

Data Structures and Algorithms

All Paths Shortest Path (Plus Review)

CS 225

November 7, 2025

Brad Solomon



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

Learning Objectives

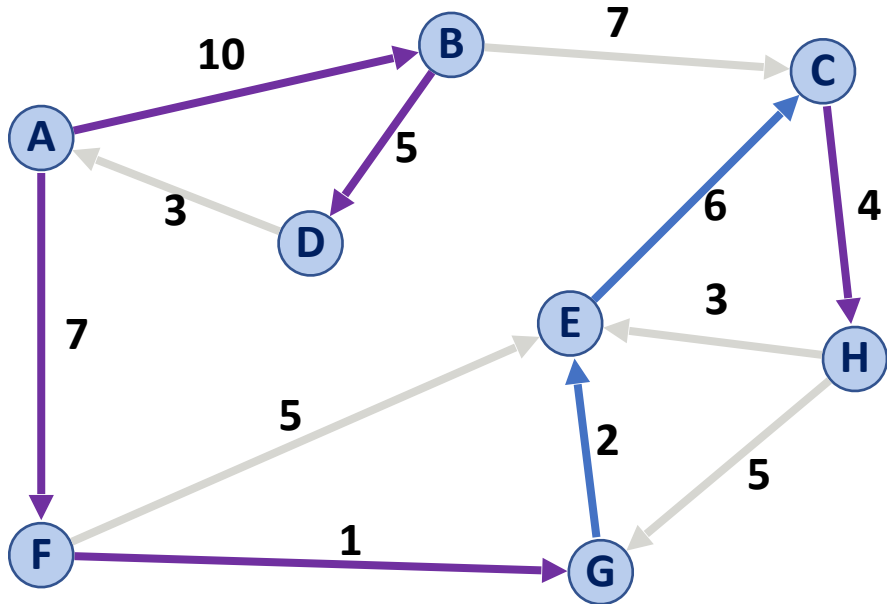
Introduce and discuss All-Paths Shortest Path

Review deterministic data structures in CS

An opportunity for Q&A for exam 4

Foreshadowing probabilistic data structures

Dijkstra's Algorithm (SSSP)



```
DijkstraSSSP(G, s):
```

```
6   foreach (Vertex v : G.vertices()):
7       d[v] = +inf
8       p[v] = NULL
9   d[s] = 0

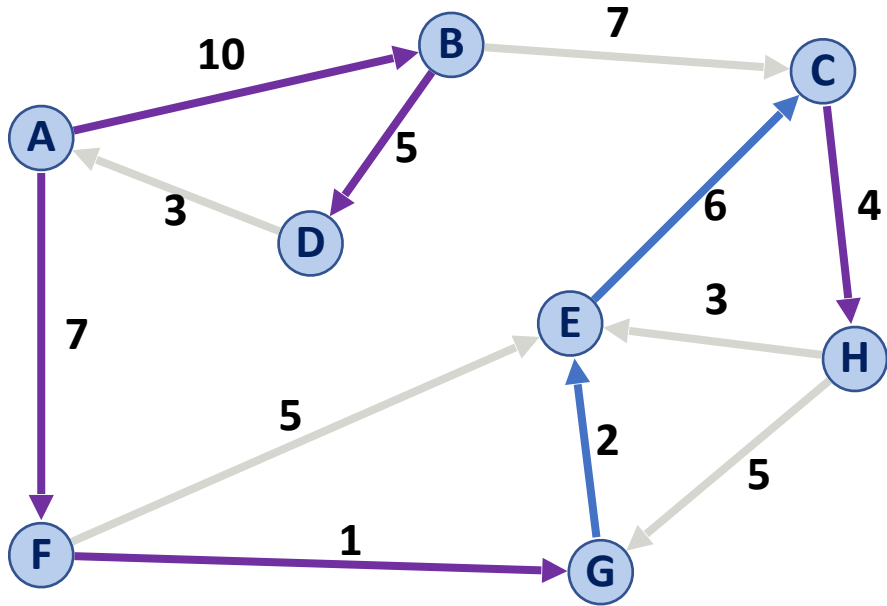
10  PriorityQueue Q // min distance, defined by d[v]
11  Q.buildHeap(G.vertices())
12  Graph T          // "labeled set"

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

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?



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)

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):
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] = u
22
23   return T
```

Floyd-Warshall Algorithm

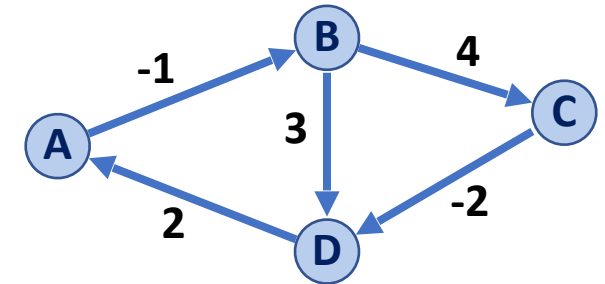
Floyd-Warshall's Algorithm is an alternative to Dijkstra in the presence of **negative-weight edges (not negative weight cycles)**.

```
1 FloydWarshall(G):
2   Let d be a adj. matrix initialized to +inf
3   foreach (Vertex v : G):
4     d[v][v] = 0
5   foreach (Edge (u, v) : G):
6     d[u][v] = cost(u, v)
7
8   foreach (Vertex u : G):
9     foreach (Vertex v : G):
10      foreach (Vertex w : G):
11        if (d[u, v] > d[u, w] + d[w, v])
12          d[u, v] = d[u, w] + d[w, v]
```

Floyd-Warshall Algorithm

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

	A	B	C	D
A				
B				
C				
D				

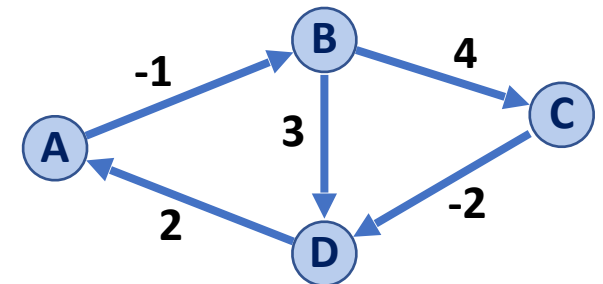


Floyd-Warshall Algorithm

```
8  foreach (Vertex w : G):  
9      foreach (Vertex u : G):  
10         foreach (Vertex v : G):  
11             if (d[u, v] > d[u, w] + d[w, v])  
12                 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

```
8  foreach (Vertex w : G) :  
9    foreach (Vertex u : G) :  
10     foreach (Vertex v : G) :  
11       if (d[u, v] > d[u, w] + d[w, v])  
12         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 $w = A$:

$u=A, v=A$

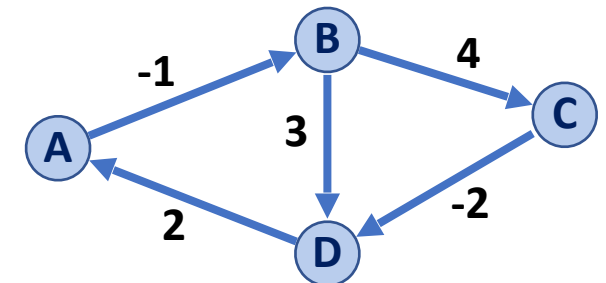


$u=A, v=B$



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

```

8  foreach (Vertex w : G) :
9    foreach (Vertex u : G) :
10   foreach (Vertex v : G) :
11     if (d[u, v] > d[u, w] + d[w, v])
12       d[u, v] = d[u, w] + d[w, v]
    
```

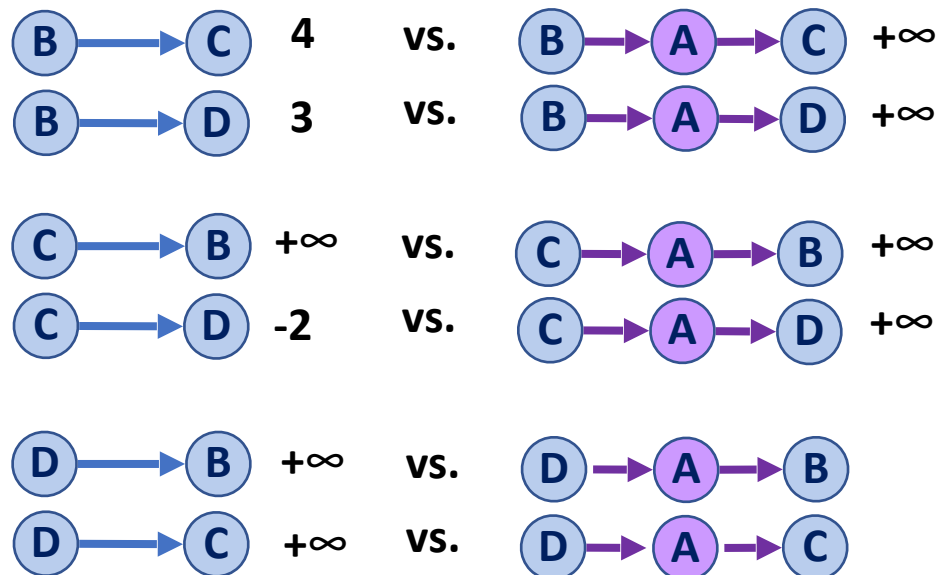
Let **w** be midpoint

Let **u** be start point

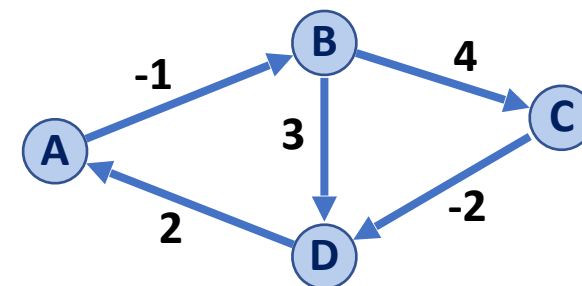
Let **v** be end point

Is our distance shorter now?

Let us consider $w = A$ (and $u \neq w$ and $v \neq w$):



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



Floyd-Warshall Algorithm

```
8  foreach (Vertex w : G) :
9    foreach (Vertex u : G) :
10     foreach (Vertex v : G) :
11       if (d[u, v] > d[u, w] + d[w, v])
12         d[u, v] = d[u, w] + d[w, v]
```

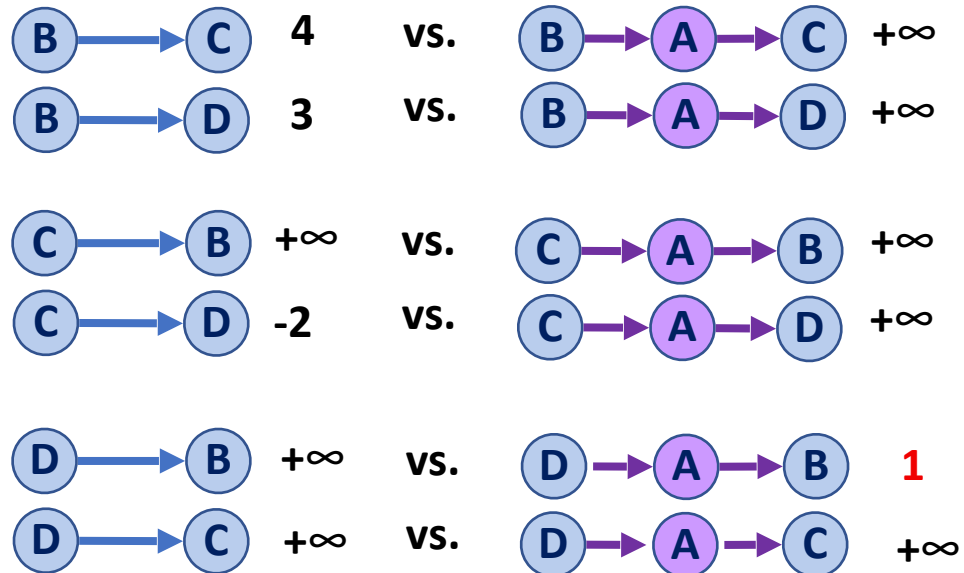
Let **w** be midpoint

Let **u** be start point

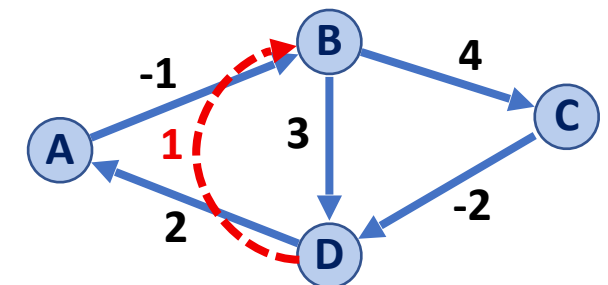
Let **v** be end point

Is our distance shorter now?

Let us consider $w = A$ (and $u \neq w$ and $v \neq w$):



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

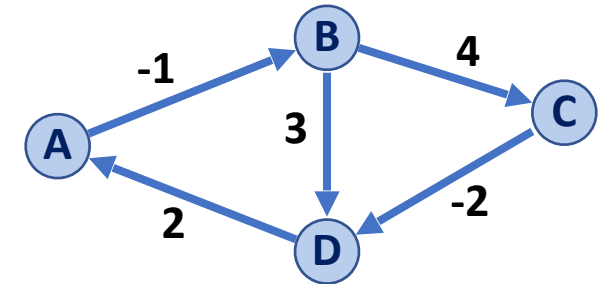
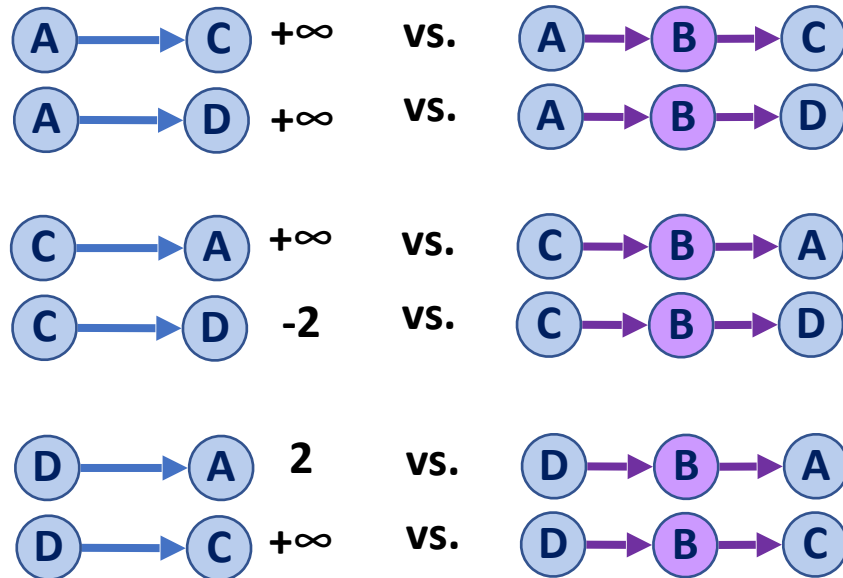


Floyd-Warshall Algorithm

```
8  foreach (Vertex w : G) :  
9    foreach (Vertex u : G) :  
10     foreach (Vertex v : G) :  
11       if (d[u, v] > d[u, w] + d[w, v])  
12         d[u, v] = d[u, w] + d[w, v]
```

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

Let us consider $w = B$ (and $u \neq w$ and $v \neq w$):



Floyd-Warshall Algorithm

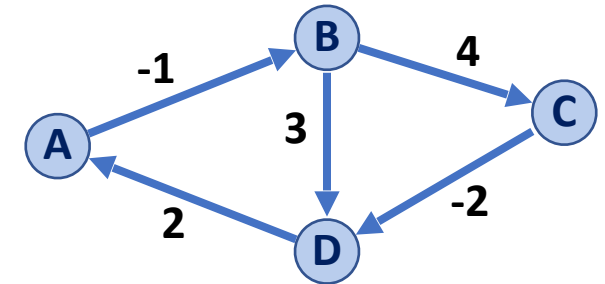
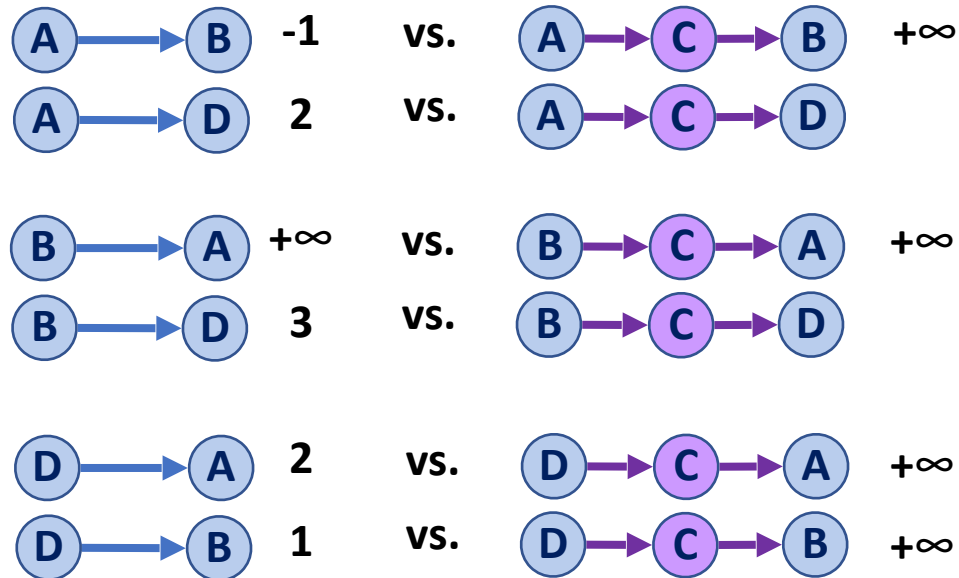
```

8  foreach (Vertex w : G) :
9    foreach (Vertex u : G) :
10   foreach (Vertex v : G) :
11     if (d[u, v] > d[u, w] + d[w, v])
12       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 $w = C$ (and $u \neq w$ and $v \neq w$):

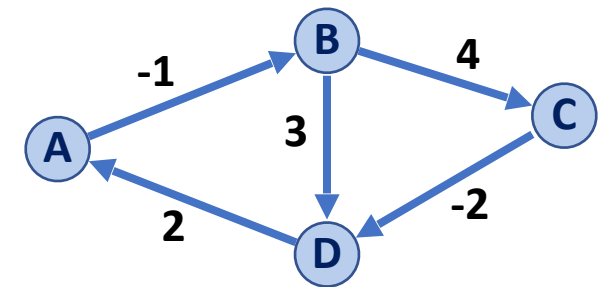


Floyd-Warshall Algorithm



```
1 FloydWarshall(G):
2   Let d be a adj. matrix initialized to +inf
3   foreach (Vertex v : G):
4     d[v][v] = 0
5   foreach (Edge (u, v) : G):
6     d[u][v] = cost(u, v)
7
8   foreach (Vertex u : G):
9     foreach (Vertex v : G):
10      foreach (Vertex w : G):
11        if (d[u, v] > d[u, w] + d[w, v])
12          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?

```
FloydWarshall(G):  
6   Let d be a adj. matrix initialized to +inf  
7   foreach (Vertex v : G):  
8       d[v][v] = 0  
9   foreach (Edge (u, v) : G):  
10      d[u][v] = cost(u, v)  
11  
12  foreach (Vertex u : G):  
13      foreach (Vertex v : G):  
14          foreach (Vertex w : G):  
15              if d[u, v] > d[u, w] + d[w, v]:  
16                  d[u, v] = d[u, w] + d[w, v]
```

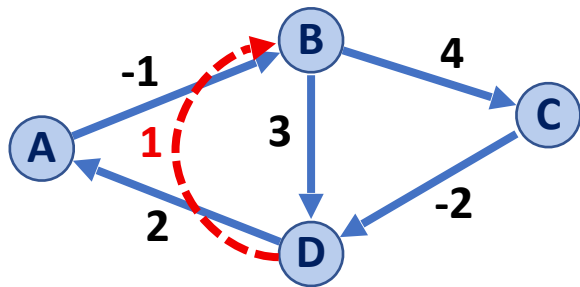
Floyd-Warshall Algorithm

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

```
FloydWarshall(G):  
  6   Let d be a adj. matrix initialized to +inf  
  7   foreach (Vertex v : G):  
  8     d[v][v] = 0  
  9   foreach (Edge (u, v) : G):  
 10     d[u][v] = cost(u, v)  
 11  
 12   foreach (Vertex w : G):  
 13     foreach (Vertex u : G):  
 14       foreach (Vertex v : G):  
 15         if (d[u, v] > d[u, w] + d[w, v])  
 16           d[u, v] = d[u, w] + d[w, v]
```


Floyd-Warshall Algorithm

```
FloydWarshall(G):  
6   Let d be a adj. matrix initialized to +inf  
7   foreach (Vertex v : G):  
8       d[v][v] = 0  
9       s[v][v] = 0  
10  foreach (Edge (u, v) : G):  
11      d[u][v] = cost(u, v)  
12      s[u][v] = v  
13  
14  foreach (Vertex w : G):  
15      foreach (Vertex u : G):  
16          foreach (Vertex v : G):  
17              if (d[u, v] > d[u, w] + d[w, v])  
18                  d[u, v] = d[u, w] + d[w, v]  
19                  s[u, v] = s[u, w]
```



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

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

We have only scratched the surface on graphs!

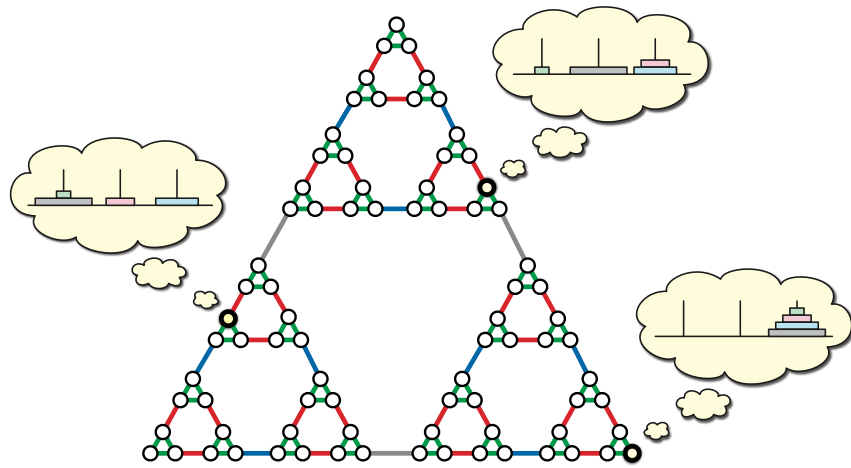


Image from Jeff Erickson Algorithms First Edition

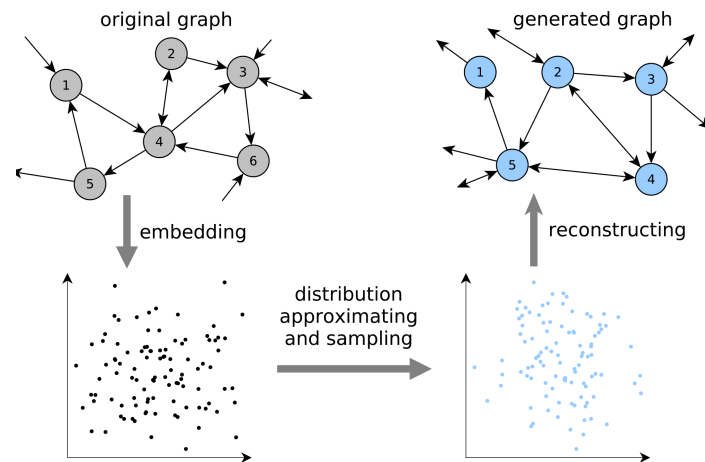
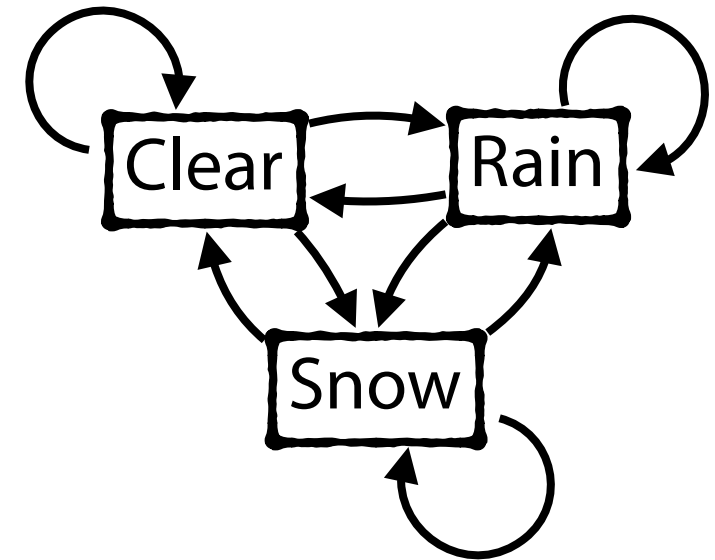


Image from Drobyshevskiy et al. **Random graph modeling: A survey of the concepts.** 2019



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



Lets review what we've seen so far!

Lets review what we've seen so far!



Lists



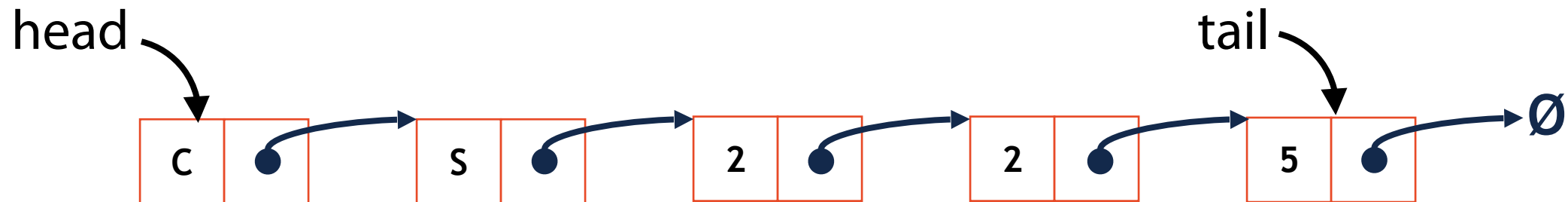
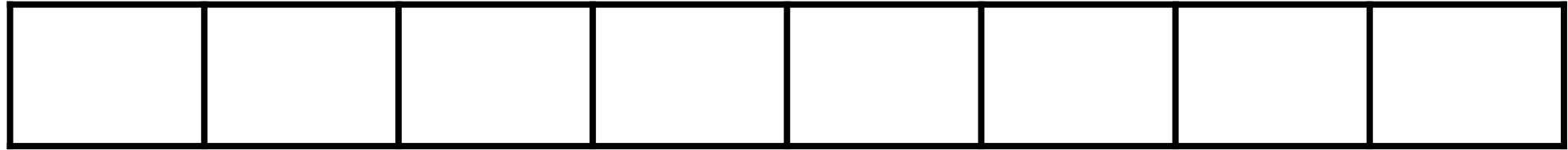
The not-so-secret underlying implementation for many things

	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:

Stack and Queue

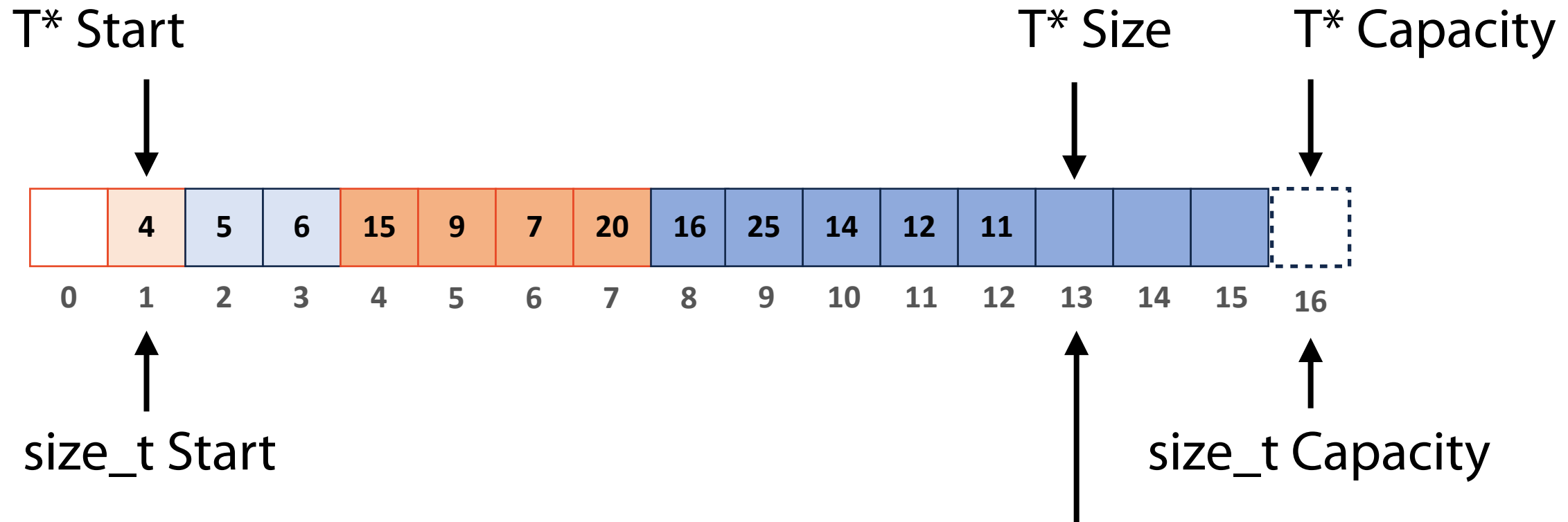
Taking advantage of special cases in lists / arrays



Heap

Taking advantage of special cases in lists / arrays

Array List (Pointer implementation)



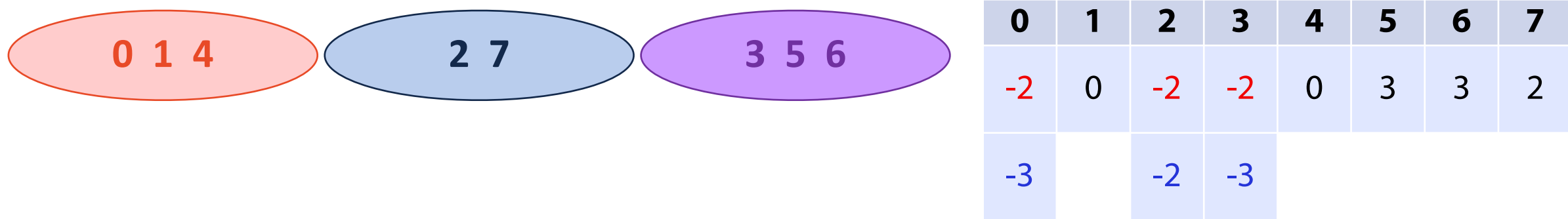
Array List (Index implementation)

size_t Size

Disjoint Set Implementation

Taking advantage of array lookup operations

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

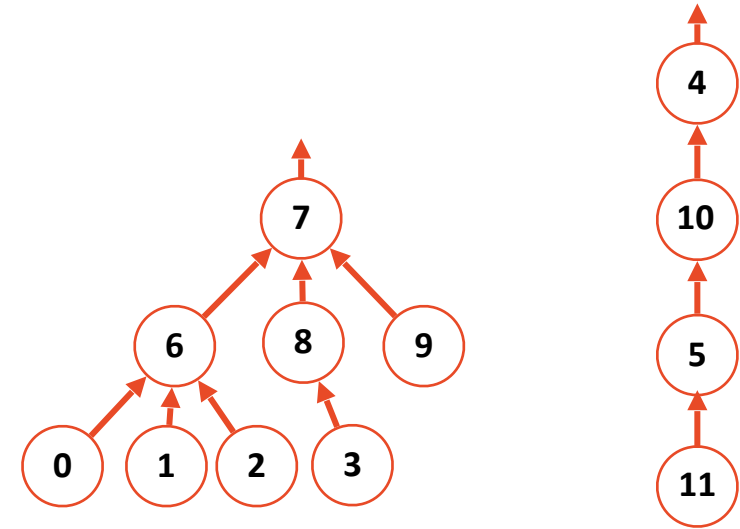


Find(k): Repeatedly look up values until **negative value**

Union(k₁, k₂): 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

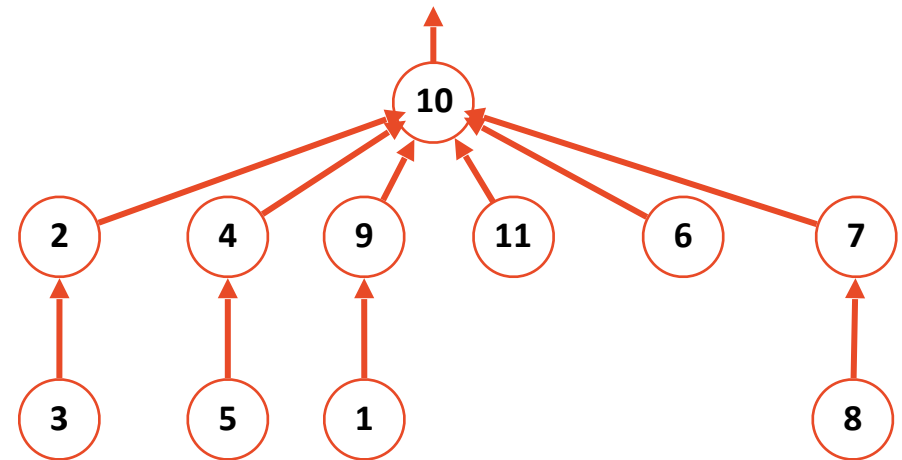
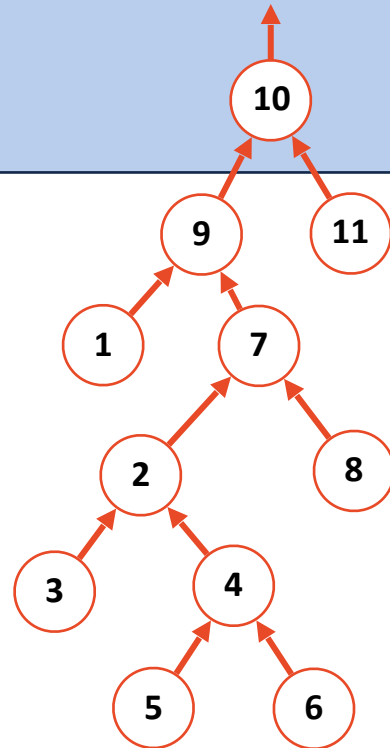
Both guarantee the height of the tree is: **$O(\log n)$** .

Disjoint Sets Path Compression

Find(6)

Minimizing number of $O(1)$ operations

```
1 int DisjointSets::find(int i) {  
2   if ( s[i] < 0 ) { return i; }  
3   else {  
4     int root = find( s[i] );  
5     s[i] = root;  
6     return root;  
7   }  
8 }
```

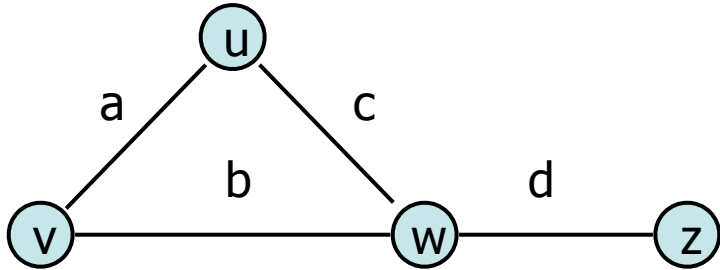


Alternative Not-Actually-A-Proof

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**. $[O(m \alpha(n))]$

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

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

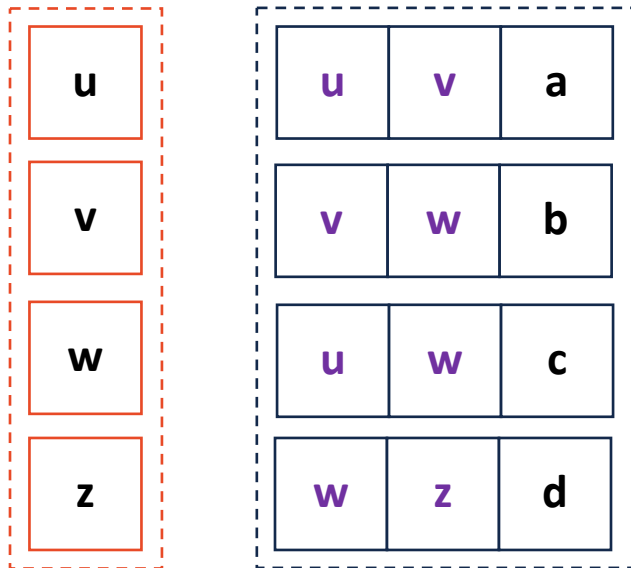


Literally just arrays

$O(1)^*$

insertVertex(K key):

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



$O(m)$

removeVertex(Vertex v):

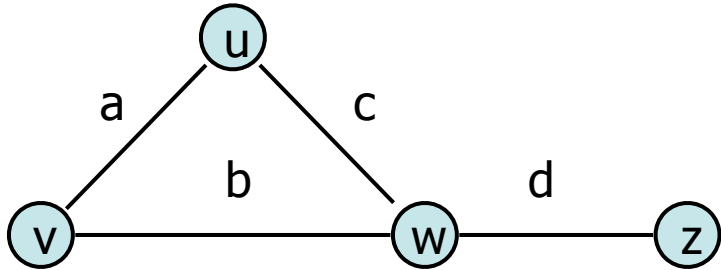
removeEdge(Vertex v1, Vertex v2, K key):

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



Literally just a matrix of arrays

$O(1)$

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

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

areAdjacent(Vertex v1, Vertex v2):

$O(n)$

incidentEdges(Vertex v):

$O(n) \text{---} O(n^2)$

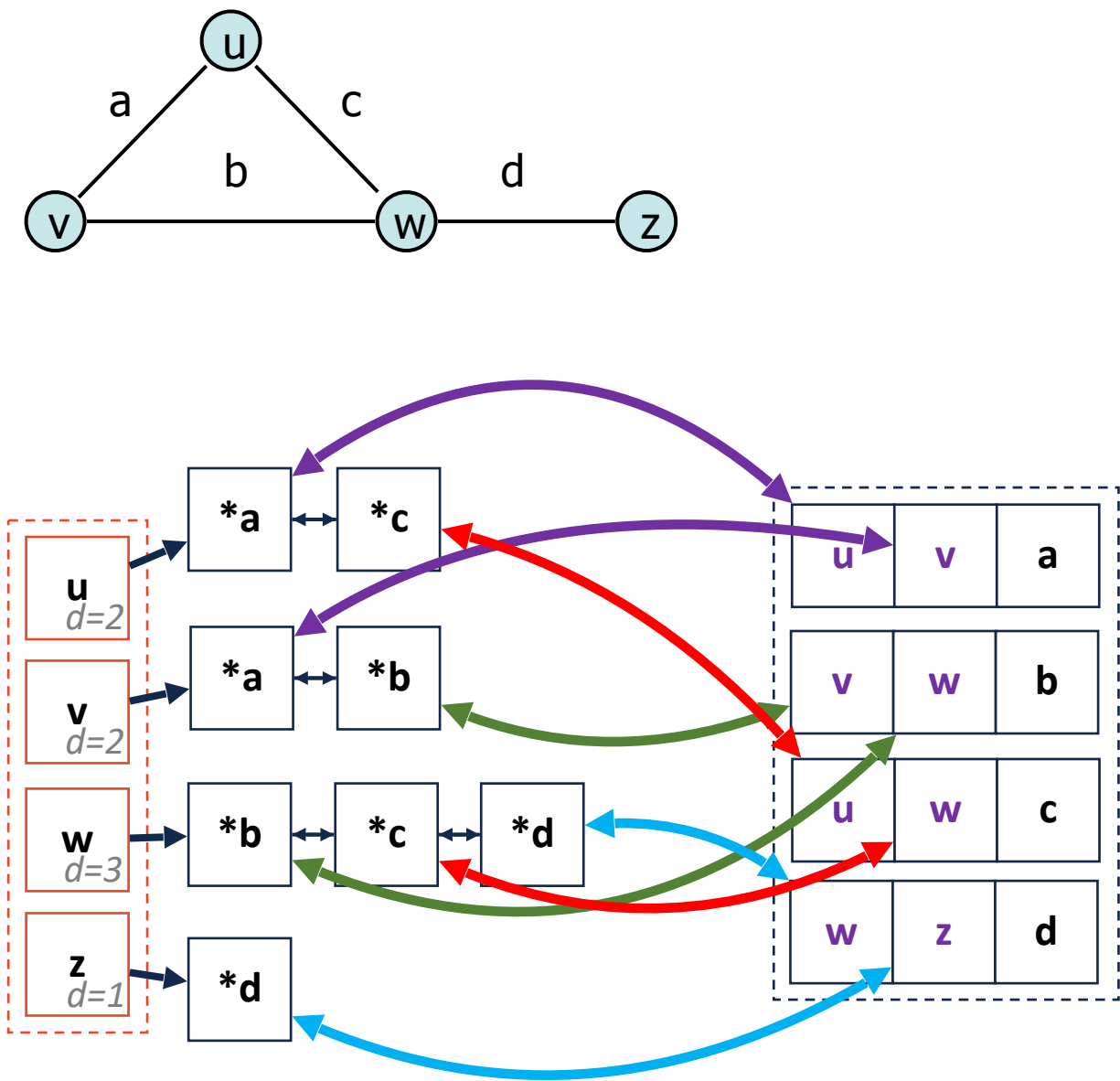
insertVertex(K key):

removeVertex(Vertex v):

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

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.

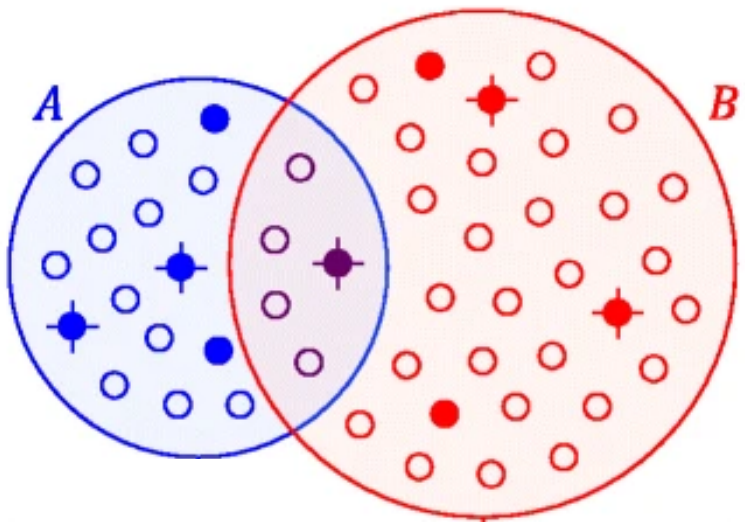
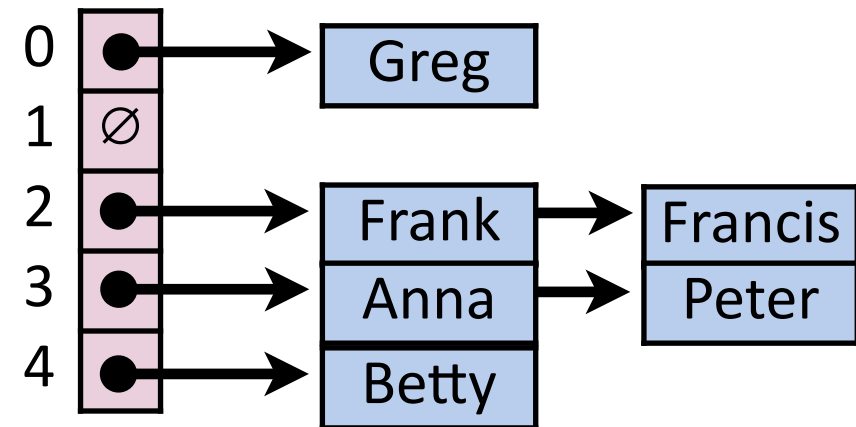
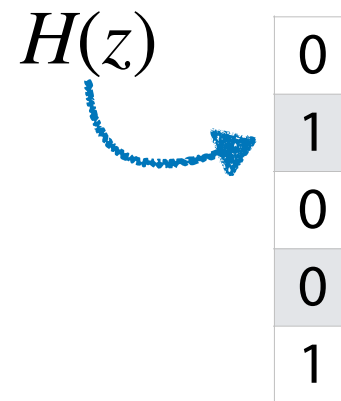


Figure from Ondov et al 2016



$H(x)$	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: