Data Structures and Algorithms

If time

All Pairs Shortest Path (Plus Review)

CS 225
Brad Solomon

November 7, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

Introduce and discuss All-Pairs Shortest Path

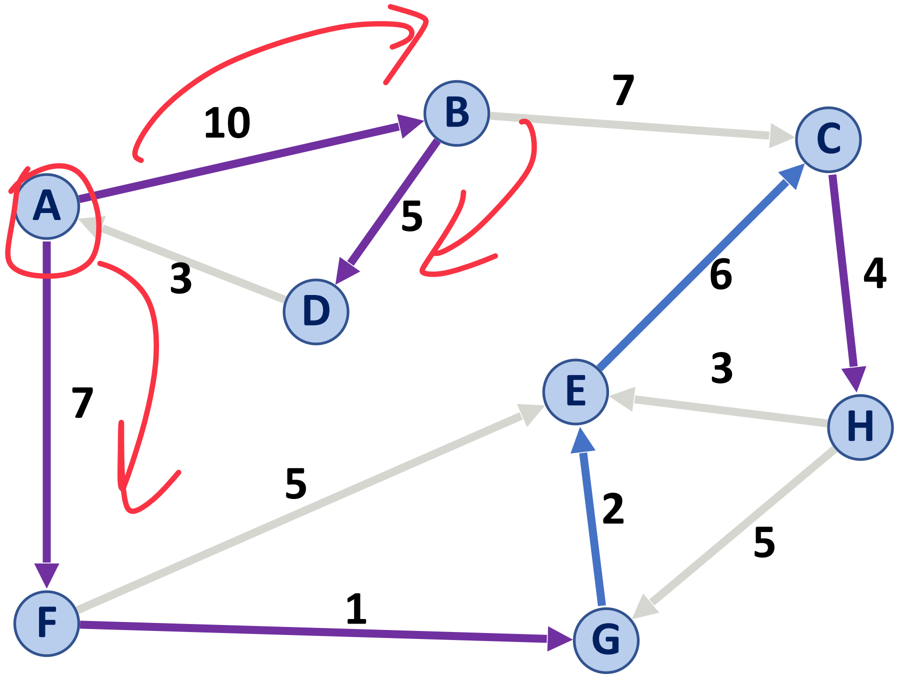
Review deterministic data structures in CS

An opportunity for Q&A for exam 4

Foreshadowing probabilistic data structures

Dijkstra's Algorithm (SSSP)

Single source shortest path



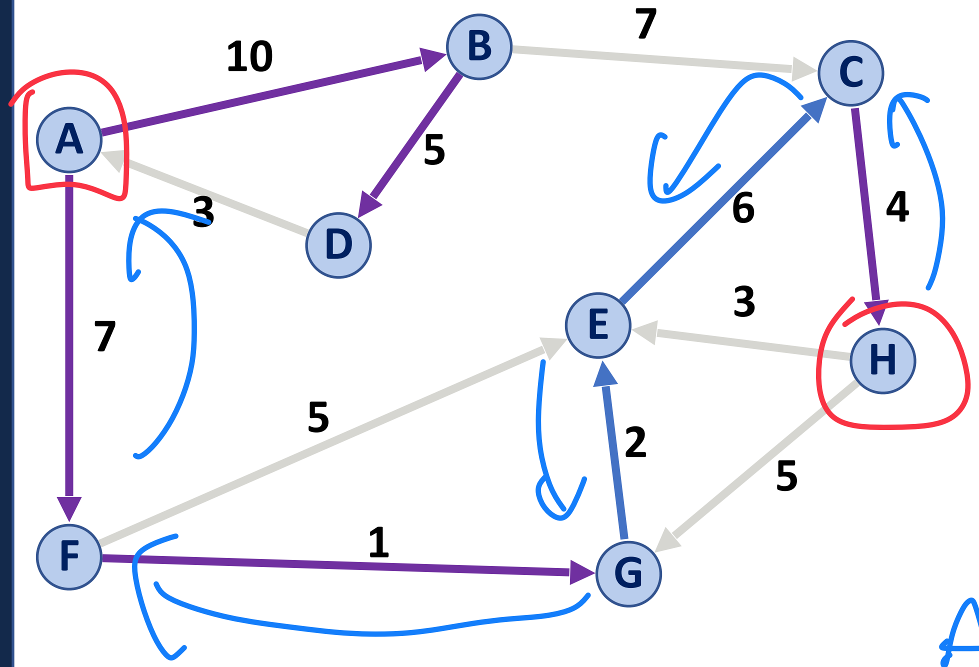

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = u
```

Only diff w/ Prim

sum of shortest path

A	B	C	D	E	F	G	H
--	A	E	B	G	A	F	C
0	10	16	15	10	7	8	20

Dijkstra's Algorithm (SSSP)



Whats the point of predecessor?

↳ stores my shortest path

To get path, start at endpoint and back track

A — F — G — E — C — H

A	B	C	D	E	F	G	H
--	A	E	B	G	A	F	C
0	10	16	15	10	7	8	20

Dijkstra's Algorithm (SSSP)

What is the running time of Dijkstra's Algorithm?

$O(m + n \log n)$

The same as Prim's!

6-9: O(n)

11-12: O(n)

15: repeat below n x

16-22: O(log n)

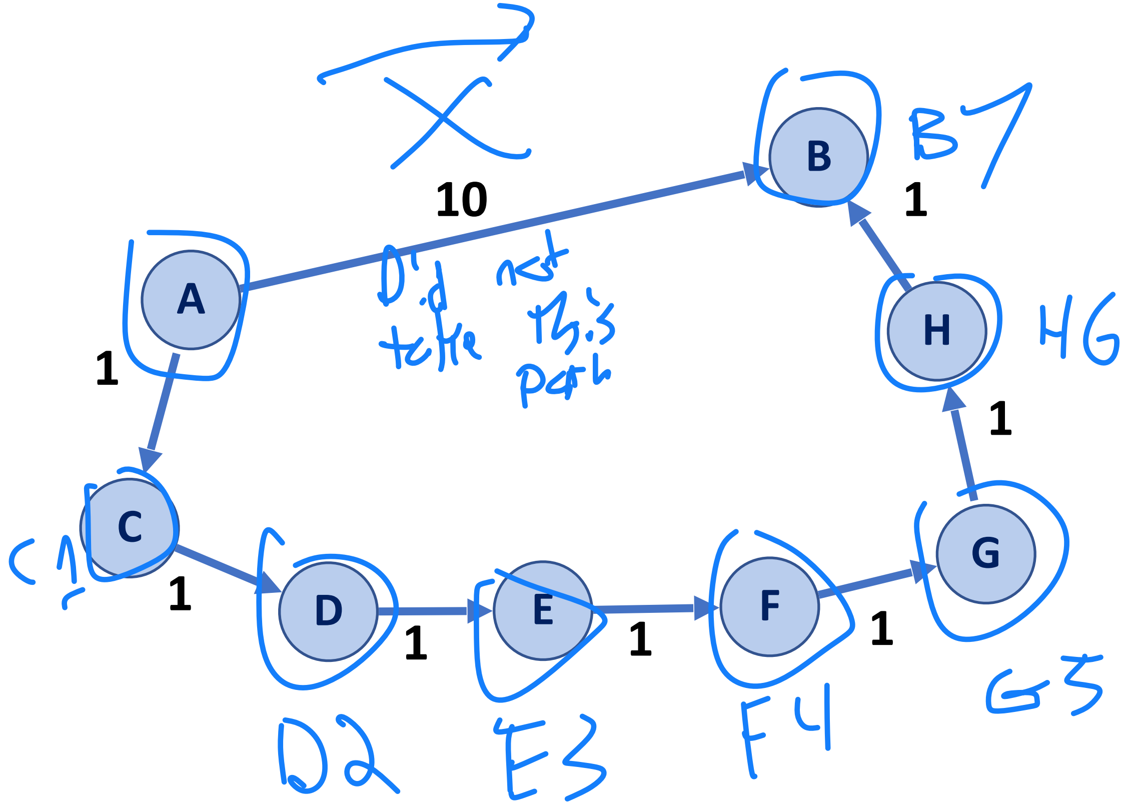
[w/ Fib Heap O(1) updates]

```
   DijkstraSSSP(G, s):
6    foreach (Vertex v : G):
7      d[v] = +inf
8      p[v] = NULL
9    d[s] = 0
10
11   PriorityQueue Q // min distance, defined by d[v]
12   Q.buildHeap(G.vertices())
13   Graph T         // "labeled set"
14
15   repeat n times:
16     Vertex u = Q.removeMin()
17     T.add(u)
18     foreach (Vertex v : neighbors of u not in T):
19       if cost(u, v) + d[u] < d[v]:
20         d[v] = cost(u, v) + d[u]
21         p[v] = m
22
23   return T
```

Dijkstra's Algorithm (SSSP)

Claim: Dijkstras will always visit a node through its optimal shortest path.

When we will visit B in the following graph?



B 10

C 1

A → ___ → B

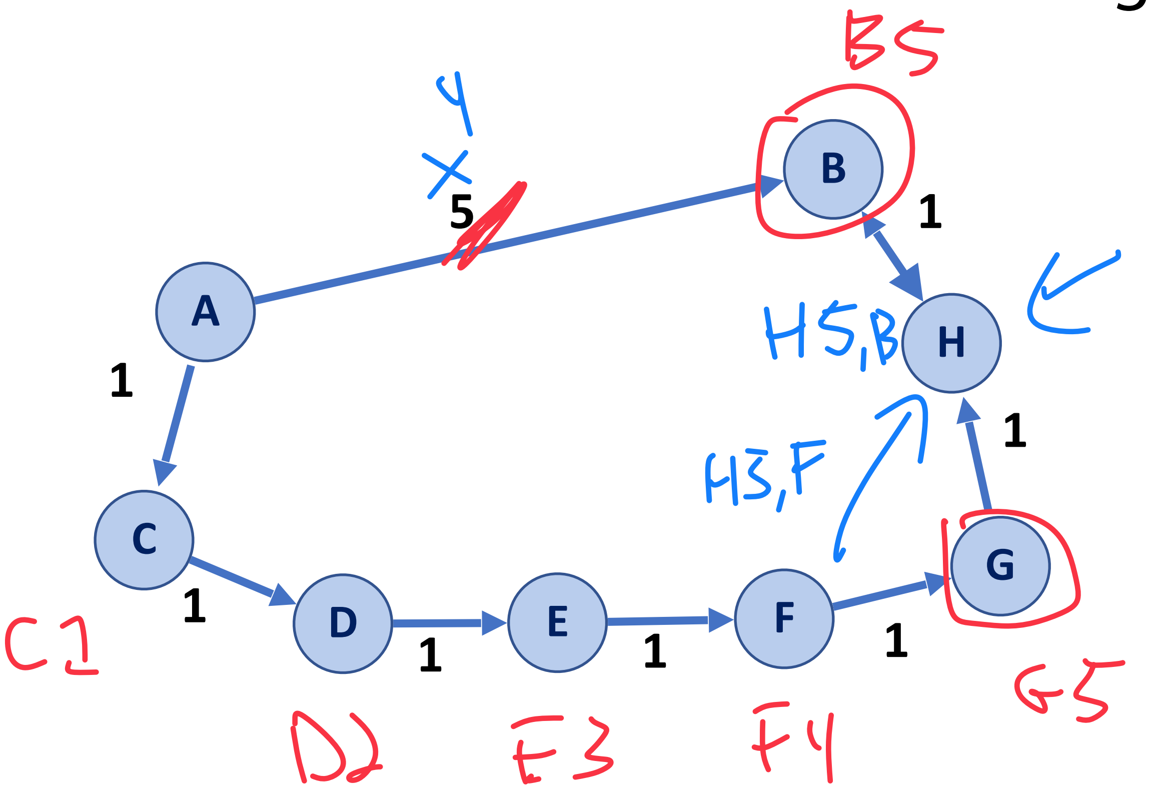
Must be smaller than 10

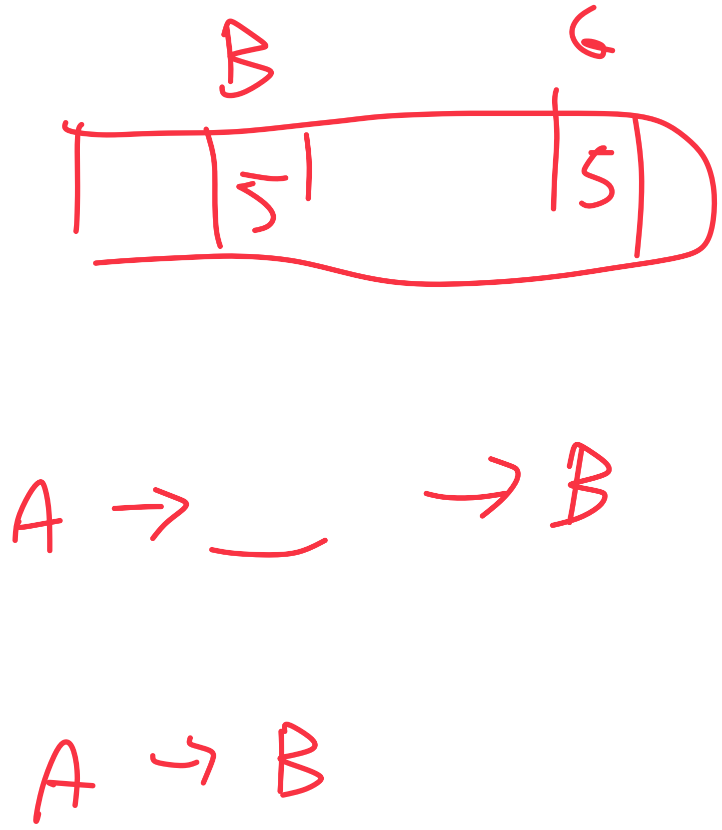
$A \xrightarrow{10} B$

Dijkstra's Algorithm (SSSP)

Claim: Dijkstras will always visit a node through its optimal shortest path.

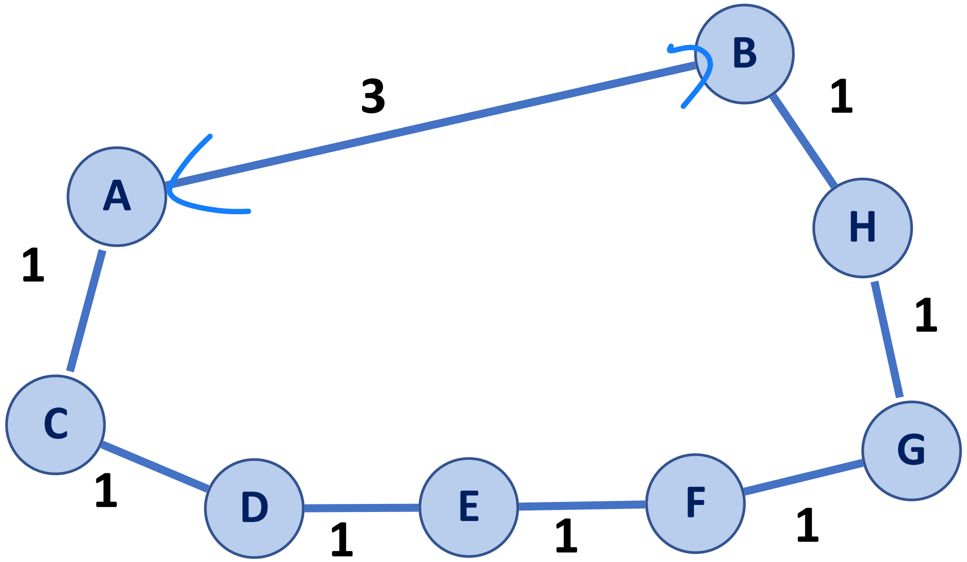
When we will visit H in the following graph?





Dijkstra’s Algorithm (SSSP)

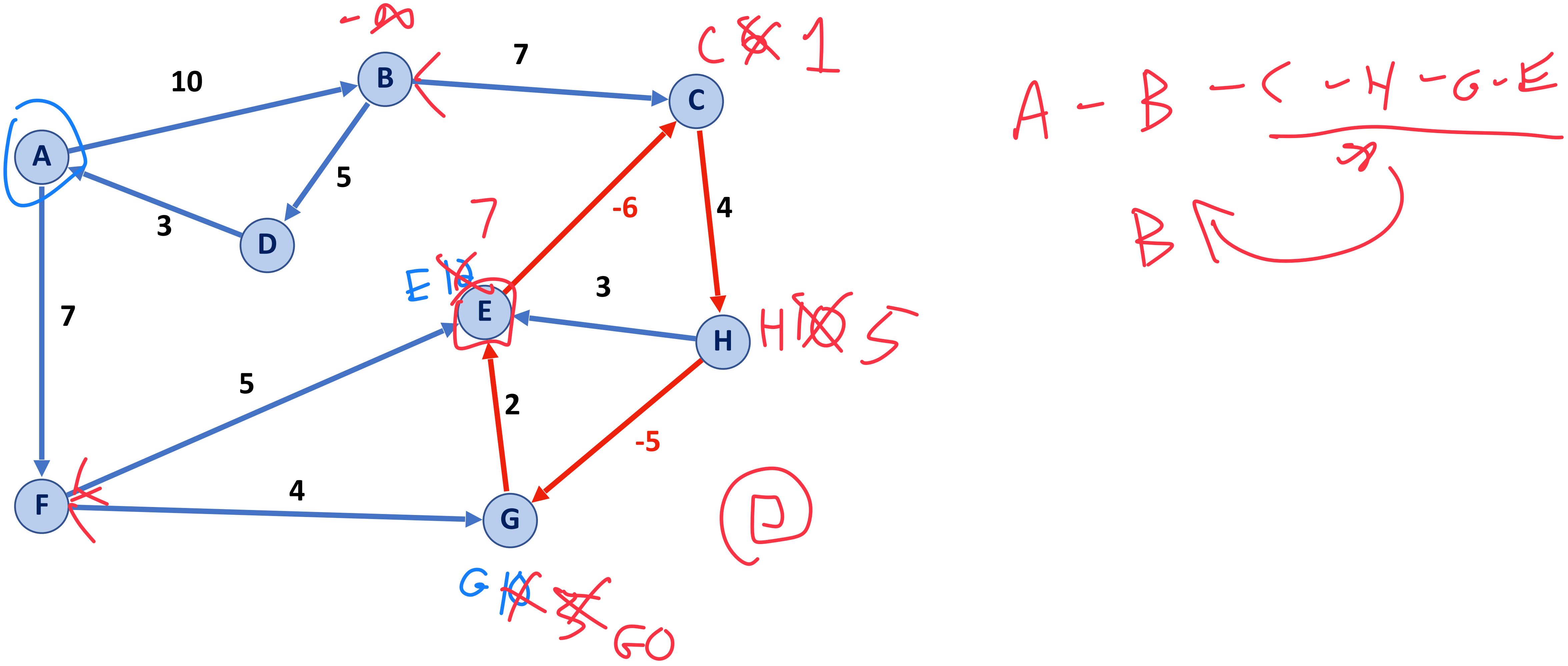
How does Dijkstra’s algorithm handle undirected graphs?



No problem here!

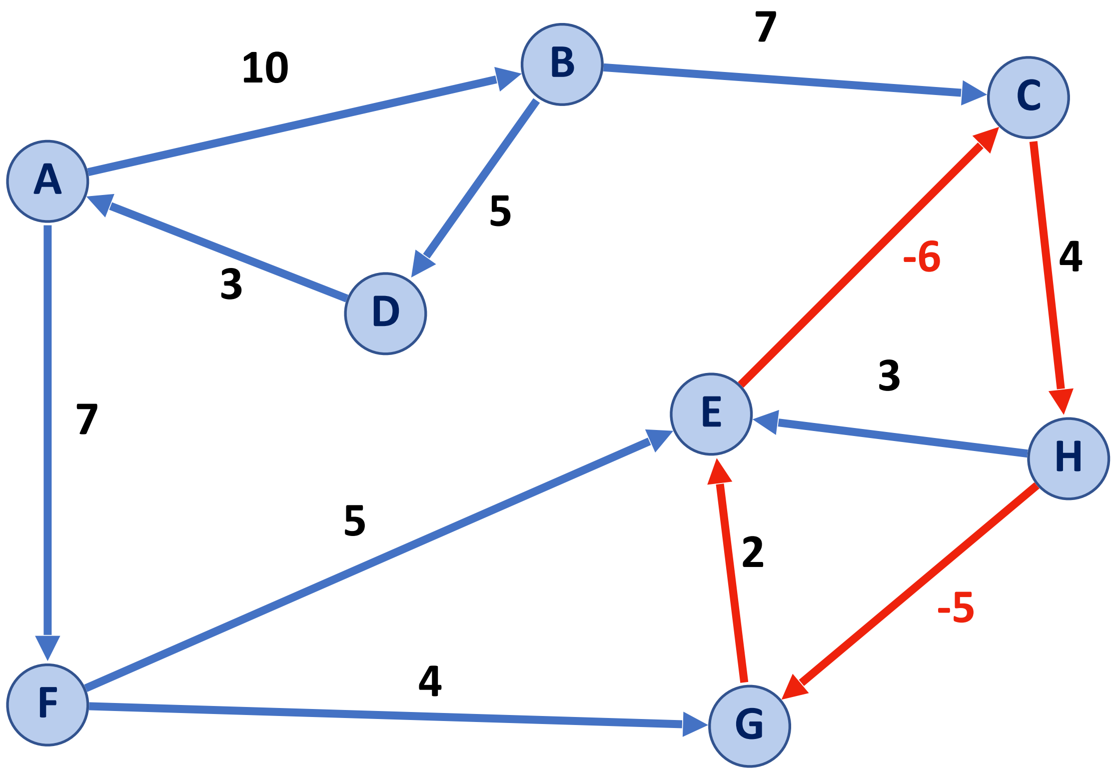
Dijkstra’s Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



Dijkstra's Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



Doesnt work!

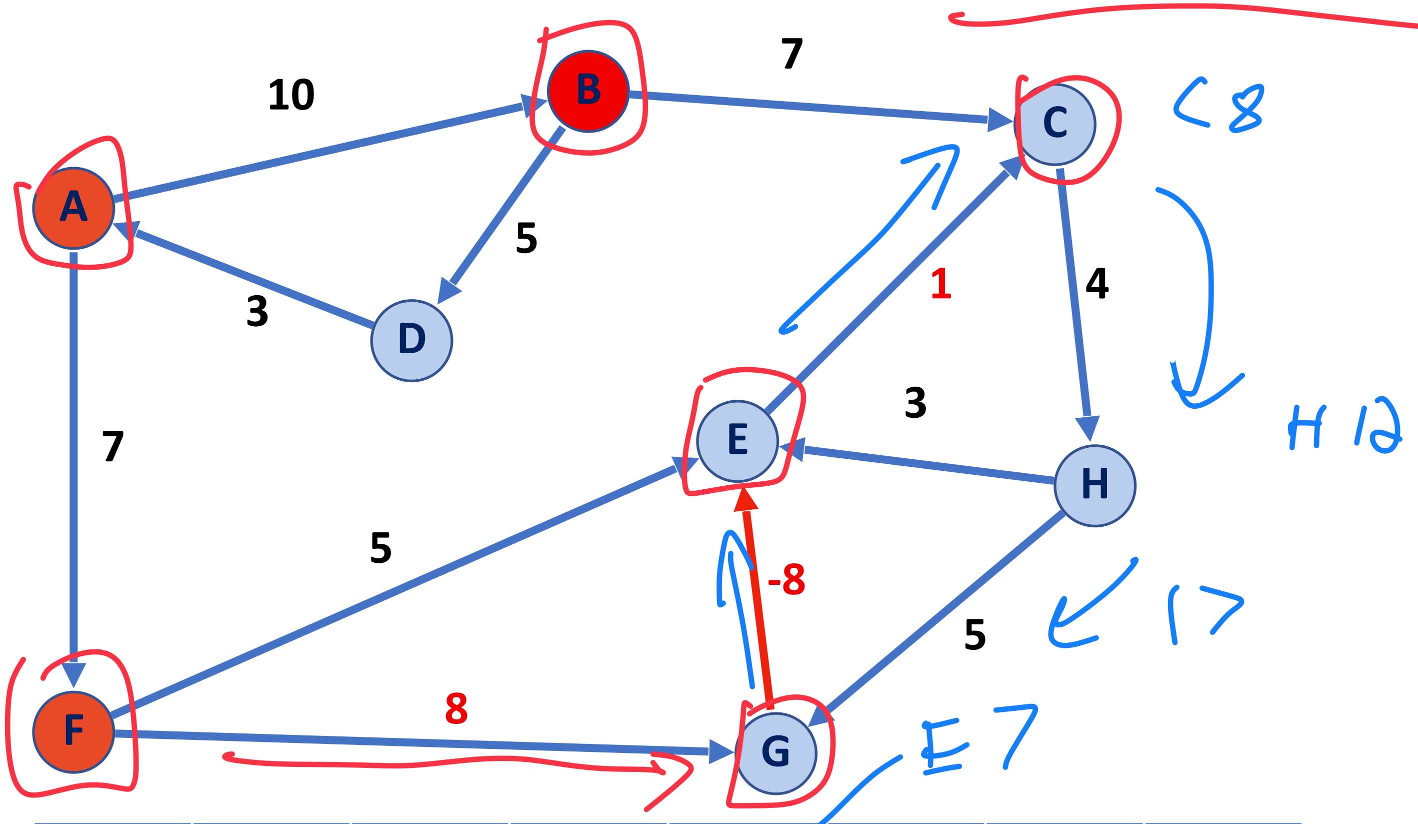
Shortest Path (A → E): A → F → E → (C → H → G → E)*

Length: 12

Length: -5 (repeatable)

Dijkstra’s Algorithm (SSSP)

How does Dijkstras handle a negative weight edge without a cycle?



A single negative weight breaks our loop from running n times to running potentially many more times!

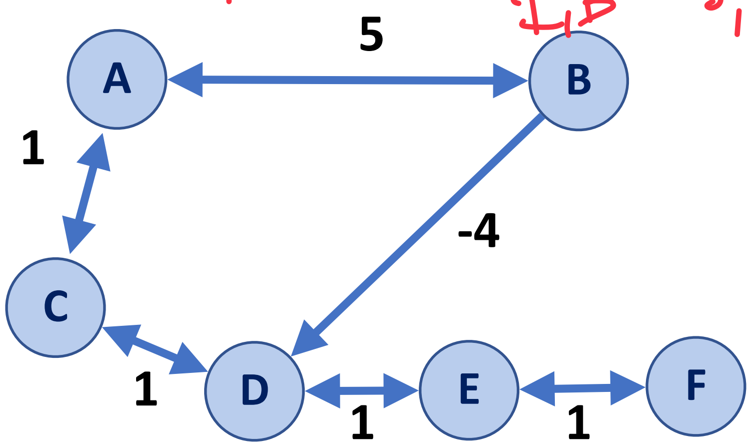
A	B	C	D	E	F	G	H
--	A	B	B	F	A	F	~~--~~ C
0	10	~~17~~ 13	15	12	7	15	~~∞~~ 17

Dijkstra's Algorithm (SSSP)

We assume that item pulled out of priority queue is **the next smallest item**

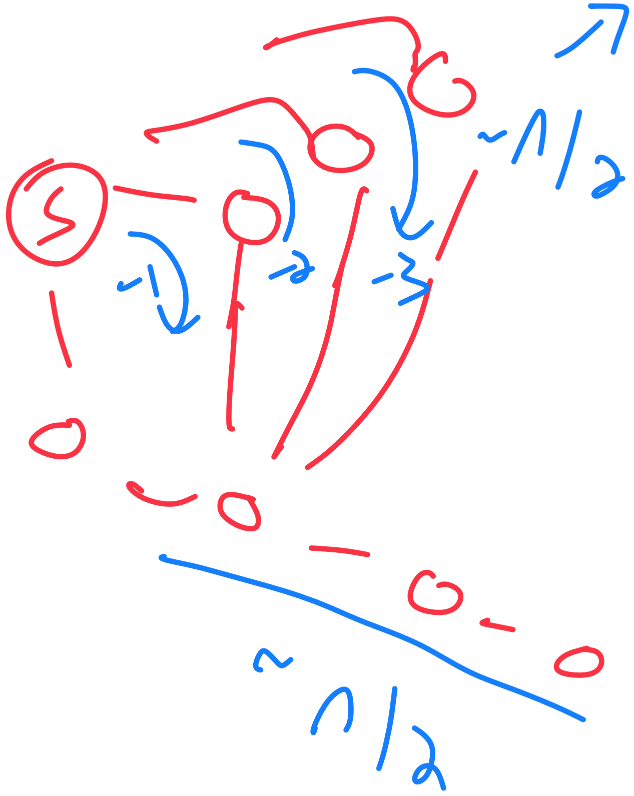
Negative weights break this assumption!

A	B	C	D	E	F
--	A	A	C	D	E
0	5	1	~~2~~	~~3~~	~~4~~
			1, B	2, 1	3



Dijkstra's Algorithm (SSSP)

Recalculating all distances is possible, but algorithm runtime is very bad!



Tangent point

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat until Q.empty():
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m
        if v not in Q:
          Q.push(v)
  return T
```

+n/2

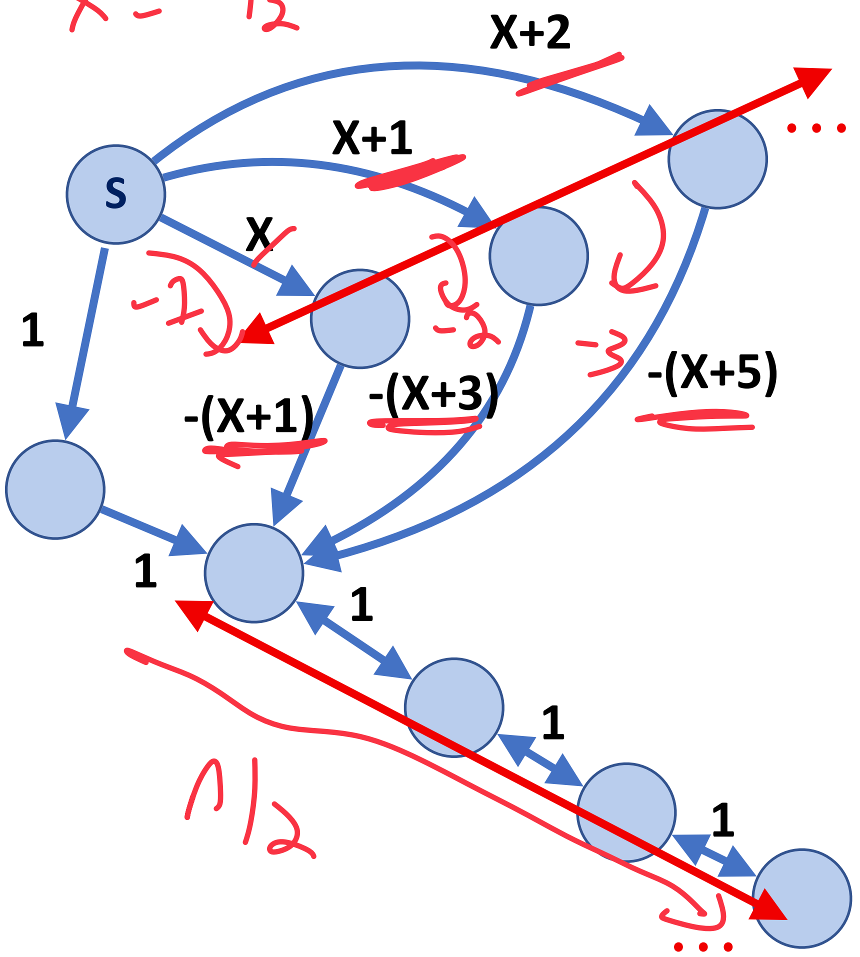
Dijkstra's Algorithm (SSSP)

A factor of n^2 worse

Recalculating all distances is possible, but algorithm runtime is very bad!

Worst case: ~n/2 nodes each updating ~n/2 nodes distances

$x \geq n/2$




```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat until Q.empty():
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m
        if v not in Q:
          Q.push(v)
  return T
```

This modification allows negative weight

Dijkstra's Algorithm (SSSP)

Dijkstras Algorithm works only on non-negative weights

Optimal implementation:

Fibonacci Heap

If dense, unsorted list ties

Optimal runtime:

Sparse: $O(m + n \log n)$

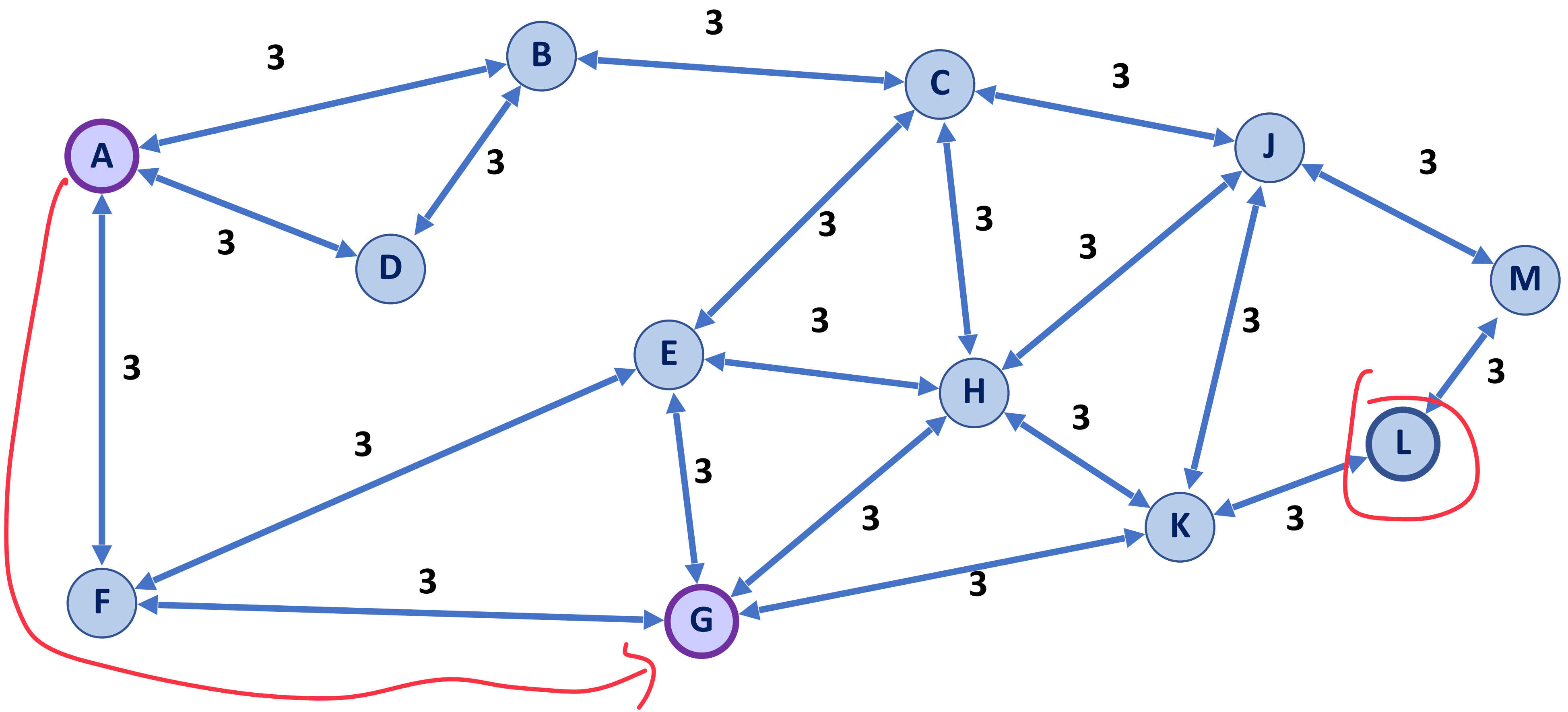
Dense: $O(n^2)$

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m

  return T
```

Landmark Path Problem

What if I wanted to get the shortest path from A to G but stopping at L along the way?

shortest path A → L
shortest path L - G



Floyd-Warshall Algorithm

Floyd-Warshall's Algorithm is an alternative to Dijkstra in the presence of negative-weight edges (not negative weight cycles).

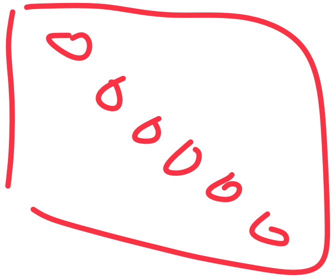
```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex u : G):
    foreach (Vertex v : G):
      foreach (Vertex w : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
```

self loops ignored

Add existing edges to matrix

Init

Make a matrix



u - v ?

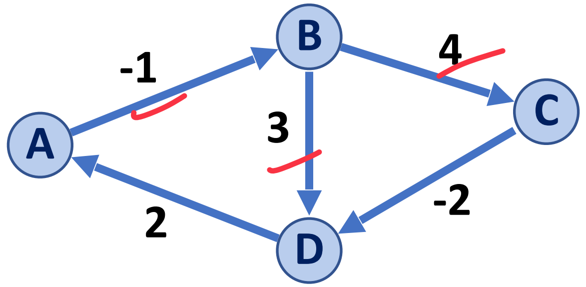
Is there a shorter path from u to v that goes through w?

u - w - v ?

Floyd-Warshall Algorithm

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)
```

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	∞	∞	0

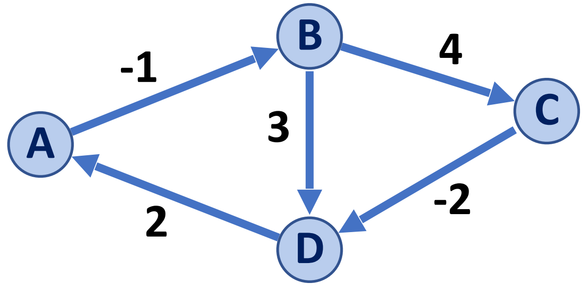


Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

Let us consider comparisons where w = A:

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	∞	∞	0



Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

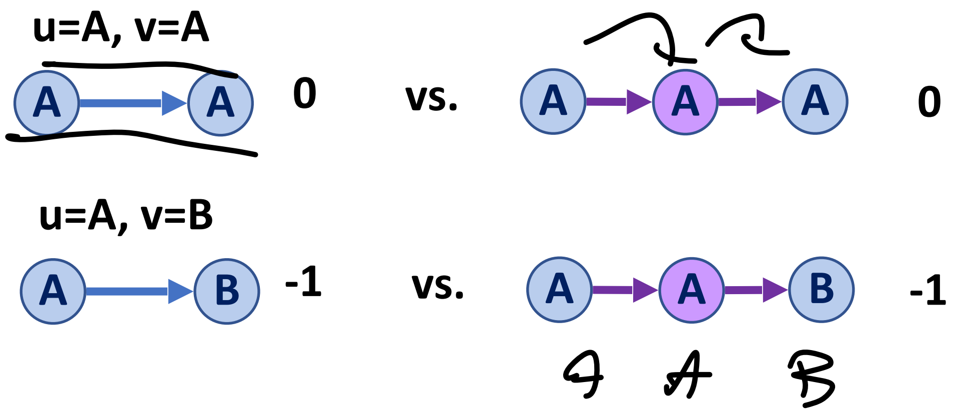
Let **w** be midpoint

Let **u** be start point

Let **v** be end point

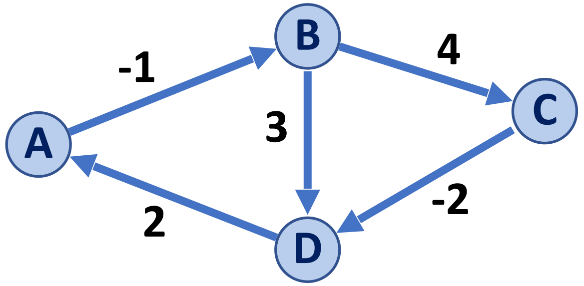
Is our distance shorter now?

Let us consider comparisons where w = A:



Don't waste time if u=w or v=w!

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	∞	∞	0



Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

Let **w** be midpoint

Let **u** be start point

Let **v** be end point

Is our distance shorter now?

Let us consider w = A (and u != w and v != w):

B → C 4 vs. B $\xrightarrow{\infty}$ A $\xrightarrow{\infty}$ C +∞

B → D 3 vs. B → A → D +∞

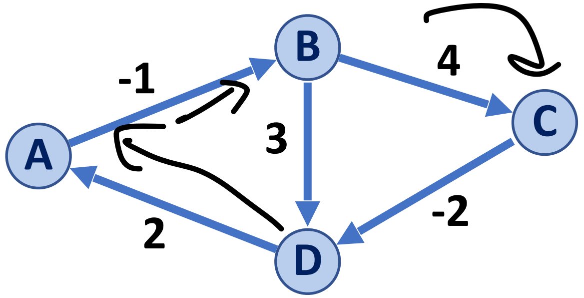
C → B +∞ vs. C → A → B +∞

C → D -2 vs. C → A → D +∞

D → B ~~+∞~~ vs. D $\xrightarrow{2}$ A $\xrightarrow{-1}$ B 1

D → C +∞ vs. D $\xrightarrow{2}$ A $\xrightarrow{\infty}$ C ∞

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	~~∞~~ 1	∞	0



Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

Let **w** be midpoint

Let **u** be start point

Let **v** be end point

Is our distance shorter now?

Let us consider w = A (and u != w and v != w):

B → C 4 vs. B → A → C +∞

B → D 3 vs. B → A → D +∞

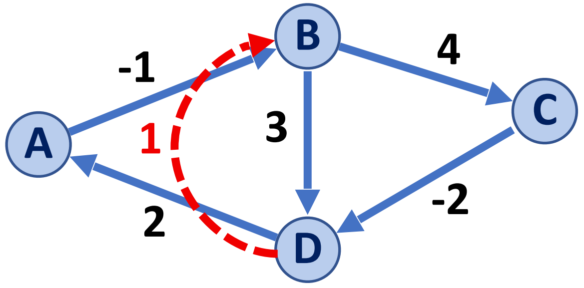
C → B +∞ vs. C → A → B +∞

C → D -2 vs. C → A → D +∞

D → B +∞ vs. D → A → B 1

D → C +∞ vs. D → A → C +∞

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	∞	0

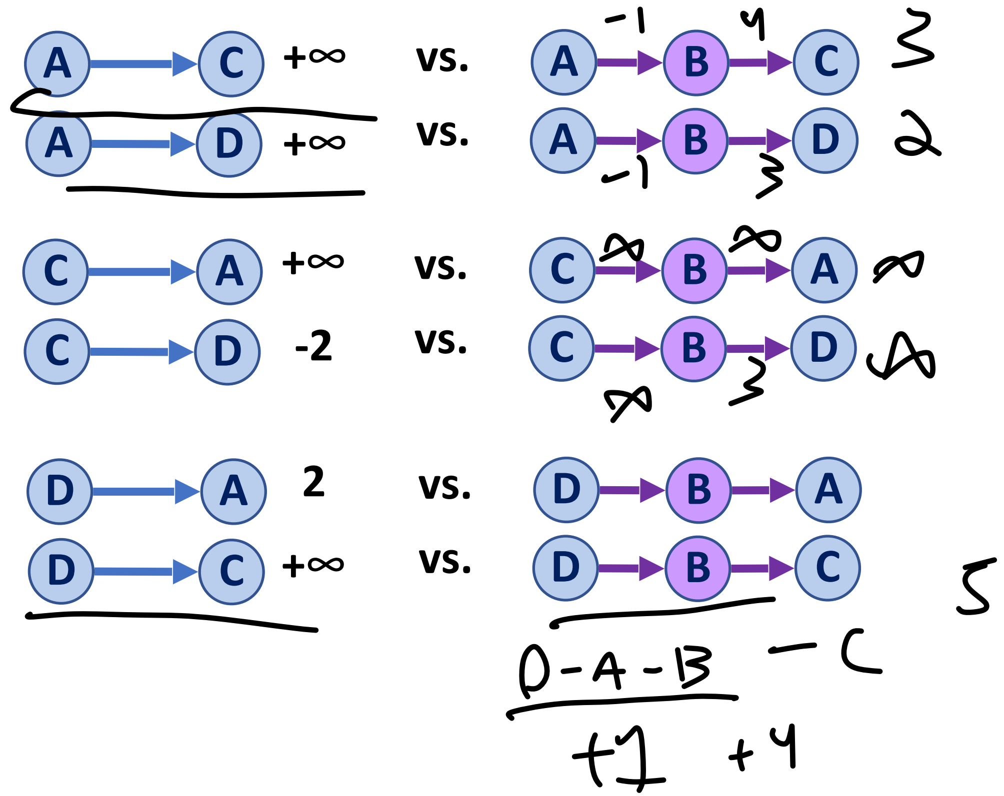


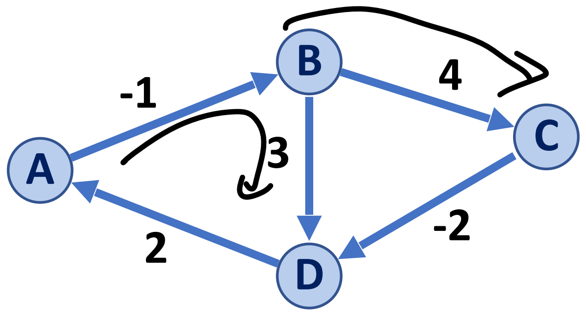
Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

	A	B	C	D
A	0	-1	~~∞~~ 3	~~∞~~ 2
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	~~∞~~ 5	0

Let us consider w = B (and u != w and v != w):





Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

	A	B	C	D
A	0	-1	3	~~2~~ 1
B	∞	0	4	~~3~~ 2
C	∞	∞	0	-2
D	2	1	5	0

Let us consider w = C (and u != w and v != w):

A → B -1 vs. A → C → B +∞

A → D 2 vs. A → C → D 1

3 -2

B → A +∞ vs. B → C → A +∞

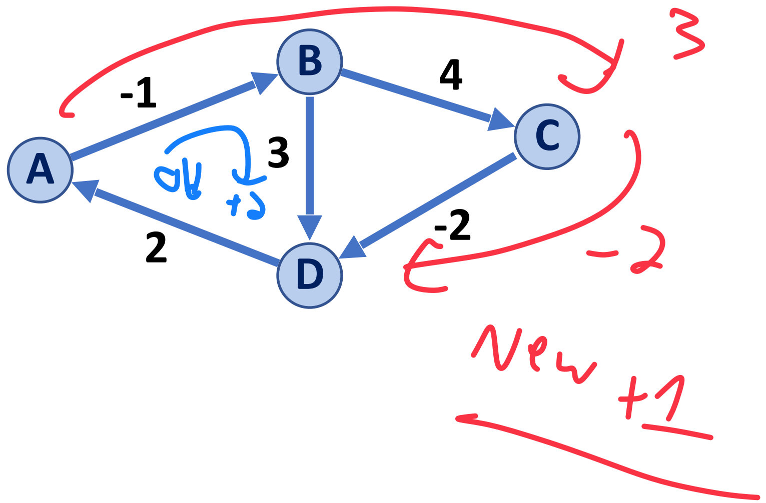
B → D 3 vs. B → C → D 2

3 4 -2

D → A 2 vs. D → C → A +∞

D → B 1 vs. D → C → B +∞

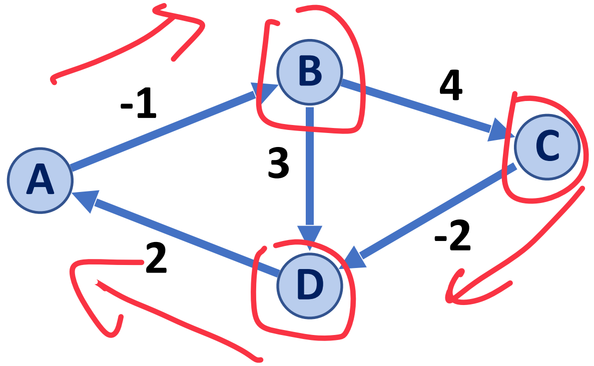
A - B - C - D



Floyd-Warshall Algorithm

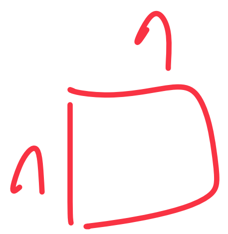
```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex u : G):
    foreach (Vertex v : G):
      foreach (Vertex w : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
```

	A	B	C	D
A	0	-1	3	1
B	5	0	4	2
C	0	-1	0	-2
D	2	1	5	0



Floyd-Warshall Algorithm

Running time? $O(n^3)$




```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex u : G):
    foreach (Vertex v : G):
      foreach (Vertex w : G):
        if d[u, v] > d[u, w] + d[w, v]:
          d[u, v] = d[u, w] + d[w, v]
```

n^3

Floyd-Warshall Algorithm

We aren't storing path information! Can we fix this?

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex w : G):
    foreach (Vertex u : G):
      foreach (Vertex v : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
```

Floyd-Warshall Algorithm

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
    s[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)
    s[u][v] = v

  foreach (Vertex w : G):
    foreach (Vertex u : G):
      foreach (Vertex v : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
          s[u, v] = s[u, w]
```

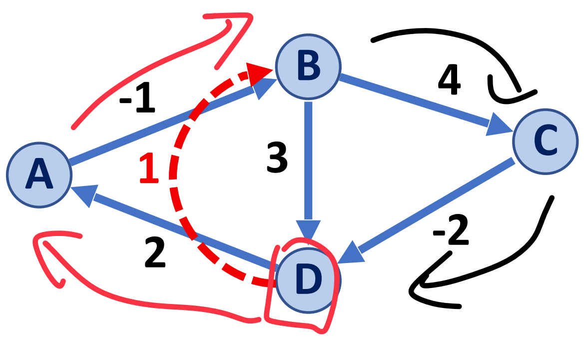
single edge tells us done

Opposite path storage

	A	B	C	D
A	0	-1	∞	~~2~~ 1
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	∞	0

if destination, done

	A	B	C	D
A		B		B
B			C	~~D~~ C
C				D
D	A	A		



A - B - D

A - B - C - D

We have only scratched the surface on graphs!

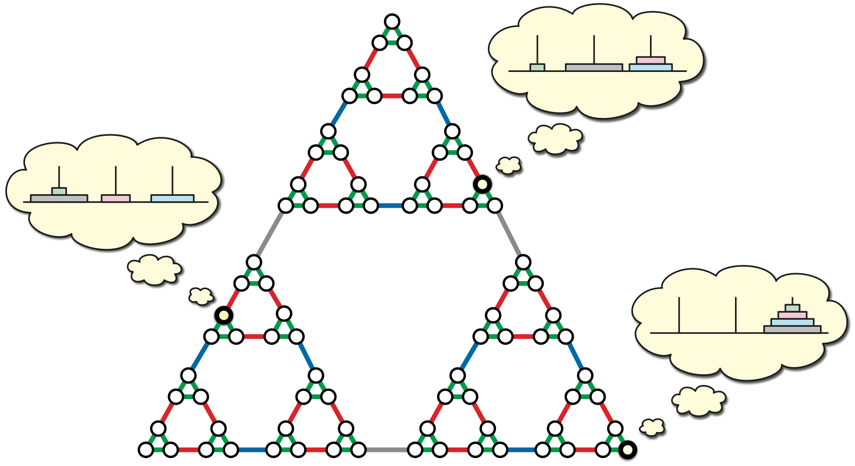


Image from Jeff Erickson Algorithms First Edition

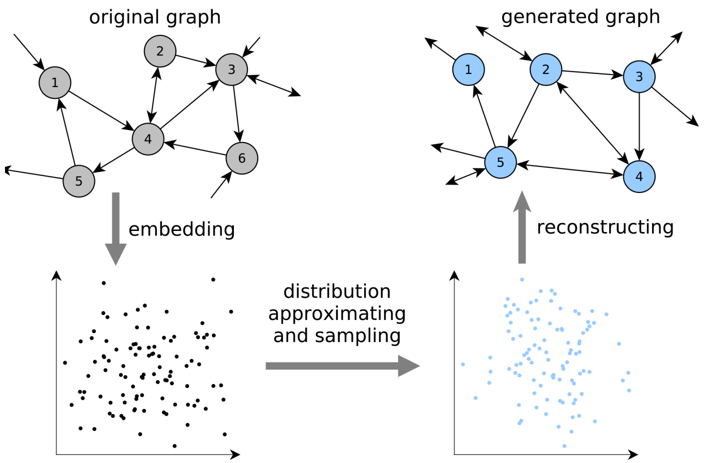
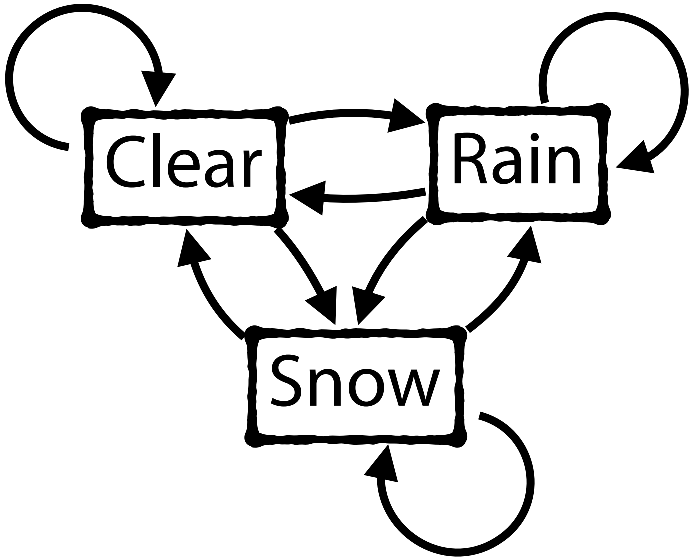


Image from Drobyshevskiy et al. **Random graph modeling: A survey of the concepts.** 2019

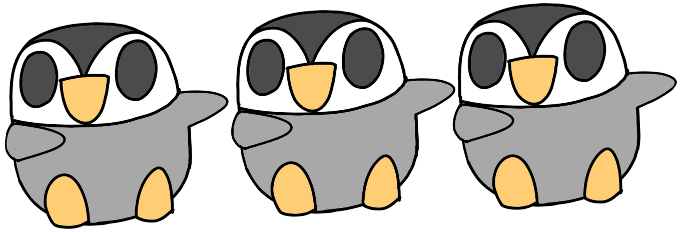


$$M = \begin{pmatrix} .5 & .3 & .2 \\ .5 & .4 & .1 \\ .2 & .1 & .7 \end{pmatrix}$$

Lets review what we’ve seen so far!

Lets review what we've seen so far!

Lists



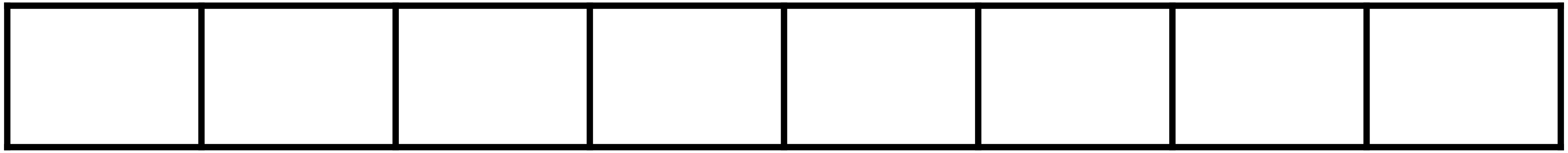
The not-so-secret underlying implementation for many things

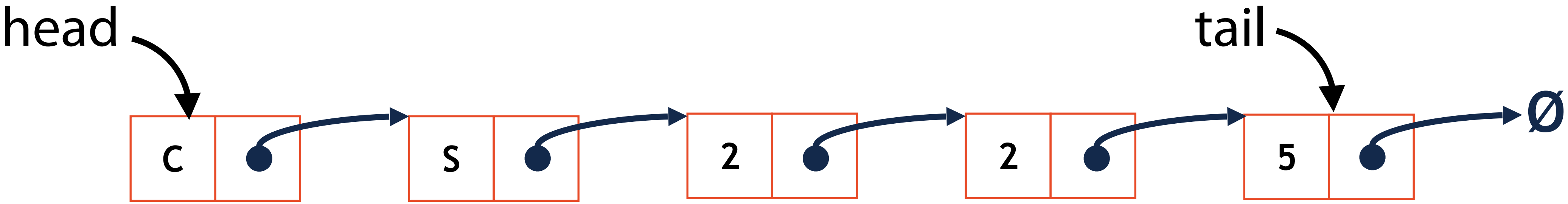
	Singly Linked List	Array
Look up **arbitrary** location	O(n)	O(1)
Insert after **given** element	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)	O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

Special Cases: insertFront insertBack (not full)

Stack and Queue

Taking advantage of special cases in lists / arrays

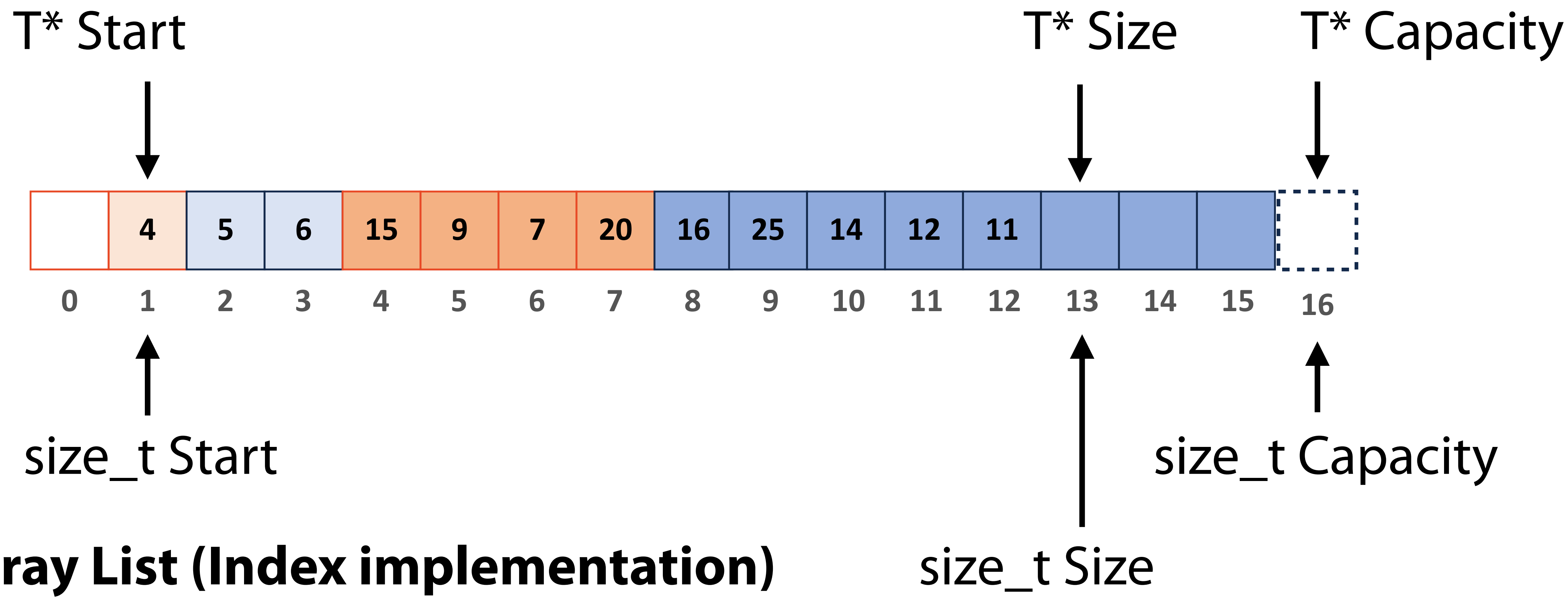




Heap

Taking advantage of special cases in lists / arrays

Array List (Pointer implementation)



Array List (Index implementation)

Disjoint Set Implementation

Taking advantage of array lookup operations

Store an UpTree as an array, canonical items store **height** / **size**

0 1 4   2 7   3 5 6

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

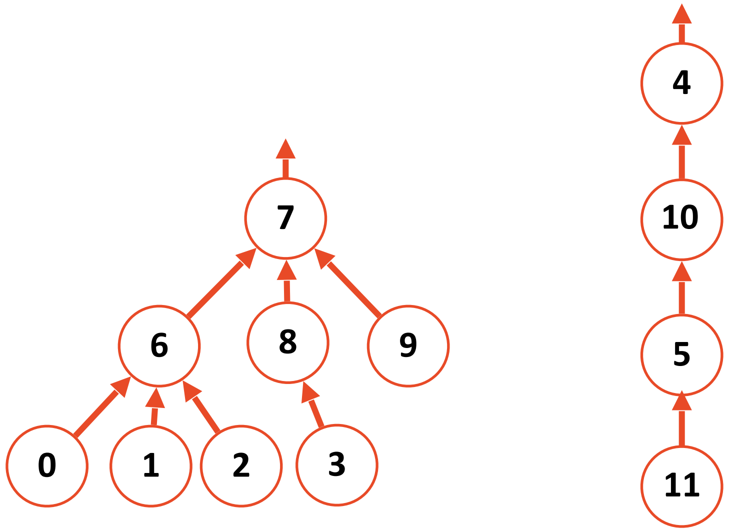
Find(k): Repeatedly look up values until **negative value**

Union(k_1, k_2): Update ***smaller*** canonical item to point to larger

Update value of remaining canonical item

Disjoint Sets – Smart Union

Minimizing number of O(1) operations



Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	**-4**	10	7	**4**	7	7	4	5

***Idea**: Keep the height of the tree as small as possible.*

Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	**7**	10	7	**-12**	7	7	4	5

***Idea**: Minimize the number of nodes that increase in height*

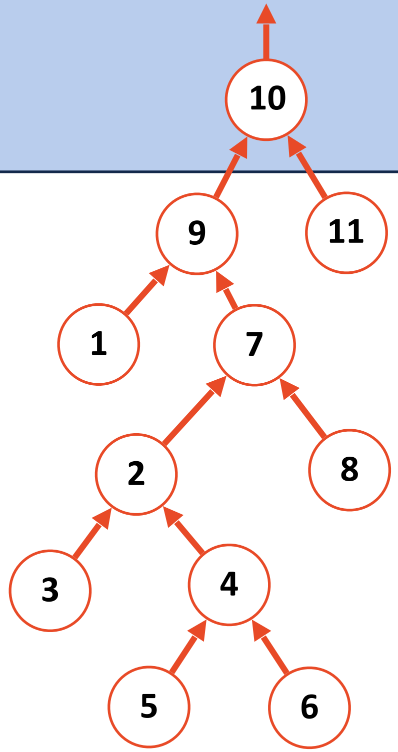
Both guarantee the height of the tree is: O(log n).

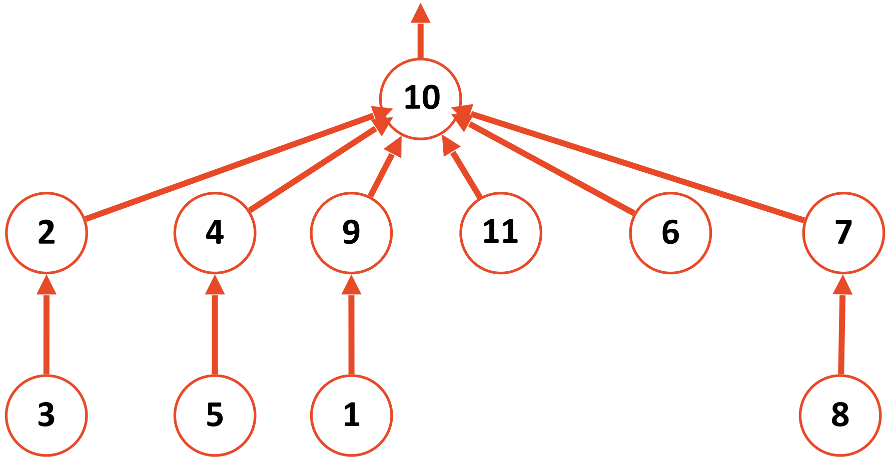
Disjoint Sets Path Compression

Find(6)

Minimizing number of O(1) operations

```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else {
    int root = find( s[i] );
    s[i] = root;
    return root;
  }
}
```



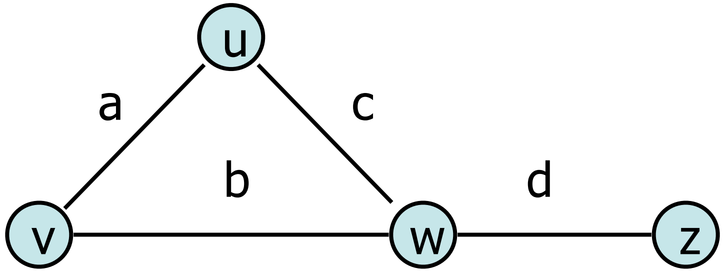


Alternative Not-Actually-A-Proof

Unproven Claim: A disjoint set implemented with smart union and path compression with **m** find calls and **n** items has a worst case running time of **inverse Ackerman.** $\left[O(m \ \alpha(n))\right]$

This grows *very* slowly to the point of being treated a constant in CS.

Graph Implementation: Edge List |V|= n,|E|= m



u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

Literally just arrays

O(1)*

insertVertex(K key):

insertEdge(Vertex v1, Vertex v2, K key):

O(m)

removeVertex(Vertex v):

removeEdge(Vertex v1, Vertex v2, K key):

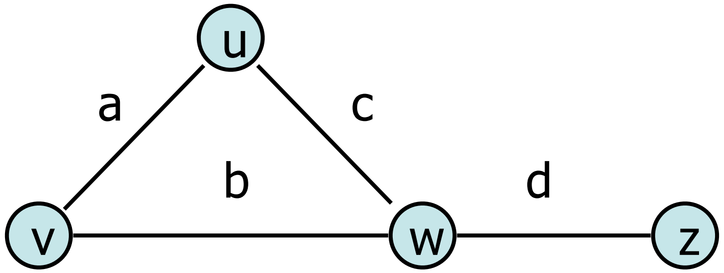
incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m

Literally just a matrix of arrays



u
v
w
z

	u	v	w	z
u	-	a	c	0
v		-	b	0
w			-	d
z				-

O(1)

insertEdge(Vertex v1, Vertex v2, K key):

removeEdge(Vertex v1, Vertex v2, K key):

areAdjacent(Vertex v1, Vertex v2):

O(n)

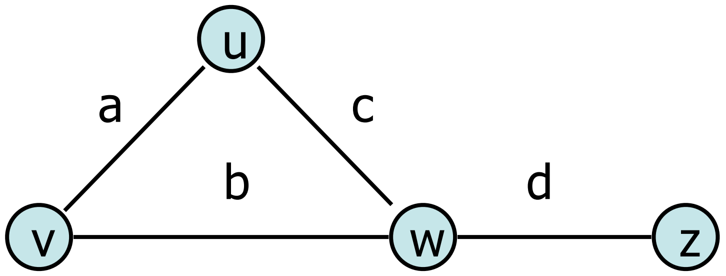
incidentEdges(Vertex v):

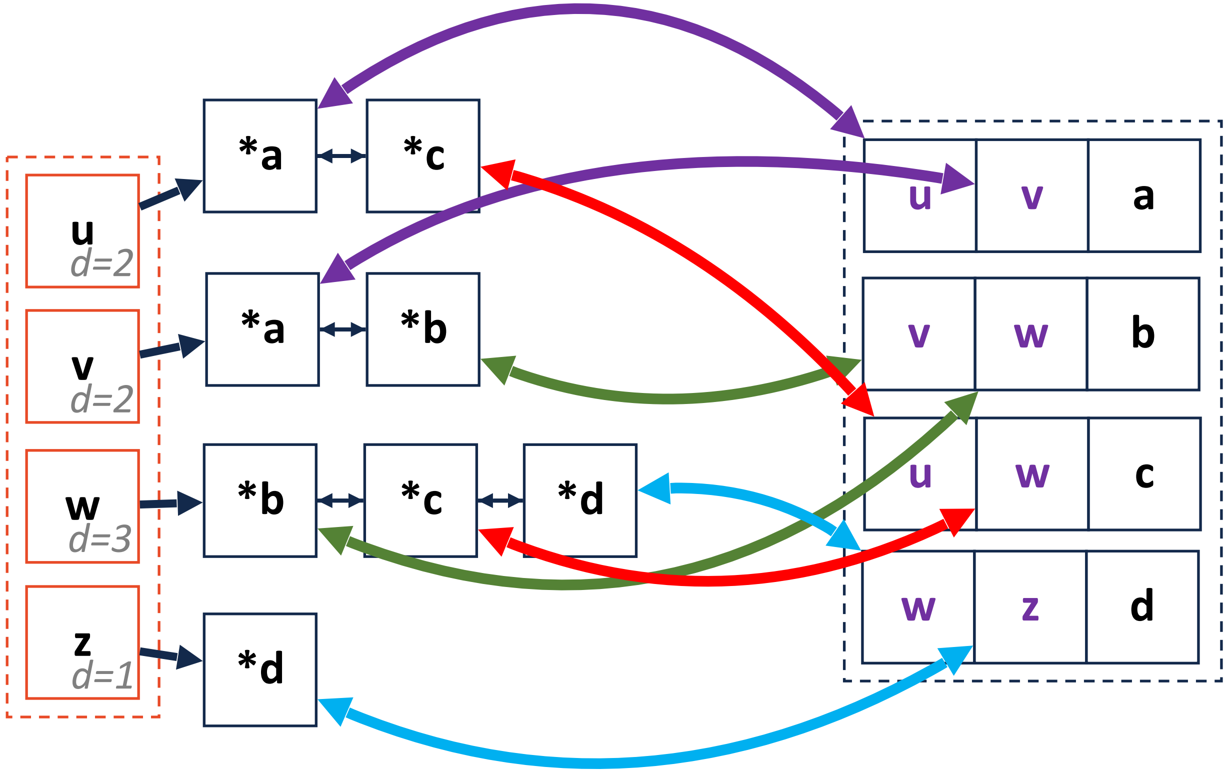
O(n)—O(n^2)

insertVertex(K key):

removeVertex(Vertex v):

Adjacency List





Technically linked lists I guess

Expressed as O(f)	Adjacency List
Space	n+m
insertVertex(v)	1*
removeVertex(v)	deg(v)
insertEdge(u, v)	1*
removeEdge(u, v)	min(deg(u), deg(v))
incidentEdges(v)	deg(v)
areAdjacent(u, v)	min(deg(u), deg(v))

… And thats most of exam 4

Randomized Algorithms

A **randomized algorithm** is one which uses a source of randomness somewhere in its implementation.

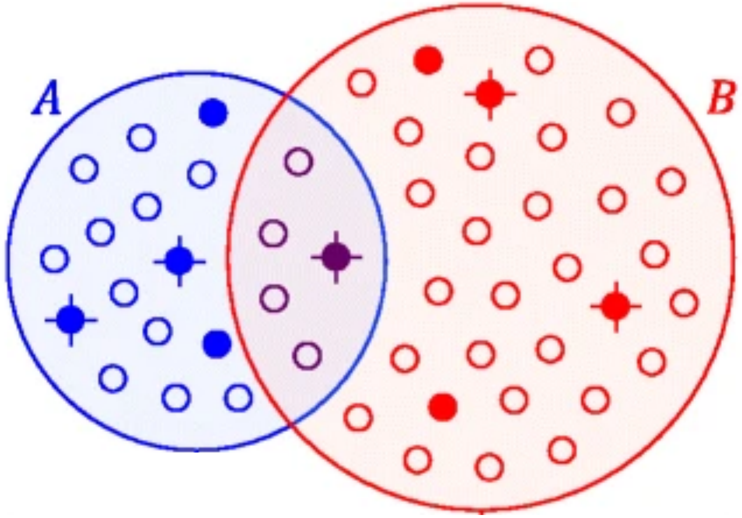
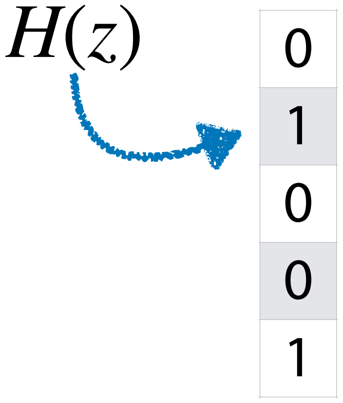
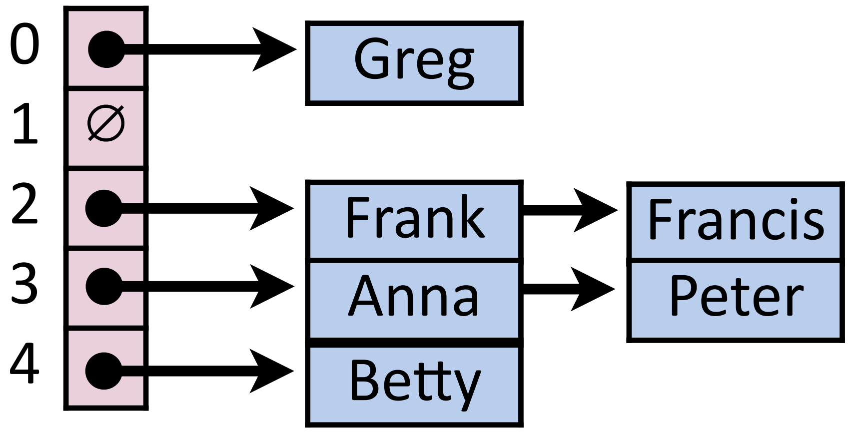


Figure from Ondov et al 2016





$H(x)$	0	2	1	0	0	4	0	2	0	6
$H(y)$	1	0	2	3	1	0	3	4	0	1
$H(z)$	2	1	0	2	0	1	0	0	7	2

A faulty list

Imagine you have a list ADT implementation ***except***…

Every time you called **insert**, it would fail 50% of the time.

Quick Primes with Fermat's Primality Test

If p is prime and a is not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$

But… ***sometimes*** if n is composite and $a^{n-1} \equiv 1 \pmod{n}$

Probabilistic Accuracy: Fermat primality test

	$a^{p-1} \equiv 1 \pmod{p}$	$a^{p-1} \not\equiv 1 \pmod{p}$
p is prime		
p is not prime		

Probabilistic Accuracy: Fermat primality test

Let's assume $\alpha = .5$

First trial: $a = a_0$ and prime test returns 'prime!'

Second trial: $a = a_1$ and prime test returns 'prime!'

Third trial: $a = a_2$ and prime test returns 'not prime!'

Is our number prime?

What is our **false positive** probability? Our **false negative** probability?

Probabilistic Accuracy: Fermat primality test

Summary: Randomized algorithms can also have fixed (or bounded) runtimes at the cost of probabilistic accuracy.

Randomness:

Assumptions: