

Lecture 19 — May 1, 2012

Prof. Erik Demaine Scribes: Justin Holmgren (2012), Jing Jian (2012), Maksim Stepanenko (2012)

Mashhood Ishaque (2007)

1 Overview

In this lecture we discuss dynamic trees that have many applications such as Network Flow and Dynamic Connectivity problems in addition to them being interesting theoretically. This is the beginning of our discussion of dynamic graphs, where in general we have an undirected graph and we want to support deletions and insertions. We will discuss one data structure called Link-Cut trees that achieves logarithmic amortized time for all operations.

2 Link-cut Trees

Using link-cut trees we want to maintain a forest of rooted trees whose each node has an arbitrary number of unordered child nodes. The data structure has to support the following operations in $O(\lg n)$ amortized time:

- *make_tree()* – Returns a new vertex in a singleton tree. This operation allows us to add elements and later manipulate them.
- *link(v,w)* – Makes vertex v a new child of vertex w , i.e. adds an edge (v,w) . In order for the representation to remain valid this operation assumes that v is the root of its tree and that v and w are nodes of distinct trees.
- *cut(v)* – Deletes the edge between vertex v and its parent, $parent(v)$ where v is not the root.
- *find_root(v)* – Returns the root of the tree that vertex v is a node of. This operation is interesting because path to root can be very long. The operation can be used to determine if two nodes u and v are connected.
- *path_aggregate(v)* – Returns an aggregate, such as max/min/sum, of the weights of the edges on the path from the root of the tree to node v . It is also possible to augment the data structure to return many kinds of statistics about the path.

Link-Cut Trees were developed by Sleator and Tarjan [1] [2]. They achieve logarithmic amortized cost per operation for all operations. Link-Cut Trees are similar to Tango trees in that they use the notions of preferred child and preferred path. They also use splay trees for the internal representation.

3 Preferred-path decomposition

Link-cut trees are all about paths in the tree, so we want to split a tree into paths. We need to define some terms first.

3.1 Definition of Link-Cut Trees

We say a vertex has been *accessed* if it was passed to any of the operations from above as an argument. We call the abstract trees that the data structure represents **represented trees**. We are not allowed to change the represented tree and it can be unbalanced. The represented tree is split into paths (in the data structure representation).

The **preferred child** of node v is equal to its i -th child if the last access within v 's subtree was in the i -th subtree and it is equal to null if the last access within v 's subtree was to v itself or if there were no accesses to v 's subtree at all. A **preferred edge** is an edge between a preferred child and its parent. A **preferred path** is a maximal continuous path of preferred edges in a tree, or a single node if there is no preferred edges incident on it. Thus preferred paths partition the nodes of the represented tree.

Link-Cut Trees represent each tree T in the forest as a tree of **auxiliary trees**, one auxiliary tree for each preferred path in T . Auxiliary trees are splay trees with each node keyed by its depth in its represented tree. Thus for each node v in its auxiliary tree all the elements in its left subtree are higher (closer to the root) than v in v 's represented tree and all the elements in its right subtree are lower. Auxiliary trees are joined together using **path-parent pointers**. There is one path-parent pointer per auxiliary tree and it is stored in the root of the auxiliary tree. It points to the node that is the parent (in the represented tree) of the topmost node in the preferred path associated with the auxiliary tree. We cannot store path-to-child pointers because there can be many children. Including auxiliary trees with the path-parent pointers as edges, we have a representation of a represented tree as a tree of auxiliary trees which potentially can have a very high degree.

3.2 Operations on Link-Cut Trees

3.2.1 Access

All operations above are implemented using an $access(v)$ subroutine. It restructures the tree T of auxiliary trees that contains vertex v so that it looks like v was just accessed in its represented tree R . When we access a vertex v some of the preferred paths change. A preferred path from the root of R down to v is formed. When this preferred path is formed every edge on the path becomes preferred and all the old preferred edges in R that had an endpoint on this path are destroyed, and replaced by path-parent pointers.

Remember that the nodes in auxiliary trees are keyed by their depth in R . Thus nodes to the left of v are higher than v and nodes to the right are lower. Since we access v , its preferred child becomes *null*. Thus, if before the access v was in the middle of a preferred path, after the access the lower part of this path becomes a separate path. What does it mean for v 's auxiliary tree? This means that we have to separate all the nodes less than v in a separate auxiliary tree. The easiest way to do this is to splay on v , i.e. bring it to the root and then disconnect its right subtree, making it a

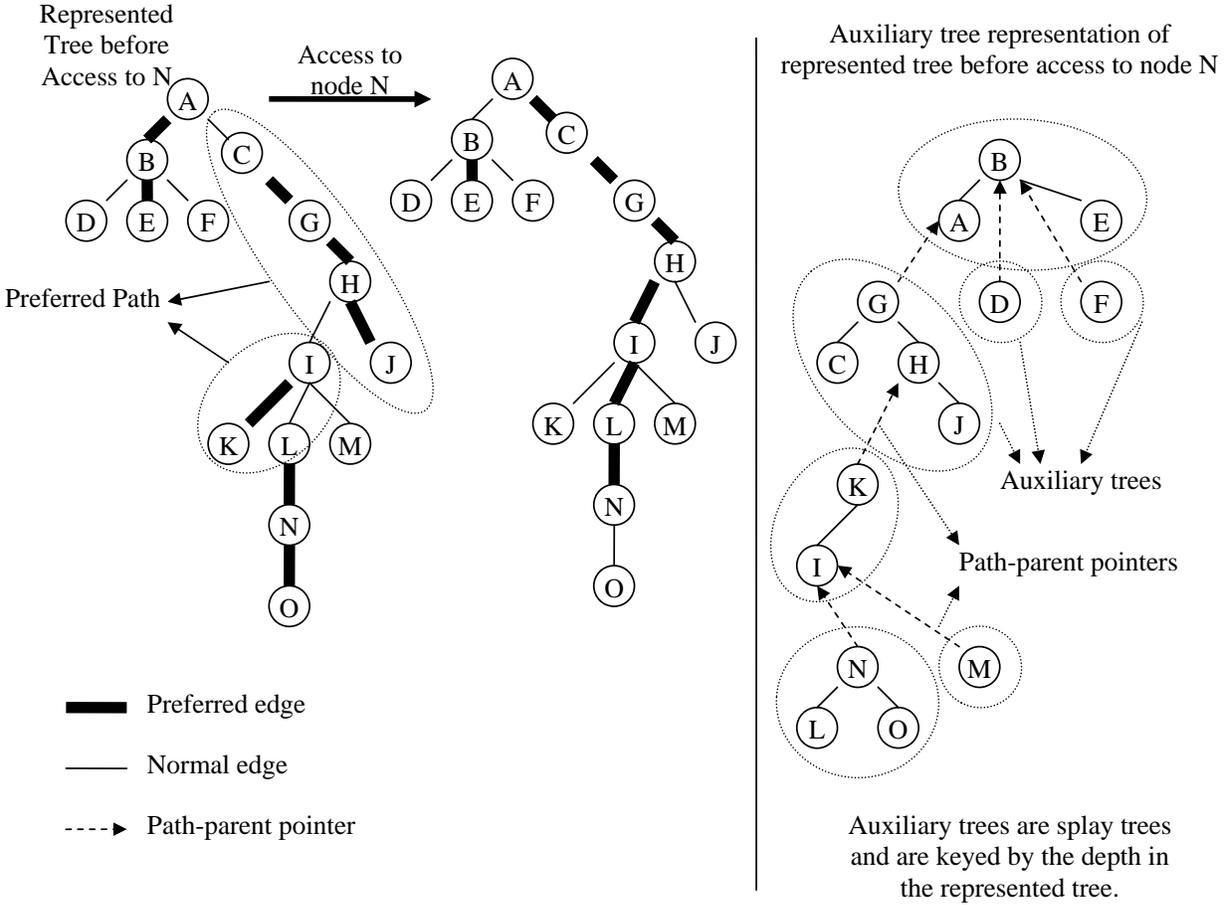


Figure 1: Structure of Link-Cut Trees and Illustration of Access Operation

separate auxiliary tree.

After dealing with v 's descendants, we have to make a preferred path from v up to the root of R . This is where path-parent pointer will be useful in guiding us up from one auxiliary tree to another. After splaying, v is the root and hence has a path-parent pointer (unless it is the root of T) to its parent in R , call it w . We need to connect v 's preferred path with the w 's preferred path, making w the real parent of v , not just path-parent. In other words, we need to set w 's preferred child to v . This is a two stage process in the auxiliary tree world. First, we have to disconnect the lower part of w 's preferred path the same way we did for v (splay on w and disconnect its right subtree). Second we have to connect v 's auxiliary tree to w 's. Since all nodes in v 's auxiliary tree are lower than any node in w 's, all we have to do is to make v auxiliary tree the right subtree of w . To do this, we simply disconnect w with its right child, and change the pointer to point to v . Finally, we have to do minor housekeeping to finish one iteration: we do a second splay of v . Since v is a child of the root w , splaying simply means rotating v to the root. Another thing to note is that we keep the path-parent pointer at all stages of concatenation.

We continue building up the preferred path in the same way, until we reach the root of R . v will

ACCESS(v)

- Splay v within its auxiliary tree, i.e. bring it to the root. The left subtree will contain all the elements higher than v and right subtree will contain all the elements lower than v
- Remove v 's preferred child.
 - $\text{path-parent}(\text{right}(v)) \leftarrow v$
 - $\text{right}(v) \leftarrow \text{null}$ (+ symmetric setting of parent pointer)
- loop until we reach the root
 - $w \leftarrow \text{path-parent}(v)$
 - splay w
 - switch w 's preferred child
 - $\text{path-parent}(\text{right}(w)) \leftarrow w$
 - $\text{right}(w) \leftarrow v$ (+ symmetric setting of parent pointer)
 - $\text{path-parent}(v) \leftarrow \text{null}$
 - $v \leftarrow w$
- splay v just for convenience

have no right child in the tree of auxiliary trees. The number of times we repeat is equal to the number of preferred-child changes.

3.2.2 Find Root

FIND_ROOT operation is very simple to implement after we know how to handle accesses. First, to find the root of v 's represented tree, we access v thus make it on the same auxiliary tree as the root of the represented tree. Since the root of the represented tree is the highest node, its key in the auxiliary tree is the lowest. Therefore, we go left from v as much as we can. When we stop, we have found the root. We splay on it and return it. The reason we need to splay is the root might be linearly deep in the tree; by splaying it to the top, we ensure that it is fast to find root next time.

FIND_ROOT(v)

- access(v)
- Set v to the smallest element in the auxiliary tree, i.e. to the root of the represented tree
 - $v \leftarrow \text{left}(v)$ until $\text{left}(v)$ is *null*
- access r
- return r

3.3 Path Aggregate

Like all of our operations, the first thing we do is $\text{access}(v)$. This is an aux. tree representing the path down to v . We augment the aux. tree with the values you care about, such as “sum” or “min” or “max.” We will see that it is easy to maintain such augmentations when we look at cut and link.

$\text{PATH-AGGREGATE}(v)$

- $\text{access}(v)$
- return v .subtree-sum (augmentation within each aux. tree)

3.3.1 Cut

To cut $(v, \text{parent}(v))$ edge in the represented tree means that we have to separate nodes in v 's subtree (in represented tree) from the tree of auxiliary trees into a separate tree of auxiliary trees. To do this we access v first, since it gathers all the nodes higher than v in v 's left subtree. Then, all we need to do is to disconnect v 's left subtree (in auxiliary tree) from v . Note that v becomes in an auxiliary tree all by itself, but path-parent pointer from v 's children (in represented tree) still point to v and hence we have the tree of auxiliary trees with the elements we wanted. Therefore there are two trees of aux trees after the cut.

$\text{CUT}(v)$

- $\text{access}(v)$
- $\text{left}(v) \leftarrow \text{null}$ (+ symmetric setting of parent pointer)

3.3.2 Link

Linking two represented trees is also easy. All we need to do is to access both v and w so that they are at the roots of their trees of auxiliary trees, and make latter left child of the former.

$\text{LINK}(v, w)$

- $\text{access}(v)$
- $\text{access}(w)$
- $\text{left}(v) \leftarrow w$ (+ symmetric setting of parent pointer)

4 Analysis

As one can see from the pseudo code, all operations are doing at most logarithmic work (amortized, because of the splay call in find_root) plus an access. Thus it is enough to bound the run time of access. First we show an $O(\lg^2 n)$ bound.

4.1 An $O(\lg^2 n)$ bound.

From `access`'s pseudo code we see that its cost is the number of iterations of the loop times the cost of splaying. We already know from previous lectures that the cost of splaying is $O(\lg n)$ amortized (splaying works even with splits and concatenations). Recall that the loop in `access` has $O(\# \text{ preferred child changes})$ iterations. Thus to prove the $O(\lg^2 n)$ bound we need to show that the number of preferred child changes is $O(\lg n)$ amortized. In other words the total number of preferred child changes is $O(m \lg n)$ for a sequence of m operations. We show this by using the Heavy-Light Decomposition of the represented tree.

4.1.1 The Heavy-Light Decomposition

The Heavy-Light decomposition is a general technique that works for any tree (not necessarily binary). It calls each edge either heavy or light depending on the relative number of nodes in its subtree.

Let $size(v)$ be the number of nodes in v 's subtree (in the represented tree).

Definition 1. An edge from vertex $parent(v)$ to v is called **heavy** if $size(v) > \frac{1}{2}size(parent(v))$, and otherwise it is called **light**.

Furthermore, let $light\text{-depth}(v)$ denote the number of light edges on the root-to-vertex path to v . Note that $light\text{-depth}(v) \leq \lg n$ because as we go down one light edge we decrease the number of nodes in our current subtree at least a factor of 2. In addition, note that each node has at most one heavy edge to a child, because at most one child subtree contains more than half of the nodes of its parent's subtree. There are four possibilities for edges in the represented tree: they can be preferred or unpreferred and heavy or light.

4.1.2 Proof of the $O(\lg^2 n)$ bound

To bound the number of preferred child changes, we do Heavy-Light decomposition on represented trees. Note that `access` does not change the represented tree, so it does not change the heavy or light classification of edges. For every change of preferred edge (possibly except for one change to the preferred edge that comes out of the accessed node) there exists a newly created preferred edge. So, we count the number of edges which change status to being preferred. Per operation, there are at most $\lg n$ edges which are light and become preferred (because all edges that become preferred are on a path starting from the root, and there can be at most $\lg n$ light edges on a path by the observation above). Now, it remains to ask how many heavy edges become preferred. For any one operation, this number can be arbitrarily large, but we can bound it to $O(\lg n)$ amortized. How come? Well, during the entire execution the number of events "heavy edge becomes preferred" is bounded by the number of events "heavy edge become unpreferred" plus $n - 1$ (because at the end, there can be $n - 1$ heavy preferred edges and at the beginning the might have been none). But when a heavy edge becomes unpreferred, a light edge becomes preferred. We've already seen that there at most $\lg n$ such events per operation in the worst-case. So there are $\leq \lg n$ events "heavy edge becomes unpreferred" per operation. So in an amortized sense, there are $\leq \lg n$ events "heavy edge becomes preferred" per operation (provided $(n - 1)/m$ is small, i.e. there is a sufficiently large sequence of operations).

4.2 An $O(\lg n)$ bound.

We prove the $O(\lg n)$ bound by showing that the cost of preferred child switch is actually $O(1)$ amortized. From `access`'s pseudo code one can easily see that its cost is

$$O(\lg n) + (\text{cost of pref child switch}) * (\# \text{ pref child switches})$$

From the above analysis we already know that the number of preferred child switches is $O(\lg n)$, thus it is enough to show that the cost of preferred child switch is $O(1)$. We do it using the potential method.

Let $s(v)$ be the number of nodes under v in the tree of auxiliary trees. Then we define the potential function $\Phi = \sum_v \lg s(v)$. From our previous study of splay trees, we have the Access theorem, which states that:

$$\text{cost}(\text{splay}(v)) \leq 3(\lg s(u) - \lg s(v)) + 1,$$

where u is the root of v 's auxiliary tree. Now note that splaying v affects only values of s for nodes in v 's auxiliary tree and changing v 's preferred child changes the structure of the auxiliary tree but the tree of auxiliary trees remains unchanged. Therefore, on `access(v)`, values of s change only for nodes inside v 's auxiliary tree. Also note that if w is the parent of the root of auxiliary tree containing v , then we have that $s(v) \leq s(u) \leq s(w)$. Now we can use this inequality and the above amortized cost for each iteration of the loop in `access`. The summation telescopes and is less than

$$3(\lg s(\text{root of represented tree}) - \lg s(v)) + O(\# \text{ preferred child changes})$$

which in turn is $O(\lg n)$ since $s(\text{root}) = n$. Thus the cost of `access` is thus $O(\lg n)$ amortized as desired.

To complete the analysis we resolve the worry that the potential might increase more than $O(\lg n)$ after cutting or joining. Cutting breaks up the tree into two trees thus values of s only decrease and thus Φ also decreases. When joining v and w , only the value of s at v increases as it becomes the root of the tree of auxiliary trees. However, since $s(v) \leq n$, the potential increases by at most $\lg s(v) = \lg n$. Thus increase of potential is small and cost of cutting and joining is $O(\lg n)$ amortized.

References

- [1] D. D. Sleator, R. E. Tarjan, *A Data Structure for Dynamic Trees*, Journal. Comput. Syst. Sci., 28(3):362-391, 1983.
- [2] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1984.