6.897: Advanced Data Structures Spring 2005

Lecture 5 — Feb 15, 2005

Prof. Erik Demaine *Scribe: Igor Ganichev*

1 Overview

In the last lecture we worked in a BST model. We discussed Wilber lower bounds for BSTs for a particular access sequence, and developed Tango trees which achieve $O(\lg \lg n)$-compatitiveness. This is a very promising result that gives hope for existence of a dynamically optimal tree.

In this lecture we discuss dynamic trees that have many applications such as to Network Flow and Dynamic Connectivity problems in addition to them being interesting theoretically. We will discuss one data structure called Link-Cut trees that achieves logarithmic amortized time for all operations.

2 Dynamic Trees

The dynamic tree problem we will discuss today is to represent a forest of rooted trees whose each node has an unordered set of children of arbitrary size. The data structure has to support the following operations:

- *MAKE_TREE()* – Returns a new vertex in a singleton tree. This operation allows us to add elements and later manipulate them.
- *CUT(v)* – Deletes the edge between vertex v and its parent, $parent(v)$.
- *JOIN(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.
- *FIND_ROOT(v)* – Returns the root of the tree that vertex v is a node of.

It is also possible to augment the data structure to return statistics about the path from the v to the root of its tree, such as the sum or minimum of the weights of each edge. This augmentation is necessary in flow algorithms.

3 Link-Cut Trees

Link-Cut Trees were developed by Sleator and Tarjan [1]. 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. The also use splay trees for the internal representation.

3.1 Definition of Link-Cut Trees

We say a vertex has been *accessed* if was passed to any of the operations from above as an argument. We call *represented trees* the abstract trees that the data structure represents.

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 every node is in exactly one preferred path.

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. 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. Including the path-parent pointers as edges, we have a representation of a represented tree as a tree of auxiliary trees.

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 prefered 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 it right subtree, making it a 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 usefull 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. 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 auxiliray 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

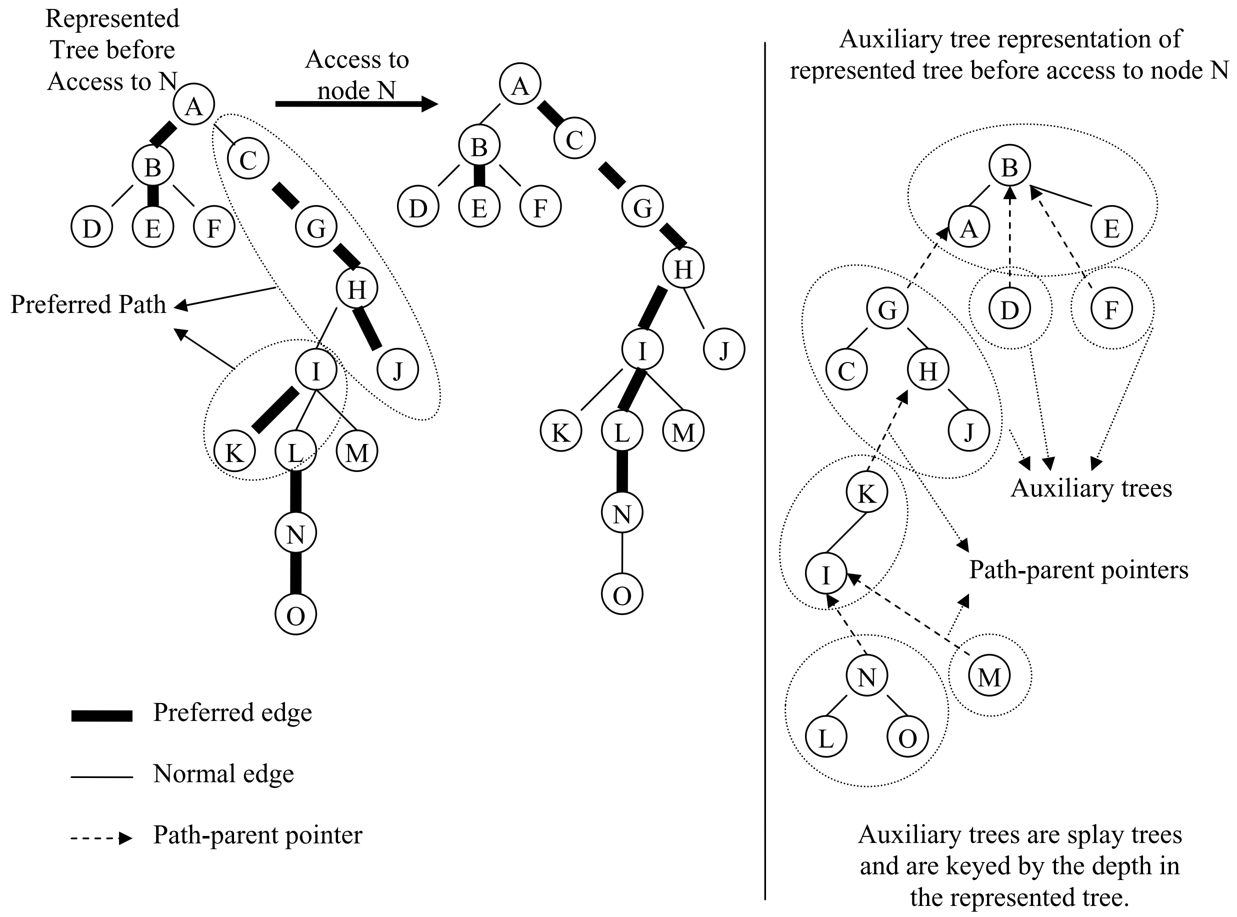
Represented Tree before Access to N
Access to node N
Auxiliary tree representation of represented tree before access to node N
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
Preferred Path
Auxiliary trees
Path-parent pointers
Preferred edge
Normal edge
Path-parent pointer
Auxiliary trees are splay trees and are keyed by the depth in the represented tree.

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

```
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.
      – path-parent(right(v)) $\leftarrow v$
      – right($v$) $\leftarrow null$
  – loop until we reach the root
      – $w \leftarrow$ path-parent($w$)
      – splay $w$
      – switch $w$'s preferred child
          – path-parent(right($w$)) $\leftarrow w$
          – right($w$) $\leftarrow v$
          – path-parent($v$) $\leftarrow null$
      – $v \leftarrow w$
  – splay $v$ just for convinience
```

make v auxiliary tree the right subtree of w. Finally, we have to do minor housekeeping to finish one iteration: since v is no longer the root of its auxiliary tree, we nullify its path-parent pointer. We continue building up the preferred path in the same way, until we reach the root of R.

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.

```
FIND_ROOT($v$)
  – access($v$)
  – Set $v$ to the smallest element in the auxiliary tree, i.e. to the root of the represented tree
      – $v \leftarrow$ left($v$) until left($v$) is $null$
  – splay $v$
  – return $v$
```

3.2.3 Cut

To cut $(v, parent(v)$ edge in the represented tree means that we have to separate nodes in v's subtree (in represented tree) from the tree of auxilary 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.

```
CUT($v$)
    – access($v$)
    – left($v$) $\leftarrow$ $null$
```

3.2.4 Join

Joining 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.

```
JOIN($v, w$)
    – access($v$)
    – access($w$)
    – left($v$) $\leftarrow$ $w$
```

4 Analysis

As one can see from the pseudocode, all operations are doing at most logarithmic work besides calls access. Thus it is enough to bound the runtime of access. First we show an $O(\lg^2 n)$ bound.

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

From access's pseudocode 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. Furthermore, the loop iterates once per jump from one preferred path to the one above (pointed to by path-parent pointer), or in other words, once per preferred child change. 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. We accomplish this by using the Heavy-Light Decomposition.

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.

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

Furthermore, let light-depth(v) denote the number of light edges from vertex v to the root. Note that light-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 coming out of it because there can be at most one child whose subtree contains more than half of the nodes of its parent's subtree.

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. 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 prseudocode one can easily see that its cost is

$$O(\lg n) + (cost\_of\_pref\_child\_switch) * (number\_of\_pref\_child\_switches)$$

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 $\Phi = \sum_v \lg s(v)$. Using the Access Theorem we get:

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

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 auxiliry tree but the tree of auxiliary trees remains unchanged. Therefore, values of s change only for nodes inside v's auxiliary tree after an access to v. 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 add equations above along the preferred path from v to the root for every preferred child change made in the access. It telescopes and is less than

$$3(\lg s(root\ of\ represented\, tree) - \lg s(v)) + O(number\ of\ preferred\, child\, changes)$$

which in turn is $O(\lg n)$ since $s(root) = n$. Thus the cost of access is $O(\lg n)$ 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.