CSE 326: Data Structures
Final Exam Review

James Fogarty

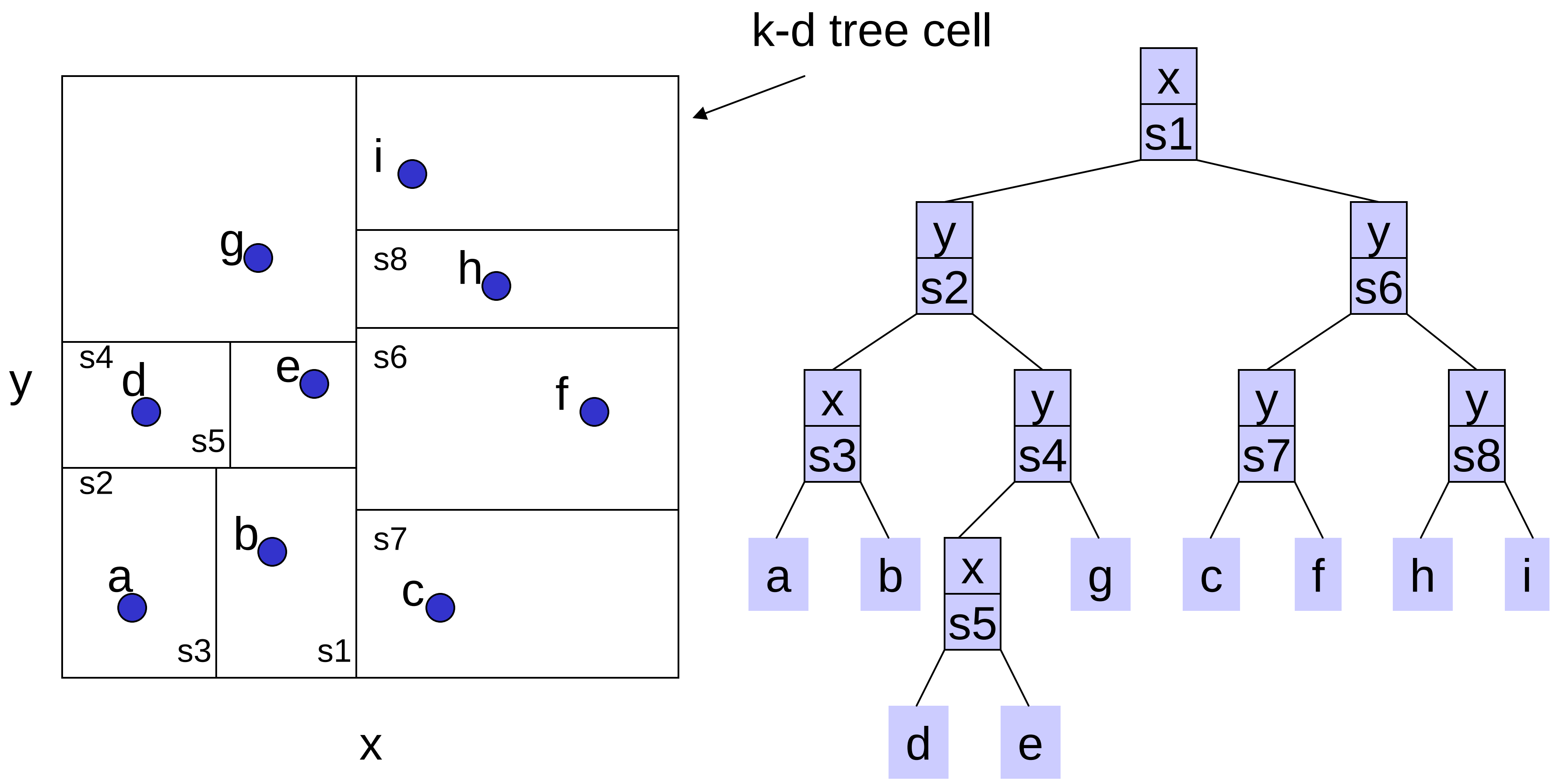
Autumn 2007

Lecture *n - 1*

Announcements

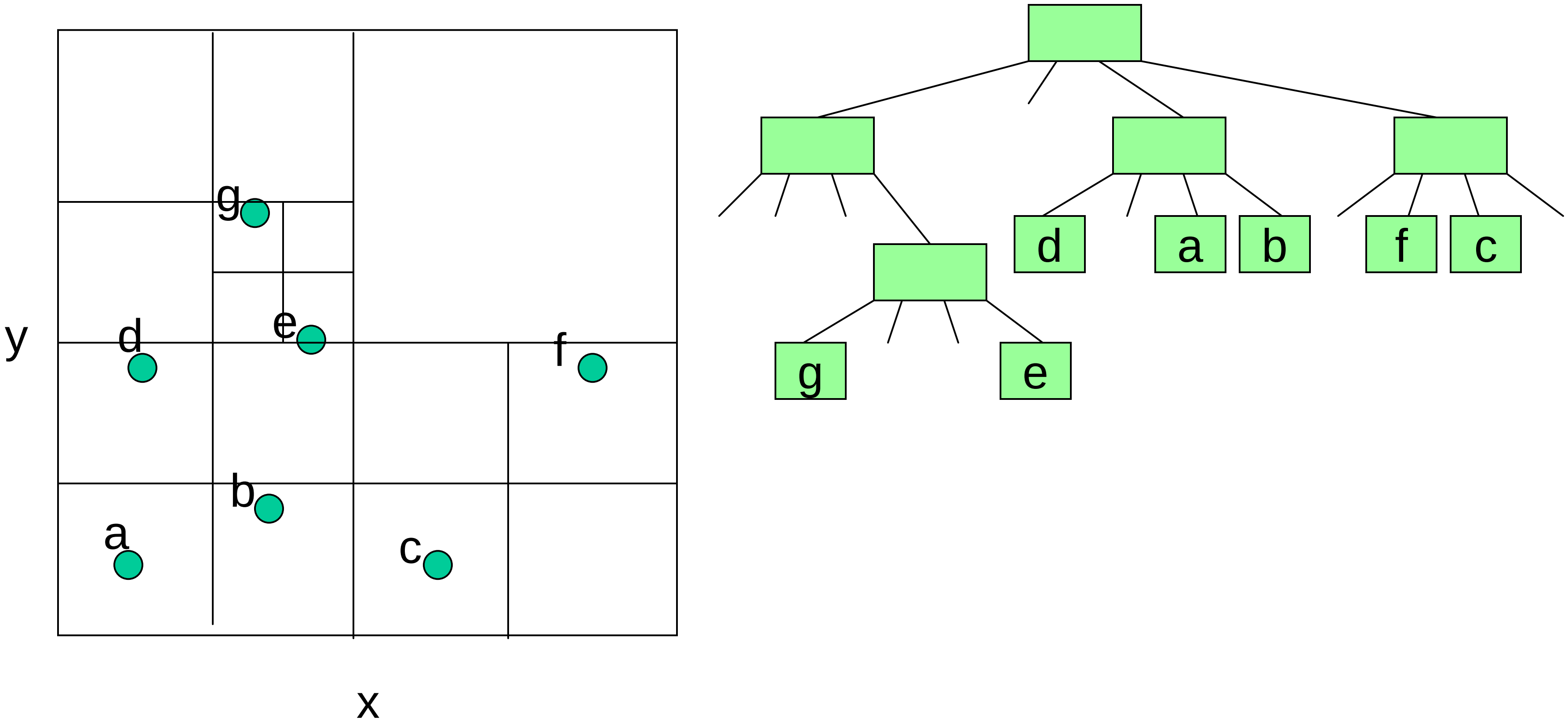
- Exam Wednesday 2:30pm, 2 hours, here in ARC 160
  - Logistics: same as midterm (closed book)
- Comprehensive
  - Everything up to, but not including, Data Compression
  - Also not anything about A*
  - So look over the midterm review again, in addition to this

k-d Tree Construction (18)



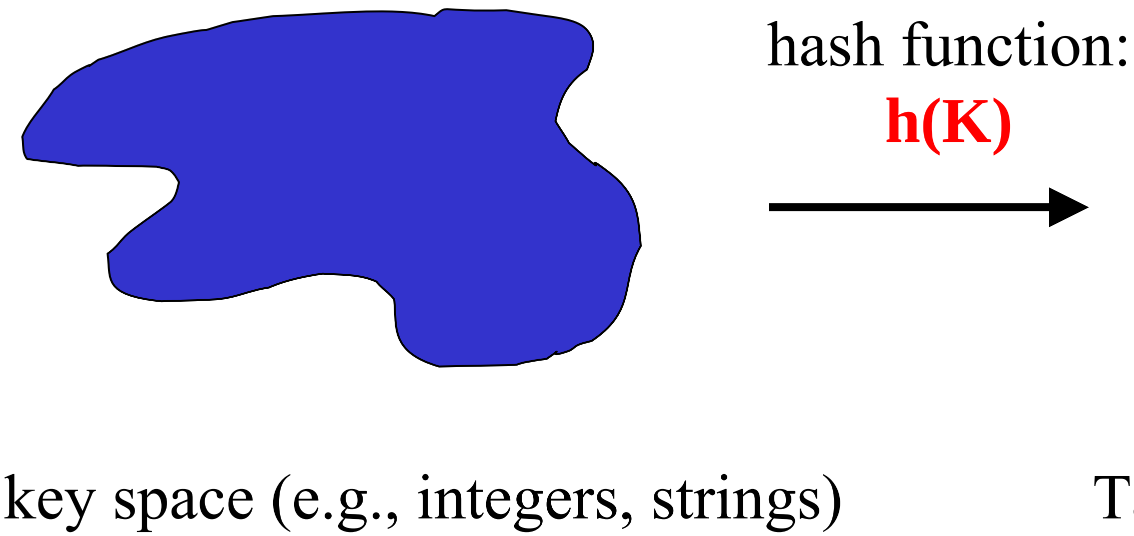
Quad Trees

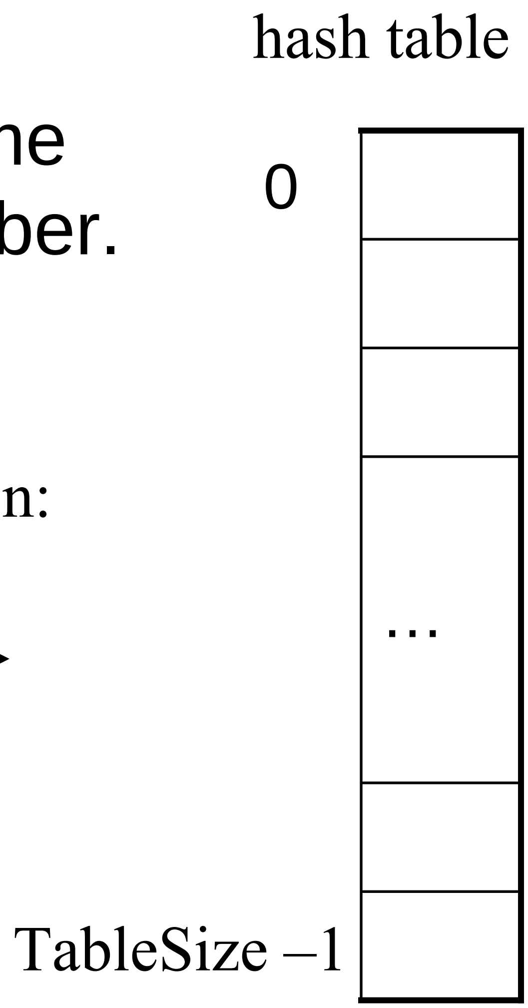
- Space Partitioning



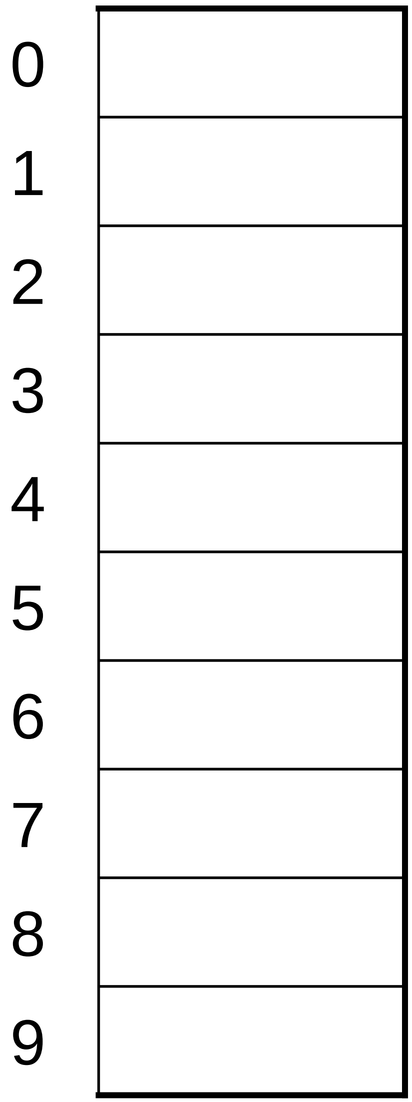
Hash Tables

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





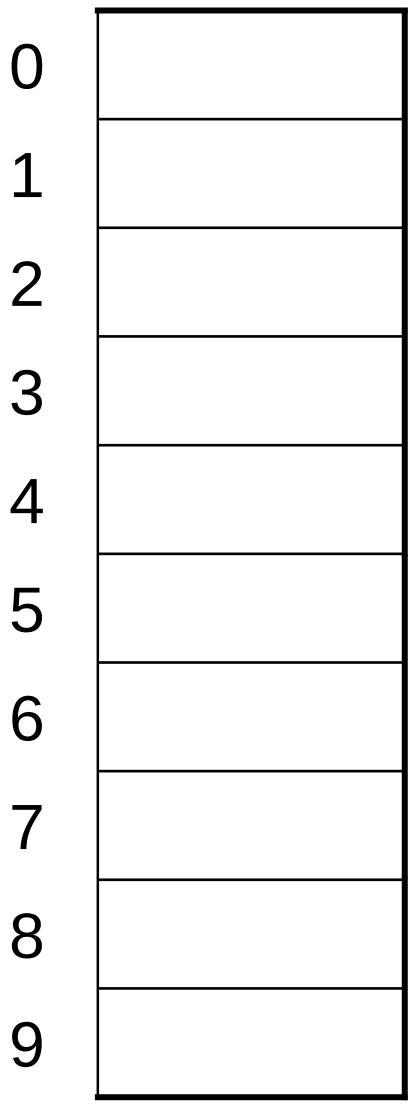
Separate Chaining



Insert:
10
22
107
12
42

- **<u>Separate chaining</u>:** All keys that map to the same hash value are kept in a list (or "bucket").

Open Addressing



Insert:
38
19
8
109
10

- **Linear Probing**: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Linear Probing

$f(i) = i$

- Probe sequence:

  0^{th} probe = $h(k) \bmod TableSize$

  1^{th} probe = $(h(k) + 1) \bmod TableSize$

  2^{th} probe = $(h(k) + 2) \bmod TableSize$

  . . .

  i^{th} probe = $(h(k) + i) \bmod TableSize$

Quadratic Probing

Less likely to encounter Primary Clustering

$f(i) = i^2$

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = $(h(k) + i^2)$ mod TableSize

Double Hashing

$f(i) = i * g(k)$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

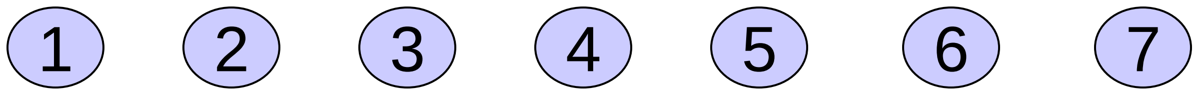
- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Disjoint Union - Find

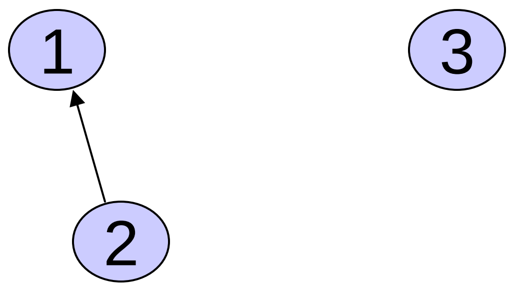
- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, one of its members
  - {3,<u>5</u>,7} , {4,2,<u>8</u>}, {<u>9</u>}, {<u>1</u>,6}
- Union(x,y) – take the union of two sets named x and y
- Find(x) – return the name of the set containing x.

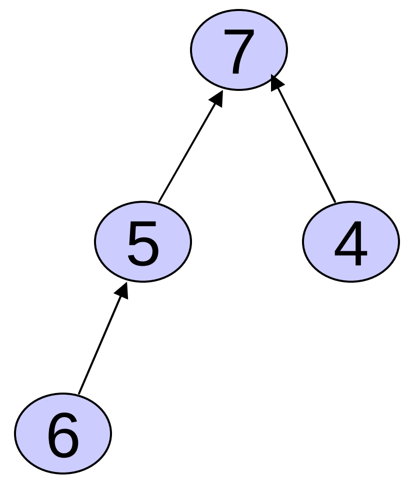
Up-Tree for DU/F

Initial state



Intermediate state

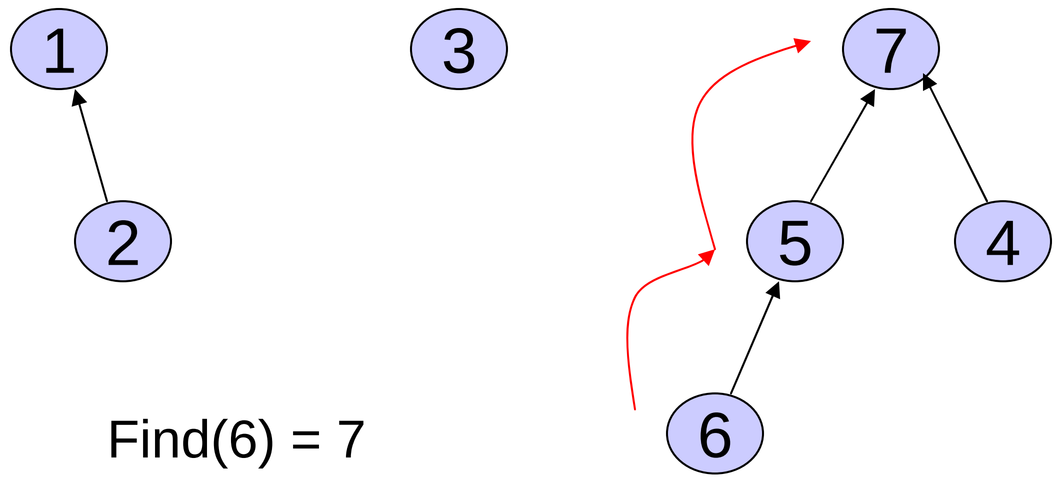




Roots are the names of each set.

Find Operation

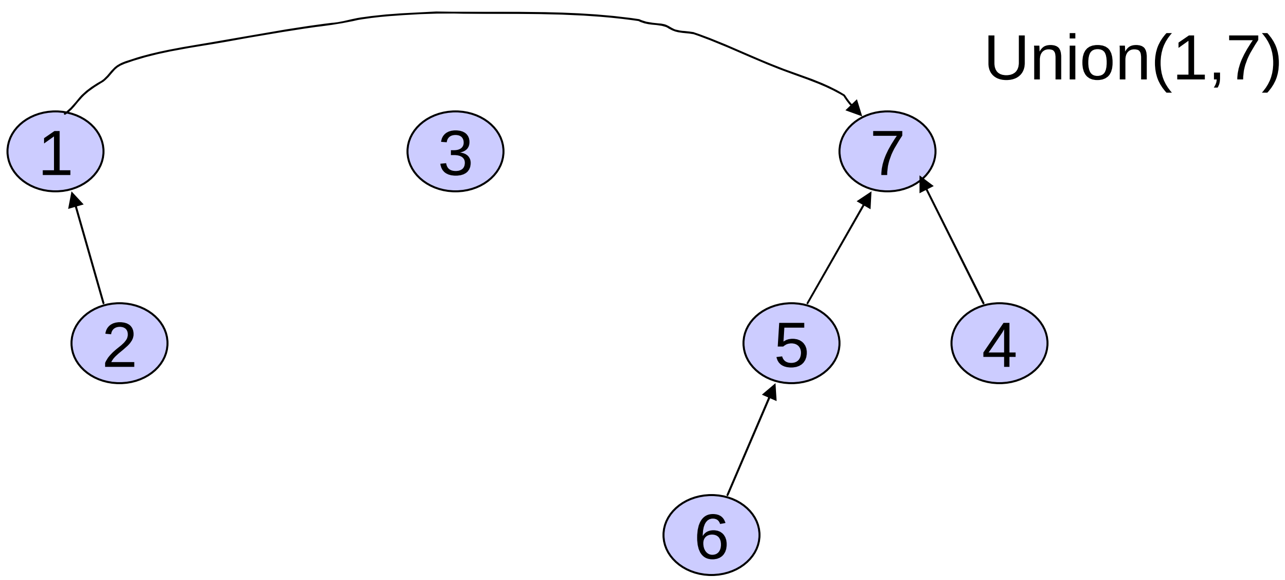
- Find(x) follow x to the root and return the root



Find(6) = 7

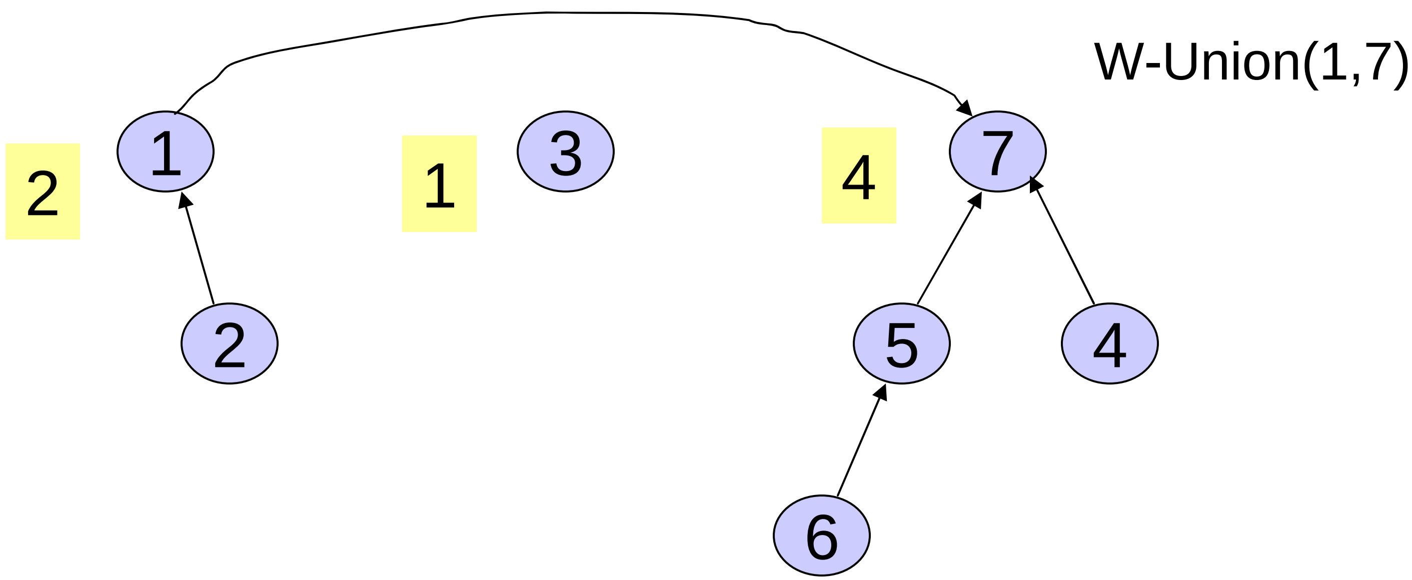
Union Operation

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



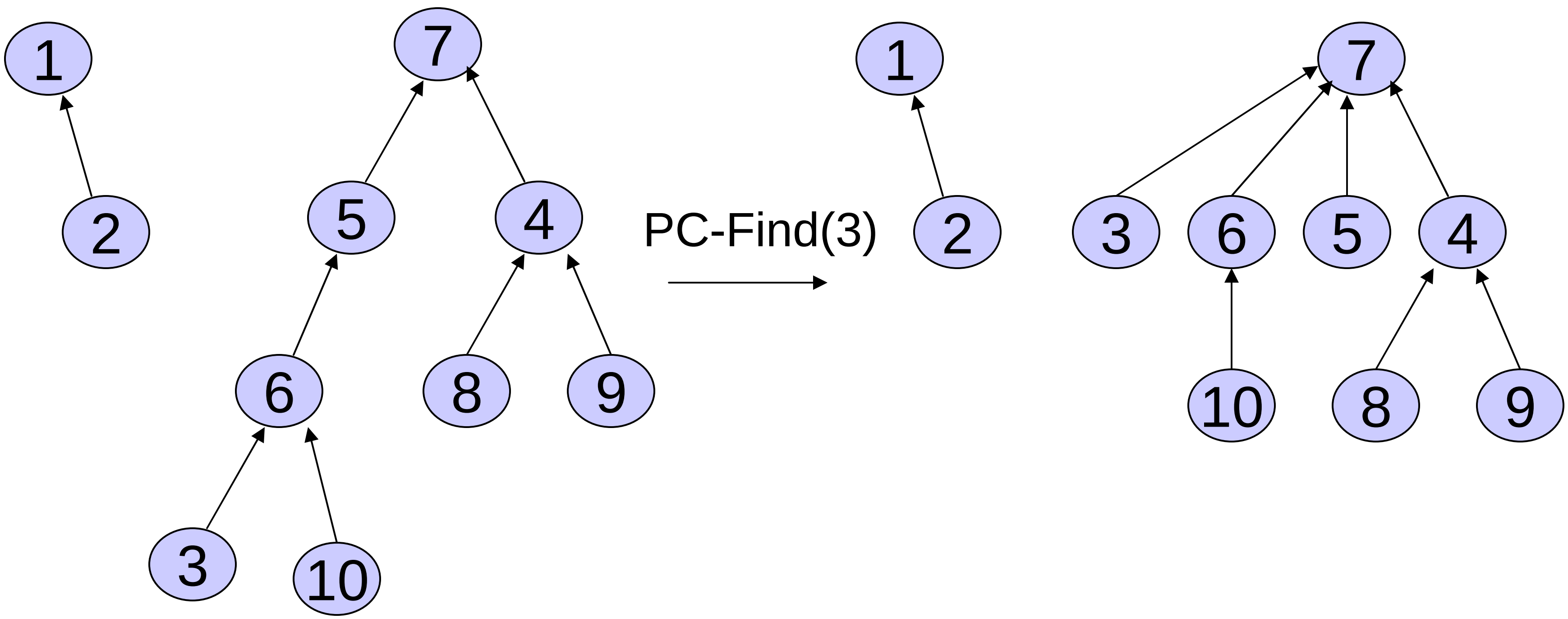
Weighted Union

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



Path Compression

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



Sorting: *The* ***Big*** *Picture*

Given *n* comparable elements in an array, sort them in an increasing order.

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

Insertion Sort: Idea

- At the k^{th} step, put the k^{th} input element in the correct place among the first k elements
- Result: After the k^{th} step, the first k elements are sorted.

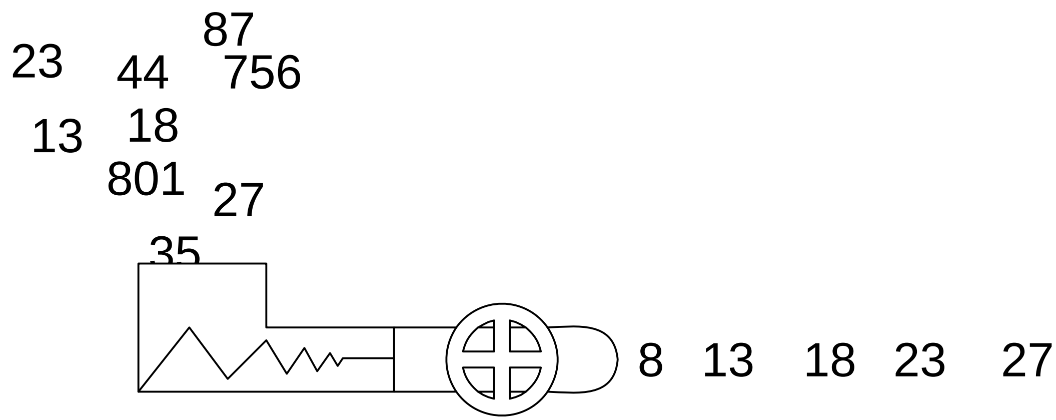
Runtime:

worst case :
best case :
average case :

Selection Sort: idea

- Find the smallest element, put it 1st
- Find the next smallest element, put it 2nd
- Find the next smallest, put it 3rd
- And so on …

HeapSort:
Using Priority Queue ADT (heap)



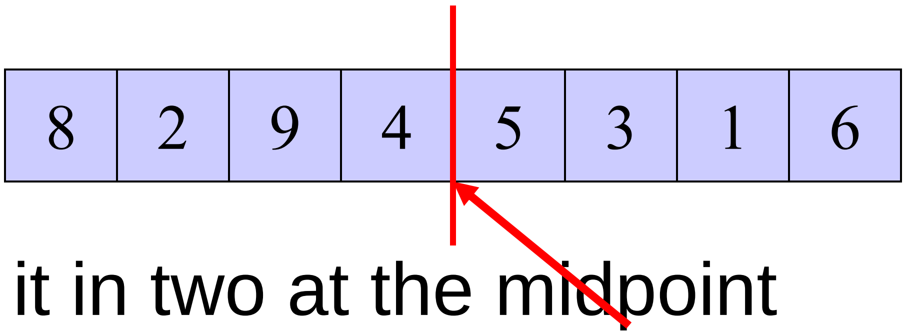
Shove all elements into a priority queue,
take them out smallest to largest.

Runtime:

“Divide and Conquer”

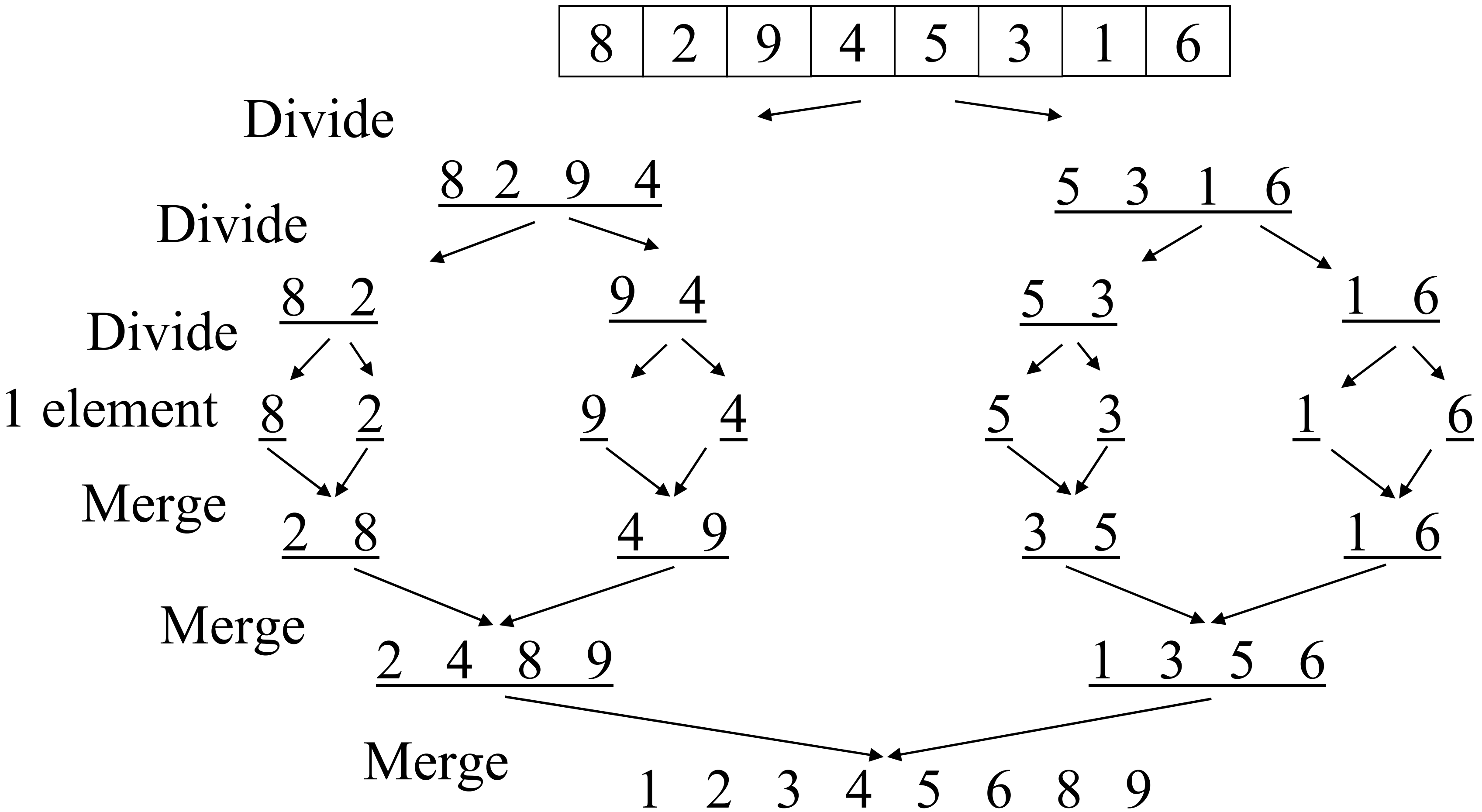
- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → known as Mergesort
- **Idea 2 :** Partition array into small items and large items, then recursively sort the two sets → known as Quicksort

Mergesort



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

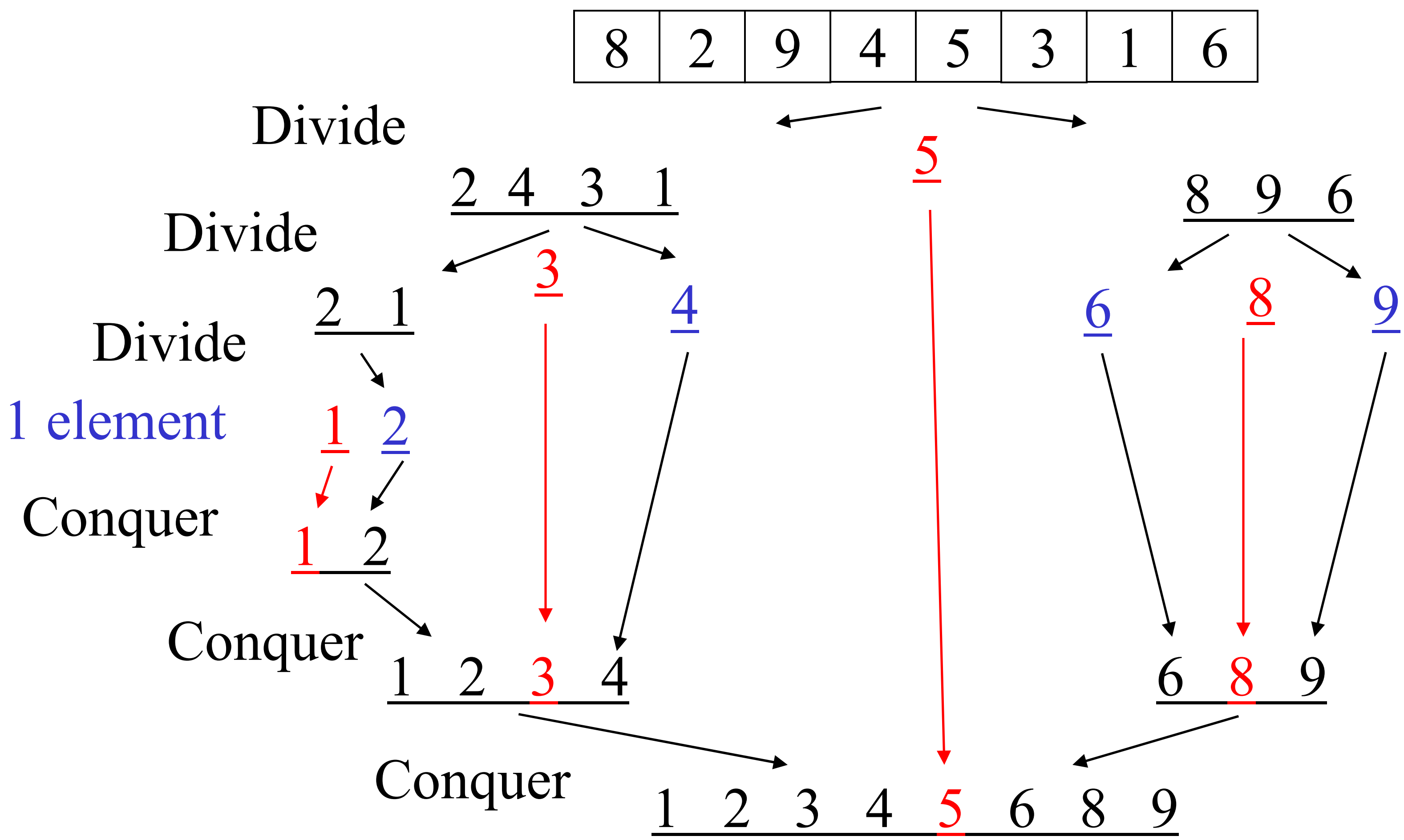
Mergesort Example



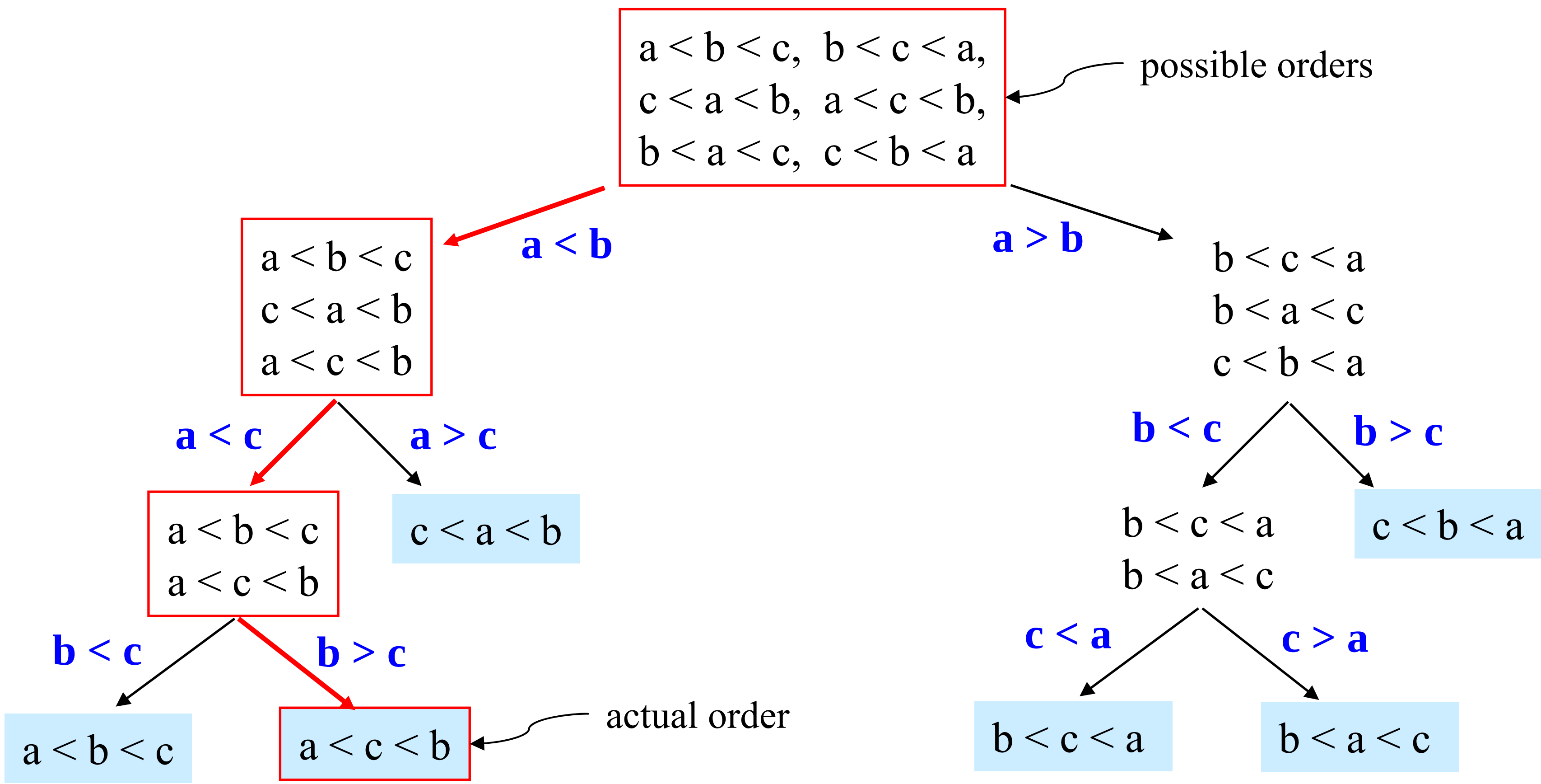
Quicksort

- 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 rigvht sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

Quicksort Example



Decision Tree Example



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?

Fixing impracticality: RadixSort

- Radix = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything

- Idea: BucketSort on each **digit**,
  least significant to most significant
  (lsd to msd)

Radix Sort Example (1st pass)

Bucket sort
by 1's digit

Input data

478
537
9
721
3
38
123
67

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

After 1st pass

721
3
123
537
67
478
38
9

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

Radix Sort Example (2nd pass)

After 1st 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 2nd pass

3
9
721
123
537
38
67
478

Radix Sort Example (3rd pass)

After 2nd pass

3
9
721
123
537
38
67
478

Bucket sort by 100's digit

0	1	2	3	4	5	6	7	8	9
003 009 038 067	123			478	537		721		

After 3rd pass

3
9
38
67
123
478
537
721

Invariant: after k passes the low order k digits are sorted.

Graph… ADT?

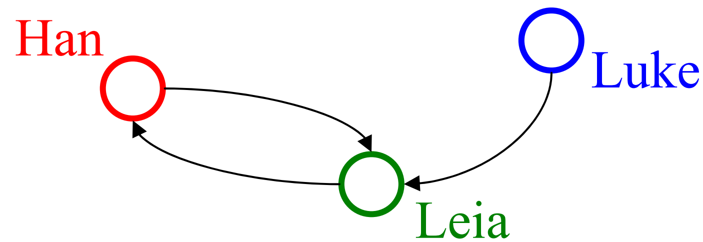
- Not quite an ADT…
  operations not clear

- A formalism for representing relationships between objects

  Graph **G = (V,E)**

  – *Set of vertices*:
  $\mathbf{V = \{v_1, v_2, \ldots, v_n\}}$

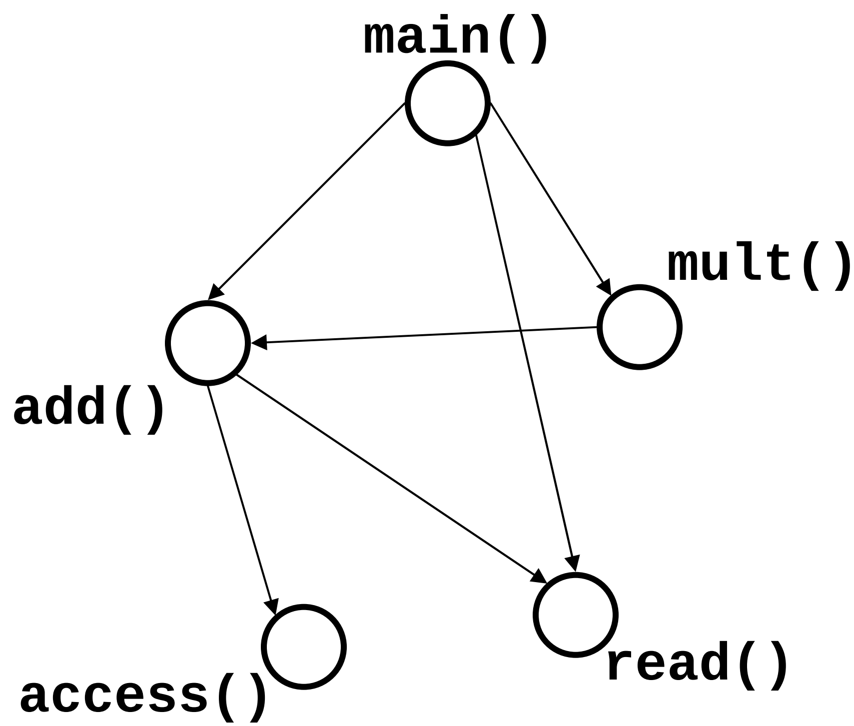
  – *Set of edges*:
  $\mathbf{E = \{e_1, e_2, \ldots, e_m\}}$
  where each $\mathbf{e_i}$ connects two vertices $\mathbf{(v_{i1}, v_{i2})}$




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

Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.

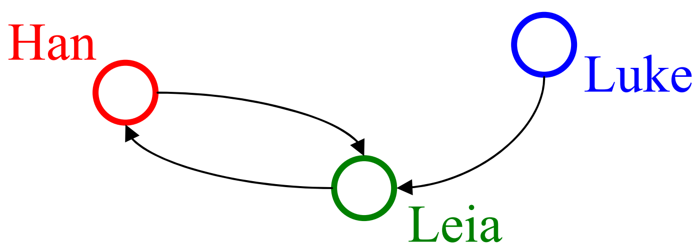


Aside: ***If program call-graph is a DAG, then all procedure calls can be in-lined***

{Tree} ⊂ {DAG} ⊂ {Graph}

Rep 1: Adjacency Matrix

A **|V| x |V|** array in which an element **(u,v)** is true if and only if there is an edge from **u** to **v**



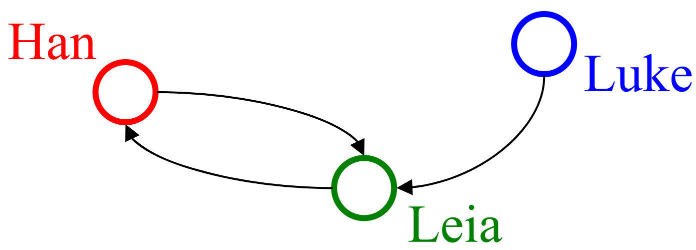
	Han	Luke	Leia
Han			
Luke			
Leia			

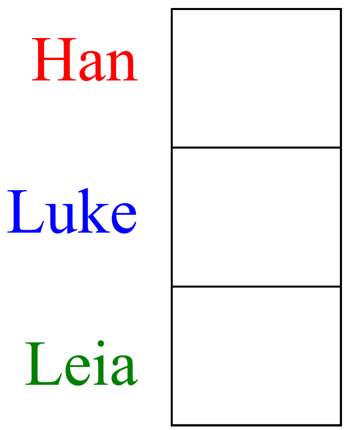
Runtimes:
Iterate over vertices?
Iterate over edges?
Iterate edges adj. to vertex?
Existence of edge?

Space requirements?

Rep 2: Adjacency List

A **|V|**-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices





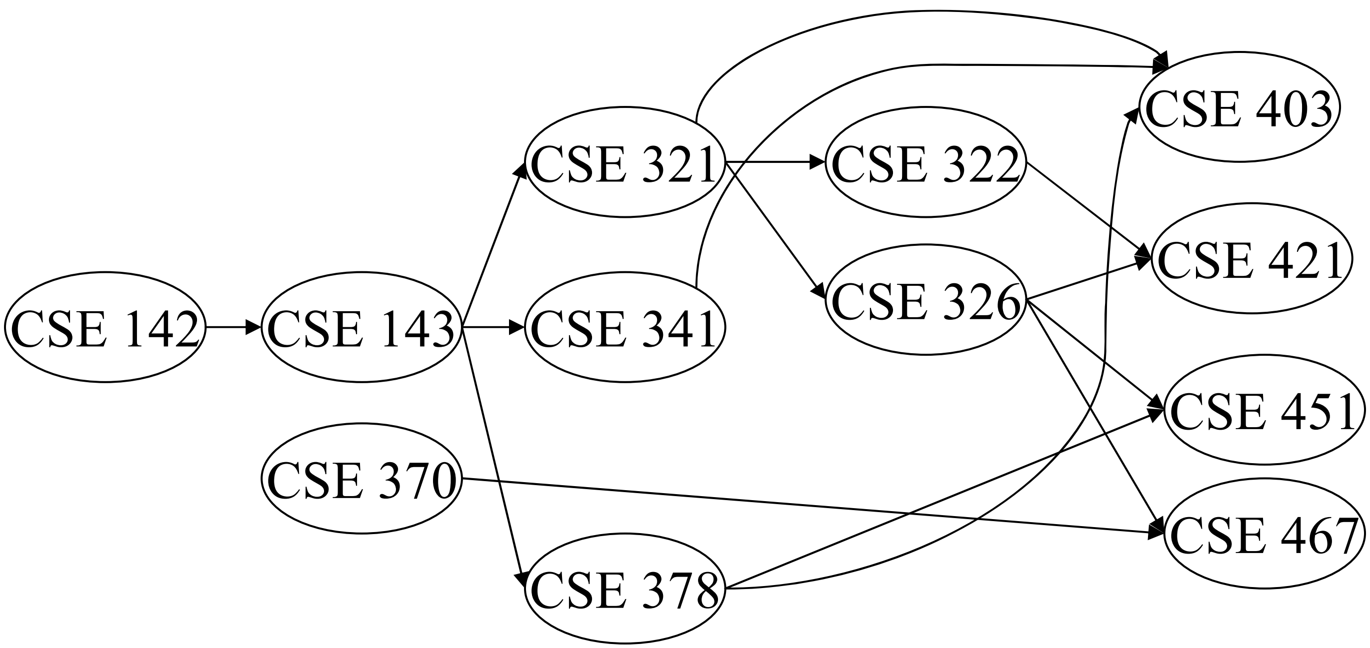
Runtimes:
Iterate over vertices?
Iterate over edges?
Iterate edges adj. to vertex?
Existence of edge?

Space requirements?

This is a partial ordering, for sorting we had a total ordering

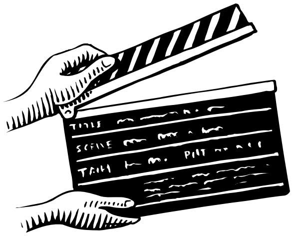
Application: Topological Sort

Given a directed graph, **`G = (V,E)`**, output all the vertices in **`V`** such that no vertex is output before any other vertex with an edge to it.



Is the output unique?

Minimize and
DO a topo sort



Topological Sort: Take Two

1. Label each vertex with its in-degree
2. Initialize a queue *Q* to contain all in-degree zero vertices
3. While *Q* not empty
   a. *v* = *Q*.dequeue; output *v*
   b. Reduce the in-degree of all vertices adjacent to *v*
   c. If new in-degree of any such vertex *u* is zero *Q*.enqueue(*u*)

Note: could use a stack, list, set, box, … instead of a queue

Runtime:

Comparison: DFS versus BFS

- Depth-first search
  - –Does not always find shortest paths
  - –Must be careful to mark visited vertices, or you could go into an infinite loop if there is a cycle
- Breadth-first search
  - –Always finds shortest paths – optimal solutions
  - –Marking visited nodes can improve efficiency, but even without doing so search is guaranteed to terminate

  - –Is BFS always preferable?

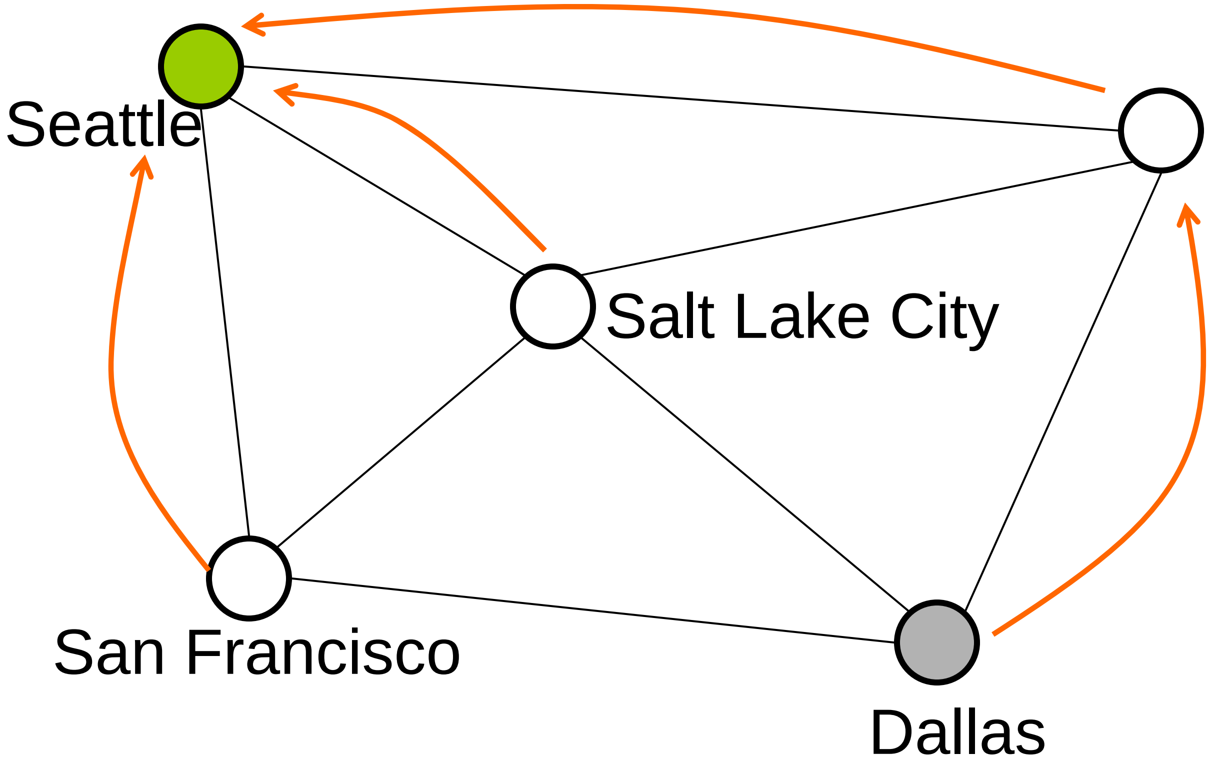
Iterative-Deepening DFS (II)

- IDFS_Search(Start, Goal_test)
-     i := 1;
-   repeat
-         answer := Bounded_DFS(Start, Goal_test, i);
-         if (answer != fail) then return answer;
-         i := i+1;
-   end

Saving the Path

- Our pseudocode returns the goal node found, but not the path to it
- How can we remember the path?
  - Add a field to each node, that points to the previous node along the path
  - Follow pointers from goal back to start to recover path

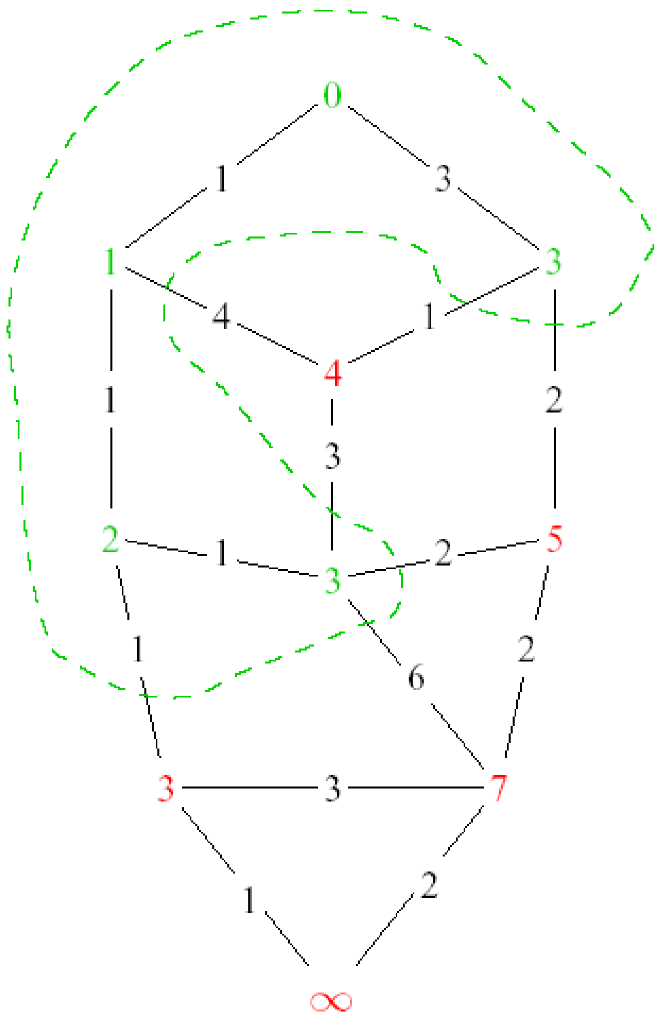
Example (Unweighted Graph)



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

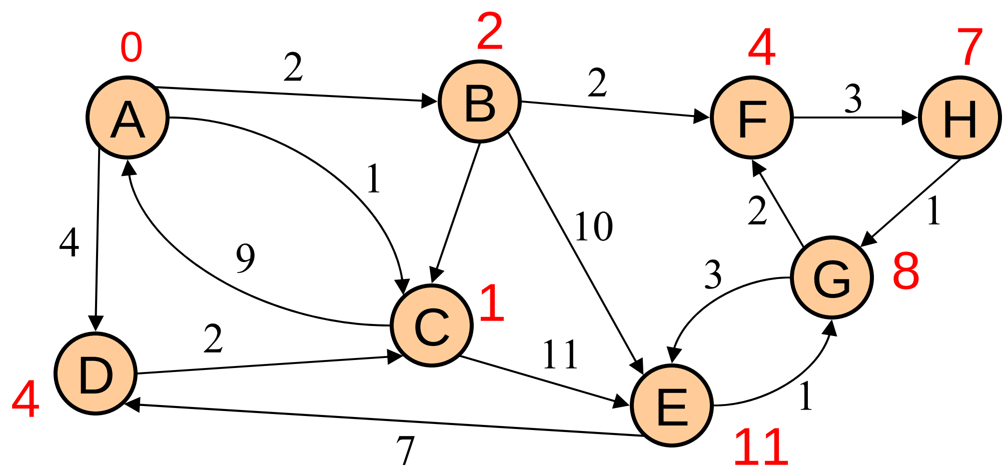


Adapt BFS to handle weighted graphs

Two kinds of vertices:

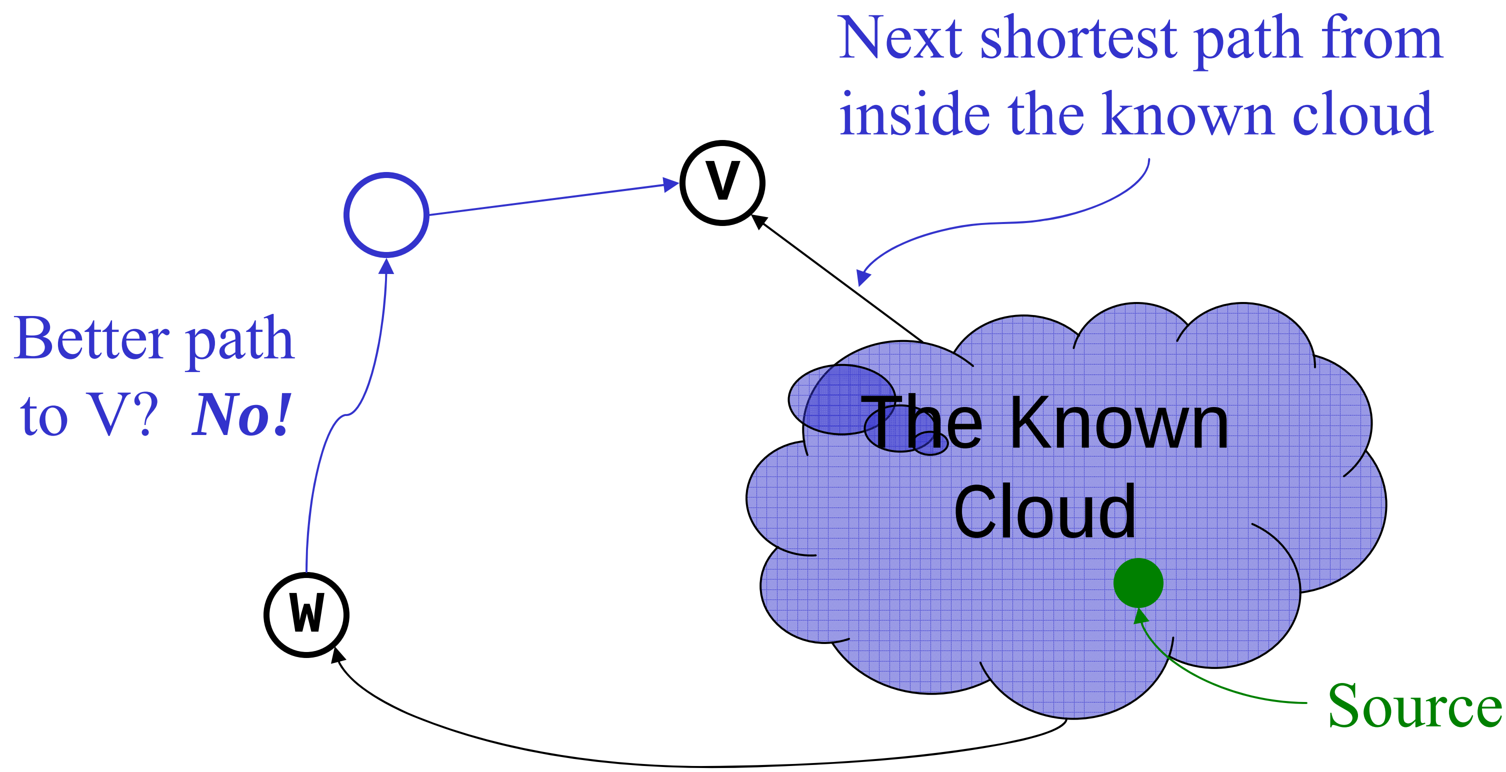
- Finished or known vertices
  - Shortest distance has been computed
- Unknown vertices
  - Have tentative distance

Dijkstra’s Algorithm in action



Vertex	Visited?	Cost	Found by
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

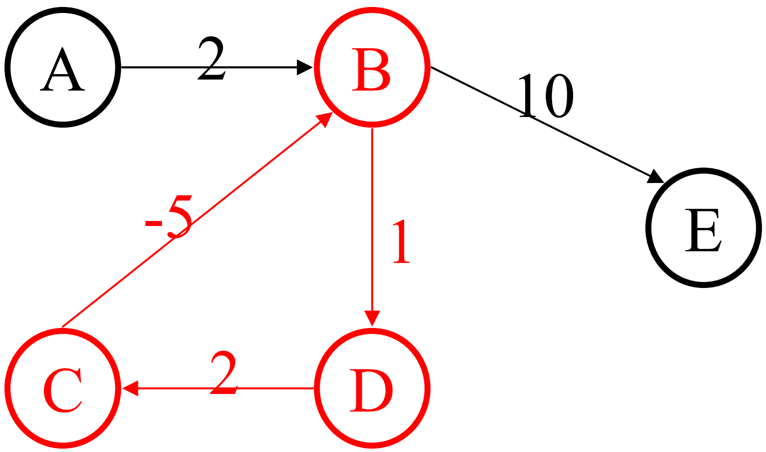
Correctness: The Cloud Proof



How does Dijkstra's decide which vertex to add to the Known set next?

- If path to **V** is shortest, path to **W** must be *at least as long*
  (or else we would have picked **W** *as the next vertex)*
- So the path through **W** to **V** cannot be any shorter!

The Trouble with Negative Weight Cycles



What's the shortest path from A to E?

Problem?

Dynamic Programming

Algorithmic technique that systematically records the answers to sub-problems in a table and re-uses those recorded results (rather than re-computing them).

Simple Example: Calculating the Nth Fibonacci number.

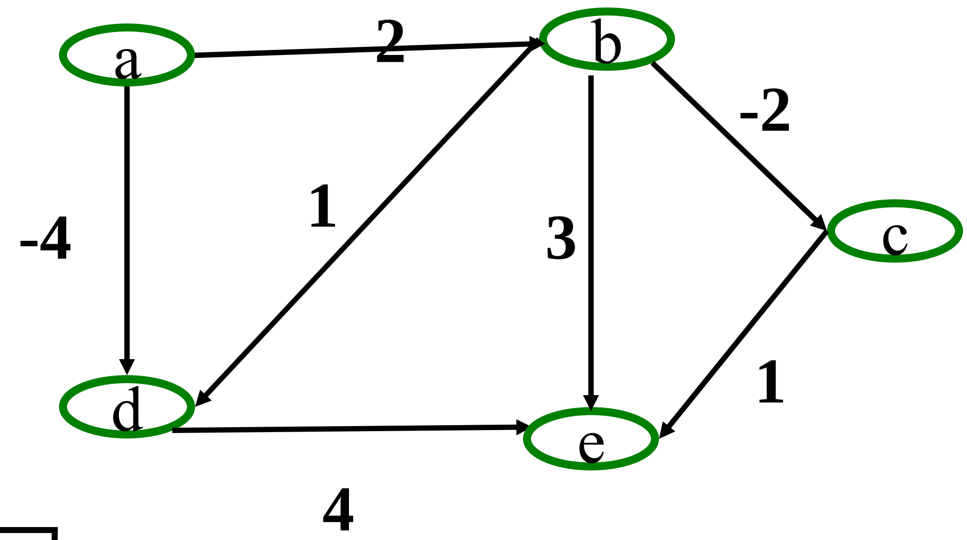
$$Fib(N) = Fib(N-1) + Fib(N-2)$$

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

Floyd-Warshall - for All-pairs shortest path

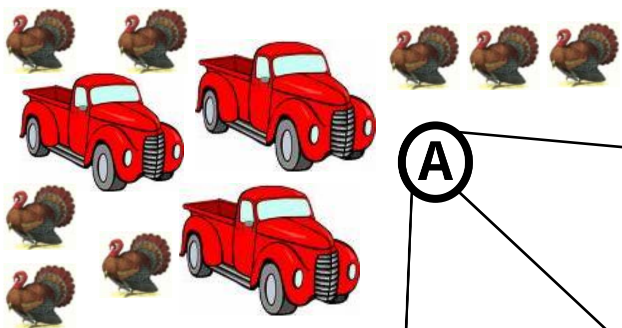


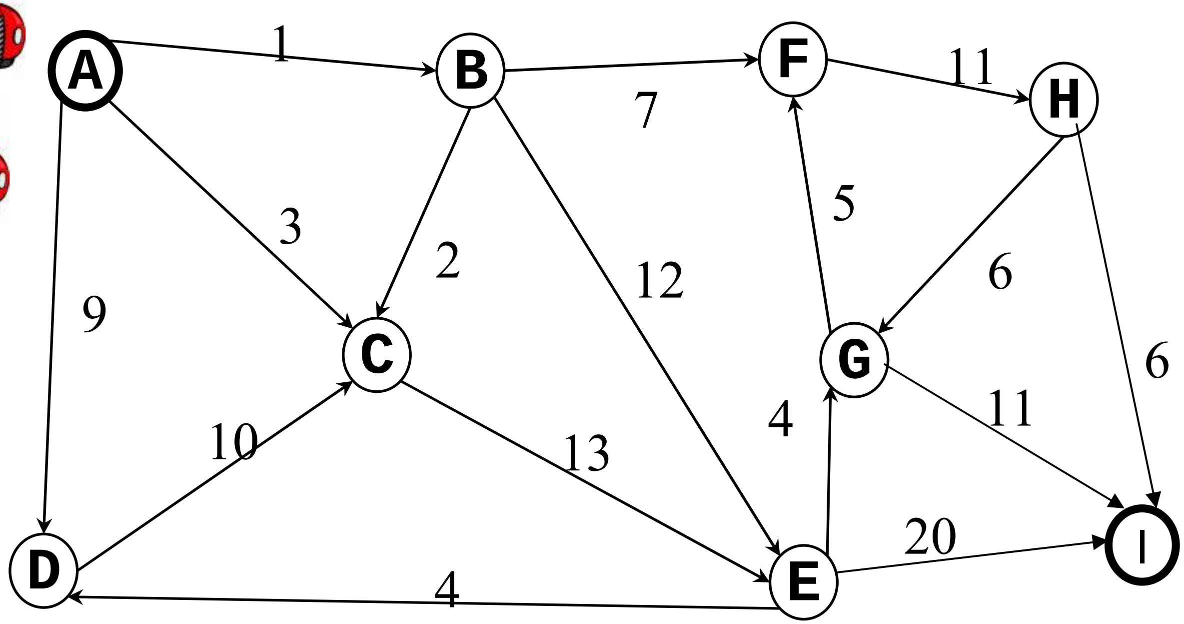
	a	b	c	d	e
a	0	2	0	-4	0
b	-	0	-2	1	-1
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0

Final Matrix Contents

Network Flows

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





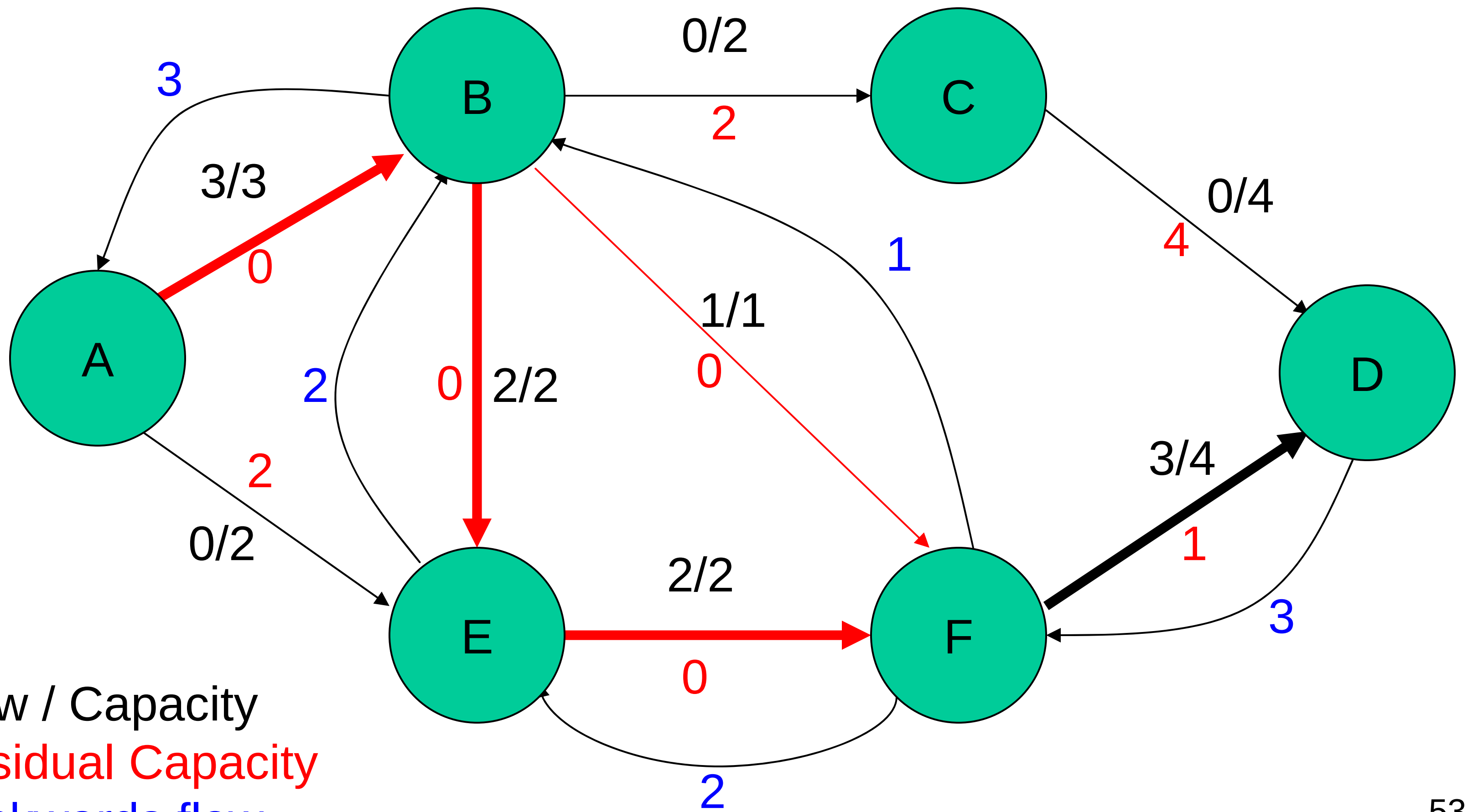
How do we know there's still room?

- Construct a residual graph:
  - Same vertices
  - Edge weights are the "leftover" capacity on the edges
  - Add extra edges for backwards-capacity too!

  - If there is a path s→t at all, then there is still room

Example (5)

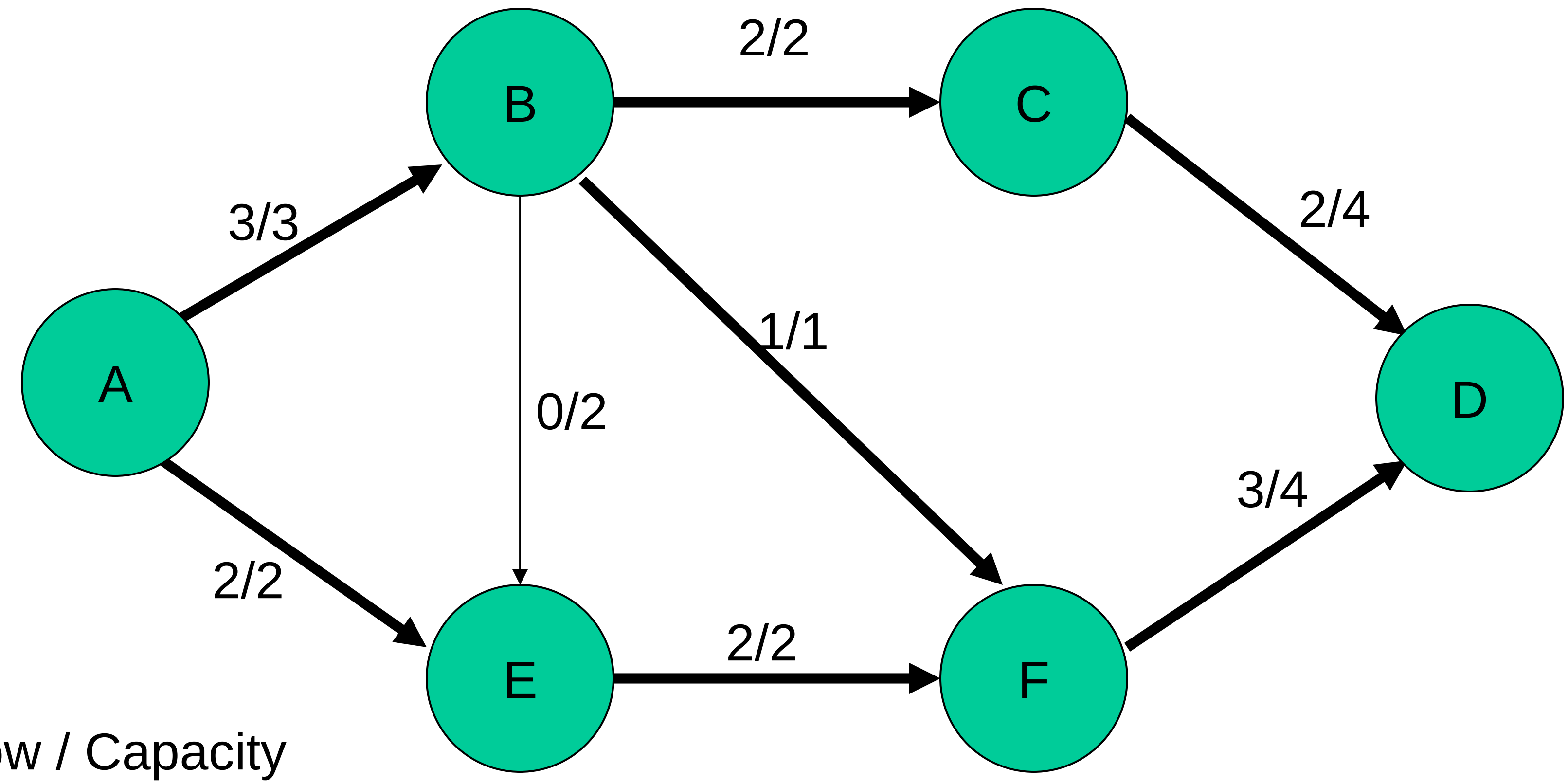
Add the backwards edges, to show we can “undo” some flow



Flow / Capacity
Residual Capacity
Backwards flow

Example (7)

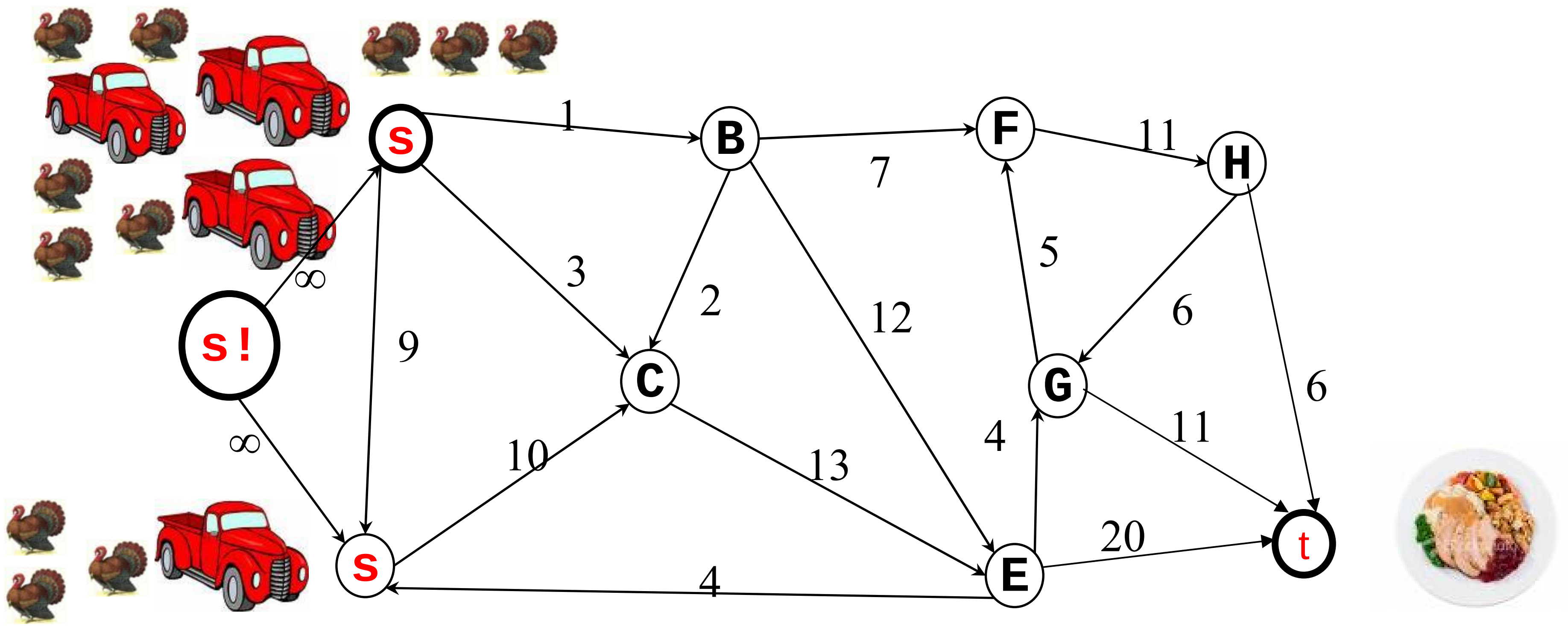
Final, maximum flow



Flow / Capacity
Residual Capacity
Backwards flow

Network Flows

- Create a single source, with infinite capacity edges connected to sources
- Same idea for multiple sinks

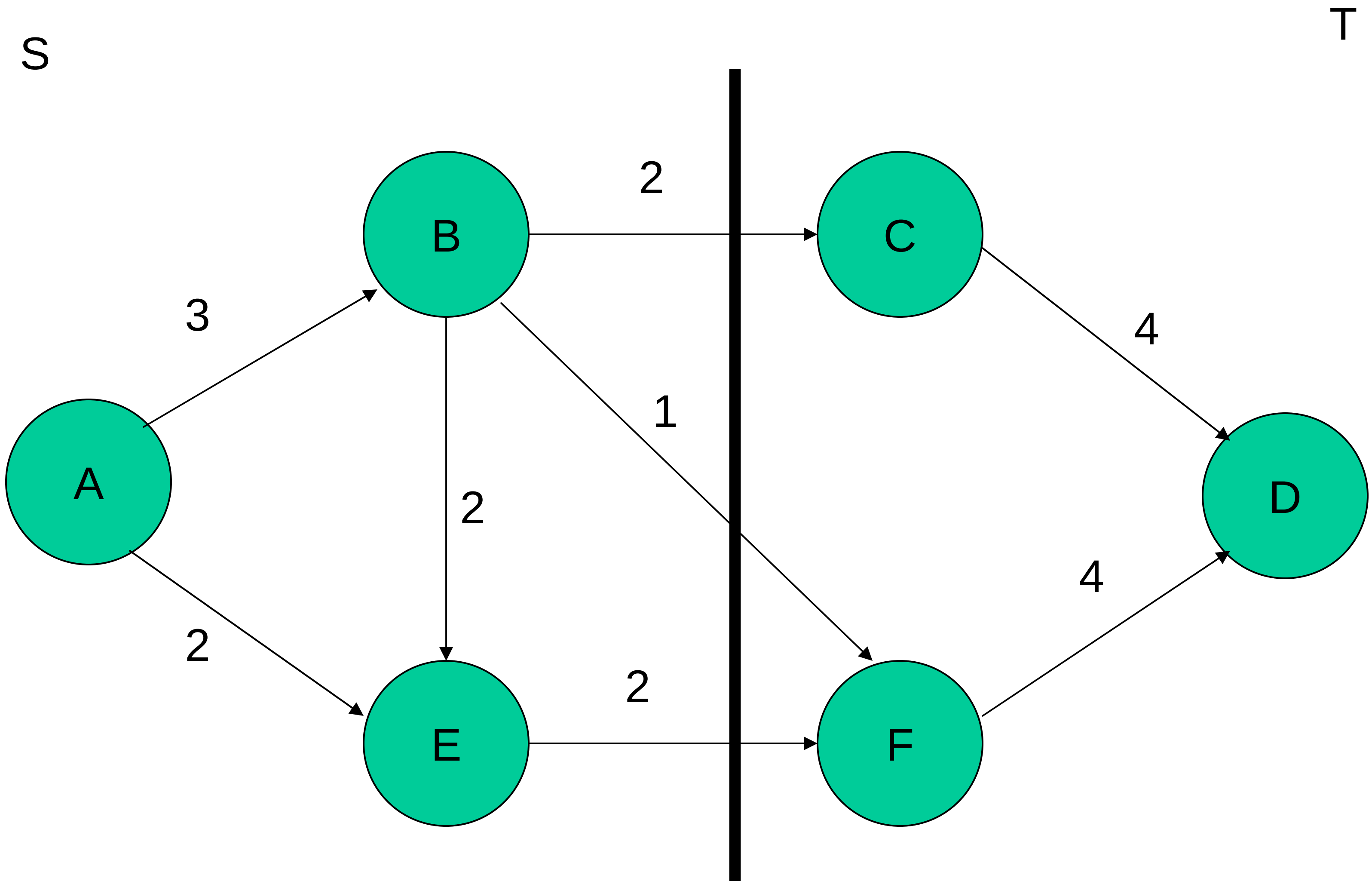


Minimum cuts

- If we cut G into (S, T), where S contains the source s and T contains the sink t,

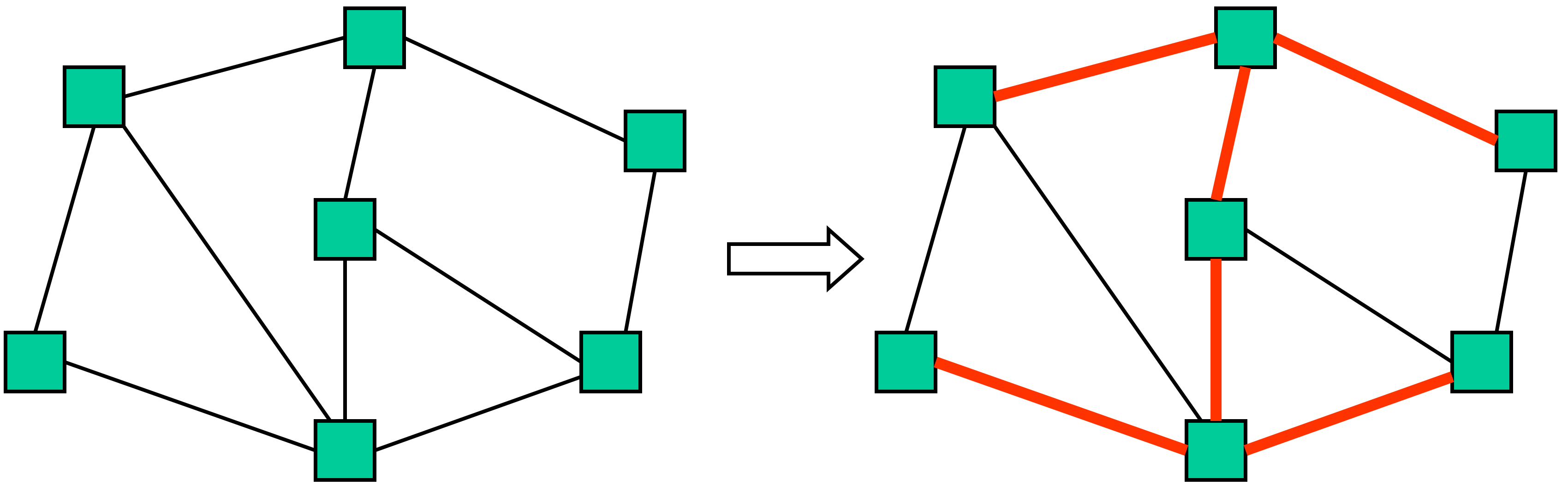
- Of all the cuts (S, T) we could find, what is the smallest (max) flow f(S, T) we will find?

Min Cut - Example (8)



Capacity of cut = 5

Spanning Tree in a Graph



Vertex = router
Edge = link between routers

Spanning tree
 - Connects all the vertices
 - No cycles

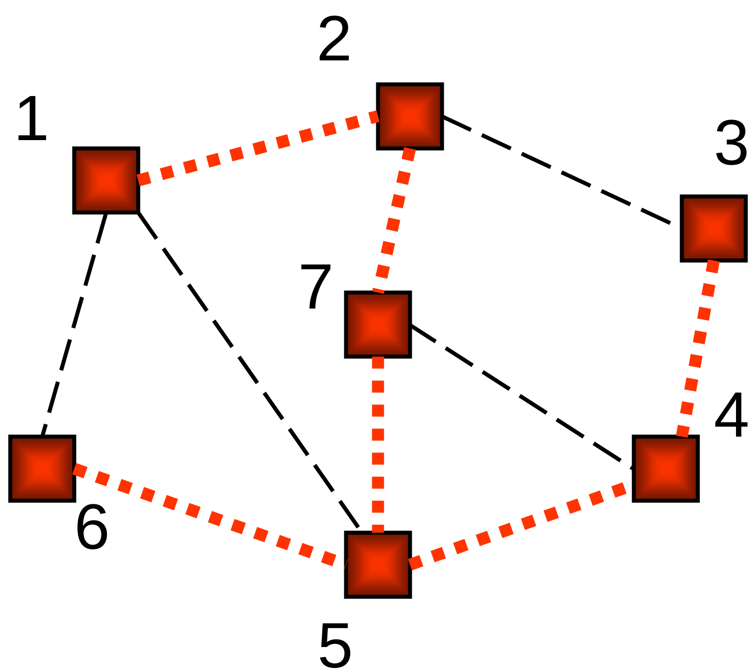
Spanning Tree Algorithm

```
ST(i: vertex)
   mark i;
   for each j adjacent to i do
      if j is unmarked then
         Add {i,j} to T;
         ST(j);
end{ST}
```

```
Main
T := empty set;
ST(1);
end{Main}
```

Example Step 16

ST(1)



{1,2} {2,7} {7,5} {5,4} {4,3} {5,6}

Minimum Spanning Trees

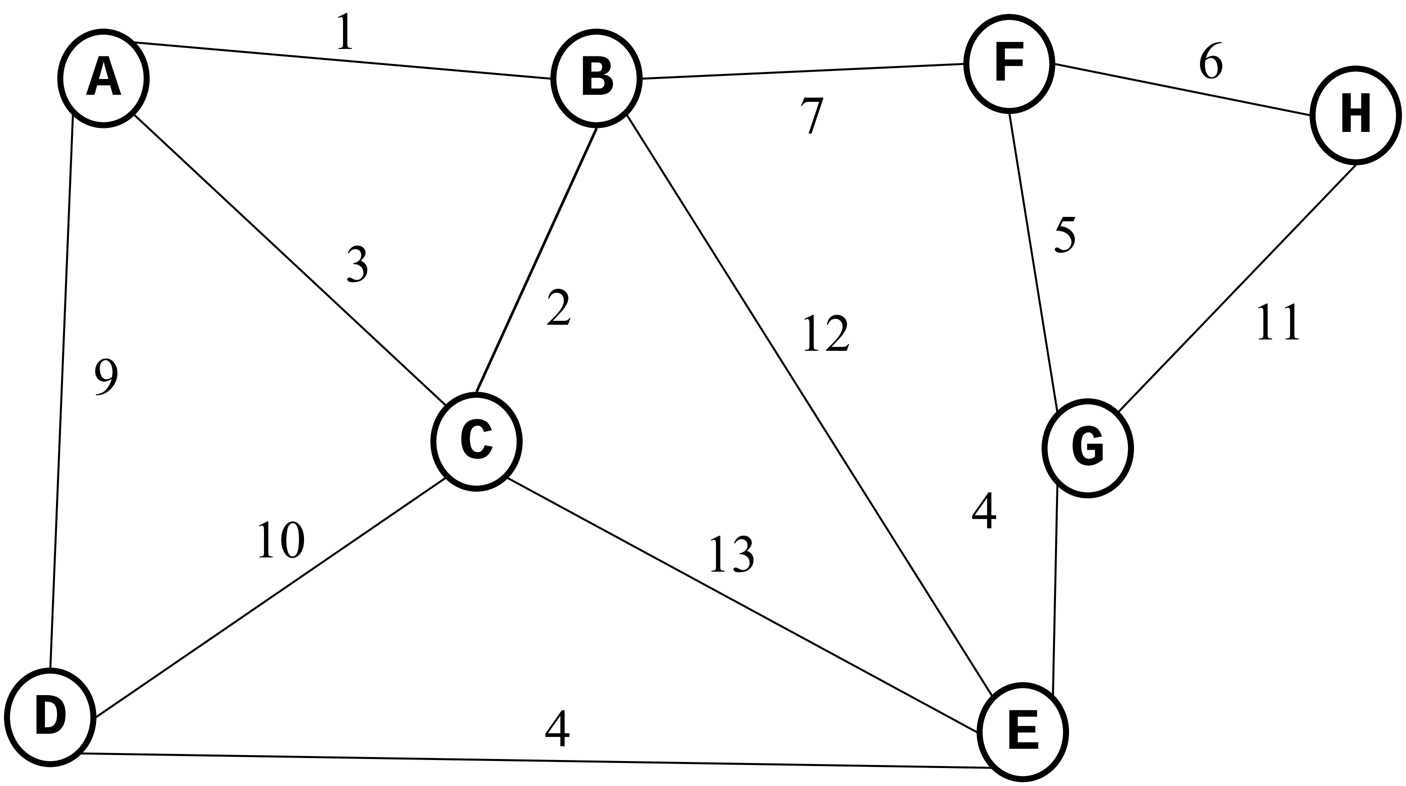
Given an undirected graph **G**=(**V**,**E**), find a graph **G'=(V, E')** such that:

- E' is a subset of E
- |E'| = |V| - 1
- G' is connected
- $\sum_{(u,v)\in E'} c_{uv}$ is minimal

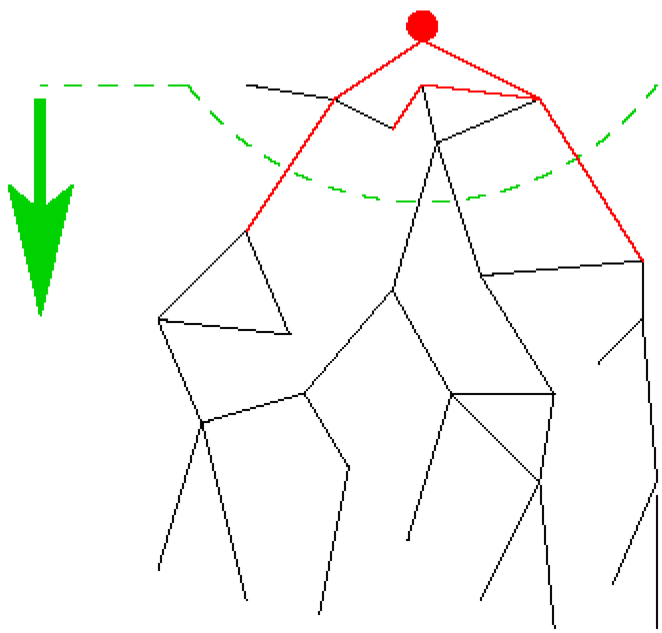
G' is a **minimum spanning tree**.

Applications: wiring a house, power grids, Internet connections

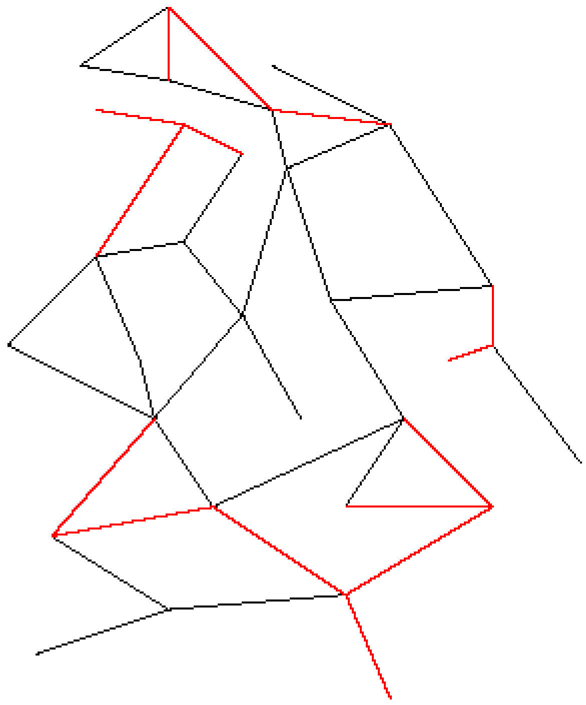
Find the MST



Two Different Approaches



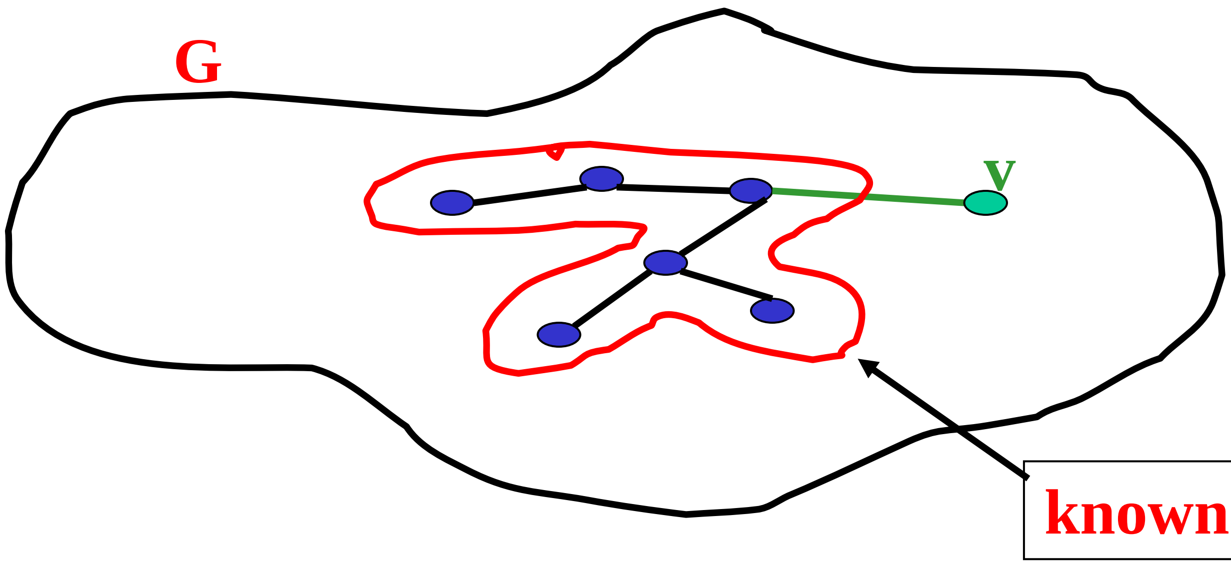
Prim's Algorithm
Looks familiar!



Kruskals's Algorithm
Completely different!

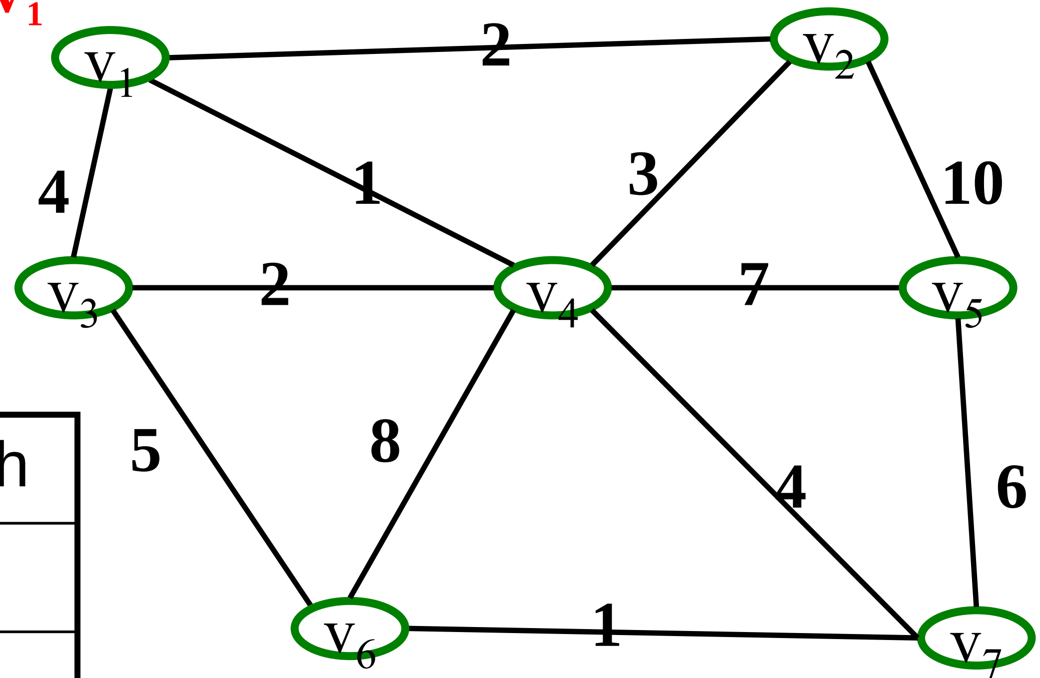
Prim's algorithm

Idea: Grow a tree by adding an edge from the "known" vertices to the "unknown" vertices. Pick the edge with the smallest weight.



Your Turn

Start with V_1



Find MST using Prim's

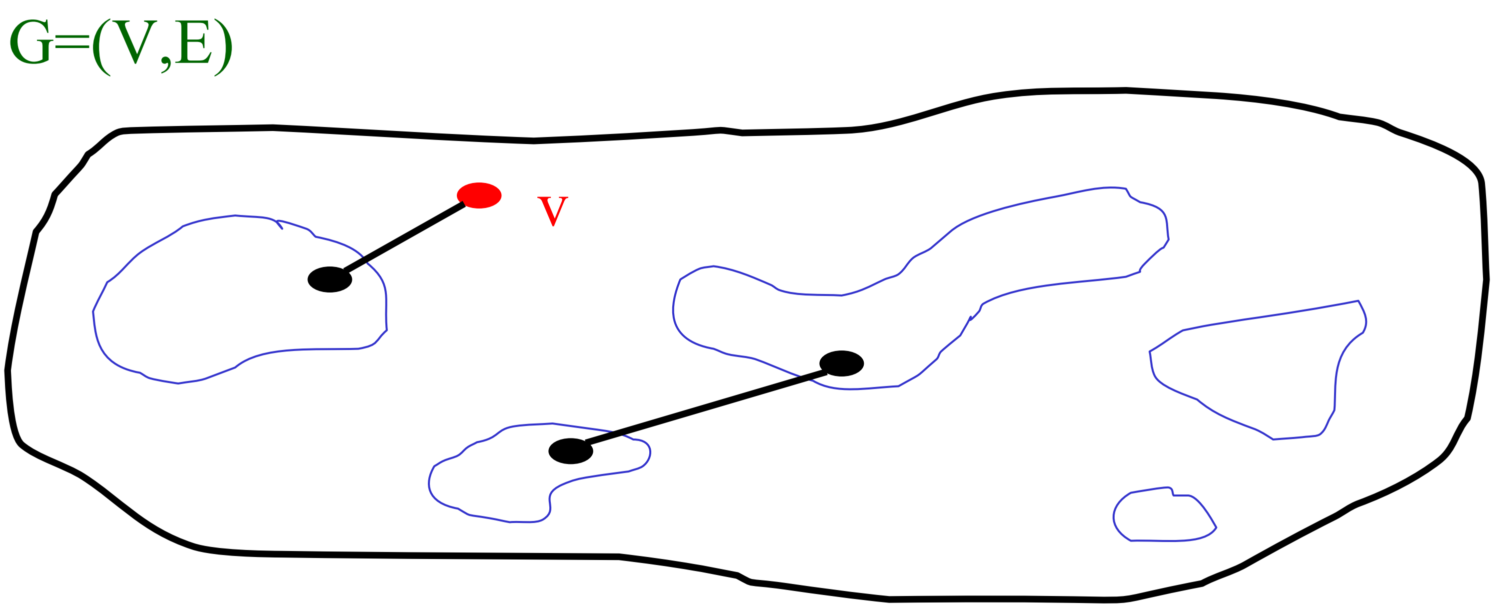
V	Kwn	Distance	path
v1			
v2			
v3			
v4			
v5			
v6			
v7			

Order Declared Known:

V_1

Kruskal's MST Algorithm

Idea: Grow a forest out of edges that do not create a cycle. Pick an edge with the smallest weight.



Kruskal's Algorithm for MST

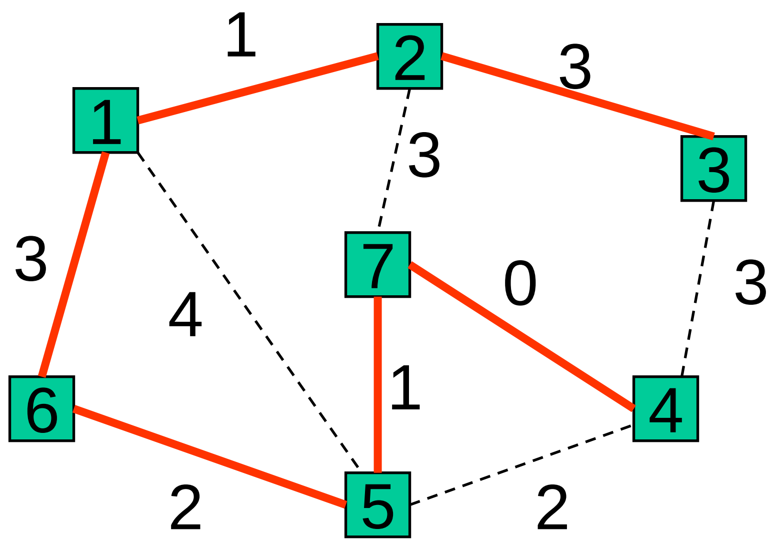
An *edge-based* greedy algorithm

Builds MST by greedily adding edges

1. Initialize with
   - empty MST
   - all vertices marked unconnected
   - all edges unmarked
2. While there are still unmarked edges
   a. Pick the lowest cost edge **`(u,v)`** and mark it
   b. If **u** and **v** are not already connected, add **`(u,v)`** to the MST and mark **`u`** and **`v`** as connected to each other

Doesn't it sound familiar?

Example of Kruskal 8,9



{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4