

Lecture 14

Lecturer: Michel X. Goemans

Scribe: Sanmay Das

In the last lecture the concept of the dynamic trees data structure was introduced and a number of operations that dynamic trees must support were described. In this lecture, we define the data structure in detail and describe the efficient implementation of operations using *expose*, an extended splay operation.

1 Dynamic Trees

Dynamic trees are a data structure intended for maintaining a representation of a set of rooted trees, and performing a number of operations (discussed below) on these trees. In today's lecture we will be using the example tree depicted in figure 1 to describe dynamic trees and the *expose* operation which is central to the efficient implementation of operations on dynamic trees.

We view rooted trees as unions of node-disjoint paths. This divides the edges of the tree into two sets. *Solid edges* are those that are on the node-disjoint paths that the tree is composed of, and *dashed edges* are those that are not on these paths. Figure 2 shows a possible partitioning of the example tree into a set of node disjoint paths. Note that each path consisting of solid edges is a directed path from top to bottom. We refer to the top of each such path as the tail, and the bottom as the head (for example h in the path from h to e is the tail and e the head).

2 Virtual Trees

The union of disjoint paths described above can be used to represent *virtual trees*. In a virtual tree, each solid path is represented by a binary search tree such that the following two conditions hold:

- 1 A successor node in the binary search tree is a parent in the rooted tree.
- 2 The root of a binary search tree is linked to the parent of the tail of the path it corresponds to in the rooted tree.

For example, figure 3 shows the virtual tree corresponding to the union of disjoint paths shown in figure 2.

There are three kinds of edges in a virtual tree, corresponding to the three types of children a node can have. Left and right children of a node are connected to the node by solid edges, and middle children of a node are connected to it by dashed edges. Thus a node can either have no parent, a parent connected to it by a solid line, or a parent connected to it by a dashed line. Note that there can be many virtual trees corresponding to a rooted tree, because there are two different degrees of freedom involved in constructing a virtual tree — the union of disjoint paths could be different, as could the structure of the binary search trees corresponding to the paths.

3 The *expose* Operation

The *expose*(v) operation is an extended splay operation on virtual trees. The important parts of this operation are to make sure that the path from v to the root is solid and that the binary search tree representing the path to which v belongs is rooted at v . We will describe the process in 3 steps although it can be implemented somewhat more efficiently in a single step (although the asymptotic efficiency is the same).

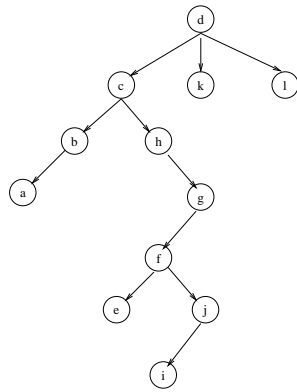


Figure 1: Example tree

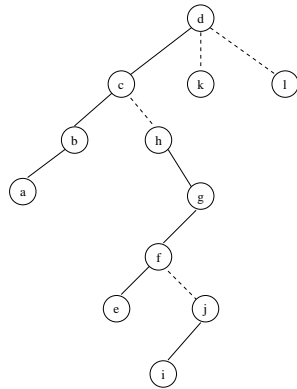


Figure 2: Tree as a union of disjoint paths

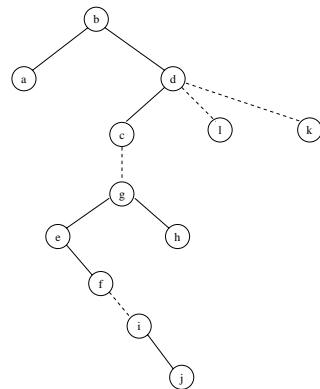


Figure 3: A virtual tree corresponding to the example

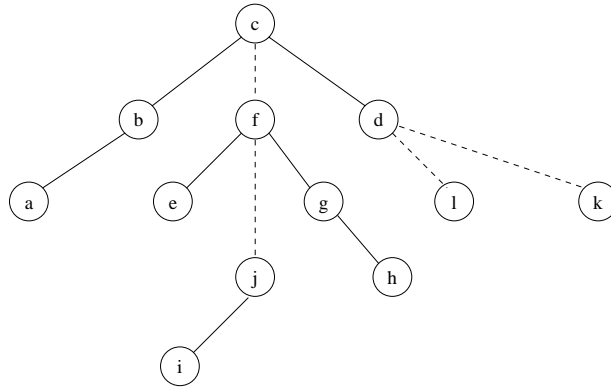


Figure 4: After step 1 of $\text{expose}(j)$

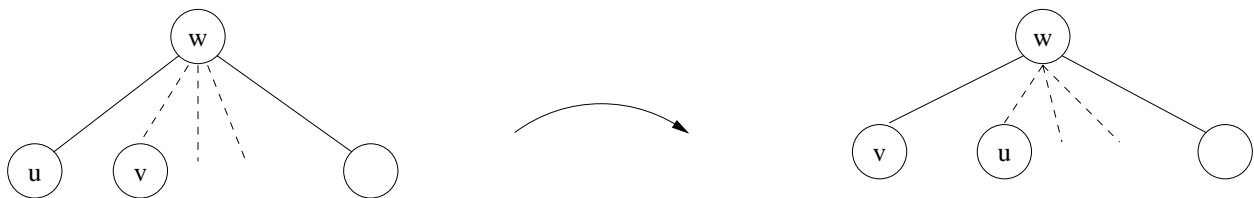


Figure 5: Splicing operation

3.1 Step 1

Step 1 consists of walking from v to the root of the virtual tree. Whenever the walk enters a binary search tree (solid edges) at some node w , a *splay*(w) operation is performed, bringing w to the root of that tree. Middle children are not affected in this step. For example in the $\text{expose}(j)$ operation on the example tree, we perform 3 splay operations, on j , f and c respectively, leading to the tree shown in figure 4. Note that at the end of step 1 of an $\text{expose}(v)$ operation, v will be connected to the root of the virtual tree only by dashed edges.

3.2 Step 2: Splicing

Step 2 consists of walking from v to the root of the virtual tree while exchanging the subtree rooted at v with the left subtree of the parent of v at each step, as illustrated in figure 5 (why the solid tree remains a binary search tree is left as an exercise). The new tree of the example for $\text{expose}(j)$ after step 2 is performed can be seen in figure 6. Note that at the end of this step, there will be a solid path from the root of the tree to the node being exposed.

3.3 Step 3

Step 3 consists of walking from v to the root in the virtual tree, splaying v to the root. The example tree after the completion of step 3 (and the entire expose operation is shown in figure 7. Note that in the analysis, we can charge the entire cost of step 2 to the final splaying operation in step 3.

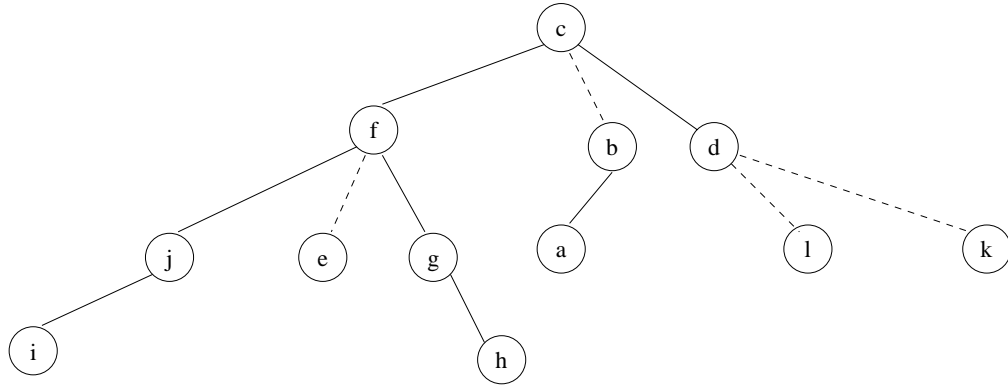


Figure 6: After step 2 (splicing) of $\text{expose}(j)$

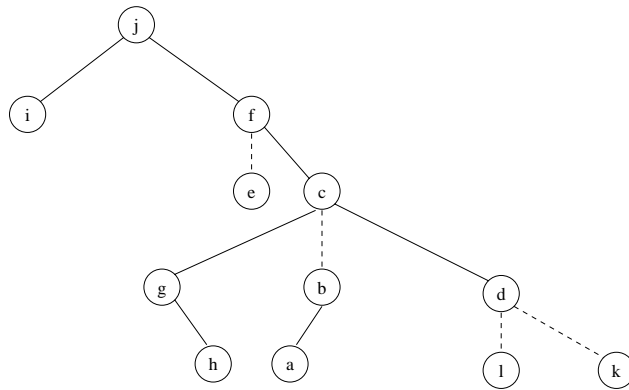


Figure 7: After step 3 of $\text{expose}(j)$

4 Operations on Dynamic Trees

4.1 Maintaining Cost Information

Some of the operations we need to perform on dynamic trees involve costs of edges and paths. At each node v , we could maintain the minimum cost in the subtree rooted at v , ignoring the virtual tree and dashed edges, but this is problematic for the *add - cost* operation. So instead we maintain two things at each node x , namely

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

where $mincost(x)$ represents the smallest cost along the path corresponding to the BST in the original tree, and:

$$\Delta cost(x) = \begin{cases} cost(x) & \text{if } x \text{ is the root of a BST,} \\ cost(x) - cost(p(x)) & \text{otherwise } (p(x) \text{ denotes the parent of } x). \end{cases}$$

Therefore, if x is the root of a BST, then $cost(x) = \Delta cost(x)$ and $mincost(x) = \Delta cost(x) - \Delta min(x)$.

$\Delta min(x)$ and $\Delta cost(x)$ can both be updated in $O(1)$ time when one does a rotation or a splice. It is important that both v and w are roots of BSTs in the splicing step.

4.2 Implementation of Operations

find - cost(v): First, *expose(v)*. Now v is the root, so return $\Delta cost(v)$.

find - root(v): *expose(v)*. Then follow right children until you reach a leaf w of the BST. *splay(w)*, then return w .

find - min(v): *expose(v)*. Examining Δmin and $\Delta cost$ go down to the minimum, say w , ignoring the left subtree of v . *splay(w)*

add - cost(v, x): *expose(v)*. Add x to $\Delta cost(v)$ and subtract x from $\Delta cost(left(v))$.

cut(v): *expose(v)* Add $\Delta cost(v)$ to $\Delta cost(right(v))$. Remove the edge $(v, right(v))$.

link(v, w, x): *expose(v)*, then *expose(w)*. Set w to be a middle child of v .