

(Rooted) Tree ADT

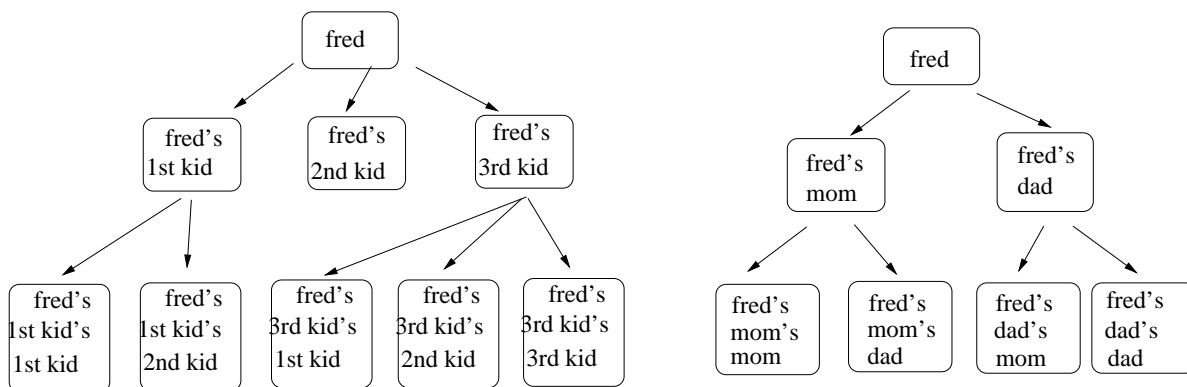
Thus far we have been working with “linear” collections, namely *lists*. For each element, it made sense to talk about the previous element (if it exists) and the next element (if it exists). We saw two data structures for representing lists, namely linked lists and arrays. Both have limitations, though, in that certain common operations are $O(n)$. For linked lists, finding the i^{th} element is $O(n)$. For arrays, add an element at the i^{th} position is $O(n)$, as is removing the element at the i^{th} position if we require that the array doesn’t have any holes.

To get around these limitations, one often organizes a collection of items 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. What is new 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, as a McGill professor in the School of Computer Science, I report to my department Chair, who reports to 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 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, 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.



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

`/home/perception/langer/.public_html/250/17 - trees.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 many other children. (Type “`ls /`” on a unix/linux command line to see the other sub-directories of the root directory.)

Terminology

Here is some terminology to get us started on trees:

- *node* (or *vertex*) - used in the same way as as with lists: we have a set of things (people, dogs, strings, etc), and we associate one node with each thing.
- *edge* - an *ordered* pair of nodes of the form $(parentnode, childnode)$.
- (*rooted*) *tree* - a set of nodes such that:
 - if the set is empty, we have an *empty tree*.
 - if the set is non-empty, we have a *non-empty (rooted) tree*. In this case, there unique node called the *root node*.
 - Every node other than the root node has a unique parent, that is, for every (non-root) node there is a *unique* edge $(v.parent, v)$.

It follows immediately that a tree with n nodes has $n - 1$ edges, i.e. each node except the root has exactly one parent.

- *sub-tree* - every node in a tree defines a subtree, namely the tree defined by this node and all its children. (Trivial fact: any tree is a subtree of itself.)

One can 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 .

- *sibling relation*: two nodes are siblings if they have the same parent.
- *leaf*: a node with no children, that is, a node v such that there does *not* exist a node w with $v = w.parent$. 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 UNIX 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).

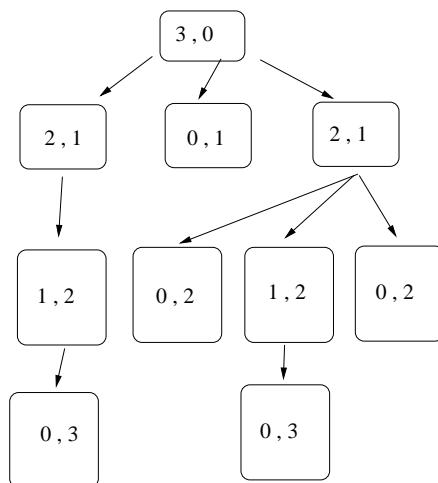
- *path*: a sequence of edges v_1, v_2, \dots, v_k where v_i is the parent of v_{i+1} , and $k \geq 1$.

- *length of a path*: the number of edges in the path. If a path has k vertices, 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 e.g. proofs by induction).

- *depth* of a node (also called *level* of the node): the length of the (unique) path from the root to the node. 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*: w is a descendent of v if there is a path from w to v . Note that “ v is an ancestor of w ” is equivalent to “ w is a descendent of v ”.

Also note that v is an ancestor/descendent of v (itself) since there is a path of length 0 from v to v . We only care about this case when proving certain mathematical statements about trees, such as “every ancestor of a node bla bla blab”. If you want to make such statements but not consider the case just mentioned, then instead you say every *proper ancestor* bla bla bla (meaning every ancestor except the node itself).

Trees and recursion

It is quite common to define operations on trees recursively. Here are a few common examples.

```

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

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 implementations

Let's briefly turn to the issue of how to implement trees. The main tricky issue is how to represent the set of children of a node. If a node can have at most two children (called a binary tree – in an upcoming lecture) then each node can be given exactly two reference variables to point to these two children.

If, however, a node can have many children, then it is less clear what to do. The most common representation is to use a linked list to represent the children of a node. This is typically called the “*first child, next sibling*” implementation of a tree.

```

class  TreeNode<T>{
  T          element;
//  TreeNode<T>  parent;          // this field is optional
  TreeNode<T>  nextSibling;
  TreeNode<T>  firstChild;
}

```

This implementation defines a singly linked list for the siblings, and either a singly or doubly linked list for the parent/first-child relationship. The parent/first-child is singly linked if only the `firstChild` field is defined, and it is doubly linked if also the `parent` field is defined. Note that to define the `depth()` method as done earlier, we would need to have a `parent` field.

If you were writing your own tree class in Java, you might prefer to write something like this:

```

class  TreeNode<T>{
  T          element;
  TreeNode<T>  parent;          // this field is optional
  LinkedList<TreeNode<T>>  children;
}

```

Alternatively, you could use an `ArrayList`

```
class  TreeNode<T>{
    T          element;
    TreeNode<T>  parent;           //  this field is optional
    ArrayList<TreeNode<T>>  children;
}
```

Either of these works fine, and is likely easier to program since you get all the `List` methods for free (that is, all the `LinkedList` or `ArrayList` methods). The only disadvantages I can think of are that:

- for `ArrayList`, it is slower to add or insert nodes, so if you are doing a lot of adding and inserting then the code might be slower than if you use a linked list
- for `LinkedList` which uses a doubly linked list, you may be wasting memory (namely if you don't need the internal "prev" reference at each node).