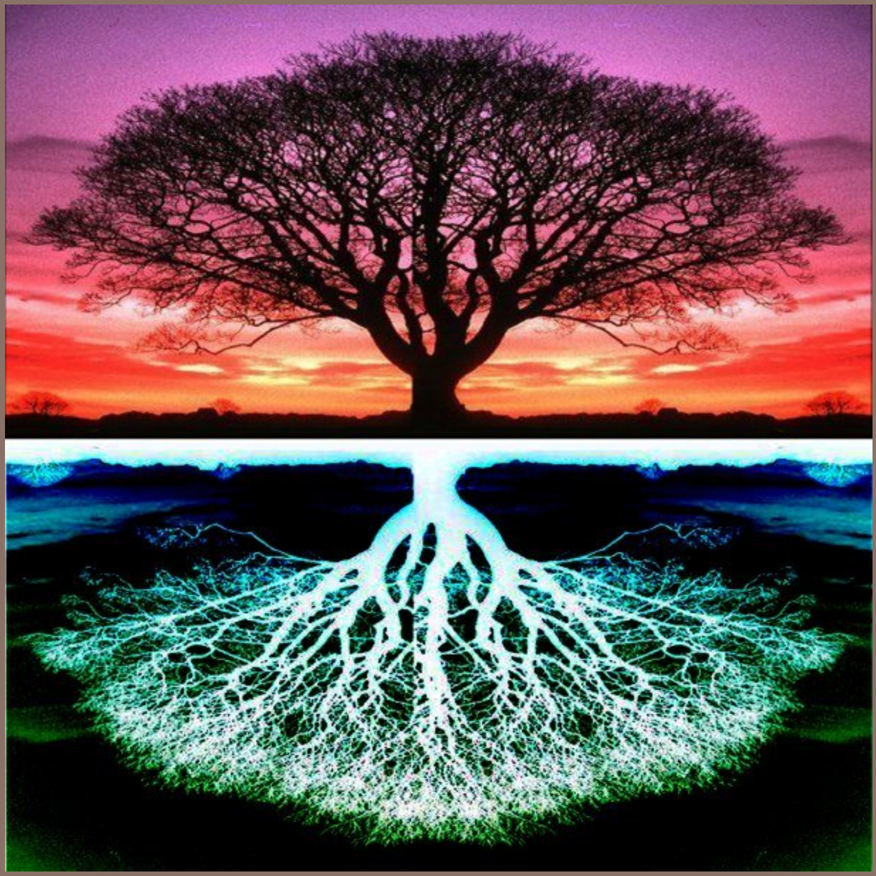


TREES

Lecture 12

CS2110 – Fall 2015

Announcements

- Prelim #1 is tonight!
  - Olin 155
  - A-L → 5:30
  - M-Z → 5:30
- A4 will be posted today
- Mid-semester TA evaluations are coming up; please participate! Your feedback will help our staff improve their teaching.

Outline

- A4 Preview
- Introduction to Trees

Readings and Homework

- Textbook, Chapter 23, 24

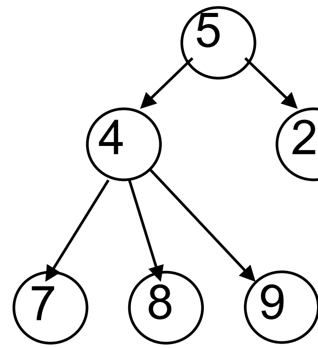
- Homework: A thought problem (draw pictures!)
  - Suppose you use trees to represent student schedules. For each student there would be a general tree with a root node containing student name and ID. The inner nodes in the tree represent courses, and the leaves represent the times/places where each course meets. Given two such trees, how could you determine whether and where the two students might run into one-another?

Tree Overview

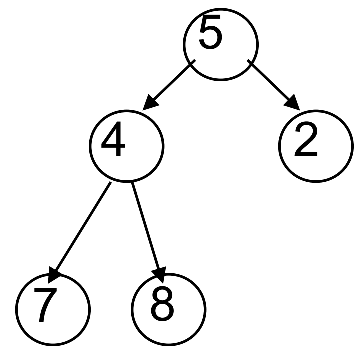
Tree: recursive data structure (similar to list)

- Each node may have zero or more *successors* (children)
- Each node has exactly one *predecessor* (parent) except the *root*, which has none
- All nodes are reachable from *root*

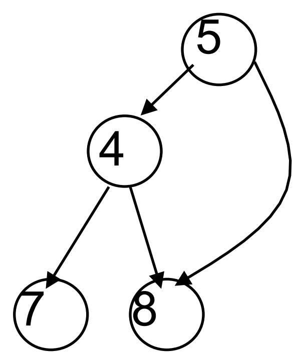
Binary tree: tree in which each node can have at most two children: a left child and a right child



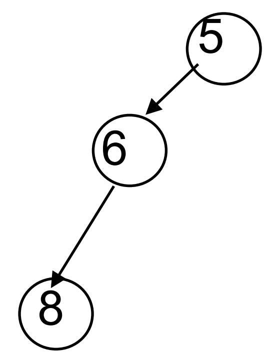
General tree



Binary tree



Not a tree

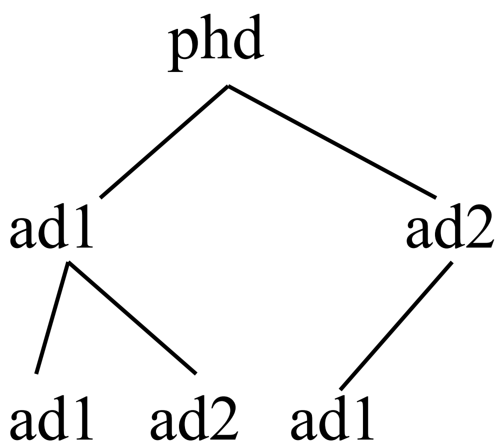


List-like tree

Binary trees were in A1!

You have seen a binary tree in A1.

A PhD object phd has one or two advisors.
Here is an intellectual ancestral tree!



Tree terminology

M: *root* of this tree

G: *root* of the *left subtree* of M

B, H, J, N, S: *leaves (their set of children is empty)*

N: *left child* of P; S: right *child* of P

P: parent of N

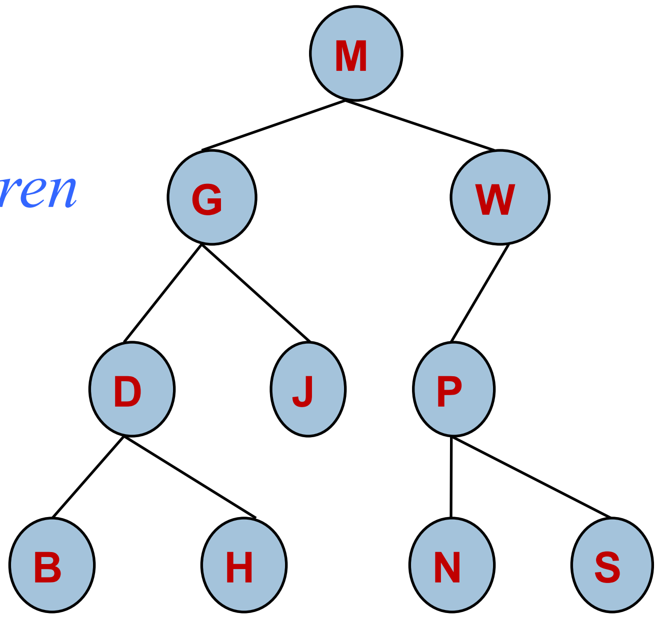
M and G: *ancestors* of D

P, N, S: *descendents* of W

J is at *depth* 2 (i.e. length of path from root = no. of edges)

W is at *height* 2 (i.e. length of longest path to a leaf)

A collection of several trees is called a ...?



Class for binary tree node


```
class TreeNode<T> {
  private T datum;
  private TreeNode<T> left, right;

  /** Constructor: one node tree with datum x */
  public TreeNode (T d) { datum= d; }

  /** Constr: Tree with root value x, left tree l, right tree r */
  public TreeNode (T d, TreeNode<T> l, TreeNode<T> r) {
    datum= d; left= l; right= r;
  }
}
```

Points to left subtree (null if empty)

Points to right subtree (null if empty)

more methods: getDatum, setDatum, getLeft, setLeft, etc.

Binary versus general tree

In a binary tree, each node has exactly two pointers: to the left subtree and to the right subtree:

- One or both could be **null**, meaning the subtree is empty (remember, a tree is a set of nodes)

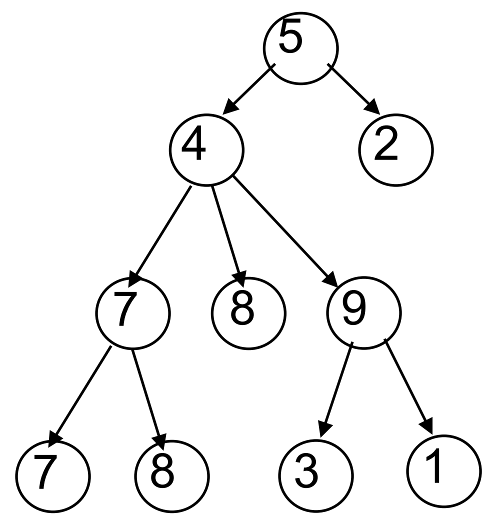
In a general tree, a node can have any number of child nodes (and they need not be ordered)

- Very useful in some situations ...
- ... one of which may be in an assignment!

Class for general tree nodes

```
class GTreeNode<T> {
    Private T datum;
    private GTreeNode<T>[] children;
    //appropriate constructors, getters,
    //setters, etc.
}
```

Parent contains an array of its children



General tree

Applications of Trees

- Most languages (natural and computer) have a recursive, hierarchical structure
- This structure is *implicit* in ordinary textual representation
- Recursive structure can be made *explicit* by representing sentences in the language as trees: Abstract Syntax Trees (ASTs)
- ASTs are easier to optimize, generate code from, etc. than textual representation
- A parser converts textual representations to AST

Use of trees: Represent expressions

In textual representation:

Parentheses show hierarchical structure

In tree representation:

Hierarchy is explicit in the structure of the tree

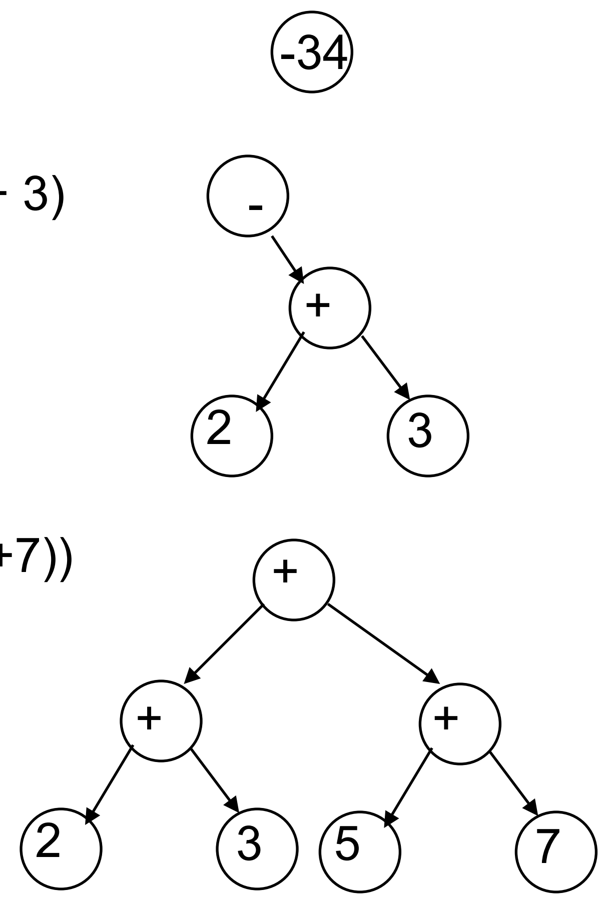
We'll talk more about expression and trees on Thursday

Text — Tree Representation

-34

- (2 + 3)

((2+3) + (5+7))



Recursion on trees

Trees are defined recursively. So recursive methods can be written to process trees in an obvious way

Base case

- empty tree   (null)
- leaf

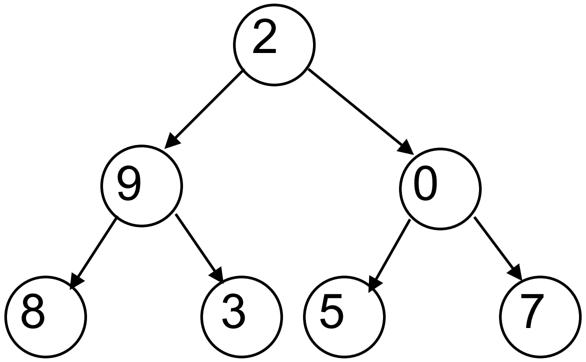
Recursive case

- solve problem on left / right subtrees
- put solutions together to get solution for full tree

Searching in a Binary Tree


```
/** Return true iff x is the datum in a node of tree  t*/
public static boolean treeSearch(T x, TreeNode<T> t) {
   if (t == null) return false;
   if (t.datum.equals(x)) return true;
   return treeSearch(x, t.left) || treeSearch(x, t.right);
}
```

- Analog of linear search in lists: given tree and an object, find out if object is stored in tree
- Easy to write recursively, harder to write iteratively



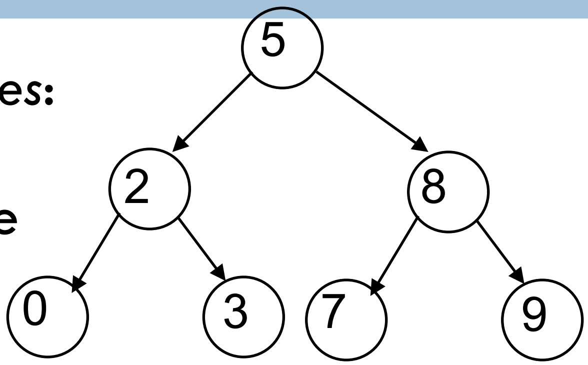
Binary Search Tree (BST)

If the tree data are *ordered and no duplicate values*:
in every subtree,

All *left* descendents of node come *before* node

All *right* descendents of node come *after* node

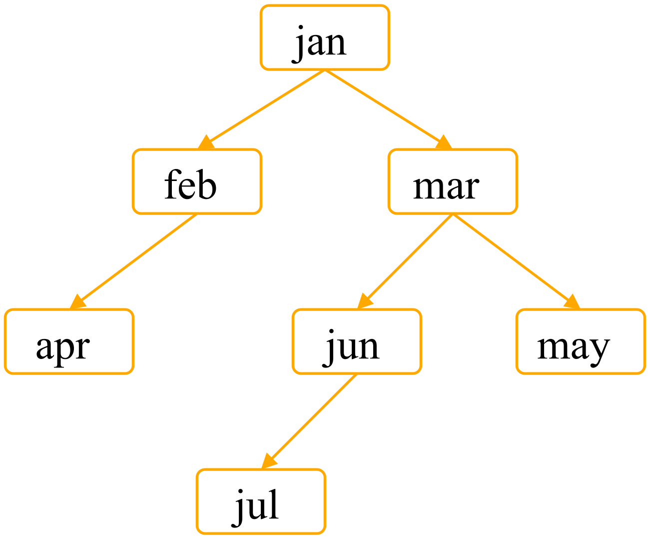
Search is MUCH faster




```
/** Return true iff x if the datum in a node of tree t.
    Precondition: node is a BST */
public static boolean treeSearch (T x, TreeNode<T> t) {
   if (t== null) return false;
   if (t.datum.equals(x)) return true;
   if (t.datum.compareTo(x) > 0)
        return treeSearch(x, t.left);
   else return treeSearch(x, t.right);
}
```

Building a BST

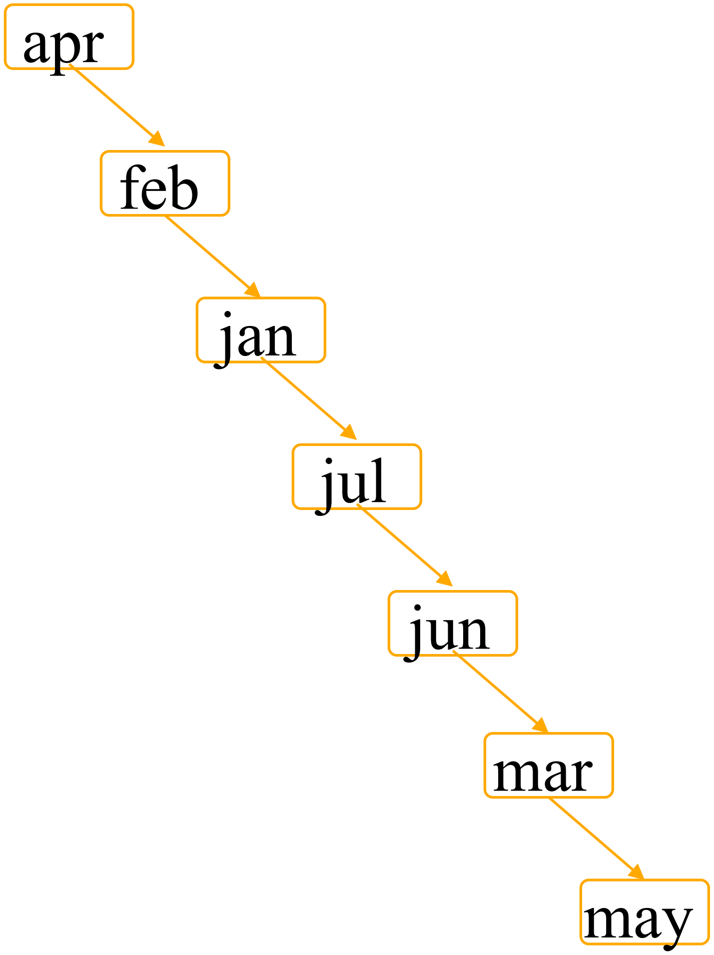
- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree
- This can be done using either recursion or iteration
- Example
  - Tree uses *alphabetical order*
  - Months appear for insertion in *calendar order*



What can go wrong?

A BST makes searches very fast, *unless*…

- Nodes are inserted in increasing order
- In this case, we're basically building a linked list (with some extra wasted space for the `left` fields, which aren't being used)



BST works great if data arrives in random order

Printing contents of BST

Because of ordering rules for a BST, it's easy to print the items in alphabetical order

- Recursively print left subtree
- Print the node
- Recursively print right subtree

```
/** Print BST t in alpha order */
private static void print(TreeNode<T> t) {
   if (t== null) return;
   print(t.left);
   System.out.print(t.datum);
   print(t.right);
}
```

Tree traversals

"Walking" over whole tree is a tree traversal

- Done often enough that there are standard names

Previous example: inorder traversal

- Process left subtree
- Process root
- Process right subtree

Note: Can do other processing besides printing

Other standard kinds of traversals

- preorder traversal
  - Process root
  - Process left subtree
  - Process right subtree
- postorder traversal
  - Process left subtree
  - Process right subtree
  - Process root
- level-order traversal
  - Not recursive uses a queue. We discuss later

Some useful methods


```
/** Return true iff node t is a leaf */
public static boolean isLeaf(TreeNode<T> t) {
   return t != null  &&  t.left == null  &&  t.right == null;
}
/** Return height of node t  (postorder traversal) */
public static int height(TreeNode<T> t) {
   if (t== null) return -1; //empty tree
   if (isLeaf(t)) return 0;
   return 1 + Math.max(height(t.left), height(t.right));
}
/** Return number of nodes  in t (postorder traversal) */
public static int nNodes(TreeNode<T> t) {
   if (t== null) return 0;
   return 1 + nNodes(t.left) + nNodes(t.right);
}
```

Useful facts about binary trees

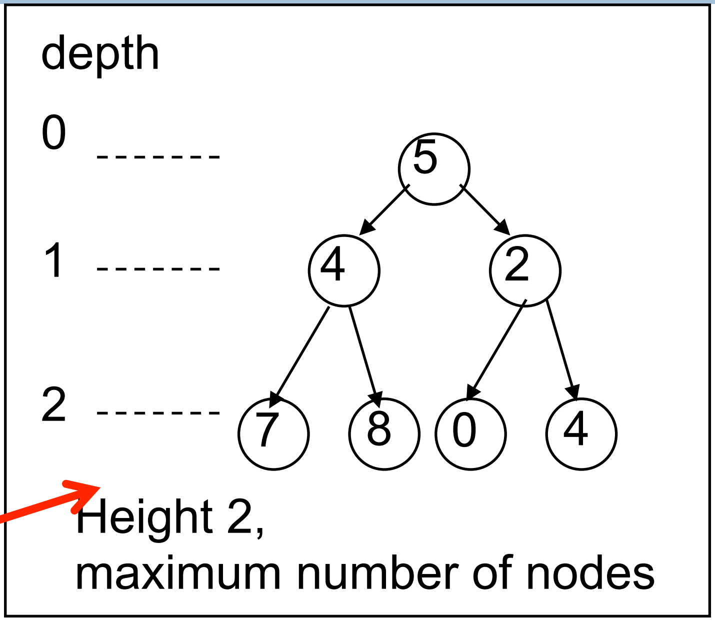
Max # of nodes at depth d: 2^d

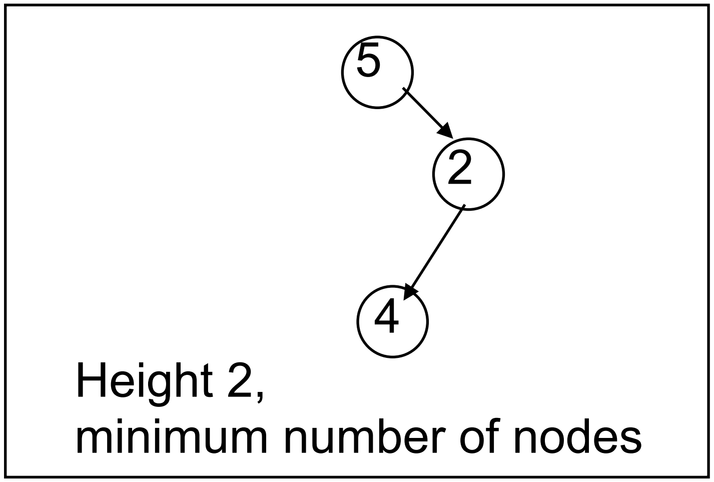
If height of tree is h

- min # of nodes: $h + 1$
- max #of nodes in tree:
- $2^0 + \ldots + 2^h = 2^{h+1} - 1$

Complete binary tree

- All levels of tree down to a certain depth are completely filled



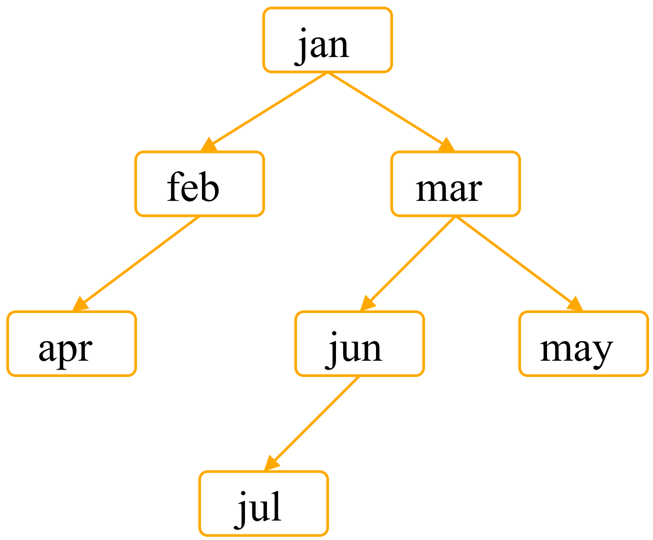


Things to think about

What if we want to *delete* data from a BST?

A BST works great as long as it's *balanced*

- How can we keep it balanced? *This turns out to be hard enough to motivate us to create other kinds of trees*



Tree Summary

- A *tree* is a recursive data structure
  - Each node has 0 or more successors (*children*)
  - Each node except the *root* has at exactly one predecessor (*parent*)
  - All node are reachable from the *root*
  - A node with no children (or empty children) is called a *leaf*
- Special case: *binary tree*
  - Binary tree nodes have a left and a right child
  - Either or both children can be empty (null)
- Trees are useful in many situations, including exposing the recursive structure of natural language and computer programs