. The bound holds for the incremental and decremental variants, and also for the amortised complexity of the fully dynamic problem.*

We sketch the proof: Inspecting the proof of Theorem 1 we find that the lower bound on the marked ancestor problem applies even if the tree T is regular. Embed such a tree

Table 2: Algorithms for variants of the marked ancestor problem

updates			query	remarks
mark	*unmark*	*add leaf*	*firstmarked*	
$O(1)^*$	–	$O(1)^*$	$O(1)$	Gabow and Tarjan [20] and [4]
–	$O(1)^*$	$O(1)^*$	$O(1)$	Gabow and Tarjan [20] and [4]
$O(\log\log n)$	$O(\log\log n)$	$O(1)^*$	$\Theta(\log n/\log\log n)^\dagger$	finds k first marked ancestors in time $O(k+\log n/\log n\log\log n)$
			exists	
$O(1)$	$O(\log\log n)$	$O(1)^*$	$\Theta(\log n/\log\log n)^\dagger$	

Dashes indicate unsupported operations
* Amortised time bound
† Optimal even for polylogarithmic update time

in the first quadrant of the plane, with the root in the origin and the nodes at depth i spread out evenly on the diagonal $y = -x + d^h - d^{h-i}$, where $d = \log^{O(1)} n$ is Ts outdegree and $h = O(\log n/\log\log n)$ is its height. The query rectangles have the upper right corner in the queried node and the lower left corner in the origin.

The upper bound for this problem is slightly larger, amortised $O(\log n \log\log n)$ using fractional cascading [24], yet for dominance queries it is $O(\log n/\log\log n)$ using [23, 31], matching our lower bound.

The algebraic models that have been used for proving lower bounds for range queries do not provide bounds for the emptiness problem, since free answers to the emptiness query are built into the model [8, p. 44]. Recently, a handful of papers have established the bound $\Omega(\log\log n/\log\log\log n)$ in the cell probe model on the query time for the *static* version of the emptiness problem, which also holds for the dynamic problem. That bound uses a completely different technique based on a result of Ajtai [2], it can be proved by combining the reduction of Miltersen, Nisan, Wigderson, and Safra [25, Thm. 18] with a bound by Beame and Fich [6].

2.2. Priority Search Trees

Priority searching, a mixture of a search structure and a priority queue, supports the following operations on a set of points $X \subseteq [n]$ with priorities $p(n) \in [n]$,

insert$(x, p(x))$: insert x into X with priority $p(x)$,

delete(x): remove x from X

max(x): return $\max\{\, p(y) \mid y \le x \,\}$.

McCreight's classic data structure [23] implements these operations (and a few more) in logarithmic time, Willard [31] shows how to improve this to $O(\log n/\log\log n)$ on a RAM. Since search trees and priority queues can be implemented in time $O(\log^{1/2} n)$ [19] and $O(\log\log n)$ [29], respectively, one might hope for even stronger improvements. However, the lower bound (1) holds also for this problem, showing that the bound from Willard's construction is optimal.

Proposition 2 *Every data structure for priority searching requires time* $\Omega(\log n/\log\log n)$ *per operation.*

Priority searching solves the emptiness problem with dominance queries, so the bound follows from Prop. 1. As before, the bound holds even for the incremental or decremental variants, and also for the amortised query time of the fully dynamic problem.

2.3. Two-dimensional Range Searching

A general setting that abstracts both the emptiness problems and priority search trees is planar range searching with queries of the form $\sum_{x\in R} \mathrm{val}(x)$, where val maps points to values from a semigroup, for instance $(\mathbf{N}, +)$ for counting and $(U^d, \cup)$ for reporting. These problems are well understood in structured models [8, 16, 30, 32]. In the cell probe model, the paper of Fredman and Saks [18] provides good lower bounds for many choices of algebraic structure, like $(\{0,1\}, \oplus)$ or $(\mathbf{N}, +)$, and Frandsen, Miltersen, and Skyum [14] analyse this problem for finite monoids of the one-dimensional case. Our lower bound shows that in two dimensions, the lower bound holds for *any* structure, if only the query can distinguish 'no points' from 'some points,' including for example $(\mathbf{N}, \max)$ or $(\{0,1\}, \vee)$.

2.4. Union–find problems

Gabow and Tarjan [20] introduce a weaker version of disjoint set union that allows linear time algorithms, called *static tree set union*. Initially every node v of a (static) tree is in a singleton set $\{v\}$, the algorithm handles the following operations:

unite with parent(v): perform the union of the set containing v with the set containing its parent, the two original sets are destroyed,

find(v): return the name of the set containing v.

This problem is easier than the usual disjoint set union problem in that the 'union-tree,' describing the structure of disjoint set unions to be performed, is known in advance. Indeed, [20] show that on a RAM, the problem's *amortised complexity* is lower, since it allows update and query operations in constant time per operation.

This problem is equivalent to the decremental marked ancestor problem: start with the entire tree marked, and unmark a node when it is united with its parent. The *find* query is just *firstmarked*. Hence, our trade-off (1) holds for the *worst-case* complexity of this problem, so in that sense knowing the tree of unions does not help:

Proposition 3 *Static tree union–find requires worst case time* $\Omega(\log n/\log\log n)$ *per operation.*

This bound is optimal by Blum's algorithm [7]. For the general union–find problem, cell probe lower bounds of the same size were provided in [18].

2.5. Computational Geometry

Range searching problems in computational geometry often consider other objects than just points, and our techniques can be applied to many of these, giving $\Omega(\log n/\log\log n)$ lower bounds. We give some examples.

Consider the following collection of segments in the plane: $\{[jd^i,(j+1)d^i]\times i \mid 0\le i\le h, 0\le j<d^{h-i}\}$, corresponding to a regular d-ary tree T of height h. When a node of T gets marked or unmarked, the corresponding segment is inserted or removed from the collection. This shows how our hardness results for points can be translated to segments. For example, the the *firstmarked* query in T corresponds to a *ray shooting* query among the segments (return the first object encountered along a line), and to answer an emptiness query in T we can use an *orthogonal segment intersection* query (report the segments crossing a given vertical line). By collapsing the family of segments onto a single line, we receive lower bounds for *interval maintenance* problems with reporting queries—for counting queries (return the *number* of intervals containing a query point) the lower bound was already given in [18].

Cell probe bounds for other geometric problems (point location and some range counting problems) are given in [22, 21].

2.6. Dynamic complexity

The lower bound for the path sum problem (2) is the largest lower bound known for any concrete dynamic problem in the cell probe model, previous bounds did not even achieve $t_u\cdot t_q=\omega(\log n/\log\log n)$. Our lower bound can be seen as a response to Fredman's challenge [17] to find stronger lower bounds than $\Omega(\log n/\log\log n)$ for dynamic problems in the cell probe model, even though we still cannot improve $t_u+t_q=\Omega(\log n/\log\log n)$, which is arguably more interesting.

We can cast the ancestor problem in a Boolean function setting; maintain $2n$ bits $x_1,\ldots,x_{2n}$ under the following operations:

update(i): change x_i to $\neg x_i$,

query: return $\bigvee_{1=i}^{n}\big(x_{2i}\wedge\bigvee\{x_j \mid v_j\in\pi(v_i)\}\big)$.

This precisely models the existential marked ancestor problem, with $x_j=1$ iff v_j is marked, and where $x_{n+1},\ldots,x_{2n}$ is used to pick out the queried path. This function is clearly in AC^0, yet by (1) has large 'dynamic hardness' in the sense of Miltersen et al. [26]. This contrasts the work in [21] were it is proved that for *symmetric* Boolean functions, there is a close correspondence between parallel and dynamic hardness.

A model with nondeterministic queries was introduced in [21] to study partial sum problems. The existential marked ancestor problem can be solved with constant time queries in this model (guess a node on $\pi(v)$ and verify that it is marked), so our results show a gap of $\Omega(\log n/\log\log n)$ between the strength of models with deterministic and nondeterministic queries. Furthermore, our results can be seen to show that nondeterminism does not help for the complement of the problem (answer 'yes' only if $\pi(v)$ is empty), so this 'nonemptiness' problem is hard even with nondeteminstic queries.

2.7. Improved upper bounds

Marked ancestor problems appear implicitly and explicitly in many algorithms in the literature. Consequently, our constructions improve a number of known upper bounds. This section provides some examples.

In the *tree colour problem* we associate a set of colours with each node of T. The following operations are considered in [12]:

colour(v,c): add c to v's set of colours,

uncolour(v,c): remove c from v's set of colours,

findfirstcolour(v,c): find the first ancestor of v with colour c.

This problem has been studied for two decades in connection with context trees [27], persistent data structures [9, 10], and recently in object oriented languages [12, 28].

Proposition 4 *Using linear time for preprocessing and linear space, we can maintain a tree with complexity* $O(\log\log n)$ *for* colour(v,c) *and* uncolour(v,c) *and complexity* $O(\log n/\log\log n)$ *for* findfirstcolour(v,c).

The construction can be found in [3] and is a simple extension of our marking algorithm using dictionaries for the colours. The best previous bound is given by a randomised algorithm of [12], which achieves $O\left((\log\log n)^2\right)$ per update, but at the cost of increasing the query time to $O(\log n/\log\log\log n)$.

The add leaf extension can be used in the tree color problem but arises also as a subproblem in other cases, see e.g. [5].

Our techniques to handle aggregate queries can be used to to dynamically maintain a string of parentheses and check if it is balanced in time $O(\log n/\log\log n)$, which is optimal and improves [13], see [3].

3. Lower bounds

3.1. Worst case bounds

Theorem 1 *The marked ancestor problem and the existential marked ancestor problem satisfy the trade-off* (1)*. Moreover, this is true even for incremental or decremental updates.*

The rest of this subsection is devoted to a proof of the above theorem.

Preliminaries. Write $[n]$ for the set $\{1,\ldots,n\}$. Identify V with $[n]$ for convenience, enumerating bottom-up from left to right, and let $M \subseteq V$ denote the set of marked nodes. The tree T is d-ary (the value of d is given in (3) below), so the number of nodes at layer j is $m_j = d^j$. The number of layers is $h = \log n/\log d$.

We model the data structure as a function D mapping the set of memory locations to b-ary contents. For convenience we assume that the data structures is no larger than 2^b (so that a pointer fits into a register), a restriction that can be removed (see [18]).

Sparse update scheme. We construct a family U of update sequences. With foresight, fix

$$\epsilon = \frac{1}{\log n}, \quad d = 4bt_{\mathrm{u}} \cdot \frac{1+\epsilon}{\epsilon^2}. \tag{3}$$

To alleviate notation we assume that d and ϵm_j for $j > 0$ are integers.

We let $\mathbf{B}(m,k)$ denote the set of bitstrings of length m with exactly k 1s.

A sequence of updates is defined by a bitstring $u = u_{(h)}\cdots u_{(0)}$ consisting of $h+1$ substrings satisfying $u_{(j)} \in \mathbf{B}(m_j, \epsilon m_j)$ $(h \geq j > 0)$ and $u_{(0)} = 0$. The set of all such strings is denoted U. Given u the nodes of T are updated in order, such that the ith update leaves node i marked iff $u_i = 1$. Thus the jth substring $u_{(j)}$ defines the updates to layer j. Since $u_{(0)} = 0$ the root of T is always left unmarked.

This update scheme marks T rather sparsely: for every $v \in V$,

$$\Pr_{u\in U}[v \in M] = \epsilon, \tag{4}$$

and the probability that none of its ancestors is marked, is

$$\Pr_{u\in U}\left[M \cap \pi(v) = \emptyset\right] \geq 1 - \epsilon h = 1 - \mathrm{o}(1). \tag{5}$$

Covers. The *weight* $W(X)$ of a set X of bitstrings is

$$W(X) = \#\bigcup_{x\in X}\{\, i \mid x_i = 1\,\}.$$

Let $0 < \delta \leq 1$. A set of strings $Y \subseteq \{0,1\}^n$ is a *δ-cover* if $W(Y) \geq \delta n$.

Clearly, $\mathbf{B}(m,k)$ contains a 1-cover of size only m/k, e.g., the set $\{\, 0^{rk}1^k0^{m-(r+1)k} \mid 0 \leq r < m/k\,\}$. The next lemma shows that every sufficiently large subset of $\mathbf{B}(m,k)$ must contain a $\frac{1}{300}$-cover of size m/k. First we recall a useful estimate of the binomial coefficient, which we will use freely in the proof.

Claim *Let $1 \leq s \leq n$. Then $\left(\frac{n}{s}\right)^s \leq \binom{n}{s} \leq \left(\frac{en}{s}\right)^s$ where e is the base of the natural logarithm.*

Lemma 5 *Let $k \leq m \leq k^2$. If $X \subseteq \mathbf{B}(m,k)$ has size*

$$\#X \geq 2^{-k}\binom{m}{k}$$

then there exists a $\frac{1}{300}$-cover $Y \subseteq X$ of size $\#Y \leq m/k$.

Proof. Let e denote the base of the natural logarithm. We construct round by round two sequences of sets $Y_r, X_r \subseteq X$ $(0 \leq r \leq m/20e^2k)$, such that

1. $W(Y_r \cup \{x\}) \geq W(Y_r) + \frac{1}{2}k$, for all $x \in X_r$
2. $\#Y_r = r$ and $\#X_r \geq 2^{-r}\#X$,
3. $rk \geq W(Y_r) \geq \frac{1}{2}rk$.

In round $m/20e^2k$ we let $Y = Y_r$, which proves the lemma. We start with $Y_0 = \emptyset$ and $X_0 = X$.

Assume X_{r-1} and Y_{r-1} have been constructed. Pick $x \in X_{r-1}$ and let $Y_r = Y_{r-1}\cup\{x\}$, satisfying the third condition.

It remains to construct X_r from X_{r-1}. Consider the set of 'bad' strings in X_{r-1}, namely those invalidating the first condition above,

$$B = \{\, z \in X_{r-1} \mid W(Y_r)\cup\{z\} < W(Y_r) + \tfrac{1}{2}k\,\}.$$

Every $z \in B$ has at least half its 1s in common with Y_r, so

$$\#B \leq \binom{W(Y_r)}{\frac{1}{2}k}\binom{m}{\frac{1}{2}k}.$$

The third condition gives $W(Y_r) \leq rk \leq m/20e^2$, since $r \leq m/20e^2k$. Using this and the estimate in the above claim, we get

$$\#B < \left(\frac{2em}{20e^2k}\right)^{k/2}\left(\frac{2em}{k}\right)^{k/2} = \left(\frac{1}{5}\right)^{k/2}\left(\frac{m}{k}\right)^{k}.$$

We assumed that $k^2 \geq m$, so $r \leq k/20e^2$. Hence we can bound the left term above by $\left(\frac{1}{5}\right)^{k/2} \leq 2^{-(\log 5)k/2} \leq 2^{-k+r}$. Thus $\#B \leq 2^{-(k+r)}\left(\frac{m}{k}\right)^k \leq 2^{-(k+r)}\binom{m}{k} \leq 2^{-r}\#X \leq \frac{1}{2}\#X_{r-1}$. This shows that B contains at most half the strings in X_{r-1}, so we can let $X_r = X_{r-1} - B$, satisfying the first and second condition. ∎

Register partition. For every update string u we define $h+1$ disjoint sets of registers $R_j(u)$ $(0 \leq j \leq h)$ in the data structure.

Let $r_j(u)$ denote the registers written during u's updates to layer j (the sets $r_j(u)$ are a good candidate for $R_j(u)$, but they are not disjoint). We construct the sets $R_j(u)$ inductively, starting with $R_0 = r_0(u)$.

Now assume that the sets $R_0(u), \ldots, R_{j-1}(u)$ have been defined for each u. Let $R_{<j}(u)$ denote the union of these registers and their contents:

$$R_{<j}(u) = \bigcup_{0 \leq k < j} \{\, (i, D(i)) \mid i \in R_k(u) \,\}.$$

For $w \in \mathbf{B}(m_j, \epsilon m_j)$ let u/w denote the bitstring u where $u_{|j}$ is replaced by w. Consider the set $[u]_j$ of update strings that are indistinguishable from u by looking only at registers in $R_k(u)$ for $k < j$,

$$[u]_j = \{\, u/w \mid R_{<j}(u) = R_{<j}(u/w) \,\}. \tag{6}$$

To finish the construction there are two cases, depending on the size of $[u]_j$:

1. If $[u]_j \leq 2^{-\epsilon m_j}\binom{m_j}{\epsilon m_j}$ then let $\langle u \rangle_j = \{u\}$.
2. Otherwise use Lemma 5 to pick a $\frac{1}{300}$-cover $Y \subseteq [u]_j$ of size $1/\epsilon$. For definiteness we choose the lexicographically first such cover Y and let $\langle u \rangle_j = Y \cup \{u\}$.

This ensures that if $[u]_j = [w]_j$ then $\langle u \rangle_j \cup \{w\} = \langle w \rangle_j \cup \{u\}$. Finally we define $R_j(u)$ by

$$R_j(u) = \bigcup_{w \in \langle u \rangle_j} r_j(w) \setminus \bigcup_{k<j} R_k(u), \tag{7}$$

which finishes the construction.

Properties of the register partition. First we note that if a register is in $R_j(u)$ then we are sure that it is not written during u's updates to layers $< j$.

Claim *If $i \in R_j(u)$ then $i \notin r_k(u)$ for any $k < j$.*

Proof. Assume $i \in r_k(u)$ for $k \neq j$. By disjointness of the partition we have $i \notin R_k(u)$. We can write the definition (7) of the latter set as $R_k(u) = A \setminus B$, where

$$A = \bigcup_{w \in \langle u \rangle_k} r_k(w), \quad B = \bigcup_{l<k} R_l(u).$$

Since $u \in \langle u \rangle_k$ by construction, we have $i \in A$ and hence $i \in B$ (because $i \notin A \setminus B$). Therefore $R_j(u)$ is one of the sets forming B, so $j = l < k$. ∎

Let D^u be the data structure resulting from updates u.

Lemma 6 *If $w \in \langle u \rangle_j$ then $D^u(i) = D^w(i)$ for all $i \notin R_j(u)$.*

Proof. First assume $i \in R_j(w)$. Since $[u]_j = [w]_j$ we have $\langle u \rangle_j \cup \{w\} = \langle w \rangle_j \cup \{u\}$. With the premise, this gives $\langle w \rangle_j \subseteq \langle u \rangle_j$ and hence

$$\bigcup_{z \in \langle w \rangle_j} r_j(z) \subseteq \bigcup_{z \in \langle u \rangle_j} r_j(z).$$

Subtracting on both sides the set $\bigcup_{k<j} R_k(u)$ (which is the same as $\bigcup_{k<j} R_k(w)$ because $w \in [u]_j$) we find $R_j(w) \subseteq R_j(u)$, see (7). Therefore $i \in R_j(u)$, so there is nothing to prove for this case.

Now assume $i \in R_k(u)$ for $k \neq j$ (else there is nothing to prove). There are two cases:

1. If $k < j$ then $(i, D^u(i)) \in R_{<j}(u)$, which agrees with $R_{<j}(w)$ by construction (6) (recall that $\langle u \rangle_j \subseteq [u]_j$).
2. Finally consider $k > j$. From the above claim, the last time i was written during u's updates was prior to the updates to layer j. Hence $D^u(i)$ is a function of the $(j-1)$th prefix of u.

   Now consider the partition induced by w and consider $R_l(w) \ni i$. First observe that $l > j$: we already considered $l = j$ in the beginning of the proof, and $l < j$ would imply $k < j$ because $R_{<j}(w) = R_{<j}(u)$. By the same reasoning as above, $D^w(i)$ is a function of the $(j-1)$th prefix of w. But u and w disagree only on the jth substring, so their prefixes up to that part are the same.

This finishes the proof. ∎

Claim *$\#R_j(u) \leq t_u m_j(1 + 1/\epsilon)$ for all $0 \leq j \leq h$ and $u \in U$.*

Proof. There are m_j nodes at layer j, each is updated at most once, incurring t_u operations. Hence $\#r_j(u) \le t_u m_j$. Since $\#R_0(u) = \#r_0(u)$ the claim holds for $j = 0$. For $j > 0$ we have $\#R_j(u) \le \#\langle u\rangle_j t_u m_j$ from the definition (7); the claim holds because $\#\langle u\rangle_j \le 1 + 1/\epsilon$. ∎

Lemma 7 $\Pr_{u\in U}\big[W(\langle u\rangle_j) \ge \frac{1}{300} m_j\big] \ge \frac{1}{2}$.

Proof. Let $s = \binom{m_j}{\epsilon m_j}$ and $t = 2^{\epsilon m_j}$. If $\#[u]_j \ge s/t$ then $\langle u\rangle_j$ contains a cover of weight $\frac{1}{300} m_j$ by construction. Hence we only have to show

$$\Pr_{u\in U}\big[\#[u]_j \ge s/t\big] \ge \tfrac{1}{2}. \tag{8}$$

From the above claim we have $\#R_{<j} \le t_u(m_0 + \cdots + m_{j-1})(1+1/\epsilon) \le t_u(2m_{j-1}-1)(1+1/\epsilon) < 2t_u m_{j-1}(1+1/\epsilon) - 1$. Thus,

$$\#R_{<j} + 1 < \frac{2t_u m_{j-1}(1+\epsilon)}{\epsilon} = \frac{2t_u m_j(1+\epsilon)}{d\epsilon} = \frac{\epsilon m_j}{2b},$$

using (3). Each set $R_{<j}(u)$ consists of tuples from $[2^b] \times [2^b]$, so the number of different $R_{<j}(u)$ is

$$\#\{\, R_{<j}(u) \mid u \in U \,\} \le (2^b \cdot 2^b)^{\#R_{<j}} \le 2^{\epsilon m_j - 1} = \tfrac{1}{2}t.$$

Let $U' = \{\, u/w \mid w \in \mathbf{B}(m_j, \epsilon m_j) \,\}$. The sets $[u]_j$ form a partition of U'. Each such set corresponds to exactly one $R_{<j}(u)$, so there are at most $\frac{1}{2}t$ different sets $[u]_j$. Call a set *small* if it contains fewer than s/t elements. The small sets contain fewer than $\frac{1}{2}s = \frac{1}{2}\#U'$ elements in total, so

$$\Pr_{u'\in U'}\big[\#[u']_j \ge s/t\big] \ge \tfrac{1}{2}.$$

Now (8) follows since the distribution on U conditioned on agreeing with u on all layers but the jth is the uniform distribution on U'. ∎

We are ready to prove Theorem 1.

Proof of Theorem 1. We prove the theorem for the (computationally easier) existential problem.

Pick $u \in U$ uniformly at random and perform the corresponding sequence of updates. Pick a leaf $v \in [m_h]$ uniformly at random and independently of u and perform the query $q(v)$. We will argue that with good probability over the choices of u and v, the query algorithm has to read registers from a constant fraction of the sets $R_j(u)$ $(1 \le j \le h)$. Since the sets are disjoint, this incurs at least as many read operations.

The probability that $\pi(v)$ contains no marked nodes is given by (5), so with that probability, the query returns 'no.'

Consider layer j and assume that the query algorithm does not read any register from $R_j(u)$. By lemma 6, it cannot distinguish u from any $w \in \langle u\rangle_j$ in the sense that its response after these update sequences is the same. The number of nodes at layer j marked by some $w \in \langle u\rangle_j$ is $W(\langle u\rangle_j)$. We call these nodes *potentially marked*—since some w marks these nodes and the algorithm is correct, a query that does not inspect $R_j(u)$ returns 'yes.' In other words: to be sure that a potentially marked nodes is unmarked, the algorithm must read a register from $R_j(u)$.

Now we show that a typical $\pi(v)$ contains many potentially marked nodes. Let v' denote the node on $\pi(v)$ at layer j. Since v is chosen uniformly at random among the leaves, and T is regular, v' is also chosen uniformly at random among the nodes at layer j. Pick any node uniformly at random (and independently of u) from the jth layer and let $X_j \in \{0,1\}$ be the indicator random variable for picking a potentially marked node in such an experiment. By Lemma 7 this happens with probability at least $\frac{1}{600}$, so $E(X_j) \ge \frac{1}{600}$. Performing this trail at every level, we expect to pick $\sum_{j=1}^{h} E(X_j) = \Omega(h)$ potentially marked nodes, by linearity of expectation.

Thus the path $\pi(v)$ is expected to contain $\Omega(h)$ potentially marked nodes. However, by (5), with probability $1 - o(1)$ the path contains no marked nodes at all. Thus with probability $\frac{1}{2} - o(1)$, the path $\pi(v)$ is unmarked but contains $\Omega(h)$ potentially marked nodes, each of which forces the query to read a register from the corresponding $R_j(u)$.

To prove the last statement in the theorem, note that if we start with T entirely unmarked, the update sequence is incremental. Likewise, if we start with T entirely marked, the update sequence is decremental. ∎

Comparison with the time stamp method. We briefly compare our partition to previous approaches. This also sheds a little light on the construction, showing why a more straightforward partition breaks the proof.

In the *time stamp* (or, *chronogram*) method of Fredman and Saks, layers correspond to *epochs*. A register 'has time stamp j' if it was written in the jth epoch (i.e., it belongs to $r_j(u)$) and no later. In our notation, this amounts to the partition

$$R_j^{\mathrm{ts}}(u) = r_j(u) \setminus \bigcup_{k<j} r_k(u),$$

cf. (7).

Consider the straightforward data structure

$$D(i) = \begin{cases} 1, & i \text{ marked,} \\ 0, & i \text{ unmarked.} \end{cases}$$

The query simply inspects all $D(i)$ for $i \in \pi(v)$. Start with an unmarked tree and $D(i) = 0$ for all i. To process an update sequence u it suffices to change the registers corresponding to marked nodes. Hence $R_j^{\mathrm{ts}}(u) = \{\, i \mid$

node i is marked }, while the registers corresponding to unmarked nodes are all in $R^{\mathrm{ts}}_{h+1}(u)$. The sets encountered on a path are R^{ts}_{h+1}, containing all registers with value 0, and R^{ts}_j for every layer j that contains a marked node on $\pi(v)$. Thus the number of such sets is $1 + \#(\pi(v) \cap M)$, with expected value $1 + h\epsilon$. A tail estimate shows that this value attains k with probability exponentially small in k. We conclude that typically only a few different time stamps are encountered, compared to $\Omega(h)$ different sets in our partition.

3.2. Amortised bounds

For the fully dynamic marked ancestor problem, our lower bound extends to the *amortised* cost per operation.

Theorem 2 *For the fully dynamic marked ancestor problem and for every $m > n$ there exists a sequence of m intermixed updates and queries taking time*

$$\Omega\left(\frac{m \log n}{\log(b \log n)}\right).$$

We omit the proof, which is an extension of the techniques used to prove Theorem 1.

3.3. Counting problems

We derive slightly better bounds (indeed, the best bounds known for any concrete problem in the cell probe model), by putting more information than just two bits into every node of the tree.

Theorem 3 *The ancestor sum problem with domain size $m = 2^b$ satisfies the trade-off* (2)*.*

We remark that this theorem can also be proved with the register partition used in the time stamp method.

4. Algorithms for Static Trees

Theorem 4 *Using linear time for preprocessing and linear space, we can maintain a tree with n nodes on-line with the following worst case time complexities.* $O(\log \log n)$ *for* mark *and* unmark *and* $O(\log n / \log \log n)$ *for* firstmarked*. Furthermore, the first k marked ancestors of any node can be found in worst case time* $O(\log n / \log \log n + k)$*. If the only query supported is* exists *then* mark *can be supported in time* $O(1)$ *and* exists *in time* $O(\log n / \log \log n)$*.*

To achieve this result we partition the nodes of the tree T into sets called *micro trees*, such that each micro tree is a connected subtree of the original tree. The set of micro tree roots induces another tree called the *macro tree*. In the macro tree, two nodes are incident iff the unique path in T between them contains no other micro tree root nodes. The macro tree is used only to simplify the presentations of the algorithms. A node is *heavy* if it belongs to the same micro tree as at least two of its children. To each of the micro trees we associate a level. If the micro tree is represented as a leaf node in the macro tree it has level 0, otherwise its level is one greater than the maximum level of its children in the macro tree. An *A-universe* of a tree has the following defining properties:

- Each micro tree has at most $O(\log n)$ heavy nodes.
- The maximum level of a micro tree is $O(\log n / \log \log n)$.

Note that the above definition does not limit the number of nodes in a micro tree.

Lemma 8 *For each tree there exists an A-universe and it can be constructed in linear time.*

The proof of the lemma can be found in [3]. In the next section we give the proof of how to maintain an A-universe for a tree which grows under addition of leaves.

Lemma 9 *Let T be a tree that has been divided into an A-universe. Assume that the marked ancestor problem in each micro tree can be solved in time t_1 for* firstmarked *and t_2 for* exists*. Then the marked ancestor problem for T can be solved with the same update time as in the micro trees and with query time* $O(t_1 + t_2 \log n / \log \log n)$*.*

Essentially we use the *exists* query to examine each of the $O(\log n / \log \log n)$ micro trees above the queried node until we find one with a marked ancestor to the queried node and then use a *firstmarked* query in the micro tree to locate it. The full proof of Lemma 9 is given in [3].

To prove Theorem 4 we have to show how to maintain micro trees efficiently, establishing the complexity of t_1 and t_2 in the above lemma. This is done below. The full proof, including how to find the first k ancestors efficiently, can be found in [3].

4.1. How to maintain micro trees

High level description Essentially we divide a micro tree into $\log n$ disjoint paths. In a single word we can for each of these paths store whether it includes a marked node or not. To answer a query we use the word to in constant time find the first path that includes a marked ancestor. This reduces the micro tree problem to a path problem. On each path P we maintain the minimum depth of a marked node on the path, $\min(P)$. The value of $\min(P)$ is sufficient to detect if a node on the path has a marked ancestor, establishing the complexity $t_2 = O(1)$ for *exists*. If we only have to answer *exists* queries, it is sufficient to use a priority queue on the path, allowing us to perform *mark* in constant time and *unmark* in time $O(\log \log n)$. If we also have to detect the

first marked ancestor on a path we use a Van Emde Boas search structure for the marked nodes on the path, giving the complexity $O(\log\log n)$ for *firstmarked* and the update operations. Details of micro trees computation and fast update (*mark*) priority queue can be found in [3].

5. Algorithms for Dynamic Trees

In this section we show how to extend the data structure above to be maintained for trees that grow under addition of leaves. In the final version of this paper we show how to extend the techniques to the link operation. Here, we show the following theorem.

Theorem 5 *We can maintain a tree under addition of leaves, with amortised complexity* $O(1)$ *for* add leaf, *worst case* $O(\log\log n)$ *for* mark *and* unmark *and worst case* $O(\log n/\log\log n)$ *for* firstmarked, *using linear space. Furthermore the first k marked ancestors can be reported in worst case time* $O(\log n/\log\log n + k)$.

Using standard methods we can also support deletion of a leaf in worst case constant time. Each time we add a new leaf, we check if more than half of the nodes have been deleted, if this is the case we rebuild the structure. If n not is known in advance we simply guess a constant and each time we exceed the guess we double our guess.

5.1. Adding a leaf

In this section we show how to maintain an A-universe for a tree that grows under additions of leaves. The structure is maintained such that each micro tree includes at most $2\log n$ heavy nodes. First we show how to limit the maximum level of a micro tree to $O(\log n/\log\log n)$ and next we will be more specific of how to maintain the structures for the micro trees and the complexity for that part.

A new leaf is added to the same micro tree as its parent belongs to if the parent is in a micro tree at level 0. Otherwise the new leaf becomes a single node in a new micro tree at level 0. When a micro tree μ contains more than $2\log n$ heavy nodes it is divided as follows. Let v be a node in the micro tree μ, such that if we remove the path P from the root of the micro tree μ to v, the micro tree is divided into a forest of trees, where each of these trees contains at most $\log n$ heavy nodes. Each of these trees is now treated as new micro trees on the same level as the micro tree μ was before the update. The path P is moved to the next level. Here we have two cases, depending on the level of the micro tree above μ. If the above micro tree level is precisely one larger, we move the path P up in this micro tree, otherwise we create a new micro tree on the next level, just containing the path.

Lemma 10 *The maximum level of a micro tree is* $O(\log n/\log\log n)$.

Proof. We will use a witness argument by induction on the level L. A node in the tree T can be a witness node for an ancestor. Two nodes in the same micro tree have no witness nodes in common. By induction we will prove: For a micro tree in level L we can for each heavy node find $\log^L n$ witness nodes. Proving this we can establish the lemma, since $L \leq \log n/\log\log n$. For micro trees at level 0 (micro trees in the bottom of the tree) the statement is clearly true, each heavy node is a witness to itself (and hence, no node is a witness node for more than one heavy node at level 0). When a micro tree is constructed it contains at most $\log n$ heavy nodes and when it is divided it contains $2\log n$ heavy nodes. Thus, from construction to destruction of a micro tree at level L at least $\log n$ new heavy nodes have been added to the micro tree. Each of these heavy nodes has by induction hypothesis been associated with $\log^L n$ witness nodes, in total $\log^{L+1} n$. Denote these witness nodes as S. When we divide a micro tree at level L, we add a new heavy node to a micro tree at level $L+1$. To the new heavy node at level $L+1$ we associate the $\log^{L+1} n$, witness nodes S. Hence, it only remains to show that the method used to associate witness nodes is such that no two nodes in the same micro tree have witness nodes in common. However, this follows immediately from the method we have used to associate witness nodes. At any level we have for any node that the set of witness nodes associated to a node v only will be associated to a new node the first time the micro tree v belongs to is divided. Hence, the same node is at most once associated as a witness nodes to a specific level. ∎

Observation 11 *At most* $O(\log n/\log\log n)$ *times is a node moved up to a new level, since a node always is moved up to a micro tree in a level one greater than the node former level.*

As in the static tree we divide a micro tree μ into disjoint paths. Thus, when a node is moved from one micro tree to another, it is also moved from one path to another. Implying it should be extracted from the search structure for one path and inserted in the search structure for the new path it belongs to. Since each such update can be performed in $O(\log\log n)$ time, it follows from the above observations:

Lemma 12 *Moving nodes from one path to another has total complexity* $O(n\log n)$.

Recall that we with each micro tree maintain a tree MT induced on the disjoint paths in the micro tree, which was represent in a single word. Thus, when a path is moved from one micro tree to another, MT trees should be created and updated. However, the total size of the two MT trees the path is moved between is $O(\log n)$, and MT trees can be updated/created in a time linear to the size, giving the complexity $O(\log n)$ for that part, which does not exceed

the complexity of moving nodes from one path to another. Details of maintaining MT trees can be found in [3] and we summarise:

Lemma 13 *We can achieve the results as expressed in Theorem 5 with the exception that adding a leaf has complexity* $O(\log n)$.

To reduce the complexity for adding a leaf to constant time, establishing Theorem 5, we essentially show how to efficiently collect paths of length $\log n$ that are inserted (and regarded) as a single node in a simple extension of the above algorithm. Details can be found in [3].

References

[1] P. K. Agarwal. Range searching. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 31, pages 575–598. CRC Press LLC, 1997.

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

[3] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. Technical Report RS-98-16, BRICS, 1998.

[4] S. Alstrup, J. Secher, and M. Spork. Optimal on-line decremental connectivity in trees. *Inf. Process. Lett.*, 64(4):161–164, 1997.

[5] A. Amir, M. Farach, R. Idury, J. L. Poutré, and A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, June 1995. See also SODA'93.

[6] P. Beame and F. Fich. On searching sorted lists: A near-optimal lower bound. Manuscript, 1997.

[7] N. Blum. On the single operation worst-case time complexity of the disjoint set union problem. *SIAM J. Comput.*, 15:1021–1024, 1986.

[8] B. Chazelle. Lower bounds for orthogonal range searching: II. the arithmetic model. *Journal of the ACM*, 37(3):439–463, July 1990.

[9] P. Dietz. Maintaining order in a linked list. In *Proc. 14th Symp. Theory of Computing (STOC)*, pages 122–127, May 1982.

[10] P. Dietz. Fully persistent arrays. In *Proc. 1st WADS*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74, 1989.

[11] P. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1st WADS*, volume 382 of *Lecture Notes in Computer Science*, pages 39–46, 1989.

[12] P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In *Proc. 4th European Symp. on Algorithms (ESA)*, Lecture Notes in Computer Science, pages 107–120, 1996.

[13] G. S. Frandsen, T. Husfeldt, P. B. Miltersen, T. Rauhe, and S. Skyum. Dynamic algorithms for the Dyck languages. In *Proc. 4th WADS*, volume 955 of *Lecture Notes in Computer Science*, pages 98–108, 1995.

[14] G. S. Frandsen, P. B. Miltersen, and S. Skyum. Dynamic word problems. In *Proc. 34th Symp. Found. of Comp. Sc. (FOCS)*, pages 470–479, 1993.

[15] G. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997.

[16] M. L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, Jan. 1982.

[17] M. L. Fredman. Lower bounds for dynamic algorithms. In *Proc. 4th SWAT*, volume 824 of *Lecture Notes in Computer Science*, pages 167–171, 1994.

[18] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Symp. Theory of Computing (STOC)*, pages 345–354, 1989.

[19] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and Systems Sciences*, 47:424–436, 1994.

[20] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and Systems Sciences*, 30(2):209–221, 1985.

[21] T. Husfeldt and T. Rauhe. Hardness results for dynamic problems by extensions of Fredman and Saks' chronogram method. In *Proc. 25th Int. Colloq. on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, 1998.

[22] T. Husfeldt, T. Rauhe, and S. Skyum. Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. *Nordic Journal of Computing*, 3(4):323–336, 1996.

[23] J. McCreight. Priority search trees. *SIAM J. Comput.*, 14:256–276, 1985.

[24] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[25] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. In *Proc. 27th Symp. Theory of Computing (STOC)*, pages 103–111. ACM, 1995.

[26] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Comput. Sci.*, 130:203–236, 1994.

[27] C. Montangero, G. Pacini, M. Simi, and F. Turini. Information management in context trees. *Acta Informatica*, 10:85–94, 1978.

[28] S. Muthukrishnan and M. Müller. Time and space efficient method-lookup for object-oriented programs. In *Proc. 7th Symp. on Discrete Alg. (SODA)*, pages 42–51, 1996.

[29] M. Thorup. On RAM priority queues. In *Proc. 7th Symp. on Discrete Alg. (SODA)*, pages 59–67, 1996.

[30] D. E. Willard. Lower bounds for the addition–subtraction operations in orthogonal range queries and related problems. *Information and Computation*, 82(1):45–64, 1989.

[31] D. E. Willard. Applications of the fusion tree method for computational geometry and searching. In *Proc. 3rd SODA*, pages 286–296, 1992.

[32] A. C. Yao. On the complexity of maintaining partial sums. *SIAM J. Comput.*, 14(2):277–288, May 1985.