CSCI 136
Data Structures &
Advanced Programming

Lecture 26

Fall 2019

Instructors: B&S

Administrative Details

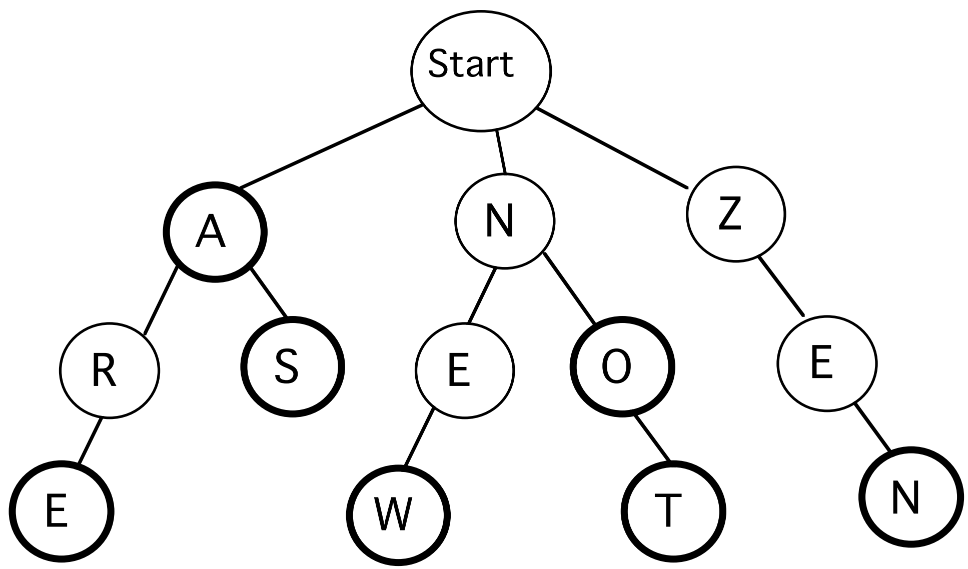
- Lab 9: Super Lexicon is online
  - Partners are permitted this week!
  - Please fill out the form by tonight at midnight
- Lab 6 back

Today

- Lab 9
- *Efficient* Binary search trees (Ch 14)
  - AVL Trees
    - Height is O(log n), so all operations are O(log n)
  - Red-Black Trees
    - Different height-balancing idea: height is O(log n)
    - All operations are O(log n)

Lab 9 : Lexicon

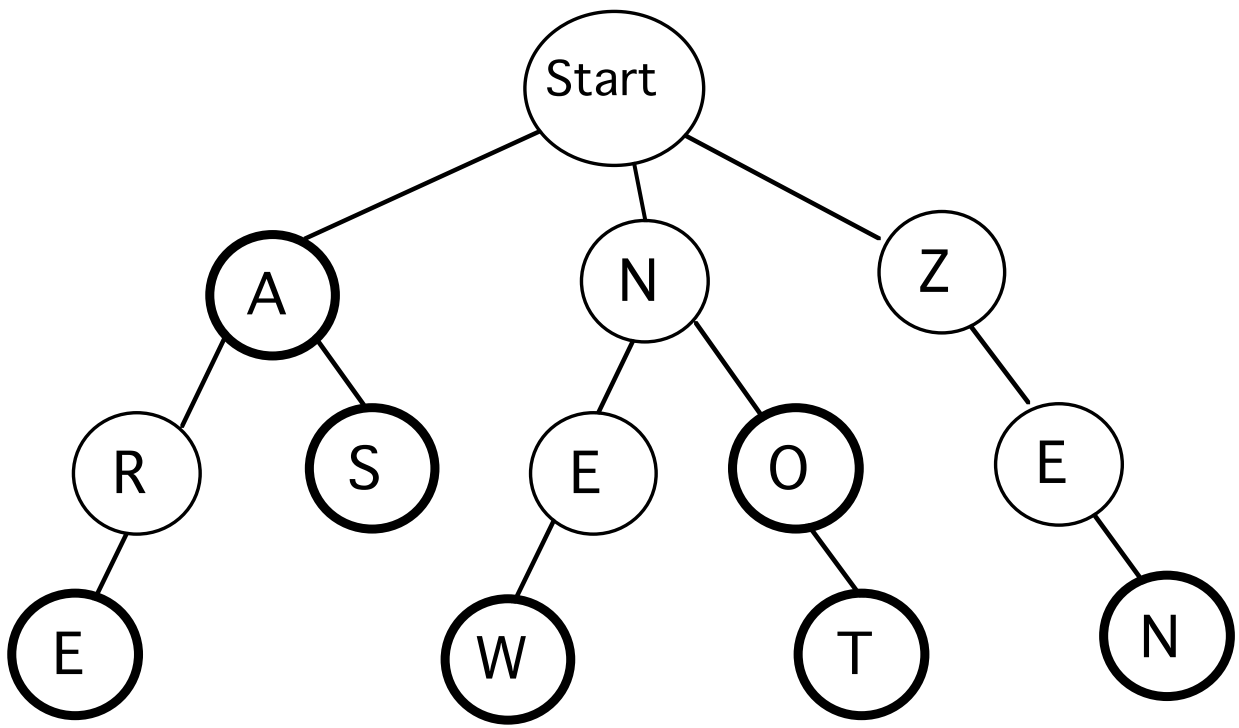
- Goal: Build a data structure that can efficiently store and search a large set of words
- A special kind of tree called a *trie*



Lab 9 : Tries

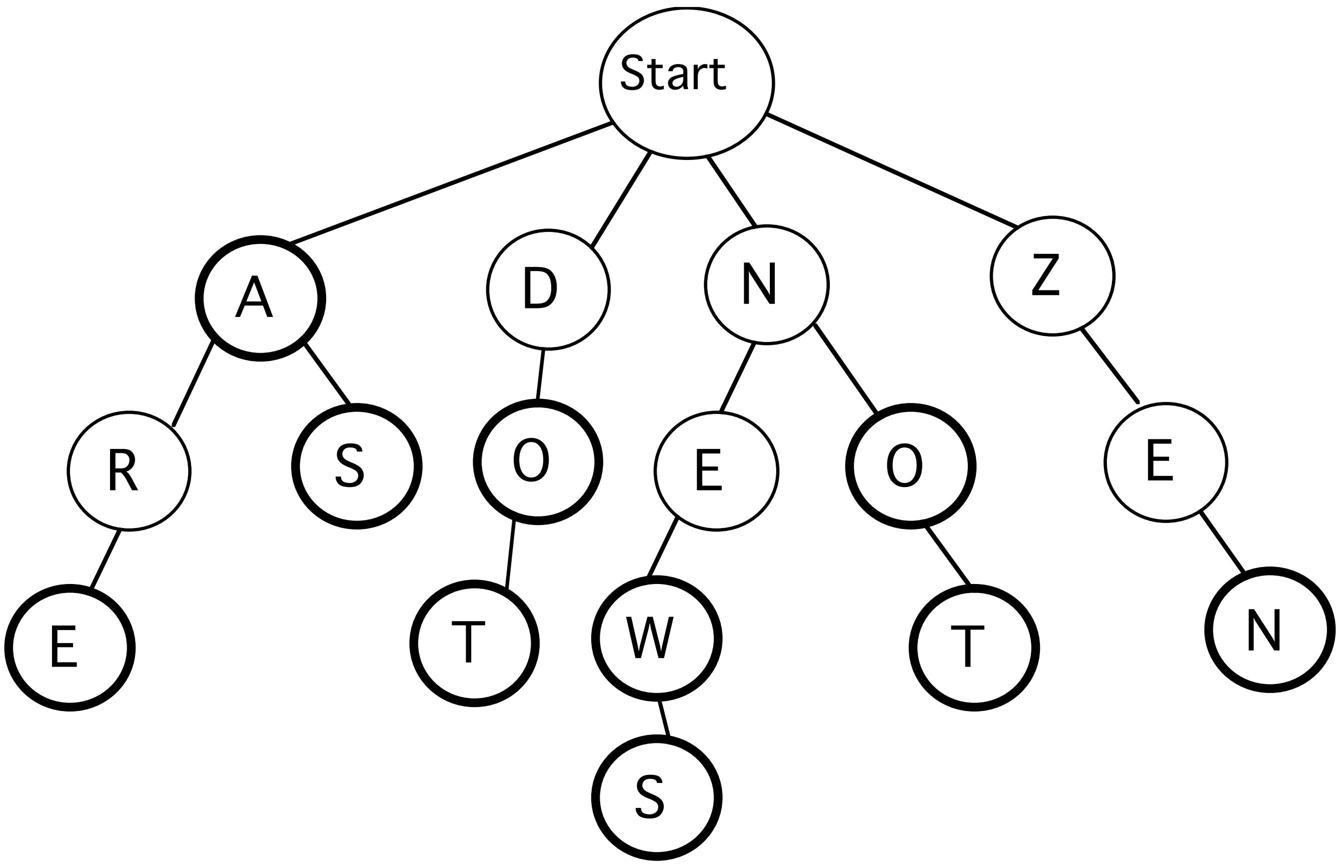
- A trie is a tree that stores words where
  - Each node holds a letter
  - Some nodes are "word" nodes (dark circles)
  - Any path from the root to a word node describes one of the stored words
  - All paths from the root form prefixes of stored words (a word is considered a prefix of itself)

Tries



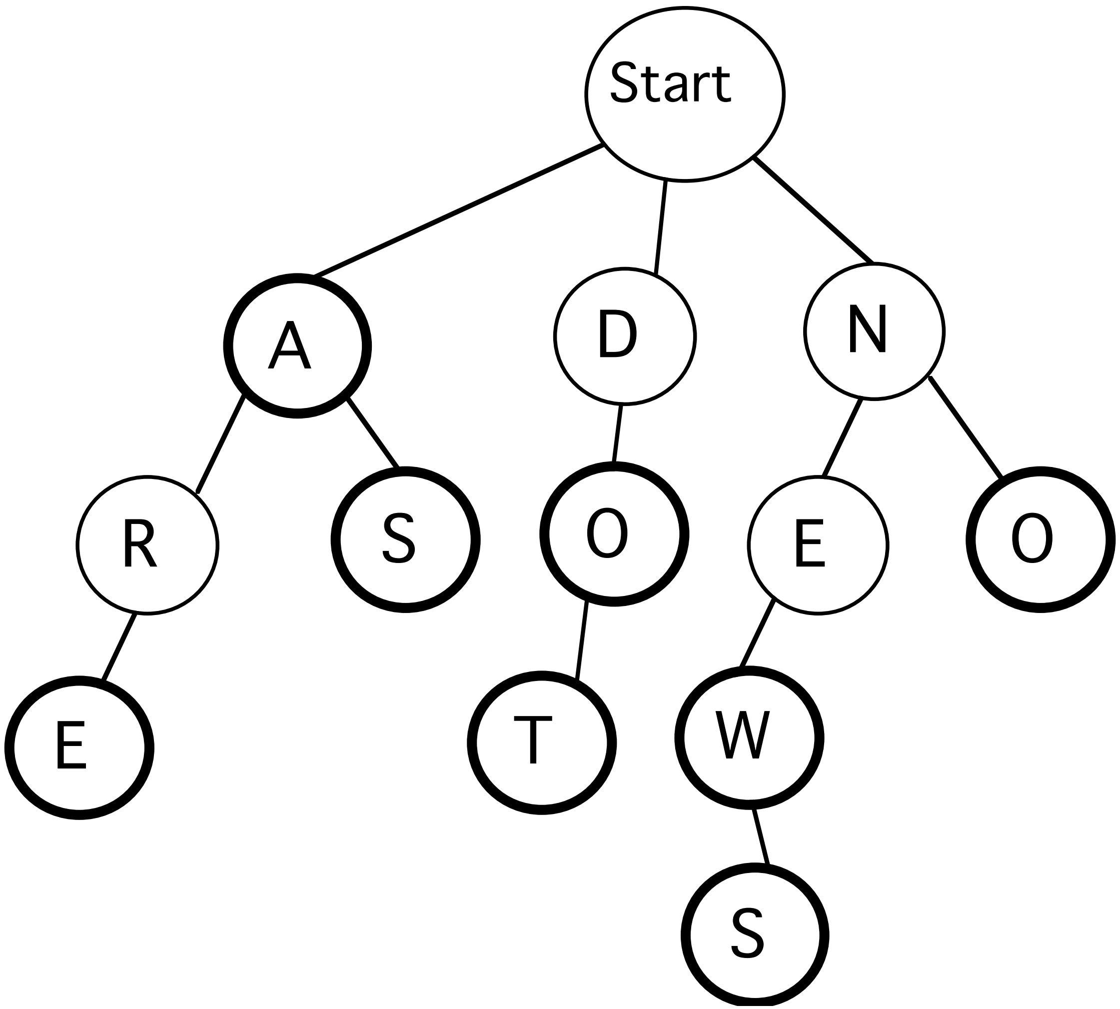
Now add “dot” and “news”

Tries



Now remove “not” and “zen”

Tries



Lab 9 : Lexicon

An interface that provides the methods

```
public interface Lexicon {
    public boolean addWord(String word);
    public int addWordsFromFile(String filename);
    public boolean removeWord(String word);
    public int numWords();
    public boolean containsWord(String word);
    public boolean containsPrefix(String prefix);
    public Iterator<String> iterator();
    public Set<String> suggestCorrections(String
            target, int maxDistance);
    public Set<String> matchRegex(String pattern);
}
```

Lab 9

- Implement a program that creates, updates, and searches a Lexicon
  - Based on a LexiconTrie class
    - Each node of the Trie is a LexiconNode
    - Analogous to a SLL consisting of SLLNodes
  - LexiconTrie implements the Lexicon Interface
  - Supports
    - adding/removing words
    - searching for words and prefixes
    - reading words from files
    - Iterating over all words

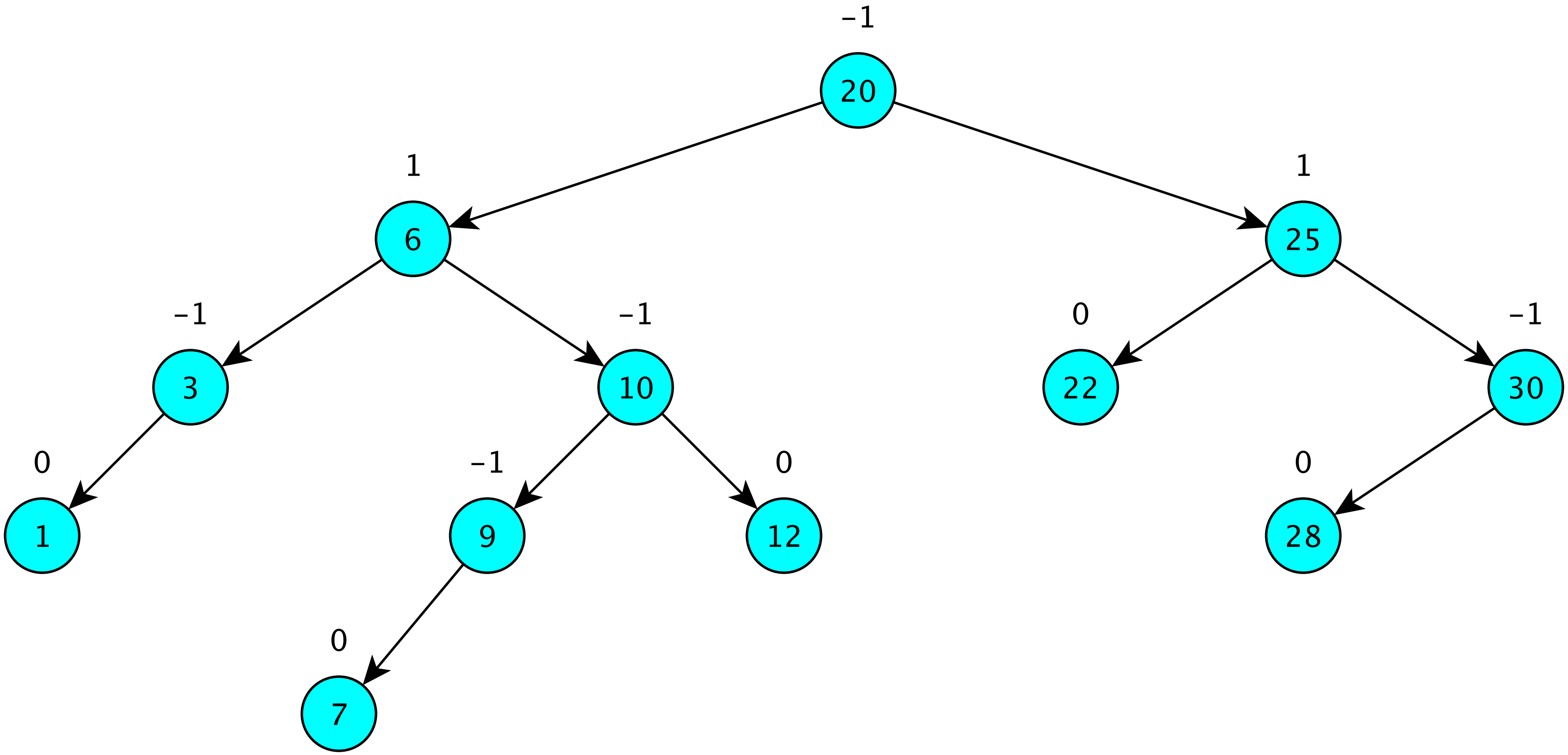
AVL Trees

One of the first balanced binary tree structures

Definition: A binary tree T is an AVL tree if

1. T is the empty tree, or
2. T has left and right sub-trees T_L and T_R such that
   a) The heights of T_L and T_R differ by at most 1, and
   b) T_L and T_R are AVL trees

AVL Trees



AVL Trees

- Balance Factor of a binary tree node:
  - height of right subtree minus height of left subtree.
  - A node with balance factor 1, 0, or -1 is considered *balanced*.
  - A node with any other balance factor is considered unbalanced and requires rebalancing the tree.
- Alternate Definition: An AVL Tree is a binary tree in which every node is balanced.

AVL Trees have O(log n) Height

Theorem: An AVL tree on n nodes has height O(log n)

Proof idea

- Show that an AVL tree of height h has at least fib(h) nodes (classic induction proof---try it!)
- Recall (HW): $fib(h) \geq \left({}^{3}\!/_{2}\right)^{h}$ if h ≥ 10
- So $n \geq \left({}^{3}\!/_{2}\right)^{h}$ and thus $\log_{3/2} n \geq h$
  - Recall that for any $a, b > 0$, $\log_a n = \frac{\log_b n}{\log_b a}$
  - So $\log_a n$ and $\log_b n$ are Big-O of one another
- So h is O(log n)

We used Fibonacci numbers in a data structures proof!!!

AVL Trees

If adding to an AVL tree creates an unbalanced node A, we rebalance the subtree with root A

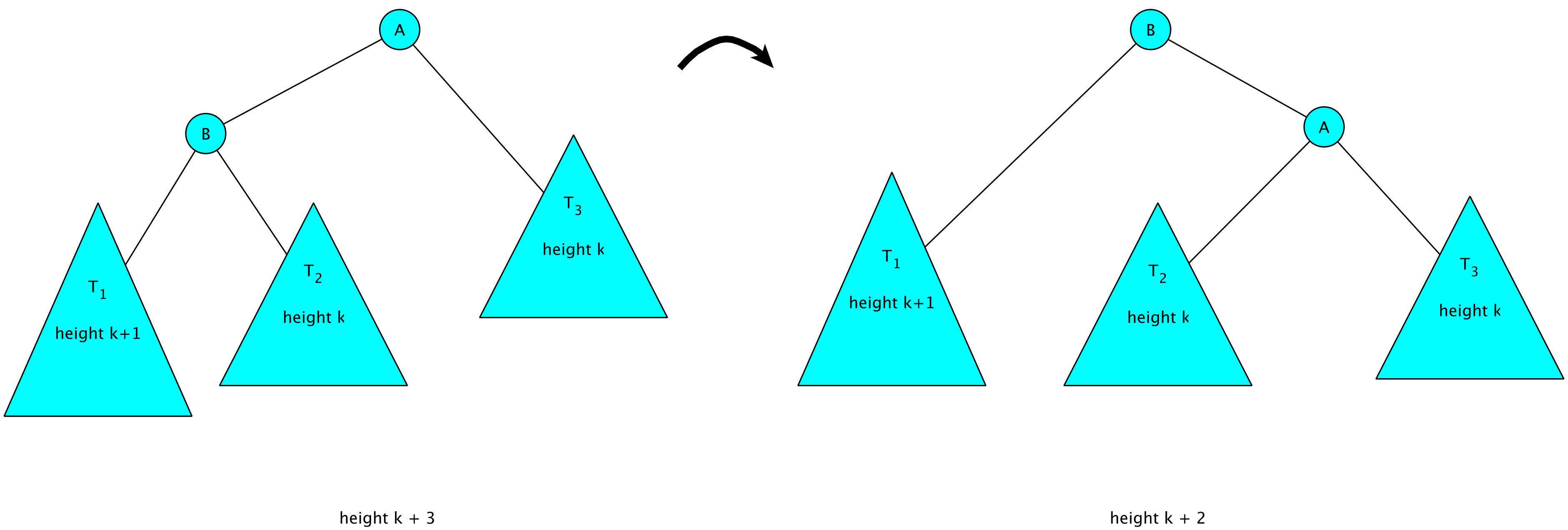
This involves a constant-time restructuring of part of the tree with root NA

The rebalancing steps are called *tree rotations*

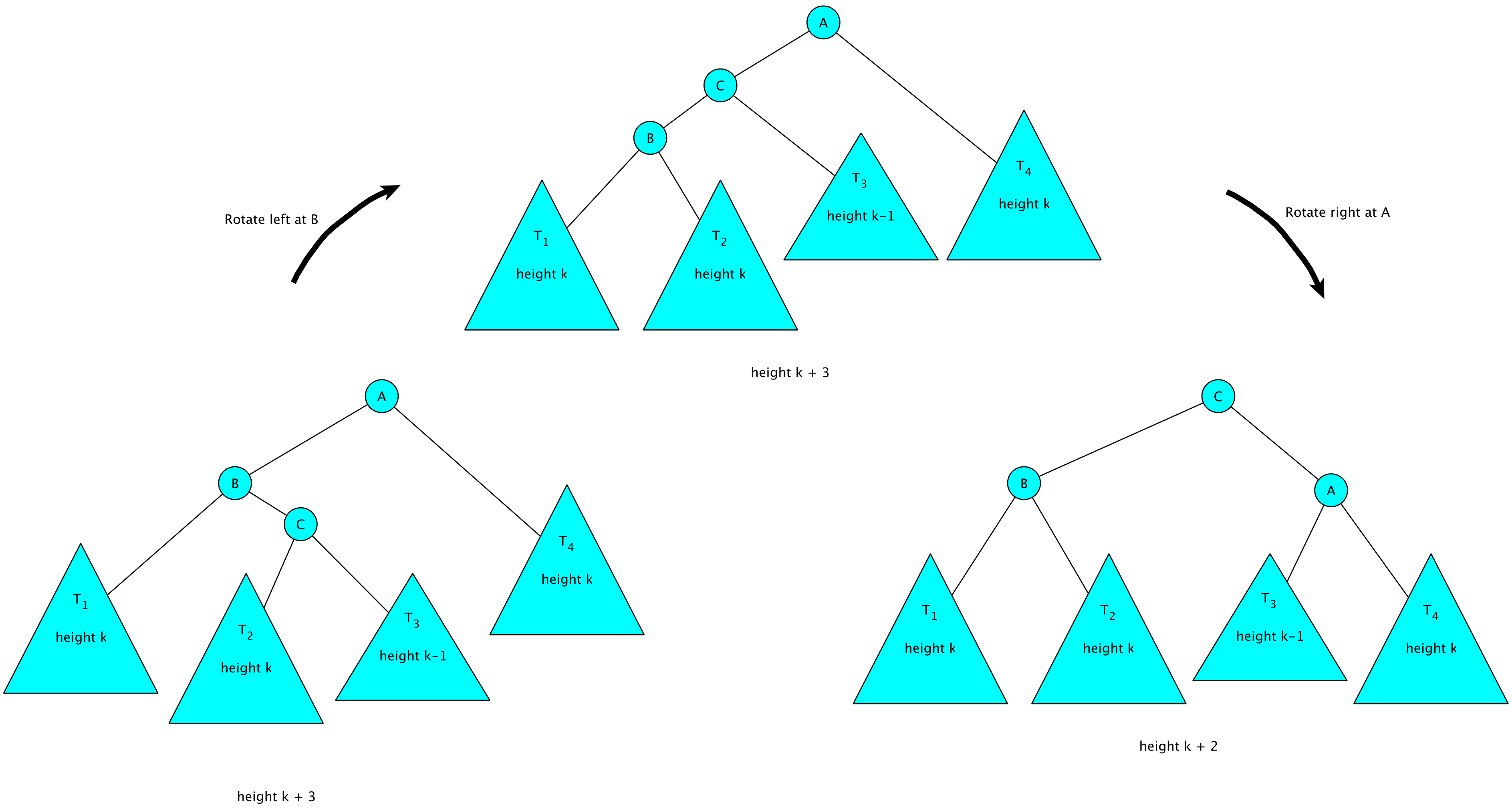
Tree rotations preserve binary search tree structure

Single Right Rotation

Assume A is unbalanced but its subtrees are AVL…



Double Rotation I



AVL Tree Facts

- A tree that is AVL except at root, where root balance factor equals ± 2 can be rebalanced with at most 2 rotations
- add(v) requires at most O(log n) balance factor changes and one (single or double) rotation to restore AVL structure
- remove(v) requires at most O(log n) balance factor changes and (single or double) rotations to restore AVL structure
- An AVL tree on n nodes has height O(log n)

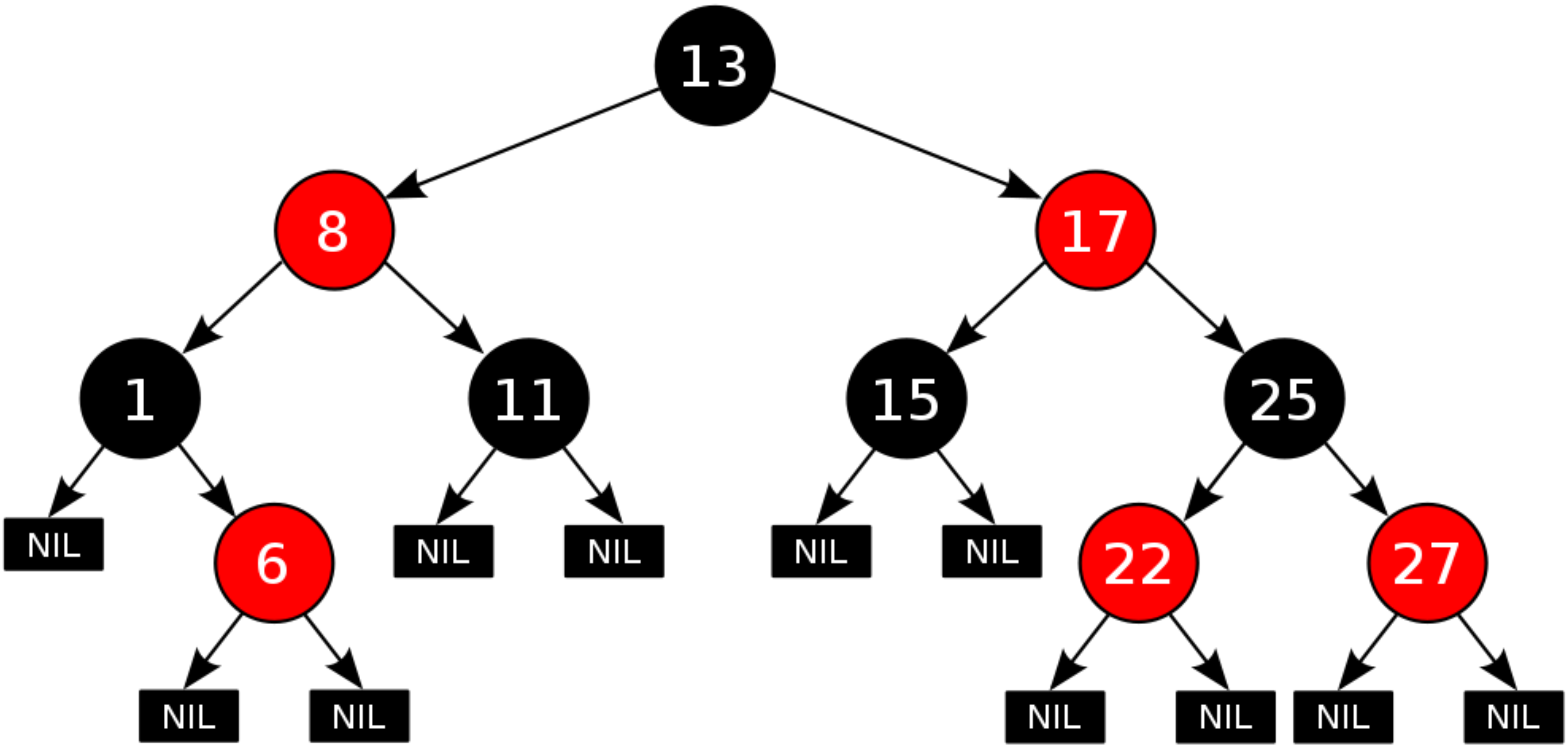
AVL Trees: One of Many

There are many strategies for tree balancing to preserve O(log n) height, including

- AVL Trees: guaranteed O(log n) height
- Red-black trees: guaranteed O(log n) height
- B-trees (not binary): guaranteed O(log n) height
  - 2-3 trees, 2-3-4 trees, red-black 2-3-4 trees, ...
- Splay trees: *Amortized* O(log n) time operations
- Randomized trees: O(log n) expected height

A Red-Black Tree

(from Wikipedia.org)



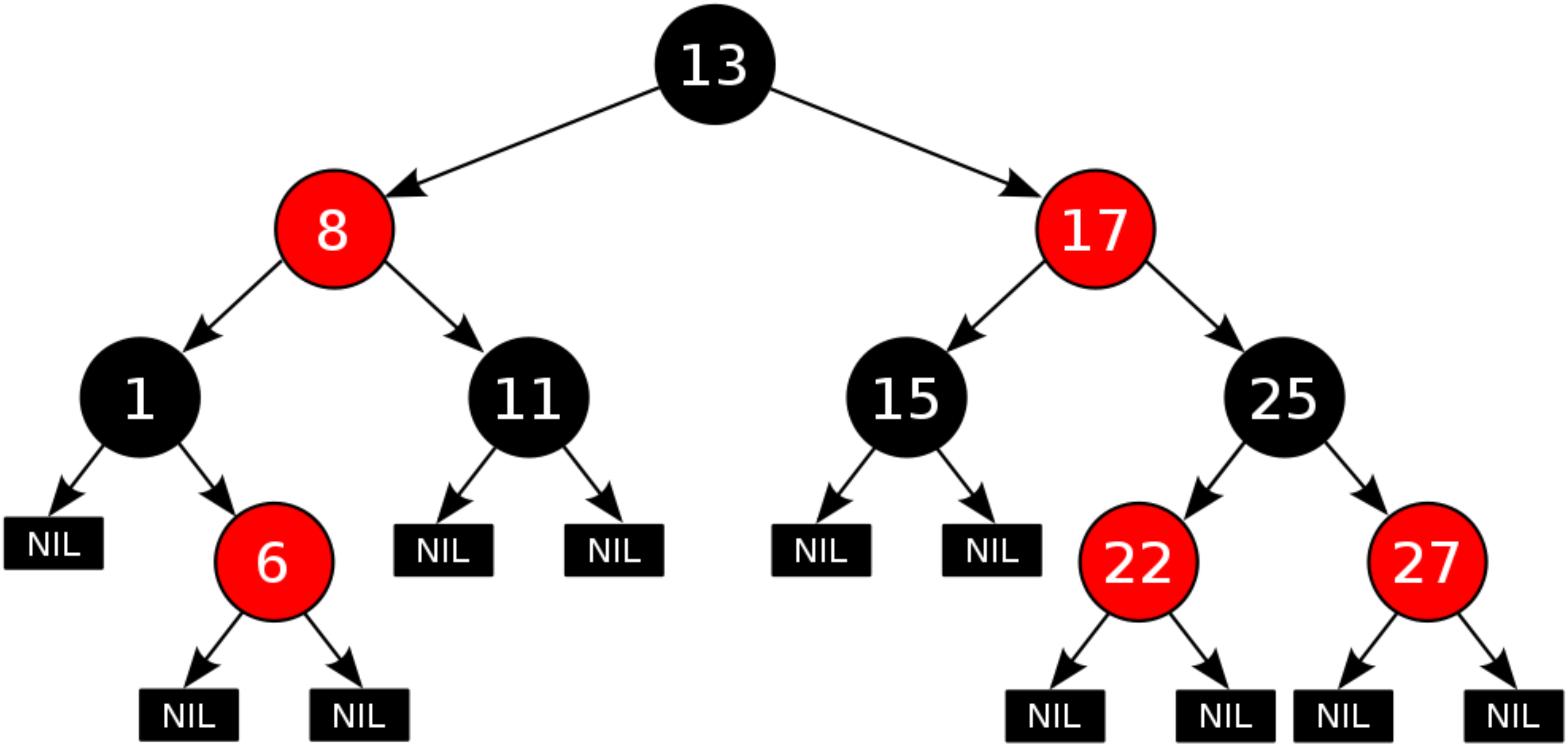
Red-Black Trees

Red-Black trees, like AVL, guarantee shallowness

- Each node is colored *red* or *black*
- Coloring satisfies these rules
  - All empty trees are black
    - We consider them to be the leaves of the tree
  - Children of red nodes are black
  - All paths from a given node to it’s descendent leaves have the *same number* of black nodes
    - This is called the *black height* of the node

A Red-Black Tree

(from Wikipedia.org)



Red-Black Trees

The coloring rules lead to the following result

Proposition: No leaf has depth more than twice that of any other leaf.

This in turn can be used to show

Theorem: A Red-Black tree with n internal nodes has height satisfying $h \leq 2\log(n+1)$

- Note: The tree will have *exactly* n+1 (empty) leaves
  - since each internal node has two children

Red-Black Trees

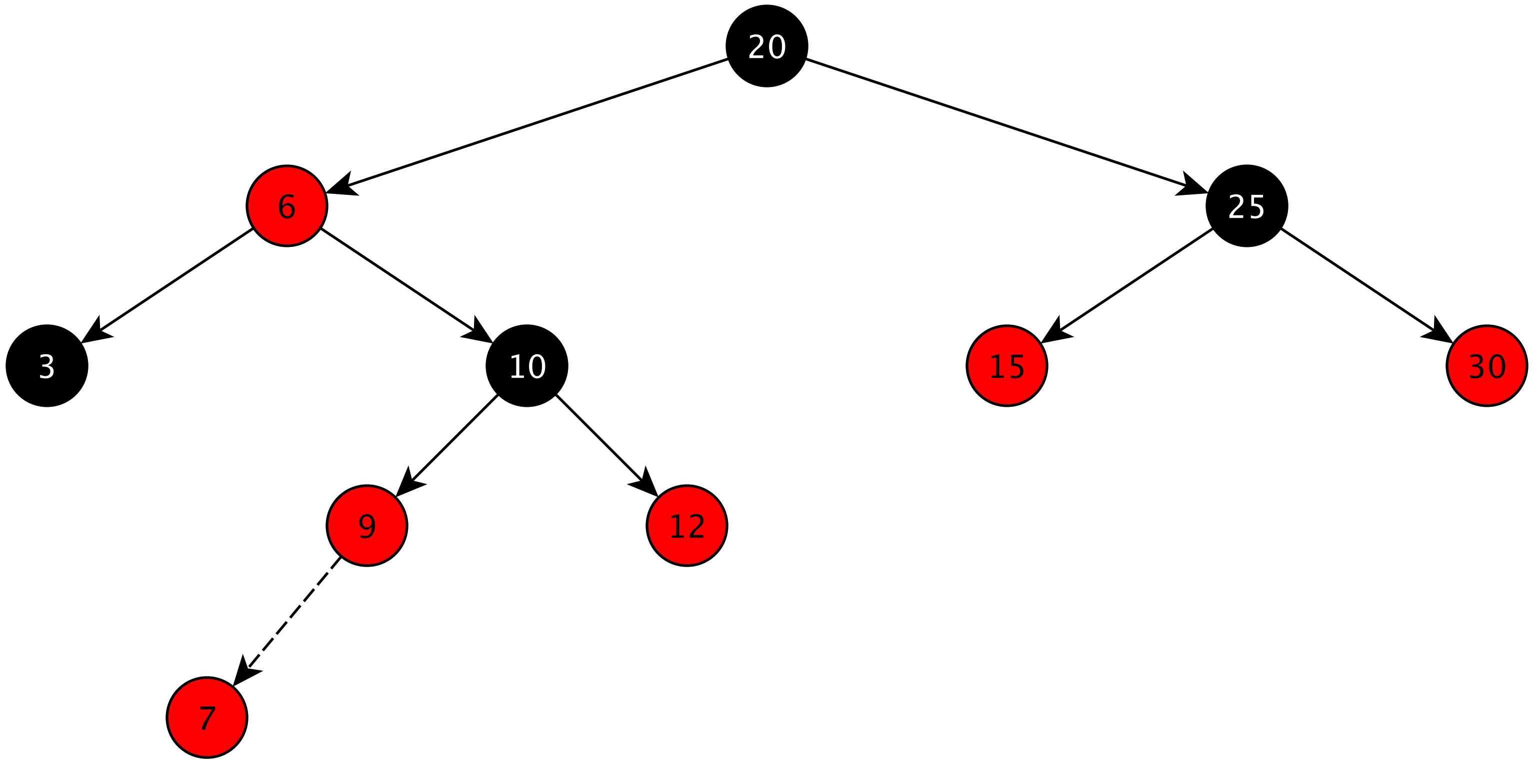
Theorem: A Red-Black tree with n *internal* nodes has height satisfying $h \leq 2\log(n+1)$

Proof sketch: Note: we count empty tree nodes!

- If root is red, recolor it black.
- Now merge red children into (black) parents
  - Now n’ ≤ n nodes and height h’ ≥ h/2
- New tree has all children with degree 2, 3, or 4
  - All leaves have depth *exactly* h’ and there are n+1 leaves
    - So $n+1 \geq 2^{h'}$, so $\log_2(n+1) \geq h' \geq \frac{h}{2}$
- Thus $2\log_2(n+1) \geq h$

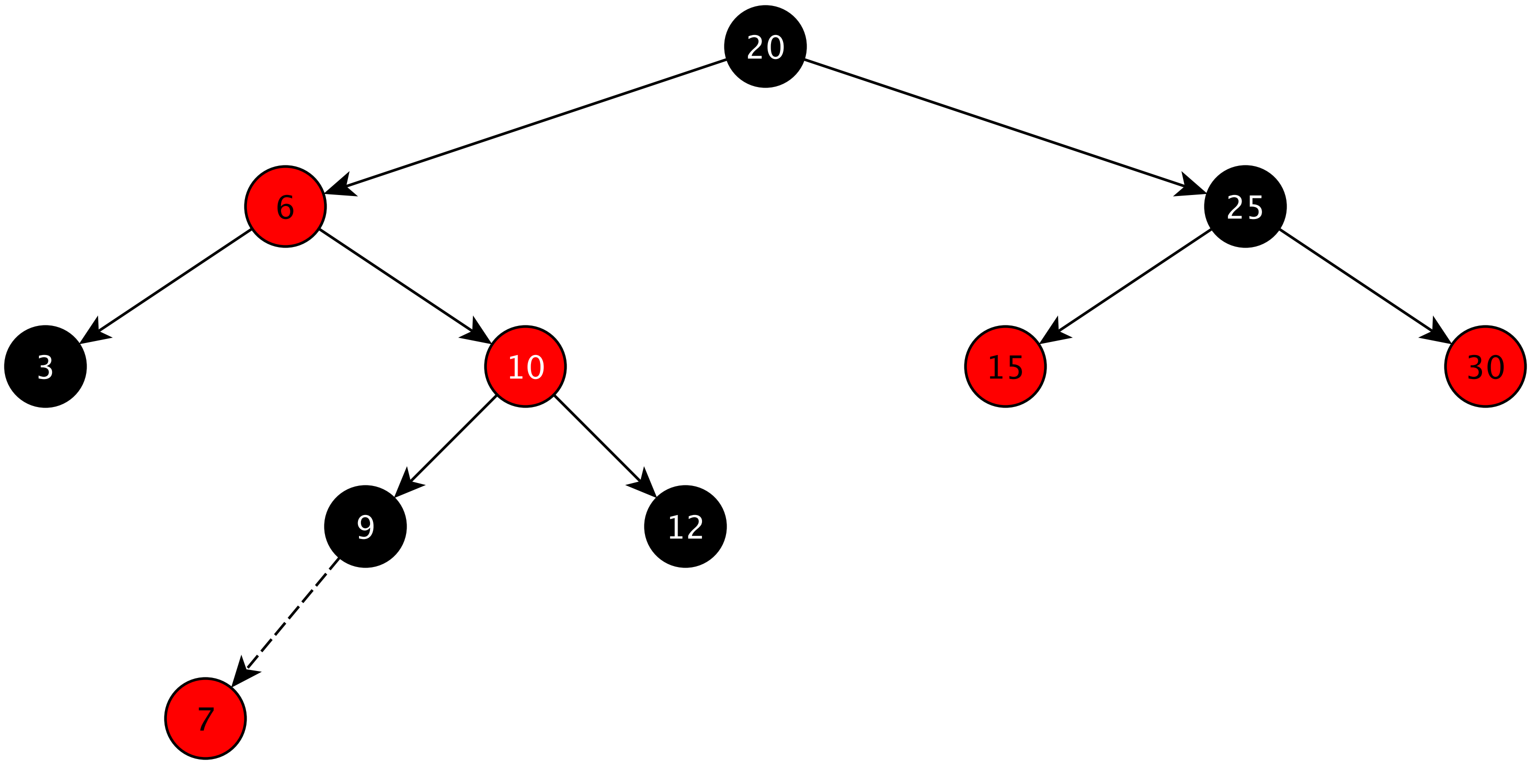
Corollary: R-B trees with n nodes have height O(log n)

Red-Black Tree Insertion



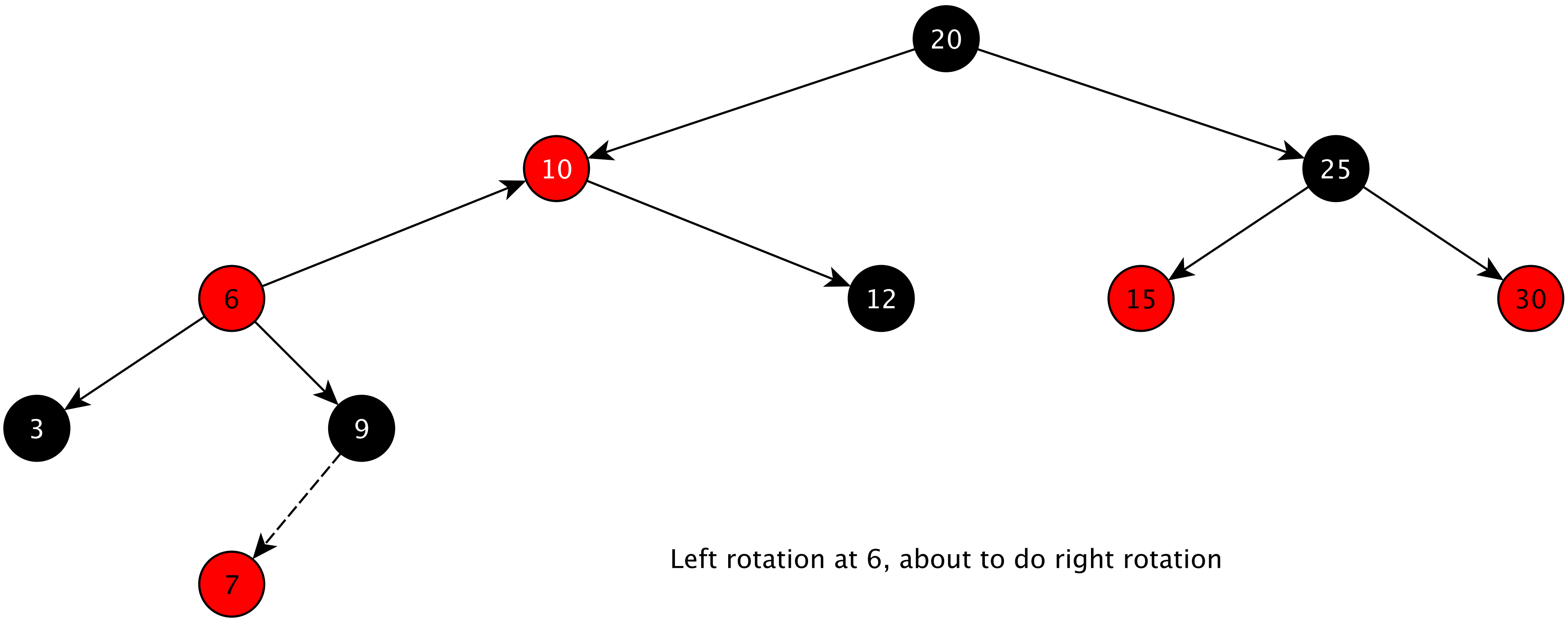
Black empty leaves not drawn. 7 just added Black-height still 2.

Red-Black Tree Insertion



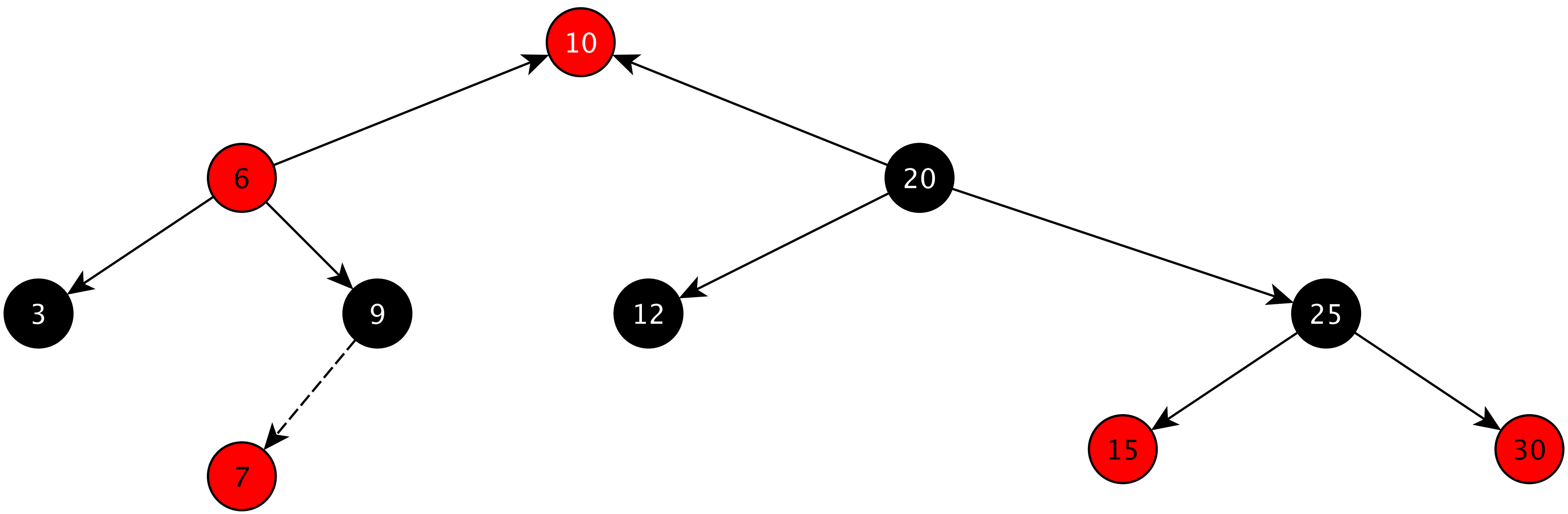
Black height still 2, color violation moved up

Red-Black Tree Insertion



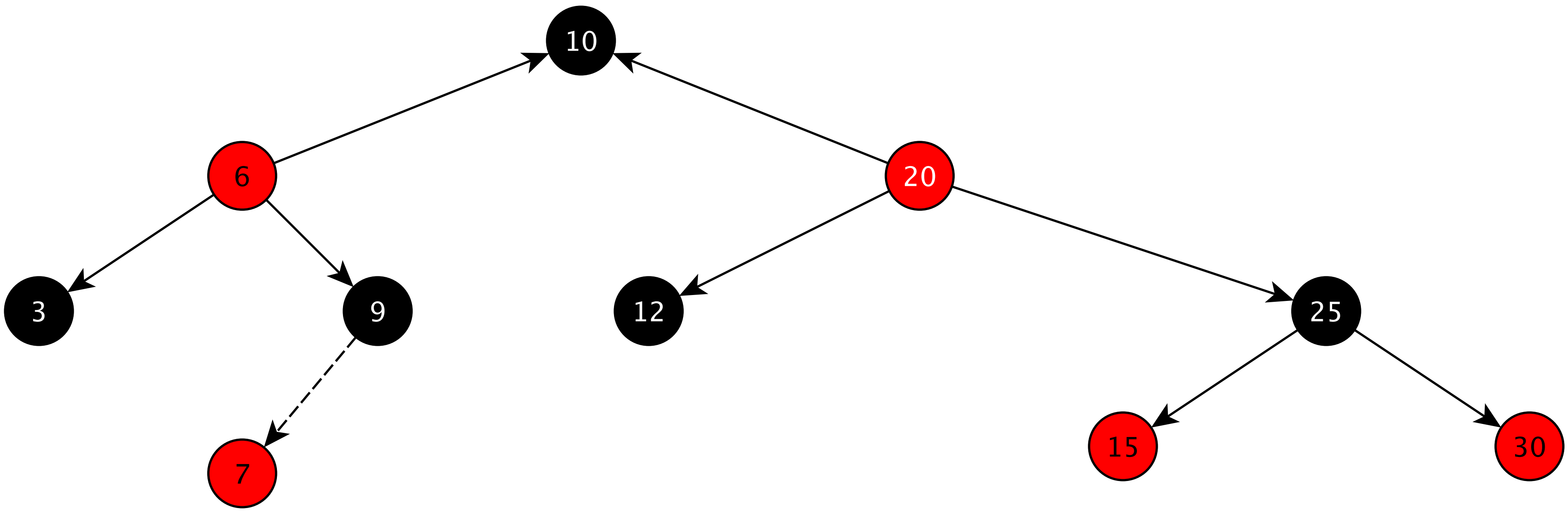
Left rotation at 6, about to do right rotation

Red-Black Tree Insertion



Right rotation at 20, black height broken, need to recolor

Red-Black Tree Insertion



Color conditions restored, black-height restored.

Balanced BSTs: What to Know

- You can keep a BST of height O(log n)
  - O(log n) insert, add, delete time
  - Reasonably efficient implementation
- AVL and red/black trees are balanced
- Rotations
- How AVL and red/black trees work (high level)
- Why AVL and red/black trees are balanced

- Don't need to know rebalancing rules

Splay Trees

Splay trees are self-adjusting binary trees

- Each time a node is accessed, it is moved to root position via rotations
- No metadata at all. Just rotate up each element you access

Splay Trees

Splay trees are self-adjusting binary trees

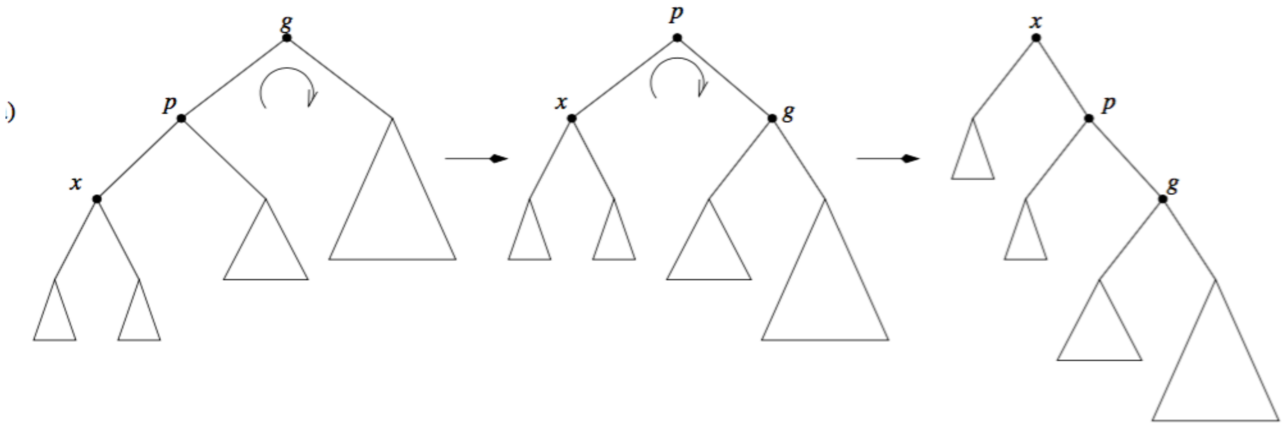
- Each time a node is accessed, it is moved to root position via rotations
- No guarantee of balance (or shallow height)
- But good *amortized* performance

Theorem: Any set of m operations (add, remove, contains, get) on an n-node splay tree take at most O(m log n) time.

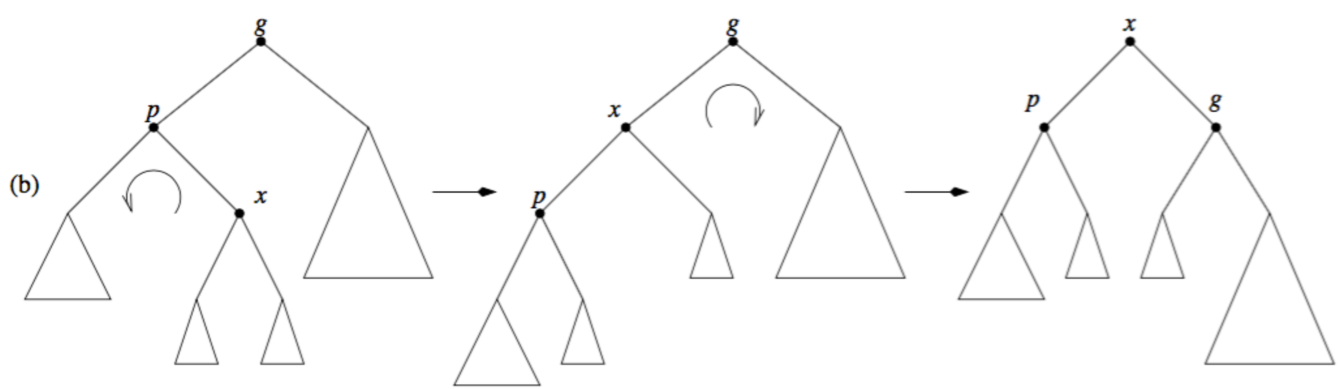
- As good as an AVL or Red-Black Tree!

Splay Tree Rotations

Right Zig-Zig Rotation (left version too)



Right Zig-Zag Rotation (left version too)



Specialized BSTs

- Sometimes I can make operations faster if I know something about the data
- What if I have n nodes in my tree, but I only ever access n' of them. How fast can I make accesses?
  - O(log n)
- What if I use my tree as a stack---I only remove the most recent thing I inserted?
  - O(1)

Dynamic Optimality

- Conjecture: For any sequence of access operations, if the best possible Binary Search Tree takes X operations, then a splay tree takes O(X) operations
- Essentially: keeping no metadata, and with no knowledge of the future, splay trees do as well as a specialized tree that knows the whole sequence in advance

Dynamic Optimality

- Conjecture: For any sequence of access operations, if the best possible Binary Search Tree takes X operations, then a splay tree takes O(X) operations

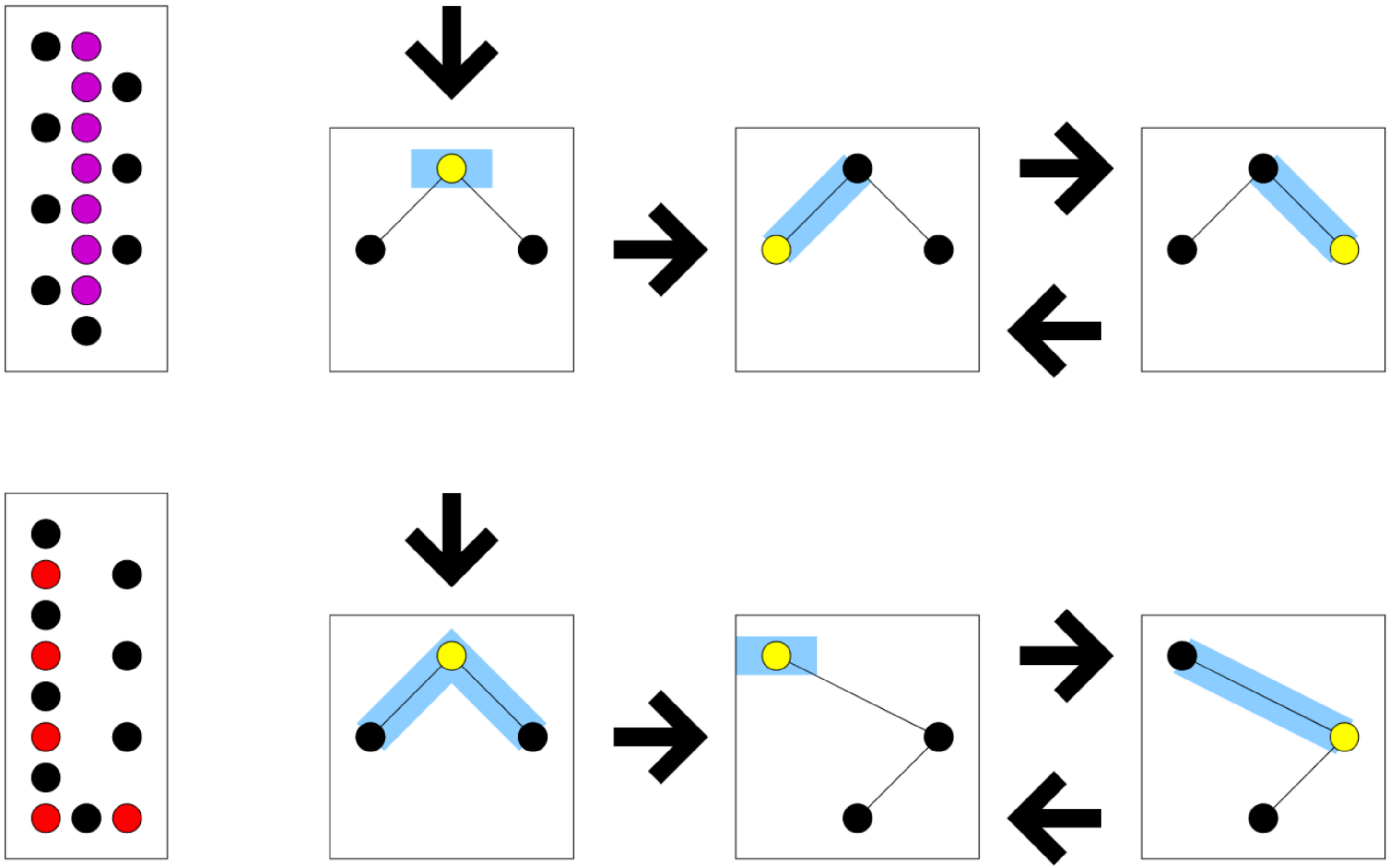
- One consequence would be: splay trees can handle stack or queue operations in O(1) average operations like a DLL

Dynamic Optimality

- Open since 1985
- Recent progress [Levy Tarjan 2019]: if a splay tree's performance only improves when we remove operations, then the splay tree is dynamically optimal

Dynamic Optimality

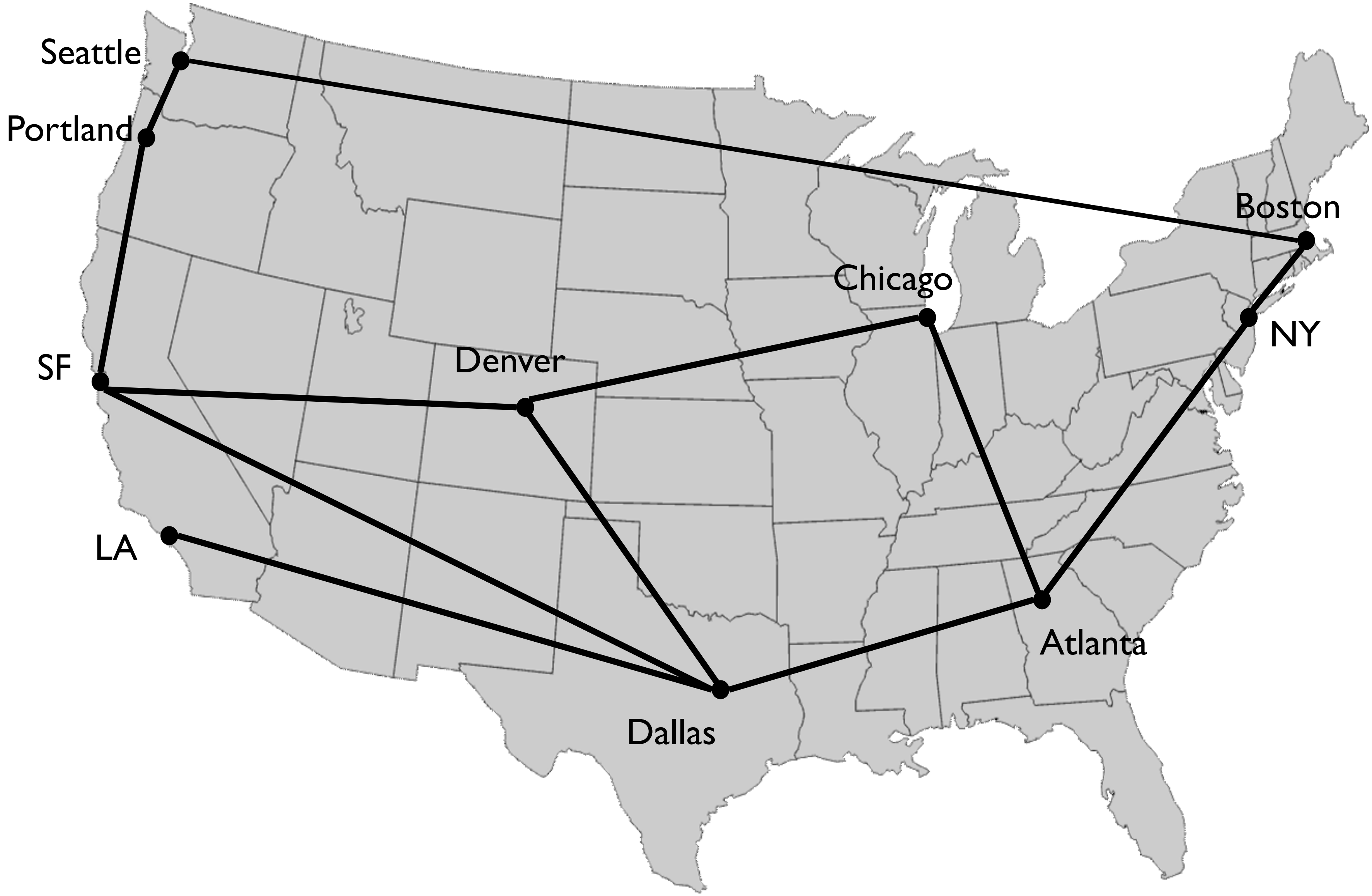
- Some really cool math in this area



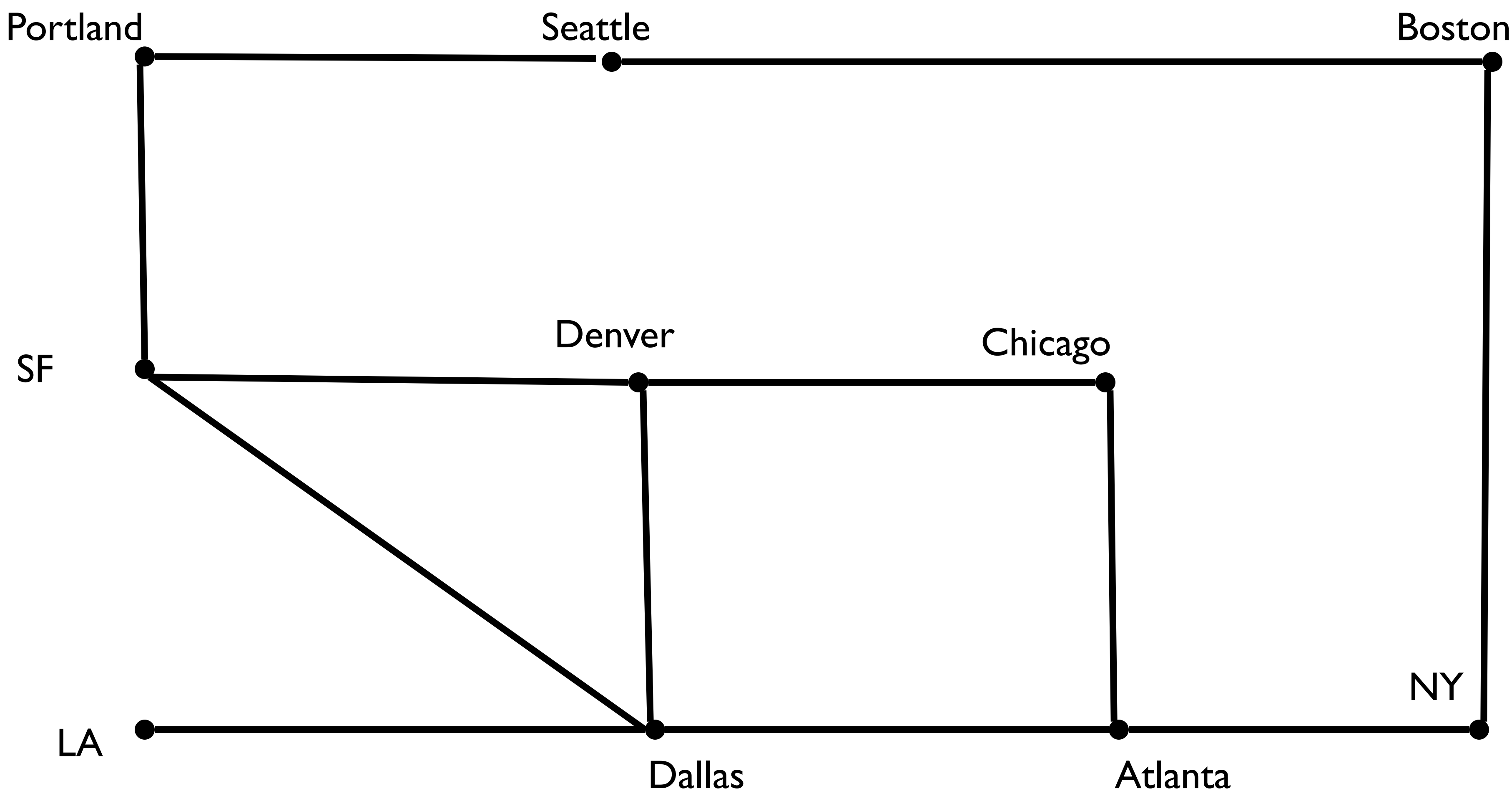
Graphs Describe the World

- Transportation Networks
- Communication Networks
- Molecular structures
- Dependency structures
- Scheduling
- Matching
- Graphics Modeling
- ....

Nodes = subway stops; Edges = track between stops

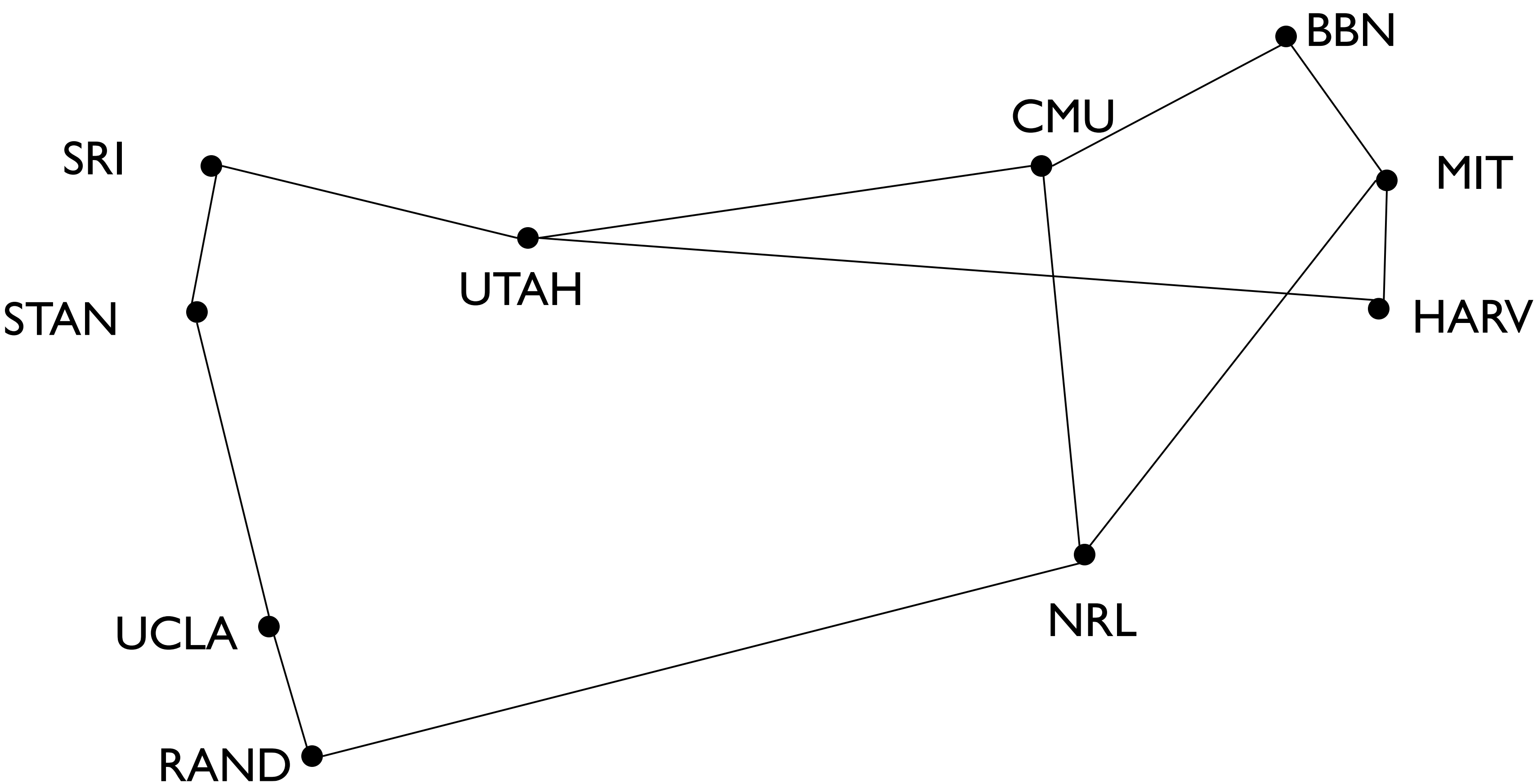


Nodes = cities; Edges = rail lines connecting cities

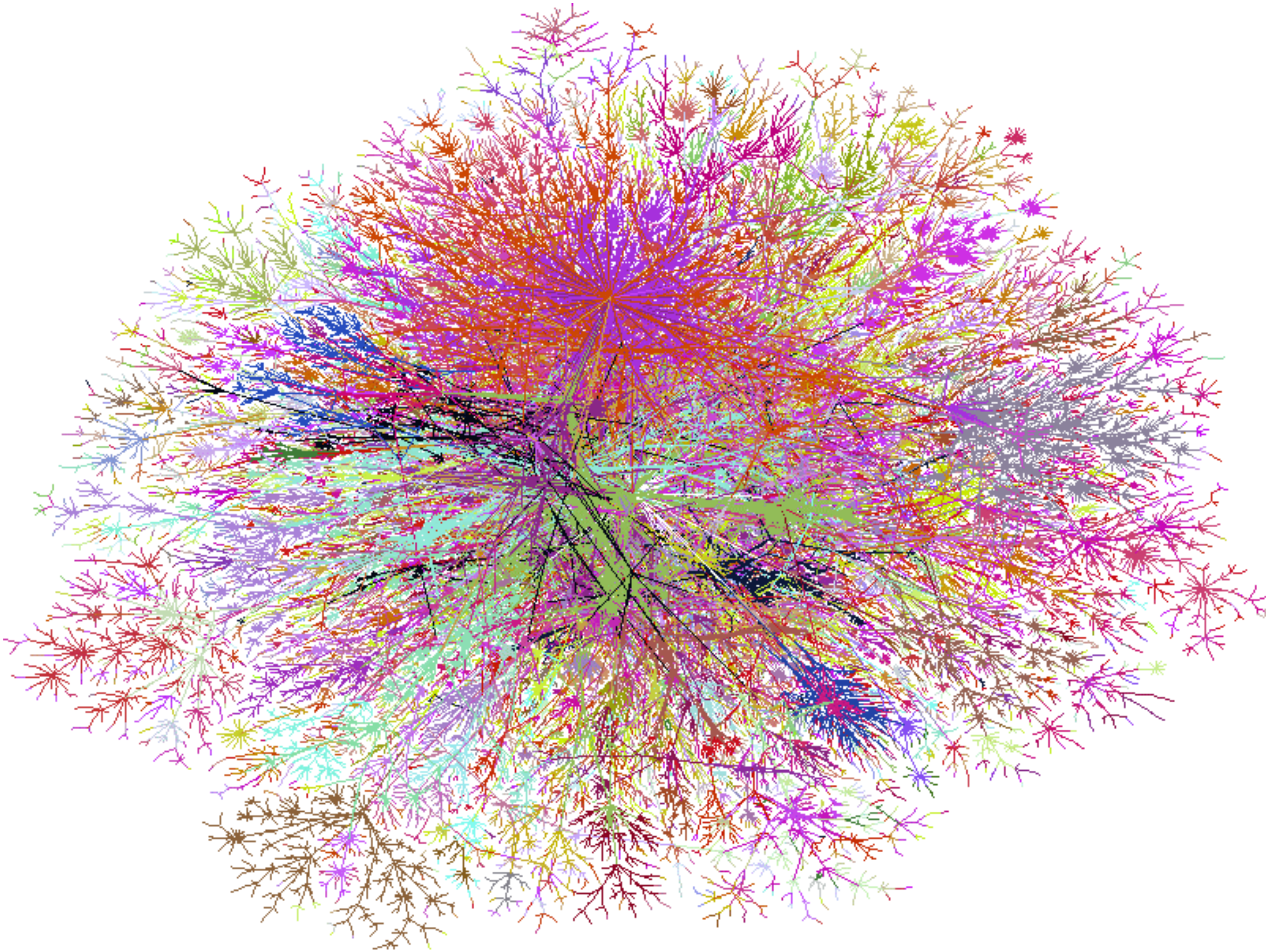


Note: Connections in graph matter, not precise locations of nodes

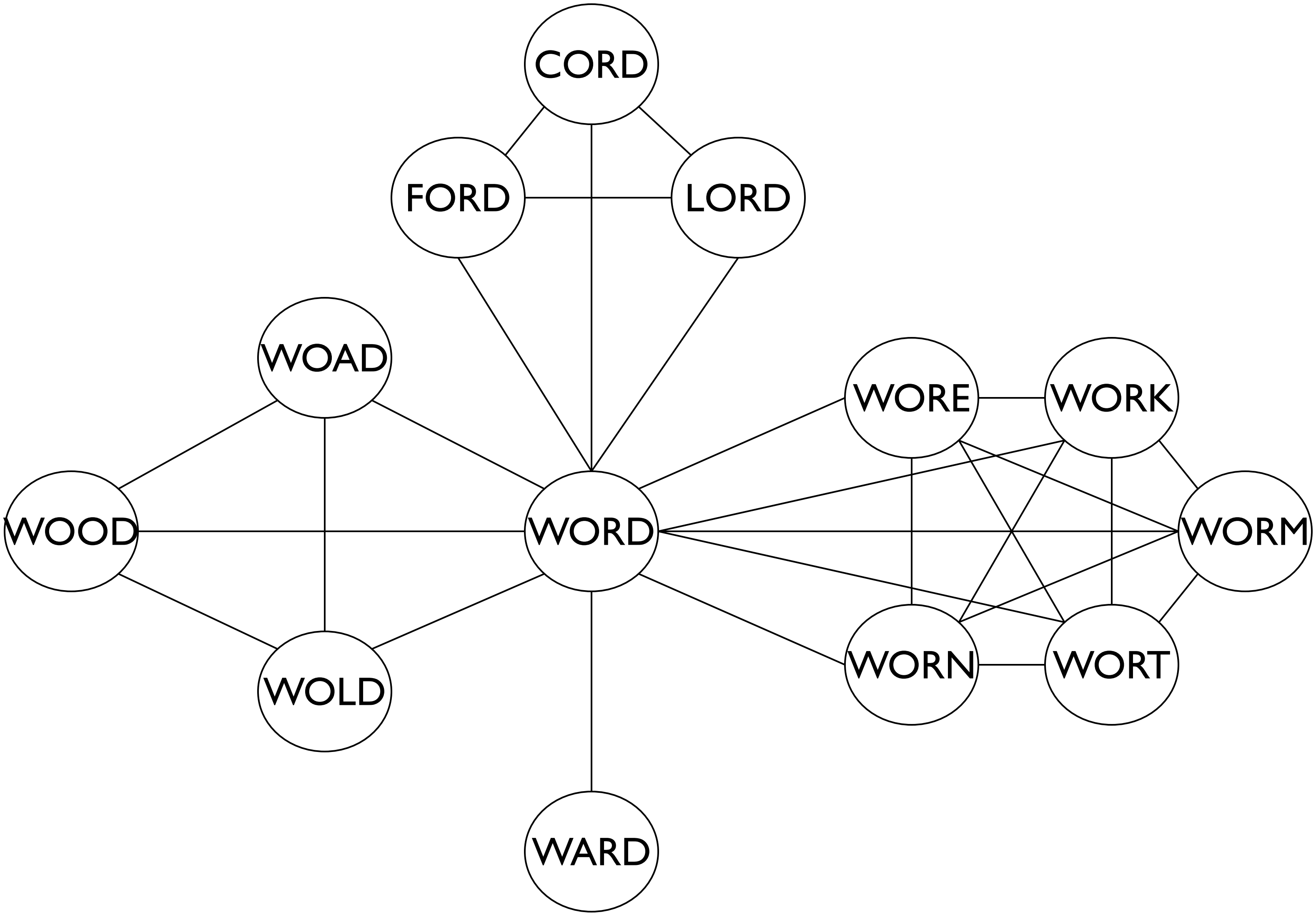
Internet (~1972)



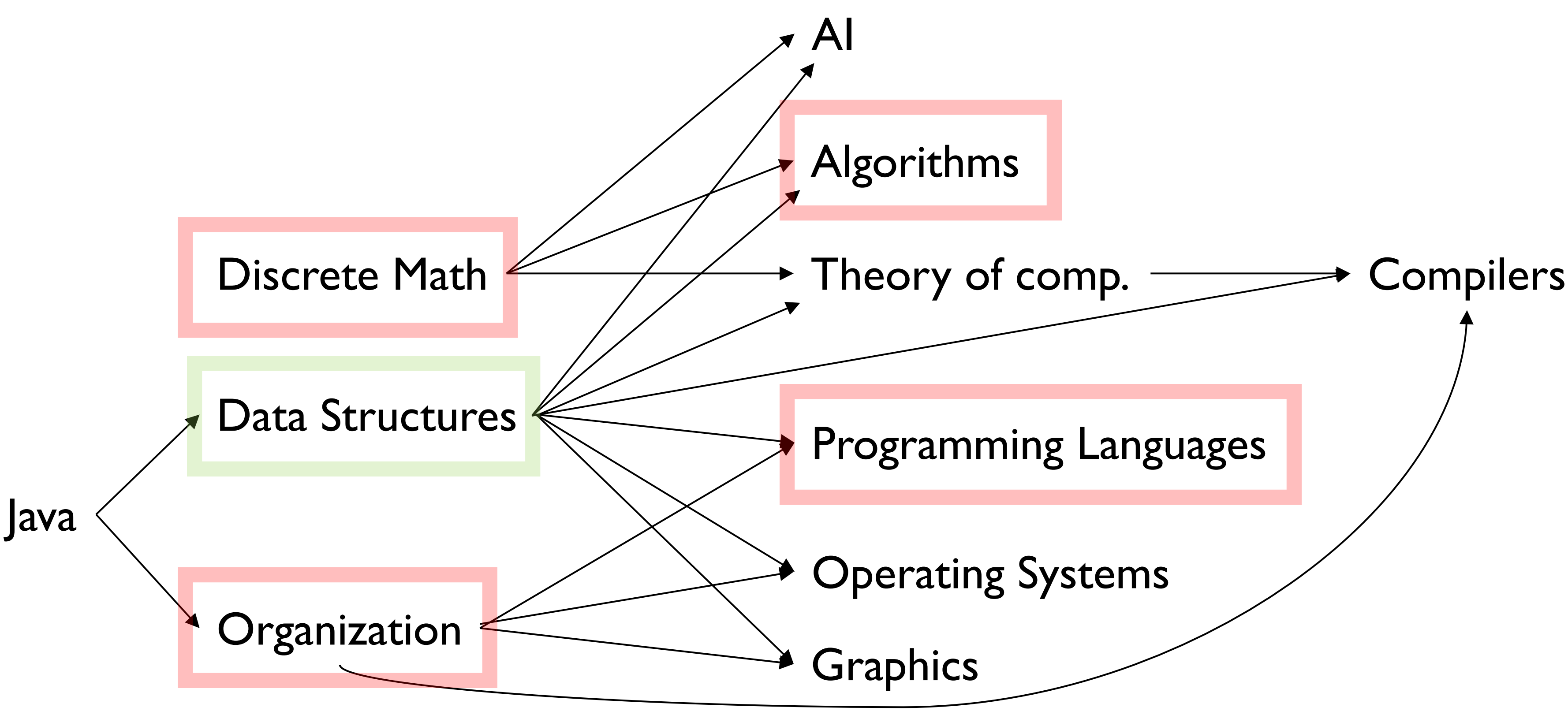
Internet (~1998)



Word Game

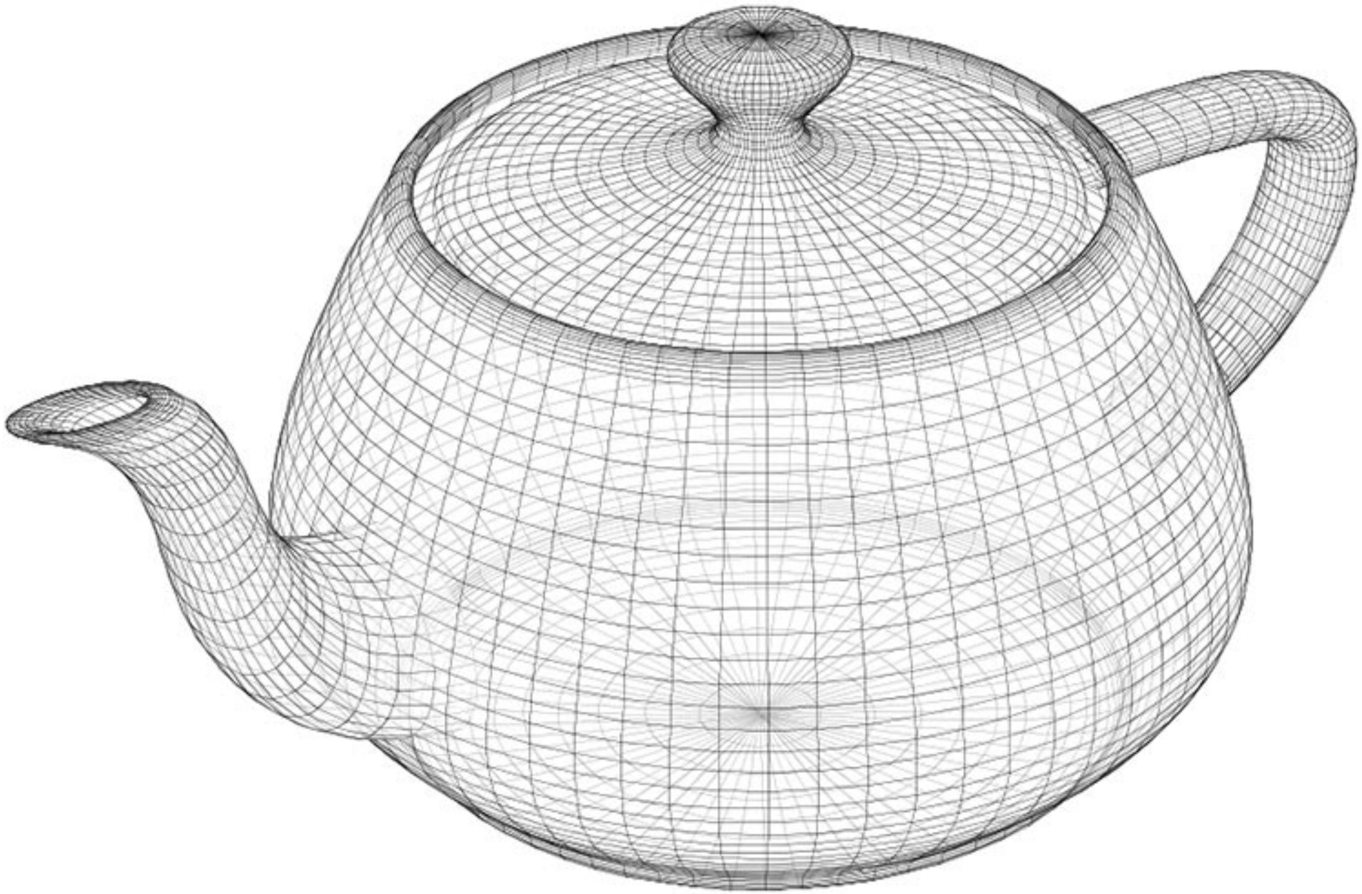

CORD
FORD
LORD
WOAD
WORE
WORK
WOOD
WORD
WORM
WORN
WORT
WOLD
WARD

CS Pre-requisite Structure (subset)

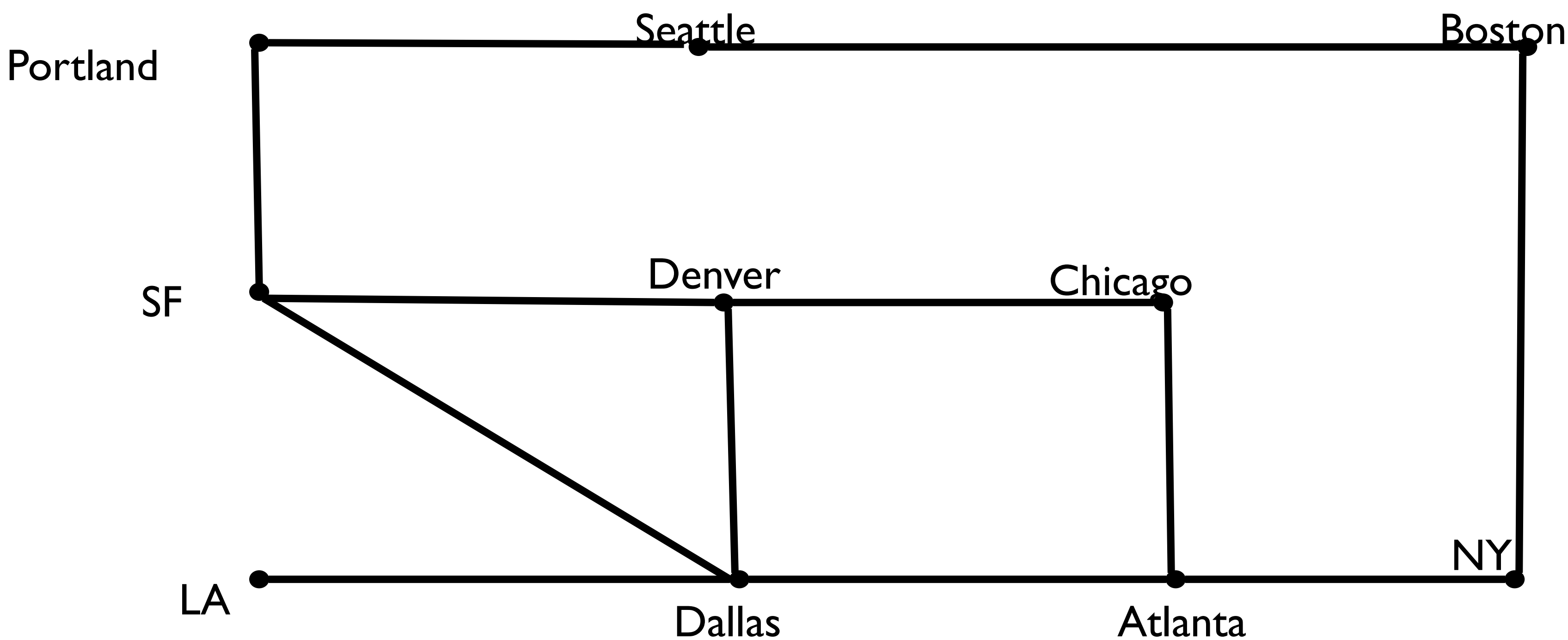


Nodes = courses; Edges = prerequisites ***

Wire-Frame Models



Basic Definitions & Concepts



Def'n: An *undirected graph* G = (V,E) consists of two sets

•V : the *vertices* of G, and E : the *edges* of G

•Each edge e in E is defined by a set of two vertices: its *incident vertices*. We write e = {u,v} and say that u and v are *adjacent.*

Walking Along a Graph

- A *walk from u to v* in a graph $G = (V,E)$ is an *alternating* sequence of vertices and edges

  $$u = v_0, e_1, v_1, e_2, v_2, \ldots , v_{k-1}, e_k, v_k = v$$

  such that each $e_i = \{v_i , v_{i+1}\}$ for $i = 1, \ldots , k$
- Note a walk starts and ends on a vertex
- If no *edge* appears more than once then the walk is called a *path*
- If no *vertex* appears more than once then the walk is a *simple path*

Walking In Circles

- A *closed walk* in a graph $G = (V,E)$ is a walk

$$v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k$$

such that each $v_0 = v_k$

- A *circuit* is a path where $v_0 = v_k$
  - No repeated edges
- A *cycle* is a *simple path* where $v_0 = v_k$
  - No repeated vertices (uhm, except for v_0!)
- The length of any of these is the number of *edges* in the sequence

Little Tiny Theorems

- If there is a walk from u to v, then there is a walk from v to u.
- If there is a walk from u to v, then there is a path from u to v (and from v to u)
- If there is a path from u to v, then there is a simple path from u to v (and v to u)
- Every circuit through v contains a cycle through v
- Not every closed walk through v contains a cycle through v! [Try to find an example!]

Another Useful Graph Fact

- Degree of a vertex *v*
  - Number of edges incident to *v*
  - Denoted by *deg(v)*
- Thm: For any graph $G = (V,E)$

$$\sum_{v \in V} \deg(v) = 2|E|$$

where $|E|$ is the number of edges in G

- Proof Hint: Induction on $|E|$: How does removing an edge change the equation?
  - Or: Count pairs *(v,e)* where *v* is incident with e

Reachability and Connectedness

- Def'n: A vertex v in G is *reachable* from a vertex u in G if there is a path from u to v
- v is reachable from u *iff* u is reachable from v
- Def'n: An undirected graph G is *connected* if for every pair of vertices u, v in G, v is reachable from u (and, of course, u from v)
- The set of all vertices reachable from v, along with all edges of G connecting any two of them, is called the *connected component of v*

Distance in Undirected Graphs

Def: The *distance* between two vertices *u* and *v* in an undirected graph $G=(V,E)$ is the minimum of the path lengths over all *u-v* paths.

- We write it as $d(u,v)$. It satisfies the properties
  - $d(u,u) = 0$, for all $u \in V$
  - $d(u,v) = d(v,u)$, for all $u,v \in V$
  - $d(u,v) \leq d(u,w) + d(w,v)$, for all $u,v,w \in V$
- This last property is called the *triangle inequality*

Algorithms on Graphs

- What are the basic operations we need to describe algorithms on graphs?
  - Given vertices u and v: are they *adjacent*?
  - Given vertex v and edge e, are they *incident*?
  - Given an edge e, get its incident vertices (*ends*)
  - How many vertices are adjacent to v? (*degree* of v)
    - The vertices adjacent to v are called its *neighbors*
  - Get a list of the vertices *adjacent* to v
    - From which we can get the edges *incident* with v

Basic Graph Algorithms

- We'll look at a number of graph algorithms
  - Connectedness: Is G connected?
    - If not, how many connected components does G have?
  - Cycle testing: Does G contain a cycle?
    - Does G contain a cycle through a given vertex?
  - If the edges of G have costs:
    - What is the cheapest connected subgraph of G that contains every vertex?
    - What is a cheapest path from u to v?
  - And more....

Testing Connectedness

- How can we determine whether G is connected?
  - Pick a vertex v; see if every vertex u is reachable from v
- How could we do this?
  - Visit the neighbors of v, then visit their neighbors, etc. See if you reach all vertices
    - Assume we can mark a vertex as "visited"
- How do we *efficiently* manage all this visiting?

Reachability: Breadth-First Search

```
BFS(G, v)       // Do a breadth-first search of G starting at v
// pre: all vertices are marked as unvisited
count ←0;
Create empty queue Q; enqueue v; mark v as visited; count++
While Q isn't empty
        current ←Q.dequeue();
        for each unvisited neighbor u  of current :
                add u to Q; mark u as visited; count++
return count;
```

Now compare value returned from BFS(G,v) to size of V

BFS Theorem

Thm. BFS(G,v) visits exactly those vertices u reachable from v.

Proof: We'll show that if u is reachable from v then BFS(G,v) visits u by induction on d = d(v,u)

- Base Case: d = 0. Then u = v.
  - v is reachable from v and BFS(G,v) visits v
- Induction Hypothesis: For some $d \geq 0$, if d(u,v) = d then BFS(G,v) visits u.

BFS Theorem

- Induction Step: Assume now that $d(u,v) = d+1$
  - Let $v = v_0, e_1, v_1, e_2, v_2, \ldots, v_d, e_{d+1}, v_{d+1} = u$ be a path of length $d+1$ from v to u
  - Then $v = v_0, e_1, v_1, e_2, v_2, \ldots, v_d$ is a path of length d from v to v_d
  - By I.H., v_d is visited by BFS(G,v) and put in Q
  - So v_d will be dequeued and all of its unvisited neighbors, including u, will be marked as visited

A similar argument shows that if u is visited by BFS(G,v) then u is reachable from v

BFS Reflections

- The BFS algorithm traced out a tree T_v: the edges connecting a visited vertex to (as yet) unvisited neighbors
- T_v is called a *BFS tree of G with root v* (or *from v*)
- The vertices of T_v are visited in *level-order*
- Every path in T_v from v to a vertex u is a *shortest possible path* from v to u
  - That is the path as length d(v,u)