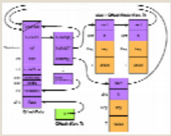


Data Structures

Richard Anderson
University of Washington

Today's topics

- Teaching Data Structures
- Active Learning in Data Structures
- Big Ideas: Average Case Analysis
- Research discussion

Re-revised Workshop Schedule

- Monday, June 30, Active learning and instructional goals
  - Morning
    - Welcome and Overview (1 hr)
    - Introductory Activity (1 hr). Determine background of participants
    - Active learning and instructional goals (1hr) in Discrete Math, Data Structures, Algorithms.
  - Afternoon
    - Group Work (1.5 hrs). Development of activities/goals from participant's classes.
    - Content lectures (Great Ideas in Computing): (1.5 hr) Problem mapping
- Tuesday, July 1, Discrete Mathematics
  - Morning
    - Discrete Mathematics Teaching (2 hrs)
    - Activities in Discrete Mathematics (1 hr)
  - Afternoon
    - Educational Technology Lecture (1.5 hrs)
    - Content Lecture: (1.5 hrs) Complexity Theory
- Wednesday, July 2, Data Structures
  - Morning
    - Data Structures Teaching (2hrs)
    - Data Structure Activities (1 hr)
  - Afternoon
    - Content Lecture: (1.5 hr) Average Case Analysis
    - Research Discussion
- Thursday, July 3, Algorithms
  - Morning
    - Algorithms Teaching (2 hrs)
    - Algorithms Activities (1 hr)
  - Afternoon
    - Content Lecture (1.5 hrs) Algorithm implementation
    - Theory discussion (optional) / Visit Mysore
- Friday, July 4, Topics
  - Morning
    - Lecture (1.5 hrs) Socially relevant computing
    - Faculty Presentations (1.5 hrs)
  - Afternoon
    - Follow up discussion with RJA (1.5 hrs)
    - Follow up discussion with Krishna

Thursday and Friday

- Final presentations
  - Short presentations by groups
  - How will you take ideas from this workshop and implement them in a class next term
  - Create a few power point slides
- Find time on Thursday to prepare talks
- Presentations after coffee break Friday morning

Highlights from Day 2

University of Washington Course

CSE 326 Data Structures (4)
Abstract data types and their implementations as data structures. Efficient of algorithms employing these data structures; asymptotic analyses. Dictionaries: balanced search trees, hashing. Priority queues: heaps. Disjoint sets with union, find. Graph algorithms: shortest path, minimum spanning tree, topological sort, search. Sorting. Prerequisite: CSE 321.

- Data Structures and Algorithm Analysis in Java 2nd Ed., Mark Allen Weiss
- Ten week term
  - 3 lectures per week (50 minutes)
  - 1 quiz section
  - Midterm, Final

Course overview

- Background (3)
- Heaps (4)
- Trees (5)
- Hashing (1)
- Union Find (2)
- Sorting (2)
- Graphs (3)
- Special Topics (4)

Analyzing the course and content

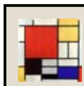
- What is the purpose of each unit?
  - Long term impact on students
- What are the learning goals of each unit?
  - How are they evaluated
- What strategies can be used to make material relevant and interesting?
- How does the context impact the content

Broader goals

- Analysis of course content
  - How does this apply to the courses that you teach?
- Reflect on challenges of your courses

Overall course context

- Discrete structures a pre-requisite
  - Students will have taken other majors classes
- Students interested in the implementations side of the course
  - Graduates remember the course positively
- Internal inconsistency in course offerings
  - Many different instructors teach the course
  - Some instructors take a different approach
- Concern that the material is out of date
- CS2 introduces some of the concepts covered in the course



Background

- Need to define the course
  - Asymptotic analysis – why constant factors don't matter
  - ADTs – this is an old program structuring concept
- Handling the interface with CS2 is tricky
  - Some variety in which course offering students had

Class Overview

- Introduction to many of the basic data structures used in computer software
  - Understand the data structures
  - Analyze the algorithms that use them
  - Know when to apply them
- Practice design and analysis of data structures.
- Practice using these data structures by writing programs.
- Make the transformation from programmer to computer scientist

Goals

- You will understand
  - what the tools are for storing and processing common data types
  - which tools are appropriate for which need
- So that you can
  - make good design choices as a developer, project manager, or system customer
- You will be able to
  - *Justify* your design decisions via formal reasoning
  - *Communicate* ideas about programs clearly and precisely

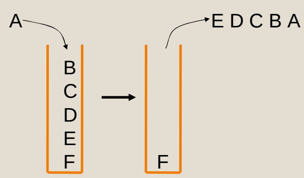
Concepts vs. Mechanisms

- Abstract
- Pseudocode
- Algorithm
  - A sequence of high-level, language independent operations, which may act upon an abstracted view of data.
- Abstract Data Type (ADT)
  - A mathematical description of an object and the set of operations on the object.

- Concrete
- Specific programming language
- Program
  - A sequence of operations in a specific programming language, which may act upon real data in the form of numbers, images, sound, etc.
- Data structure
  - A specific way in which a program's data is represented, which reflects the programmer's design choices/goals.

Second Example: Stack ADT

- LIFO: Last In First Out
- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty



Algorithm Analysis: Why?

- Correctness:
  - Does the algorithm do what is intended.
- Performance:
  - What is the running time of the algorithm.
  - How much storage does it consume.
- Different algorithms may be correct
  - Which should I use?

Asymptotic Analysis

- Eliminate low order terms
  - $4n + 5 \Rightarrow$
  - $0.5\ n \log n + 2n + 7 \Rightarrow$
  - $n^3 + 2^n + 3n \Rightarrow$
- Eliminate coefficients
  - $4n \Rightarrow$
  - $0.5\ n \log n \Rightarrow$
  - $n \log n^2$ =>

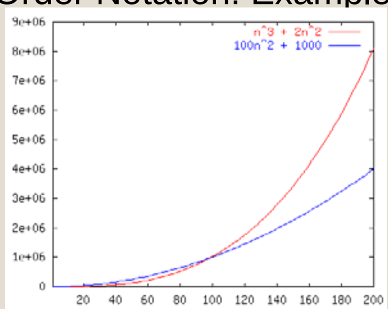
Definition of Order Notation

- Upper bound: $T(n) = O(f(n))$ Big-O
  Exist positive constants c and n' such that
  $T(n) \le c\, f(n)$ for all $n \ge n'$

- Lower bound: $T(n) = \Omega(g(n))$ Omega
  Exist positive constants c and n' such that
  $T(n) \ge c\, g(n)$ for all $n \ge n'$

- Tight bound: $T(n) = \theta(f(n))$ Theta
  When both hold:
  $T(n) = O(f(n))$
  $T(n) = \Omega(f(n))$

Order Notation: Example



$100n^2 + 1000 \le 5\ (n^3 + 2n^2)$ for all $n \ge 19$

So $f(n) \in O(\ g(n)\ )$

Types of Analysis

Two <u>orthogonal</u> axes:

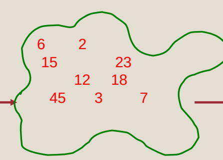
- Bound Flavor
  - Upper bound (O, o)
  - Lower bound (Ω, ω)
  - Asymptotically tight (θ)
- Analysis Case
  - Worst Case (Adversary)
  - Average Case
  - Best Case
  - Amortized

Heaps

- Multiple heaps are introduced
  - Priority Queue
  - Leftist Heaps
  - Skew Heaps
  - Binomial Queues
- Idea of Priority Queue is absolutely fundamental
- Other concepts introduced with other flavors of heaps
  - e.g., d-heaps allow higher branching factor and tradeoffs in operation costs
- Introducing other queues only makes sense if underlying concepts are emphasized

Queues that Allow Line Jumping

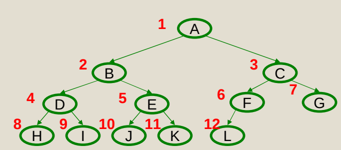
- Need a new ADT
- Operations: Insert an Item,
  Remove the "Best" Item



Priority Queue ADT

1. **PQueue <u>data</u>** : collection of data with priority
2. **PQueue <u>operations</u>**
   – insert
   – deleteMin
3. **PQueue <u>property</u>**: for two elements in the queue, *x* and *y*, if *x* has a **<u>lower</u>** priority value than *y*, *x* will be deleted before *y*

Representing Complete Binary Trees in an Array



From node **i**:

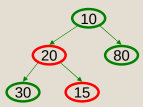
left child:
right child:
parent:

implicit (array) implementation:

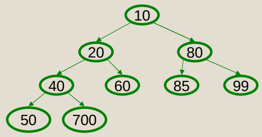
	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap **Order** Property

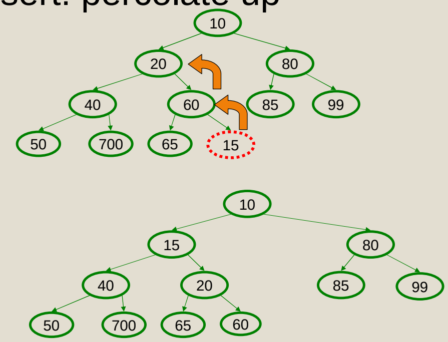
Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.



not a heap



Insert: percolate up



More Priority Queue Operations

- **decreaseKey**
  - given a pointer to an object in the queue, reduce its priority value

  Solution: change priority and ________________________

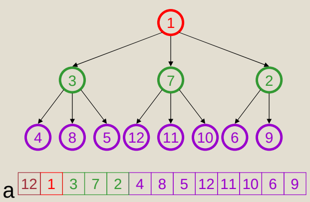
- **increaseKey**
  - given a pointer to an object in the queue, increase its priority value

  Why do we need a *pointer*? Why not simply data value?

  Solution: change priority and ________________________

A Solution: *d*-Heaps

- Each node has *d* children
- Still representable by array
- Good choices for *d*:
  - (choose a power of two for efficiency)
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block

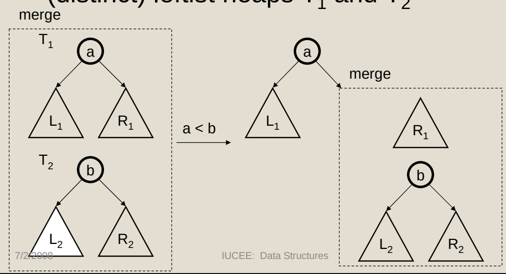


Leftist Heap Properties

- Heap-order property
  - parent's priority value is ≤ to childrens' priority values
  - <u>result</u>: minimum element is at the root
- Leftist property
  - For every node x, $npl(\text{left}(x)) \geq npl(\text{right}(x))$
  - <u>result</u>: tree is at least as "heavy" on the left as the right

Merging Two Leftist Heaps

- merge(T_1,T_2) returns one leftist heap containing all elements of the two (distinct) leftist heaps T_1 and T_2



Yet Another Data Structure: Binomial Queues

- Structural property
  - Forest of binomial trees with at most one tree of any height
- Order property
  - Each binomial tree has the heap-order property

What's a forest?

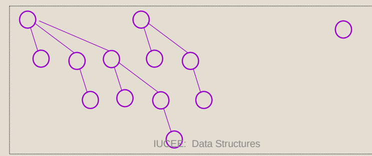
What's a binomial tree?

Binomial Queue with *n* elements

Binomial Q with *n* elements has a *unique* structural representation in terms of binomial trees!

Write *n* in binary: $n = 1101_{\text{(base 2)}} = 13_{\text{(base 10)}}$

1 B_3 1 B_2 No B_1 1 B_0



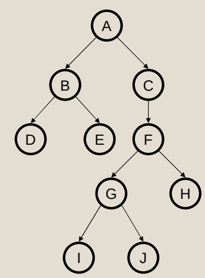
Trees

- Understanding binary trees and binary search trees is critical
- Material may have been covered in CS2
  - but I want students to really understand it
  - implementation assignment can really help
  - long term understanding of search and deletion
- Concept of balanced trees (e.g. AVL) important
  - Details less so

Binary Trees

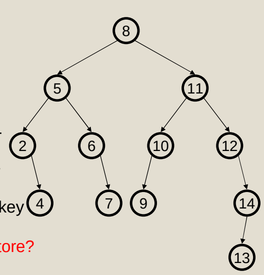
- Binary tree is
  - a root
  - left subtree *(maybe empty)*
  - right subtree *(maybe empty)*
- Representation:

Data	
left pointer	right pointer

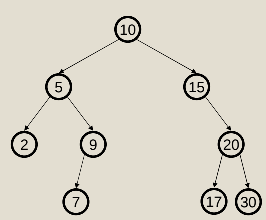


Binary Search Tree Data Structure

- Structural property
  - each node has ≤ 2 children
  - result:
    - storage is small
    - operations are simple
    - average depth is small
- Order property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result: easy to find any given key
- What must I know about what I store?



Find in BST, Recursive




```
Node Find(Object key,
            Node root) {
  if (root == NULL)
    return NULL;

  if (key < root.key)
    return Find(key,
                root.left);
  else if (key > root.key)
    return Find(key,
                root.right);
  else
    return root;
}
```

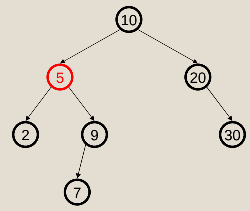
Runtime:

Non-lazy Deletion

- Removing an item disrupts the tree structure.
- Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.
- Three cases:
  - node has no children (leaf node)
  - node has one child
  - node has two children

Deletion – The Two Child Case

Delete(5)



What can we replace 5 with?

Balanced BST

<u>Observation</u>

- BST: the shallower the better!
- For a BST with n nodes
  - Average height is $O(\log n)$
  - Worst case height is $O(n)$
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

<u>Solution</u>: Require a **Balance Condition** that

1. ensures depth is $O(\log n)$ – strong enough!
2. is easy to maintain – not too strong!

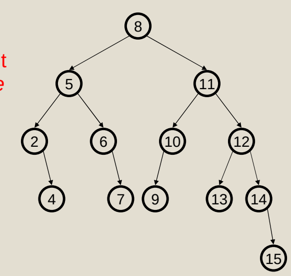
The AVL Tree Data Structure

<u>Structural properties</u>

1. Binary tree property (0,1, or 2 children)
2. Heights of left and right subtrees of *every node* **differ by at most 1**

Result:

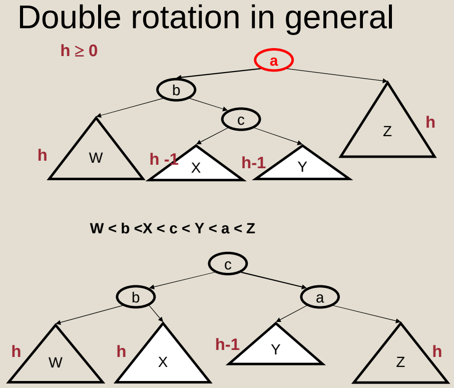
Worst case depth of any node is: $O(\log n)$



<u>Ordering property</u>

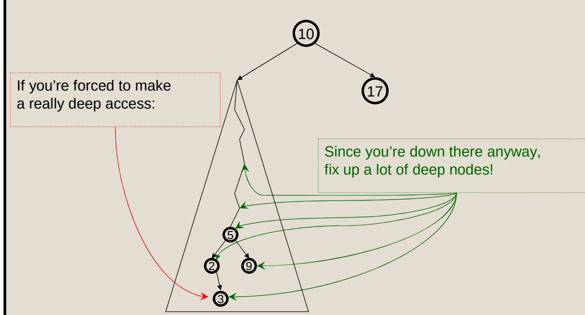
– Same as for BST

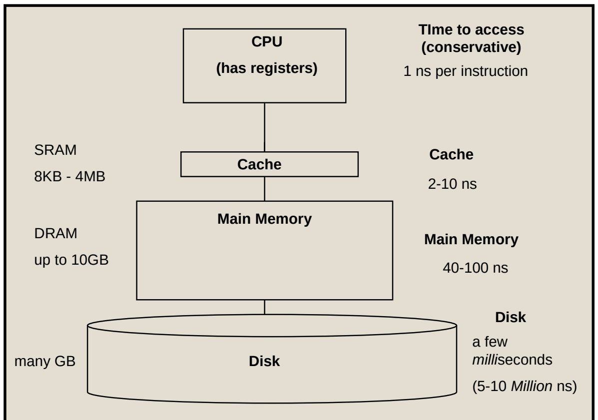
Double rotation in general



Height of tree before? Height of tree after? Effect on Ancestors?

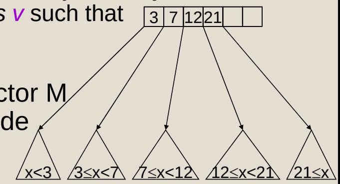
The Splay Tree Idea





Solution: B-Trees

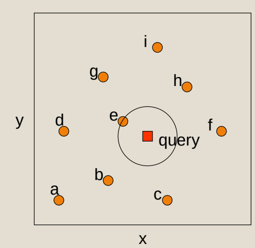
- specialized *M*-ary search trees
- Each **node** has (up to) M-1 keys:
  - subtree between two keys *x* and *y* contains leaves with *values v* such that $x \le v < y$
- Pick branching factor M such that each node takes one full {*page, block*} of memory



Range Queries

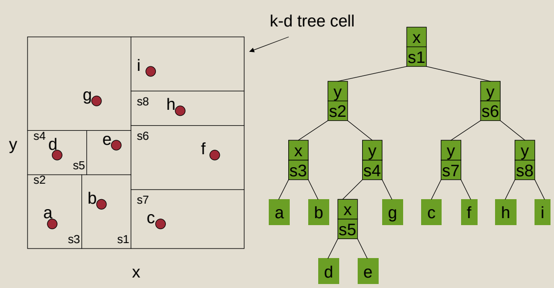
- Think of a range query.
  - "Give me all customers aged 45-55."
  - "Give me all accounts worth $5m to $15m"
- Can be done in time ________.
- What if we want both:
  - "Give me all customers aged 45-55 with accounts worth between $5m and $15m."

Nearest Neighbor Search



Nearest neighbor is e.

k-d Tree Construction

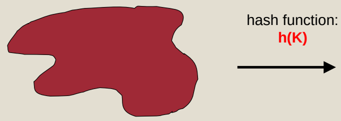


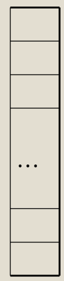
Hashing

- Great idea – but the idea can be conveyed quickly
- Implementation of hash tables less important than in the past
  - Programmers should use build in HashTable class

Hash Tables

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





Collision Resolution

Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

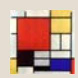
1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

Analysis of find

- Defn: The load factor, λ, of a hash table is the ratio: $\frac{N}{M}$ $\leftarrow$ no. of elements, $\leftarrow$ table size

For separate chaining, λ = average # of elements in a bucket

- Unsuccessful find:
- Successful find:



Union Find

- Classic data structure
- Some neat ideas
  - In-tree data structure
  - Path compression
  - Weighted union
- Touches on deep theoretical results
- Not that useful
  - Programmers rarely implement Union-Find

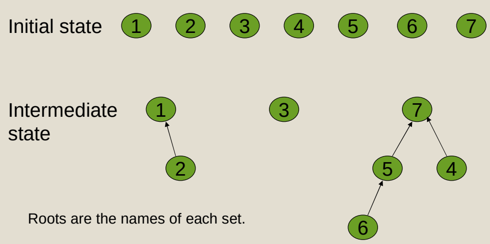
Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, one of its members
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Find(x) – return the name of the set containing x
- Union(x,y) – take the union of two sets named x and y

Find

- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5
  - Find(4) = 8

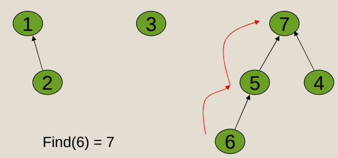
Up-Tree for DU/F



Roots are the names of each set.

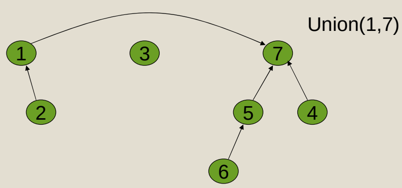
Find Operation

- Find(x) follow x to the root and return the root



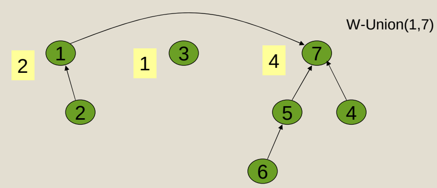
Union Operation

- Union(i,j) - assuming i and j roots, point i to j.



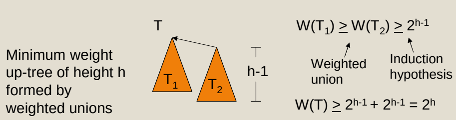
Weighted Union

- Weighted Union
  - Always point the smaller tree to the root of the larger tree



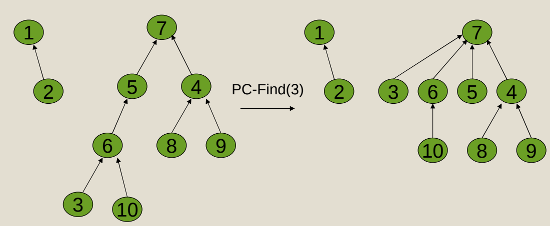
Analysis of Weighted Union

- With weighted union an up-tree of height h has weight at least 2^h.
- Proof by induction
  - Basis: h = 0. The up-tree has one node, $2^0 = 1$
  - Inductive step: Assume true for all h' < h.



Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



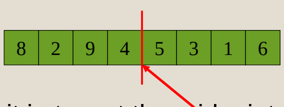
Disjoint Union / Find with Weighted Union and PC

- Worst case time complexity for a W-Union is O(1) and for a PC-Find is O(log n).
- Time complexity for m ≥ n operations on n elements is O(m log* n)
  - Log * n < 7 for all reasonable n. Essentially constant time per operation!
- Using “ranked union” gives an even better bound theoretically.

Sorting

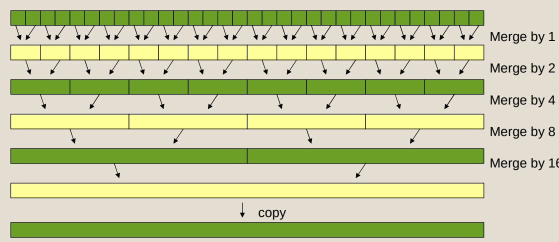
- Important – but programmers should not be writing sort routines
- The motivation for seeing lots of sort algorithms is to see the algorithmic ideas and issues
- Quicksort probably the most important

Mergesort



- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Iterative Mergesort



Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  - Partition array into left and right sub-arrays
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

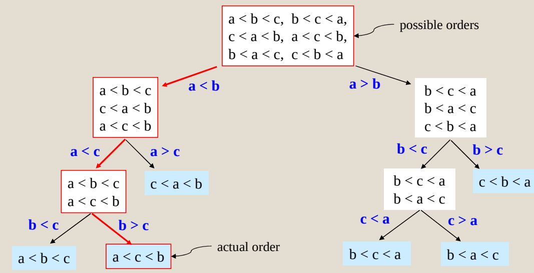
“Four easy steps”

- To sort an array **S**
  - If the number of elements in **S** is 0 or 1, then return. The array is sorted.
  - Pick an element *v* in **S**. This is the *pivot* value.
  - Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1$ = {all values $x \le v$}, and $\mathbf{S}_2$ = {all values $x \ge v$}.
  - Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

Features of Sorting Algorithms

- In-place
  - Sorted items occupy the same space as the original items. (No copying required, only O(1) extra space if any.)
- Stable
  - Items in input with the same value end up in the same order as when they began.

Decision Tree Example



Decision Trees and Sorting

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - ie, by making comparisons
  - Each decision reduces the possible solution space by one half
- Run time is ≥ maximum no. of comparisons
  - maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree

BucketSort (aka BinSort)

If all values to be sorted are *known* to be between 1 and *K*, create an array `count` of size *K*, **increment** counts while traversing the input, and finally output the result.

Example *K*=5. Input = (5,1,3,4,3,2,1,1,5,4,5)

`count` array	
1	
2	
3	
4	
5	

Running time to sort n items?

Radix Sort Example (2^{nd} pass)

After 1^{st} pass

721
3
123
537
67
478
38
9

Bucket sort by 10's digit

0	1	2	3	4	5	6	7	8	9
03 09		721 123	537 38			67	478		

After 2^{nd} pass

3
9
721
123
537
38
67
478

Summary of sorting

- Sorting choices:
  - *O(N²)* – Bubblesort, Insertion Sort
  - *O(N log N)* average case running time:
    - Heapsort: In-place, not stable.
    - Mergesort: *O(N)* extra space, stable.
    - Quicksort: claimed fastest in practice, but *O(N²)* worst case. Needs extra storage for recursion. Not stable.
  - *O(N)* – Radix Sort: fast and stable. Not comparison based. Not in-place.

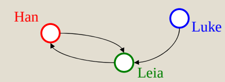
Graphs

- This shifts the course from data structures to algorithms
- Definitions and concepts of graphs from discrete mathematics, but algorithms should be new

Graphs

- A formalism for representing relationships between objects

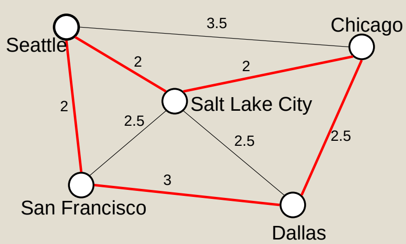
  Graph `G = (V,E)`
  - *Set of vertices*:
    $V = \{v_1, v_2, \ldots, v_n\}$
  - *Set of edges*:
    $E = \{e_1, e_2, \ldots, e_m\}$
    where each e_i connects two vertices (v_{i1}, v_{i2})




```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```


Path Length and Cost

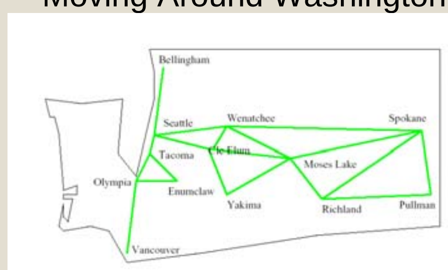
- *Path length*: the number of edges in the path
- *Path cost*: the sum of the costs of each edge



length(p) = 5

cost(p) = 11.5

Some Applications: Moving Around Washington

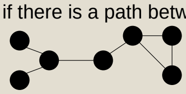


What's the *fastest way* to get from Seattle to Pullman?

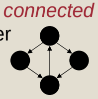
Edge labels:

Graph Connectivity

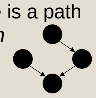
Undirected graphs are *connected* if there is a path between any two vertices



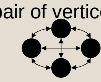
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



A *complete* graph has an edge between every pair of vertices



Depth-First Graph Search

Open – Stack

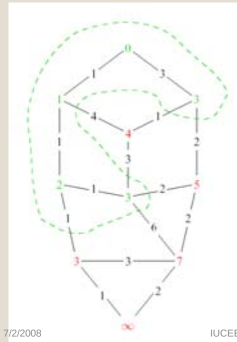
Criteria – Pop

```
DFS( Start, Goal_test)
  push(Start, Open);
  repeat
      if (empty(Open)) then return fail;
      Node := pop(Open);
      if (Goal_test(Node)) then return Node;
      for each Child of node do
          if (Child not already visited) then push(Child, Open);
      Mark Node as visited;
  end
```

Dijkstra's Algorithm for Single Source Shortest Path

- Similar to breadth-first search, but uses a heap instead of a queue:
  - Always select (expand) the vertex that has a lowest-cost path to the start vertex
- Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges

Dijkstra's Algorithm: Idea



At each step:

1) Pick closest unknown vertex
2) Add it to known vertices
3) Update distances

Dijkstra's Algorithm: Pseudocode

Initialize the cost of each node to ∞

Initialize the cost of the source to 0

While there are unknown nodes left in the graph
- Select an unknown node *b* with the lowest cost
- Mark *b* as known
- For each node *a* adjacent to *b*
  - *a*'s cost = min(*a*'s old cost, *b*'s cost + cost of (*b*, *a*))
  - *a*'s prev path node = *b*

Floyd-Warshall

```
for (int k = 1; k =< V; k++)
 for (int i = 1; i =< V; i++)
  for (int j = 1; j =< V; j++)
   if ( ( M[i][k]+ M[k][j] ) < M[i][j] )
     M[i][j] =  M[i][k]+ M[k][j]
```

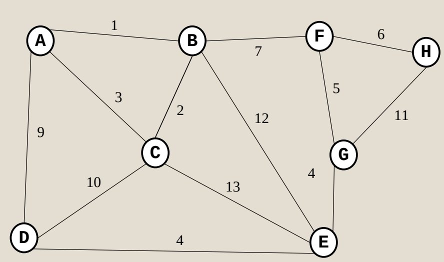
Invariant: After the kth iteration, the matrix includes the shortest paths for all pairs of vertices (i,j) containing only vertices 1..k as intermediate vertices

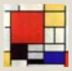
Minimum Spanning Tree Problem

- Input: Undirected Graph G = (V,E) and a cost function C from E to the reals. C(e) is the cost of edge e.
- Output: A spanning tree T with minimum total cost. That is: T that minimizes

$$C(T) = \sum_{e \in T} C(e)$$

Find the MST





Special Topics

- Although these topics are interesting, it is not clear what there purpose is

Problem: Large Graphs

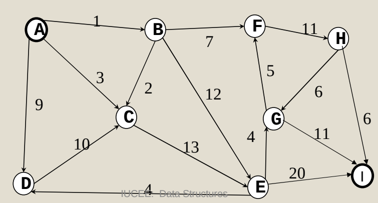
- It is expensive to find optimal paths in large graphs, using BFS or Dijkstra’s algorithm (for weighted graphs)
- How can we search large graphs efficiently by using “commonsense” about which direction looks most promising?
- Best-first search
- A*: Exactly like Best-first search, but using a different criteria for the priority queue:
  - minimize (distance from start) + (estimated distance to goal)

Speech Recognition as Shortest Path

- Convert to a shortest-path problem:
  - Utterance is a “layered” DAG
  - Begins with a special dummy “start” node
  - Next: A layer of nodes for each word position, one node for each word choice
  - Edges between every node in layer i to every node in layer i+1
    - Cost of an edge is smaller if the pair of words frequently occur together in real speech
      - Technically: - log probability of co-occurrence
  - Finally: a dummy “end” node
  - Find shortest path from start to end node

Network Flows

- Given a weighted, directed graph G=(V,E)
- Treat the edge weights as *capacities*
- How much can we flow through the graph?



Dictionary Coding

- Does not use statistical knowledge of data.
- Encoder: As the input is processed develop a dictionary and transmit the index of strings found in the dictionary.
- Decoder: As the code is processed reconstruct the dictionary to invert the process of encoding.
- Examples: LZW, LZ77, Sequitur,
- Applications: Unix Compress, gzip, GIF