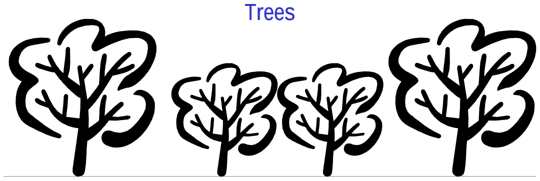
CSE 143 Notes 5/15/06

Trees



Overview

- Topics
  - Trees: Definitions and terminology
  - Binary trees
  - Tree traversals
  - Recursive tree algorithms

Trees

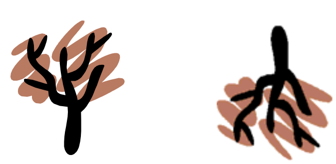
- Most of the structures we've looked at so far are linear
  - Arrays
  - Linked lists
- There are many examples of structures that are not linear
  - Organization charts
  - Book contents (chapters, sections, paragraphs)
  - Class inheritance diagrams
- *Trees* can be used to represent hierarchical structures

Looking Ahead To An Old Goal

- Finding algorithms and data structures for fast searching
  - Sorted arrays are faster than unsorted arrays, for searching
    - Can use binary search algorithm
    - Not so easy to keep the array in order
  - LinkedLists were faster than arrays (or ArrayLists), for insertion and removal operations
    - The extra flexibility of the "next" pointers avoided the cost of sliding
    - But... LinkedLists are hard to search, even if sorted
- Is there a way to get the best of both worlds?
- The answer will be...Yes: a particular type of tree

Drawing Trees

- For whatever reason, computer scientists usually draw trees upside down with the root at the top



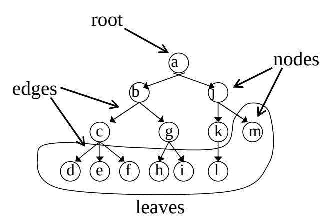
Tree Definitions (1)

- A *tree* is a collection of *nodes* connected by *edges*
- A *node* contains
  - Data (e.g. an int, an Object, or whatever we want)
  - References (edges) to two or more *subtrees* or *children*
- Equivalently: a tree is either
  - An empty tree, or
  - A root node with left and right subtrees
- Both definitions are recursive: the first focuses on the implementation (nodes, edges), while the second is a bit more abstract (trees & subtrees)
  - We'll look at trees both ways depending on the situation
  - Often we will use this structure to help formulate algorithms and analysis (recursive data ⇔ recursive algorithms)

Tree Definitions (2)

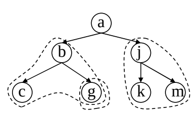
- Trees are hierarchical
  - A node is said to be the *parent* of its *children* (subtrees)
    - We can also speak of the collection of *ancestors* (parent, grandparent, …) and *descendants* (children, grandchildren) of a node
  - There is a single unique *root* node that has no parent
  - Nodes with no children are called *leaf nodes*
    - Nodes with one or more children are often called *interior* nodes
  - A tree with no nodes is said to be *empty*

Tree Terminology



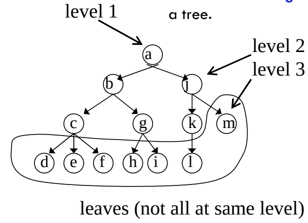
Subtrees

- A *subtree* in a tree is any node in the tree together with all of its descendants (its children, and their children, recursively)



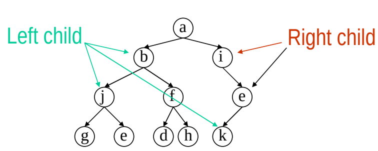
Level and Height

Definition: The root has **level** 1
Children have level 1 greater than their parent
Definition: The **height** is the highest level of any node in a tree.



Binary Trees

- A *binary tree* is a tree each of whose nodes has no more than two children
  - The two children are called the *left child* and *right child*
  - The subtrees rooted at those children are called the *left subtree* and the *right subtree*



Binary Tree Nodes

- A node for a binary tree holds some sort of data and references to its subtrees
  - For example, tree nodes to hold a integer values

```
class TreeNode {
    public int data;              // data item in this node
    public TreeNode left;         // left subtree, or null if none
    public TreeNode right;        // right subtree, or null if none
    public TreeNode(int data, TreeNode left, TreeNode right) { … }
}
```

Binary Tree Implementation

- A collection that uses a tree as its underlying data structure normally just needs a single instance variable pointing to the root node, or null if the tree is empty
  - (The fact that a tree is the underlying data structure is usually a private detail, just as the use of an array or linked list is private in a list structure)

```
// collection of integers
public class IntCollection {
    private TreeNode root;             // root of tree, or null if empty
    public IntCollection( ) { root = null; }
    …
}
```


Tree Algorithms

- The definition of a tree is naturally recursive:
  - A tree is either null (empty),
    or data + left (sub-)tree + right (sub-)tree
  - Base case(s)?
  - Recursive case(s)?
- Given a recursively defined data structure, recursion is often a very natural technique for algorithms on that data structure
  - Don't fight it!

A Typical Tree Algorithm: nPositive( )

```
public class IntCollection {
    …
    /** Return the number of positive (>0) ints stored in this tree */
    public int nPositive( ) {
        return nPositive(root);
    }
    // Return the number of nodes with positive ints in the (sub-)tree with root r
    private int nPositive(TreeNode r) {
        if (r == null) {
            return ___________________ ;
        } else {

        }
    }
```


Tree Traversal

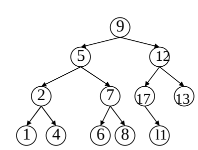
- Functions like nPositive systematically "visit" each node in a tree
  - This is called a *traversal*
  - We also used this word in connection with lists
- Traversal is a common pattern in many algorithms
  - The processing done during the "visit" varies with the algorithm
- What order should nodes be visited in?
  - Many are possible
  - Three have been singled out as particularly useful for binary trees: *preorder*, *postorder*, and *inorder*

Traversals

- Preorder traversal:
  - "Visit" the (current) node first
    i.e., do whatever processing is to be done
  - Then, (recursively) do preorder traversal on its children, left to right
- Postorder traversal:
  - First, (recursively) do postorder traversals of children, left to right
  - Visit the node itself last
- Inorder traversal:
  - (Recursively) do inorder traversal of left child
  - Then visit the (current) node
  - Then (recursively) do inorder traversal of right child

Example of Tree Traversal

In what order are the nodes visited, if we start the process at the root?



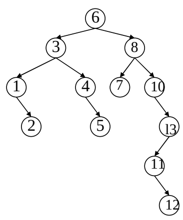
Preorder:

Inorder:

Postorder:

More Practice

What about this tree?



Preorder:

Inorder:

Postorder:

New Algorithm: *contains*

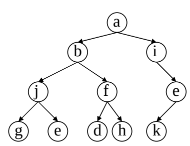
- Return whether or not a value is in the tree

```
public class IntCollection {
    …
    /** Return whether n is in the tree */
    public boolean contains(int n) {
        return contains(root, n);
    }
    // Return whether n is in (sub-)tree with root r
    private boolean contains(TreeNode r, int n) {
        if (r == null) {
            return ________________ ;
        } else if (r.data == n) {
            return ________________ ;
        } else {
            return ______________________________________________ ;
        }
    }
```


Test

contains(d)

contains(c)



Cost of *contains*

- Work done at each node:
- Number of nodes visited:
- Total cost:
- Can we do better?