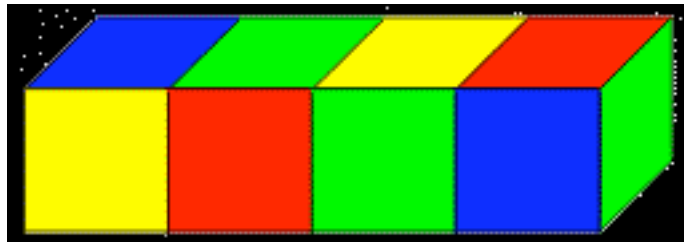


Today's topics

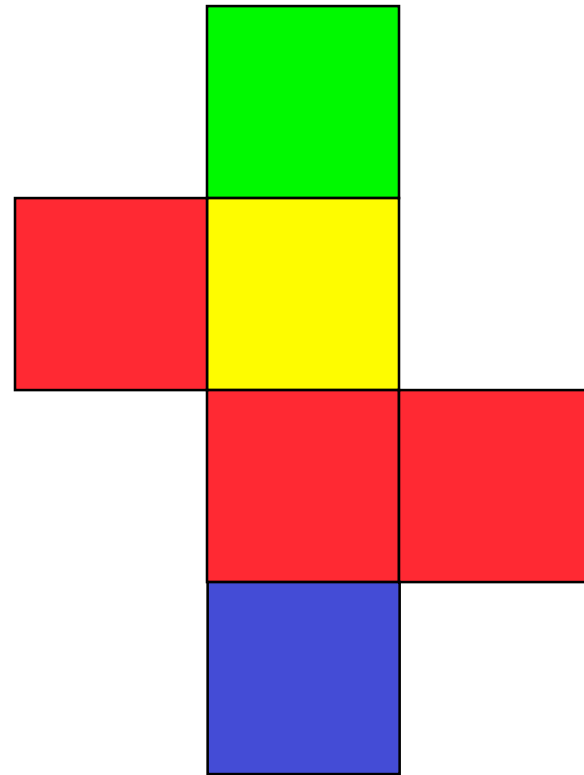
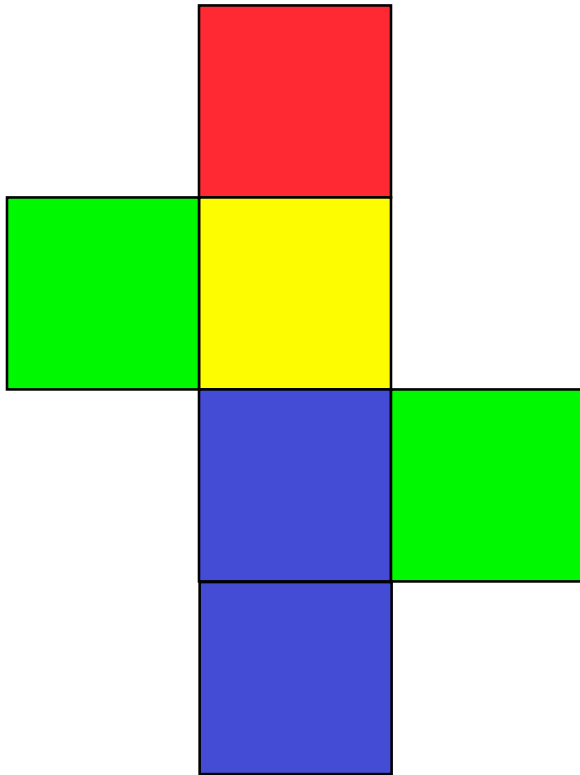
- Instant Insanity
- Trees
 - Properties
 - Applications
- Reading: Sections 9.1-9.2

Instant Insanity

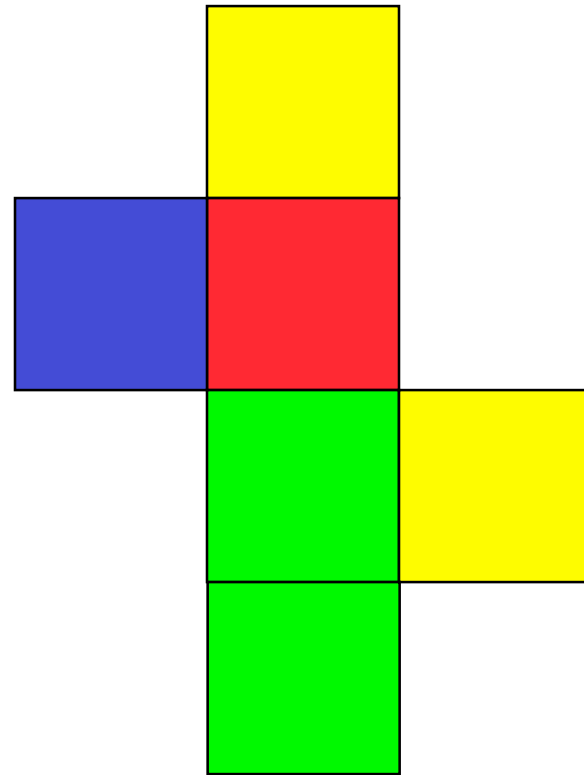
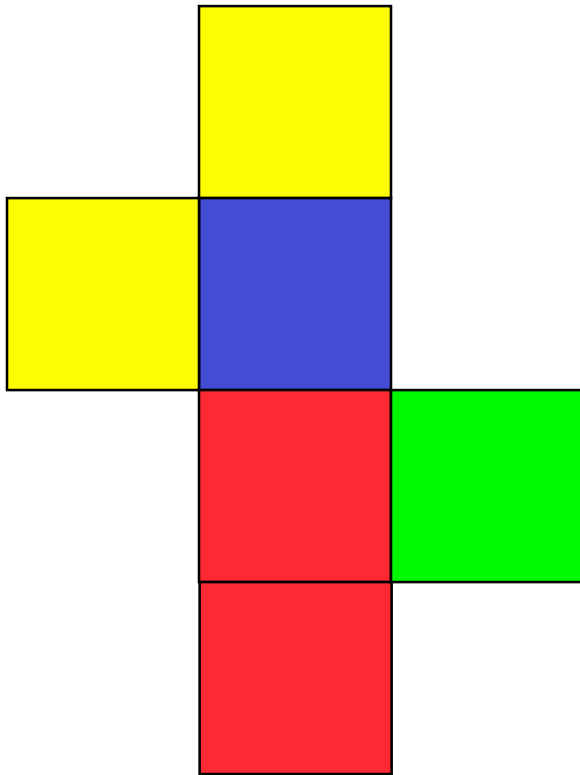
- Given four cubes, how can we stack the cubes, so that whether the cubes are viewed from the front, back, left or right, one sees all four colors



Cubes 1 & 2



Cubes 3 & 4



Creating graph formulation

- Create multigraph
 - Four vertices represent four colors of faces
 - Connect vertices with edges when they are opposite from each other
 - Label cubes
- Solving
 - Find subgraphs with certain properties
 - What properties?

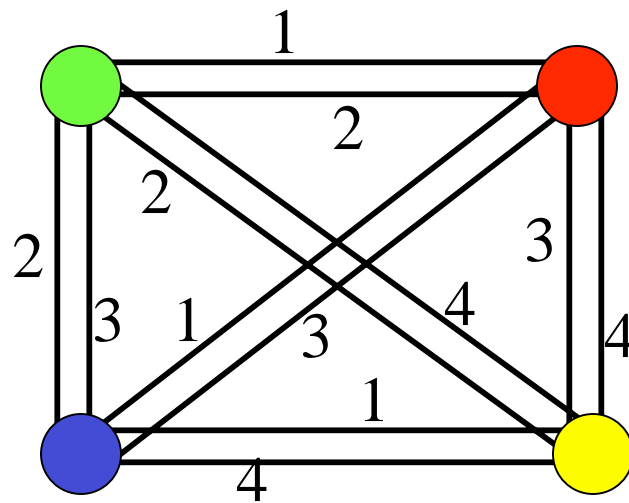
Summary

Graph-theoretic Formulation of Instant Insanity:

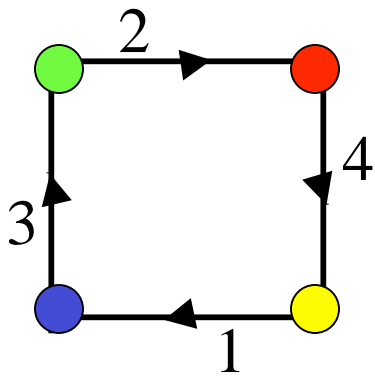
- ✓ Find two edge-disjoint labeled factors in the graph of the Instant Insanity puzzle, one for left-right sides and one for front-back sides
- ✓ Use the clockwise traversal procedure to determine the left-right and front-back arrangements of each cube.

Try it out:

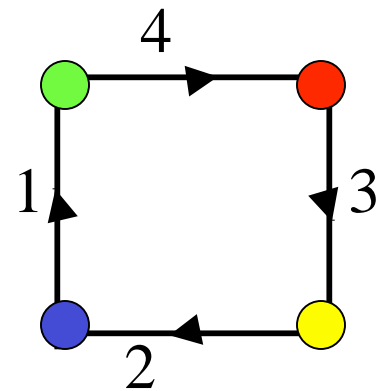
Find all Instant Insanity solution to the game with the multigraph:



Answer:



and



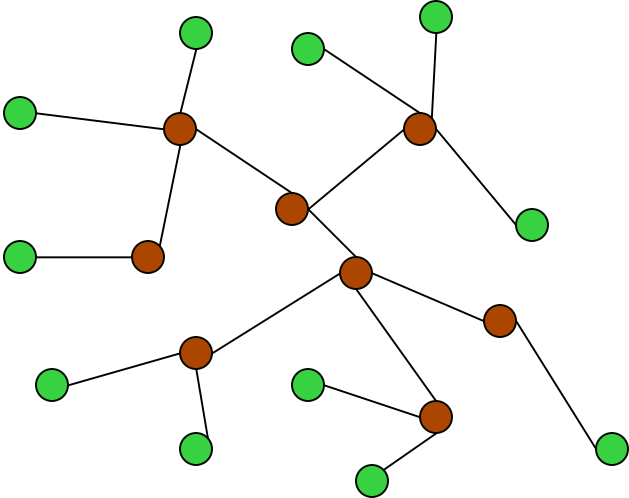
§9.1: Introduction to Trees

- A *tree* is a connected undirected graph that contains no circuits.
 - **Theorem:** There is a unique simple path between any two of its nodes.
- A (not-necessarily-connected) undirected graph without simple circuits is called a *forest*.
 - You can think of it as a set of trees having disjoint sets of nodes.
- A *leaf* node in a tree or forest is any pendant or isolated vertex. An *internal* node is any non-leaf vertex (thus it has degree \geq ____).

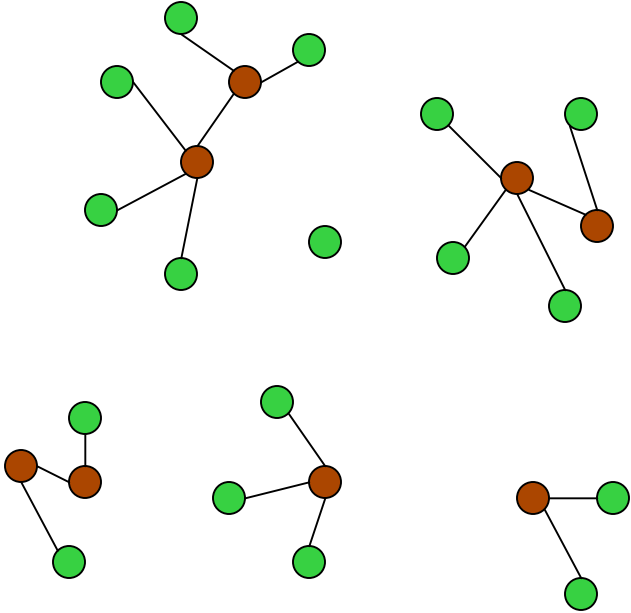
Tree and Forest Examples

Leaves in green, internal nodes in brown.

- A Tree:



- A Forest:

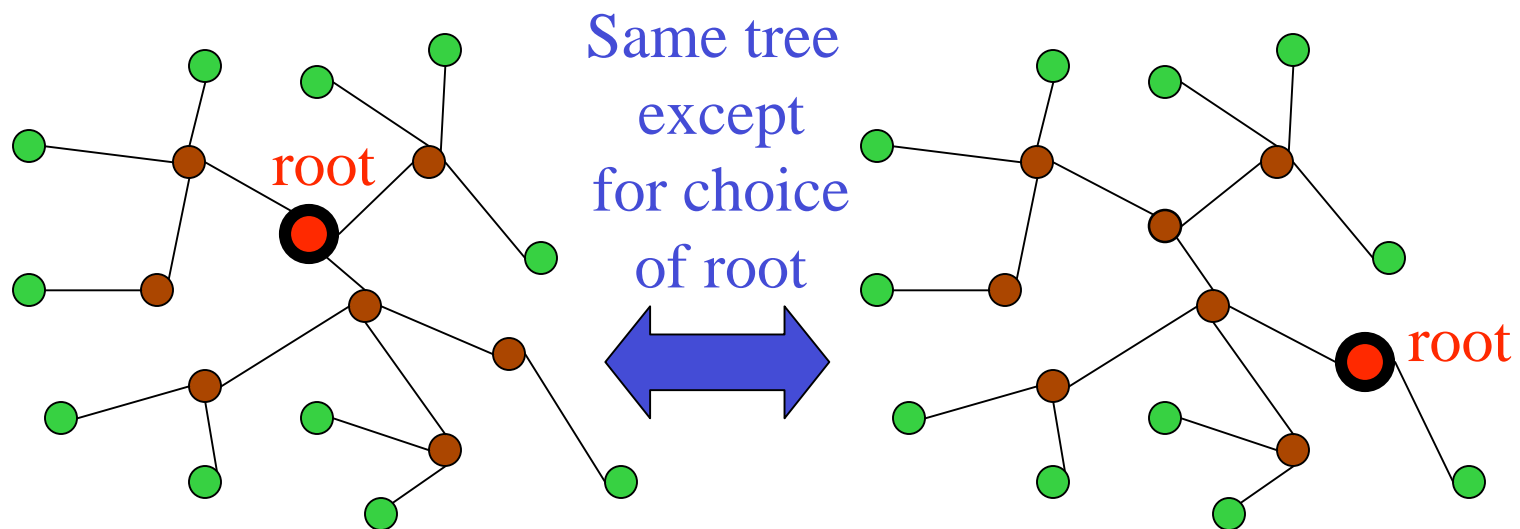


Rooted Trees

- A *rooted tree* is a tree in which one node has been designated the *root*.
 - Every edge is (implicitly or explicitly) directed away from the root.
- You should know the following terms about rooted trees:
 - Parent, child, siblings, ancestors, descendants, leaf, internal node, subtree.

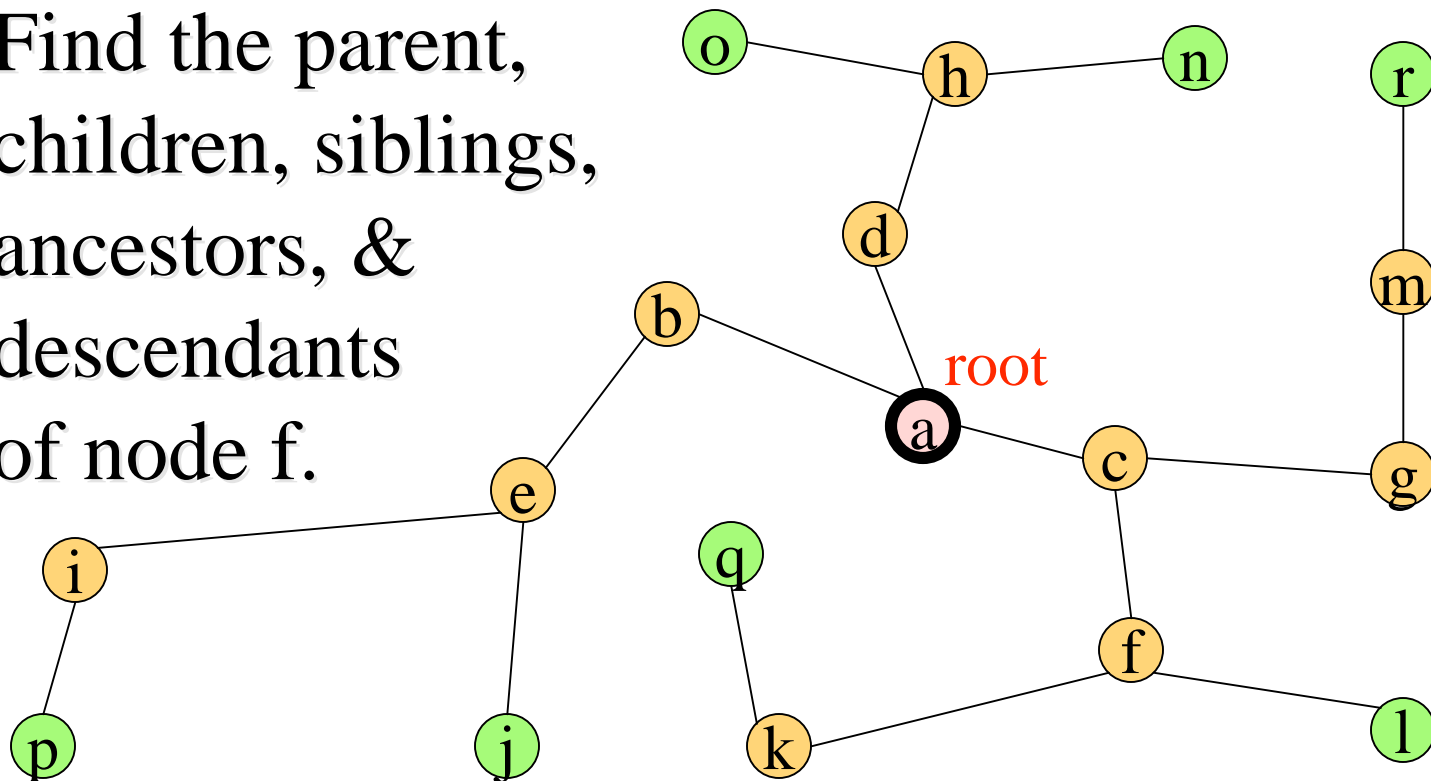
Rooted Tree Examples

- Note that a given unrooted tree with n nodes yields n different rooted trees.



Rooted-Tree Terminology Exercise

- Find the parent, children, siblings, ancestors, & descendants of node f.

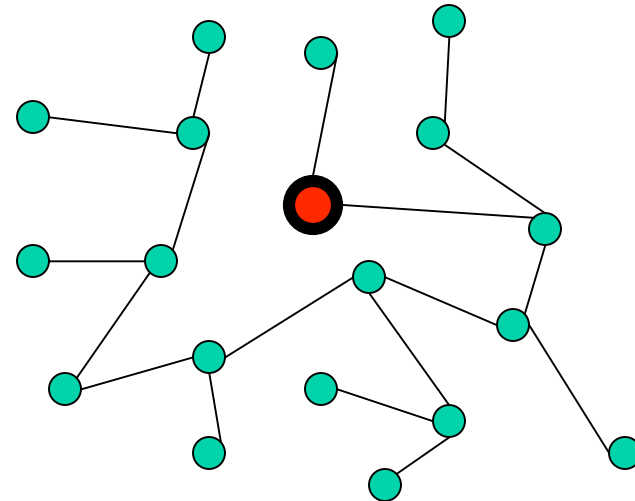
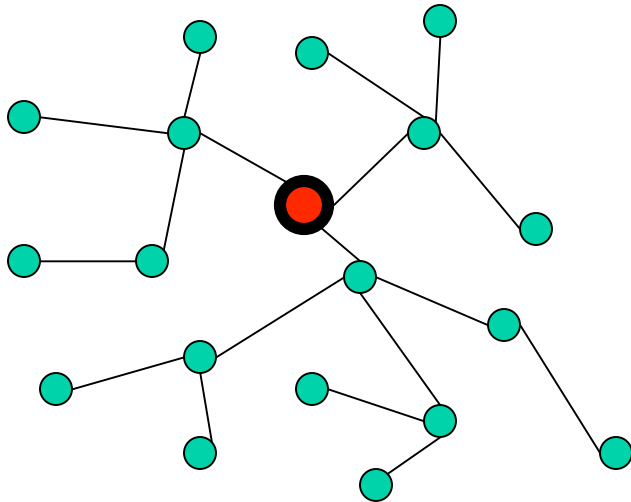


n-ary trees

- A rooted tree is called *n*-ary if every vertex has no more than *n* children.
 - It is called *full* if every internal (non-leaf) vertex has *exactly n* children.
- A 2-ary tree is called a *binary tree*.
 - These are handy for describing sequences of yes-no decisions.
 - **Example:** Comparisons in binary search algorithm.

Which Tree is Binary?

- **Theorem:** A given rooted tree is a binary tree iff every node other than the root has degree ≤ 2 , and the root has degree ≤ 3 .



Ordered Rooted Tree

- This is just a rooted tree in which the children of each internal node are ordered.
- In ordered binary trees, we can define:
 - left child, right child
 - left subtree, right subtree
- For n -ary trees with $n > 2$, can use terms like “leftmost”, “rightmost,” *etc.*

Trees as Models

- Can use trees to model the following:
 - Saturated hydrocarbons
 - Organizational structures
 - Computer file systems
- In each case, would you use a rooted or a non-rooted tree?

Some Tree Theorems

- Any tree with n nodes has $e = n - 1$ edges.
 - **Proof:** Consider removing leaves.
- A full m -ary tree with i internal nodes has $n = mi + 1$ nodes, and $l = (m - 1)i + 1$ leaves.
 - **Proof:** There are mi children of internal nodes, plus the root. And, $l = n - i = (m - 1)i + 1$. \square
- Thus, when m is known and the tree is full, we can compute all four of the values e , i , n , and l , given any one of them.

Some More Tree Theorems

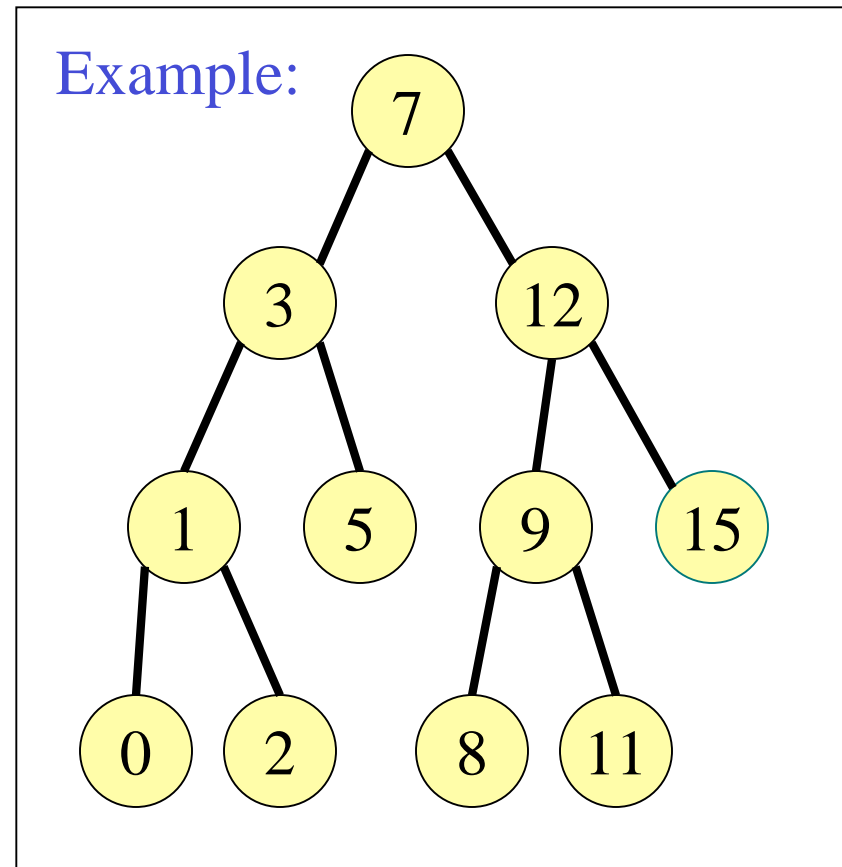
- **Definition:** The *level* of a node is the length of the simple path from the root to the node.
 - The *height* of a tree is maximum node level.
 - A rooted m -ary tree with height h is called *balanced* if all leaves are at levels h or $h-1$.
- **Theorem:** There are at most m^h leaves in an m -ary tree of height h .
 - **Corollary:** An m -ary tree with l leaves has height $h \geq \lceil \log_m l \rceil$. If m is full and balanced then $h = \lceil \log_m l \rceil$.

Binary Search Trees

- A representation for sorted sets of items.
 - Supports the following operations in $\Theta(\log n)$ average-case time:
 - Searching for an existing item.
 - Inserting a new item, if not already present.
 - Supports printing out all items in $\Theta(n)$ time.
- Note that inserting into a plain sequence a_i would instead take $\Theta(n)$ worst-case time.

Binary Search Tree Format

- Items are stored at individual tree nodes.
- We arrange for the tree to always obey this invariant:
 - For every item x ,
 - Every node in x 's left subtree is less than x .
 - Every node in x 's right subtree is greater than x .



Coin-Weighing Problem

- Imagine you have 8 coins, one of which is a lighter counterfeit, and a free-beam balance.
 - No scale of weight markings is required for this problem!
- How many weighings are needed to guarantee that the counterfeit coin will be found?

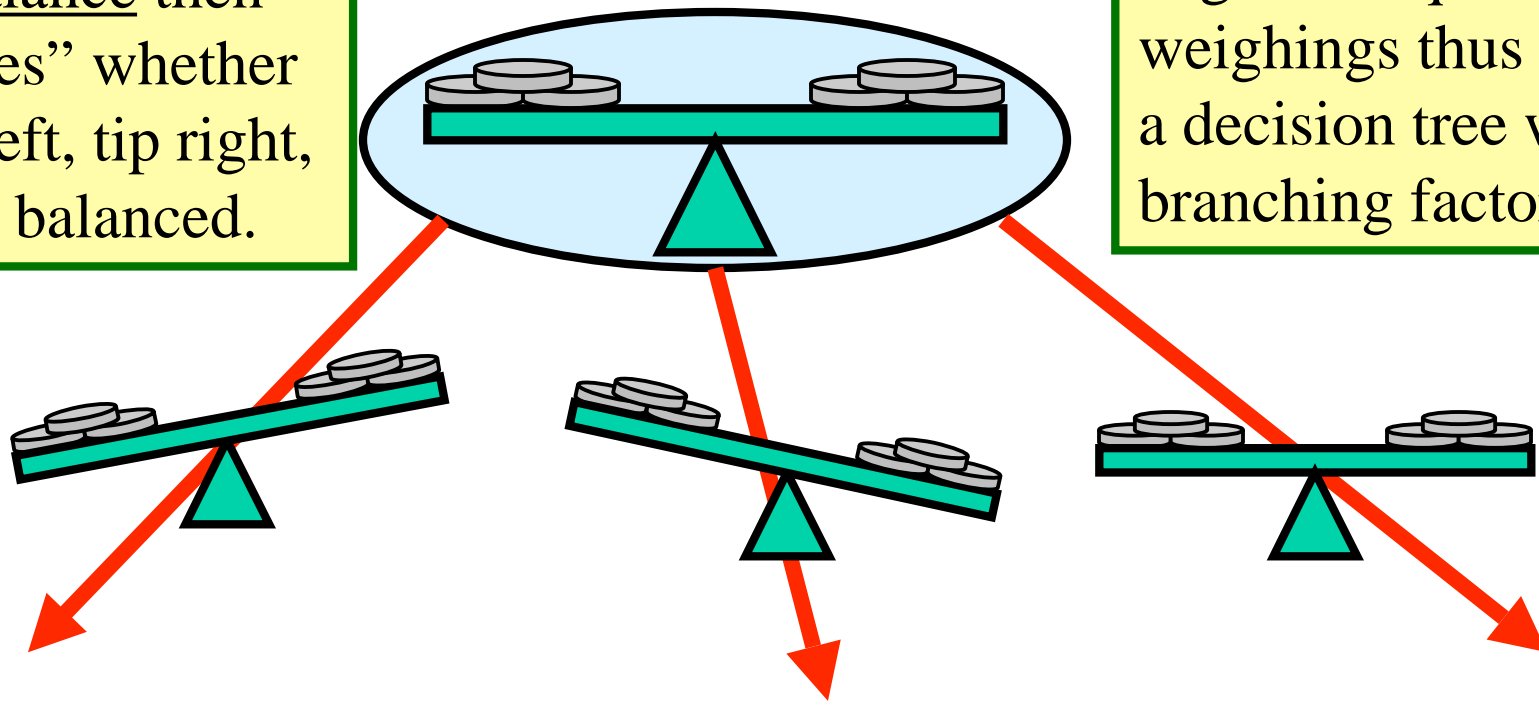


As a Decision-Tree Problem

- In each situation, we pick two disjoint and equal-size subsets of coins to put on the scale.

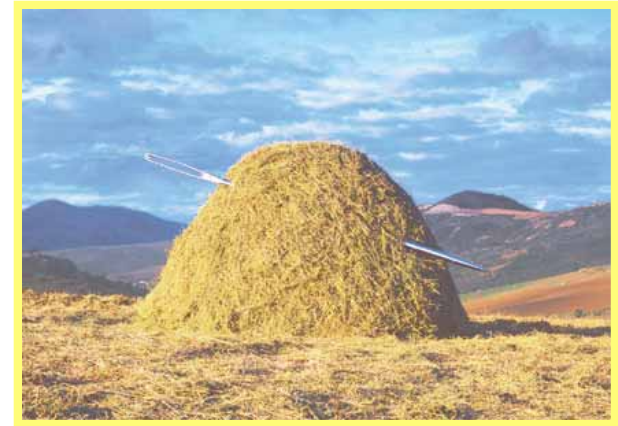
The balance then “decides” whether to tip left, tip right, or stay balanced.

A given sequence of weighings thus yields a decision tree with branching factor 3.



Applying the Tree Height Theorem

- The decision tree must have at least 8 leaf nodes, since there are 8 possible outcomes.
 - In terms of which coin is the counterfeit one.
- Recall the tree-height theorem, $h \geq \lceil \log_m n \rceil$.
 - Thus the decision tree must have height $h \geq \lceil \log_3 8 \rceil = \lceil 1.893\dots \rceil = 2$.
- Let's see if we solve the problem with *only* 2 weighings...



General Solution Strategy

- The problem is an example of searching for 1 unique particular item, from among a list of n otherwise identical items.
 - Somewhat analogous to the adage of “searching for a needle in haystack.”
- Armed with our balance, we can attack the problem using a divide-and-conquer strategy, like what’s done in binary search.
 - We want to narrow down the set of possible locations where the desired item (coin) could be found down from n to just 1, in a logarithmic fashion.
- Each weighing has 3 possible outcomes.
 - Thus, we should use it to partition the search space into 3 pieces that are as close to equal-sized as possible.
- This strategy will lead to the minimum possible worst-case number of weighings required.

General Balance Strategy

- On each step, put $\lceil n/3 \rceil$ of the n coins to be searched on each side of the scale.
 - If the scale tips to the left, then:
 - The lightweight fake is in the right set of $\lceil n/3 \rceil \approx n/3$ coins.
 - If the scale tips to the right, then:
 - The lightweight fake is in the left set of $\lceil n/3 \rceil \approx n/3$ coins.
 - If the scale stays balanced, then:
 - The fake is in the remaining set of $n - 2\lceil n/3 \rceil \approx n/3$ coins that were not weighed!
- Except if $n \bmod 3 = 1$ then we can do a little better by weighing $\lfloor n/3 \rfloor$ of the coins on each side.

You can prove that this strategy always leads to a balanced 3-ary tree.

Coin Balancing Decision Tree

- Here's what the tree looks like in our case:

