Maintaining Tree Projections in Amortized $O(\log n)$ Time

Stephen B. Peckham

TR 89-1034
September 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This research was supported by NSF grant CCR-8514862 and ONR grant N000014-88K-0594.
Maintaining Tree Projections in Amortized $O(\log n)$ Time.

Stephen B. Peckham

Cornell University

Abstract

The projection of a set of marked nodes in a tree can be represented by a structure tree, that is, a subtree containing the marked nodes and the lowest common ancestors of all pairs of marked nodes. As an application modifies a forest of trees by linking and cutting trees and by marking and unmarking nodes, the structure tree associated with each tree must be updated in order to reflect the current set of marked nodes. Previous algorithms have used $O(n)$ time per operation [Hoo87] to maintain structure trees. This algorithm makes use of self-adjusting binary trees [ST85] and reduces the running time to amortized $O(\log n)$ per operation.

1 Introduction

Many algorithms make use of trees as a fundamental data structure. At times, it is useful to select a distinguished set of nodes in a tree and perform some operations just on those nodes. For example, Hoover [Hoo87] uses structure trees to keep track of uses of a particular defining occurrence of an attribute value. In [RMT86] and [Pec89], structure trees are used in an incremental attribute evaluation algorithm. If the trees are not modified and the relative position of nodes in the tree is not important, then the set of nodes can be maintained as a list in time $O(m)$ where $m$ is the number of distinguished nodes. On the other hand, if the trees are being modified as nodes are marked and unmarked, then a structure tree may be necessary. The term “structure tree” was introduced by Hoover, but the origin of the concept itself is unknown. Hoover's algorithm for maintaining structure trees [Hoo87] takes $O(n)$ time per operation, where $n$ is the number of nodes in the forest of trees. The algorithm presented here takes amortized $O(\log n)$ time per operation. It uses self-adjusting binary trees [ST85] as an auxiliary data structure in order to achieve this bound.
2 Structure Trees

A structure tree is a "structure-preserving projection of a tree onto a vertex subset" of the tree [Hoo87]. A graph projection is defined as follows:

Definition 1 Let $G = (V, E)$ be a graph with vertices $V$ and edges $E$, and let $N$ be a set of vertices with $N \subseteq V$. Then the projection of $G$ onto $N$, denoted $G \setminus N$, is a graph with vertices $N$ and edges

$$E' = \{ [v_1, v_2] \mid \text{there exists a path from } v_1 \text{ to } v_2 \text{ in } G \text{ that contains no vertices in } N \text{ other than } v_1 \text{ and } v_2 \}.$$ 

This definition assures us that if $v_1, v_2 \in N$ and $v_1$ is reachable from $v_2$ in $G$, then $v_1$ is reachable from $v_2$ in $G \setminus N$.

Definition 2 Let $T = (V, E)$ be a tree and let $N$ be a set of nodes $N \subseteq V$. Let $\hat{N}$ be the smallest subset of $V$ containing $N$ and closed under lowest common ancestor. That is

- if $v \in N$ then $v \in \hat{N}$, and
- if $v_1, v_2 \in N$ and $v$ is the lowest common ancestor of $v_1$ and $v_2$ in $T$, then $v \in \hat{N}$.

Let $T_N = T \setminus \hat{N}$. Then $T_N$ is the structure tree of $T$ with respect to $N$. $T$ is the underlying tree, $N$ is the set of marked nodes, and $\hat{N}$ is the set of structure tree nodes. Edges in $T_N$ are called structure tree edges.\(^1\)

Hoover [Hoo87] shows that this definition indeed defines a tree, and furthermore, each node in $\hat{N}$ has at most as many children in $T_N$ as the same node has in $T$. Finally, it is shown that $|\hat{N}| < 2|N|$, if $N \neq \emptyset$, and for each new node $v$ added to $N$, at most two new nodes must be added to $\hat{N}$: $v$, and if necessary, the lowest common ancestor of $v$ and another node in $N$.

Structure trees have several uses. They are used by Hoover to bypass copy rule chains in an incremental evaluator for an attribute grammar. They can also be used in algorithms for updating an attributed tree incrementally after multiple changes [RMT86,Pec89] and for displaying nodes that contain special values in a language-based editor [RT89].

\(^1\)Hoover adds the root of $T$ to the structure tree whenever $N$ is nonempty. We choose instead to associate with each tree a separate pointer to its structure tree root. Depending on the application, either option may be more appropriate.
3 Structure Tree Operations

We wish to maintain a forest of trees, for which each tree contains a set of marked nodes represented by a structure tree. The structure tree operations that we need are of two varieties. First, we must be able to create and modify underlying trees containing marked nodes. Second, we must be able to mark and unmark nodes in any underlying tree.

If two trees containing marked nodes are linked, a single underlying tree results, and a new structure tree is defined containing all the marked nodes originally contained in the two trees. Similarly, cutting a tree in two may cause an existing structure tree to be split into two structure trees.

All the structure tree operations must make sure that the structure tree nodes and edges for each underlying tree are correct according to the structure tree definition. Therefore, the structure tree associated with each underlying tree is maintained by each of the following operations:

\textbf{maketree}(v) Return a new tree containing the single unmarked node v.

\textbf{link}(T_1, T_2, w) Link trees T_1 and T_2 by adding an edge between the root of T_2 and node w, which must be in tree T_1. Tree T_1 will be updated, along with its associated structure tree, and tree T_2 will be destroyed.

\textbf{cut}(T, v) Split tree T into two trees by removing the edge between v and its parent. Node v must not be the root of T. Return the new tree whose root is v, along with its structure tree. Tree T and its structure tree will be updated.

\textbf{mark\_node}(T, v) Add node v to the set of marked nodes for tree T, which must contain node v. Update the structure tree in T as needed.

\textbf{unmark\_node}(T, v) Remove node v from the set of marked nodes for tree T, which must contain node v. Update the structure tree in T as needed.

In addition, for each node in the underlying tree, we will be able to determine whether the node is a structure tree node or is marked.

4 Representing Trees With Paths

We will represent underlying trees as a collection of paths, the same representation used by Sleator and Tarjan in the implementation of link-cut trees [ST83]. Each path consists of one or more nodes, and each node in the tree is a member of exactly one path. Elements of each path are ordered relative to their depths in the tree. (We
assume that edges are directed from nodes to their parents, for the purposes of this representation.) It is helpful to visualize trees using this representation by drawing them with solid edges between nodes that are in the same path, and dotted edges between the last node of a path and its parent. A path tree, that is, a tree represented by paths, is shown in Figure 1. The successor of the path whose last node is the root of the tree is null, so in the figure, the successor of the path containing b is null. For every other path, the successor is the parent of the last node in the path. E.g., the successor of the path containing p is e. Nodes with no solid edges incident to them (such as e or j) are the only members of one-node paths.

The basic operation used by Sleator and Tarjan to maintain link-cut trees is expose, which creates a single solid path starting at a node v and ending at the root of the tree. In the process, solid edges incident to this path are converted to dotted edges. For example, Figure 2 shows the tree in Figure 1 after we have executed expose(n). Although it is possible for a single expose to take $O(n)$ time, it will be seen that over a sequence of operations, and with an appropriate implementation for the path operations, all the tree operations can be performed in amortized $O(\log n)$ time.

5 Representing Paths With Self-Adjusting Binary Trees

In order to achieve the desired bound on running time, we use self-adjusting binary trees [ST85] to represent paths. The symmetric order of nodes in the self-adjusting binary tree is the same as the order of nodes in the path represented by the tree. Each binary tree representing a path is called a solid tree. Most of the path operations restructure solid trees by splaying [ST85], that is, moving a designated node to the
root of the solid tree while preserving the symmetric order. The path operations that we need are:

**makepath**(v) Create and return a new path containing the single node v.

**findpath**(v) Return the path containing the node v. (As a side-effect, node v will become the root of its solid tree.)

**findlast**(p) Return v, the last node of path p. (As a side-effect, node v will become the root of its solid tree.)

**join**(p, v, q) Create and return a new path containing the nodes of path p, a one-node path v, and path q. The nodes of p will be before v in the new path, and the nodes of q will be after v. Either p or q can be null.

**split**(v) Split the path containing v into three parts: the part before v, a one-node path containing v, and the part after v. Return the “before” and “after” paths as a pair, either of which can be null. Node v must be the root of its solid tree.

Tree operations can restructure a solid tree, but at all times, a path is designated by the node at the current root of its solid tree. This means that we can treat a path as a node in the algorithm, but we can only use a node as a path if we are sure that the node is the root of its solid tree.

A virtual tree shows the internal, self-adjusting binary tree representation of a path tree. Figure 3 shows a virtual tree corresponding to the path tree of Figure 1. Because the binary tree representation for a path is not unique, the virtual tree corresponding to a given path tree is not unique either.
6 Representing Structure Trees With Path Trees

We use path trees to represent structure trees by letting paths represent structure tree edges in addition to representing the underlying tree. Each structure tree edge is represented by a single path. The paths must satisfy the following structure tree invariant, if any structure tree nodes exist:

- There is a solid path from the structure tree root to the underlying tree root.

- All edges entering structure tree nodes are dotted. That is, each structure tree node is the beginning of a path.

- Each structure tree edge from a structure tree node \( v \) to its structure tree parent \( w \) is represented by a single solid path from \( v \) to a child \( x \) of \( w \) and a dotted edge from \( x \) to \( w \). If \( w \) has \( m \) children in the underlying tree, then we number the structure tree children of \( w \) from 1 to \( m \), even though some of them may be \textbf{null}. Therefore, if \( x \) is the \( i \)-th child of \( w \), then \( v \) will be the \( i \)-th structure tree child of \( w \).

A node \( v \) in the underlying tree is \textit{on-path} if it is part of a path representing a structure tree edge, but is not a structure tree node. Paths outside a structure tree are not constrained in any special way, but they are still modified by structure tree operations.

In order to maintain the structure tree invariant, we must modify the \textit{expose} operation. Otherwise, a single path created by \textit{expose} could contain many structure tree nodes, and reestablishing the structure tree invariant could take too much time. Instead, the \textit{expose} operation used here will create a solid path from a node \( v \) to a child of the first-encountered structure tree node on the path from \( v \) to the root of the underlying tree. (If no structure tree nodes are encountered, then \textit{expose} will create a solid path from \( v \) to the root, as usual.)

As structure tree operations are called, we need to be sure that the structure tree invariant is maintained. For example, we must be sure that \textit{mark\_node} splits paths when new structure tree edges are added, and that \textit{unmark\_node} joins paths whenever structure tree nodes are removed from the interior of a structure tree.

7 Data Structures Needed

In the underlying trees, we choose to provide direct pointers to the parent and children of each node. Each node in the underlying tree has the following fields:

\textbf{node parent} — The parent of the node (or \textbf{null} for the root node).
node child[1…m] — The children of the node.

integer child.no — The child number of the node with respect to its parent. Children are numbered from left to right. This field is undefined for the root of the tree.

boolean is.marked — True if and only if this node is in the marked set.

boolean is.structure.tree.node — True if and only if this node is a structure tree node. Marked nodes are always structure tree nodes.

In addition, structure tree nodes must have additional fields st.parent, st.child, and st.child.no, which are defined just like parent, child, and child.no, respectively. The st.parent field is null for the root of the structure tree, and st.child[i] is null if and only if no descendant of child[i] (including child[i] itself) is a structure tree node. If st.child[i] is not null, then st.child.no[st.child[i]] = child.no[child[i]] = i.

In order to maintain the path representation for underlying trees, we must add two or three fields (depending on the implementation) to each node. Our algorithm uses the parent and child pointers and maintains the extra fields used to keep track of the paths. (We assume that a client program maintains the parent and child pointers for link and cut operations.)

Finally, so that we can find the roots of structure trees, each tree variable in the algorithm will have two fields: root, the node that is the root of the underlying tree, and structure.tree.root, the root of the current structure tree, if the tree contains any structure tree nodes.

8 The Algorithm

The routines in this section are shown as concisely as possible, so we do not show the path operations. They can be found in [Tar83]. First, we introduce two functions that modify a virtual tree without affecting the underlying path tree.

8.1 Splice

Splice converts a dotted edge connecting a path p to its successor v into a solid edge. The successor of p must be v, and v must be the root of its structure tree root. If necessary, an existing solid edge entering v is converted into a dotted edge by splitting the original path containing v. Path p may be null in which case the returned path will have v as its first node. Otherwise, p is joined to the path containing v. The returned path will still have v as the root of its new solid tree.

In an actual implementation, splice would probably be expanded in-line, but it is shown separately here for clarity. The code for splice is in Figure 4.
8.2 Expose

The expose function is used by most of the structure tree operations. It creates a solid path beginning at a node \( v \) and ending at the root of the underlying tree, unless a structure tree node is encountered. A structure tree node is detected by the condition “\( v.is\_structure\_tree\_node \)” in the until statement. That is, if node \( v \), which is the successor of path \( p \) after each iteration of the repeat loop, is a structure tree node, then expose returns immediately, rather than continuing toward the root. The path returned by expose will be represented by the node where the last splice was performed. Expose does not maintain the structure tree invariant, so routines calling expose must make sure that the invariant is restored, if necessary. The code for expose is shown in Figure 5.

```plaintext
path function expose(node v);    node w;
    path p := null;
    repeat
        w := successor(findpath(v));    /* v becomes a solid tree root. */
        p := splice(v, p);
        v := w;
        until v = null or v.is_structure_tree_node;
        successor(p) := v;
    return p;
end expose;
```

Figure 5: Expose Operation
8.3 Maketree

Maketree creates a new tree containing a single, unmarked node \( v \). The code for maketree is shown in Figure 6.

```plaintext
tree function maketree(node v);
  tree t;
  t.root := v;
  t.structure.tree_root := null;
  successor(makepath(v)) := null;
  return t;
end maketree;
```

Figure 6: Maketree Operation

8.4 Mark_node

The mark_node operation adds a node \( v \) to the set of marked nodes, ensuring that the structure tree definition is still satisfied in the resulting tree. It is the most complicated of the structure tree operations, because there are so many ways that a new structure tree node can be placed into the existing structure tree. There are three primary cases:

Case 1 Node \( v \) is already a structure tree node, although it was not previously marked.

Case 2 No ancestor of \( v \) is in the structure tree.

Case 3 Some ancestor of \( v \) is in the structure tree.

The code in Figure 7 distinguishes the three main cases. As can be seen, Case 1 is trivial, since the structure tree is unchanged.

For the other two cases, we check the successor of the path returned by expose. If the successor is null, then expose reached the root, and no ancestor of \( v \) is in the structure tree. In this case, we have the three subcases shown in Figure 8 and described here:

Case 2.1 Node \( v \) is the first node to be added to the structure tree.

Case 2.2 Node \( v \) is an ancestor of the existing structure tree root \( r \). Node \( v \) will be the new structure tree root, and a single structure tree edge will be added between \( v \) and \( r \).
procedure mark_node(node v; tree t);
    path p;
    v.is_marked := true;
    if v.is_structure_tree_node then
        return; /* Case 1: v is already in structure tree */
    v.is_structure_tree_node := true;
    p := expose(v);
    if successor(p) = null then [Case 2 code] /* Expose reached the root */
    else [Case 3 code]
end mark_node;

Figure 7: Skeleton code for mark_node operation

Case 2.3 Node v is not an ancestor of the existing structure tree root r, so r', the
lowest common ancestor of v and r, will have to be added to the structure
tree, and two new structure tree edges will be added. Because a single solid
path existed from r to the root of the underlying tree, then r' is found easily,
since it will be the node returned by expose.

The code for Case 2 is shown in Figure 9. In all three subcases, the structure tree
will acquire a new root. New structure tree edges are made by calling one of two
functions: make_st_edge(m(p, c, m)), which makes c the m-th structure tree child
of p, or make_st_edge(p, c), which makes c the structure tree child of p, but calls
findlast(findpath(c)) to ascertain the proper child number.

If the successor of the path returned by expose is not null, then the successor
must be an existing structure tree node, and we have the third main case. Again,
we have three subcases, which are depicted in Figure 10:

Case 3.1 Node v is on-path. The existing structure tree edge containing v will be
split into two edges.

Case 3.2 Node v has an ancestor w that is on-path, and no structure tree or on-path
nodes occur between v and w. A structure tree edge will be added between v
and w, and the existing structure tree edge containing w will be split into two
edges.

Case 3.3 Node v has an ancestor x that is a structure tree node, and no structure
tree or on-path nodes occur between v and x. A new structure tree edge will
be added between v and x. No existing structure tree edges are affected.

The code for Case 3 is shown in Figure 11.
Case 2.1: The new node is first in the structure tree.

Case 2.2: The new node is an ancestor of the old root.

Case 2.3: An ancestor of the new node becomes the structure tree root.

Key: $\otimes$ = the node being marked
$\odot$ = a structure tree node added according to the structure tree definition
$\bullet$ = an existing structure tree node.
$\ldots\ldots$ = a new structure tree edge

Figure 8: Subcases for Case 2 of the Mark_node operation

/* Node $v$ is outside structure tree. The structure tree acquires a new root. */
if t.struct_tree_root $\neq$ null then
  r := t.struct_tree_root;
  r' := p; /* r' is the root of the solid tree representing path $p$. */
  if r' = v then make_st_edge(v, r); /* Subcase 2.2 */
  else /* Subcase 2.3 */
    successor(splice(r', null)) := null; /* Create path from $r'$ to root. */
    make_st_edge(r', v);
    make_st_edge(r', r);
    r'.is_structure_tree_node := true;
  end
else r' = v; /* Subcase 2.1 */
t.struct_tree_root := r'; /* All subcases */

Figure 9: Code for Case 2 of the Mark_node operation
Case 3.1: The new node is *on-path*.

Case 3.2: An ancestor of the new node is *on-path*.

Case 3.3: An ancestor of the new node is a structure tree node.

Key: $\otimes$ = the node being marked  
$\bigcirc$ = a structure tree node added according to the structure tree definition  
$\bullet$ = an existing structure tree node.  
$\ldots$ = a new structure tree edge  
$\ldots $ = an existing structure tree edge (which might be broken)

Figure 10: Subcases for Case 3 of the Mark_node operation

8.5 Path_compress

The *unmark_node* and *cut* operations will need to remove unmarked structure tree nodes that are no longer necessary according to the structure tree definition. An auxiliary procedure *path_compress* is used to check whether a node *v*, which must not be a structure tree leaf, should be removed from the structure tree. If *v* has more than one structure tree child, then it must remain in the tree. Otherwise, *v* is removed from the structure tree, while maintaining the structure tree invariant. We only need to deal with two cases. First, if *v* is the former root of the structure tree, then the only child of *v* will be the new root. Otherwise, *v* is removed from the structure tree, and a single structure tree edge replaces the two former structure tree edges incident to *v*. The *path_compress* function is shown in Figure 12. The function *only_st_child*(v) returns the structure tree child of *v* if the node has exactly one structure tree child. Otherwise, it returns *null*. 

12
/* successor(p) is a structure tree node, and v is a descendant of it. */
ode x := successor(p);
ode w := p;       /* Save representative node of path returned by expose. */
ode c := findlast(p);
if x.st_child[c.child_no] ≠ null then
  y := x.st_child[c.child_no];
  if w ≠ v then /* Subcase 3.2 */
    w := findpath(w);   /* Make sure w is solid tree (path) root. */
    successor(splice(w, null)) := x;
    make_st_edge(w, v);
    w.is_structure_tree_node := true;
  end
  make_st_edge(w, y);   /* Subcases 3.1 & 3.2 */
  make_st_edge_m(x, w, c.child_no);
else
  make_st_edge_m(x, v, c.child_no);   /* Subcase 3.3 */
Figure 11: Code for Case 3 of the Mark_node operation

procedure path_compress(node v; tree t);
  node child, parent;
  if v.is_marked then return;     /* Cannot remove marked node */
    child := only_st_child(v);
  if child = null then return;     /* Node v still needed. */
    parent := v.st_parent;
    v.st_parent := null;
    v.st_child[child.st_child_no] := null;
    v.is_structure_tree_node := false;
  if parent ≠ null then make_st_edge_m(parent, child, v.st_child_no);
else
    t.structure_tree_root := child;
    child.st_parent := null;
end
  successor(splice(v, findpath(child))) := parent;
end path_compress;

Figure 12: Path_compress Operation
8.6 Unmark_node

The code for unmark_node is much simpler than that for mark_node because it uses path_compress. Unmark_node unmarks a node \( v \) in a structure tree, possibly removing \( v \) and another node from the structure tree in order to satisfy the structure tree definition. Two cases need to be distinguished for unmark_node. First, if \( v \) is not a leaf in the structure tree, then path_compress is called in case \( v \) is no longer needed as a structure tree node. Otherwise, the structure tree edge between \( v \) and its parent \( w \) is removed, and path_compress is called with \( w \) as its argument. Of course, if \( v \) is a leaf node and the structure tree root as well, then \( v \) was the only node in the structure tree, and the structure tree becomes empty. Code for unmark_node is in Figure 13. The findpath calls are needed, because the first argument to path_compress must be a solid tree root.

```
procedure unmark_node(node v; tree t);
    node w := v.st_parent;
    v.is_marked := false;
    v := findpath(v);  /* v becomes a structure tree root. */
    if v has no structure tree child then /* Removing structure tree leaf */
        if w = null then t.structure_tree_root := null;
        else
            w.st_child[v.st_child_no] := null;
            v.is_structure_tree_node := false;
            v.st_parent := null;
            path_compress(findpath(w), t);
        end
    else path_compress(v, t);
end unmark_node;
```

Figure 13: Unmark_node Operation

8.7 Link

The link operation links trees \( t_1 \) and \( t_2 \) by adding an edge from the root of \( t_2 \) to node \( w \), which must be in \( t_1 \). To simplify link, we mark node \( w \) temporarily if it is not already a structure tree node. Once this has been done, the structure trees in \( t_1 \) and \( t_2 \) can be linked easily. Afterwards, if we marked node \( w \), then we unmark the node. removing \( w \) from the structure tree if it is not needed. The code for the link operation is in Figure 14.
procedure link(tree t1, t2; node w);
    node v := t2.root;
    boolean remove := not w.is_structure_tree_node;
    if remove then mark_node(w, t1);
    successor(findpath(v)) := w;
    if t2.structure_tree_root ≠ null then
        make_st_edge_m(w, t2.structure_tree_root, v.child_no);
    if remove then unmark_node(w, t1);
end link;

Figure 14: Link Operation

8.8 Cut
The cut function removes a subtree rooted at a node v from an existing tree t by removing the edge from v to its parent. The removed subtree is returned. By marking node w, the parent of v, we can easily determine whether the subtree being cut contains any structure tree nodes. Afterwards, w is unmarked, if necessary. The code for cut is in Figure 15.

tree function cut(tree t; node v)
    node w := v.parent;  /* Node v must have a parent. */
    tree t';
    boolean remove := not w.is_structure_tree_node;
    if remove then mark_node(w, t);
    t'.root := v;
    successor(findpath(v)) := null;
    t'.structure_tree_root := w.st_child[v.child_no];  /* may be null */
    if w.st_child[v.child_no] ≠ null then
        w.st_child[v.child_no].st_parent := null;
        w.st_child[v.child_no] := null;
    if remove then unmark_node(w, t) else path_compress(findpath(w), t);
return t';
end cut;

Figure 15: Cut Operation
9 Analysis of Running Time

In analyzing the running time of these routines, much of the analysis in [Tar83] can be used. The main difference has to do with the expose operation. Because expose as described here does not always create a solid path all the way to the root, the analysis must be modified. First, we need a few definitions regarding nodes in the underlying tree.

**Definition 3** The size of a node $v$ is the number of descendants of $v$ in the underlying tree, including $v$ itself.

**Definition 4** An edge from $v$ to its parent $w$ is heavy if $2 \cdot \text{size}(v) > \text{size}(w)$ and light otherwise.

**Lemma 1** If $v$ is any node, there is at most one heavy edge entering $v$.

**Lemma 2** There are at most $\lceil \log n \rceil$ light edges on the tree path from $v$ to the tree root.

Proof: The proof of Lemma 1 is obvious. For Lemma 2, we call a node light if the edge to its parent is light. If the size of a light node is $x$, then the size of its parent must be at least $2x$. If there are more than $\lceil \log n \rceil$ light edges on the tree path from $v$ to the tree root, then the weight of the root must be at least $2^{\lceil \log n \rceil + 1} > n$, a contradiction. □

These lemmas allow us to prove the following theorem (cf. [Tar83]).

**Theorem 1** A sequence of $m$ structure tree operations including $n$ maketree operations requires $O(m)$ path operations and in addition, at most $m$ calls to expose. The expose calls require $O(m \log n)$ splices, each of which requires $O(1)$ time.

Proof: The split and join calls in splice are constant time operations, since no splaying is needed. Therefore, each splice takes constant time. Each structure tree operation requires a constant number of path operations, so $m$ structure tree operations require $O(m)$ path operations. Mark\_node can call expose, as can link and cut since they can call mark\_node. Unmark\_node and maketree do not call expose at all. So overall, $m$ structure tree operations require at most $m - n < m$ expose calls.

Next, we prove that the $m$ expose calls require $O(m \log n)$ splice calls. By Lemma 2, for any given expose, each of at most $\lceil \log n \rceil$ splices will turn a light, dotted edge into a light, solid edge. Each of the rest of the splices will convert a heavy, dotted edge into a heavy, solid edge, increasing the number of heavy, solid edges by one, since any edge converted to dotted must be light.
We bound the number of heavy, solid edges created by noting that after the \( m \) operations, the tree has at most \( n - 1 \) edges. Therefore, the number of heavy, solid edges created by the \( m \) `expose` calls is at most \( n - 1 \) plus the number of heavy, solid edges destroyed by the operations.

Each call to `expose` destroys at most \( \lceil \log n \rceil + 1 \) heavy, solid edges, possibly one for the first `splice` call, and at most one for each light, solid edge created. In addition to the heavy, solid edges destroyed by `expose`, the other operations can destroy additional heavy, solid edges as follows:

**maketree, unmark_node** No edges are created or affected by `maketree`, and no solid edges are destroyed by `unmark_node`.

**link** `Link` (making a node \( w \) the parent of a node \( v \)) increases the size of all nodes on the path from \( w \) to the root of the underlying tree, possibly making those edges heavy, and making edges incident to the path light. At most \( \lceil \log n \rceil \) edges incident to the path can be heavy, so `link` can destroy at most \( \lceil \log n \rceil \) additional heavy, solid edges. But these heavy, solid edges would have been destroyed anyway if `expose` had reached the root, so at most \( \lceil \log n \rceil \) solid, heavy edges are destroyed in total by the `expose` and by the addition of the edge from \( v \) to \( w \).

**cut** `Cut` can convert edges on the path from the cut node to the root from heavy to light, but at most \( \lceil \log n \rceil \) of them can be made light. In addition, the cut edge might have been solid and heavy, so `cut` can destroy as many as \( \lceil \log n \rceil + 1 \) heavy, solid edges.

**mark_node** `Mark_node` can call `splice`, splitting the path returned by its call to `expose`, so at most one heavy, solid edge can be destroyed.

When we combine these bounds, we see that at most \( O(m \lceil \log n \rceil) \) solid, heavy edges are destroyed by the `expose` calls, and \( O(m \lceil \log n \rceil) \) additional solid, heavy edges are destroyed by the other operations. So \( O(n - 1 + m \lceil \log n \rceil) \) solid, heavy edges are created by the exposes. Adding the \( O(m \lceil \log n \rceil) \) splices that do create solid, light edges, our theorem is proved. \( \square \)

To prove an overall amortized bound on the operations, we use a modified definition of `size`:

\[
\text{size}(v) = \begin{cases} 
1 & \text{if } v \text{ is a structure tree node,} \\
1 + \sum_i \text{size}(i), \text{for all children } i \text{ of } v & \text{otherwise.}
\end{cases}
\]

In other words, `size(v)` treats the underlying tree as if it were partitioned into a forest of trees by removing every edge between a structure tree node and its children.
Then we use an accounting scheme that depends on the paths used to represent the tree. Each node in a path tree has an individual weight

\[
iw(v) = \begin{cases} 
\overline{\text{size}}(v) - \overline{\text{size}}(u) & \text{if } [u,v] \text{ is a solid edge,} \\
\overline{\text{size}}(v) & \text{otherwise.}
\end{cases}
\]

When a node \( v \) becomes a structure tree node, the \( \overline{\text{size}} \) of nodes above \( v \) can be affected, but except for \( v \), no individual weights are affected. The total weight \( tw(v) \) of a node \( v \) is the sum of the individual weights of the descendants of \( v \) in the solid tree containing \( v \). For any node \( v \), \( tw(v) \) is the number of descendants of \( v \) in the virtual tree containing it, not including any proper descendants of a structure tree node \( w \) that is a descendant of \( v \). Finally, let \( rank(v) = \lfloor \log tw(v) \rfloor \). We wish to maintain \( rank(v) \) credits on each node at all times. Each structure tree operation will be assessed an amortized cost of \( O(\log n) \) credits. Credits not used to pay for the actual operation will be used to maintain the invariant. Extra operations can be accounted for by credits released during structure tree operations.

We will prove that if we use self-adjusting binary trees, we obtain an \( O(m \log n) \) time bound on a sequence of \( m \) structure tree operations. We state the following lemma from Tarjan without proof [Tar83].

**Lemma 3** Splaying a solid tree with root \( v \) at a node \( x \) while maintaining the credit invariant takes \( 3(rank(v) - rank(x)) + 1 \) credits.

Since \( 1 \leq tw(v) \leq n \) for all nodes \( v \), then \( 0 \leq rank(v) \leq \log n \) for all nodes, and so a single solid path operation takes \( O(\log n) \) credits.

Examining the expose operation, we see that the first findpath(\( v \)) operation takes \( 3(rank(u) - rank(v)) + 1 \) credits where \( u \) is the root of the solid tree containing \( v \). After the findpath call, \( v \) will be the root of its solid tree. The split and join operations in splice do not affect any total weights, so one additional credit pays for the rest of the splice. The successor of \( v \), which is the next node that will be splayed, will have rank at least as great as that of \( u \), unless it is a structure tree node, in which case expose returns.

Summing over all splices during an expose, we notice that the sum telescopes, and so the total number of credits needed is \( 3(rank(w) - rank(v)) \) plus two per splice, where \( v \) is the node exposed and \( w \) is the root of the last solid tree reached by the expose. By Theorem 1, we conclude that the number of credits needed for all \( O(m) \) exposes required by \( m \) tree operations is \( O(m \log n) \).

Total weights can be affected by other operations, as follows:

**maketree** Node \( v \), the only node in the new tree, will have \( rank(v) = 0 \).

**mark_node** This operation can only reduce total weights, since the individual weight of the marked node is reduced from its previous value to one.

18
link  *Link* adds an edge from a subtree root \( v \) to an existing node \( w \). Because \( w \) is marked first, \( iw(w) = 1 \) before and after the edge \([v, w]\) is added. Therefore, *link* does not affect any total weights or ranks, other than those affected by *mark_node* and *unmark_node*.

cut  *Cut*, like *link*, marks \( w \), the parent of \( v \). Therefore, removing the subtree rooted at \( v \) doesn't affect any total weights or ranks.

path_compress  *Path.compress*, which can unmark a node \( v \), calls *findpath* on any node being unmarked, making it a solid tree root. After \( v \) is removed from the structure tree, its individual weight increases. But the only total weight affected is \( v \)'s, because it is the root of its solid tree. Therefore, by adding \( O(\log n) \) credits to \( v \), we restore the credit invariant.

unmark_node  *Unmark_node*, which can be called by *link* and *cut*, can modify the individual weight of a node, by removing it from the structure tree. If the node being removed is a leaf in the structure tree, then the *findpath*(\( v \)) call makes \( v \) the root of its solid tree. As with *path.compress*, an extra \( O(\log n) \) credits takes care of any violation of the credit invariant. If we need to remove other nodes from the structure tree, we call *path.compress* to do the work, so an additional \( O(\log n) \) credits might be needed to maintain the invariant.

Taking all these operations into account, we have the following theorem:

**Theorem 2**  *If we use self-adjusting binary trees to represent solid paths, a sequence of \( m \) structure tree operations including \( n \) maketree operations requires \( O(m \log n) \) time.*

10  **Conclusion**

The algorithm presented here is an improvement over currently published methods for maintaining structure trees. While some additional storage overhead is incurred to maintain paths as self-adjusting binary trees, it is modest, and the improvement in execution time should outweigh the storage penalty.
References


