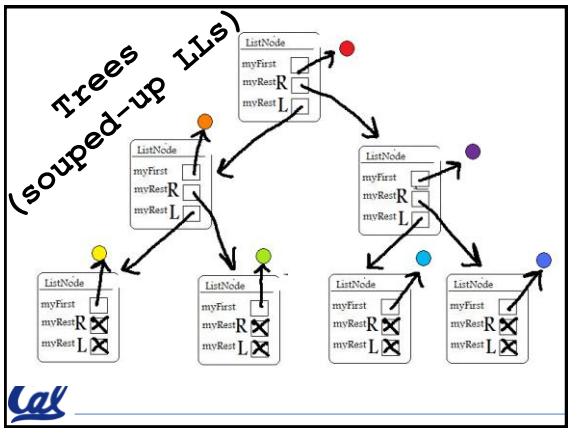
Trees

7/14/2009

Important Dates

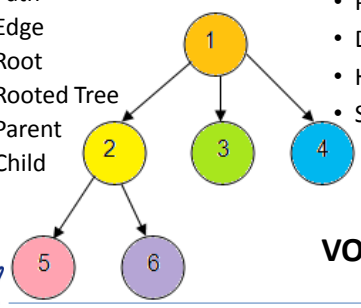
- Project 2 due
  - DESIGN, DESIGN, DESIGN!!!
  - Friday 7/24/2009 – 10pm
- Midterm Review
  - Saturday 7/26/2009 – 1-4pm in 306 Soda
- Midterm 2
  - Tuesday 7/28/2009 – 5-6pm in 10 Evans



- Tree
- Node
- Path
- Edge
- Root
- Rooted Tree
- Parent
- Child

- Leaf
- Sibling

- Ancestor
- Descendent
- Path Length
- Depth
- Height
- Subtree



VOCAB

Tree Definitions: Overview

- A ***tree*** consists of a *set of nodes* and a *set of edges* that connect *pairs of nodes*.
- There is *exactly one* path between any two nodes of the tree.
- A ***path*** is a connected sequence of zero or more edges.

Tree Definitions: Rooted Tree

- In a ***rooted tree***, one distinguished node is called the ***root***.
- Every node *c*, except the root, has exactly one ***parent*** *node p*, which is the first node traversed on the path from c to the root. *c* is *p*'s ***child***.
- The root has no parent.
- A node can have any number of children.

Tree Definitions: More

- A ***leaf*** is a node with no children.
- ***Siblings*** are nodes with the same parent.
- The ***ancestors*** of a node *d* are the nodes on the path from *d* to the root.
  - Technically, the ancestors of *d* also include *d* itself.
  - The root is an ancestor of every node in the tree.
- If *a* is an ancestor of *d*, then *d* is a ***descendant*** of *a*.

Definitions

- The ***length of a path*** is the number of edges in the path.
- The ***depth of a node*** *d* is the length of the path from *d* to the root.
  - The depth of the root is zero.
- The ***height of a node*** *d* is the length of the path from *d* to its deepest descendant.
  - The height of a leaf node is zero.
  - The *height of a tree* = height of the root.
- The ***subtree*** rooted at node *d* is the tree formed by *d* and its descendants.

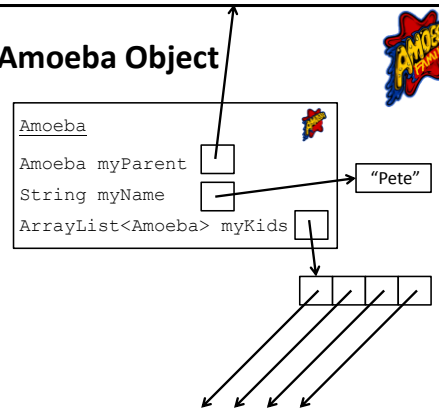
Announcements (Text)

- Please recycle your paper – there is a bin in the printer room
- Please throw away your trash
- Please use the power strips. NEVER use the floorboards!!!!

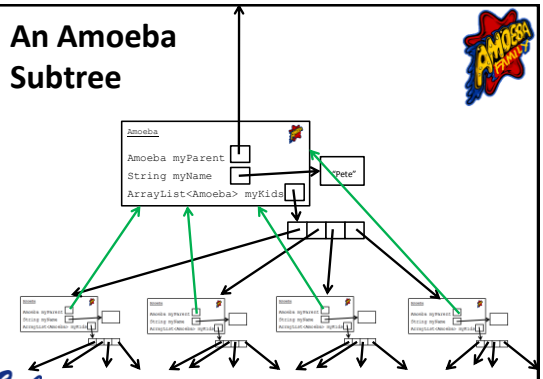
```
public class AmoebaFamily {

    private Amoeba myRoot = null;

    // A constructor that starts an
    // Amoeba family with an amoeba
    // with the given name.
    public AmoebaFamily (String name) {
        myRoot = new Amoeba (name, null);
    }
```

```
private static class Amoeba {

    public String myName;
    public Amoeba myParent;
    public ArrayList<Amoeba> myChildren;

    public Amoeba (String name, Amoeba parent) {
        myName = name;
        myParent = parent;
        myChildren = new ArrayList<Amoeba> ( );
    }

    // Add an amoeba with the given name as
    // this amoeba's youngest child.
    public void addChild (String childName) {
        Amoeba child = new Amoeba (childName, this);
        myChildren.add (child);
```

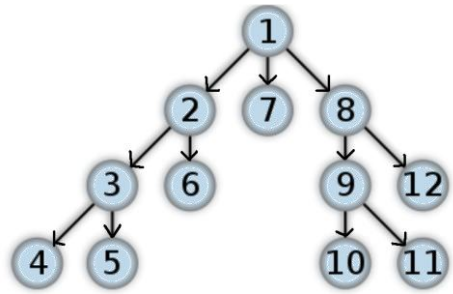
One Amoeba Object



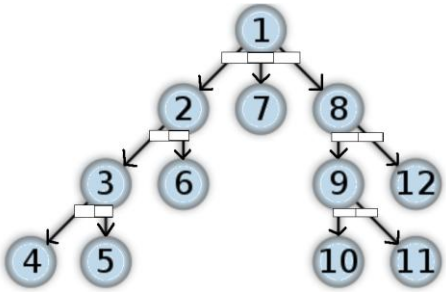
An Amoeba Subtree



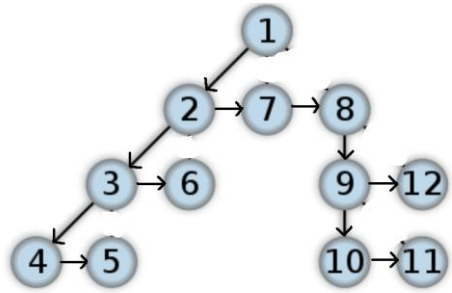
Design 1 (How we think about it)



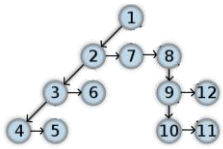
Design 2 (Amoeba)



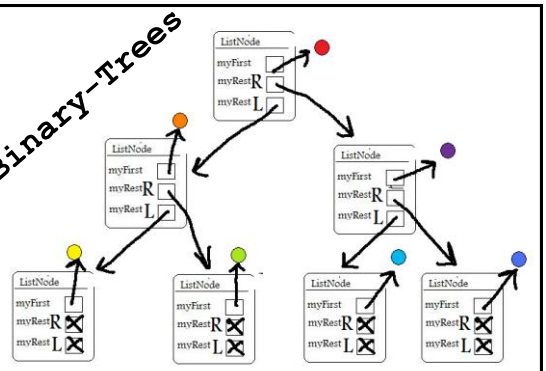
Design 3 – LinkedList of Children




```
public class SibTree {
    SibTreeNode root;
    int size;

    private class SibTreeNode {
        Object item;
        SibTreeNode parent;
        SibTreeNode firstChild;
        SibTreeNode nextSibiling;
```

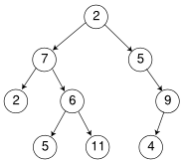


Binary-Trees



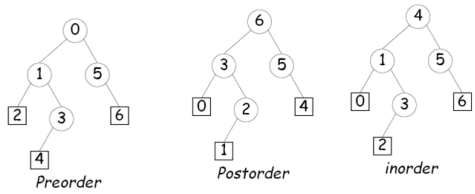
Binary Tree

- A binary tree is a tree in which no node has more than two children, and every child is either a ***left child*** or a ***right child***, even if it's the only child its parent has.



Traversal

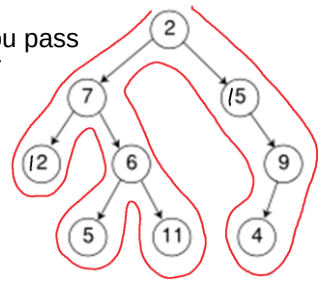
- **Preorder:** visit node, traverse its children.
- **Postorder:** traverse children, visit node.
- **Inorder:** traverse first child, visit node, traverse second child (binary trees only).




```
private static class Amoeba {

    public String myName;
    public Amoeba myParent;
    public ArrayList<Amoeba> myChildren;

    public Amoeba (String name, Amoeba parent) {
        myName = name;
        myParent = parent;
        myChildren = new ArrayList<Amoeba> ( );
    }

    public void [                    ] {
        System.out.println(myName);
        for(Amoeba child : myChildren){
            child.printPreOrder();
        }
    }
```

Preorder: visit node, traverse its children.

As you pass LEFT



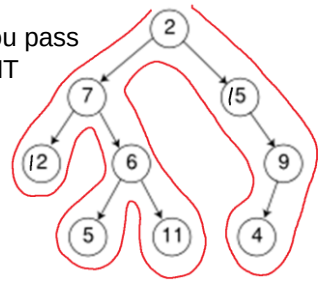
2 → 7 → 12 → 6 → 5 → 11 → 15 → 9 → 4

"PRE" => start with "ME"

```
private static class Amoeba {

    public String myName;
    public Amoeba myParent;
    public ArrayList<Amoeba> myChildren;

    public Amoeba (String name, Amoeba parent) {
        myName = name;
        myParent = parent;
        myChildren = new ArrayList<Amoeba> ( );
    }

    public void [                    ] {
        for(Amoeba child : myChildren){
            child.printPreOrder();
        }
        System.out.println(myName);
    }
```

Postorder: traverse children, visit node

As you pass RIGHT



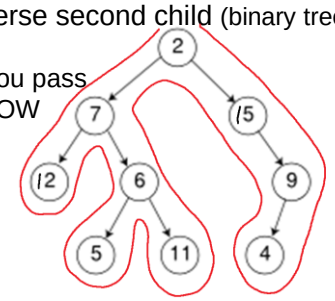
12 → 5 → 11 → 6 → 7 → 4 → 9 → 15 → 2

"POST" => be a good "HOST"

```
private static class Amoeba {
    public Amoeba myLeft;
    public Amoeba myRight;

    public String myName;
    public Amoeba myParent;

    public void printInOrder()
    {
        if (myLeft != null){
            myLeft.printInOrder();
        }
        System.out.println(myName);
        if (myRight != null)
        {
            myLeft.printInOrder();
        }
    }
```

Inorder: traverse first child, visit node, traverse second child (binary trees only).

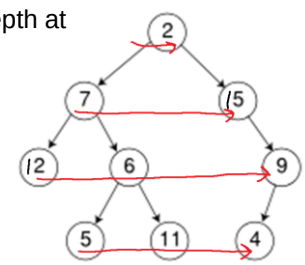
As you pass BELOW



12 → 7 → 5 → 6 → 11 → 2 → 15 → 4 → 9

Level Order: Breadth First

One depth at a time



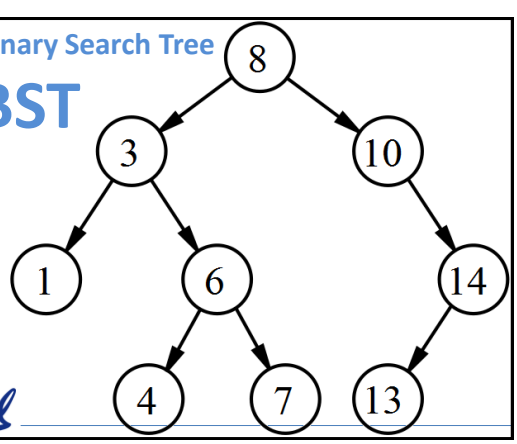
2 → 7 → 15 → 12 → 6 → 9 → 5 → 11 → 4

Level-Order (Breadth-First) Traversal

- Traverse all nodes at depth 0, then depth 1…
- Unlike the other traversals, this algorithm is not naturally recursive
- Use a queue, which initially contains only the root. Then repeat the following steps:
  - Dequeue a node from the front of the list.
  - Visit it.
  - Enqueue its children (in order from left to right). Continue until the queue is empty.

Binary Search Tree

BST

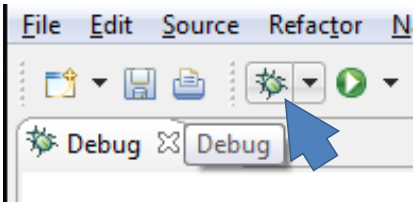


Binary Search Tree

- Binary Search Tree is a binary tree.
- Generally, each node of the Binary Search Tree contains an <Key, Value> pair called an Entry.
  - The key can be the same as the value.
- Binary Search Tree satisfies the following property, called the *binary search tree invariant*: For any node X,
  - every key in the *left* subtree of X is *less than or equal* to X's key, and
  - every key in the *right* subtree of X is *greater than or equal* to X's key.
  - A key equal to the parent's key can go into either subtree.
- Example.

Inserting

- `insert()` starts by following the same path through the tree as `find()`.
- When it reaches a `null` reference, replace the `null` with a reference to a new node <Key, Value> .
- Duplicate keys are allowed.
  - If `insert()` finds a node that already has the key, it puts it the new entry in the left subtree of the older one.
  - We could just as easily choose the right subtree; it doesn't matter.



Breakpoints | Variables

Name	Value
ars	String[0] (id=16)
n1	NonemptyListNode (id=...
myFirst	"A" (id=21)
myRest	EmptyListNode (id=25)

Breakpoints | Variables

Name	Value
ars	String[0] (id=16)
n1	NonemptyListNode (id=...
myFirst	"A" (id=22)
myRest	EmptyListNode (id=26)
n2	NonemptyListNode (id=...
myFirst	"A" (id=22)
myRest	NonemptyListNode (id=...
myFirst	"B" (id=29)
myRest	EmptyListNode (id=30)

Breakpoints | Variables

Name	Value
ars	String[0] (id=16)
n1	NonemptyListNode (id=...
myFirst	"A" (id=22)
myRest	EmptyListNode (id=26)
n2	NonemptyListNode (id=...
myFirst	"A" (id=22)
myRest	NonemptyListNode (id=...
myFirst	"B" (id=29)
myRest	EmptyListNode (id=30)
n3	NonemptyListNode (id=...
myFirst	"A" (id=22)
myRest	NonemptyListNode (id=...
myFirst	"B" (id=29)
myRest	NonemptyListNode (id=...
myFirst	"C" (id=37)
myRest	EmptyListNode (id=38)

Debugging

- Your TA will help you LEARN to use the debugger!
  - Your TA will not debug your code! SORRY!
- Your FRIENDS will help you learn to use the debugger AND help you debug your code!!!
  - AWESOME!