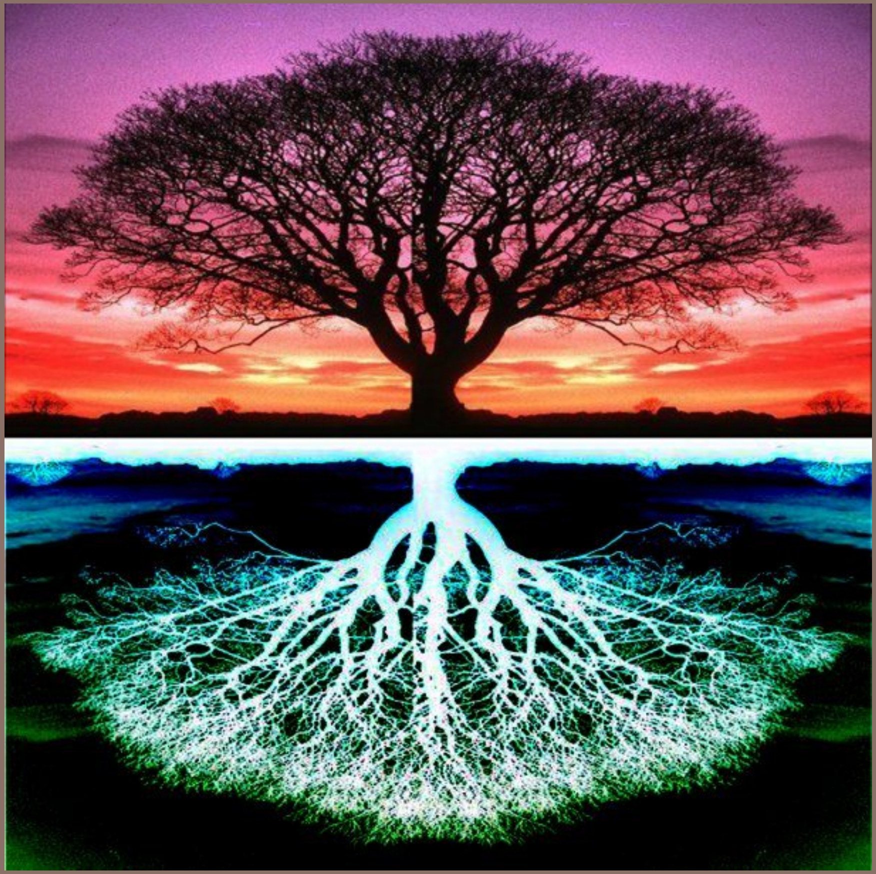


TREES

Lecture 10

CS2110 – Spring2014

Readings and Homework

- Textbook, Chapter 23, 24

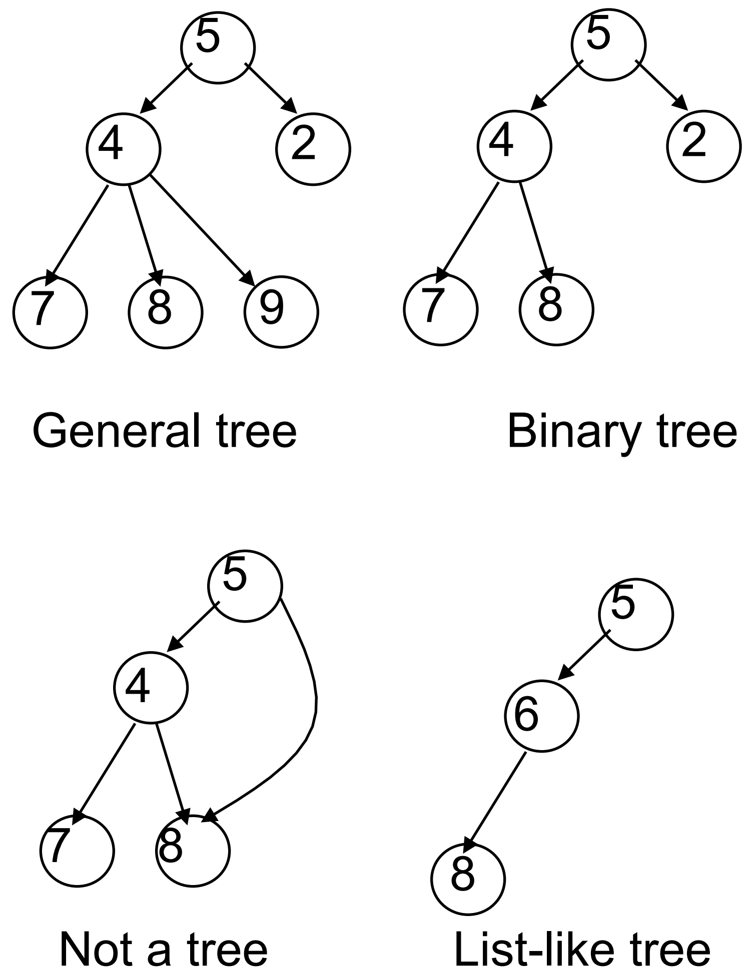
- Homework: A thought problem (draw pictures!)
  - Suppose you use trees to represent student schedules. For each student there would be a general tree with a root node containing student name and ID. The inner nodes in the tree represent courses, and the leaves represent the times/places where each 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*

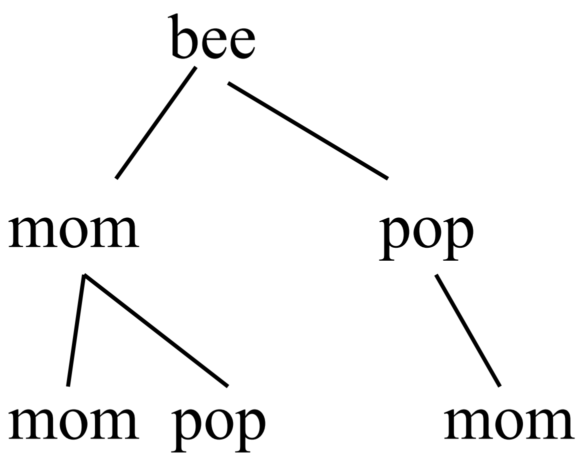
Binary tree: tree in which each node can have at most two children: a left child and a right child



Binary Trees were in A1!

You have seen a binary tree in A1.

A Bee object has a mom and pop. There is an ancestral tree!



Tree Terminology

M: *root* of this tree

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

B, H, J, N, S: *leaves*

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

P: parent of N

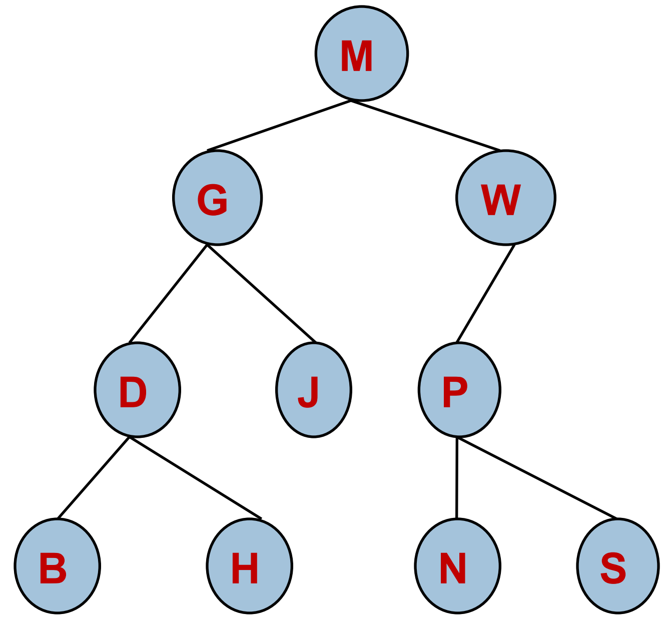
M and G: *ancestors* of D

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 <u>longest</u> 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 x) { datum= x; }

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

Points to left subtree

Points to right subtree

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

- Of course one or both could be *null*

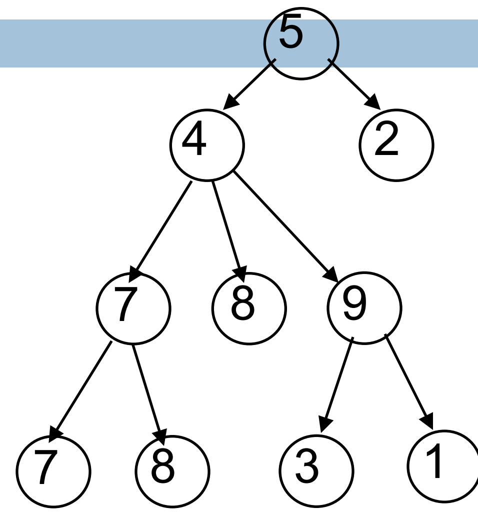
In a general tree, a node can have any number of child nodes

- Very useful in some situations ...
- ... one of which will be our assignments!

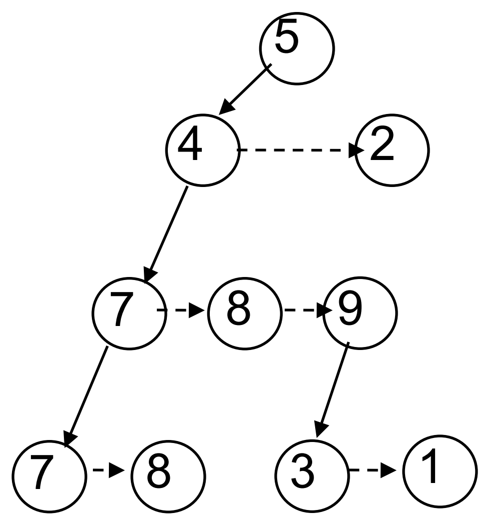
Class for General Tree nodes


```
class GTreeNode {
    private Object datum;
    private GTreeCell left;
    private GTreeCell sibling;
    appropriate getters/setters
}
```

- Parent node points directly only to its leftmost child
- Leftmost child has pointer to next sibling, which points to next sibling, etc.



General tree



Tree represented using GTreeCell

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

Example

Expression grammar:

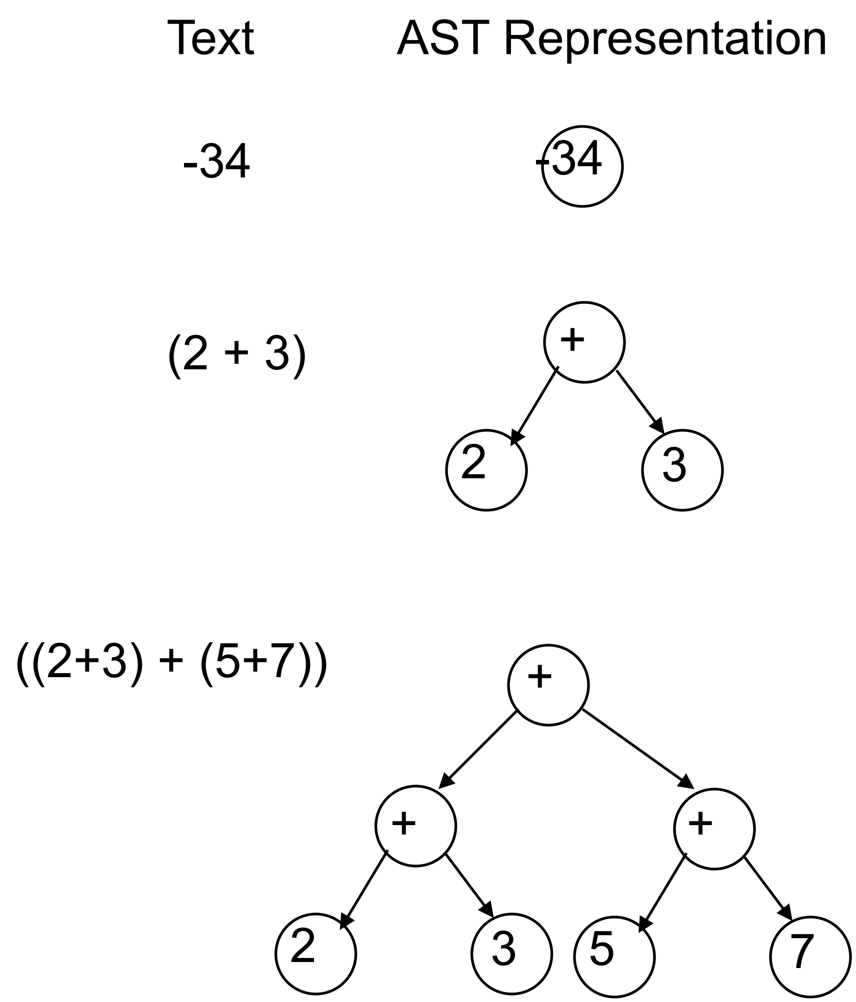
- E → integer
- E → (E + E)

In textual representation

- Parentheses show hierarchical structure

In tree representation

- Hierarchy is explicit in the structure of the tree



Recursion on Trees

Recursive methods can be written to operate on trees in an obvious way

Base case

- empty tree
- leaf node

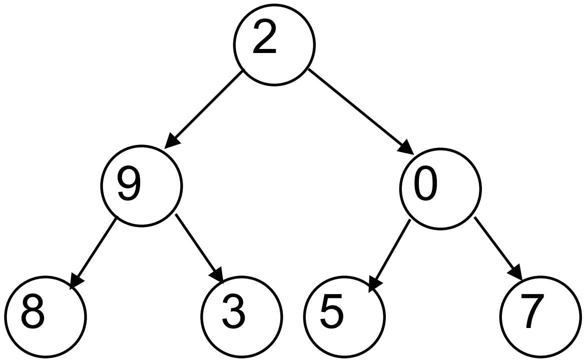
Recursive case

- solve problem on left and 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(Object x, TreeNode 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



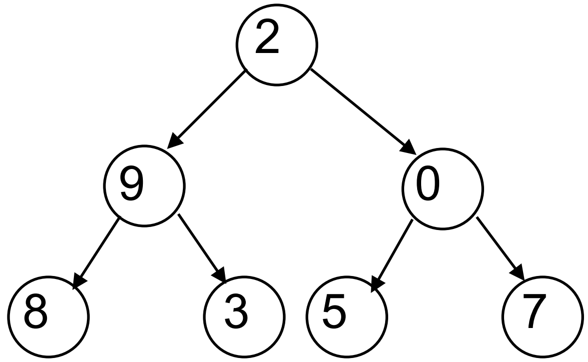
Searching in a Binary Tree


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

Important point about t. We can think of it either as

(1) One node of the tree OR

(2) The subtree that is rooted at t



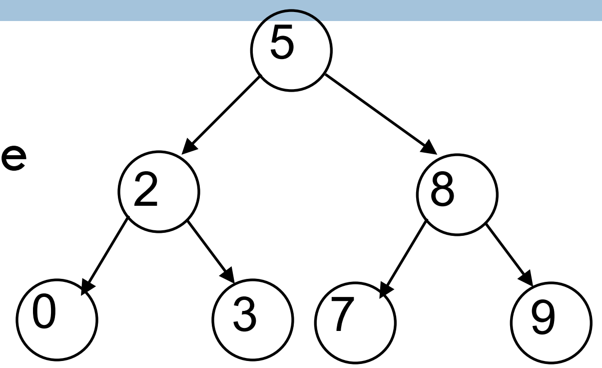
Binary Search Tree (BST)

If the tree data are *ordered*: in every subtree,

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

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

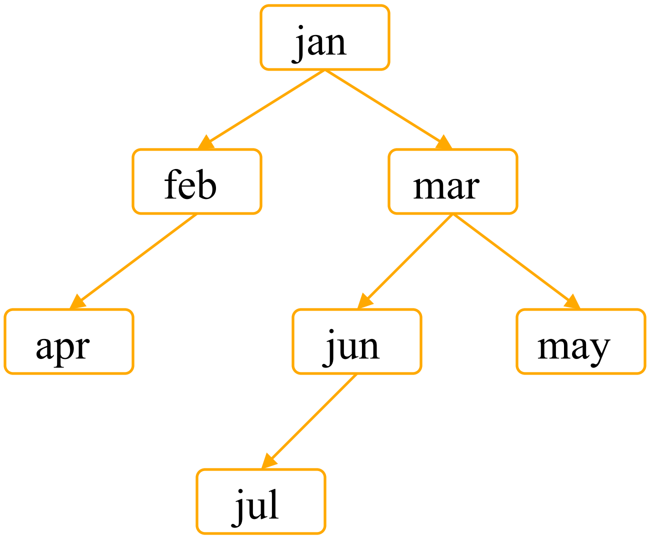
Search is MUCH faster




```
/** Return true iff x if the datum in a node of tree t.
    Precondition: node is a BST */
public static boolean treeSearch (Object x, TreeNode 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

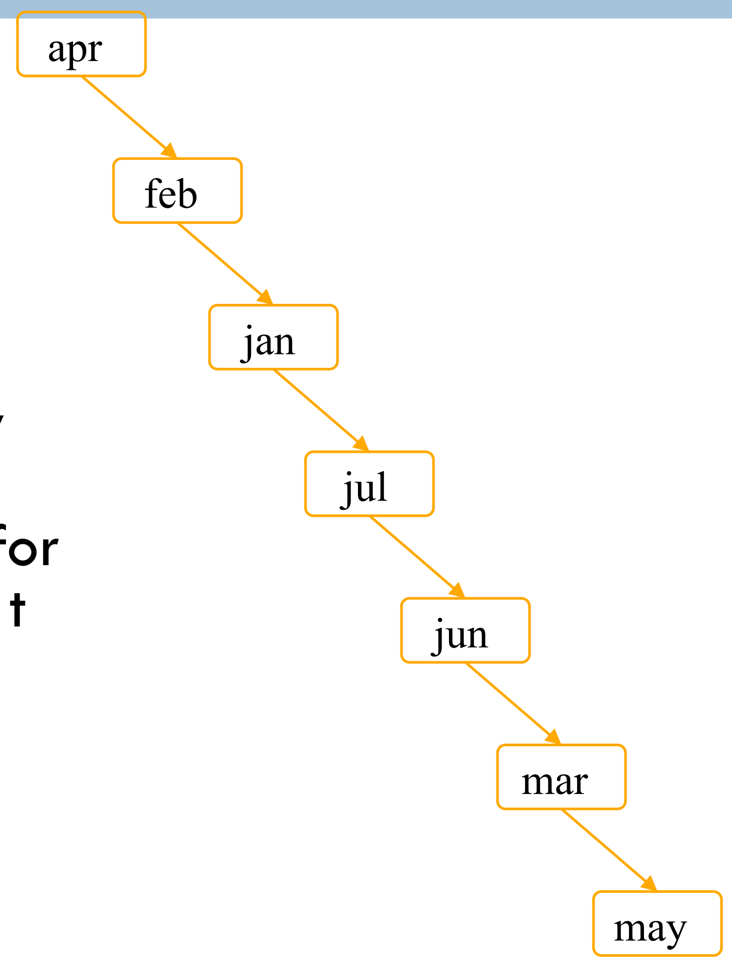
- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree
- This can be done using either recursion or iteration
- Example
  - Tree uses *alphabetical order*
  - Months appear for insertion in *calendar order*



What Can Go Wrong?

- A BST makes searches very fast, *unless*…
  - Nodes are inserted in alphabetical order
  - In this case, we're basically building a linked list (with some extra wasted space for the `left` fields that 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 the BST in alpha. order.  */
public void show () {
   show(root);
   System.out.println();
}
/** Print BST t in alpha order */
private static void show(TreeNode t) {
   if (t== null) return;
   show(t.lchild);
   System.out.print(t.datum);
   show(t.rchild);
}
```

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 node
    - Process right subtree
- Note: Can do other processing besides printing

Other standard kinds of traversals

- Preorder traversal
  - Process node
  - Process left subtree
  - Process right subtree
- Postorder traversal
  - Process left subtree
  - Process right subtree
  - Process node
- Level-order traversal
  - Not recursive uses a queue

Some Useful Methods

```
/** Return true iff node t is a leaf */
public static boolean isLeaf(TreeNode t) {
  return t!= null && t.left == null && t.right == null;
}
/** Return height of node t using postorder traversal
public static int height(TreeNode 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 using postorder traversal */
public static int nNodes(TreeNode t) {
  if (t== null) return 0;
  return 1 + nNodes(t.left) + nNodes(t.right);
}
```

Useful Facts about Binary Trees

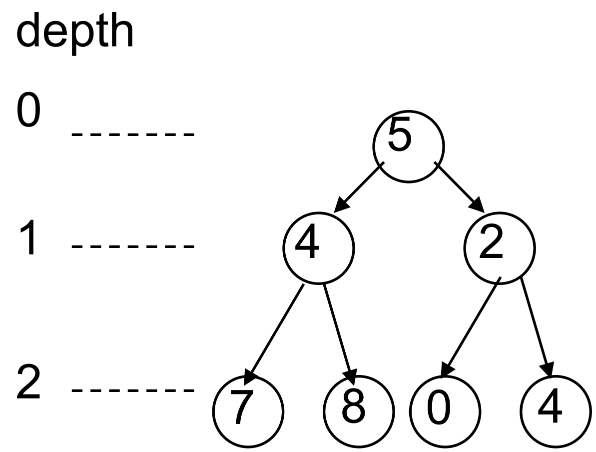
Max number of nodes at depth d: 2^d

If height of tree is h

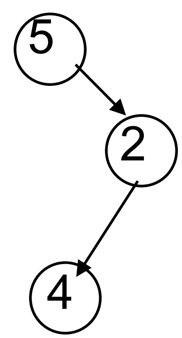
- min number of nodes in tree: $h + 1$
- Max number of nodes in tree:
- $2^0 + \ldots + 2^h = 2^{h+1} - 1$

Complete binary tree

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



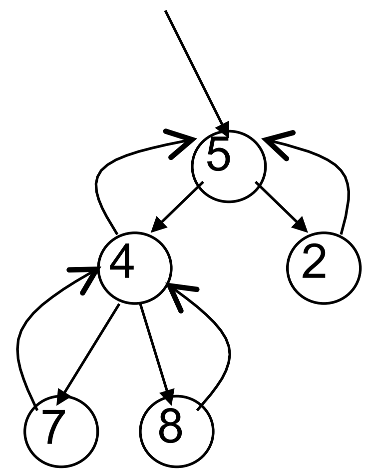
Height 2,
maximum number of nodes



Height 2,
minimum number of nodes

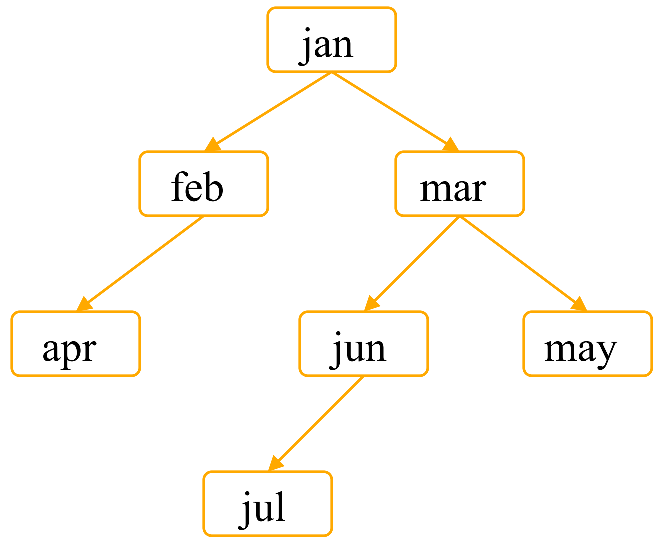
Tree with Parent Pointers

- In some applications, it is useful to have trees in which nodes can reference their parents
- Analog of doubly-linked lists



Things to Think About

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



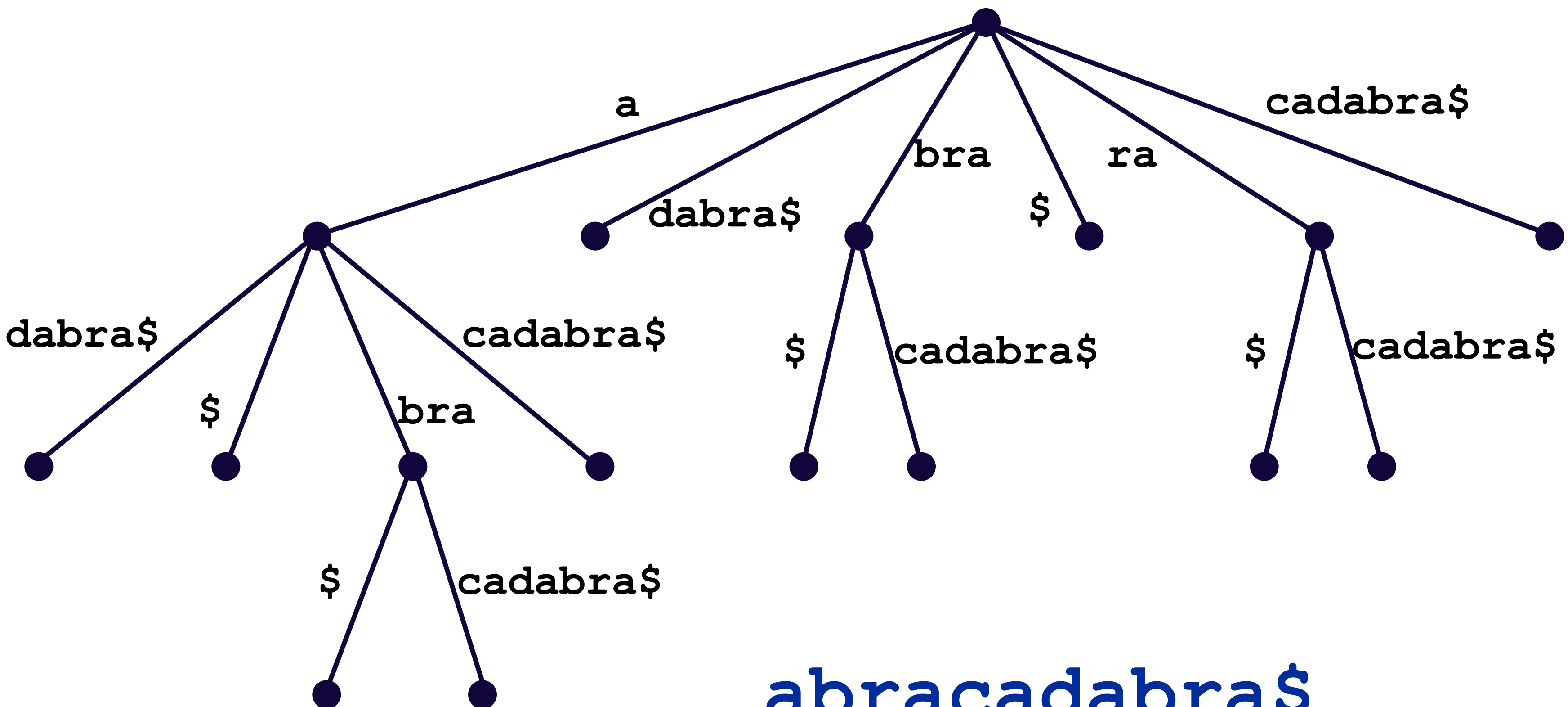
A BST works great as long as it's *balanced*

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

Suffix Trees

- Given a string s, a suffix tree for s is a tree such that
  - each edge has a unique label, which is a nonnull substring of s
  - any two edges out of the same node have labels beginning with different characters
  - the labels along any path from the root to a leaf concatenate together to give a suffix of s
  - all suffixes are represented by some path
  - the leaf of the path is labeled with the index of the first character of the suffix in s
- Suffix trees can be constructed in linear time

Suffix Trees



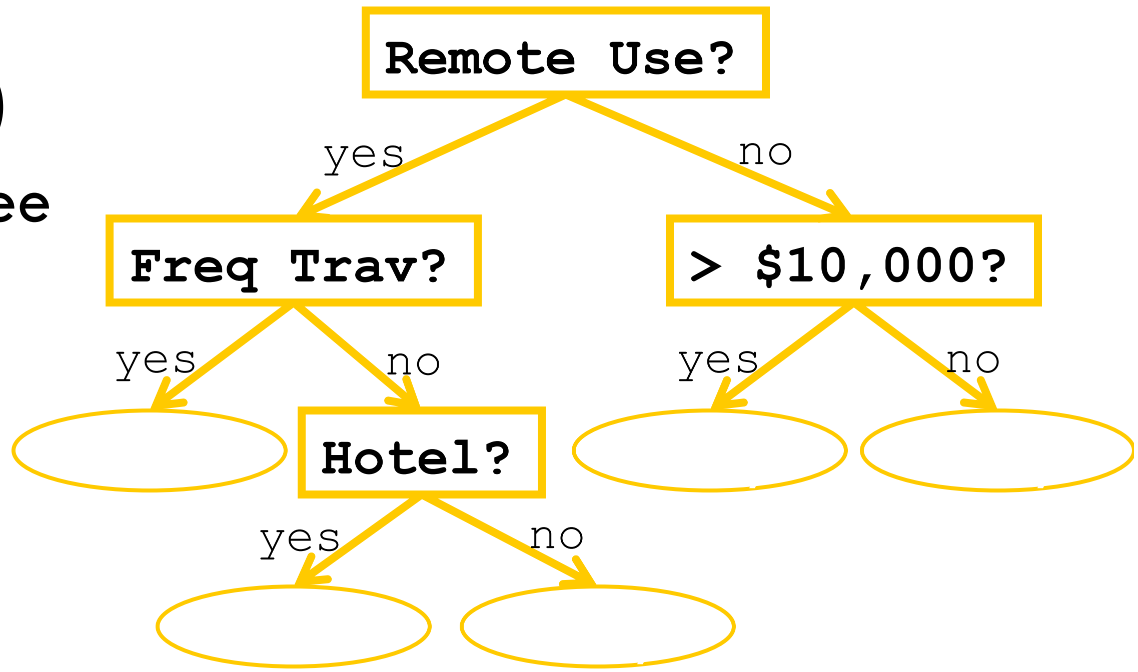
abracadabra$

Suffix Trees

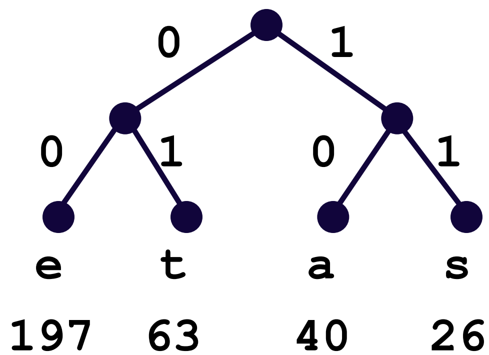
- Useful in string matching algorithms (e.g., longest common substring of 2 strings)
- Most algorithms linear time
- Used in genomics (human genome is ~4GB)

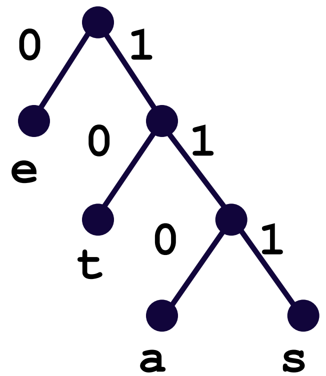
Decision Trees

- Classification:
  - Attributes (e.g. is CC used more than 200 miles from home?)
  - Values (e.g. yes/no)
  - Follow branch of tree based on value of attribute.
  - Leaves provide decision.
- Example:
  - Should credit card transaction be denied?



Huffman Trees





Fixed length encoding
197*2 + 63*2 + 40*2 + 26*2 = 652

Huffman encoding
197*1 + 63*2 + 40*3 + 26*3 = 521

Huffman Compression of "Ulysses"

' '	242125	00100000	3	110
'e'	139496	01100101	3	000
't'	95660	01110100	4	1010
'a'	89651	01100001	4	1000
'o'	88884	01101111	4	0111
'n'	78465	01101110	4	0101
'i'	76505	01101001	4	0100
's'	73186	01110011	4	0011
'h'	68625	01101000	5	11111
'r'	68320	01110010	5	11110
'l'	52657	01101100	5	10111
'u'	32942	01110101	6	111011
'g'	26201	01100111	6	101101
'f'	25248	01100110	6	101100
'.'	21361	00101110	6	011010
'p'	20661	01110000	6	011001

Huffman Compression of "Ulysses"

...

```
'7'       68  00110111  15  111010101001111
'/'       58  00101111  15  111010101001110
'X'       19  01011000  16  0110000000100011
'&'        3  00100110  18  011000000010001010
'%'        3  00100101  19  0110000000100010111
'+'        2  00101011  19  0110000000100010110
original size   11904320
compressed size  6822151
42.7% compression
```

BSP Trees

- BSP = Binary Space Partition (not related to BST!)
- Used to render 3D images composed of polygons
- Each node n has one polygon p as data
- Left subtree of n contains all polygons on one side of p
- Right subtree of n contains all polygons on the other side of p
- Order of traversal determines occlusion (hiding)!

Tree Summary

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