Dynamic LCA Queries on Trees*

Richard Cole† Ramesh Hariharan‡

Abstract

We show how to maintain a data structure on trees which allows for the following operations, all in worst-case constant time.

1. Insertion of leaves and internal nodes.
2. Deletion of leaves.
3. Deletion of internal nodes with only one child.
4. Determining the Least Common Ancestor of any two nodes.

We also generalize the Dietz-Sleator "cup-filling" scheduling methodology, which may be of independent interest.

1 Introduction

Finding *Least Common Ancestors (LCAs)* in trees is a fundamental problem that arises in a number of applications. For example, it arises in computing maximum weight matchings in graphs [Ga90], in computing longest common extensions of strings, finding maximal palindromes in strings, matching patterns with k mismatches, and finding k-mismatch tandem repeats [Gus97]. The tree involved in all but the first of these applications is a *Suffix Tree*.

The primary use of LCA computations in a suffix tree is to determine the longest common prefix of two substrings in constant time. This operation is used heavily in the above applications. The suffix tree for a given string can be constructed in linear time [M76]. Each node in this tree corresponds to a substring of the given string. The longest common prefix of any two substrings is the string corresponding to the least common ancestor of the corresponding nodes.

The first constant time LCA computation algorithm was due to Harel and Tarjan [HT84]. This algorithm preprocesses a tree in linear time and subsequently answers LCA queries in constant time. Subsequently, Schieber and Vishkin [SV88], Berkman and Vishkin [BV94], and Bender and Farach-Colton [BF00], gave simpler algorithms with the same performance.

In this paper, we consider the dynamic version of the problem, i.e., maintaining a data structure which supports the following tree operations: insertion of leaves and internal nodes, deletion of internal nodes with only one child, and LCA queries. We assume that when a new node is inserted, a pointer to the insertion site in the tree is also given. The motivation is to maintain a suffix tree under insertion of new strings, deletion of strings, and longest common prefix queries. One application of this problem arises in maintaining a database

*This work was supported by NSF grants CCR9503309, CCR9800085, and CCR0105678.

†Courant Institute, New York University, cole@cs.nyu.edu.

‡Indian Institute of Science, Bangalore, ramesh@csa.iisc.ernet.in.

of strings in order to answer queries of the following kind: given a pattern string, find all its occurrences with up to k mismatches in the strings in the database. Efficient algorithms for finding all occurrences of a pattern in a text with up to k mismatches [LV86, CH97] require maintaining the suffix tree of the text and processing it for LCA queries. Extending these algorithms to maintain a database of strings supporting k-mismatch queries would require maintaining LCAs dynamically[1]. Additions and deletions of new strings to the database will change the suffix tree as well as the LCA data structure. A new string can be inserted into a suffix tree in time proportional to its length. The number of nodes inserted in the process is proportional to the length of the string inserted. These nodes could be leaves or internal nodes. Similarly, deletion of a string will cause the removal of some nodes. Our goal is to minimize the work needed to maintain the data structure for each node inserted or deleted.

Harel and Tarjan [HT84] gave an algorithm to maintain a forest under *linking* and finding LCAs. This is useful in computing maximum weight matchings in graphs. The link operation generalizes insertions of new leaves. Harel and Tarjan's link operation allowed only linking of whole trees, not linking of a tree to a *subtree* of another tree. The amortized time taken by their link operation was $\alpha(m,n)$, where n is the size of the tree and m is the number of operations. LCA queries were answered in constant time. Gabow [Ga90] gave an algorithm which performs additions and deletions of leaves in constant amortized time. It also supported linking of trees to subtrees in $\alpha(m,n)$ amortized time. The worst-case time for update operations in both these algorithms was $\Omega(n)$. The worst-case time for an LCA query was $O(1)$. Both the above algorithms were motivated by the maximum weighted matching problem in graphs.

Since our focus is different, namely suffix trees, we consider insertions and deletions of leaves and internal nodes, but not the link operation. Note that neither of the above algorithms considered insertions of internal nodes. Westbrook [We92] built upon Gabow's approach above to give an $O(1)$ amortized time algorithm which could perform insertions and deletions of leaves as well as internal nodes. Our focus, however, is on worst-case insertion time rather than amortized time.

We give an algorithm which performs insertions and deletions of leaves and internal nodes while supporting LCA queries, all in *constant worst-case* time. This algorithm is obtained in two stages. First, we give an algorithm which takes $O(\log^3 n)$ worst-case time for insertions and deletions and $O(1)$ worst-case time for queries. This is the core of our algorithm. Subsequently, we show how to improve the worst-case time for insertions and deletions to $O(1)$ by using a standard multi-level scheme. The space taken by our algorithm is $O(n)$.

Our basic $O(\log^3 n)$ worst-case time algorithm broadly follows Gabow's and Schieber and Vishkin's algorithm. The overall approach is to decompose the tree into centroid paths, and assign a code to each node. From the codes for two given nodes, the centroid path at which their paths from the root separate can be easily determined in constant time. And given two vertices on the same centroid path, the one closer to the root can be determined by a simple numbering. Together, these suffice to find the LCA. The basic problem we have to solve is to maintain the centroid paths and codes over insertions and deletions. Gabow's algorithm does this in bursts, reorganizing whole subtrees when they grow too large. This makes the worst-case time large. However, the amortized time is $O(\log n)$ because each reorganization is coupled with a doubling in size; this time is reduced to $O(1)$ using a multi-level scheme. Note, also, that Gabow does not consider insertions of internal nodes. Thus, two main issues need to be tackled to get constant worst-case time.

[1]However, just maintaining LCAs alone is not sufficient to solve the dynamic k mismatches problem with query time smaller than the obvious static algorithm. Therefore, we do not obtain any results on the dynamic k mismatches problem here.

The first issue is that of maintaining numbers on centroid paths so that the LCA of two given nodes on the same centroid path can be found in constant time. For this purpose, we use the Dietz-Sleator [DS87] data structure (or the Tsakalidis [Ts84] data structure) which maintains order in a list under insertions and deletions.[2]

The second and the more serious issue by far is that of reorganizing trees to maintain centroid paths and codes in constant worst-case time. Since we seek constant worst-case time, there is little option but to delay this reorganization. We amortize this reorganization over future insertions and deletions, i.e., spread the $O(1)$ amortized work of Gabow's algorithm over future insertions/deletions so each insertion and deletion does only a constant amount of work. This approach is not new and has been used by Dietz and Sleator [DS87] and Willard [W82] among others. However, the problems caused by this approach are non-trivial and specific to this setting.

The problem with this approach is that any change at a node v causes the codes at all the nodes in the subtree rooted at v to change. Since updates of these codes are spread over future insertions and deletions, queries at any given instant will find a mixture of updated and not yet updated codes. This could potentially give wrong answers. We avoid this with a two phase update of a two copy code.

What further complicates the situation is that reorganizations could be taking place at many subtrees simultaneously, one nested inside the other. This implies that the variation amongst nodes in the degree to which their codes have been updated at any given instant could be arbitrarily large. Nonetheless, the two phase update ensures correct answers to the queries.

An additional complication is that the various nested reorganizations could proceed at very different speeds, depending upon the distribution of the inserted nodes. In this respect, the situation is analogous to that encountered in asynchronous distributed computing, where interacting processes proceeding at arbitrarily different speeds need to ensure they collectively make progress on their shared computation.

Our main contribution is to organize the various nested processes so that they complete in time and also maintain codes which give correct answers to queries. This is obtained by a non-trivial scheduling procedure coupled with an analysis which bounds the total sizes of nested processes.

To understand our scheduling procedure it is helpful to recall the Dietz-Sleator cup-filling methodology. It concerns a collection of tasks in which priorities increase in an unknown but bounded way (i.e. adversarially set) each time unit; the scheduling is very simple: simply select the current highest priority task and run it to completion. They show this algorithm has a good performance which they tightly bound. We are concerned with a similar scenario, but in which priorities are only approximately known; naturally, we schedule the apparently highest priority task. We also allow the tasks to be somewhat divisible so that they need not be run to completion once started. Details appear in Section 6.8.

2 Overview

We assume that the tree is a binary tree, without loss of generality.

First, we consider only insertions. Deletions are handled easily by just ignoring them until they form a significant fraction of the number of nodes, at which point the entire data structure is rebuilt. The original

[2] We can also use the data structure given here but supporting only leaf insertion and deletion. This results in the centroid paths being modified only at their endpoints, and a trivial numbering scheme suffices to maintain order. This approach was suggested by Farach-Colton [F99].

data structure is also maintained until this rebuilding is complete in order to answer queries. Details on handling deletions are deferred to Section 8.

We also assume that the insertions at most double the tree size. This assumption is also handled easily by rebuilding when the size of the tree increases by some suitable constant factor, and again is addressed in Section 8.

The paper is organized as follows. We give some definitions in Section 3. Then we describe the algorithm for the static case in Section 4 and Gabow's dynamic algorithm in Sections 5. In Section 6, we describe our $O(\log^3 n)$ worst-case time algorithm. Sections 7 and 8 describe the improvement to $O(1)$ time and the handling of deletions, respectively.

3 Definitions

We partition the tree T into paths, called *centroid paths*, as follows. Let T_y denote the subtree of T rooted at node y. Suppose $2^i \leq |T_y| < 2^{i+1}$. Then, y is called a *tail* node if $|T_z| < 2^i$ for all children z of y, if any. Such vertices y will lie in distinct centroid paths and will be the *tails*, i.e., bottommost vertices, in their respective centroid paths. The centroid path containing y connects y to its farthest ancestor x such that $2^i \leq |T_x| < 2^{i+1}$. x is called the *head* of this path. It is easy to see that centroid paths defined as above are disjoint.

A centroid path π is said to be an ancestor of a node x if π contains an ancestor[3] of x. A centroid path π is said to be an ancestor of another path π' if π is an ancestor of the head of π'. A centroid path π is a child of another path π' if the head of π is a child of a node on π'.

The *Least Common Centroid Path (LCCP)* of two nodes is the centroid path containing their LCA. An *off-path node* with respect to a particular centroid path π is a node not on π whose parent is on π. The *Branching Pair (BP)* of two nodes x, y is the pair of nodes x', y' on the LCCP which are the least common ancestors of x, y, respectively.

4 Outline of the Static Algorithm

The nodes of the tree are partitioned into centroid paths. The nodes are then numbered so that parents have smaller numbers than their children. In fact, the numbering need satisfy only the following property: if x and y are distinct vertices on the same centroid path and x is a strict ancestor of y then $number(x) < number(y)$.

Each vertex is given a code of length $O(\log n)$ with the following property: the LCCP and BP of x and y can be determined easily from the first bit in which the codes for x and y differ. Let $code(x)$ denote the code for node x.

The LCA of two nodes x, y is now easy to determine. The LCCP and BP of x, y are found in constant time using a RAM operation for finding the leftmost bit which differs in $code(x)$ and $code(y)$[4]. Note that the nodes in the BP need not be distinct (see Fig.1). The node in the BP with the smaller number is the desired LCA.

---

[3] All references to ancestors in this paper will be in the non-strict sense, unless otherwise stated.

[4] Or perhaps, using table look-up on a precomputed set of answers.

4.1 The Codes

It remains to describe the assignment of codes to nodes. Note that if the tree was a complete binary tree, all centroid paths would be just single nodes. Further, *code*(x) could be the canonical code obtained by labelling the left-going edges 0 and right-going edges 1, and reading off the path labels from the root to x.

For a general tree, *code*(x) is a concatenation of smaller bit strings, one for each centroid path containing an ancestor of x.

First, we assign to each centroid path π, a bit string, called *separator*(π). These strings have the following property. For each centroid path π, the separator strings assigned to children centroid paths of π form a prefix-free set (i.e., no string is a prefix of another string). The length of *separator*(π) is $O\left(\log \frac{|T_x|}{|T_y|}\right)$, where y is the head of π and x is the head of the centroid path containing the parent of y.

$Code(x)$ is a concatenation of the separator strings assigned to ancestor centroid paths of x (including the path containing x) in order of increasing distance from the root. It is easy to show that the length of the code is $O(\log n)$ (take any sequence of centroid paths encountered on a path from the root to a leaf and let $x_1 \ldots x_k$ be the heads of these centroid paths; then the sum $\sum_{i=2}^{k} \log \frac{|T_{x_{i-1}}|}{|T_{x_i}|}$ equals $O(\log |T_{x_1}|) = O(\log n)$).

It will be convenient to have *separator*(π) be of length $a(\lceil \log |T_x| \rceil - \lceil \log |T_y| \rceil)$ for a suitable constant integer $a \geq 1$, if need be by padding *separator*(π) with zeros at the right end. This ensures that the length and position of *separator*(π) in a code is fully determined by $|T_x|$ and $|T_y|$.

Each separator string in *code*(x) is *tagged* with the name of the corresponding centroid path, i.e., given the index of a bit in *code*(x), we can recover the name of the path within whose separator this bit lies, in $O(1)$ time. *number*(x), *separator*(π), *code*(x) and the above tagging can all be computed in $O(n)$ time (we comment briefly on this below).

The LCCP of nodes x and y is determined from *code*(x) and *code*(y) in $O(1)$ time as follows. We find the leftmost bit in which *code*(x) and *code*(y) differ; subsequently, using the above tagging, we find the name of the two paths whose separators contain this mismatch bit in *code*(x) and *code*(y), respectively. The parents of the heads of these two paths will give the BP (see Fig.1) and the path containing this BP is the $LCCP$. To see this, note that the separator strings in both codes corresponding to the LCCP and centroid paths above the LCCP are identical. In addition, due to the above prefix-free property, the separator strings corresponding to the two children paths of the LCCP which are ancestors of x and y, respectively, necessarily differ in some bit.

A special case arises when one or both of x, y are part of the LCCP. If both are part of the LCCP then the one with smaller *number*() is the LCA. Otherwise, if x is part of the LCCP but y is not, then *code*(x) is a prefix of *code*(y). The path containing x is the LCCP; BP is easy to determine as well.

paragraphComputation. We briefly touch upon how *number*(x), *separator*(π), *code*(x) can be computed in $O(n)$ time, and further, how each separator string in *code*(x) can be tagged with the name of the corresponding path.

Computing *number*(x) is clearly easy: in any centroid path the numbers only need to be in increasing order of distance from the root.

Computing *separator*(π) involves assigning prefix-free codes. We outline how this is done for children paths of a centroid path π with head x, given that the separator for π has already been determined. Let $\pi_1 \ldots \pi_k$ denote the children paths of π, and $x_1 \ldots x_k$ their respective heads. We construct a weight-balanced binary search tree on the weights $|T_{x_1}| \ldots |T_{x_k}|$. This tree can be constructed in $O(k)$ time [Meh77] and has

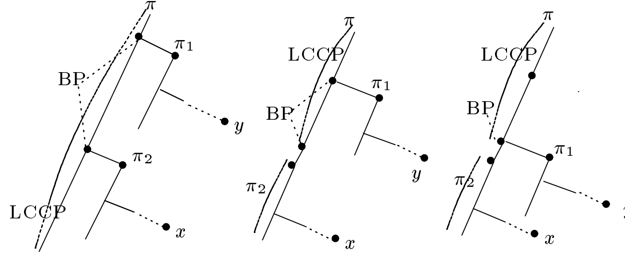

Figure 1: LCCP and BP.

the property that the height of the leaf corresponding to x_i is $O\left(\log \frac{\sum_{j=1}^{k}|T_{x_j}|}{|T_{x_i}|}\right) = O\left(\log \frac{|T_x|}{|T_{x_i}|}\right)$. Separator codes for $\pi_1 \ldots \pi_k$ are obtained by encoding left edges in the weight-balanced tree by 0, right edges by 1, and reading off the labels on the path from the root to the appropriate leaves in this tree. Clearly, codes thus obtained are prefix-free. The whole procedure takes $O(k)$ time, which translates to $O(n)$ time over all of T.

code(x) is computed in $O(1)$ time from *code*(y), where y is the parent of x, as follows. If x and y are in the same centroid path then the codes are the same. Otherwise, x is in a child path π of the path containing y, and *code*(x) is obtained by concatenating *code*(y) and *separator*(π). This is done in $O(1)$ time using a RAM operation.

There is one issue which needs clarification. Recall the tagging mentioned above. One method to find the name of the centroid path whose separator string contains a particular mismatch bit is to keep an array of size $O(\log n)$ for each vertex x; the array for vertex x stores the relevant path name for each potential mismatch bit. Clearly, given the leftmost bit in which *code*(x) differs from *code*(y), indexing into the above arrays (one each for x and y) using the location of the mismatch bit will give us the names of the required separator paths in $O(1)$ time. However, setting this up would require $O(n \log n)$ space and therefore, $O(n \log n)$ time, over all nodes x. Both terms can be reduced to $O(n)$ in one of two ways.

The first involves using a multi-level data structure, similar to the one used by Gabow [Ga90] and the one we use to get $O(1)$ query time for the dynamic case; this is elaborated upon further in Section 7. In this paper, we will assume the framework of this solution.

In the second solution, this tagging is avoided altogether. Instead, centroid paths are named by the code given to their respective heads and the name of the LCCP of two given nodes x and y is easily recovered in $O(1)$ time, given their codes. Indeed, only the following operation needs to be performed to determine the name of the LCCP: given the mismatch bit in *code*(x), return the prefix of *code*(x) comprising separators of all those centroid paths which are ancestors of that centroid path whose separator contains the mismatch bit (and likewise for *code*(y)). This is easily done using look-up tables of $O(n)$ size.

5 The Dynamic Case: Gabow's Amortized Bound

The main problem in the dynamic case is to maintain the centroid paths along with the quantities *number*(x), *separator*(π) and *code*(x).

Gabow [Ga90] gave an algorithm for the case when only leaf insertions were allowed. Maintenance of $number(x)$ is trivial in this case: new leaves are assigned successively larger numbers. However, if insertions of internal nodes is allowed, then it is not clear how to maintain $number(x)$.

Gabow's approach to maintaining centroid paths is as follows. As insertions are made, the centroid paths in the tree will change, in a manner yet to be described. Gabow updates the centroid paths not incrementally but in bursts. Whenever the subtree rooted at the head of a centroid path doubles[5] in size, the entire subtree is reorganized, i.e., reprocessed to construct new centroid paths, separators and codes.

Gabow maintains separators and codes as follows. Instead of prefix-free separators, Gabow maintains a family of nested intervals. The interval for a centroid path is a subinterval of the interval for any ancestor centroid path. In addition, the intervals for the centroid paths which are children of a path π are all disjoint. A constrained version of this approach is equivalent to maintaining separators, as we shall describe shortly in Section 6.2.

When a new off-path node y with respect to a particular centroid path π is inserted, a new interval within the interval for π and to the right of all other intervals for children of π is assigned to y. Gabow shows that there is always sufficient space for this new interval, given that a subtree is reprocessed whenever its size doubles, at which point intervals nested within another are packed together. We follow a similar approach.

The time taken by Gabow's algorithm on any single insertion is proportional to the size of the subtree which is reorganized. Thus the worst-case time for an insertion could be $\Omega(n)$. However, since the reorganization of a subtree is coupled with the doubling in its size, the amortized time for an insertion is $O(\log n)$. Gabow converts this to $O(1)$ amortized time by using a multi-level approach.

6 Our $O(\log^3 n)$ Time Worst-Case Algorithm

As described above, there are two main hurdles to improving Gabow's scheme to run in constant worst-case time, or even poly-logarithmic worst-case time. The first is the maintenance of $number(x)$ when internal nodes are inserted. The second is the reorganization of subtrees.

The first problem is easy to overcome using an algorithm for maintaining order in a list under insertions and deletions in $O(1)$ worst-case time, due to Dietz and Sleator [DS87]. We maintain each centroid path as an ordered list using this algorithm, allowing us to answer queries about which node in a particular branching pair is closer to the root in $O(1)$ worst-case time.

The second problem is more serious. Our basic approach is to distribute the reorganization of a subtree caused by a particular insertion over subsequent insertions. In other words, the various operations involved in reorganizing a subtree are performed, a poly-logarithmic number at a time, over future insertions[6]. This means that queries which come while a subtree is being reorganized will see a partially reorganized subtree, and therefore risk returning wrong answers. We describe our algorithm for the reorganization in further detail next.

[5]When it crosses a power of two boundary, actually.

[6]This general approach has also been followed by Dietz and Sleator [DS87] and by Willard [W82] to convert algorithms with good amortized performance to worst-case performance.

6.1 Weighted Nodes

Our $O(1)$ time algorithm for the LCA problem uses as a subroutine an $O(\log^3 n)$ algorithm for a slightly generalized problem. We indicate the reasons behind the need for a generalization next.

Let T be the tree on which the LCA queries are being performed. Our approach is to select $\Theta(n/\log^3 n)$ nodes of T, which partition T into subtrees of size $O(\log^3 n)$, called *small* subtrees. The selected nodes are formed into an induced tree T_1, to which we apply the $O(\log^3 n)$ time algorithm. It will be the case that for each $\Theta(\log^3 n)$ insertions into one of these small subtrees just $O(1)$ nodes are added to T_1. To achieve the $O(1)$ worst case time bound, we need to perform the $O(\log^3 n)$ operations stemming from an insertion to T_1 over the corresponding $O(\log^3 n)$ insertions to the relevant small subtree of T, at a rate of $O(1)$ operations per insertion. To control this appropriately, we weight the nodes of T_1 as follows.

Weight Constraints:

(i) All weights are integer multiples of $1/\log^3 n$.[7]

(ii) Node weights are in the range $[0,1]$.

(iii) If a node has weight less than 1, its parent has weight 1.

(iv) A weight 1 node has at most one child of weight less than 1.

Weight increases occur in integer multiples of $1/\log^3 n$; the largest possible increase is by $1/\log^2 n$, as we will see in Remark 6.18. We will show that we can maintain T_1 with $O(\log^3 n)$ operations per unit increase in weight. Later we will see that each insertion to T results in at most a $4/\log^3 n$ increase in weight, and we will show that T_1 can be maintained with $O(1)$ work per insertion to T. For intuition, the reader may find it helpful to think of nodes being inserted with weight 1 with the caveat that this is not exactly the scenario we are describing.

When a node is inserted in T_1 it will have weight zero initially. As the relevant inservations to T occur, its weight is increased. Until its weight reaches 1, no further nodes can be inserted in its neighborhood in T_1, so as to meet constraints (iii) and (iv) above.

6.2 Updating Centroid Paths

When a node's weight is increased (by up to $1/\log^2 n$), each of its $O(\log n)$ ancestor centroid paths could shift down by one or two nodes as shown in Fig. 2, Case 1. New centroid paths of one or two nodes could begin as well, as shown in Fig. 2, Case 2. (If all node weights are equal to 1, the shifts are only by one node and the new paths each have only one node.)

We would like to maintain the invariant that for each centroid path there is an integer i such that for each node w on the path, $2^i \le |T_w| < 2^{i+1}$, where $|T_w|$ denotes the weight of the subtree T_w rooted at w. We call such paths *i-paths*. Unfortunately, as it is expensive to update the codes associated with the subtrees of a centroid path, the changes to a centroid path may lag the increases in the sizes of the trees T_w. Consequently, we will allow centroid paths to overlap.

More specifically, an i-path π, as in the static structure, comprises a maximal sequence of nodes w with $2^i \le |T_w| < 2^{i+1}$, but in addition may include further nodes z with $2^{i+1} \le |T_z| < 2^{i+1} + 2^{i-1} + 2^{i-2}$, for

[7] n is restricted to a range $[2^i, 2^{i+a}]$ for some constant a, and we take $\log^3 n$ to be the fixed value $(i+a)^3$.

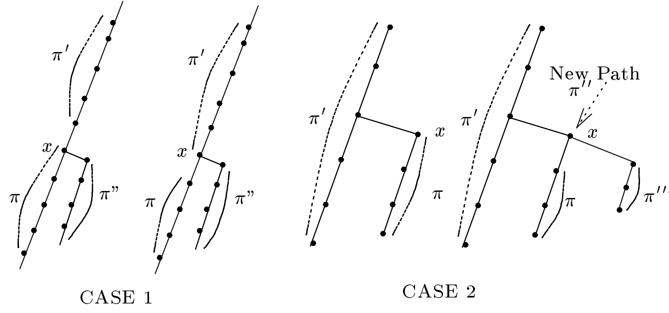

Figure 2: Changing Centroid Paths.

$i \geq 1$. Any such node z is also part of an $(i+1)$-path π'. Naturally, the nodes of π are required to form a path.

Further, to accommodate leaves, which may have weight 0, we define a 0-path to comprise a maximal sequence of nodes w such that $0 \leq |T_w| < 2$, and in addition it may include further nodes z with $2 \leq |T_z| < 2\frac{3}{4}$.

If node v lies on both an i-path π and an $(i+1)$-path π', π is said to be its *primary* path and π' its *secondary* path. If v lies on a single path π, π is its primary path.

We need to redefine the notion of parenthood for these paths.

Definition. Let π be a centroid path with topmost node x, called *head*(π). If x is secondary on path π', then π' is the parent of π. Otherwise, if x is not secondary on any path, the parent of π is given by the primary path of *parent*(x).

Lemma 6.1. *Suppose that u and v are adjacent secondary nodes on $(i+1)$-path π'. Then u and v are primary nodes on the same i-path π, where π' is the parent of π.*

Proof. WLOG let u be the parent of v. $|T_v| \geq 2^{i+1}$ as v lies on π'. Let w be u's other child, if any. Suppose, for a contradiction, that u and v were on different i-paths. Also suppose that u is on i-path π. Then w must have been part of π before $|T_v|$ reached 2^i, as otherwise at that point v would have joined π. For $i \geq 1$, it follows that $|T_w| \geq 2^i$. For $i = 0$, by weight constraint (iv), $weight(w)$ must reach 1 before v is inserted (with initial weight 0) in T_1; thus here too, $|T_w| \geq 2^i$. Thus $|T_u| = wt(u) + |T_w| + |T_v| \geq 2^{i+1} + 2^i$, and thus u cannot be on an i-path, contrary to assumption. □

The increment in weight of a node z may cause changes to the tails of some or all of its ancestral centroid paths. Changes to the heads of the paths may be deferred; their handling is described in subsequent sections. The reason for delaying changes to the head is that such a change entails updating the codes of all the nodes in the off-path subtree of the head node. The following changes may occur to an i-path with tail y and head x.

1. If node z is not a descendant of y then the tail of π is unchanged.

2. If z is a descendant of x and $|T_x|$ increases from less than 2^{i+1} to at least 2^{i+1} due to the weight change, then x is added to an $(i+1)$-path. If the path π'', the parent of π, is an $(i+1)$-path, then x becomes the tail of π''. If not, a new $(i+1)$-path is created comprising the single node x. In any event, x remains on π for now. Note that there may be two nodes x_1 and x_2 for which $|T_{x_h}|$, $h = 1, 2$, increases from less than 2^{i+1} to at least 2^{i+1}.

Remark 6.2. *Clearly, as weights increase in the subtree rooted at the head x of path π, eventually x must be removed from the head of π in order to maintain the invariant that $|T_{head(\pi)}| < 2^{i+1} + 2^{i-1} + 2^{i-2}$. Thus, over time, the path π will migrate down the (growing) tree, but will never disappear.*

Remark 6.3. *Actually, the tail node u of an i-path might have children v and w with $|T_v|$, $|T_w| < 2^{i-1}$, but at least one of $|T_v|$, $|T_w|$ will be of size 2^{i-2} or larger (for to contradict this we would need $2^i \leq |T_u| = wt(u) + |T_v| + |T_w| < 1 + 2 \cdot 2^{i-2}$, i.e. $2^{i-1} < 1$ or $i < 1$, and then u is a leaf). Thus as $(i-1)$-path π' migrates down the tree from such a node u it might disappear; to avoid this we preserve it as a zero length path at the "bottom" of node u and ancestral to both v and w. Later if either $|T_v|$ or $|T_w|$ reaches size 2^{i-1}, then the corresponding node (v or w) joins π'.*

When a node z is inserted it joins a centroid path according to the following rules.

1. If z is inserted between two nodes in π then z is added to π at the appropriate place (possibly, z is added to two paths in this way, once as a primary node and once as a secondary node).

2. If node z is inserted between x, the head of path π, and x's parent z', then z is added to the centroid path or paths to which x belongs. If as a result z is on both an i- and an $(i+1)$-path, it will be the case that $|T_z| < 2^{i+1} + 2^{i-1} + 2^{i-2}$, since $|T_z| = |T_x|$ at this point.

6.3 Updating Separators

Separators and codes have to be updated to reflect the above changes in the centroid paths; the update begins when a node previously on an i-path joins an $(i+1)$-path; when the update completes, the node leaves the i-path. In between times it lies on both paths.

The following interpretation of separators in terms of intervals is helpful.

Separators as Constrained Intervals. Consider an interval of length $m = 2^k$. All subintervals we consider will have lengths which are powers of 2. Each integer point on this interval has an associated k bit code (the leftmost point is coded with the all 0s string, subsequent points are coded by the binary encoding of their distance from the leftmost point; this encoding has length k). We allow a subinterval of length 2^i to begin only at those integer points which have i trailing 0s in their bits (see Fig.3); with such a subinterval, we associate a bit string of length $k-i$ given by the leading $k-i$ bits of the code for the starting point. It can easily be seen that given a set of disjoint subintervals with this property, the bit strings assigned to the subintervals form a prefix-free set. Thus assigning prefix-free separators is identical to assigning subintervals with the above constraints. Henceforth, all our references to intervals will be to intervals with the above constraints.

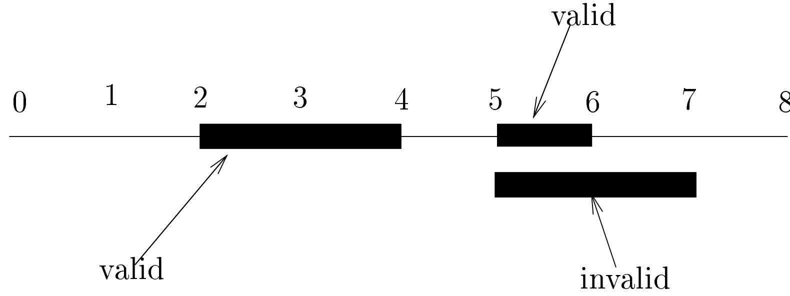

Figure 3: Constrained Intervals

Mapping Paths to Intervals. With each i-path π we maintain an interval int_π of length either 2^{ic} or $2^{ic+c'}$, $c' < c$, where c and c' are constants to be determined. When π is created, int_π has length 2^{ic}. At some point between the time $T_{head(\pi}$ reaches size $2^{i+1} - 2^{i-3}$ and the time it reaches size 2^{i+1}, int_π will gain length $2^{ic+c'}$. There are two goals. The first is to ensure that if the parent of $head(\pi)$ lies on an i-path (not π) as well as an $(i+1)$-path, then int_π has length 2^{ic}. The second is to ensure that if a node originally on π is also secondary on path π' then int_π has length $2^{ic+c'}$. By definition, once $T_{head(\pi)}$ first reaches size $2^i + 2^{i-1} + 2^{i-2}$ the first situation no longer applies. The second situation applies once $T_{head(\pi)}$ reaches size 2^{i+1} (as $head(\pi)$ changes the size of $T_{head(\pi)}$ may subsequently drop). Note that constraints on c and c' will be imposed by Lemma 6.6 below; setting $c' = 5$ and $c = 10$ suffices.

A crucial property of an interval for an i-path π is that the rightmost bit for its separator ends at a fixed location, so that in each code in which it occurs there are a specified number of bits to its right whose values depend only on the separators for h-paths, $h < i$, to which the relevant node belongs. The number of these bits is either ic or $ic + c'$, corresponding to the size of int_π. This allows the code for int_π to be changed without requiring any change to the separators for h-paths, $h < i$, contained in the codes in which int_π occurs. The one exception arises when the size of the interval for π increases. But this can be viewed as simply prefixing c' zeros to the separator following π in the code for each node in $T_{head(\pi)}$.

Updating Intervals. The following updates need to be performed. See Fig. 2.

1. Node x lies on paths π and π'; x is the head of π and is being removed from π; in addition, there is a proper ancestor of x that is or had been on π'. Then the centroid path π'' whose head was the off-path child of x (with respect to π) must be assigned a new interval. The interval for π'' was earlier nested within int_π and $int_{\pi'}$. Now this interval must reassigned so as to be disjoint from int_π but still nested within $int_{\pi'}$. The process which does this is called $Reassign(x)$.

2. Node x is on paths π and π', and x is the head node of π and π'. Then π' is a new centroid path, and a new interval has to be assigned to π'. This interval $int_{\pi'}$ must be nested within the interval $int_{\pi''}$ for the path π'', the previous parent of π, and now the parent of π'. Further, int_π must be reassigned so it is nested within $int_{\pi'}$. This is done by a procedure $Assign(\pi')$. In addition, the interval associated with the path π'' containing the off-path child of x (with respect to π) must be reassigned so that it is

also nested within $int_{\pi'}$. This is done by a procedure $Reassign(x)$. There are a few details associated with this case which will be explained later in Section 6.4.

3. Node x is the head of i-path π, and its parent y lies on class i path $\pi' \neq \pi$ and also necessarily on class $i+1$ path π'' (for by the maximality of π, $|T_y| \geq 2^{i+1}$). Note that by the time $|T_{head(\pi)}| = 2^{i+1}$, y will no longer be on π'. Between this time and before the time $|T_{head(\pi)}|$ reaches $2^{i+1} + 2^{i-1} + 2^{i-2}$, a new larger interval will have been assigned to path π; this new interval will be contained within $int_{\pi''}$. This is done by a process $Rescale(\pi)$.

Thus as time proceeds, subintervals move from one interval to another and new intervals are created. This movement and creation is done as follows. When a subinterval has to be removed from an interval, the subinterval is just marked as deleted but not removed. When a new subinterval has to be assigned to π within $int_{\pi'}$, where π' is the parent path of π, it is assigned in either the first half of $int_{\pi'}$ or the second half of $int_{\pi'}$, based on a logic to be described. In either case, it is assigned to the leftmost available slot to the right of all assigned subintervals in constant time. Note that a particular weight increase creates at most constant number of *Rescale*, *Reassign* or *Assign* processes at each ancestor centroid path of the reweighted node.

We need to ensure that $Assign(\pi)$, $Rescale(\pi)$, and $Reassign(x)$ will always find an empty slot as above to assign to the new interval for π. This is not hard to ensure if non-deleted subintervals are separated by small gaps only. However, large gaps could build up as subintervals enter and leave intervals. Consequently, we need a background process which will start removing deleted subintervals and compacting non-deleted subintervals within an interval, once the interval reaches a certain fraction of its capacity. This process is described next.

The Compacting Process for an i-path π. It maintains the interval int_π as two halves. At any given time, one half, the insertion half, will be receiving newly inserted subintervals. At the same time the deleted subintervals in the other half are being removed and the non-deleted subintervals are being moved to the insertion half. By the time the insertion half is filled, the non-insertion half will be empty, and then their roles are toggled. Actually, the toggling occurs at a predetermined time when it is guaranteed that the non-insertion half is empty and the insertion half has not overflowed. WLOG consider the instant at which the right half becomes the insertion half. The compaction process moves through the subintervals in the left half from left to right, removing deleted subintervals, and reassigning undeleted subintervals at the leftmost possible slot in the right half (to the right of already assigned subintervals in this half). Insertions subsequent to the beginning of the compaction process will also be made in the right half until the stopping time, which is no later than when the right half becomes full. We will show that the compaction process in the left half will have finished by the time this happens. At this point, the compaction process will become active in the right half and insertions will be performed in the left half. We call the above process $Compact(\pi)$. Note that a single insertion can initiate a compaction process at each of its ancestor centroid paths.

Thus there are four kinds of processes which could be created when a node is inserted: *Assign*( ), *Rescale*( ), *Reassign*( ), and *Compact*( ). Each process is expensive and takes time at least proportioned to the size of the subtree in which it modifies codes (this updating of codes will be elaborated upon shortly in Section 6.4). Thus these proceses have to be performed, a poly-algorithmic number of operations at a time, over future insertions. Therefore, at any instant, a number of such processes could be underway.

6.4 Updating Codes

We consider the updates that need to be made to the various codes as a result of changes made to the intervals by *Assign*( ), *Reassign*( ), *Rescale*( ), and *Compact*( ) processes (as described in Section 6.3).

First, consider an $Assign(\pi')$ process initiated by some node x, which is the head node on i-path π, and becomes the first node on $(i+1)$-path π'. $Assign(\pi')$ must assign a new interval to π', and a new interval to π nested within the interval for π'. It must then change the codes at all nodes in T_x to reflect the change to the above two intervals. This is done as follows. The old separator string for π in the codes at nodes in T_x (including x itself) will be updated to the new separator string for π. In addition, the separator string for π' will be incorporated into the codes at all nodes in T_x. Thus the effect of $Assign(\pi')$ on the codes in T_x is to make π' appear as a path in their codes, but as a path with no primary nodes. $Reassign(x)$ will perform the changes needed to make x a primary node on π'.

Next, consider a $Reassign(x)$ process. It is initiated when x is in the process of leaving i-path π on which it is currently primary (recall x is also secondary on $(i+1)$-path π' in this scenario). The process must remove the separator string for π from the code at x and from the codes at all the nodes in the subtree rooted at that child y of x which is not on π. In addition, this process must assign a new separator string for the path containing y and modify the codes at all nodes in the subtree rooted at y to reflect this.

We need to specify the scheduling of $Reassign(x)$ in more detail. When $size(x)$ reaches 2^{i+1}, $Reassign(x)$ is made pending. At some future point, $Reassign(x_i)$ processes, $1 \leq i \leq k$, are all initiated, where nodes $x_1, x_2, \ldots, x_k$ are all the nodes currently primary on π and secondary on π'. These *Reassign*( ) processes are performed in top to bottom order (i.e. if the nodes $x_1, x_2, \ldots, x_k$ are in top to bottom order, then $Reassign(x_1)$, $Reassign(x_2)$, $\ldots$ $Reassign(x_k)$ are performed in turn). This collection of *Reassign* processes is called a *Reassign* superprocess.

A $compact(\pi)$ process for an i-path π must assign a new separator code to all children paths of primary nodes on π (secondary nodes are themselves contained in a child path of a primary node on π by Lemma 6.1). $Compact(\pi)$ must also update codes at all nodes in the subtree rooted at the head of π (other than codes for primary nodes on π). If π happens to be a zero length path, the above description applies to its one or two children paths.

Finally, a $Rescale(\pi)$ process must assign a new interval int_π to π contained within $int_{\pi'}$, where π' is the parent of π. In addition, it must update the codes of all nodes in $T_{head(\pi)}$, replacing the old separator for π with the new separator (corresponding to the change to interval int_π).

In addition to updating the codes, it is also necessary to update the annotations on the codes; recall the annotations label each bit in the code with the name of the centroid path whose separator contains this bit (also note the tagging mentioned in Section 4.1). This can be done in $O(\log n)$ time per node.

We have now described the overall structure of the algorithm. For each unit weight increase in a node (from 0 to 1), which occurs as a node is inserted, $O(\log n)$ work is done in updating the ancestor centroid paths of the inserted node and initiating a constant number of *Assign*, *Rescale*, *Reassign* and *Compact* processes for each such centroid path. Another $O(\log n)$ work is done to construct the code for the inserted node along with the annotations, using the code for its parent or child (this is explained in Section 6.5). The work needed to be done on this insertion in order to further the progress of unfinished processes will be described in Section 6.6. This part of the algorithm is what leads to the $O(\log^3 n)$ bound. Before describing this work, we introduce one more crucial aspect of the algorithm and also some crucial invariants which we will maintain.

Two Codes Instead of One. Even though all the above description has been in terms of one code per

node, we will actually maintain, not one, but two codes at each node. We will refer to these two codes as the *first code* and the *second code*. The reason for two codes is the following.

Consider two nodes x, y and consider any process which needs to modify codes at both these nodes. At some intermediate time instant, this process could have modified the code at x but not at y; as a consequence, the first bit of difference will no longer be related to the LCCP of x and y. To make matters worse, there could be several such processes which have modified codes at one but not both of x and y (actually, our algorithm will ensure that there is only one such process).

To ensure that LCAs are indeed related to the very first bit of difference, we will maintain two codes instead of one at each vertex. Each process will be required to update both codes for all vertices of interest. However, all first codes will be updated first, and all second codes will be updated only after all first codes have been updated. The critical property is that at any instant of time, either both the first codes at x, y would have been modified, or neither second code at x, y has been modified. Thus, either the first codes or the second codes will retain the relationship of the leftmost difference bit to the LCCP, at each instant of time.

In what follows, unless explicitly stated, we will refer to both the codes for node x collectively as the code at node x, or as $code(x)$.

6.5 Invariants Maintained

To ensure that queries are answered correctly at every instant, we schedule the various processes so that the two invariants described below are maintained.

Invariant 1 states that each process finishes before the situation which caused the initiation of the process changes too dramatically. Some additional terminology will be helpful.

Definition. A process associated with an i-path is called an i-process (i-*Assign*, etc.).

Definition. The size of path π, $size(\pi)$, is defined to be $|T_{head(\pi)}|$.

Definition. A weight increase is in the domain of path π, or into π for short, if it is applied to a node in the subtree rooted at the current head of π.

Invariant 1. Each process associated with i-path π completes within a weight increase of 2^{i-3} into π from the time the process was initiated. In addition:

(a) $Assign(\pi)$ is initiated when $size(\pi)$ reaches 2^i. (As $size(\pi)$ may never be exactly 2^i, it is helpful to pretend that time is continous and that the weight increases occur continuously, and then we can define the initiation of $Assign(\pi)$ to occur exactly when $size(\pi) = 2^i$.)

(b) $Compact(\pi)$ is initiated following each weight increase of 2^{i+1} into π from the time of π's creation (i.e. $Assign(\pi)$'s initiation).

(c) Pending $Reassign(x)$ processes associated with π are initiated following weight increase $h\,2^{i-2} + 2^{i-3}$ into π, from the time of π's creation, for each integer $h \geq 4$.

(d) $Rescale(\,)$ is initiated when $size(\pi)$ reaches $2^{i+1} - 2^{i-3}$.

Corollary 6.4. *A Reassign(x) process associated with $(i+1)$-path π', which removes x from i-path π, becomes pending when size$(\pi) = 2^{i+1}$ and head$(\pi) = x$, and completes before a further weight increase of $2^{i-1} + 2^{i-2}$ into π. In addition, if a Reassign(z) process is created when node z is inserted as the parent of node x with an already pending Reassign(x) process, the Reassign(z) completes before the Reassign(x) process completes.*

Finally, size$(\pi) < 2^{i+1} + 2^{i-1} + 2^{i-2}$.

Proof. *Reassign*(x) becomes pending when x becomes secondary on π', i.e. when $size(\pi) = 2^{i+1}$ with $head(\pi) = x$. By Invariant 1(c) applied to π', *Reassign*(x) is initiated before a further weight increase of 2^{i-1} into π' and completes within another weight increase of 2^{i-2} into π'. Any weight increase into π during this time is also into π'. The first claim follows. The claim about *Reassign*(z) follows due to the scheduling of *Reassign*( ) processes in top to bottom order.

To obtain the bound on $size(\pi)$ we show that $size(\pi)$ is less than $2^{i+1} + 2^{i-1}$ at the moment when a bunch of *Reassign*( ) processes removing nodes from π are initiated. Let x be the highest node on π not being reassigned. Then, at that moment, $|T_x| < 2^{i+1}$. Following a weight increase of 2^{i-1} into π', all the nodes being reassigned have been removed, and the next bunch of *Reassign*( ) processes is being initiated; at that moment again $size(\pi) < 2^{i+1} + 2^{i-1}$. The maximum size for π therefore occurs just before the *Reassign*( ) processes are completed, i.e. just before a weight increase of 2^{i-2} into π' (and hence into π) from the moment the *Reassign*( ) processes are initiated. This yields the claimed bound on $size(\pi)$. □

Corollary 6.5. *For each path π, there is at most one Assign(π), Rescale( ), Reassign(x) for x secondary on π, or Compact(π) underway at any given time.*

Invariant 2. For any pair of nodes x, y there is at most one *Assign*( ), *Reassign*( ), *Rescale*( ), or *Compact*( ) process which must modify the codes at both x and y and which has modified one but not both the first codes at these nodes. Similarly, there is at most one process which must modify the codes at both x and y and which has modified one but not both the second codes at these nodes. Finally, if a process with the former description exists then a process with the latter description cannot exist.

We remark that Invariant 1 is not hard to maintain. Similarly, Invariant 2 is easy to maintain using a simple blocking mechanism for processes. However, maintaining both invariants simultaneously is non-trivial. This is because weight increases in different parts of the tree may occur at different rates and unfinished processes will therefore be driven at different speeds by these weight increases. In particular, a process could be blocked indefinitely. One solution to this problem is to make a blocked process help the blocking process. This will be described in detail in Section 6.6.

We are now ready to show that the intervals are sufficiently large.

Lemma 6.6. *Suppose the Invariant 1 holds and that $c' \geq 5$ and $c = 2c'$. Let π be an i-path.*

(a) The insertion side of int$_\pi$ cannot be filled up prior to the completion of Rescale(π) (recall that int$_\pi$ has size 2^{ic} before Rescale(π) completes).

(b) The insertion side of int$_\pi$ following the completion of Rescale(π) will not fill up before the first initiation of Compact(π) (int$_\pi$ now has size $2^{ic+c'}$).

(c) Consider an initiation of Compact(π). Over a subsequent weight increase of 2^{i+1} into π, the side of int$_\pi$ being filled by Compact(π) will have room for the subintervals being added to int$_\pi$.

Proof. The proof uses an induction on i. The result is trivially true for $i = 0$. Thus it remains to prove the inductive step. We start with part (a). Up to the completion of $Rescale(\pi)$, $size(\pi)$ remains less than 2^{i+1}. We show that all the subintervals that could be generated until $Rescale(\pi)$ completes will fit into int_π.

We sum the length of all the subintervals inserted into int_π since the initiation of $Assign(\pi)$. Some of these intervals may have been deleted by the time $Rescale(\pi)$ completes. The length of each interval that is inserted is at most $2^{(i-1)\,c+c'}$. A particular subtree rooted at an off-path node could repeatedly delete and insert subintervals increasing in size by successive factors of $2^{2c'}$. Thus, such a subtree, whose root is a primary node in an h-path, could be responsible for a total subinterval of length $\sum_{j=0}^{2h+1} 2^{jc'} < \frac{2^{(2h+2)c'}}{2^{c'}-1}$. The total length of gaps between subintervals is at most the total length of the subintervals. Since $Rescale(\pi)$ completes following a weight increase of 2^i into π from the time π was created, the total length of int_π occupied by deleted and undeleted subintervals and the gaps between them is at most $2\sum_r \frac{2^{c'(2h_r+2)}}{2^{c'}-1}$, where $\sum_r 2^{h_r} \le 2^{i+1} = 4 \cdot 2^{i-1}$, and $h_r \le i-1$. This is at most $\frac{8 \cdot 2^{ic'}}{2^{c'}-1} \le \frac{2^{2ic'}}{2}$ for $c' \ge 5$.

The proof of part (c) is broadly similar. WLOG we assume that $Compact(\pi)$ inserts into the right half of int_π (now int_π has size $2^{ic+c'}$). We show that from the time of the initiation of $Compact(\pi)$, the subintervals in int_π when $Compact(\pi)$ was initiated together with any new subintervals inserted up till the next initiation of $Compact(\pi)$ will all fit in the right half of int_π. When $Compact(\pi)$ is initiated, $size(\pi) < 2^{i+1} + 2^{i-1} + 2^{i-2}$; the next initiation of $Compact(\pi)$ occurs following a weight increase of 2^{i+1} into π. The length of each subinterval inserted into int_π is at most $2^{2ic'}$; such a subinterval would be due to a subtree of size at least 2^i. Similarly to before, the total length of subintervals for which such a subtree could be responsible is less than $2^{2ic'} + \sum_{j=0}^{2i-1} 2^{jc'} < \frac{2^{(2i+1)\,c'}}{2^{c'}-1}$. A smaller subtree, whose root is a primary node on an h-path could contribute subintervals of total length no more than $\frac{2^{(2h+2)\,c'}}{2^{c'}-1}$. As before, the total length of the right half of int_π occupied by deleted and undeleted subintervals and the gaps between them is less than $\frac{2a \cdot 2^{(2i+1)\,c'}}{2^{c'}-1} + 2\sum_r \frac{2^{(2h_r+2)\,c'}}{2^{c'}-1}$, where $a \cdot 2^i + \sum_r 2^{h_r} < 2^{i+1} + 2^{i-1} + 2^{i-2} + 2^{i+1}$ and $h_r \le i-1$. This is at most $\left(8 \cdot 2^{(2i+1)\,c'} + 4 \cdot 2^{2ic'}\right)/(2^{c'}-1) \le 12 \cdot 2^{(2i+1)\,c'}/(2^{c'}-1) \le 2^{(2i+1)\,c'}/2$, if $c' \ge 5$.

The proof of part (b) is identical to that of part (c) except that we need only consider the intervals due to an initial weight of 2^i (when π was created) and a weight increase of 2^{i+1} (namely, up to the time when $Compact(\pi)$ is first initiated). □

6.6 Some Details of Processes

Before proceeding, some details of *Assign*( ), *Reassign*( ), *Rescale*( ), and *Compact*( ) processes need to be carefully examined. These details arise because a process is performed over a sequence of future insertions. A fundamental issue here is that nodes can lie on two paths, on one as a primary node, on the other as a secondary node.

Our goal, which enables the $O(1)$ query procedure is to establish the following claim.

Claim 6.7. *The changes due to x becoming primary on path π' and ceasing to be primary on path π are reflected in the code at a descendant node y of x only due to the modifications performed by the process carrying out x's change of primary paths.*

Process Blocking. To maintain Invariant 2, the following strategy is used. A process first updates all the first codes for the nodes it needs to process in preorder and then updates all the second codes, again in

preorder. When the process begins work on the first codes of a subtree rooted at a node z it will mark z as blocked and will only remove the mark when it has finished updating the first codes in z's subtree. It will follow the same blocking procedure when updating second codes. If a process P comes to a blocked node it will not proceed with its work until the node is unblocked (later, we explain the helping of the blocking process undertaken by P while it is waiting).

In addition, a process keeps the topmost node it is updating blocked until it completes. Finally, a *Reassign*(x) process, in addition to keeping x blocked throughout its processing, will as its last step make x primary on the path on which x had been secondary.

We specify later just how a newly inserted node is marked.

Constructing Codes for Newly Inserted Nodes. Recall that each node has two associated codes, a first code and a second code. For a newly inserted node x, each code is obtained from the corresponding code for its parent or child as follows.

1. If x is inserted as a child of a leaf node then it is given the same code as its parent. This reflects the fact that x lies on the same 0-path as its parent. Of course, if and when x's weight increases sufficiently, its parent will leave this 0-path.

2. If x is inserted as a leaf child of an internal node v then it forms a new singleton path. A new interval is assigned for this path. The code for x is just the code for its parent appended with the separator string for this new singleton path. There is one more complicated scenario, which arises when a *Reassign*(v) process is underway, and the first code for v has been updated, but the second has not. Let π be v's primary path and π' its secondary path. In this case, x is assigned two subintervals, one in $int_{\pi'}$ for its first code, and one in int_{π} for its second code. When the *Reassign*(v) process updates second codes it will replace the subinterval in x's second code with the subinterval in $int_{\pi'}$ used in its first code.

3. If x is inserted as an internal node, then the code for x is made identical to that for its child. This makes x primary on the same path π as its child. In addition, if x's child is marked then so is x. Of course, it may be that x is also added to the parent path of π, in which case a *Reassign*(x) process is created. There is one exception. Let y be x's child. If a *Reassign*(y) is already underway, the code for x is set to the updated code for y, i.e. the code with separator(π) removed, and in this case x is not marked.

We mention here that Invariant 2 also applies to newly inserted nodes x and y which inherit their codes from partially processed nodes (i.e., nodes for which one but not both codes have been updated by some process).

Process Roots. Each process must modify codes at certain nodes in the tree. The node closest to the root of T_1 whose code a process must modify is called the *root* of the process. Since nodes are inserted dynamically, the root of a process needs to be defined more precisely. For an *Assign*(π), *Compact*(π), or *Rescale*(π) process, *head*(π) is the root of the process; if a new node z is inserted and becomes the head of π as one of these processes is underway, z becomes the new root of the process. While one of these processes is underway no node x leaves π due to a *Reassign*(x) process until the *Assign*(π), *Rescale*(π) or *Compact*(π) has completed; this is a consequence of the blocking strategy. For a *Reassign*(x) process, the root is always x.

Helping a Blocking Process. If a process P reaches a marked node x, it will seek to help an unblocked process in T_x so as to advance the processing that will lead to the removal of the mark on x. To this end it traverses a path of marked nodes from x; at the last marked node y, it discovers the process Q that marked y and performs the next code update for Q. This may entail unmarking $O(\log n)$ nodes and marking up to one node (since Q marks a non-leaf node prior to updating it). To facilitate this process, each mark includes the name of the process making the mark. As it suffices to have P help some unblocked process on which it is waiting either directly or indirectly, it suffices that P traverse a maximal path of marked nodes in T_x; thus this proces takes $O(\log n)$ time. It is called a *basic step.*

i-process P could be blocked by an ancestral process or by a descendant process. We will need to ensure that P is blocked by at most one ancestral process. Further, this only happens at P's initiation. If P is so blocked, it puts a subsidiary mark on its root. The meaning of this mark is that as soon as Q's mark is removed, where Q is the blocking process, P's mark is then instantiated. As there is only one active process per path, there are at most two such subsidiary marks per node (two paths can share a root, either because a new path has only secondary nodes and hence shares its root with a child path, or because a path temporarily has no nodes; the nodeless path will use its parent node as its root for any associated processes). In the case that two marks are present, the mark for the deeper path will be instantiated. It remains the case that each process P is blocked by at most one ancestral process.

Process Interleaving and its Effect on Code Updates. We need to examine the interleaving of processes for paths π and π', π' a child of π, and how this may affect the update of code portions corresponding to $sep(\pi)$ and $sep(\pi')$.

Because of their relative timing, $Compact(\pi)$, $Rescale(\pi)$, $Assign(\pi)$ and $Reassign(x)$ for x secondary on π do not overlap.

$Compact(\pi)$ and $Reassign(x)$ for x secondary on π, update only the portion of the code corresponding to $sep(\pi')$ for π' a child of π; $Rescale(\pi)$ only updates the portion of the code corresponding to $sep(\pi)$; $Assign(\pi)$, on the other hand, updates the portions of the code corresponding to both $sep(\pi)$ and $sep(\pi')$, where π' is the primary path for nodes secondary on π. Thus we will need to consider the possible interaction of $Compact(\pi)$ and $Assign(\pi')$ or $Rescale(\pi')$, and of $Reassign(x)$ and $Assign(\pi''')$ or $Rescale(\pi''')$ where x is secondary on π, primary on π', and π''' is the off-path child of x (w.r.t. π').

Consider such a $Compact(\pi)$ process. Let y be an off-path child of π, secondary on π' and primary on π''. So there is an $Assign(\pi')$ process at hand. If the $Compact(\pi)$ process updates $code(y)$ first, then π'' receives a new subinterval in the insertion half of int_π, the right half say; subsequently, the $Assign(\pi')$ gives π' a new subinterval in the right half of int_π. On the other hand, if the $Assign(\pi')$ process updates $code(y)$ first, but after the $Compact(\pi)$ process has been initiated, then the $Assign(\pi')$ process gives π' a new subinterval in the right half of int_π, and subsequently the $Compact(\pi)$ process does nothing further to $int_{\pi'}$ and the codes for nodes in T_y. The possible interactions of $Compact(\pi)$ and $Rescale(\pi')$ are identical.

Next, we consider a $Reassign(x)$ process for a node x, primary on π' and secondary on π. Let y be the child of x which is not on π'. Let π'' be the path containing y. After the creation of the $Reassign(x)$ process suppose that y becomes secondary on a new centroid path π''' (see Fig. 4). π''' now becomes the parent of π'' and an $Assign(\pi''')$ process is initiated. We consider two cases in turn.

First, suppose that $Assign(\pi''')$ modifies the code for y before $Reassign(x)$ does so. Then the separator for π' will be in $code(y)$ when it is processed by $Assign(\pi''')$. Thus $int_{\pi'''}$ will be assigned so that it is nested within $int_{\pi'}$. At some subsequent point, $Reassign(x)$ will remove the separator string for π' from $code(y)$

and reassign $int_{\pi'''}$ so that it is nested within int_{π}.

Second, suppose that $Reassign(x)$ has updated $code(y)$ before $Assign(\pi''')$. Then, when $Assign(\pi''')$ processes $code(y)$ it no longer contains the separator for π'. The behavior of $Assign(\pi''')$ is now as expected with $int_{\pi'''}$ being assigned so as to be nested within int_{π}. But note that when $Reassign(x)$ processed $code(y)$, $code(y)$ indicated that y was primary on π''. So π'' was reassigned a new subinterval within int_{π} and the resulting separator string was included within $code(y)$. At some later instant, $Assign(\pi''')$ included the separator string for π''' in $code(y)$, and replaced the current separator string for π'' with a new separator string corresponding to a subinterval nested within $int_{\pi'''}$. Ultimately, the $Reassign(y)$ process created by y becoming secondary on π''' removed the latter separator string from $code(y)$. Again, the possible interactions of $Reassign(x)$ and $Rescale(\pi''')$ are identical.

Consider Claim 6.7. Note that it would not hold, if in the first case above, $Assign(\pi''')$ assigned an interval to π''' nested within the interval for π and incorporated the associated separator into $code(y)$. For if $Assign(\pi''')$ did so then the change resulting from x becoming primary on π would be reflected in $code(y)$ by a modification made by $Assign(\pi''')$ and not by $Reassign(x)$.

It is possible for the updates described above involving two processes ($Compact(\pi)$ and $Assign(\pi')$ or $Rescale(\pi')$, $Reassign(x)$ and $Assign(\pi''')$ or $Rescale(\pi''')$) to occur in a different order on the two codes. The only possible interleaving for the $Compact(\pi)$ process is that first $Compact(\pi)$ updates the first codes of nodes in T_y, then $Assign(\pi')$ (or $Rescale(\pi')$) updates both codes of nodes in T_y, then $Compact(\pi)$ examines T_y but makes no further updates to the codes. For the $Reassign(x)$ process, the only possible interleaving is that first $Reassign(x)$ updates the first codes of nodes in T_y, then $Assign(\pi''')$ (or $Rescale(\pi''')$) updates both codes of nodes in T_y, the $Reassign(x)$ updates the second codes of nodes in T_y. Each update follows the appropriate rules for the codes it encounters which will differ for the first and second codes. All we have to ensure is that the subintervals chosen for the second codes within a particular interval are the same as those selected for the first codes within the same interval. Thus for example, in the $Reassign(x)/Assign(\pi''')$ scenario above, $Assign(\pi''')$ assigns a separator for π''' to $code(y)$, $(sep(\pi'''))^1$ say, contained in int_{π}, while for the second code it assigns a separator $(sep(\pi'''))^2$ for π''' contained within $int_{\pi'}$, and then $Reassign(x)$ replaces this separator with $(sep(\pi'''))^1$.

6.7 Processing Queries

The algorithm ensures that the first bit of difference between either the first codes at x and y, or the second codes at x and y, is related to the LCCP, which itself can be obtained using the algorithm detailed below. The first step is to find the leftmost bit of difference d_{first}, between the first codes at x and y, and thereby to find the rightmost path π_{first} such that the first codes at x and y agree on all separators up to and including that for π_{first} Similarly, d_{second} and π_{second} are identified with respect to the second codes at x and y.

Consider $z_{first} = head(\pi_{first})$ and $z_{second} = head(\pi_{second})$. As we will see, if the first code of z_{first} agrees with the first code of x (and of y) on all separators up to π_{first} and the same is true for z_{second}, then one of π_{first} and π_{second} is the LCCP of x and y; if it only holds for z_{first} then π_{first} is the LCCP, and similarly for z_{second}.

Lemma 6.8. *If a process P updates the separator $sep(\pi)$ for a path π, then $sep(\pi)$ is identical either for all first codes for nodes in $T_{head(\pi)}$ or for all second codes for nodes in $T_{head(\pi)}$, and when P unmarks $T_{head(\pi)}$ it is identical both for all first codes for nodes in $T_{head(\pi)}$ and for all second codes, although its encoding in the first and second codes may differ.*

Proof. The proof uses an induction on time. So assume that the result is true at the moment P marks $T_{head(\pi)}$. Any process that updates $sep(\pi)$ must mark $T_{head(\pi)}$, thus until P unmarks $T_{head(\pi)}$, only P can update $sep(\pi)$ in the codes for nodes in $T_{head(\pi)}$. The lemma follows. □

Comment. The interleaving described in the previous section shows that the two codes may differ following completion of a process, though when the second process touching these codes also completes, the equality will be restored.

Lemma 6.9. *If $sep(\pi)$ appears in $code(x)$ then $head(\pi)$ is an ancestor of x.*

Proof. The proof is by induction on time. When a node is inserted, if a leaf it inherits its code from its parent and thus the claim is true at this point; if an internal node, it inherits its code from its child and the claim is true at this point too.

An update which changes $head(\pi)$ will occur only after $sep(\pi)$ is appropriately updated in the codes for all descendants of $head(\pi)$ and thus the claim continues to hold. □

Let z be the LCA of x and y, and suppose that z is primary on π and let $w = head(\pi)$. Then:

Lemma 6.10. *Suppose either that w is not marked, or if w is marked, then WLOG it is the second codes in T_w that are begin updated. Then z and z_{first} lie on the same path. Also, the first code of z_{first} agrees with the first code of x on all the separators up to π_{first}. Finally, π_{first} is the LCCP of x and y.*

Proof. By Lemma 6.8 any encoding $sep(\tilde{\pi})$ for path $\tilde{\pi}$ appearing in the first code of z must also appear in the first codes of x and y. Thus z_{first} lies on the same path as z or is a descendant of z. But clearly if $sep(\tilde{\pi})$ appears in $code(x)$ then there is a node v on $\tilde{\pi}$ with v an ancestor of x (v may be primary or secondary on $\tilde{\pi}$). Thus there are nodes on π_{first} ancestral to each of x and y, which must include $z_{first} = head(\pi_{first})$. Thus z_{first} and z lie on the same path. □

Corollary 6.11. *One of π_{first} and π_{second} is the LCCP of x and y.*

By Lemma 6.9 π_{first} and π_{second} are both ancestral to each of x and y. Thus if $\pi_{first} = \pi_{second}$ they are both the LCCP. Otherwise, let π_{first} be a j-path and π_{second} a k-path. The one with larger index (among j and k) provides the LCCP.

This yields the $O(1)$ algorithm for finding the LCCP and hence the LCA.

6.8 Running Time

In order to maintain Invariant 2 while meeting Invariant 1, a process may need to help (perform the work of) other processes that are blocking it. Thus the main issue is to determine how much work a process may do. We measure this work in *Basic Steps.*

Definition. A basic step of a process is the traversal of $O(\log n)$ edges of the subtree it is processing followed by the updating of one of the codes at a single node in this subtree. (The final basic step entails only edge traversals.)

Claim 6.12.

(i) A basic step takes $O(\log n)$ time to perform.

(ii) A process rooted at node x performs at most $2|T_x| + 1$ basic steps on it task, where $|T_x|$ is the size of tree T_x in vertices immediately before the completion of the process.

Lemma 6.13. *An Assign(π), Compact(π), Rescale(π), or a Reassign superprocess associated with class i path π performs at most $e \cdot 2^i \cdot i$ basic steps on itself and all the tasks it helps, for a suitable constant $e > 0$.*

Lemma 6.13 depends on a scheduling algorithm which ensures that for each weight increase of 2^{i-3} into path π, at least $e \cdot i \cdot 2^i$ basic steps are performed on process P and the tasks it helps, where P is associated with path π. We describe the scheduler later. Clearly, if the scheduler exists, then the truth of Lemma 6.13 up to a given time immediately implies Invariant 1 also holds up to that time.

Proof of Lemma 6.13. The proof uses a double induction, the outer induction being on t, the weight increase since the data structure was initiated (time for short), and the inner induction being on the class i of the i-path π. Note that time proceeds in increments of $1/\log^3 n$. Our proof depends on the following claim, whose proof uses the inductive hypothesis.

Claim 6.14. *Suppose Lemma 6.13 holds through time t, and for processes associated with h-paths, $h < i$, at time $t + 1/\log^3 n$. Then consider an i-path π at time $t + 1/\log^3 n$ with head x. Consider the collection of all h-processes, $h < i$, having roots in T_x which have been activated by time $t + 1/\log^3 n$. The total number of basic steps that have been performed on all these processes through time $t + 1/\log^3 n$ since their first activation is $O(i \cdot 2^i)$.*

Proof of Claim 6.14. We first note that if Lemma 6.13 holds for a given process P associated with an i-path at its termination, then the size of the subtree rooted at P's root at its termination is less than $2^{i+1} + 2^{i-1} + 2^{i-2}$, by Corollary 6.4. Consequently, if a process associated with an i-path is not complete at a time t' for which Lemma 6.13 holds, then subtree rooted at the process root has size less than $2^{i+1} + 2^{i-1} + 2^{i-2}$. In particular, the subtree rooted at P's root has size less than $2^{i+1} + 2^{i-1} + 2^{i-2}$ at time t, and hence size less than $2^{i+1} + 2^{i-1} + 2^{i-2} + 1/\log^2 n$ at time $t + 1/\log^3 n$.

Now, we bound the number of basic steps performed by each type of process having its root in T_x, where x is P's root.

***Assign( )* processes.** There is one *Assign*( ) process for each path inside T_x. By the inductive hypothesis, for an *Assign*(π') process associated with h-path π', $h < i$, the associated subtree has size less than $2^{h+1} + 2^{h-1} + 2^{h-2}$ at time $t + 1/\log^3 n$ and hence the *Assign*(π') process has had at most $O(2^h)$ basic steps performed on it. For each $h < i$, each h-path has a distinct set of nodes of combined size at least 2^h associated with it, namely either the subtree rooted at the head of the path, or if the h-path π' has h-path π'' as a child, the associated nodes are given by $T_{head(\pi')} - T_{head(\pi'')}$. Thus there are at most $\left(2^{i+1} + 2^{i-1} + 2^{i-2} + 1/\log^2 n\right)/2^h$ h-paths in T_x, and the total number of basic steps performed through time $t + 1/\log^3 n$ on their *Assign*( ) processes is $O(2^i)$. Summing over all $h < i$ gives a bound of $O(i \cdot 2^i)$ on the number of basic steps performed on *Assign*( ) processes for h-paths, $h < i$, inside T_x.

***Rescale( )* processes.** The analysis is identical to that for the *Assign*( ) processes.

***Compact*( ) processes.** Recall that successive $\mathit{Compact}(\pi')$ processes for h-path π' are separated by weight increases of 2^{h+1} into π'. Likewise the first $\mathit{Compact}(\pi')$ process occurs following a weight increase 2^{h+1} from the initiation of $\mathit{Assign}(\pi')$. Further, each weight increase at a node contributes only to the weight increase for paths that are the node's ancestors, hence for at most two h-paths for each value of h. Thus at most $2\left(2^{i+1}+2^{i-1}+2^{i-2}+1/\log^2 n\right)/2^{h+1}$ $\mathit{Compact}(\pi')$ processes have been activated for h-paths π' contained in T_x. By the inductive hypothesis, these processes have each had at most $O(2^h)$ basic steps performed on them by time $t+1/\log^3 n$, and hence summing over all $h<i$, and all paths in T_x, yields a bound of $O(i\cdot 2^i)$ basic steps performed on the *Compact*( ) processes for h-paths, $h<i$, inside T_x.

***Reassign*( ) processes.** The argument is very similar to that for the *Compact*( ) processes, with each bunched *Reassign*( ) superprocess resembling a *Compact*( ) process in its cost. We account for a bunched h-superprocess by associating it with the at least 2^{h-2} insertions to its associated h-path π' from either the time of the creation of π' or the start of the previous *Reassign*( ) superprocess associated with π', whichever is more recent, to the moment the current superprocess begins to be processed.

Consider a weight increase; it is charged for the *Reassign*( ) superprocesses at its ancestors, i.e., for at most two *Reassign*( ) h-superprocesses, for each h. It follows that there have been at most $2\left(2^{i+1}+2^{i-1}+2^{i-2}+1/\log^2 n\right)/2^{h-2}$ such superprocesses activated in T_x. Summing over all paths and $h<i$, we conclude that a total of $O(i\cdot 2^i)$ basic operations have been performed on the *Reassign*( ) superprocesses with roots in T_x.

This concludes the proof of Claim 6.14.

We now complete the proof of Lemma 6.13.

Consider the currently active process P associated with i-path π, if any. By Claim 6.14, P has performed at most $O(i\cdot 2^i)$ basic operations helping processes associated with h-paths, $h<i$. Since P has not completed at time t, as already noted, $|T_x|<2^{i+1}+2^{i-1}+2^{i-2}+1/\log^2 n$ at time $t+1/\log^3 n$. Thus the number of basic steps performed on P's task and the up to one other task it may help associated with some ancestral j-path, $j>i$, is $O(2^i)$. This gives a total bound of $O(i\cdot 2^i)$ on the basic steps performed by P. □

It remains to describe the processing performed following a weight increase. Following a $\Theta(1/\log n)$ weight increase at a node, the size of each of its $O(\log n)$ ancestral central paths are incremented, up to $O(\log n)$ new secondary nodes are added to the paths to which they newly belong, and up to $O(\log n)$ new processes are initiated. Then e basic steps are performed on the active process at each ancestral centroid path, if any, for a total of $O(\log n)$ basic steps, which takes $O(\log^2 n)$ time. Node insertion costs $O(\log n)$ per node, but there are $O(1)$ node insertions per unit weight increase, so this is relatively insignificant.

The $O(1)$ time algorithm requires a more elaborate scheduling procedure for two reasons. First, we cannot keep the recorded sizes of the centroid paths completely up to date and so processes may be late in getting underway, and second, we do not want to have multiple basic steps partially completed. This leads us to perform basic steps over a $\Theta(1/\log^2 n)$ weight increase once started, even if the weight increase is not all occuring in the relevant subtree. Our scheduling procedure is based on a variant of the Dietz-Sleator "cup-filling" methodology [DS87], which we develop in the next section.

6.9 Dietz-Sleator "Cup-Filling" with Dated Priorities

We seek to handle a task scheduling scenario of the following flavor. There are at most k tasks at any one time. Tasks have associated priorities which increase, possibly at different rates, as they are delayed.

Further, the known priorities may be somewhat dated and hence potentially inaccurate. Our goal is to determine conditions under which the following strategy is effective: schedule for *atom* steps the task with current highest known priority and iterate.

So let $\Gamma = \{P_1, P_2, \cdots\}$ be a collection of no more than k tasks. Suppose each task is performed in atomic chunks of length *atom* and suppose each task has length at most ℓ, an integer multiple of atom. Task P_i has an associated priority $p_i \geq 0$. Priorities only increase. At any time a new task of priority 0 may be created so long as there are at most k tasks altogether. It will be convenient to keep placeholder tasks of priority 0 to ensure that there are always exactly k tasks at hand.

After every atom steps a task P_i is chosen to be executed for the next *atom* steps (possibly, but not necessarily, the same task as on the previous *atom* steps) which satisfies the following rule:

$$\lambda p_i - \text{work performed on } P_i + \lambda \cdot \mathit{error} \geq \lambda p_j - \text{work performed on } P_j \text{for all } j \neq i,$$

where *error* represents the maximum error in the recorded priorities (as opposed to the actual priorities p_i, p_j) and $\lambda > 0$ is a scale parameter relating priorities to steps executed. After executing these *atom* steps, the priorities of an arbitrary subset of tasks are increased by a combined total of at most *p-inc*.

Lemma 6.15. *The above scheduling algorithm, if atom* $\geq 4\lambda \max\{\mathit{error}, \textit{p-inc}\}$*, satisfies*

$$\lambda p_i + \ell - \textit{work performed on } P_i \leq \lambda(\mathit{error} + \textit{p-inc}) + (\mathit{atom} + \ell) + 4\lambda(\mathit{error} + \textit{p-inc}) \log k.$$

Corollary 6.16. $p_i \leq (\mathit{error} + \textit{p-inc}) + 1/\lambda(\mathit{atom} + \ell) + 4(\mathit{error} + \textit{p-inc}) \log k.$

Proof of Lemma 6.15. We use a potential argument. We show that if there is a task P_j with priority plus $1/\lambda$ times remaining work (strictly, $1/\lambda(\ell - \text{work performed on } P_j)$) at least $\ell/\lambda + \mathit{error} + \mathit{atom}/\lambda$, when a task is chosen to have *atom* steps executed, then the potential will not increase following these *atom* steps being performed and the applying of the combined *p-inc* increment to the priorities.

We associate potential c^{r_i} with task P_i, $1 \leq i \leq k$, where $c > 1$ is a suitable constant and r_i is defined as follows: $r_i = \lambda p_i + \ell -$ work performed on P_i.

Clearly, following *atom* steps being executed on P_i, and potentials being incremented by *p-inc*, the maximum increase in potential occurs if all the incremental priority is concentrated on the task P_j with largest r_j. We note that $r_i + \lambda \cdot \mathit{error} \geq r_j$ prior to the execution of *atom* steps on P_i. Thus if $r_j \geq \ell + \lambda \cdot \mathit{error} + \mathit{atom}$, then $r_i \geq \ell + \mathit{atom}$, so after *atom* steps of work are performed on P_i, P_i's potential decreases by a multiplicative factor of c^{atom}. We want to ensure that the potential does not increase. For this, it suffices that:

$$c^{r_j + \lambda\, \textit{p-inc}} + c^{r_i - \mathit{atom}} \leq c^{r_j} + c^{r_i}.$$

Clearly, this is hardest to satisfy with $r_i + \lambda \cdot \mathit{error} = r_j$; so it suffices that:

$$c^{\lambda(\mathit{error}\, +\, \textit{p-inc})} + c^{-\mathit{atom}} \leq c^{\lambda \cdot\, \mathit{error}} + 1$$

Let $\Delta = \lambda \max\{\mathit{error}, \textit{p-inc}\}$. Choosing c so that $c^{2\Delta} = \sqrt{2}$, and choosing $\mathit{atom} \geq 4\Delta$ yields a sufficient condition of

$$\sqrt{2} + \left(1/\sqrt{2}\right)^2 \leq 1 + 1,$$

which is true.

Thus the largest potential possible is less than

$$(k-1)\,c^{\lambda error+(atom+\ell)}+c^{\lambda(error+p\text{-}inc)+(atom+\ell)}\le k\,c^{\lambda(error+p\text{-}inc)+(atom+\ell)}\ .$$

Hence $c^{r_i}\le k\,c^{\lambda(error+p\text{-}inc)+(atom+\ell)}$ for all i, from which the claimed bound on r_i follows. □

A special case arises when $error=0$ and $atom=\ell$.

Corollary 6.17. *If $error=0$ and $atom=\ell$, then $p_i\le(9+4\log k)$ p-inc.*

Proof. Set $\lambda=atom/(4p\text{-}inc)$. □

Dietz and Sleator proved this bound with somewhat tighter constants, namely $p_i\le p\text{-}inc\cdot(\log k+O(1))$. This is often called the Dietz-Sleator cup-filling lemma.

6.10 Scheduling in the $O(1)$ Time Algorithm

In the $O(1)$ time algorithm a layered structure will be used. The tree is binarized and partitioned into subtrees of size $O(\log^3 n)$. The roots of these subtrees and their LCAs form an implicit tree on which the previous $O(\log^3 n)$ time update algorithm is run. Intuitively, the subtree roots change only every $\Theta(\log^3 n)$ insertions, which provides enough time to perform the $O(\log^3 n)$ operations needed for an update.

Let T denote the tree of n nodes on which LCA queries are being performed and let T_1 denote the implicit tree. As we will see, T_1 has at most $4n/\log^3 n$ nodes, and this relationship applies to each subtree of T and the corresponding subtrees of T_1. An insertion of a node v in T will result in a weight increase of either 0 or $4/\log^3 n$ to the following node in T_1: the node that is the root of the size $O(\log^3 n)$ subtree containing v in the binarized version of T. The rule for the weight increase is discussed later in Section 7 when we discuss how to maintain the size $O(\log^3 n)$ subtrees.

Also, a new node of weight zero may be inserted in T_1 as a result of an insertion into T. As already noted, weight 0 nodes are adjacent only to weight 1 nodes. Again, details on when this happens are given in Section 7.

At this point, we describe a schedule that updates path weights and performs *Assign*( ), *Rescale*( ), *Reassign*( ) and *Compact*( ) processes as needed but with only $O(1)$ work per insertion to T.

The major difficulty we face is that on updating the weight of a node (as a result of an insertion in T), we cannot immediately update the weights of all the ancestral centroid paths (note that we only track the weight of the subtrees rooted at the heads of centroid paths—together with the individual node weights, this suffices to track the weight changes when a head node leaves a centroid path). Instead we create a weight update task for each node in T_1. The weight update task for node v is responsible for updating the weights of head nodes ancestral to v to reflect an increase in v's weight. It may be run multiple times. The execution of weight update tasks is alternated with the execution of *Assign*( ), *Rescale*( ), *Reassign*( ), and *Compact*( ) processes in stages of length $\Theta(\log n)$, with each update weight task being run to completion once initiated. An update weight task will take $O(\log n)$ time to run to completion, as we will see. This ensures that the *Assign*( ), *Rescale*( ), *Reassign*( ), and *Compact*( ) processes, when underway, always see a tree T_1 with consistent weights at the different head nodes.

Recall that we also need to track the weight of insertions made to each i-path so as to know when to initiate an i-process (it suffices to keep track of this value $\bmod 2^{i+1}$). To this end, each update weight task also increments these weights. The update weight task for node v needs to store two values: the first value

is the increment being made to each of its ancestral centroid paths if it is underway, which equals the weight increment to v between the start of the previous and current runs of the task; the second value is the weight increment to v since the task last began running.

The weight update tasks are scheduled using the standard Dietz-Sleator cup-filling scheduler. A task's priority is given by the sum of the two increment values it holds. Here $atom = \Theta(p\text{-}inc) = \Theta(1/\log^2 n)$ and $k = \Theta(n/\log^3 n)$, for $p\text{-}inc$, the increase in priority during the execution of one task, is defined to be a tight upper bound on the total weight increase to T_1 when performing one run of one task. We choose the constant of proportionality so that the start of successive runs of the task for a node v are separated by at most a weight increase of $\alpha/\log n$ on v's part, for a suitable constant $\alpha > 0$.

Remark 6.18. *This implies that when a weight is updated, it is updated by at most* $p\text{-}inc \leq \alpha/(4\log^2 n) \leq 1/\log^2 n$*, as we will choose* $\alpha \leq 1$.

By Corollary 6.17 this entails that $(9+4\log n)\, p\text{-}inc \leq \alpha/\log n$, i.e. that an update task runs to completion during a period bounded by a weight increase of $\alpha/[(9+4\log n)\log n]$ to T_1. But such a weight increase requires $\Omega(\log n)$ insertions, and as one run of the task takes $O(\log n)$ time, this takes $O(1)$ time per insertion to T.

Now we can show:

Lemma 6.19. *The recorded size of an i-path is at most* $\alpha\left(2^{i+1}+2^{i-1}+2^{i-2}\right)/\log n$ *smaller than the actual size and no larger than the actual size, assuming the actual size is less than* $2^{i+1}+2^{i-1}+2^{i-2}$.

Proof. Consider the subtree rooted at the head node of the i-path. If it has r nodes of weight 1 it has at most $2r+1$ nodes of weight less than 1 (since all but possibly one node of weight less than 1 has a parent, necessarily of weight 1, and since each weight 1 node has at most one child of weight less than 1). Since the subtree has size less than $2^{i+1}+2^{i-1}+2^{i-2}$, it has at most $2^{i+1}+2^{i-1}+2^{i-2}$ nodes of weight less than 1. But the recorded weight of each such node is in deficit by at most $\alpha/\log n$, and the result follows. □

We are ready to describe the scheduling of the *Assign*( ), *Rescale*( ), *Reassign*( ) and *Compact*( ) processes. For each i, we run a separate modified cup-filling procedure for the i-processes. The priority of a process is simply the weight of insertions in the associated subtree since the moment when the process would have been initiated in our original algorithm. For an *Assign*(π) process this is approximated using the current size of i-path π, minus 2^i; the size of π when *Assign*(π) should have been initiated. For *Compact*(π) and *Reassign*( ) superprocesses, we need to record the weight of insertions mod 2^{i+1} that have occurred in the subtree rooted at the head of i-path π since π was created. This term minus the starting time of the process mod 2^{i+1}, as specified in Invariant 1 yields the priority (for *Reassign*( ), the priority is calculated with a shift of $2^{i-3} \mod 2^{i+1}$). It follows the recorded priority maybe too small, but by at most $\alpha(2^{i+1}+2^{i-1}+2^{i-2})/\log n$.

We perform each scheduled process for one basic step, cycling among the classes of processes for each class of i-path ($i = 0, 1, 2, \cdots$) in round robin order. We alternate between performing one basic step and one complete task updating path sizes. This ensures the recorded sizes of the centroid paths are always consistent when basic steps are being performed. Thus every $\Theta(\log^2 n)$ insertions, one basic step is performed on one process associated with a class i-path for each $i = 1, 2, \ldots$.

Lemma 6.20. *With the following parameter choices each i-process finishes within the time for $2^i/8$ weighted insertions into the corresponding i-path, assuming $n \geq 2$. The parameter choices are: error* $= 11\,\alpha 2^{i-1}/\log n$,

$\ell = e' \cdot i \cdot 2^i$ *(measured in basic steps),* $\lambda = d \log n$*,* *p-inc* $=$ *atom*$/(4d \log n)$*,* *atom* $= 44\alpha\, d\, 2^{i-1}$*,* $d = 2/\alpha$*,* $e' = \lceil \frac{e}{44} \rceil \cdot 44$*, and* $\alpha = 1/\left[4(154 + e')\right]$*.*

Proof. We note that for these parameter values, ℓ is an integer multiple of *atom*, *atom* $\geq 4\lambda \max\{error, p\text{-}inc\}$, and so Lemma 6.15 applies. Thus the process priorities are always bounded by $11\alpha\, 2^{i-1}/\log n + 11\alpha\, 2^{i-1}/\log n + 1/(d \log n)\left(44\alpha\, d\, 2^{i-1} + e' \cdot i \cdot 2^i\right) + 4\left(11\alpha\, 2^{i-1}/\log n + 11\alpha\, 2^{i-1}/\log n\right)\log n \leq \alpha\, 2^{i-1}\left(66/\log n + e' \cdot i/\log n + 88\right) \leq 2^i/[8 \cdot 4\,(154 + e')\,\alpha] = 2^i/8$, assuming $n \geq 2$. □

It remains to show that $O(1)$ work per weight increase of $4/\log^3 n$ suffices.

Lemma 6.21. *It suffices to perform* $O(1)$ *work per weight increase of* $4/\log^3 n$ *to ensure that in the period in which an* i*-process performs atom basic steps the overall weight and hence priority increase by at most p-inc.*

Proof. These atom basic steps are performed over a period of $\Theta(atom \log^2 n)$ insertions assuming $O(1)$ work per insertion. (Recall, we cycle among the i-processes for $i = 0, 1, \ldots, \log n$, in round robin order, performing one basic step on an i-process for each i, and each basic step takes $O(\log n)$ work.) Since each insertion causes a weight increase of at most $4/\log^3 n$, the resulting weight gain is $O(atom/\log n)$. Notice that the more work per insertion, the fewer insertions needed to complete the *atom* basic steps and the smaller the weight increase. Thus with a large enough $O(1)$ work per insertion, the result holds. □

We have shown:

Theorem 6.22. *The implicit tree* T_1 *described above can be maintained in* $O(1)$ *work per insertion, assuming each insertion results in a weight increase of* 0 *or* $4/\log^3 n$*, and each insertion adds at most one weight* 0 *node* v *to* T_1*, and further that such a node* v *is adjacent only to weight 1 nodes. Further, LCA queries on* T_1 *can be answered in* $O(1)$ *time.*

7 The $O(1)$ Worst Case Algorithm

The tree T on which LCA queries are being performed is made binary, using a standard binarization. More specifically, a node v with d children is represented by d copies of v forming a chain of right children. The actual children of v will be stored as the left children of this chain of nodes. Note that if n items are inserted in an initially empty tree the binarized tree will contain at least n nodes and at most $2n - 1$. As a result, an insertion may entail the addition of two nodes to the binarized tree, called T henceforth. To simplify the discussion, from now on we term a node addition to T an insertion, understanding that a real insertion may induce two insertions into T.

As already noted, T is kept partitioned in at most $4n/\log^3 n$ subtrees, called partitioning subtrees, each of size at most $\log^3 n/4$ (strictly, $\lfloor(\log n)/4\rfloor^3$). We assume that n lies in the range $[n, 2n)$; Section 8 explains how to handle n increasing to $2n$ or beyond. We create a tree T_1 which contains the root of each subtree in the partition of T. These subtrees are chosen so that the LCA of the roots of any two partitioning subtrees is itself the root of a partitioning subtree. A node v in T_1 is the parent of node w in T_1 if v is the nearest ancestor of w in T such that v is in T_1. Clearly, as T is binary, so is T_1.

When a partitioning subtree grows too large it is split, causing the addition of one or two nodes to T_1 (two nodes may be needed to maintain the LCA property on subtree roots). But, as we will see, a newly created partitioning subtree once initiated, is itself partitioned only following $\Theta(\log^3 n)$ insertions into itself.

A partition of partitioning subtree S, rooted at node v, proceeds as follows. Once initiated, within $\frac{1}{4}\log^3 n$ further insertions into S it determines the root(s) of the new subtrees. If then inserts one of the news roots in T_1 as a child of v, giving it weight 0. Over the next $\frac{1}{4}\log^3 n$ insertions into S the weight of the new root is increased in increments of $4/\log^3 n$ until its actual weight is 1. Within the next $\frac{1}{8}\log^3 n$ insertions into S, we ensure v's recorded weight becomes 1, as follows. Instead of following the previously stated rule for giving priorities to weight update tasks, once the actual weight of a node reaches 1, on each insertion to subtree S, we continue incrementing its priority by $4/\log^3 n$. To bring the new root's recorded weight to 1 may need the completion of one run of its weight increase task and a full second run of the task. We have ensured this occurs within a weight increase of $2\alpha/\log n$, so it suffices that $2\alpha/\log n \leq \frac{1}{8}\log^3 n \cdot 4/\log^3 n$, and $\alpha \leq 1/4$ suffices for $n \geq 2$. The second new root, if any, is then inserted in the same way. Note that this ensures that any node in T_1 of weight less than 1 is adjacent only to nodes of weight 1, and nodes of weight 1 have at most one child of weight less than 1.

To answer a query LCA(u, v) we first determined if u and v are in different partitioning subtrees, by finding, in $O(1)$ time, the roots r_u and r_v of their respective partitioning subtrees. If $r_u \neq r_v$, we compute LCA (r_u, r_v) on T_1 in $O(1)$ time as previously described (see Theorem 6.22). Otherwise, the query is handled recursively.

To support queries on the partitioning subtrees, they are partitioned in turn into subtrees of size at most $4\log\log^3 n$ [8]. For each partitioning subtree S of T we maintain a tree S_1 comprising the roots of S's partitioning subtrees. Updates are performed using our previous algorithm, i.e. with $O(\log\log^3 n)$ work over $\Theta(\log\log^3 n)$ insertions. Queries are performed as in the previous paragraph. It is helpful to let T_2 denote the union of all the S_1 trees.

The recursion bottoms out at the partitioning subtrees of size $O(\log\log^3 n)$ for, as we will see, there are $o(n)$ distinct partitioning trees of this size, and their updating can be done via table lookup in $O(1)$ time per insertion, as can LCA queries. The requisite tables use $o(n)$ space.

Details of the Data Structures.

Each node in T keeps a pointer to its newest ancestor in T_2, the root of the size $O(\log\log^3 n)$ partitioning subtree to which it belongs. Similarly, each node in T_2 keeps a pointer to its nearest ancestor in T_1, the root of the size $O(\log^3 n)$ partitioning subtrees to which it belongs. On an insertion, the weight of the appropriate nodes in T_2 and T_1 are incremented in $O(1)$ time, using the above pointers.

Definition. The size of a partitioning subtree is the sum of the weights of the nodes it contains.

The size of partitioning subtrees are recorded with their roots. On an insertion, the up to two subtree sizes that change are incremented (by $4/\log\log^3 n$ and $4/\log^3 n$, respectively); these sizes are stored at the subtrees' roots.

Additional data is needed to support the splitting of the partitioning subtrees. We begin by describing what is needed for splitting the size $O(\log\log^3 n)$ partitioning subtrees. In addition to storing the subtrees themselves, we keep a table of all possible trees, represented canonically. Using the canonical representation, in $O(1)$ time we will be able to answer LCA queries and to determine the new canonical tree resulting from an insertion. Finally, by linking the nodes of the actual tree to those of the corresponding canonical tree, we will be able to translate query answers on canonical tree to answers on the actual tree in $O(1)$ time.

The following information is maintained for each actual tree S.

[8] $\log\log^3 n$ is our notation for $(\log\log n)^3$.

1. For each node v in S, a distinct label, denoted $label(v)$ in the range $1 \ldots 4 \log\log^3 n$. In addition, the up to two edges going to children outside S are also recorded. (The structure of S along with the associated labels provides the appropriate canonical labelled tree used to answer LCA queries on S.)

2. An array $\ell_to_n(S)$ storing, for each label in the range $1 \ldots 4 \log\log^3 n$, the node in S corresponding to this label, if any. This inverse map is used to convert the LCA obtained using lookup tables on the canonical tree from a label to an actual node (for the canonical tree nodes are named by their labels).

3. The name $name(S)$ of the labelled canonical tree associated with S, the root $root(S)$, of S, along with a pointer $pointer(v)$ from each node v in S to the location storing $\ell_to_n(S)$, $name(S)$, $root(S)$, $size(S)$, and $flag(S)$. The role of $flag(S)$ is explained next.

4. Actually, two copies of $label(v)$ and $pointer(v)$ are maintained for each node v in S. One of these copies will be "old" and the other "current". This will be indicated by the $flag(S)$ field above. The $flag(S)$ field pointed to by the "old" $pointer(v)$ will be set to 0 while that pointed to by the "current" $pointer(v)$ will be set to 1.

5. In addition to the above, there is a static table for each labelled tree of size at most $4 \log\log^3 n$ supporting the following queries: given two labels in the tree, return the label of the LCA, and given a new label (corresponding to a newly inserted node) and the label(s) corresponding to the node(s) at the insertion site, return the name of the resulting labelled tree. Note that labels for nodes in S are allocated in sequential order of insertion.

   Since there are $O(2^{8 \log\log^3 n} (4 \log\log^3 n)^{4 \log\log^3 n})$ labelled binary trees of size at most $4 \log\log^3 n$ with labels in the range $1 \ldots 4 \log\log^3 n$, the total space occupied by the above tables is $O(n)$. These tables can also be built in $O(n)$ time.

Processing Insertions. Each insertion will do $O(1)$ work at each of the 3 levels. This will result in $\Theta(\log^3 n)$ work being available for each insertion into T_1 and $\Theta(\log\log^3 n)$ work for each insertion into T_2.

Insertions into T will require the following actions. First, the insertion into the appropriate size $O(\log\log^3 n)$ partitioning subtree S of T rooted at a node in T_2 is made. This is done using a constant time table lookup to calculate the name of the new subtree after insertion. Second, if S reaches size $3(\log\log^3 n)$ then it is partitioned into two or three subtrees, each of size at most $3 \log\log^3 n$, over the next $\log\log^3 n$ insertions to S.

An insertion of node u into T is processed as follows.

Let v be the parent of u in T. u is viewed as being inserted into the partitioning subtrees S_b and S_a containing v, of sizes $O(\log\log^3 n)$ and $O(\log^3 n)$ and rooted in T_2 and T_1, respectively. On following $pointer(v)$, $\ell_to_n(S_b)$, $name(S_b)$, and $size(S_b)$ are readily updated in $O(1)$ time (using table lookup for $name(S_b)$). If $size(S_b)$ reaches $3 \log\log^3 n$, a split of S_b is initiated. It is carried out as described below. $O(1)$ work is then performed on the tasks associated with the trees T_1 and T_2.

Splitting $S = S_b$. The first step is to find a splitting location that divides the tree into two pieces each of size at least $\log\log^3 n$. This can be done by depth first search in $O(\log\log^3 n)$ time, or by table lookup in $O(1)$ time. To ensure that the new trees have at most two external children each, we find the LCAs of the new roots and the up to two external children; if one of these LCAs is not a new root, it is also introduced

as a third root. The one or two new roots are added to tree T_2 with the already explained timing (it suffices to carry out the depth first search within $\frac{1}{4}\log\log^3 n$ insertions). To simplify the notation, we continue to suppose that only two new trees are created; the changes if there are three new trees are evident.

The new roots define S_1 and S_2, the trees that S_b is split into. Next, $root(S_1)$, $\ell_to_n(S_1)$, $name(S_1)$, $size(S_1)$, $root(S_2)$, $\ell_to_n(S_2)$, $size(S_2)$, and $name(S_2)$ are computed in the obvious way in $O(\log\log^3 n)$ time (e.g. by traversing each of S_1 and S_2 in turn and "inserting" their nodes one by one). Then for each node w the "old" $label(w)$ and $pointer(w)$ are updated to be in accordance with S_1 or S_2, whichever contains v, also in $O(\log\log^3 n)$ time.

Note that all this while, the "current" $label(v)$, $pointer(v)$, $name(S)$, $root(S)$, and $\ell_to_n(S)$ are used to answer LCA queries; further these structures are updated with each insertion that occurs even after the splitting process starts. Also note that after the splitting process starts, new insertions are neglected in constructing S_1 and S_2 and the associated fields $name(S_1)$, $root(S_1)$, $\ell_to_n(S_1)$, $name(S_2)$, $root(S_2)$, $\ell_to_n(S_2)$. This is easily implemented by putting a time-stamp on each inserted node and ignoring nodes which are time-stamped later than the start of the splitting process. These insertions are just queued up and performed on S_1 or S_2 as appropriate after they have been constructed. When all $\log\log^3 n$ insertions have been performed, $flag(S_1)$ and $flag(S_2)$ are set and $flag(S)$ is reset so that for each v in S_1 and S_2 the "old" $label(v)$ and $pointer(v)$ become "current" and vice versa; this takes $O(1)$ time. Note that S_1 and S_2 each have size at most $3\log\log^3 n$ at this point.

Splitting algorithm for a size $O(\log^3 n)$ partitioning subtree S_a. The splitting algorithm on S_a begins when its size reaches $3\log^3 n$ and is similar to the previous splitting algorithm but is done without table lookup. For each partitioning S_a the following information is maintained, in addition to the data structure storing the tree itself.

1. For each node v, $code(v)$, along with the annotations, and $number(v)$.
2. The structure $cpath(S_a)$ storing centroid paths and associated information for paths in S_a.
3. For each node v, a pointer $pointer(v)$ to the location storing $cpath(S_a)$, $root(S_a)$, and $flag(S_a)$.
4. Two copies of $code(v)$, $number(v)$, $pointer(v)$, maintained as before. One of these copies will be "old" and the other 'current". The appropriate $flag(S_a)$ indicates which of these copies is current.

The algorithm proceeds as before. First, the location of the split is determined in $O(\log^3 n)$ time using depth-first search. This splits S_a into two pieces each of size at least $\log^3 n$, thereby defining S_1 and S_2. Then the data structures for S_1 and S_2 are computed, the backlog of insertions is applied, and finally the appropriate flats are set. This all takes $O(\log^3 n)$ time.

We have shown:

Lemma 7.1. *There is a data structure for trees of size in the range $[n, 2n)$ which answers LCA queries in $O(1)$ time and performs insertions to the tree in $O(1)$ time.*

Remark 7.2. *Although space is not freed when subtrees are split the space used is still linear. For each split of a size s tree, using space $\Theta(s)$, only happens after $\Theta(s)$ insertions.*

8 Handling Deletions and Changing Values of n

We note that the need for the limited range in Lemma 7.1 arises for two reasons: first, the construction in Section 7 requires a fixed value of $\log^3 n$ (and of $\log\log^3 n$), and second the basic algorithm takes $O(\log^3 n)$ time per update, and here the $\log^3 n$ is not fixed. But we could readily change the range to $[n, 2^d n)$ for any fixed n by replacing $\log^3 n$ by $(\log n + d)^3$ in Section 7. As we will see, $d = 3$ suffices.

We do not perform deletions explicitly. Instead, deleted items will just be marked as such, or rather the topmost copy of the corresponding node will be so marked. Thus the size of the current tree would include the count of both deleted and non-deleted nodes. We will periodically rebuild the data structure from scratch, in the background, so as to maintain the following invariant.

Invariant 3. The insertion count, the number of items in the data structure plus the number of items marked deleted, lies in the range $[4 \cdot 2^i, 32 \cdot 2^i]$; the actual number of items in the data structure lies in the range $[6 \cdot 2^i, 32 \cdot 2^i]$. In addition, the number of deleted items is at most $3 \cdot 2^i$. Further, immediately after being rebuilt the insertion count is in the range $[8 \cdot 2^i, 31 \cdot 2^i]$, the actual count in the range $[7 \cdot 2^i, 31 \cdot 2^i]$, and the deletion count is at most 2^{i+1}.

Invariant 3 implies that the number of nodes in the tree lies in the range $[4 \cdot 2^i, 64 \cdot 2^i)$, and thus $d = \log\ 64/4 = 4$ suffices.

Let the actual size of the data structure denote the number of undeleted items. If the insertion count reaches $31 \cdot 2^i$, the data structure is rebuilt in the range $[8 \cdot 2^i, 64 \cdot 2^i]$; if the actual size decreases to $7n$, then it is rebuilt in the range $[2 \cdot 2^i, 16 \cdot 2^i]$; if neither of these apply but the number of items marked deleted reaches 2^{i+1}, then it is rebuilt either in the range $[4 \cdot 2^i, 32 \cdot 2^i]$ if the actual size is at least $8n$, and in the range $[2 \cdot 2^i, 16 \cdot 2^i]$ otherwise. The rebuilding is completed within 2^i insertions; this takes $O(1)$ time per insertion. It is readily checked that Invariant 3 continues to hold following the rebuilding.

An easy way to perform the rebuilding is to traverse the current tree determining the undeleted items, and then inserting them in the new tree, one by one, using the appropriate value for $\log^3 n$. Of course, new insertions and deletions are performed on the current tree and queued so they can performed on the new tree.

We have shown:

Theorem 8.1. *There is a linear time data structure that supports LCA queries on a tree undergoing insertions and deletions such that each update and query can be performed in worst case $O(1)$ time.*

9 Acknowledgements

We thank the referees for urging us to seek a less complex solution which led us to the present version of the data structure.

References

[BF00] M. Bender, M. Farach-Colton. *The LCA Problem Revisited.* Proceedings of Latin American Theoretical Informatics, 2000, pp. 88–94.

[BV94] O. Berkman, U. Vishkin. *Finding Level Ancestors in Trees.* Journal of Computer and System Sciences, 48, 2, 1994, pp. 214–230.

[CH97] R. Cole, R. Hariharan. *Simpler Faster Approximate String Matching.* Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 463–472.

[DS87] P. Dietz, D. Sleator. *Two Algorithms for Maintaining Order in a List.* Proceedings of the 19th ACM Symposium on Theory of Computing, 1987, pp. 365–371.

[F99] M. Farach-Colton. *Private Communication*, 1999.

[Ga90] H. Gabow. *Data Structures for Weighted Matching and Nearest Common Ancestors with Linking.* Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 434–443.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997, pp. 196–207.

[HT84] D. Harel, R. Tarjan. *Fast Algorithms for Finding Nearest Common Ancestors.* SIAM Journal on Computing, 13, 2, 1984, pp. 338-355.

[LV86] G. Landau, U. Vishkin. *Fast parallel and serial approximate string matching.* Journal of Algorithms, 10, 1989, pp. 157-169.

[M76] E. McCreight. *A Space-economical Suffix Tree Construction Algorithm.* Journal of the Association of Computing Machinery, 23, 1976, pp. 262–272.

[Meh77] K. Mehlhorn. *A Best Possible Bound for the Weighted Path Length of Binary Search Trees.* SIAM Journal on Computing, 6, 2, 1977, pp. 235–239.

[SV88] B. Schieber, U. Vishkin. *On Finding Lowest Common Ancestors: Simplification and Parallelization.* SIAM Journal on Computing, 17, 6, 1988, pp. 1253–1262.

[Ts84] A. Tsakalidis. *Maintaining Order in a Generalized Link List.* Acta Informatica, 21, 1, 1984, pp. 101–112.

[We92] J. Westbrook. *Fast Incremental Planarity Searching.* Proceedings of the 19th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, Springer-Verlag 623, 1992, pp. 342–353.

[W82] D. Willard. *Maintaining Dense Sequential Files in a Dynamic Environment.* Proceedings of the 24th ACM Symposium on Theory of Computing, 1992, pp. 114–121.