# Combine and Conquer

Robert F. Cohen and Roberto Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912

**CS-92-19**
April 1992

# Combine and Conquer *

Robert F. Cohen
Roberto Tamassia
Department of Computer Science
Brown University
Providence, Rhode Island 02912–1910
(rfc@cs.brown.edu, rt@cs.brown.edu)

**Abstract**

We present a general technique for dynamizing a class of problems whose underlying structure is a computation graph embedded in a tree. We associate values, called *attributes*, with the nodes, paths, and subtrees of our trees. Path attributes form a *path attribute system*, if they are maintained in constant time under path concatenation. Additionally, attributes form a *tree attribute system* if the tree attributes of the tail of a path $\Pi$ are determined in constant time from the path attributes of $\Pi$.

We also introduce a new data structure called a *linear attribute grammar*. An *attribute grammar* is a tree-based expression where the values a node $\mu$ are calculated from the values at the parent, siblings, and/or the children of $\mu$. A linear attribute grammar, is an attribute grammar where all dependencies are linear.

Our contributions can be summarized as follows. We provide a framework for maintaining attribute systems on trees in a fully dynamic environment. We show that given a semiring $\mathcal{S}$, a set of linear expressions with binary and unary operators over $\mathcal{S}^k$ can be dynamically maintained in a fully dynamic environment using linear space and logarithmic time per operation. We show that a linear attribute grammar can be dynamically maintained in a fully dynamic environment using linear space and logarithmic time per operation. Finally, we present many examples of application of our technique to dynamic graph and geometric problems.

(April 1992)

*Research supported in part by the National Science Foundation under grant CCR-9007851, by the U.S. Army Research Office under grant DAAL03-91-G-0035, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052, ARPA order 8225.

# 1 Introduction

The development of dynamic algorithms and data structures is a challenging area of research that has attracted increasing interest in the last years. While considerable progress has been made in geometric searching problems (see, e.g., the survey paper [5]), considerably fewer results exist on the dynamization of graph algorithms. A crucial development in the area of dynamic geometric searching has been the identification of general methods that apply to a large class of problems. In particular, the techniques developed by Overmars et al. (summarized in [21]) for the class of decomposable search problems constitute a fundamental contribution toward the dynamization of large classes of geometric problems. The availability of such general techniques appears instead to be lacking in the area of dynamic graph algorithms. The goal of this research is exactly to provide such generalized techniques in the realm of dynamic tree and graph problems.

Our approach is motivated by the observation that a number of dynamic graph algorithms [10,11,13,14,15,16,19,23,31], developed mostly for connectivity problems, appear to be based on the following fundamental idea: Decompose a graph into subgraphs with limited overlap, and represent such a decomposition by means of a tree so that dynamic operations on the graph are reflected into corresponding dynamic tree operations, which are in turn supported by variations of the link-cut trees of Sleator and Tarjan [28].

We associate values, called *attributes*, with the nodes, paths, and subtrees of our trees. Path attributes form a *path attribute system*, if they are maintained in constant time under path concatenation. Additionally, attributes form a *tree attribute system* if the tree attributes of the tail of a path $\Pi$ are determined in constant time from the path attributes of $\Pi$.

We also introduce a new data structure called a *linear attribute grammar*. An *attribute grammar* is a tree-based expression where the values a node $\mu$ are calculated from the values at the parent, siblings, and/or the children of $\mu$. A linear attribute grammar, is an attribute grammar where all dependencies are linear.

Our contributions can be summarized as follows:

- We provide a framework for maintaining attribute systems on trees in a fully dynamic environment. Our technique extends and generalizes the dynamic trees of [28] and the dynamic expression trees of [9]. Our technique also extends and generalizes also the work on order-decomposable search problems by Overmars [Overmars Notes 83].
- We show that given a semiring $\mathcal{S}$, a set of linear expressions with binary and unary operators over $\mathcal{S}^k$ can be dynamically maintained in a fully dynamic environment using linear space and logarithmic time per operation.
- We show that a linear attribute grammar can be dynamically maintained in a fully dynamic environment using linear space and logarithmic time per operation. Linear attribute grammars can be used as the data structure for dynamic algorithms for several problems in graph drawing.

The rest of this paper is organized as follows. Section 2 describes fully dynamic algorithms to maintain attributes on paths and trees. Separate algorithms are presented for trees of bounded and unbounded degrees. Section 3.1 applies the results of the previous section to present a fully dynamic algorithm for maintaining the solutions of linear expressions. Section 3.2 presents a fully dynamic algorithm for linear attribute grammars. Finally, section 4 presents other applications including two types of generalized heaps.

# 2 Attribute Systems on Paths and Trees

In this section we present a framework for dynamically maintaining an important class of path and tree attributes. We introduce the concept of path attribute systems and tree attribute systems which significantly extend and generalize the dynamic trees of [28]. We begin by discussing dynamic algorithms on a collection of paths, and then show that trees can be maintained as a collection of paths.

## 2.1 Path Attribute Systems

Paths are directed. The first and last nodes of a path $\Pi$ are called the *head* and *tail* of $\Pi$, respectively, and are denoted with $head(\Pi)$ and $tail(\Pi)$. The *reversed path* $\overline{\Pi}$ of $\Pi$ is the path obtained by reversing all edges of $\Pi$. Formally, we have:

**Definition 1** *A* path *is a collection of nodes and directed edges such that:*

- *The empty path with no nodes is a path.*
- *A single node $\mu$ is a path with* $\mathrm{head}(\mu) = \mathrm{tail}(\mu) = \mu$.
- *If $\Pi'$ and $\Pi''$ are paths then the concatenation of $\Pi'$ and $\Pi''$ is a path $\Pi$ formed by adding a directed edge from* $\mathrm{tail}(\Pi')$ *to* $\mathrm{head}(\Pi'')$.

**Definition 2** *A* node attribute *$N$ is a function on nodes. The values $N$ can take are arbitrary, but can be stored in $O(1)$ space. A* path attribute *$P$ is a function on paths. The value of $P(\Pi)$ for a path $\Pi$ is dependent on the values of $N(\mu)$ for each node $\mu$ on $\Pi$, and on the order of the nodes of $\Pi$. The value of $P(\Pi)$ can be stored in $O(1)$ space.*

**Definition 3** *If the values of a node attribute $N$ are taken from a monoid, then $N$ is said to be* globally updatable*, otherwise $N$ is* locally updatable. *We define $\mathcal{N}_G(\mu)$ and $\mathcal{N}_L(\mu)$ to be the set of globally and locally updatable node attributes of $\mu$.*

**Definition 4** *A* node attribute set *$\mathcal{N}$ is a finite collection of node attributes $N_1, \ldots, N_r$. Similarly, a* path attribute set *$\mathcal{P}$ is a finite collection of path attributes $P_1, \ldots, P_s$. For a node $\mu$, the* value *of $\mathcal{N}(\mu)$ is the vector $(N_1(\mu), \ldots, N_r(\mu))$. For a path $\Pi$, the value of $\mathcal{P}(\Pi)$ is the vector $(P_1(\Pi), \ldots, P_s(\Pi))$.*

For each path $\Pi$, we consider the node attribute sets $\mathcal{N}(head(\Pi))$ and $\mathcal{N}(tail(\Pi))$ to be included in path attribute set $\mathcal{P}(\Pi)$.

**Definition 5** *Suppose $\mathcal{N}$ is a node attribute set $\mathcal{P}$ is a path attribute set, and $F : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$ is a function . The triple $(\mathcal{N}, \mathcal{P}, F)$ is a* path attribute system *$\mathcal{Q}$ if for any path $\Pi$ that is the concatenation of paths $\Pi'$ and $\Pi''$, $\mathcal{P}(\Pi)$ can be determined in $O(1)$ time as $F(\mathcal{P}(\Pi'), \mathcal{P}(\Pi''))$. Function $F$ is called the* concatenation function *of $\mathcal{Q}$.*

**Example 1** *Suppose for each node $\mu$ we keep the following node attribute:*

- $\mathrm{weight}(\mu)$ *— the real-valued weight of node $\mu$.*

*Additionally, we have the following path attributes for a path $\Pi$:*

- $\text{sum}(\Pi)$ — *the sum of the weights of the nodes of* $\Pi$.
- $\text{squaredsum}(\Pi)$ — *the square of the sum of the weights of the nodes of* $\Pi$.

*Suppose path* $\Pi$ *is the concatenation of paths* $\Pi'$ *and* $\Pi''$. *If our path attribute set* $\mathcal{P}(\Pi)$ *consists solely of* $\text{squaredsum}(\Pi)$*, then we can not find a concatenation function to maintain* $\mathcal{P}(\Pi)$*, since we cannot determine* $\text{squaredsum}(\Pi)$ *from* $\text{squaredsum}(\Pi')$ *and* $\text{squaredsum}(\Pi'')$. *However, if* $\mathcal{P}(\Pi)$ *is the pair* $(\text{sum}(\Pi), \text{squaredsum}(\Pi))$*, then we can use concatenation function* $F$ *such that:*

$$\begin{aligned} \text{sum}(\Pi) &= \text{sum}(\Pi') + \text{sum}(\Pi'') \\ \text{squaredsum}(\Pi) &= (\text{sum}(\Pi))^2 \end{aligned}$$

Each node attribute $N$ has a *secondary value* $N^*(\mu)$ in addition to its "normal" *primary value* $N(\mu)$. For node attribute set $\mathcal{N} = (N_1, \ldots, N_r)$, $\mathcal{N}^*$ denotes $(N_1^*, \ldots, N_r^*)$. We also have corresponding secondary values for path attributes, and similarly define $\mathcal{P}^*(\Pi)$ for a path $\Pi$.

A decomposable search problem $\pi$ [21] on a set $D$, locates a distinguished element $x = \pi(D)$ with the conditions of given any partition of $D$ into subsets $D'$ and $D''$, we can determine in constant time if $x$ is in $D'$ or $D''$. A number of methods are presented to maintain the solution for a decomposable search problem.

We extend this notion to paths as follows:

**Definition 6** *Given a path attribute system* $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$*, a* path-selection query $Q$ *for* $\mathcal{Q}$ *maps a path* $\Pi$ *and a query argument* $q$ *into a node* $\mu = Q(\Pi, q)$ *of* $\Pi$. *Suppose path* $\Pi$ *is the concatenation of* $\Pi'$ *and* $\Pi''$. *A* path-selection function $S(\Pi, q)$ *for* $Q$*, determines in* $O(1)$ *time whether* $\mu$ *is in* $\Pi'$ *or* $\Pi''$ *from* $q$ *and the values* $\mathcal{P}(\Pi')$ *and* $\mathcal{P}(\Pi'')$.

We will show that we can perform a selection query given the associated selection function.

We consider a rather general dynamic environment for a set of paths equipped with a path attribute system $\mathcal{Q}$ and a collection $\mathcal{S}$ of path-selection functions. Update operations include: *splitting*, *concatenating*, and *reversing paths*; and *updating node attributes*; and *exchanging* primary and secondary attribute values for all the nodes on a path. Query operations include: *evaluating* node and path attributes, and *computing path-selection functions*.

We allow both "local" and "global" updates of node attributes. A *local update* consists of changing the value $N(\mu)$ of a single node $\mu$. A *global update* can be performed on a globally updatable node attribute $N$ with values taken from a monoid with operator $\odot$, and consists of applying an incremental change $\delta$ to the attribute of every node in a path $\Pi$. Formally, a global update sets $N(\mu) = N(\mu) \odot \delta$ for every node $\mu$ of $\Pi$, where we assume that $x \odot y$ can be computed in $O(1)$ time. If a path attribute $P$ depends on a globally updatable node attribute $N$, we require that the new value of $P(\Pi)$ can be computed in $O(1)$ time given the variation $\delta$ of $N$ in $\Pi$.

We support the following repertory of operations on a set of (uniquely identified) paths equipped with a path attribute system $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$ and a collection $\mathcal{F}$ of path-selection functions for $\mathcal{Q}$:

- *makepath*(*node* $\nu$; *value* $x$) — return a new path consisting of a single node $\mu$ and set $\mathcal{N}(\nu) = x$.
- *deletepath*(*node* $\nu$) — remove the single-node path consisting of node $\nu$.

- *getpath(node* $\nu$*)* — return the unique identifier of the path containing $\nu$.
- *head(path* $\Pi$*)* — return the head of path $\Pi$.
- *tail(path* $\Pi$*)* — return the tail of path $\Pi$.
- *before(node* $\mu$*, integer* $k$*)* — return the $k$ nodes preceding $\mu$ on *getpath*$(\mu)$. If there are fewer than $k$ nodes preceding $\mu$, then return the nodes found. If $\mu$ is the head of its path then return *nil*.
- *after(node* $\mu$*, integer* $k$*)* — return the $k$ nodes succeeding $\mu$ on *getpath*$(\mu)$. If there are fewer than $k$ nodes succeeding $\mu$, then return the nodes found. If $\mu$ is the tail of its path then return *nil*.
- *precedes(node* $\mu, \nu$*)* — return **true** if nodes $\mu$ and $\nu$ are on the same path with $\mu$ preceding $\nu$. Otherwise, returns **false**.
- *pathreport(path* $\Pi$*)* — report the value of path attribute set $\mathcal{P}$ for path $\Pi$.
- *nodereport(node* $\mu$*)* — report the value of node attribute set $\mathcal{N}$ for node $\mu$.
- *concatenate(path* $\Pi', \Pi''$*)* — concatenate paths $\Pi'$ and $\Pi''$ to get path $\Pi$.
- *split(node* $\mu$*)* — split path $\Pi$ into three subpaths by removing the edges from $\mu$ to *before*$(\mu, 1)$ and $\mu$ to *after*$(\mu, 1)$.
- *reverse(path* $\Pi$*)* — reverse the direction of path $\Pi$, i.e., set $\Pi = \overline{\Pi}$.
- *exchange(path* $\Pi$*)* — exchange the values $\mathcal{N}(\mu)$ and $\mathcal{N}^*(\mu)$ for each node $\mu$ on $\Pi$.
- *localupdate(node* $\mu$*, nodeattribute* $N$*, value* $x$*)* — set the value of node attribute $N$ of $\mathcal{N}$ to be $x$.
- *globalupdate(path* $\Pi$*, nodeattribute* $N$*, value* $\delta$*)* — for each node $\mu$ on path $\Pi$, set $N(\mu) = N(\mu) \odot \delta$, where $N$ is a globally updatable node attribute with monoid operator $\odot$.
- *find(path* $\Pi$*, selectionfunction* $S$*, value* $q$*)* — find node $\mu$ of $\Pi$ returned by the path-query associated with path-selection function $S$ using query argument $q$.

Our data structure consists of representing each path $\Pi$ as a balanced binary tree $\mathcal{B}_\Pi$, called the *path-tree* of $\Pi$, and identifying $\Pi$ by the root $\zeta$ of $\mathcal{B}_\Pi$. Each leaf $\lambda$ of $\mathcal{B}_\Pi$ represents a node of $\Pi$, and the left-to-right order of the leaves of $\mathcal{B}_\Pi$ corresponds to the head-to-tail order of the nodes of $\Pi$. We store at $\lambda$ the values $\mathcal{N}(\lambda)$ and $\mathcal{N}^*(\lambda)$. For a globally updatable node attribute $N_i$ of $\mathcal{N}$, the value stored at $\lambda$ is not the actual value of $N_i$, but a "partial value" that can be used to compute $N_i(\lambda)$, as will become apparent later.

Each internal node $\eta$ of $\mathcal{B}_\Pi$ represents the subpath $\Pi(\eta)$ of $\Pi$ associated with the leaves in the subtree of $\eta$. Note that we identify path $\Pi(\eta)$ with $\eta$. We store at $\eta$ pointers *head*$(\eta)$ and *tail*$(\eta)$ to the head and tail of $\Pi(\eta)$. Additionally, we store at node $\eta$ the four values $\mathcal{P}(\Pi(\eta))$, $\mathcal{P}(\overline{\Pi(\eta)})$, $\mathcal{P}^*(\Pi(\eta))$, and $\mathcal{P}^*(\overline{\Pi(\eta)})$. Notice that we can consider the pointers *head* and *tail* as part of path attribute set $\mathcal{P}$. For the path attributes that depend on globally updatable node attributes, we store partial values. We also keep in $\eta$ a vector $\Delta(\eta)$ (and a secondary vector $\Delta^*(\eta)$) such that for any globally updatable node attribute $N_i$ of $\mathcal{N}$ the $i$-th component $\Delta_i(\eta)$ represents a change to the value of $N_i$ for each node $\mu$ on $\Pi(\eta)$. (If $N_j$ is a locally updatable node attribute, then we conventionally set $\Delta_j(\eta) = 0$.) Therefore, for any node $\mu$ of $\Pi$, $N_i(\mu)$ is the combination (using operation $\odot_i$, the monoid operator for values of $N_i$) of the "partial value" of $N_i$ stored at $\mu$ and of the "offsets" $\Delta_i(\eta)$ of all the nodes $\eta$

on the path from $\mu$ to the root $\zeta$ of $\mathcal{B}_\Pi$. Similarly, $\mathcal{P}(\mu)$ can be computed from the partial value of $\mathcal{P}$ stored at $\mu$ and the "offsets" $\Delta(\eta)$ of all the nodes $\eta$ on the path from $\mu$ to $\zeta$ in $\mathcal{B}_\Pi$. (Analogous arguments apply to the reverse and secondary values of the attributes.)

At each node $\mu$ of a path $\Pi$, we store pointers $left(\mu)$ and $right(\mu)$ to the left and right neighbors of $\mu$ on $\Pi$. At each path-tree node $\eta$ we store pointers to $head(\Pi(\eta))$ and $tail(\Pi(\eta))$. We keep at node $\mu$ two globally updatable bits $reversed(\mu)$ and $exchanged(\mu)$. At each path tree node $\eta$, we keep the offset values $reversed(\eta)$ and $exchanged(\eta)$. If $reversed(\mu) = 1$, then $left(\mu)$ points to the right neighbor of $\mu$ and $right(\mu)$ points to the left neighbor of $\mu$. Similarly, if $exchanged(\mu) = 1$, then the meaning of $\mathcal{N}(\mu)$ and $\mathcal{N}^*(\mu)$ is swapped. The meaning of the partial values $reversed(\eta)$ and $exchanged(\eta)$ is similar. If $reversed(\eta) = 1$, then $head(\eta)$ points to $tail(\Pi(\eta))$ and $tail(\eta)$ points to $head(\Pi(\eta))$. Also, $reversed(\eta) = 1$ swaps the meaning of the left and right child pointers in the subtree of $\eta$. The values of these two bits also determine the actual meaning of the four values kept for $\mathcal{P}$ at $\eta$.

Operations *makepath* and *deletepath* are trivially implemented in $O(1)$ time. Operation $pathreport(\Pi)$ is performed in $O(1)$ time and consists of returning the value of $\mathcal{P}$, computed from the "partial value" stored at the root $\zeta$ of $\mathcal{B}_\Pi$ and the vector $\Delta(\zeta)$ . Operations $reverse(\zeta)$ and $exchange(\zeta)$ are implemented by flipping the *reversed* and *exchanged* bits at $\eta$. Operations $head(\zeta)$ and $tail(\zeta)$ determine the correct node based on the value of $reversed(\zeta)$ and the *head* and *tail* pointers stored at $\zeta$. Operation $globalupdate(\Pi, N_i, \delta)$ is performed by setting $\Delta_i(\zeta) = \Delta_i(\zeta) \odot \delta$, where $\zeta$ is the root of the path-tree of $\Pi$. Each of these operations is implemented in $O(1)$ time.

Operation $getpath(\mu)$ is implemented by following pointers from $\mu$ to the root of the path-tree containing $\mu$. Operation $before(\mu, k)$ is implemented by calling operation *push* on each node of the path from $getpath(\mu)$ to $\mu$. The immediate predecessor of $\mu$ is then pointed at by $left(\mu)$. We use the *left* and *right* pointers in the following way to return the remaining (at most) $k-1$ predecessors of $\mu$: Suppose we have returned $i < k$ nodes with $\mu_{i-1}$ and $\mu_i$ the $i-1$st and $i$th nodes returned. Then one of $left(\mu_i)$ and $right(\mu_i)$ will point to $\mu_{i-1}$. The other will point to the predecessor of $\mu_i$. (We cannot simply follow the *left* pointers, since each $\mu_i$ only stores a partial value of $reversed(\mu_i)$.) The implementation of operation $after(\mu, k)$ is symmetric. Each of these operations takes $O(d_\mu + k)$ time where $d_\mu$ is the depth of $\mu$ in $\mathcal{B}_\Pi$. Note that for a path of length $\ell$, $d_\mu$ is $O(\log \ell)$.

Operation $precedes(\mu, \nu)$ is implemented by finding the least common ancestor $\eta$ of $\mu$ and $\nu$ in the path tree containing both $\mu$ and $\nu$. We then calculate the reversal state of $\eta$ to determine which of $\mu$ and $\nu$ is a descendant of the left child of $\eta$. If $\mu$ is a descendant of the left child then return **true** else return **false**.

The following operation moves the values of $\Delta$ towards the leaves of a path-tree in $O(1)$ time:

- $push(node\ \eta)$ — For internal path-tree node $\eta$ with children $\eta'$ and $\eta''$ combine $\Delta(\eta')$ and $\Delta(\eta'')$ with $\Delta(\eta)$ and set $\Delta(\eta) = 0$.

Operation $push(\eta)$ is implemented by using the monoid operator for each globally updatable node attribute to apply the partial value at $\eta$ to the partial values at $\eta'$ and $\eta''$. We also swap values depending on $reversed(\eta)$ and $exchanged(\eta)$. If $reversed(\eta) = 1$, then we swap pointers $head(\eta)$ and $tail(\eta)$, the values $\mathcal{P}(\eta)$ and $\overline{\mathcal{P}(\eta)}$, and the values $\mathcal{P}^*(\eta)$ and $\overline{\mathcal{P}^*(\eta)}$. If $exchanged(\eta) = 1$, then we swap the values $\mathcal{P}(\eta)$ and $\mathcal{P}^*(\eta)$, and the values $\overline{\mathcal{P}(\eta)}$

and $\overline{\mathcal{P}^*(\eta)}$. If $\eta$ is a tree node $\mu$, then it is a leaf of its path tree. If $reversed(\mu) = 1$, we swap pointers $left(\mu)$ and $right(\mu)$. If $exchanged(\mu) = 1$, then we swap the values $\mathcal{N}(\mu)$ and $\mathcal{N}^*(\mu)$. (Equivalently, we can include the pointers *left* and *right* in node attribute set $\mathcal{N}$ and the pointers *head* and *tail* in path attribute set $\mathcal{P}$. In this way, operation *push* would only swap attribute sets, not individual pointers.)

Therefore, operation $nodereport(\mu)$ is implemented by finding the root $\zeta$ of the path-tree containing $\mu$ and then calling *push* along this path starting at $\zeta$. In this way, the partial values of the globally updatable node attributes of $\mathcal{N}(\mu)$ will be the actual values. Therefore, we then return $\mathcal{N}(\mu)$.

Operations *concatenate* and *split* use the following *elementary tree operations* which can be performed in $O(1)$ time:

- $join(node\ \zeta', \zeta'')$ — Given the roots $\zeta'$ and $\zeta''$ of the path-trees for solid paths $\Pi'$ and $\Pi''$, combine the trees into a new tree by creating a new root $\zeta$ with left child $\zeta'$ and right child $\zeta''$. We set the value of $\Delta(\zeta)$ to be 0. We set the *left* and *right* neighbor pointers at $tail(\zeta')$ and $head(\zeta'')$ to point at each other. Choose the one of *left* or *right* that is initially *nil* at $tail(\zeta')$ and set that value to $head(\zeta'')$. If both *left* and *right* are *nil* at $tail(\zeta')$ then we can determine in constant time the value of *reversed* at $tail(\zeta')$, so we set the *right* pointer to be $head(\zeta')$. We perform a similar update at $head(\zeta'')$. We calculate the values of $\mathcal{P}$ stored at $\zeta$ in $O(1)$ time using concatenation function $F$.
- $separate(node\ \zeta)$ — Given the root $\zeta$ of a binary tree, first call $push(\zeta)$ then divide the tree into two trees with roots $\zeta'$ and $\zeta''$, where $\zeta'$ is the root of the left subtree and $\zeta''$ is the root of the right subtree. The values of $\mathcal{P}$ at $\zeta'$ and $\zeta''$ do not change.
- $rotateleft(node\ \eta)$ ($rotateright(node\ \eta)$) — Perform a left (right) rotation at node $\eta$. This operation can be performed with $O(1)$ *separate* and *join* operations.

We conclude operation $split(\mu)$ by calling $push(\mu)$. This step is not needed to maintain the values of paths, yet is needed in our implementation of trees (see section 2.2).

Operation $localupdate(\mu, x)$ is implemented by first replacing the value of $\mathcal{N}(\mu)$ with $x$. Suppose path tree $\mathcal{B}$, with root $\zeta$, contains node $\mu$. Suppose $\eta$ is a path tree node in $\mathcal{B}$. The value of $\mathcal{P}(\eta)$ may change only if $\eta$ is on the path from $\mu$ to $\zeta$ in $\mathcal{B}$. Therefore, we calculate these values starting at $\mu$ and ending at $\zeta$. This takes $O(d_\mu)$ time where $d_\mu$ is the depth of $\mu$ in $\mathcal{B}$.

We need the following operation from [29] in our implementation of $find(\Pi, S, q)$:

- $splaystep(node\ \eta)$ — For binary tree $\mathcal{B}$ with root $\zeta$ and node $\eta$ a grandchild of $\zeta$, restructure $\mathcal{B}$ such that the relative order of the leaves of $\mathcal{B}$ remain fixed, and every node in the subtree rooted at $\eta$ has its depth reduced by one or two. We accomplish this as follows. Let $\eta'$ be the child of $\zeta$ that is the parent of $\eta$. If $\eta'$ and $\eta$ are both left children then perform $rotateright(\zeta)$ twice. If $\eta'$ is a left child and $\eta$ is a right child, then perform $rotateleft(\eta')$ followed by $rotateright(\zeta)$. The other two cases are symmetric. (See Fig. 1.)

Operation *splaystep* is implemented with a constant number of *rotateleft* and *rotateright* operations. Hence, *splaystep* takes $O(1)$ time.

**Lemma 1** *Suppose $S$ is a path-selection function for path attribute set $\mathcal{P}$ and $\zeta$ is the root of a path-tree $\Pi$. Then given query argument $q$, we can determine in $O(1)$ time if $S(\Pi, q)$*

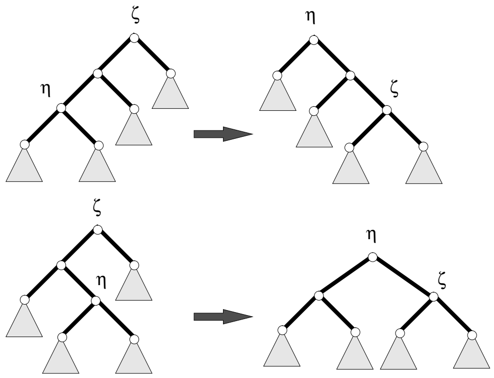


**Figure 1:** The rotations performed in operation *splaystep*$(\eta)$.

*is a child or grandchild of* $\zeta$*, or which grandchild of* $\zeta$ *is the root of the subtree containing* $S(\Pi, \zeta)$*.*

**Proof:** Suppose $\eta_1$ and $\eta_2$ are the children of $\zeta$. By definition, in $O(1)$ time we can determine which subtree rooted at $\eta_1$ or $\eta_2$ contains $S(\Pi, q)$. If the subtree found is a single node, then it is $S(\Pi, q)$. Without loss of generality, suppose we know that $S(\Pi, q)$ is in the subtree rooted at $\eta_1$, and $\eta_1$ is not a leaf. Then let $\eta_{11}$ and $\eta_{12}$ be the children of $\eta_1$. We then issue *splaystep*$(\eta_{11})$ to restructure $\mathcal{B}_\Pi$ such that $\eta_{11}$ is the left child of the root. Then, in $O(1)$ time we can determine if $S(\Pi, q)$ is in the subtree rooted at $\eta_{11}$. Otherwise, $S(\Pi, q)$ is in the subtree rooted at $\eta_{12}$. □

Operation *find*$(\zeta)$ is implemented as follows, starting at node $\zeta$ and repeating until a value is returned: if $\mu$ is a child or grandchild of $\zeta$ then return $\mu$. Otherwise, determine the subtree rooted at the grandchild $\eta$ of $\zeta$ which contains $\mu$. Do *splaystep* at $\eta$ and recur. After $\mu$ is found, we undo all the *splaystep*s to restore the balance of the path-tree. This takes $O(d_\mu)$ time where $d_\mu$ is the depth of the returned node $\mu$.

**Theorem 1** *Let* $\mathcal{Q} = (\mathcal{P}, \mathcal{N}, F)$ *be a path attribute system and* $\mathcal{S}$ *a collection of path-selection functions for* $\mathcal{Q}$*. There is a fully dynamic data structure for maintaining* $\mathcal{Q}$ *and* $\mathcal{S}$ *over a set of paths with the following performance:*

1. *a path of length* $\ell$ *uses* $O(\ell)$ *space;*
2. *operations* makepath, deletepath, head, tail, reverse *and* exchange *each take* $O(1)$ *time;*
3. *operations* before *and* after *each take* $O(\log \ell + k)$ *time to return* $k$ *nodes from a path of length* $\ell$*;*
4. *operations* getpath, concatenate, split, precedes, localupdate, globalupdate, pathreport, nodereport *and* find *each take* $O(\log \ell)$ *time for a path of length* $\ell$*.*

**Example 2** *Suppose we associate a value* cost$(\mu)$ *with each node* $\mu$ *of a path* $\Pi$*. We want to answer the following query in a dynamic environment (including operations* reverse, concatenate, *and* join*):*

- FindMin(*path* $\Pi$) — *return the minimum cost node of path* $\Pi$ *closest to* head$(\Pi)$.

*We keep the following value for each path* $\Pi$*:*

- mincost(*path* $\Pi$) — *the minimum cost of any node of path* $\Pi$.

*Therefore, we keep path attribute system* $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$*, where for node* $\mu$ *and path* $\Pi$*,* $\mathcal{N}(\mu) = (\mathrm{cost}(\mu))$ *and* $\mathcal{P}(\Pi) = (\mathrm{mincost}(\Pi))$*. Suppose path* $\Pi$ *is the concatenation of paths* $\Pi'$ *and* $\Pi''$*. Concatenation function* $F$ *calculates* $\mathrm{mincost}(\Pi) = \min(\mathrm{mincost}(\Pi'), \mathrm{mincost}(\Pi''))$*.*

*Operations* FindMin *is the selection query associated with the following path selection function.*

**Selection Function $\mathbf{S_1}$** Suppose $\eta$ is the root of a path tree with children $\eta'$ and $\eta''$. If $\mathit{mincost}(\eta') = \mathit{mincost}(\eta)$ then return $\eta'$, else return $\eta''$.

*Operation* FindMin$(\Pi)$ *is then implemented as* $\mathrm{find}(\Pi, S_1)$*.*

**Example 3** *Suppose each node is associated one of a finite number of possible types. We wish to support the following query.*

- FindLongest(*path* $\Pi$, *nodetype* $X$) — *return the tail of the longest subpath* $\Pi'$ *of* $\Pi$ *such that* $\mathrm{head}(\Pi') = \mathrm{head}(\Pi)$ *and* $\Pi'$ *does not contain a node of type* $X$*.*

*We keep the following path attribute system* $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$*. For each node* $\mu$*, node attribute set* $\mathcal{N}(\mu)$ *consists of the single node attribute* $\mathrm{type}(\mu)$*, the type of node* $\mu$*. For path* $\Pi$*, path attribute set* $\mathcal{P}(\Pi)$ *consists of the set of boolean valued path attributes* $\mathrm{contains}(\Pi, X)$*. The value of* $\mathrm{contains}(\Pi, X)$ *is* **true** *if path* $\Pi$ *contains a node of type* $X$*. Suppose path* $\Pi$ *is the concatenation of paths* $\Pi'$ *and* $\Pi''$*. Concatenation function* $F$ *sets* $\mathrm{contains}(\Pi, X)$ *to be* **true** *if either* $\mathrm{contains}(\Pi', X)$ *is* **true** *or* $\mathrm{contains}(\Pi'', X)$ *is* **true**.

*We utilize the following path selection function with query argument* $X$*.*

**Selection Function $\mathbf{S_2}$** Suppose $\eta$ is the root of a path tree with children $\eta'$ and $\eta''$. If $\mathit{contains}(\eta', X)$ is **true**, or if $\mathit{head}(\eta'')$ has type $X$, then return $\eta'$, else return $\eta''$.

*Query* FindLongest$(\Pi, X)$ *is then implemented as follows. If* $\mathrm{head}(\Pi)$ *has type* $X$*, then return nil. Otherwise, return* $\mathrm{find}(\Pi, S_2, X)$*.*

**Example 4** We consider a problem that arises in dynamically maintaining trees (see Section 2.2). Consider paths whose nodes have positive real weights. A node $\mu$ is *light* if the weight of $\mu$ is greater than the sum of the weights of the predecessors of $\mu$ on $\Pi$. Each path $\Pi$ has at least one light node since $\mathit{head}(\Pi)$ is light. We want to answer the following queries in a dynamic environment (including operations *reverse*, *concatenate*, and *join*):

- *totalpathwt*(*path* $\Pi$) — return the total weight of all the nodes in path $\Pi$.
- *light*(*path* $\Pi$) — return the light node $\mu$ of $\Pi$ closest to $\mathit{tail}(\Pi)$.

Let the *tilt* of a node $\mu$ of path $\Pi$ be the sum of the weights of the nodes preceding $\mu$ in $\Pi$, minus the weight of $\mu$. Therefore a node $\mu$ is light if $\mathit{tilt}(\mu) < 0$. Let $\mathit{mintilt}(\Pi)$ be the minimum value of *tilt* among all nodes of $\Pi$. Notice that $\mathit{totalpathwt}(\Pi) = \mathit{totalpathwt}(\overline{\Pi})$, but in general $\mathit{negtiltnode}(\Pi)$ is different from $\mathit{negtiltnode}(\overline{\Pi})$.

To solve this problem we keep the following path attribute system $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$: the node attribute of a node $\mu$ is the weight of $\mu$, i.e., $\mathcal{N}(\mu) = (\mathit{weight}(\mu))$; the path attributes of a path $\Pi$ are $\mathcal{P}(\Pi) = (\mathit{totalpathwt}(\Pi), \mathit{mintilt}(\Pi))$ (see Fig. 2). Suppose path $\Pi$ is the concatenation of paths $\Pi'$ and $\Pi''$. Concatenation function $F$ calculates $\mathcal{P}(\Pi)$ as follows:

$$\begin{aligned}
\mathit{totalpathwt}(\Pi) &= \mathit{totalpathwt}(\Pi') + \mathit{totalpathwt}(\Pi'') \\
\mathit{mintilt}(\Pi) &= \min(\mathit{mintilt}(\Pi'), \mathit{totalpathwt}(\Pi') - \mathit{weight}(\mathit{head}(\Pi'')), \\
&\quad\ \mathit{totalpathwt}(\Pi') + \mathit{mintilt}(\Pi''))
\end{aligned}$$

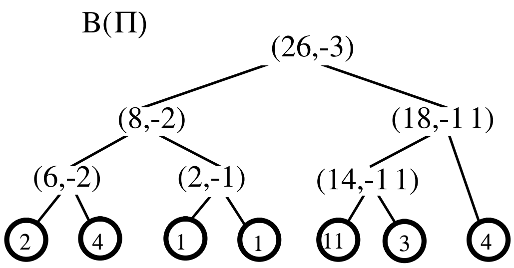


**Figure 2:** Path-tree for the path attribute system of Example 4. Each leaf is labeled with its weight. Each internal node $\eta$ is labeled with $\mathcal{P}(\eta) = (totalpathwt(\eta), mintilt(\eta))$.

Operation $totalpathwt(\Pi)$ is implemented with $pathreport(\Pi)$. Operation $light(\Pi)$ is implemented as a selection-query via the following path-selection function without query argument.

**Selection Function $\mathbf{S_3}$** *Suppose $\eta$ is the root of a path tree with children $\eta'$ and $\eta''$. if* $\text{totalpathwt}(\Pi') - \text{weight}(\text{head}(\Pi'')) < 0$ *or* $\text{totalpathwt}(\Pi') + \text{mintilt}(\Pi'') < 0$ *then return* $\eta''$. *Otherwise, return* $\eta'$.

## 2.2 Tree Attribute Systems

We now discuss a general class of dynamic algorithms on trees. We consider rooted ordered trees with edges directed from the child to parent. Hence, a path in a tree is always directed from a descendant to an ancestor.

**Definition 7** *Rooted tree $T$ has* bounded degree *if there is a constant $d$ such that no node of $T$ has more than $d$ neighbors, otherwise it is of* unbounded degree. *Tree $T$ is* ordered *if for each node $\mu$ the left to right order of the children of $\mu$ is fixed.*

If $T$ is a tree and $\mu$ is a node of $T$ we use the notation $T_\mu$ to indicate the subtree rooted at $\mu$.

We develop our dynamic algorithm for trees in the same manner as for paths. We introduce the concept of tree attribute systems, and show a large collection of dynamic operations on a tree storing a tree attribute system.

**Definition 8** *A node attribute set $\mathcal{N}$ for a tree $T$ consists of a finite collection of node attributes for the nodes of $T$. Similarly, a path attribute set $\mathcal{P}$ for $T$ consists of a fixed collection of path attributes for the paths of $T$. A* tree attribute *$R$ is a function on trees such that, for a tree $T$, $R(T)$ depends on the values $\mathcal{N}(\mu)$ for each node $\mu$ on $T$, on the parent-child relationships between nodes of $T$, and on the order of the children of each node of $T$. We assume that $R(T)$ can be stored in $O(1)$ space. A* tree attribute set *$\mathcal{R}$ is a fixed collection of tree attributes. For a node $\mu$ of tree $T$, $\mathcal{R}(\mu)$ denotes the tree attribute set of the subtree of $T$ rooted at $\mu$.*

We consider both bounded and unbounded degree trees. We first present the algorithm for bounded degree trees, then extend this result to unbounded degree trees.

### 2.2.1 Bounded Degree Trees

In this section, we present a fully dynamic algorithm for tree attribute systems on bounded degree trees. We begin by formally defining a tree attribute system.

**Definition 9** *Suppose $\mathcal{N}$ is a node attribute set, $\mathcal{P}$ is a path attribute set, $\mathcal{R}$ is a tree attribute set, and $F$ is a concatenation function. Given a bounded degree tree $T$ and a path $\Pi$ of $T$, we denote with $\tilde{\mathcal{N}}$ the extended node attribute set that, for a node $\mu$ of $\Pi$, consists of $\mathcal{N}(\mu)$, $\mathcal{R}(\mu')$ for each child $\mu'$ of $\mu$ not on $\Pi$, and the ordering of the children of $\mu$. We say that the triplet $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ is a* tree attribute system *if:*

- *The triple $(\tilde{\mathcal{N}}, \mathcal{P}, F)$ is a path attribute system for the set of paths of $T$ (where concatenations are restricted to paths $\Pi'$ and $\Pi''$ in $T$ such that* $\mathrm{head}(\Pi'')$ *is the parent of* $\mathrm{tail}(\Pi')$ *in $T$).*
- *The value of $\mathcal{R}(\mathrm{head}(\Pi))$ can be determined in constant time from the values $\mathcal{P}(\Pi)$, $\tilde{\mathcal{N}}(\mathrm{head}(\Pi))$, and $\tilde{\mathcal{N}}(\mathrm{tail}(\Pi))$.*

We extend the concept of selection to trees. That is we define a tree selection function, which, if available, allows us to quickly find a distinguished node of tree $T$.

**Definition 10** *Given a tree attribute system $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$, a* tree-selection query *$Q$ for $\mathcal{Z}$ maps tree $T$ and a query argument $q$ into a node $\mu = Q(T, q)$ of $T$. Suppose path $\Pi$ is the concatenation of $\Pi'$ and $\Pi''$ in $T$ such that* $\mathrm{tail}(\Pi)$ *is the root of $T$. Suppose node $\mu'$ is the node closest to* $\mathrm{tail}(\Pi)$ *such that $\mu$ is a descendant of $\mu'$. A* tree-selection function *$S(\Pi, q)$ for $Q$ is a path-selection function to find node $\mu'$.*

**Definition 11** *Consider a tree $T$ containing a node $\nu$ with children $\mu_1, \ldots, \mu_d$ in left-to-right order. Suppose $\Pi$ is a path of $T$ containing both $\nu$ and some child $\mu_i$. The* left-tree *of $\nu$ with respect to $\Pi$ $T_\ell(\nu)$ is defined as follows. The root of $T_\ell(\nu)$ is a node $\nu_\ell$ with $\mathcal{N}(\nu_\ell) = \mathcal{N}(\nu)$. Root $\nu_r$ has children $\mu_1, \ldots, \mu_{i-1}$. The* right-tree *of $\nu$ with respect to $\Pi$ $T_r(\nu)$ is defined similarly. The root of $T_r(\nu)$ is a node $\nu_r$ with $\mathcal{N}(\nu_r) = \mathcal{N}(\nu)$. Root $\nu_r$ has children $\mu_{i+1}, \ldots, \mu_d$.*

We provide three types of global restructuring of trees: reflecting, everting and cycling. Consider a tree $T$. We allow a node $\mu$ of $T$ to be *fixed* indicating that the structure of the subtree rooted at $\mu$ may not be restructured.

**Definition 12** *Given a tree $T$ and a node $\mu$ of $T$ that is not fixed, reflecting $T$ at $\mu$ consists of reversing the left-to-right order of the children of $\mu$ and the children of all descendants that can be reached from $\mu$ without passing through a fixed node.*

**Definition 13** *Given a tree $T$ with node $\mu$ not the root,* everting *$T$ at $\mu$ consists of reversing the parent-child relationships between all nodes on the path $\Pi$ from $\mu$ to the root of $T$. If $\Pi$ contains a fixed node, then everting at $\mu$ is not allowed.*

*Each node $\mu$ also keeps an* eversion rule *which describes changes to the order of the children of $\mu$. Possibilities include:*

- *Simple — the left and right-trees of $\mu$ remain unchanged.*
- *Steady — the left and right-trees of $\mu$ are exchanged.*
- *Mirrored — the left and right-trees of $\mu$ are reflected.*

Consider a node $\mu$, its child $\mu'$, and its parent $\nu$, all on path $\Pi$. Simple eversion at $\mu$ substitutes $\nu$ for $\mu'$ in the ordering of the children of $\mu$. Steady eversion at $\mu$ maintains the clockwise order of the neighbors of $\mu$. Finally, mirrored eversion at $\mu$ reverses the clockwise order of the neighbors of $\mu$. Each node of a tree $T$ may use a different eversion rule. Therefore, we consider the eversion rule a node property.

**Definition 14** *Given a tree $T$ and a node $\mu$ and its parent $\nu$ of $T$ that is not fixed, cycling $\nu$ at $\mu$ consists of cyclicly permuting the children of $\nu$ such that node $\mu$ is the final node in the order.*

Our algorithm supports the following operations on a collection of trees with a tree attribute system $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R})$ and a collection $\mathcal{F}$ of path-selection and tree-selection functions:

- *Evaluate*(*node* $\mu$) — Return the values of $\mathcal{N}(\mu)$ and $\mathcal{R}(\mu)$.
- *PathFind*(*node* $\mu', \mu''$; *selectionfunction* $S$) — Let $\Pi$ be the path of a tree from node $\mu'$ to node $\mu''$. Find the node of $\Pi$ returned by the path-selection function $S$.
- *TreeFind*(*node* $\nu$; *selectionfunction* $S$) — Find the node of the subtree rooted at $\mu$ returned by the tree-selection function $S$.
- *LocalUpdate*(*node* $\mu$, *nodeattribute* $N$, *value* $x$) — Set the value of node attribute $N$ of $\mathcal{N}$ to be $x$.
- *GlobalUpdate*(*node* $\mu', \mu''$, *nodeattribute* $N$, *value* $\delta$) — Let $\Pi$ be the path from $\mu'$ to $\mu''$ in $T$. For each node $\mu$ on path $\Pi$, set $N(\mu) = N(\mu) \odot \delta$, where $\odot$ is the operator of the monoid for $N$.
- *MakeTree*(*value* $x$) — Create a new tree consisting of a single leaf whose node attribute set has value $x$.
- *DeleteTree*(*node* $\lambda$) — Remove the single-node tree consisting of leaf $\lambda$.
- *LCA*(*node* $\mu', \mu''$) — Returns the least common ancestor of nodes $\mu'$ and $\mu''$. Nodes $\mu'$ and $\mu''$ are assumed to be nodes of the same tree.
- *Evert*(*node* $\mu$) — Let $T$ be the tree containing node $\mu$. Evert $T$ at $\mu$.
- *Link*(*node* $\rho, \mu, \mu', \mu''$) — This operation assumes that nodes $\mu$ and $\rho$ are in different trees, and $\rho$ is the root of its tree $T$. Add an edge from $\rho$ to $\mu$ with $\mu'$ its immediate left sibling and $\mu''$ its immediate right sibling. If $\mu'$ or $\mu''$ is *nil* then $\rho$ becomes the first or last child of $\mu$. Tree $T$ becomes a subtree of the tree containing $\mu$.
- *Cut*(*node* $\mu$) — This operation assumes that $\mu$ is not the root of a tree. Remove the edge from $\mu$ to its parent, thus separating the subtree rooted at $\mu$.
- *Parent*(*node* $\mu$) — Return the parent of $\mu$ or *nil* if $\mu$ is a tree root.
- *Sibling*(*node* $\mu$, *integer* $k$, *direction* $X$) — Return the $k$ adjacent siblings of node $\mu$ in the $X$ direction, where $X$ is either *left* or *right*. If there are fewer than $k$ siblings in the $X$ direction, then return the nodes found. If there are no siblings in the $X$ direction, then return *nil*.
- *Reflect*(*node* $\mu$) — Reflect the subtree rooted at $\mu$.
- *Cycle*(*node* $\nu, \mu$) — Cyclicly permute the children of $\nu$ such that node $\mu$ is the final node in the order. Node $\mu$ is assumed to be a child of node $\mu$.

| $\eta'$ | $\eta''$ | $\eta' \odot_r \eta''$ |
|---|---|---|
| $(x,0)$ | $(y,z)$ | $(x+y,0)$ |
| $(0,1)$ | $(y,z)$ | $(0,1)$ |

Table 1 The rules to calculate the values of operator $\odot_r$. Variables $x$, $y$, and $z$ take boolean values. Operator $+$ is boolean addition.

Our data structure consists of representing an $n$-node tree $T$ as a collection of disjoint paths. We partition the edges of $T$ into *solid* or *dashed* such that at most one solid edge is incoming into a node. Therefore, every node is in exactly one maximal path of solid edges (of length 0 or more), called a *solid path* of $T$.

The partition of edges into solid and and dashed is obtained by means of the following *size invariant*: let $size(\mu)$ denote the number of nodes in the subtree of $T$ rooted at node $\mu$. An edge $(\mu, \nu)$ of $T$ is solid if $size(\mu) > size(\nu)/2$. If the partition satisfies the size invariant, then every path of $T$ has $O(\log n)$ dashed edges.

Also, in order to achieve the logarithmic time per dynamic operation, we use biased search trees [2] to represent the path-trees. For a node $\nu$ of a solid path $\Pi$ we define the node-attribute $weight(\nu)$ as one plus the sum of the sizes of the subtrees connected to $\mu$ by dashed edges. For each solid path $\Pi$, we also define the corresponding path attribute $weight(\Pi)$ as the sum of the weights of the nodes of $\Pi$. The path-tree $\mathcal{B}_\Pi$ is then implemented as a search tree biased by the node-weights of its leaves.

The size invariant may be temporarily violated by the algorithms that operate on the solid paths of $T$. Hence, we set-up a mechanism to restore it, using a path attribute system on the solid paths.

Consider a node $\mu$ of $T$ on solid path $\Pi$. We keep a globally updatable node attribute $reflected(\mu)$ which indicates the *path reflection status*, which is the left-to-right direction of the children of $\mu$, ignoring any reflection done by ancestors of $tail(\Pi)$. At each path tree node $\eta$, we keep the associated offset value $reflected(\eta)$. Rather than use the *exchange* bits for paths, we use the *reflected* bits which are handled somewhat differently. We also keep a locally updatable node attribute $fixed(\mu)$ which indicates if node $\mu$ is fixed.

Even though the values for *reflected* are boolean, we do not use simple boolean addition as the monoid operator for calculating the actual value of $reflected(\mu)$, for a node $\mu$. At a path tree node $\eta$, we consider $reflected(\eta)$ to be part of a pair of values $(a(\eta), b(\eta))$, where $a(\eta) = reflected(\eta)$ and $b(\eta) = fixed(tail(\eta))$. The monoid operator $\odot_r$ on these pairs is described in table 1. The value $(1,1)$ is not allowed since we do not allow reflection of a fixed node. Note that operator $\odot_r$ does not modify the value of $fixed(tail(\eta))$.

Let $\mu$ be a node of tree $T$. Let $\nu$ be the node closest to the root of $T$ such that $\nu$ is reached by a dashed edge on the path $\Pi$ from $\mu$ to the root of $T$, and $\nu$ is a descendant of the closest ancestor of $\mu$ that is fixed. If $\mu$ has no fixed ancestors, then $\mu$ will be the parent of the last dashed edge of $\Pi$. The *reflection status* of node $\mu$ is then calculated by finding the exclusive-or of the path reflection status of $\mu$ and the nodes reached over dashed edges on the path from $\mu$ to $\nu$.

We use the *reflected* bits much like the *exchanged* bits of section 2.1. Let $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R})$ be the tree attribute system we are maintaining for tree $T$. For node $\mu$, let $\mathcal{R}^*(\mu)$ be the tree

| | *Initial Value* | | *Restructured* | |
|---|---|---|---|---|
| *Rule* | *reversed* | *reflected* | $T_\ell(\mu)$ | $T_r(\mu)$ |
| any | 0 | 0 | $T_\ell(\mu)$ | $T_r(\mu)$ |
| any | 0 | 1 | $T_r^*(\mu)$ | $T_\ell^*(\mu)$ |
| simple | 1 | 0 | $T_\ell(\mu)$ | $T_r(\mu)$ |
| simple | 1 | 1 | $T_r^*(\mu)$ | $T_\ell^*(\mu)$ |
| steady | 1 | 0 | $T_r(\mu)$ | $T_\ell(\mu)$ |
| steady | 1 | 1 | $T_\ell^*(\mu)$ | $T_r^*(\mu)$ |
| mirrored | 1 | 0 | $T_\ell^*(\mu)$ | $T_r^*(\mu)$ |
| mirrored | 1 | 1 | $T_r(\mu)$ | $T_\ell(\mu)$ |

Table 2 Restructuring done by operation $push(\mu)$ for tree node $\mu$ with left-tree $T_\ell(\mu)$ and right-tree $T_r(\mu)$. Eversion rules are given in definition 13. The notation $T_\ell^*(\mu)$ and $T_r^*(\mu)$ indicate the reflections of $T_\ell(\mu)$ and $T_r(\mu)$.

attribute set for the reflection of $\mu$. We have defined the (primary) extended node attribute set $\tilde{\mathcal{N}}(\mu)$ to be the union of $\mathcal{N}(\mu)$ and $\mathcal{R}(\mu')$ for each child $\mu'$ of $\mu$ not on $\Pi$. The secondary extended node attribute set $\tilde{\mathcal{N}}^*(\mu)$ is the union of $\mathcal{N}(\mu)$ and $\mathcal{R}^*(\mu')$ for each child $\mu'$ of $\mu$ not on $\Pi$. Then for each path tree node $\eta$ we keep the four values $\mathcal{P}(\Pi(\eta))$, $\mathcal{P}(\overline{\Pi(\eta)})$, $\mathcal{P}^*(\Pi(\eta))$, and $\mathcal{P}^*(\overline{\Pi(\eta)})$, the meaning of which depend on the values of *reversed*$(\eta)$ and *reflected*$(\eta)$.

We keep a value in $\mathcal{P}(\Pi)$ called *containsfixed*$(\Pi)$ indicating if path $\Pi$ contains a fixed node. If so, then the values of $\mathcal{P}(\overline{\Pi(\eta)})$ and $\mathcal{P}^*(\overline{\Pi(\eta)})$ are undefined.

For node $\mu$ on path $\Pi$ in tree $T$, we extend the restructuring done by operation $push(\mu)$. If the value of *reversed*$(\mu) = 1$ or *reflected*$(\mu) = 1$ prior to the call to $push(\mu)$, then we may swap the left and right-trees of $\mu$, reflect the left and right trees of $\mu$, or both. The restructuring done is based on the eversion rule for $\mu$ as described in definition 13. Table 2 describes this restructuring. For bounded degree tree $T$, we swap the left and right-trees of $\mu$ by reordering the children of $\mu$. We reflect the left (right) tree of $\mu$ by reversing the order of the children of $\mu$ in the left (right) tree and then flipping the *reflected* bit at the roots of the path trees containing the children of $\mu$ in the left (right) tree. Since $T$ has bounded degree, we can perform all this in $O(1)$ time.

Now, we show how to perform operation *Evaluate*$(\mu)$. Let $\Pi$ be the solid path of length $\ell$ containing $\mu$. If $\mu$ is the tail of $\Pi$ then, by the definition of tree attribute systems, we can use path attribute system $(\tilde{\mathcal{N}}, \mathcal{P})$ to find $\mathcal{R}(\mu)$ in $O(1)$ time. Otherwise, we make $\mu$ the tail of a path by calling *split*$(\mu)$. We then restore path $\Pi$ with the *concatenate* operation. Hence, this can all be accomplished in $O(\log \ell)$ time.

Operation *LocalUpdate*$(\lambda, x)$ affects only the values of the nodes in the path from $\lambda$ to the root $\rho$. If such path contains dashed edges, then we temporarily convert it into a solid path by changing dashed edges to solid and solid edges to dashed such that any node continues to have at most one incoming solid edge. This is done with the following operations, derived from dynamic trees [28]:

- *splice*(*path* $\Pi$) — This operation assumes that $\Pi$ is a solid path ending at $\mu \neq \rho$. Convert the dashed edge leaving $\mu$ to solid and convert the solid edge (if it exists) entering the parent $\nu$ of $\mu$ to dashed. Let $\Pi'$ be the solid path containing $\nu$ and $\mu'$ be the sibling of $\mu$ with the solid edge $(\mu', \nu)$. To implement *splice*$(\Pi)$ we need to convert edge $(\mu', \nu)$ from solid to dashed. Let $\Pi''$ be the resulting solid path starting at $\nu$. We obtain $\Pi''$ by splitting $\Pi'$. We complete *splice*$(\Pi)$ by concatenating $\Pi$ and $\Pi''$.

- *expose*$(\mu)$ — Convert to dashed the solid edge entering $\mu$, if such edge exists. Create a solid path from $\mu$ to the root by converting to solid all the dashed edges $(\nu', \nu'')$ of such path, and converting to dashed the edges $(\mathrm{sib}(\nu'), \nu'')$. Operation *expose*$(\mu)$ consists of a sequence of *splice* operations on the solid paths containing the nodes on the path from $\mu$ to $\rho$. This operation is always followed by a *conceal* operation, which undoes its effect.
- *conceal*$(\mu)$ — Restore the original type (solid or dashed) of the edges entering the nodes on the path $\Pi$ from node $\mu$ to the root $\rho$. This operation is the inverse of *expose*, and also consists of a sequence of *splice* operations. The topmost node where to *splice* is given by *light*$(\Pi)$, as described in Example 4. Operation *light*$(\Pi)$ returns the node $\nu$ of $\Pi$ closest to *tail*$(\Pi)$ such that the weight of $\nu$ is larger than the sum of the weights of the nodes preceding $\nu$ in $\Pi$. As shown in Example 4, *light*$(\Pi)$ is found using a path-selection function.

Since path-trees are biased by the weights, the sum of the number of elementary tree operations at each *splice* operation telescopes, so that operations *expose* and *conceal* can be performed with $O(\log n)$ elementary tree operations.

Operation *LocalUpdate*$(\mu, x)$ is realized as follows. First, we issue *expose*$(\mu)$. Next, we change the value of the constant node attribute at $\mu$ to $x$. Finally, we perform *conceal*$(\mu)$ to restore the original solid and dashed edges. Hence, operation *LocalUpdate* is implemented in $O(\log n)$ time.

Operation *GlobalUpdate*$(\mu', \mu'', N, \delta)$ first makes two calls to *expose* to get the path $\Pi$ from $\mu'$ to $\mu''$. Let $\Delta_N$ be the entry for node attribute $N$ in the $\Delta$ vector at the root of the path-tree for $\Pi$. We set $\Delta_N$ to $\Delta_N \odot \delta$. We restore the weight invariant by calling *expose*$(\mu')$ followed by *conceal*$(\mu')$. All this can be done in $O(\log n)$ time.

To implement $LCA(\mu', \mu'')$ we first issue *expose*$(\mu')$ followed by *expose*$(\mu'')$. If $\mu'$ and $\mu''$ are on the same resulting solid path, then $\mu'$ is an ancestor of $\mu''$, so we return $\mu''$. Otherwise, when we *expose*$(\mu'')$, we split the path containing $\mu'$ at the least common ancestor. Therefore, $LCA(\mu', \mu'')$ will be the parent of the tail of the path containing $\mu'$. We restore the size invariant by calling *conceal*$(\mu'')$, *expose*$(\mu')$, then *conceal*$(\mu')$. All this can be performed in $O(\log n)$ time.

For a path $\Pi$, we use the eversion rules described in table 2 to calculate the value of $P(\bar{\Pi})$ and $P^{(}\bar{\Pi})$. We perform operation *Evert*$(\mu)$ by first issuing *expose*$(\mu)$ to get the path $\Pi$ from $\mu$ to the root $\rho$, flipping the reverse bit of $\Pi$, and finally calling operation *conceal*$(\rho)$ to restore the size invariant. Therefore, operation *Evert* is performed in $O(\log n)$ time.

Operation *Reflect*$(\mu)$ is performed as follows. First, we call *expose*$(\mu)$. The left edge-path of $\mu$ then contains nodes for all the children of $\mu$. We then reverse the order of the children of $\mu$ by flipping the *reflected* and *reversed* bits at the root of the path-tree representing $\Pi_\ell(\mu)$. Finally, we perform *conceal*$(\mu)$ to restore the original solid and dashed edges. Hence, operation *Reflect* is implemented in $O(\log n)$ time.

Operation $Cycle(\nu, \mu)$ is performed by first calling $expose(\nu)$. We then reorder the children of $\nu$. Suppose $T_r(\nu)$ is the right-tree of $\nu$ with respect to path $\Pi$ before the *Cycle* operation is performed. We flip the *Reflect* bit at the roots of the path trees containing the children of $\nu$ in $T_r(\nu)$. Finally, we perform $conceal(\nu)$ to restore the original solid and dashed edges. All this can be done in $O(\log n)$ time.

Operation $PathFind(\mu', \mu'', S, q)$ makes two calls to *expose* to get the path $\Pi$ from $\mu'$ to $\mu''$, then calls $find(\Pi, S, q)$. We then restore the weight invariant by calling $expose(\nu')$ followed by $conceal(\nu')$.

Consider tree selection function $S$. We implement operation $TreeFind(\nu, S, q)$ as follows. If $\nu$ is not the root of its tree, we first call *expose* at $Parent(\nu)$. Node $\nu$ then becomes the tail of its solid path $\Pi$. Let $\mu$ be the node we are searching for. We repeat the following until node $\mu$ is found. Let node $\mu' = find(\Pi, S_p, q)$. Create path $\Pi'$ with head $\mu'$ and tail $\nu$. This is done with single *split* and *concatenate* operations.

Suppose $\mu_1, \ldots, \mu_d$ are the children of $\mu'$ not on $\Pi$ and $\Pi_1, \ldots, \Pi_d$ are the solid paths containing each $\mu_j$. Repeat the following until we find a child $\mu_i$ of $\mu'$ such that node $\mu$ is a descendant of $\mu_i$. Consider child node $\mu_j$. Let path $\Pi'$ be the result of joining paths $\Pi_j$ and $\Pi'$. By the definition of tree and path selection functions, we can determine in $O(1)$ time if node $\mu$ is in the subtree rooted at $\mu_j$.

If no child of $\mu'$ has $\mu$ as a descendant, then $\mu' = \mu$. Otherwise, we reset path $\Pi$ to be $\Pi'$ and continue. After $\mu$ is found, we undo the restructuring to restore the path trees.

At each iteration, the time to find $\mu'$ and create $\Pi'$ is $O(d_{\mu'})$, where $d_{\mu'}$ is the depth of $\mu'$ in $\mathcal{B}_\Pi$, the path tree for path $\Pi$. Operation *join* increases the depth of a path tree node by 1. Therefore, $d_{\mu'}$ is at most one more than the depth of $\mu'$ in its original path-tree. Therefore, the time to find $\mu$ is $O(\log n)$ plus the time to perform $expose(\mu)$. The time to restore the path trees is equivalent. Therefore operation *TreeFind* is implemented in $O(\log n)$ time.

Operations *MakeTree* and *DeleteTree* can be trivially implemented in $O(1)$ time. Operation $Parent(\mu)$ is performed in $O(\log n)$ time by calling $expose(\mu)$, returning $after(\mu)$ on the solid path containing $\mu$, and calling $conceal(\mu)$.

Operation $Sibling(\mu, k, X)$ is realized as follows. If node $\mu$ is the root of $T$, then return *nil*. Otherwise, we call $expose(\mu)$. Let node $\nu$ be the parent of $\mu$. Let $\zeta$ be the root of the resulting path tree. We call operation *push* on the path from $\zeta$ to $\nu$. Node $\nu$ then stores the actual value of $reflected(\nu)$. We then return the (at most) $k$ siblings of $\mu$ in the $X$ direction. Hence, operation *Sibling* is performed in $O(\log n + k)$ time.

Operations *Link* and *Cut* are implemented in $O(\log n)$ time using variations of the same named dynamic tree operations. We implement operation $Link(\rho, \mu, \mu', \mu'')$ by first calling $expose(\mu)$. We then add $\rho$ as a child of $\mu$ between $\mu'$ and $\mu''$. Finally, we call $conceal(\mu)$ to maintain the size invariant.

Operation $Cut(\mu)$ is the inverse of *Link*. We first call $expose(Parent(\mu))$. We then remove node $\mu$ as a child of its parent by removing $\mu$ from the order of children stored at $\mu$ and destroying the dashed edge. We conclude by calling $conceal(\mu)$ to maintain the size invariant.

**Theorem 2** *Let $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ be a tree attribute system, and $\mathcal{S}$ be a collection of path-selection and tree-selection functions. There is a fully dynamic data structure for maintaining $\mathcal{Z}$ and $\mathcal{S}$ over a set of trees with the following performance:*

1. *a tree of size $n$ uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Evaluate, LocalUpdate, GlobalUpdate, Link, Cut, Evert, Reflect, Cycle, *LCA*, PathFind, TreeFind, Sibling, *and* Parent *take each $O(\log n)$ time, where $n$ is the total size of the trees involved in the operation.*

**Example 5** *A* concatenable heap *consists of a collection of* heaps *which are sets of weighted nodes that support the following operations:*

- FindMin(*heap $H$*) — *returns the cost of a minimum cost node in heap $H$.*
- InsertHeap(*node $\mu$*; *heap $H$*) — *insert node $\mu$ in heap $H$.*
- DeleteHeap(*node $\mu$*; *heap $H$*) — *delete node $\mu$ from heap $H$.*
- ConcatenateHeap(*heap $H_1, H_2$*) — *combine heaps $H_1$ and $H_2$ into a single heap.*

*We can implement a concatenable heap as either a path or a tree. If we store each heap as a path, then operation* FindMin($H$) *is implemented as operation* FindMin *in example 2. Operations* InsertHeap($\mu, H$), DeleteHeap($\mu, H$), *and* ConcatenateHeap($H_1, H_2$) *are directly implemented using operations* concatenate *and* split.

*The implementation of a concatenable heap as a tree is more complicated. We present this technique since it is used in developing more generalized heaps that can not be implemented as paths.*

*The nodes of a heap are stored as leaves of an arbitrary tree. A heap is identified as its representative tree. We maintain the following tree attribute for the nodes of tree $T$.*

- mincost($\mu$) — *the minimum cost of a node in the subtree rooted at $\mu$.*

*We also maintain the following path attribute for the paths of tree $T$.*

- minpathcost($\Pi$) — *the minimum cost of a leaf that is a descendant of* tail($\Pi$), *but is not* head($\Pi$) *or a descendant of* head($\Pi$).

*It is easy to see that these values can be maintained as part of a tree attribute system. We implement operation* FindMin *using the following tree selection function which takes a cost $q$ as a query argument:*

**Selection Function $\mathbf{S_4}$** Suppose node $\eta$ the root of the path tree with children $\eta'$ and $\eta''$. If *minpathcost*(*tail*($\eta$)) = *minpathcost*(*tail*($\eta'$)) then return $\eta'$, else return $\eta''$.

*For tree $T$ associated with heap $H$, let $c$ be the value of* mincost *at the root of $T$. Operation* FindMin($H$) *is implemented as* TreeFind($\rho, S_1, c$). *Operations* InsertHeap($\mu, H$), DeleteHeap($\mu, H$), *and* ConcatenateHeap($H_1, H_2$) *are directly implemented using operations* Link, Cut, *and* MakeTree.

*Therefore, both implementations of concatenable heaps are implemented with $O(\log n)$ time per operation, where $n$ is the total size of the heaps involved.*

### 2.2.2 Unbounded Degree Trees

The algorithm presented in the previous section for bounded degree trees takes advantage of the fact that reordering and reflecting the children of a node can be accomplished in constant time. In the case of unbounded degree trees, we need to develop additional techniques to accomplish the needed restructuring.

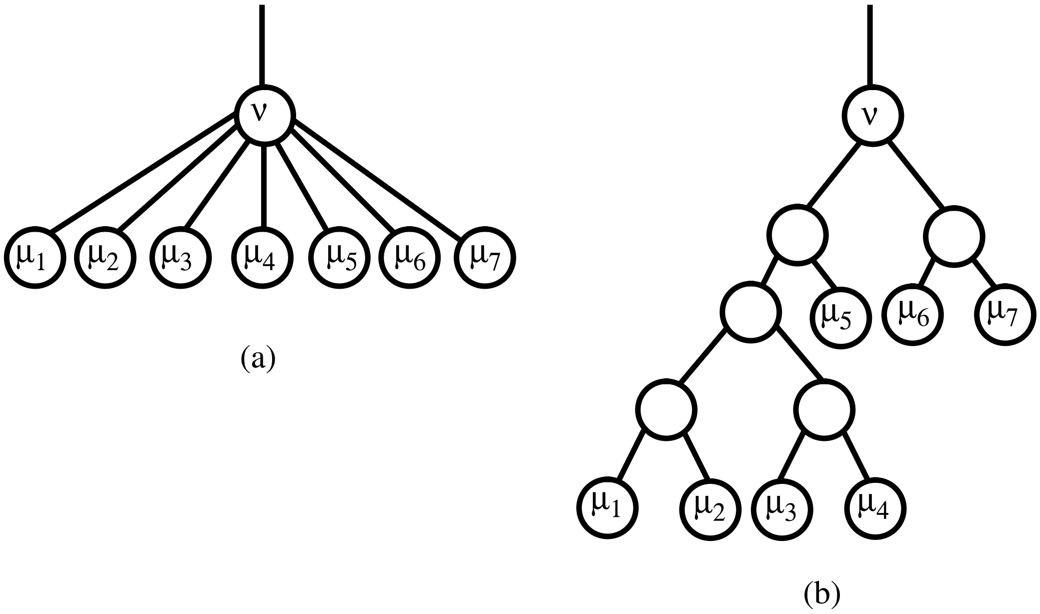


**Figure 3:** Expanding a node of unbounded degree. (a) The original node. (b) The expansion of the node.

**Definition 15** *Suppose $T$ is a (unbounded-degree) tree containing node $\nu$ with children $\mu_1, \ldots, \mu_d$.* Expanding *$\nu$ between $\mu_i$ and $\mu_j$ ($1 \leq i \leq j \leq d$) consists of restructuring $T$ such that we replace nodes $\mu_i, \ldots \mu_j$ in the ordering of the children of $\nu$ with a new node $\mu$. Node $\mu$ has children $\mu_i, \ldots \mu_j$. We set $\mathcal{N}_L(\mu) = \mathcal{N}_L(\nu)$ and $\mathcal{N}_G(\mu) = 0$. The inverse operation is called* contracting *child $\mu$ of $\nu$. In order to contract $\mu$, $\mathcal{N}_L(\mu)$ must equal $\mathcal{N}_L(\nu)$ and $\mathcal{N}_G(\mu)$ must equal* $0$*.*

For unbounded degree trees, we require that tree attribute set $\mathcal{R}$ meet the following *expansion invariant*: For any node $\nu$ of tree $T$, the value of $\mathcal{R}(\nu)$ is unchanged as a result of expanding $\nu$ or contracting a child of $\nu$. Therefore, given an unbounded degree tree $T$ we can create an equivalent tree $T'$ of bounded degree 2, called a *binary expansion* of $T$, by expanding nodes with more than two children (see Fig. 3). Note that if $T$ has $n$ nodes, then $T'$ will have $O(n)$ nodes.

**Definition 16** *Suppose $T$ is an unbounded degree tree. Then $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ is a tree attribute system on $T$ if and only if node attribute set $\mathcal{R}$ keeps the expansion invariant and $\mathcal{Z}$ is a tree attribute system on any binary expansion $T'$ of $T$.*

For each node $\nu$ in an unbounded degree tree $T$, we keep two paths $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$ associated with the children to the left and right of a specified child $\mu_i$ of $\nu$ (see Fig. 4). We write $\nu(\mu_j)$ to represent the node in either $\Pi_\ell(\nu)$ or $\Pi_r(\nu)$ associated with $\mu_j$. Formally,

**Definition 17** *Consider an unbounded degree tree $T$ containing a node $\nu$ with children $\mu_1, \ldots, \mu_d$ in left-to-right order. Suppose $\Pi$ is a path of $T$ containing $\nu$. The* left edge-path *of $\nu$ with respect to $\Pi$, $\Pi_\ell(\nu)$, consists of a node $\nu(\mu_j)$ for each child $\mu_j$ of $\nu$ in the left-tree of $\nu$ with respect to $\Pi$. Path $\Pi_\ell(\nu)$ is directed in the left-to-right order of the chil-*

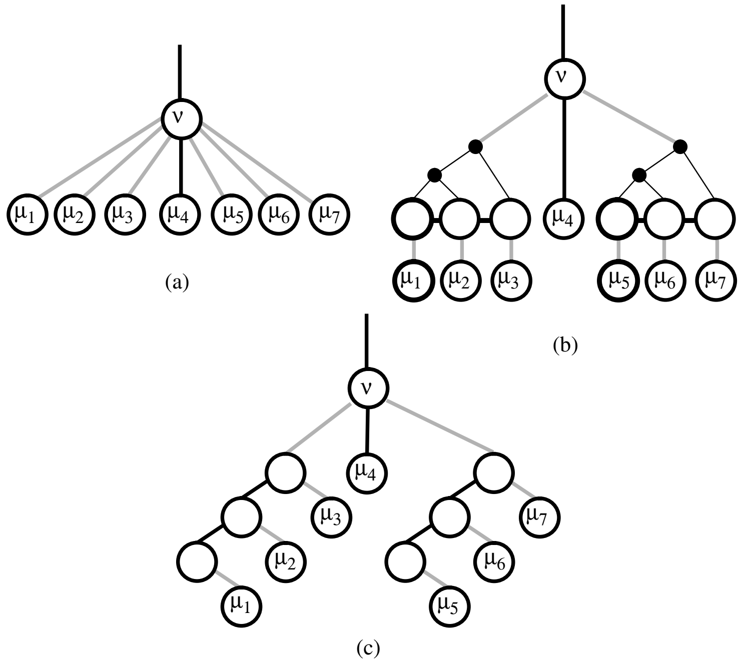


**Figure 4:** A node $\nu$ in an unbounded degree tree. (a) The node its solid path, and its children. (b) The left and right edge-paths. (c) An equivalent bounded degree representation.

*dren. The value of* $\mathcal{N}_L(\nu(\mu_j))$ *is defined to be* $\mathcal{N}_L(\nu)$. *The* right edge-path $\Pi_r(\nu)$ *is similarly defined for the children of* $\nu$ *in the right-tree of* $\nu$.

Consider an unbounded degree tree $T$ and tree attribute system $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ on $T$. Let $\Pi$ be a solid path in $T$ and $\Pi_X(\nu)$ be an edge path of some node $\nu$. We define the extended node attribute set $\hat{\mathcal{N}}$ that, for a node $\mu$ of $\Pi$, $\hat{\mathcal{N}}(\mu)$ consists of $\mathcal{N}(\mu)$, $\mathcal{R}(\mu_\ell)$, and $\mathcal{R}(\mu_r)$ for the roots $\mu_\ell$ and $\mu_r$ of the left and right-trees of $\mu$. For a node $\nu(\mu')$ of $\Pi_X(\nu)$, $x = \ell$ or $r$, extended node attribute set $\hat{\mathcal{N}}(\nu(\mu'))$ consists of $\mathcal{N}(\nu)$ and $\mathcal{R}(\mu')$.

**Lemma 2** *Suppose* $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ *is a tree attribute system on unbounded degree tree* $T$. *Then,*

- *The triplet* $(\hat{\mathcal{N}}, \mathcal{P}, F)$ *is a path attribute system for paths in* $T$ *where concatenations are restricted to adjacent subpaths.*
- *The triplet* $(\hat{\mathcal{N}}, \mathcal{P}, F)$ *is a path attribute system for edge-paths where concatenations are restricted to adjacent subpaths of the edge-paths.*

**Proof:** Suppose $\Pi$ is a solid path from node $\sigma$ to node $\tau$. Consider the path $\hat{\Pi}$ formed by expanding every node $\mu$ on $\Pi$ into two nodes $\mu'$ and $\mu''$ on $\hat{\Pi}$ with $\mu'$ the parent of $\mu''$. Recall that nodes $\mu_\ell$ and $\mu_r$ are the roots of the left and right-trees of node $\mu$ with respect to $\Pi$. Suppose we know the values $\mathcal{R}(\mu_\ell)$ and $\mathcal{R}(\mu_r)$. We make $\mu_\ell$ the left child of $\mu'$ and $\mu_r$ the right child of $\mu''$. Path $\hat{(\Pi)}$ is the path from $\sigma''$ to $\tau'$ in a binary expansion of $T$. Therefore, the first part of the lemma is true if we define $\mathcal{P}(\Pi)$ to be $\mathcal{P}(\hat{\Pi})$.

The argument for edge-paths is even more direct. Suppose $\nu$ is a node and $X$ is either $\ell$ or $r$. Each node of edge-path $\Pi_X(\nu)$ has degree bounded by 2. Therefore each edge path is a path in a binary expansion of $T$, so the lemma holds by definition 16.

$\square$

In addition to the operations previously defined for bounded degree trees, we support the following operations on unbounded degree trees:

- *Expand*(*node* $\mu, \mu', \mu''$) — Expand node $\mu$ between children $\mu'$ and $\mu''$. Node $\mu'$ is assumed to precede $\mu''$ in the left-to-right order of the children of $\mu$.
- *Contract*(*node* $\mu$) — Contract $\mu$ into its parent node $\nu$. The values $\mathcal{N}(\mu)$ and $\mathcal{N}(\nu)$ are assumed to be equal.

Edge-paths are stored in the same structure as solid paths. If $\Pi_X(\nu)$ is the left or right edge-path of node $\nu$, then reversing $\Pi_X(\nu)$ is equivalent to reflecting the left or right-tree of $\nu$. Therefore, path tree nodes use eversion rule 2 (see definition 13).

Recall that operation *splice* converts a dashed edge to solid and possibly a solid edge to dashed. Suppose $\mu$ is a node connected to its parent $\nu$ by a solid edge. Let $\Pi$ be the solid path containing nodes $\mu$ and $\nu$. Let $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$ be the left and right edge-paths of $\nu$. We convert solid edge $(\mu, \nu)$ to dashed as follows. We begin by splitting path $\Pi$ at $\nu$. As described in section 2.2.1, operation *split* calls operation push along the path from the root of the path tree for $\Pi$ to node $\nu$. Operation *push*($\nu$) may require the left and right-trees of $\nu$ to be swapped or exchanged (see table 2). This is performed by either exchanging the meaning of $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$, or by flipping the *reversed* and *reflected* bits at the root of the path tree representing $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$. We conclude by creating a path tree node $\nu(\mu)$ for the new dashed edge, and setting the left edge-path of $\nu$ to be the concatenation $\Pi_\ell(\nu)$, $\nu(\mu)$, and $\Pi_r(\nu)$ and the right edge-path to be the empty path.

Converting a dashed edge to solid is performed similarly, reversing the roles of the solid and edge-paths. Suppose the edge between node $\mu$ and its parent $\nu$ is now dashed, and there is no solid edge entering $\nu$. Let $\Pi'$ and $\Pi''$ be the paths with $\mu = tail(\Pi')$ and $\nu = head(\Pi'')$. Let $\Pi_\ell(\nu)$ be the left edge-path of $\nu$. Split $\Pi_\ell(\nu)$ at $\nu(\mu)$. The resulting paths to the left and right of $\nu(\mu)$ become the left and right edge-paths of $\nu$. We then concatenate $\Pi'$ and $\Pi''$ to create the solid edge.

For a node $\mu$ of a solid path $\Pi$ we now define the node-attribute $weight(\mu)$ as one plus the sum of the sizes of the left and right edge-paths of $\mu$. For a node $\nu(\mu')$ of an edge-path $\Pi_X(\nu)$ we define the node-attribute $weight(\nu(\mu'))$ as one plus the sum of the sizes of solid path containing $\mu'$. We bias by weight the path trees for both the edge-paths and solid paths. Therefore, the sum of the number of elementary tree operations at each *splice* operation telescopes, so that operations *expose* and *conceal* can be performed with $O(\log n)$ elementary tree operations.

The techniques to perform operations *TreeFind*, *Sibling*, *Reflect* and *Cycle* are slightly modified for unbounded degree trees. All other operations remain unchanged.

We implement operation *TreeFind*($\nu, S, q$) as in the bounded-degree tree case, with following difference. Let $\mu$ be the node we a searching for. Suppose at an intermediate step node $\mu'$ is the deepest node on path $\Pi$ such that the node $\mu$ is a descendant of $\mu'$. We proceed as in the bounded case, except we next consider the edge-paths connected to $\mu'$, rather than the solid paths containing the children of $\mu'$. Hence, we join an edge-path to $\Pi$. This is well defined by the expansion invariant. We perform a similar operation when $\mu'$ is an edge-path node.

Operation *Sibling*$(\mu, k, X)$ is realized by first calling *expose*$(\mu)$. We then use operation *push* to get the value of *reverse* and *Reflect* at *Parent*$(\mu)$. We find the siblings by either calling *before* or *after* on the left or right path trees of $\nu$, depending on direction $X$. We conclude by restoring the size invariant by calling *conceal*$(\mu)$.

Operation *Reflect*$(\mu)$ is performed as follows. First, we call *expose*$(\mu)$. We then flip the *reverse* bit for the left edge-path of $\mu$. Finally, we perform *conceal*$(\mu)$ to restore the original solid and dashed edges.

Operation *Cycle*$(\nu, \mu)$ is performed by first calling *expose*$(\nu)$. We then split the left path tree of $\nu$ at path tree node $\nu(\mu)$. Let $\Pi'$ and $\Pi''$ be the resulting subpaths representing the children of $\nu$ before and after $\mu$. We reorder the children of $\nu$ by flipping the *reflected* and *reversed* bits at the root of the path tree representing $\Pi''$, concatenating $\Pi''$, $\Pi'$ and $\nu(\mu)$, and making this new edge-path the left edge-path of $\nu$. Finally, we perform *conceal*$(\nu)$ to restore the original solid and dashed edges.

Operation *Expand*$(\mu, \mu', \mu'')$ begins by calling *expose*$(\mu)$ which results in the left edge-path of $\mu$, $\Pi_\ell(\mu)$ containing all the edges from children to $\mu$. Then we perform a constant number of split and join operations to form three subpaths $\Pi'$, $\Pi''$, and $\Pi'''$ of $\Pi_\ell(\mu)$, with $\Pi'$ containing the edge-path nodes associated with the children of $\mu$ preceding $\mu$, $\Pi''$ containing the edge-path nodes associated with the children of $\mu$ between $\mu'$ and $\mu''$, and $\Pi'''$ containing the edge-path nodes associated with the children of $\mu$ following $\mu''$. We extend the solid path containing $\mu$ by concatenating a new node $\nu$ before node $\mu$. We make $\Pi''$ the left edge-path of $\nu$, and paths $\Pi'$ and $\Pi''$ the left and right edge-paths of node $\mu$. Finally, we call *conceal*$(\nu)$ to restore the size invariant.

Operation *Contract*$(\mu)$ is performed similarly. We begin by calling *expose*$(\mu)$ which results in the left edge-path of $\mu$, $\Pi_\ell(\mu)$ containing all the edges from children to $\mu$. Let node $\nu$ be the parent of $\mu$. We remove node $\mu$. We set $\Pi_\ell(\nu)$ to the concatenation of $\Pi_\ell(\nu)$, $\Pi_\ell(\mu)$, and $\Pi_r(\nu)$. We set $\Pi_r(\nu)$ to the empty path. Finally, we call *conceal*$(\nu)$ to restore the size invariant.

**Theorem 3** *Let $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ be a tree attribute system, and $\mathcal{S}$ be a collection of path-selection and tree-selection functions. There is a fully dynamic data structure for maintaining $\mathcal{Z}$ and $\mathcal{S}$ over a set of unbounded degree trees with the following performance:*

1. *a tree of size $n$ uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Sibling *takes $O(\log n + k)$ time to return $k$ nodes.*
4. *operations* Evaluate, LocalUpdate, GlobalUpdate, Link, Cut, Evert, Reflect, Cycle, *LCA*, PathFind, TreeFind, Parent, Expand, *and* Contract *take each $O(\log n)$ time, where $n$ is the total size of the trees involved in the operation.*

# 3 Expression Trees

## 3.1 Linear Expression Trees

In this section, we consider the dynamic evaluation of linear expressions, which extends and generalizes the dynamic expression trees of [9]. We show how to solve this problem using a

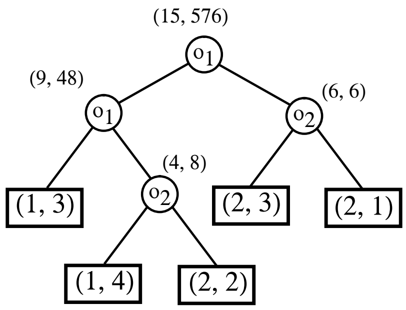


**Figure 5:** A linear expression tree with two binary linear operators. For $x = (x_1, x_2)$ and $y = (y_1, y_2)$ operators are defined to be: $o_1(x, y) = (x_1 + y_2, 2y_1y_2)$ and $o_2(x, y) = (2x_1 + x_1y_2, x_2y_1)$. The unary operators associated with the edges are the identity operators.

tree attribute system. Let $\mathcal{S}$ be a semiring with binary operators $\oplus$ and $\otimes$, such that $\otimes$ is distributive with respect to $\oplus$. We assume that elements of $\mathcal{S}$ can be stored in $O(1)$ space and binary operations can be performed in $O(1)$ time. The extension to the general case is straightforward. We consider operations in $\mathcal{S}^r$, the space of $r$-tuples of $\mathcal{S}$ for some fixed $r$. Let $x = (x_1, \ldots, x_r)$ and $y = (y_1, \ldots, y_r) \in \mathcal{S}^r$. A *binary linear operator* $\odot : \mathcal{S}^r \times \mathcal{S}^r \to \mathcal{S}^r$ is defined by

$$(x \odot y)_i = x^T A_i y \oplus b_i^T x \oplus c_i^T y \oplus d_i$$

for $i = 1, \cdots, r$, where $A_i \in \mathcal{S}^r \times \mathcal{S}^r$, $b_i, c_i \in \mathcal{S}^r$, and $d_i \in \mathcal{S}$. A *unary linear operator* $\nabla : \mathcal{S}^r \to \mathcal{S}^r$ is defined by

$$\nabla x = Ax \oplus b$$

where $A \in \mathcal{S}^r \times \mathcal{S}^r$, and $b \in \mathcal{S}^r$. (In the above definitions we have used standard linear algebra notation, and convention that $xy$ denotes $x \otimes y$.) Note that, given a binary linear operator $\odot$, for any fixed $\bar{y}$, $x \odot \bar{y}$ is a unary linear operator. Also, given a unary linear operator $\nabla$, there exists a binary linear operator $\odot$ such that $\nabla x = x \odot 0$.

A *linear expression* $\mathcal{E}$ over $\mathcal{S}^r$ is an expression with variables in $\mathcal{S}^r$ involving binary and unary linear operators. A linear expression is represented by a tree $T$ called a linear expression tree, such that the internal nodes are the operators and the leaves are the variables of $\mathcal{E}$ (see an example in Fig. 5). We associate a binary linear operator with each internal node of $T$. If $\mu$ is an internal node associated with a binary linear operator, but with only one child, then $\mu$ takes the value of its child.

**Example 6** Fig. 5 shows an example of linear expression and its associated binary tree.

We allow unbounded degree linear expression trees. Suppose $\odot$ is a binary linear operator. The unbounded version of $\odot$ takes a variable and unbounded number of operands. We write $\odot(x_1, \ldots, x_d)$ to indicate applying $\odot$ to values $x_1, \ldots, x_d$. If $d = 1$ then the value of $\odot(x_1)$ is $x_1 \odot 0$. Otherwise, the value of $\odot(x_1, \ldots, x_d)$ is $((x_1 \odot x_2) \ldots) \odot x_d$. An *unbounded degree linear expression* is an expression containing unbounded versions of binary linear operators.

An unbounded linear expression is represented by an unbounded linear expression tree. In order to keep the value invariant each binary linear operator must be associative.

We consider a dynamic environment where linear expressions are manipulated by changing the values of the variables and by linking and cutting the corresponding trees. We also allow a somehow unorthodox modification of an expression $\mathcal{E}$ by everting its tree $T$.

In order to give a meaning to the eversion of $T$, we require the following. Suppose $\mu$ is a node that is not the root of $T$. Everting at node $\mu$ results in $\mu$ having an additional child. We do not allow eversion at a leaf of a linear expression tree. In order to give meaning to eversion at a node with 1 child, each internal node calculates a binary linear operator. A unary linear operator is simulated by a binary linear operator with only one child. Finally, we allow the root $\rho$ of $T$ to have three children. If $\odot$ is the binary linear operator at $\rho$, then the value of $\rho$ is calculated as in the unbounded version of $\odot$. As with general unbounded linear operators, we require that if $\mu$ is a node of $T$ with 2 children, then eversion is allowed at $\mu$ only if the operator $\odot$ at $\mu$ is associative.

Given a linear expression tree $T$ we will show that we can maintain a tree attribute system to find the values of subexpressions of $T$.

**Lemma 3** *Suppose $\mathcal{E}$ is a linear expression. The dependence of the value $y$ of $\mathcal{E}$ on a single variable $x$ can be expressed by a unary linear operator $y = Ax \oplus b$.*

**Proof:** The proof is by induction on the level of the subexpression containing $x$. If $y$ is $x$, then the unary linear operator is $y = x$. Now suppose $y = y' \odot y''$ and $x$ is used in the calculation of $y'$. By the inductive hypothesis, we have $y' = A'x \oplus b'$ for some matrix $A'$ and vector $b'$. When we fix the value of $y''$, we get by the earlier observation matrix $A''$ and vector $b''$ such that $y = A''y' \oplus b''$. By composition, we have $y = A''A'x \oplus (A''b' \oplus b'')$ □

We call *transfer pair* a (matrix, vector) pair $(A, b)$ that by Lemma 3 characterizes the dependency of a linear expression on a variable. In particular, given a (solid or edge) path $\Pi$ of $T$, we denote with $P(\Pi)$ the transfer pair associated with the dependency of the value of $tail(\Pi)$ from the value of $head(\Pi)$.

**Lemma 4** *Let $\mathcal{E}$ be a linear expression represented by a tree $T$, and let $x(\mu)$ be the value of the subexpression given by the subtree of $T$ rooted at $\mu$. Then the transfer pairs $P(\Pi)$ of the solid or edge paths $\Pi$ of $T$ form a path attribute system, and there exists a tree attribute system that supports tree attribute $x(\mu)$ by means of the path attribute $P(\Pi)$.*

**Proof:** Suppose path $\Pi$ of $T$ is the concatenation of paths $\Pi'$ and $\Pi''$ with transfer pairs $(A', b')$ and $(A'', b'')$, respectively. The value of $head(\Pi'')$ is obtained from the values of the children of $head(\Pi'')$ by means of a linear operator; hence $x(head(\Pi'')) = Cx(tail(\Pi')) \oplus d$. By Lemma 3 we have $x(head(\Pi)) = A''(C(A'x(tail(\Pi)) \oplus b') \oplus d) \oplus b''$. Thus, $x(head(\Pi)) = A''CA'x(tail(\Pi)) \oplus A''(Cb' \oplus d) \oplus b''$, so that the transfer pair $P(\Pi) = (A, b)$ is given by $A = A''CA'$ and $b = A''(Cb' \oplus d) \oplus b''$. This shows that the transfer pairs form a path attribute system. Clearly, transfer pair $P(\Pi)$ allows to compute in $O(1)$ time $x(tail(\Pi))$ from $x(head(\Pi))$. Also, the transfer pairs depend on an extended node attribute set that for a node $\mu$ of a path $\Pi$ consists of the operator of $\mu$ and the value $x(\mu')$ of the child $\mu'$ of $\mu$ not in $\Pi$. Hence, the triplet $(O, P, x)$, where $O(\mu)$ denotes the operator of node $\mu$, is a tree attribute system, and the lemma follows from Theorem 2. □

**Theorem 4** *Let $\mathcal{S}$ be a semiring and $r$ an integer. There is a fully dynamic data structure for maintaining a set of expressions with (unary and binary) linear operators over $\mathcal{S}^r$ with the following performance:*

1. *an expression of size $n$ uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Evaluate, LocalUpdate, Link, Cut, Evert, Reflect, Cycle, PathFind, TreeFind, Parent, *and* Sibling, *take each $O(\log n)$ time, where $n$ is the total size of the trees involved in the operation.*

If we restrict ourselves to associative binary linear operators, then by lemma 2, lemma 4, and Theorem 2, we get:

**Theorem 5** *Let $\mathcal{S}$ be a semiring and $r$ an integer. There is a fully dynamic data structure for maintaining a set of unbounded linear expressions with associative binary linear operators over $\mathcal{S}^r$ with the following performance:*

1. *an expression of size $n$ uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Sibling *takes $O(\log n + k)$ time to return $k$ nodes.*
4. *operations* Evaluate, LocalUpdate, Link, Cut, Evert, Reflect, Cycle, *LCA,* PathFind, TreeFind, Parent, Expand, *and* Contract *take each $O(\log n)$ time, where $n$ is the total size of the trees involved in the operation.*

## 3.2 Linear Attribute Grammars

In this section, we consider the dynamic evaluation of linear attribute grammars and show that this problem can also be solved using a tree attribute system.

An *attribute grammar* $T$ is a rooted, ordered tree such that each node $\mu$ of $T$ stores an r-tuple of attributes $x(\mu) = (x_1(\mu), \ldots, x_r(\mu))$. *Synthesized* attributes are calculated from the children of $\mu$. *Inherited* attributes are calculated from the attributes of the parent of $\mu$, and the synthesized attributes of $\mu$ and its siblings. We assume, without loss of generality, that the synthesized attributes of $\mu$ precede the inherited attributes of $\mu$. A *binary attribute grammar* is an attribute grammar on a binary tree. We consider attribute grammars where the values of each attribute are taken from a semiring $\mathcal{S}$.

Attribute grammars [20] are much studied and have applications in areas such as language based editors [3,26], and VLSI design [18]. Incremental algorithms for attribute grammars have been presented [17,25] which give a $O(a \cdot t)$ running time per operation where $a$ is the number of attributes affected by a change and $t$ is the time to make a single change. In general, $a = O(n)$ for an attribute grammar of size $n$. Incremental evaluation of general circuits is studied in [1].

The precedence graph for an attribute grammar $T$ is a digraph consisting of a vertex for each attribute $x_i(\mu)$ and a directed edge from $x_i(\mu)$ to $x_j(\nu)$ if the value $x_i(\mu)$ is used in the calculation of $x_j(\nu)$.

For a node $\mu$ of attribute grammar $T$, let $s(\mu) = (s_1(\mu), \ldots s_r(\mu))$ and $h(\mu) = (h_1(\mu), \ldots h_r(\mu))$ be the r-tuples defined by:

$$s(\mu) = \begin{cases} x_j(\mu) & \text{if } x_j(\mu) \text{ is synthesized} \\ 0 & \text{if } x_j(\mu) \text{ is inherited} \end{cases}$$

$$h(\mu) = \begin{cases} x_j(\mu) & \text{if } x_j(\mu) \text{ is inherited} \\ 0 & \text{if } x_j(\mu) \text{ is synthesized} \end{cases}$$

where $1 \leq j \leq r$.

Suppose node $\mu$ has $p < r$ synthesized attributes. Then $s^R(\mu) = (x_1(\mu), \ldots, x_p(\mu))$, and $h^R(\mu) = (x_{p+1}(\mu), \ldots, x_r(\mu))$.

**Definition 18** *Consider a node $\mu$ of a binary attribute grammar $T$. Let node $\nu$ be the parent of $\mu$, node $\mu'$ be the sibling of $\mu$, and nodes $\mu_1$ and $\mu_2$ be the children of $\mu$. A* bidirectional binary operator $\odot = (\odot_S, \odot_I, \odot_\ell, \odot_r)$ *is a 4-tuple of binary linear operators such that:*

$$s(\mu) = x(\mu_1) \odot_S x(\mu_2)$$

*and, if $\mu$ is the left child of $\nu$:*

$$h(\mu) = s(\mu) \odot_I (x(\nu) \odot_r s(\mu'))$$

*otherwise, if $\mu$ is the right child of $\nu$:*

$$h(\mu) = s(\mu) \odot_I (x(\nu) \odot_\ell s(\mu'))$$

**Definition 19** *A* binary linear attribute grammar *$T$ is a binary attribute grammar such that:*

- *The value of all attributes come from a semiring $\mathcal{S}$.*
- *Each node $\mu$ of $T$ has an associated bidirectional operator to calculate the value $x(\mu)$.*
- *The precedence graph of $G$ is acyclic.*
- *The dependence of the value an attribute $x_i(\mu)$ on the value of any other attribute $x_j(\nu)$ is linear. That is, we can express the dependence in the form $x_i(\mu) = ax_j(\nu) + b$ where $a$ and $b$ are members of $\mathcal{S}$.*

A (binary) linear expression tree is a (binary) linear attribute grammar without inherited attributes.

Suppose $G$ is the precedent graph of a linear attribute grammar $T$ and $x_i(\nu_1)$ and $x_j(\nu_2)$ are two nodes of $G$. Suppose $\pi'$ and $\pi''$ are two paths in $G$ from $x_i(\nu_1)$ to $x_j(\nu_2)$ that only meet at their endpoints. Let $x_h(\mu')$ and $x_k(\mu'')$ be the predecessors of $x_j(\nu_2)$ on $\pi'$ and $\pi''$. Then the attributes $x_h(\mu')$ and $x_k(\mu'')$ are not multiplied by the operation performed at $\nu_2$.

Suppose $x = (x_1, \ldots, x_h)$ is a vector. The power vector *power*$(x)$ is a vector with $2^h$ entries $(1, x_1, \ldots, x_h, x_1x_2, \ldots, x_{h-1}x_h, \ldots, x_1 \cdots x_h)$.

**Lemma 5** *Suppose $G$ is a precedence graph such that all dependencies are linear. Consider attributes $y_0, \ldots, y_f$ of $G$. The dependence of the value of $y_0$ on the values of $y_1, \ldots, y_f$ can be expressed as $y_0 = a \cdot \mathrm{power}((y_1, \ldots, y_f))$ for some vector $a$ with $2^f$ components.*

**Proof:** Let $G$ be the precedence graph of $T$. Our proof is by induction on the length of a longest directed path from some $y_i$ $(1 \leq i \leq f)$ to $y_0$ in $G$. If the length is 0, then all $y_i$ $(0 \leq i \leq f)$ are identical, and the lemma follows with $a = (0, 1, 0, \ldots)$.

Otherwise, let $(\nu_1, \ldots, \nu_k)$ be the attributes of $T$ such that for each $\nu_j$ $(1 \leq j \leq k)$ there is an edge from $\nu_j$ to $y_0$ and a path of length greater than or equal to 0 from some $y_i$ to $\nu_j$. Then for each $\nu_j$ there is a vector $a_j$ such that $\nu_j = a_j \cdot power((y_1, \ldots, y_j))$. Additionally, there is a vector $b$ such that $y_0 = b \cdot power((\nu_1, \ldots, \nu_k))$. By substitution, we get a polynomial formula for $y_0$ in terms of $y_1, \ldots, y_j$. No term of this formula can contain $y_j^g$, $g \geq 2$, for $1 \leq j \leq k$ since this would violate the linear dependency of attributes in a linear attribute grammar.

□

**Corollary 1** *Suppose $\mu$ is a node of linear attribute grammar $T$. The dependence of a synthesized attribute $x_i(\mu)$ on the inherited attributes $h^R(\mu)$ can be expressed as $x_i(\mu) = a \cdot \mathrm{power}(h^R(\mu))$.*

The vector $a$ depends only on the calculations of attribute values performed at nodes in the subtree rooted at $\mu$, excluding the calculation of the inherited attributes of $\mu$.

**Corollary 2** *Suppose $\Pi$ is a path in linear attribute grammar $T$ with head $\sigma$ and tail $\tau$. Let $y$ be the vector consisting of the attributes $s^R(\sigma)$ followed by the attributes $h^R(\tau)$. The dependence of an inherited attribute $x_i(\sigma)$ on the attributes of $y$ can be expressed as $x_i(\sigma) = a \cdot \mathrm{power}(y)$ for some vector $a$. Similarly the dependence of synthesized attribute $x_j(\tau)$ on the attributes of $x$ can be expressed as $x_j(\tau) = b \cdot \mathrm{power}(y)$ for some vector $b$.*

The vectors $a$ and $b$ depends on the calculations of attribute values performed at nodes in the subtree rooted at $\tau$ excluding the subtree rooted at $\sigma$.

Suppose we are given a directed acyclic graph $G$. Graph $G$ is an *evaluation graph* if each vertex $v$ of $G$ calculates a value $value(v)$ by an equation $value(v) = a \cdot power(y)$, where $y$ is the vector of values of vertices $u$ with edges in $G$ from $u$ to $v$. A *topological sort* of $G$ orders the vertices of $G$ such that if there is an edge from vertex $u$ to vertex $v$ in $G$, then $u$ precedes $v$ in the ordering. A *simple evaluation scheme* calculates the values for each vertex of $G$ in topological order. In this way, the input values are available when calculating the value of a vertex.

For a node $\mu$ in linear attribute grammar $T$, corollary 1 shows that we can summarize the dependence of the attributes of $s^R(\mu)$ on the attributes of $h^R(\mu)$ as an evaluation graph $summarygraph(\mu)$. There is a vertex of $summarygraph(\mu)$ for each attribute of $\mu$. Edges are directed from attributes of $s^R(\mu)$ to attributes of $h^R(\mu)$. Similarly, for a path $\Pi$ in $T$ with head $\sigma$ and tail $\tau$, corollary 2 shows that we can summarize the dependence of the attributes of $h^R(\sigma)$ and $s^R(\tau)$ on the attributes of $s^R(\sigma)$ and $h^R(\tau)$ as an evaluation graph $summarygraph(\Pi)$. There is a vertex of $summarygraph(\Pi)$ for each attribute of $\sigma$ and $\tau$. Edges are directed from attributes of $s^R(\sigma)$ and $h^R(\tau)$ to attributes of $h^R(\sigma)$ and $s^R(\tau)$. Notice that for any node $\mu$ and any path $\Pi$, evaluation graphs $summarygraph(\mu)$ and $summarygraph(\Pi)$ will be of constant size.

**Lemma 6** *Let $T$ be a linear attribute grammar, $R(\mu)$ be the graph* summarygraph($\mu$), *$N(\mu)$ be the bidirectional binary operator $\odot$, and $P(\Pi)$ be the directed graph* summarygraph($\Pi$). *Then there exists a tree attribute system that supports tree attribute $R(\mu)$ by means of the path attribute $P(\Pi)$.*

**Proof:** Suppose $\Pi$ is a path in $T$ from node $\sigma$ to node $\tau$. Evaluation graph $summarygraph(\Pi)$ gives the dependencies of the attributes of $h^R(\sigma)$ and $s^R(\tau)$ on the attributes of $s^R(\sigma)$ and $h^R(\tau)$. Evaluation graph $summarygraph(\sigma)$ gives the dependencies of the attributes of $s^R(\sigma)$

on the attributes of $h^R(\sigma)$. We combine these two graphs and perform the simple evaluation scheme to get the dependencies of the attributes of $s^R(\tau)$ on the attributes of $h^R(\tau)$, which is $summarygraph(\tau)$. All this can be done in constant time, since these graphs are of bounded size.

To show that $P(\Pi)$ can be maintained in a path attribute system, consider path $\Pi'$ from head $\sigma'$ to $\tau'$ and path $\Pi''$ from head $\sigma''$ to $\tau''$ with node $\sigma''$ the parent of node $\tau'$. Suppose node $\mu$ is the sibling of node $\tau'$. The evaluation graph $summarygraph(\mu)$ will be part of the extended node attribute set $\tilde{\mathcal{N}}(\sigma'')$.

Suppose path $\Pi$ is the concatenation of paths $\Pi'$ and $\Pi''$. To find $summarygraph(\Pi)$, we form an evaluation graph $H$ with vertices for each of the attributes of $x(\sigma')$, $x(\tau')$, $x(\sigma'')$, $x(\tau'')$, and $x(\mu)$. The edges of $H$ are the edges of $summarygraph(\Pi')$, $summarygraph(\Pi'')$, $summarygraph(\mu)$, the dependencies for calculating $s^R(\sigma'')$ from $x(\tau')$ and $x(\mu)$, and the dependencies for calculating $h^R(\tau')$ from $s^R(\tau')$, $s^R(\mu)$, and $x(\sigma'')$. We perform the simple evaluation scheme on $H$ to determine the dependencies of the attributes of $h^R(\sigma')$ and $s^R(\tau'')$ on the attributes of $s^R(\sigma')$ and $h^R(\tau'')$, which is $summarygraph(\Pi)$ (see Fig. 6). All this can be done in constant time, since these graphs are of bounded size. $\square$

For a node $\mu$ of linear attribute grammar $T$, the value of $Evaluate(\mu)$ is the evaluation graph $summarygraph(\mu)$. In order to return the actual values of $x(\mu)$, we present the following operation for linear attribute grammars.

- *EvaluateAttributes(node $\mu$)* — Returns the values of the attributes $x(\mu)$.

We implement operation $EvaluateAttributes(\mu)$ as follows. First we call $Evaluate(\mu)$ to get evaluation graph $summarygraph(\mu)$. We then call $expose(\mu)$ to get path $\Pi$ from $\mu$ to root $\rho$. By combining the evaluation graphs $summarygraph(\Pi)$ and $summarygraph(\mu)$, and performing the simple evaluation scheme, we get the dependencies of $x(\mu)$ on $h^R(\rho)$. We then calculate the actual values of $x(\mu)$ by plugging in the default inherited value for the root $\rho$ of $T$. We finish by calling $conceal(\mu)$ to restore the size invariant.

**Theorem 6** *Let $\mathcal{S}$ be a semiring and $r$ an integer. There is a fully dynamic data structure for maintaining a set of linear attribute grammars over $\mathcal{S}^r$ with the following performance:*

1. *an linear attribute grammar of size $n$ uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* EvaluateAttributes, LocalUpdate, Link, Cut, Evert, Reflect, Cycle, *LCA,* PathFind, TreeFind, Parent, *and* Sibling, *take each $O(\log n)$ time, where $n$ is the total size of the linear attribute grammars involved in the operation.*

A binary linear operator is an *unbounded linear operator* if it keeps the expansion invariant. That is, if $\mu$ is a node of linear attribute grammar $T$, then $summarygraph(\mu)$ is unchanged after expanding any children of $\mu$. Note that the actual value of attributes may change as as a result of an *Expand* operation.

Consider the following example. Given a node $\mu$ in linear attribute grammar $T$, let $depth(\mu)$ be the depth of node $\mu$ in $T$. The attribute $depth(\mu)$ is inherited. The value $depth(\mu)$ is either 0, if $\mu$ is the root of $T$, or $depth(\nu)+1$, where node $\nu$ is the parent of $\mu$. If $\mu$ is an internal node and $\mu'$ is the leftmost child of $\mu$. Let $synthesizeddepth(\mu) = depth(\mu')-1$. Therefore, we get the relation $synthesizeddepth(\mu) = depth(\mu)$, which remains true after any

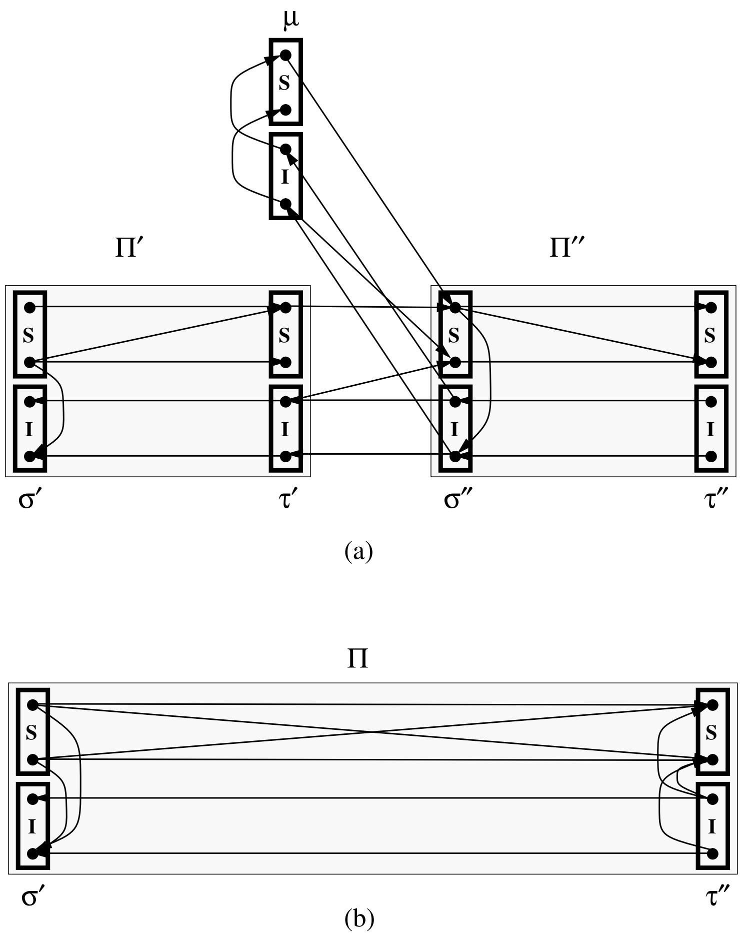


**Figure 6:** An example of the concatenation of summary graphs as described in lemma 6. (a) The combined evaluation graph. (b) The resulting summary graph

expansion of a node of $T$. Yet, when expanding children of $\mu$, we increment the depth of the descendants of the expanded nodes.

An *unbounded linear attribute grammar* is a linear attribute grammar with unbounded degree nodes such that each node calculates an unbounded linear operator.

By by lemma 2, lemma 6, and Theorem 2, we get:

**Theorem 7** *Let $\mathcal{S}$ be a semiring and $r$ an integer. There is a fully dynamic data structure for maintaining a set of unbounded linear attribute grammars over $\mathcal{S}^r$ with the following performance:*

1. *an linear attribute grammar of size $n$ uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Sibling *takes $O(\log n + k)$ time to return $k$ nodes.*
4. *operations* EvaluateAttributes, LocalUpdate, Link, Cut, Evert, Reflect, Cycle, *LCA* PathFind, TreeFind, Parent, Expand, *and* Contract *take each $O(\log n)$ time, where $n$ is the total size of the linear attribute grammars involved in the operation.*

# 4 Applications

In this section we show applications of tree attribute systems, linear expression trees, and linear attribute grammars. We show how a number of existing data structures can be expressed as tree attribute systems. We also present new applications using our techniques.

## 4.1 Tree Structures for Graph Problems

A number of existing data structures can be expressed as tree attribute systems. This list includes dynamic trees, dynamic expression trees, and edge ordered trees. In this section, we review these data structures, and existing applications for tree attribute systems, linear expression trees, and linear attribute grammars.

Dynamic trees [28] are unordered, unbounded, rooted trees supporting operations *Link*, *Cut* and *Evert*. Each node $\mu$ stores a globally updatable cost associated with the edge from $\mu$ to its parent. Global updates are only performed on the path from a node to the root. A specialized *PathFind* operator is included to find the minimum cost edge on a path. Dynamic trees were first presented as an internal data structure in several (sequential) maximum flow algorithms.

Dynamic expression trees [9] are a restricted form of linear expression trees. Values are scalars taken from a semi-ring $\mathcal{S}$. Operations are the $+$ and $\times$ operators of semi-ring $\mathcal{S}$. The major presented algorithm for dynamic expression trees is fully dynamic algorithms for maximum flow and shortest path in series-parallel digraphs.

In [8], linear expression trees are used to present fully dynamic algorithms for a large number of problems on tree-width two graphs, the class of graphs where each biconnected component is a biconnected series-parallel graph. Using linear expression trees augmented with additional node, path, and tree attributes, fully dynamic algorithms are given for queries such as the size of a vertex cover, dominating set, chromatic number, maximum independent set, connectible pairs shortest path, and connectible pairs maximum flow. Queries are performed in $O(\log m)$ time, while updates are performed in $O(\log^2 m)$ time for a graph with $m$ edges.

A number of data structures have been presented which maintain the recursive decomposition of graphs into subgraphs with constant size overlap. SPQ-trees [9] and SPQC-trees are used as a fully dynamic data structure to maintain a decomposition of series-parallel digraphs and tree-width two graphs. SPQR-trees [11] are used to maintain the tri-connected components of an initially biconnected graph. FWRT-trees [19] used to maintain the 4-connected components of an initially tri-connected graph. These last two data structures have a specialized *PathFind* operation to find the closest ancestor of a given type.

Edge-ordered trees [12] are ordered, unbounded, rooted trees supporting operations *Link*, *Cut*, *Expand*, *Contract*, *Cycle* and *Evert*. Each node $\mu$ stores a locally updatable cost associated with the edge from $\mu$ to its parent. A specialized *PathFind* operator is included to find the minimum cost edge on a path. Edge-ordered trees were presented as the data structure for a fully dynamic algorithm to maintain the minimum or maximum spanning tree of planar graphs.

In [7], simplified linear attribute grammars are used to maintain drawings of series-parallel graphs and trees. Queries include location of vertices and edges of the drawing, point location, and window queries. Queries take $O(k \log m)$, where $k$ is the number of vertices and edges returned, and $m$ is the number of edges of the tree or series-parallel graph.

## 4.2 Summation Heaps

In this section we present a generalization of heaps called summation heaps. Recall that in the tree implementation of a heap, the structure of the tree is arbitrary (see example 5). Heap elements are stored at the leaves of the tree. Internal nodes are only used to direct us when selecting the minimum value of a heap.

In summation heaps, the structure of the heap is important. We extend costs to include internal nodes. A summation heap is an expression tree over operators + and min. The cost of a node $\mu$ is the value of the subexpression rooted at $\mu$.

We replace operation *FindMin* for heaps, with the following operation.

- *FindMinNode*(*node* $\nu$) — return a minimum cost node $\mu$ in the subtree rooted at $\nu$.

To implement operation *FindMinNode*, we calculate two values for each node $\mu$:

- *cost*($\mu$) — the cost of node $\mu$.
- *mincost*($\mu$) — the minimum cost of a node in the subtree rooted at $\mu$.

These values are calculated as linear expressions The value of *cost*($\mu$) is defined to be the result of a linear expression. The value of *mincost*($\mu$) is the minimum of *cost*($\mu$) and *mincost* for the children of $\mu$. Operation *FindMin*($\mu$) is performed by calling operation *TreeFind*($\mu, S_4$), where $S_4$ is the selection function of example 5.

**Theorem 8** *There exists a fully dynamic $O(n)$ space data structure for maintaining a collection of summation heaps of total size $n$, such that a query or update operation takes time $O(\log n)$*

## 4.3 Blocking Heaps

In this section we present another generalization of heaps called blocking heaps, which are used in the implementation of the dynamic algorithm for tree-width two graphs.

Suppose $\mathcal{S}$ is a totally-ordered set. We consider operations in $\mathcal{S}^r$, the space of $r$-tuples of $\mathcal{S}$ for some fixed $r$. Let $x = (x_1, \ldots, x_r)$ and $y = (y_1, \ldots, y_r) \in \mathcal{S}^r$. A *binary minimization operator* $\odot : \mathcal{S}^r \times \mathcal{S}^r \to \mathcal{S}^r$ is defined such that each component $(x \odot y)_i$ calculates the minimum of some subset of the components of $x$ and some subset of the components of $y$. Given a component $\mu_i$ at node $\mu$, a *source* of $\mu_i$ is a component $\lambda_j$ at leaf $\lambda$ such that reducing the value of $\lambda_j$, reduces the value of $\mu_i$.

Consider the following operation on a tree associated with an expression of binary minimization operators:

- *FindSources*(*node* $\mu$) — return a source for each of the components of $\mu$.

**Definition 20** *A* blocking heap *is a linear expression tree of binary minimization operators which also supports operation* FindSources.

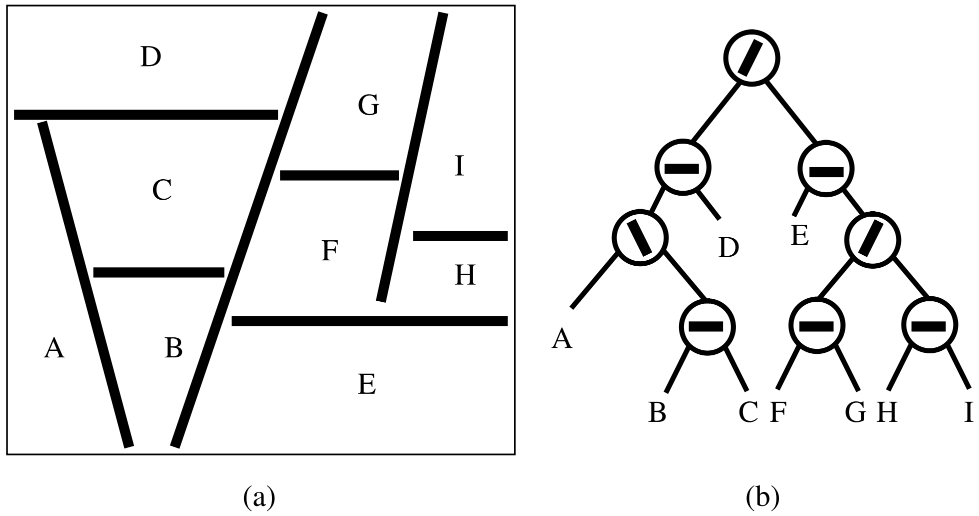


**Figure 7:** (a) A binary space partition of the plane into trapezoids. (b) The associated decomposition tree.

Suppose $T$ is a blocking heap. For each path $\Pi$ in $T$ from node $\sigma$ to node $\tau$, we keep the following boolean table:

- *depends*$(\Pi, i, j)$ — **true** if there is a path in the dependency graph of $T$ from $\sigma_i$ to $\tau_j$.

Operation *FindSources*$(\mu)$ is performed by calling operation *TreeFind*$(\mu, S_5, i)$, for each $1 \leq i \leq r$, where $S_5$ is the following tree selection function:

**Selection Function $\mathbf{S_5}$** *Suppose* $\Pi$ *is a path, node* $\zeta$ *the root of the path tree* $\mathcal{B}_\Pi$*, and nodes* $\zeta'$ *and* $\zeta''$ *the left and right children of* $\zeta$*. Let* $\tau'$ *be* tail$(\zeta')$*. Use the transfer pairs at* $\zeta$ *and* $\zeta'$*, the binary minimization operator at* head$(\zeta'')$*, and the* depends *table for* $\zeta''$ *to determine if there is a component* $\tau'_j$ *at* $\tau'$ *such that the value of* $\tau'_j$ *is the same as* $\mu_i$*, and* $\mu_i$ *depends on* $\tau_j$*. If so, return* $\zeta'$*, else return* $\zeta''$*.*

**Theorem 9** *There exists a fully dynamic* $O(n)$ *space data structure for maintaining a collection of blocking heaps of total size* $n$*, such that a query or update operation takes time* $O(\log n)$

## 4.4 Point Location in Unbalanced Binary Space Partition Trees

A binary space partition is a binary tree where each node corresponds to a connected region of the space (or plane) and to a geometric object that bipartitions such a region (see Fig. 7). E.g. a node corresponds to a trapezoid and to a segment that divides it into two trapezoids (for details see [22]). Each leaf $\lambda$ of binary space partition $T$ represents a region referred to as *region*$(\lambda)$. If the discrimination of a point with respect to the dividing object (i.e., figuring out which of the two children regions contains the point) can be done in polylog time (usually O(1) time), then one can do point location in polylog time. Previous approaches

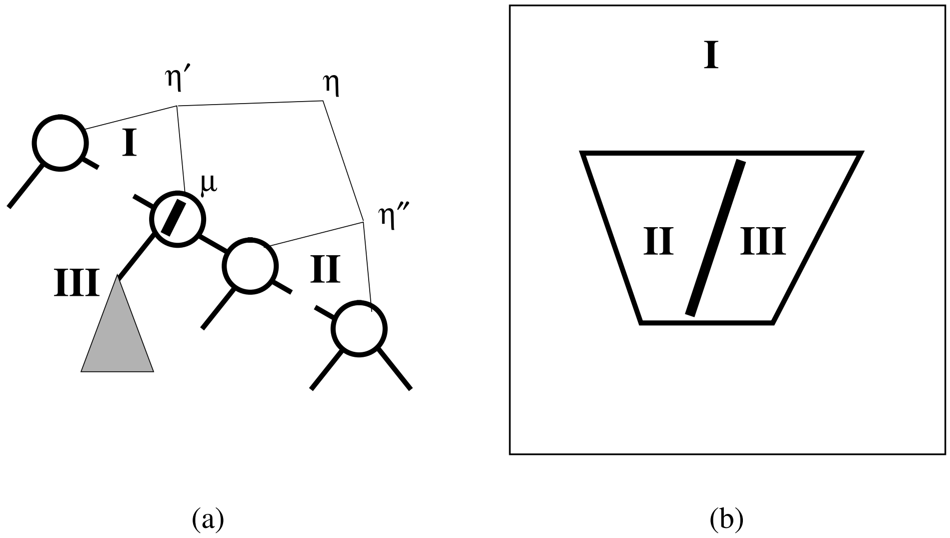


**Figure 8:** (a) A solid path in a decomposition tree; and (b) some regions of a binary space partition. For a point $p$, consider selection node $\zeta = S_6(\eta, p)$. If $\zeta = \eta''$ then $p$ is in regions I or III. If $\zeta = \eta'$ then $p$ is in regions II. If node $\mu = \mathit{find}(\Pi, S_6, p)$, then $p$ is in region III.

try to keep the binary space partition balanced [24,27]. With a selection function one can use unbalanced partitions. A specialized version of this technique was introduced in [4].

We support operations on a collection of binary space partitions:

- $\mathit{Region}(\mathit{node}\ \rho; \mathit{point}\ p)$ — Return the region containing point $p$ in the binary space partition rooted at node $\rho$.
- $\mathit{MakeBSP}()$ — Create a new elementary binary space partition with no partitions.
- $\mathit{DeleteBSP}(\mathit{node}\ \lambda)$ — Remove the elementary binary space partition represented by node $\lambda$.
- $\mathit{Divide}(\mathit{node}\ \lambda, \mathit{partition}\ P)$ — Use partition $P$ to divide the region represented by leaf $\lambda$.
- $\mathit{Compose}(\mathit{node}\ \lambda, \rho)$ — Partition the region represented by leaf $\lambda$ using the binary space partition with root $\rho$.
- $\mathit{Decompose}(\mathit{node}\ \mu)$ — Replace the sub-partition represented by node $\mu$ with a single region.

Operations *MakeBSP* and *delbsp* correspond to operations *MakeTree* and *DeleteTree* respectively. Operations $\mathit{Compose}(\lambda, \rho)$ and $\mathit{Decompose}(\mu)$ each are implemented with a constant number of *Link* and *Cut* operations.

Operation $\mathit{Divide}(\lambda, P)$ consists of replacing node $\lambda$ with a node $\mu$ representing partition $P$. Node $\mu$ has two children representing the regions created by partition $P$.

We implement operation $\mathit{Region}(\rho, p)$ using the following selection function, which takes a point $p$ as an argument.

**Selection Function $\mathbf{S_6}$** *Suppose $\eta$ is the root of a path tree with children $\eta'$ and $\eta''$. If $p$ is*

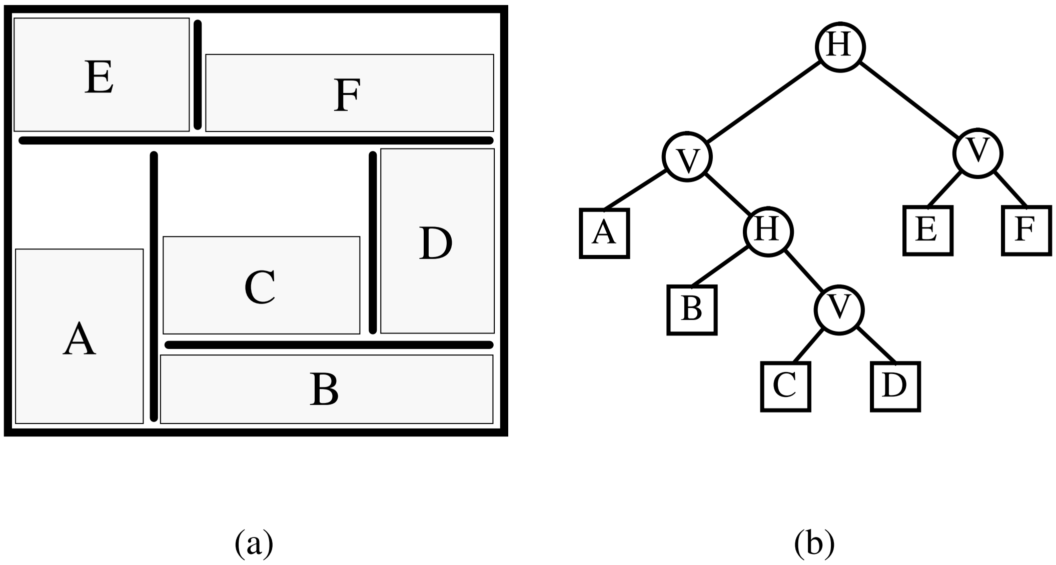


**Figure 9:** (a) A slicing floorplan; and (b) its associated parse tree.

*contained in* region(tail($\eta'$)), *then return* $\eta'$, *else return* $\eta''$.

Operation *TreeFind*$(\rho, S_6, p)$ returns a leaf $\lambda$ of $T$ such that point $p$ is contained in *region*$(\lambda)$ (see Fig. 8). We then return either *region*$(\lambda)$, or if $p$ is on the boundary of *region*$(\lambda)$, the portion of the boundary containing $p$.

**Theorem 10** *There is a fully dynamic data structure for maintaining a collection of unbalanced binary space partitions with the following performance:*

1. *operations* MakeBSP *and* DeleteBSP *take each* $O(q_1(n))$ *time, where* $q_1(n)$ *is the time required to build a single partition;*
2. *operations* Compose, Decompose, *and* Divide *take each* $O(q_1(n) \cdot \log n)$ *time;*
3. *operation* Region *takes* $O(q_2(n) \cdot \log n)$ *time, where* $q_2(n)$ *is the time to perform point location in a single region. The value* $n$ *is the total size of the binary space partitions involved in each operation.*

The trapezoid method for planar point location described in [6] is an example of an existing method that can be expressed as a selection function.

## 4.5 Slicing Floorplan Compaction

Linear attribute grammars have immediate application to the problem of compacting *slicing floorplans*, a layout technique widely used in VLSI (see, e.g., [30]). A slicing floorplan is either a rectangle (called basic rectangle), or is the union of two slicing floorplans that share a horizontal side (called horizontal slice) or a vertical side (called vertical slice). Each basic rectangle $r$ has a minimum width $w_r$ and a minimum height $h_r$, which describe the dimensions of a circuit module to be placed inside $r$.

An important problem in VLSI layout is determining the location of basic rectangles

| *Value* | *Vertical* | *Horizontal* |
|---|---|---|
| $height(\mu)$ | $\max(height(\mu_1), height(\mu_2))$ | $height(\mu_1) + height(\mu_2)$ |
| $width(\mu)$ | $width(\mu_1) + width(\mu_2)$ | $\max(width(\mu_1), width(\mu_2))$ |
| $xpos(\mu_1)$ | $xpos(\mu)$ | $xpos(\mu)$ |
| $ypos(\mu_1)$ | $ypos(\mu)$ | $ypos(\mu)$ |
| $xpos(\mu_2)$ | $xpos(\mu) + width(\mu_1)$ | $xpos(\mu)$ |
| $ypos(\mu_2)$ | $ypos(\mu)$ | $ypos(\mu) + height(\mu_1)$ |

Table 3 The equations to calculate the compaction of floorplans for a node $\mu$ and its children $\mu_1$ and $\mu_2$. The choice of equations depend on if $\mu$ represents a vertical or horizontal slice.

while minimizing the area of the slicing floorplan, subject to the above constraints on the height and width of the basic rectangles. (For full details, see [30].) This problem can be solved using a linear attribute grammar $T$. Leaves of $T$ represent basic rectangles, and internal nodes represent horizontal or vertical slices (see Fig. 9). The *reference point* of a sub-floorplan is its bottom-left corner.

We keep the following synthesized attributes at each node $\mu$ of linear attribute grammar $T$ with root $\rho$:

- $height(\mu)$ — The height of the sub-floorplan represented by node $\mu$.
- $width(\mu)$ — The width of the sub-floorplan represented by node $\mu$.

We also keep the following inherited attributes:

- $xpos(\mu)$ — The $x$-coordinate of the reference point of the sub-floorplan represented by node $\mu$. The value $xpos(\rho)$ is assumed to be 0.
- $ypos(\mu)$ — The $y$-coordinate of the reference point of the sub-floorplan represented by node $\mu$. The value $ypos(\rho)$ is assumed to be 0.

The equations to calculate these values is shown in table 3. Using linear attribute grammars, we can solve a dynamic version of the compaction problem for slicing floorplans, where query operations ask for the minimum area of a floorplan (or sub-floorplan) or the location of a basic rectangle and update operations change the dimensions of a basic rectangle or join (or separate) two floorplans.

**Theorem 11** *There exists a fully dynamic $O(n)$-space data structure for maintaining a collection of slicing floorplans of total size $n$, such that a query or update operation takes $O(\log n)$ worst-case time.*

# References


[1] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and F.K. Zadeck, "Incremental Evaluation of Computational Circuits," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 32–42.

[2] S.W. Bent, D.D. Sleator, and R.E. Tarjan, "Biased Search Trees," *SIAM J. Computing* 14 (1985), 545–568.

[3] G. M. Beshers and R. H. Campbell, "Maintained and Constructor Attributes," *Proc. ACM Symp. on Language Issues in Programming Environments, ACM SIGPLAN Notices* (1985), 34–42.

[4] Y.-J. Chiang, F.P. Preparata, and R. Tamassia, "A Unified Approach to Dynamic Point Location, Ray Shooting and Shortest Paths in Planar Maps," Dept. Computer Science, Brown Univ., Technical Report CS-92-07, 1992.

[5] Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," Dept. of Computer Science, Brown Univ., Technical Report CS-91-24, 1991.

[6] Y.-J. Chiang and R. Tamassia, "Dynamization of the Trapezoid Method for Planar Point Location," *Int. J. of Computational Geometry Applications* (to appear).

[7] R. F. Cohen, G. Di Battista, R. Tamassia, I.G. Tollis, and P. Bertolazzi, "A Framework for Dynamic Graph Drawing," *Proc. ACM Symp. on Computational Geometry* (1992).

[8] R.F. Cohen, S. Sairam, R. Tamassia, and J.S. Vitter, "Dynamic Algorithms for Bounded Tree-Width Graphs," Brown University, Manuscript, 1992.

[9] R.F. Cohen and R. Tamassia, "Dynamic Expression Trees and their Applications," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 52–61.

[10] G. Di Battista and R. Tamassia, "Incremental Planarity Testing," *Proc. 30th IEEE Symp. on Foundations of Computer Science* (1989), 436–441.

[11] G. Di Battista and R. Tamassia, "On-Line Graph Algorithms with SPQR-Trees," *Automata, Languages and Programming (Proc. 17th ICALP), Lecture Notes in Computer Science* 442 (1990), 598–611.

[12] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, and M. Yung, "Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph," *J. of Algorithms* 13 (1992), 33–54.

[13] G.N. Frederickson, "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications," *SIAM J. Computing* 14 (1985), 781–798.

[14] G.N. Frederickson, "Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and $k$ Smallest Spanning Trees," *Proc. 32th IEEE Symp. on Foundations of Computer Science* (1991).

[15] Z. Galil and G.F. Italiano, "Fully Dynamic Algorithms for Edge-Connectivity Problems," *Proc. 23th ACM Symp. on Theory of Computing* (1991), 317–327.

[16] Z. Galil and G.F. Italiano, "Maintaining Biconnected Components of Dynamic Planar Graphs," *Automata, Languages and Programming (Proc. 18th ICALP), Lecture Notes in Computer Science* (1991).

[17] L. G. Jones, "Incremental Compaction of Flat Symbolic IC Layouts," Department of Computer Science, University of Illinois, Urbana, Illinois, Technical Report No. UIUCDCS-R-87-1386, 1987.

[18] L. G. Jones and J. Simon, "Hierarchical VLSI Design Systems Based on Attribute Grammars," *Proc. 13th ACM Symp. on Principles of Programming Languages* (1986), 58–69.

[19] A. Kanevsky, R. Tamassia, J. Chen, and G. Di Battista, "On-Line Maintenance of the Four-Connected Components of a Graph," *Proc. 32th IEEE Symp. on Foundations of Computer Science* (1991), 793–801.

[20] D. E. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory* 2 (1968), 127–145.

[21] M. Overmars, "The Design of Dynamic Data Structures," *Lecture Notes in Computer Science* 156 (1983).

[22] M.S. Patterson and F.F. Yao, "Optimal Binary Space Partitions for Orthoganal Objects," *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms* (1990), 100–106.

[23] J.A. La Poutre, "Dynamic Graph Algorithms and Data Structures," Dept. of Computer Science, University of Utrechet, Utrechet, Ph.D. Thesis, 1991.

[24] F.P. Preparata, "A New Approach to Planar Point Location," *SIAM J. Computing* 10 (1981), 473–483.

[25] T.W. Reps, *Generating Language-Based Environments*, The MIT Press, 1984.

[26] T.W. Reps and T. Teitelbaum, *The Synthesizer Generator*, Springer-Verlag, 1989.

[27] C. Schwarz, M. Smid, and J. Snoeyink, "An Optimal Algorithm for the On-line Closest-Pair Problem," *Proc. ACM Symp. on Computational Geometry* (1992).

[28] D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees," *J. Computer Systems Sciences* 24 (1983), 362–381.

[29] D.D. Sleator and R.E. Tarjan, "Self-Adjusting Binary Search Trees," *J. ACM* 32 (1985), 652–686.

[30] L. Stockmeyer, "Optimal Orientation of Cells in Slicing Floorplan Design," *Information and Control* 57 (1983), 91–101.

[31] J. Westbrook and R.E. Tarjan, "Maintaining Bridge-Connected and Biconnected Components On-Line," *Algorithmica* (1989 (to appear in Algorithmica)).