# 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.

## 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. Schieber and Vishkin [SV88] gave a simpler algorithm 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 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

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

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

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

and processing it for LCA queries. Extending these algorithms to maintain a database of strings supporting $k$-mismatch queries would require maintaining LCAs dynamically. 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. In addition, our focus 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 $O(\log^4 n)$ 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 $O(\log^4 n)$ 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.

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 which maintains order in a list under insertions and deletions.[1]

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

[1] 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].

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.

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. So queries will result in not just "slightly" wrong answers but "arbitrarily" wrong answers.

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 yet result in codes which give only "slightly" wrong answers to queries. These slightly wrong answers are easy to correct. This is obtained by a non-trivial scheduling procedure coupled with an analysis which bounds the total sizes of nested processes.

## 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 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.

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 Section 5. In Section 6, we describe our $O(\log^4 n)$ worst-case time algorithm. Section 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 $y$. The *size* of a node $y$ is just the size of $T_y$. 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 $\pi$ is said to be an ancestor of a node $x$ if $\pi$ contains an ancestor of $x$. A centroid path $\pi$ is said to be an ancestor of another path $\pi'$ if $\pi$ is an ancestor of the head of $\pi'$. A centroid path $\pi$ is a child of another path $\pi'$ if the head of $\pi$ is a child of a node on $\pi'$.

The *Least Common Centroid Path (LCCP)* of two nodes is the path containing their LCA. An *off-path node* with respect to a particular centroid path $\pi$ is a node not on $\pi$ whose parent is on $\pi$. The *Branching Pair (BP)* of two nodes $x, y$ is the pair of nodes $x', y'$ on the LCCP which are the least 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 on the same centroid path and $x$ is an 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 $code(y)$[2]. Note that the nodes in the BP need not be distinct (see Fig.1). The node with the smaller number in the BP is the desired LCA.

## 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 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 $\pi$, a bit string, called $separator(\pi)$. These strings have the following property. For each centroid path $\pi$, the separator strings assigned to children centroid paths of $\pi$ form a prefix-free set. The length of $separator(\pi)$ is $O(\log \frac{|T_y|}{|T_x|})$, where $x$ is the head of $\pi$ and $y$ is the head of the centroid path containing the parent of $x$.

$code(x)$ is a concatenation of the separator strings assigned to ancestor centroid paths of $x$ (the path containing $x$ inclusive) in order of increasing distance from the root. It is easy to show that the length of the code is $O(\log n)$. Each separator string in $code(x)$ is tagged with the name of the corresponding centroid path. $number(x), separator(\pi), code(x)$ can all be assigned easily in $O(n)$ time.

The LCCP of nodes $x$ and $y$ is determined from $code(x)$ and $code(y)$ in $O(1)$ time as follows. 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

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

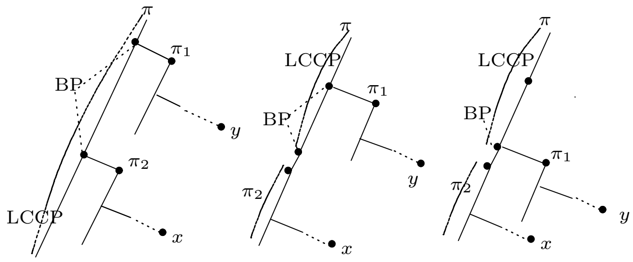


Figure 1: LCCP and BP.

children paths of the LCCP which are ancestors of $x$ and $y$, respectively, necessarily differ in some bit. The parents of the heads of these two paths will give the BP (see Fig.1). The path containing this BP is the $LCCP$. Therefore, BP and LCCP are easy to determine from the leftmost bit of difference between $code(x)$ and $code(y)$. 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.

There is one issue which needs clarification. Note that tagging each separator string in $code(x)$ with the name of the corresponding path 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. 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 is easily recovered in $O(1)$ time, given their codes, using look-up tables of $O(n)$ size. In this paper, we will assume the framework of the former solution.

# 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 $number(x)$, $separator(\pi)$ 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[3] 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 sub-interval of the interval for any ancestor

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

centroid path. In addition, the intervals for the centroid paths which are children of a path $\pi$ 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 $\pi$ is inserted, a new interval within the interval for $\pi$ and to the right of all other intervals for children of $\pi$ 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 $O(\log^4 n)$ 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[4]. 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.

## 6.1 Updating Centroid Paths

When an insertion is made, each of the $O(\log n)$ ancestor centroid paths could shift down by one node as shown in Fig.2, Case 1. New single node centroid paths could begin as well, as shown in Fig.2, Case 2.

Or aim is to associate a particular path with a particular size range. Consider a particular centroid path $\pi$ with head $x$ and tail $y$. Let $2^i \leq |T_x| < 2^{i+1}$. The nodes in this path will change as insertions occur, but will maintain the invariant that their sizes will be in the range $[2^i, 2^{i+1})$; we refer to this range as *size class* $i$. The insertion of a single leaf or internal node $z$ can cause only the following changes to $\pi$.

1. If node $z$ is not a descendant of $x$ then $\pi$ does not change (except as in (5) below).
2. If node $z$ is inserted between two nodes in $\pi$ then it is added to $\pi$ at the appropriate place.

[4]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.

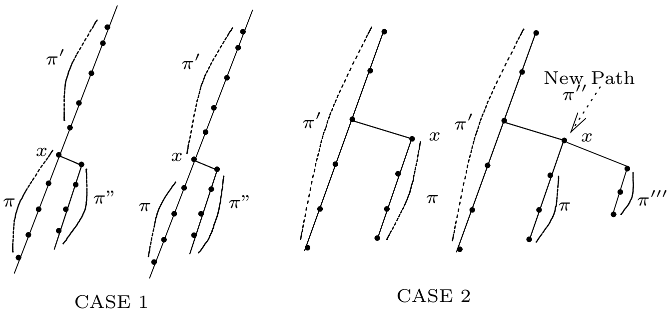


Figure 2: Changing Centroid Paths.

3. If node $z$ is a descendant of $x$ and and $|T_x|$ becomes at least $2^{i+1}$ after the insertion, then $x$ is removed from $\pi$. In this case, it can be shown that $x$ is not the only node in $\pi$. The child of $x$ in $\pi$ becomes the new head of $\pi$. If the parent of $x$ has size less than $2^{i+2}$ after the insertion, then $x$ becomes the new tail node on the path containing its parent. Otherwise, if the parent of $x$ has size at least $2^{i+2}$ after the insertion, then $x$ becomes a new centroid path by itself.

4. If node $z$ is a descendant of $y$ and and $|T_{y'}|$ becomes at least $2^i$ after the insertion for some child $y'$ of $y$, $y'$ is removed from its centroid path and added to $\pi$; it becomes the tail node of $\pi$.

5. If node $z$ is inserted between $x$ and its parent $z'$, then there are three cases. If $2^i \leq |T_z| < 2^{i+1}$ then $z$ becomes the new head of $\pi$. If $|T_z| = 2^{i+1}$ and $|T_{z'}| < 2^{i+2}$ then $z$ becomes the tail node on the centroid path containing $z'$. And if $|T_z| = 2^{i+1}$ and $|T_{z'}| \geq 2^{i+2}$ then $z$ becomes a singleton centroid path by itself.

**Remarks.** Note that new centroid paths can be created by an insertion but existing centroid paths never vanish. Also note that if node $y$ moves from one centroid path to another on an insertion, the size class of the latter path is one larger than that of the former path. Further, if $z$ is a newly inserted node which forms a new singleton path by itself, then this path is in the size class which is one larger than that of the path containing the only child of $z$.

Upon an insertion, the above changes are easy to make in $O(\log n)$ time, since there are at most $\log n$ centroid paths that could be ancestors of $z$ and each can be updated in constant time.

## 6.2 Updating Separators

Separators and codes have to be updated to reflect the above changes in the centroid paths. The main change to the centroid paths above is that the head vertex of a path is removed and becomes either a new singleton centroid path or the tail vertex of the parent centroid path. Before we show how separators are updated in response to these changes, we need the following interpretation of separators in terms of intervals.

**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, starting from all 0s at the left end. We allow a subinterval of length $2^i$ to begin only at those integer points which have $i$ trailing 0s in their bits; with such a subinterval, we associate a bit string of length $k-i$ given by the leading $k-i$ bits of 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.

**Mapping Paths to Intervals.** With each centroid path $\pi$, we maintain an interval $int_\pi$ of length $2^{ic}$, where $2^i \leq |T_{head(\pi)}| < 2^{i+1}$ and $c$ is a constant to be determined. Intervals associated with paths which are children of $\pi$ are subintervals of $int_\pi$ and are mutually disjoint. It is important to note that the separator bit string for a centroid path $\pi'$ is assigned relative to the interval for its parent path $\pi$, and therefore, when a new interval is assigned to $\pi$ or any of its ancestor paths, the separator code for $\pi'$ remains unchanged.

Note that constraints on $c$ will be imposed by Lemma 6.1 and Lemma 6.2; setting $c \geq 6$ will satisfy these constraints.

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

1. When the head node $x$ is removed from path $\pi$ and made the tail node of the parent path $\pi'$ of $\pi$, the centroid path $\pi''$ whose head was the off-path (with respect to $\pi$) child of $x$ must be assigned a new interval. The interval for this path was earlier nested within $int_\pi$ and $int_{\pi'}$. Now, this interval must be disjoint from the $int_\pi$ but still nested within $int_{\pi'}$. The process which does this is called *Reassign*($x$).

2. When the head node $x$ of $\pi$ is removed and becomes a new centroid path $\pi''$, a new interval has to be assigned to this path. This interval $int_{\pi''}$ must be nested within the interval $int_{\pi'}$ for the parent path $\pi'$ of $\pi$; further, $int_\pi$ 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 $\pi'''$ containing the off-path child of $x$ (with respect to $\pi$) 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 that will be explained later in Section 6.3.

3. When the new node $z$ is inserted between head node $x$ of $\pi$ and its parent and forms a singleton centroid path $\pi''$ by itself, we treat $z$ as if it were first added to $\pi$ as its head node and then immediately moved out to form a new singleton path; the updates made to intervals then are as in the previous case, with $z$ replacing $x$.

Thus as time proceeds, sub-intervals 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 $\pi$ within $int_{\pi'}$, where $\pi'$ is the parent path of $\pi$, 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 subintervals of $int_{\pi'}$ assigned so far (deleted or not) in that half. If such a slot exists, then it is easy to assign this subinterval in constant time. Note that a particular insertion initiates only a constant number of *Reassign* or *Assign* processes at each ancestor centroid path of the inserted node.

We need to ensure that *Assign*() and *Reassign*() will always find an empty slot as above to assign the new interval for $\pi$. Lemma 6.1 shows that this would always be true if non-deleted subintervals do not leave large gaps in between. However, large gaps could build up as as subintervals enter and leave intervals. Therefore, we need a background process which will start removing deleted intervals and compacting non-deleted subintervals within an interval, once the interval reaches a certain fraction of its capacity. This process is described next. It will remove deleted subintervals and compact undeleted intervals in one half of an interval leaving the other half free for further insertions, toggling alternately between the two halves.

**Lemma 6.1** *All subintervals corresponding to children paths of path $\pi$ at any given instant can fit into the first quadrant of $int_\pi$.*

**Proof.** Let $2^{ic}$ be the length of $int_\pi$. Then any subinterval corresponding to a child path of $\pi$ has length at most $2^{(i-1)c}$. The total sizes of the heads of children paths of $\pi$ is at most $2^{i+1}$. Therefore, using simple calculus, the total length of all such subintervals is at most $4.2^{(i-1)c}$. When packed, the sum of the gaps between these subintervals is at most the sum of the lengths of the subintervals. The lemma now follows since $8.2^{(i-1)c} < 2^{ic}/4$, for $c \geq 5$. $\Box$

**The Compacting Process.** When the half into which intervals are currently being inserted becomes full, without loss of generality, the left half, the compaction process in the left half begins. This compaction moves through the subintervals in the left half from left to right, removing deleted intervals, and reassigning undeleted intervals at the leftmost possible slot in the right half. Insertions subsequent to the beginning of the compaction process will also be made in the right half until it becomes full. We will show that the compaction process in the left half will have finished by the time this happens. At that point, the compaction process will become active in the right half and insertions will be performed in the left half. We call the above compaction process *Compact*$(\pi)$. Note that a single insertion can initiate a compaction process at each of its ancestor centroid paths.

Thus, there are three kinds of processes which could be initiated when a node is inserted: *Assign*$()$, *Reassign*$()$ and *Compact*$()$. Each process is expensive and takes time at least proportional to the size of the subtree in which it modifies codes (this updating of codes will be elaborated upon shortly in Section 6.3). Thus these processes have to be performed, a poly-logarithmic number of operations at a time, over future insertions. Therefore, at any instant, a number of such processes could be active.

## 6.3 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* and *Compact* processes (as described in Section 6.2).

First, consider an *Assign*$(\pi')$ process initiated by some node $x$ moving out of $\pi$ to form singleton path $\pi'$ (recall from Section 6.2 that $x$ could be a newly inserted node, but if it forms a new singleton path then we treat it as coming out of $\pi$). *Assign*$(\pi')$ must assign a new interval to $\pi'$, and a new interval to $\pi$ nested within the interval for $\pi'$. It must then change the codes at all nodes rooted at $T_x$ to reflect the change to the above two intervals. This is done as follows. The old separator string for $\pi$ in the codes at nodes in $T_x$ (including $x$ itself) will be updated to the new separator string for $\pi$. In addition, the separator string for $\pi'$ will be incorporated into the codes at all nodes in $T_x$. Thus note that the effect of *Assign*$(\pi')$ on the codes in $T_x$ corresponds to the creation of a new *empty path* as the parent path of $\pi$; currently, the code at $x$ still indicates it is in $\pi$ and not in $\pi'$. *Reassign*$(x)$ will perform the changes required in actually moving $x$ to this empty path.

Next, consider a *Reassign*($x$) process. Such a process initiated by the move of $x$ from $\pi$ to $\pi'$ must remove the separator string for $\pi$ 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 $\pi$. 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.

Finally, a *Compact*($\pi$) process must assign new separator codes to all children paths of $\pi$ and update the codes at all nodes in the subtree rooted at the head of $\pi$ (other than the codes for nodes on $\pi$ itself).

In addition to updating the codes, the annotations on the codes labelling each separator string with the corresponding centroid path have to be updated in each of the above cases. This can be done in $O(\log n)$ time per node. This work is done over future insertions, a poly-logarithmic amount per insertion, in a manner yet to be described.

We have now described the overall structure of the algorithm. On an insertion, $O(\log n)$ work is done in updating the ancestor centroid paths of the inserted node and initiating a constant number of *Assign*, *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 *Reassign* and *Compact* processes will be described in Section 6.6. This part of the algorithm is what leads to the $O(\log^4 n)$ bound. Before describing this work, we describe some crucial invariants which we will maintain.

## 6.4 Invariants Maintained

To ensure that queries are answered correctly at every instant, we schedule the various processes so that the 3 invariants described below are maintained.

Invariant 1 states that an *Assign*() or *Reassign*() process should finish before the situation which caused the initiation of this process changes dramatically. Invariant 2 states that there is at most one process *Compact*($\pi$) at any given instant, for each path $\pi$. Invariant 3 states that, for any node $z$, there is at most one process which must modify all codes in $T_z$ but has changed codes at some but not all the nodes in $T_z$, at any instant.

Note that Invariants 1 and 2 ensure that the codes at any instant reflect situations which occurred not too far back in the past. Clearly, this is desirable in order to answer queries in $O(1)$ time. Invariant 3 ensures that the codes at any two nodes $x, y$ in the subtree $T_z$ reflect situations which occurred with a not too large time interval in between. The absence of this invariant would allow an arbitrary number of processes which must modify both $code(x), code(y)$, but have modified exactly one of the two codes. If this indeed happens, then it is not clear how to answer queries as arbitrarily many initial bits of difference between $code(x), code(y)$ would be unrelated to the LCCP of $x, y$.

**Definitions.** An insertion is said to be in the subtree rooted at the head of a path $\pi$ if the inserted node is in the subtree rooted at the head of $\pi$ at the time of insertion. We say that path $\pi$ *occurs* in $code(x)$ if the separator string for $\pi$ (which is not necessarily the latest separator string for $\pi$) appears in $code(x)$.

### 6.4.1 Invariant 1

**Invariant 1.** Process *Reassign*($y$) initiated when node $y$ switches to path $\pi$, will complete before $y$ leaves $\pi$. Further, process *Assign*($\pi$) initiated when node $y$ switches to a new singleton path $\pi$ will complete before either some node from $\pi$ moves out of $\pi$ to form a new singleton path, or a singleton path formed by a newly inserted node becomes the parent of $\pi$. Finally, a process $Reassign(x)$ initiated by the move of $x$ to $\pi$ will finish before process $Reassign(y)$ initiated by the move of $y$ to $\pi$ is activated, where $x$ is closer to the root of $T$ than $y$.

If path $\pi$ is in the size range $[2^i, 2^{i+1})$ then note that there will be at least $2^i$ insertions in the subtree rooted at the head of $\pi$ before any of the events mentioned in the first two sentences of Invariant 1 occur; these insertions will be useful in completing the above processes.

Below, we state a weaker version of Invariant 1, which we call Invariant 1'. Invariant 1' will be implied by Invariant 1, i.e., if the latter holds then the former holds as well. We will need this weaker invariant when we describe the $O(1)$ time algorithm; Invariant 1 need not hold for this algorithm. However, to avoid reproving results, we will work with Invariant 1' (instead of Invariant 1) for the current $O(\log^4 n)$ time algorithm as well.

**Invariant 1'.** The following three facts will hold.

**(a)** Consider the processes initiated when node $y$ switches to path $\pi$ (if $\pi$ is an existing path then there is a single *Reassign* process and if $\pi$ is a new path there is an *Assign* and a *Reassign* process). $y$ will be either in $\pi$ or in the parent path $\pi'$ of $\pi$ when these processes finish. Further, if these processes finish after $y$ has moved to $\pi'$, then the processes initiated by the move of $y$ to $\pi'$ will also finish before $y$ leaves $\pi'$. In addition, the second set of processes will become active only after the first set of processes finish.

**(b)** Consider the $Assign, Reassign$ processes initiated when a node $y$ switches to a new singleton path $\pi$. These processes will finish before any one of the following events happens.

1. Some node currently in $\pi$ changes paths twice in the future.
2. A new singleton path $\pi'$ becomes the parent of $\pi$ in the future, and subsequently some node moves out of $\pi'$ or a new singleton path becomes the parent of $\pi'$.

In addition, if the subsequent event in case 2 above happens first and the above processes finish only after $\pi'$ comes into existence, then the $Assign, Reassign$ processes initiated by the creation of $\pi'$ will also finish before the subsequent event occurs; further, these processes will become active only after the first set of processes have finished.

**(c)** A process $Reassign(x)$ initiated by the move of $x$ to $\pi$ will finish before process $Reassign(y)$ initiated by the move of $y$ to $\pi$ is activated, where $x$ is closer to the root of $T$ than $y$.

Next, we describe Invariants 2 and 3. Before doing so, we mention that other than the lemmas which show that Invariants 1' and 2 hold for the $O(\log^4 n)$ time algorithm (namely, Lemmas 6.14, 6.15, 6.16, 6.17, and 6.5), all lemmas in the $O(\log^4 n)$ time algorithm will depend only on Invariants 1', 2 and 3. Therefore, they will continue to hold for the $O(1)$ time algorithm once Invariants 1', 2 and 3 are established. Further, the proof of Lemma 6.11 which shows Invariant 3 is easily seen to hold for the $O(1)$ time algorithm as well. The same is true for the proof of Lemma 6.5 which shows Invariant 1'(c). Thus, only Invariants 1'(a), 1'(b), and 2 will need reproving for the $O(1)$ time algorithm.

### 6.4.2 Invariant 2

**Invariant 2.** One initiation of $Compact(\pi)$ will finish before the next initiation. Further, $Assign(\pi)$ will finish before the first initiation of $Compact(\pi)$.

Lemma 6.2 will be useful in proving this invariant.

**Lemma 6.2** *Consider two successive initiations of* $Compact(\pi)$. *Let* $\pi$ *be in the size range* $[2^i, 2^{i+1})$. *Then, between the above two initiations, there must be at least* $2^{i+2}$ *insertions in the subtree rooted at the head of* $\pi$. *The same holds for the interval between the time that* $\pi$ *comes into existence and the first initiation of* $Compact(\pi)$.

**Proof.** Assume, without loss of generality, that the first compaction inserts into the right half of $int_\pi$.

Consider insertions into the right half. Suppose this half gets full and yet the number of nodes inserted in the subtree rooted at the head of $\pi$ is less than $2^{i+2}$. Then we show a contradiction as follows.

We count the length of all subintervals inserted into this half since the first initiation. Some of these might have been deleted by the time of the next initiation. The length of each interval that is inserted is at most $2^{(i-1)c}$. A particular subtree rooted at an off-path node with respect to $\pi$ could repeatedly delete and insert intervals of successively doubling length. Thus such a subtree of size (at the time of the second initiation) in the range $[2^j, 2^{j+1})$ could be responsible for a total subinterval length of $\sum_{k=0}^{j} 2^{ck} \leq \frac{2^{c(j+1)}-1}{2^c-1}$. The total length of the gaps between subintervals is at most the total length of the subintervals. Since the number of nodes inserted into the subtree rooted at the head of $\pi$ during the first compaction is less than $2^{i+2}$, the total length of the right half of $int_\pi$ occupied by deleted and undeleted subintervals inserted since the first initiation is at most $2.\sum_r \frac{2^{c(h_r+1)}-1}{2^c-1}$, where $\sum_r 2^{h_r} < 2^{i+2} + 2^{i+1} - 1$ and $h_r \leq i-1$. Here the first factor of 2 is for the gaps. This is at most $12.2.\frac{2^{ci}-1}{2^c-1} \leq 2^{ic}/2$, for $c \geq 6$. Thus the right half of $int_\pi$ cannot be full, a contradiction. □

### 6.4.3 Invariant 3

**Invariant 3.** For any pair of nodes $x, y$, there is at most one $Assign()$, $Reassign()$, or $Compact()$ procedure which must modify both $code(x), code(y)$ but has updated the code for exactly one of $x, y$.

We remark that Invariants 1' and 2 themselves are not hard to maintain. Similarly, Invariant 3 is easy to maintain using a simple blocking mechanism for processes. However, maintaining all three invariants simultaneously is non-trivial. This is because insertions in different portions of the tree may occur at different rates and unfinished processes will therefore be driven at different speeds by these insertions. In particular, a process could get blocked indefinitely. Our solution to this problem is to make a blocked process help the blocking process. This will be described in detail in Section 6.6.

Finally, note that Invariants 1', 2 and 3 will hold in our $O(1)$ time algorithm as well.

## 6.5 Some Details of Processes

Before proceeding, some details of $Assign()$, $Reassign()$, $Compact()$ processes need to be carefully examined. These details arise because a process is performed over a sequence of future insertions.

**Remarks on Insertions.** The code for a newly inserted node $x$ is determined as follows.

1. If $x$ is inserted as a child of a leaf node then it is first given the same code as its parent. This reflects the fact that $x$ is in the same path as its parent. Of course, this parent will now move out of this centroid path.

2. If $x$ is inserted as a leaf child of an internal node 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.

3. If $x$ is inserted as an internal node, then the code for $x$ is made identical to that for its child. This corresponds to putting $x$ is the same path as its child. Of course, it may happen that $x$ immediately moves out of this path, in which case a $Reassign(x)$ procedure will be responsible for changing its code.

Note that $code(x)$ constructed this way need not indicate the current path it is in, if $x$ is an internal node. If $x$ is currently in $\pi$ and $code(x)$ indeed indicates that $x$ is in some other path $\pi'$ then $Reassign(x)$ processes are initiated at all paths between $\pi$ and $\pi'$ (including $\pi$ but excluding $\pi'$); by Invariant 1', there are at most two such paths, including $\pi$. Thus, one can think of $x$ as having been inserted in $\pi'$ and not $\pi$, and having moved to $\pi$ subsequently, initiating a $Reassign(x)$ process with each change of path. Also note that if $\pi$ is a new singleton path formed by $x$ then the $Assign(\pi)$ procedure will of course be initiated.

Finally, note that subsequent to the insertion of $x$ and before the separator string for $\pi'$ is removed from $x$ by the above $Reassign(x)$ processes, a new leaf node $z$ may be inserted as an off-path child of $x$ with respect to $\pi$. As above, $code(z)$ will be constructed from $code(x)$, which currently records that $x$ is in $\pi'$. Therefore, the path $\pi''$ containing $z$ will be treated as a child of path $\pi'$ and not of $\pi$. The interval $int_{\pi''}$ will be assigned so that it nests within $int_{\pi'}$. $Reassign(x)$ will ultimately reassign $int_{\pi''}$ so that it nests within $int_{\pi}$ and is disjoint from $int_{\pi'}$.

**Remarks on** $Assign()/Reassign()$ **Processes.** Suppose a node $x$ moves from centroid path $\pi'$ to centroid path $\pi$ at some instant, causing initiation of a $Reassign(x)$ process. Let $y$ be the child of $x$ at this instant which is not on $\pi'$ and let $\pi''$ be the path containing $y$. Subsequently, suppose $y$ moves out of $\pi''$ to form a new centroid path $\pi'''$. $\pi'''$ now becomes the parent of $\pi''$ and an $Assign(\pi''')$ procedure is initiated. We consider two cases in turn.

First, suppose $Assign(\pi''')$ modifies the code for $y$ before $Reassign(x)$ has modified the code for $x$. $Assign(\pi''')$ will take the code for $x$ as the base. Since $code(x)$ indicates that $x$ is in $\pi'$, $int_{\pi'''}$ will be assigned so that it is nested within $int_{\pi'}$ and the separator string for $\pi'$ will be included in $code(y)$. At some subsequent point, $Reassign(x)$ will remove the separator string for $\pi'$ from both $code(x)$ and $code(y)$.

Second, suppose $Reassign(x)$ modifies the code for $x$ before $Assign(\pi''')$ modifies the code for $y$. The behaviour of $Assign(\pi''')$ is now as expected with $int_{\pi'''}$ being assigned so as to be nested within $int_{\pi}$. However, consider $Reassign(x)$. Suppose, it modifies $code(y)$ before $Assign(\pi''')$ does. $code(y)$ at this instant will indicate that it is still on $\pi''$. Then $\pi''$ will be reassigned a new interval within $int_{\pi}$ and the resulting separator string will be included in $code(y)$. As some later instant, $Assign(\pi''')$ will include the separator string for $\pi'''$ in $code(y)$, and replace the current separator string for $\pi''$ with a new separator string corresponding to an interval nested within $int_{\pi'''}$; the $Reassign(y)$ procedure initiated by $y$ moving to $\pi'''$ will ultimately remove the latter separator string.

**Remarks on** $Compact()$ **Processes.** Suppose the parent $x$ of the head of a path $\pi'$ has moved off path $\pi$ in the past but its code still reflects that it is in $\pi$. An initiation of $Compact(\pi)$ now cannot delete the interval for $\pi'$ which is nested within that of $\pi$. Thus, $\pi'$ will be treated as a child of $\pi$ if the code for $x$ states that it is still in $\pi$.

The above situations suggest that at each instant we have two views of the partition of nodes into centroid paths. The first view is the actual view in which nodes change paths immediately with each insertion. The second view is the view reflected by the codes at the various nodes; this view could lag behind the first, though not by too much, as stated by Invariant 1'. The first view is used to initiate the appropriate processes. Processes modify codes based on the second view, as illustrated in the situations above.

At this point, we state Fact 6.3. Note that this fact would not hold if, for instance, $Assign(\pi''')$ assigned an interval to $\pi'''$ nested within the interval for $\pi$ and incorporated the associated separator string into $code(y)$, in the first case above in the remarks on $Assign()/Reassign()$ processes. If $Assign(\pi''')$ indeed did so then the fact that $x$ had changed from $\pi'$ to $\pi$ would be reflected in $code(y)$ by a modification made by $Assign(\pi''')$ and not by $Reassign(x)$. In the rest of the paper, this fact will implicitly appear at several places in arguments of the following form: for instance, if a $Reassign(x)$ procedure initiated by $x$ moving off $\pi$ has not modified the code at descendant $y$ of $x$, then the separator string for $\pi$ is still present in $code(y)$ (provided, of course that $Assign(\pi)$ has already inserted this separator string into $code(y)$).

**Fact 6.3** *The change of a vertex $x$ from one centroid path to another is reflected in the code at some descendant node $y$ of $x$ only by the modifications performed by the process initiated by the change of paths at $x$.*

**Root of a Process.** Each process must modify codes at certain nodes in the tree. The node closest to the root of $T$ 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. As we will see in Section 6.6, a process is *activated* at some time instant subsequent to its initiation. Modifications to codes will be made by the process only after it is activated. We define the root for each process type in turn, based on the situation at the time of activation.

First, consider an $Assign(\pi)$ process. There are two cases. The first case is when a new singleton path $\pi'$ becomes the parent of $\pi$ before the moment of activation of the above process; in this case, the root of this process is the head node of $\pi'$ when $Assign(\pi)$ is activated. The second case is when the above does not happen; the root of this process is then the node closest to the root of $T$ which was ever on $\pi$.

We make a few remarks on the above definition. First, note that if some node in $\pi$ moves out of $\pi$ to an existing path before the activation of $Assign(\pi)$, then the first case cannot occur; this follows from Invariant 1' which says that $Assign(\pi)$ will finish before this node changes paths yet again, which it must to allow for the first case. Second, note that the root in either case is an ancestor (not necessarily strict) of all nodes which were ever on $\pi$. This is trivially true for the second case. For the first case, this follows from Invariant 1' which says that no node can leave $\pi'$ before $Assign(\pi)$ and $Assign(\pi')$ both finish. Third, from Invariant 1', the root of the process in either case is either in $\pi$ or in the parent of $\pi$. Finally, note that until the activation of $Assign(\pi)$, all nodes which were ever in $\pi$ or in $\pi'$ will have codes which indicate that they are in some descendant path of $\pi$; this justifies the definition of the root as above.

For a $Compact(\pi)$ process, we would like to define the root to be the topmost node $x$ amongst nodes in $\pi$, its parent, and its grandparent path whose code reflects that it is still in $\pi$. The restriction to $\pi$ and its parent and grandparent paths is given by Invariant 1' which says that nodes which have moved out of $\pi$ without the associated $Reassign()$ process finishing must be in either the parent or the grandparent path of

$\pi$. However, we have to make a small change to the above definition to account for situations in which such a node $x$ does not exist, for instance, when the *Reassign*() processes for nodes exiting $\pi$ finish but for those entering $\pi$ do not get started. To handle this situation, we define the root be the topmost node $x$ amongst nodes in $\pi$, its parent, and its grandparent path whose code reflects that it is still in $\pi$ or in a path of a yet smaller size class.

Finally, for a *Reassign*($x$) process, the root is always $x$. This process must change the codes at $x$ and in the subtree rooted at one of the children of $x$.

## 6.6 Scheduling Processes

Each centroid path $\pi$ maintains a list *list*($\pi$) of unfinished processes associated with this path. These include the processes *Assign*($\pi$) and *Compact*($\pi$), and the processes *Reassign*($x$), where node $x$ moved from some other path to path $\pi$ in the past. *Assign*($\pi$) and *Compact*($\pi$) processes have the highest priorities. Note that by Invariant 2, there will be at most one *Assign*() or *Compact*() process in *list*($\pi$) at any instant. *Reassign*($x$) processes are always ordered by increasing distance of $x$ from the root of $T$. A process is said to be *initiated* at the instant at which it enters this list. A process is said to be *active* when it is actually scheduled; once scheduled it remains active until completed. At most one process in *list*($\pi$) will be active at any instant. When an active process is completed, it is removed from *list*($\pi$).

We now describe how the various processes are performed so as to maintain Invariants 1, 2 and 3.

**Algorithm for Scheduling Processes.** On an insertion, $O(\log^3 n)$ work is done on processes in the list at each (not necessarily strict) ancestor path and at each of the children paths of each ancestor path which are in a size class one smaller than their respective parents.

Consider any such path $\pi$ on which a particular insertion does work. If there is an already active process in *list*($\pi$), then $O(\log^3 n)$ operations are performed on this process. Otherwise, if there is no such process, then the highest priority process in *list*($\pi$), if any, is made active and then $O(\log^3 n)$ operations are performed on this process. The total time taken on an insertion is thus $O(\log^4 n)$. The reason for our choice of $O(\log^3 n)$ will become clear in Lemma 6.12 in Section 6.8.

**Performing a Process.** We now describe what the act of performing one operation on a particular active process means. Note that, left to itself, each process would perform a preorder traversal on the appropriate subtree and update the codes at each node in this subtree. Thus each operation would involve changing the code and the path annotations at some node, or backtracking up a node for which this is already done. However, to maintain the invariants (recall that processes must sometimes help other processes), a process has to be performed differently.

Maintaining Invariant 3 implies the following. If a process *Assign*(), *Reassign*(), *Compact*() is working on the nodes of a subtree $T_x$ rooted at node $x$, then another process which must also modify codes at nodes in $T_x$ cannot work on the nodes in $T_x$ yet. This implies that the second process is blocked until the first process finishes. In this situation, the second process blocks itself and helps the first process complete, following which it itself becomes active. Note that helping the second process is essential to avoid indefinite blocking. Note also that there could be up to $O(\log n)$ levels of nesting with one process blocked by another, which, in turn, is blocked by another, and so on. To enforce the above rule, each process will mark the nodes it traverses on the way down, unmarking them on the way up. The mark will indicate the name of the process. At each instant, each process will also remember the last node it visited.

The following steps are executed when an operation is performed on a process $\mathcal{P}$. First, the next actual operation of $\mathcal{P}$ is determined. This operation involves either changing the code for a new node or backtracking up an already processed node. In the latter case, the operation is performed and the node is unmarked. In the former case, if the node whose code needs to be updated is not already marked by some other process, then the update is performed and the node marked. In both these cases, we say that this process is *unblocked.* It remains to consider the case when the next operation is to update the code of a new node and that node is already marked by another process. In this case, we consider the process *blocked.*

To tackle this case, we keep a *blocking list.* If process $\mathcal{A}$'s next node is marked by $\mathcal{B}$ then the blocking list node for $\mathcal{A}$ would point to $\mathcal{B}$. The last process $\mathcal{Q}$ in this list is always unblocked. The next operation for the last process is then performed; thus $\mathcal{P}$ helps $\mathcal{Q}$. Note that determining the last process in the blocking list might required traversing the list until the front is reached. This will require $O(\log n)$ time because no two processes in a blocking list will be associated with the same centroid path (recall that at most one process in the list at any centroid path is active at any given instant).

We need to add one more feature to our algorithm in order to ensure that the following fact holds. This fact will be critical in proving Invariants 1 and 2.

**Fact 6.4** *A process with root $x$ helps at most 1 process whose root is a strict ancestor of $x$.*

Fact 6.4 is ensured as follows. Consider an active process $\mathcal{P}$ which gets blocked at its root node $x$ because it is marked by some other process $\mathcal{Q}$ within which it is strictly nested. Then $\mathcal{P}$ raises a *demand flag* at $x$. Note that a demand flag can be raised only at the root of a process. All active processes which raise a demand flag at $x$ are organized into a list. When $\mathcal{Q}$ unmarks $x$, it picks the first process $\mathcal{P}$ from this list, resets its demand flag, performs the actual operations of $\mathcal{P}$ on $x$, and marks $x$ for $\mathcal{P}$, thus unblocking $\mathcal{P}$. Then when $\mathcal{P}$ finishes, it will perform a similar unblocking of the next process in the list. Thus once a process has become active and raised a demand flag, there is at most one process within which it is strictly nested which it can help.

**Remark.** Recall the construction of codes for newly inserted nodes from Section 6.5. Note that if a process has already updated the codes at a node $x$ before a new node $z$ is inserted between $x$ and its parent, then the effect of this process will automatically be visible at $z$. Therefore, if $x$ is marked by some process, the same mark is put on $z$. If $x$ is unmarked then so is $z$. Also, if a new leaf $z$ is inserted as a child of $x$ then the effect of the above process will automatically be visible at $z$. In addition, $z$ will not be marked.

We conclude this section with the following lemma, which proves one part of Invariant 1'.

**Lemma 6.5** *A process Reassign$(x)$ initiated by the move of $x$ to $\pi$ will finish before process Reassign$(y)$ initiated by the move of $y$ to $\pi$ is activated, where $x$ is closer to the root of $T$ than $y$.*

**Proof.** This follows from the priorities established for processes in *list*$(\pi)$ in this section, with one clarification. Recall from Section 6.5 that *Reassign*$(x)$ may be initiated after *Reassign*$(y)$ if $x$ is inserted after $y$ has already moved to $\pi$. This leads to the danger that *Reassign*$(y)$ may actually be activated before $x$ is inserted. But recall the way codes are assigned to newly inserted nodes. If the above situation occurs then the code for $x$ is easily seen to indicate that $x$ is in $\pi$ (or in an ancestor path of $\pi$), and thus the *Reassign*$(x)$ procedure corresponding to the change of path of $x$ to $\pi$ will not be initiated at all. $\Box$

### 6.7 Processing Queries

We need to establish several properties before describing the query processing algorithm.

Fact 6.6 is a direct consequence of the fact that centroid paths only move down the tree.

**Fact 6.6** *Suppose paths $\pi_1, \pi_2$ occur in code$(x)$ at some instant, with $\pi_1$ occurring before $\pi_2$. Then $\pi_1$ must be in a strictly larger size class compared to $\pi_2$. Further, let $u$ be the head of $\pi_1$ when it came into existence. Then the current head of $\pi_2$ must be a descendant of $u$. Clearly, at no instant can $\pi_1$ be a descendant of $\pi_2$; in addition, if $\pi_1, \pi_2$ both occur in code$(y)$ as well, for some vertex $y$ at some other instant, then $\pi_1$ must occur before $\pi_2$ in code$(y)$.*

Fact 6.7 follows from the priorities given to various processes in $list(\pi)$.

**Fact 6.7** *A Reassign$(x)$ process initiated by the move of $x$ to path $\pi$ becomes active only after Assign$(\pi)$ has completed.*

We need the following properties derived from Invariants 1', 2 and 3 in order to answer queries. First, we need a definition.

**Definition.** The *predecessor* path of a path $\pi$ is defined as follows. Consider the instant at which $\pi$ comes into existence. If the only node $z$ in $\pi$ at this instant is a newly inserted node, then the predecessor is the only child path of $\pi$ at this instant. Otherwise, if $z$ is not a newly inserted node, then its predecessor is the path which contained $z$ before it moved to $\pi$. Note that the predecessor of $\pi$ is in a size class one smaller than $\pi$.

**Lemma 6.8** *Suppose paths $\pi_1, \pi_2$ occur consecutively in code$(x)$ at some instant $t$, with $\pi_2$ following $\pi_1$. Then, if $\pi_1$ is an ancestor of $\pi_2$ at this instant, it is either the parent, grandparent, or great-grandparent of $\pi_2$. Similarly, if $\pi_1$ is not an ancestor of $\pi_2$ at this instant, then the path containing the LCA of the heads of $\pi_1, \pi_2$ is either the parent or grandparent of $\pi_1$ and either the parent, grandparent, or great-grandparent of $\pi_2$.*

**Proof.** We consider the two cases in the lemma in turn.

**Case 1.** Suppose $\pi_1$ is an ancestor of $\pi_2$ at instant $t$ but not its parent. At instant $t$, let $\pi_3$ denote that child path of $\pi_1$ which is an ancestor of $\pi_2$. Clearly, such a path $\pi_3$ must exist. Let $\pi_3'$ be the predecessor of $\pi_3$ and let $\pi_3'$ be the predecessor of $\pi_3'$. It now suffices to show that either $\pi_2 = \pi_3'$ or $\pi_2 = \pi_3''$. Suppose this is not the case. Then we show that $\pi_3$ must occur between $\pi_1$ and $\pi_2$ in *code*$(x)$ at instant $t$, a contradiction.

We will need the following fact. Let $z'$ be the head of $\pi_3'$ when it comes into existence. Let $z''$ be the head of $\pi_3''$ when it comes into existence. Then $z', z''$ must be ancestors of $\pi_2$ at instant $t$. This can be seen as follows for $z'$; the proof for $z''$ is similar. Since the head of $\pi_3$ is an ancestor of $\pi_2$ at instant $t$ and since paths only move down the tree, the head $h$ of $\pi_3$ at the time it came into existence is also an ancestor of $\pi_2$ at instant $t$. By the definition of predecessor, $h$ must have moved from $\pi_3'$ to $\pi_3$ in the past or the head of $\pi_3'$ was the only child of $h$ if $h$'s insertion created $\pi_3$. It follows that $z'$ must be an ancestor of $h$ or the only child of $h$ and therefore $z'$ is an ancestor of $\pi_2$ at the time $h$ moves to $\pi_3$ and hence at subsequent time $t$.

By Invariant 1', $Assign(\pi_3'')$ must complete before $\pi_3$ comes into existence. When this process completes, $\pi_3''$ will appear between $\pi_1$ and $\pi_2$ in *code*$(x)$. Since $\pi_3''$ does not appear in *code*$(x)$ at time $t$, it must have moved off the path between $\pi_3$ and $\pi_2$ at some instant before $t$; further, the $Reassign(y)$ process which removed $\pi_3''$ from $code(x)$ would have processed $code(x)$ before instant $t$ (here $y$ is the last node on the path between $\pi_3$ and $\pi_2$ occupied by $\pi_3''$; note that $y$ moves from $\pi_3''$ to $\pi_3'$). By Fact 6.7, this $Reassign(y)$ process could not have become active until $Assign(\pi_3')$ completed (note that this $Reassign(y)$ procedure will be in $list(\pi_3')$). Thus $Assign(\pi_3')$ would have processed $x$ by instant $t$ and added $\pi_3'$ between $\pi_1$ and $\pi_2$ in $code(x)$.

Next, since $\pi_3'$ does not appear in $code(x)$ at time $t$, it must have moved off the path between $\pi_3$ and $\pi_2$ at some instant before $t$; further, the $Reassign()$ procedure which removed $\pi_3'$ from $code(x)$ would have processed $code(x)$ before instant $t$. By Fact 6.7, this $Reassign()$ procedure could not have become active until $Assign(\pi_3)$ completed. It follows that $Assign(\pi_3)$ would have processed $x$ by instant $t$ and $\pi_3$ must appear between $\pi_1$ and $\pi_2$ in $code(x)$ at instant $t$, as required.

**Case 2.** Suppose $\pi_1$ is not an ancestor of $\pi_2$ at instant $t$. Let $u$ be the head of $\pi_1$ when it came into existence at instant $t_1$, say. By Fact 6.6, the head $w$ of $\pi_2$ at instant $t$ must be a descendant of $u$. Consider the path of nodes from $u$ to $\pi_1$, at instant $t$. Let $v$ be the deepest node on this path which is an ancestor of $w$. Let $v$ change paths from $\pi$, to, say, $\pi_4$, at instant $t_2 + 1$. $\pi_4$ must be in a size class one larger than $\pi_1$. We claim that, at instant $t$, $v$ must either be in $\pi_4$ or in the parent path $\pi_5$ of $\pi_4$, and that the path containing $v$ must be the parent, grandparent, or great-grandparent of $\pi_2$. The lemma follows.

That $v$ must be in $\pi_4$ or its parent at instant $t$ is shown as follows. Suppose this is not true. By Invariant 1', the $Reassign(v)$ procedure initiated by the move of $v$ from $\pi_1$ to $\pi_4$ would have completed by instant $t$. Then $\pi_1$ will not be in $code(x)$ at instant $t$, a contradiction.

It remains to show that the path $\pi$ containing $v$ at instant $t$ must be the parent, grandparent, or great-grandparent of $\pi_2$. At instant $t$, let $\pi_3$ denote that child path of $\pi$ which is an ancestor of $\pi_2$. Clearly, such a path $\pi_3$ must exist. Let $\pi_3'$ be the predecessor of $\pi_3$ and let $\pi_3''$ be the predecessor of $\pi_3'$. It now suffices to show that either $\pi_2 = \pi_3$, $\pi_2 = \pi_3'$ or $\pi_2 = \pi_3''$. Suppose this is not the case. Then we show that $\pi_3$ must occur between $\pi_1$ and $\pi_2$ in $code(x)$ at instant $t$, a contradiction. This proof is identical to the one in the last paragraphs of Case 1. We repeat it here for completeness.

As in Case 1, let $z'$ be the head of $\pi_3'$ when it comes into existence. Let $z''$ be the head of $\pi_3''$ when it comes into existence. Then, as in Case 1, $z', z''$ must be ancestors of $\pi_2$ at instant $t$.

By Invariant 1', $Assign(\pi_3'')$ must complete before $\pi_3$ comes into existence. When this process completes, $\pi_3''$ will appear between $\pi_1$ and $\pi_2$ in $code(x)$. Since $\pi_3''$ does not appear in $code(x)$ at time $t$, it must have moved off the path between $\pi_3$ and $\pi_2$ at some instant before $t$; further, the $Reassign()$ procedure which removed $\pi_3''$ from $code(x)$ would have processed $code(x)$ before instant $t$. By Fact 6.7, this $Reassign()$ procedure could not have become active until $Assign(\pi_3')$ completed. Thus $Assign(\pi_3')$ would have processed $x$ before time $t$ and added $\pi_3'$ between $\pi_1$ and $\pi_2$ in $code(x)$. Since $\pi_3'$ does not appear in $code(x)$ at time $t$, it must have moved off the path between $\pi_3$ and $\pi_2$ at some instant before $t$; further, the $Reassign()$ procedure which removed $\pi_3'$ from $code(x)$ would have processed $code(x)$ before instant $t$. By Fact 6.7, this $Reassign()$ procedure could not have become active until $Assign(\pi_3)$ completed. It follows that $Assign(\pi_3)$ would have processed $x$ by instant $t$ and $\pi_3$ must appear between $\pi_1$ and $\pi_2$ in $code(x)$ at instant $t$, as required. $\Box$

Lemma 6.8 is used to prove Lemma 6.9 below.

**Lemma 6.9** *Let $\pi$ be the last path which occurs in both code$(x)$ and code$(y)$. Let $\pi_1, \pi_2, \pi_3, \pi_4$ be the paths following $\pi$ in code$(x)$ and $\pi_1', \pi_2', \pi_3', \pi_4'$ be the paths following $\pi$ in code$(y)$. Then the current LCCP of $x$ and $y$ is also the LCCP of the current heads of $\pi_4$ and $\pi_4'$. Further, the number of paths strictly between $\pi_4$ and this LCCP is $O(1)$, and likewise for $\pi_4'$.*

**Proof.** Let $z$ denote the LCA of $x$ and $y$, and let $x'$ and $y'$ be the children of $z$ which are ancestors of $x$ and $y$, respectively.

First, consider the first part of the lemma. We show that $\pi_4$ must be completely in $T_{x'}$ and $\pi'_4$ in $T_{y'}$. Suppose $\pi_4$ is not completely in $T_{x'}$. We derive a contradiction as follows. A similar proof holds for $\pi'_4$. There are two possible cases.

**Case 1.** Suppose $\pi_4$ is completely in $T_{y'}$. Then, since $\pi_4$ is present in *code*$(x)$, $\pi_4$ must have had its head on the path from the root to $z$ at some instant. At some later instant, the head of $\pi_4$ must have moved into $T_{y'}$; $z$ changes paths from $\pi_4$ to some path $\pi'$ at this instant. By Fact 6.6, none of $\pi_1 \dots \pi_3$ is completely contained in $T_{x'} \cup T_{y'}$ at this instant. By Invariant 1', the $Reassign(z)$ process initiated by $z$ moving to $\pi'$ will complete before $z$ changes paths for the second time after moving to $\pi'$; this process would also remove $\pi_4$ from *code*$(x)$. But, since $\pi_4$ is currently in *code*$(x)$, $z$ must have changed paths at most once since moving to $\pi'$. It follows that neither of $\pi_1, \pi_2$ can be completely contained in $T_{x'} \cup T_{y'}$. Then, by Invariant 3, at most one of these two paths can be present in one of *code*$(x)$, *code*$(y)$ without being present in the other, a contradiction.

**Case 2.** Suppose $\pi_4$ is not completely in $T_{y'}$. Since $\pi_4$ is not completely in $T_{x'}$, it follows from Lemma 6.8 that none of the paths $\pi_1, \pi_2, \pi_3$ has any node in $T_z$. Again, by Invariant 3, at most one of these three paths can be present in one of *code*$(x)$, *code*$(y)$ without being present in the other, a contradiction.

Next, consider the second part of the lemma. We show that if $\pi$ is not the LCCP or an ancestor of the LCCP then it must be its child or grandchild. The second part of the lemma then follows from Lemma 6.8.

Since $\pi$ occurs in both *code*$(x)$ and *code*$(y)$, its head must have been an ancestor of $z$ at some instant. If $\pi$ is a descendant but not a child or grandchild of the LCCP currently, then $z$ must have changed paths at least twice after moving off path $\pi$. By Invariant 1', the $Reassign(z)$ process initiated by $z$ moving off $\pi$ will have completed; this procedure would also have removed $\pi$ from both *code*$(x)$ and *code*$(y)$, a contradiction. □

Note that a special case arises if $\pi_4$ is not defined; in this case $\pi_4$ is taken to be the path containing $x$ (i.e., the last path in code $(x)$). $\pi'_4$ is defined analogously. Lemma 6.9 continues to hold. The following lemma shows a property which will be needed to determine $\pi_4, \pi'_4$ in $O(1)$ time. Invariant 3 is crucial for this property to hold.

**Lemma 6.10** *Let $\pi$ be the last path in code$(x)$ which occurs in code$(y)$ as well. Consider the sequence of separator strings of paths in code$(x)$ which precede $\pi$ and consider the analogous sequence for code$(y)$. If these two sequences are not identical, then one of the following possibilities holds:*

1. *The two sequences are identical but for one path which has distinct separator strings in the two sequences; these two distinct separator strings have the same length.*

2. *The two sequences are identical but for the following. There is one path which is present in exactly one of the two sequences (without loss of generality, say, the first sequence). Further, the next path in the first sequence is also present in the second sequence but with a different separator string. Finally, $|s_1| + |s_2| = |s_3|$ and $s_1$ is not a prefix of $s_3$, where $s_1$ is the separator string for the former path in the first sequence, $s_2$ is the separator string for the latter path in the first sequence, and $s_3$ is the separator string for the latter path in the second sequence.*

**Proof.** Let $z, x', y'$ be as in the proof of Lemma 6.9.

By Invariant 1', either $\pi$ is not completely in $T_{x'} \cup T_{y'}$, or $\pi$ is one of the two most recent paths on which $z$ was present before it moved to its current path. For, if the head of $\pi$ is a descendant of $z$ and $\pi$ is not one of the above two paths then the *Reassign*$(z)$ procedure which removes $\pi$ from either *code*$(x)$ or *code*$(y)$ will have finished by now. Let $\pi_1$ and $\pi_2$ denote these two most recent paths, with $z$ being on $\pi_2$ after it leaves $\pi_1$, and let $\pi_3$ be $z$'s current path. We consider the two cases separately: $\pi \neq \pi_1$ and $\pi = \pi_1$. The main property used in these cases is the following. Unless $\pi = \pi_1$, no path which occurs before $\pi$ in even one of *code*$(x)$, *code*$(y)$ can lie completely in $T_{x'}$ or $T_{y'}$. And if $\pi = \pi_1$, then $\pi_2$ is the only possible path with this property. This is shown as follows.

Suppose $\pi'$ precedes $\pi$ in *code*$(x)$. By Fact 6.3, $\pi'$ is in a larger size class than $\pi$. All paths descendant from $z$ are in a smaller size class than $z$. Thus if $\pi = \pi_2$ or if $\pi$ is not fully in $T_{x'} \cup T_{y'}$ then $\pi'$ is not a descendant of $z$, i.e. $\pi'$ is not fully in $T_{x'} \cup T_{y'}$.

If $\pi = \pi_1$, then possibly $\pi' = \pi_2$. If this is not the case, and $\pi'$ is fully in $T_{x'} \cup T_{y'}$, by Fact 6.3, $\pi'$ must be in the same size class as $\pi_2$; thus $\pi_2$ and $\pi'$ are unrelated. Since $\pi_2$ once contained $z$, it follows that $\pi'$ was always a strict descendant of $z$. As $\pi'$ precedes $\pi$ in *code*$(x)$, by Fact 6.3, the current head of $\pi$ is a descendant of the original head of $\pi'$, and therefore the original head of $\pi'$ lies on one of $\pi_2$ or $\pi_3$; it cannot lie on $\pi_2$ as $\pi'$ and $\pi_2$ are unrelated; likewise, it cannot lie on $\pi_3$, as $\pi'$ and $\pi_2$ were always unrelated. Thus the only possible way in which $\pi'$ is fully in $T_{x'} \cup T_{y'}$ is for $\pi' = \pi_2$.

We now proceed with the two cases. First, suppose $\pi \neq \pi_1$. Then no path which occurs before $\pi$ in even one of *code*$(x)$, *code*$(y)$ can lie completely in $T_{x'}$ or $T_{y'}$. Therefore, processes associated with such paths will either modify both or none of *code*$(x)$, *code*$(y)$. The lemma now follows from Invariant 3 and the various possible scenarios described below.

1. Every path present in the first sequence is also present in the second. Further, at most one of these paths has a different separator code in the first sequence as compared to the second. This case corresponds to the situation when a *Compact*() process has updated the separator string for a particular path in exactly one of the two sequences. Note that in this case, the old and new separator strings for the above path have the same lengths.

2. With the exception of one path $\pi'$, every path present in the first sequence is also present in the second sequence. Further, one path $\pi''$ which is present in both sequences has distinct separator strings in the two sequences. In addition, $\pi''$ immediately follows $\pi'$ in the sequence containing $\pi'$.

   This case corresponds to the situation when process *Reassign*() initiated by the change of parent of $\pi''$ from $\pi'$ to some other path has removed $\pi'$ and simultaneously included the new separator code for $\pi''$ in exactly one of the two sequences. This case also occurs when an *Assign*$(\pi')$ process has included $\pi'$ and simultaneously included the new separator code for its child $\pi''$ in exactly one of the two sequences.

   Note that in both situations above, the lengths of the separators strings of $\pi'$ and $\pi''$ satisfy the properties stated in the lemma by virtue of the following facts: a path in the size class $[2^i, 2^{i+1})$ which is a child of a path in the size class $[2^j, 2^{j+1})$ gets a separator string of length $c(j-i)$, and two paths which are children of the same path get separator strings which constitute a prefix-free set.

Second, suppose $\pi = \pi_1$. In this case, we show in the next paragraph that processes associated with $\pi_2$ (namely, the *Reassign*() processes initiated when some node leaves $\pi_2$, *Assign*$(\pi_2)$ and *Compact*$(\pi_2)$) are either not yet active or modify both or none of $code(x)$ and $code(y)$. Since no path other than $\pi_2$ which occurs before $\pi$ in even one of $code(x), code(y)$ can lie completely in $T_{x'}$ or $T_{y'}$ (as was proved earlier), processes associated with such paths will either modify both or none of $code(x), code(y)$. The lemma now follows from Invariant 3 and the various possible scenarios described in Case 1.

We consider each of the above listed processes in turn.

First, consider $Reassign(u)$ processes initiated by $u$ moving off $\pi_2$. If $u$ is a strict ancestor of $z$ then this process affects neither $code(x)$ nor $code(y)$. If $u = z$, then this process is not yet active because the $Reassign(z)$ process initiated by $z$ moving off $\pi_1$ has not finished yet (as $\pi = \pi_1$ is still is both codes) and Invariant 1' says that the latter process must finish before the former is activated. This leaves the case when $u$ is a descendant of $z$. Since $z$ is currently on, say, $\pi_3$, which is in a size class one larger than $\pi_2$, $u$ must have moved to $\pi_3$ from $\pi_2$. Then the above $Reassign(u)$ process cannot be activated before the $Reassign(z)$ process initiated by $z$ moving off $\pi_2$ to $\pi_3$ is finished, by Invariant 1'.

Next, consider $Assign(\pi_2)$ and $Compact(\pi_2)$ processes. ¿From the definition of the roots of these processes in Section 6.5, the roots must be $z$ or ancestors of $z$. For $Assign(\pi_2)$, this follows because $z$ was in $\pi_2$ at some point. For $Compact(\pi_2)$, this can be seen from the following fact. Since the $Reassign(z)$ process initiated by the move of $z$ off $\pi = \pi_1$ has not yet finished, $z$'s code indicates it is either in $\pi_2$ (this process could have begun and modified $code(z)$) or in $\pi_1$, or in some descendant path of $\pi_1$. □

Given Lemma 6.9 and Lemma 6.10, the processing of queries in $O(1)$ times proceeds as follows. The aim is to find $\pi_4$ and $\pi'_4$ (see Lemma 6.9) in $O(1)$ time. Once this is done, a walk up the tree from each of $\pi_4$ and $\pi'_4$, moving from the head of one centroid path to the head of the parent path until a common path is reached, will terminate in $O(1)$ time and give the LCCP and BP, by Lemma 6.9. Determining $\pi_4$ and $\pi'_4$ is itself straightforward given $\pi_1$ and $\pi'_1$ (see Lemma 6.9). We show how to determine $\pi_1, \pi'_1$ in $O(1)$ time. This is done as follows.

The first bit of difference between $code(x), code(y)$ is found. Next, the separator strings in the two codes containing this bit difference are identified. Let these strings correspond to paths $\pi_x$ and $\pi_y$, respectively. If $\pi_x$ does not occur in $code(y)$ and $\pi_y$ does not occur in $code(x)$, then $\pi_x = \pi_1$ and $\pi_y = \pi'_1$, by Lemma 6.10. So assume without loss of generality that $\pi_x$ occurs in $code(y)$. Clearly, it must occur after $\pi_y$ in $code(y)$. By Lemma 6.10, it must actually follow $\pi_y$ in $code(y)$. Further, by Fact 6.6, $\pi_y$ cannot appear in $code(x)$. The first bit of difference in the portions of $code(x), code(y)$ starting after the separator strings of $\pi_x$ in each code is found next. By Lemma 6.10, the separator strings in $code(x)$ and $code(y)$ containing this bit difference correspond to the paths $\pi_1$ and $\pi'_1$, respectively.

## 6.8 Proofs of Invariants

It remains to show that Invariants 1', 2, and 3 are maintained.

**Lemma 6.11** *The above algorithm maintains Invariant 3.*

**Proof.** Consider a particular instant when a process (either $Assign()$, $Reassign()$ or $Compact()$) has already modified $code(x)$ and is yet to modify $code(y)$. Then $z$, the LCA of $x$ and $y$, would currently be marked by this process, even if it was inserted after $x$ was marked. It follows that any other process rooted at an ancestor of $z$ which has updated $x$ must have updated $y$ and vice versa, otherwise it would have its mark on $z$. □

**Definition.** The size of an *Assign*(), *Reassign*(), or *Compact*() process is defined to be the number of nodes whose codes the process modifies before it finishes (this does not include codes modified while helping other processes).

By Invariant 1' and the definition of process root, the size is $\Theta(2^i)$ for $Assign(\pi)$ and $Compact(\pi)$, if the nodes in $\pi$ are in the range $[2^i, 2^{i+1})$. By the definition of $Reassign(x)$, the size of a $Reassign(x)$ procedure initiated when $x$ moves from path $\pi$ to $\pi'$ is upper bounded by the maximum size reached by the subtree rooted at the appropriate child of $x$ before $Reassign(x)$ finishes. By Invariant 1', this happens while $x$ is still in $\pi'$ or in the parent path of $\pi'$. Here, the appropriate child denotes that child which is contained in a path which is neither $\pi$ nor $\pi'$.

The following lemma is the key reason why Invariants 1' and 2 can be maintained; it also explains the $O(\log^3 n)$ bound for our algorithm. In fact, this lemma will itself use Invariant 1'. However, the argument will not be circular because we will use induction on time.

**Lemma 6.12** *Consider a subtree rooted at vertex $x$ having size in the range $[2^i, 2^{i+1})$ at a particular instant. The total sizes of all processes initiated until this instant and having roots in $T_x$ is $O(i^2 2^i) = O(2^i \log^2 n)$, assuming Invariant 1' holds up to this instant. The total number of operations required to complete all these processes is $O(2^i \log^3 n)$.*

**Proof.** We show that the total sizes of all processes in $T_x$ initiated until this instant is $O(i^2 2^i) = O(2^i \log^2 n)$. Since each code takes $O(\log n)$ time to update ($O(\log n)$ time is spent in traversing the blocking list to determine which process to help, and $O(\log n)$ time is spent in updating the code and annotations at a particular node), the total number of operations required to complete all these processes is $O(2^i \log^3 n)$.

We consider each process type separately.

*Assign*()**.** *Assign*() processes are initiated exactly once per path. Recall that the size of an $Assign(\pi)$ process is $\Theta(2^j)$, if the nodes in $\pi$ are in the range $[2^j, 2^{j+1})$. It follows that the sum of the sizes of *Assign*() processes over all paths in $T_x$ is $O(i2^i)$.

*Reassign*()**.** Consider centroid paths $\pi'$ completely in $T_x$ whose nodes are in the range $[2^h, 2^{h+1})$, $h < i$. We show that the sizes of *Reassign*() processes initiated by the change of paths at the parent of the head of $\pi'$ can be charged to nodes in $T_x$ so that each node is charged $O(i)$. Summing over all $h < i$ gives the $O(i^2 2^i)$ bound.

We partition the sequence of initiations of the above *Reassign*() processes into maximal subsequences. Within a subsequence, the size classes of successive parents of the head of $\pi'$ are strictly increasing. So the last initiation of *Reassign*() in a subsequence has the following property: the next move (the move subsequent to the one causing initiation of the above process) of the parent of the head of $\pi'$ is to a newly created path in the size class $[2^{h+1}, 2^{h+2})$. It follows that the first *Reassign*() process in each subsequence (except possibly the first subsequence) corresponds to the parent of the head of $\pi'$ moving to a newly created path in the size class $[2^{h+1}, 2^{h+2})$.

In each subsequence, we account for the last two initiations separately from the others. First, we show that, excluding the last two initiations, the size of each initiation within a subsequence is $\Theta(2^h)$. Then we account for the sizes of the last two initiations separately. Finally, we account for the sizes of all but the last two initiations.

Consider an initiation of $Reassign(a)$ which is not the penultimate or the last member in its subsequence. Suppose this process was initiated because $a$ moved to path $\pi$. Then the next member of this subsequence would have been initiated because $a$ moves to the parent $\pi''$ of $\pi$. By Invariant 1, the current initiation finishes before $a$ leaves $\pi''$. Since this initiation is not amongst the last two in its subsequence, $\pi'$ must be a child of either $\pi$ or $\pi''$ until $a$ leaves $\pi''$. The size of this initiation of *Reassign*$(a)$ is thus 1 (for the node $a$) plus the size of the subtree rooted at the head of $\pi'$, which is $\Theta(2^h)$, as claimed.

Next, consider initiations of *Reassign*() which are the last two in their respective subsequences. Let the first be initiated by a move to path $\pi$ and the second by a move from $\pi$ to its parent $\pi''$. Note that several new paths could emerge between $\pi'$ and $\pi''$ before or while these initiations of *Reassign*() become active and thus the size of this process could be much more than $2^h$. Consider the largest of these paths, i.e., the one in the largest size class. Let $\pi'''$ be this path and let it be in the size class $[2^k, 2^{k+1})$. Then the size of each of the above initiations of *Reassign*() is $O(2^k)$ and this size can be charged to the size of the *Assign*$(\pi''')$ process (recall that sizes of *Assign*() processes have already been accounted for). By Invariant 1', *Assign*$(\pi''')$ will be charged by just two *Reassign*() process.

It remains to account for the $\Theta(2^h)$ work done for all but the last two initiations in each subsequence. Consider a particular initiation which is not amongst the last two members of its subsequence. This initiation began because the parent of the head of $\pi'$ moves from one centroid path to another. Let $a$ be this parent node and let its size at the moment of the initiation be $2^{k+1}$. For each insertion in the subtree rooted at the other child of $a$ occurring after $a$ had size $2^k$, charge the inserted node $2^{h-k+1}$. Note that there must have been at least $2^k - 2^h \geq 2^{k-1}$ such insertions if $\pi'$ was created before $a$ had size greater than $2^k$; otherwise this initiation is charged to the *Assign*$(\pi')$ process. It remains to count the charges at each node.

We now count the total charge at each node $b$. We consider each value of $k$ from 1 to $i$ separately and consider the charge given to $b$ by an initiation in which the parent node $a$ has size $2^{k+1}$, as above. There are at most $2^{k+1-h}$ paths in the size class $[2^h, 2^{h+1})$ for which initiations of the above form charge $b$. This is because all such paths are mutually disjoint and present in the subtree rooted at the highest ancestor $c$ of $b$ (at the time of its insertion) such that $|T_c| < 2^{k+1}$. Therefore, the total charge given to a node at the time of insertion is at most $2^{k+1-h}2^{h-k+1} = O(1)$, for a particular $k$. This sums to $O(i)$ over all $k \leq i$, as claimed.

*Compact*()**.** Consider centroid paths $\pi'$ completely in $T_x$ whose nodes are in the range $[2^h, 2^{h+1})$, $h \leq i$. Recall that the size of a *Compact*() process associated with $\pi'$ is $\Theta(2^h)$. We show that these sizes can be charged to nodes in $T_x$ so that each node is charged $O(1)$. Summing over all $h \leq i$ gives the lemma.

We ignore the first initiation of *Compact*$(\pi')$. The total size of the first initiations over paths of all size classes is $O(i2^i)$. We account for subsequent initiations.

Consider a particular initiation. By Lemma 6.2, there are at least $2^{h+2}$ insertions between the previous initiation and the current one in the subtree rooted at the head of $\pi'$. The size of the current initiation is charged to these insertions; each node inserted is charged $O(1)$. Consider one such node $b$. Clearly, two initiations for the same path do not both charge $b$. We now show that no initiation for any other path $\pi''$ in the size range $[2^h, 2^{h+1})$ charges node $b$. This follows because the only path in the above size range which charges $b$ is the unique path in this size range which is an ancestor of $b$ at the time of $b$'s insertion. The lemma follows. □

**Invariants 1 and 2 hold.** Broadly, Invariants 1 and 2 hold because of the following 3 reasons.

1. By Lemma 6.12 and Fact 6.4, the total amount of work done by a process of size $2^i$ in helping other processes is limited to $O(2^i \log^3 n)$.

2. Each insertion does $O(\log^3 n)$ work on processes at each ancestor path and at each of the children paths of each ancestor path which are in a size class one smaller than their respective parents.

3. ¿From Invariant 1' in Section 6.4 and by Lemma 6.2, there are sufficiently many insertions available to finish processes in time.

The formal proofs of Invariants 1 and 2 are given next.

## 6.9 Proofs of Invariants 1 and 2.

First, we need the following lemma.

**Lemma 6.13** *Consider an Assign(), Reassign(), or Compact() process with root $x$. Suppose $x$ has size $s$ at some later instant and this process has already performed $\Theta(s\log^3 n)$ operations since it became active. If Invariant 1' holds up to this instant then the above process must have completed by this instant.*

**Proof.** Given Invariant 1', by Lemma 6.12, the total sum of the sizes of all processes rooted in $T_x$ is $O(s\log^2 n)$ and $O(s\log^3 n)$ operations suffice to complete these processes. Therefore, $O(s\log^3 n)$ work will be performed by the process in consideration in helping nested processes. The work done by this process on itself is $O(s\log n)$. By Fact 6.4, there is just 1 process within which any process is strictly nested and on which it does work; this work is also $O(s\log n)$. The lemma follows. □

Invariant 2 follows from Lemmas 6.14 and 6.15. Both lemmas assume that Invariant 1' holds. Invariant 1' itself will be shown shortly.

**Lemma 6.14** *Consider a process Assign($\pi$) which is initiated when $\pi$ has head, say, $x$. If Invariant 1' holds until a Compact($\pi$) process is initiated then Assign($\pi$) completes before a Compact($\pi$) process is initiated.*

**Proof.** Let $\pi$ be in the size range $[2^i, 2^{i+1})$. Recall that the size of the *Assign*($\pi$) process is $O(2^i)$ (this follows from Invariant 1' and the definition of root for this process). Consider the first $2^i$ insertions into the subtree rooted at the head of $\pi$ made since *Assign*($\pi$) is initiated, starting with the insertion which caused the above initiation. Each of these insertions will do $\Theta(\log^3 n)$ work on *Assign*($\pi$). It follows from Lemma 6.13 that *Assign*($\pi$) will complete by the time these insertions are made. The lemma follows from Lemma 6.2. □

**Lemma 6.15** *Consider a process Compact($\pi$) which is initiated when $\pi$ has head, say, $x$. If Invariant 1' holds until the next initiation of Compact($\pi$) then this Compact($\pi$) process completes before the next initiation of Compact($\pi$).*

**Proof.** Let $\pi$ be in the size range $[2^i, 2^{i+1})$. Recall that the size of the *Compact*($\pi$) process is $O(2^i)$ (this follows from Invariant 1' and the definition of root for this process). Consider the first $2^i$ insertions into the subtree rooted at the head of $\pi$ made starting with the initiation of *Compact*($\pi$) when $\pi$ has head $x$. Recall that *Compact*($\pi$) has the highest priority amongst processes in *list*($\pi$). Each of the above insertions will do $\Theta(\log^3 n)$ work on either the only process in *list*($\pi$) which was active at the time of the first initiation of *Compact*($\pi$), or the first initiation of *Compact*($\pi$) itself. It follows from Lemma 6.13 that both these processes will complete by the time these insertions are made. The lemma follows from Lemma 6.2. □

It remains to show Invariant 1'; in fact, we will now prove Invariant 1, which clearly implies Invariant 1'. Recall that the last part of Invariant 1 follows from Lemma 6.5, which has been shown earlier.

**Lemma 6.16** *Process Reassign$(y)$ initiated when node $y$ switches to path $\pi$ will complete before $y$ leaves $\pi$.*

**Proof.** Suppose $\pi$ is in the size range $[2^i, 2^{i+1})$. Assume that Invariant 1 holds inductively until just before $y$ leaves $\pi$. We show that the above process will have finished when $y$ leaves $\pi$.

Let $z$ be the head of $\pi$ when $y$ switches to path $\pi$. We will show that *Reassign*$(y)$ will complete within at most $2^i$ insertions into $T_z$, starting with the insertion causing $y$ to switch to $\pi$.

By the induction hypothesis, the only processes *Reassign*$(x)$ in the list at $\pi$ just before $y$ moves to $\pi$ are those with the property that $x$ is in $\pi$. It follows that the sum of the sizes of all such processes will be $O(2^i)$, even after accounting for the next $2^i$ insertions into the subtree rooted at $T_z$. Thus, the sum of the sizes of all processes in the list at $\pi$, including the processes initiated by the switch of $y$ to $\pi$ and including any *Assign*$(\pi)$ or *Compact*$(\pi)$ process, is $O(2^i)$, by Lemma 6.14 and 6.15, accounting for the next $2^i$ insertions into the subtree rooted at $T_z$.

Consider the next $2^i$ insertions made into $T_z$ starting with the insertion causing $z$ to switch to $\pi$; $y$ will be in $\pi$ even after these insertions and $z$ will be in $\pi$ or its parent. Each such insertion will perform $O(\log^3 n)$ work on some process in *list*$(\pi)$ (recall that each insertion does work not only on ancestor paths but also at children of ancestor paths which are in a size class one smaller). By Lemma 6.13, all the above processes in the list at $\pi$ will be complete by the end of these insertions. □

**Lemma 6.17** *Process Assign$(\pi)$ initiated when node $y$ switches to a new singleton path $\pi$ will complete before either some node from $\pi$ moves out of $\pi$ to form a new singleton path, or a singleton path formed by a newly inserted node becomes the parent of $\pi$.*

**Proof.** Suppose $\pi$ is in the size range $[2^i, 2^{i+1})$. Assume that Invariant 1 holds inductively until just before one of the two events in the lemma happens. We show that the above process will have finished by the time one of these events happens.

There will be at least $2^i$ insertions into the subtree rooted at the head of $\pi$ before either of these events happens. Consider the first $2^i$ such insertions. Each such insertion will do $O(\log^3 n)$ work on processes in *list*$(\pi)$. Since the process considered in the lemma has the highest priority amongst processes in *list*$(\pi)$ and since the size of this process is $O(2^i)$, the lemma follows from Lemma 6.13. □

# 7 The $O(1)$ Worst-Case Algorithm

We maintain three levels of trees, denoted by $T_0, T_1, T_2$, respectively. Level $T_0$ contains all nodes in $T$ partitioned into subtrees of size[5] $\Theta(\log\log^4 n)$. Level $T_1$ contains one representative (i.e., the root) from each subtree in $T_0$; in addition, it contains all nodes in $T_0$ which are LCAs of roots of subtrees in $T_0$. $T_1$, in turn, is partitioned into subtrees of size $\Theta(\log^4 n)$. Level $T_2$ contains one representative (i.e., the root) from each subtree of $T_1$ along with all nodes in $T_1$ which are LCAs of roots of subtrees in $T_1$. Thus $|T_2| = O(\frac{n}{\log^4 n})$ and $|T_1| = O(\frac{n}{\log\log^4 n})$.

$T_0$ is made binary, using a standard binarization. Then, it follows from the above that $T_1$ and $T_2$ are binary trees. Maintaining all three trees as binary trees is crucial for technical reasons.

**Data Structures Maintained.** We maintain data structures for each subtree into which $T_1$ is partitioned and the tree $T_2$ as per the $O(\log^4 n)$ time algorithm described above.

[5] $\log\log^4 n$ is our notation for $(\log\log n)^4$.

We maintain for each node in $T_0$, its representative in $T_1$, and for each node in $T_1$, its representative in $T_2$.

In addition, we maintain the following information for subtrees in $T_0$.

1. For each labelled tree of size at most $3\log\log^4 n$ and each pair of labels, the LCA of the corresponding nodes.

2. For each labelled tree of size at most $3\log\log^4 n$, and any specified node $x$ or edge $e$ in this tree, the name for the new tree obtained by inserting a new node as a leaf of $x$ or inside $e$.

Since there are $O(2^{6\log\log^4 n}(3\log\log^4 n)^{3\log\log^4 n})$ labelled binary trees of size at most $3\log\log^4 n$ with labels in the range $1\ldots3\log\log^4 n$, the total space occupied by above tables is $O(n)$. These tables can also be built in $O(n)$ time.

**Processing Insertions.** Each insertion into $T$ will do $O(1)$ work at each of the 3 levels. This will result in $\Theta(\log^4 n)$ work being available for each insertion into $T_2$ and $\Theta(\log\log^4 n)$ work for each insertion into $T_1$.

Insertions into $T$ will require the following actions. First, the insertion into the appropriate subtree $S$ of $T_0$ is made. This is done using a constant time table look-up to calculate the name of the new subtree after insertion. Second, if this subtree size reaches $3\log\log^4 n$ then it is partitioned into two subtrees, $S_1$ and $S_2$, each of size at most $2.5\log\log^4 n$. Here, the root of $S_1$ is the same as the root of $S$. The root $r'$ of $S_2$ and possibly one additional node are now inserted into the appropriate subtree $S3$ in $T_1$ (recall that $T_1$ comprises roots of subtrees in $T_0$ along with their LCAs), if these are not already present in $T_1$. These two nodes are inserted sequentially, with the latter being inserted first. Note that these nodes will be inserted in $T_1$ either in the middle of an edge or as leaves. The following changes are required on insertion.

1. The tree sizes and centroid paths in $S3$ need to be updated to reflect the addition of $r'$.

2. Work on $Assign(), Reassign(), Compact()$ processes needs to be done at ancestor paths in $S3$.

These changes are actually performed by the previous $.5\log\log^4 n$ insertions into the subtree $S$ of $T_0$ rooted at $r'$, one operation per insertion. Thus the changes required by an insertion into $T_1$ are actually performed in anticipation of this insertion; the actual insertion happens only after these operations complete. Next, if the subtree $S3$ reaches $3\log^4 n$ in size then it is split into two subtrees, each of size at most $2.5\log^4 n$ (this is not strictly true, as we describe shortly in Section 7.1). The root of the second subtree and possibly one additional node are inserted into $T_2$. Updates similar to that performed for inserting a node into $S3$ need to be performed for these insertions as well. These are performed over the previous $\Theta(\log^4 n)$ insertions into $S3$, one operation per insertion.

There are three issues which need further elaboration. The first is the algorithm for splitting a subtree into two. The second is showing that performing the updates due to insertions into $T_1, T_2$ over a sequence of insertions into $T_0, T_1$, respectively, preserves correctness. The third is the processing of queries. We discuss these issues in turn.

## 7.1 Splitting Subtrees upon Insertion

**Splitting Algorithm for $T_0$.** Consider one of the subtrees $S$ into which $T_0$ is partitioned. When the size of $S$ reaches $2.5\log\log^4 n$, a process which splits this tree into two begins. The operations in this process are performed over the next $.5 \log\log^4 n$ insertions into $S$. This process finishes by the time the size of $S$ reaches $3 \log\log^4 n$. Since each insertion into $S$ takes $O(1)$ time, $S$ would have been split into two, the last $.5 \log\log^4 n$ insertions (except the very last one) performed on the appropriate subtree, and the data structures for the LCA processing computed, by the time the size of $S$ reaches $3 \log\log^4 n$. We elaborate on this shortly. One important issue needs to be addressed at this point. The above process is actually done on a copy of $S$, with the original copy being maintained as such so that queries which come in the meantime can be answered. Insertions are performed on both copies. Once the second copy is ready, a flag associated with the root is set so that future queries can read from that copy.

Details of the data structure for $T_0$ are as follows. For each subtree $S$ into which $T_0$ is partitioned, the following information is maintained, in addition to the data structure storing the tree itself.

1. For each node $v$ in $S$, a label for $v$ called $label(v)$. The subtree $S$ along with the associated labels will determine the appropriate labelled tree used to answer LCA queries for $S$.

2. An array $lton(S)$ storing, for each label $l$ in the range $1 \ldots 3 \log\log^4 n$, the node corresponding to this label, if any, in $S$. This inverse map from labels to nodes is needed because the LCA obtained using look-up tables on labelled trees will return only a label rather than an actual node.

3. The name $name(S)$ of the labelled 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 $lton(S)$, $name(S)$, $root(S)$, $flag(S)$. $flag(S)$ is explained next.

4. Actually, two copies of $label(v)$, $pointer(v)$ 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 $flag(S)$ pointed to by the "old" $pointer(v)$ will be set to 0 while that pointed to by the new $pointer(v)$ will be set to 1.

Of course, in addition to the above, there is a static table for each labelled tree of size at most $3 \log\log^4 n$ supporting the following queries: given two labels in this 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 here labels to nodes in $S$ are allocated in sequential order of insertion.

The algorithm to split subtree $S$ into subtrees $S_1$ and $S_2$ involves the following changes. First, the location of the split is determined in $O(\log\log^4 n)$ time using depth-first search; the best split (which will be $\frac{1}{3} - \frac{2}{3}$ at worst) is found. This defines $S_1, S_2$. Next, $name(S_1), root(S_1), lton(S_1)$ and $name(S_2), root(S_2), lton(S_2)$ are computed in the obvious way in $O(\log\log^4 n)$ time. Then the "old" $label(v), pointer(v)$ are now updated to to be in accordance with $S_1$ or $S_2$, whichever contains $v$, in $O(\log\log^4 n)$ time.

Note that all this while, the "current" $label(v)$, $pointer(v)$ and $name(S)$, $root(S)$, $lton(S)$ 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, insertions made are neglected in determining $S_1$, $S_2$, $name(S_1)$, $root(S_1)$, $lton(S_1)$, $name(S_2)$, $root(S_2)$, $lton(S_2)$ and in updating the "old" $label(v), pointer(v)$ above. 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 after the above structures have been computed. When all the $.5 \log\log^4 n$

insertions made after the splitting process starts have been performed, $flag(S_1)$ and $flag(S_2)$ are set and $flag(S)$ is reset so that for each $v$ in $S_1, S_2$, the "old" $label(v), pointer(v)$ become "current" and vice versa; this takes $O(1)$ time. Note that both $S_1, S_2$ have size at most $2\frac{1}{6}\log\log^4 n < 2.5\log\log^4 n$ each.

**Splitting Algorithm for $T_1$.** Note that each insertion into subtree $S$ of $T_1$ actually represents $\Theta(\log\log^4 n)$ insertions into $T$ in the portion represented by $S$. Therefore, $\Theta(\log\log^4 n)$ work is available for each insertion into $S$. The actual insertion into $S$ occurs only after the $\mathcal{W} = \Theta(\log\log^4 n)$ work associated with this insertion is done on $S$. An insertion into $S$ will be called *partial* if some but not all of the $\mathcal{W}$ work associated with it has already been performed. The *capacity* of $S$ is defined as its size times $\mathcal{W}$ plus the total work done on it by partial insertions.

The splitting algorithm on $S$ begins not when its size reaches $2.5\log^4 n$ but when its capacity reaches $2.5\log^4 n\mathcal{W}$ (this could happen before the size reaches $2.5\log^4 n$ but no later). This splitting is performed over the next $.5\log^4 n\mathcal{W}$ units of work performed on $S$ by current partial insertions and future insertions. It follows that this splitting process completes by the time the size of $S$ reaches $3\log^4 n$. When this splitting process completes, the capacity is less than $2.5\log^4 n\mathcal{W}$ for each of two resulting subtrees $S_1$ and $S_2$ (and therefore the size is less than $2.5\log^4 n$). Further, at this instant, each partial insertion into $S$ will also be partial to at least the same extent (i.e., in terms of the number of operations performed on this insertion) in one of $S_1, S_2$.

The splitting algorithm itself is similar to the splitting algorithm for $T_0$, but is done without using table look-up. For each subtree $S$ into which $T_1$ is partitioned, 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*$(S)$ storing centroid paths and associated information for paths in $S$.

3. A pointer *pointer*$(v)$ from each node $v$ to the location storing *cpath*$(S)$, *root*$(S)$, $flag(S)$. $flag(S)$ is explained next.

4. Two copies of *code*$(v)$, *number*$(v)$, *pointer*$(v)$ are maintained as before. One of these copies will be "old' and the other "current". The appropriate $flag(S)$ indicates which of these copies is current.

5. For each node $v$, there is at most one associated partially inserted node. The *capacity* of this partial insertion, i.e., the total work done so far by this partial insertions, is also maintained (this is to enable balanced splits based on the capacity, as explained below).

The algorithm to split subtree $S$ into subtrees $S_1$ and $S_2$ involves the following changes. First, the location of the split is determined in $O(\log^4 n)$ time using depth-first search. This split divides the capacity in $S$ as equally as possible (close to $\frac{1}{3} - \frac{2}{3}$, at worst). This defines $S_1, S_2$. The remainder of the procedure involves computing the data structures for $S_1, S_2$, applying the backlog of insertions, and finally setting the appropriate flag, as in the case of $T_0$. This takes $O(\log^4 n)$ time. The only point to note is that each partial insertion is performed to the extent of the number of operations already available for it in $S$.

## 7.2 Processing of Insertions into $T_1, T_2$.

We outline the processing of insertions into $T_2$. Insertions into $T_1$ are similar.

Recall that a new node is inserted into $T_2$ only when some subtree $S$ of $T_1$ splits into two. Each insertion into $S$ after its capacity reaches $2.5\log^4 n\mathcal{W}$ does one unit of work on $T_2$. By the time the capacity of $S$ reaches $3\log^4 n\mathcal{W}$, it splits into two and up to two new nodes are inserted into $T_2$. Thus the actual insertion into $T_2$ occurs only after the $\Theta(\log^4 n)$ work associated with this insertion is done on $T_2$. An insertion into $T_2$ will be called *partial* if some but not all of the $\Theta(\log^4 n)$ work associated with it has already been performed. An insertion into $T_2$ will be called *complete* if all of the $\Theta(\log^4 n)$ work associated with it has already been performed. In the remainder of the paper, the *size* of a subtree denotes the actual number of nodes in that subtree; this is in contrast to the *recorded size* of the subtree, which could be larger because an insertion is actually performed after all updates for it are performed, and these updates are performed sequentially from the root downwards, not simultaneously. The *size class* of a centroid path will be defined using the recorded sizes.

The following fact will be of importance. It follows from the fact that each subtree split in $T_1$ causes up to two nodes to be inserted into $T_2$, and the insertion of the second of these nodes begins after the insertion of the first is over.

**Fact 7.1** *Consider any node $y$ in $T_2$ such that the subtree rooted at $y$ has size $\alpha$ at a given instant. Then the number of partial insertions into this subtree at this instant is at most $\alpha$. In other words, the recorded size of node $y$ cannot exceed twice its actual size.*

The above $\Theta(\log^4 n)$ work associated with the insertion of a new node $x$ into $T_2$ is performed by a procedure $Insert(x)$. It proceeds down the tree towards the actual insertion site, starting at the root. The following actions are performed on each centroid path encountered.

1. The size count[6] is incremented. A partial insertion is said to have increased the recorded size of a particular node if it has incremented the size count in some descendant centroid path.

2. The centroid path is then modified in accordance with the updated counts, and the appropriate *Assign*(), *Reassign*(), and *Compact*() procedures are initiated.

3. $\Theta(\log^3 n)$ work is done on the processes in the lists at each of these centroid paths. Further, $\Theta(\log^3 n)$ work is done on processes in the lists at each of those children and grandchildren paths of the above centroid paths which are in a size class one smaller than their respective parents. Work on these children and grandchildren paths will be useful in maintaining Invariants 1 and 2 and will be required in Lemma 7.2.

The following issues need to be described in further detail.

1. How $Insert(x)$ traces the path down from the root to the insertion site.

2. How $Insert(x)$ maintains counts at centroid paths.

3. How the code and annotations for a node are updated by *Assign*(), *Reassign*(), and *Compact*() processes.

4. How blocking lists are traversed by *Assign*(), *Reassign*(), and *Compact*() processes.

[6]Recall from Section 6.1 that we have to keep a count of the number of nodes in the subtree rooted at the head of each centroid path. Further, the centroid paths themselves need to be modified as these counts change.

5. How Invariants 1 and 2 are maintained. It is easy to see that Invariant 3 is maintained by our algorithm for scheduling processes in spite of the processing of insertions into $T_2$ being spread out over insertions into $T_1$.

We address issue 5 first, followed by issues 1, 2, 3 and 4, in that order. Before proceeding, we need two clarifications.

**Disappearing Centroid Paths.** Note that since counts at centroid paths are updated from the root downwards and sequentially for each insertion, a centroid path could become empty temporarily. This happens, for instance, when a singleton path with a size count of $2^i - 1$ has its count incremented by 1, while the counts at the children paths are not incremented yet. We will not abandon empty paths; they will remain "on an edge" until they become non-empty again. Note that, by Fact 7.1, there is at most one empty centroid path on any edge. This needs to be kept in mind for the remainder of this paper.

**Lemmas and Invariants which continue to hold.** As can be easily checked (see the paragraph following the definition of Invariant 1'), once we establish Invariants 1'(a), 1'(b), and 2, all the lemmas in the $O(\log^4 n)$ algorithm will continue to hold with the exception of those lemmas which actually establish these invariants, namely, Lemmas 6.14, 6.15, 6.16, and 6.17. These lemmas need not hold now because we have partial insertions as well; these insertions are not guaranteed to perform $\Theta(\log^3 n)$ work on processes in the lists at each ancestor centroid path. Therefore, it suffices to reprove Invariants 1'(a), 1'(b), and 2 keeping partial insertions in mind.

### 7.2.1 Maintaining Invariants 1' and 2

**Maintaining Invariant 2.** Even though Invariant 1 required modification to Invariant 1', Invariant 2 can be maintained as such for the following reason. Recall Lemma 6.2. Between any two initiations of $Compact(\pi)$, there must be at least $2^{i+2}$ insertions into the subtree rooted at the head of $\pi$, where $\pi$ is in the size class $[2^i, 2^{i+1})$. Let $x$ be the head of $\pi$ at the time of the first initiation. It follows that there must be at least $2^{i+2}$ insertions into the subtree rooted at $x$ in the above time interval. Consider the first $2^{i+2}$ such insertions, complete or partial, each of which has increased the recorded size of $x$. Let the recorded size of $x$ before these insertions be $\alpha$, $2^i \leq \alpha \leq 2^{i+1} - 1$. Then the recorded size of $x$ after all these insertions is $\alpha + 2^{i+2} \leq 3.2^{i+1} - 1$. Therefore, $x$ is either in $\pi$, $\pi$'s parent, or $\pi$'s grandparent at the end of these $2^{i+2}$ insertions (recall size classes of paths are based on recorded sizes). By Fact 7.1, the size of $x$ is at least $\frac{\alpha+2^{i+2}}{2}$ at the end of these $2^{i+2}$ insertions. It follows that there must have been at least $\frac{\alpha+2^{i+2}}{2} - \alpha \geq 2^i$ complete insertions into $T_x$ between the two initiations of $Compact(\pi)$ above.

Each of these complete insertions will perform $\Theta(\log^3 n)$ work on processes in $list(\pi)$ (recall from point 3 in the actions listed at the beginning of Section 7 that each insertion does work not just on ancestor paths but also children and grandchildren of ancestor paths which are in a size class one and two smaller, respectively). These complete insertions will suffice to finish the first initiation of $Compact(\pi)$. We remark that Lemmas 6.14 and 6.15 can be shown formally now.

**Maintaining Invariant $1'$.** We will not be able to maintain Invariant 1 itself in the current setting for the following reason. Suppose node $y$ in $T_2$ moves to path $\pi$ which is in the size class $[2^i, 2^{i+1})$. Next, suppose there are $2^i$ partial insertions, each of which has increased the count at $y$, thus forcing it into a centroid path $\pi'$ one size class higher. Further, suppose that not all these partial insertions have done work on the

*Reassign*() process initiated by the move of $y$ to $\pi$. Then this *Reassign*() procedure need not have finished before $y$ leaves $\pi$, thus violating Invariant 1.

However, we will be able to maintain Invariant 1' for the following reason. In the above scenario, note that after $y$ moves to $\pi$ and before it leaves $\pi'$, there must be at least $2^i$ complete insertions into the subtree of $T_2$ rooted at $y$. This follows from Fact 7.1. Thus we can certainly ensure that the above *Reassign*() procedure finishes before $y$ leaves $\pi'$. In fact, we can also ensure that if this procedure does not finish before $y$ leaves $\pi$ for $\pi'$ then the second *Reassign*() procedure initiated by the move to $\pi'$ also finishes before $y$ leaves $\pi'$. The reason for this is the following.

Even if half of the $\theta(2^i \log^3 n)$ work associated with the above $2^i$ insertions is performed before $y$ leaves $\pi$, then this work will suffice to finish the first *Reassign*() process. And if less than half of this work is performed before $y$ leaves $\pi$, more than half of it must be performed after $y$ moves to $\pi$ and before it leaves $\pi'$. This work will be sufficient to finish both the first and the second *Reassign*() procedures.

Next, Lemmas 7.2 and 7.3 will show parts (a) and (b), respectively, of Invariant 1'.

**Lemma 7.2** *Consider the processes initiated when node $y$ switches to path $\pi$ (if $\pi$ is an existing path then there is a single Reassign process and if $\pi$ is a new path there is an Assign and a Reassign process). $y$ will be either in $\pi$ or in the parent path $\pi'$ of $\pi$ when these processes finish. Further, if these processes finish after $y$ has moved to $\pi'$, then the processes initiated by the move of $y$ to $\pi'$ will also finish before $y$ leaves $\pi'$. In addition, the second set of processes will become active only after the first set of processes finish.*

**Proof.** Assume inductively that the Invariant 1' holds until just before $y$ leaves $\pi'$. Let $\pi$ be in the size range $[2^i, 2^{i+1})$.

By the induction hypothesis, the only *Reassign*($x$) processes in the list at $\pi$ at the point when $y$ switches to $\pi$ are those with the property that $x$ is either in $\pi$ or in the parent path $\pi'$ of $\pi$. Therefore, the sum of the sizes of these processes is $O(2^i)$, even accounting for the next $2^i$ complete insertions into $z'$, the current head of $\pi'$, starting with the insertion which caused $y$ to move to $\pi$ (for even following these insertions $y$ remains in $\pi$ or $\pi'$). Since by assumption Invariant 1' holds while $y$ remains on $\pi$ or $\pi'$, by Lemmas 6.14 and 6.15, Invariant 2 holds at this time too, and so there is at most one *Compact*($\pi$) procedure at any given instant during this time. It follows that the sum of the sizes of all processes in this list, including the processes initiated by the switch of $y$ to $\pi$ and including any *Assign*($\pi$) or *Compact*($\pi$) process, is $O(2^i)$, even accounting for the next $2^i$ complete insertions into $z'$.

Consider the next $2^i$ complete insertions made into $T_{z'}$; by Fact 7.1, $z'$ must be in the parent path of $\pi'$ even after these complete insertions (the recorded size of $z'$ is at most $2^{i+2} - 1 + 2.2^i \leq 2^{i+3} - 1$). Each such insertion will perform $\Theta(\log^3 n)$ work on some process in $list(\pi)$ (recall from point 3 in the actions listed at the beginning of Section 7 that each insertion does work not just on ancestor paths but also on children and grandchildren of ancestor paths which are in a size class one and two smaller, respectively). By Lemma 6.13 along with the induction hypothesis, all the above processes in the list at $\pi$ will be complete by the end of these insertions.

Thus, the *Assign*() or *Reassign*() procedure initiated by $y$ moving to $\pi$ will finish by the time there are $2^i$ complete insertions into $T_{z'}$. By Fact 7.1, these complete insertions must occur while $y$ is still in $\pi$ or its parent. This proves the first part of the lemma.

The second part of the lemma follows easily from the argument in the paragraphs preceding the lemma. The last part can be ensured by preventing activation of any process initiated by the move of a node to $\pi'$ if the process initiated by the move of that node to $\pi$ is not yet complete. This implies that a process might have to wait until a process at a child path finishes. The above argument can still be seen to hold with this restriction; the only change will be that the size bound derived in paragraph 2 of this proof will at most double to include sizes of processes in a child path. Finally, note that if a process in the list at $\pi'$ is waiting for a process in the list at $\pi$ to complete, then the latter process cannot be waiting for another process to complete, by the induction hypothesis. □

**Lemma 7.3** *Consider the Assign, Reassign processes initiated when a node $y$ switches to a new singleton path $\pi$. These processes will finish before any one of the following events happens.*

1. *Some node currently in $\pi$ changes paths twice in the future.*
2. *A new singleton path $\pi'$ becomes the parent of $\pi$ in the future, and subsequently some node moves out of $\pi'$ or a new singleton path becomes the parent of $\pi'$.*

*In addition, if the latter event above happens first and the above processes finish only after $\pi'$ comes into existence, then the Assign, Reassign processes initiated by the creation of $\pi'$ will also finish before the latter event occurs; further, these processes will become active only after the first set of processes have finished.*

**Proof.** Assume inductively that Invariant 1' holds until just before the first of the two events in the lemma occurs. Suppose $\pi$ is in the size range $[2^i, 2^{i+1})$.

The only processes in the list at $\pi$ at the instant when $y$ switches to $\pi$ are $Assign(\pi)$ and $Reassign(y)$. However, more processes could get added over subsequent insertions. Consider the next $2^i$ complete insertions into $z'$, which is the current head of the current parent of $\pi$ if the first event in the lemma occurs earlier, and the head of $\pi'$ if the second event in the lemma occurs earlier. By Fact 7.1, neither of the two events in the lemma could have occurred by the end of these insertions; further $z'$ will still be in the parent or grandparent path of $\pi$ by the end of these insertions.

By the induction hypothesis, the total sum of the sizes of $Reassign()$ processes in $list(\pi)$ is $O(2^i)$, even accounting for the above $2^i$ complete insertions. The sum of the sizes of all processes in this list, including the processes initiated by the switch of $y$ to $\pi$ and including any $Assign(\pi)$ or $Compact(\pi)$ process, is $O(2^i)$ (as in the proof of Lemma 7.2, there exists only one $Compact(\pi)$ procedure at any given instant), even accounting for the next $2^i$ complete insertions into $z'$.

The first part of the lemma follows from Lemma 6.13 in conjunction with the induction hypothesis along with the fact that each of the above insertions performs $\Theta(\log^3 n)$ work on some process in $list(\pi)$.

The remainder of the proof is identical to that for Lemma 7.2. □

### 7.2.2 Tracing Paths and Maintaining Counts

Due to the processing of insertions into $T_1$ being spread out over insertions into $T_2$, counts at centroid paths on the path from the root to the insertion site $x$ must be updated over a period of time.

Let $y$ be the parent of the node $x$ to be inserted. $Insert(x)$ traces the path down from the root to $y$ as follows. It starts at the centroid path containing the root and updates the count at this path. In general, suppose it has just updated the count at path $\pi$; in the process, suppose it has also determined the bottommost node $z$ on $\pi$ which is an ancestor of $y$. The next path $\pi'$ and the associated node $z'$ are found in $O(1)$ time as follows.

Note that by the time $Insert(x)$ comes around to determining $\pi'$, $\pi$ might have moved an arbitrary distance below $z$. Thus $\pi'$ must be found using only $z$ and not $\pi$. Let $\pi''$ be the path containing $z$ currently.

Then $\pi'$ is that child of $\pi''$ which lies on the path from $z$ to $y$. To determine $\pi'$, $Insert(x)$ first determines the LCA of $y$ and the head $u$ of that child path of $\pi''$ which is in a size class one smaller than $\pi''$ (note that there could be two such child paths, in which case, any one is considered). If this LCA is $u$ itself then $\pi'$ is the path containing $u$. Otherwise, $\pi'$ is the path containing that off-path child of this LCA (with respect to $\pi''$) which is not the same as $u$. The associated node $z'$ can be determined similarly.

We now turn to the maintenance of counts. As mentioned above, after $Insert(x)$ increments the count at $\pi$ and before it increments the count at the next centroid path, $\pi$ might have moved an arbitrary distance below $z$. This causes the following problem. Consider that off-path child $z_1$ of $z$ with respect to $\pi$ which is an ancestor of $y$. Suppose $Insert(x)$ has incremented the count at $\pi$ but not at the path containing $z_1$. Consider the situation when $z$ moves from path $\pi$ to path $\pi_1$. Then some of the count at path $\pi$ needs to be transferred to path $\pi_1$.

In order to facilitate this transfer, we keep a *partial counter* at each node of $\pi$. The partial counter for $z$ records the number of increments of $\pi$'s count that have get to increment the counter for $z$'s off-path child $\pi'$. The partial counter for $z$ is incremented when $Insert(x)$ increments the count at $\pi$. Subsequently, when $Insert(x)$ increments the count at the next path $\pi'$ (as in the previous paragraph), the partial counter at $z$ is decremented. And if $z$ moves off $\pi$ in this time interval, the count at $\pi$ is decremented by the partial count at $z$ plus the count at the centroid path rooted at the off-path child of $z$, with respect to $\pi$. This operation takes only constant time and is done by that $Insert(x)$ process which moves $z$ off $\pi$.

Thus $Insert(x)$ can trace the path down from the root to $x$ and update counts on the centroid paths encountered, in $O(1)$ time per path encountered.

### 7.2.3 Constructing Codes

$Insert(x)$ constructs the code for $x$ using the code for the child/parent of $x$ (see Section 6.5), after all the above operations have been performed. Constructing this code and the associated annotations takes $O(\log n)$ operations, which are performed over a series of insertions into the subtree $S$ of $T_1$. Thus, there is a chance that while the code for a node is being read by one process, another procedure *Assign*(), *Reassign*(), or *Compact*() is modifying this code. To prevent such situations, all modifications to codes are actually made on copies of the code, with the copies replacing their respective originals after the modification is complete. This also facilitates answering LCA queries which appear while the modification is being done. Our blocking protocol ensures that two processes do not attempt to modify a code simultaneously.

### 7.2.4 Traversing Blocking Lists

Recall that in our $O(\log^4 n)$ time algorithm, a process $\mathcal{P}$ traverses the entire blocking list it lies in, finds the head process in this list, and does $O(\log n)$ work on this head process. This takes $O(\log n)$ time, which is also the maximum size of any blocking list. In the current setting, $\mathcal{P}$'s traversal of the blocking list will be spread out over several insertions into $T_1$, each relevant insertion incrementing this traversal by one step. The following problem arises now.

Suppose $\mathcal{P}$'s traversal has currently reached a node representing process $\mathcal{Q}$ in the blocking list. Further, due to other processes on this blocking list helping $\mathcal{Q}$, $\mathcal{Q}$ finishes and is taken off the list before $\mathcal{P}$ can do any work on $\mathcal{Q}$. Then the work done by $\mathcal{P}$'s traversal does not contribute to helping any process and thus, cannot be accounted for. Note that this wasteful work is at most $O(\log n)$; further, for each process $\mathcal{Q}$, this wasted work is $O(\log^2 n)$, $O(\log n)$ per process in the blocking list. We can charge this work to $\mathcal{Q}$ itself provided the size of $\mathcal{Q}$ is at least $\log^2 n$. However, this may not be the case.

The easiest way to handle the above problem is to work with subtrees of size $\theta(\log\log^5 n)$ in $T_0$ and $\theta(\log^5 n)$ in $T_1$, as opposed to $\theta(\log\log^4 n)$ and $\theta(\log^4 n)$ earlier. Since the work done on a process of size $2^i$ was $O(2^i \log n)$ earlier (see the first paragraph of the proof of Lemma 6.12: traversing the blocking list and updating the codes at each of the $2^i$ nodes required $O(2^i \log n)$ time), it will now become possible to afford $\theta(2^i \log^2 n)$ work. The wasted work of $O(\log^2 n)$ in the previous paragraph can now be charged to $\mathcal{Q}$, even if $\mathcal{Q}$ has size 1. It can be easily verified that the above change does not affect the asymptotic complexity of our algorithm.

Finally, we give the algorithm for $\mathcal{P}$'s traversal. Each unit of work on $\mathcal{P}$'s traversal will increment this traversal by one step, provided the current process in the blocking list reached by this traversal is an unfinished process. In case the current process in the blocking list reached by this traversal is an already finished process, then this traversal is abandoned and a new traversal begins from the node for $\mathcal{P}$.

# 8 Handling Deletions

We will not perform deletions explicitly. Instead deleted nodes will just be marked as such. Thus the size of the current tree would include the count of both deleted and non-deleted nodes. We will periodically reconstruct the data structure from scratch so as to maintain the following invariants.

The number of deleted nodes is at most $n/2$ and the size of the tree is in the range $n \dots 2n$, where $n$ is the number of undeleted nodes at the moment the previous reconstruction began. If the number of deleted nodes exceeds $n/4$ then the entire data structure will be reconstructed. Similarly, when the size of the tree exceeds $7n/4$, the data structure is reconstructed. Deleted nodes will be removed in the reconstruction process. Reconstruction will be performed over the next $\frac{n}{8}$ insertions and deletions, a constant number of operations per insertion/deletion. There will be at most one reconstruction process at any instant. During the reconstruction process, both the current data structure and the data structure being reconstructed are maintained. This is to enable processing of queries, which are still processed using the current data structure, until the reconstruction is complete. At this point, the current data structure is abandoned.

# References

[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.

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

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

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

[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.