CSE373: Data Structures & Algorithms
Lecture 28: Final review and class wrap-up

Catie Baker

Spring 2015

Final Exam

As also indicated on the web page:

- Bring your student ID
- Next Tuesday, 2:30-4:20 in this room
- Cumulative but topics post-midterm about 2/3 of the questions
- See information on course web-page
- Not unlike the midterms in style, structure, etc.

Terminology

- Abstract Data Type (ADT)
  - Mathematical description of a “thing” with set of operations
  - Not concerned with implementation details
- Algorithm
  - A high level, language-independent description of a step-by-step process
- Data structure
  - A specific organization of data and family of algorithms for implementing an ADT
- Implementation of a data structure
  - A specific implementation in a specific language

Asymptotic and Algorithm Analysis

1. Add up time for all parts of the algorithm

   e.g. number of iterations = $(n^2 + n)/2$
2. Eliminate low-order terms i.e. eliminate n: $(n^2)/2$
3. Eliminate coefficients i.e. eliminate 1/2: (n^2)

Examples:

- $4n + 5$ = O(n)
- $0.5n \log n + 2n + 7$ = O(n log *n)*
- $n^3 + 2^n + 3n$ = $O(2^n)$
- $n \log (10n^2)$
  - 2n log (10n) = O(n log *n)*

Amortized Analysis

- In amortized analysis, the time required to perform a sequence of data structure operations is averaged over all the operations performed.

- Typically used to show that the average cost of an operation is small for a sequence of operations, even though a single operation can cost a lot

The Queue ADT

- Operations

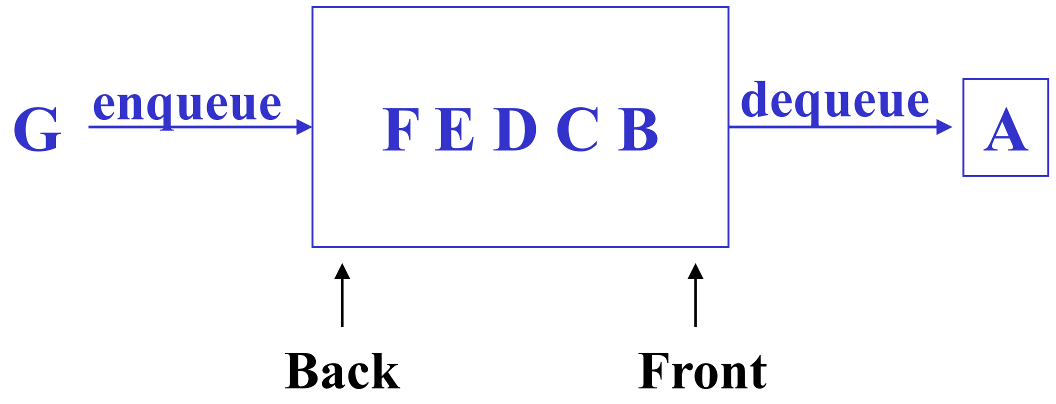
  `create`

  `destroy`

  `enqueue`

  `dequeue`

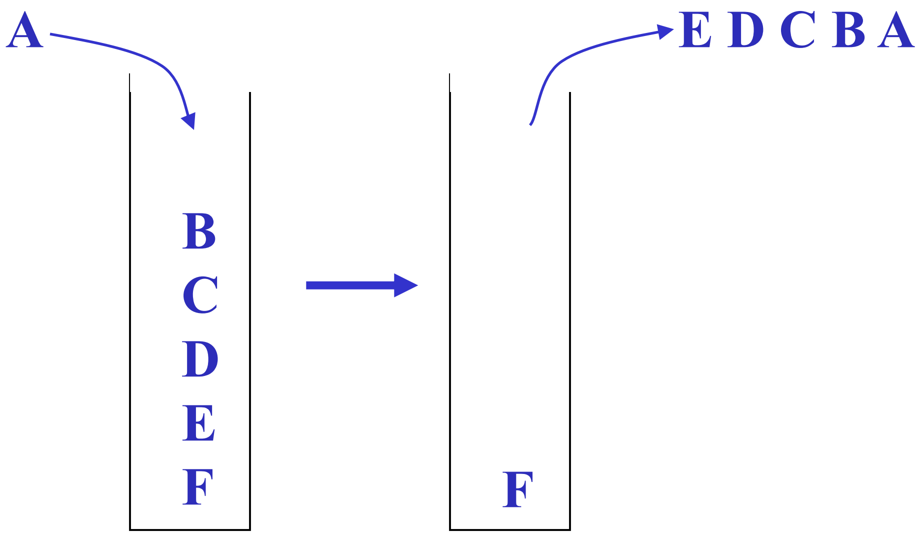
  `is_empty`



The Stack ADT

Operations:

```
create
destroy
push
pop
top
is_empty
```



The Dictionary (a.k.a. Map) ADT

- Data:
  - set of (key, value) pairs
  - keys must be comparable

- Operations:
  - **`insert(key,value)`**
  - **`find(key)`**
  - **`delete(key)`**
  - …

insert(catie, ….) →

← **find(rama)**

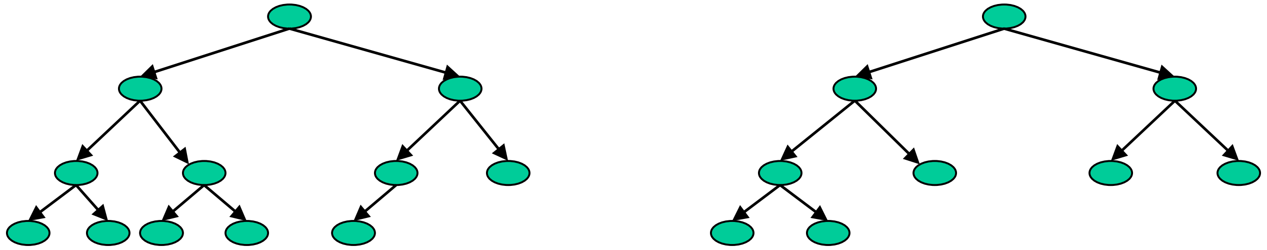
Rama Gokhale, …

- **catie**
  Catie Baker
  OH: Wed 11.00-12.00
  …
- **rama**
  Rama Gokhale
  OH: Fri 3.30-4.30
  …
- **conrad**
  Conrad Nied
  OH: Wed 4:00-5:00
  …

Will tend to emphasize the keys;
don't forget about the stored values

Trees

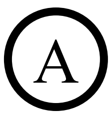
- Binary tree: Each node has at most 2 children (branching factor 2)
- *n*-ary tree: Each node has at most *n* children (branching factor *n*)
- Perfect tree: Each row completely full
- Complete tree: Each row completely full except maybe the bottom row, which is filled from left to right



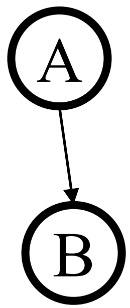
Tree Calculations

Recall: Height of a tree is the maximum number of edges from the root to a leaf.

What is the height of this tree?



Height = 0



Height = 1

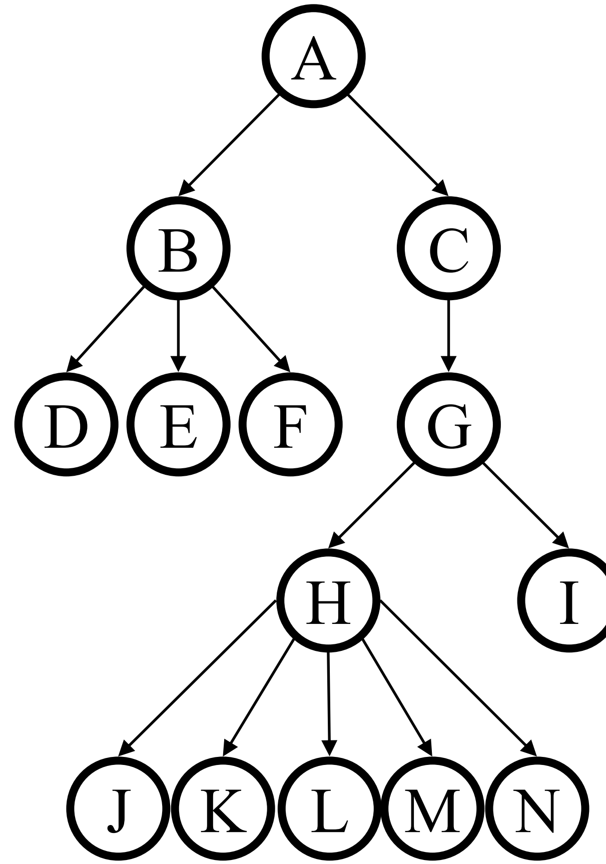
What is the depth of node G?

Depth = 2

What is the depth of node L?

Depth = 4

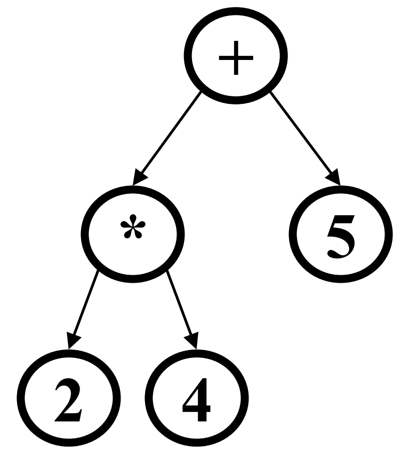
Height = 4



Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

- *Pre-order*: root, left subtree, right subtree
  + * 2 4 5
- *In-order*: left subtree, root, right subtree
  2 * 4 + 5
- *Post-order*: left subtree, right subtree, root
  2 4 * 5 +

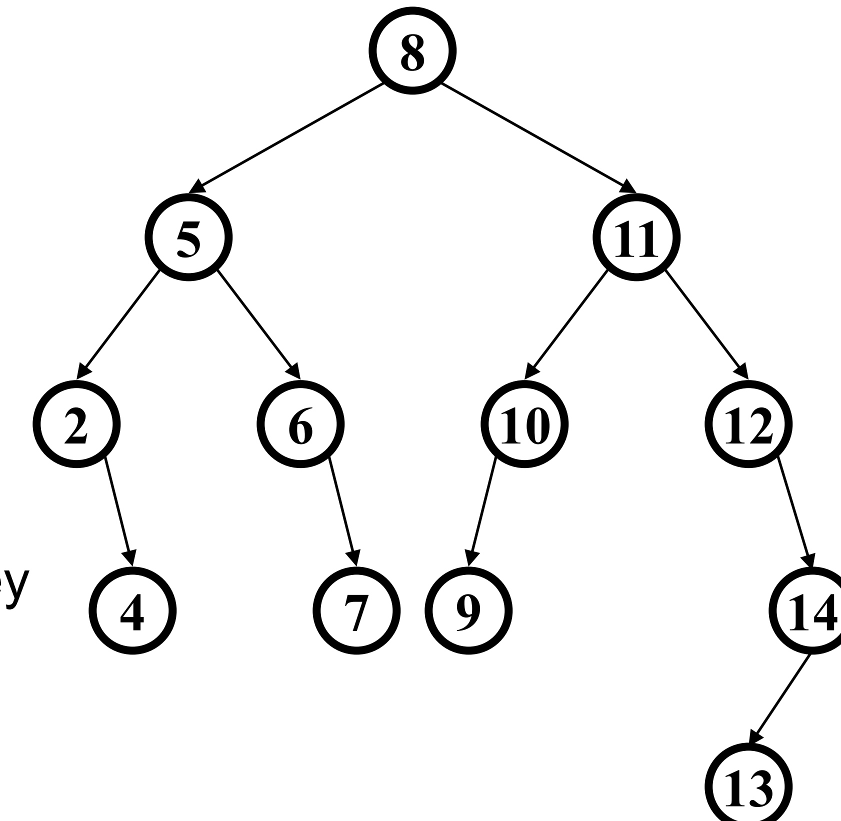


(an expression tree)

Binary Search Tree (BST) Data Structure

- Structure property (binary tree)
  - Each node has ≤ 2 children
  - Result: keeps operations simple

- Order property
  - All keys in left subtree smaller than node's key
  - All keys in right subtree larger than node's key
  - Result: easy to find any given key



- Operations
  - Find, insert, delete, BuildTree

The AVL Tree Data Structure

An AVL tree is a self-balancing binary search tree.

Structural properties

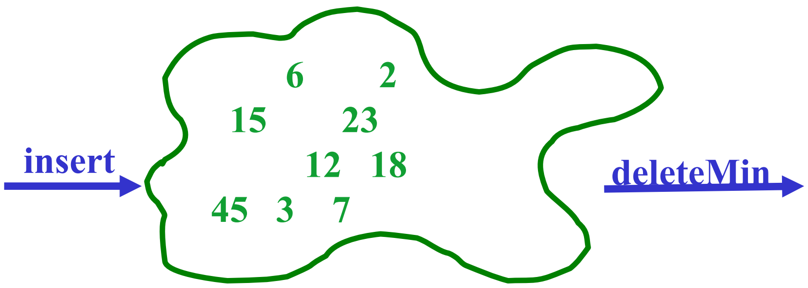
1. Binary tree property (same as BST)
2. Order property (same as for BST)
3. Balance property:
   balance of every node is between -1 and 1

Result: **Worst-case** depth is O(log *n*)

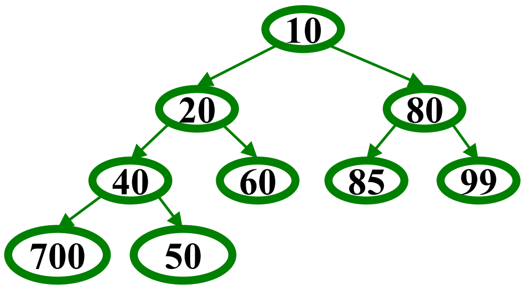
- **Operations**
  - **`find`**
  - **`insert`**: First BST **`insert`**, *then* check balance and potentially “fix” the AVL tree (4 cases).

Priority Queues and Binary Heaps

- Priority Queue ADT:
  - **`insert`** comparable object,
  - **`deleteMin`**



- Binary heap data structure:
  - Complete binary tree
  - Each node has less important priority value than its parent



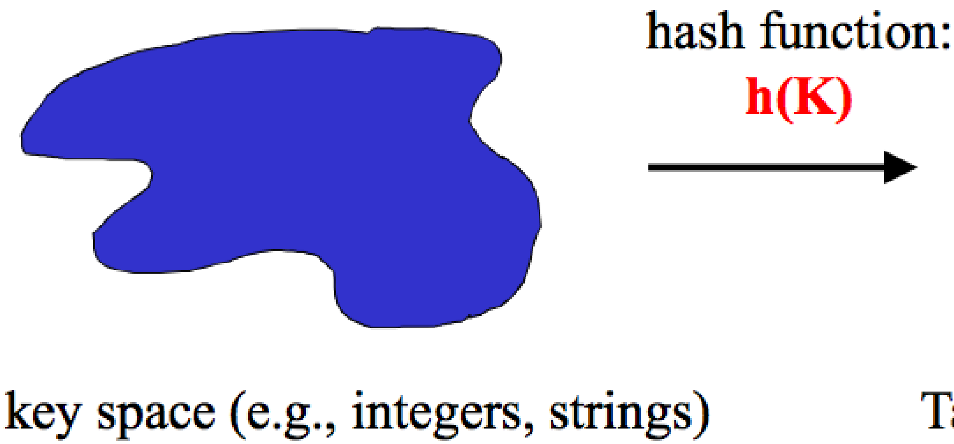
- **`insert`** and **`deleteMin`** operations = O(height-of-tree)=O(**`log`** n)
  - **`insert`**: put at new last position in tree and percolate-up
  - **`deleteMin`**: remove root, put last element at root and percolate-down

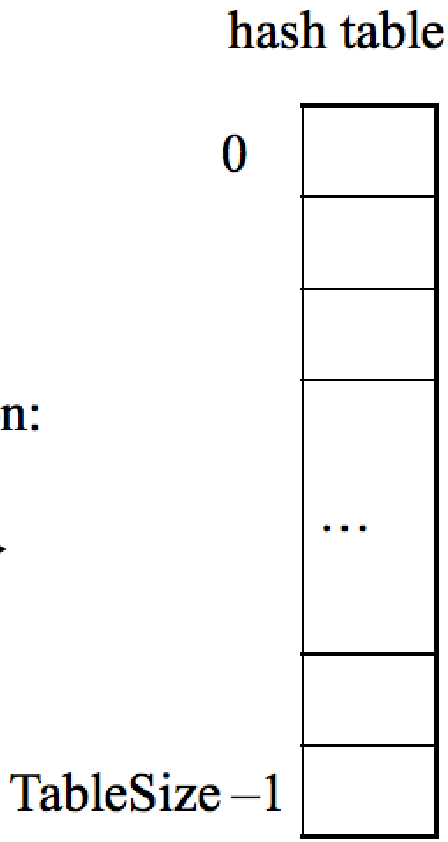
Union-Find ADT

- Given an unchanging set *S*, **`create`** an initial partition of a set
  - Typically each item in its own subset: {a}, {b}, {c}, …
  - Give each subset a "name" by choosing a *representative element*
- Operations
  - **`find`** takes an element of *S* and returns the representative element of the subset it is in
  - **`union`** takes two subsets and (permanently) makes one larger subset
- Up-tree data structure
  - With path compression and union by size

Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:





- Collision: when two keys map to the same location in the hash table.
- Two ways to resolve collision:
  - Separate chaining
  - Open Addressing (linear probing, quadratic probing, double hashing.)

Memory Locality

- Temporal Locality (locality in time)
  - If an item (a location in memory) is referenced, **that same location** will tend to be referenced again soon.

- Spatial Locality (locality in space)
  - If an item is referenced, items **whose addresses are close by** tend to be referenced soon.

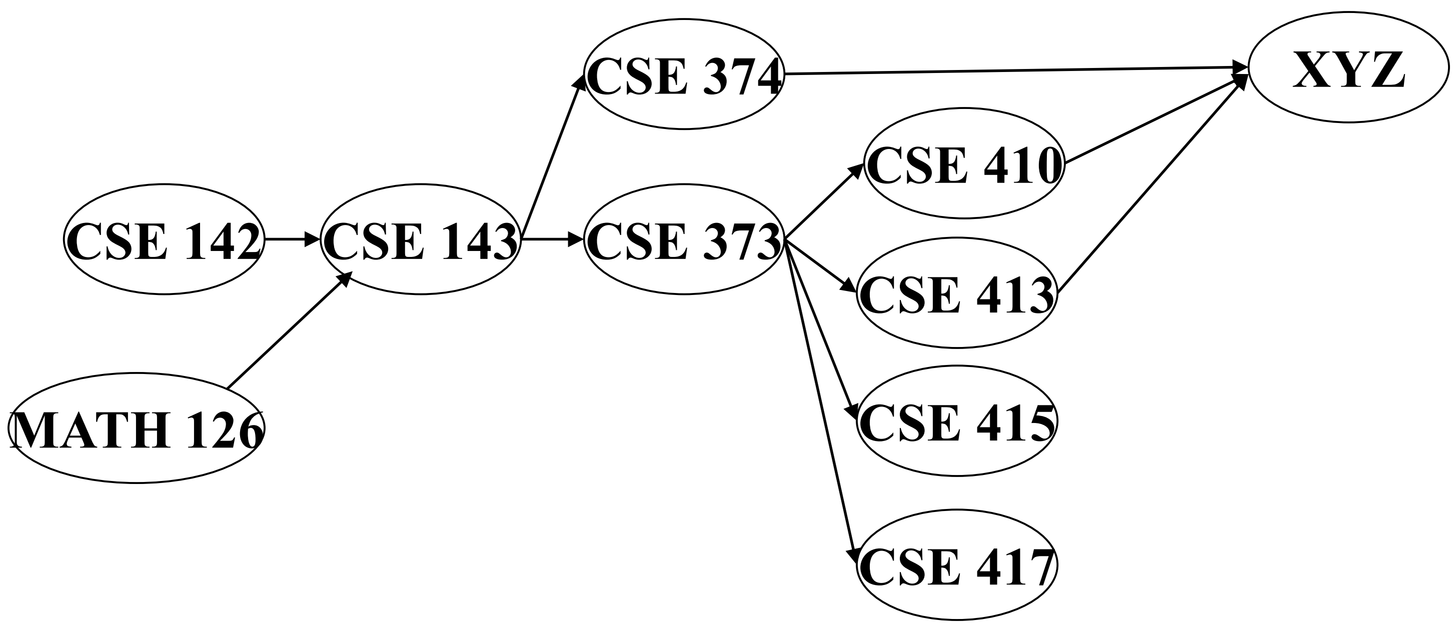
Graphs

- Vertex, node, edge
- Directed, undirected
- Weighted, unweighted
- Connected, disconnected, strongly/weakly connected
- Paths, cycles
- DAGs

- Adjacency lists and matrices

Topological Sort

Problem: Given a DAG **`G=(V,E)`**, output all vertices in an order such that no vertex appears before another vertex that has an edge to it



One example output:

126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415

Graph Traversals

For an arbitrary graph and a starting node **v**, find all nodes *reachable* from **v** (i.e., there exists a path from **v**)

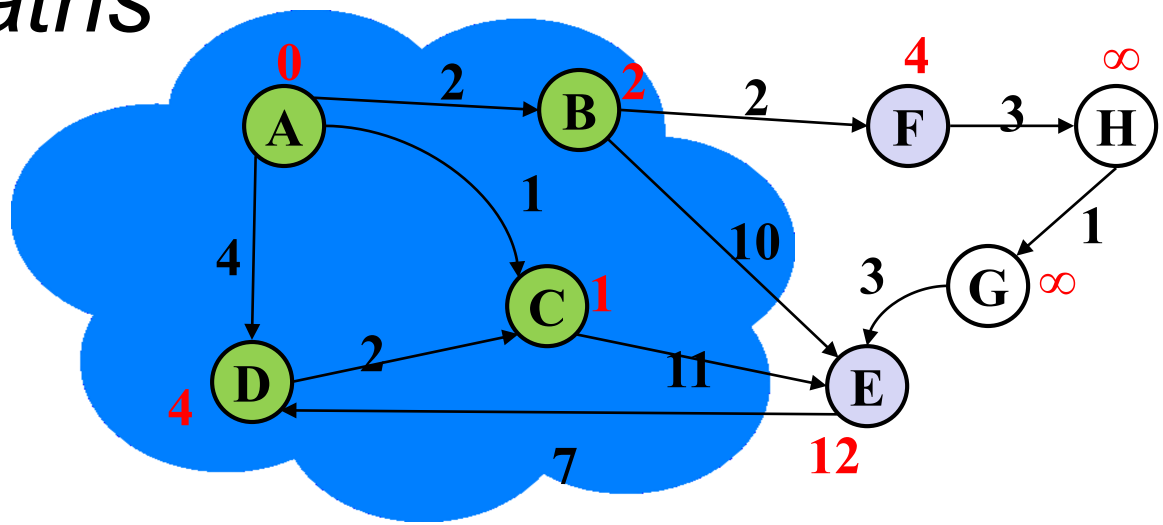
Basic idea:

- Keep following nodes
- But "mark" nodes after visiting them, so the traversal terminates and processes each reachable node exactly once

Important Graph traversal algorithms:

- "Depth-first search" "DFS": recursively explore one part before going back to the other parts not yet explored
- "Breadth-first search" "BFS": explore areas closer to the start node first

Dijkstra's Algorithm: Lowest cost paths



- Initially, start node has cost 0 and all other nodes have cost ∞
- At each step:
  – Pick closest unknown vertex **v**
  – Add it to the "cloud" of known vertices
  – Update distances for nodes with edges from **v**
- That's it!

Minimum Spanning Trees

The minimum-spanning-tree problem

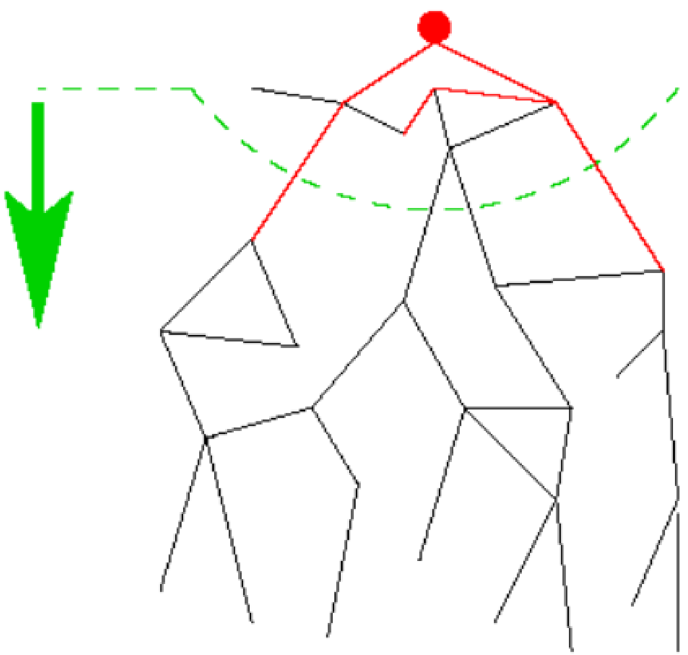
- Given a weighted undirected graph, compute a spanning tree of minimum weight

Given an undirected graph **G**=(**V**,**E**), find a graph **G'=(V, E')** such that:

- E' is a subset of E
- |E'| = |V| - 1
- G' is connected

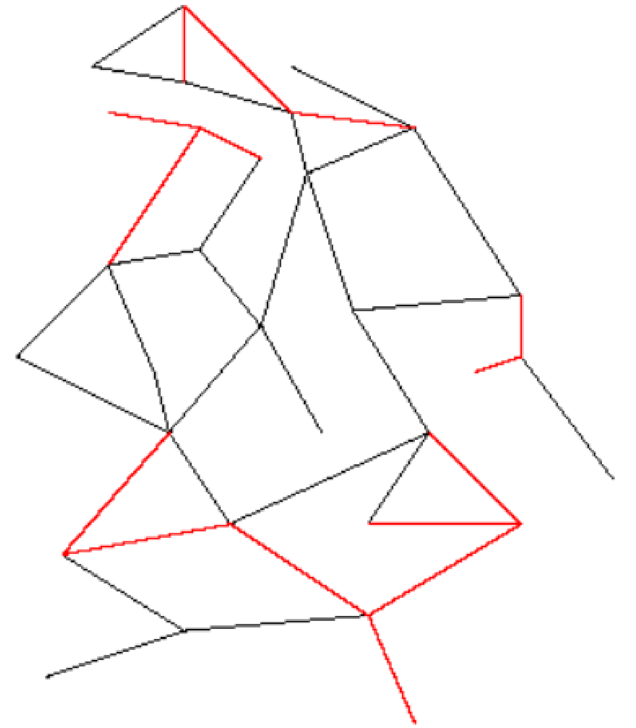
G' is a minimum spanning tree.

Two different approaches



Prim's Algorithm

Almost identical to Dijkstra's



Kruskals's Algorithm

Completely different!

Sorting: The Big Picture

Surprising amount of neat stuff to say about sorting:

- **Simple algorithms: O(n^2)**
  - Insertion sort
  - Selection sort
  - Shell sort
  - …
- **Fancier algorithms: O(n log n)**
  - Heap sort
  - Merge sort
  - Quick sort
  - …
- **Comparison lower bound: Ω(n log n)**
- **Specialized algorithms: O(n)**
  - Bucket sort
  - Radix sort
- **Handling huge data sets**
  - External sorting

Preserving Abstractions

- Need to deep-copy data passed into abstractions to avoid pain and suffering

- Need to deep-copy data passed out of abstractions to avoid pain and suffering (unless data is "new" or no longer used in abstraction)

- If objects are immutable (no way to update fields or things they refer to), then copying unnecessary

Algorithm Design Techniques

- Greedy (Shortest path, minimum spanning tree, …)
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
- Backtracking (A clever form of exhaustive search)
- P vs. NP (Know what it means for an algorithm to be in NP, in P.)
- Parallelism
  - Use threads to split work among many processors.

Phew! That's it.

- Good luck ☺

Victory Lap

A victory lap is an extra trip around the track

- By the exhausted victors (that's us) ☺

Review course goals

- Slides from Lecture 1
- What makes CSE 373 special

Thank you!

Big thank-you to your TAs

- Amazingly cohesive “big team”
- Prompt grading and question-answering
- Optional TA sessions weren’t optional for them!

Conrad Nied Yunyi Song Andy Li Rama Gokhale Luyi Lu Cyndi Ai Johnson Goh

Thank you!

And huge thank you to all of **you**

- Great attitude
- Showed up to class (most of the time)
- Occasionally laughed at stuff ☺

Now a few slides from Lecture 1

– Hopefully they make more sense now
– Hopefully we succeeded

Data Structures

- Introduction to Algorithm Analysis
- Lists, Stacks, Queues
- Trees, Hashing, Dictionaries
- Heaps, Priority Queues
- Sorting
- Disjoint Sets
- Graph Algorithms
- Introduction to Parallelism and Concurrency

Goals

- Be able to make good design choices as a developer, project manager, etc.
  - Reason in terms of the general abstractions that come up in all non-trivial software (and many non-software) systems
- Be able to justify and communicate your design decisions

You will learn the key abstractions used almost every day in just about anything related to computing and software.

- This is not a course about Java! We use Java as a tool, but the data structures you learn about can be implemented in any language.

Last slide

I had a lot of fun and learned a great deal this quarter.

You have learned the key ideas for organizing data, a skill that far transcends computer science.