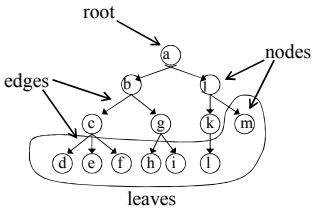
CSC 143 Java

Trees

Tree Definitions

- A *tree* is a collection of *nodes* connected by *edges*
- A *node* contains
  - Data (e.g. an Object)
  - References (edges) to two or more *subtrees* or *children*
- Trees are hierarchical
  - A node is said to be the *parent* of its *children* (subtrees)
  - There is a single unique *root* node that has no parent
  - Nodes with no children are called *leaf nodes*
  - A tree with no nodes is said to be *empty*

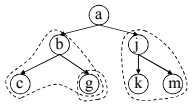
Tree Terminology



Trees usually drawn upside-down, from root to leaves

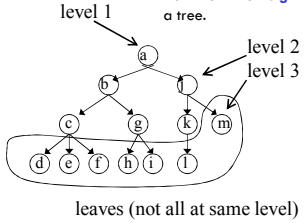
Subtrees

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



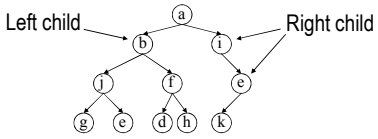
Level and Height

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 belonging to those children are called the *left subtree* and the *right subtree*



Binary Tree Nodes

- A node for a binary tree holds the item and references to its subtrees

```
class BTNode<E> {
    E item;             // data item in this node
    BTNode left;        // left subtree, or null if none
    BTNode right;       // right subtree, or null if none

    BTNode(Object item, BTNode left, BTNode right) { … }
}
```

  - Where could we put this class definition?
  - What would BinaryTree class instance variable(s)/constructor look like?
  - What operations on a BinaryTree might there be (add, remove, query)?

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: size( )

```
// within BinTree
    /** Return the number of items in this tree */
    public int size( ) {
        return subtreeSize(root);
    }
    // Return the number of nodes in the (sub-)tree with root n
    private int subtreeSize(BTNode<E> n) {
        if (n == null)
            return _____ ;
        else
            return _____ ;
    }
```

What is runtime cost of this algorithm?

Exercises

Do try these at home!

1. Print all the contents of the binary tree (BT)
2. Find the sum of all the values in a BT of integers
3. Find the smallest value in a BT of integers
4. (A little harder) Find the average of all the values in a BT (one approach: think in terms of a "kickoff" method)

Tree Traversal

- Functions like subtreeSize 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***

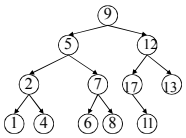
Traversals

- Preorder traversal:
  - "Visit" the (current) node first
    i.e., do what ever 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

    Footnote: pre- and postorder make sense for all trees; inorder only for binary trees

Example of Tree Traversal

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



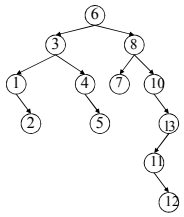
Preorder:

Inorder:

Postorder:

More Practice

What about this tree?



Preorder:

Inorder:

Postorder:

Runtime analysis?

New Algorithm: *contains*

Return whether or not a value is an item in the tree

```
// within class BinTree

/** Return whether elem is in tree */
public boolean contains(Object elem) {
    return subtreeContains(root, elem);
}

// Return whether elem is in (sub-)tree with root r
private boolean subtreeContains(BTNode<E> r, Object elem) {
    if (r == null)
        return _____;
    else if (r.item.equals(elem))
        return _____;
    else
        return _____
}
```

Let's trace this on a tree..

Cost of *contains*

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