CS 61B: Lecture 24
Wednesday, March 19, 2014

Today's reading: Goodrich & Tamassia, Chapter 7.

ROOTED TREES

A _tree_ consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the property that there is exactly one path (no more, no less) between any two nodes of the tree. A _path_ is a sequence of one or more nodes, each consecutive pair being connected by an edge.

In a _rooted_ tree, one distinguished node is called the _root_. Every node c, except the root, has exactly one _parent_ node p, which is the first node after c on the path from c to the root. c is p's _child_. The root has no parent. A node can have any number of children.

Some other definitions:

- A _leaf_ is a node with no children.
- An _internal_node_ is a non-leaf node (having one or more children).
- _Siblings_ are nodes with the same parent.
- The _ancestors_ of a node d are the nodes on the path from d to the root. These include d's parent, d's parent's parent, d's parent's parent's parent, and so forth up to the root. Technically, the ancestors of d also include d itself, which makes you wonder about d's sex life. The root is an ancestor of every node in the tree.
- If a is an ancestor of d, then d is a _descendant_ of a.
- The _length_ of a path is the number of edges in the path.
- The _depth_ of a node n is the length of the path from n to the root. (The depth of the root is zero.)
- The _height_ of a node n is the length of the path from n to its deepest descendant. (The height of a leaf node is zero.)
- The height of a tree is the depth of its deepest node = height of the root.
- The _subtree_ rooted at node n is the tree formed by n and its descendants.
- A _binary_tree_ is a tree in which no node has more than two children, and every child is either a _left_child_ or a _right_child_, even if it's the only child its parent has.

A commonly encountered application of trees is the directory structure of a file system.

```
                       _______~jrs/61b_______                <-- Root node
                      /       |        |      \
                     /        |        |       \
                   hw   index.html   lab      _lec__
                  / \                /\      / /\ \ \_
                 /   \     ^        /  \    / /  \ \  \
               hw1  hw2    |      lab1 lab2 01 02 03 04 05   <-- Leaf nodes
                       Leaf node
```

Representing Rooted Trees

Goodrich and Tamassia present a data structure in which each node has three references: one reference to an item, one reference to the node's parent, and one reference to the node's children, which can be stored in any reasonable data structure like a linked list. Directories are typically stored this way, but the lists they use are represented very differently than our list ADTs.

Another popular tree representation spurns separately encapsulated linked lists so that siblings are directly linked. It retains the "item" and "parent" references, but instead of referencing a list of children, each node references just its leftmost child. Each node also references its next sibling to the right. The "nextSibling" references are used to join the children of a node in a singly-linked list, whose head is the node's "firstChild".

I'll call this tree a "SibTree", since siblings are central to the representation. The nodes I call "SibTreeNodes".

```
class SibTreeNode {                    |  class SibTree {
  Object item;                         |    SibTreeNode root;
  SibTreeNode parent;                  |    int size;
  SibTreeNode firstChild;              |  }
  SibTreeNode nextSibling;             |
}                                      |
```

```
================================================================================
+ ROOTED TREE | ---------------------             ---------------------------- +
=============== |---           ---- |             |          parent          | +
+               ||.|root   size|14| |             ---------------------------- +
+               |-+-           ---- |             |           item           | +
+               --|------------------             ---------------------------- +
+                 v     SibTree object            | firstChild | nextSibling | +
+               -----                             ---------------------------- +
+               | * |                               structure of SibTreeNodes  +
+               -----                                                          +
+ Root node =>  |jrs|                                                          +
+               -----<---------                                                +
+               |.|*|          \                                               +
+               /----<----      \                                              +
+              /  ^^      \      \                                             +
+             v  /  \      \      \                                            +
+            ---/-  -\---  -\---  -\---                                        +
+            | . |  | . |  | . |  | . |                                        +
+            -----  -----  -----  -----                                        +
+            |hw |  |ind|  |lab|  |lec|<-------------------------              +
+            -----  -----  -----  -----<-------------------      \             +
+            |.|.+->|*|.+->|.|.+->|.|*|<-------------      \      \            +
+            /----  -----  /----  --\--<------       \      \      \           +
+           /  ^^         /   ^^      \ ^      \      \      \      \          +
+          v  /  \       v   /  \      \ \      \      \      \      \         +
+         ---/-  -\---   ---/-  -\---   >-\---  -\---  -\---  -\---  -\---     +
+         | . |  | . |   | . |  | . |    | . |  | . |  | . |  | . |  | . |     +
+         -----  -----   -----  -----    -----  -----  -----  -----  -----     +
+         |hw1|  |hw2|   |lb1|  |lb2|    |01 |  |02 |  |03 |  |04 |  |05 |     +
+         -----  -----   -----  -----    -----  -----  -----  -----  -----     +
+         |*|.+->|*|*|   |*|.+->|*|*|    |*|.+->|*|.+->|*|.+->|*|.+->|*|*|     +
+         -----  -----   -----  -----    -----  -----  -----  -----  -----     +
================================================================================
```

```
Tree Traversals
---------------
A _traversal_ is a manner of _visiting_ each node in a tree once.  What you do
when visiting any particular node depends on the application; for instance,
you might print a node's value, or perform some calculation upon it.  There
are several different traversals, each of which orders the nodes differently.

Many traversals can be defined recursively.  In a _preorder_ traversal, you
visit each node before recursively visiting its children, which are visited
from left to right.  The root is visited first.

class SibTreeNode {
  public void preorder() {
    this.visit();
    if (firstChild != null) {
      firstChild.preorder();
    }
    if (nextSibling != null) {
      nextSibling.preorder();
    }
  }
}

Suppose your method visit() numbers the nodes in the order they're visited.
A preorder traversal visits the nodes in this order.

        1
       / \
      /   \
     2     6
    /|\   / \
   3 4 5 7   8

Each node is visited only once, so a preorder traversal takes O(n) time, where
n is the number of nodes in the tree.  All the traversals we will consider take
O(n) time.

A preorder traversal is a natural way to print a directory's structure.
Simply have the method visit() print each node of the tree.

~jrs/61b
   hw
      hw1
      hw2
   index.html
   lab
      lab1
      lab2
   lec
      01
      02
      03
      04
      05
```

```
In a _postorder_ traversal, you visit each node's children (in left-to-right
order) before the node itself.

  public void postorder() {
    if (firstChild != null) {
      firstChild.postorder();
    }
    this.visit();
    if (nextSibling != null) {
      nextSibling.postorder();
    }
  }

A postorder traversal visits the nodes in this order.

        8
       / \
      /   \
     4     7
    /|\   / \
   1 2 3 5   6

The postorder() code is trickier than it looks.  The best way to understand it
is to draw a depth-two tree on paper, then pretend you're the computer and
execute the algorithm carefully.  Trust me on this.  It's worth your time.

A postorder traversal is the natural way to sum the total disk space used in
the root directory and its descendants.  The method visit() sums "this" node's
disk space with the disk space of all its children.  In the example above, a
postorder traversal would first sum the sizes of the files in hw1/ and hw2/;
then it would visit hw/ and sum its two children.  The last thing it would
compute is the total disk space at the root ~jrs/61b/, which sums all the
files in the tree.

Binary trees allow for an _inorder_ traversal:  recursively traverse the root's
left subtree (rooted at the left child), then the root itself, then the root's
right subtree.  The preorder, inorder, and postorder traversals of an
expression tree will print a _prefix_, _infix_, or _postfix_ expression,
respectively.

        +
       / \         Prefix:  + * 3 7 ^ 4 2
      /   \
     *     ^        Infix:  3 * 7 + 4 ^ 2
    / \   / \
   3   7 4   2    Postfix:  3 7 * 4 2 ^ +

In a _level-order_ traversal, you visit the root, then all the depth-1 nodes
(from left to right), then all the depth-2 nodes, et cetera.  The level-order
traversal of our expression tree is "+ * ^ 3 7 4 2" (which doesn't mean much).

Unlike the three previous traversals, a level-order traversal is not
straightforward to define recursively.  However, a level-order traversal can be
done in O(n) time.  Use a queue, which initially contains only the root.  Then
repeat the following steps:
  - Dequeue a node.
  - Visit it.
  - Enqueue its children (in order from left to right).
Continue until the queue is empty.

A final thought:  if you use a stack instead of a queue, and push each node's
children in reverse order--from right to left (so they pop off the stack in
order from left to right)--you perform a preorder traversal.  Think about why.
```