CSCI 136
Data Structures &
Advanced Programming

Jeannie Albrecht
Lecture 30
May 2, 2014

Administrative Details

- Darwin lab
  - Final lab + creature due Monday at noon
  - Any questions/problems?
- How do you infect another creature?
  - Make sure you don't create new creatures…just change the species of an existing one

Last Time (Monday)

- Continued talking about BSTs
- Learned how to add elements to a BST

Today's Outline

- Wrap up binary search trees
- Maybe start talking about Graphs (Ch 16)
  - Learn a bit more about graphs during next lab

Recap: locate

```
protected BT<E> locate(BT<E> top, E value) {
  // pre: top and value are non-null
  // post: returns "highest" node with the desired value,
  //       or node to which value should be added
  E topValue = top.value();
  BT<E> child;
  // found at top: done
  if (topValue.equals(value)) return top;
  // look left if less-than, right if greater-than
  if (ordering.compare(topValue,value) < 0) {
      child = top.right();
  } else {
      child = top.left();
  }
  // if no child there: not in tree, return this node,
  if (child.isEmpty()) { return top; }
  // else keep searching
 else { return locate(child, value); }
}
```

Recap: contains

```
public boolean contains (E value) {
  if (root.isEmpty()) return false;

  BinaryTree<E> node = locate(root, value);
  return node.value().equals(value);
}
```

Recap: add

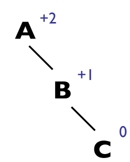
```
public void add(E value) {
  BT<E> newNode = new BT<E>(value);
  if (root.isEmpty()) { root = newNode; }
  else {
   BT<E> node = locate(root,value);
   E nodeValue = node.value();
    // node is either successor or predecessor of newNode
    if (ordering.compare(nodeValue,value) < 0) {
        //locate returned predecessor; add as right child
        node.setRight(newNode);
    } else {  //locate returned successor
      if (!node.left().isEmpty()) {
        // duplicate: if value is in tree, we insert before it
        predecessor(node).setRight(newNode);
      } else {
        node.setLeft(newNode);
      }
    }
  }
  count++;
}
```

Removal

- Removing the root is the hardest
- Let's figure that out first
  - If we figure out how to remove the root, we can remove any element in BST in same way (why?)
- We need to implement:
  - `public E remove(E item)`
  - `protected BT<E> removeTop(BT<E> top)`

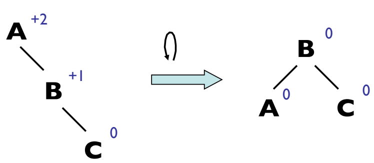
Food for thought…

- Can we design a binary search tree that is always balanced?
- Yes!
- AVL trees
  - Named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published a paper about AVL trees in 1962 called "An algorithm for the organization of information"



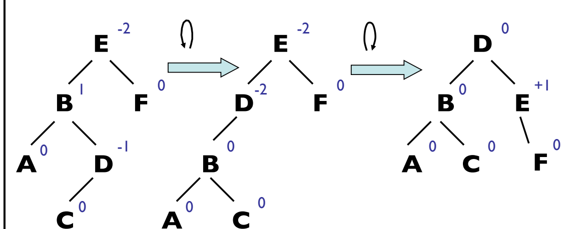
- The balance factor of a node is the *height* of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced.
- A node with any other balance factor is considered unbalanced and requires rebalancing the tree.

Single Rotation



Unbalanced trees can be rotated to achieve balance.

Double Rotation



Moving on…

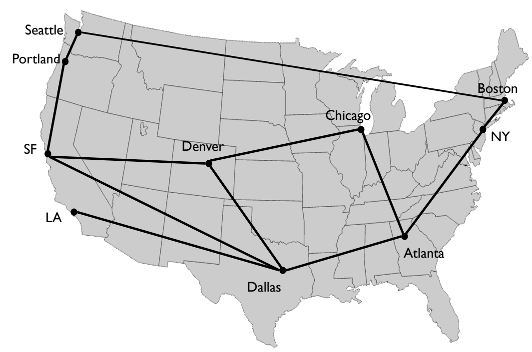
- You won't be tested on AVL trees
- Any questions on BSTs before we move on to graphs?

Introduction to Graphs

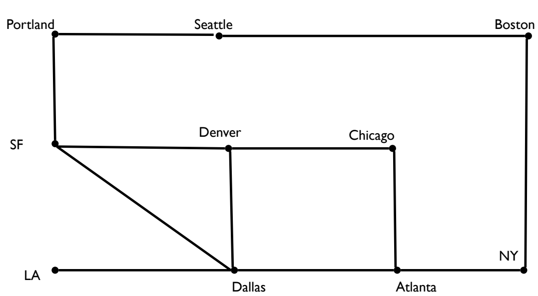
- Types of data structures
  - Basic - Lists/Vectors (no ordering relation)
  - Linear - ordered by insertion
  - Ordered - value ordering
  - Tree - hierarchical ordering
  - BST - value ordering (in a hierarchical fashion)
- Next up: Graphs
  - The most general way to describe relationships between data

Graphs

- Definition
  - A graph is a collection of vertices (nodes) and edges connecting them
- Examples?



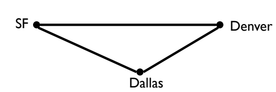
Nodes = cities; Edges = lines connecting cities



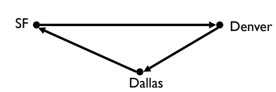
Note: Structure of graph matters, not actual placement of nodes

Types of Graphs

- Undirected
  - All edges are bi-directional

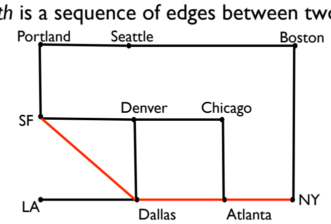


- Directed
  - Edges have a source and destination



Paths

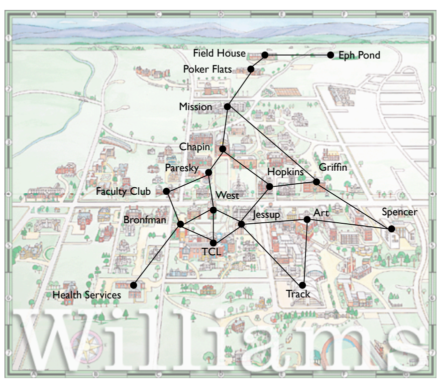
- A *path* is a sequence of edges between two nodes



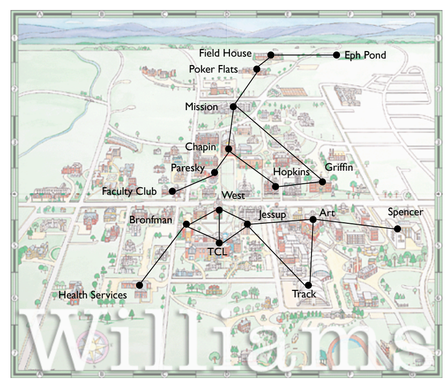
- Questions:
  - What is the shortest path from SF to NY?
  - What is the shortest *cycle* from SF to SF that goes through Atlanta and Chicago?

Connectedness

- Nodes U and V are *connected* if there is an edge between U and V
- A *connected component* is a set S where there is a path between every pair of vertices in S
  - A *fully connected component* is a set S where there is an edge between every pair of vertices in S



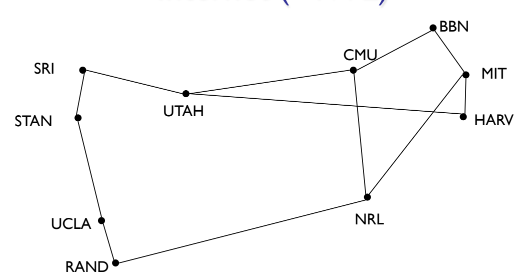
What's *reachable* from TCL?



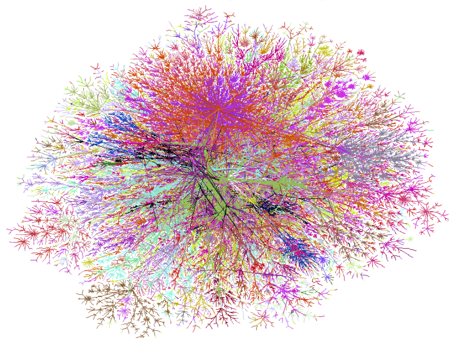
Graph Applications

- Connectedness in the real world
  - Flights, campus, networks, etc.
  - Useful for finding shortest number of steps/hops

Internet (~1972)

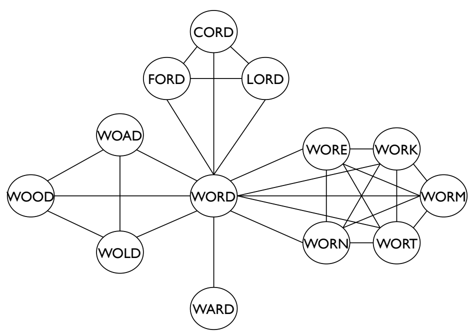


Internet (~1998)



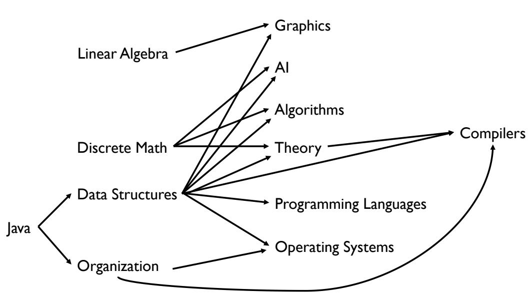
Graph Applications

- Connectedness in the real world
  - Flights, campus, networks, etc.
  - Useful for finding shortest number of steps
- Word "changlings"
  - Change in degree, paths, size, etc.



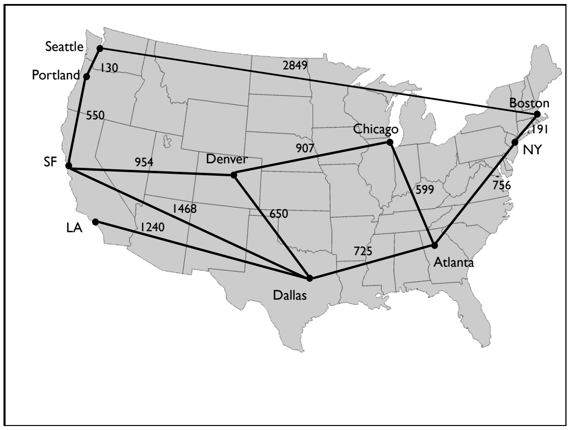
Graph Applications

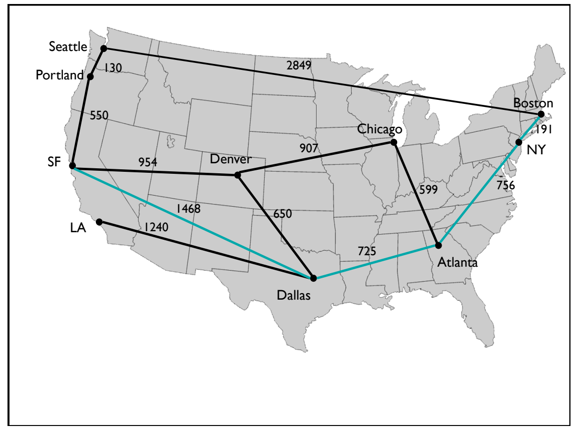
- Connectedness in the real world
  - Flights, campus, networks, etc.
  - Useful for finding shortest number of steps
- Word "changlings"
  - Change in degree, paths, size, etc.
- Schedules
  - In edges/out edges indicate prerequisite relationships (why no cycles?)



Labeled Edges

- Not all edges are the same "weight"
  - Edges can carry extra info
- Weight = the *cost* of traversing that edge
  - Cost may be a function of time, distance, price to pay, etc.
- May lead to different solutions to previously answered questions
  - What is shortest path between SF and NY given edge weights?


Seattle
130
Portland
550
SF
954
Denver
2849
907
Chicago
Boston
191
NY
599
756
1468
650
LA
1240
725
Atlanta
Dallas


Seattle
130
Portland
550
SF
954
Denver
2849
907
Chicago
Boston
191
NY
599
756
1468
650
LA
1240
725
Atlanta
Dallas