**CS 573: Topics in Analysis of Algorithms** **Spring 2006**

Lecture 7 — February 7

*Lecturer: Jeff Erickson* *Scribe: Chris Osborn*

## 7.1 Overview

Previously, we investigated Wilber lower bounds for binary search trees, and we looked at Tango trees which are $O(\lg\lg n)$-competitive with this bound.

In this lecture we investigate the Link-Cut tree, for which all operations are performed in $O(\lg n)$ amortized time. The Link-Cut tree, while less useful as a general-purpose tree data structure, is useful for applications such as Network Flow.

## 7.2 Operations

We wish to support the following operations:

- MAKE_TREE(x)

  Creates and returns singleton tree with root value $x$
- CUT(v)

  Deletes the edge from $v$ to its parent
- JOIN(v, w)

  Where $v$ is a root vertex, assigns $\mathbf{parent}(\mathbf{v}) \leftarrow \mathbf{w}$ (makes $v$ a child of $w$)
- FIND_ROOT(v)

  Returns the root of the tree containing $v$

## 7.3 Link-Cut Trees

Link-Cut Trees were developed by Sleator and Tarjan[1].

A Link-Cut Tree represents a standard binary tree (augmented to indicate preferred children, edges and paths), in a non-standard way.

The "represented tree" is actually a forest of rooted trees, each of which is no different from a standard, rooted tree except that the following "preferences" may be indicated:

- A node may "prefer" one of its children. The child becomes a **preferred child**.
- An edge between a parent and a *preferred child* is a **preferred edge**.
- A path containing only *preferred edges* is a **preferred path**. May be a single vertex.

Usually, we are interested in what happens to the represented tree, so we will often first indicate how a particular operation would be performed on the represented tree, and then explain how such an operation is carried out in the Link-Cut Tree.

Link-Cut Trees are defined as follows:

- A **path tree** is a tree representing some **preferred path** in the *represented tree*. The underlying data structure is a splay tree containing all nodes of the *preferred path*, keyed by their depth in the *represented tree*.
- A **Link-Cut tree** is a decomposition of the *represented tree* into *preferred paths*, such that each *preferred path* is represented by a corresponding *path tree*.
- The root node of each *path tree* has a unidirectional **path pointer** to its parent node in whatever *path tree* the parent node resides.

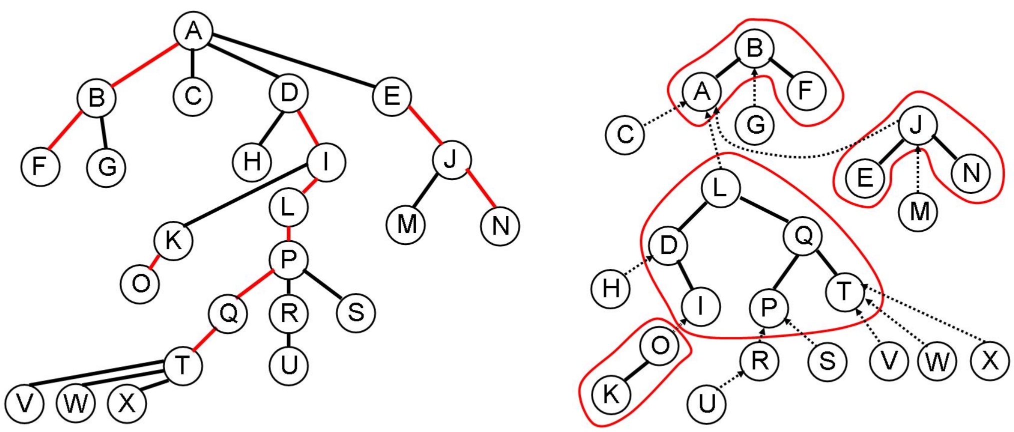

**Table 7.1.** On the left, a represented tree, with preferred edges in red. On the right, the corresponding Link-Cut Tree. Path-parent pointers are dotted.

## 7.4 Operations on Link-Cut Trees

All Link-Cut tree operations call a function Access($v$) to do the majority of the work.

Access($v$) reorganizes the *represented tree* so that $v$ is on the preferred path containing the root, and makes $v$ the root of its *path tree* in the Link-Cut Tree.

To accomplish this, Access first removes the "preference" from any preferred edge adjacent to $v$ and one of $v$'s children.

Then, Access climbs up the tree to the root, at each vertex $w$ updating $w$'s *preferred edge* to be that which extends the preferred path containing $v$. The process terminates when $w$ is the root, and $w$'s preferred edge has been updated.

Here is the pseudocode for Access:

**Splay**($v$) **// in its path tree**

**// Remove** $v$**'s preferred child**

**pathparent(right**($v$)**)** $\leftarrow v$

**right**($v$) $\leftarrow$ **Null**

$v_t \leftarrow v$

**while** $v_t \neq$ **root**

  $w \leftarrow$ **pathparent**($v_t$)

  **Splay**($w$)

  **// Update** $w$**'s preferred child**

  **pathparent(right**($w$)**)**$\leftarrow w$

  **right**($w$) $\leftarrow v_t$

  **pathparent**($v_t$) $\leftarrow$ **Null**

  $v_t \leftarrow w$

**Splay**($v$)

Note that when we splay some vertex $v$ within its path tree $T$, $v$ becomes the root of $T$. Because $T$ is keyed by depth, the right subtree $W$ of $v$ now contains all vertices $w \in (W \subset T)$ for which $depth(w) > depth(v)$. And these vertices $w$ are precisely those which are deeper than $v$ in the preferred path $T$ containing $v$. $W$ contains the vertices that we desire to "cut off" from the preferred path.

To remove the vertices in $W$ from $T$, we first set pathparent(right($v$)) to $v$. This means that $W$ is now its **own** preferred path, with "path parent" $v$.

Lastly, we set right($v$) to point to the **new** preferred path, and set the "path parent" of the root of that path to Null.

The code at the beginning is a special case of the code in the loop, where we "unprefer" a preferred edge but do not "prefer" any new edge.

## 7.5 More Pseudocode

### 7.5.1 Cut

**Cut**($v$)

> **Access**($v$)
>
> **left**($v$) $\leftarrow$ **Null**

The call to Access puts $v$ in a preferred path $T$, containing the root of the Link-Cut Tree. Further, $v$ is at the root of $T$. Thus, any vertices that are shallower than $v$ in $T$ are in the left subtree of $v$. We therefore achieve a cut by setting left($v$) $\leftarrow$ Null.

### 7.5.2 Join

**Join**($v$, $w$)

> **Access**($v$)
>
> **Access**($w$)
>
> **left**($v$) $\leftarrow w$

The first call to Access puts $v$ in a preferred path. The second call makes $w$ the root of a preferred path $T$, also containing $v$. By setting $left(v)$ to $w$, we make $v$ a child of $w$.

### 7.5.3 FindRoot

**FindRoot**($v$)

> **Access**($v$)
>
> **while left**($v$) $\neq$ **Null**
>
>     $v \leftarrow$ **left**($v$)
>
> **Splay**($v$)
>
> **Return**($v$)

Note the root of $v$ will be in the same path tree as $v$ after Access is called. Furthermore, the root will be the leftmost node in that path tree.

## 7.6 Heavy-Light Decomposition

The run-time complexity of every function we have documented is dominated by the complexity of Access. We aim to show that Access has run-time complexity $O((\lg n)^2)$.

Since Access works by iteratively splaying, where each splay is done in $O(\lg n)$ time, it suffices to show that the number of splays is $O(\lg n)$.

Let $size(v)$ be the number of nodes in the subtree rooted at $v$.

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

Let $lightdepth(v)$ be the number of light edges from in the path from $v$ to its root.

Note that $lightdepth(v) \leq \lg n$: suppose that the tree contains $n$ vertices. Let $m$ be a lower-bound on $n$, which starts at 1, counting only $v$. Now we traverse the path taken by Access, starting at $v$ and ending at the root. Each time we take a light edge, the value of $m$ must at least double. Therefore, after only a logarithmic number of light-edge traversals, we have $m > n$.

So the analysis is as follows:

$$\begin{aligned}
&\#\text{edges that become preferred} \\
&\qquad \leq \#\text{light edges pref.} + \#\text{heavy edges pref.} \\
&\qquad \leq \lg n + \#\text{heavy edges pref.}
\end{aligned}$$

Over a series of $m$ calls to Access:

$$\begin{aligned}
&\text{total } \#\text{ heavy edges that become pref.} \\
&\qquad \leq \text{total } \#\text{heavy edges that become } \underline{\text{un}}\text{-pref.} + (n-1) \\
&\qquad \leq \text{total } \#\ \underline{\text{light}} \text{ edges that become pref.} + (n-1) \\
&\qquad \leq m \lg n + n - 1
\end{aligned}$$

Thus, the total number of prefered edges changes is $O(m \lg n + n)$, and the amoritized number of prefered edge changes per call to Access is $O(\lg n)$.

This completes the $O((\lg n)^2)$ bound.

## 7.7 Improving the Bound to $O(\lg n)$

We can do even better and achieve an amortized cost of $O(\lg n)$. To do so, we show that the amortized cost of switching a preferred child is actually $O(1)$. We use the potential method.

Let $s(v) = \#$ of nodes in $v$'s subtree in $T$ (path tree).
Let $\Phi(T) = \sum_{v \in T} \lg s(v)$.

The Access Lemma tells us that the amortized cost of a splay is bounded by:

$$3 \lg(size(root(v))) - 3 \lg(size(v)) + 1$$

Note that after splaying $v$, $v$ is joined to its path-parent $w$, and we have that $size(w) > size(v)$.

This results in a telescoping sum, bounded by:

$$3 \lg n - 3 \lg size(v) + O(\#\text{pref. edge changes})$$

The last term is $O(\lg n)$, so the result is an amortized $O(\lg n)$ bound.

# Bibliography

[1] D. D. Sleator, R.E. Tarjan, *A Data Structure for Dynamic Trees*, Journal. Comput. Syst. Sci., 1983.