CHAPTER 5

Linking and Cutting Trees

5.1. The problem of linking and cutting trees. The trees we have studied in Chapters 2, 3 and 4 generally occur indirectly in network algorithms, as concrete representations of abstract data structures. Trees also occur more directly in such algorithms, either as part of the problem statement or because they express the behavior of the algorithm. In this chapter we shall study a generic tree manipulation problem that occurs in many network algorithms, including a maximum flow algorithm that we shall study in Chapter 8.

The problem is to maintain a collection of vertex-disjoint rooted trees that change over time as edges are added or deleted. More precisely, we wish to perform the following operations on such trees, each of whose vertices has a real-valued cost (see Fig. 5.1):

maketree (v): Create a new tree containing the single vertex v, previously in no tree, with cost zero.

findroot (v): Return the root of the tree containing vertex v.

findcost (v): Return the pair $[w, x]$ where x is the minimum cost of a vertex on the tree path from v to findroot (v) and w is the last vertex (closest to the root) on this path of cost x.

addcost (v, x): Add real number x to the cost of every vertex on the tree path from v to findroot (v).

link (v, x): Combine the trees containing vertices v and w by adding the edge $[v, w]$. (We regard tree edges as directed from child to parent.) This operation assumes that v and w are in different trees and v is a root.

cut (v): Divide the tree containing vertex v into two trees by deleting the edge out of v. This operation assumes that v is not a tree root.

In discussing this problem we shall use m to denote the number of operations and n to denote the number of vertices (maketree operations). One way to solve this problem is to store with each vertex its parent and its cost. With this representation each maketree, link, or cut operation takes $O(1)$ time, and each findroot, findcost, or addcost operation takes time proportional to the depth of the input vertex, which is $O(n)$.

By representing the structure of the trees implicitly, we can reduce the time for findroot, findcost and addcost to $O(\log n)$, amortized over a sequence of operations, while increasing the time for link and cut to $O(\log n)$. In the next two sections we shall study a data structure developed by Sleator and Tarjan [4] that has these bounds.

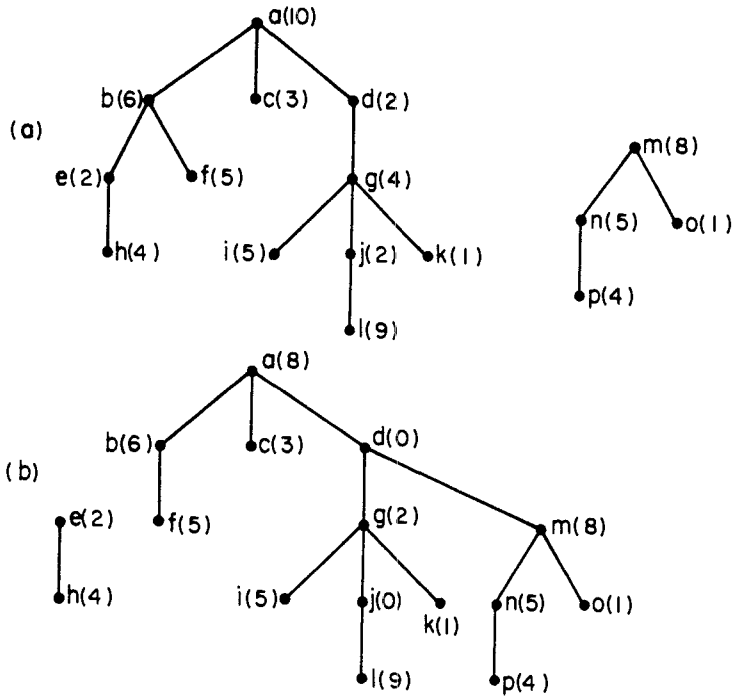

FIG. 5.1. *Tree operations.* (a) *Two trees. Numbers in parentheses are vertex costs. Operation* findroot (j) *returns* a, findcost (j) *returns* $[d, 2]$. (b) *After* addcost ($j, -2$), link (m, d), cut (e).

5.2. Representing trees as sets of paths. There are no standard data structures that directly solve the problem of linking and cutting trees. However, there do exist methods for the special case in which each tree is a path. In this section we shall assume the existence of such a special-case method and use it to solve the general problem.

Suppose we have a way to carry out the following operations on vertex-disjoint paths, each of whose vertices has a real-valued cost:

makepath (v): Create a new path containing the single vertex v, previously on no path, with cost zero.

findpath (v): Return the path containing vertex v.

findtail (p): Return the tail (last vertex) of path p.

findcost (p): Return the pair $[w, x]$, where x is the minimum cost of a vertex on path p and w is the last vertex on p of cost x.

addpathcost (p, x): Add real number x to the cost of every vertex on path p.

join (p, v, q): Combine path p, the one-vertex path containing only v, and path q into a single path by adding an edge from the tail of p to v and an edge from v to the head (first vertex) of q. Return the new path. This operation allows either p or q to be empty, in which case the corresponding edge is not added.

split (v): Divide the path containing vertex v into at most three paths by deleting the edges incident to v. Return the pair $[p, q]$, where p is the part of the original path before but not including v, and q is the part after but not including v. If v is the head of the original path, p is empty; if v is the tail, q is empty.

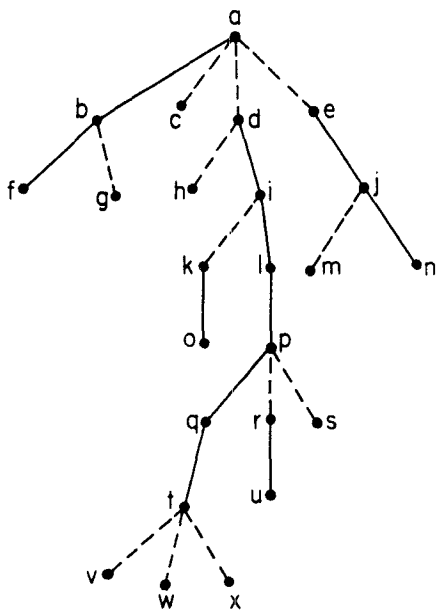

FIG. 5.2. *A tree partitioned into solid paths. Path* [*t, q, p, l, i, d*] *has head t and tail d.*

By using these path operations as primitives, we can solve the problem of linking and cutting trees. We partition each tree into a set of vertex-disjoint paths and carry out each tree operation by means of one or more path operations. To define the paths, we partition the edges of each tree into two kinds, *solid* and *dashed,* so that at most one solid edge enters any vertex. (See Fig. 5.2.) The solid edges partition the tree into vertex-disjoint *solid paths.*

To represent the dashed edges, we store with each solid path p its *successor,* defined to be the vertex entered by the dashed edge leaving the tail of p; if the tail of

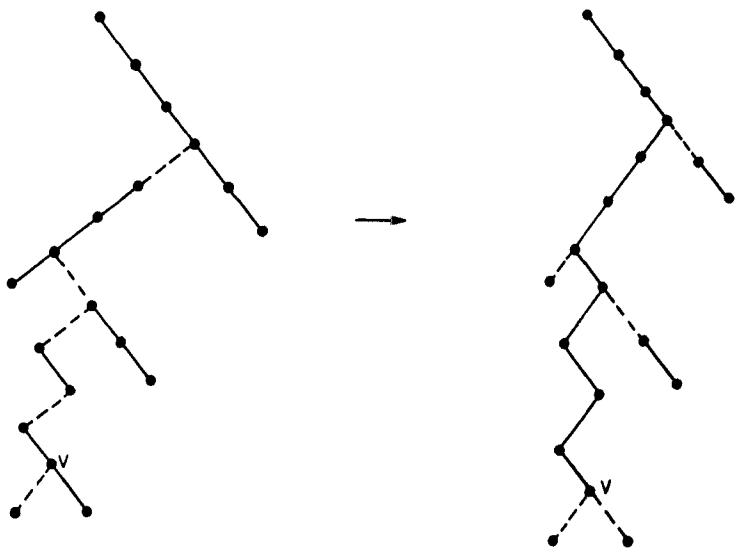

FIG. 5.3. *Exposing a vertex v.*

p is a tree root, its successor is **null.** To carry out the tree operations we need the following composite operation on paths (see Fig. 5.3):

expose (v): Make the tree path from v to findroot (v) solid by converting dashed edges along the path to solid and solid edges incident to the path to dashed. Return the resulting solid path.

Our implementation of the dynamic tree operations identifies each path by a vertex on it and uses **null** to identify the empty path. The following programs implement the six tree operations with the help of expose:

```
procedure maketree (vertex v);
   makepath (v);
   successor (v) := null
end maketree;

vertex function findroot (vertex v);
   return findtail (expose (v))
end findroot;

list function findcost (vertex v);
   return findpathcost (expose (v))
end findcost;

procedure addcost (vertex v, real x);
   addpathcost (expose (v), x)
end addcost;

procedure link (vertex v, w);
1. successor (join (null, expose (v), expose (w))) := null
end link;

procedure cut (vertex v);
   path p, q;
2. expose (v);
3. [p, q] := split (v);
   successor (v) := successor (q) := null
end cut;
```

Notes. The expose in line 1 of link makes v into a one-vertex solid path. After line 2 in cut, vertex v has no entering solid edge; thus the path p returned by the split in line 3 is empty. Since in cut vertex v is not a tree root, the path **q** returned in line 3 is nonempty. □

The following program implements expose:

```
path function expose (vertex v);
   path p, q, r; vertex w;
   p := null;
   do v ≠ null →
      w := successor (findpath (v));
```

```
    1. [q, r] := split (v);
       if q ≠ null → successor (q) := v fi;
    2. p := join (p, v, r);
       v := w
    od;
    successor (p) := null
end expose;
```

Note. The solid edge from v to the head of r broken by the split in line 1 is restored by the join in line 2. Each iteration of the **do** loop converts to solid the edge from the tail of p to v (if $p \neq$ **null**) and to dashed the edge from the tail of q to v (if $q \neq$ **null**). □

We shall call each iteration of the **do** loop in expose a *splice*. Each tree operation takes $O(1)$ path operations and at most one expose. Each splice within an expose takes $O(1)$ path operations. At this level of detail, the only remaining task is to count the number of splices per expose. In the remainder of this section we shall derive an $O(m \log n)$ bound on the number of splices caused by a sequence of m tree operations. This bound implies that there are $O(\log n)$ splices per expose amortized over the sequence.

To carry out the proof we need one new concept. We define the *size* of a vertex v to be the number of descendants of v, including v itself. We define an edge from v to its parent w to be *heavy* if $2 \cdot size\ (v) > size\ (w)$ and *light* otherwise. The following result is obvious.

LEMMA 5.1. *If v is any vertex, there is at most one heavy edge entering v and there are at most $\lfloor \lg n \rfloor$ light edges on the tree path from v to* findroot (v).

To bound the number of splices, we consider their effect on the number of tree edges that are both solid and heavy. There are no such edges (indeed, there are no edges at all) initially, and there are at most $n - 1$ such edges after all the tree operations. Consider exposing a vertex. During the expose, each splice except the first converts a dashed edge to solid. At most $\lfloor \lg n \rfloor$ of these splices convert a light dashed edge to solid. Each splice converting a heavy dashed edge to solid increases the number of heavy solid edges by one, since the edge it makes dashed, if any, is light. Thus m tree operations cause a total of $m\ (\lfloor \lg n \rfloor + 1)$ splices plus at most one for each heavy solid edge created by the tree operations.

To bound the number of heavy solid edges created, we note that all but $n - 1$ of them must be destroyed. Each expose destroys at most $\lfloor \lg n \rfloor + 1$ heavy solid edges, at most one for each splice that does not increase the number of heavy solid edges. Links and cuts can also destroy heavy solid edges by changing node sizes.

An operation link (v, w) increases the size of all nodes on the tree path from w to findroot (w), possibly converting edges on this path from light to heavy and edges incident to this path from heavy to light. After the operation expose (w) in the implementation of link, the edges incident to the path are dashed, and adding the edge $[v, w]$ thus converts no solid edges from heavy to light.

An operation cut (v) decreases the size of all nodes on the tree path from the parent of v to findroot (v), converting at most $\lfloor \lg n \rfloor$ heavy edges to light. The cut

also deletes a solid edge that may be heavy. Thus a cut destroys at most $\lfloor \lg n \rfloor + 1$ heavy solid edges, not including those destroyed by the expose that begins the cut.

Combining our estimates, we find that at most $(3m/2)(\lfloor \lg n \rfloor + 1)$ heavy solid edges are destroyed by m tree operations, since there are at most $m/2$ cuts (an edge deleted by a cut must have been added by a previous link). Thus the total number of splices is at most $(5m/2)(\lfloor \lg n \rfloor + 1)$, and we have the following theorem:

THEOREM 5.1. *A sequence of m tree operations including n maketree operations requires $O(m)$ path operations and in addition at most m exposes. The exposes require $O(m \log n)$ splices, each of which takes $O(1)$ path operations.*

5.3. Representing paths as binary trees, To complete our solution to the problem of linking and cutting trees, we need a way to represent solid paths. We shall represent each solid path by a binary tree whose nodes in symmetric order are the vertices on the path. (See Fig. 5.4.) Each node x contains three pointers: *parent* (x), *left* (x) and *right* (x), to its parent, left child and right child, respectively. We use the root of the tree to identify the path; thus the root contains a pointer to the successor of the path. To distinguish between the trees defined by the link and cut operations and the trees representing solid paths, we shall call the latter *solid trees.*

To represent vertex costs, we use two real-valued fields per node. For any vertex x, let *cost* (x) be the cost of x, and let *mincost* (x) be the minimum cost of a descendant of x in its solid tree. With x we store $\Delta cost\ (x)$ and $\Delta min\ (x)$, defined as follows (see Fig. 5.4):

$$\Delta cost\ (x) = cost\ (x) - mincost\ (x),$$

$$\Delta min\ (x) = \begin{cases} mincost\ (x) & \text{if } x \text{ is a solid tree root,} \\ mincost\ (x) - mincost\ (parent\ (x)) & \text{if } x \text{ is not a solid tree root.} \end{cases}$$

Note that $\Delta cost\ (x) \geq 0$ for any vertex x and $\Delta min\ (x) \geq 0$ for any nonroot vertex x. Given $\Delta cost$ and Δmin, we can compute *mincost* (x) for any vertex x by

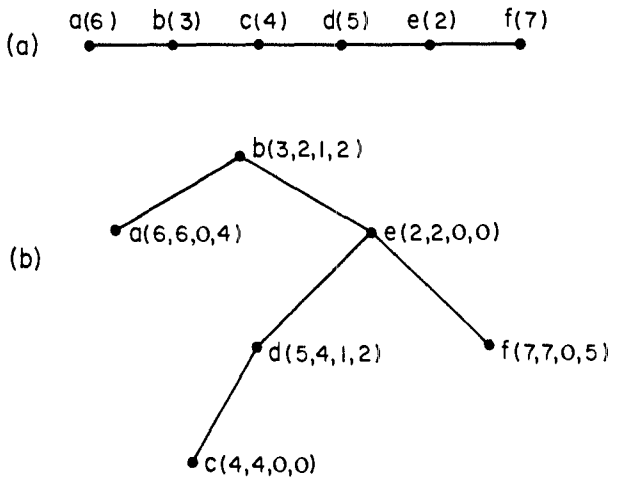

FIG. 5.4. *Representation of a path.* (a) *Path with head a and tail f.* (b) *Binary tree representing the path. Numbers labeling nodes are cost, mincost, Δcost, and Δmin, respectively.*

summing Δmin along the solid tree path from the root to x and compute $cost\ (x)$ as $mincost\ (x) + \Delta cost\ (x)$.

With this representation, we can perform a single or double rotation in $O(1)$ time. We can also perform in $O(1)$ time two operations that assemble and disassemble solid trees:

assemble (u, v, w): Given the roots u, v, w of three solid trees, such that the tree with root v contains only one node, combine the trees into a single tree with root v and left and right subtrees the trees rooted at u and w, respectively. Return v.

disassemble (v): Given the root v of a solid tree, break the tree into three parts, a tree containing only v and the left and right subtrees of v. Return the pair consisting of the roots of the left and right subtrees.

Note. The effect of a rotation depends on whether or not the rotation takes place at the root of a solid tree. In particular, if we rotate at a root we must move the sucessor pointer for the corresponding solid path from the old root to the new root. □

Since assembly, disassembly and rotation are $O(1)$-time operations, we can use any of the kinds of binary search trees discussed in Chapter 4 to represent solid paths. We perform the six path operations as follows:

makepath (v): Construct a binary tree of one node, v, with $\Delta min\ (v) = \Delta cost\ (v) = 0$.

findpath (v): In the solid tree containing v, follow parent pointers from v until reaching a node with no parent; return this node.

findtail (p): Node p is the root of a solid tree. Initialize vertex v to be p and repeatedly replace v by $right\ (v)$ until $right\ (v) =$ **null.** Then return v.

findpathcost (p): Initialize vertex w to be p and repeat the following step until $\Delta cost\ (w) = 0$ and either $right\ (w) =$ **null** or $\Delta min\ (right\ (w)) > 0$: If $right\ (w) \neq$ **null** and $\Delta min\ (right\ (w)) = 0$, replace w by $right\ (w)$; otherwise if $\Delta cost\ (w) > 0$ replace w by $left\ (w)$. (In the latter case $\Delta min\ (left\ (w)) = 0$.) Once the computation of w is completed, return $[w, \Delta min\ (p)]$.

addpathcost (p, x): Add x to $\Delta min\ (p)$.

join (p, v, q): Join the solid trees with roots p, v and q using any of the methods discussed in Chapter 4 for joining search trees.

split (v): Split the solid tree containing node v using any of the methods discussed in Chapter 4 for splitting search trees.

Both makepath and addpathcost take $O(1)$ time. The operations findpath, findtail and findpathcost are essentially the same as an access operation in a search tree (although findpath proceeds bottom-up instead of top-down), and each such operation takes time proportional to the depth of the bottommost accessed node. Both join and split are equivalent to the corresponding operations on search trees. If we use balanced binary trees to represent the solid paths, the time per path operation is $O(\log n)$, and by Theorem 5.1 a sequence of m tree operations selected from maketree, findroot, findcost, addcost, link and cut takes $O(m\ (\log n)^2)$ time. The ideas behind this result are due to Galil and Naamad [2] and Shiloach [3].

Since the *successor* field is only needed for tree roots and the *parent* field is only needed for nonroots, we can save space in the implementation of the tree operations by storing parents and successors in a single field, if we store with each vertex a bit indicating whether it is a solid tree root. Another way to think of the data structure representing a tree T is as a *virtual tree* T': T' has the same vertex set as T and has a parent function *parent'* defined as follows (see Fig. 5.5):

$$parent'(v) = \begin{cases} parent(v) & \text{if } v \text{ is not the root of a solid tree (we call } [v, parent'(v)] \text{ a } \textit{solid virtual edge}), \\ successor(v) & \text{if } v \text{ is a solid tree root (we call } [v, parent'(v)] \text{ a } \textit{dashed virtual edge}). \end{cases}$$

That is, T' consists of the solid trees representing the solid paths of T connected by edges corresponding to the dashed edges of T. To distinguish between a virtual tree T' and the tree T it represents, we shall call the latter an *actual tree.*

We can improve the $O(m(\log n)^2)$ time bound for m tree operations by a factor of $\log n$ if we use self-adjusting trees rather than balanced trees to represent the solid paths. This also has the paradoxical effect of simplifying the implementation. Let

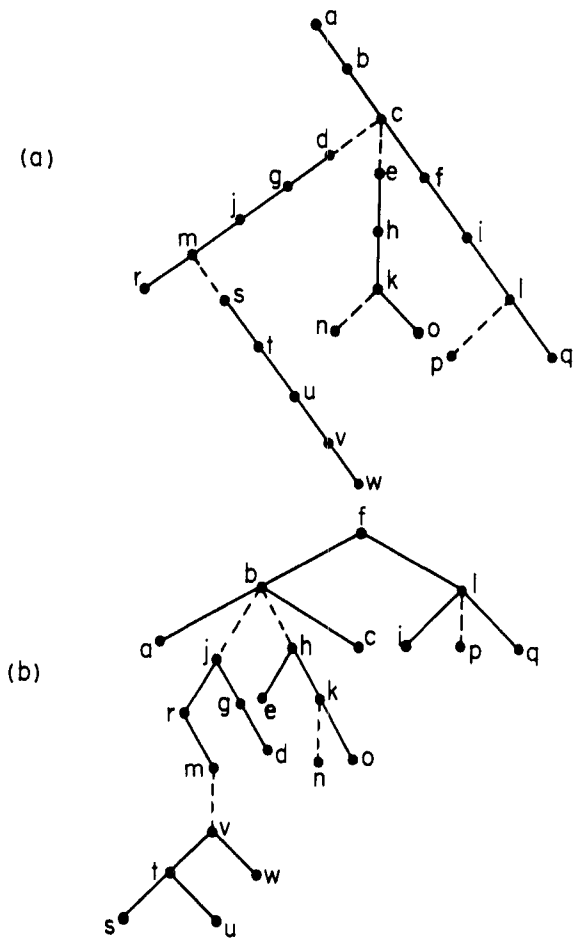

FIG. 5.5. *An actual tree and its virtual tree.* (a) *Actual tree.* (b) *Virtual tree.*

splay (x) be the operation of splaying a solid tree at a node x. Recall that splay (x) moves x to the root of the tree by making single rotations. The following programs implement the path operations:

```
procedure makepath (vertex v);
  parent (v) := left (v) := right (v) := null;
  Δcost (v) := Δmin (v) := 0
end makepath;

vertex function findpath (vertex v);
  splay (v);
  return v
end findpath;

list function findpathcost (path p);
  do right(p) ≠ null and Δmin (right(p)) = 0 → p := right(p)
  | (right (p) = null or Δmin (right (p)) > 0) and Δcost (p) > 0 → p := left (p)
  od;
  splay (p);
  return [p, Δmin (p)]
end findpathcost;

vertex function findtail (path p):
  do right (p) ≠ null → p := right (p) od;
  splay (p);
  return p
end findtail;

procedure addpathcost (path p, real x);
  Δmin (p) := Δmin (p) + x
end addpathcost;

procedure join (path p, v, q);
  assemble (p, v, q)
end join;

list function split (vertex v);
  splay (v);
  return disassemble (v)
end split;
```

To analyze the running time of this implementation, we use the analysis of splaying given in §4.3. Recall that we have defined the size of a vertex v in an actual tree T to be the number of descendants of v, including v itself. We define the *individual weight* of a vertex v as follows:

$$iw(v) = \begin{cases} size(v) & \text{if } v \text{ has no entering solid edge,} \\ size(v) - size(u) & \text{if } [u, v] \text{ is a solid edge.} \end{cases}$$

We define the total weight $tw(v)$ of a vertex v to be the sum of the individual weights of the descendants of v in the solid tree containing v. If v is the root of a solid tree, $tw(v)$ is the number of descendants of v in the virtual tree containing it. Also $1 \leq tw(v) \leq n$ for all vertices v.

Let the rank of a vertex v be $rank(v) = \lfloor \lg tw(v) \rfloor$. Suppose we use the accounting scheme of §4.3 to keep track of the time needed for operations on solid trees. Recall that we must keep $rank(v)$ credits on each vertex v and that $3(rank(u) - rank(v)) + 1$ credits suffice to maintain this invariant and to pay for a splay at a vertex v in a solid tree with root u. Since $0 \leq rank(v) \leq \lg n$ for any vertex v, this means that any solid path operation takes $O(\log n)$ credits.

The heart of the matter is the analysis of expose. We shall prove that a single expose takes $O(\log n)$ credits plus $O(1)$ credits per splice. Consider a typical splice. Let v be the current vertex, u the root of the solid tree containing v, and w the parent of u in the virtual tree. Referring to the program implementing expose, the operation findpath (v) that begins the splice takes $3(rank(u) - rank(v)) + 1$ credits and makes v the root of its solid tree. After this the operations split (v) and join (p, v, r) take $O(1)$ time and do not change $tw(v)$, since $tw(v)$ is the number of descendants of v in the virtual tree, and the effect of the join and split on the virtual tree is merely to change from dashed to solid or vice versa at most two virtual edges entering v. (See Fig. 5.6.) Thus one additional credit pays for the split, the join and

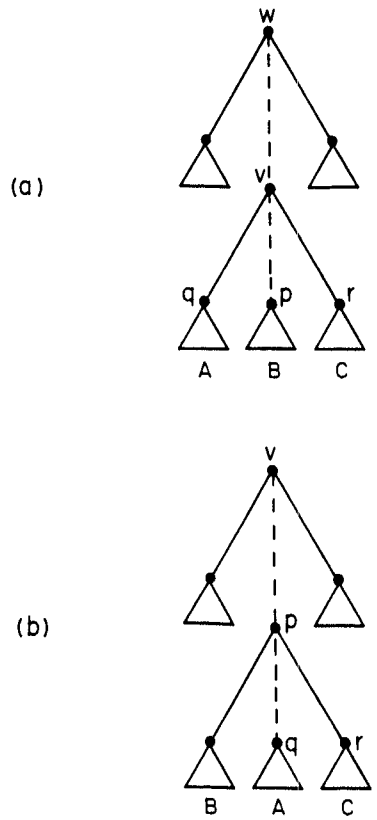

FIG. 5.6. *Effect of a splice on a virtual tree.* (a) *Relevant part of virtual tree after operation* findpath (v). *Triangles denote solid subtrees. Vertex* v *is now the root of its solid tree.* (b) *After completing the splice.*

the rest of the splice. No ancestor of w in the virtual tree has its individual or total weight affected by the splice; thus no credits are needed to maintain the credit invariant on other solid trees. Vertex w, which is the next current vertex, has rank at least as great as that of u.

Summing over all splices during an expose, we find the total number of credits needed is 3 (*rank* (u) − *rank* (v)) plus two per splice, where v is the vertex exposed and u is the root of the virtual tree containing v. This bound is $O(\log n)$ plus two per splice. By Theorem 5.1 the total number of credits for all m exposes caused by m tree operations is $O(m \log n)$.

We must also account for the fact that link and cut change the total weights of vertices and thus affect the credit invariant. Consider an operation link (v, w). This operation joins the empty path, the one-vertex path formed by exposing v, and the path formed by exposing w into a single solid path. The only effect on total weight is to increase the total weight of v, requiring that we place $O(\log n)$ additional credits on v. Consider the operation cut (v). After the operation expose (v), the remaining effect of the cut is to delete the solid edge leaving v, which changes no individual weights and only decreases the total weights of some of the ancestors of v in the actual tree originally containing it. This only releases credits.

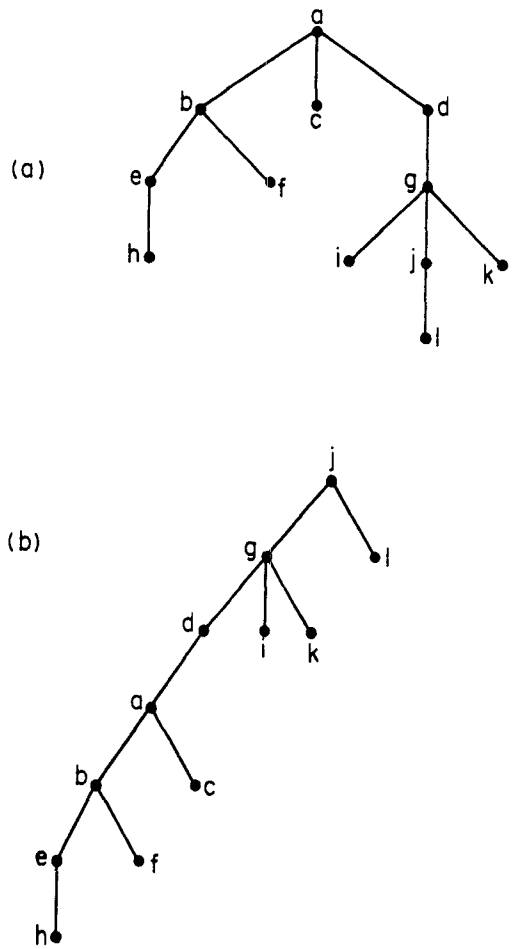

FIG. 5.7. *Everting a tree.* (a) *A rooted tree.* (b) *After the operation evert* (j).

Combining these estimates we have the following theorem:

THEOREM 5.2. *If we use self-adjusting binary trees to represent solid paths, a sequence of m tree operations including n maketree operations requires* $O(m \log n)$ time.

5.4. Remarks. We can add other operations to our repertoire of tree operations without sacrificing the bound of $O(\log n)$ amortized time per operation. Perhaps the most interesting is evert (v), which turns the tree containing vertex v "inside out" by making v the root. (See Fig. 5.7.) When this operation is included the data structure is powerful enough to handle problems requiring linking and cutting of free (unrooted) trees; we root each tree at an arbitrary vertex and reroot as necessary using evert. We can associate costs with the edges of the trees rather than with the vertices. If worst-case rather than amortized running time is important, we can modify the data structure so that each tree operation takes $O(\log n)$ time in the worst case; the resulting structure uses biased search trees [1] in place of self-adjusting trees and is more complicated than the one presented here. Details of all these results may be found in Sleator and Tarjan [4].

References

[1] S. W. BENT, D. D. SLEATOR AND R. E. TARJAN, *Biased search trees*, SIAM J. Comput., submitted.

[2] Z. GALIL AND A. NAAMAD, *An* $O(E V \log^2 V)$ *algorithm for the maximal flow problem*, J. Comput. System Sci., 21 (1980), pp. 203–217.

[3] Y. SHILOACH, *An* $O(n \cdot I \log^2 I)$ *maximum-flow algorithm*, Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford Univ., Stanford, CA, 1978.

[4] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), to appear; also in Proc. 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 114–122.