CSE 373
Data Structures & Algorithms

Lectures 24

Final Review

Third Midterm (a.k.a. Final)

- Friday, 12:30 – 1:30, here in class
- Logistics: Closed Book
- Comprehensive
  - Everything up to and including Network Flow
  - Not the material we will cover this Wednesday

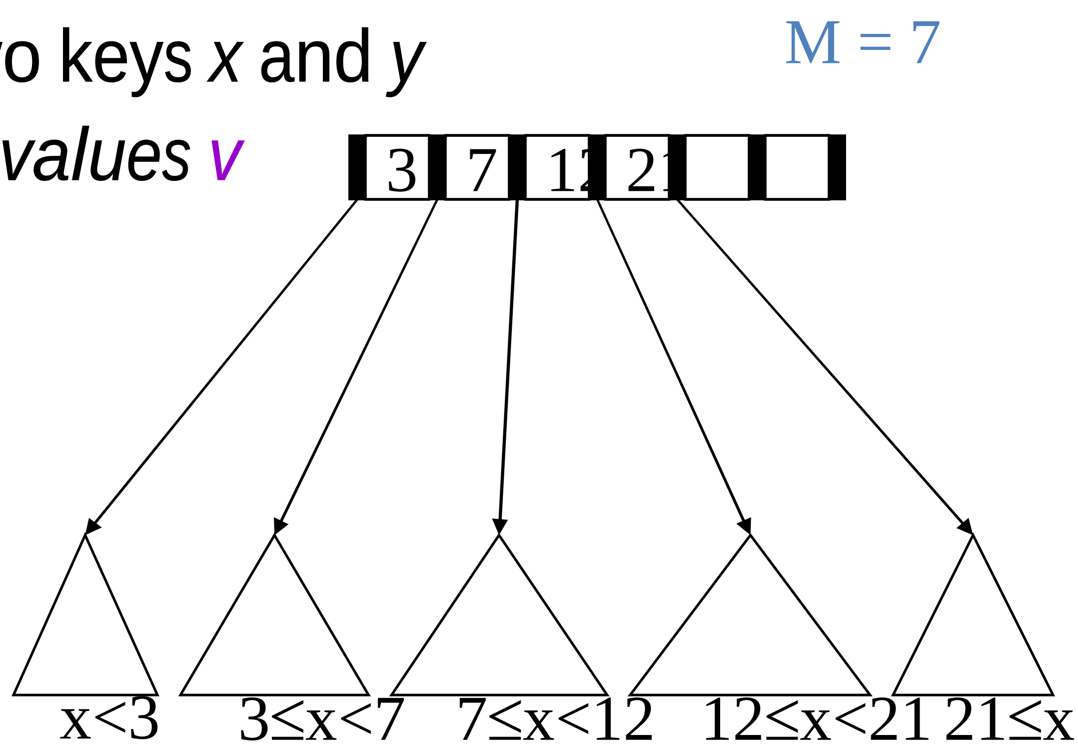
B+ Trees

(book calls these B-trees)

- Each internal still has (up to) *M*-1 keys:

- Order property:
  - subtree between two keys *x* and *y* contain leaves with *values* *v* such that $x \le v < y$
  - Note the "≤"

- Leaf nodes contain up to *L* sorted keys.



B+ Tree Structure Properties

Root (special case)

– has between 2 and ***M*** children (or root could be a leaf)

Internal nodes

Nodes are at least ½ full

– store up to M-1 keys

– have between $\lceil M/2 \rceil$ and M children

Leaf nodes

– where data is stored

– all at the same depth

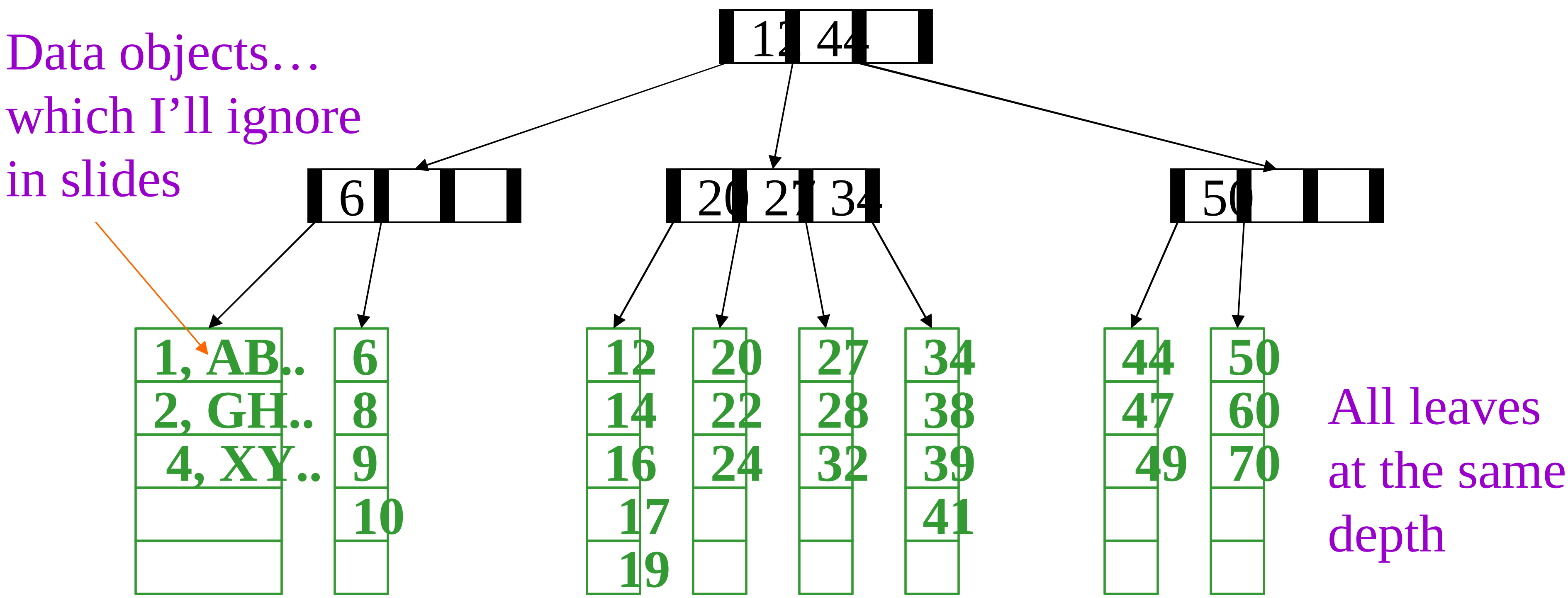
Leaves are at least ½ full

– contain between $\lceil L/2 \rceil$ and L data items

B+ Tree: Example

B+ Tree with ***M* = 4** (# pointers in internal node)

and ***L* = 5** (# data items in leaf)



Definition for later: "neighbor" is the next sibling to the left or right.

B+ trees vs. AVL trees

Suppose again we have $n = 2^{30} \approx 10^9$ items:

- Depth of AVL Tree

43

- Depth of B+ Tree with M = 256, L = 256

Log_128 10^9 = 4.3

So let's see how we do this…

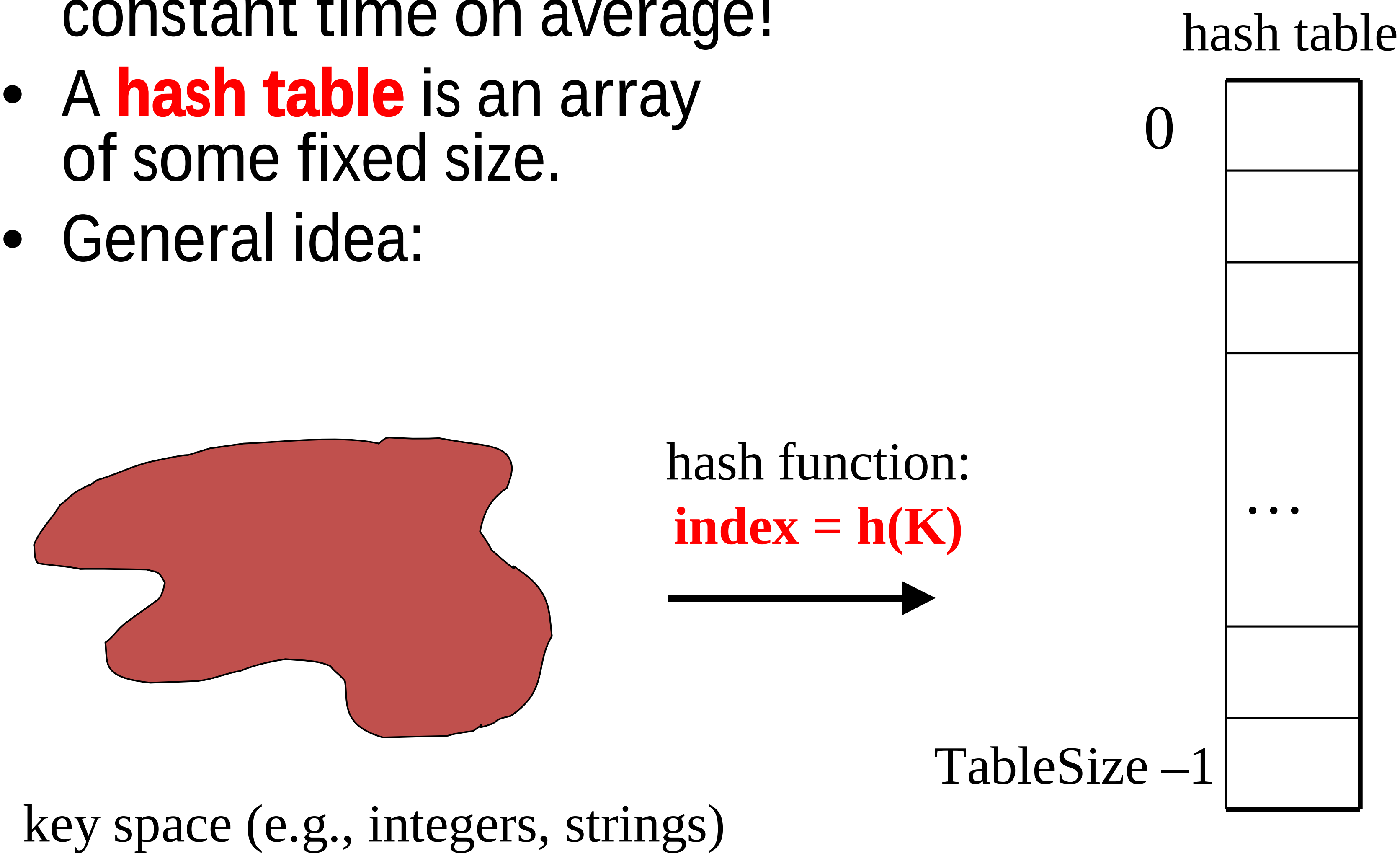
Thinking about B+ Trees

- B+ Tree insertion can cause (expensive) splitting and propagation up the tree
- B+ Tree deletion can cause (cheap) adoption or (expensive) merging and propagation up the tree
- Split/merge/propagation is rare if ***M*** and ***L*** are large *(Why?)*
- Pick branching factor ***M*** and data items/leaf ***L*** such that each node takes one full page/block of memory/disk.

Only 1/L inserts cause split, only 1/M of these go up!

Hash Tables

- Find, insert, delete: constant time on average!
- A **hash table** is an array of some fixed size.
- General idea:



Separate Chaining

0	10
1	
2	42, 12, 22
3	
4	
5	
6	
7	107
8	
9	

Insert:
10
22
107
12
42

Thoughts about this?

Our goal is to keep it such that a simple list is good enough

Separate chaining: All keys that map to the same hash value are kept in a list (or "bucket").

Open Addressing

0	8
1	109
2	10
3	
4	
5	
6	
7	
8	38
9	19

Insert:
38
19
8
109
10

Try h(K)

If full, try h(K)+1.

If full, try h(K)+2.

If full, try h(K)+3.

Etc…

What is f(i)?

Linear Probing

$$f(i) = i$$

- Probe sequence:

  0^{th} probe = h(K) % TableSize

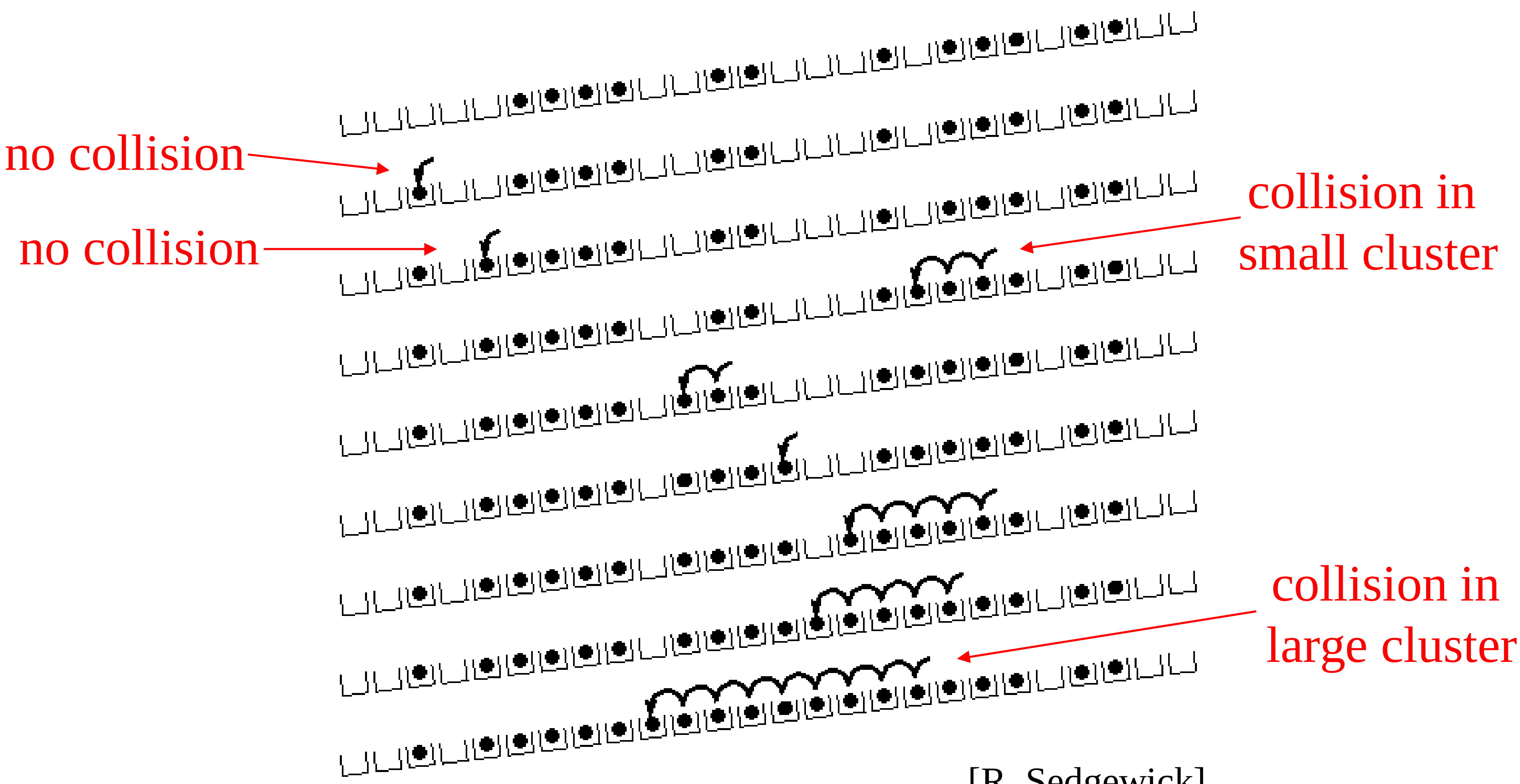
  1^{th} probe = (h(K) + 1) % TableSize

  2^{th} probe = (h(K) + 2) % TableSize

  . . .

  i^{th} probe = (h(K) + i) % TableSize

Linear Probing – Clustering



[R. Sedgewick]

Quadratic Probing

Less likely to encounter Primary Clustering

$f(i) = i^2$

- Probe sequence:

  0^{th} probe = h(K) % TableSize

  1^{th} probe = (h(K) + 1) % TableSize

  2^{th} probe = (h(K) + 4) % TableSize

  3^{th} probe = (h(K) + 9) % TableSize

  . . .

  i^{th} probe = (h(K) + i^2) % TableSize

Double Hashing

Idea: given two different (good) hash functions h(K) and g(K), it is unlikely two keys to collide with both.

So…let's try probing with a second hash function:

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

- Probe sequence:
  - 0^{th} probe = h(K) % TableSize
  - 1^{th} probe = (h(K) + g(K)) % TableSize
  - 2^{th} probe = (h(K) + 2*g(K)) % TableSize
  - 3^{th} probe = (h(K) + 3*g(K)) % TableSize
  - . . .
  - i^{th} probe = (h(K) + i*g(K)) % TableSize

Deletion in Separate Chaining

How do we delete an element with separate chaining?

Easy, just delete the item from the bucket

Deletion in Open Addressing

Can we do something similar for open addressing?

- Delete
- Find
- Insert

0	
1	
2	16
3	X
4	59
5	
6	76

$h(k) = k \% 7$
Linear probing

Delete(23)
Find(59)
Insert(30)

Need to leave a marker of a deletion

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?
  - Separate chaining: full ($\lambda = 1$)
  - Open addressing: half full ($\lambda = 0.5$)
  - When an insertion fails
  - Some other threshold
- Cost of a single rehashing? O(N)

Why Sort?

- Allows binary search of an N-element array in O(log N) time
- Allows O(1) time access to *k*th largest element in the array for any *k*
- People tend to like their output sorted

- Sorting algorithms are a frequently used and heavily studied family of algorithms in computer science

Stability

A sorting algorithm is **stable** if:

– Items in the input with the same value end up in the same order as when they began.

Input		**Stable Sort**		**Unstable sort**	
Adams	1	Adams	1	Adams	1
Black	2	Smith	1	Smith	1
Brown	4	Black	2	Washington	2
Jackson	2	Jackson	2	Jackson	2
Jones	4	Washington	2	Black	2
Smith	1	White	3	White	3
Thompson	4	Wilson	3	Wilson	3
Washington	2	Brown	4	Thompson	4
White	3	Jones	4	Brown	4
Wilson	3	Thompson	4	Jones	4

[Sedgewick]

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 …	Heap sort Binary tree sort Merge sort Quick sort (avg case) …		Bucket sort Radix sort	External sorting

Selection Sort: Idea

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

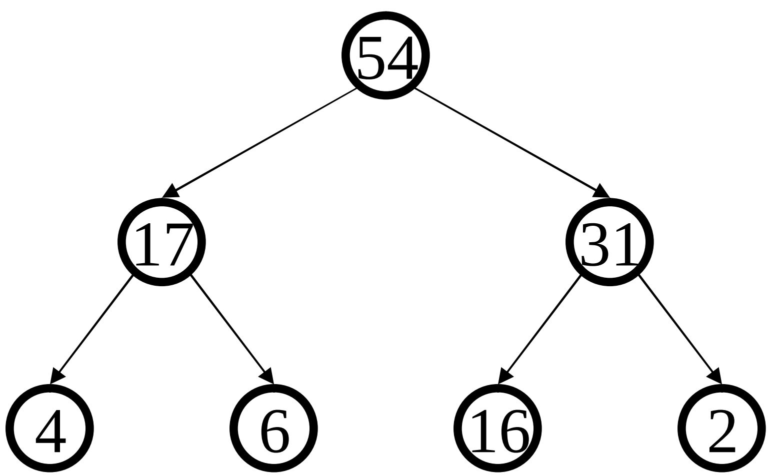
Bubble Sort Idea

- Take a pass through the array
  – If a pair of neighboring elements are out of sort order, swap them.
- Take passes until no swaps are needed at any point in the pass.

Insertion Sort: Idea

1. One element is by definition sorted
2. Sort first 2 elements.
3. Insert 3rd element in order.
   - (First 3 elements are now sorted.)
4. Insert 4th element in order
   - (First 4 elements are now sorted.)
5. And so on…

Heap Sort: Sort with a Binary Heap



2, 4, 6, 16, 17, 31, 54

Use a max-heap, do it in-place

Runtime: O(n log n)

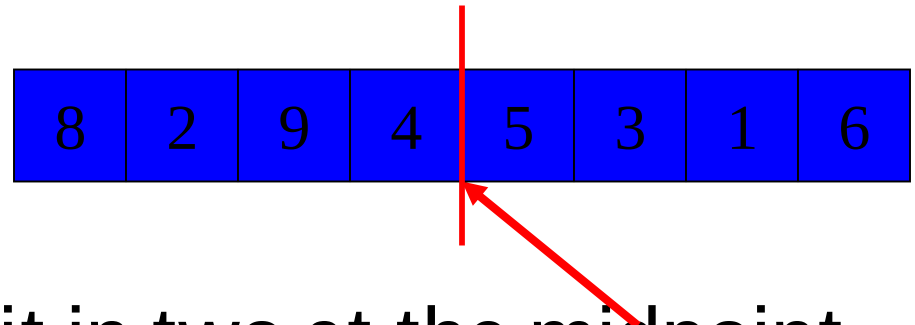
"Divide and Conquer"

- Two divide and conquer sorting methods:

- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → known as Mergesort

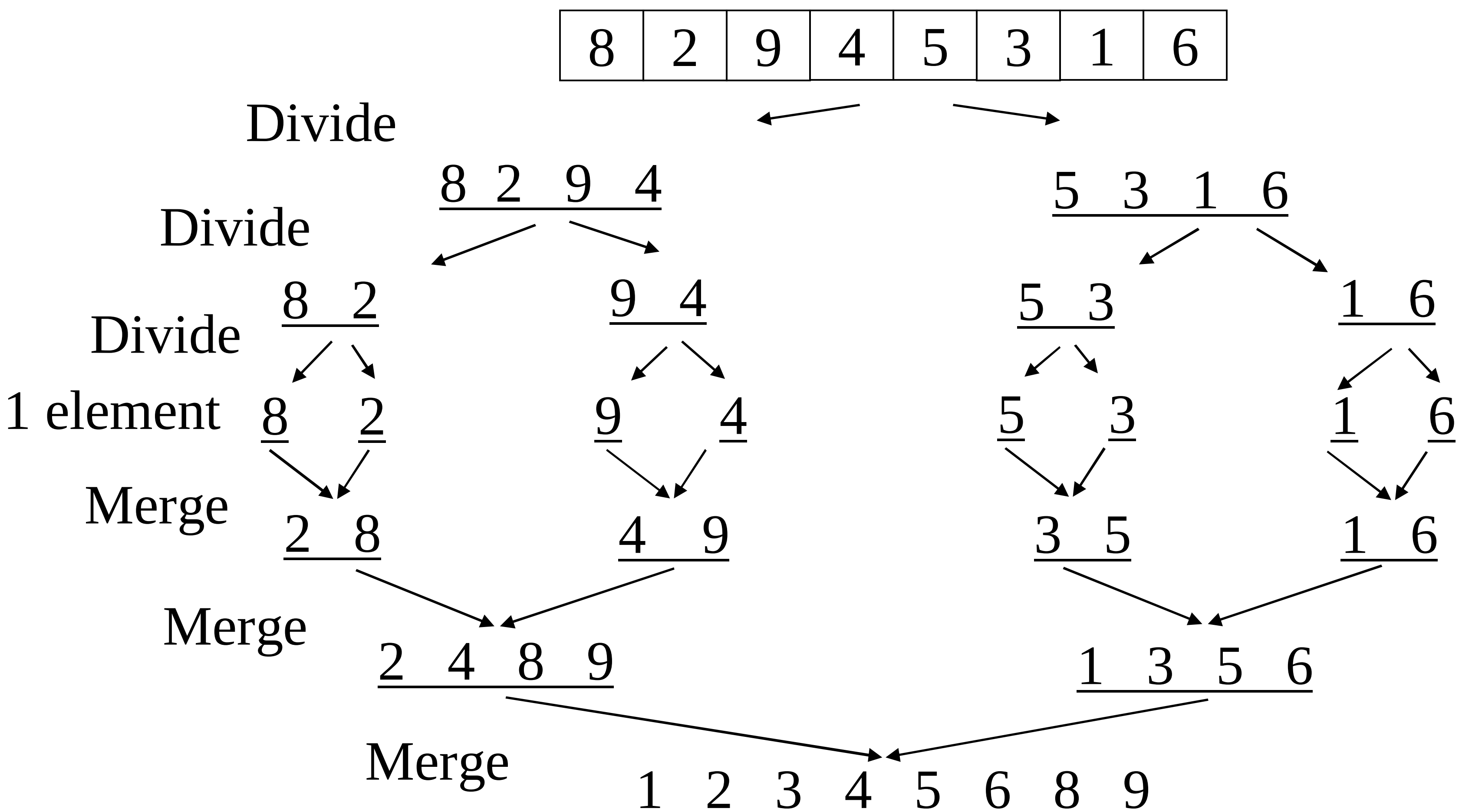
- **Idea 2 :** Partition array into small items and large items, then recursively sort the two smaller portions → known as Quicksort

Mergesort

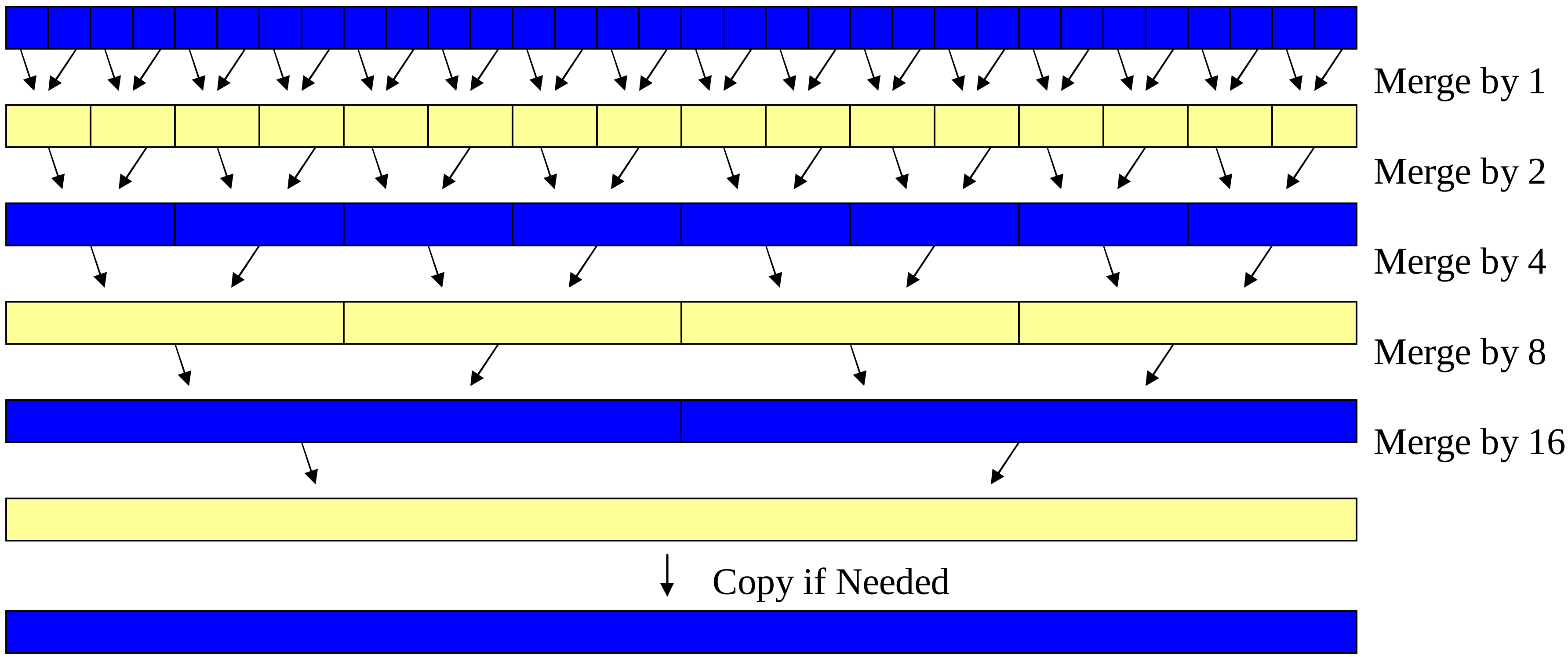


- Divide it in two at the midpoint
- Conquer each side in turn
  (by recursively sorting)
- Merge two halves together

Mergesort Example



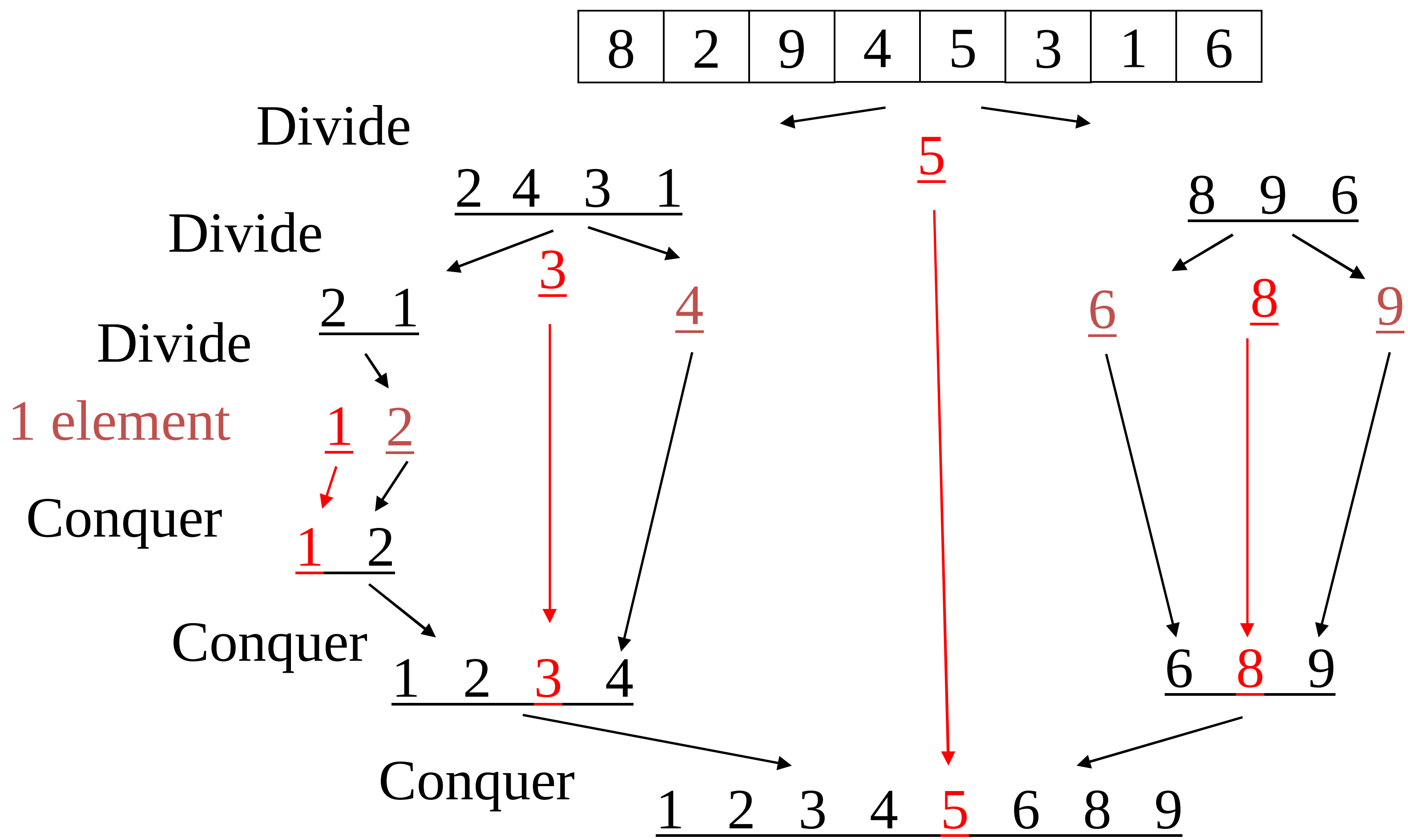
Iterative Mergesort



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

Quicksort Example



So Which Is Best?

- It's a trick question, a naïve question
- Myth: "Quicksort is the best in-memory sorting algorithm."
- Mergesort and Quicksort make different tradeoffs regarding the cost of comparison and the cost of a swap
- Mergesort is also the basis for external sorting algorithms (large N sorting)

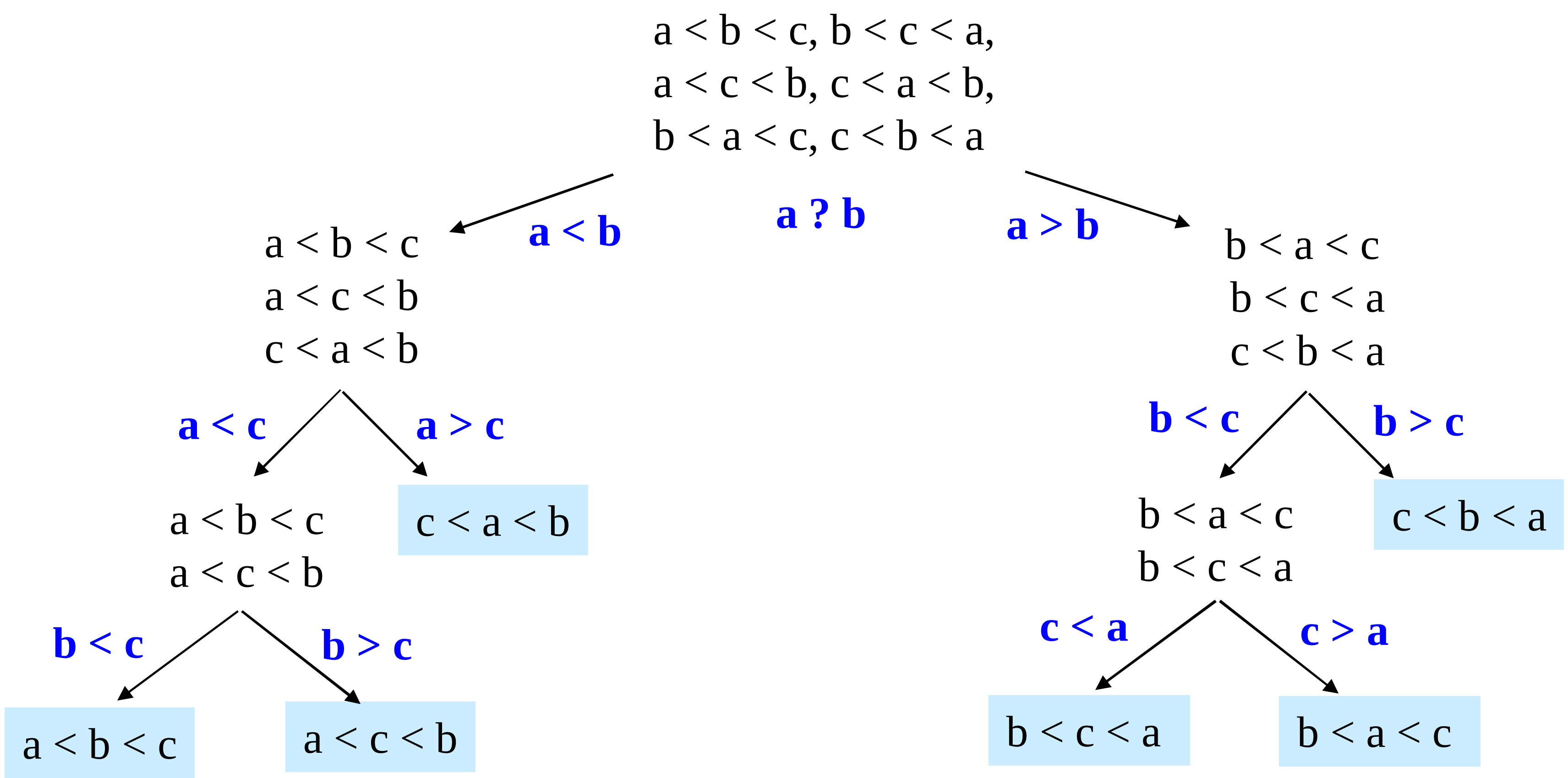
Permutations

- How many possible orderings are there?

- Example: a, b, c

$a < b < c$	$b < a < c$	$c < a < b$
$a < c < b$	$b < c < a$	$c < b < a$

Decision Tree



The leaves contain all the possible orderings of a, b, c

Lower bound on Height

- The decision tree has how many leaves:

$$L = N!$$

- A binary tree with L leaves has height at least:

$$h \geq \log_2 L$$

- So the decision tree has height:

$$h \geq \log_2 (N!)$$

log(N!)

select just the first N/2 terms

each of the selected terms is ≥ logN/2

$$
\begin{aligned}
\log(N!) &= \log\big(N\cdot(N-1)\cdot(N-2)\cdots(2)\cdot(1)\big)\\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1\\
&\ge \log N + \log(N-1) + \log(N-2) + \cdots + \log\frac{N}{2}\\
&\ge \frac{N}{2}\log\frac{N}{2}\\
&\ge \frac{N}{2}(\log N - \log 2) = \frac{N}{2}\log N - \frac{N}{2}\log 2\\
&= \Omega(N\log N)
\end{aligned}
$$

Ω(N log N)

- No matter how clever you are about which comparisons you perform, your sorting algorithm with always be Ω(N log N)
- Your worst case will be at least N log N
- Proving this saves us the trouble of trying to do better than this, because we cannot
- Now that's some Computer Science

Doing Better

- So how can we do better?
  - Need to dodge one of the proof's assumptions
- What's our proof based in?
  - Comparisons
- Can we sort without doing comparisons?

BucketSort (aka BinSort)

If all values 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	3
2	1
3	2
4	2
5	3

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

Running time to sort n items?

N + K

RadixSort

- Radix = “The base of a number system”
  - We’ll use 10 for convenience
  - Use a larger number in any implementation
  - ASCII Strings, for example, might use 128
- Idea:
  - BucketSort on one digit at a time
    - Requires stable sort!
  - After sort k, the last k digits are sorted
  - Set number of buckets: B = radix.

RadixSort

- Input:126, 328, 636, 341, 416, 131, 328

BucketSort on lsd:

	341 131					126 636 416		328 328	
0	1	2	3	4	5	6	7	8	9

BucketSort on next-higher digit:

	416	126 328 328	131 636	341					
0	1	2	3	4	5	6	7	8	9

BucketSort on msd:

	126 131		328 328 341	416		636			
0	1	2	3	4	5	6	7	8	9

Output: 126, 131, 328, 328, 341, 416, 636

Summary of sorting

$O(N^2)$ average, worst case:

– **Selection Sort**, **Bubblesort**, **Insertion Sort**

$O(N \log N)$ average case:

– **Heapsort**: In-place, not stable.

– **Mergesort**: $O(N)$ extra space, stable, massive data.

– **Quicksort**: claimed fastest in practice, but $O(N^2)$ worst case. Recursion/stack requirement. Not stable.

$\Omega(N \log N)$ worst and average case:

– **Any comparison-based sorting algorithm**

$O(N)$

– **Radix Sort**: fast and stable. Not comparison based. Not in-place. Poor memory locality can undercut performance.

Disjoint Set ADT

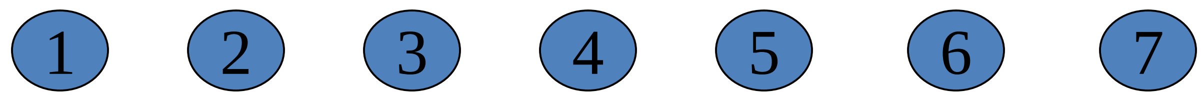
- Data: set of pairwise **disjoint sets**.
- Required operations
  - **Union** – merge two sets to create their union
  - **Find** – determine which set an item appears in

- A common operation sequence:
  - Connect two elements if not already connected:

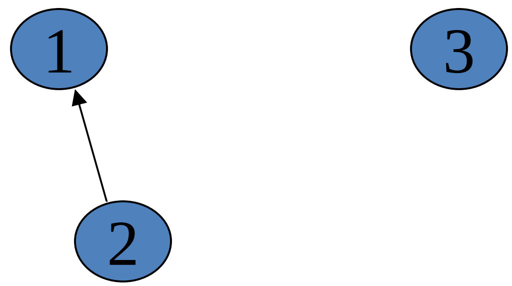
    if (Find(x) != Find(y)) then Union(x,y)

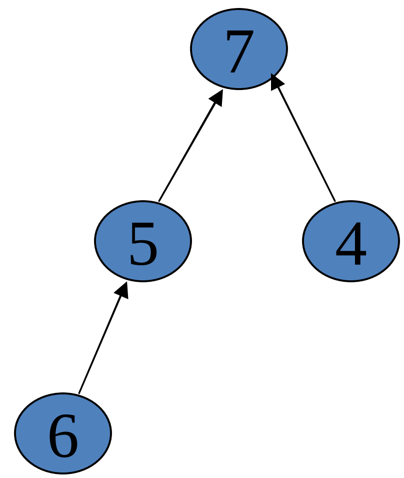
Up-Tree for DU/F

Initial state



Intermediate state

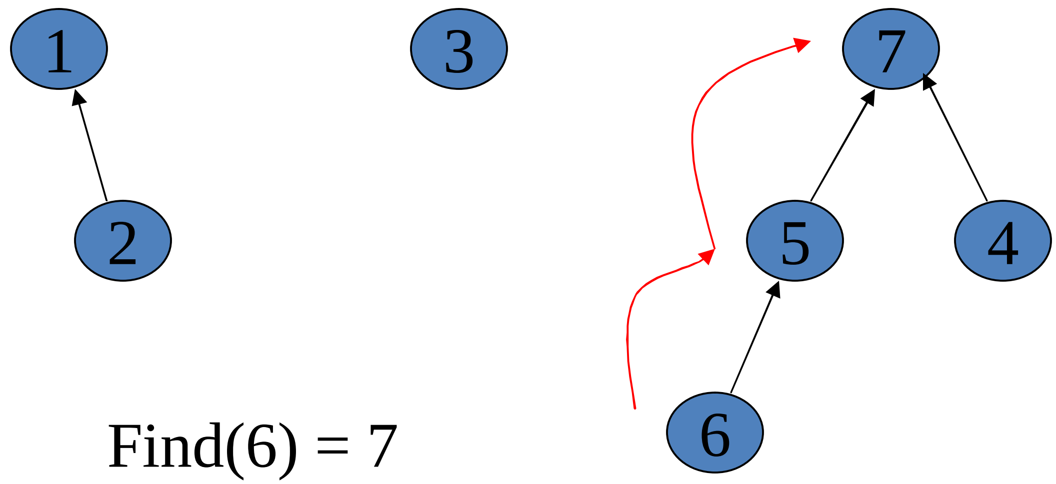




Roots are the names of each set.

Find Operation

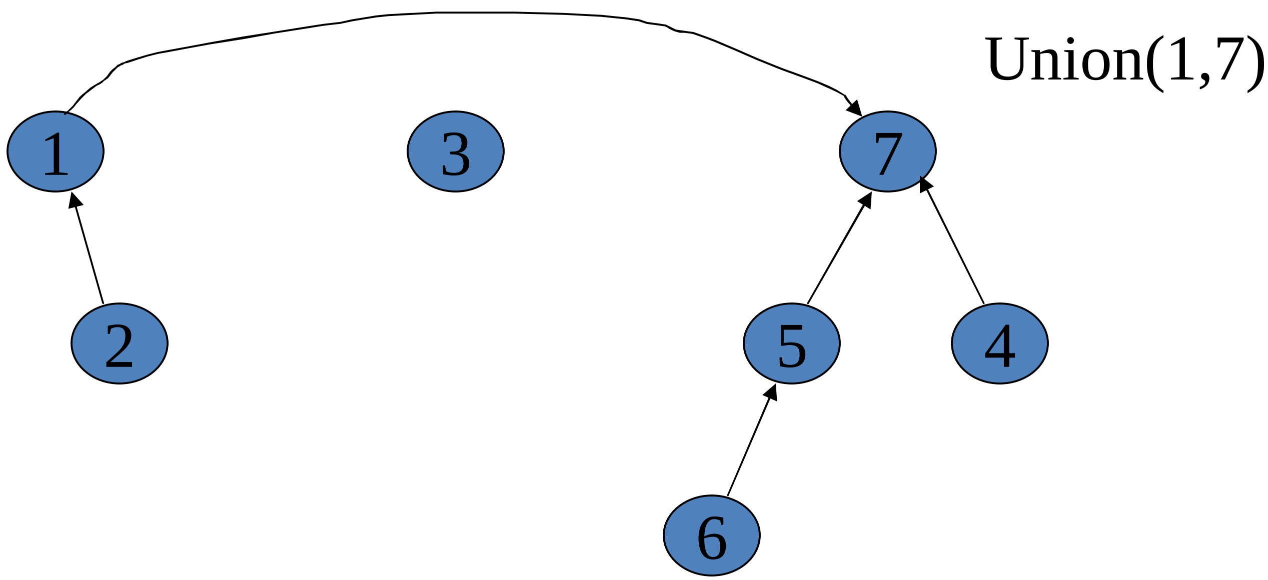
- Find(x):
  follow x to the root and return the root



Find(6) = 7

Union Operation

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

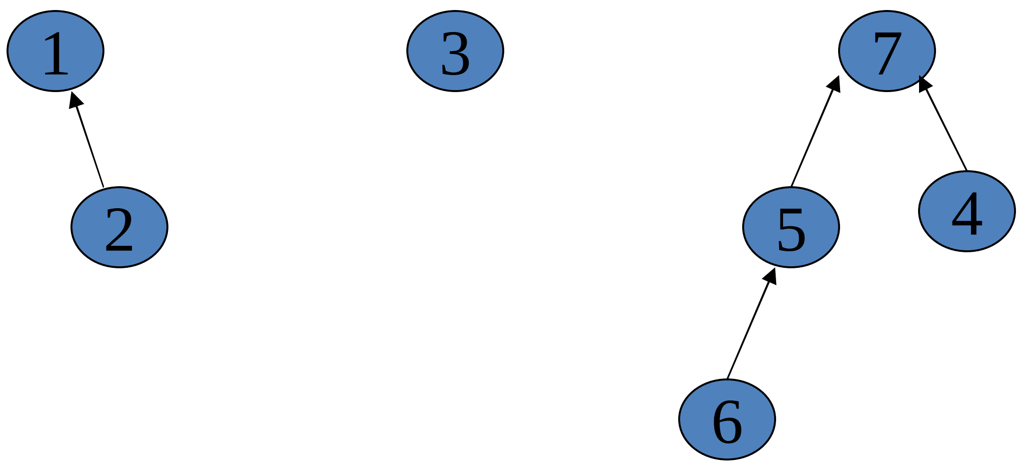


Simple Implementation

- Array of indices

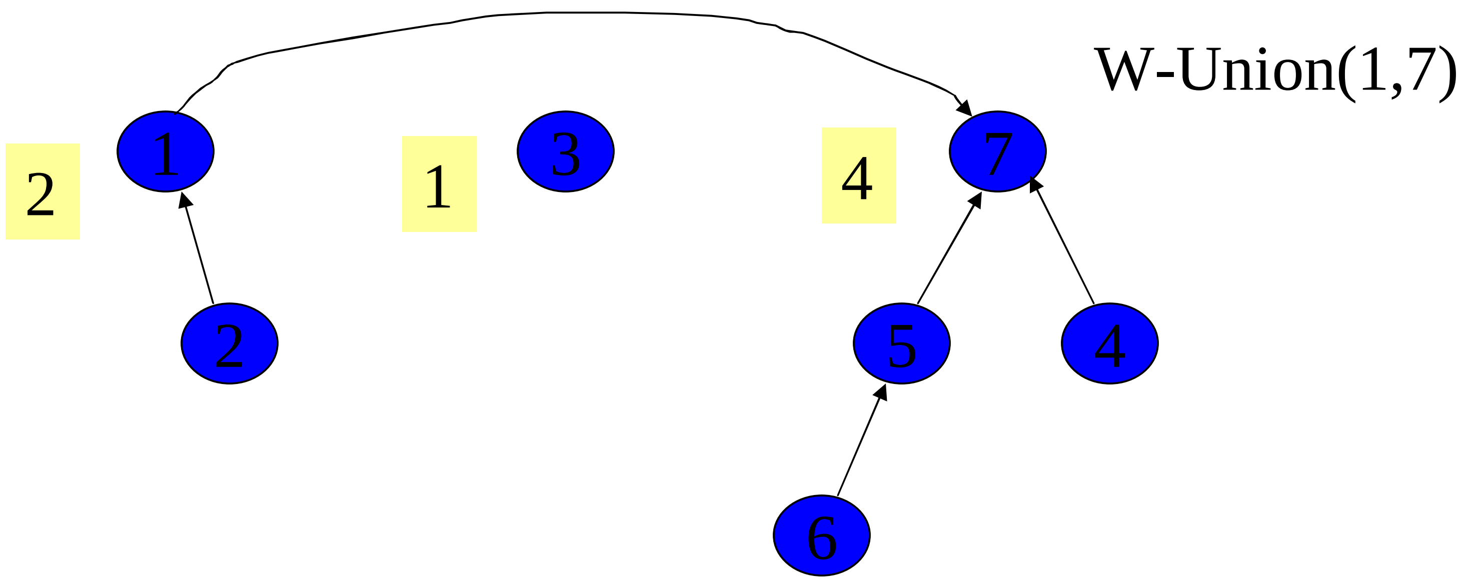
	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0

Up[x] = 0 means
x is a root.

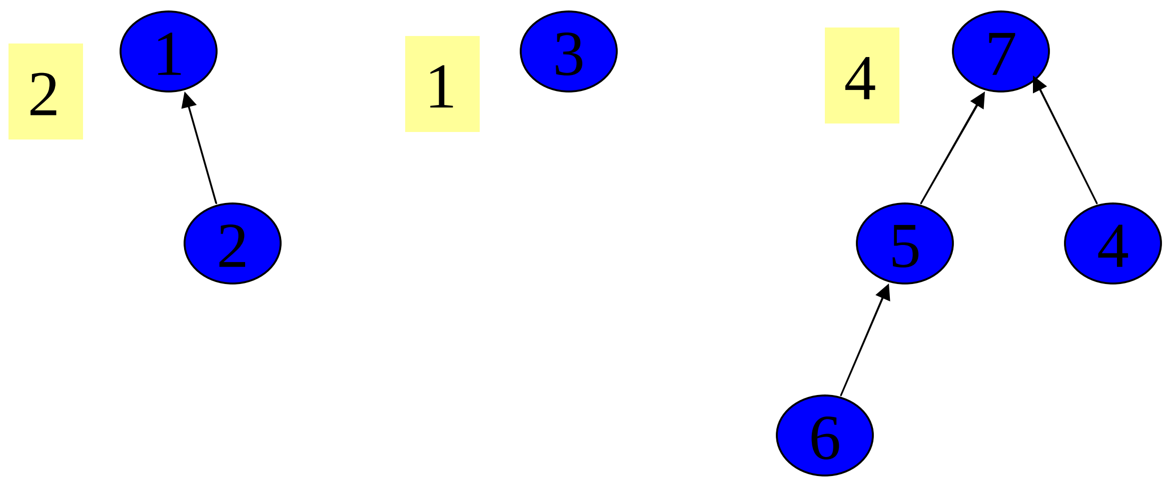


Weighted Union

- Weighted Union
  – Instead of arbitrarily joining two roots, always point the smaller root to the larger root

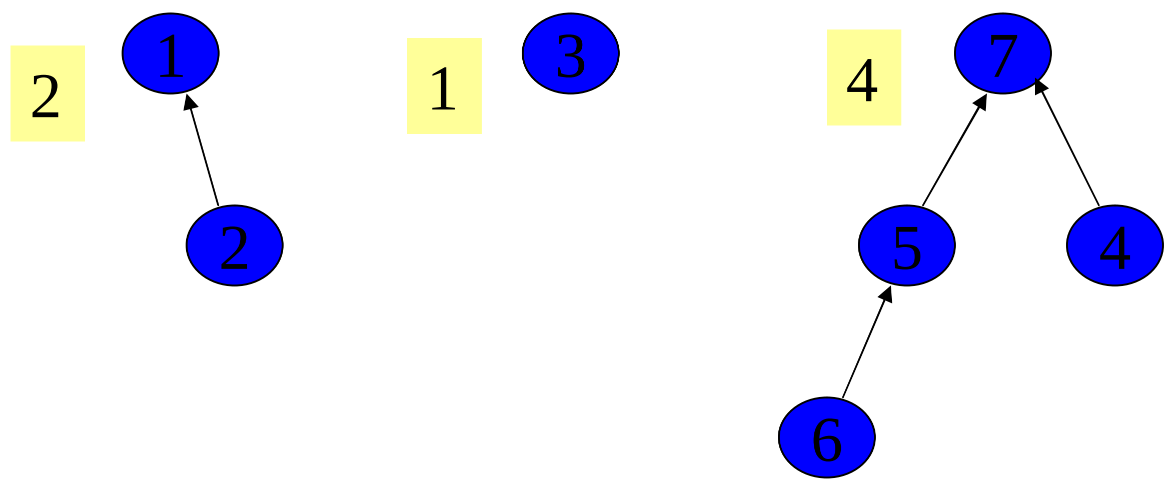


Elegant Array Implementation



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

Elegant Array Implementation

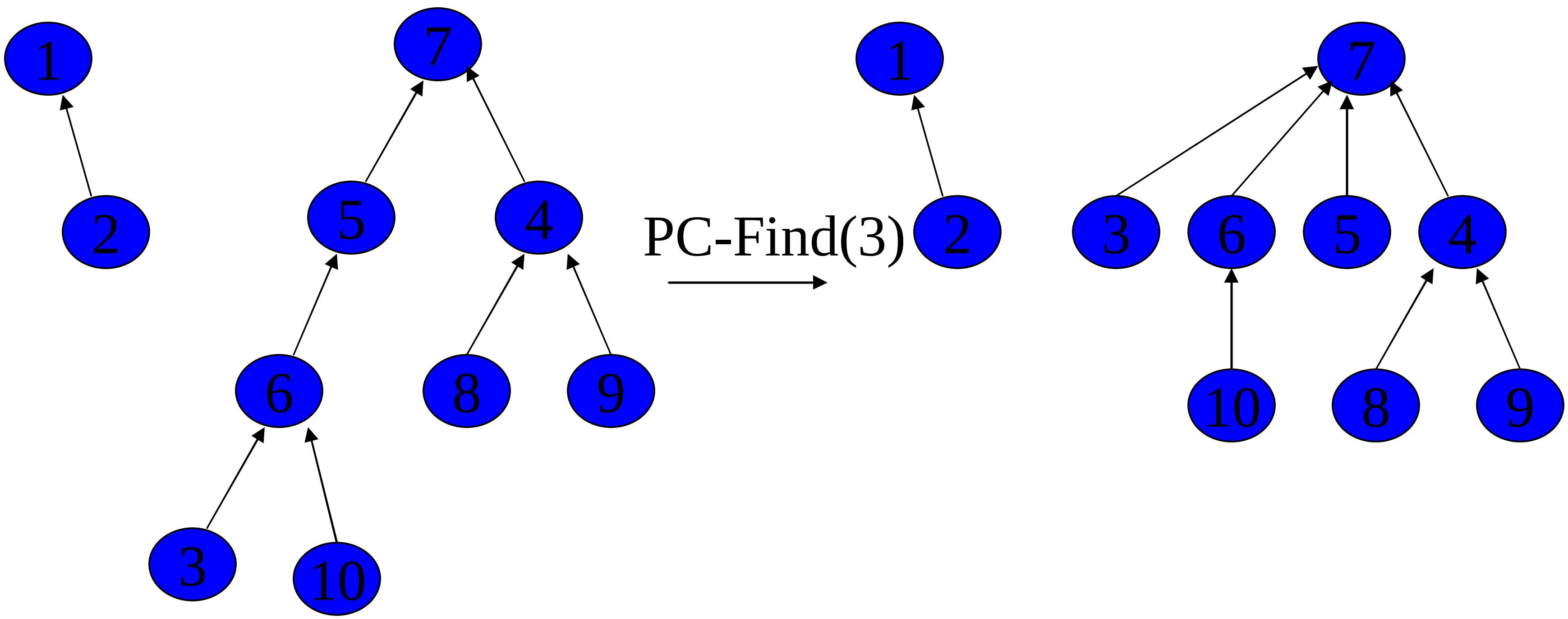


	1	2	3	4	5	6	7
up -2		1	-1	7	7	5	-4

Instead of a separate weight array,
can re-use the empty parent reference

Path Compression

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

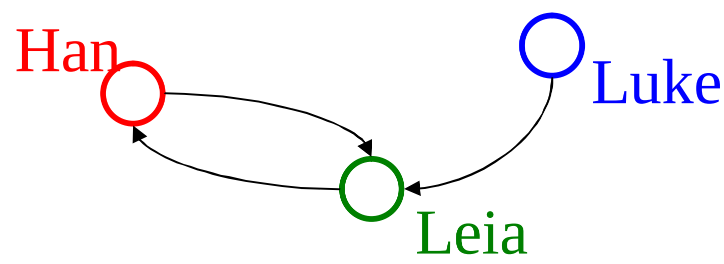


Graphs

Formalism representing relationships among objects

Graph $\mathbf{G = (V,E)}$

- *Set of vertices* (aka nodes): $\mathbf{V = \{v_1, v_2, \ldots, v_n\}}$



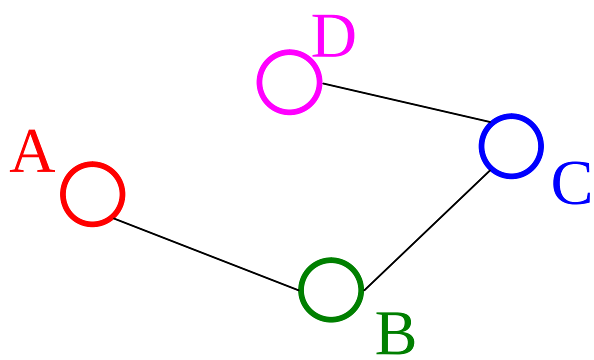
- *Set of edges*: $\mathbf{E = \{e_1, e_2, \ldots, e_m\}}$ where each $\mathbf{e_i}$ connects one vertex to another $\mathbf{(v_j, v_k)}$

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

Graphs can be *directed* or *undirected*

Undirected Graphs

In *undirected* graphs, edges have no specific direction (edges are always two-way):

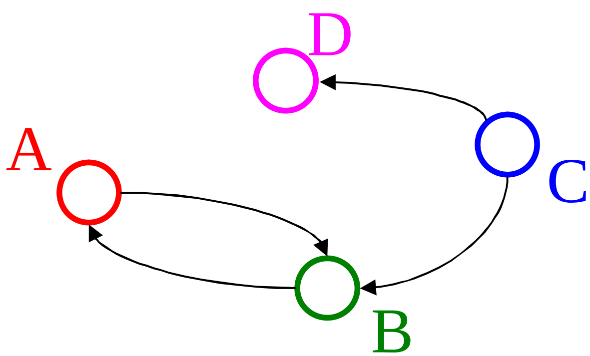


Thus, **(u,v)**$\in$**E** implies **(v,u)**$\in$**E**. Only one of these edges needs to be in the set; the other is implicit.

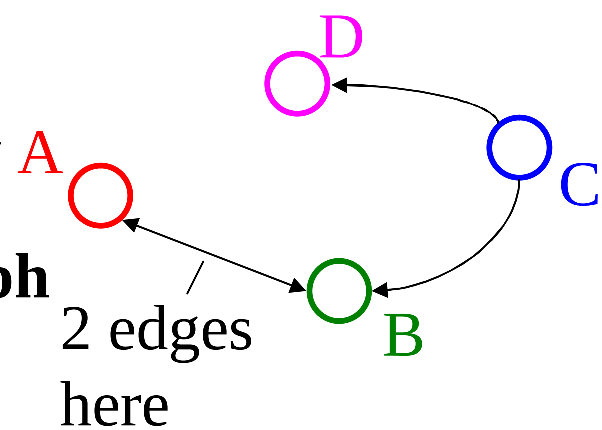
Degree of a vertex: number of edges containing that vertex. (Same as number of adjacent vertices.)

Directed Graphs

In *directed* graphs (aka *digraphs*), edges have a specific direction:



Two common depictions of the same graph

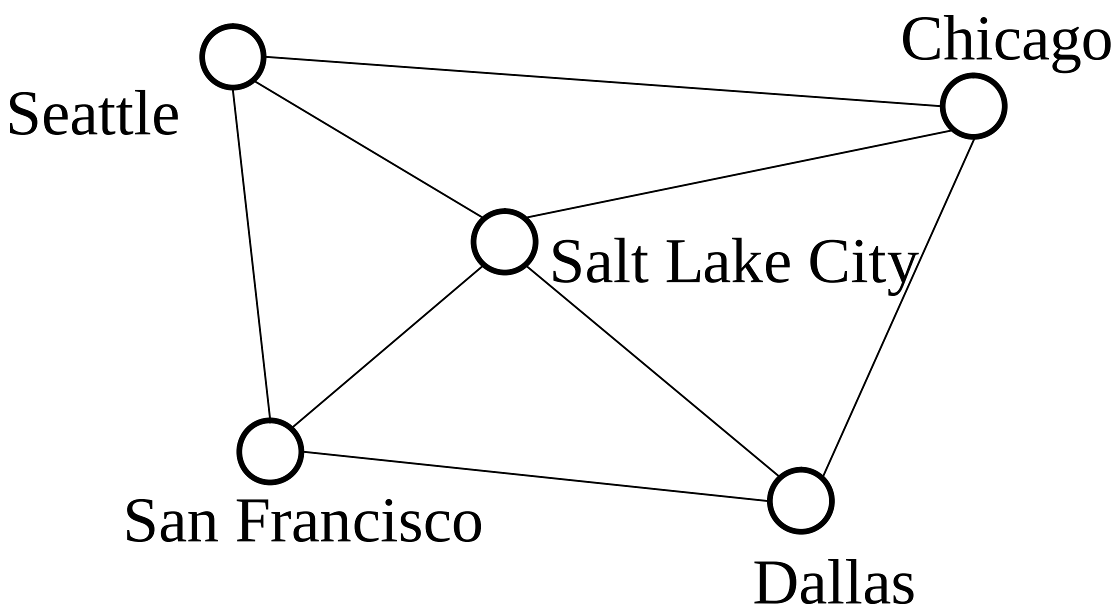


Thus, **`(u,v)`**$\in$**`E`** does *not* imply **`(v,u)`**$\in$**`E`**.

In-degree of a vertex: number of inbound edges.
Out-degree of a vertex : number of outbound edges.

Paths and Cycles

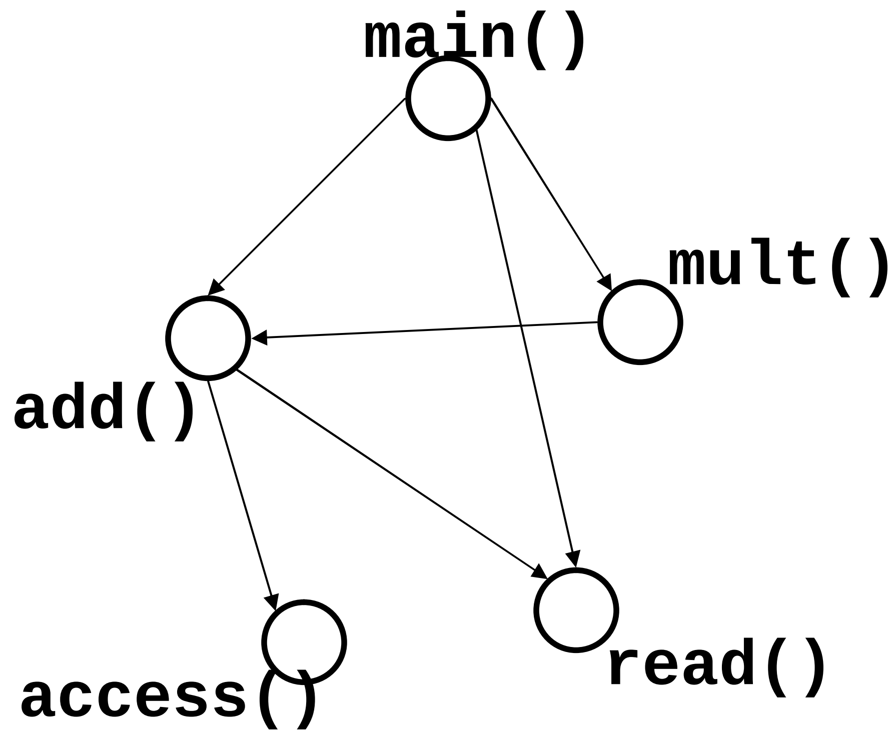
- A *path* is a list of vertices $\{v_1, v_2, \ldots, v_n\}$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$.
- A *cycle* is a path that begins and ends at the same node.



- *p = {Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle}*

Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.



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

{Rooted, directed tree} ⊂ {DAG} ⊂ {Graph}

|E| and |V|

How many edges |E| in a graph with |V| vertices?

$0 \le |E| \le |V|^2$

What if the graph is directed?

$0 \le |E| \le 2|V|^2$

What if it is undirected and connected?

$|V|-1 \le |E| \le |V|^2$

Can the following bounds be simplified?

- Arbitrary graph: $O(|E| + |V|^2)$

  $O(|V|^2)$

- Undirected, connected: $O(|E| \log|V| + |V| \log|V|)$

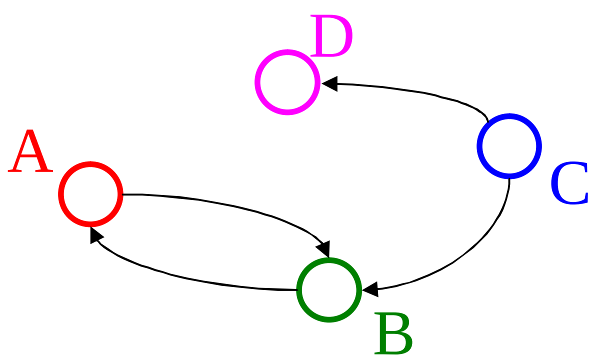
  $O(|E| \log|V|)$

Some (semi-standard) terminology:

- A graph is *sparse* if it has $O(|V|)$ edges (upper bound).
- A graph is *dense* if it has $\Theta(|V|^2)$ edges.

Representation 1: Adjacency Matrix

A **`|V|x|V|`** matrix **`M`** in which an element **`M[u,v]`** is true if and only if there is an edge from **`u`** to **`v`**



	A	B	C	D
A				
B				
C				
D				

Runtimes:

Iterate over vertices? O(|V|)

Iterate over edges? $O(|V|^2)$

Iterate edges adj. to vertex? O(|V|)

Existence of edge? O(1)

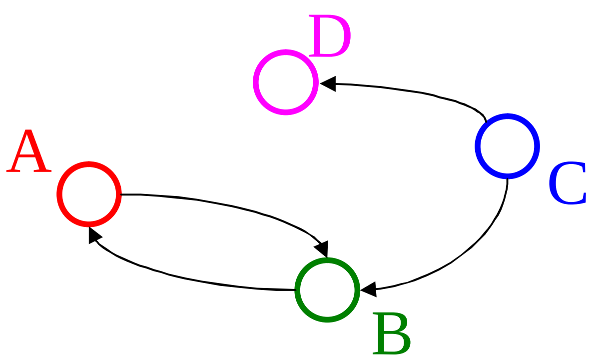
Space requirements? $O(|V|^2)$

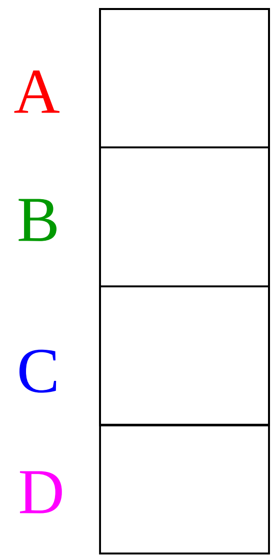
Best for what kinds of graphs?

dense

Representation 2: Adjacency List

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





Runtimes:

Iterate over vertices? O(|V|)

Iterate over edges? O(|V|+|E|)

Iterate edges adj. to vertex? O(d)

Existence of edge? O(d)

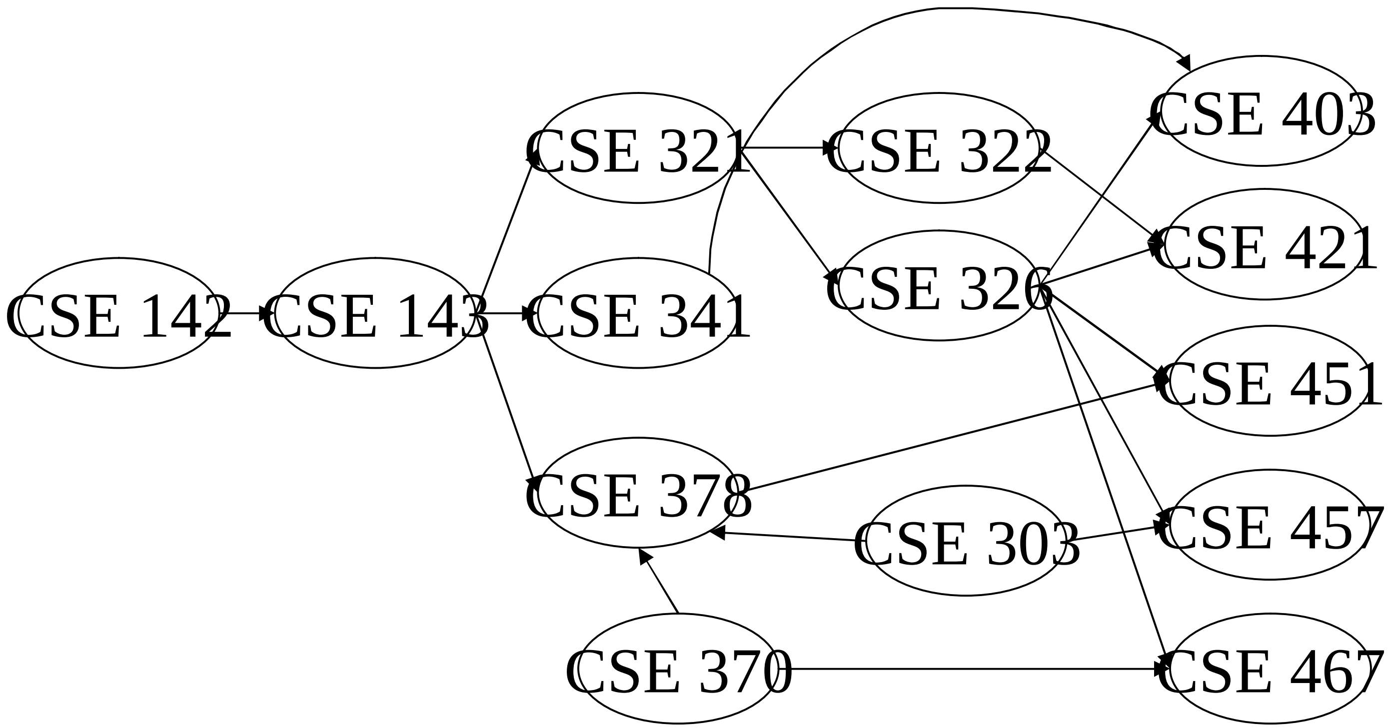
Space requirements? O(|V|+|E|)

Best for what kinds of graphs?

sparse

Application: Topological Sort

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

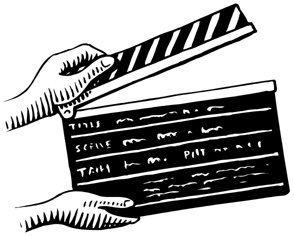


What kind of input graph is allowed?

DAG

Is the output unique?

No, often called a partial ordering



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*)

Is the use of a queue here important?

Runtime:

O(|V| + |E|)

No, can use a stack, list, set, box, etc.
Changes behavior, but result is still topological sort

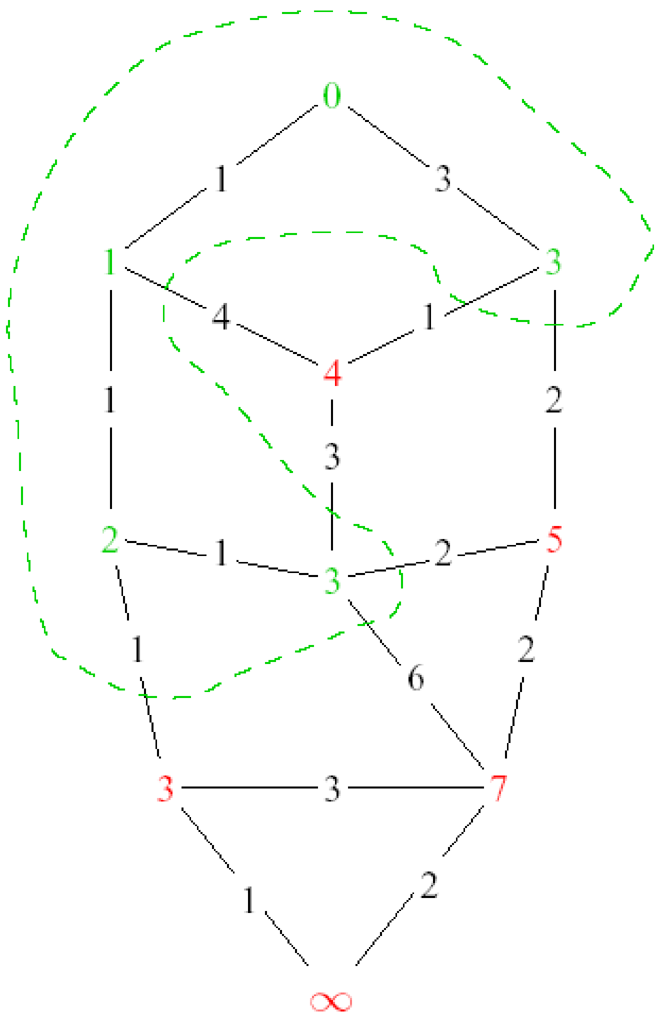
Comparison: DFS versus BFS

- Breadth-first search
  - –Always finds shortest paths – optimal solutions
  - –Marking visited nodes can improve efficiency, but even without this search guaranteed to terminate
- 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
- Is BFS always preferable?

Single Source Shortest Paths (SSSP)

- Given a graph G, edge costs $c_{i,j}$, and vertex s, find the shortest paths from s to all vertices in G.
- Is finding paths to all the vertices harder or easier than the previous problem?
  - The same difficulty (imagine the one we want is the last one we reach)
- But we still haven't dealt with edge costs...

Dijkstra's Algorithm: Idea



At each step:

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

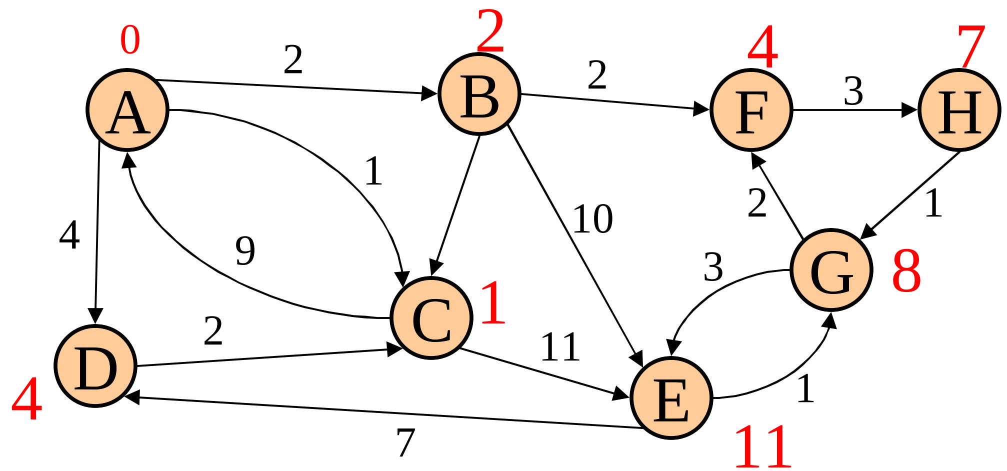
Dijkstra's Algorithm: Pseudocode

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*
  - if *b*'s cost + cost of (*b*, *a*) < *a*'s old cost
    - *a*'s cost = *b*'s cost + cost of (*b*, *a*)
    - *a*'s prev path node = *b*

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

```
void Graph::dijkstra(Vertex s){
  Vertex v,w;

  Initialize s.dist = 0 and set dist of all other
  vertices to infinity

  while (there exist unknown vertices, find the
  one b with the smallest distance)
    b.known = true;

    for each a adjacent to b
      if (!a.known)
        if (b.dist + weight(b,a) < a.dist){
          a.dist = (b.dist + weight(b,a));
          a.path = b;
        }
  }
}
```

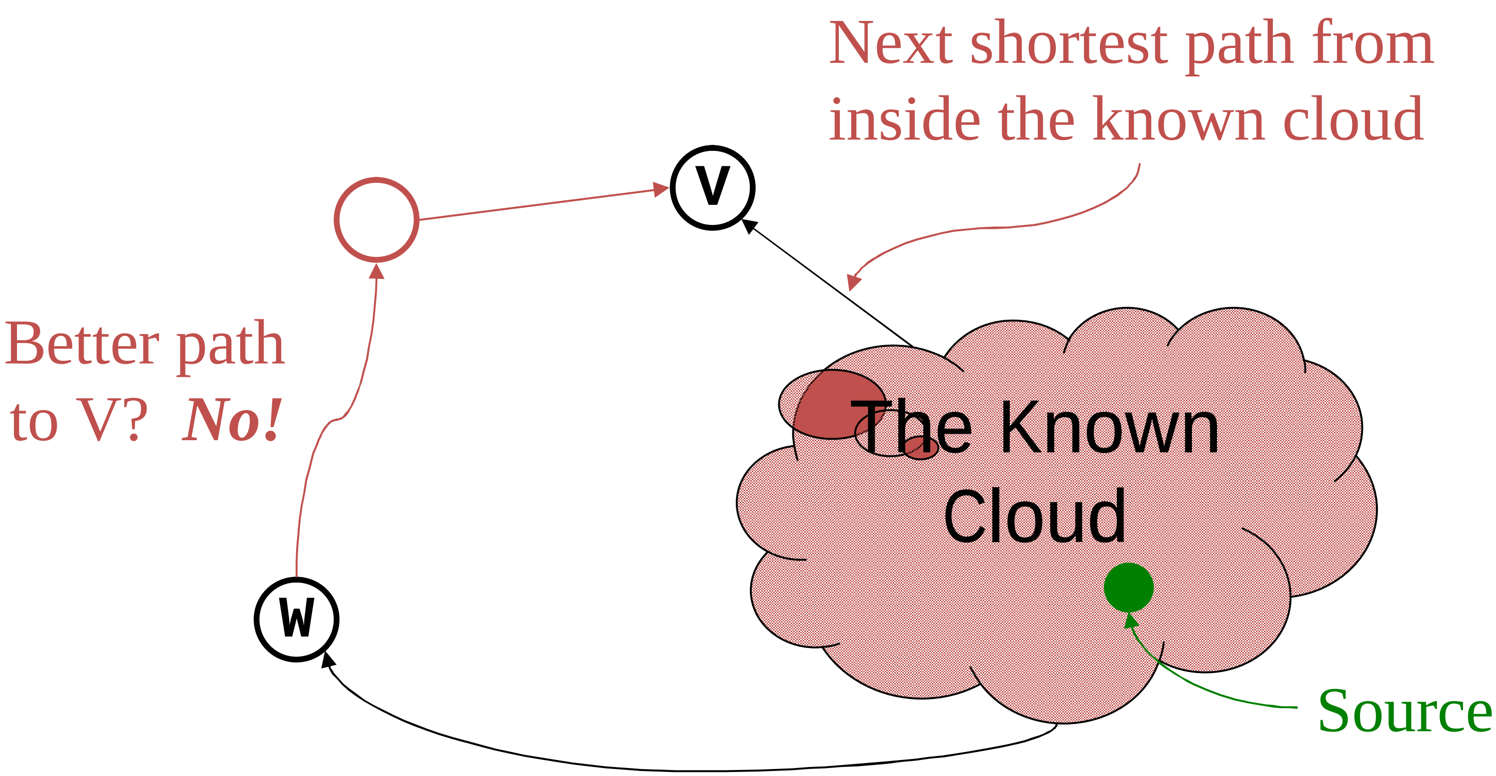
Sounds like deleteMin on a heap…

Sounds like adjacency lists

Sounds like decreaseKey

Running time: O(|E| log |V|) – there are |E| edges to examine, and each one causes a heap operation of time O(log |V|)

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!

Follow-On Question

- What if I had multiple potential start points, and need to know the minimum cost of reaching each node from any start point?
- Can do this by changing the algorithm
  – Add each start point to initial queue with cost 0
- If the algorithm is encapsulated (and highly tuned for efficiency), this seems bad
  – You need to re-implement the whole thing
  – Your implementation probably isn't as good

Thinking About Graph Structure

- Working with graphs is often a problem of setting up the right graph so that you can apply the unmodified standard algorithm

- Change the graph, apply the encapsulated and optimized SSSP implementation
  - Add a meta-start node
  - Include 0 cost edges from it to the start nodes

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

Simple for loop implementation intended to be fast (especially with the help of a modern compiler). Does not bother with if statements to filter out comparisons that will never result in a change.

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?

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**.

Reducing Best to Minimum

Let P(e) be the probability that an edge doesn't fail.
Define:

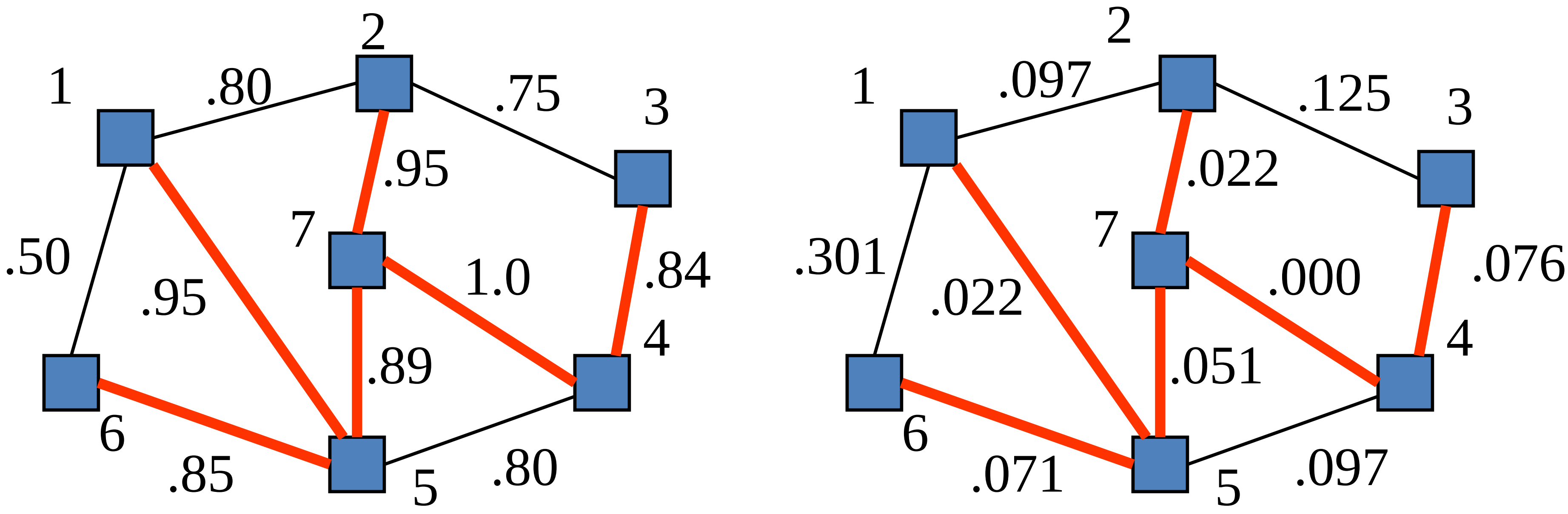
$$C(e) = -\log_{10}(P(e))$$

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

is equivalent to maximizing $\prod_{e \in T} P(e)$

because $\prod_{e \in T} P(e) = \prod_{e \in T} 10^{-C(e)} = 10^{-\sum_{e \in T} C(e)}$

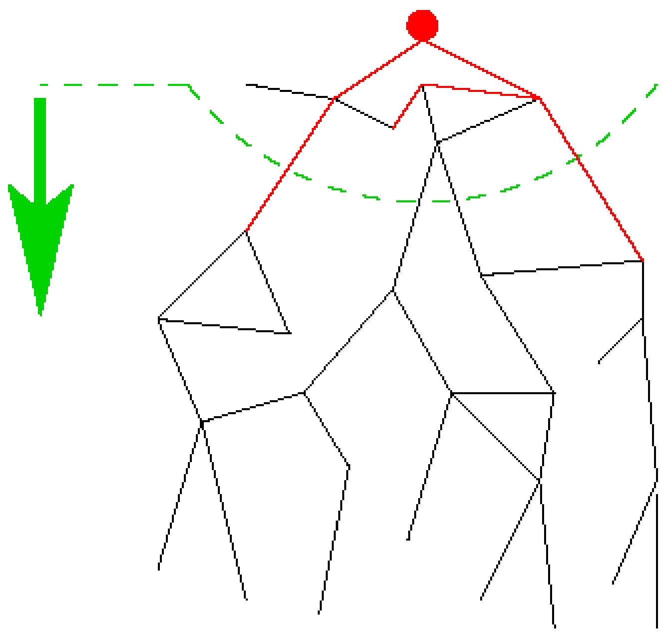
Example of Reduction



Best Spanning Tree Problem

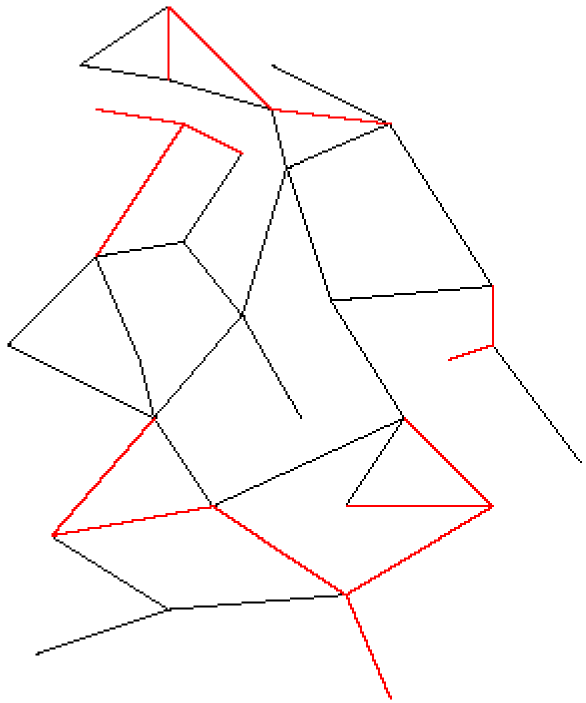
Minimum Spanning Tree Problem

Two Different Approaches



Prim's Algorithm

Looks familiar!

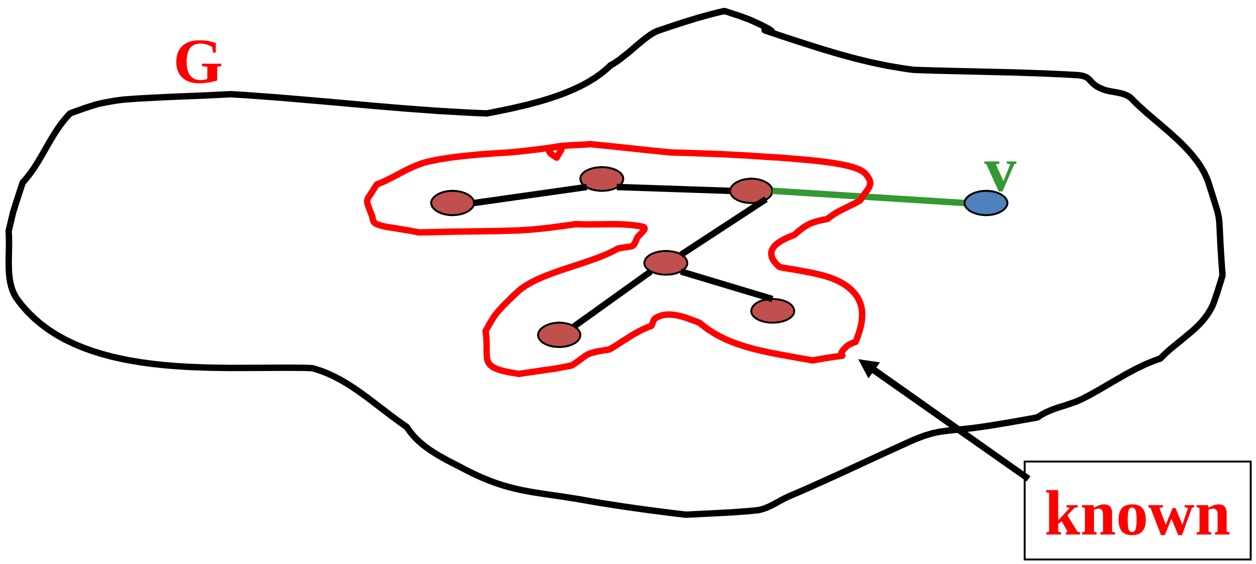


Kruskals's Algorithm

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.



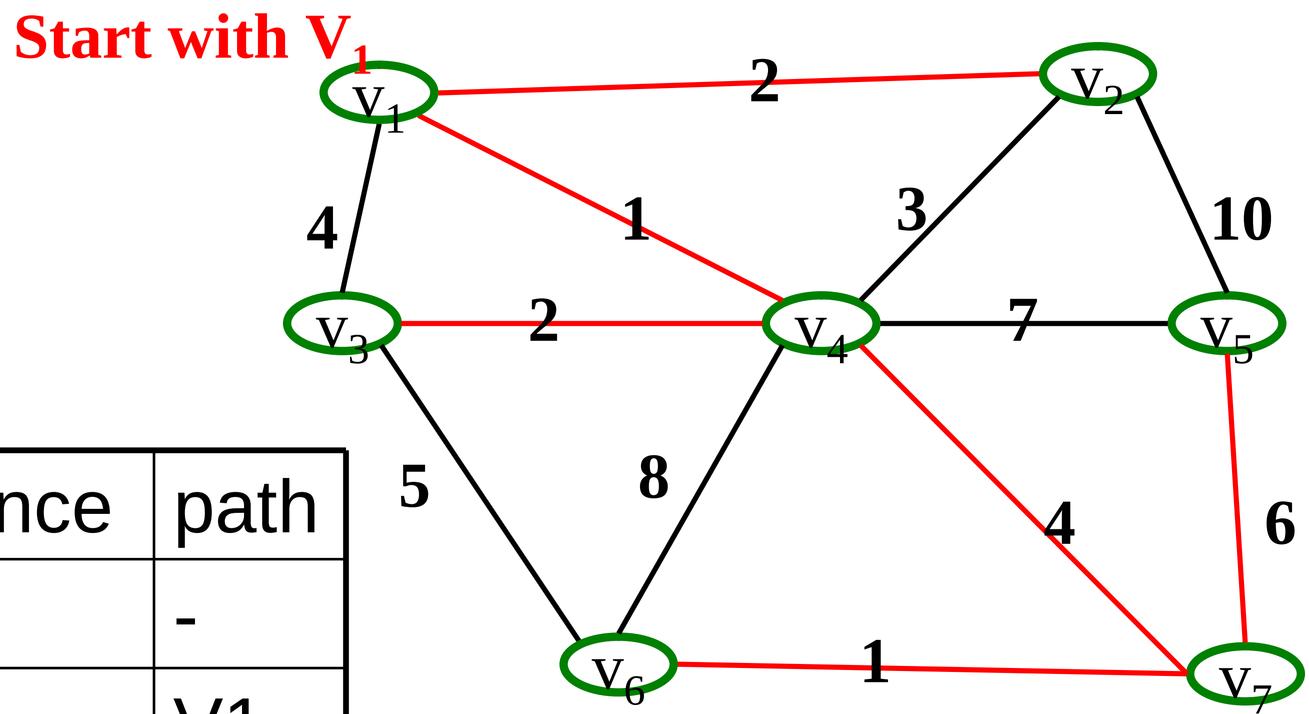
Prim's Algorithm for MST

A *node-based* greedy algorithm

Builds MST by greedily adding nodes

1. Select a node to be the "root"
   - mark it as known
   - Update cost of all its neighbors
2. While there are unknown nodes left in the graph
   a. Select an unknown node *b* with the smallest cost to reach from some *known* node *a*
   b. Mark *b* as known
   c. Add (*a*, *b*) to MST
   d. Update cost of all nodes adjacent to *b*

Find MST using Prim's



V	Kwn	Distance	path
v1	Y	-	-
v2	Y	2	V1
v3	Y	2	V4
v4	Y	1	V1
v5	Y	6	V7
v6	Y	1	V7
v7	Y	4	V4

Order Declared Known:

V1, V4, V2, V3, V7, V6, V5

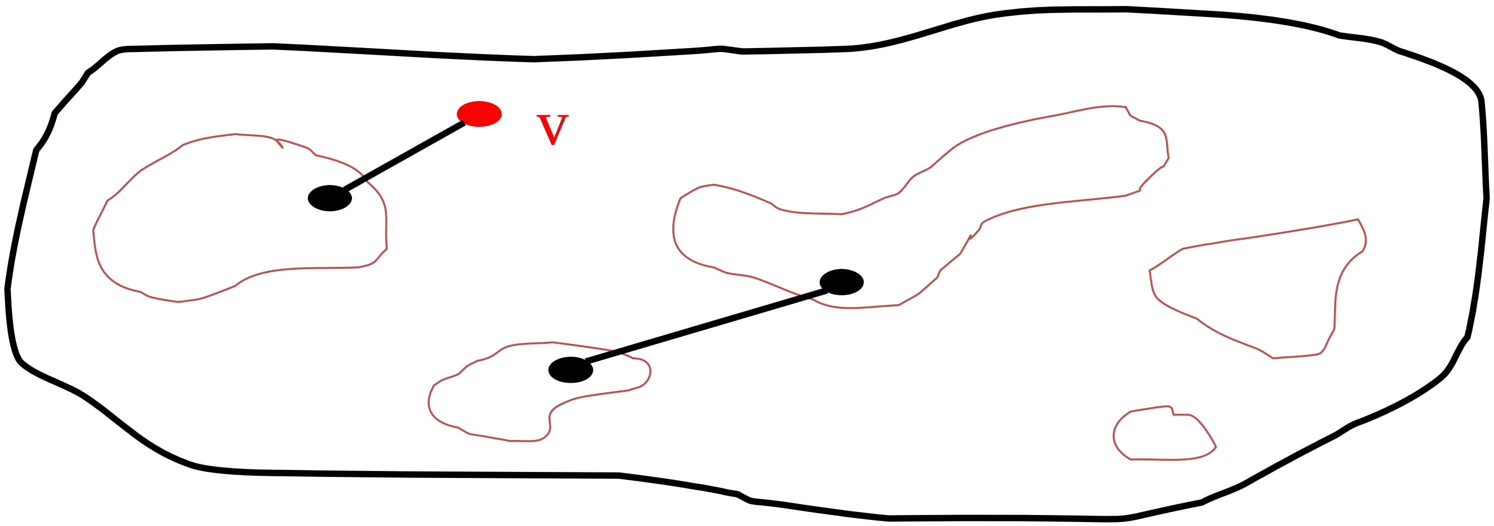
Selected Edges:

{V2, V1}, {V3, V4}, {V4, V1}, {V5, V7}, {V6, V7}, {V7, V4}

Kruskal's MST Algorithm

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

G=(V,E)



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 <u>lowest cost edge</u> **`(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

Sound familiar?

Optimized Kruskal code

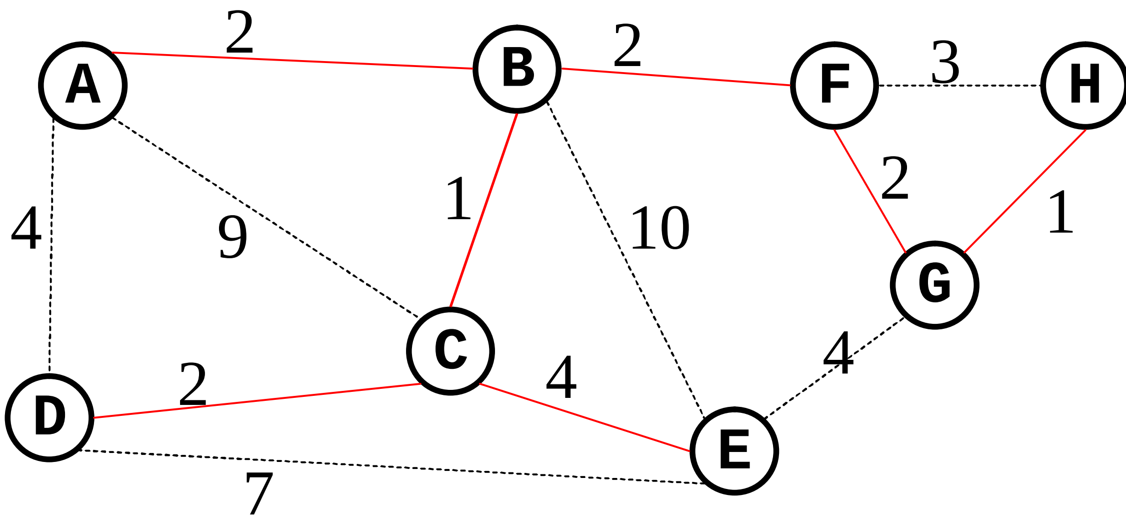
```
void Graph::kruskal(){
  int edgesAccepted = 0;
  DisjSet s(NUM_VERTICES);

  while (edgesAccepted < NUM_VERTICES – 1){
    e = smallest weight edge not deleted yet;
    // edge e = (u, v)
    uset = s.find(u);
    vset = s.find(v);
    if (uset != vset){
      edgesAccepted++;
      s.unionSets(uset, vset);
    }
  }
}
```

|E| heap ops

2|E| finds

|V| unions

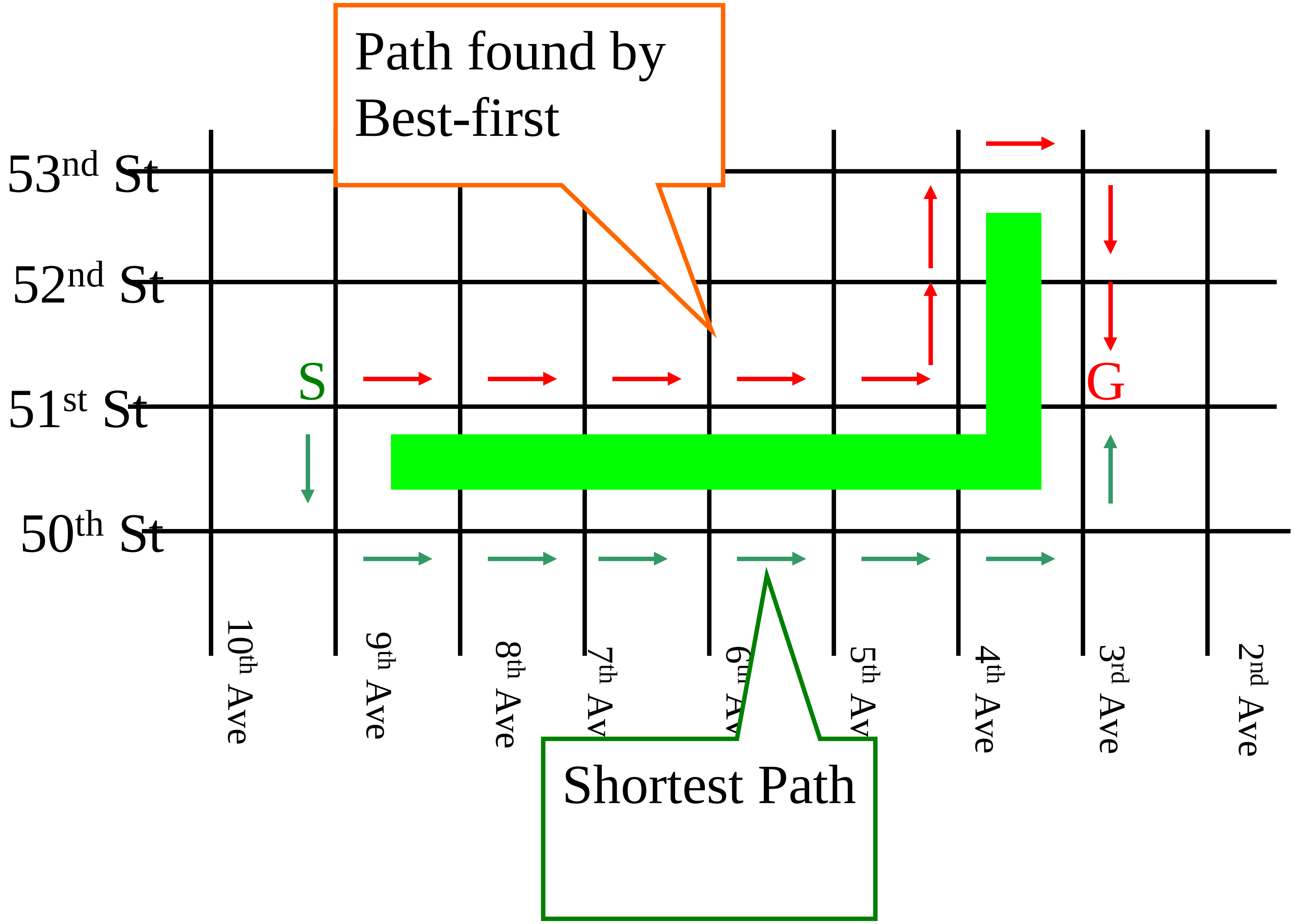
Find MST using Kruskal's



Total Cost: 14

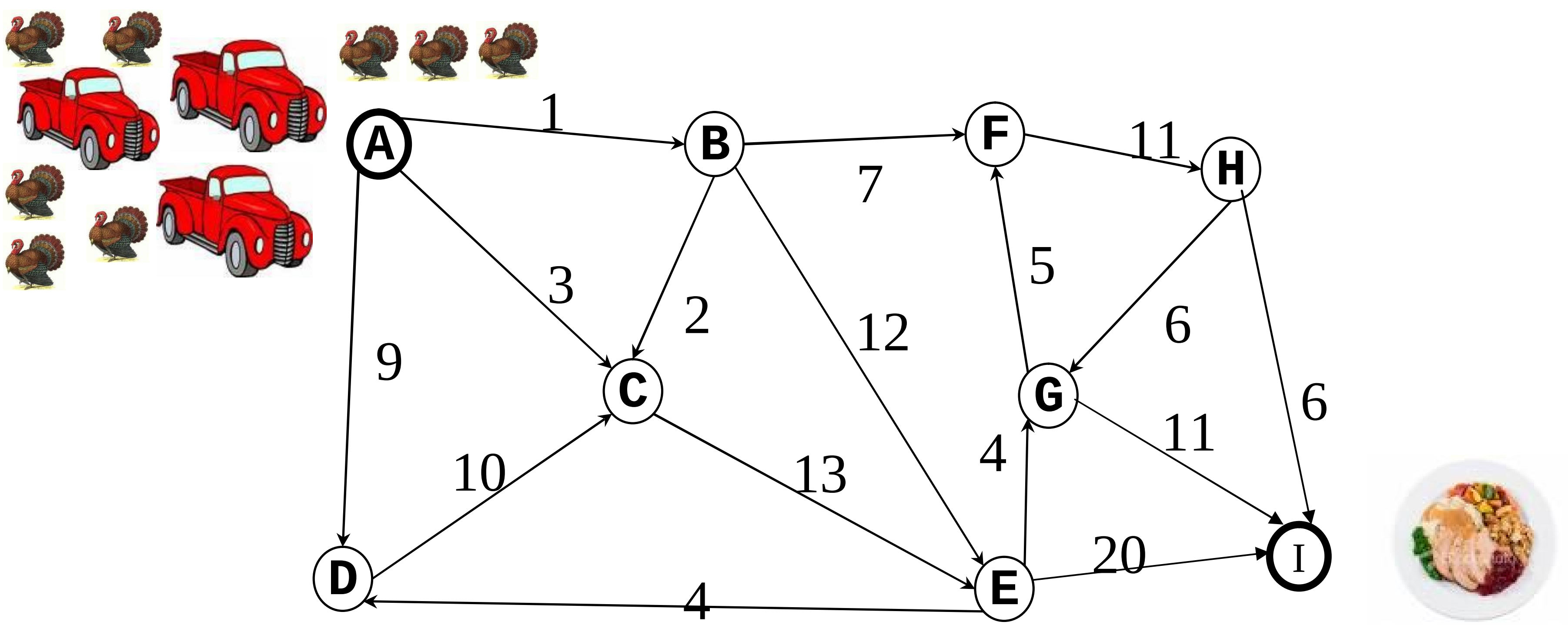
- Is this MST unique?
- Under what condition is an MST unique?
  - Unique edge weights guarantee uniqueness

Best-First


Path found by Best-first
53nd St
52nd St
51st St
50th St
S
G
10th Ave
9th Ave
8th Ave
7th Ave
6th Ave
5th Ave
4th Ave
3rd Ave
2nd Ave
Shortest Path

Network Flow

- So, how do we want to go about this?



Ford-Fulkerson Method

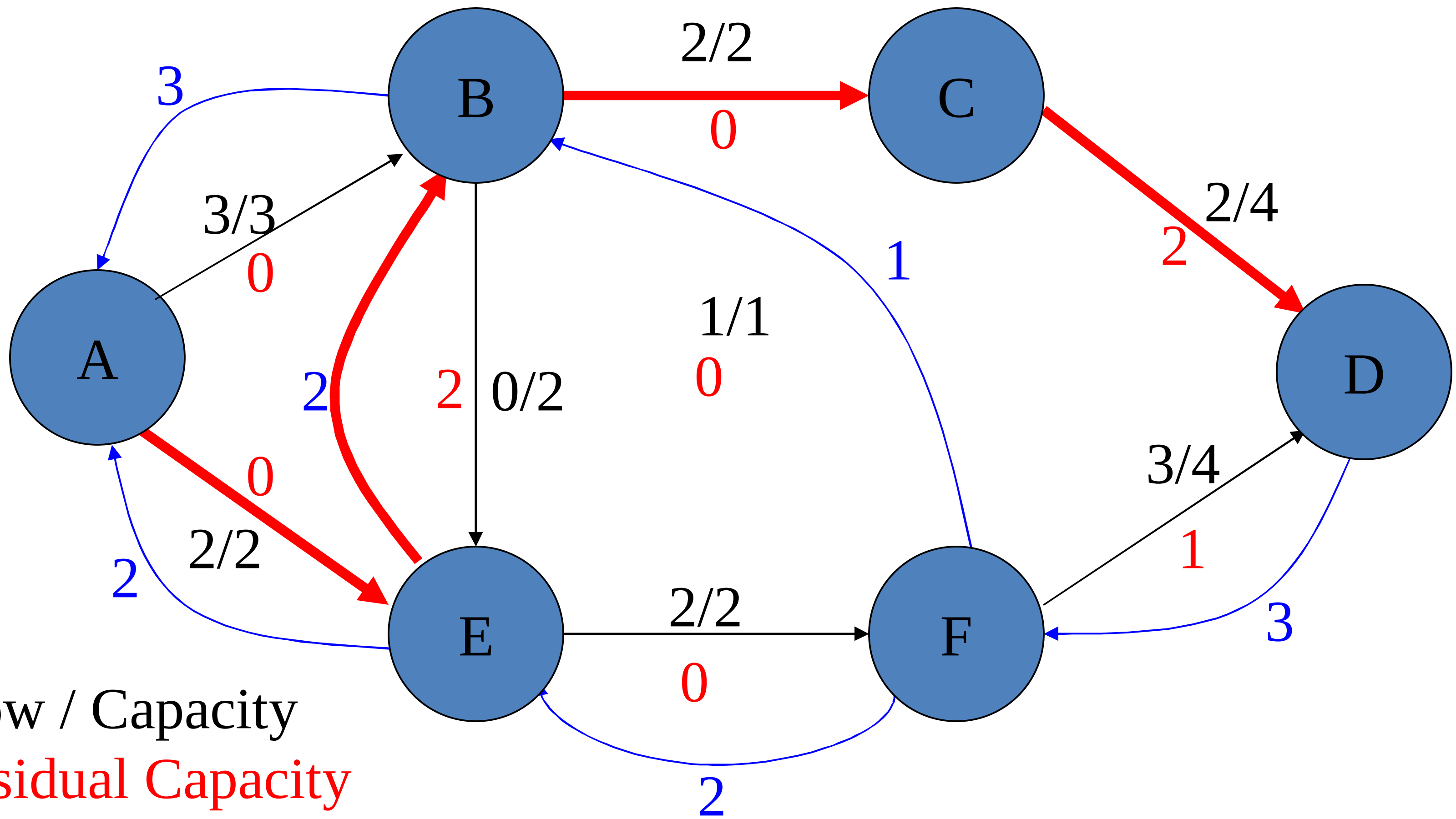
- Our greedy algorithm makes choices about how to route flow, and we never reconsider those choices
- Can we develop a way to efficiently reconsider the choices we already made?
- Can we do it by just modifying the graph?

Residual Graph

- Constructing a residual graph:
  - Use the same vertices
  - Edge weights are the remaining capacity on the edges, given the existing augmenting paths
  - Add additional edges for backward capacity
  - If there is a path from *s* to *t* in the residual graph, then there is available capacity there

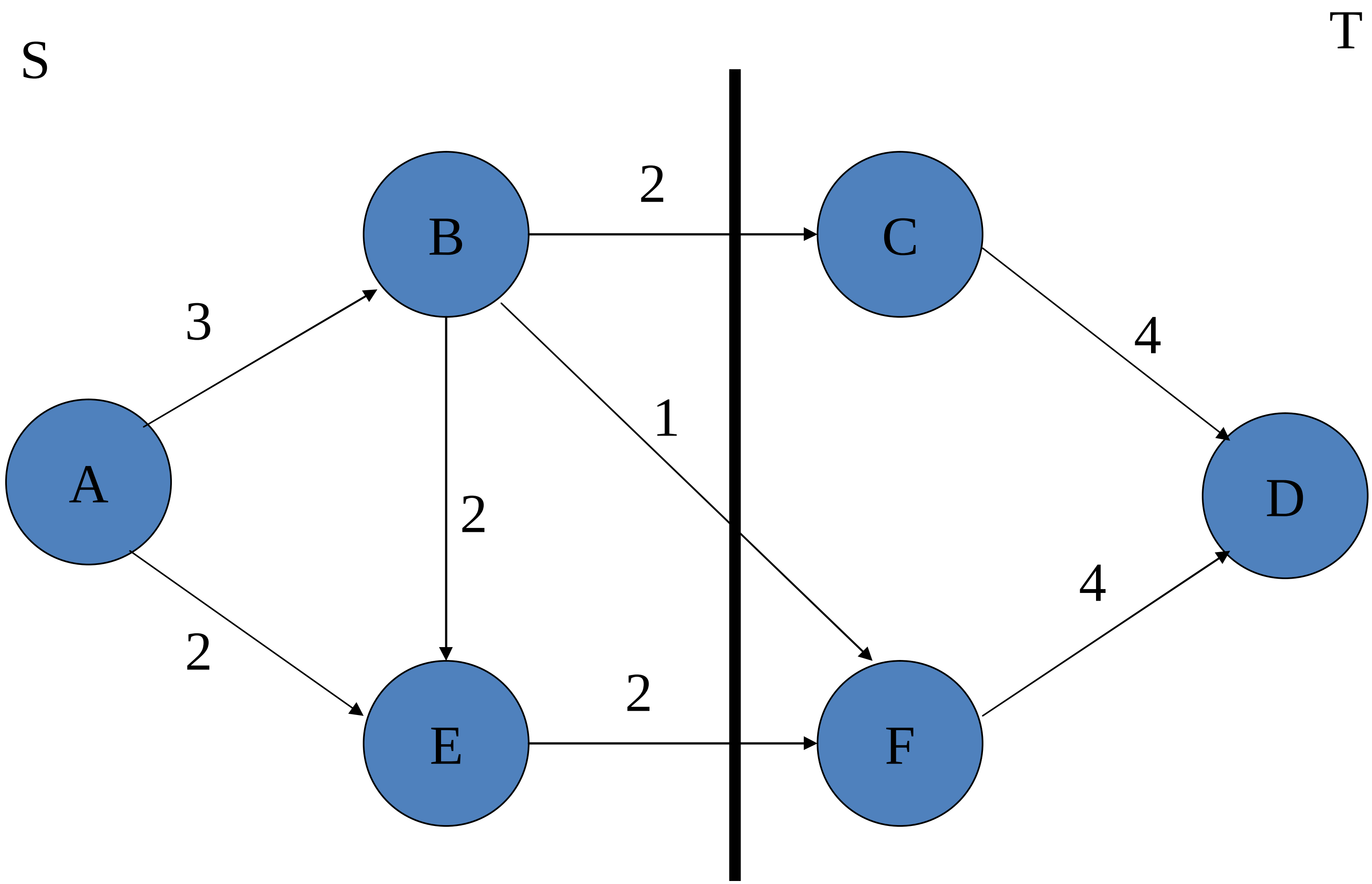
Example

Augment along AEBCD (which saturates AE and EB, and empties I



Flow / Capacity
Residual Capacity
Backwards flow

Min Cut - Example



Capacity of cut = 5

Coincidence?

- No, Max-flow always equals Min-cut
  - If there is a cut with capacity equal to the flow, we have a maxflow:
    - We can't have a flow that's bigger than the capacity cutting the graph! So any cut puts a bound on the maxflow, and if we have an equality, then we must have a maximum flow.
  - If we have a maxflow, then there are no augmenting paths left
    - Or else we could augment the flow along that path, which would yield a higher total flow.
  - If there are no augmenting paths, we have a cut of capacity equal to the maxflow
    - Pick a cut (S,T) where S contains all vertices reachable in the residual graph from s, and T is everything else. Then every edge from S to T must be saturated (or else there would be a path in the residual graph). So c(S,T) = f(S,T) = f(s,t) = |f| and we're done.