(Rooted) Trees

Thus far we have been working mostly with "linear" collections, namely *lists*. For each element in a list, it makes sense to talk about the previous element (if it exists) and the next element (if it exists). It is often useful to organizes a collection of elements in a "non-linear" way. In the next several lectures, we will look at some examples. Today we begin with *(rooted) trees*.

Like a list, a rooted tree is composed of nodes that reference one another. With a tree, each node can have one "parent" and multiple "children". You can think of the parent as the "prev" node and the children as "next" nodes. The key difference here is that a node can have multiple children, whereas in a linked list a node has (at most) one "next" node.

You are familiar with the concept of rooted trees already. Here are a few examples.

- Many organizations have a hierarchical structures that are trees. For example, see the McGill organizational chart. `http://www.mcgill.ca/orgchart/` which is (almost) a tree. Note that the "lowest" level in this chart contains the Deans but of course there are thousands of employees at McGill "below" the Deans, which are not shown. For example, as a McGill professor in the School of Computer Science, I report to my department Chair (Professor Bettina Kemme), who reports to the Dean of Science, who reports to the McGill Provost, who to the Principal. A professor in the Department of Electrical and Computer Engineering reports to the Chair of ECE, who reports to the Dean of Engineering, who like the Dean of Science reports to the Provost, who reports to the Principal. (The "B reports to A" relationship in an organization hierarchy defines a "B is a child of A" relation in a tree – see below).

- Family trees. There are two kinds of family trees you might consider. The first defines the parent/child relation literally, so that the tree children of a node correspond to the actual children (sons and daughters) of the person represented by the node. The second tree is less conventional. It defines each person's mother and father as its "children". In such a tree, each person has two children (the person's real parents), and they each of two children (the person's four grandparents), etc.

- A file directory on a MS Windows or UNIX/LINUX operating system. For example, this lecture notes file is stored at

  `C:\Users\Michael\Dropbox\TEACHING\250\LECTURENOTES\17-trees.pdf`

[ASIDE: The lecture slides used many figures to illustrate the ideas that I will be presenting in today's notes. I am not including the figures in these notes. But you should definitely consult those slides when reading these notes, or you will have a tough time!]

Terminology for rooted trees

A (rooted) tree consists of a set of *nodes* or *vertices*, and *edges* which are ordered pairs of nodes or vertices. When we write an edge (v_1, v_2), we mean that the edge goes from node v_1 to v_2. The "node" vs. "vertex" terminology is perhaps a bit confusing at first. When one is discussing data

structures, it is more common to use the term "node" and when one is talking about trees as abstract data types one typically uses the term "vertex."

In COMP 250 we will only deal with a special type of tree called a *rooted* tree. A rooted tree has special node called the *root node*. Rooted trees have the following properties:

- For any node v in the tree (except the root node), there is a unique node p such that (p, v) is an edge in the tree. The node p is called the *parent* of v, . Naturally, v is called a *child* of p. A parent can have multiple children.

  Notice that a tree with n nodes has $n-1$ edges. Why? Because for each node v except the root node ($n-1$ of these), there is a unique edge (p, v) and these $n-1$ edges are exactly the set of edges in the tree.

- *sibling relation*: Two nodes are siblings if they have the same parent.

- *leaf*: A node with no children is called a leaf node. A more complicated way of saying a "node with no children" is "a node v such that there does *not* exist an edge (v, w) where w is a node in the tree". Leaves are also called *external* nodes.

- *internal* node: a node that has a child (i.e. a node that is not a leaf node).

  For example, in the linux or windows file system, files and empty directories are leaf nodes, and non-empty directories are internal nodes. A directory is a file that contains a list of references to its children nodes, which may be files (leaves) or subdirectories (internal nodes).

- *path*: a sequence of nodes $v_1, v_2, \ldots, v_k$ where v_i is the parent of v_{i+1} for all i.

- *length of a path*: the number of edges in the path. If a path has k nodes, then it has length $k-1$.

  You can define a path of length 0, namely a path that consists of just one node v. (This is useful sometimes, if you want to make certain mathematical statements bulletproof.)

- *depth* of a node in a tree (also called *level* of the node): the length of the (unique) path from the root to the node. Note that the root node is at depth 0.

- *height* of a node v in a tree: the maximum length of a path from v to a leaf. Note: a leaf has height 0.

- *height* of a tree: the height of the root node

- *ancestor*: v is an ancestor of w if there is a path from v to w

- *descendent*: w is a descendent of v if there is a path from v to w. Note that "v is an ancestor of w" is equivalent to "w is a descendent of v".

- A subtree is a subset of nodes and edges in a tree which itself is a tree. In particular, a subtree has its own root which may or may not be the same as the root of the original tree. Every node in a tree defines a subtree, namely the tree defined by this node and all its children. In particular, any tree is a subtree of itself. Every node in a tree also defines a subtree in which that node is the only node.

- *recursive definition of a rooted tree:* A rooted tree T either has no nodes (empty tree), or it consists of a root node r together with a set of zero or more non-empty subtrees $T_1, \ldots, T_k$ whose roots are the children of r.

We can also define operations on trees recursively. Here are a few common examples. The first is to compute the depth of a node.

```
depth(v){
  if (v is a root node)    //  that is,  v.parent == null
    return 0
  else
    return 1 + depth(v.parent)
}
```

Notice that this method requires that we can access the parent of a node. We have defined edges to be of the form (parent, child). If one implements an edge by putting a child reference in a parent node then, given a node v, we can only reference the child of this node, not the parent, and so we would not be able to compute the depth. Therefore, to use the above method for computing depth, we would need a node to have a reference to its parent. (Whether we need references to children too depends on what we will use the tree for. In most figures in the slides today, I drew trees that had references only from parents to children.)

Another example operation is to compute the height of a node. Just like the depth, the height can be computed recursively.

```
height(v){
  if (v is a leaf)    //  that is,  v.child == null for any child
    return 0
  else{
    h = 0
    for each child w of v
       h = max(h, height(w))
    return 1 + h
  }
}
```

Tree implementation in Java

The main decision one needs to make in implementing trees is how to represent the set of children of a node. If node can have at most two children (as is the case of a binary tree, which we will describe soon) then each node can be given exactly two reference variables for the children. If a node can have many children, then a more flexible approach is needed.

One common approach is to define the children by an array list or by a linked list.

```
class  TreeNode<T>{
    T   element;
    ArrayList<TreeNode<T>>   children;

    TreeNode<T>   parent;   // optional
     :
     :                     //  methods
}
```

This approach is perhaps overkill though, since one typicallly doesn't need to index the children by their number. Using a `LinkedList` (doubly linked list in Java) is also perhaps overkill if one doesn't need to be able to go both directions in the list of children. A singly linked list is often enough, and that is the approach below.

The *"first child, next sibling"* implementation of a tree uses the following node definition.

```
  class  TreeNode<T>{
      T   element;
      TreeNode<T>   firstChild;
      TreeNode<T>   nextSibling;
        :
        :                      //  methods
  }
```

It defines a singly linked list for the siblings, where the head is the `firstChild` and the next is the `nextSibling`. It is simpler than the arraylist implementation since it uses just two fixed references at each node (or 3, if one also has a parent link)

Finally, to define the rooted tree, we could use:

```
  class  Tree<T>{
     TreeNode<T>   root;
         :
         :                      //  methods
  }
```

The `root` here serves the same role as the `head` field in our implementation of `SLinkedList`. It gives you access to the nodes of the tree. The `TreeNode` class would then be defined as an inner class inside this `Tree` class.

Exercise: Representing trees using lists

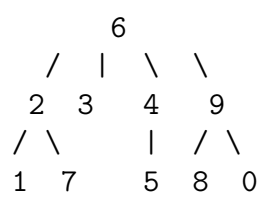
A tree can be represented using lists, as follows:

```
tree             =   root | ( root listOfSubTrees )
listOfSubTrees   =   tree |   tree listOfSubTrees
```

For example, let's draw the tree that corresponds to the following list, where the `root` elements are single digits. (Apologies for the ASCII art below. See the slides for prettier pictures.)

```
( 6 ( 2 1 7 ) 3 ( 4 5 ) ( 9 8 0 ) )
```

The first uses a separate edge for each parent/child pair.



The second uses the "first child, next sibling" representation.