Basic Research in Computer Science

Marked Ancestor Problems

(Preliminary Version)

Stephen Alstrup
Thore Husfeldt
Theis Rauhe

BRICS Report Series **RS-98-7**

ISSN 0909-0878 **April 1998**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/98/7/`

MARKED ANCESTOR PROBLEMS

PRELIMINARY VERSION

STEPHEN ALSTRUP, UNIVERSITY OF COPENHAGEN,
THORE HUSFELDT, LUND UNIVERSITY AND BRICS, AND
THEIS RAUHE, BRICS, UNIVERSITY OF AARHUS

Abstract. Consider a rooted tree whose nodes can be marked or unmarked. Given a node, we want to find its nearest marked ancestor. This generalises the well-known predecessor problem, where the tree is a path.

We show tight upper and lower bounds for this problem. The lower bounds are proved in the cell probe model, the upper bounds 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 intersection problems, proximity problems, and ray shooting. Our upper bounds improve a number of algorithms from various fields, including dynamic dictionary matching and coloured ancestor problems.

1. Introduction

Consider a rooted tree whose nodes can be marked or unmarked. Given a node we want to find its nearest marked ancestor, i.e., the first marked node on the path from the given node to the root. This generalises the well-known predecessor problem, where the tree is a path.

The technical contribution of the present paper is an analysis of the complexity of the marked ancestor problem; all our lower bounds

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 TFR. The content of this report is identical with the content of Technical Report DIKU-TR-98/9, Department of Computer Science, University of Copenhagen.

are given in the cell probe model, and our algorithms run on a unit-cost RAM. The results close a number of central open problems about other concrete computational problems that have been posed in the literature, some of which have received considerable attention. We briefly mention these corollaries below, because they make a strong case for the fundamentality of the ancestor problem. A detailed account of the applications is given in Sect. 2.

emptiness problem (aka. existential range queries): We prove optimal bounds on maintaining points in the plane and check if a given rectanglecontains any points. Finding a lower bound for this problem is Open Problem 1 in a recent handbook chapter on range queries by Agarwal [1].

range searching and partial sums: Lower bounds for range searching problems in the plane are known for structured or algebraic models [13, 22, 39, 42]. We extend these to the stronger cell probe model.

priority search trees: We show that Willard's RAM improvement [40] of McCreight's classic data structure [29] is optimal.

union–find problems: Gabow and Tarjan [26] showed that a version of the union–find problem (static tree union–find) is solvable in constant amortised time per operation, provably easier than the general problem. We show that with respect to *single operation worst case bounds*, the problem is not easier than the general problem.

lower bounds for computational geometry: Cell probe lower bounds are given for a number of fundamental problems from dynamic computational geometry, including interval maintenance, intersection, ray shooting and proximity problems.

tree colouring problems: Our algorithms for marked trees extend to coloured trees, improving a number of previous results [17, 36].

dynamic dictionary matching: Improved algorithms for dynamic dictionary matching [7].

cell probe complexity: The largest trade-off known for any concrete problem in this model.

parallel vs. dynamic computation: We exhibit a concrete function that is easy for parallel computation (it is in AC^0) but hard in the dynamic sense.

The marked ancestor problem. We let T denote 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 node v to the root will be denoted $\pi(v)$,

which includes v and the root. The *depth* of a node v is $\mathrm{depth}(v) = |\pi(v)| - 1$, its distance to 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 the first marked node on the path from v to the root, i.e., the marked node of largest depth on $\pi(v)$.

The *incremental* problem does not support *unmark*, the *decremental* problem does not support *mark*, while the *fully dynamic* problem supports both update operations.

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},

$$t_{\mathrm{q}} = \Omega\Big(\frac{\log n}{\log(t_{\mathrm{u}} b \log n)}\Big). \tag{1}$$

Previously, bounds of this size were known for counting problems, e.g., returning the number of marked predecessors.

To prove 1 we use an argument similar to the *chronogram* or *time stamp* technique of Fredman and Saks [25], and we re-prove their Thm. 3 using our framework for expository reasons. Interestingly, while our result is stronger than [25], the proof is simpler.

For logarithmic word size the bound (1) implies a lower bound of $\Theta(\log n/\log\log n)$ per operation. The bound 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.

updates		complexity pr. operation	
mark	*unmark*	amortised	worst case
•		$\left.\begin{matrix}\\ \end{matrix}\right\}$ O(1)	$\left.\begin{matrix}\\ \\ \end{matrix}\right\}\Theta(\log n/\log\log n)$
	•		
•	•	$\Theta(\log n/\log\log n)$	

TABLE 1. Complexity of marked ancestor problems for logarithmic word size.

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

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 [20, 26], 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.

Comparing upper and lower bounds we see that the query time is optimal, even if we would allow polylogarithmic time for each update or consider amortised bounds. The only target for improvement is the double-logarithmic update time; we can exhibit constant time bounds for some special cases of the problem, but the gap remains an open problem even if T is a path.

Variants and extensions.

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

exists(v): return 'yes' if $\pi(v)$ contains any marked nodes.

We can show that our lower bound (1) holds even for this, potentially easier problem, which is our main tool for proving lower bounds in §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

$$t_{\mathrm{q}} \log t_{\mathrm{u}} = \Omega(\log n) \tag{2}$$

between the amortised update t_{u} and query time t_{q}. Note that this bound does not depend on the register size.

If the update operation is somewhat restricted (the change to $\mathrm{val}(k)$ 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 [16]. By a related technique we show how 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 [19].

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 $s \leq k$ is the size of the output.

Adding leaves. The variants of the marked ancestor problem studied in [7, 26] 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, yielding improved algorithms for dynamic dictionary matching.

Summary. Table 1 shows the complexity of each 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*.

updates			query	remarks
mark	*unmark*	*add leaf*	*firstmarked*	
$O(1)^*$	–	$O(1)^*$	$O(1)^*$	Gabow and Tarjan [26]
–	$O(1)^*$	$O(1)^*$	$O(1)^*$	Gabow and Tarjan [26]
$O(\log \log n)$	$O(\log \log n)$	$O(1)^*$	$\Theta(\log n / \log \log n)^\dagger$	finds k first marked ancestors in time $O(k + \log n / \log n \log \log n)$
			exists	
$O(1)$	$O(\log \log n)$	$O(1)^*$	$\Theta(\log n / \log \log n)^\dagger$	

*Amortised time bound

†Optimal even for polylogarithmic update time

TABLE 2. Upper bounds for algorithms supporting various combinations of updates and queries; a dash indicates lack of support for a particular operation.

Computational models. 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 of Fredman [21] and A.C. Yao [41]. The model 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.

Nondeterminism. The worst case lower bounds for the marked ancestor problem can be seen to hold even if we allow a model with *nondeterministic* queries as introduced in [27]. In short this model extends query computation with the additional power of accessing the data structure through nondeterministic guesses and such that positive query answers are verified in the usual sense. This should be contrasted with the fact that if T is a path, there is a provable gap between nondeterministic and deterministic query time as observed in [27], since nondeterminism allows worst case constant time complexity for *firstmarked* while supporting *mark* and *unmark* in worst case $O(\log \log n)$ time.

Furthermore, the existential marked ancestor problem has nondeterministic query algorithms that take constant time: guess a node on $\pi(v)$ and verify that it is marked. However, the complement of the problem, to verify that a path contains *no* marked nodes, does not allow such an algorithm. Indeed, our lower bound holds for the co-nondeterministic complexity of the query.

2. Applications

In this section we consider logarithmic word size for concreteness, unless otherwise stated.

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 determine whether

$$S \cap R \stackrel{?}{=} \emptyset,$$

for rectangle R. 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 or decremental variants, and also for the amortised query time of the fully dynamic problem.*

We sketch the proof: First observe that the proofs of the lower bounds on the marked ancestor problem hold even if the tree T is regular. Now embed such a tree 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 $y = -x + \delta^h - \delta^{h-i}$, where $\delta = \log^{O(1)} n$ 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.

This problem may be the most convincing application of our lower bound (1); it holds in the cell probe model, so it makes minimal assumptions on the computational model and none on the data structure. The upper bound for this problem is slightly larger, amortised $O(\log n \log\log n)$ using fractional cascading [31], yet for dominance queries it is $O(\log n / \log\log n)$ using [29, 40], matching our lower bound.

The traditional, algebraic models 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 [13, 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. That bound uses a completely different technique based on a result of Ajtaj [2], this can be proved by combining the reduction of Miltersen, Nisan, Wigderson, and Safra [32, Thm. 18] with a bound by Beame and Fich [10].

2.2. **Priority Search Trees.** *Priority searching*, a mixture of a search structure and a priority queue sometimes nicknamed '$1\frac{1}{2}$-dimensional searching,' 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 [29] implements these operations (and a few more) in logarithmic time, Willard [40] 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 $O(\log^{1/2} n)$ and $O(\log\log n)$, 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 the last section. 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 and priority searching examples is range queries, where we want answers to questions 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. For many choices of algebraic structure, like $(\{0,1\}, \oplus)$ or $(\mathbf{N}, +)$, the paper of Fredman and Saks [25] provides good lower bounds both in the group and in the cell probe model, and Frandsen, Miltersen, and Skyum [18] 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 for $(\mathbf{N}, \max)$ or $(\{0,1\}, \vee)$.

Many bounds for range searching are provided in a variety of algebraic or structured models [13, 22, 39, 42]. However, these bounds are sometimes overly pessimistic. For example it is known that the one-dimensional partial sum problem for $(\{0,1\}, \vee)$ can be solved in time $O(\log \log n)$ per operation on a RAM [38], exponentially faster than Yao's lower bound in an algebraic setting of $\Omega(\log n / \log \log n)$ [42]. As Agarwal explains, this is because bounds in the semigroup model assume "the weights of points to be part of the input. That is, the data structure is not tailored to a special set of weight, and it should work for any set of weight. It is conceivable that faster algorithm can be developed for answering orthogonal range-counting queries, exploiting the fact that the weight of each point is 1 in each case." [1, p. 579]. To extend the semigroup lower bounds to stronger models, e.g., to allow subtraction, has been (and for dimensions higher than 2 still is) an open problem for more than a decade, posed by Fredman [22], A.C. Yao [42], and lately by Agarwal [1].

2.4. **Union–find problems.** Gabow and Tarjan [26] 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, [26] show that on a random access machine, 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 [11]. For the general union–find problem, cell probe lower bounds of the same size were provided in [25].

2.5. **Tree colour problems.** In the *tree colour problem* [17] we associate with each node of T a set of colours. The following operations are considered in [17]:

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.

The problem arises in Object Oriented Languages (OOL) [17, 36] to handle methods in a program. For a long period of time very similar problems have been investigated in areas like persistent data structures, logic programming and context trees, see, e.g., [14, 15, 34]. It is known that the colour problem can be solved using $O(\log n/\log\log n)$ time for both updates and queries, some of the best results can be found in [7, 36], see [17] for a list of references. In [17] on the background of empirical examination it is documented that long update times are unacceptable for this problem, and a randomized algorithm is given with complexity $O\left((\log\log n)^2\right)$ per update, but at the cost of increasing the query time to $O(\log n/\log\log\log n)$. With our algorithms we can provide even better update times without slowing down the queries.

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

The construction can be found in the appendix and is a simple extension of our marking algorithm using dictionaries for the colours.

2.6. **Dynamic dictionary matching.** Our algorithm with the *add leaf* extension has many applications, e.g., it can be used in dynamic variants of the above problem [17, 36]. However, here we will concentrate on the application studied by Amir, Farach, Idury, La Poutre and Schaffer [7], called *dynamic dictionary matching*, see [9, 5, 8, 6, 7] for further references to this problem.

The problem, a generalisation of pattern matching, is to maintain a set of *patterns* P_i (the *dictionary*) and a *text* T over a bounded alphabet. An update operation adds or removes patterns from the dictionary, and the query is

scan: return all pairs (i, j) such that pattern P_i matches the piece of text beginning the jth letter of T.

By combining our algorithm with the reduction of [7] we obtain the following bounds:

insert/delete(P): $\mathrm{O}(\log\log d + |P|\log\sigma)$,
scan: $\mathrm{O}\big(tocc + |T|(\log d/\log\log d + \log\sigma)\big)$,

where d is the total length of all patterns in the dictionary, $\sigma \le d$ is the number of distinct letters in the dictionary, and *tocc* is the number of pattern occurrences (i.e., the size of the output). The bounds given in [7] are $\mathrm{O}\left(|P|(\log d/\log\log d + \log\sigma)\right)$ and $\mathrm{O}\left((|T| + tocc)\log d/\log\log d + |T|\log\sigma\right)$.

The bound is obtained by improving the prefix lookup problem of [7], where the updates are as above and the query is

lookup(j, k): output the patterns that are a prefix of the first k letters of P_j.

Using our algorithm for the marked ancestor problem in the reduction of [7, Theorem 3.37, Lemma 3.1], we obtain query time $\mathrm{O}\left(|out_{j,k}|\log d/\log\log d\right)$ for this problem, where $out_{j,k}$ is the output.

2.7. **Lower bounds for dynamic algorithms.** It is easy to dress up the emptiness problem to provide lower bounds for a number of central problems in computational geometry, a field where lower bounds for many problems only exist in structured models, even though there is a large interest in dynamic algorithms, also on the RAM.

For example, using lines instead of points in our reduction, we receive a lower bound for orthogonal segment intersection, which in turn gives lower bounds for several similar problems like ray shooting, i.e., finding the first object in the intersection. *Proximity problems* are at least as hard as orthogonal range queries for some metrics (but not the Euclidean), so our lower bound holds for these.

A similar construction shows the same bounds for interval (or, segment) trees, i.e., to maintain a set of intervals of the form $[i, j]$, $1 \leq i, j \leq n$, under insertions and deletions. The query returns the name of an interval containing a given point.

Cell probe bounds for other geometric problems (point location) are given in [28, 27], and some more can be seen to follow from [25].

2.8. **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

$$t_{\mathrm{u}} \cdot t_{\mathrm{q}} = \omega(\log n / \log\log n).$$

Our lower bound can be seen as a response to Fredman's challenge [23], even though we still cannot improve

$$t_{\mathrm{u}} + t_{\mathrm{q}} = \Omega(\log n / \log\log n),$$

which is arguably more interesting. Note that the word size does not appear in (2) (but it does appear in the statement of the problem).

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

$$\bigvee_{1 \leq i \leq n} x_{2i} \wedge \Big(\bigvee_{v_j \in \pi(v_i)} x_j \Big).$$

This precisely models the existential marked ancestor problem 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. [33]. This contrasts the work in [27] were it is proved that for *symmetric* Boolean functions, there is a close correspondence between parallel and dynamic hardness.

3. Lower bounds

3.1. **Preliminaries.** Our lower bounds work in the cell probe model, a nonuniform version of the unit-cost RAM, where memory access is the only resource. We consider registers of size bounded by b bits (a typical choice is $b = \Theta(\log n)$). We let M denote the memory. The maximal time for any update (*mark* or *unmark*) is denoted t_{u}, and the maximal time for the query (varying with the particular problem) is t_{q}.

The proofs for our lower bounds use an information theoretic argument in which it is necessary to limit the number of memory registers that can be accessed by the algorithm. In the RAM model the quantity 2^b bounds the number of registers that can be accessed, and this would

be a sufficient bound for our proofs. But in the cell probe model we need a more careful analysis. Instead we will split M into two parts; early registers and late registers. The *early* registers are those that can be reached by a query computation within the first $\lceil \log n \rceil$ memory reads. That is those registers occuring within depth $\lceil \log n \rceil$ in any of the cell probe query trees. The late memory is the remaining memory registers in the description of the algorithm. Clearly for the worst case bounds, we can without loss of generality restrict our attention to registers in early memory since a single read in late memory immediately witnesses a query computation using time $\Omega(\log n)$ (and we never prove bounds better than this). For the amortised bound, it also turns out that the bound $\Omega(\log n)$ for every query computations reading from late memory in our hard operation sequences is sufficient to establish the overall amortised bound. Let R denote the least power of two larger than the total number of registers in early memory. Clearly $\log R \in \mathrm{O}(b \log n)$. For the rest of this section C denotes the quantity $b + \log R$, so the number of early memory configurations that differ on x registers is bounded by 2^{Cx}.

We can view a marked tree $T = (V, E)$ as a labeling $e\colon V \mapsto \{0, 1\}$, where $e(v) = 1$ iff v is marked, and in general we will extend our labelings to larger domains D. Thus let $e : V \mapsto D$ be a *labeling* of the nodes V of tree T. For any subset $W \subseteq V$ the *partial labeling* $e|_W\colon W \mapsto D$ denotes the restriction of e to W. Let F be a set of labelings $V \mapsto D$. Let F^W denote the set of partial labelings $W \mapsto D$ that are a restriction to W of some labeling in F, i.e., $F^W = \{\, e|_W \mid e \in F \,\}$. We partition V into *layers* $W_1, W_2, \ldots, W_h$ according to depth, so the ith layer W_i contains the nodes of depth i.

Consider a family $\mathcal{U}$ of operation sequences consisting of updates and queries. Let e^I denote some initial labeling of T. We let e^u denote the labeling resulting from execution of the updates of sequence u starting from initial input configuration corresponding to e^I. Define the set of *reachable labelings* as:

$$F_{\mathcal{U}} = \{\, e^u \mid u \in \mathcal{U} \,\},$$

we often drop the subscript $\mathcal{U}$. A probability distribution on $\mathcal{U}$ induces a probability distribution on $F_{\mathcal{U}}$ by letting the probability of picking $e \in F_{\mathcal{U}}$ be

$$\Pr(e) = \sum_{u \in \mathcal{U} : e^u = e} \Pr(u).$$

In a similar way for subsets of nodes W the probability distribution on $\mathcal{U}$ induces distributions on the set of partial labelings F^W. We say

that a distribution on $\mathcal{U}$ *labels* T *uniformly* with respect to $F_{\mathcal{U}}$ iff the distributions on $F_{\mathcal{U}}$ and $F_{\mathcal{U}}^{W_t}$ for all $1 \leq t \leq h$ are uniform.

To each register in early memory we associate an *age*. A register has age x in M if there have been performed x write operations since the register was written. We say two early memory configurations M, M' are x-equivalent, denoted $M \equiv_x M'$, iff the set of registers of age x or less and contents are the same. Let $M[u]$ denote the early memory configuration resulting from execution of sequence $u \in \mathcal{U}$ beginning with initial memory configuration M. Define $Z(u, x, W) \subseteq F^W$ relative to a family $\mathcal{U}$ by:

$$Z(u, x, W) = \{\, e^{\hat{u}}|_W \mid \hat{u} \in \mathcal{U} \text{ where } M[u] \equiv_x M[\hat{u}] \,\}.$$

For $Z \subseteq F^W$ we associate a set of *unfixed* nodes $\rho(Z) \subseteq W$ defined by;

$$v \in \rho(Z) \text{ iff } \exists e, e' \in Z : e(v) \neq e'(v).$$

Lemma 5. *Consider a probability distribution on a family* $\mathcal{U}$ *of operation sequences that labels* T *uniformly with respect to* F*. Then for any layer of nodes* W *and* $x > 0$*:*

$$\Pr_{u \in \mathcal{U}}(|Z(u, x, W)| \geq |F^W|/2^{Cx+1}) \geq \tfrac{1}{2}.$$

Proof. Consider the equivalence relation where $e_1, e_2 \in F^W$ are equivalent iff there exists $u_1, u_2 \in \mathcal{U}$ such that $e_1 = e^{u_1}|_W$, $e_2 = e^{u_2}|_W$ and $M[u_1] \equiv_x M[u_2]$. Then the sets $Z(u, x, W)$ correspond to equivalence classes of F^W with respect to this relation. Hence we can bound the number of these classes of F^W by counting the number of early memory configurations differing among registers of age x or less which by the choice of C is bounded by 2^{Cx}. Hence the average size of a class $Z(e, x, W)$ is at least $|F^W|/2^{Cx}$. By the assumption that the probability distribution on F^W is uniform, the claim follows by a standard averaging argument. ∎

3.2. **Counting problems.** For expository reasons we start with yet another problem, which turns out to be particularly well suited for our technique. Here, the query asks for the parity of the number of marked nodes on the path from v to the root,

mark(v): mark v,
parity(v): return $\sum_{u \in \pi(v)} m(v) \mod 2$.

If T is a path then this is the parity prefix problem of Fredman and Saks [25], so its hardness is well known.

Theorem 1 (Fredman and Saks)**.** *The ancestor parity problem satisfies the trade-off* (1)*.*

Proof. Let T be a complete tree with n leafs and internal nodes of degree $\delta(n) \geq 4Ct_{\mathrm{u}}$ but such that $\log \delta(n) \in \mathrm{O}(\log(bt_{\mathrm{u}} \log n))$. Let h denote the height of T. Our goal is to establish the existence of an update sequence such that a random query performed immediately after this sequence performs a constant fraction of h read operations.

We argue probabilistic. Let F be the set of all $\{0,1\}$-labelings of T. We define a family $\mathcal{U}$ of update sequences relative to F such that there is an one-to-one correspondence between a labeling $e \in F$ and update sequence $u \in \mathcal{U}$. The correspondence is as follows. The update sequence labels T bottom-up such that after $|W_h| + |W_{h-1}| + \cdots + |W_i|$ updates, the actual labeling e' of T is:

$$e'(v) = \begin{cases} e(v), & v \in W_h \cup \cdots \cup W_i \\ e^I(v), & \text{otherwise.} \end{cases}$$

The order in which the nodes of each layer are updated can be chosen arbitrarily. Our choice of probability distribution on $\mathcal{U}$ are the uniform and by the bijection between F and $\mathcal{U}$ this clearly labels T uniformly.

Fix some layer $2 \leq t \leq h$ of the tree. Let x denote the number of registers written during updates for nodes of depth $t-1$ and above. Then

$$|W_t| = 4Ct_{\mathrm{u}}|W_{t-1}| > 2Ct_{\mathrm{u}}\Big|\bigcup_{i<t} W_i\Big| \geq 2Cx. \tag{3}$$

The size of F^{W_t} is $2^{|W_t|}$. Hence by Lemma 5 and (3) with probability at least a half we have $\log |Z(e,x,W_t)| \geq |W_t| - Cx - 1 \geq \frac{1}{2}|W_t|$. Since $\log |Z(e,x,W_t)| \leq |\rho(e,x,W_t)|$ we obtain for random $v \in W_t$ and $u \in \mathcal{U}$:

$$\begin{aligned} \Pr(\exists e_1, e_2 \in Z(u,x,W_t) : e_1(v) \neq e_2(v)) &\geq \Pr(v \in \rho(Z(u,x,W_t))) \\ &\geq \tfrac{1}{2}\Pr(|\rho(Z(u,x,W_t))| \geq \tfrac{1}{2}|W_t|) \\ &\geq \tfrac{1}{4}. \end{aligned}$$

Divide the registers into generations relative to their age. The registers belonging to generation t consists of those registers last written to during the updates for labels of depth t in T. Formally letting x and x' denote the number of write operations performed from and above layer $t-1$ and t respectively, the t generation is the registers of ages y where $x < y \leq x'$.

Pick a random $u \in \mathcal{U}$ and leaf l. If *parity*(l) reads outside early memory then by definition at least $\lceil \log n \rceil > h$ read operations has been performed before this. On the other hand if this is not the case we claim that for any t the probability that *parity*(l) reads a register from generation t is at least $\frac{1}{4}$. Let v be a node on the path from l to the root.

Let t denote the layer containing v. The contents of registers belonging to generation $t+1, t+2, \ldots, h$ can not depend on the value of $e(v)$, since the labeling of $e(v)$ is performed after these registers were changed for the last time. Furthermore if $\exists e_1, e_2 \in Z(u, x, W_t) : e_1(v) \neq e_2(v)$ then by definition of $Z(u, x, W)$ the labeling of v is also independent of the contents of registers of generation $t-1$ and below. Since this event occurs with probability $\frac{1}{4}$, and the answer for *parity*(l) depends upon $e(v)$, the algorithm needs to read a register from generation t with probability $\frac{1}{4}$ in order to distinguish the two different answers. By linearity of expectation we conclude that the query has lower bound $\Omega(h) = \Omega(\log n / \log(t_{\mathrm{u}} b \log n))$. ∎

The technique can be used to give 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. Recall the definition of the ancestor sum problem from Sec. 1 and let G denote the domain of values.

Theorem 2. *The ancestor sum problem satisfies the trade-off* (2).

Proof. Let T be complete with n leafs and out-degree $\delta(n)$ greater than $ct_{\mathrm{u}}^2 > 8t_{\mathrm{u}}^2$ for a constant c. Let h denote the height of T as before. Let F be the set of all labelings $V \mapsto G$.

In this proof we need to bound R in terms of the worst case update time t_{u} since the general bound $2^{b \log n}$ used so far, is to weak for very fast update t_{u}. In order to bound R we observe that we can restrict our attention to those registers which occur among the forest of update trees in the cell probe description of the algorithm. That is any register in some query tree which is *not* among these "update registers" is clearly superfluous and can without loss of generality be disregarded in our analysis. Furthermore for each node in tree T, there are at most $|G|$ possible valid assignments for which we have an associated cell probe update tree. Such update tree contains at most $|G|^{t_{\mathrm{u}}}$ nodes. That is $C = \log R + b$ is bounded by:

$$\log(2n|G||G|^{t_{\mathrm{u}}}) + \log|G| \leq 2t_{\mathrm{u}} \log|G| - 1.$$

The update sequences $\mathcal{U}$ are defined relative to F as in the previous proof, i.e., bottom-up and such that the probability distribution on $\mathcal{U}$ labels T uniformly with respect to F.

Again as before for fixed $2 \leq t \leq h$ and x denoting the number of registers written during updates performed for the nodes at layer $t-1$

and above we bound

$$|W_t| = 8t_{\mathrm{u}}^2|W_{t-1}| \geq 4t_{\mathrm{u}}^2|\bigcup_{i<t} W_i| \geq 4t_{\mathrm{u}}x.$$

Since $|F^{W_t}| = |G|^{|W_t|}$, by Lemma 5 and the bound on $|W_t|$ with probability a half

$$\begin{aligned}\log|Z(u,x,W_t)| &\geq |W_t|\log|G| - Cx - 1\\ &\geq |W_t|\log|G| - (2t_{\mathrm{u}}\log|G|)x \geq \tfrac{1}{2}|W_t|\log|G|.\end{aligned}$$

Since $\log|Z(u,x,W_t)| \leq |\rho(Z(u,x,W_t))|\log|G|$ this implies that $|\rho(Z(u,t,W_t))| \geq \frac{1}{2}|W_t|$ with probability a half. As in previous proof for random $v \in W_t$ and $u \in \mathcal{U}$ we obtain $\Pr(\exists e_1, e_2 \in Z(u,x,W_t) : e_1(v) \neq e_2(v)) \geq \frac{1}{4}$. The lower bound of $\frac{1}{4}h$ for the expected time for a query for a random leaf l follows as before, implying $t_{\mathrm{q}} \in \Omega(h) = \Omega(\log n/\log t_{\mathrm{u}})$.
∎

3.3. **Marked ancestor queries.** We now turn to our main result, the lower bound for ancestor queries.

The main difference in our construction is that the marking constructed by a typical update sequence is *sparse* in the sense that there is a fair chance that a node and its immediate ancestors are unmarked, thereby forcing the algorithm to examine a large portion of $\pi(v)$.

Theorem 3. *The marked ancestor problem and the existential marked ancestor problem problem satisfy the trade-off* (1)*. Moreover, this is true even for incremental or decremental updates.*

Proof. We show the result for the (computationally easier) existential problem.

Assume n is a power of two. Let T be a tree with n leafs and outdegree greater than $4Ct_{\mathrm{u}}\log n$ but $\mathrm{O}(Ct_{\mathrm{u}}\log n)$. Each layer of nodes W_i, $i \geq 2$, is partitioned into blocks $W_{i,j}$, $j \in [|W_i|/\log n]$, consisting of exactly $\log n$ nodes, i.e., $W_i = \bigcup_j W_{i,j}$, where $W_{i,j} \cap W_{i,j'} = \emptyset$ for $j \neq j'$ and $|W_{i,j}| = \log n$ for all j. Let F denote the set of labelings of T satisfying

$$\forall i,j : \sum_{v \in W_{i,j}} e(v) = 1$$

and root always zero. Let the update sequences $\mathcal{U}$ be defined relative to F as in the proof for Theorem 1. Note that depending upon the initial labeling, the update sequence can take the form as either an entirely incremental or entirely decremental update sequence.

Fix a layer $2 \leq t \leq h$. As before we can bound the number of nodes at layer t in terms of the number of write operations which take place during execution of updates associated layer $t-1$ and above. Let x denote this number of write operations. Then $|W_t| \geq 2Cx \log n$. By this bound and by Lemma 5 with probability a half:

$$Z(u, x, W_t) \geq |F^{W_t}|/2^{Cx+1} \geq |F^{W_t}|/2^{\frac{1}{2}|W_t|/\log n+1}.$$

For $Z(u, x, W_t)$ satisfying this bound, a simple counting argument shows that $|\rho(Z(u, x, W_t))| \geq \frac{1}{2}|W_t|$.

Hence we conclude as in previous proofs that the probability of the event:

$$\exists e_1, e_2 \in Z(u, x, W_t) : e_1(v) \neq e_2(v) \tag{4}$$

is at least $\frac{1}{4}$. In order to achieve the lower bound as in the previous proofs we need one more observation. Obviously the computation for an *exists* query returning 'yes' does not need to rely on all the labels on the path, i.e., a single node with 1 determines the answer. On the other hand, for a negative answer, the query computation is obviously sensitive to the label of any node on the path in question. That is for a query which returns answer 'no', every layer t that satisfies (4) above, witnesses a read operation of a register belonging to generation t. The probability of a 'yes' answer is less than $\sum_{2 \leq t \leq h} \log^{-1} n \leq \frac{h-1}{\log n}$, by the choice of F (there is only one uniformly chosen 1-entry in each block of size $\log n$). Hence for a random node on a random path from leaf l to root, the probability that the *exists*(l) computation reads a node from generation $l(v)$ is at least $\frac{1}{4} - \Pr(\textit{exists}(l) \text{ returns 'yes'}) \geq \frac{1}{8}$ for large n. Hence the expected number of read operations for *exists*(l) is $\frac{1}{8}h \in \Omega(\log n / \log(t_{\mathrm{u}} b \log n))$ as desired. ∎

3.4. **Amortised bounds.** Let t_{u} and t_{q} denote the amortised cost of updates and queries respectively, i.e., for $m, m' > n$, the amount of time used to perform m updates and m' queries is at most $mt_{\mathrm{u}} + m't_{\mathrm{q}}$.

Theorem 4. *For the fully dynamic marked ancestor problem and for every $m > n$ there exists a sequence of m intermixed updates and queries taking time*

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

Proof. Let T be a complete tree with n leafs and out-degree greater than $ct_{\mathrm{u}}C \log^2 n$ for an integer constant $c \geq 8$. We consider the same set of labelings F as in the proof of Theorem 3. The family of sequences $\mathcal{U}$ we consider is defined relative to a certain infinite traversal of the

leafs of T. Let $k \geq 2$ and consider k infinite sequences $A^i = \langle a_1^i, a_2^i, \ldots$ for $1 \leq i \leq k$. Then the merge of $A^1, \ldots, A^k$ is:

$$\mathit{merge}(A^1, \ldots, A^k) = \langle a_1^1, a_1^2, a_1^3, \ldots, a_1^k, a_2^1, a_2^2, \ldots$$

For a tree T' we define an infinite sequence of leafs $\mathit{seq}(T')$ inductively by; if T' is a leaf, say l, then $\mathit{seq}(T') = \langle l, l, \ldots$. Otherwise for a tree T' with the root having subtrees $T_1, T_2, \ldots, T_k$, the sequence are defined to be $\mathit{seq}(T') = \mathit{merge}(\mathit{seq}(T_1), \ldots, \mathit{seq}(T_k))$. Let $\mathit{seq}(T')[i]$ denote the ith element in $\mathit{seq}(T')$.

Assume the initial labeling e^I of T is a member of F. Let $m \geq n$. We are ready to define $\mathcal{U}$. It consists of all possible sequences of the form:

$$u_1 q_1 u_2 q_2 \ldots u_{2m} q_{2m}$$

where u_i is a small subsequence of $2h$ updates to be defined in a moment and each q_i is a query of the form $\mathit{exists}(l)$ for some leaf of T. After any prefix $u_1 q_1 \ldots u_j$ of a sequence $u \in \mathcal{U}$ the updates maintain the invariant that the actual labeling after execution of $u_1 q_1 \ldots u_j$ is a member of F. The updates in u_i are associated the path from leaf $l = \mathit{seq}(T)[i]$ to the root in T. For each $v \in \pi(l)$ two updates are performed. Let t denote the layer for v and let $W_{t,j}$ be the block of size $\log n$ containing v. The first update unmark the single node labeled one in $W_{t,j}$ and the next update assigns a single node among the $\log n$ nodes (including the node just unmarked) in $W_{t,j}$ to one. After execution of the updates u_i, the labeling of the nodes in $W_{t,j}$ does not depend on updates which have taking place before u_i in the sense that the choice of which node set to one in $W_{j,t}$ by u_i is never influenced by the actual labeling when executing of u_i.

It is straightforward to show that the prefix closure of family $\mathcal{U}$ labels T uniformly with respect to F. Let $j \geq m$ and let $u \in \mathcal{U}$. Let e_j denote the actual labeling of T after execution of the prefix up to q_j of u. We let $s(j,t)$ denote the subsequence $u_{j-|W_t|}, \ldots, u_j$ ending just before q_j and $\tilde{u} = u_1 \ldots u_j$ be the full prefix ending before q_j. Let $x(s^u(j,t))$ denote the number of write operations performed during execution of $s^u(j,t)$.

Let S^u be the set of pairs $\{\, (j,t) \mid x(s^u(j,t-1)) \leq 4t_{\mathrm{u}} \log n |W_{t-1}| \,\}$. That is S^u captures those subsequences which from an outside viewpoint performed as fast as if the updates were performed in worst case time t_{u}. The next two lemmata provides us with the necessary tools allowing us to argue as in the proof for the worst case bound of Theorem 3.

Lemma 6. *For any sequence $u \in \mathcal{U}$:*

$$|S^u| \geq \tfrac{1}{2}m(h-1).$$

Proof. By definition of t_{u} we have $\sum_{j \geq m} x(s^u(j,t)) \leq |W_t| 2mt_{\mathrm{u}}h$, i.e., the average size of $x(s^u(j,t))$ is less than $2t_{\mathrm{u}}h|W_t| \leq 2t_{\mathrm{u}} \log n|W_t|$. That is at least half of the js greater than m must satisfy $x(s^u(j,t)) \leq 4t_{\mathrm{u}} \log n|W_t|$, i.e., $(j, t+1)$ is in S^u. Hence for each $t < h$ there is $\frac{1}{2}m$ entries j such that $(j,t) \in S^u$ implying $|S^u| \geq \frac{1}{2}m(h-1)$. ∎

Lemma 7. *For any j and t the labeling $e_j(v)$ for any $v \in \bigcup_{i \leq t} W_i$ only depends on updates in $s(j,t)$.*

Proof. Let $\pi_0, \pi_1, \ldots, \pi_{|s(j,t)|-1}$ denote the paths associated the update subsequences in $s(j,t) = u'_0, u'_1, \ldots, u'_{|s(j,t)|}$. Using $|s(j,t)| = |W_t| + 1$ and by the construction of the traversal sequence of the leafs, it is easily proven by induction in t that $\bigcup_{i \leq t} W_i \subseteq \bigcup_j \pi_j$. Hence consider a node $v \in \bigcup_{i \leq t} W_i$, say at layer t', and let π_j be such path containing v. By definition of the update subsequence u'_j relative to π_j, the one entry within some block $W_{t',j'}$ containing v is affected by u'_j and hence v does not depend on updates prior to u'_j, i.e., only on $s(j,t)$. ∎

As before we divide the registers into generations relative to their age, and keep a correspondence between the layers of T and generations. At a particular point, say q_j, the registers belonging to generation t are those of age y such that $x(s^u(j,t-1)) < y \leq x(s^u(j,t))$.

For each $(j,t) \in S^u$ either query q_j reads from late memory, i.e., it performs at least $\lceil \log n \rceil$ read operations in which case we can associate a read to each (j,t') for all $t' \leq h < \lceil \log n \rceil$ layers. Otherwise with a constant probability the query q_j performs a read of a register belonging to generation t. Let e_j denote the labeling when executing q_j and let $x = x(s^u(j,t-1))$, i.e., $e_j \in Z(\tilde{u}, x, W_t)$. Since $(j,t) \in S^u$ we get

$$x \leq 4t_{\mathrm{u}} \log n|W_{t-1}| \leq \tfrac{1}{2}|W_t|(\log nC)^{-1}.$$

By Lemma 7 we know that the labeling e_j of nodes at layer $t+1$ and above are independent of contents of registers belonging to generation $t+1$ or older. By Lemma 5 the probability that $|Z(\tilde{u}, x, W_t)| \geq |F^{W_t}|/2^{Cx+1}$ is larger than a half. Arguing as in proof of Theorem 3 this implies for random v at layer t:

$$\exists e \in Z(\tilde{u}, x, W_t) : e(v) \neq e_j(v)$$

with probability $\frac{1}{4}$. As before the probability of a 'yes' answer for query q_j is less than $\frac{1}{8}$ for sufficiently large n even if we restrict us to j occurring as first component of pairs in S^u. Hence arguing as before, if q_j does not read outside early memory, then q_j reads a register of generation t with probability at least $\frac{1}{8}$ (i.e., in order to distinguish e and e_j which both can occur as a result of operation sequences from $\mathcal{U}$ ending with memory configurations which agrees on all registers outside generation t). Hence the expected number of read operations performed during execution of u is at least $\frac{1}{8}|S^u| \in \Omega(mh) = \Omega(m \log n / \log(t_{\mathrm{u}} b \log n))$ as desired. ∎

4. Algorithms for Static Trees

We will show the following theorems.

Theorem 5. *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.* $\mathrm{O}(\log \log n)$ *for* mark *and* unmark *and* $\mathrm{O}(\log n / \log \log n)$ *for* firstmarked. *Furthermore, the first k marked ancestors of any node can be found in worst case time* $\mathrm{O}(\log n / \log \log n + k)$.

Theorem 6. *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.* $\mathrm{O}(1)$ *for* mark, $\mathrm{O}(\log \log n)$ *for* unmark *and* $\mathrm{O}(\log n / \log \log n)$ *for* exists.

In order to achieve these results we will present a new micro/macro-division of trees in section 4.1. Next in section 4.2 we show how to handle micro trees and finally in section 4.3 we proof the above theorems.

4.1. ART-universe: A micro-macro division of a tree.

A division of T into a micro/macro universe consists of a partition of the nodes into micro trees, such that each micro tree is a connected subtree of the original tree. The macro tree is the tree induced by the root nodes of the micro trees. Hence, in the macro tree, two nodes are incident iff there is path in the tree T between the two nodes that does not contain other micro tree root nodes. We will only use the macro tree to simplify the presentations of the algorithms. A node with more than one child in a micro tree/tree is defined as a *heavy* node in the micro tree/tree.

Definition 8. *An ART-universe of a tree has the following defining properties:*

- *Each micro tree has at most* $\mathrm{O}(\log n)$ *heavy nodes.*

- *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.*
- *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 and that a node in the tree T that has more than one child is only a heavy node in the micro tree if more than one child of the node is in the same micro tree.

Lemma 9. *An ART-universe exists and can be constructed in linear time.*

Proof. An ART-universe can be constructed recursively in $O(\log n / \log\log n)$ steps as follows. For any node v, let $w(v)$ be the number of heavy descendants to v in T. For any node v, where $w(v) < \log n$ and $w(\text{parent}(v)) \geq \log n$, we say v is the root in a micro tree and the nodes in the micro tree is v and its descendents. Let T' be the tree T from which all the micro trees have been removed, hence T' is the tree induced by the nodes which are not removed. Now we apply the same process to the tree T', removing micro trees with less than $\log n$ heavy nodes, and keep on recursively until the root of T is included in a micro tree. The level of the micro tree then equals the step in which it was constructed in the above algorithm. Since the algorithm trivially is implemented in linear time, we only need to show that the result is an ART-universe. That is we show that the maximum level of a micro tree is $O(\log n / \log\log n)$, since the remaining properties of an ART-universe follow explicitly by construction. Let T and T' be as above. Each leaf in T' must be a node in T with more than $\log n$ heavy descendants. Thus, T' has at most $O(n/\log n)$ heavy nodes, implying that the remaining tree in the ith iteration has at most $n/(\log n)^i$ heavy nodes, as a consequence, after $O(\log n / \log\log n)$ recursions, the root of T will be included in a micro tree. ∎

Lemma 10. *In a tree T we can support the operations* mark *and* unmark*, in a time identical to the time we can support the same operations in a micro tree. Furthermore, if a query in a micro tree takes time* $O(t_1)$ *it takes* $O(t_1 + t_2(\log n / \log\log n))$ *in the tree T, where* $O(t_2)$ *is the complexity to answer* exists *in a micro tree.*

Proof. Let the tree be divided into an ART-universe. Each of the micro trees is maintained separately, thus updating a node in the tree

only affects the structure associated with the micro tree the node belongs to. Hence, the complexity for updating a node is identical to the cost of updating the structure associated to the micro tree. Now a query for a node in the tree can be done as follows. First we examine if the node has a marked ancestor in the micro tree it belongs to, this takes $O(t_2)$ time. If this is not the case we jump to its first ancestor in the micro tree on the next level and proceed in the same way from this ancestor until we get to the root of the tree or to a micro tree with a marked ancestor to the node. In total we visit at most $O(\log n/\log\log)$ micro trees (the maximum level of a micro tree) where we in each of these trees use $O(t_2)$ time and finally in the last micro tree visit we use $O(t_1)$ time to answer the query, establishing the complexity $O(t_1+t_2(\log n/\log\log n))$ to answer a query in the tree. ∎

4.2. **Algorithms in a micro tree.** In this section we show how to efficiently perform updates and queries in a micro tree, μ. The use of the terms heavy and light node now refer to the number of children a node has in a micro tree μ. Thus, a heavy node in the original tree T can be light in the micro tree μ to which it belongs, if less than two of its children in the original tree T are included in the micro tree μ.

High level description of how to maintain micro trees. Essentially we divide a micro tree into $\log n$ paths and the tree induced by the paths is represented in a single word. To answer a query we use the word to in constant time find the first path including a marked ancestor. Finally, using a search structure on the path we complete the query.

In particular each micro tree is divided into $\log n$ disjoint paths as follows. The paths starts from a heavy node in the micro tree and goes up to but below the first heavy proper ancestor. As a special case, we have that the micro tree root belongs to a path with only one node if the root is heavy. Furthermore the division of the paths does not include the paths which start from a leaf and go up to but do not include the first heavy proper ancestor of the leaf. The essential observation of this division of the paths is:

Observation 11. *Let v be a node in a micro tree whose paths are divided as described above and let P be the nodes on the path from v to the root in the micro tree. Let v belong to a path P_0 and let $P_1 \cdots P_k$ be the remaining paths from the node v up to the root in the micro tree. Then P is identical to $P_1 \cdots P_k$ plus a part of the path P_0.*

Let $P_0 \cdots P_k$ be the disjoint path from a node in a micro tree to the root. In order to in constant time detect the smallest i, $0 < i \leq k$, such that P_i includes a marked node for the node, we use a tree induced on

the disjoint paths in the micro tree to which we apply a method used in [4].

Let MT be the tree induced by the disjoint paths in the micro tree. That is, in MT we have a node for each heavy node, and a parent pointer between two nodes iff the parent is the first heavy proper ancestor to the child in the micro tree. If the heavy node does not have a heavy proper ancestor its parent pointer is nil and the node is the root in MT. Since a micro tree includes at most $\log n$ heavy nodes, the tree MT has size at most $\log n$ and thus it can be represented in a single word as follows. The nodes in MT are numbered from 1 to $\log n$ in a top-down manner. To each node in MT we associate a word, where the ith bit is set to 1 iff the node in MT numbered i is an ancestor to the node. Such an induced tree can clearly be build in linear time (for details see [4]). To the tree MT we also maintain a single word. The ith bit in this word is 1 iff there is a marked node on the ith path.

Lemma 12. *In worst case constant time per update (*mark *and* unmark*) we can maintain a micro tree such that in worst case constant time from any heavy node we can detect on which of the disjoint paths (if any) the first marked ancestor to the heavy node belongs.*

Proof. This is done by using a constant number of simple machine operations (e.g. XOR and AND) to compare the word for tree MT (indicating on which paths there is a marked node) and the word for the heavy node representing which paths there are from the heavy node to the root, see [4] for details. ∎

Thus, we can reduce micro tree problems to path problems as stated in the next lemma. Here, we will regard the paths as rooted, such that the root of the path is the node on the path nearest the root of the tree to which the path belongs.

Lemma 13. *If we for each path know the depth of the marked ancestor nearest the root of the path, we can support the operations* exists*,* firstmarked*,* mark *and* unmark *in a micro tree in the same worst case complexity as on a path.*

Proof. Let v be a node using the paths $P_0 \cdots P_k$. First we detect if the node v has a marked ancestor on the path P_0, by comparing the depth of the marked ancestor on the path P_0 nearest the root with the depth of v. If there is no marked ancestor to the node v on the path P_0 we can in constant time, according to lemma 12, find the smallest i such that P_i includes a marked ancestor, if any such i exist. ∎

Lemma 14. *On a (rooted) path we can support in worst case the following operations:* firstmarked *in* $O(\log\log n)$ *time,* exists *in constant time,* unmark *in* $O(\log\log n)$ *time and* mark *in* $O(\log\log n)$ *time or constant time respectively if* firstmarked *is supported or not. Furthermore, we can in worst case constant time report the marked node on the path with minimum depth.*

Proof. For each path we maintain the depth of the marked node with minimum depth on the path, denoted as $\min(P)$ for the path P. This value is sufficient to answer *exists* queries. However, if we also need to find the first marked node we need a search structure on the path including the depth of the marked nodes on the path. Thus, essentially the only difference between *firstmarked* and *exists* is that we in the first case need a search structure on the path where we in the second case only need a priority queue. As a search structure we can use Van Emde Boas with worst case complexity $O(\log\log n)$ per operation [30, 37], however if searching is not needed, we can perform *mark* (insertion in the structure) in constant time, as shown in in lemma 23 below, which give an efficient black box for insertions in priority queues. ∎

4.3. **Compilation.** *Proof.* **for Theorems 5 and 6** According to Lemma 9 we can in linear time build an ART-universe. Now the theorem is established by using path results from Lemma 14 in Lemma 13 reducing micro tree problems to path problems, and then use this result in Lemma 10 which reduces tree problems to micro tree problems. Finally we sketch (a more detailed proof will be presented in the final version of this paper) how to find the first k ancestor in $O(\log n/\log\log n + k)$ worst case time. Let k' be the number of marked ancestor to a node in a micro tree. We show that in a time $O(min\{k, k'\})$ we can detect if $k > k'$ and if so in the same time find the k' marked ancestor. To each path in a micro tree we associate a list of the marked nodes on the path, this can clearly be done within the same complexity as updating the search structure on the path. To find the marked ancestors to the node we scan the micro tree top-down, that is we examined the disjunct path with marked nodes from the root of the micro tree and down to the node from which we have to find marked ancestors. To find the path with a marked ancestor we use the word associated to the micro tree and to find the marked ancestors on the path we use the list of marked ancestors associated to the path. ∎

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 7. *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 ART-universe (recall definition 8) for a tree that grows under additions of leaves. For a micro tree μ, we let $\text{Heavy}(\mu)$ denote the set of heavy nodes in μ. We will maintain the structure such that each micro tree includes at most $2 \log n$ heavy nodes, thus $|\text{Heavy}(\mu)| \leq 2 \log n$. 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 M 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 r 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 15. *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 a micro trees at level 0 (micro trees at level 0 are 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 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 $\log^{L+1} n$, S, witness nodes. 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 16. *At most* $\mathrm{O}(\log n / \log\log n)$ *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 and maintain a tree MT induced on the paths, which we represent in a single word. Thus, it remains to show how to maintain these structures efficiently. Recall that we to each path P have a search structure (or a priority queue) and maintain the minimum depth of a marked node on the path $\min(P)$. Thus, when we move a path P from one micro tree to another micro tree, each node on the path P should be extracted from the structure of the path it belongs to before the update and inserted in a structure for the path P. Let us first establish the complexity for this part of the update.

Lemma 17. *Moving nodes from one path to another has total complexity* $\mathrm{O}(n \log n)$.

Proof. Each of the n nodes is according to observation 16 moved at most $O(\log n / \log\log n)$ times from one path to another, each time at a cost of $O(\log\log n)$ for updating a Van Emde Boas structure. ∎

Beside constructing of the new path moved from one micro tree to another we also have to update the structures associated to the micro trees. Here, we divide the description into two parts: to add a path to a micro tree and to remove a path from a micro tree. Adding a path P to a micro tree can create a new heavy node on a path Q in the micro tree and thereby force us to divide the path Q into two paths and reorganise the tree MT. The tree MT can be rebuild in time $O(\log n)$, however we cannot afford to directly split the search structure for the path Q. Instead we will with each path maintain the set of heavy nodes where the path Q has been split. Let Q be a path split by the set of heavy nodes $\text{Heavy}(Q)$. A $mark(v)$ operation, for a node v on path Q, is now performed as follows. We insert the node in the search structure associated to Q and if v depth is less than $\min(Q)$ we update this value. By using the set $\text{Heavy}(Q)$ we can in a time no greater than updating the search structure calculate to which part Q_i of the path Q the node v belongs and updating the bit for Q_i in the word representing the tree MT. The operation *unmark* is performed similarly. For a query in a micro tree from a node v on a path P, we can still use $\min(P)$ to detect if there on P is a marked ancestor, if this is not the case we use the tree MT to find the first path with a marked ancestor and continue from here in the same fashion.

Lemma 18. *Adding a new path to a micro tree has complexity* $O(\log n)$.

Proof. It follows from the discussion above that we have to rebuild the tree MT, which has complexity $O(\log n)$, and perform a constant number of operations in a search structure of cost $O(\log\log n)$. ∎

Lemma 19. *The dividing of a micro tree into several micro trees when removing a path has complexity* $O(\log n)$.

Proof. When we remove a path the nodes on the path can have been a part of several paths in the micro tree and the micro tree can be split into several micro trees. For each of these new micro tree we have to create new MT trees, however MT trees can be build in a time linear to the size, and the total size is limited by the number of heavy nodes in the divided micro tree. Hence the complexity is $O(\log n)$. ∎

Lemma 20. *We can achieve the results expressed in theorem 7 with the exception that adding a leaf has complexity* $O(\log n)$.

Proof. The complexity for *mark*, *unmark*, and queries follows immediately from the discussion above since we maintain an ART-universe. The complexity for adding leaves can be divided into 1) constructing new paths which are removed from one micro tree and inserted in another micro tree, and 2) dividing/constructing micro trees when a path is moved. From lemma 17 we know that the total cost of constructing new paths is $O(n \log n)$. Each time we move a path upwards we have to divide and construct micro trees with additional cost of $O(\log n)$ according to lemma 18 and lemma 19. However, since at least $\log n$ nodes have been added to a micro tree before it is destroyed and at most $n \log n / \log \log n$ times a node is added to a micro tree, this does not exceed the amortised complexity. ∎

Next we show how to use lemma 20 as a tool to achieve the results stated in theorem 7. In order to use lemma 20 as a black box for further computation we will have to extend the lemma with an extra operation, $AddParent(v, w)$ for nodes v, w. Let v have the parent p in the tree, then $AddParent(v, w)$ inserts w into the tree such that w becomes a child of p instead of v whose new parent becomes w. Since the operation only extends an existing path it has no influence on the maximum level of micro tree. The only difficult task is to assign the node w a number between the number associated to v and p in order to search on the path. However, this task we assign to the user of the algorithm and as we will see it is easy in our context. A new node inserted with the operation *AddParent* can, as other nodes, be moved to new micro trees $O(\log n / \log \log n)$ times and each time with complexity $O(\log \log n)$, thus it has amortised complexity $O(\log n)$. We conclude:

Lemma 21. *We can achieve the results as expressed in lemma 20 and furthermore within the same complexity we can support a number of* AddParent *operations which does not exceed the number of* add leaf *operations.*

With the above lemma we can now achieve theorem 7. Essentially we will do as follows. We divide the nodes from the tree T into two levels such that a node in level 1 and all its ancestors belong to level 1. Hence, the nodes in a connected subtree with root equal the root of T is at level 1 and the remaining nodes are at level 2. At level 2 we collect paths of length $O(\log n)$ and insert these paths, at most $n / \log n$, into level 1. In level 1 we regard the paths of length $O(\log n)$

as pathnodes and will maintain level 1 using lemma 21. Since at most $n/\log n$ pathnodes are inserted in level 1, each taking amortised time $O(\log n)$, we achieve amortized constant time complexity in total. In order to implement this idea we have to show how to collect nodes into paths of size $O(\log n)$ and how to use lemma 21 on pathnodes. First we show how to collect nodes into paths at level 2.

5.2. **To collect paths.**

Lemma 22. *We can achieve the results as expressed in theorem 7 for a forest of trees, where we are allowed $n/\log n$ times to divide a tree into smaller trees by removing a path of size $O(\log n)$ from the root of a tree to another node in the tree.*

A very simple algorithm could be established as follows. When we add a new leaf we simply add it and a query is answered by following the path from the node to its root. If this path has size $> \log n$ it is removed. The problems with this simple algorithm is that a query now takes $O(\log n)$ amortised time and worst case $O(n)$ time.

Proof of lemma 22. First we show how to improve the worst case complexity to $O(\log n)$ for queries of the above simple algorithm and next how to reduce to worst case $O(\log n/\log\log n)$ time for a query. The forest is divided into two levels, A and B (in the same way as we divided the tree into level 1 and 2). The nodes in level A are above the nodes in level B. A node in level B with a parent in level A is denoted as a level B root node. We will maintain the invariant that the length of a path from a node in level A to its root node in level A is at most $\log n$. When a new node is inserted in level B we associate to the node a pointer to its level B root and increment the total number of decedent to this root. If this number becomes $\log n$ we proceed as follows. The path from the new node's first ancestor in level A and up to the root in level A is removed and the tree in level B to which the new node belongs is moved to level A. Since we only remove a path each time a new tree of size $\log n$ has been constructed in level B, at most $n/\log n$ paths is removed. When a tree is moved from level B to level A its size is at most $\log n$ and the root of the tree becomes also a root in level A, establishing the worst case complexity for a query, since the maximum height is $O(\log n)$. In order to reduce the complexity for a query to $O(\log n/\log\log n)$ we do as follows. When a node is added to the tree, it is first added to a small tree, SmallTree, of size at most $\log n$ or becomes the root in a new SmallTree if its parent belongs to a SmallTree of maximal size. Since the SmallTrees have size $\log n$ they can be represented in a word meaning we can perform all

the operation, *mark*, *unmark*, etc. in constant time. Now essentially we let the macro universe induced by the SmallTrees be maintained by the above worst case $O(\log n)$ algorithm. More specifically, let the macro tree be the tree induced by the SmallTrees. The nodes in the macro tree are divided into two levels, A and B, as above. Now, as above, we limit the size of a tree in level B, but this time we set the limit of the size to $\log n/\log\log n$. Again we remove paths when a tree in level B gets larger than the limit $\log n/\log\log n$. This implies that we remove at most $n/(\log^2 n/\log\log n)$ paths (which should be divided into paths of size $\log n$) and a query now is performed by examining at most $O(\log n/\log\log n)$ SmallTrees in which we perform each operation in constant time. ∎

5.3. **Compilation.** Now we show how to combine lemma 21 with lemma 22 to get theorem 7.

Proof. **theorem 7** The nodes in the tree are divided into two levels, A above B. Level A is maintained by lemma 21 and level B is maintained by lemma 22. When a leaf is added to the tree it is inserted in level B. The algorithm, see lemma 22, to maintain level B pushs up to $n/\log n$ times a path of length $\log n$ up to level A. Each such path, P, will be represented as a single node, $PathNode(P)$, in level A, which is maintained by the algorithm giving lemma 21. This is done as follows. Iff a node in a path P is marked, $PathNode(P)$ is also marked. Furthermore the structure is maintained such that if a node v on a path P_v has a parent w on another path P_w in the same micro tree then w is the node on the path P_w with largest depth in the tree. This implies that the *firstmarked* ancestor to a node v on a path P_v either can be found on the path P_v or on the path which is the *firstmarked* ancestor to $PathNode(P_v)$. Given the $PathNode$, with the first marked ancestor, we can easily in constant time detect the first marked ancestor on the associated path, by letting the marked nodes on a path be represented in a word. Thus, it only remains to show that we can maintain the structure in level A such that if a node v on a path P_v has a parent w on another path P_w in the same micro tree then w is the node on the path with largest depth. Let $MovePath$ be a path which we move from one micro tree to another in the next level. $MovePath$ consist of $PathNodes$ which all, per induction, furfill our condition, thus the only problem is the $PathNode$ Q on the path $MovePath$ nearest the root in the tree which parent is a $PathNode$ P in the new micro tree the path $MovePath$ is moved to. Therefore we split the path P (and the nodes on the path) into two PathNodes using $AddParent$ in order to fullfill

the condition. The split is done in $O(\log n)$ time which does not exceed the complexity for dividing/constructing micro trees. Since, each path moved from level B to level A can result in at most one *AddParent* operation, the number of *AddParent* operations do not exceed the number of *add leaf*'s operations as required in lemma 21. Furthermore, we can easily assign a number to each of the $PathNodes$, as also required by lemma 21, by simply letting the number for a $PathNode$ be the greatest depth in the tree for any of the nodes on the path. To find the first k ancestor efficiently we use the same methode as described in the proof of theorem 5. ∎

6. Aggregate Queries

6.1. **Dyck Languages.** Let S be a string of letters from 1 to n. Each element in S is either a (or). We consider the following two operation $reverse(i)$ which change letter i in the string, either from (to) or vice versa. The query operation is $balanced(i)$ which return Yes iff the first i letters is a balanced string. See [19] for previous work on this problem, including an algorithm with running time $O(\log n)$.

The ith letter in the string will be denoted as $s(i)$. We will show the following theorem.

Theorem 8. *Given linear time for preprocessing and linear space, we can update in* $O(\log n/\log\log n)$ *time and answer a query in* $O(\log n/\log\log n)$ *time. Specially, balanced*(n) *can be answered in constant time. The complexities are all worst case.*

From now we will represent (by a 1 and) by a -1. Let $sum(i,j) = \sum_{k\in i\cdots j} s(k)$ and $minsum(i,j) = \min_{k\in i\cdots j} sum(i,k)$. The answer to the query $balanced(i)$ is now Yes iff $sum(1,i) = minsum(1,i) = 0$.

Let T be a balanced tree with branch $B = (\log n/(\log\log n)^c)$, where c is a small constant > 1. The leaves in the tree are in two levels and numbered $1..n$ from left to right. The leaf numbered i have the value $s(i)$. For each node v in the tree let $left(v)$ and $right(v)$ respectively be the number of the left and right most leaf decedent to the node. To each internal node will we maintain to values: $nodesum(v) = sum(left(v), right(v))$ and $nodeminsum(v) = minsum(left(v), right(v))$. When a leaf changes value we can easily update *nodesum* in time equal to the height of the tree. The difficult part is to update *nodeminsum*. Let the children of a node v be numbered $v_1 \cdots v_k$, for the node v we will for each child maintain the value $childminsum(v_i) = minsum(v_i) + \sum_{v_j\in v_1\cdots(v_i-1)} nodesum(v_j)$, hence $nodeminsum(v) = \min_{v_i\in v_1\cdots v_k} childminsum(v_i)$. This can be done in

constant time per update by tabulation iff the branch B times the number of bits to represent childminsum is no larger than $\mathrm{O}(\log n)$. In order to achieve this we do as follows. Let $X = \min_{v_i \in v_1 \cdots v_k} childminsum(v_i)$ and let $Xchildminsum(v_i) = childminsum(v_i) - X$. If this value is too large to be represented by $\log\log n$ bits we let it have the maximum value which can be represented by $\log\log n$ bit, in order to mark the value as not useful. Now $Xchildminsum$ can be updated in constant time for at least $\log n$ rounds and $childminsum$ can be computed in constant time as $Xchildminsum - X$. After $\log n$ rounds we update the tables. Here, the update complexity is not worst case, but it should be easy to make it worst case. After half of the rounds we start a rebuilding process for the nodes.

To answer a $balanced(1)$ query, we simply test the values in the root of the tree. This can be done in constant time. To answer a $balanced(i)$ query we have to traverse the tree from the root to the ith leaf a make some prefix sum computation which requires more structure, however this will not increase the update time. Furthermore note that tabulation not is necessary, see Dietz [16].

6.2. **Tree generalisation of Dietz' results and a simple ancestor algorithm.** In [16] Dietz showed the following theorem:

Theorem 9. *Prefix sum can be computed in worst case* $\mathrm{O}(\log n/\log\log n)$ *time per operation if the update is restricted to integers which can be represented by* $\mathrm{O}(\log\log n)$ *bits*

Here we extend the results to :

Theorem 10. *Let T be a tree with edge and node weights. An update increase the weight of a node or an edge. The update is restricted to integers which can be represented by* $\mathrm{O}(\log\log n)$ *bits. A query consist of two nodes and return the sum of the edge and node weight on the path between the nodes. We show how each operation can be done in worst case* $\mathrm{O}(\log n/\log\log n)$ *time per operation.*

In order to achieve this we will use top-trees as presented in [3]. A cluster in a tree T is a connected subtree of the tree T with at most two boundary nodes. A boundary node in a cluster is a node incident with a node in the tree T which not belongs to the same cluster the node belongs too. A top-tree of a tree T, is an abstract binary tree where each node in the top-tree represents a cluster of the original tree T. Each leaf in the top-tree contains an edge from T not contained by other leaves. An each internal top-tree node is a cluster which represents the combination of the two child clusters. A top-tree has height $\mathrm{O}(\log n)$ and can be constructed in linear time. Clearly it follows, we

can construct an abstract tree T' with degree $O(\log n/\log\log n)$ and height $O(\log n/\log\log n)$ where each internal node in the top-tree is a cluster which combine its $O(\log n/\log n\log n)$ children clusters. Now we give a simple $O(\log n/\log\log n)$ worst case per operation algorithm for *mark*, *unmark*, and *firstmarked*. By definition each internal node C in T' is a combination of $\log n/\log\log n$ clusters. Let these children clusters to a cluster be named $children(C)$. The tree induced by the boundary nodes from the clusters $children(C)$ we defined as the micro tree fro the cluster C. The micro tree have size $O(\log n/\log\log n)$ and can be represented in a word. When a node is marked we update the micro tree for all the clusters including the node, where the node not is a boundary node. In this way each query/update requires a constant number of operation on $O(\log n/\log\log n)$ micro trees (see [3] for details), which each can be performed in constant time. Furthermore we can in amortized $O(\log n/\log\log n)$ time add a leaf to the tree, by collecting small trees of size $O(\log n/\log\log n)$, before updating the tree T'. This give the same results as presented in [7]. In order to achieve theorem 10 we combine the representation of T' with the results of Dietz [16], which is possible since the size of a micro tree is $O(\log n/\log\log n)$. In the final version of this paper we will give more details.

References

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

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

[3] S. Alstrup, J. Holm, K. De Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th Int. Colloq. on Automata, Languages and Programming (ICALP)*, volume 1256 of *Lecture Notes in Computer Science*, 1997.

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

[5] A. Amir and M. Farach. Adaptive dictionary matching. In IEEE, editor, *Proc. 32nd Symp. Found. of Comp. Sc. (FOCS)*, pages 760–766, San Juan, Porto Rico, October 1991. IEEE Computer Society Press.

[6] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *Journal of Computer and Systems Sciences*, 49(2):208–222, October 1994.

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

[8] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. In *Proc. 3rd Combin. Pattern Matching (CPM)*, volume 644 of *Lecture Notes in Computer Science*, pages 262–275. Springer, 1992.

[9] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. In *Proc. 1st Symp. on Discrete Alg. (SODA)*, pages 344–357. SIAM, 1990.

[10] Paul Beame and Faith Fich. On searching sorted lists: A near-optimal lower bound. Manuscript, 1997.

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

[12] G.S. Brodal. Worst-case efficient priority queues. In *Proc. 7th Symp. on Discrete Alg. (SODA)*, pages 52–58, 1996.

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

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

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

[16] P.F. 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.

[17] 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.

[18] Gudmund S. Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. In *Proc. 34th Symp. Found. of Comp. Sc. (FOCS)*, pages 470–479, 1993.

[19] Gudmund Skovbjerg Frandsen, Thore Husfeldt, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum. Dynamic algorithms for the Dyck languages. In *Proc. 4th WADS*, volume 955 of *Lecture Notes in Computer Science*, pages 98–108, 1995.

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

[21] Michael L. Fredman. Observations on the complexity of generating quasi-Gray codes. *SIAM Journal on Computing*, 7(2):134–146, 1978.

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

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

[24] Michael L. Fredman, Janos Komlós, and Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31:538–544, July 1984.

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

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

[27] Thore Husfeldt and Theis 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.

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

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

[30] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(N)$ space. *Journal of the ACM*, 35(4):183–189, 1990.

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

[32] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. In *Proc. 27th Symp. Theory of Computing (STOC)*, pages 103–111. ACM, 1995.

[33] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130:203–236, 1994.

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

[35] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[36] 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.

[37] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

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

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

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

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

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

Appendix

Fast Insertions in a Priority Queue. Above we need the operations $\min(P)$, *mark*, and *unmark* on a path. These operations are identical to the operations $Insert$, $Delete$ and $Findmin$ in a priority queue. By standard methods the operation $Findmin$ is supported in worst case constant time, whereas the insertion operations only have been supported in worst case constant time by manipulations of specific priority queues, see [12] for a survey. In this section we give a general

black box for priority queues to achieve worst case constant time for insertion. We show the following lemma.

Lemma 23. *Given a priority queue H with worst case complexity $O(f)$ per operation for Insertion, Delete and Findmin, we can construct a priority queue H' with worst case complexity $O(1)$ for Insertion and Findmin and worst case complexity $O(f)$ for Delete.*

Proof. The heap H' consist of a heap H and a bucket of size $O(f)$. When an element is inserted in H' we insert it in the bucket in constant time. When we have inserted $O(f)$ elements, we empty the bucket and insert the minimum element from the bucket in the heap H. The remaining elements from the bucket becomes a tail to the inserted element. If the inserted element is *Deleted* from H the elements in its tail is reinserted in the bucket. Since only $1/f$ of *Insertions* lead to a insertion in the heap H we have achieved amortized complexity $O(1)$ for insertion. In order to make the complexity worst case, we just make use of simple rebuilding methods, thus an element is inserted in the heap H over a sequence of f insertion operation. If a deletion operation should be executed before such a sequence we have time to finish the insertion before extracting the deleted element. ∎

Note : For decrease keys randomly performed on the elements this structure also give complexity $O(1)$ for decrease key. We work with simple extensions to the above structure to achieve this result for any distributions of decrease key, but have not been able to prove or disprove the value of this method for decrease key.

Proof of Prop. 4. In [17] they essential give an algorithm which also use *mark* and *unmark* and extend it to several colours as follows. To each colour is associated a data structure and constant look up time [24, 35] is used to find the structure associated with a specific colour. In essential we use the same method for the algorithm presented in section 4 for *mark*, *unmark* and *firstmarked*. To each structure S from that algorithm we now have a set of colours, where we to each colour associate a structure S. That is for a path we associate a set of colours. To each colour in the set we associate a search structure. Now a search for a given colour a path can be processed as follows. First we use the colour set to in constant time find the structure associated to the colour and next search for the specific colour on the path. Since all necessary set operations can be done in constant time, the result follows from theorem 5.

Recent BRICS Report Series Publications

RS-98-7 **Stephen Alstrup, Thore Husfeldt, and Theis Rauhe.** ***Marked Ancestor Problems (Preliminary Version)*****. April 1998.**

RS-98-6 **Kim Sunesen.** ***Further Results on Partial Order Equivalences on Infinite Systems*****. March 1998. 48 pp.**

RS-98-5 **Olivier Danvy.** ***Formatting Strings in ML*****. March 1998. 3 pp. This report is superseded by the later report BRICS RS-98-12.**

RS-98-4 **Mogens Nielsen and Thomas S. Hune.** ***Deciding Timed Bisimulation through Open Maps*****. February 1998.**

RS-98-3 **Christian N. S. Pedersen, Rune B. Lyngsø, and Jotun Hein.** ***Comparison of Coding DNA*****. January 1998. 20 pp. To appear in** ***Combinatorial Pattern Matching: 9th Annual Symposium*****, CPM '98 Proceedings, LNCS, 1998.**

RS-98-2 **Olivier Danvy.** ***An Extensional Characterization of Lambda-Lifting and Lambda-Dropping*****. January 1998.**

RS-98-1 **Olivier Danvy.** ***A Simple Solution to Type Specialization (Extended Abstract)*****. January 1998. 7 pp.**

RS-97-53 **Olivier Danvy.** ***Online Type-Directed Partial Evaluation*****. December 1997. 31 pp. Extended version of an article to appear in** ***Third Fuji International Symposium on Functional and Logic Programming*****, FLOPS '98 Proceedings (Kyoto, Japan, April 2–4, 1998), pages 271–295, World Scientific, 1998.**

RS-97-52 **Paola Quaglia.** ***On the Finitary Characterization of π-Congruences*****. December 1997. 59 pp.**

RS-97-51 **James McKinna and Robert Pollack.** ***Some Lambda Calculus and Type Theory Formalized*****. December 1997. 43 pp.**

RS-97-50 **Ivan B. Damgård and Birgit Pfitzmann.** ***Sequential Iteration of Interactive Arguments and an Efficient Zero-Knowledge Argument for NP*****. December 1997. 19 pp. To appear in** ***25th International Colloquium on Automata, Languages, and Programming*****, ICALP '98 Proceedings, LNCS, 1998.**