(Rooted) Trees

Thus far we have been working with "linear" collections, namely lists. For each element, it made sense to talk about the previous element (if it exists) and the next element (if it exists). Lists have limitations, though. If you use a linked list, then accessing an arbitrary element can be slow (though adding and removing the element can be fast *once you have accessed it*). If you use an array, then access is fast, but adding and removing can be slow.

To get around these limitations, one often organizes a collection of items in a "non-linear" way. We now turn to our first example: rooted trees.

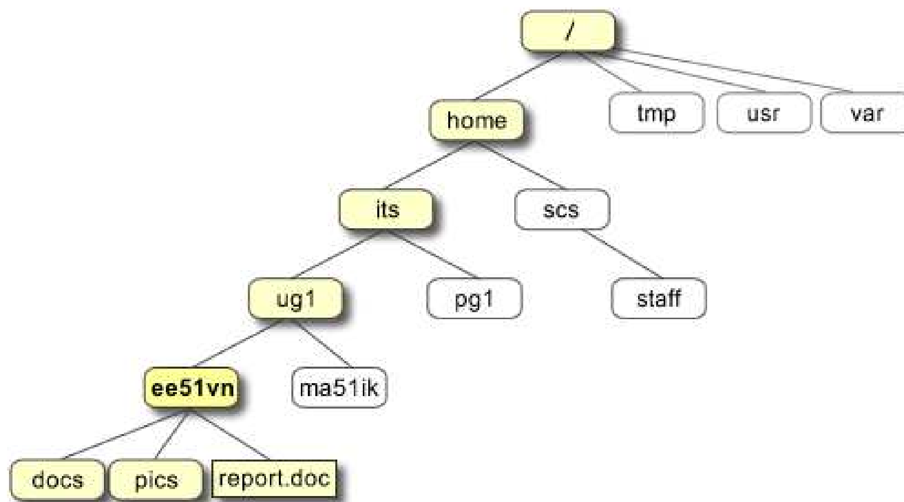
Like a list, a tree is composed of nodes that reference one another. Each node (except the "root node") has exactly one "parent", and each node can have multiple "children". You can think of the parent as the "prev" node and the children as "next" nodes. So what's new here is that a node can have multiple children.

You are familiar with (rooted) trees already. Here are a few examples.

- directory and file structures on a MS Windows or UNIX/LINUX operating system. For example, this lecture notes file is stored on a UNIX system and has a path

  `/home/perception/langer/public_html/250/lecture22.pdf`

  where the backslashes indicate a parent/child relationship. The "/home" at the beginning indicates that "home" is a child of the root directory "/". The root directory has other children such as "usr", "sys", "lib". Here is another example (shamelessly stolen from the web):

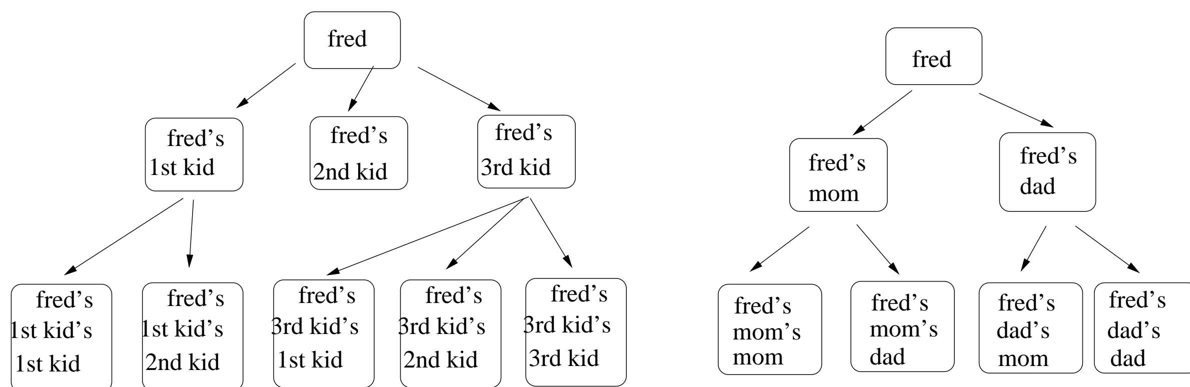


- Many organizations have a hierarchical structures which define trees. For example, as a McGill professor in the School of Computer Science, I report to my department Chair, who reports to the the Dean of Science, who reports to the McGill Provost, who reports to the Principal. A professor in the Department of Electrical and Computer Engineering would report 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. See `https://home.mcgill.ca/orgchart` Hopefully you have access to it.

Note: the "B reports to A" relationship in an organization hierarchy defines a "B is a child of A" relation in a tree (see definitions below.)

- Inheritence relationships between classes in Java. "class B extends class A" defines a "B is a child of A" relation. (Note: interfaces do not need to obey a tree structure, since a one interface can extend multiple interfaces, and a class can implement multiple interfaces.)
- Family trees. There are two trees you might consider, both having a person "fred" at the root. The first tree is the more conventional "family tree. It defines the parent/child relation literally, so that the tree children of a node correspond to the actual children ("kids") of the person represented by the node. The second tree is less conventional, but it is interesting too so I'll mention it. It defines each person's mother and father as its "tree children". A person's real mother and father are obviously not the person's "children", but here we are using "children" only in a formal sense of a tree definition. Note that, in this sense, each person has two children (the person's real parents). Thus, each person has four grandparents, eight greatgrandparents, etc.



Definitions

Here are a few definitions of terms we will use. We assume we have a finite collection (set) of *nodes*[1]

- *edge* - an ordered pair of nodes of the form $(parent, child)$.
- *(rooted) tree* - a collection of nodes such that:
  - If the collection is empty, we have an empty tree. Otherwise, there is a unique node called the *root node* of the tree.

[1] We do not formally define a *node* here. But if this bothers you, then takes a Java programmer's perspective and consider a node to be an object of some unspecified class.

  - Every non-root node v in the tree has a unique parent, that is, there is a *unique* edge $(v.parent, v)$ in the tree. The root node does not have a parent, that is, there is no edge of the form $(v.parent, v)$.

  Notice that a tree with n nodes has $n-1$ edges. The reason is that each node (except the root) has exactly one parent.

- *sibling relation*: two nodes are siblings if they have the same parent. (The parent of a node is unique.)

- *leaf*: a node with no children, that is, a node v such that there does *not* exist a node w with $v = parent(w)$. Leaves are also called *external* nodes.

- *internal* node: a node that has a child (i.e. a node that is not a leaf node).[2]

  For example, in the UNIX file system, files and empty directories are leaf nodes, and non-empty directories are internal nodes. (Think of a directory as a file which contains a list of references to the children nodes.)

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

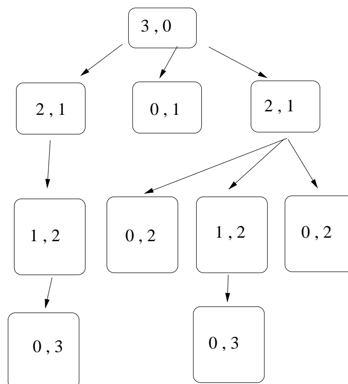
- *length of a path*: the number of edges in the path. If a path has k vertices, then it has length $k-1$

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

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

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

  Here is an example of some (height,depth) pairs for a tree:



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

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

[2]The original posting of these nodes specified that a root is not an internal node.

Trees and recursion

Many operations on trees can be done recursively. Indeed one can even give a recursive definition of a (rooted) tree: A tree T is a collection of nodes. If T is not empty, then there is a unique root node r and k non-empty (sub)trees T_1 to T_k whose roots r_i are children of r, i.e. (r, r_i) is an edge in T.

Let's next look at several recursive algorithms on rooted trees.

Computing depth and height of a node

These were discussed in class. See the textbook p. 275 for more.

```
depth(v, T){
  if (v is the root of T)
    return 0
  else
    return 1 + depth(parent(v), T)
}

height(v, T){
  if (v is a leaf)
    return 0
  else{
    h = 0;
    for each child w of v
       h = max(h, height(w,T));
    return 1 + h;
  }
}
```

Pre-order traversal

Often we would like to examine and possibly carry out operations on all the nodes of the tree. It is common to refer to the examination and operations as "visiting" the node. Visiting all the nodes in the tree in some order is called *traversing* the tree.

How do we choose the order? There are several recursive ways we can do this. Here are three common ones.

The first is a pre-order traversal. It first visits a node, and then visits all its children.

```
PreorderTraversal(tree, root){
  visit root
  for each child
    PreorderTraversal(tree, root.child)
}
```

Note that any node v in a tree can be considered as the root node of a subtree, namely a subcollection of nodes (and a subset of edges of the original tree) whose root node is v. This idea of a *sub-tree* will come up again (many times).

Post-order traversal

A post-order traversal delays visiting a node until after it has visited all its children.

```
PostOrderTraversal(tree, root){
  for each child of root
    PreorderTraversal(tree, root.child){
  visit root
```

Level-order traversal (also called "breadth first")

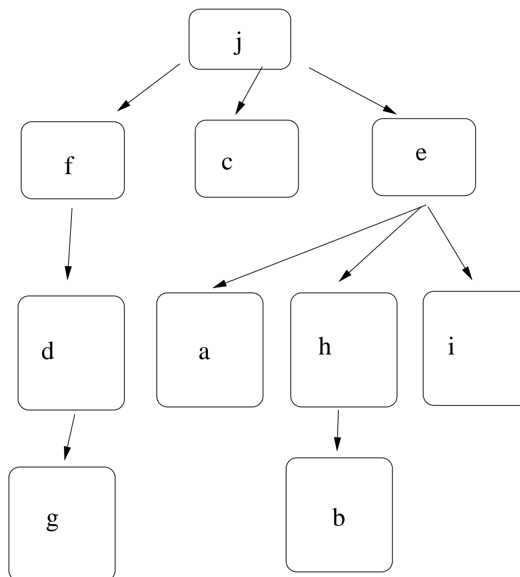
The third traversal method is not written recursively.

```
  for i = 0 to depth(tree)
     visit all nodes at level i
```

It is sometimes called "breadth first" since you descend the tree in levels and finish each level before going down to the next. This is in contrast to "depth first" where you plunge straight down to the leaves. Both pre-order and post-order are considered depth first.

Example

In the example below, assume the children of each node are ordered from left to right.



Here is the ordering of nodes visited using the various traversal methods:
Level-order: j, f, c, e, d, a, h, i, g, b
Pre-order: j, f, d, g, c, e, a, h, b, i
Post-order: g, d, f, c, a, b, h, i, e, j