CS 573: Topics in Algorithms: Advanced Data Structures **Spring 2006**

Lecture 8 — February 9

Lecturer: Jeff Erickson *Scribe: Joseph Elble*

Review

Last time we investigated Sleator and Tarjan's link-cut trees (referred to in [5] as ST-Trees). The link-cut tree is based on path decompositions. Every node in the represented tree T has no more than one preferred child. Preferred paths are represented as splay trees. The root of each path tree points to its parent in T. We saw that the operations CUT, JOIN, and FINDROOT are all implemented via a function we called ACCESS. The function ACCESS(v) made the path from v to the root a preferred path with the amortized cost of $O(\log n)$.

8.1 Path-Cost Queries via Lazy Propagation

In this section, the functions FINDCOST(v) and ADDCOST(v, Δ) will be introduced. To support these functions and maintain $O(\log n)$ amortized time, it will be necessary to maintain three different values at every node in the path tree.

Among other graph-theoretic problems, link-cut trees allowed Sleator and Tarjan to develop a new fast algorithm for network flows including max-flow problems [4]. In order to attain this application, the authors implemented the following operations:

- FINDCOST (v): This function returns the minimum value among ancestors of v (including v itself).
- ADDCOST (v, Δ): This function adds Δ to the value of every ancestor of v (again, including v itself).

The three values stored at each node to support the above operations include:

- value(v): stores what node v thinks its value is.
- min(v): stores the minimum value of the descendants of v (including v itself).
- shift(v): stores the value shift for all descendants of v.

The *real* value of v is given by

$$\left(\sum_{u \text{ ancestor of } v} shift(u) \right) + value(v) \ .$$

Similarly, the *real* minimum value in v's subtree is given by

$$\text{realmin}\ (v) = \min \begin{cases} \text{realvalue}(v) \\ \text{realmin}(\text{left}(v)) \\ \text{realmin}(\text{right}(v)) \end{cases} .$$

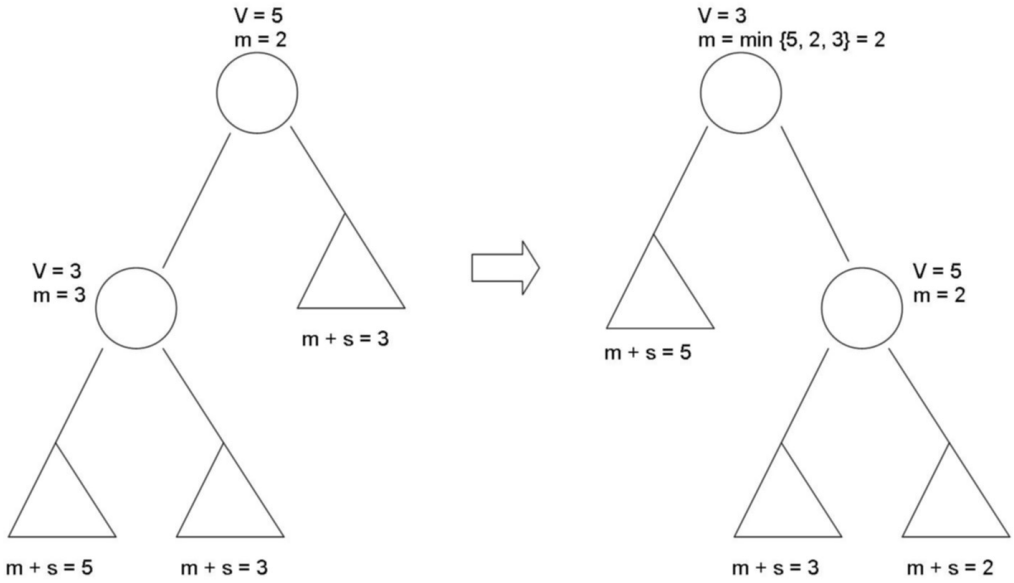
We maintain the following invariant at all times (this function is recursive)

$$\min\ (v) = \min \begin{cases} \text{value}(v) \\ \min(\text{left}(v)) + \text{shift}(\text{left}(v)) \\ \min(\text{right}(v)) + \text{shift}(\text{right}(v)) \end{cases} .$$

It is important to note that if one were to perform the AddCost function without the shift(v) value, then one would have to visit all the descendants of v and add the shift value at every iteration. The trick involved with this data structure is to maintain the value and shift variables correctly as the tree is rotated.

We will now see the Force function [Alg. 1]. This function is responsible for the invaluable task of maintaining value and shift during a rotation (see figure 8.1). It is trivial to maintain min during rotations, splits and joins.

Figure 8.1. A rotation (notice the recalculation of min)



Now if one were to perform a Cut operation, one would force the two nodes to be cut. Similarly, if one were to perform a Link operation, one would force the two nodes to be linked. So on every access, we wish to force the node v. The path operations, FindCost and AddCost and are now easy [Alg. 2 and Alg. 3]. Both algorithms take $O(\log\ n)$ amortized time.

Algorithm 1 FORCE(v)

{ Do whatever local calculations that you need to do in order to make shift(v) = 0 }
value(v) ← value(v) + shift(v)
min(v) ← min(v) + shift(v)
shift(left(v)) ← shift(left(v)) + shift(v)
shift(right(v)) ← shift(right(v)) + shift(v)
shift(v) ← 0

Algorithm 2 FINDCOST(v)

ACCESS (v) { forces v }
return min(v)

8.2 Euler-Tour Trees

We saw that link-cut trees are good for maintaining aggregates on paths of a tree. In particular, this information is useful in the course of solving network flow problems. In an Euler-tour tree, Henzinger and King sought to keep aggregate information on subtrees [3].

Euler-tour trees are an alternative dynamic tree representation that supports easy subtree queries. The Euler-tour tree data structure was introduced by Henzinger and King in [3]. An Euler-tour tree data structure maintains a path along the tree that begins at the root and ends at the root, traversing each edge twice.

The Euler-tour of a tree, T, is maintain in a balanced binary search tree with one node for each time a node in T was visited. This balanced binary tree (eg. splay tree) must support splits and joins. Each node in T holds pointers to its first and last occurances in its Euler-tour tree. The Euler-tour tree algorithm [Alg. 4] for an n node tree has length $2n-2$ (two outputs per edge). A degree d node is output d times, except the root which is output $d+1$ times.

Our Euler-tour tree will implement MAKETREE(x), FINDROOT(v), CUT(v) [Alg. 5], JOIN(v,u) [Alg. 6], FINDCOST(v), and ADDCOST(v, Δ). CUT, JOIN, and FINDROOT each take O(log n) amortized time. It is possible to speed up FINDROOT to $O(1)$ worst-case by connecting the root to a splay tree (see figure 8.3).

- MAKETREE(x): return new node with value x
- FINDROOT(v): return left-most node in v's Euler-tour tree.
- FINDCOST(v): return minimum value among v's descendants
- ADDCOST(v, Δ): add Δ to the values of v's descendants

We can use the same lazy propagation trick to achieve O(log n) amortized time. Any subtree of T is an interval in the Euler-tour tree of T. By splaying, it is possible to isolate any interval as a subtree of the Euler-tour tree of T. Both link-cut and Euler-tour trees support the EVERT operation.

Figure 8.2. An Euler-Tour Tree Example

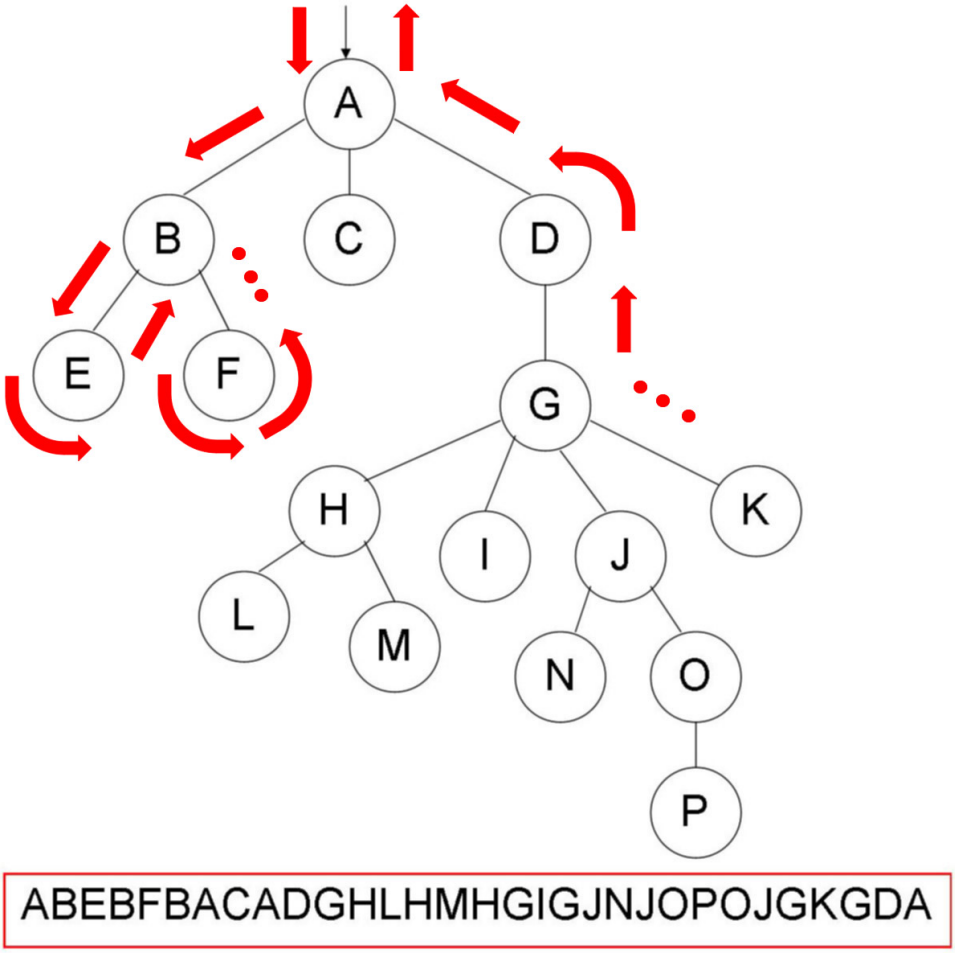
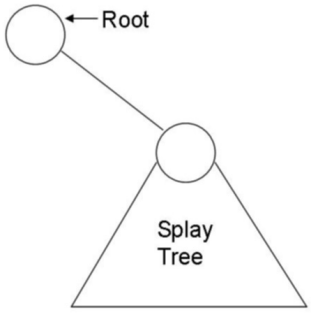


Figure 8.3. FindRoot in $O(1)$



Algorithm 3 AddCost(v)

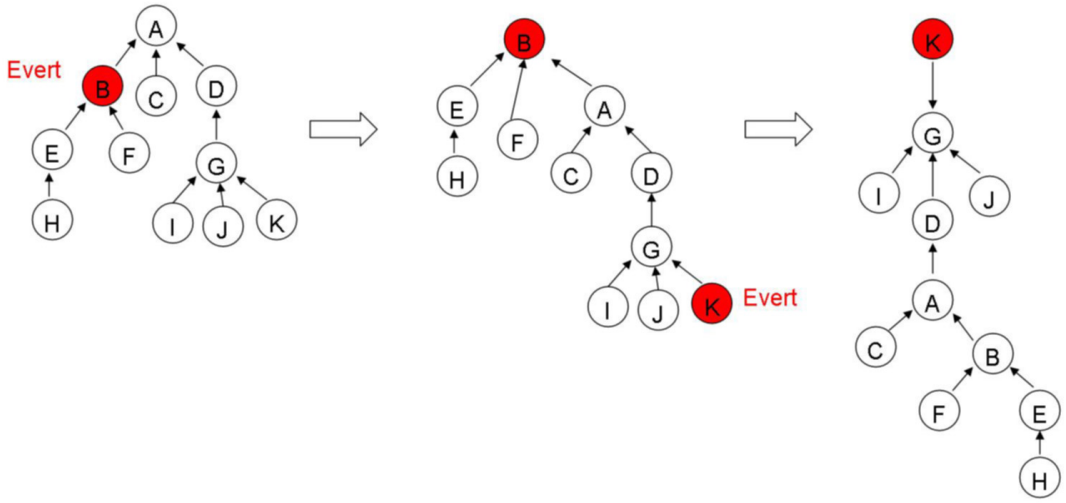
```
Access (v) { forces v }
shift(v) ← Δ
```

Algorithm 4 EulerTour(v)

```
output v
for each child w of v do
  EulerTour(w)
  output v
end for
```

- Evert(v): Make v the root of the tree. The underlying free tree is unchanged, but all directed paths now lead to v (see figure 8.4).

Figure 8.4. Evert operation



If you Cut(v) and Join(v) for some node v, then you may **not** end up with the same Euler-tour before and after the operations. Link-cut and Euler-tour trees give rise to the question: can one perform both path and subtree queries with the same data structure while maintaining $O(\log n)$ amortized time? We have seen link-cut trees provide $O(\log n)$ amortized time for path queries, and we have seen Euler-tour trees provide $O(\log n)$ amortized time for subtree queries. The remaining sections of these notes are a mere overview of dynamic trees.

Algorithm 5 CUT(v)

Find the first an last occurances of v in its Euler-tour tree.
{ $O(1)$ since this information is maintained }
A u v B v u C $\leftarrow$ EulerTour(v)
where $v \notin$ A, C and $u =$ parent(v)
return A u C, v B v

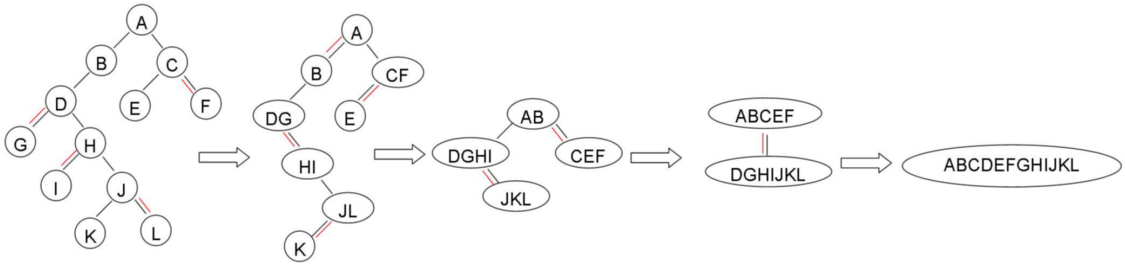
Algorithm 6 JOIN(v, u)

A u B $\leftarrow$ EULERTOUR(u)
v C $v \leftarrow$ EULERTOUR(v)
return A u v C v u B

8.3 Topology Trees

Topology trees were introduced by Greg Frederickson in [2], and were developed for maintaing binary trees dynamically. Topology trees can be used to implement link-cut trees. These trees are based on what the author refers to as a *vertex cluster*. A vertex cluster with respect to an undirected tree, T, is a set of vertices such that the subgraph of T induced on the cluster is connected. Every node in the topology tree points to its parent and kids in its own level. The topology tree shown in the figure below (figure 8.5) is contracting its independent sets of edges to leaves or between degree-2 vertices.

Figure 8.5. Topology Tree



8.4 Top Trees

Top trees were created by Alstrup, Holm, de Lichtenberg, and Thorup in 2005 [1]. They are based on edge contraction. Every cluster has one or two exposed vertices, and again there is a hierarchical clustering. The top-level cluster looks like a path, in that it has two exposed ends. Neighboring clusters share vertices. One might think of this as vertex contraction.

8.5 Rake-Compress Trees

The modern rake-compress tree was introduced by Tarjan and Werneck in [5]. There are two types of nodes: rake and compress nodes. The leaves in a rake-compress tree are the edges in the represented tree T. There is a one-to-one correspondence between compress nodes and nodes in T. However, this one-to-one correspondence does not hold for rake nodes. The root of a rake-compress tree has the degree of one. In the rake-compress tree presented in [5], any vertex with degree less than or equal to three requires a compress operation (see figure 8.7). A rake operation is performed to merge two neighboring leaves and is generally performed when there are more than 3 leaves from a given vertex (see figure 8.6).

To make the recursion simpler, we assume that the root of the represented tree T has degree 1. (For undirected trees, we choose an arbitrary leaf to be the root; for directed trees, we may need to add an extra node above the real root.) Each non-leaf node in T has exactly one *preferred child*. The choices of preferred children partition T into several *preferred paths*. The preferred path containing the root is called the *main path*.

The rake-compress tree for T is constructed recursively as follows. We first recursively construct rake-compress trees for every subtree hanging off the main path of T. (For each subtree, its parent p on the main path is considered the root of the subtree for purposes of recursive construction. All other children of p are ignored, so that the root of the subtree has degree 1, as required.) The recursive construction replaces each subtree with a single edge, so that the tree becomes a caterpillar.

If any node on the main path has more than one leaf off the main path, we merge these leaves together in pairs. (See Figure 8.6.) Each pairwise merge is recorded in a *rake node*, whose two children correspond to the two edges being merged. Thus, the leaves hanging off any node on the main path are clustered into a binary *rake tree*. (The operation is called 'rake' because it removes about half of the leaves.)

Finally, we repeatedly compress the main path, by replacing interior path vertices with edges, until T consists of a single edge. (See Figure 8.7.) Each vertex removal is recorded in a *compress node*, which has either two or three children, corresponding to the edges adjacent to the vertex being removed. In particular, the edges on the main path are clustered into a binary *compress tree*, each of whose nodes may also be the parent of a rake tree.

The final rake-compress tree for T has one leaf for each edge in T and one compress node for each internal node in T. Rake nodes do not correspond directly to either nodes or edges in T, and leaves in T are not directly represented in the rake-compress tree. Figure 8.9 shows a rake-compress tree that represents the tree in Figure 8.8. Square nodes are leaves; circular nodes with labels are compress nodes; doubled circular nodes are rake nodes. Compress trees are connected by solid edges; rake trees are connected by doubled edges; and dotted edges join the root of each rake tree to its parent in a compress tree.

The leaves of rake trees are recursive rake-compress trees. Compress nodes are associated with a deleted vertex.

Compress trees are effectively equivalent to path trees in link-cut trees. The role of the rake tree is to organize the path parent pointers in a link-cut tree. The final rake tree (see figure 8.8 and 8.9) has degree less than or equal to three. The operations LINK, FINDROOT,

Figure 8.6. RAKE operation

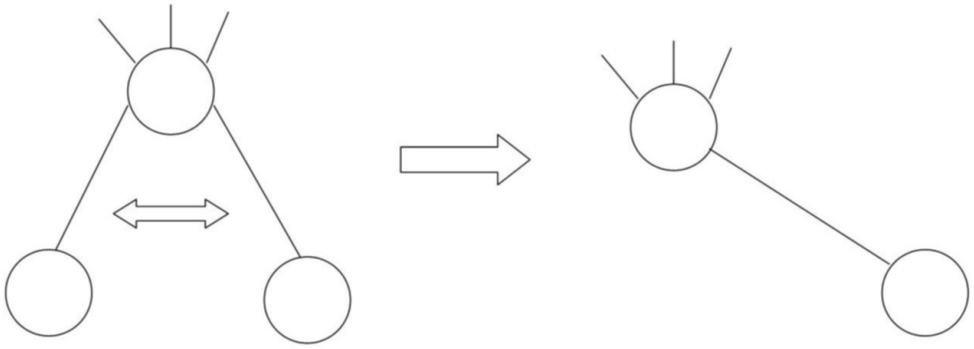
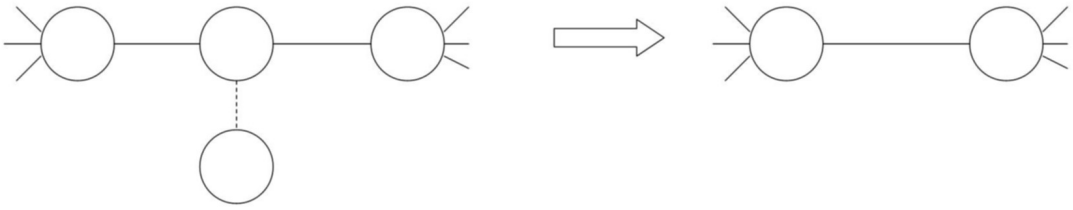


Figure 8.7. COMPRESS operation



CUT, EVERT, path queries, and subtree queries can all be performed in $O(\log n)$ amortized time.

Figure 8.8. Rake-Compress Tree [Part 1]

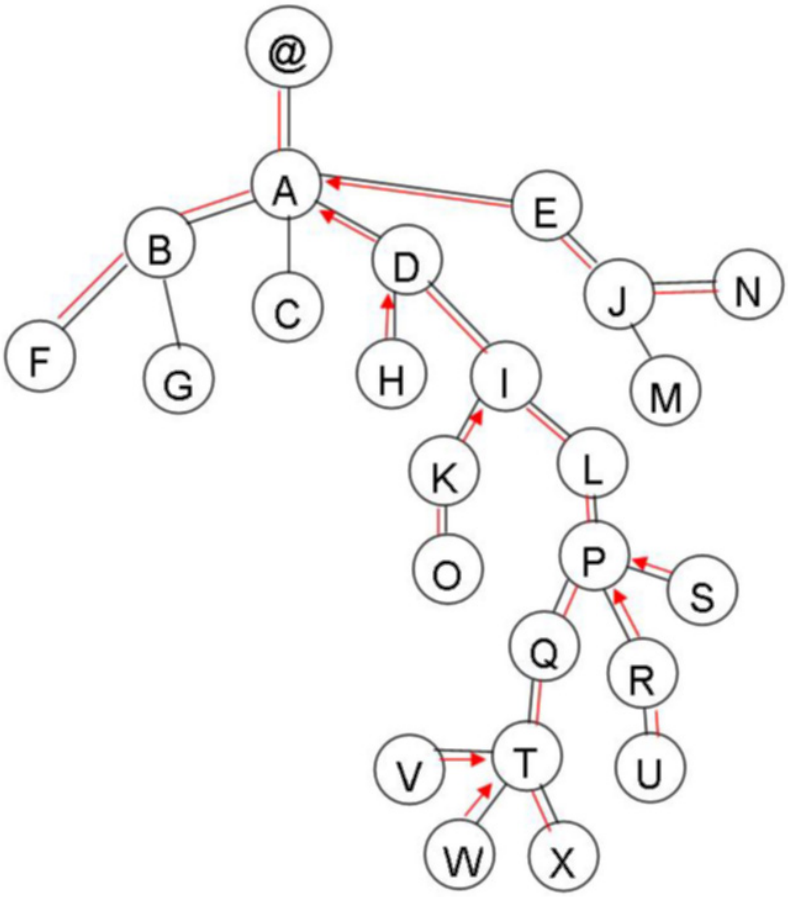
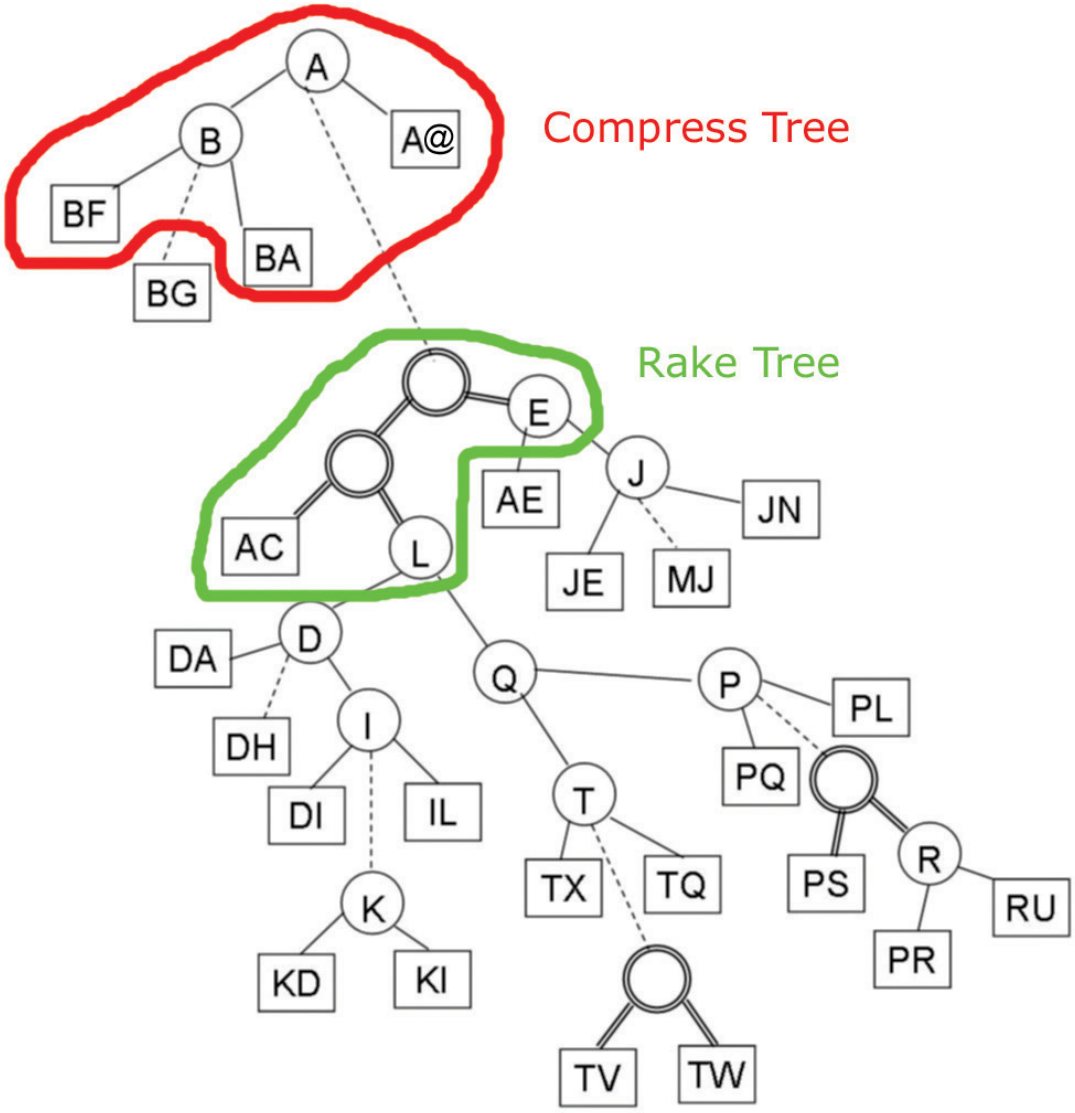


Figure 8.9. Rake-Compress Tree [Part 2]



Bibliography

[1] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1:243–264, 2005.

[2] Greg Frederickson. A data structure for dynamically maintaining rooted trees. *J. Algorithms*, 24:37–65, 1997.

[3] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46:502–516, 1999.

[4] Daniel D. Sleator and Robert E. Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and Systems Sciences*, 26:362–391, 1983.

[5] Robert E. Tarjan and Renato Werneck. Self-adjusting top trees. *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 813–822, 2005.