Data Structures in Java

Session 25

Instructor: Bert Huang

http://www.cs.columbia.edu/~bert/courses/3134

Announcements

- Homework 6 due Monday Dec. 14th.
    - Nikhil's office hours moved from 5-7. Drop off theory homework at office hours.
- Final exam Thursday, Dec. 17th, 4-7 PM, Hamilton 602 (this room)
    - same format as midterm (open book/notes)

Review

- A couple topics on data structures in Artificial Intelligence:
    - Game trees
    - Graphical Models
- Final Review (part 1)

Course Topics

- Lists, Stacks, Queues
- General Trees
- Binary Search Trees
  - AVL Trees
  - Splay Trees
- Tries
- Priority Queues (heaps)
- Hash Tables
- Graphs
  - Topological Sort, Shortest Paths, Spanning Tree
- Disjoint Sets
- Sorting Algorithms
- Complexity Classes
- kd-Trees

Hash Table ADT

- Insert or delete objects by **key**
- Search for objects by **key**
- **No** order information whatsoever

- Ideally O(1) per operation

Implementation

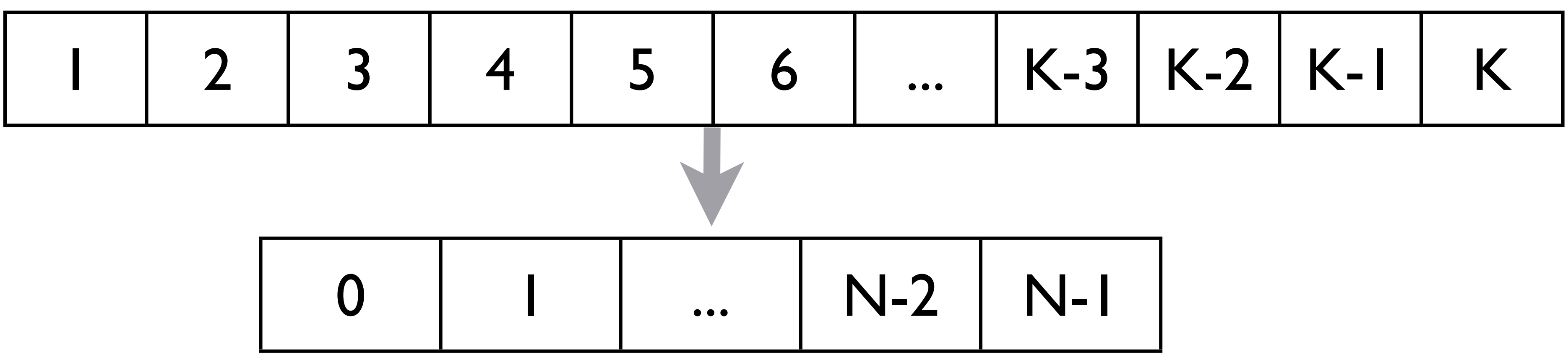
- Suppose we have keys between 1 and K
- Create an array with K entries
- Insert, delete, search are just array operations

1	2	3	4	5	6	...	K-3	K-2	K-1	K

- Obviously too expensive

Hash Functions

- A **hash function** maps any key to a valid array position
  - Array positions range from 0 to N-1
  - Key range possibly unlimited



Hash Functions

- For integer keys, (key mod N) is the simplest hash function
- In general, **any** function that maps from the space of keys to the space of array indices is valid
- but a good hash function spreads the data out evenly in the array
- A good hash function avoids **collisions**

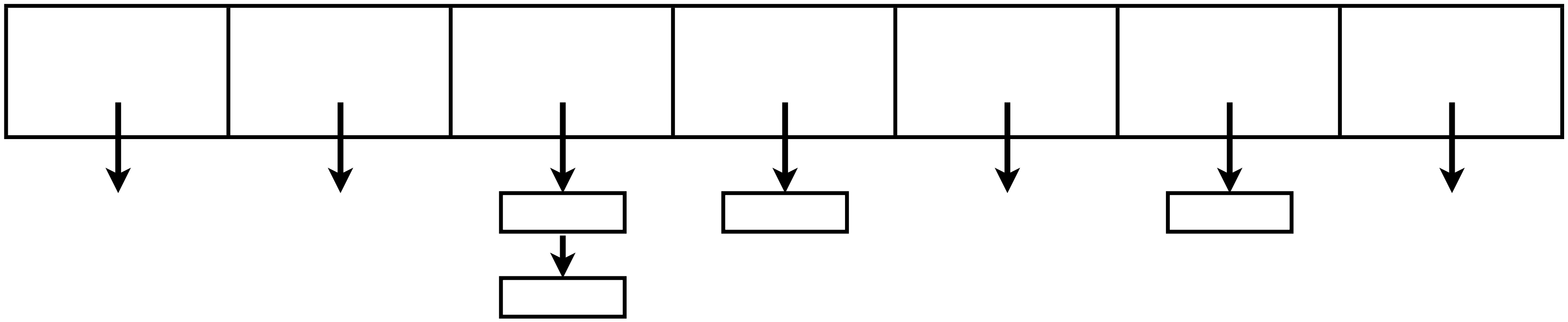
Collisions

- A **collision** is when two distinct keys map to the same array index
  - e.g., $h(x) = x \bmod 5$
    $h(7) = 2$, $h(12) = 2$
- Choose h(x) to minimize collisions, but collisions are inevitable
- To implement a hash table, we must decide on collision resolution policy

Collision Resolution

- Two basic strategies
  - Strategy 1: Separate Chaining
  - Strategy 2: Probing; lots of variants

Strategy 1: Separate Chaining



- Keep a list at each array entry
  - Insert(x): find h(x), add to list at h(x)
  - Delete(x): find h(x), search list at h(x) for x, delete
  - Search(x): find h(x), search list at h(x)

Separate Chaining Average Case

- **Load Factor** λ = # objects / TableSize
- Average list length is λ
- Time to insert = constant, or constant + λ
- Time to search = constant + λ or constant + $\lambda/2$

Strategy 2: Probing

- If h(x) is occupied, try **h(x)+f(i) mod N** for i = 1 until an empty slot is found
- Many ways to choose a good f(i)
- Simplest method: Linear Probing
  - f(i) = i

Primary Clustering

	x		x	x	x						x	

- If there are many collisions, blocks of occupied cells form: **primary clustering**
- Any hash value inside the cluster adds to the end of that cluster
- (a) it becomes more likely that the next hash value will collide with the cluster, and (b) collisions in the cluster get more expensive

Quadratic Probing

- f(i) = i^2
- Avoids primary clustering
- Sometimes will never find an empty slot even if table isn't full!
- Luckily, if load factor $\lambda \leq \frac{1}{2}$,

  guaranteed to find empty slot

Double Hashing

- If $h_1(x)$ is occupied, probe according to

$$f(i) = i \times h_2(x)$$

- 2nd hash function must never map to 0

- Increments differently depending on the key

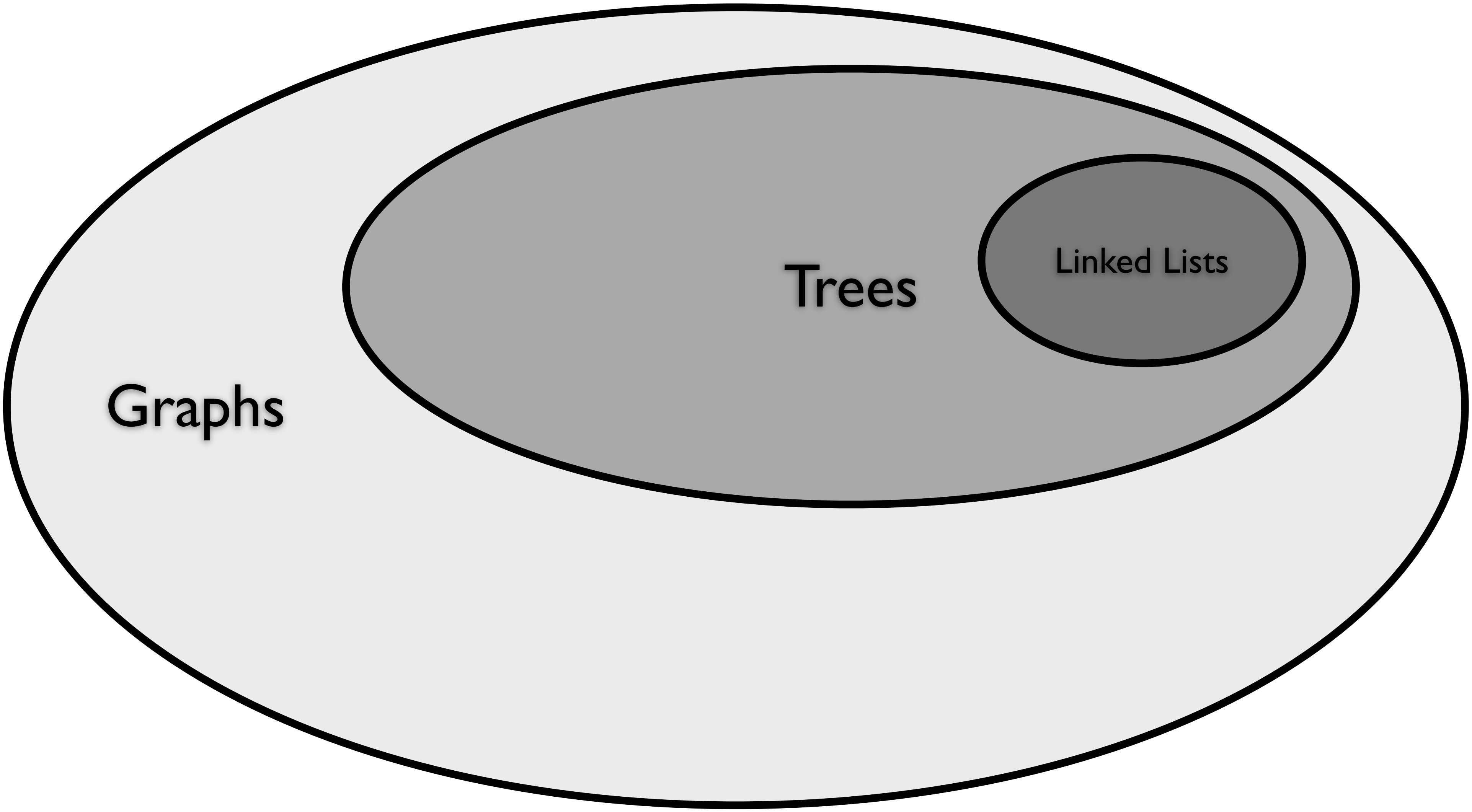
Hashing

- Indexing by the key needs too much memory
- Index into smaller size array, pray you don't get collisions
- If collisions occur,
  - separate chaining, lists in array
  - probing, try different array locations

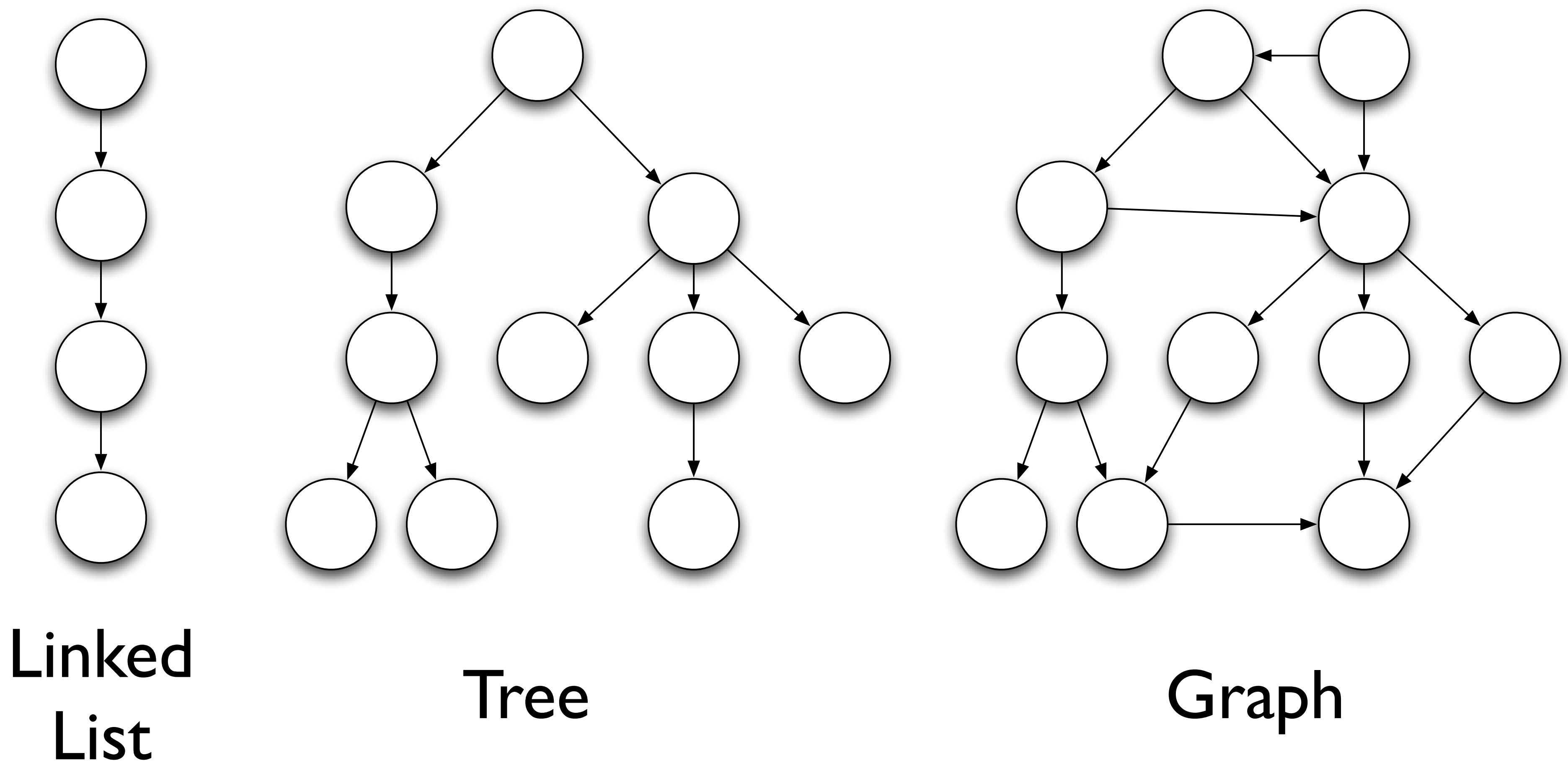
Rehashing

- Like ArrayLists, we have to guess the number of elements we need to insert into a hash table
- Whatever our collision policy is, the hash table becomes inefficient when load factor is too high.
- To alleviate load, **rehash**:
  - create larger table, scan current table, insert items into new table using new hash function

Graphs



Graphs



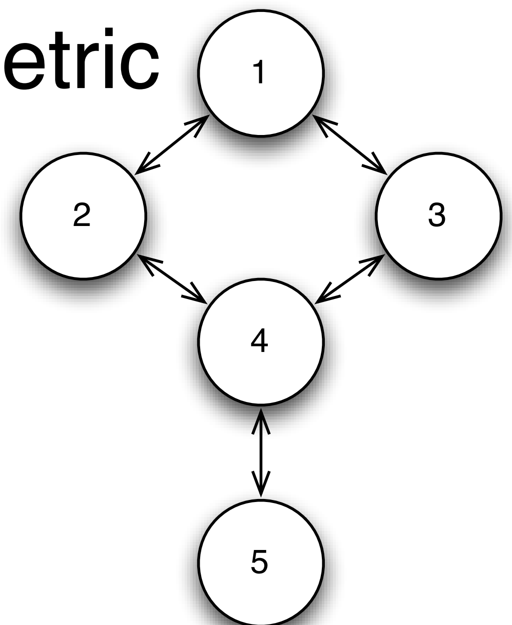
Implementation

- Option 1:
  - Store all nodes in an indexed list
  - Represent edges with **adjacency matrix**
- Option 2:
  - Explicitly store **adjacency lists**

Adjacency Matrices

- 2d-array **A** of boolean variables
- A[i][j] is true when node **i** is adjacent to node **j**
  - If graph is undirected, A is symmetric

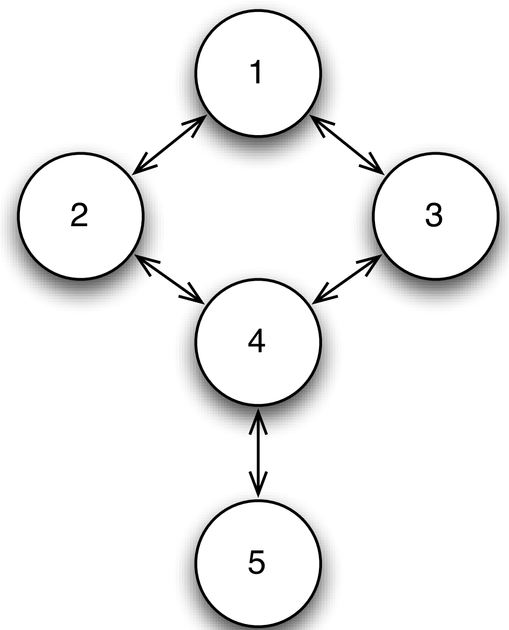
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	0
4	0	1	1	0	1
5	0	0	0	1	0



Adjacency Lists

- Each node stores references to its neighbors

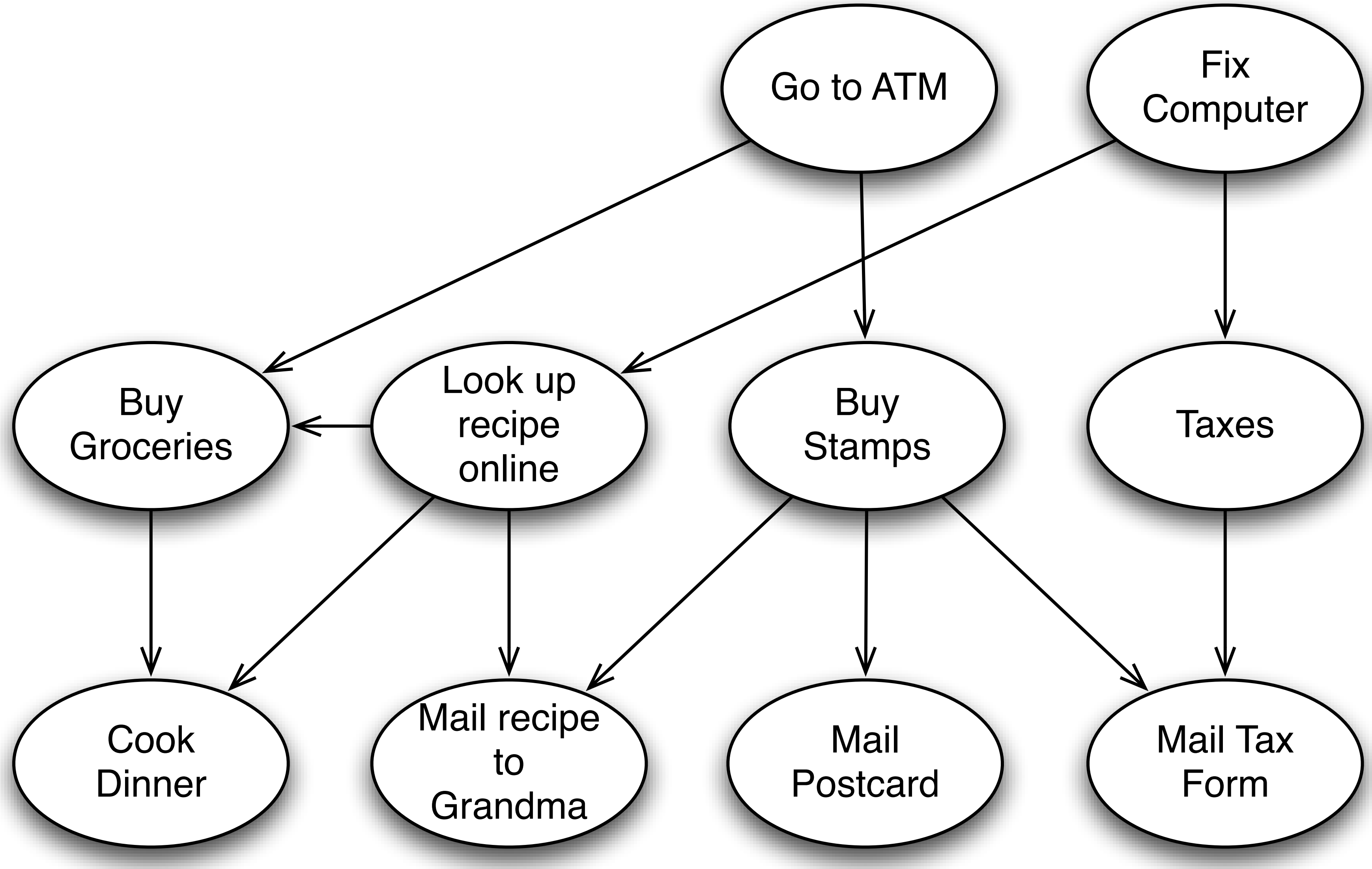
1	2	3	
2	1	4	
3	1	4	
4	2	3	5
5	4		



Topological Sort

- Problem definition:
  - Given a directed acyclic graph G, order the nodes such that for each edge $(v_i, v_j) \in E$, v_i is before v_j in the ordering.
- e.g., scheduling errands when some tasks depend on other tasks being completed.

Topological Sort Ex.



Topological Sort
Naïve Algorithm

- **Degree** means # of edges,
  indegree means # of incoming edges
- 1. Compute the **indegree** of all nodes
- 2. Print any node with indegree 0
- 3. Remove the node we just printed. Go to 1.
- Which nodes' indegrees change?

Topological Sort
Better Algorithm

- 1. Compute all indegrees
- 2. Put all indegree 0 nodes into a Collection
- 3. Print and remove a node from Collection
- 4. Decrement indegrees of the node’s neighbors.
- 5. If any neighbor has indegree 0, place in Collection. Go to 3.

Topological Sort Running time

- Initial indegree computation: O(|E|)
  - Unless we update indegree as we build graph
- |V| nodes must be enqueued/dequeued
- Dequeue requires operation for outgoing edges
- Each edge is used, but never repeated
- Total running time O(|V| + |E|)

Shortest Path

- Given **G = (V,E)**, and a node **s** ∈ **V**, find the shortest (weighted) path from **s** to every other vertex in **G**.
- Motivating example: subway travel
  - Nodes are junctions, transfer locations
  - Edge weights are estimated time of travel

Breadth First Search

- Like a level-order traversal
- Find all adjacent nodes (level 1)
- Find *new* nodes adjacent to level 1 nodes (level 2)
- ... and so on
- We can implement this with a queue

Unweighted Shortest Path Algorithm

- Set node s’ distance to 0 and enqueue s.
- Then repeat the following:
  - Dequeue node **v**. For unset neighbor **u**:
    - set neighbor **u**’s distance to **v**’s distance +1
    - mark that we reached **v** from **u**
    - enqueue **u**

Weighted Shortest Path

- The problem becomes more difficult when edges have different weights
- Weights represent different costs on using that edge
- Standard algorithm is **Dijkstra's Algorithm**

Dijkstra’s Algorithm

- Keep distance overestimates **D(v)** for each node **v** (all non-source nodes are initially infinite)
- 1. Choose node **v** with smallest *unknown* distance
- 2. Declare that **v**’s shortest distance is *known*
- 3. Update distance estimates for neighbors

Updating Distances

- For each of **v**'s neighbors, **w**,
- if min(**D(v)+ weight(v,w)**, **D(w)**)
    - i.e., update **D(w)** if the path going through **v** is cheaper than the best path so far to **w**

Computational Cost

- If the graph is dense, we scan the vertices to find the minimum edge **O(V)**
- This happens **|V|** times
- We also update the distances once per edge, **O(|E|)**
- Thus, total running time is $O(|E| + |V|^2)$

Computational Cost (sparse)

- Keep a priority queue of all unknown nodes
- Each stage requires a **deleteMin**, and then some **decreaseKey**s (the # of neighbors of node)
- We call **decreaseKey** once per edge, we call **deleteMin** once per vertex
- Both operations are O(log |V|)
- Total cost: O(|E| log |V| + |V| log |V|) = O(|E| log |V|)

All Pairs Shortest Path

- Dijkstra's Algorithm finds shortest paths from one node to all other nodes
- What about computing shortest paths for all pairs of nodes?
- We can run Dijkstra's |V| times. Total cost: $O(|V|^3)$
- Floyd-Warshall algorithm is often faster in practice (though same asymptotic time)

Recursive Motivation

- Consider the set of numbered nodes **1** through **k**
- The shortest path between any node **i** and **j** using only nodes in the set {**1**, ..., **k**} is the minimum of
  - shortest path from **i** to **j** using nodes {**1**, ..., **k-1**}
  - shortest path from **i** to **j** using node **k**
- dist(i,j,k) = min(dist(i,j,k-1),
  dist(i,k,k-1)+dist(k,j,k-1))

Dynamic Programming

- Instead of repeatedly computing recursive calls, store lookup table
- To compute dist(i,j,k) for any i,j, we only need to look up dist(-,-, k-1)
  - but never k-2, k-3, etc.
- We can incrementally compute the path matrix for k=0, then use it to compute for k=1, then k=2...

Floyd-Warshall Code

- Initialize `d = ` weight matrix
-
```
for (k=0; k<N; k++)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      if (d[i][j] > d[i][k]+d[k][j])
        d[i][j] = d[i][k] + d[k][j];
```
- Additionally, we can store the actual path by keeping a “midpoint” matrix

Midpoint Matrix

- We can store the N^2 paths efficiently with a midpoint matrix:

  path(i,j) = path(i, midpoint[i][j]) +
                  path(midpoint[i][j], j)

- We only need a NxN matrix to store all the paths

Transitive Closure

- For any nodes i, j, is there a path from i to j?
- Instead of computing shortest paths, just compute Boolean if a path exists
- path(i,j,k) = path(i,j,k-1) OR
  path(i,k,k-1) AND path(k,j,k-1)
- Transitive closure can tell you whether a graph is **connected**

Minimum Spanning Tree Problem Definition

- Given connected graph **G**, find the connected, acyclic subgraph **T** with minimum edge weight
  - A tree that includes every node is called a **spanning tree**
- The method to find the MST is another example of a greedy algorithm

Prim's Algorithm

- Grow the tree like Dijkstra's Algorithm
- Dijkstra's: grow the set of vertices to which we know the shortest path
- Prim's: grow the set of vertices we have added to the minimum tree
- Store shortest edge **D**[ ] from each node to tree

Prim’s Algorithm

- Start with a single node tree, set distance of adjacent nodes to edge weights, infinite elsewhere
- Repeat until all nodes are in tree:
  - Add the node **v** with shortest known distance
  - Update distances of adjacent nodes **w**: **D**[w] = min(**D**[w], weight(**v**,**w**))

Implementation Details

- Store “previous node” like Dijkstra’s Algorithm; backtrack to construct tree after completion
- Of course, use a priority queue to keep track of edge weights. Either
  - keep track of nodes inside heap & decreaseKey
  - or just add a new copy of the node when key decreases, and call deleteMin until you see a node not in the tree

Prim's Running Time

- Each stage requires one deleteMin $O(\log |V|)$, and there are exactly $|V|$ stages
- We update keys for each edge, updating the key costs $O(\log |V|)$ (either an insert or a decreaseKey)
- Total time:
  $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$

Kruskal's Algorithm

- Somewhat simpler conceptually, but more challenging to implement
- Algorithm: repeatedly add the shortest edge that does not cause a cycle until no such edges exist
- Each added edge performs a union on two trees; perform unions until there is only one tree
- Need special ADT for unions (Disjoint Set)

Kruskal's Running Time

- First, buildHeap costs O(lEl)
- In the worst case, we have to call lEl deleteMins $|E| \leq |V|^2$
- Total running time O(lEl log lEl); but

$$O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|)$$

Motivating Example

- One interpretation of Kruskal's Algorithm:
  - Think of trees as sets of connected nodes
  - Merge sets by connecting nodes
  - Never merge nodes that are in the same set
- Simple idea, but how can we implement it?

Equivalence Classes

- Equivalence class: the set of elements that are all related to each other via an equivalence relation
- Due to transitivity, each member can only be a member of one equivalence class
- Thus, equivalence classes are **disjoint sets**
  - Choose any distinct sets S and T, $S \cap T = \emptyset$

Disjoint Set ADT

- Collection of objects, each in an equivalence class
- **find**(x) returns the class of the object
- **union**(x,y) puts x and y in the same class
  - as well as every other relative of x and y
- Even less information than hash; no keys, no ordering

Data Structure

- Store elements in equivalence (general) trees
- Use the tree's root as equivalence class label
- **find** returns root of containing tree
- **union** merges tree
- Since all operations only search up the tree, we can store in an array

Implementation

- Index all objects from 0 to N-1
- Store a parent array such that **s[i]** is the index of **i**'s parent
- If **i** is a root, store the negative size of its tree*
- **find** follows **s[i]** until negative, returns index
- **union**(x,y) points the root of x's tree to the root of y's tree

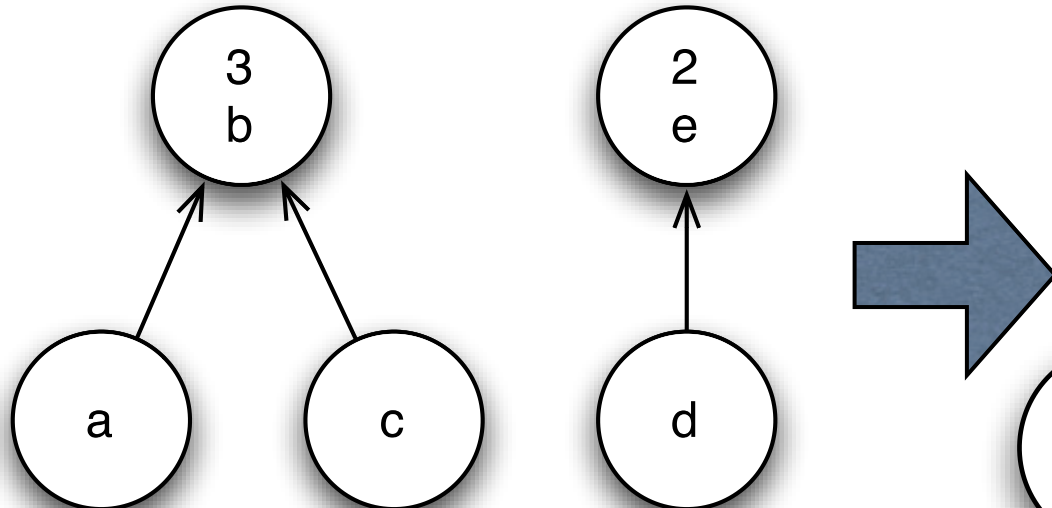
Analysis

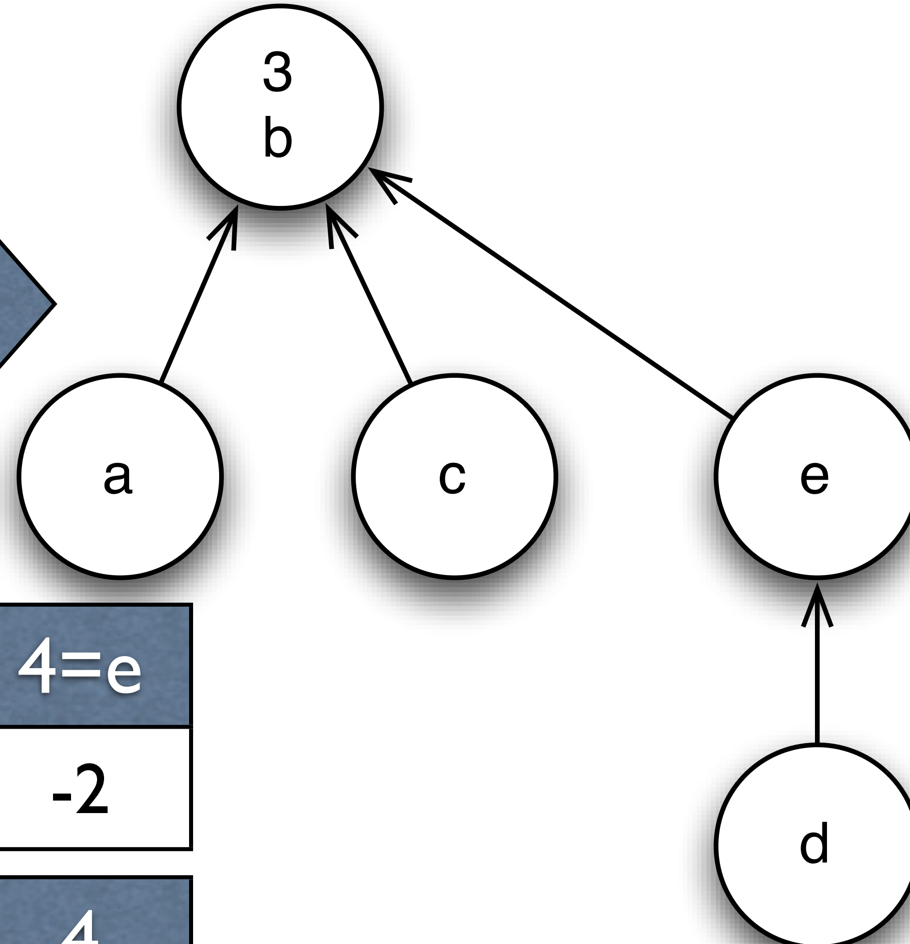
- **find** costs the depth of the node
- **union** costs O(1) after **find**ing the roots
- Both operations depend on the height of the tree
- Since these are general trees, the trees can be arbitrarily shallow

Union by Size

- Claim: if we union by pointing the smaller tree to the larger tree's root, the height is at most log N
- Each union increases the depths of nodes in the smaller trees
- Also puts nodes from the smaller tree into a tree at least twice the size
  - We can only double the size log N times

Union by Size Figure





0=a	1=b	2=c	3=d	4=e
1	-3	1	4	-2

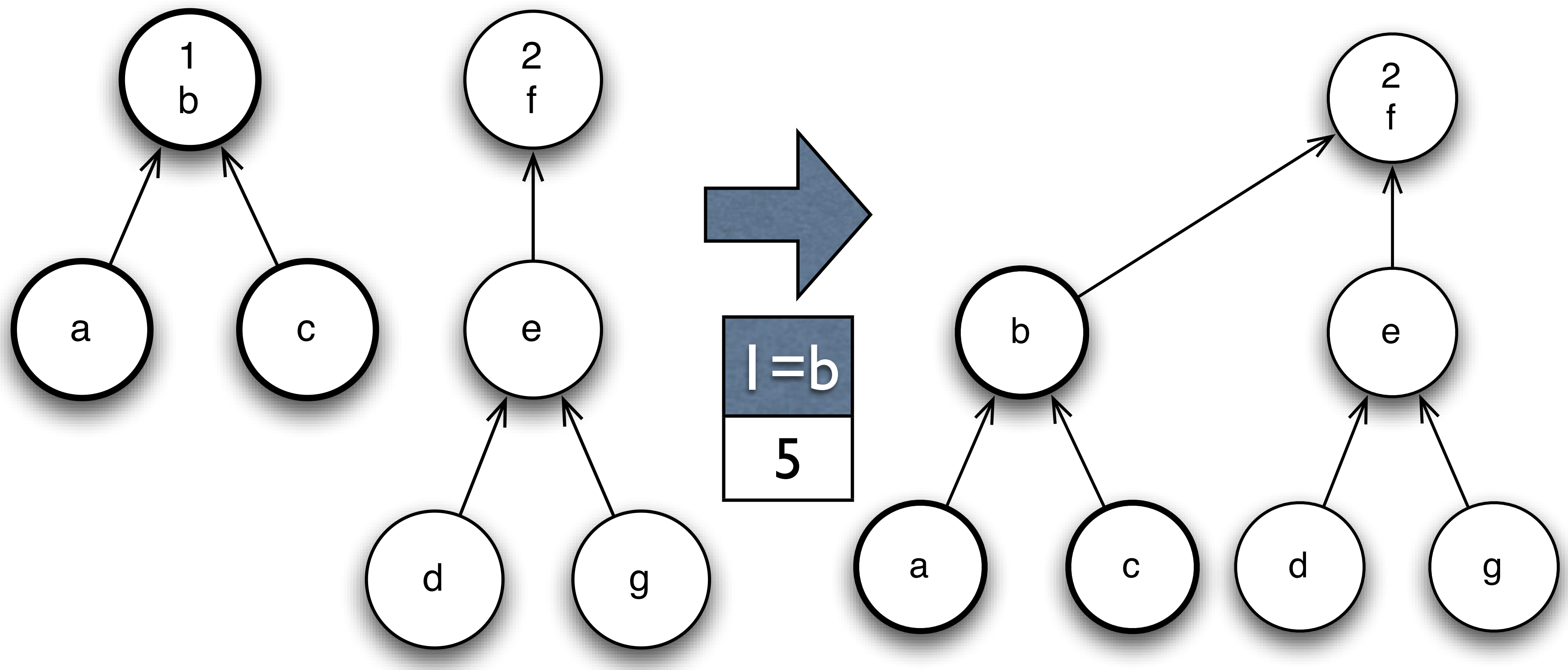
0	1	2	3	4
1	-5	1	4	1

Union by Height

- Similar method, attach the tree with less height to the taller tree
- overall height only increases if trees are equal height

Union by Height Figure

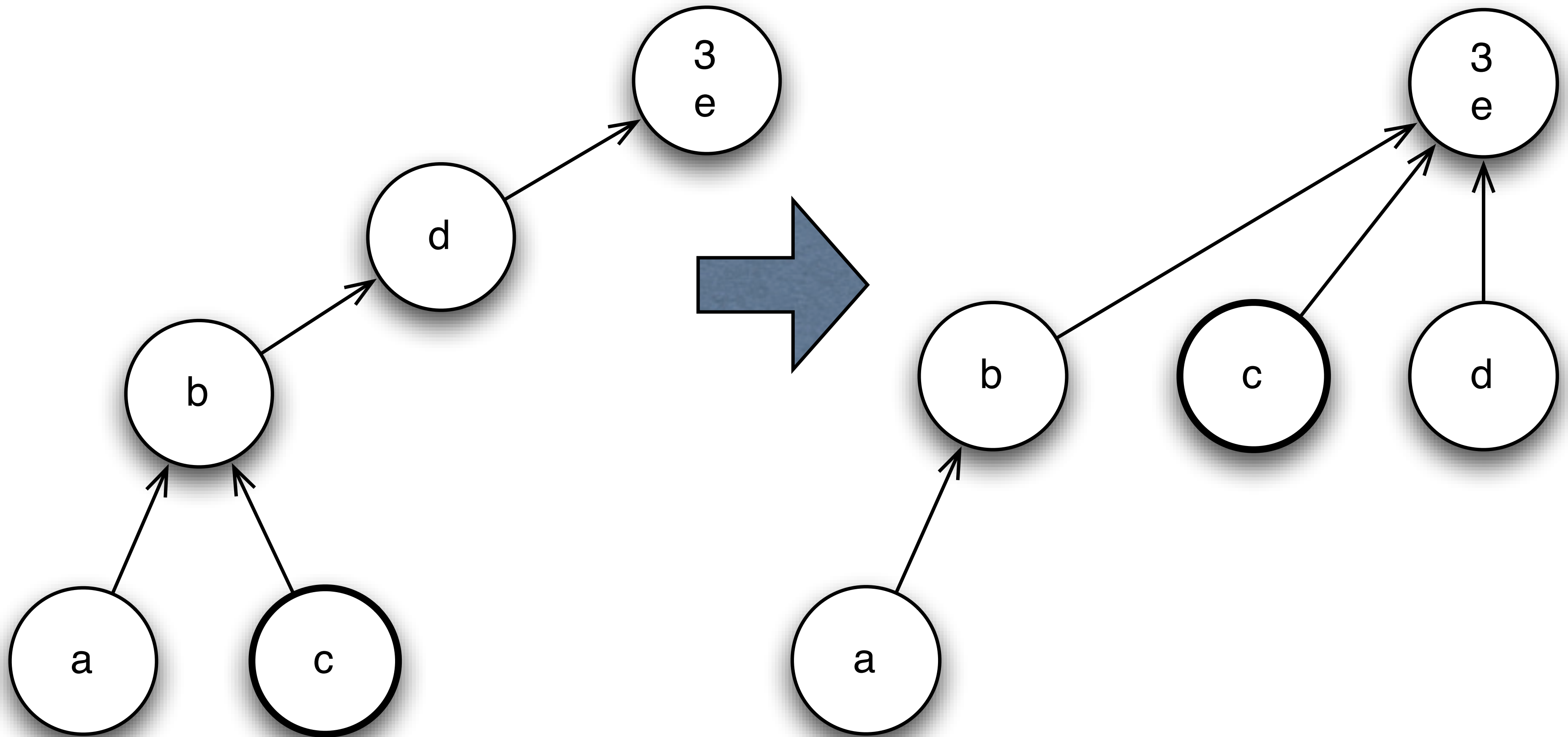
0=a	1=b	2=c	3=d	4=e	5=f	6=g
1	-1	1	4	5	-2	4



Path Compression

- Even if we have log N tall trees, we can keep calling **find** on the deepest node repeatedly, costing O(M log N) for M operations
- Additionally, we will perform **path compression** during each **find** call
  - Point every node along the find path to root

Path Compression Figure



0=a	1=b	2=c	3=d	4=e
1	3	1	4	-3

0=a	1=b	2=c	3=d	4=e
1	4	4	4	-3

Union by Rank

- Path compression messes up union-by-height because we reduce the height when we compress
- We could fix the height, but this turns out to gain little, and costs **find** operations more
- Instead, rename to **union by rank**, where **rank** is just an overestimate of height
- Since heights change less often than sizes, rank/height is usually the cheaper choice

Worst Case Bound

- Any sequence of $M = \Omega(N)$ operations will cost **O(M log* N)** running time
- log* N is the number of times the logarithm needs to be applied to N until the result is ≤ 1
- So for all realistic intents, each operation is amortized constant time

Note about Kruskal's

- With this bound, Kruskal's algorithm needs N-1 unions, so it should cost almost linear time to perform unions
- Unfortunately the algorithm is still dominated by heap deleteMin calls, so asymptotic running time is still O(E log V)

Sorting

- Given array A of size N, reorder A so its elements are in order.
  - "In order" with respect to a consistent comparison function

Radix Sort

- Radix Sort sorts by looking at one digit at a time
- We can start with the least significant digit or the most significant digit
  - least significant digit first provides a **stable** sort
  - tries use most significant, so let's look at least...

Radix Sort with Least Significant Digit

- BucketSort according to the least significant digit
- Repeat: BucketSort contents of each multi-item bucket according to the next least significant digit
- Running time: **O(Nk)** for maximum of **k** digits
- Space: **O(Nk)**

Radix Sort with Least Significant Digit

- CountingSort according to the least significant digit
- Repeat: CountingSort according to the next least significant digit
- Each step must be **stable**
- Running time: **O(Nk)** for maximum of **k** digits
- Space: **O(N+b)** for base-**b** number system*

Comparison Sorts

- Of course, Radix Sort only works well for sorting keys representable as digital numbers
- In general, we must often use comparison sorts
- We have proven a $\Omega(N \log N)$ lower bound for running time
- But algorithms also have other desirable characteristics

Sorting Algorithm Characteristics

- Worst case running time
- Worst case space usage (can it run in place?)
- Stability
- Average running time/space
- (simplicity)
- (Best case running time/space usage)

Preview

	Worst Case Time	Average Time	Space	Stable?
Selection	$O(N^2)$	$O(N^2)$	$O(1)$	No
Insertion	$O(N^2)$	$O(N^2)$	$O(1)$	Yes
Shell	$O(N^{3/2})$	?	$O(1)$	No
Heap	$O(N \log N)$	$O(N \log N)$	$O(1)$	No
Merge	$O(N \log N)$	$O(N \log N)$	$O(N)/O(1)$	Yes/No
Quick	$O(N^2)$	$O(N \log N)$	$O(\log N)$	No

Selection Sort

- Swap least unsorted element with first unsorted element
- Unstable if in place
- Running time $O(N^2)$
- In place O(1) space

Selection Sort

3	7	5	2	6	1	0	4
0	7	5	2	6	1	3	4
0	1	5	2	6	7	3	4
0	1	2	5	6	7	3	4
0	1	2	3	6	7	5	4
0	1	2	3	4	7	5	6
0	1	2	3	4	5	7	6
0	1	2	3	4	5	6	7

Insertion Sort

- Assume first **p** elements are sorted. Insert **(p+1)**'th element into appropriate location.
  - Save **A[p+1]** in temporary variable **t**, shift sorted elements greater than **t**, and insert **t**
- Stable
- Running time $O(N^2)$
- In place **O(1)** space

Insertion Sort

3	7	5	2	6	1	0	4
3	7	5	2	6	1	0	4
3	5	7	2	6	1	0	4
2	3	5	7	6	1	0	4
2	3	5	6	7	1	0	4
1	2	3	5	6	7	0	4
0	1	2	3	5	6	7	4
0	1	2	3	4	5	6	7

Insertion Sort Analysis

- When the sorted segment is **i** elements, we may need up to **i** shifts to insert the next element

$$\sum_{i=2}^{N} i = N(N-1)/2 - 1 = O(N^2)$$

- Stable because elements are visited in order and equal elements are inserted after its equals

Shellsort

- Essentially splits the array into subarrays and runs Insertion Sort on the subarrays
- Uses an increasing sequence, $h_1, \ldots, h_t$, such that $h_1 = 1$.
- At phase **k**, all elements h_k apart are sorted; the array is called h_k-sorted
- for every **i**, $A[i] \leq A[i + h_k]$

Shell Sort Correctness

- Efficiency of algorithm depends on that elements sorted at earlier stages remain sorted in later stages

- Unstable. Example: 2-sort the following: [5 5 1]

Increment Sequences

- Shell suggested the sequence $h_t = \lfloor N/2 \rfloor$ and $h_k = \lfloor h_{k+1}/2 \rfloor$, which was suboptimal
- A better sequence is $h_k = 2^k - 1$
- Using better sequence sorts in $\Theta(N^{3/2})$
- Often used for its simplicity and sub-quadratic time, even though **O(N log N)** algorithms exist

Shell Sort I

3	7	5	2	6	1	0	4
3	7	5	2	6	1	0	4
2	7	5	3	6	1	0	4
0	7	5	2	6	1	3	4
0	6	5	2	7	1	3	4
0	4	5	2	6	1	3	7
0	4	1	2	6	5	3	7

Shell Sort II

0	4	1	2	6	5	3	7
0	1	4	2	6	5	3	7
0	1	2	4	6	5	3	7
0	1	2	4	5	6	3	7
0	1	2	3	4	5	6	7

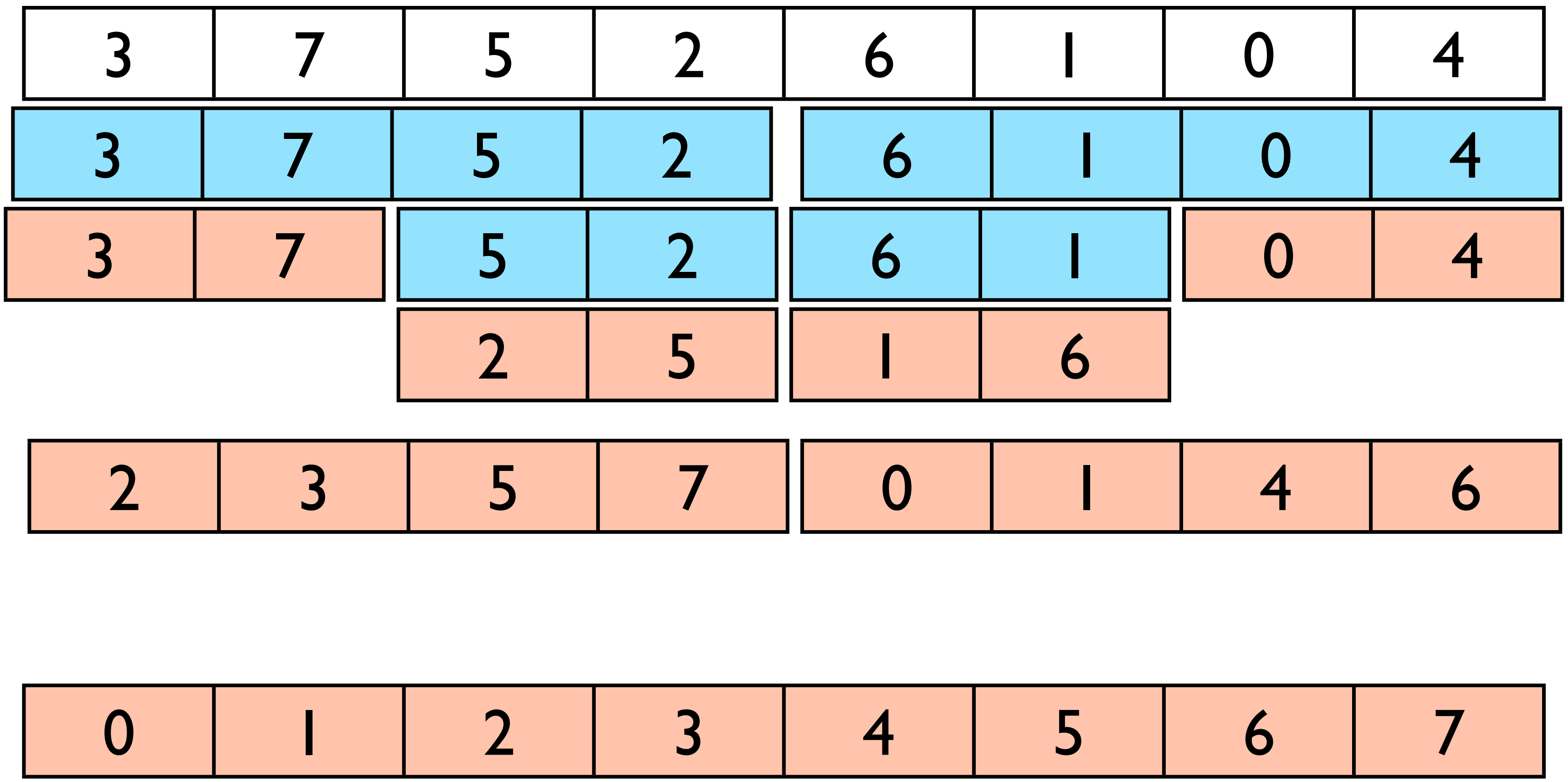
Heapsort

- Build a **max** heap from the array: **O(N)**
- call deleteMax **N** times: **O(N log N)**
- **O(1)** space
- Simple if we abstract heaps
- Unstable

Mergesort

- Quintessential divide-and-conquer example
- Mergesort each half of the array, merge the results
- Merge by iterating through both halves, compare the current elements, copy lesser of the two into output array

Merge Sort



Mergesort Recurrence

- Merge operation is costs **O(N)**
- **T(N) = 2 T(N/2) + N**
- A few ways to solve this recurrence, i.e., visualizing equation as a tree

$$= \sum_{i=0}^{\log N} 2^i c \frac{N}{2^i}$$

$$= \sum_{i=0}^{\log N} cN = cN \log N$$

Quicksort

- Choose an element as the **pivot**
- Partition the array into elements greater than pivot and elements less than pivot
- Quicksort each partition

Choosing a Pivot

- The worst case for Quicksort is when the partitions are of size zero and **N-1**
- Ideally, the pivot is the median, so each partition is about half
- If your input is random, you can choose the first element, but this is very bad for presorted input!
- Choosing randomly works, but a better method is...

Median-of-Three

- Choose three entries, use the median as pivot
- If we choose randomly, **2/N** probability of worst case pivots
- Median-of-three gives **0** probability of worst case, tiny probability of 2nd-worst case. (Approx. $2/N^3$)
- Randomness less important, so choosing (first, middle, last) works reasonably well

Partitioning the Array

- Once pivot is chosen, swap pivot to end of array. Start counters **i**=1 and **j**=**N-1**
- Intuition: **i** will look at less-than partition, **j** will look at greater-than partition
- Increment **i** and decrement **j** until we find elements that don't belong (**A[i]** > **pivot** or **A[j]** < **pivot**)
- Swap (**A[i]**, **A[j]**), continue increment/decrements
- When **i** and **j** touch, swap pivot with **A[j]**

Quicksort Worst Case

- Running time recurrence includes the cost of partitioning, then the cost of 2 quicksorts
- We don't know the size of the partitions, so let **i** be the size of the first partition
- **T(N) = T(i)+T(N-i-1) + N**
- Worst case is **T(N) = T(N-1) + N**

Quicksort Properties

- Unstable
- Average time O(N log N)
- Worst case time $O(N^2)$

Quick Sort

3	7	5	2	6	1	0	4
3	7	5	2	6	1	0	4
3	0	5	2	6	1	7	4
3	0	5	2	6	1	7	4
3	0	1	2	6	5	7	4
2	0	1	3	6	5	7	4
0	1	2	3	6	5	7	4
0	1	2	3	6	5	7	4
0	1	2	3	6	5	4	7
0	1	2	3	4	5	6	7

QuickSort Space

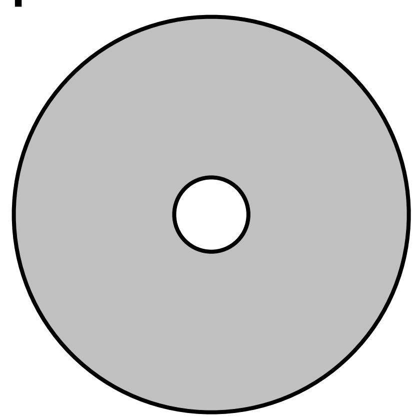
- QuickSort is a recursive algorithm
- Each recursive call sorts a segment of the array, it must store the beginning and end of the segment
- When the deepest recursive call is made, between N-1 and log N nested calls have occurred

External Sorting

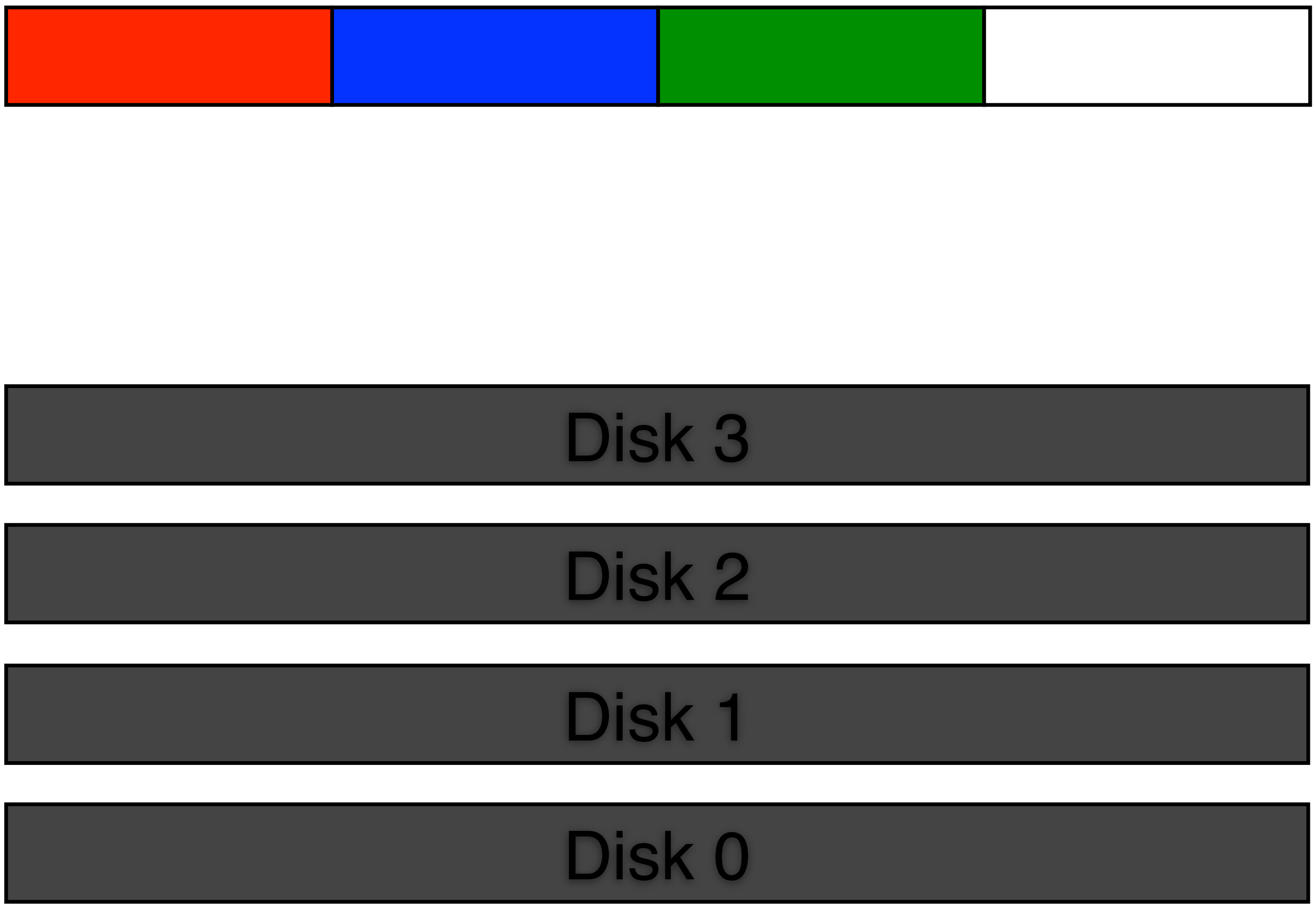
- So far, we have looked at sorting algorithms when the data is all available in RAM
- Often, the data we want to sort is so large, we can only fit a subset in RAM at any time
- We could run standard sorting algorithms, but then we would be swapping elements to and from disk
  - Instead, we want to minimize disk I/O, even if it means more CPU work

MergeSort

- We can speed up external sorting if we have two or more disks (with free space) via Mergesort
- One nice feature of Mergesort is the merging step can be done online with streaming data
- Read as much data as you can, sort, write to disk, repeat for all data, write output to alternating disks
  - merge outputs using 4 disks



External Sorting

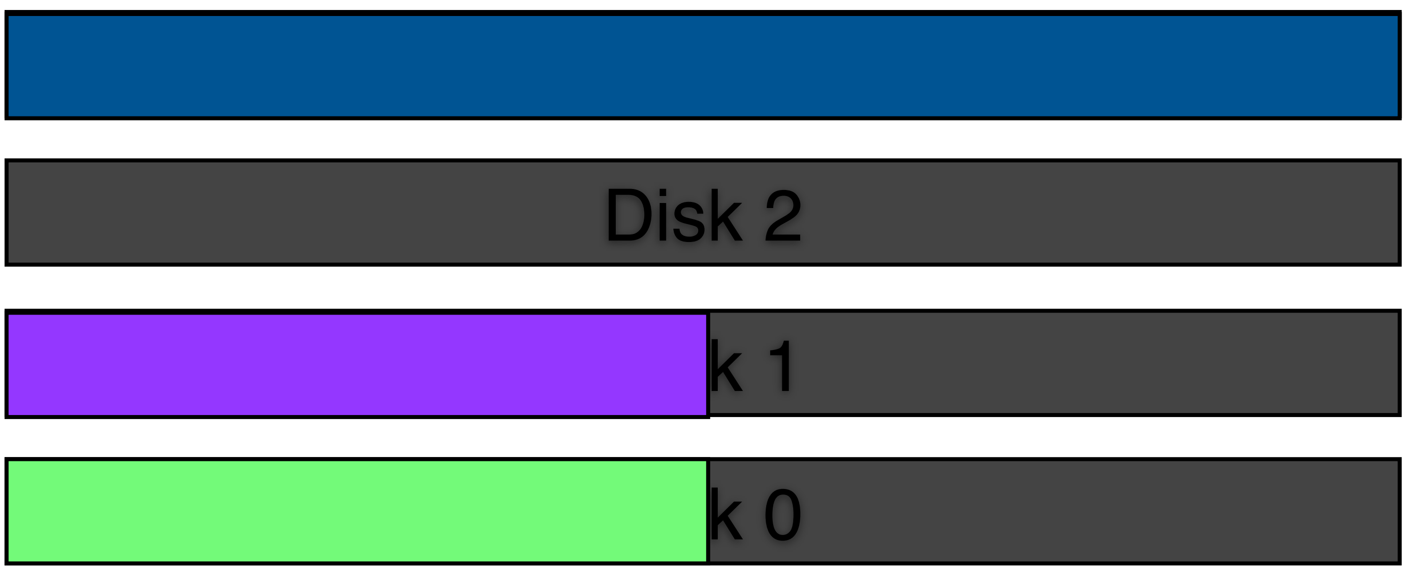


External Sorting

External Sorting

External Sorting

External Sorting



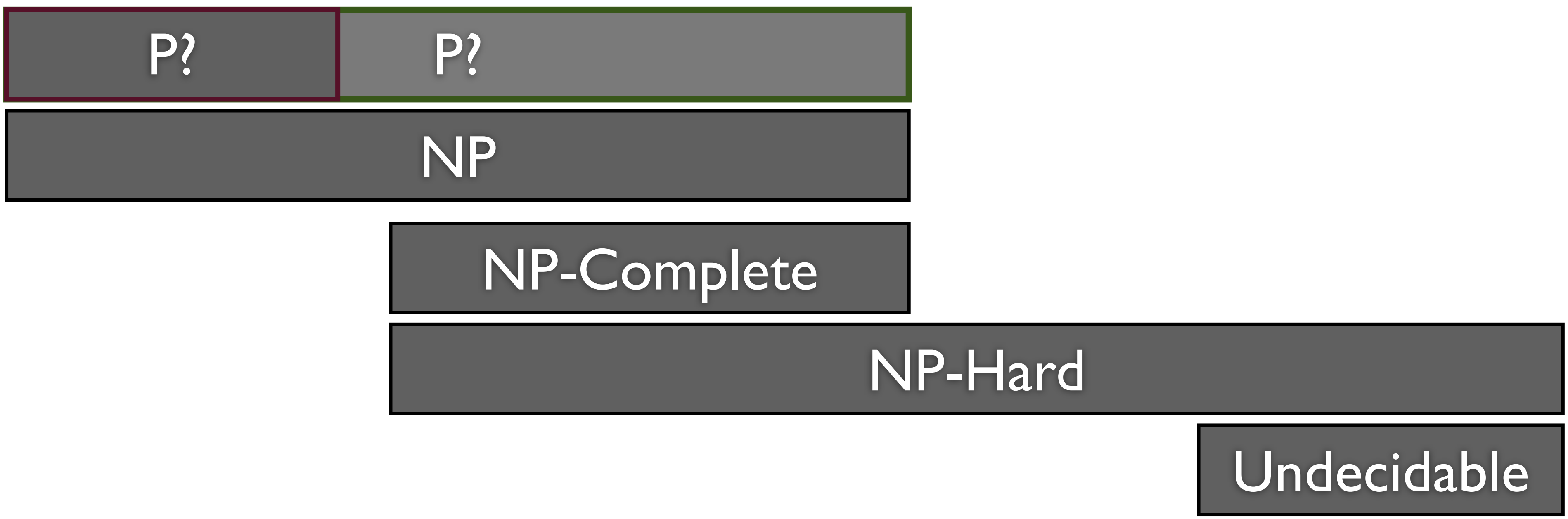
Complexity of Problems

- We've been concerned with the complexity of *algorithms*
- It is important to also consider the complexity of *problems*
- Understanding complexity is important for theory, and also for practice
  - understanding the hardness of problems helps us build better algorithms

Complexity Classes

- **P** - solvable in polynomial time
- **NP** - solvable in polynomial time by a nondeterministic computer
  - i.e., you can check a solution in polynomial time
- **NP-complete** - a problem in NP such that any problem in NP is polynomially reducible to it
- NP-Hard
- **Undecidable** - no algorithm can solve the problem

Complexity Class Hierarchy



NP-Complete Problems Satisfiability

- Given Boolean expression of N variables, can we set variables to make expression true?
- First NP-Complete proof because Cook's Theorem gave polynomial time procedure to convert any NP problem to a Boolean expression
- I.e., if we have efficient algorithm for Satisfiability, we can efficiently solve any NP problem

NP-Complete Problems Graph Coloring

- Given a graph is it possible to color with **k** colors all nodes so no adjacent nodes are the same color?
- Coloring countries on a map
- Sudoku is a form of this problem. All squares in a row, column and blocks are connected. **k** = 9

NP-Complete Problems
Hamiltonian Path

- Given a graph with N nodes, is there a path that visits each node exactly once?

NP-Hard Problems Traveling Salesman

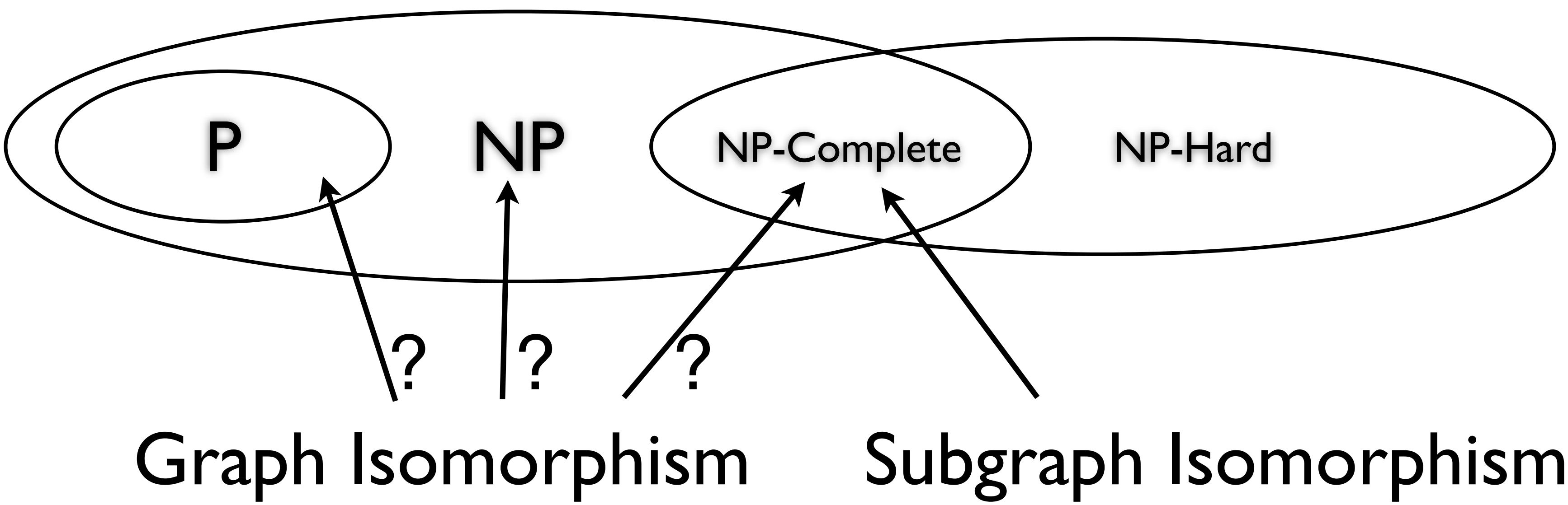
- Closely related to Hamiltonian Path problem
- Given complete graph **G**, find the shortest path that visits all nodes
- If we are able to solve TSP, we can find a Hamiltonian Path; set connected edge weight to constant, disconnected to infinity
  - TSP is NP-hard

Poly. Time Approximation

- Certain optimization NP-Hard problems have **polynomial time approximation schemes** (PTAS)
  - An efficient method to find a solution within a constant of the true optimum
    - e.g., Optimal TSP path length = ℓ
      PTAS TSP path length $\leq \ell(1+\epsilon)$
  - For fixed constant, must be poly. time, but can scale poorly w.r.t. constant
  - E.g., $O(p(N)^{(\frac{1}{\epsilon}!)})$ is a valid PTAS time

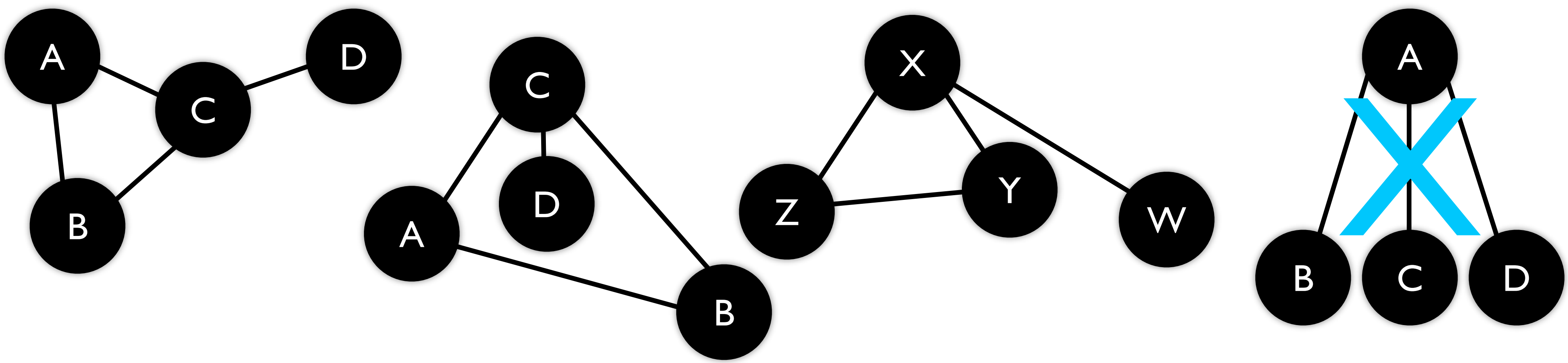
Graph Isomorphism Complexity

- The Graph Isomorphism problem is NP,
  - but is unknown if NP-Complete/Hard,
  - and no poly. time algorithm is known



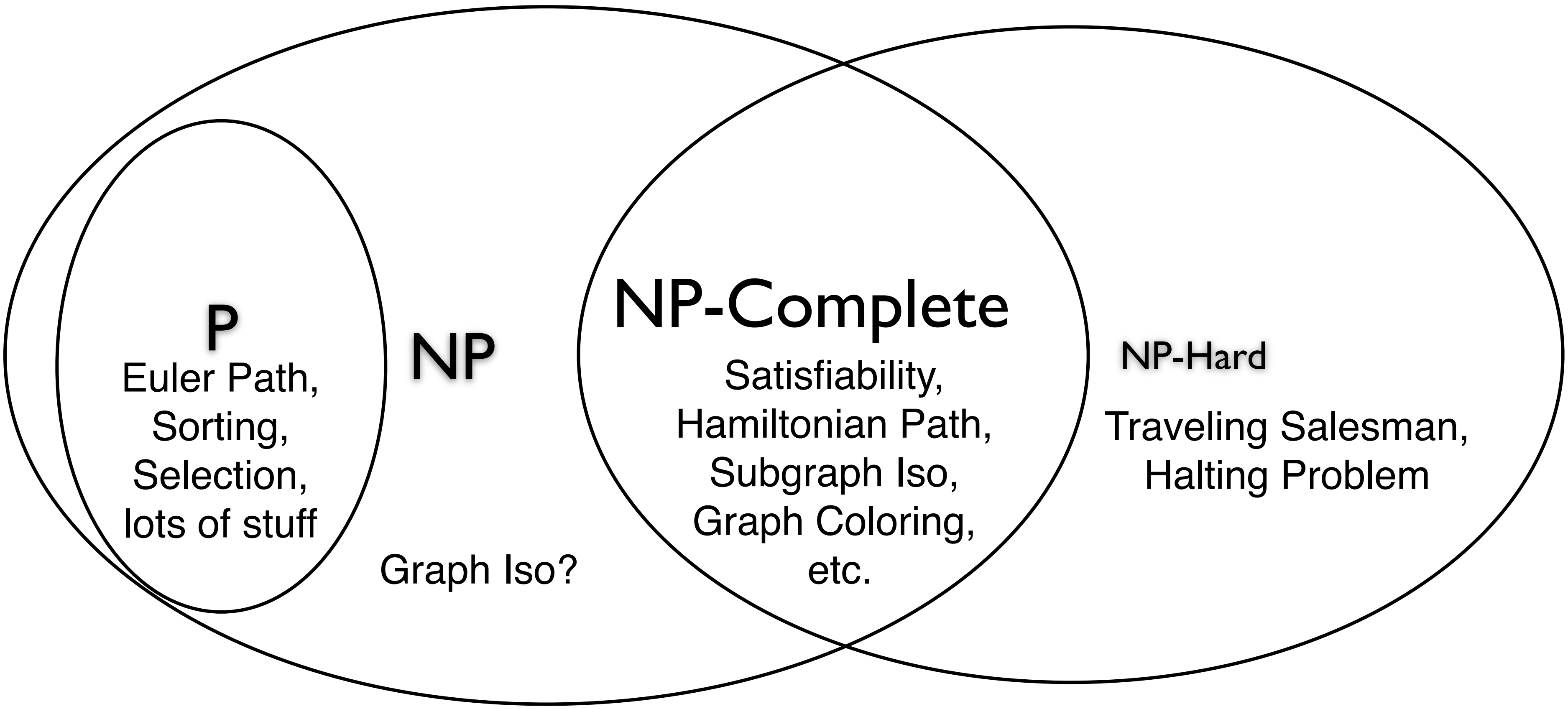
Graph Isomorphism Definition

- Given graphs G and H, is there a 1-to-1 mapping of vertices from G to vertices from H that preserves the edge structure?



- Subgraph Isomorphism: is a subgraph of G isomorphic to H?

Complexity


P
Euler Path,
Sorting,
Selection,
lots of stuff
NP
Graph Iso?
NP-Complete
Satisfiability,
Hamiltonian Path,
Subgraph Iso,
Graph Coloring,
etc.
NP-Hard
Traveling Salesman,
Halting Problem

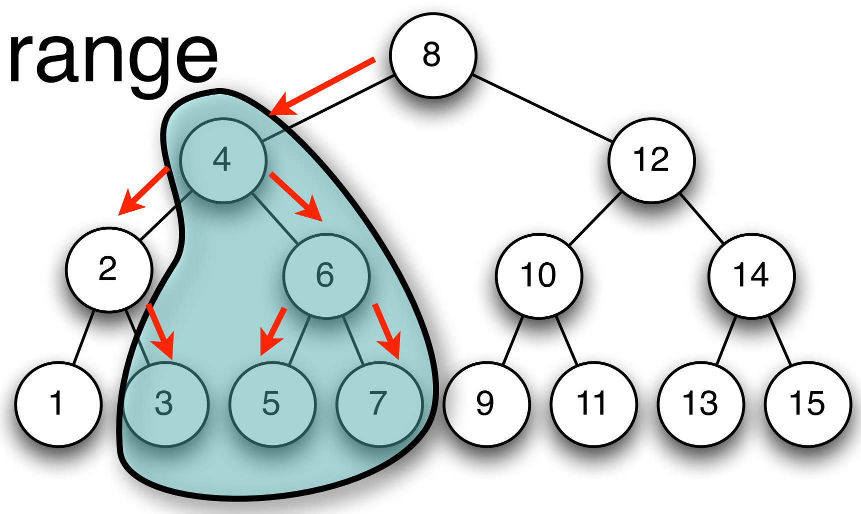
kd-Trees

- Useful data structure for data mining and machine learning applications
- Store elements by k-dimensional keys
  - e.g., age, height, weight
- Retrieve elements by ranges in the k dimensions
  - e.g., Searching for a new basketball center
    18-24 year olds, 6’6”-7’4”, 200+ lbs

1-d Range Search

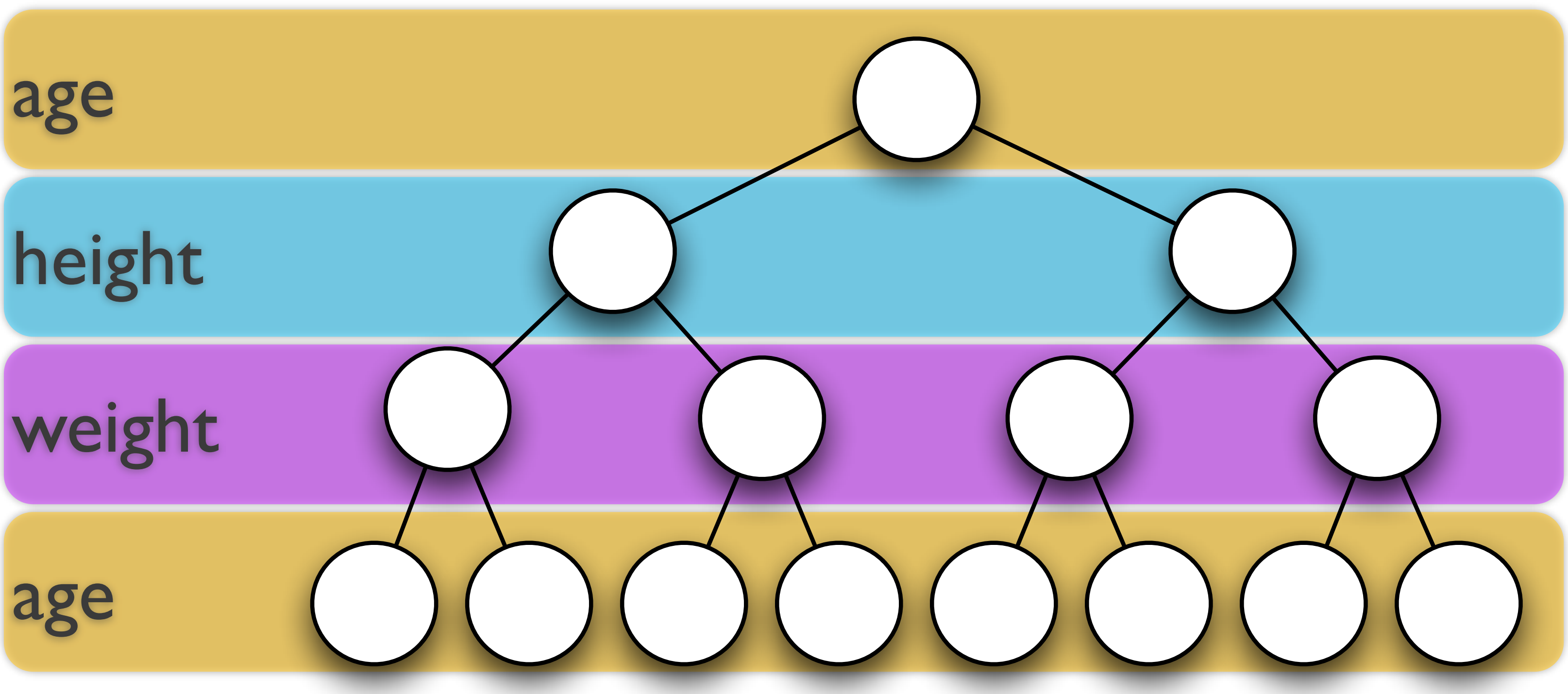
- BST recursive search:
  (1) if **key** is in range, print node
  (2) if **key** > **lower bound**, search left
  (3) if **key** < **upper bound**, search right
- **O(M+log N)** for **M** items returned
  - **O(log N)** to find nodes in range
  - **O(1)** at each node

Search for 3-7



kd-Tree Structure

- Binary search tree
  - each level splits on alternating keys



Search Algorithm

- Given lower and upper bounds for each dimension
- If key is in range, print
  If key > current dimension's lower bound
  search left child
  If key < current dimension's upper bound
  search right child
- Insert recursion is just like BST

kd-Tree Analysis

- Since each level represents a different keys, balancing is not possible
- If we have all the points, we can build a perfectly balanced tree. How?
- Then worst case $O(M + kN^{1-1/k})$

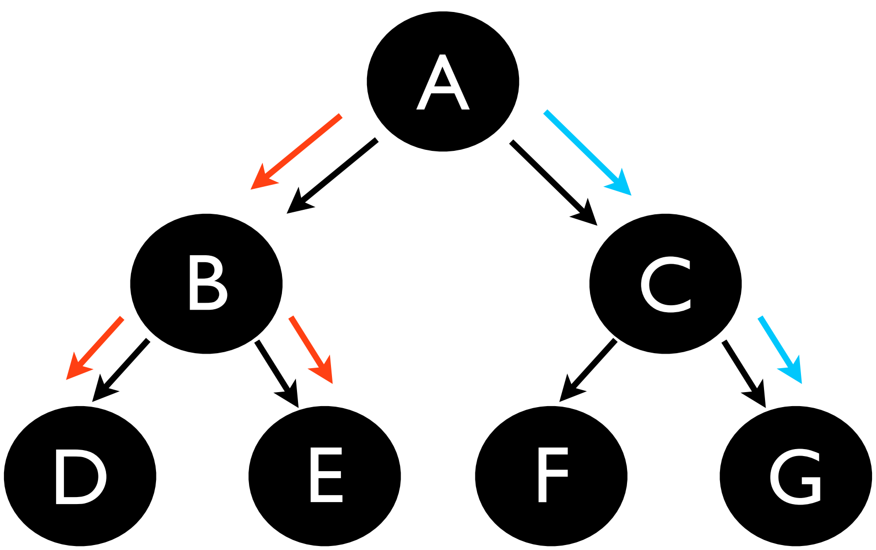
Nearest Neighbor Search

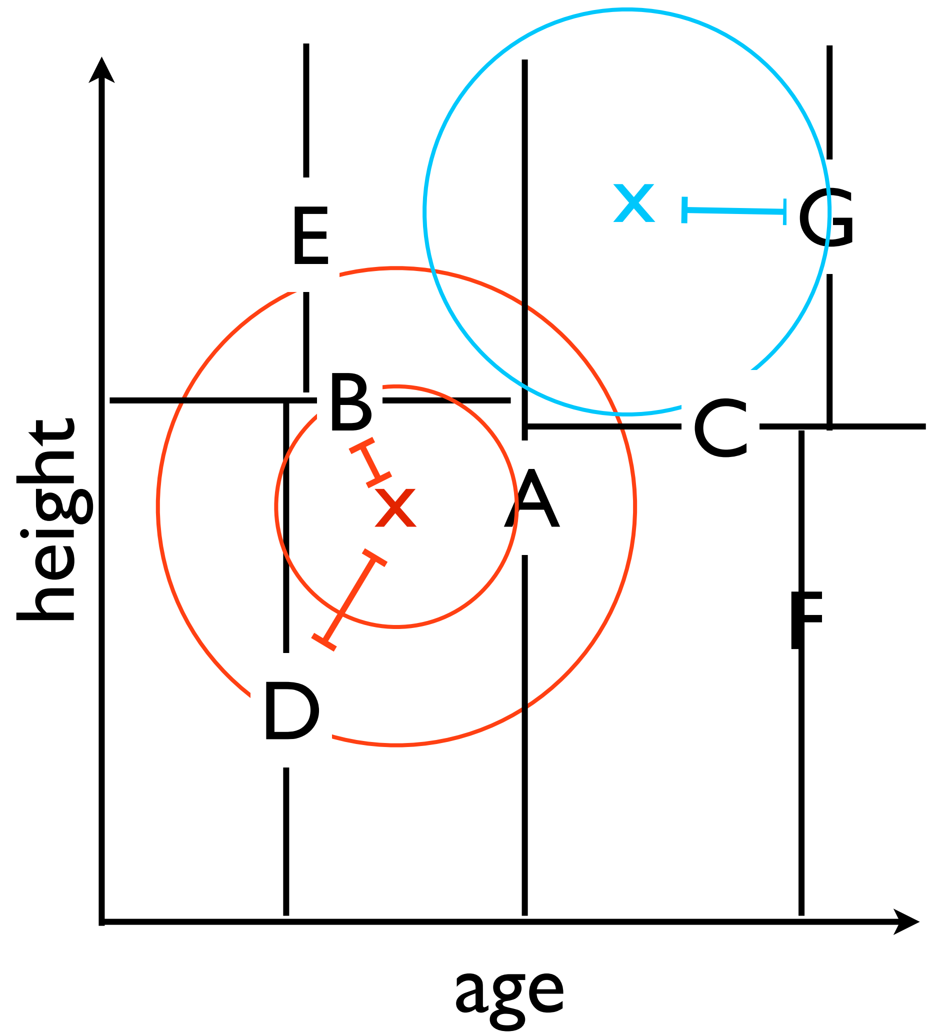
- kd-trees are especially helpful for finding nearest neighbors
- Given a data set, find the nearest point to any element **x**
- Naive O(N) approach is to compute distances everywhere
- Instead, kd-tree offers $O(kN^{1-1/k})$

Nearest Neighbor Algorithm

- Search for **x** in the kd-tree until you reach a leaf
  - Consider leaf point *current-best*
- Backtrack along search path, and at each node:
  - If current point is better, redefine *current-best*
  - If best can be in the unexplored child*, recurse down the unexplored child

Algorithm Illustration





Reading

- pre-midterm: Weiss Ch. 2, 3, 4, 6
- post-midterm: Weiss Ch. 5, 7, 8, 9, 12.6
- See schedule on class website for specific sections (i.e., which to skip)