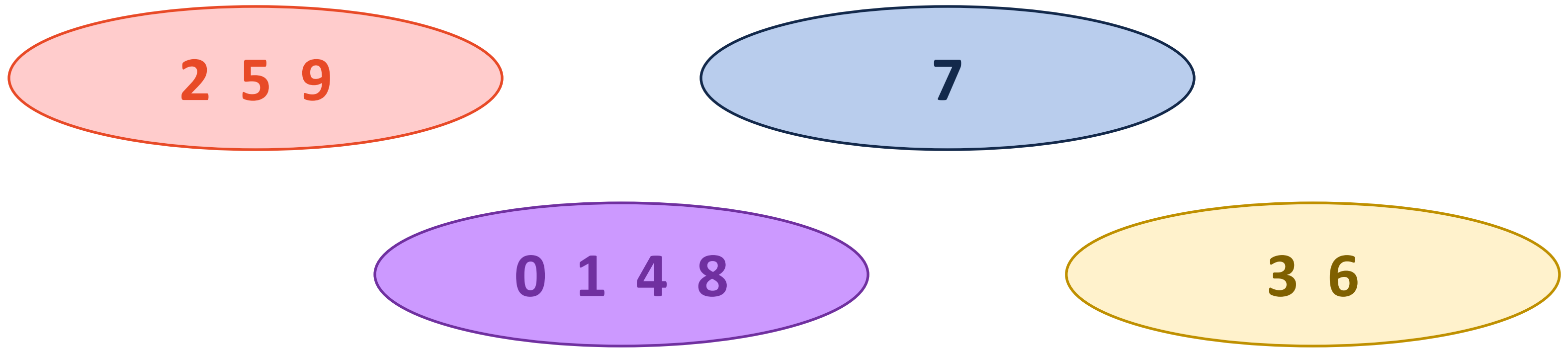
CS225

Final Exam Review

By Mariam Vardishvili,
thanks to: Milica Hadzi-Tanovic
Wade Fagen-Ulmschneider

Disjoint Sets



Key Ideas:

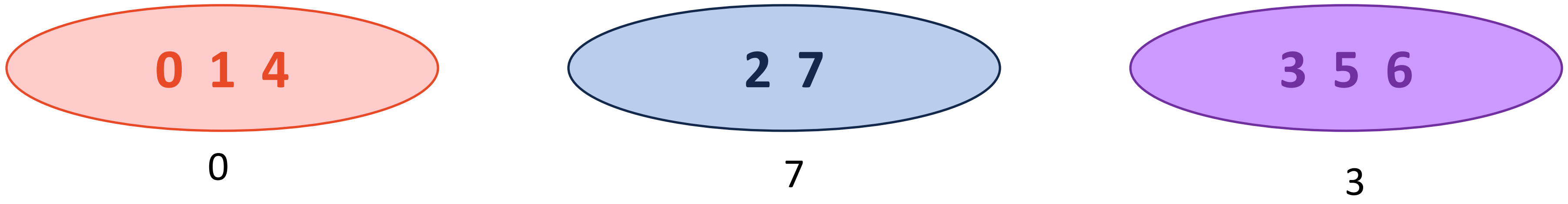
- Each element exists in exactly one set.
- Every set is an equitant representation.
  - Mathematically: $4 \in [0]_R \rightarrow 8 \in [0]_R$
  - Programmatically: find(4) == find(8)

Disjoint Sets ADT

- Maintain a collection S = {s_0, s_1, … s_k}
- Each set has a representative member.
- API:

```
void makeSet(const T & t);
void union(const T & k1, const T & k2);
T & find(const T & k);
```

Implementation #1



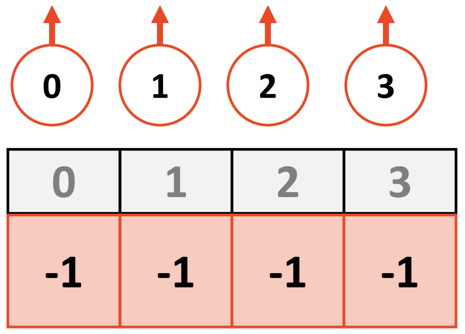
0	1	2	3	4	5	6	7
0	**0**	**7**	**3**	**0**	**3**	**3**	**7**

Find(k): O(1)

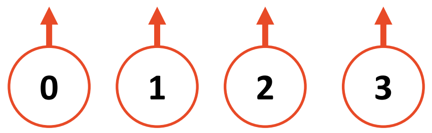
Union(k1, k2): O(n)

Implementation #2

- We will continue to use an array where the index is the key
- The value of the array is:
  - **-1,** if we have found the representative element
  - **The index of the parent**, if we haven't found the rep. element
- We will call theses **UpTrees**:



UpTrees



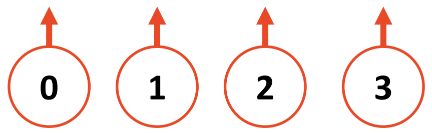
0	1	2	3
-1	-1	-1	-1

0	1	2	3

0	1	2	3

0	1	2	3

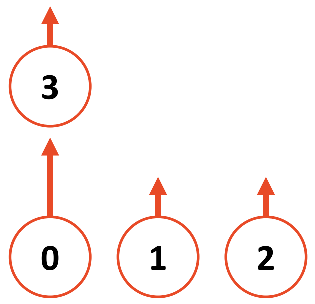
UpTrees



0	1	2	3
-1	-1	-1	-1

0	1	2	3

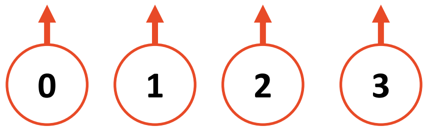
Union (3,0)



0	1	2	3
3	-1	-1	-1

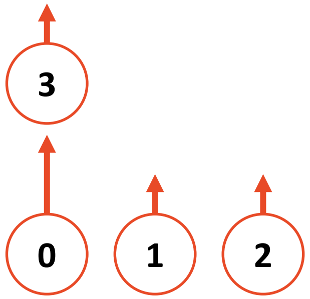
0	1	2	3

UpTrees



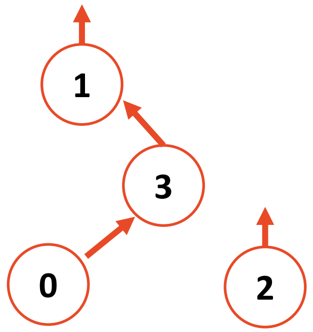
0	1	2	3
-1	-1	-1	-1

Union (3,0)



0	1	2	3
3	-1	-1	-1

Union (1,3)

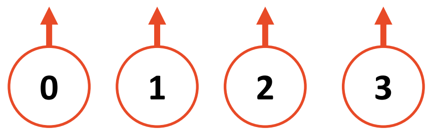


0	1	2	3
3	-1	-1	1

Find(0) = 1

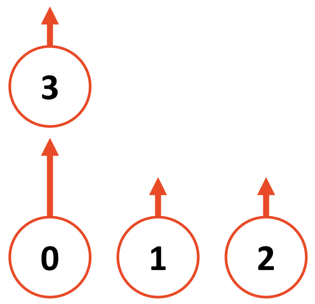
0	1	2	3

UpTrees



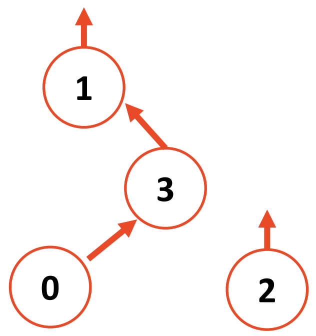
0	1	2	3
-1	-1	-1	-1

Union (3,0)



0	1	2	3
3	-1	-1	-1

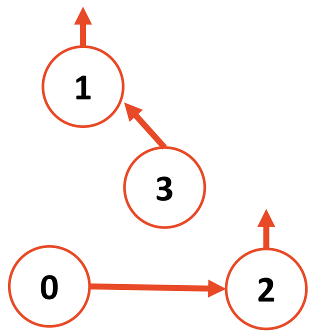
Union (1,3)



0	1	2	3
3	-1	-1	1

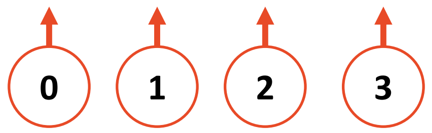
Find(0) = 1

Union (2,0)



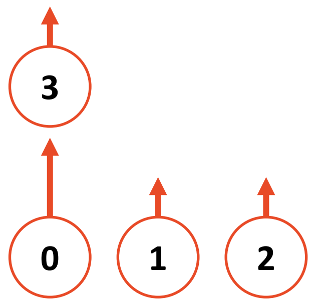
0	1	2	3

UpTrees



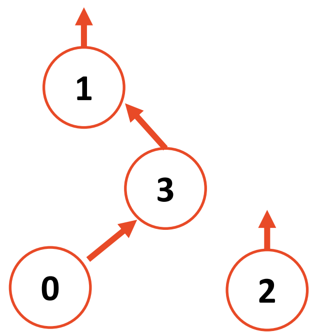
0	1	2	3
-1	-1	-1	-1

Union (3,0)



0	1	2	3
3	-1	-1	-1

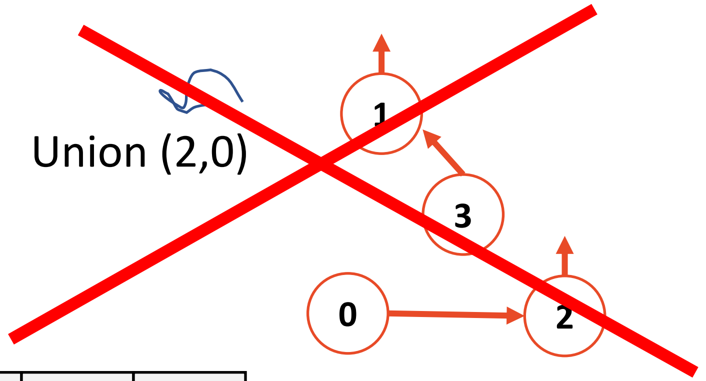
Union (1,3)



0	1	2	3
3	-1	-1	1

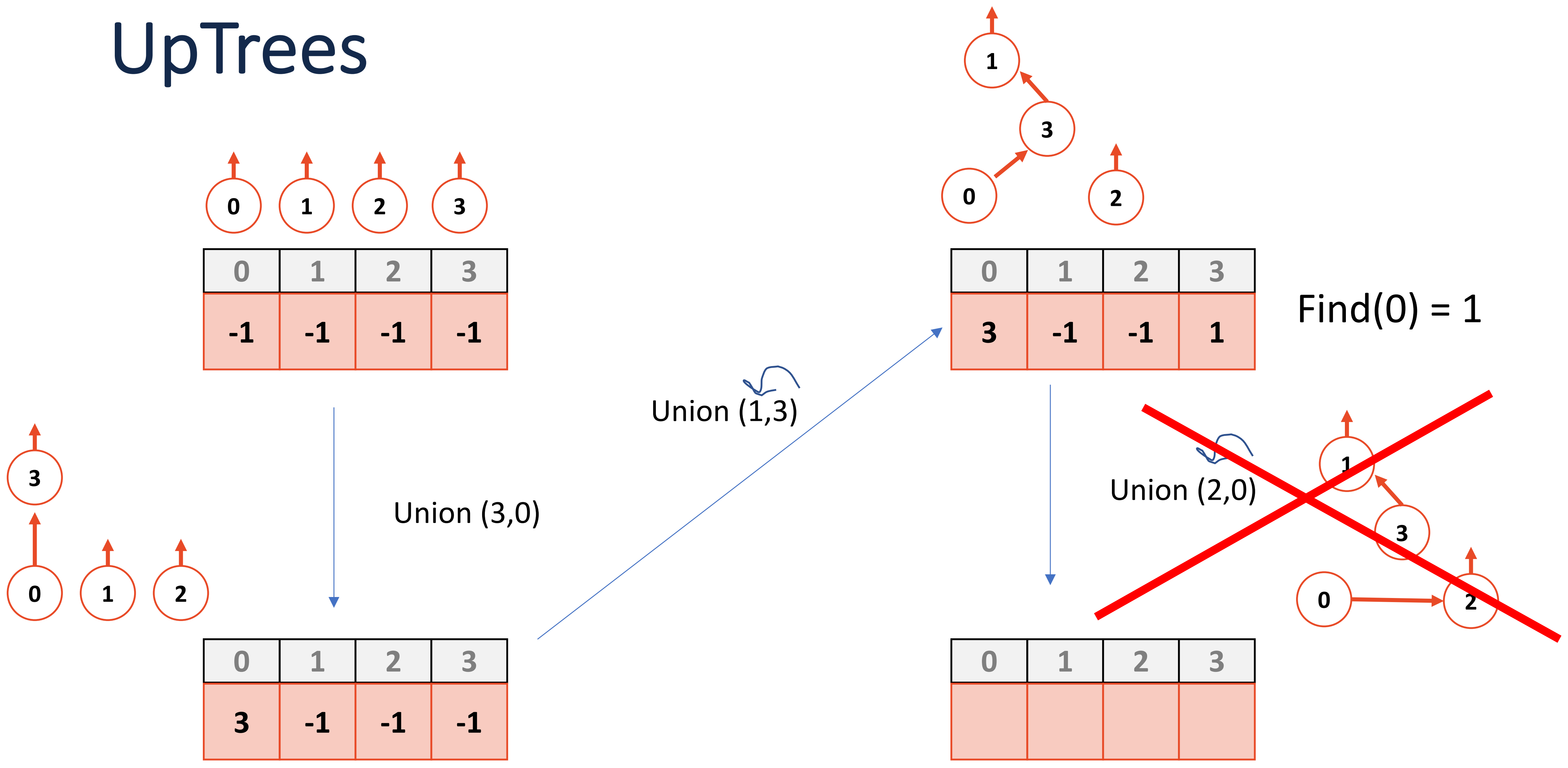
Find(0) = 1

Union (2,0)



0	1	2	3

UpTrees



0	1	2	3
-1	-1	-1	-1

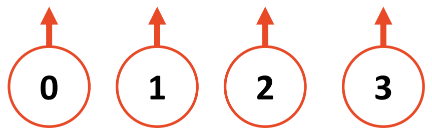
0	1	2	3
3	-1	-1	-1

0	1	2	3
3	-1	-1	1

0	1	2	3

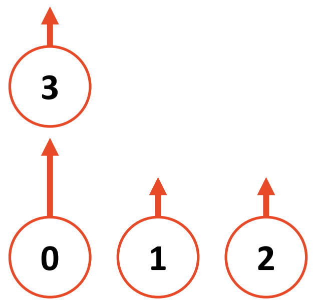
We have to find representative elements for each set to do Union

UpTrees



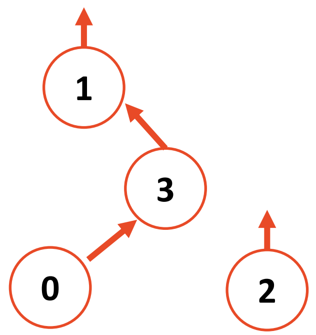
0	1	2	3
-1	-1	-1	-1

Union (3,0)



0	1	2	3
3	-1	-1	-1

Union (1,3)



0	1	2	3
3	-1	-1	1

Find(0) = 1

Union (2,0)

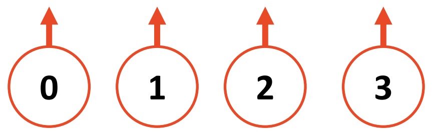
Find(2) = 2

Find(0) = 1

Union (2,1)

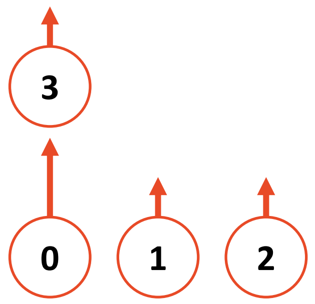
0	1	2	3

UpTrees



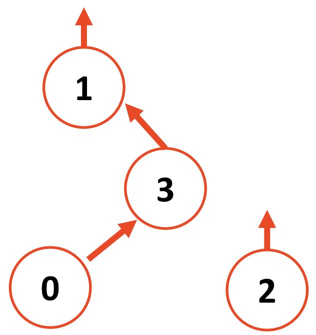
0	1	2	3
-1	-1	-1	-1

Union (3,0)



0	1	2	3
3	-1	-1	-1

Union (1,3)



0	1	2	3
3	-1	-1	1

Find(0) = 1

Union (2,0)

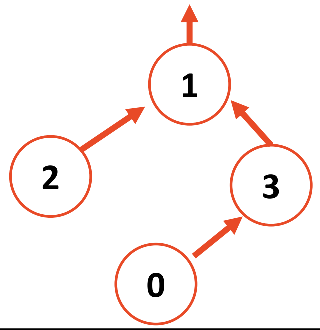
Find(2) = 2

Find(0) = 1

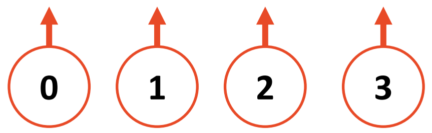
Union (2,1)

0	1	2	3

1. 2-> 1

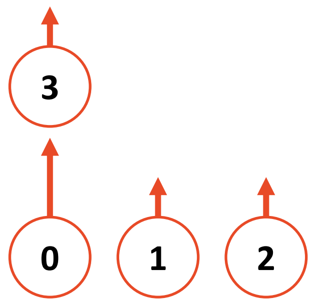


UpTrees



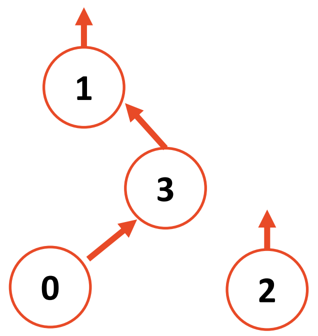
0	1	2	3
-1	-1	-1	-1

Union (3,0)



0	1	2	3
3	-1	-1	-1

Union (1,3)



0	1	2	3
3	-1	-1	1

Find(0) = 1

Union (2,0)

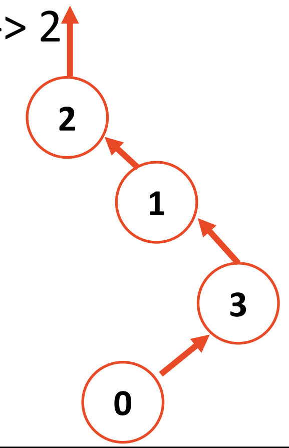
Find(2) = 2

Find(0) = 1

0	1	2	3
3	2	-1	1

Union (2,1)

2. 1-> 2



Disjoint Sets Find

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

Running time?

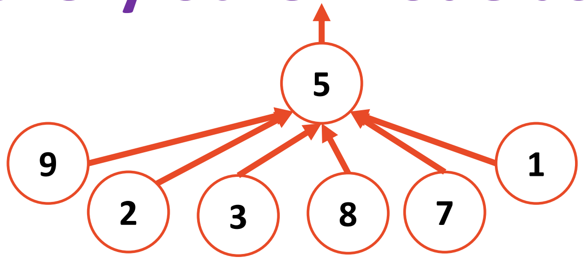
Structure: A structure similar to a linked list

Running time: O(h) == O(n)

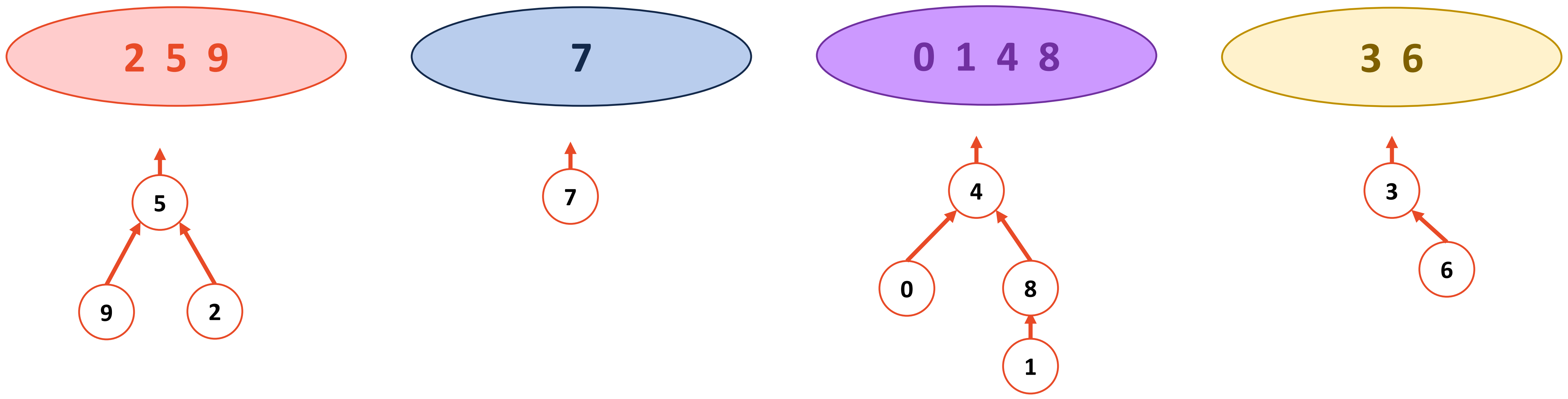
What is the ideal UpTree?

Structure: One root node with every other node as it's child

Running Time: O(1)

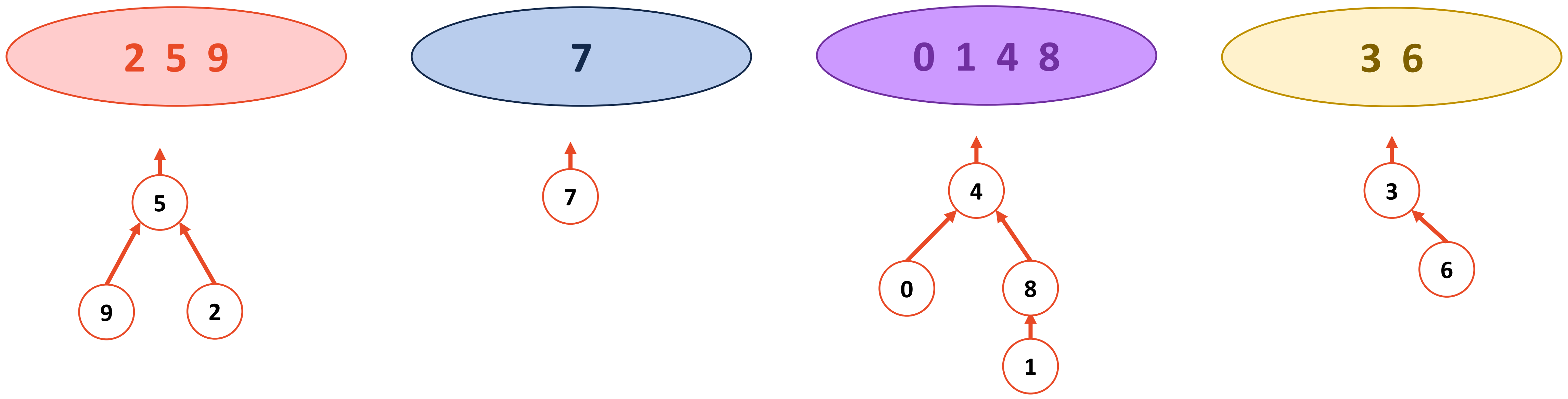


Disjoint Sets



0	1	2	3	4	5	6	7	8	9
4	8	5	-1	-1	-1	3	-1	4	5

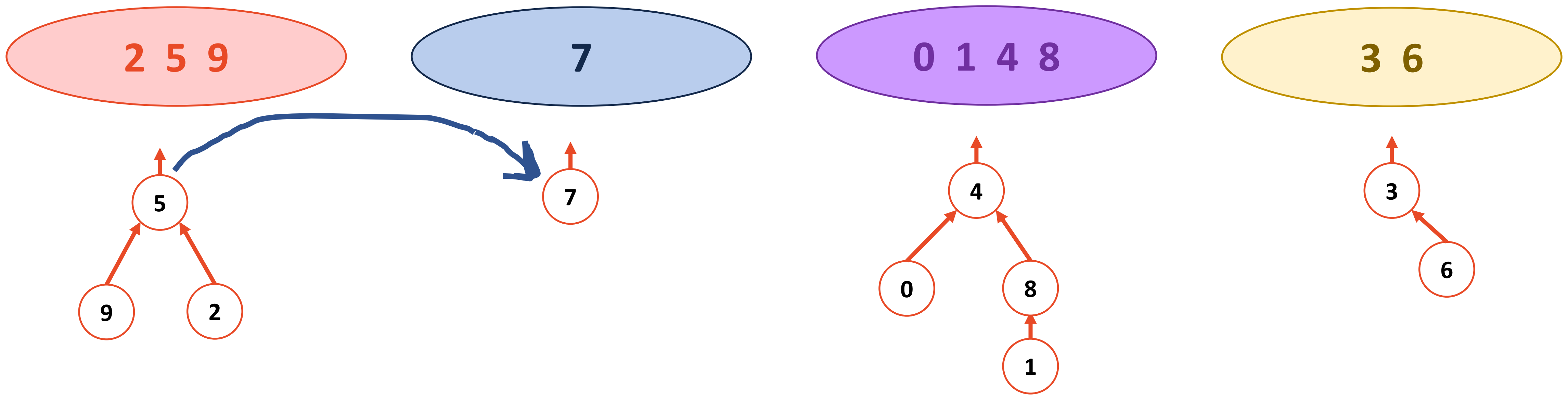
Disjoint Sets



Union(5,7) vs Union (7, 5)

0	1	2	3	4	5	6	7	8	9
4	8	5	-1	-1	-1	3	-1	4	5

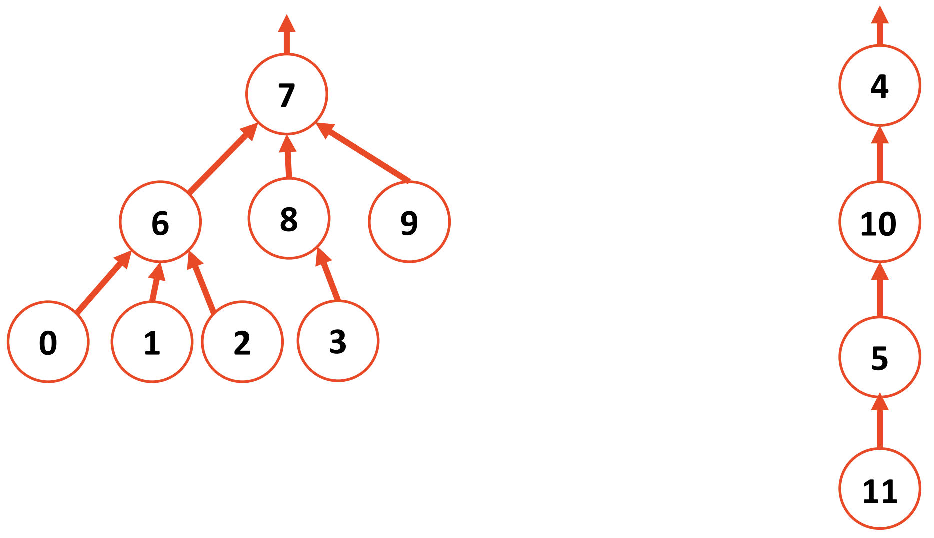
Disjoint Sets



Union (7, 5) – height increases! ☹

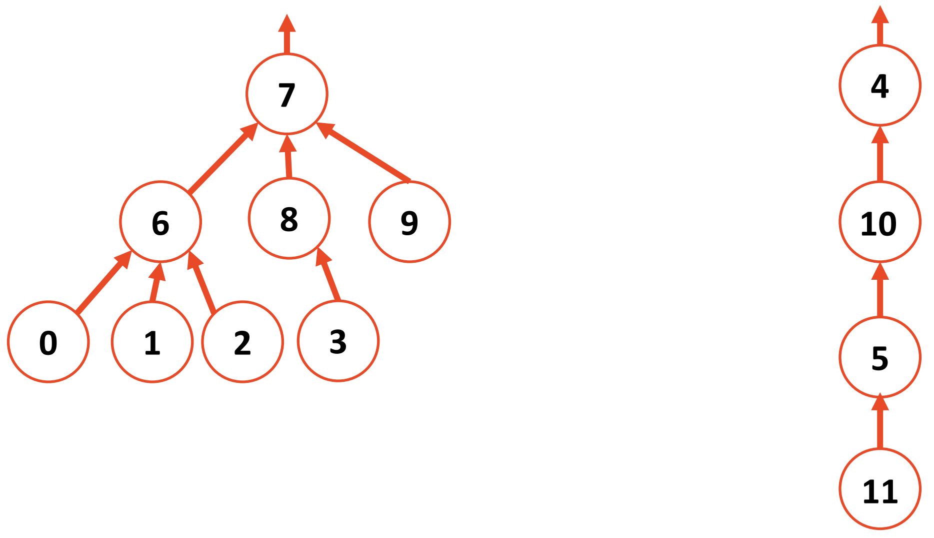
0	1	2	3	4	5	6	7	8	9
4	8	5	-1	-1	-1	3	-1	4	5

Disjoint Sets – Union



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

Disjoint Sets – Smart Union

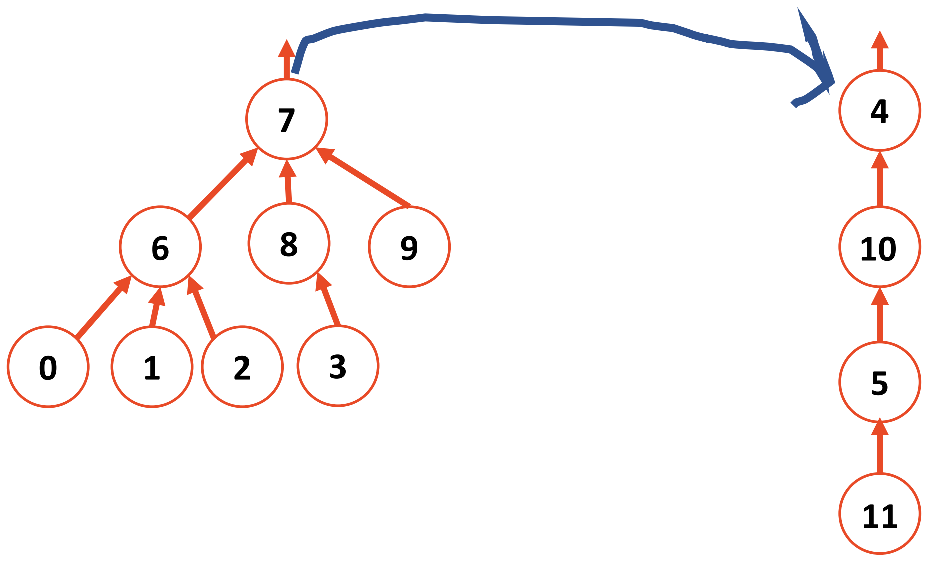


Union by height

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

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

Disjoint Sets – Smart Union



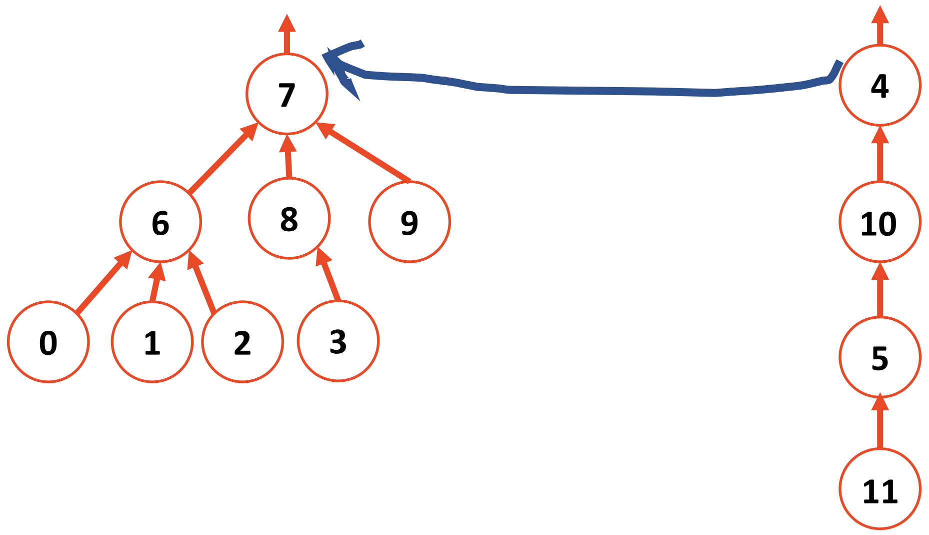
Union by height

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

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

Average node is further away from the root node ☹

Disjoint Sets – Smart Union



Union by height

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

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

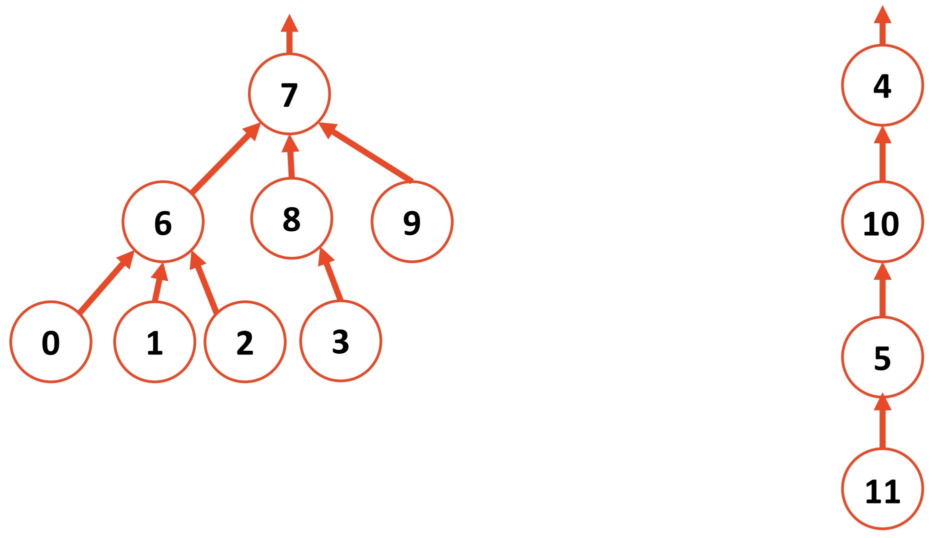
Union by size

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

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

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

Disjoint Sets – Smart Union



Union by height

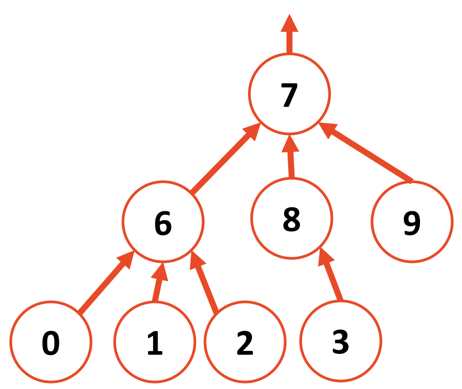
0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	-3	7	7	4	5

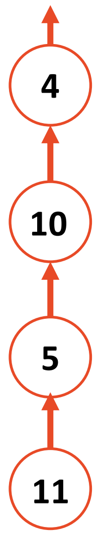
Value = -h -1
(to avoid 0s)

Union by size

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

Disjoint Sets – Smart Union





Union by height

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

Value = h -1
(to avoid 0s)

Union by size

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

Value = - n

Disjoint Sets Find

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

```
void DisjointSets::unionBySize(int root1, int root2) {
  int newSize = arr_[root1] + arr_[root2];

  // If arr_[root1] is less than (more negative), it is the larger set;
  // we union the smaller set, root2, with root1.
  if ( arr_[root1] < arr_[root2] ) {
    arr_[root2] = root1;
    arr_[root1] = newSize;
  }

  // Otherwise, do the opposite:
  else {
    arr_[root1] = root2;
    arr_[root2] = newSize;
  }
}
```

Disjoint Sets Find

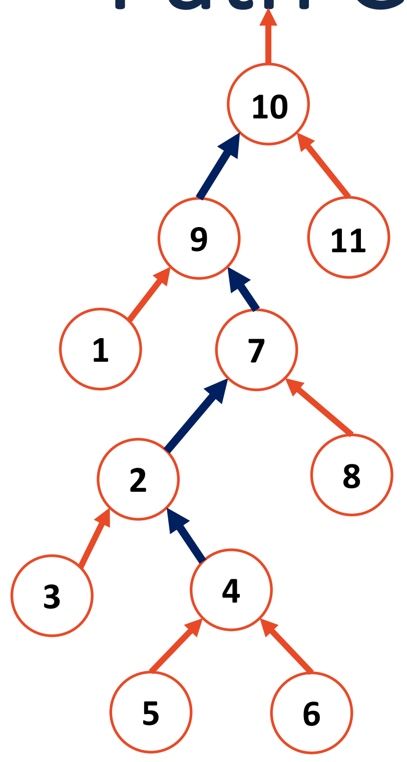
```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else { return _find( s[i] ); }
}
```

$O(h) \equiv O(\lg n)$

```
void DisjointSets::unionBySize(int root1, int root2) {
  int newSize = arr_[root1] + arr_[root2];

  // If arr_[root1] is less than (more negative), it is the larger set;
  // we union the smaller set, root2, with root1.
  if ( arr_[root1] < arr_[root2] ) {
    arr_[root2] = root1;
    arr_[root1] = newSize;
  }

  // Otherwise, do the opposite:
  else {
    arr_[root1] = root2;
    arr_[root2] = newSize;
  }
}
```

Running time of Union is O(1), given root nodes, O(h) otherwise.

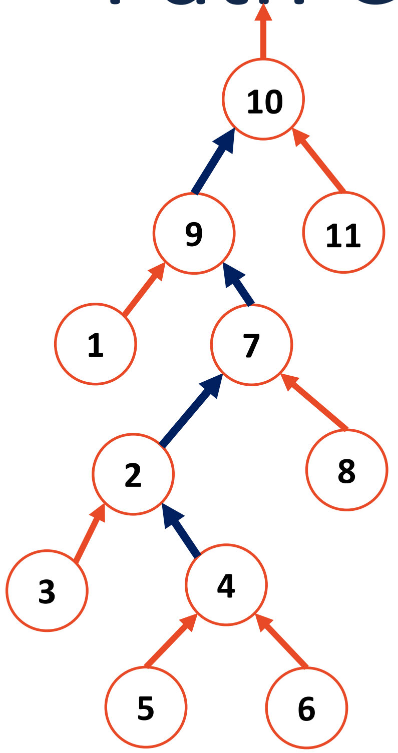
Path Compression



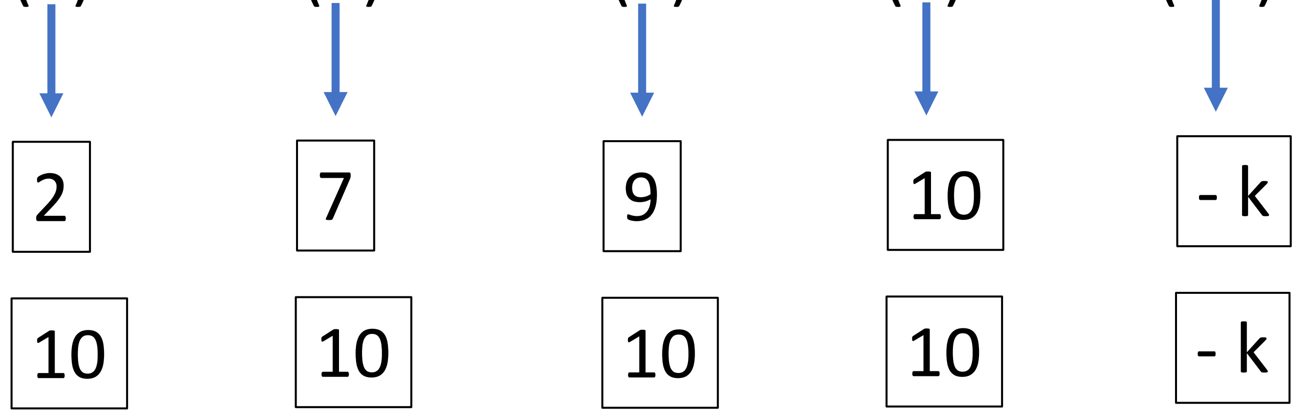
$$\begin{tikzcd}
\text{Find}(4) \arrow[r, "->"] \arrow[d] & \text{Find}(2) \arrow[r, "->"] \arrow[d] & \text{Find}(7) \arrow[r, "->"] \arrow[d] & \text{Find}(9) \arrow[r, "->"] \arrow[d] & \text{Find}(10) \arrow[d] \\
\boxed{2} & \boxed{7} & \boxed{9} & \boxed{10} & \boxed{-k}
\end{tikzcd}$$

When we unwind recursion, we update every element we visited to point to the root node

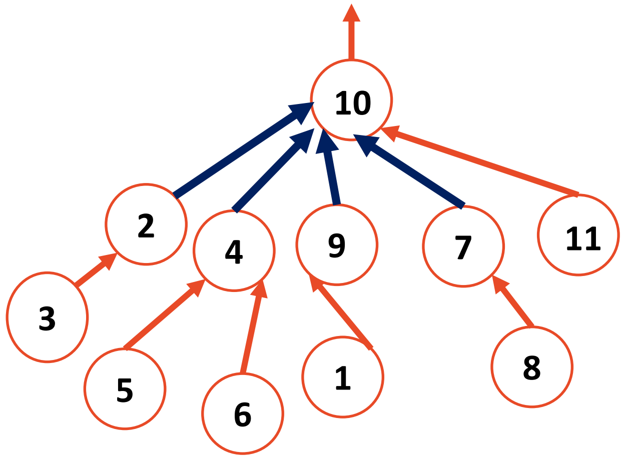
Path Compression



Find(4) ->Find(2) ->Find (7)->Find(9)->Find(10)



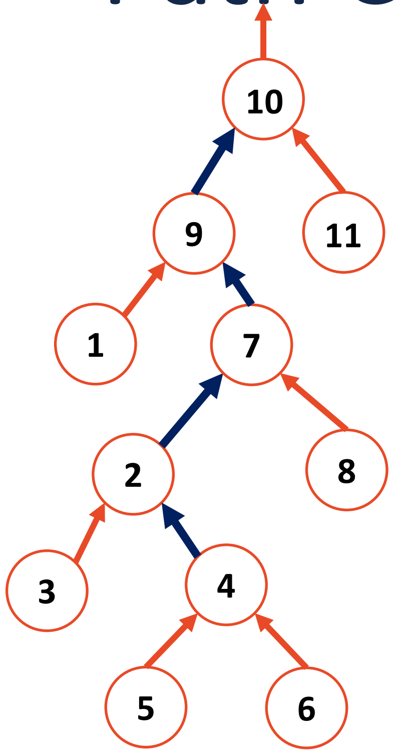
When we unwind recursion, we update every element we visited to point to the root node

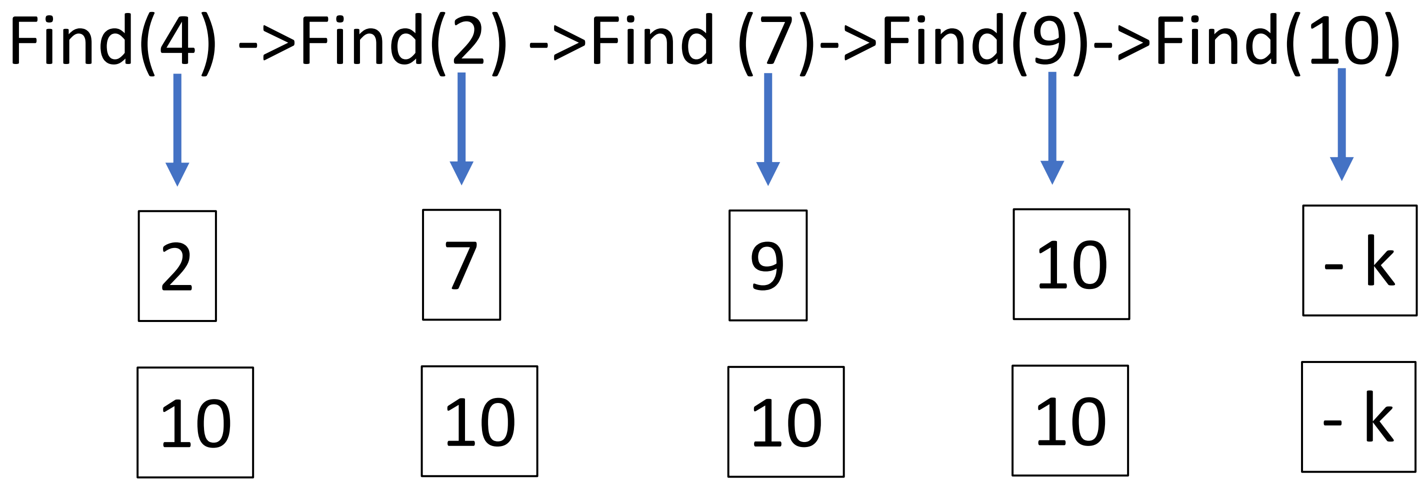


This is self improving algorithm. Running time gets better and better.

Overall running time of the algorithm is going to be better than AVL tree.

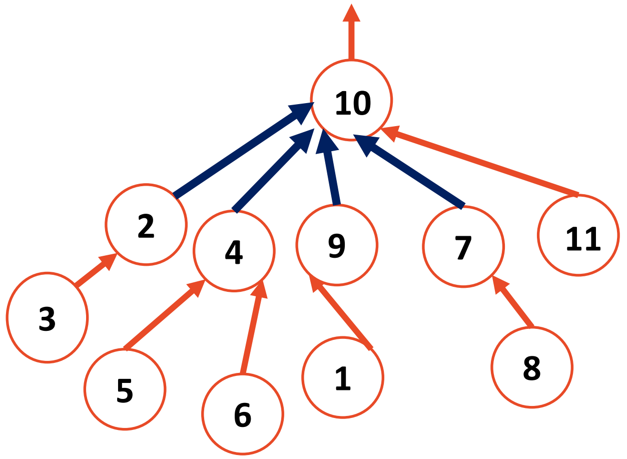
Path Compression





When we unwind recursion, we update every element we visited to point to the root node

Since first find takes O(log n) time, we cannot call this algorithm O(1).



This is self improving algorithm. Running time gets better and better.

Overall running time of the algorithm is going to be better than AVL tree.

Disjoint Sets Analysis

The **iterated log** function:

The number of times you can take a log of a number.

$\log^*(n) =$

$0 \qquad\qquad\qquad\quad , n \leq 1$

$1 + \log^*(\log(n)) , n > 1$

What is $\mathbf{lg^*(2^{65536})}$?

Disjoint Sets Analysis

The **iterated log** function:

The number of times you can take a log of a number.

log*(n) =

0 , $n \leq 1$

1 + log*(log(n)) , $n > 1$

What is **lg*(2^{65536})** -> 65536 -> 16 -> 4 -> 2 -> 1

Disjoint Sets Analysis

In an Disjoint Sets implemented with smart **unions** and path compression on **find**:

Any sequence of **m** **union** and **find** operations result in the worse case running time of O($m \lg^*(n)$),
where **n** is the number of items in the Disjoint Sets.

When used **with other algorithms** we will say running time is O(1)

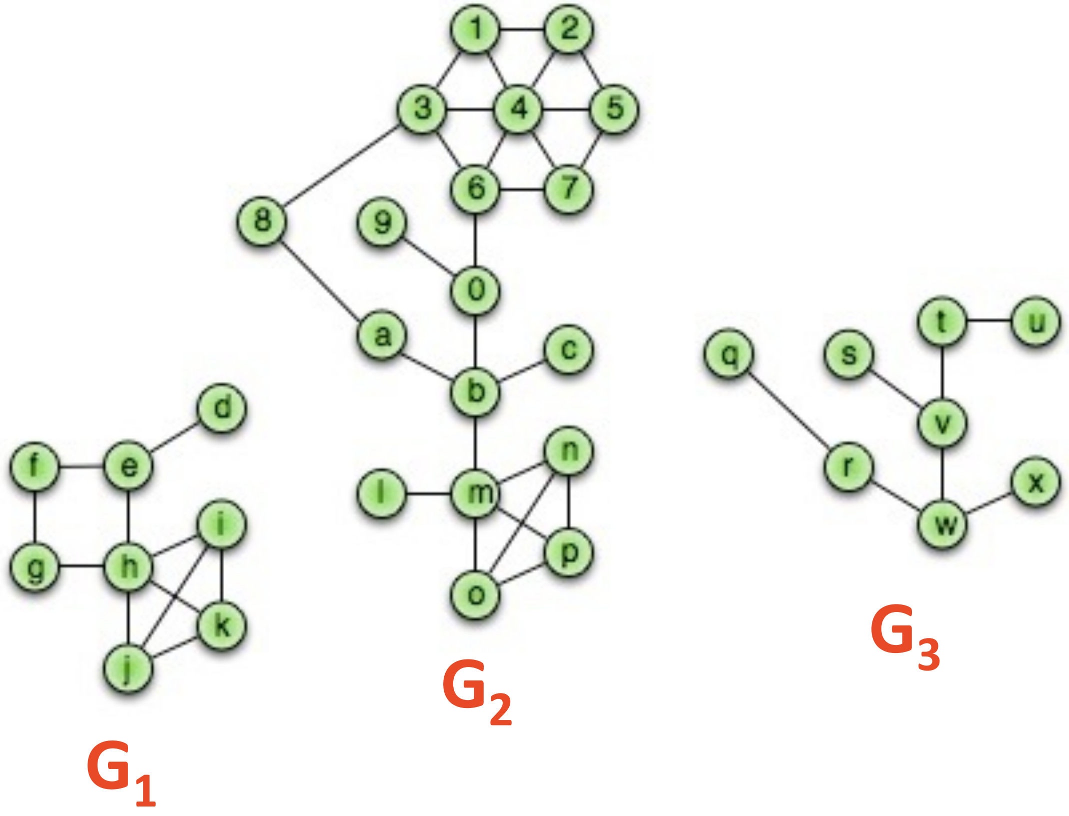
Graphs

1. A common vocabulary
2. Graph implementations
3. Graph traversals
4. Graph algorithm

Graph vocabulary

Graph vocabulary

A graph G is a tuple of a set of vertices V, and a set of edges E



$G = (V, E)$

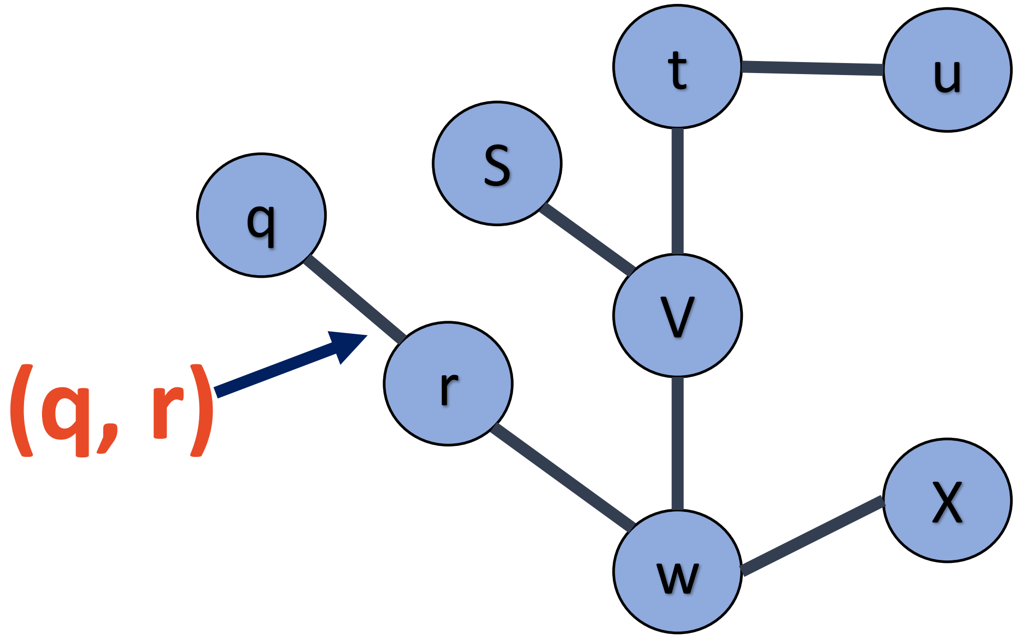
$|V| = n$ //number of vertices

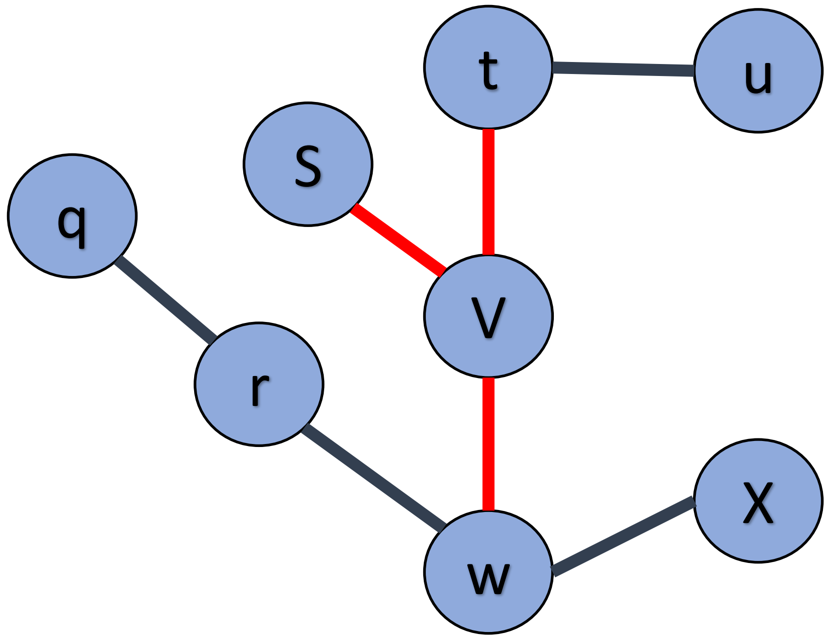
$|E| = m$ //number of edges

Graph vocabulary

We identify an edge by stating two vertices it connects.

- **Incident edges** → all edges that touch that node
  - $I(v) \ = \ \{\,\{x, v\}\ in\ E\}$

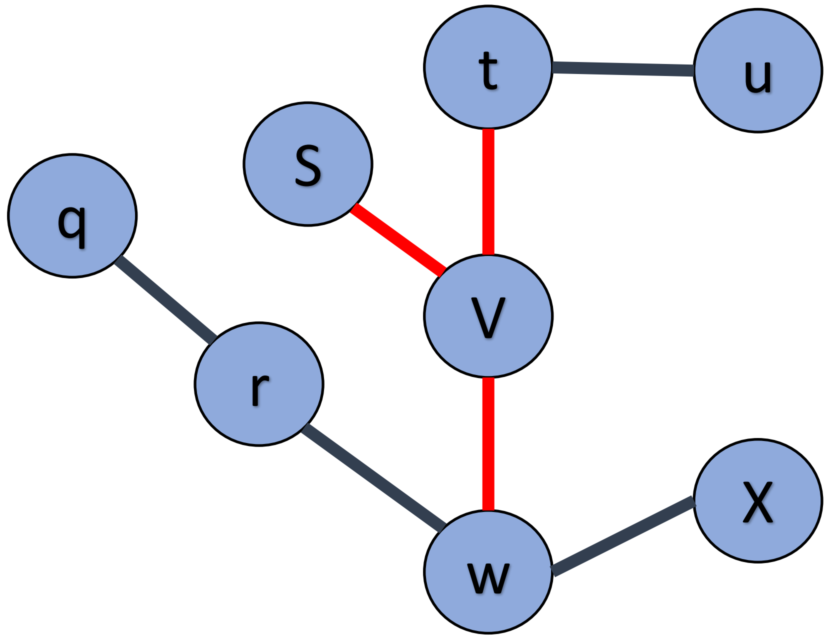




Incident edges for V are $(\boldsymbol{v}, \boldsymbol{s}), (\boldsymbol{v}, \boldsymbol{t}), (\boldsymbol{v}, \boldsymbol{w})$

Graph vocabulary

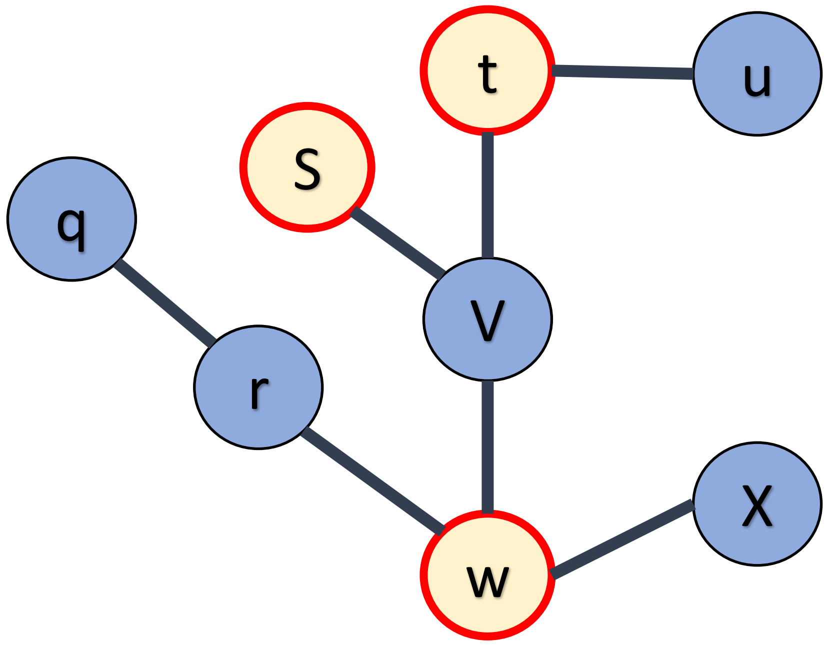
- **Degree** → the number of incident edges.
  - $Degree(v) = |I(v)|$



Degree(v) = 3

Graph vocabulary

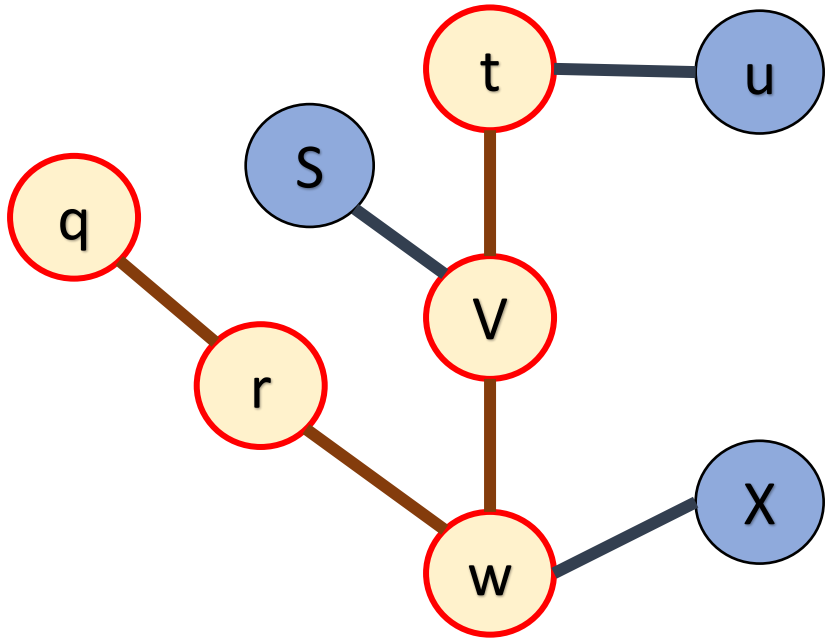
- **Adjacent vertex** → a vertex at the other end of the incident edge.
  - $A(v) = \{x: (x, v) in\ E\}$



$\boldsymbol{A(v) = \{s, w, t\}}$

Graph vocabulary

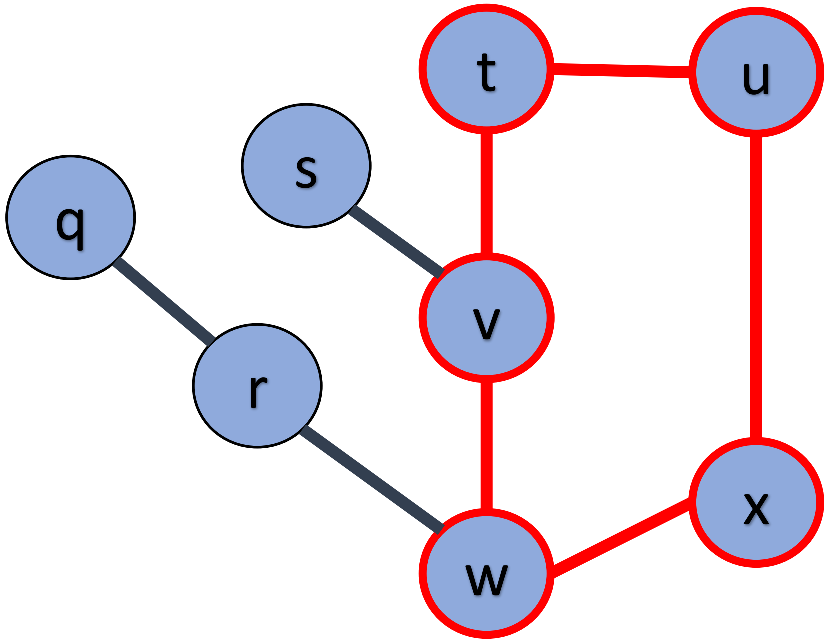
- **Path** → a sequence of vertices connected by edges.



Path from q to t is: $\{\boldsymbol{q}, \boldsymbol{r}, \boldsymbol{w}, \boldsymbol{v}, \boldsymbol{t}\}$

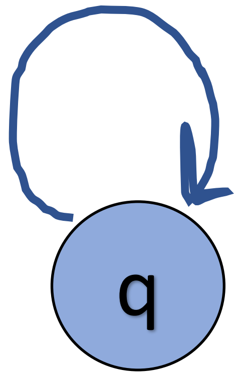
Graph vocabulary

- **Cycle** → a path with common beginning and end.

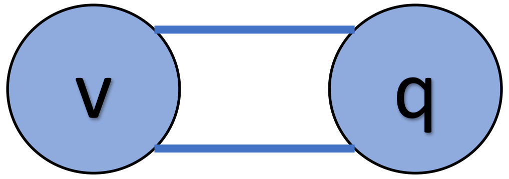


Graph vocabulary

- **Simple Graph** →A graph with **no** self loops and multi-edges



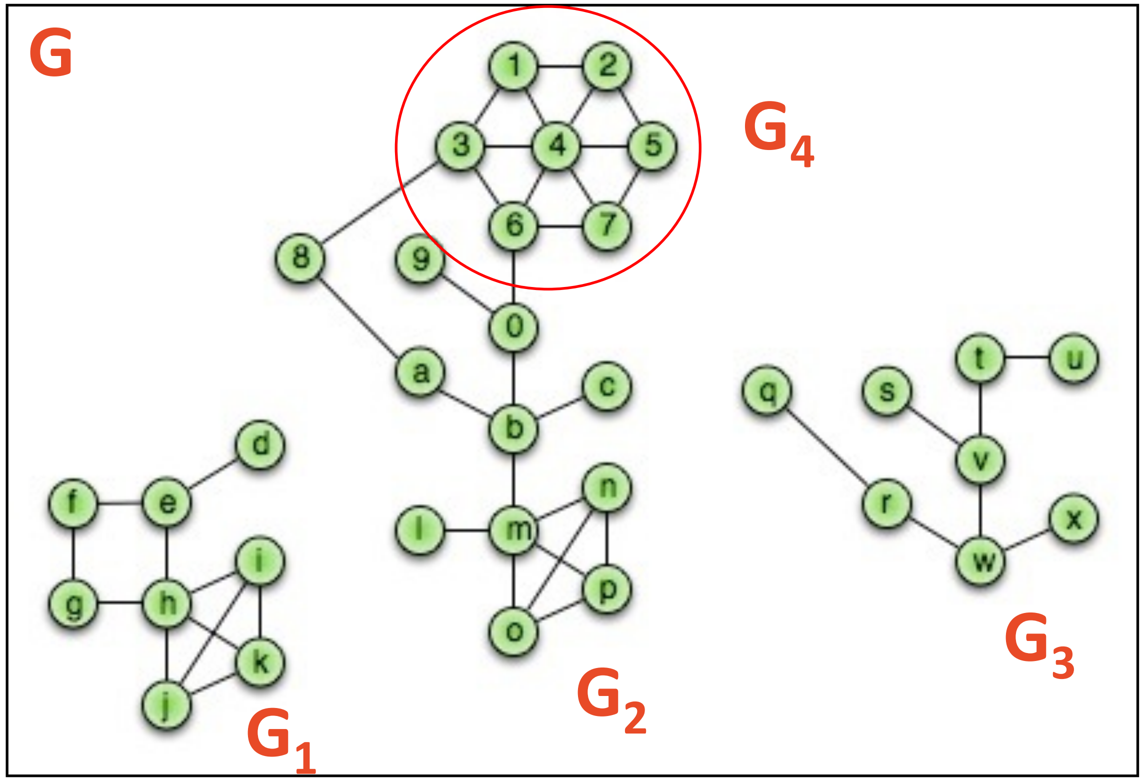
Self loop



Multi-edges

Graph vocabulary

- **Subgraph** →any subset of vertices such that every edge in the subgraph implies that both vertices that are incident to that edge are part of that graph



Subgraph(G):

G' = (V', E'):

V' ∈ V, E' ∈ E, and

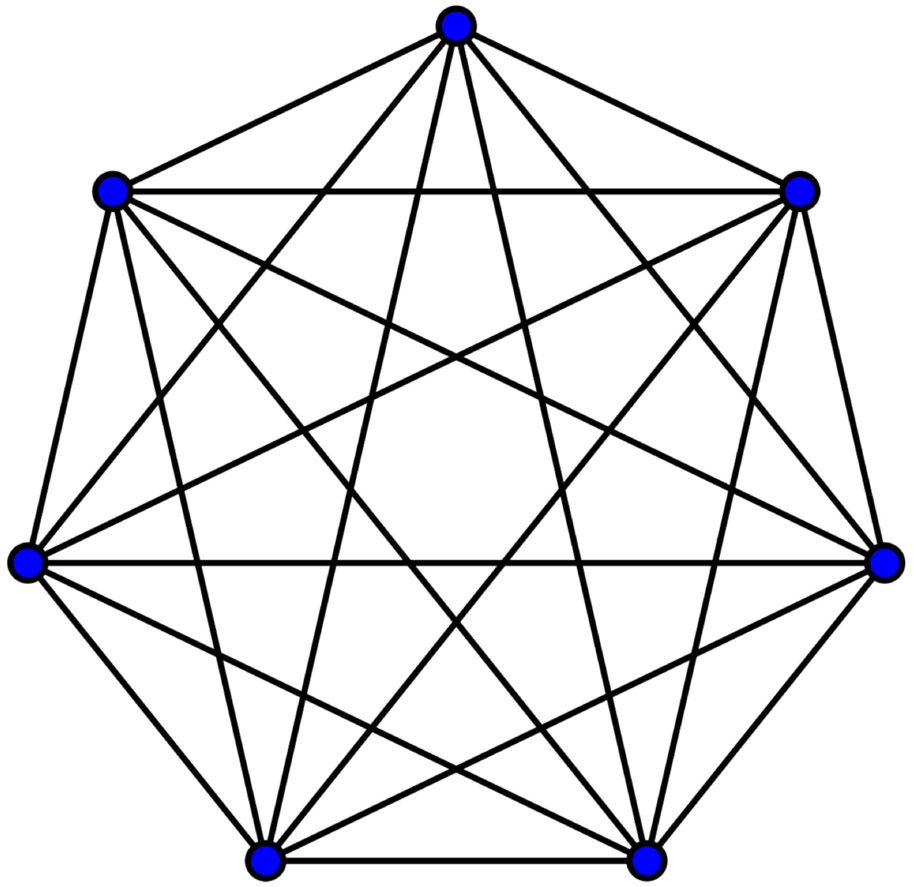
(u, v) ∈ E → u ∈ V', v ∈ V'

✓ G1 G2, G3 and G4 are subgraphs of G

✓ G4 is also a subgraph of G2

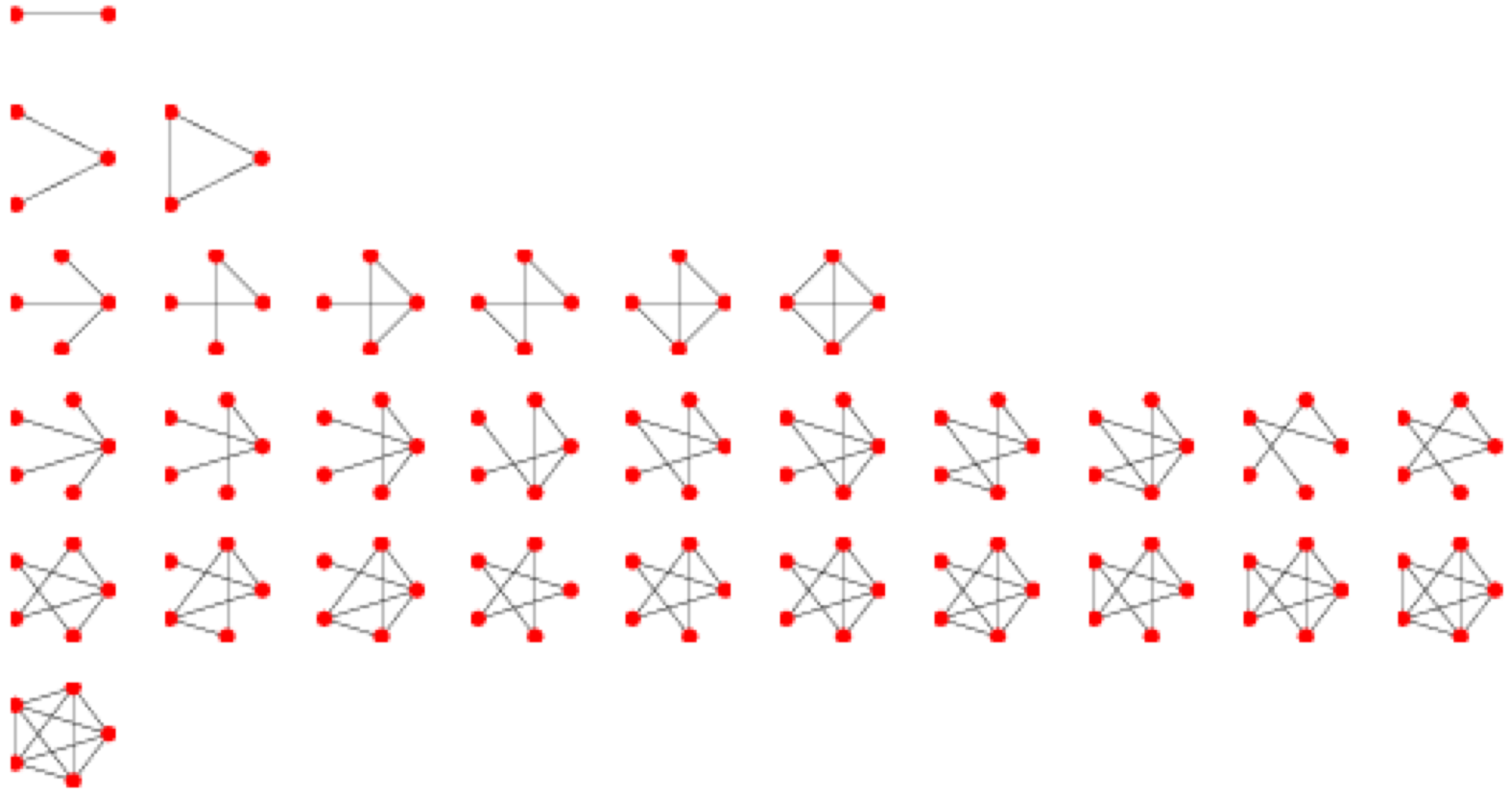
Graph vocabulary

- **Complete subgraph**: every two distinct vertices are adjacent.



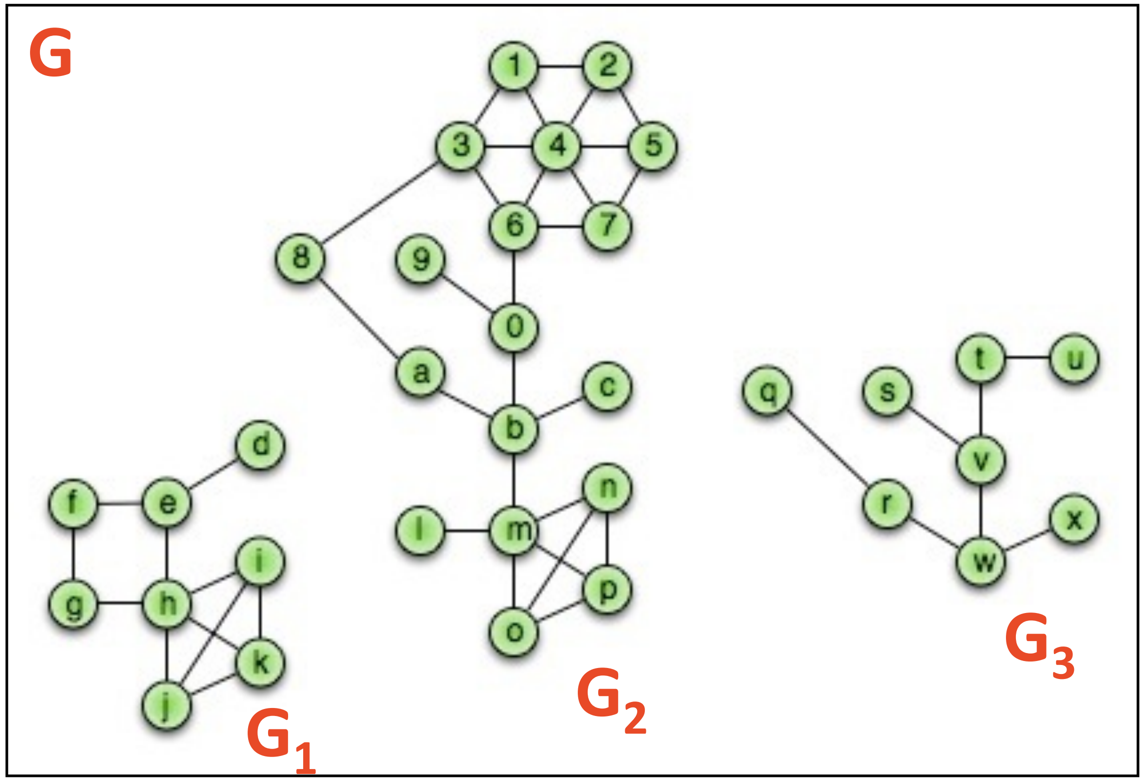
Graph vocabulary

- **Connected subgraph**: there is a path between every two vertices in the graph.



Graph vocabulary

- **Connected component**: a connected subgraph where non of the vertices are connected to the rest of the graph.



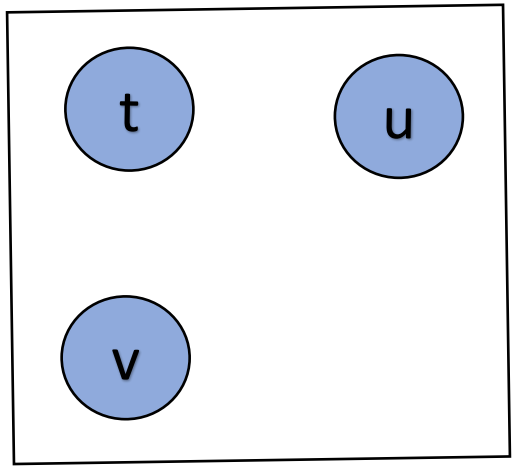
G1, G2 and G3 are connected components.

Properties of Graph

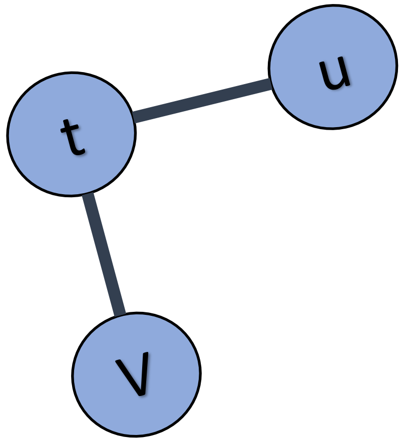
Properties of Graph

Running times are often reported by n (the number of vertices) but often depend on m (the number of edges).

- **Minimum number of edges (m):**
    - Not Connected: m = 0
    - Connected: m = n-1



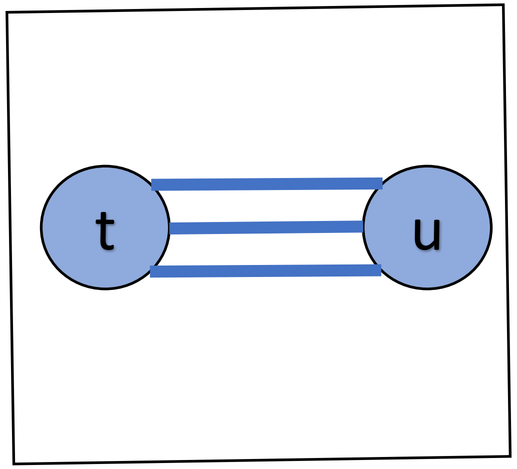
Example 1.



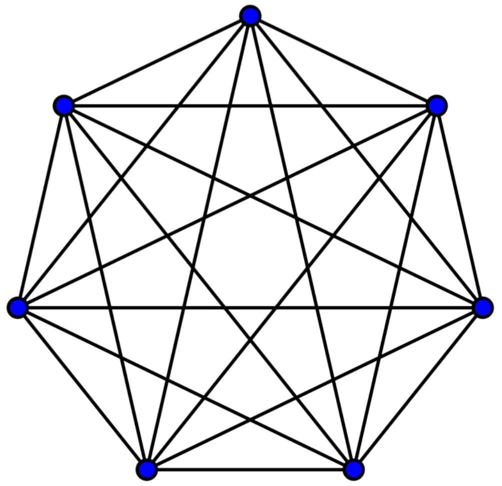
Example 2.

Properties of Graph

- **Maximum edges (m):**
  - Not simple: **m = ∞,** since we can have multiple edges between vertices.
  - Simple: $\sum_{k=1}^{n-1} k$



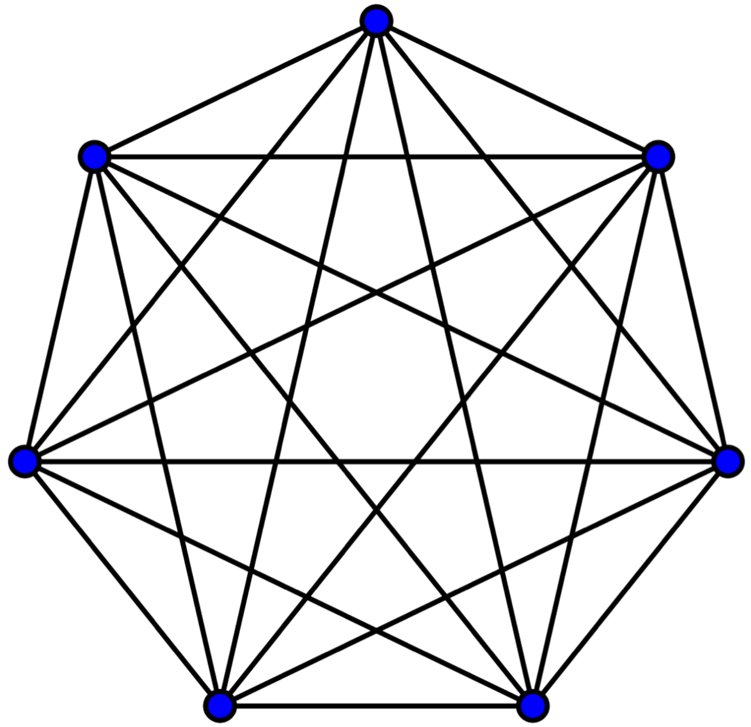
Example 1.



Example 2.

Maximum edges in simple graph:

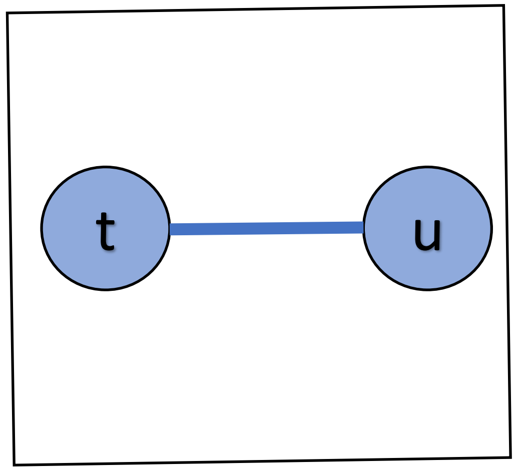
n	m
1	0
2	1
3	3
4	6
5	10
...	...
n	$\sum_{k=1}^{n-1} k$



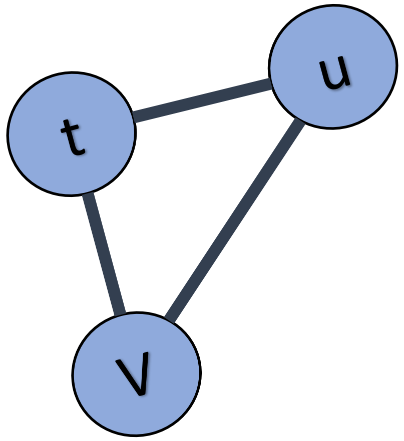
Properties of Graph

Sum of all degrees of all vertices:

$$\sum_{v \epsilon V} \deg(v) = 2 \, * m$$



$$\sum_{v \epsilon V} \deg(v) = 2$$



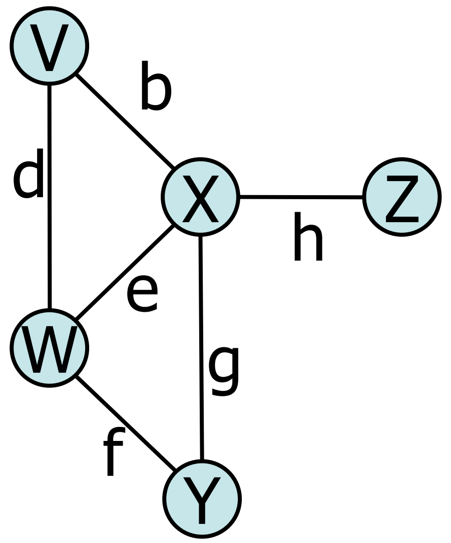
$$\sum_{v \epsilon V} \deg(v) = 6$$

Graph ADT

Graph ADT

Data:

- **Vertices**
- **Edges**
- **Some data structure maintaining the structure between vertices and edges.**



Functions:

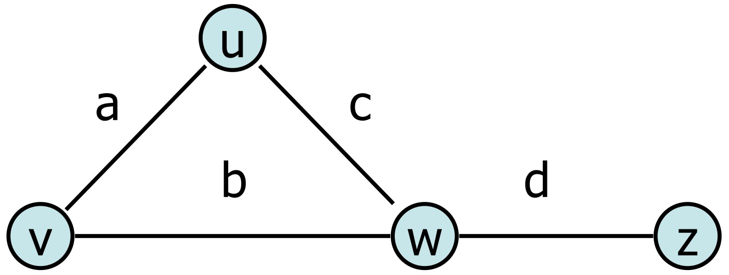
- **insertVertex(K key);**
- **insertEdge(Vertex v1, Vertex v2, K key);**

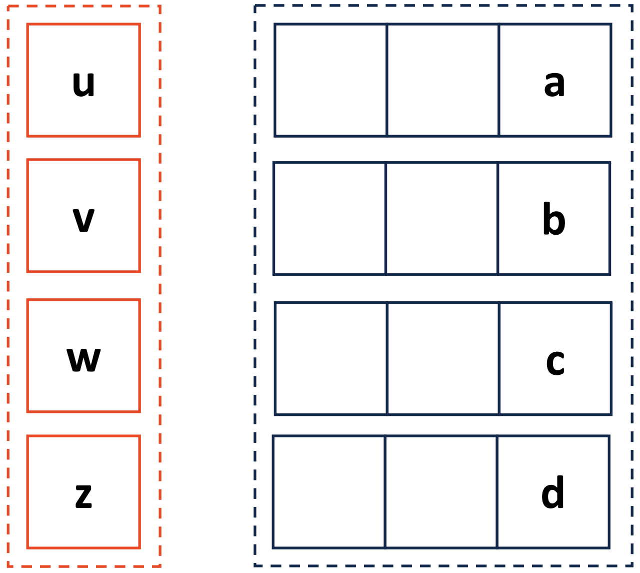
- **removeVertex(Vertex v);**
- **removeEdge(Vertex v1, Vertex v2);**

- **incidentEdges(Vertex v);**
- **areAdjacent(Vertex v1, Vertex v2);**

- **origin(Edge e);**
- **destination(Edge e);**

Graph Implementation: Edge List





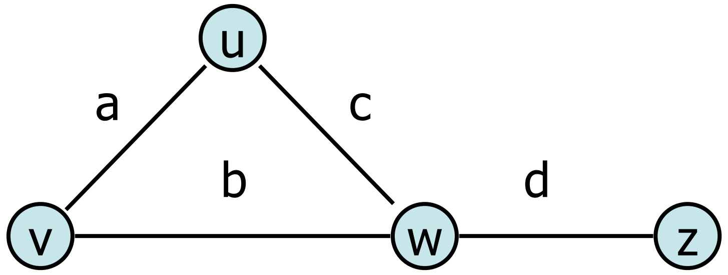
Array of vertices:

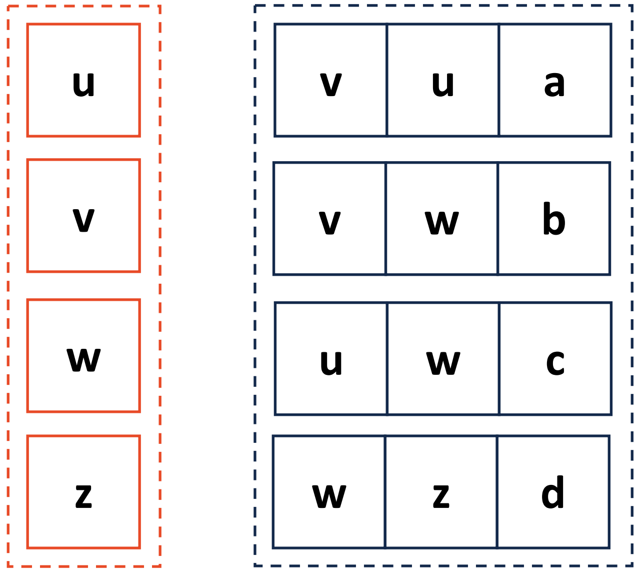
- × Takes time to find a specific vertex

Hash table:

- ✓ find takes O(1) time

Graph Implementation: Edge List





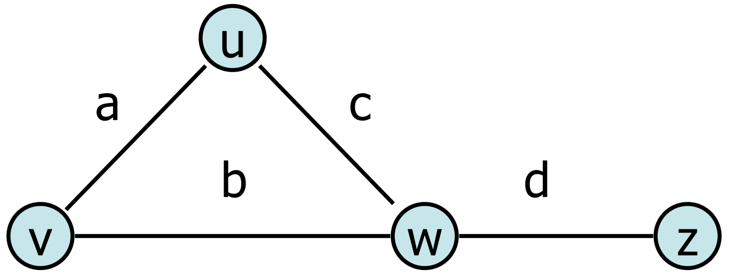
Hash table

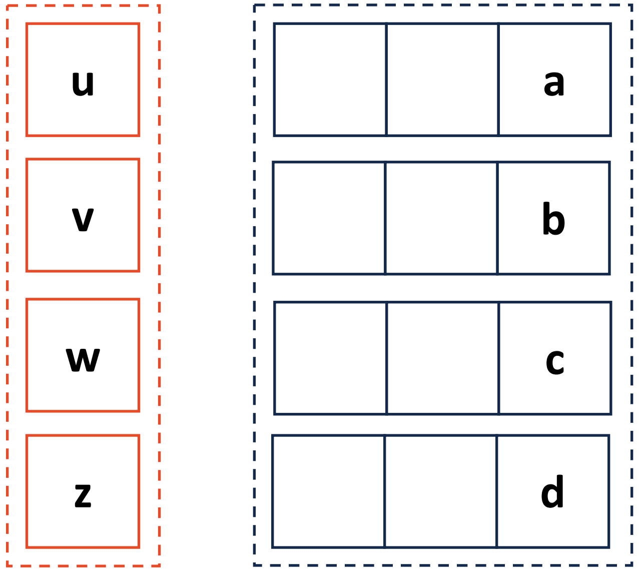
List

Given we use list for edges, what is the running time of insertVertex and removeVertex?

- InsertVertex take O(1) time, since inserting into hash table takes O(1) time.
- Removing vertex means removing vertex from hash table and removing corresponding edges from the list. Running time will be: O(1) + O(m)

Graph Implementation: Edge List





Hash table List

insertVertex(K key) – O(1)

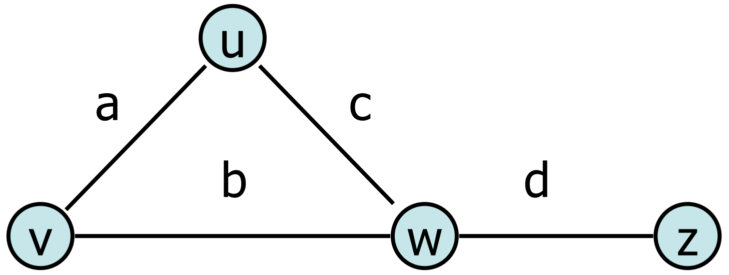
removeVertex(Vertex v) – O(m)

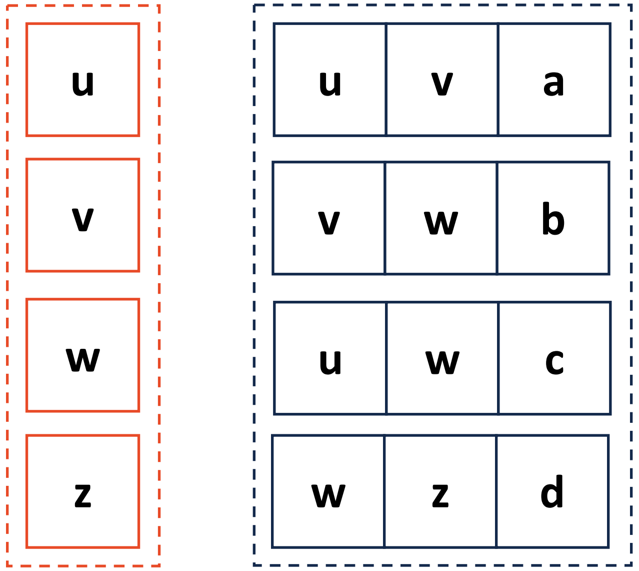
areAdjacent(Vertex v1, Vertex v2) – O(m)

incidentEdges(Vertex v) – O(m)

The relationship between number of nodes and the number of edges can be n^2; which means that O(m) could in fact be $O(n^2)$

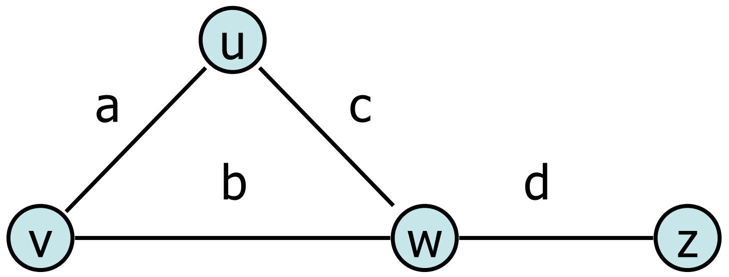
Graph Implementation~~: Hash table for edges~~



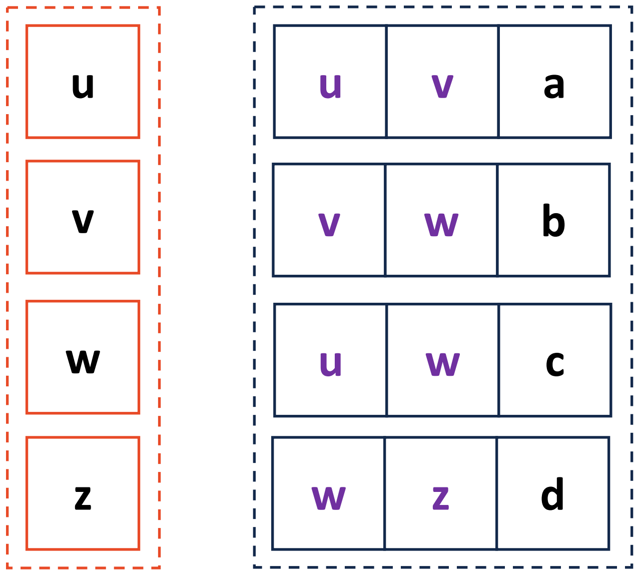


We cannot use a hash table for edges, there is no random distribution (no SUHA) and that would defeat the purpose of the hash table.

Graph Implementation: Adjacency Matrix

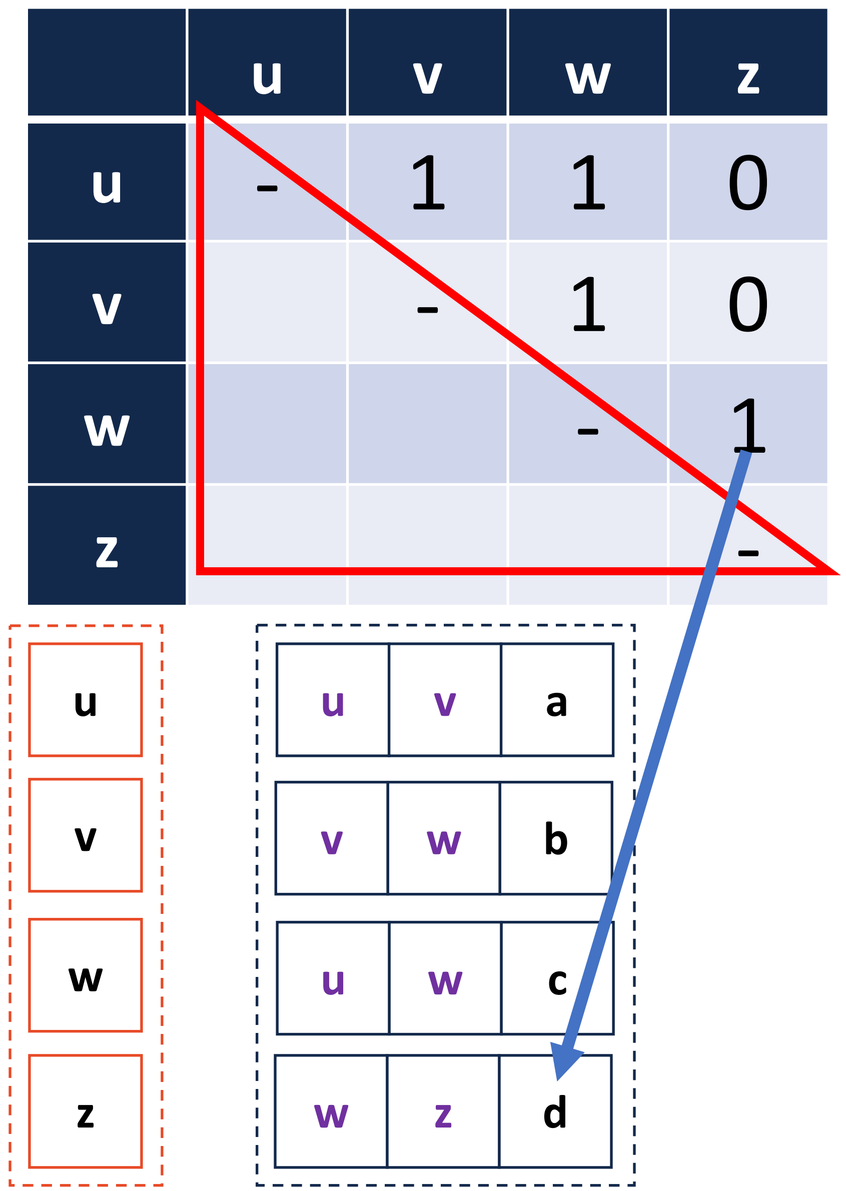


insertVertex(K key) :
removeVertex(Vertex v) :
areAdjacent(Vertex v1, Vertex v2): O(1)
incidentEdges(Vertex v) :



	u	v	w	z
u	-	1	1	0
v		-	1	0
w			-	1
z				-

Adjacency Matrix



	u	v	w	z
u	-	1	1	0
v		-	1	0
w			-	1
z				-

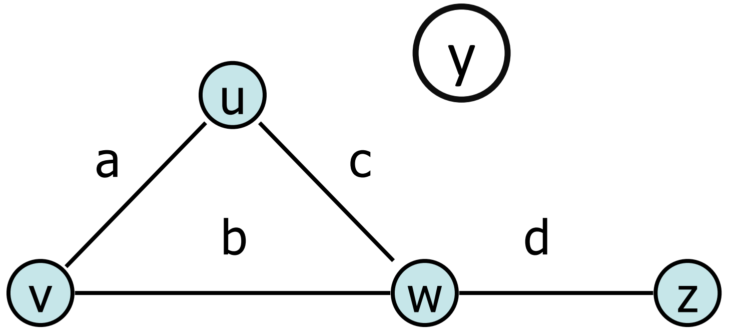
u
v
w
z

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

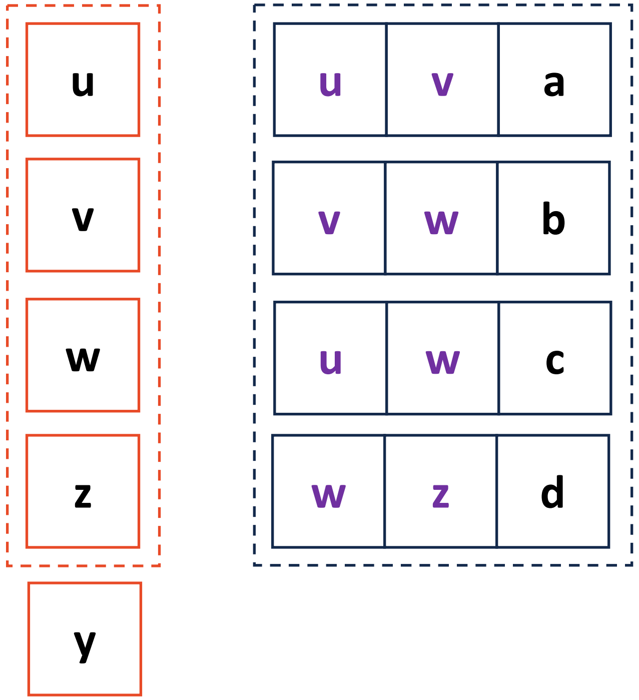
- Number one denotes a pointer to the edge in the edge list when two nodes are adjacent;
- If the graph is not directed bottom triangle is symmetric to the upper triangle, so we can ignore it.

Space complexity $O(n^2)$

insertVertex(K key) :



- Insert y in the hash table O(1);
- Insert data into the matrix:
  - If matrix is full double the size of matrix (double rows and columns)

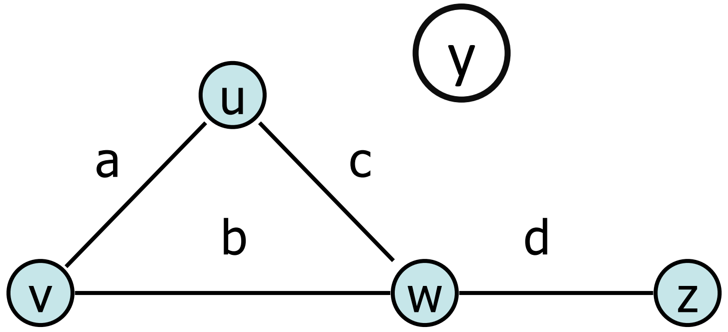


u
v
w
z
y

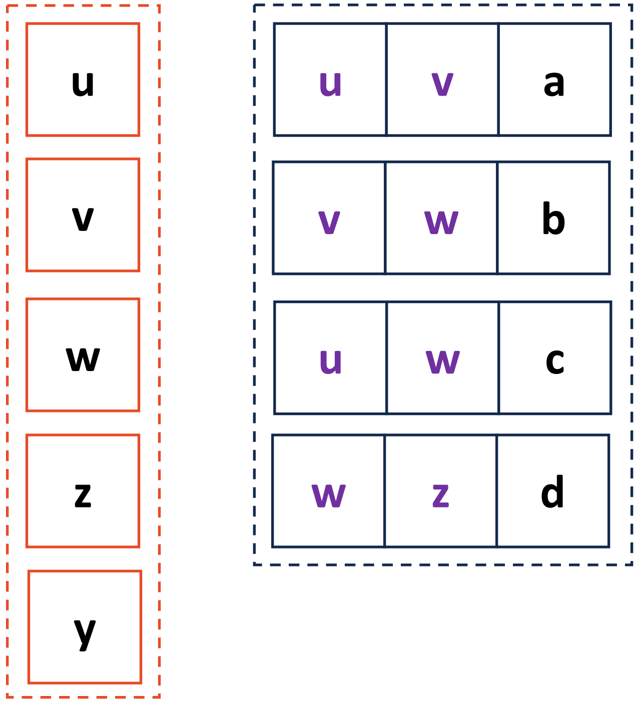
u	v	a
v	w	b
u	w	c
w	z	d

	u	v	w	z
u	-	1	1	0
v		-	1	0
w			-	1
z				-

insertVertex(K key) : O(n)*



- Insert y in the hash table O(1);
- Insert data into the matrix:
  - If matrix is full double the size of matrix (double rows and columns): O(n)* = $O(n^2)/O(n)$



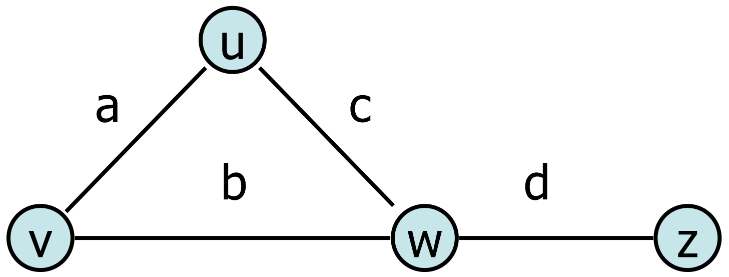
u	
v	
w	
z	
y	

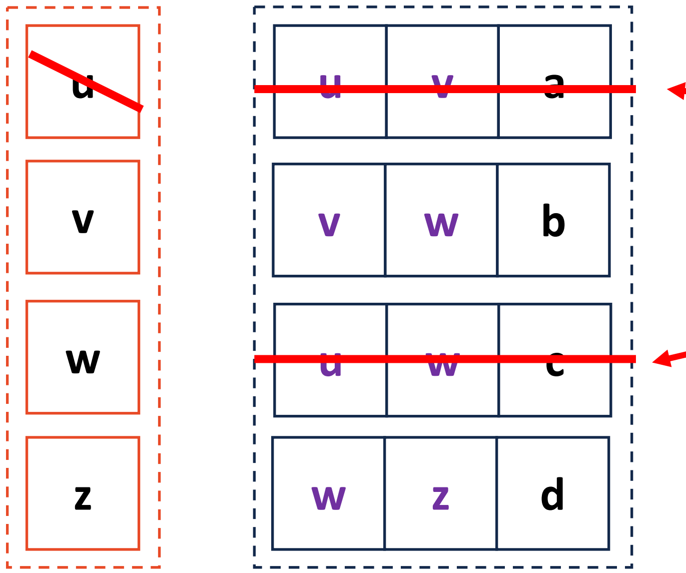
u	v	a
v	w	b
u	w	c
w	z	d

	u	v	w	z	y			
u	-	1	1	0				
v		-	1	0				
w			-	1				
z				-				
y					-			

removeVertex(u) :

- Remove u from the hash table - O(1);
- Go through the row and column u and remove all the edges (O(n)):
  - Removing element from the list (after it is located, takes O(1) time.
  - …

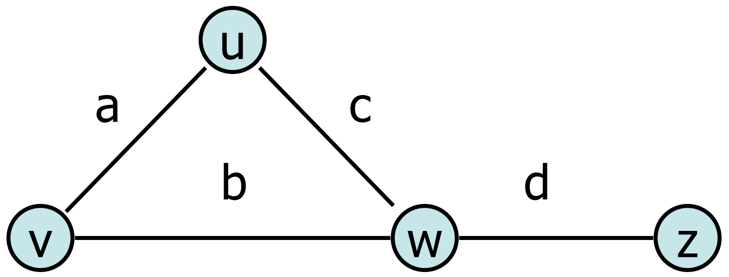


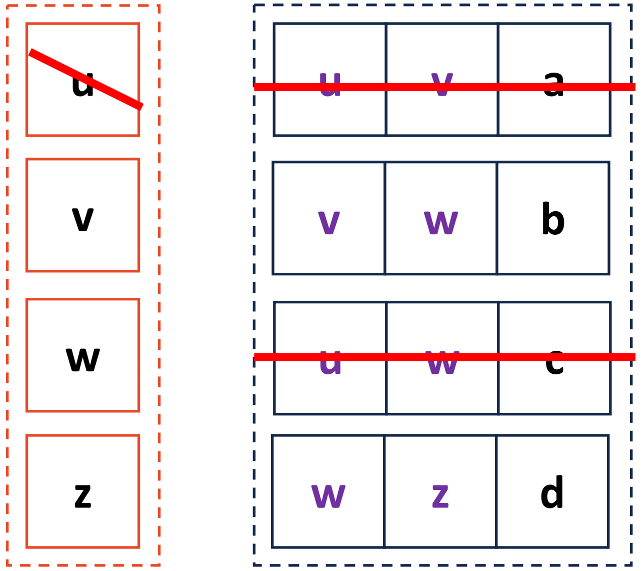


	u	v	w	z
u	-	1	1	0
v		-	1	0
w			-	1
z				-

removeVertex(u) :

- Remove u from the hash table - O(1);
- Go through the row and column u and remove all the edges (O(n)):
  - Removing element from the list (after it is located, takes O(1) time.
  - Repair structure of the table - O(n)

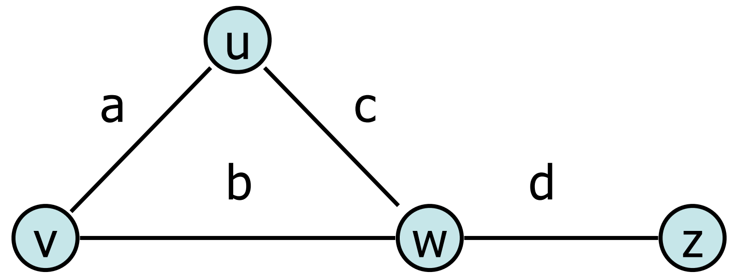


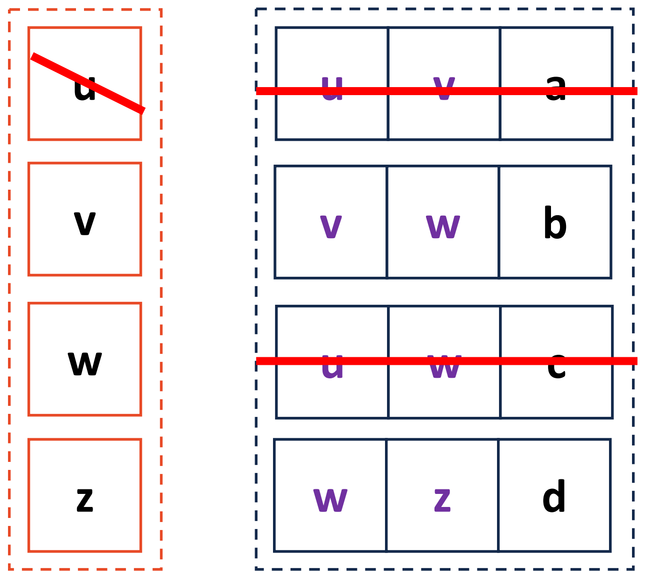


		v	w	z
v		-	1	0
w			-	1
z				-

removeVertex(u) : **O(n)**

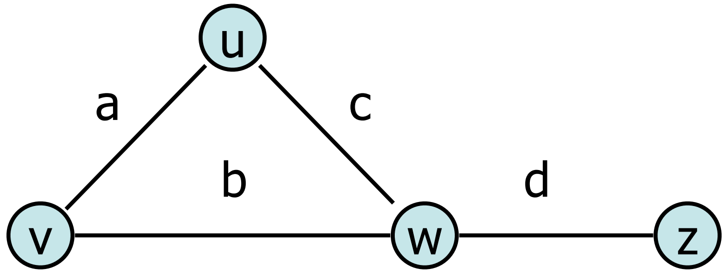
- Remove u from the hash table - O(1);
- Go through the row and column u and remove all the edges (O(n)):
  - Removing element from the list (after it is located, takes O(1) time.
  - Repair structure of the table - O(n)





	z	v	w	
z	-	0	1	
v		-	1	
w			-	

incidentEdges(Vertex v): O(n)



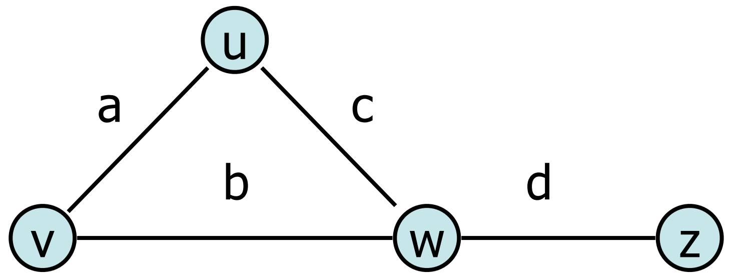
Finding incident edge requires going through the row and column – O(n)

u
v
w
z

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

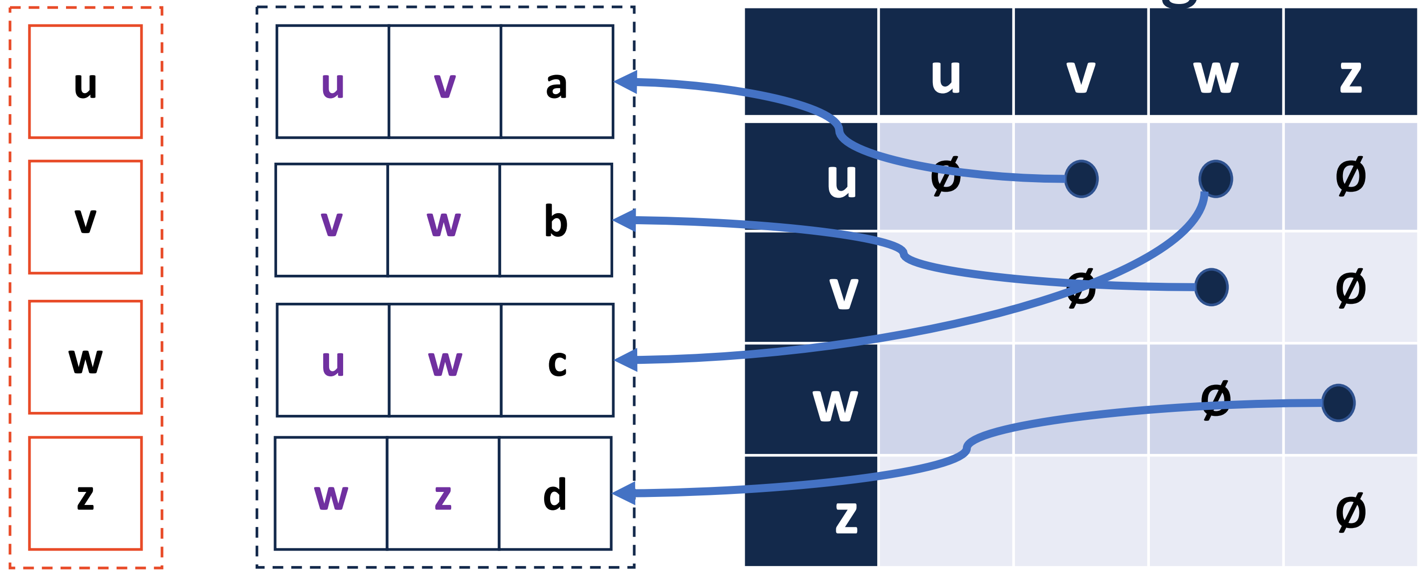
	u	v	w	z
u	-	1	1	0
v		-	1	0
w			-	1
z				-

Adjacency Matrix



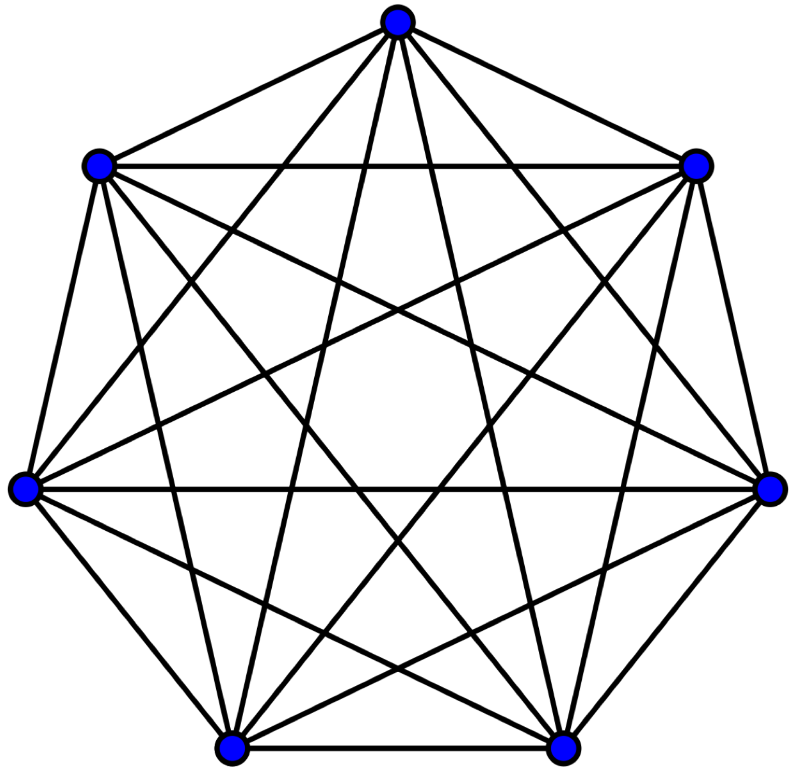
Key Ideas:

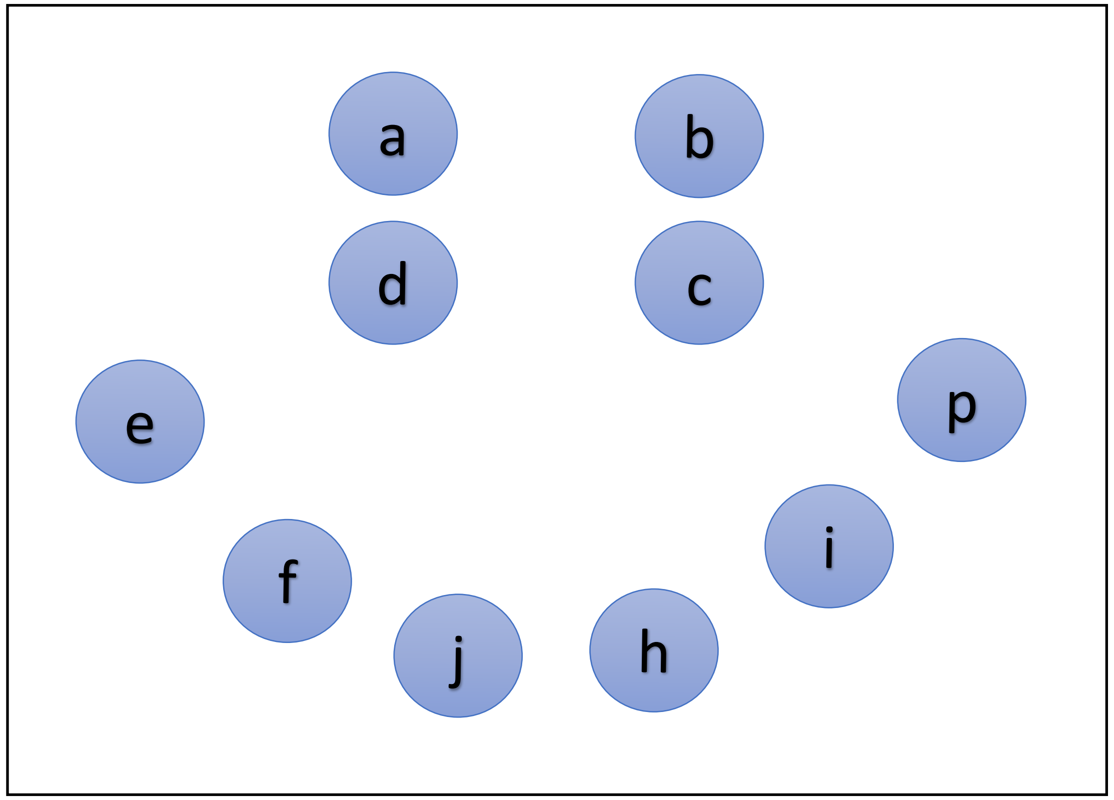
- Given a vertex, O(1) lookup in vertex list
- Given a pair of vertices (an edge), O(1) lookup in the matrix
- Undirected graphs can use an upper triangular matrix



	u	v	w	z
u	ø	●	●	ø
v		ø	●	ø
w			ø	●
z				ø

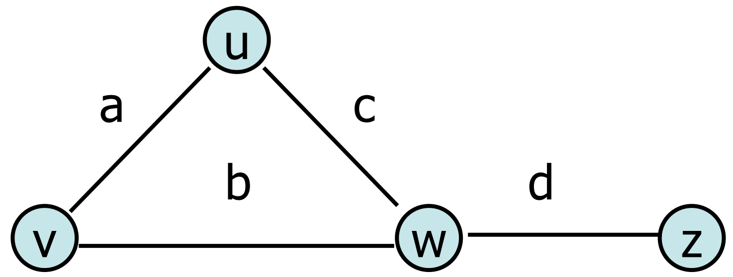
Better running time: O(n) or O(m)?

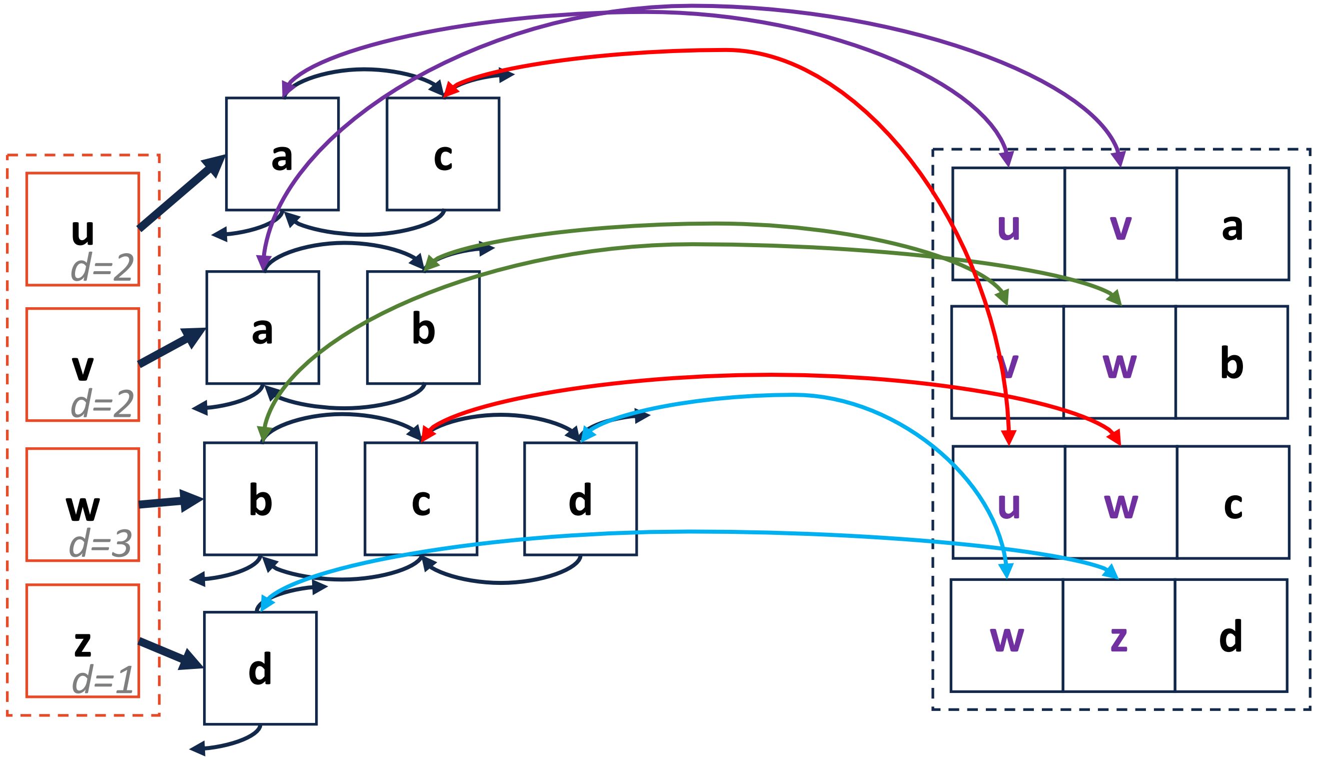




There is no clear winner!

Adjacency List

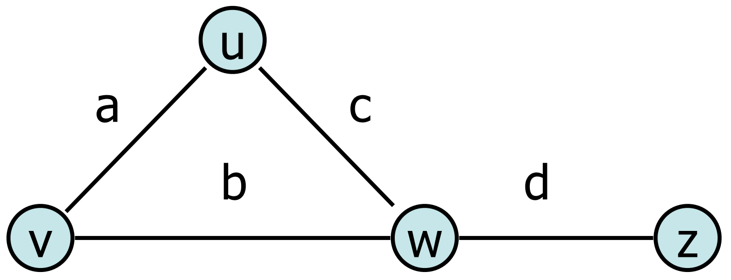


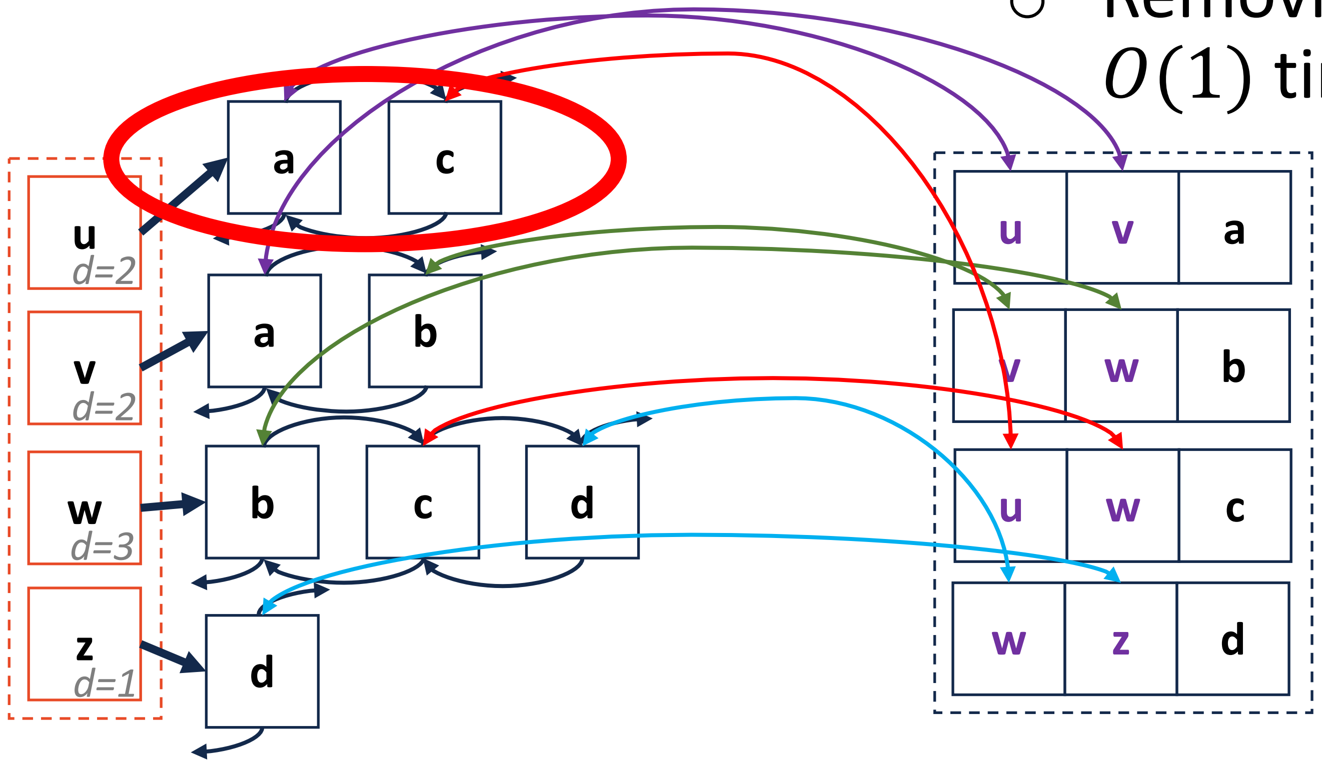


Key Ideas:

- O(1) lookup in vertex list
- Vertex list contains a doubly-linked adjacency list
  - O(1) access to the adjacent vertex's node in adjacency list (via the edge list)
- Vertex list maintains a count of incident edges, or **deg(v)**
- Many operations run in O(deg(v)), and deg(v) ≤ n-1, O(n).

removeVertex(u) : $O(\deg(u))$

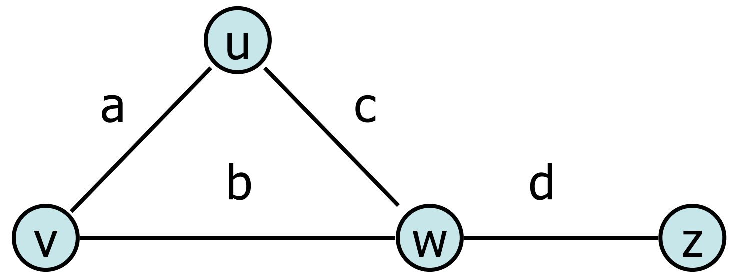




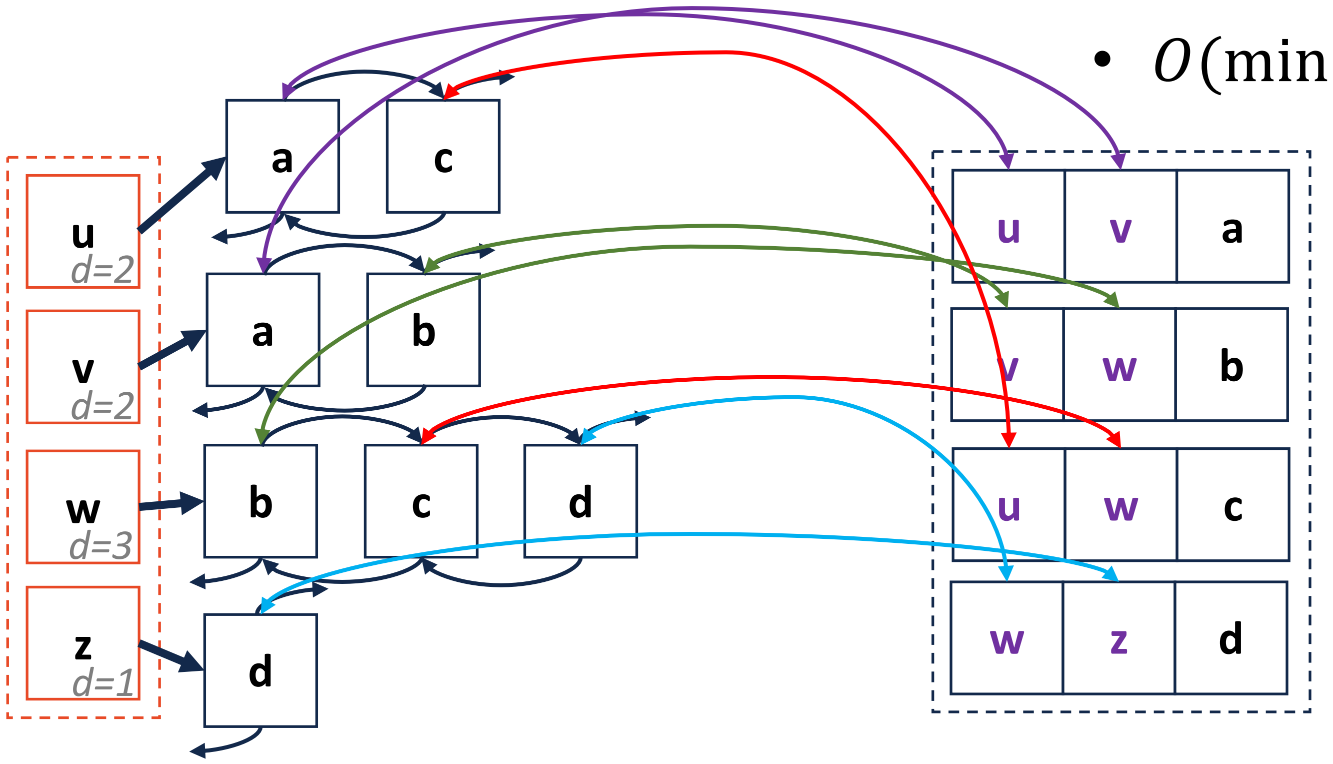
- Remove u from the hash table - O(1);
- Go though the incident list and remove all the edges:
  - u has $\deg(u)$ edges in the list;
  - Removing element from the adj list takes $O(1)$ time – removing all the edges will take

    $\deg(u) \; * \; O(1)$

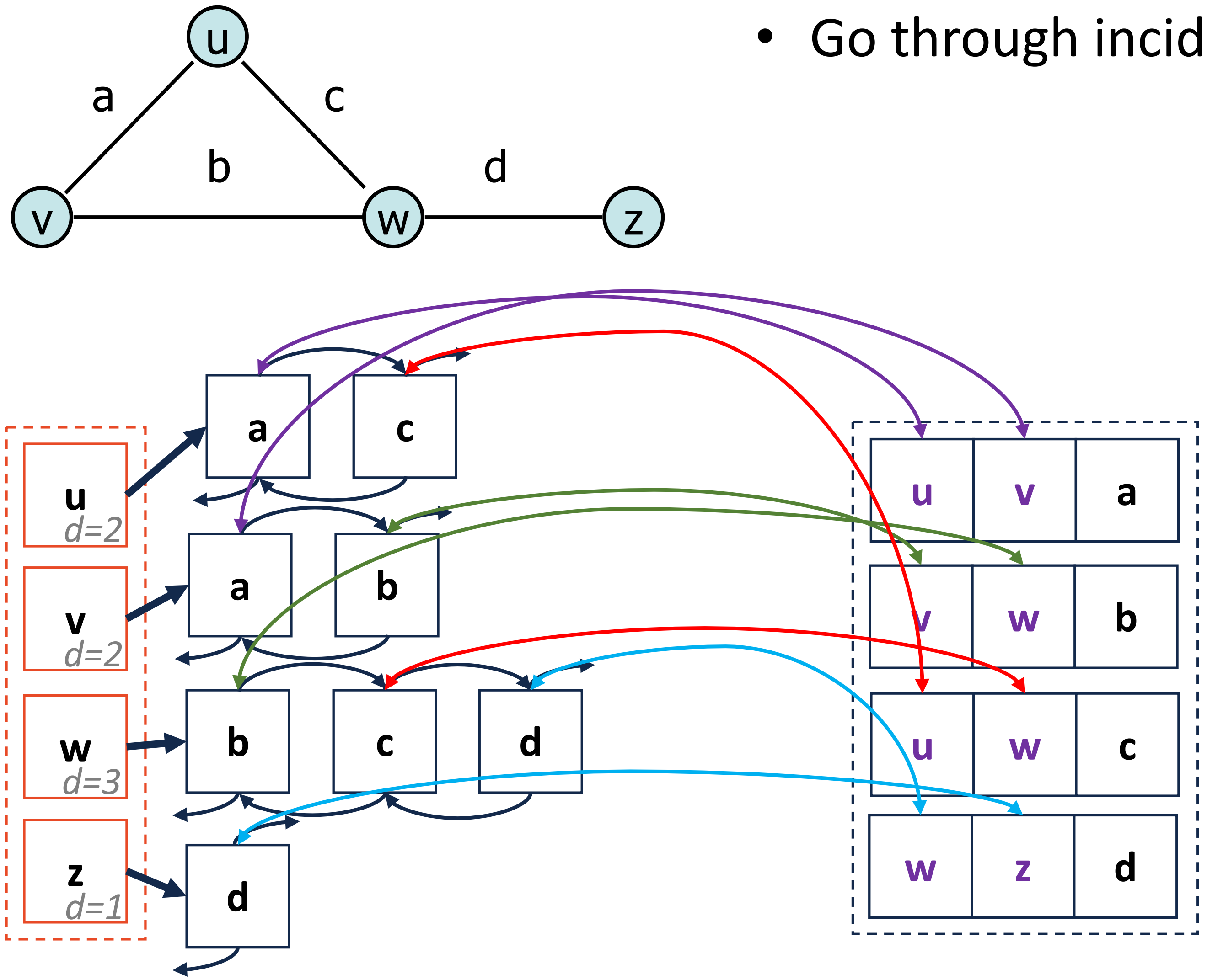
areAdjacent(Vertex v1, Vertex v2): $O(\min(\deg(v1), \deg(v2)))$



- To check adjacent nodes, we need to go through incident edges of one of the vertices:
  - ✓ Choose the vertex with smaller list:
    - $O(\min(\deg(v1), \deg(v2)))$



incidentEdges(Vertex v):



- Go through incident edges of the vertices:

$$O(\deg(v))$$

Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1	n	1
removeVertex(v)	m	n	deg(v)
insertEdge(v, w, k)	1	1	1
removeEdge(v, w)	1	1	1
incidentEdges(v)	m	n	deg(v)
areAdjacent(v, w)	m	1	min(deg(v), deg(w))

Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1 ☺	n	1 ☺
removeVertex(v)	m	n	deg(v) ☺
insertEdge(v, w, k)	1 ☺	1 ☺	1 ☺
removeEdge(v, w)	1 ☺	1 ☺	1 ☺
incidentEdges(v)	m	n	deg(v) ☺
areAdjacent(v, w)	m	1 ☺	min(deg(v), deg(w))

Use cases:

Sparse graphs

The graph is not connected →

$$m < n \Rightarrow \deg(v) < n \Rightarrow 2m < n.$$

Advantage to use: adjacency list implementation

Dense graphs

The graph is almost fully connected →

$$m \sim n^2, \quad \mathrm{degree(v) \sim n}$$

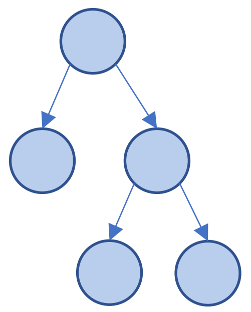
We can use either adjacency list or adjacency matrix.
It depends on the operations we need (are adjacent or insert vertex).

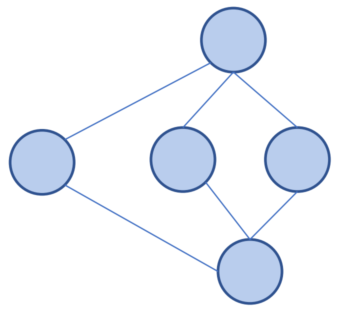
Traversal:

Objective: Visit every vertex and every edge in the graph.

Purpose: Search for interesting sub-structures in the graph.

Tree traversal vs Graph traversal





- Ordered
- Obvious Start
-

- Any order
- Arb. Starting point
- No notion of completeness

Traversal: BFS

```
BFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
       BFS(G, v)
```

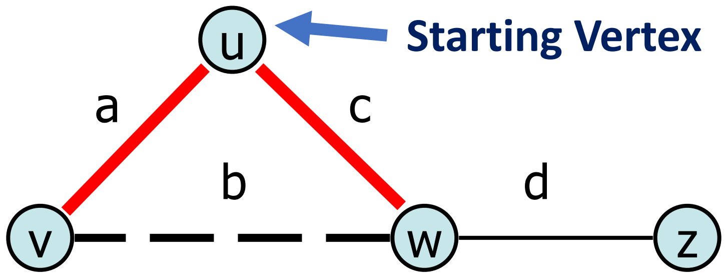
```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

Traversal: BFS

```
BFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
       BFS(G, v)
```

```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

If we have already **visited** vertex w, but have **not explored** the edge (v, w), it means that edge is the cross edge.



Algorithm setup:

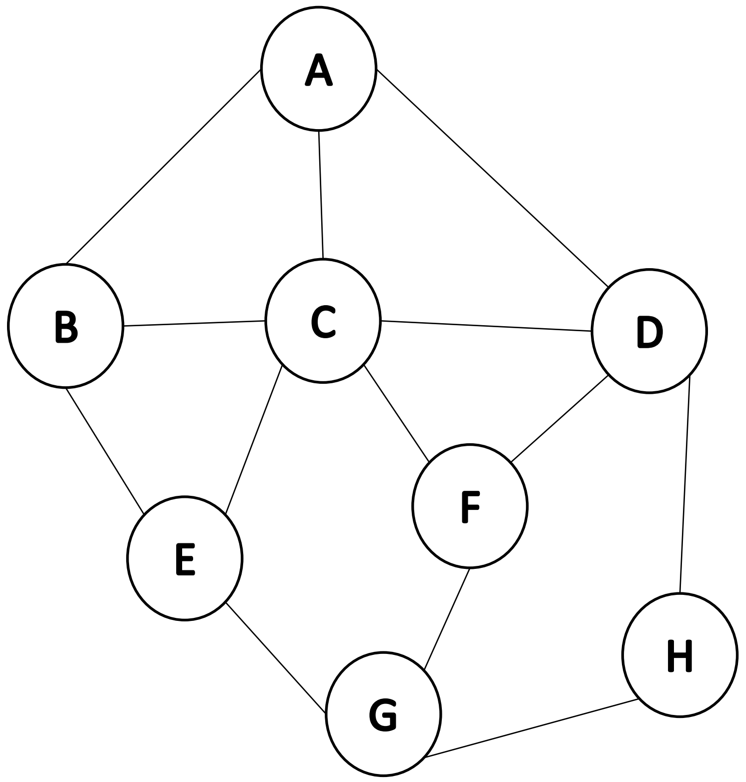
Label each edge:

- Discovery edge (bolded) or
- Cross edge (dashed)

Table of vertices with following features:

- Vertex name - key
- Boolean flag - visited
- Distance it took to get to the vertex
- Predecessor
- List of adjacent vertices

- Queue

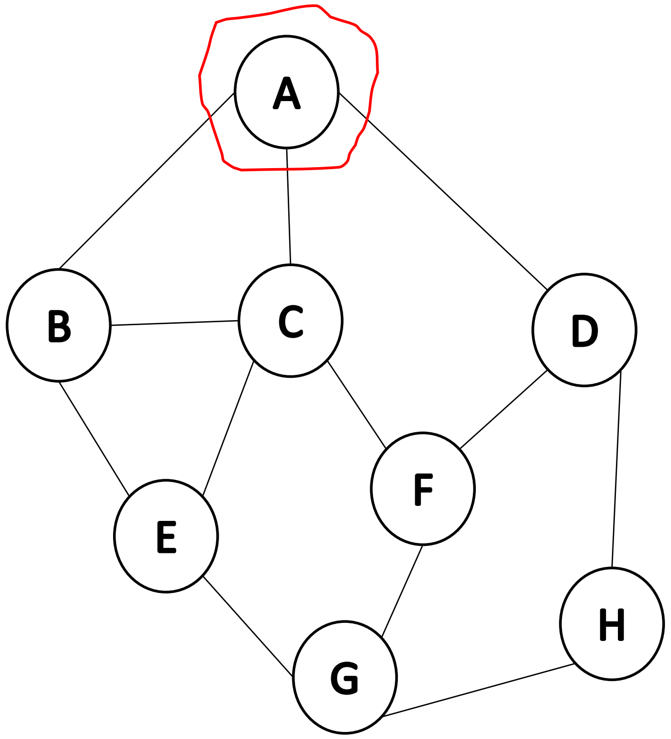


Queue

key	visited	dist.	pred.	adj. vertices
A				C B D
B				A E C
C				A B D E F
D				A C F H
E				B C G
F				C D G
G				E F H
H				D G

- Chose a starting point, add it to the queue, set its visited flag in the table, set distance value to 0, and predecessor value to null.

Starting point - A

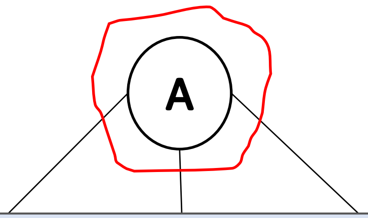


Queue

A

key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B				A E C
C				A B D E F
D				A C F H
E				B C G
F				C D G
G				E F H
H				D G

Starting point - A



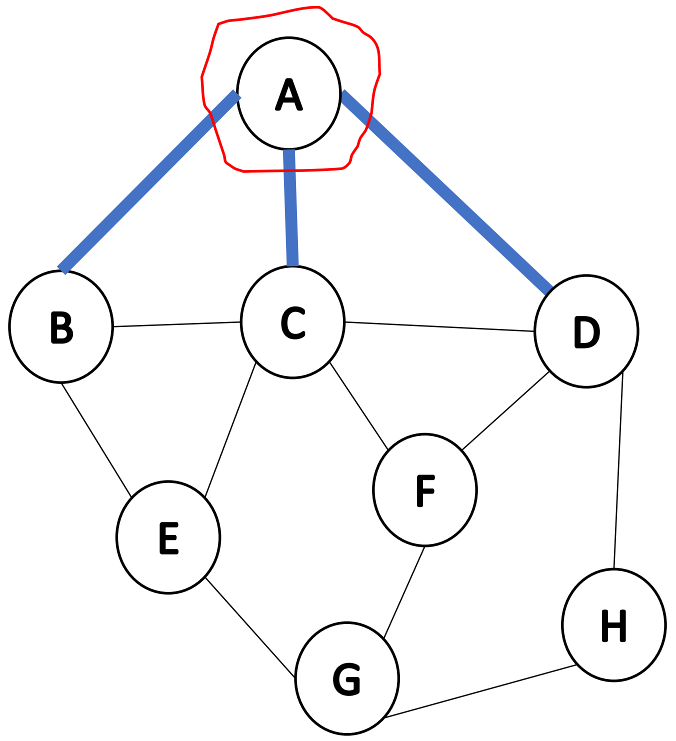
key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B				A E C

Dequeue and loop over the adjacent vertices of the dequeued element.
Examine each adjacent vertex:

- **If the vertex has not been visited**, mark the edge to the vertex as discovery edge; update it's visited flag, distance, and predecessor, and add the vertex to the queue.
- **Otherwise if the edge is not explored** yet just mark the edge as cross edge and move on to the next vertex.

We will dequeue A and examine vertices C, B, and D.

Starting point - A

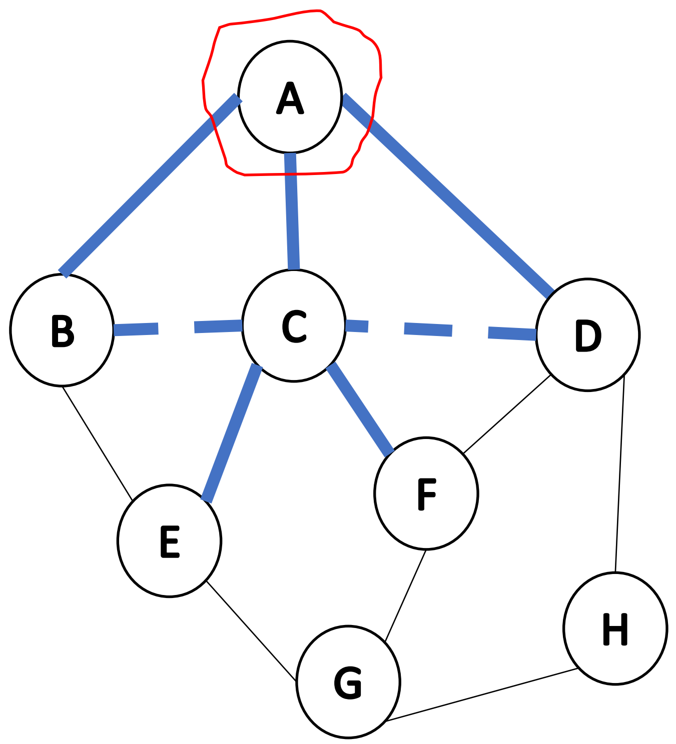


Queue

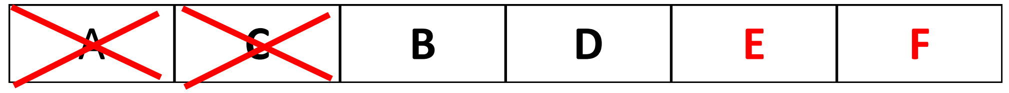
~~A~~	C	B	D

key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E				B C G
F				C D G
G				E F H
H				D G

Starting point - A

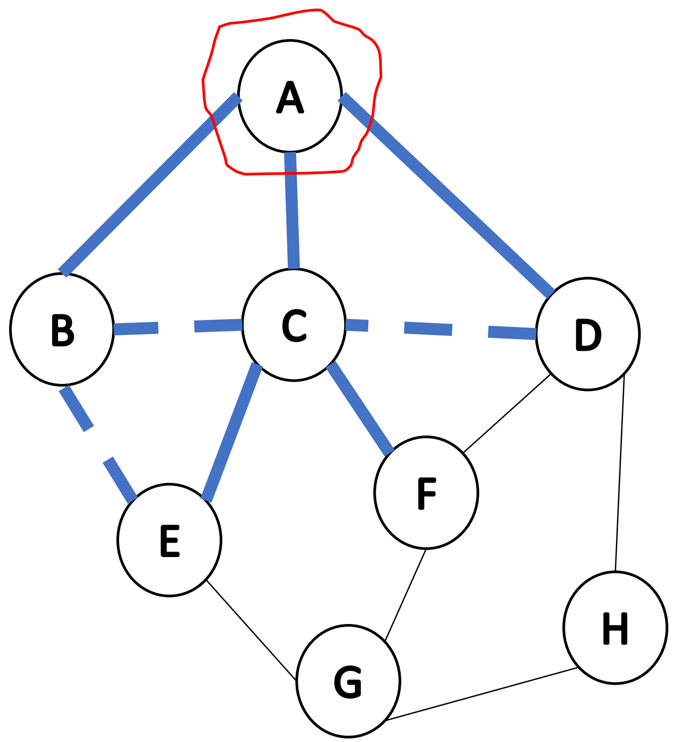


Queue



key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E	✓	2	C	B C G
F	✓	2	C	C D G
G				E F H
H				D G

Starting point - A

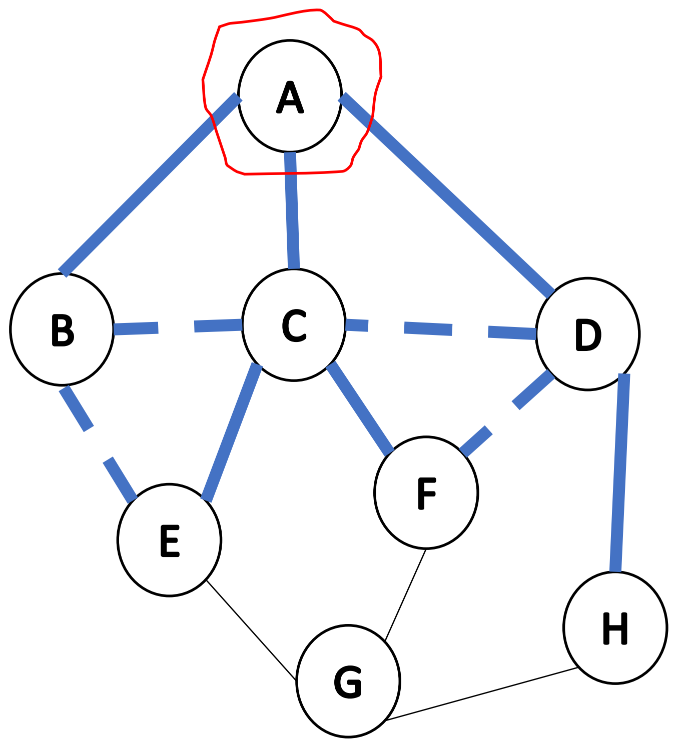


Queue

~~A~~	~~C~~	~~B~~	D	E	F

key	visited	dist.	pred.	adj. vertices
A	√	0	null	C B D
B	√	1	A	A E C
C	√	1	A	A B D E F
D	√	1	A	A C F H
E	√	2	C	B C G
F	√	2	C	C D G
G				E F H
H				D G

Starting point - A

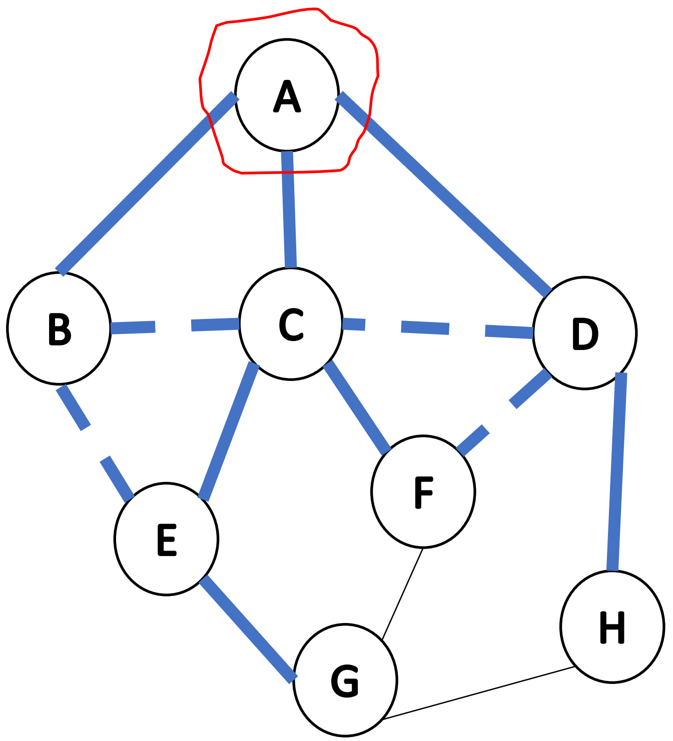


Queue

~~A~~	~~C~~	~~B~~	~~D~~	E	F	H

key	visited	dist.	pred.	adj. vertices
A	√	0	null	C B D
B	√	1	A	A E C
C	√	1	A	A B D E F
D	√	1	A	A C F H
E	√	2	C	B C G
F	√	2	C	C D G
G				E F H
H	√	2	D	D G

Starting point - A

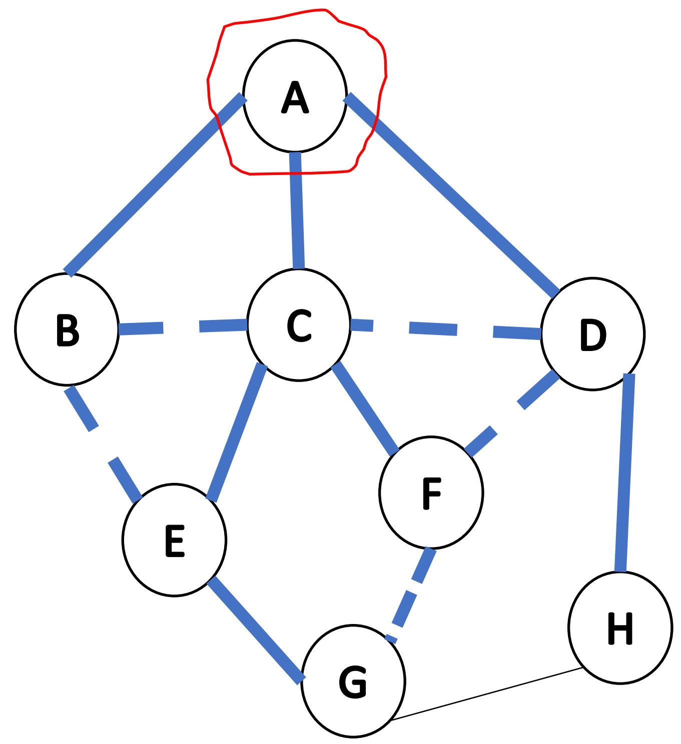


Queue

~~A~~	~~C~~	~~B~~	~~D~~	~~E~~	F	H	G

key	visited	dist.	pred.	adj. vertices
A	√	0	null	C B D
B	√	1	A	A E C
C	√	1	A	A B D E F
D	√	1	A	A C F H
E	√	2	C	B C G
F	√	2	C	C D G
G	√	3	E	E F H
H	√	2	D	D G

Starting point - A

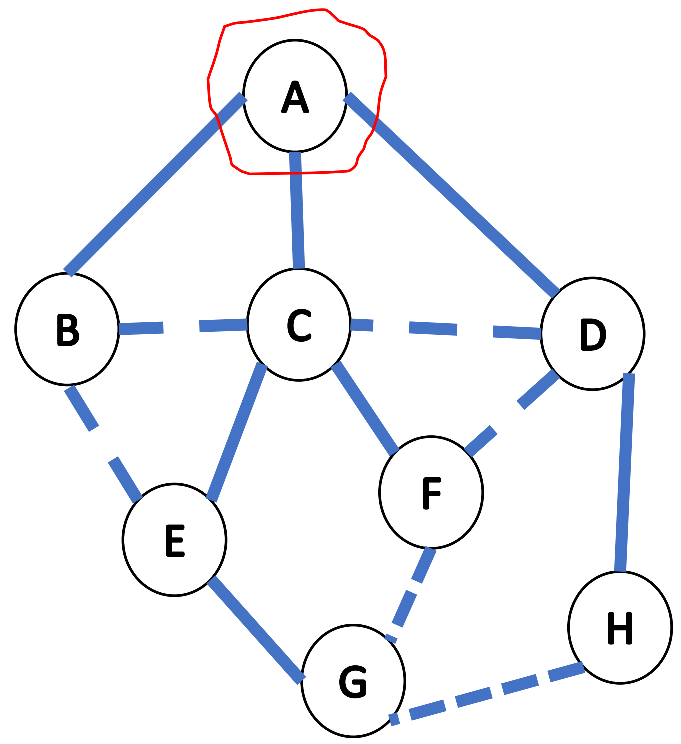


Queue

~~A~~	~~C~~	~~B~~	~~D~~	~~E~~	~~F~~	H	G

key	visited	dist.	pred.	adj. vertices
A	√	0	null	C B D
B	√	1	A	A E C
C	√	1	A	A B D E F
D	√	1	A	A C F H
E	√	2	C	B C G
F	√	2	C	C D G
G	√	3	E	E F H
H	√	2	D	D G

Starting point - A

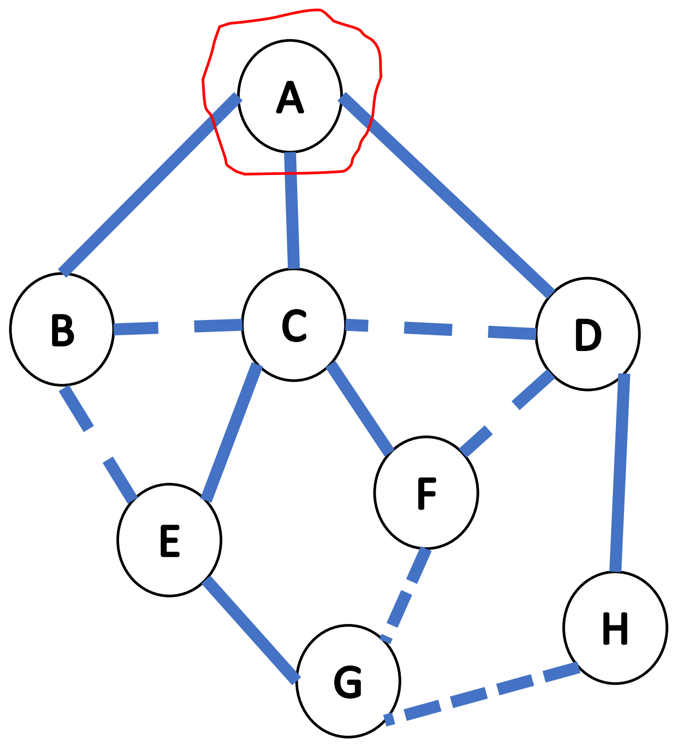


Queue

~~A~~	~~C~~	~~B~~	~~D~~	~~E~~	~~F~~	~~H~~	G

key	**visited**	**dist.**	**pred.**	**adj. vertices**
A	√	0	null	C B D
B	√	1	A	A E C
C	√	1	A	A B D E F
D	√	1	A	A C F H
E	√	2	C	B C G
F	√	2	C	C D G
G	√	3	E	E F H
H	√	2	D	D G

Starting point - A



Queue

~~A~~	~~C~~	~~B~~	~~D~~	~~E~~	~~F~~	~~H~~	~~G~~

key	visited	dist.	pred.	adj. vertices
A	√	0	null	C B D
B	√	1	A	A E C
C	√	1	A	A B D E F
D	√	1	A	A C F H
E	√	2	C	B C G
F	√	2	C	C D G
G	√	3	E	E F H
H	√	2	D	D G

Traversal: BFS

```
BFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
       BFS(G, v)
```

```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

Our implementation handles disjoint graphs.

How do we use this to count components?

Add component counter before BFS call;

Traversal: BFS

```
BFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
       comps++;
       BFS(G, v)
```

```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

Our implementation handles disjoint graphs.

How do we use this to count components?

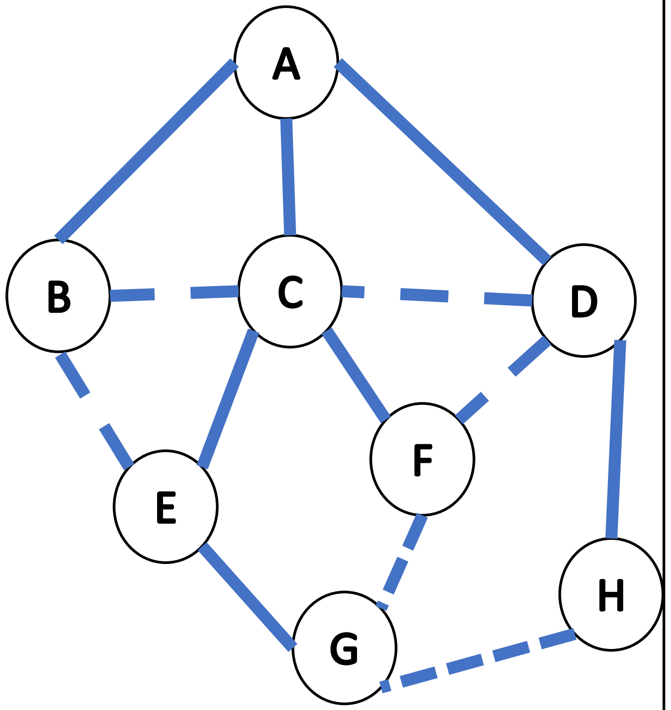
Add component counter before BFS call;

BFS Analysis

Q: Does our implementation detect a cycle?

- ***How do we update our code to detect a cycle?***

Yes. Existence of at least one cross edge guarantees cycle.



```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

Running time of BFS **O(n+m)**

```
 1  BFS(G):
 2    Input: Graph, G
 3    Output: A labeling of the edges on
 4        G as discovery and cross edges
 5
 6    foreach (Vertex v : G.vertices()):
 7      setLabel(v, UNEXPLORED)
 8    foreach (Edge e : G.edges()):
 9      setLabel(e, UNEXPLORED)
10    foreach (Vertex v : G.vertices()):
11      if getLabel(v) == UNEXPLORED:
12         BFS(G, v)
```

```
14  BFS(G, v):
15    Queue q
16    setLabel(v, VISITED)
17    q.enqueue(v)
18
19    while !q.empty():
20      v = q.dequeue()
21      foreach (Vertex w : G.adjacent(v)):
22        if getLabel(w) == UNEXPLORED:
23           setLabel(v, w, DISCOVERY)
24           setLabel(w, VISITED)
25           q.enqueue(w)
26        elseif getLabel(v, w) == UNEXPLORED:
27           setLabel(v, w, CROSS)
```

6-7: O(n)
8-9: O(m)
If the graph is connected component we call BFS only once (**12**)
15-17: O(1)

19 -> while runs n times
20 – O(1)
21: degree(v) times, $\sum_{v\epsilon V} \deg(v) = 2\ * m$ =>
19-27: O(n+m) time
22-27: O(1)

Running time of BFS **O(n+m)**

```
BFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
       BFS(G, v)
```

```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

This is optimal running time because we know we have to visit every edge and vertex, therefore we cannot do better than O(n+m).

BFS Observations

Q: What is a shortest path from **A** to **H**?

Path: A,D,H

Q: What is a shortest path from **E** to **H**?

No information about this.

BFS finds shortest path only from starting vertex (in graphs without weights) ;

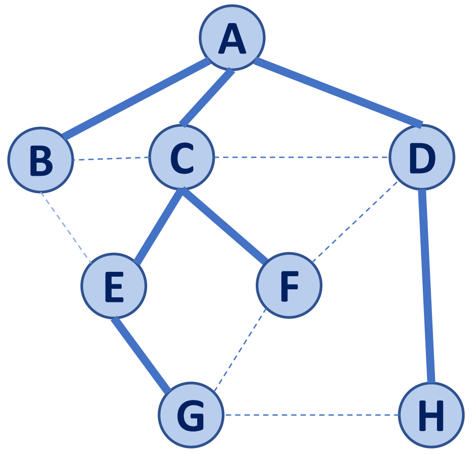
Q: How does a cross edge relate to **d**?

$$|\Delta d| \leq 1$$

Q: What structure is made from discovery edges?

We get new graph structure: spanning tree!

d	p	v	Adjacent
0	A	A	C B D
1	A	B	A C E
1	A	C	B A D E F
1	A	D	A C F H
2	C	E	B C G
2	C	F	C D G
3	E	G	E F H
2	D	H	D G



DFS – Depth First Search

Algorithm setup:

Everything is the same as BFS except for:

- We will use stack instead of a queue.
- We will label cross edges as back edges.

Algorithm setup:

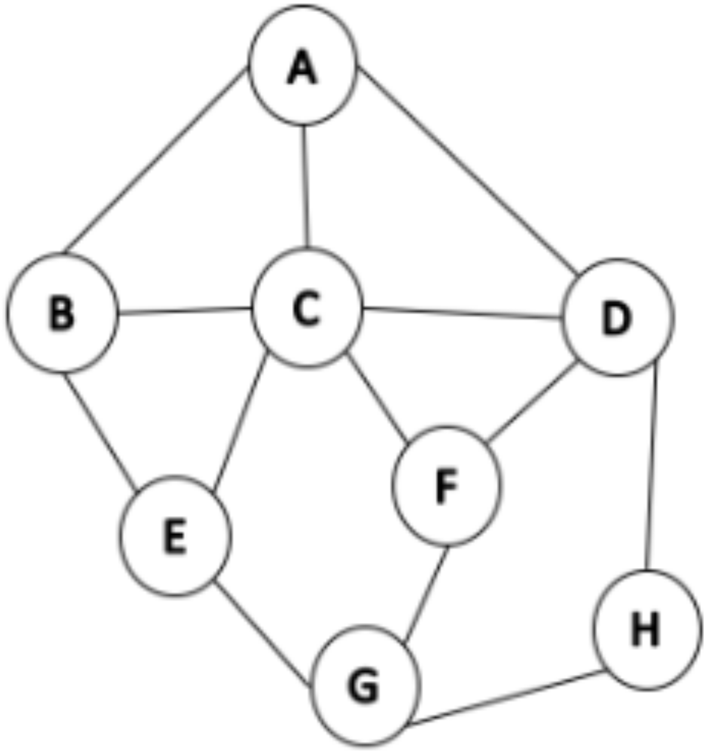
Label each edge:

- Discovery edge (bolded) or
- **back edge (dashed)**

Table of vertices with following features:

- Vertex name - key
- Boolean flag - visited
- Distance it took to get to the vertex
- Predecessor
- List of adjacent vertices

- **Stack**

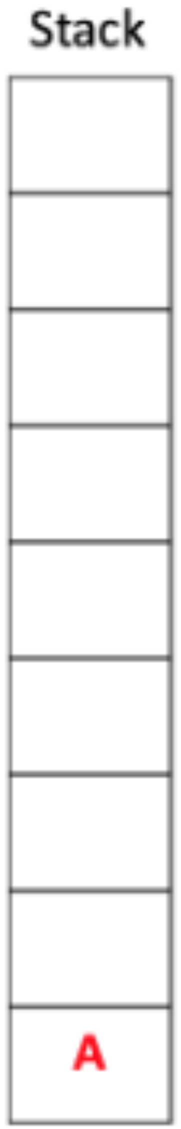


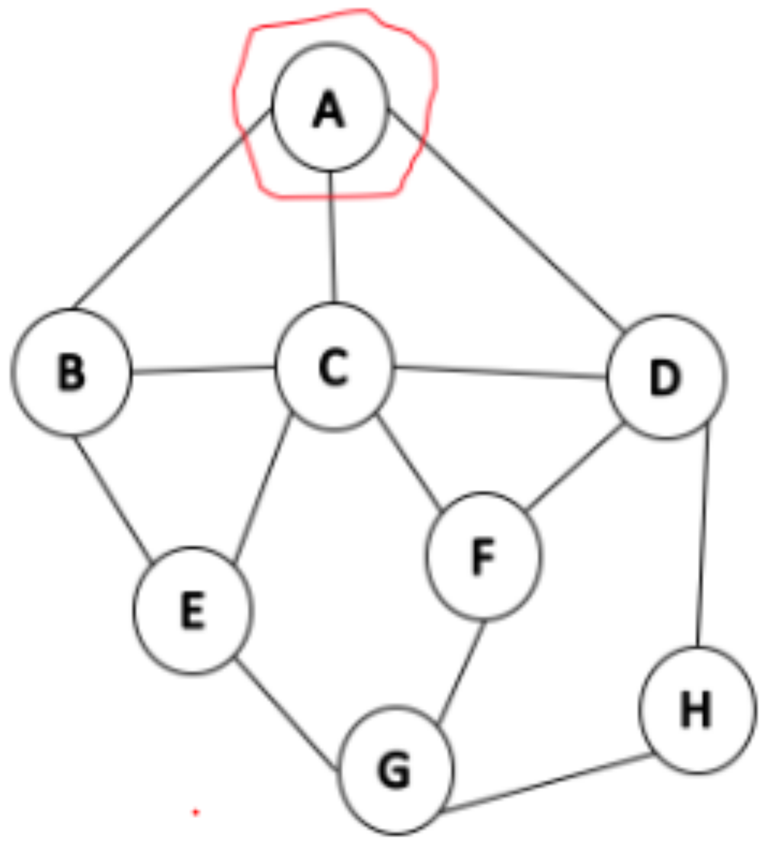
key	visited	dist.	pred.	adj. vertices
A				C B D
B				A E C
C				A B D E F
D				A C F H
E				B C G
F				C D G
G				E F H
H				D G

Algorithm logic:

- We will start with A and run the same algorithm as we did for BFS but we will be using stack.

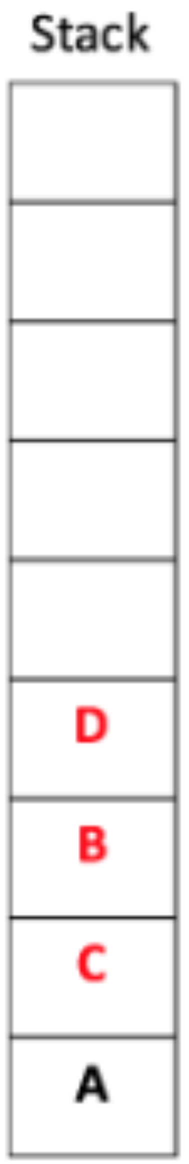
Add A to the stack and update A

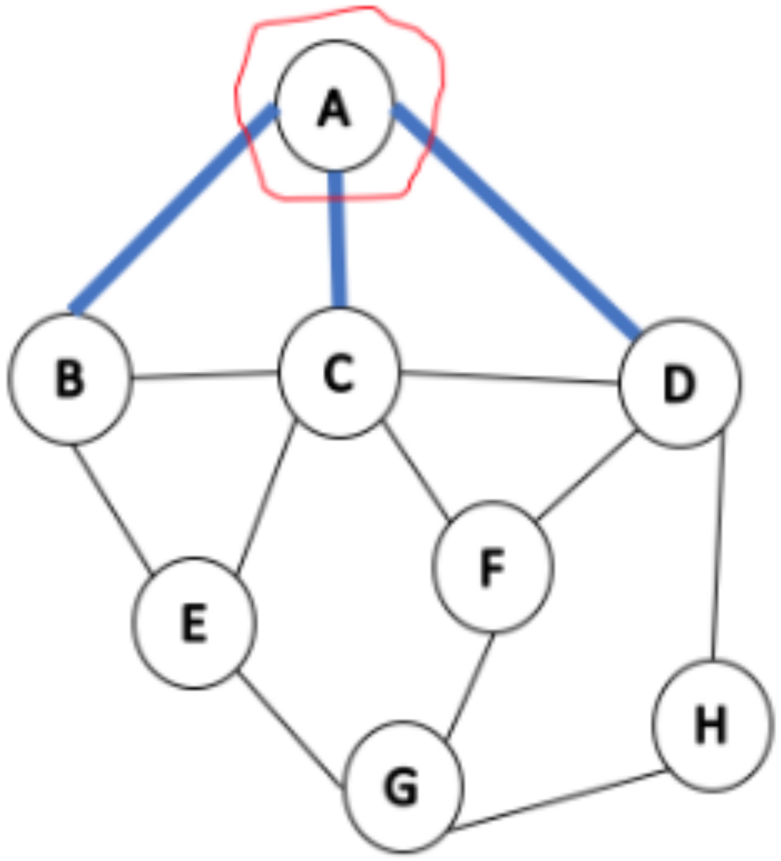




key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B				A E C
C				A B D E F
D				A C F H
E				B C G
F				C D G
G				E F H
H				D G

Pop A and examine adjacent vertices.

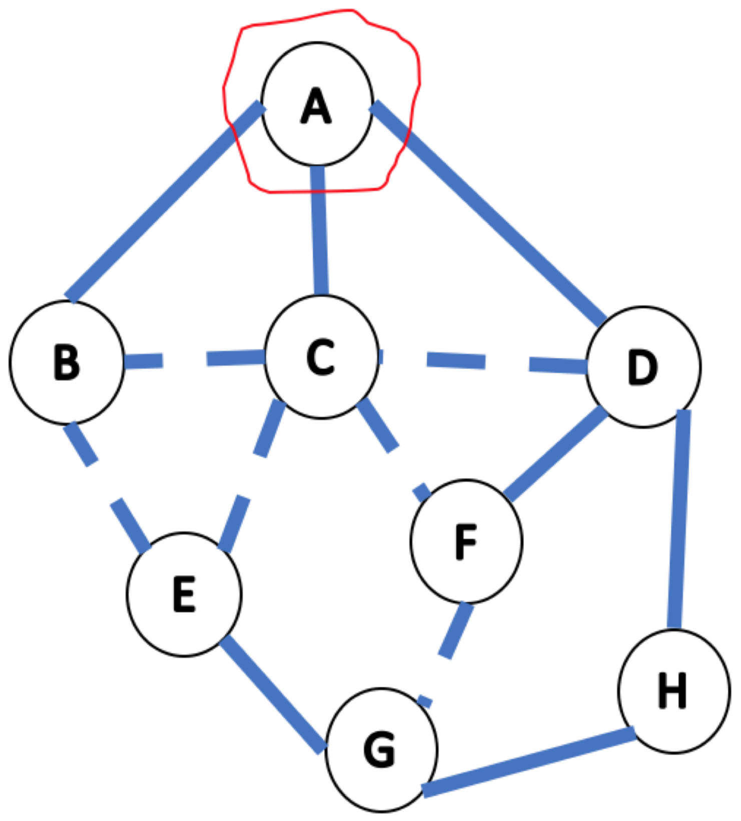




key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E				B C G
F				C D G
G				E F H
H				D G

One Eternity Later

After visiting all vertices and edges, we get the following result:

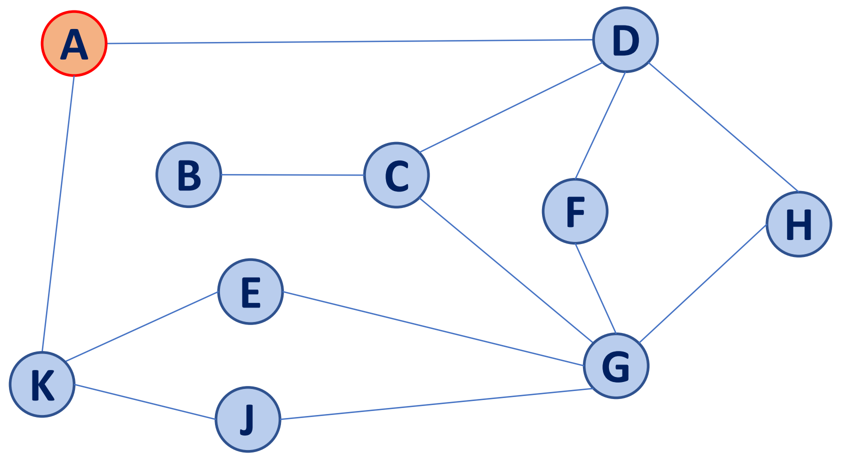


key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E	✓	4	G	B C G
F	✓	2	D	C D G
G	✓	3	H	E F H
H	✓	2	D	D G

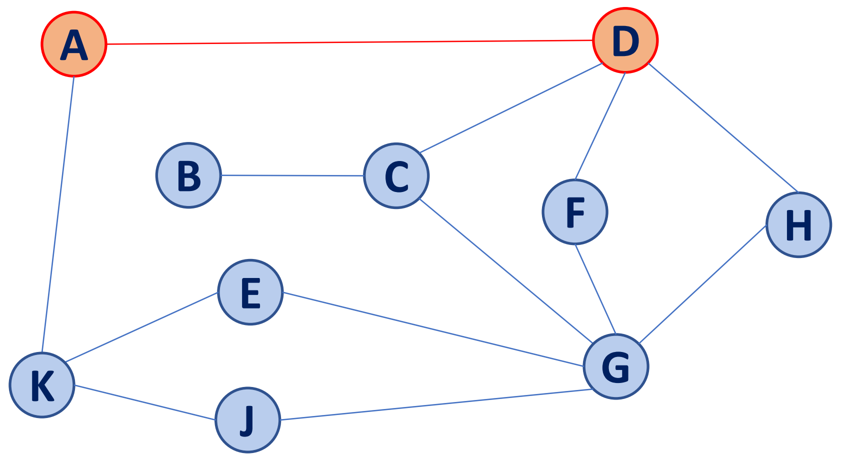
```
DFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and back edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
      DFS(G, v)
```

```
DFS(G, v):
  ~~Queue q~~
  setLabel(v, VISITED)
  ~~q.enqueue(v)~~

  ~~while !q.empty():~~
    ~~v = q.dequeue()~~
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
        setLabel(v, w, DISCOVERY)
        ~~setLabel(w, VISITED)~~
        DFS(G, w)
      elseif getLabel(v, w) == UNEXPLORED:
        setLabel(v, w, BACK)
```

DFS with recursion:

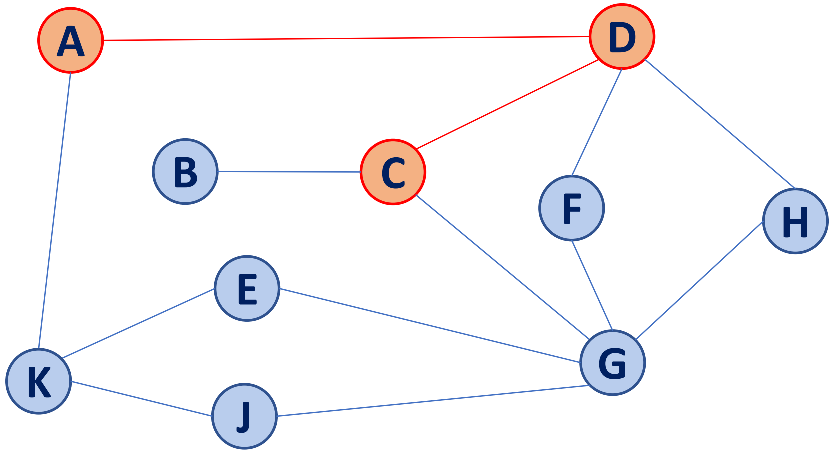


We visit D first and we are immediately recusing from D.

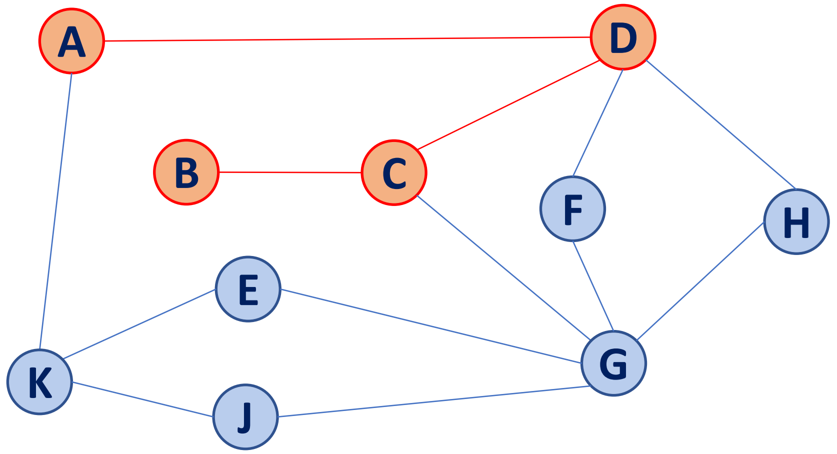


Order of vertices does not matter.

DFS with recursion:

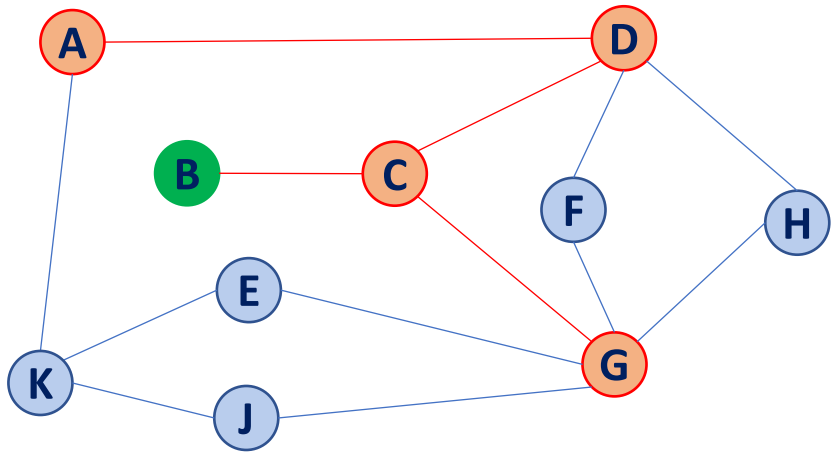


Next we visit C first and we are immediately recusing from C.

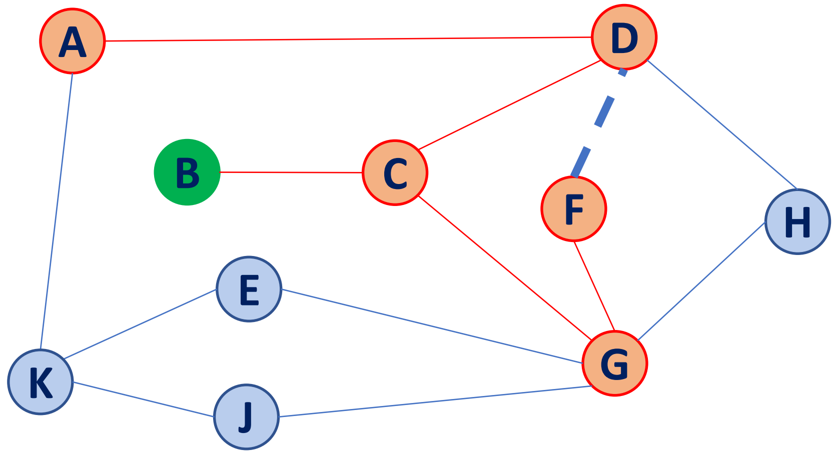


Next we visit B first.
We visited all neighbors for B, so we will go back to C.

DFS with recursion:

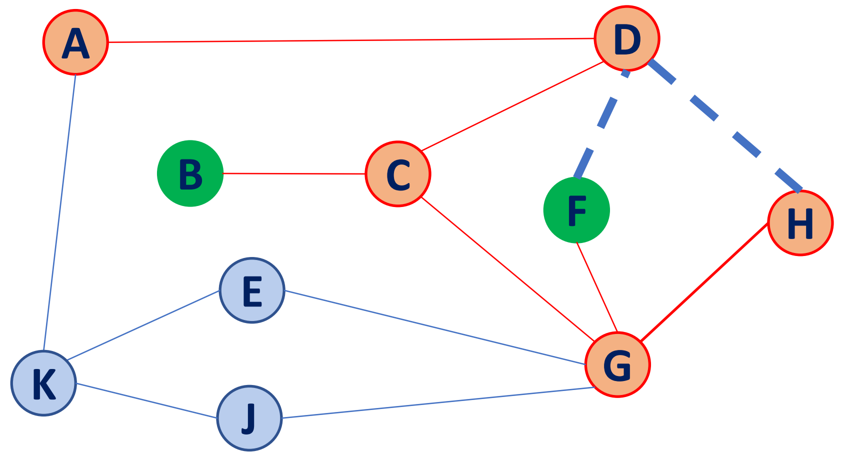


Next we visit G first and we are immediately recusing from G.

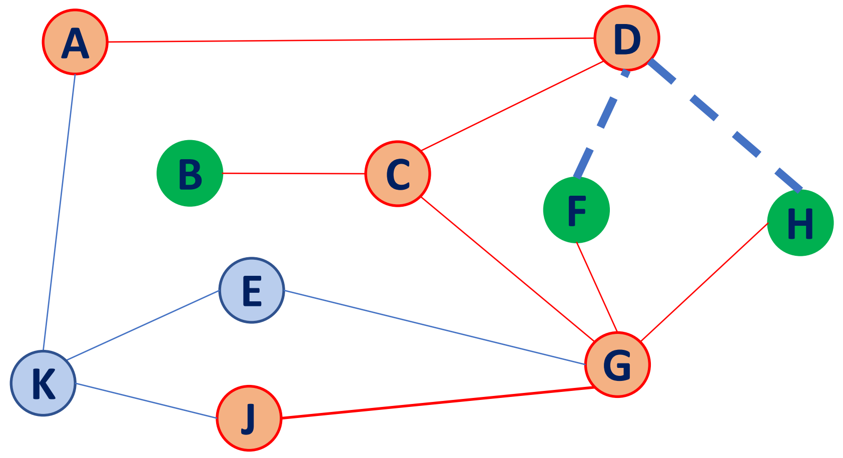


Next we visit F first.
Since D is already visited (F,D) is labeled as back edge.
F is done and we go back to G.

DFS with recursion:

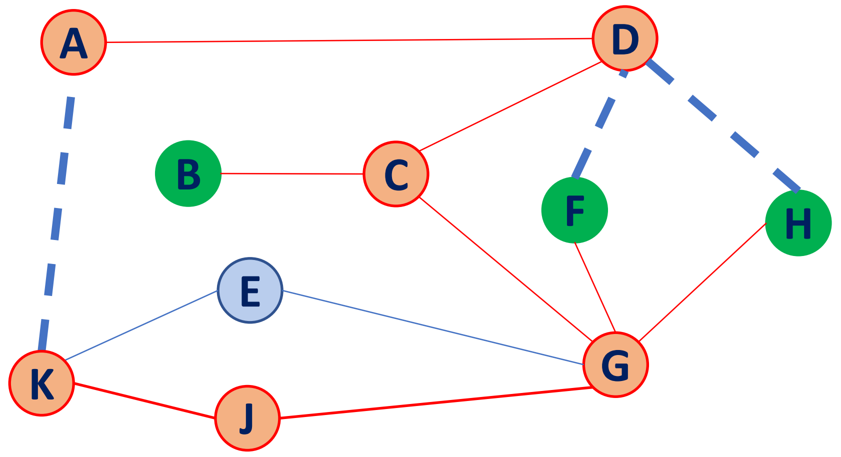


Next we visit H and we label another back edge (H,D). H will be done, we will go back to G.

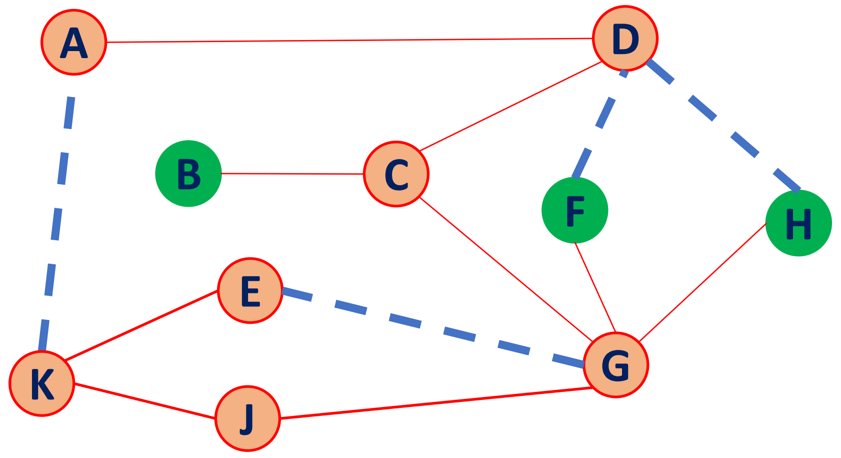


Next we visit J.

DFS with recursion:



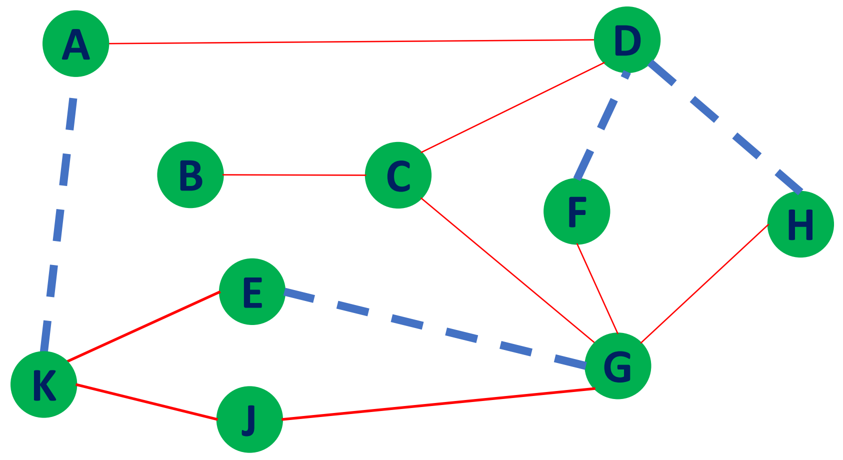
Next we visit K.
(A,K) labeled as back edge.



Next we visit E.
(E,G) becomes back edge and E will be done.

* You should also keep track of distance and parents.

DFS with recursion:



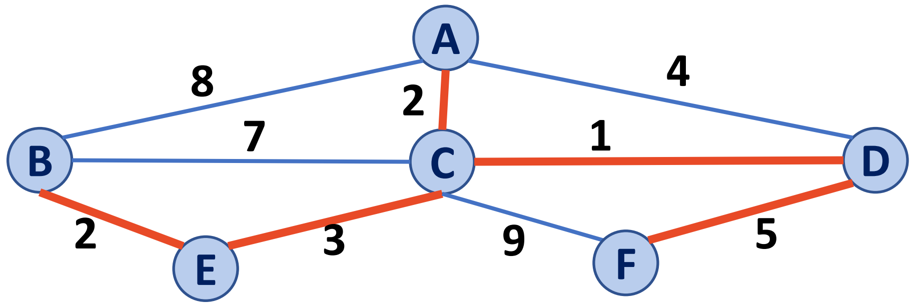
- Back edge is getting us closer to starting vertex;
- Existence of back edges means there is a cycle;
- Discovery edges gives us spanning tree;
- DFS can gives us component count;

Minimum Spanning Tree Algorithms

Input: Connected, undirected graph **G** with edge weights (unconstrained, but must be additive)

Output: A graph G' with the following properties:

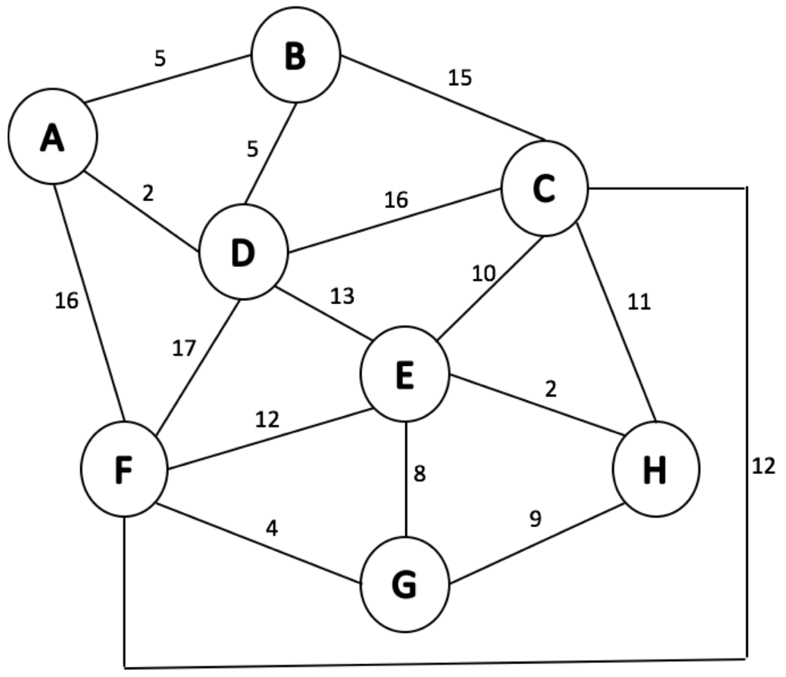
- G' is a spanning graph of G
- G' is a tree (connected, acyclic)
- G' has a minimal total weight among all spanning trees

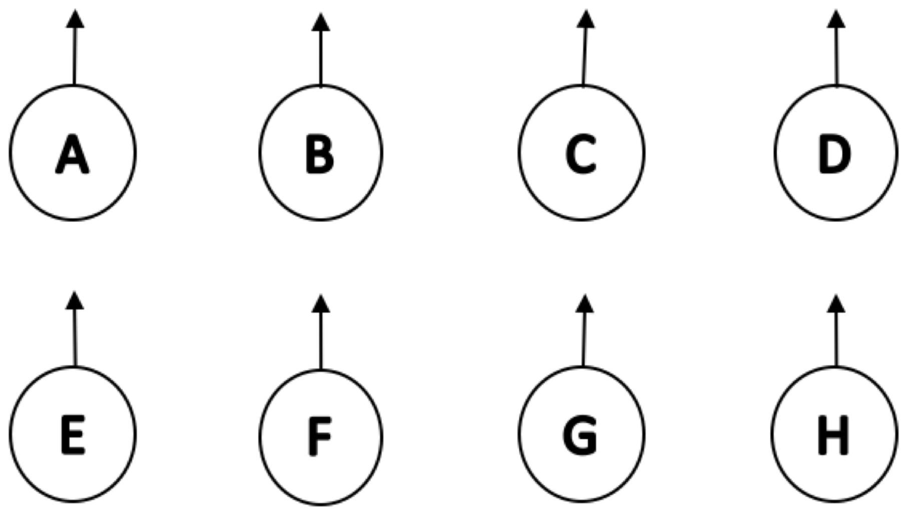


Kruskal’s Algorithm

Algorithm setup:

- Maintain a list of edges sorted by weight in increasing order → min heap.
- Initialize a disjoint set (up tree) for each vertex.





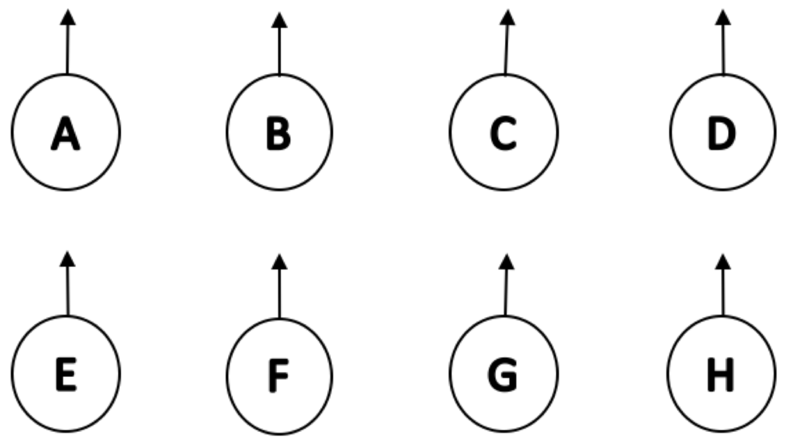
Sorted list of edges:

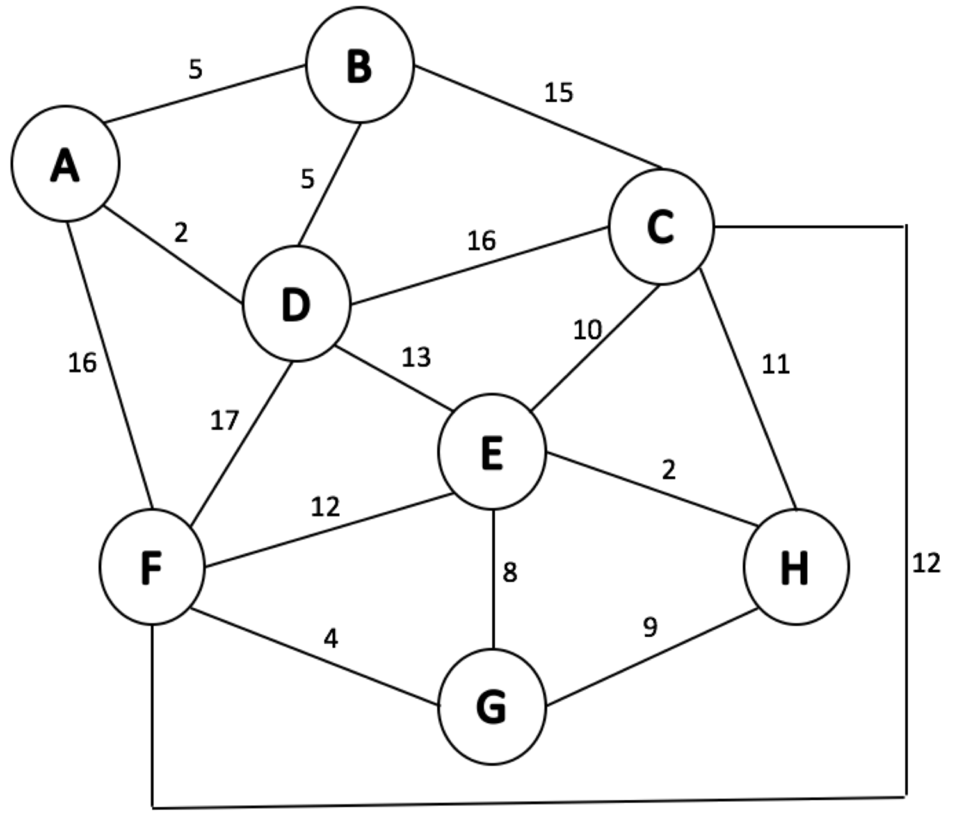
(A, D)
(E, H)
(F, G)
(A, B)
(B, D)
(G, E)
(G, H)
(E, C)
(C, H)
(E, F)
(F, C)
(D, E)
(B, C)
(C, D)
(A, F)
(D, F)

Kruskal's Algorithm

- Remove minimum from the heap;
- Check that the two vertices, that form the removed edge, are in different disjoint sets.
    - If they are, add the edge to the spanning tree and union the two sets.
    - Otherwise, ignore that edge and move on.

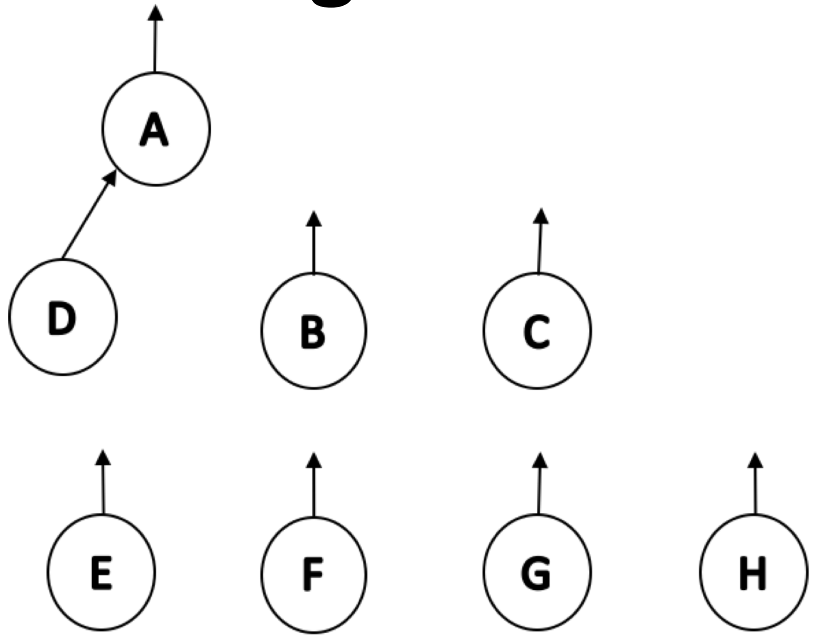
Kruskal’s Algorithm

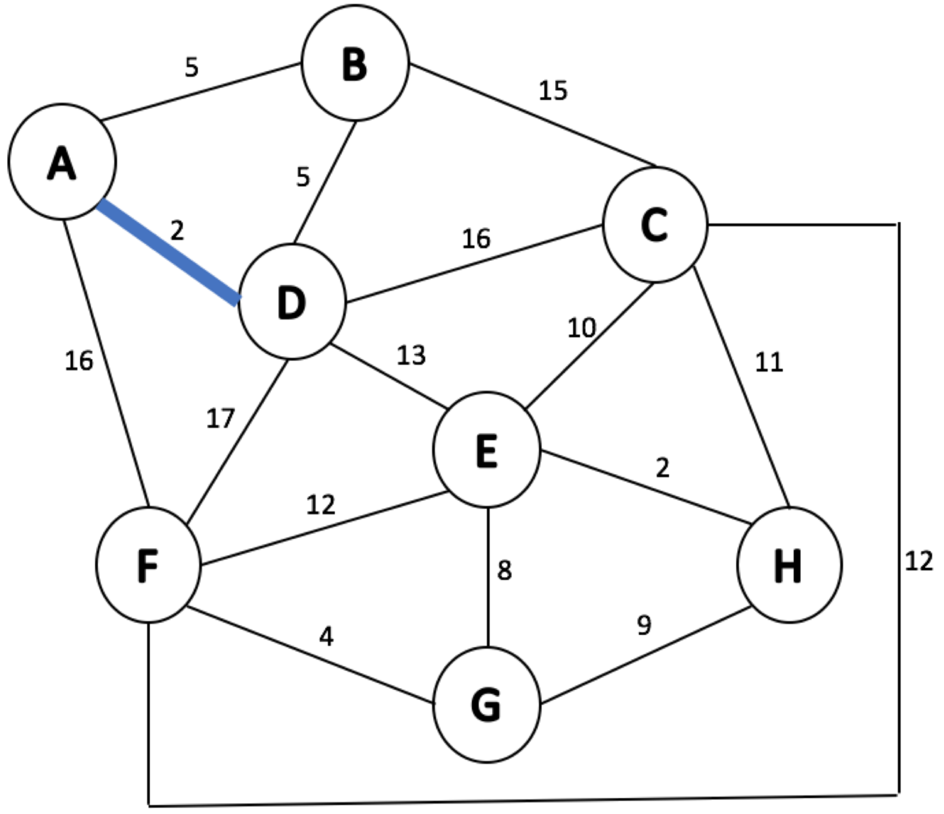




(A,D)
(E,H)
(F,G)
(A,B)
(B,D)
(G,E)
(G,H)
(E,C)
(C,H)
(E,F)
(F,C)
(D,E)
(B,C)
(C,D)
(A,F)
(D,F)

Kruskal’s Algorithm

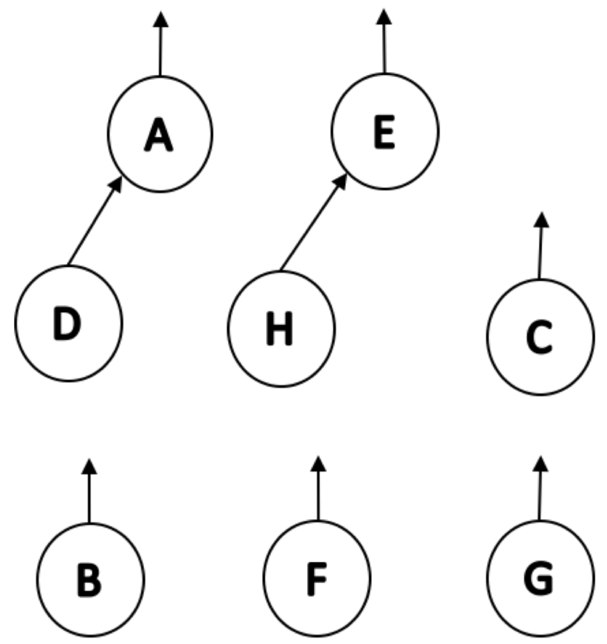


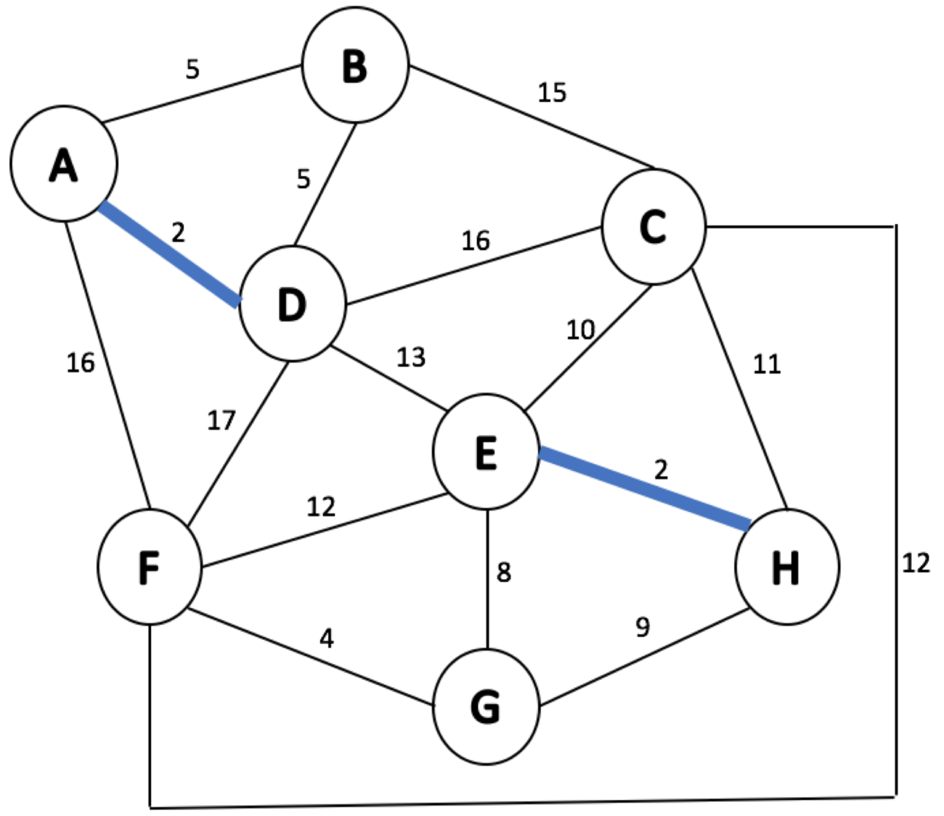


~~(A,D)~~
(E,H)
(F,G)
(A,B)
(B,D)
(G,E)
(G,H)
(E,C)
(C,H)
(E,F)
(F,C)
(D,E)
(B,C)
(C,D)
(A,F)
(D,F)

- remove edge (A, D) from the heap.
- Vertex A and vertex D are in different sets. Therefore, we can add edge (A, D) and union sets {A} and {D}.

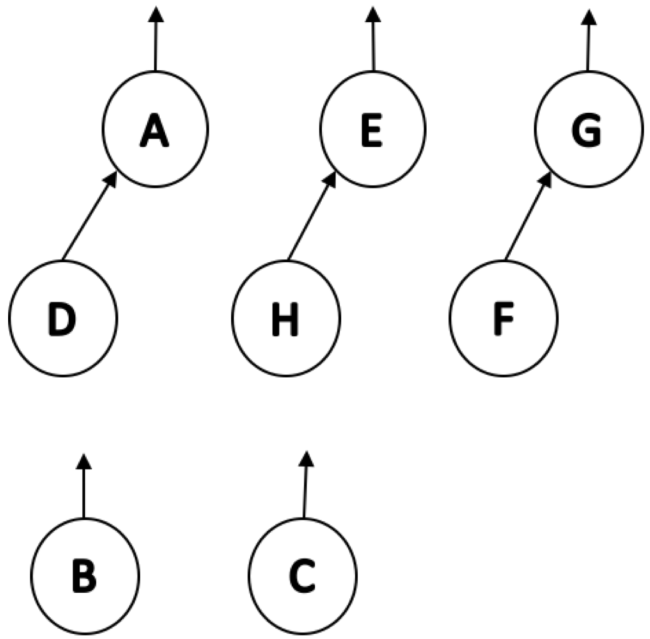
Kruskal’s Algorithm

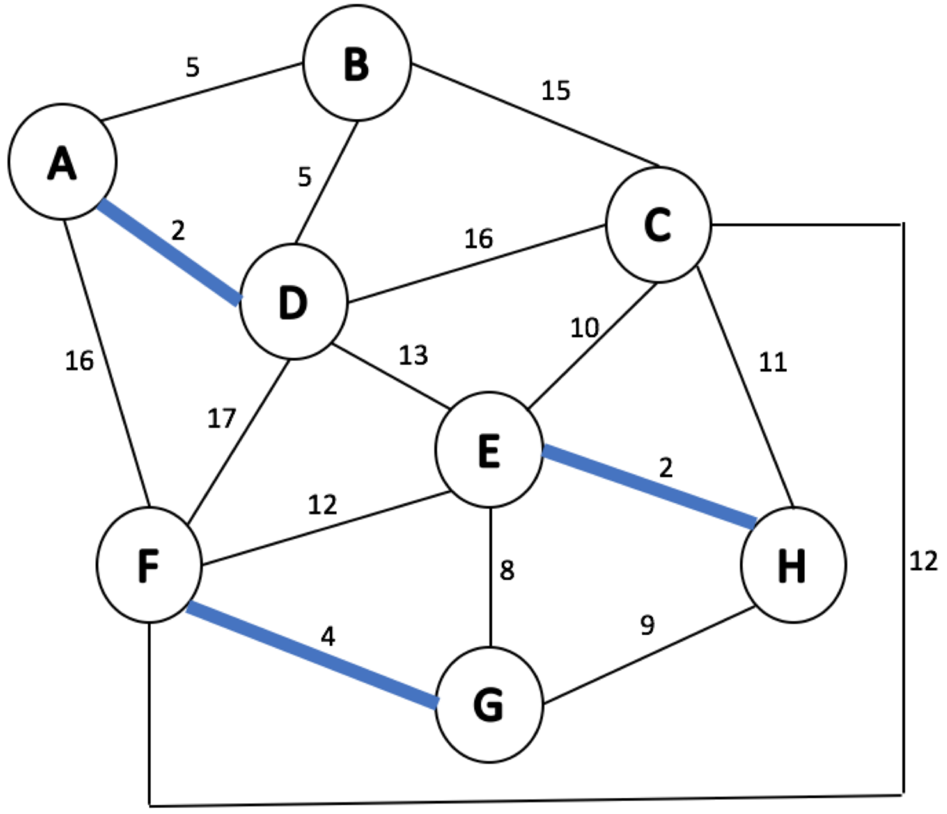




~~(A,D)~~
~~(E,H)~~
(F,G)
(A,B)
(B,D)
(G,E)
(G,H)
(E,C)
(C,H)
(E,F)
(F,C)
(D,E)
(B,C)
(C,D)
(A,F)
(D,F)

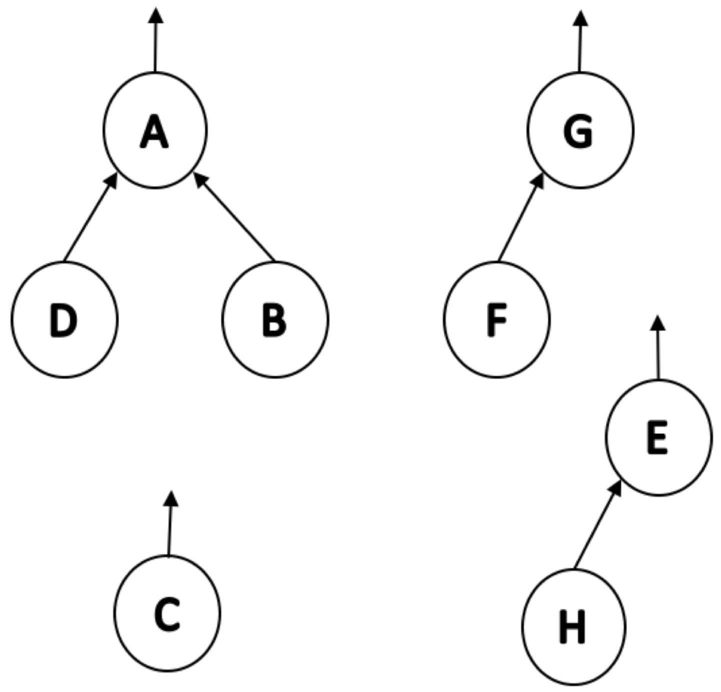
Kruskal's Algorithm

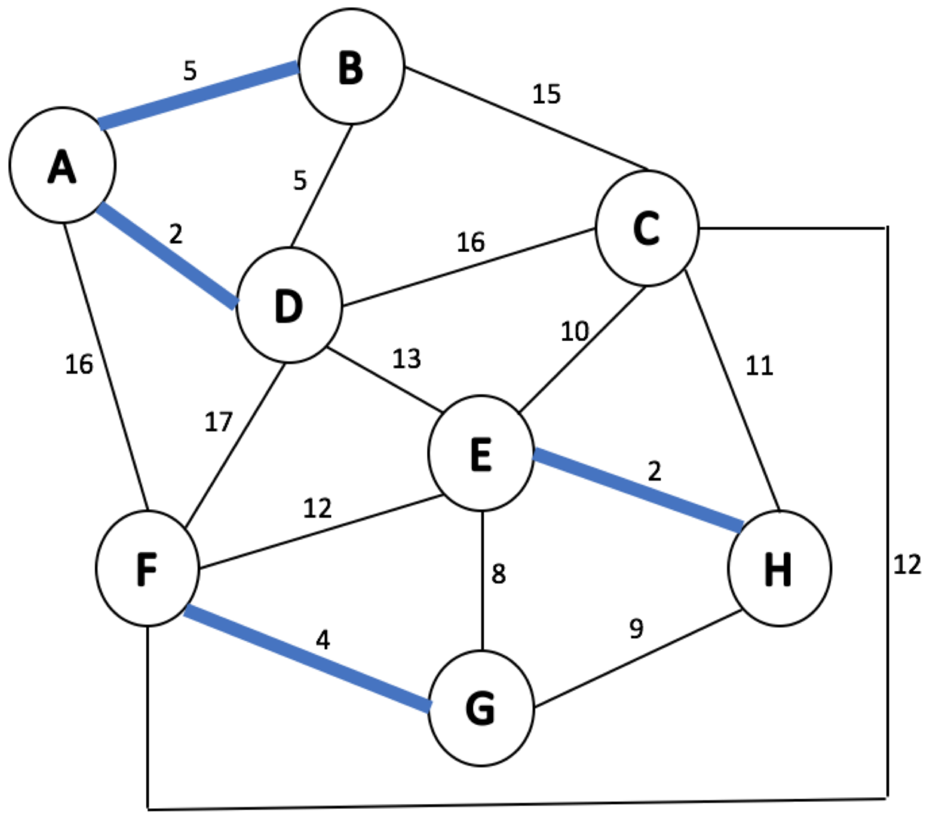




~~(A,D)~~
~~(E,H)~~
~~(F,G)~~
(A,B)
(B,D)
(G,E)
(G,H)
(E,C)
(C,H)
(E,F)
(F,C)
(D,E)
(B,C)
(C,D)
(A,F)
(D,F)

Kruskal’s Algorithm

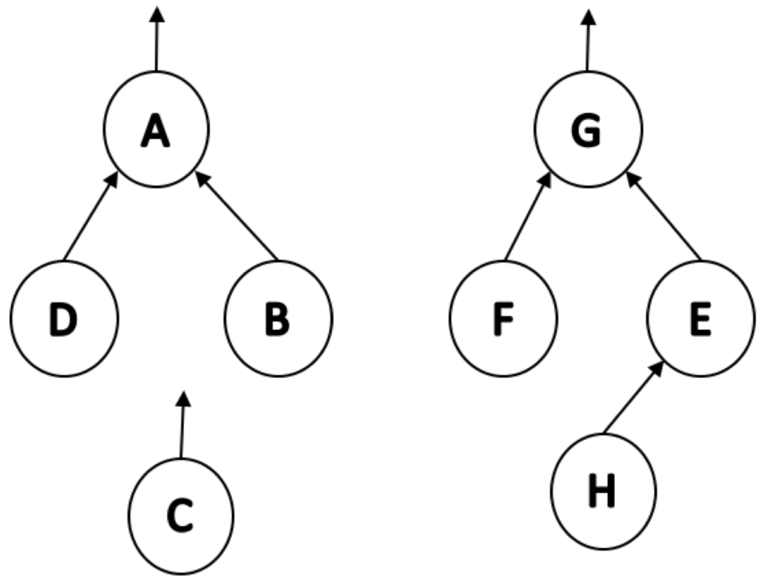


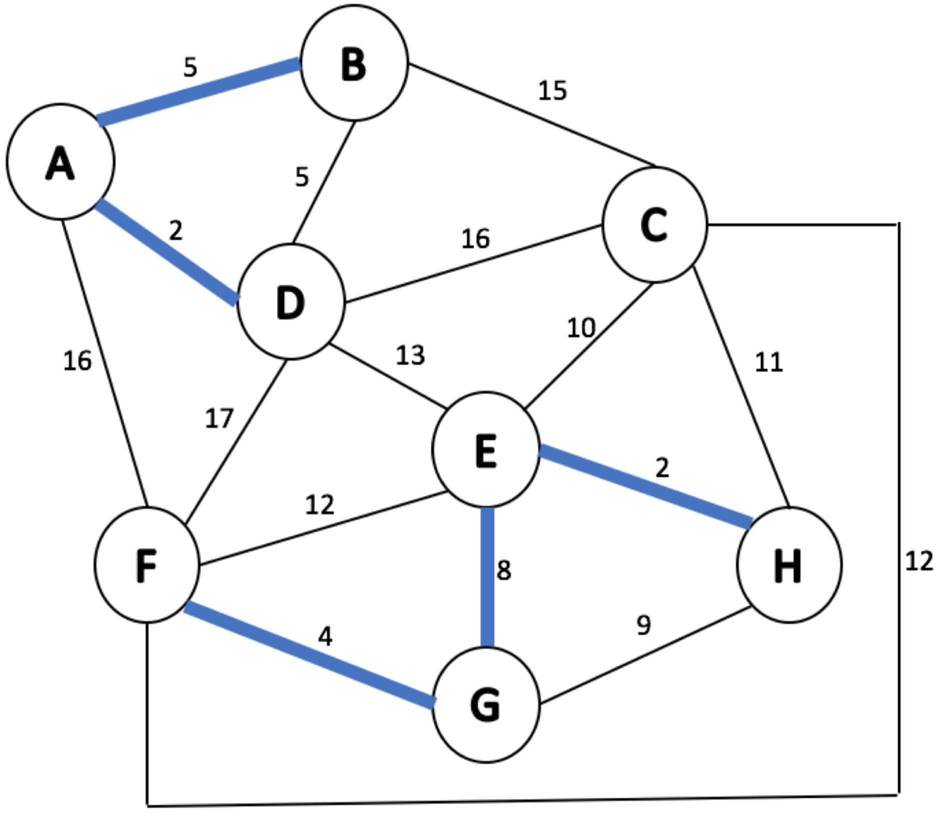


~~(A,D)~~
~~(E,H)~~
~~(F,G)~~
~~(A,B)~~
(B,D)
(G,E)
(G,H)
(E,C)
(C,H)
(E,F)
(F,C)
(D,E)
(B,C)
(C,D)
(A,F)
(D,F)

Next:
We skip (B,D) since they are in the same set.

Kruskal’s Algorithm

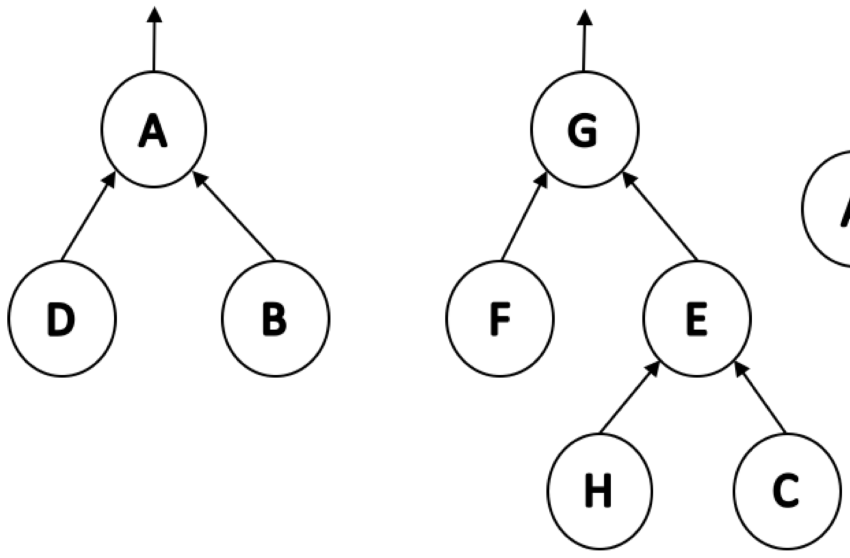


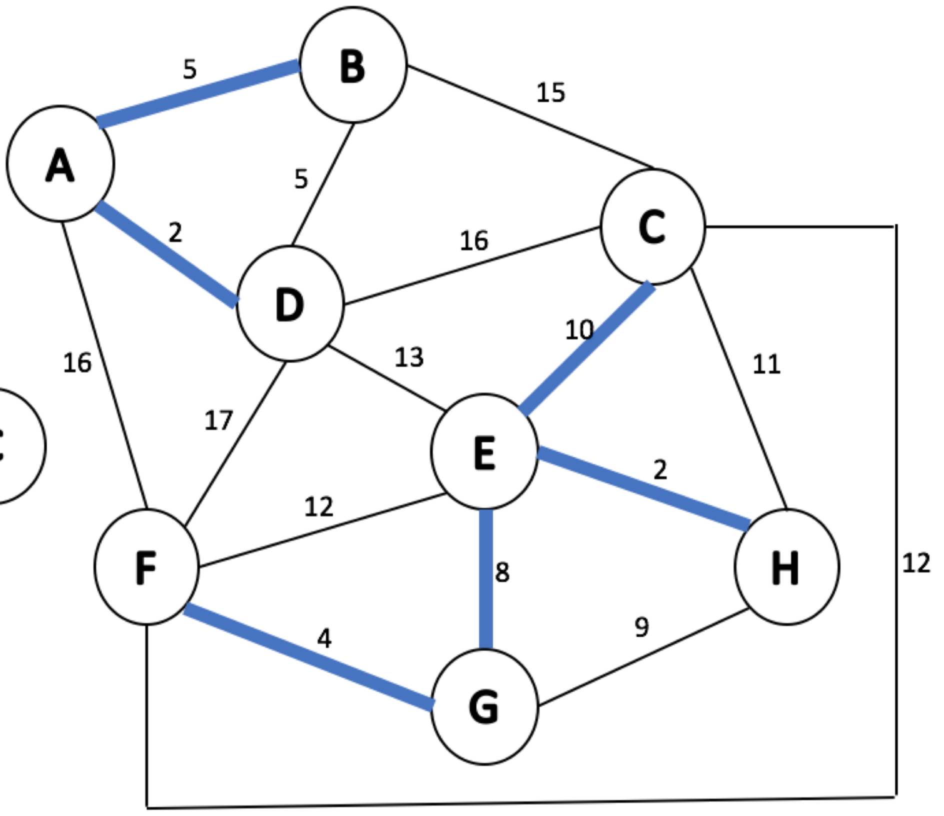


~~(A,D)~~
~~(E,H)~~
~~(F,G)~~
~~(A,B)~~
~~(B,D)~~
~~(G,E)~~
(G,H)
(E,C)
(C,H)
(E,F)
(F,C)
(D,E)
(B,C)
(C,D)
(A,F)
(D,F)

Next:
We skip (G,H) since they are in the same set.

Kruskal’s Algorithm

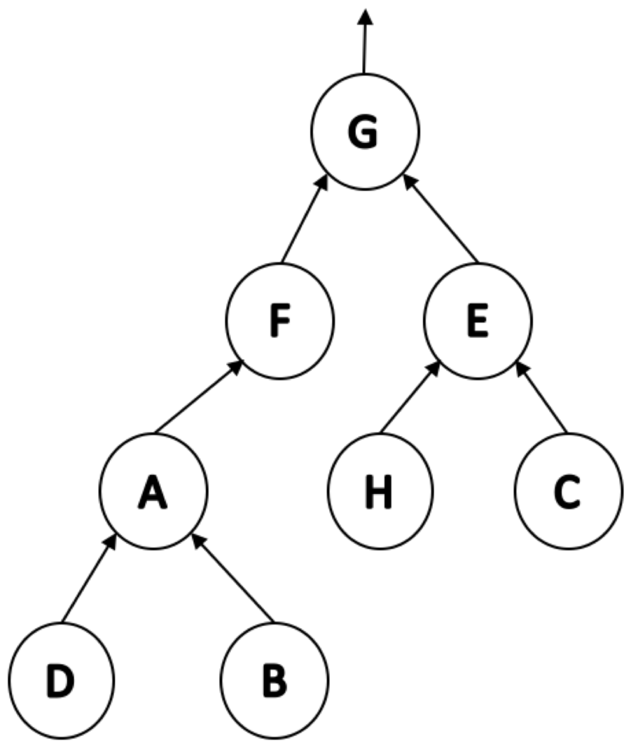


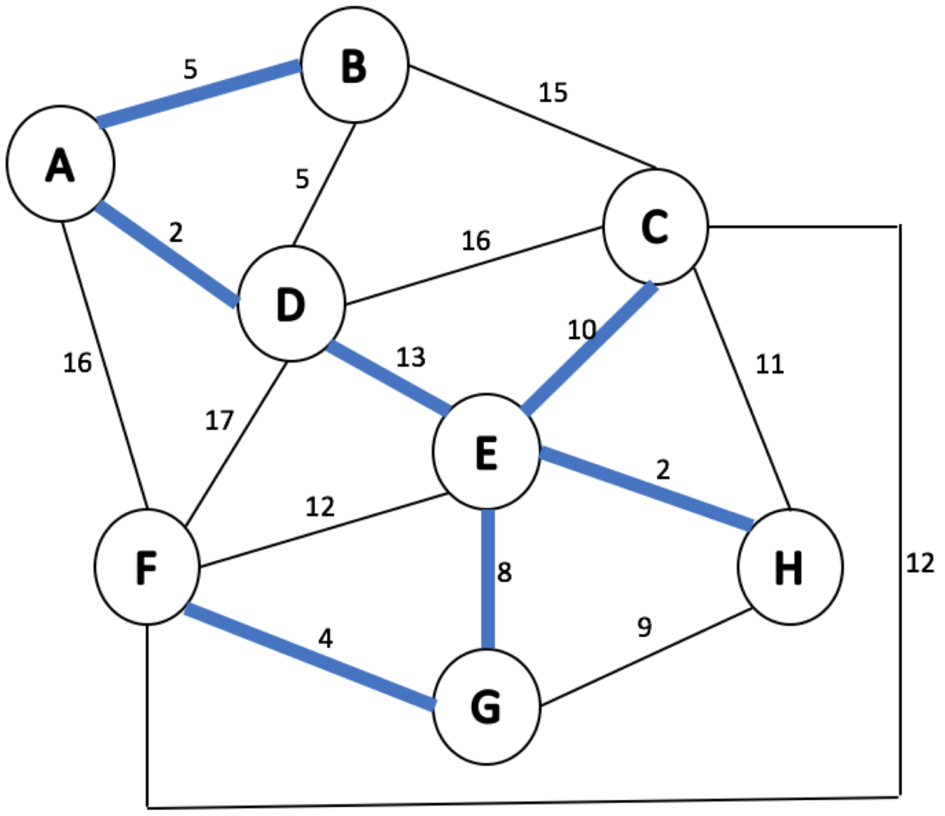


~~(A,D)~~
~~(E,H)~~
~~(F,G)~~
~~(A,B)~~
~~(B,D)~~
~~(G,E)~~
~~(G,H)~~
~~(E,C)~~
(C,H)
(E,F)
(F,C)
(D,E)
(B,C)
(C,D)
(A,F)
(D,F)

Next:
We skip (C, H), (E,F), (F,C) since they are in the same set.

Kruskal’s Algorithm

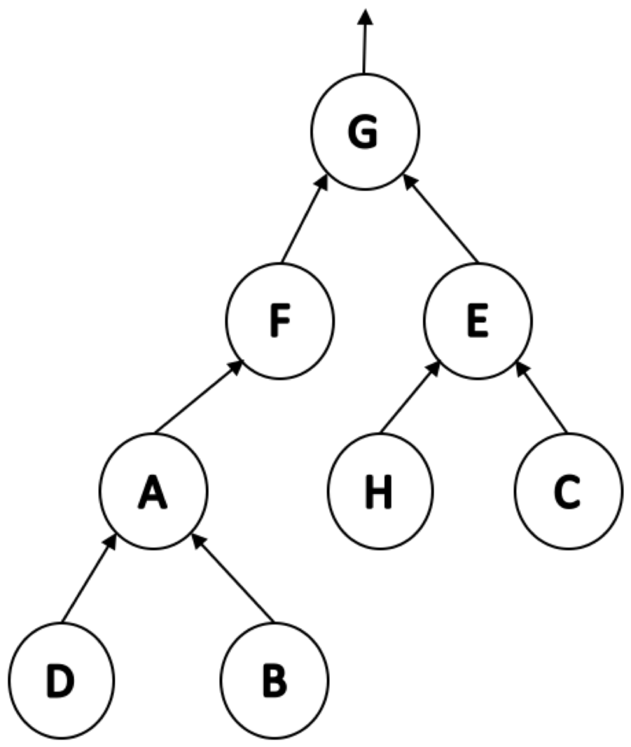


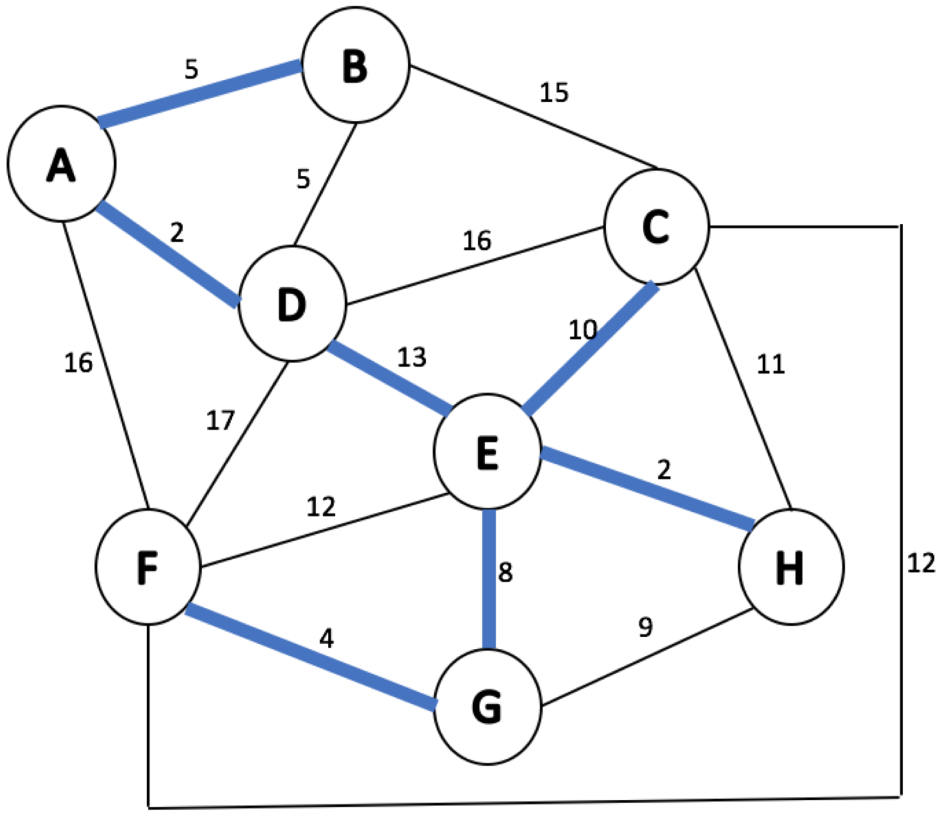


~~(A,D)~~
~~(E,H)~~
~~(F,G)~~
~~(A,B)~~
~~(B,D)~~
~~(G,E)~~
~~(G,H)~~
~~(E,C)~~
~~(C,H)~~
~~(E,F)~~
~~(F,C)~~
~~(D,E)~~
(B,C)
(C,D)
(A,F)
(D,F)

We pop the rest of the edges and ignore them all because now all vertices are in one set.

Kruskal’s Algorithm





We have created an MST → total sum of all edges is the smallest possible on this graph.

Kruskal’s Algorithm

```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  foreach (Edge e : G):
    Q.insert(e)

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

Stopping condition:

`|T.edges()| < n-1`

Worst case:

We visit every edge

Kruskal’s Algorithm

Graph can have multiple spanning trees => it can have multiple minimum spanning trees, but there will always be at least one minimum spanning tree.

Kruskal’s Algorithm

Priority Queue:	Heap	Sorted Array
Building :6-8	O(m)	O(m lg m)
Each removeMin :13	O(lg m)	O(1)

Lines 2-4: O(n)
Line 10: O(n)
Line 13: loop will run m times;
Line 14-17: O(1)

```
 1  KruskalMST(G):
 2    DisjointSets forest
 3    foreach (Vertex v : G):
 4      forest.makeSet(v)
 5
 6    PriorityQueue Q    // min edge weight
 7    foreach (Edge e : G):
 8      Q.insert(e)
 9
10    Graph T = (V, {})
11
12    while |T.edges()| < n-1:
13      Vertex (u, v) = Q.removeMin()
14      if forest.find(u) != forest.find(v):
15         T.addEdge(u, v)
16         forest.union( forest.find(u),
17                       forest.find(v) )
18
19    return T
```

Priority Queue:	Heap	Sorted Array
Building :6-8	O(m)	O(m lg n)
Each removeMin :13	O(lg n)	O(1)

$$m \;=\; O(n^2) \;<=\; c * n^2$$
$$\ldots \log \mathit{is\ an\ increasing\ function} \ldots$$
$$\lg(m) \;<=\; \lg(\,c \;*\; n^2\,)$$
$$<=\; \lg(c) \;+\; 2\lg(n)$$

Kruskal’s Algorithm - total running time:

$O(n + m)$ for set up with heap

$O(n + m \lg n)$ for set up with sorted array.

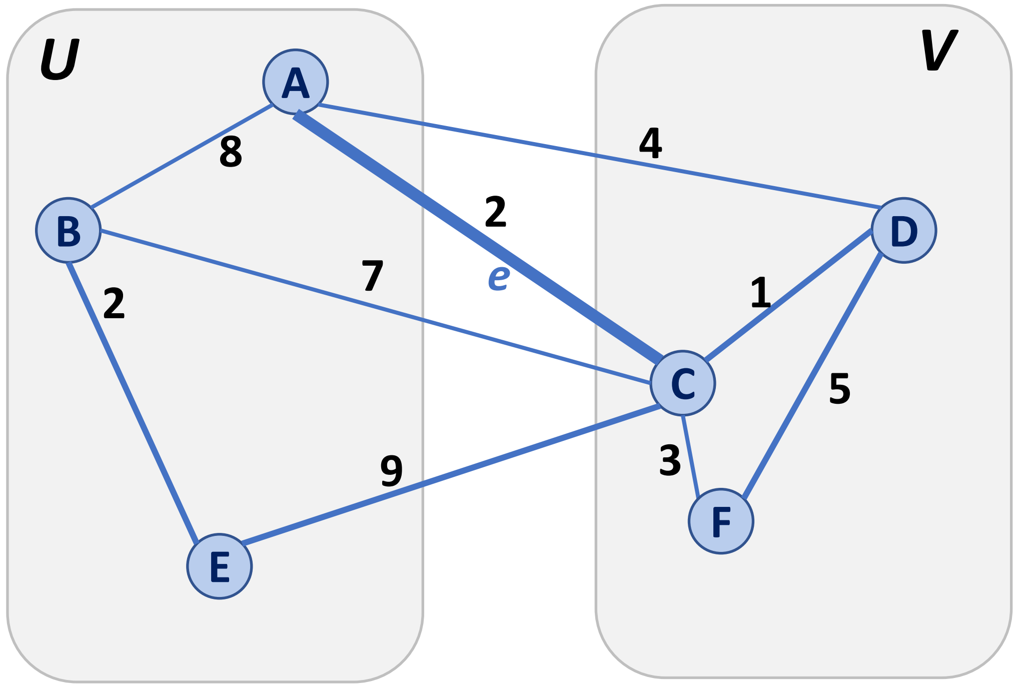
Priority Queue:	Total Running Time
Heap	$O(n + m) \ + \ O(m \lg n)$
Sorted Array	$O(n + m \lg n) \ + \ O(m)$

Partition Property

Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**.

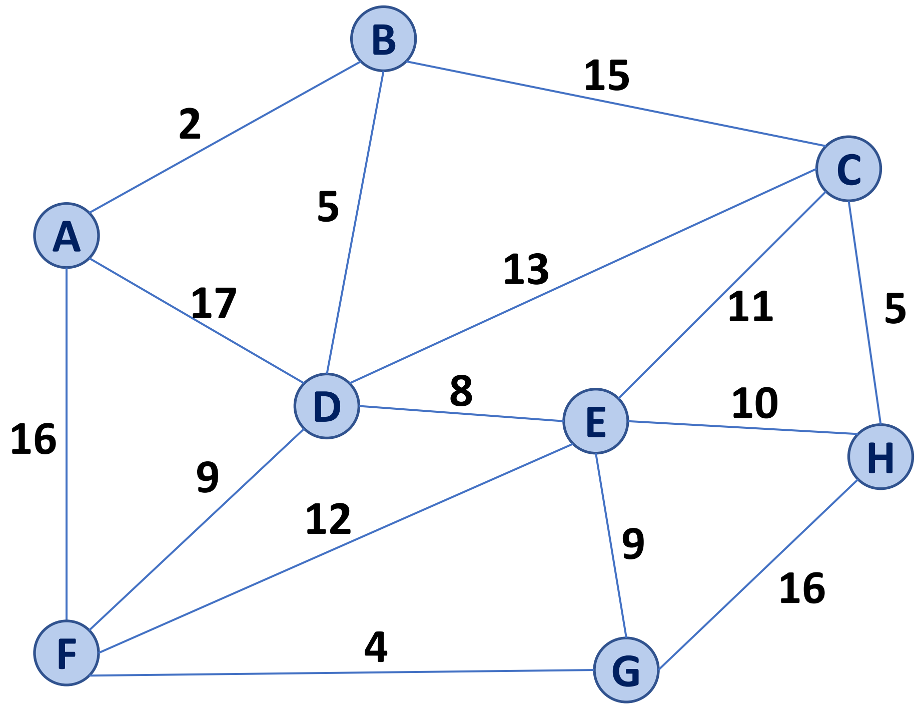
Let **e** be an edge of minimum weight across the partition.

Then **e** is part of some minimum spanning tree.

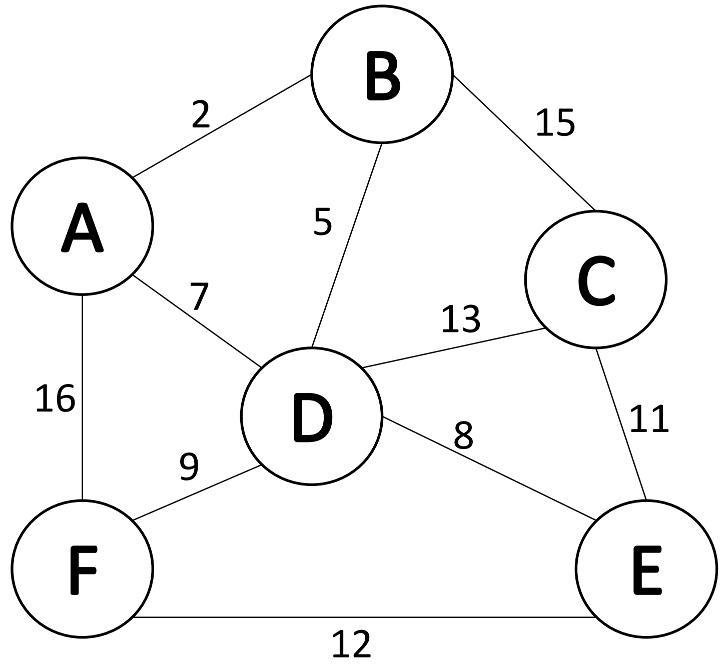


Partition Property

The partition property suggests an algorithm:



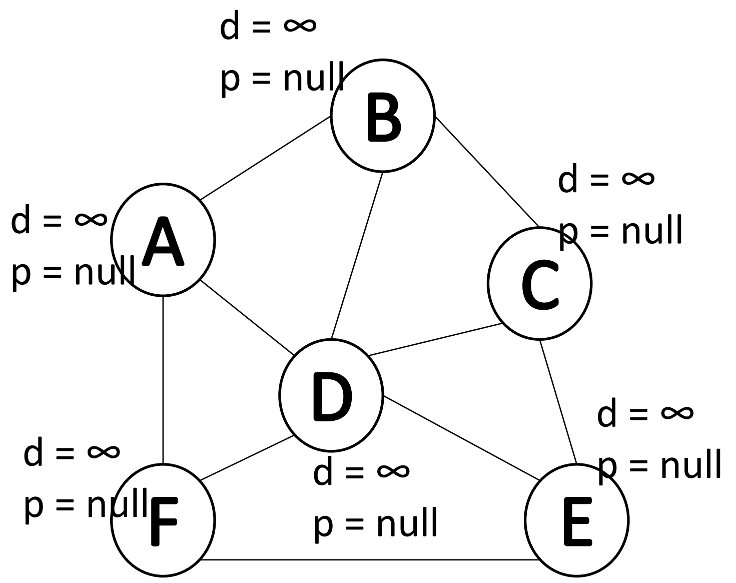
Prim’s Algorithm



```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

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

  return T
```

Prim's Algorithm

A	∞
B	∞
C	∞
D	∞
E	∞
F	∞



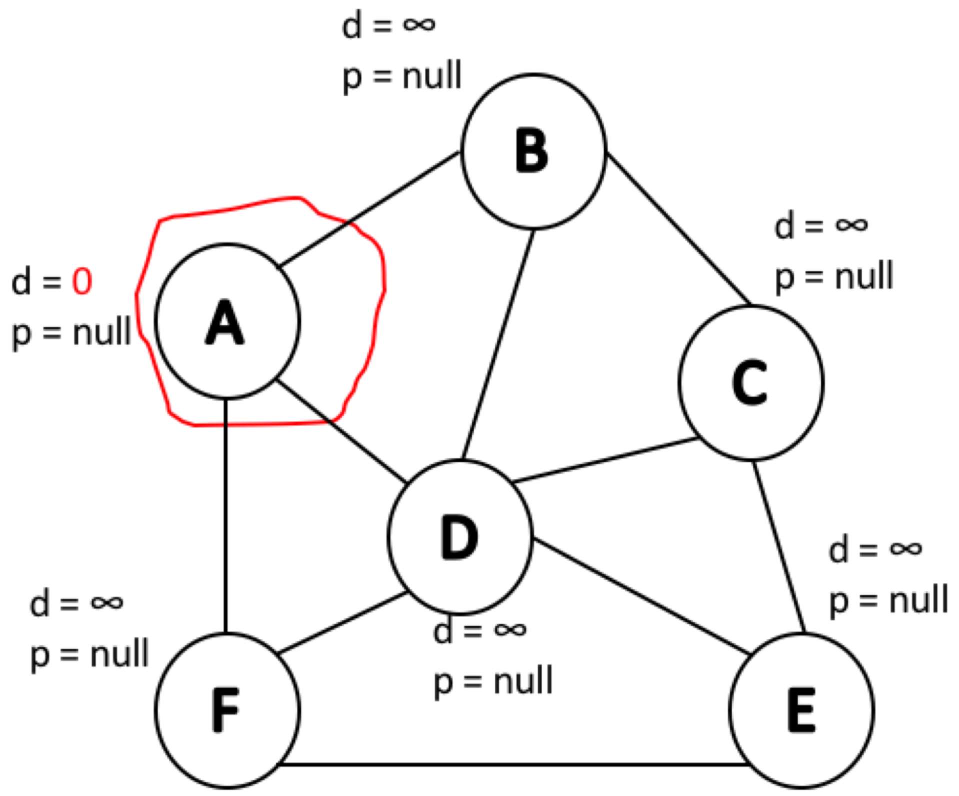
```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

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

  return T
```

Prim’s Algorithm

Algorithm logic:

Choose an arbitrary starting point and set its distance to 0.

Pop the starting vertex from the heap and update the distance/predecessor of adjacent vertices.

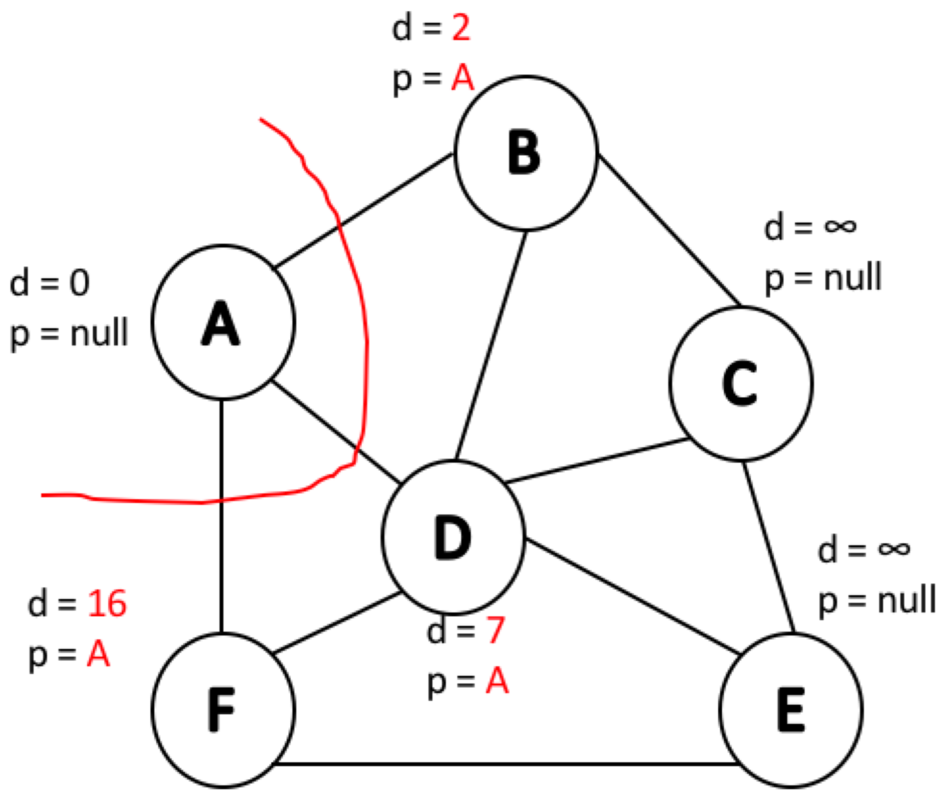


A	0
B	∞
C	∞
D	∞
E	∞
F	∞

Prim’s Algorithm

We pop A and update adjacent vertices B, D, and F.

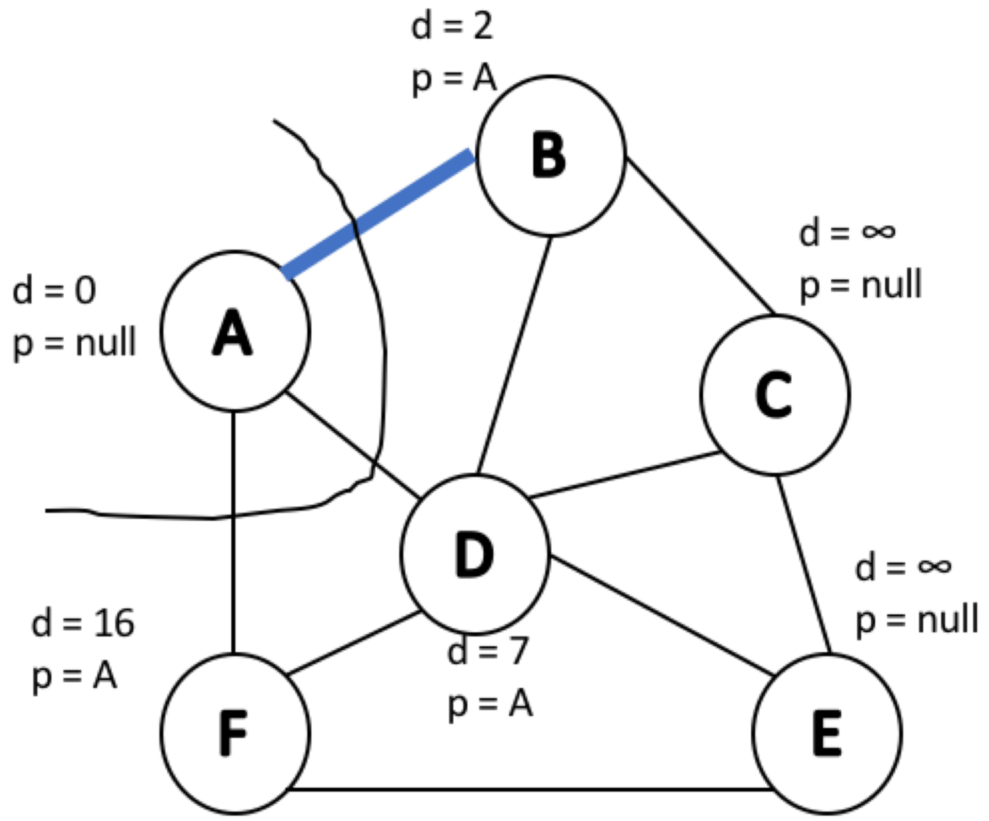
Next: remove minimum element from the heap and add the edge to the MST



~~A~~	~~0~~
B	2
C	∞
D	7
E	∞
F	16

Prim’s Algorithm

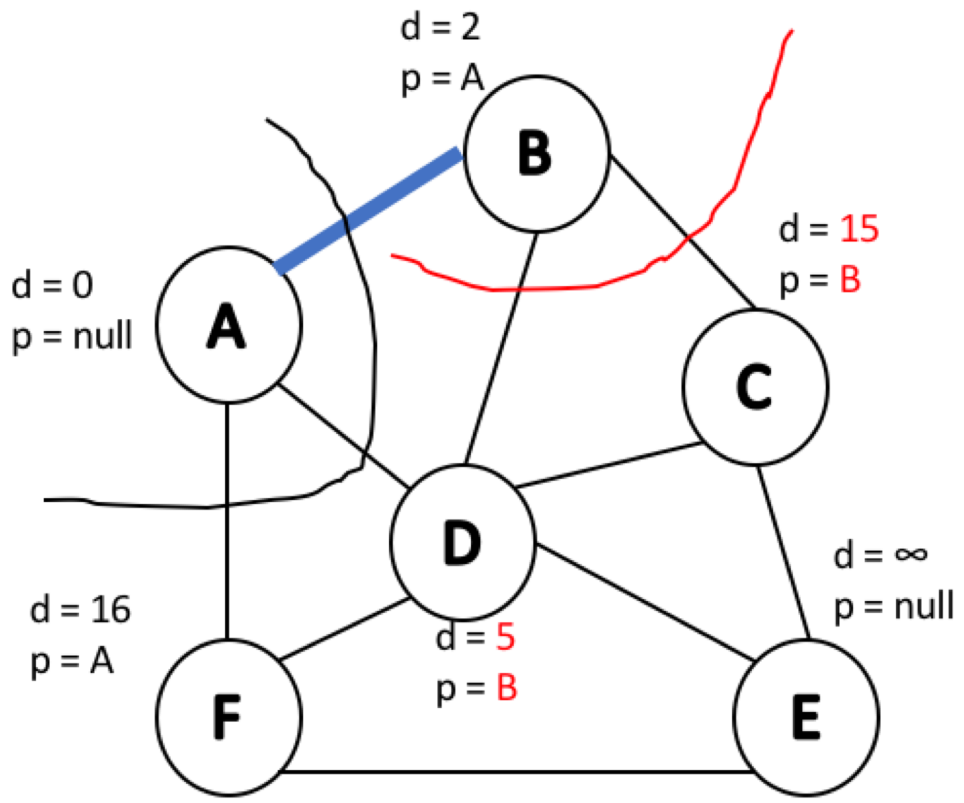
Next, we pop a vertex with the smallest distance and update adjacent vertices. However, we update vertices only if the distance is smaller than the current.



~~A~~	~~0~~
B	2
C	∞
D	7
E	∞
F	16

Prim's Algorithm

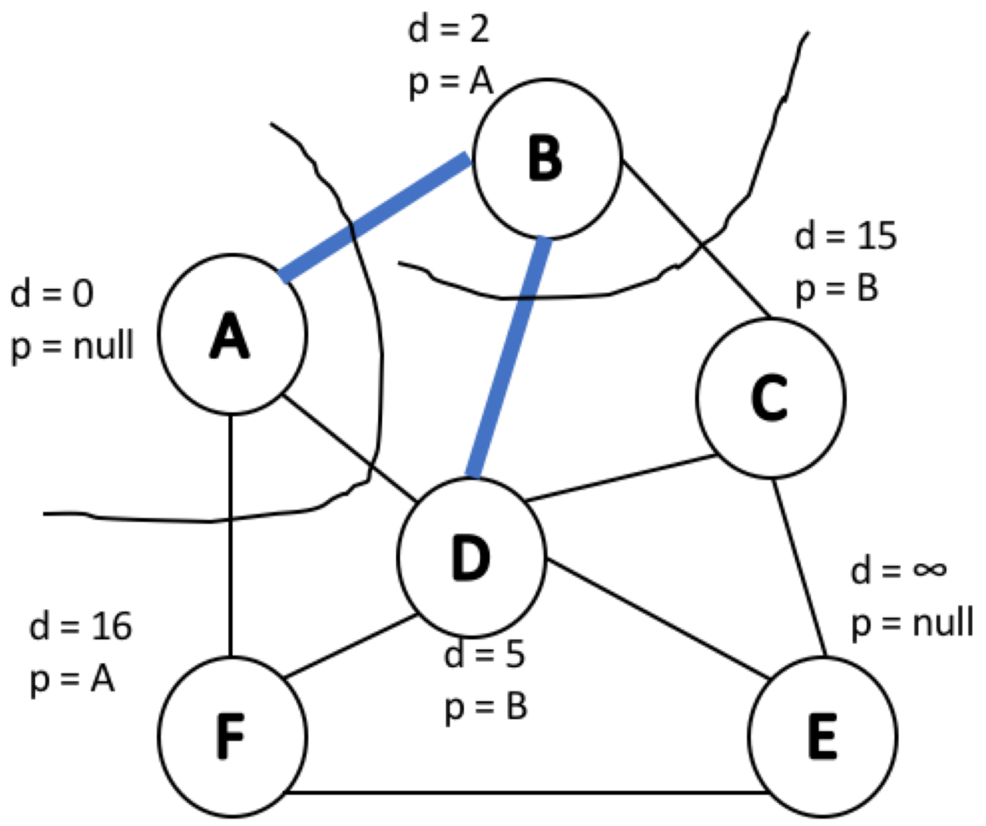
Next: remove minimum element from the heap and add the edge to the MST
We will add edge (D, B)



~~A~~	~~0~~
~~B~~	~~2~~
C	15
D	5
E	∞
F	16

Prim’s Algorithm

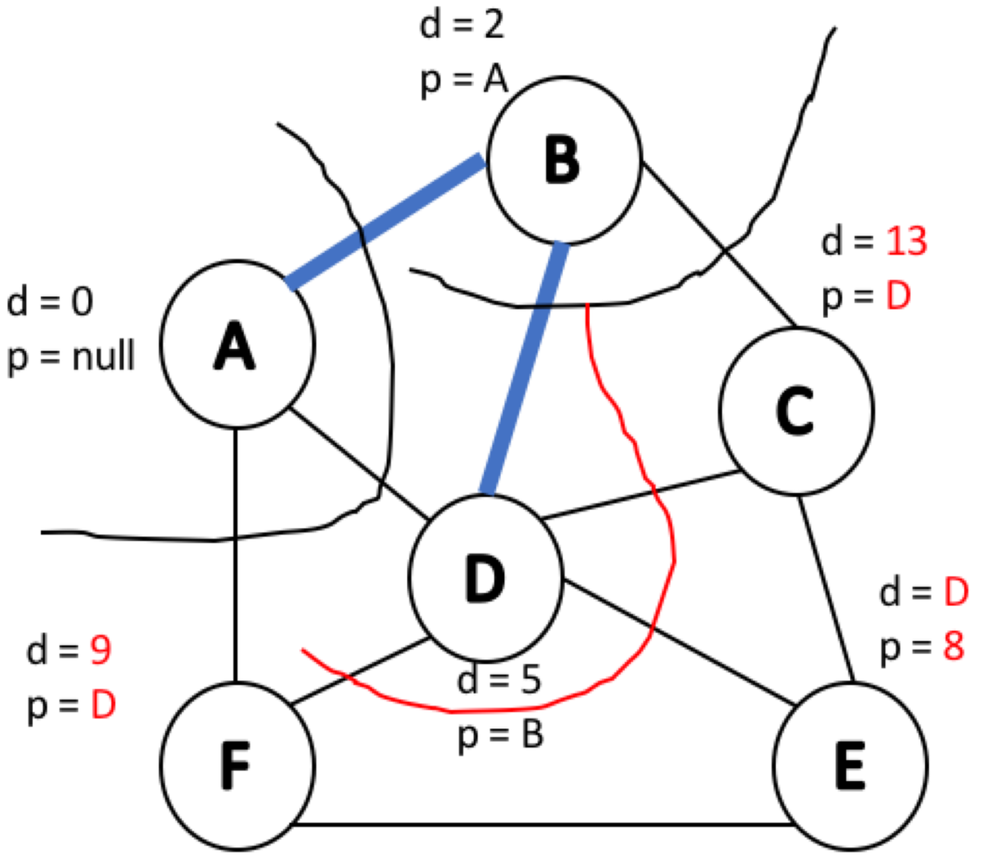
Next: pop a vertex with the smallest distance, update adjacent vertices if needed, and add the edge with the smallest distance.
These steps are repeated until the heap is empty .



~~A~~	~~0~~
~~B~~	~~2~~
C	15
D	5
E	∞
F	16

Prim’s Algorithm

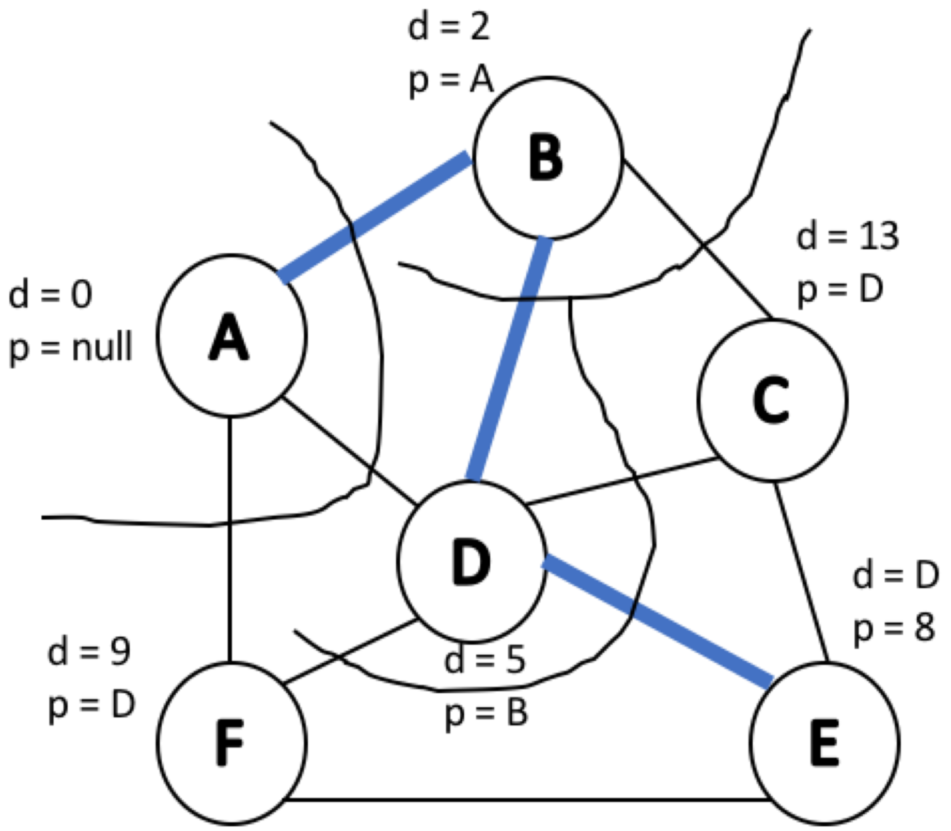
we pop D and we update all its adjacent vertices F, E, and C



~~A~~	~~0~~
~~B~~	~~2~~
C	13
~~D~~	~~5~~
E	8
F	9

Prim’s Algorithm

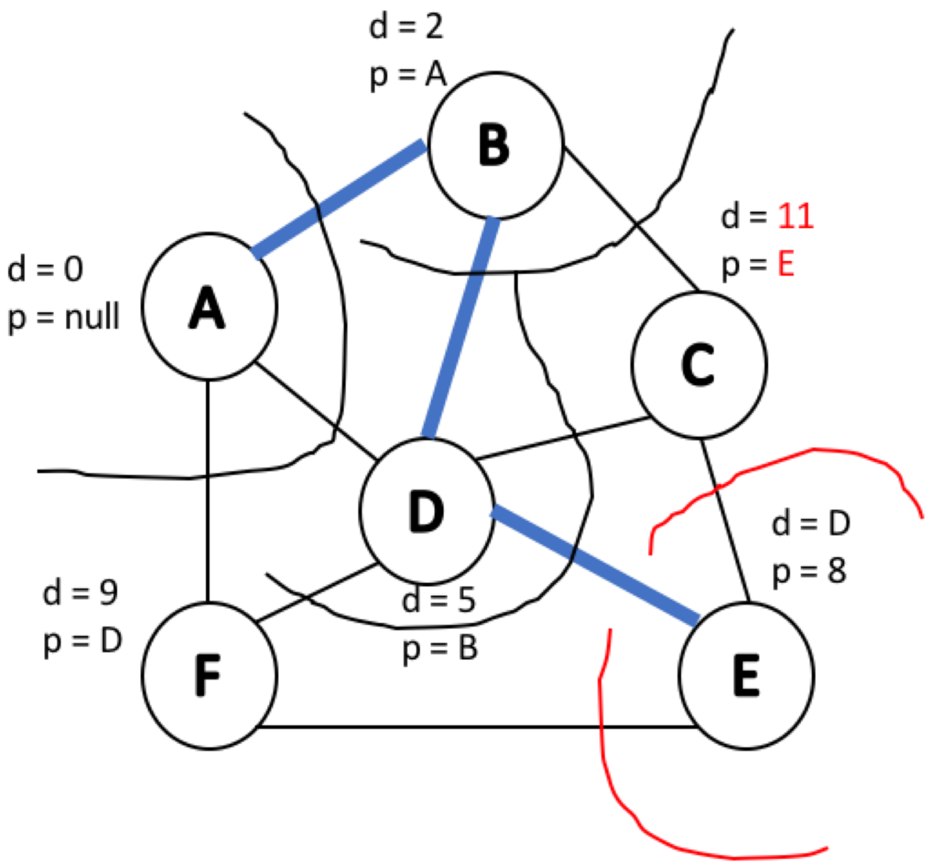
The next vertex with smallest distance is E. We add the edge from D to E.



~~A~~	~~0~~
~~B~~	~~2~~
C	13
~~D~~	~~5~~
E	8
F	9

Prim’s Algorithm

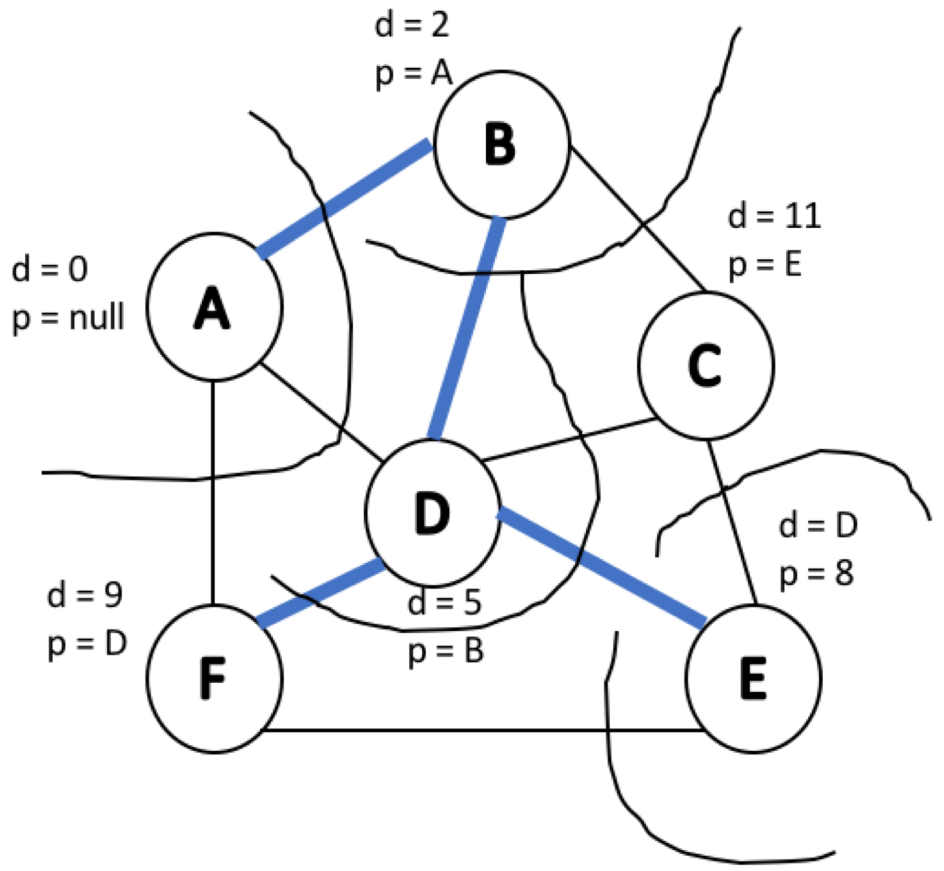
pop E and we only update C, because F’s current distance is smaller than the one from E to F.



~~A~~	~~0~~
~~B~~	~~2~~
C	11
~~D~~	~~5~~
~~E~~	~~8~~
F	9

Prim’s Algorithm

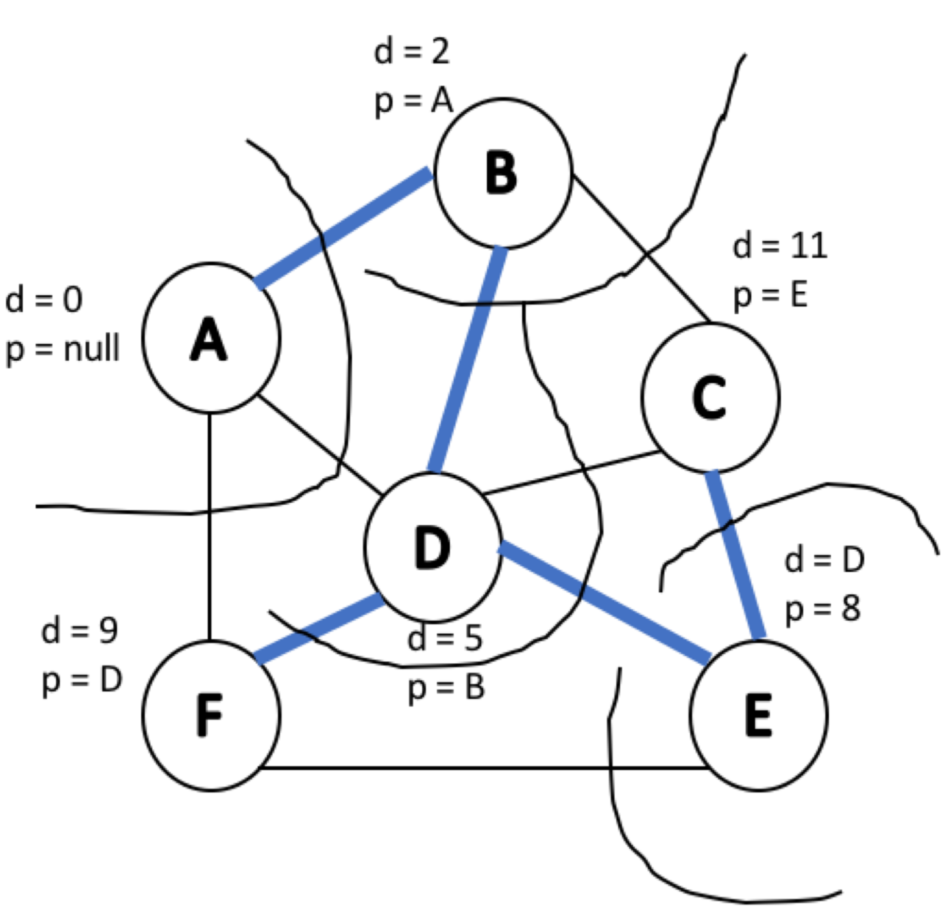
- The shortest distance is from D to F, so we add that edge to the graph.
- We pop 9 and we don’t have anything to update because all neighboring edges have been added to the graph.

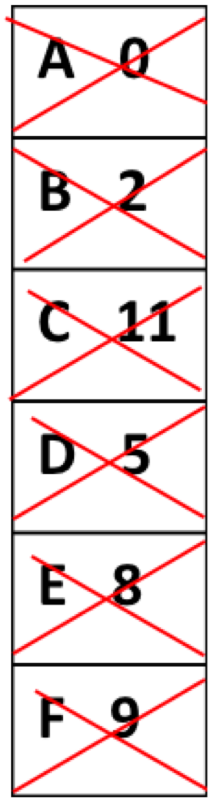


~~A~~	~~0~~
~~B~~	~~2~~
C	11
~~D~~	~~5~~
~~E~~	~~8~~
~~F~~	~~9~~

Prim’s Algorithm

- Finally, we pop C and add an edge from E to C. After this step the heap is empty and we are done.





~~A~~	~~0~~
~~B~~	~~2~~
~~C~~	~~11~~
~~D~~	~~5~~
~~E~~	~~8~~
~~F~~	~~9~~

Prim’s Algorithm – Runtime analysis

- Lines 6 to 10: O(n)
- Lines 12 - 15: O(n)
- 15: loop O(n) :
  - Line 16: log(n) (remove)
  - Line 17: O*(n) (adj. matrix)
    - $O(n^2)$
  - Running 18-19 lines n times: O(m)
  - 20-22: Lg(n) (restoring heap property)

- $O(n^2 + m \lg n)$

```
 1  PrimMST(G, s):
 2    Input: G, Graph;
 3           s, vertex in G, starting vertex
 4    Output: T, a minimum spanning tree (MST) of G
 5
 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 m = Q.removeMin()
17      T.add(m)
18      foreach (Vertex v : neighbors of m not in T):
19        if cost(v, m) < d[v]:
20          d[v] = cost(v, m)
21          p[v] = m
22
23    return T
```

Prim's Algorithm – Runtime analysis

- $O(n^2 + m \lg n)$
- The reason we have n^2 in the running time is because adding a vertex takes O(n). Therefore, we consider adjacency list → 16-17 will run in $n \ \log(n)$

How can we reduce the cost of updating? - We can use an unsorted array.
Line 16 → remove takes O(n) because we need to loop over the whole array to find the vertex to remove.
Line 17 → O(n) as previously explained.
Lines 19 to 22 → will now take O(m) because we don't need to update anything, we are just looping over edges.
Total running time will be O(n^2).

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$	$O(n \lg(n) + m \lg(n))$
Unsorted Array	$O(n^2)$	$O(n^2)$

Based on the analysis, heap with the adjacency matrix gives worst running time;

- Case 1: the data is sparse → use (heap + adj list) and the running time will be O($nlog(n)$) $(n{\sim}m)$
- Case 2: the data is dense → use (unsorted array + adj matrix/list) and the running time will be O(n^2). $m{\sim}n^2$

MST Algorithm Runtime:

- Kruskal's Algorithm:

  O(n + m lg(n))

- Prim's Algorithm:

  O(n lg(n) + m lg(n))

- What must be true about the connectivity of a graph when running an MST algorithm?

  Graph is a connected graph.

  - How does n and m relate?

$$\boldsymbol{m \geq n - 1 \rightarrow O(n) = O(n)}$$

Running time: $m \lg n$

Fibonacci heap

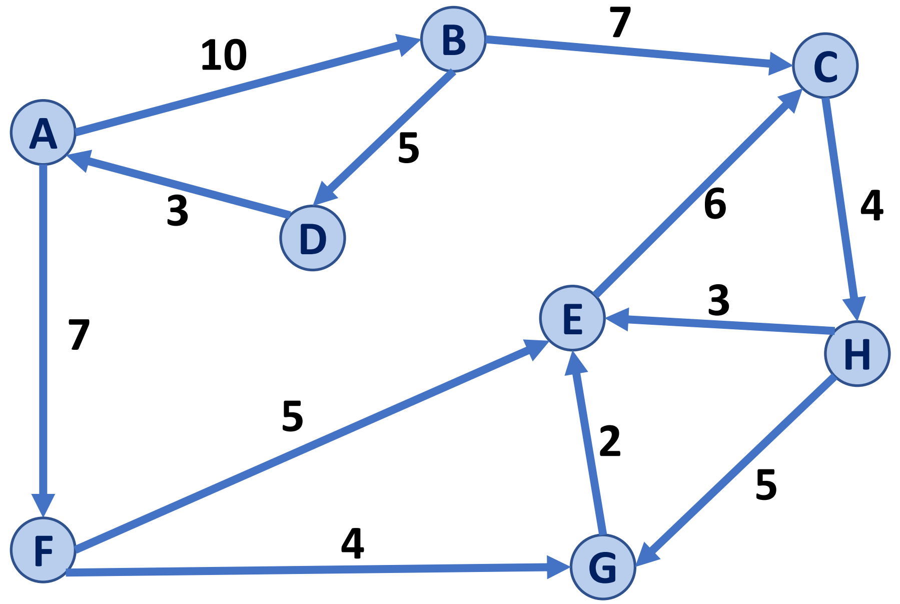
Decrease key operation in Fibonacci heap takes O(1)* time.

	Binary Heap	Fibonacci Heap
Remove Min	O(lg(n))	O(lg(n))
Decrease Key	O(lg(n))	O(1)*

If we use Fibonacci heap for our algorithm, updated value will take O(1) time, since we are always decreasing key.

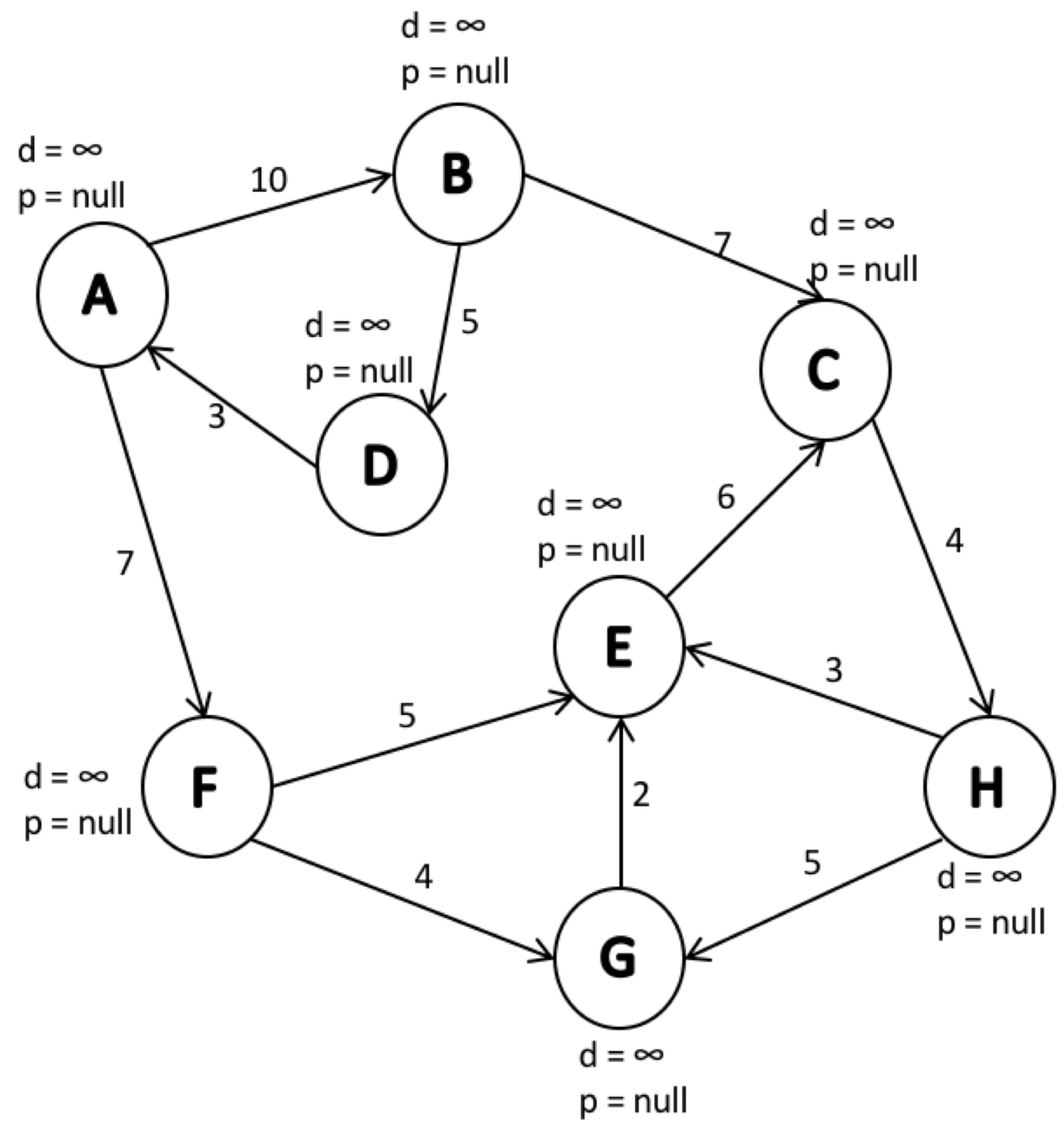
Adj. List with Fibonacci heap: $O(n \lg n + m) \rightarrow fastest\ running\ time\ for\ MST$

Dijkstra’s Algorithm



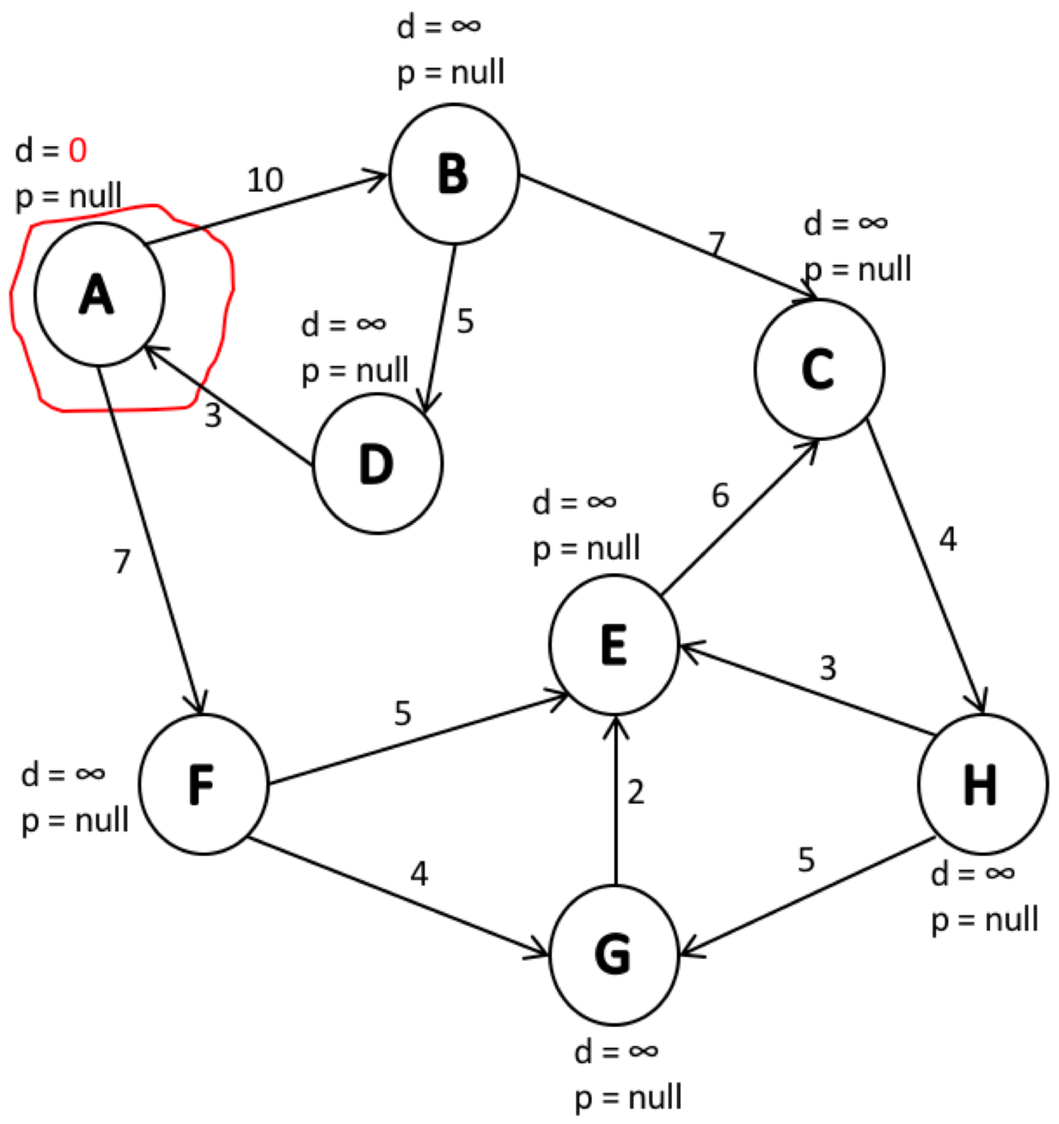
```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

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

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

Choose an arbitrary starting point and set its distance to 0.



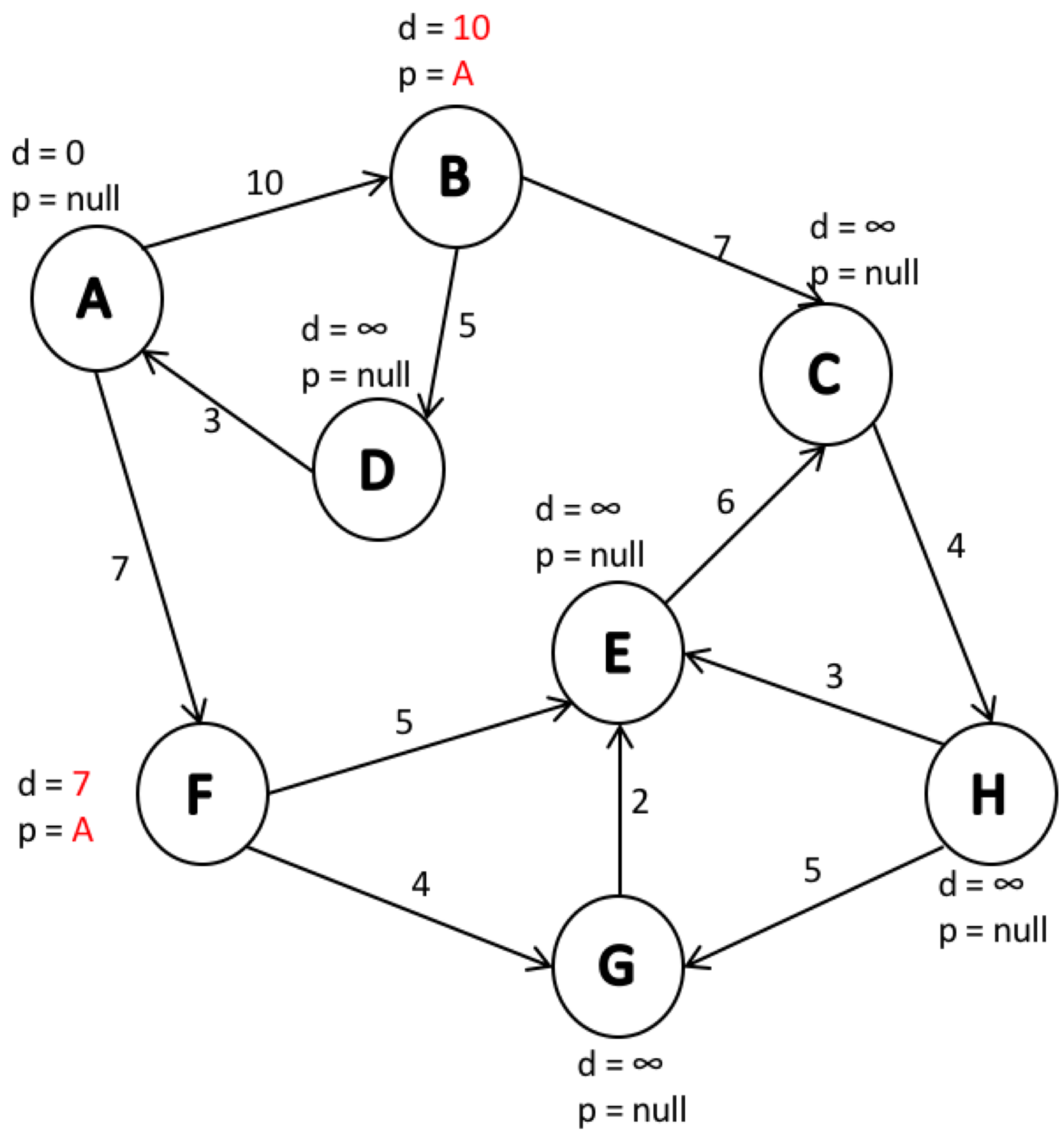
V	d	p
A	∞	null
B	∞	null
C	∞	null
D	∞	null
E	∞	null
F	∞	null
G	∞	null
H	∞	null

Starting point A.



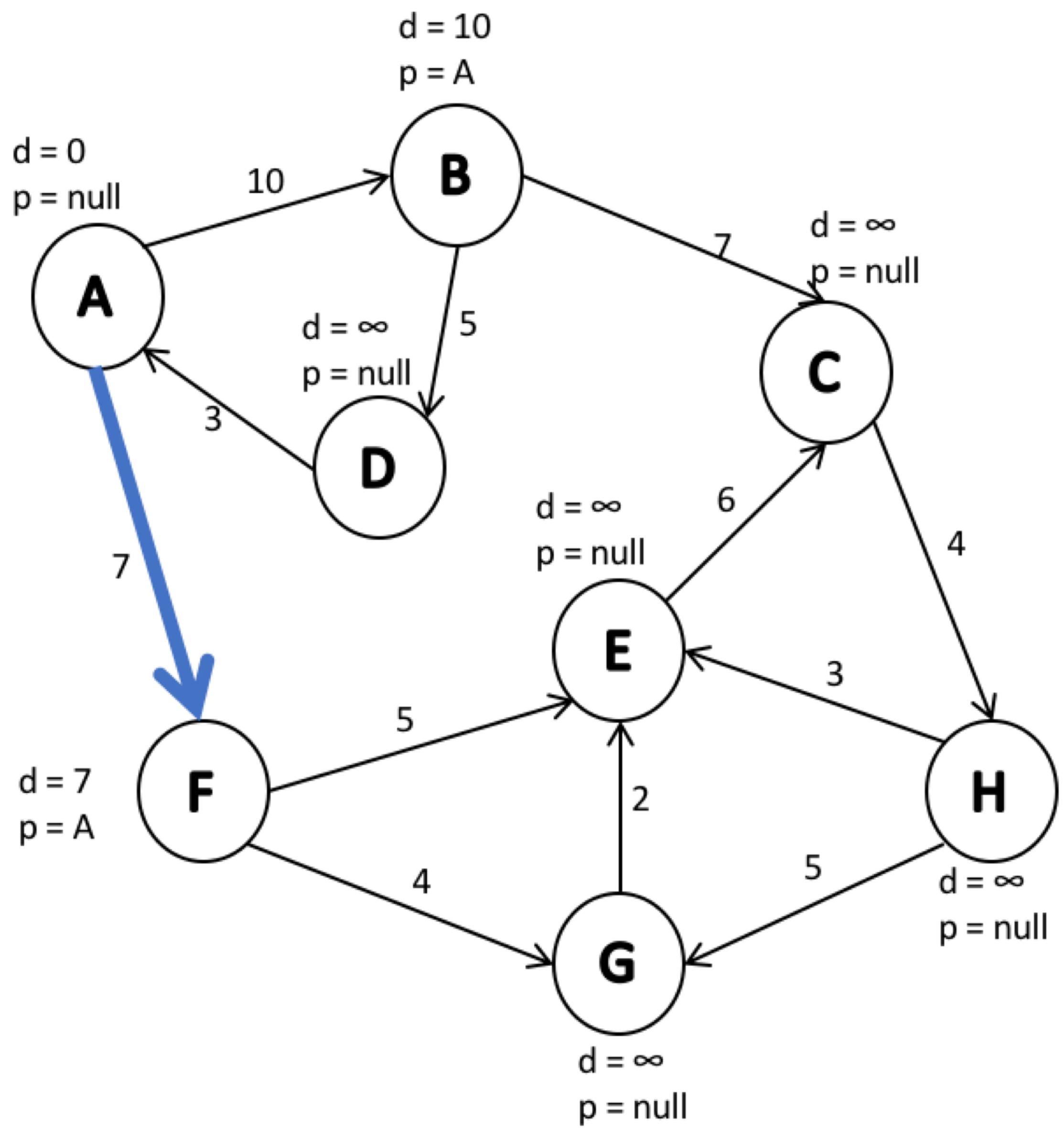
V	d	p
A	0	null
B	∞	null
C	∞	null
D	∞	null
E	∞	null
F	∞	null
G	∞	null
H	∞	null

We pop A and update adjacent vertices B and F. *Notice: edges are directed*



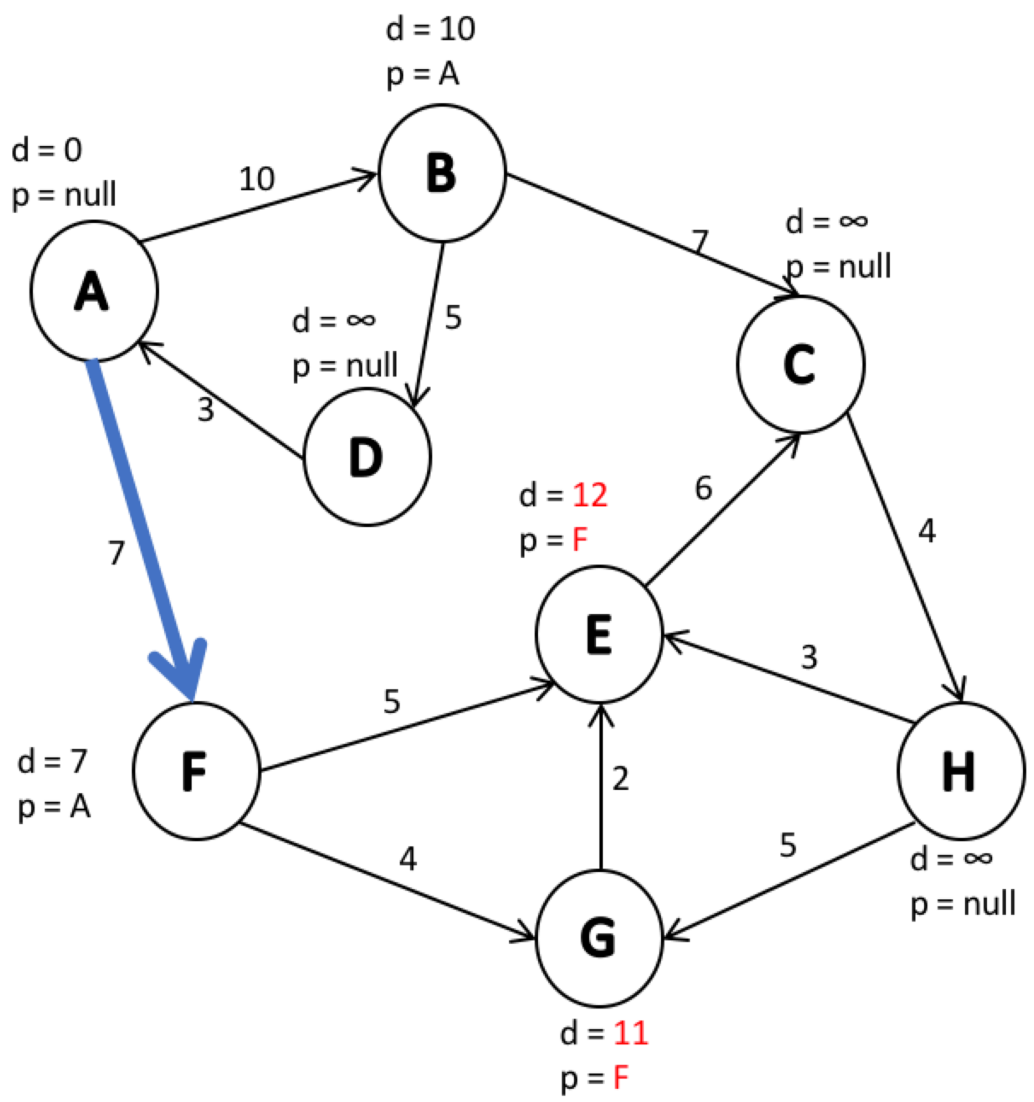
V	d	p
~~A~~	~~0~~	~~null~~
B	10	A
C	∞	null
D	∞	null
E	∞	null
F	7	A
G	∞	null
H	∞	null

add an edge to the node with the smallest distance



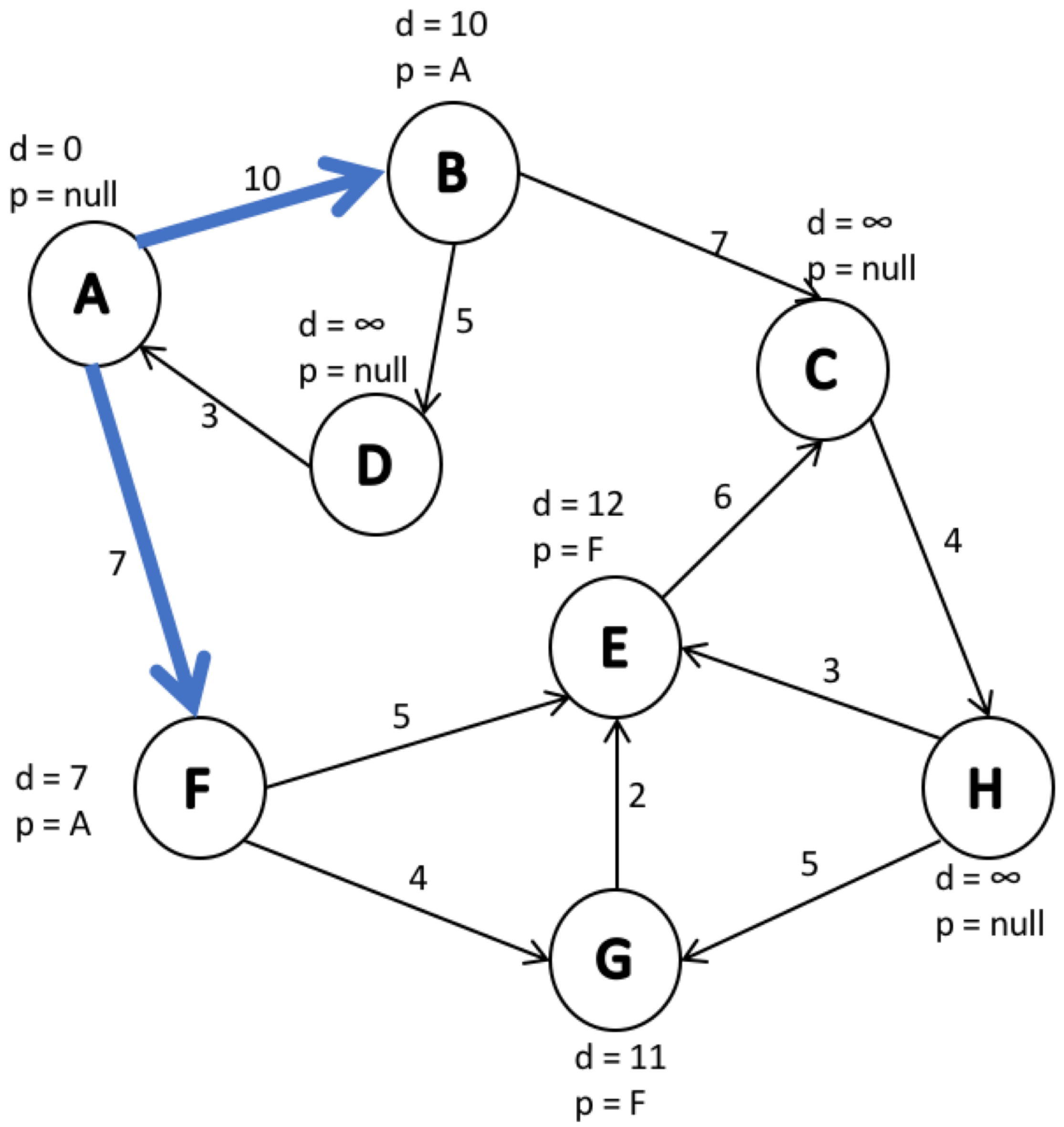
V	**d**	**p**
~~A~~	~~0~~	~~null~~
B	10	A
C	∞	null
D	∞	null
E	∞	null
F	7	A
G	∞	null
H	∞	null

Pop a vertex with the smallest distance and update adjacent vertices only if the distance from the start is smaller than the current d.



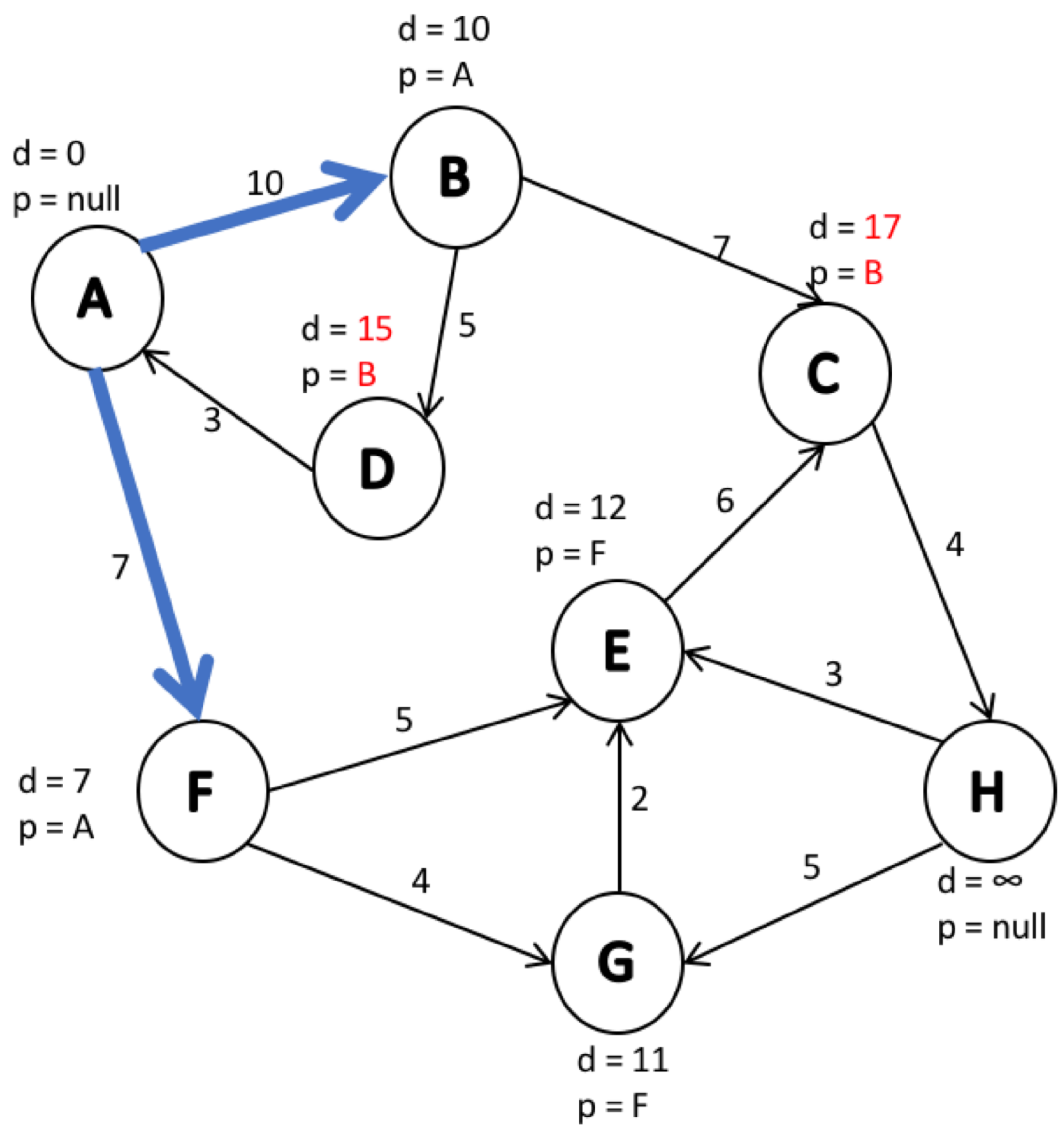
V	d	p
~~A~~	~~0~~	~~null~~
B	10	A
C	∞	null
D	∞	null
E	12	F
~~F~~	~~7~~	~~A~~
G	11	F
H	∞	null

Add an edge to the node with the smallest path



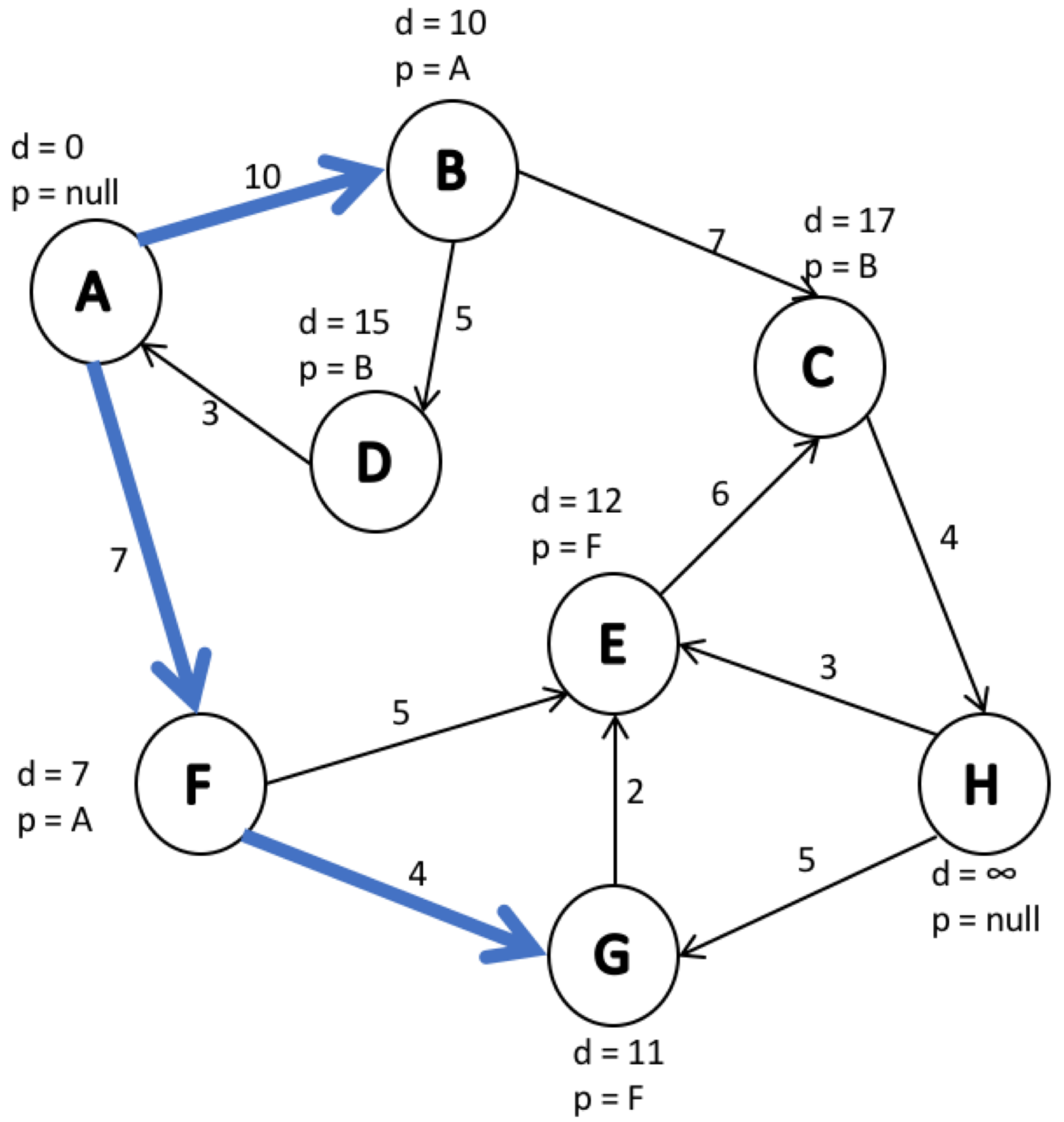
V	d	p
~~A~~	~~0~~	~~null~~
B	10	A
C	∞	null
D	∞	null
E	12	F
~~F~~	~~7~~	~~A~~
G	11	F
H	∞	null

Pop and update if needed:



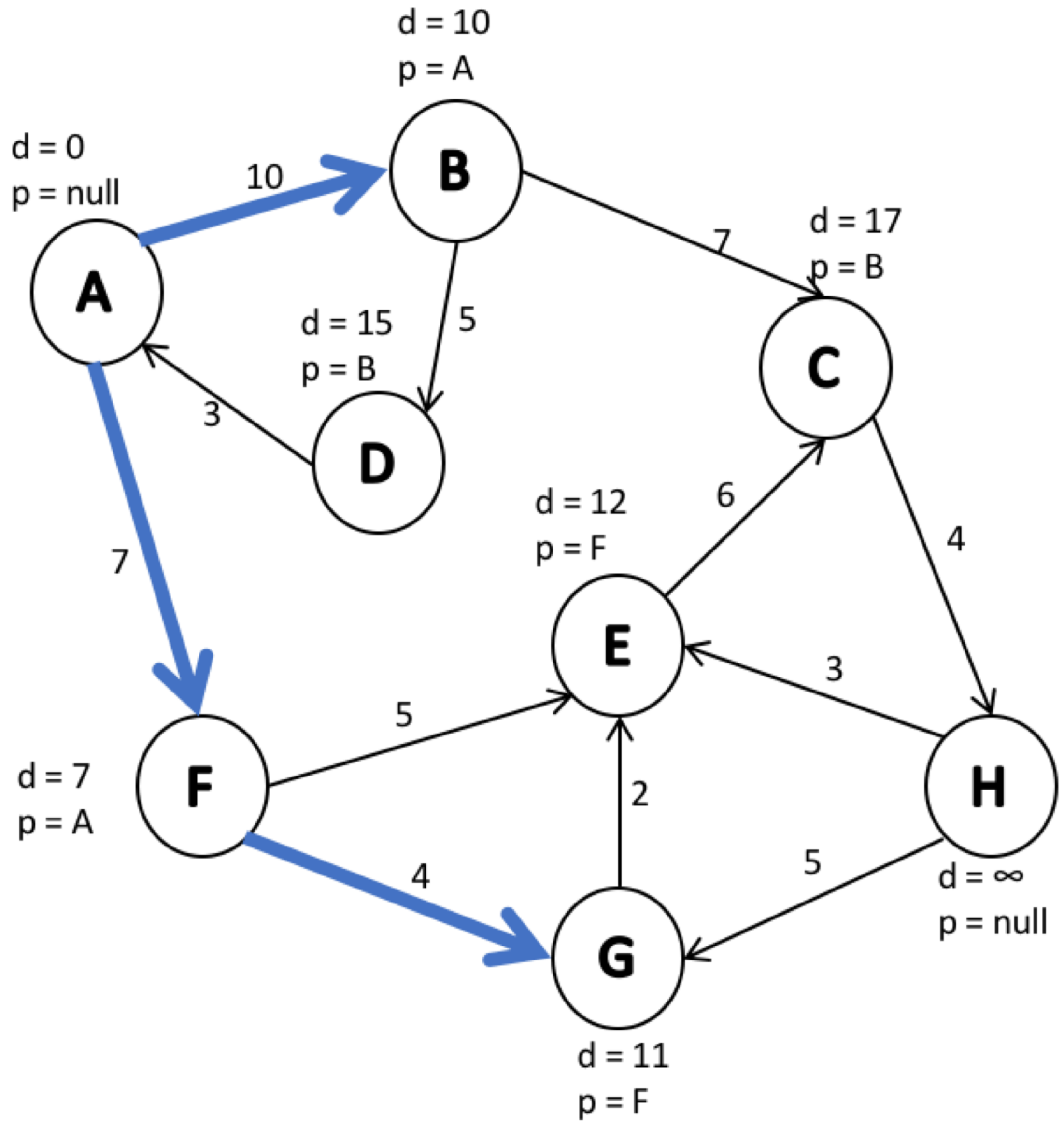
V	d	p
~~A~~	~~0~~	~~null~~
~~B~~	~~10~~	~~A~~
C	17	B
D	15	B
E	12	F
~~F~~	~~7~~	~~A~~
G	11	F
H	∞	null

Add the edge:



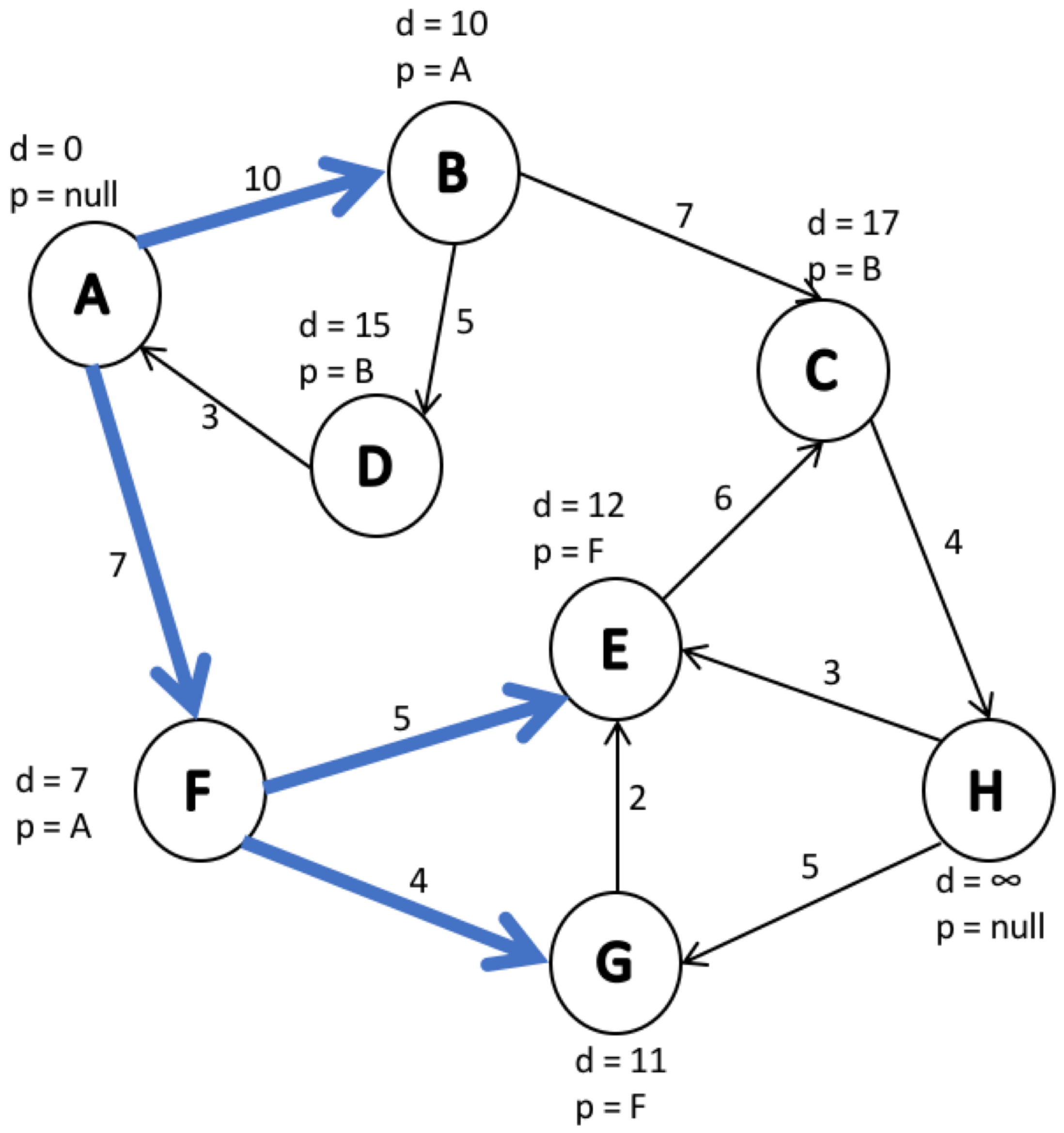
V	d	p
~~A~~	~~0~~	~~null~~
~~B~~	~~10~~	~~A~~
C	17	B
D	15	B
E	12	F
~~F~~	~~7~~	~~A~~
G	11	F
H	∞	null

Pop and update if needed:



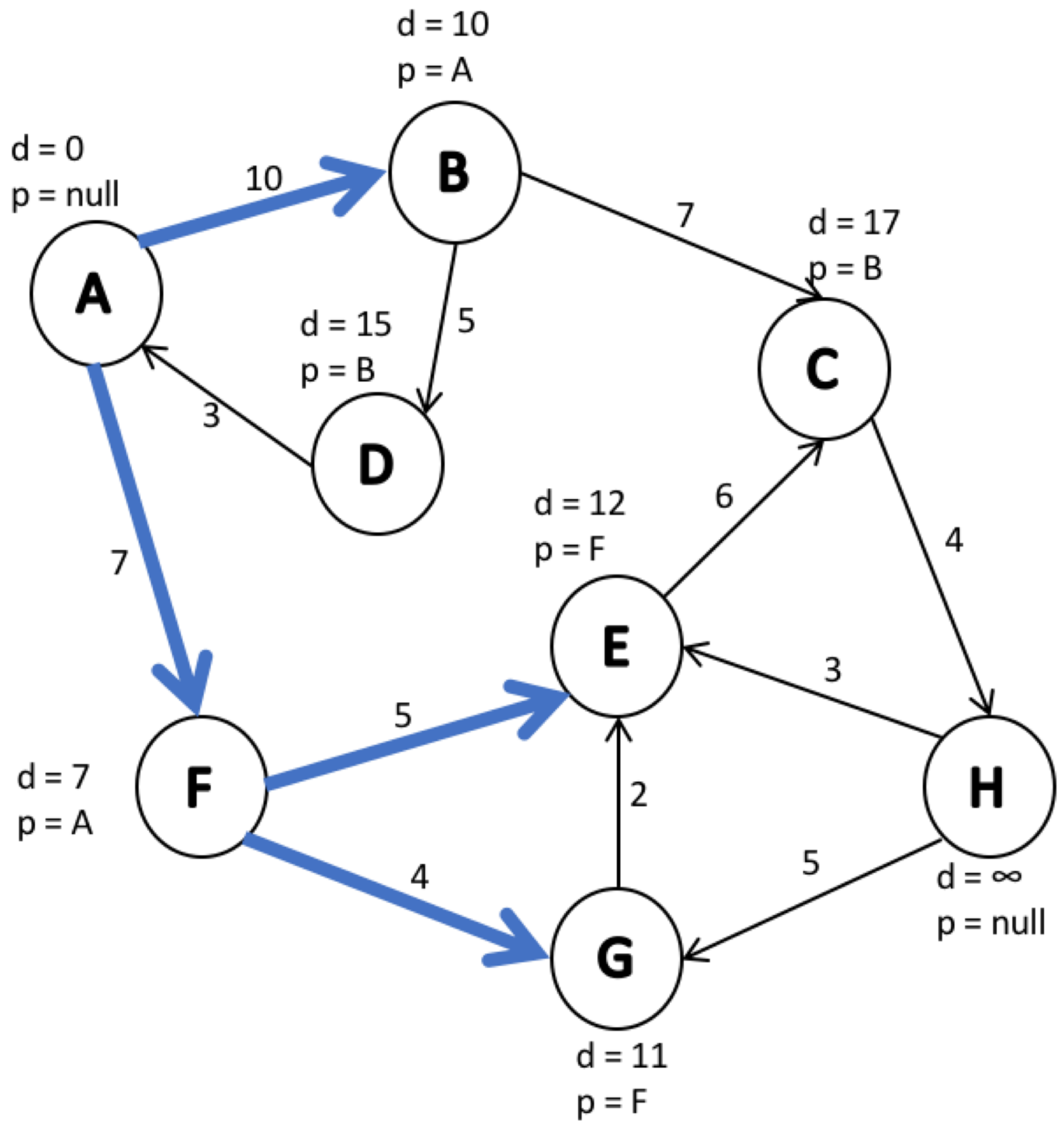
V	d	p
~~A~~	~~0~~	~~null~~
~~B~~	~~10~~	~~A~~
C	17	B
D	15	B
E	12	F
~~F~~	~~7~~	~~A~~
~~G~~	~~11~~	~~F~~
H	∞	null

Add the edge:



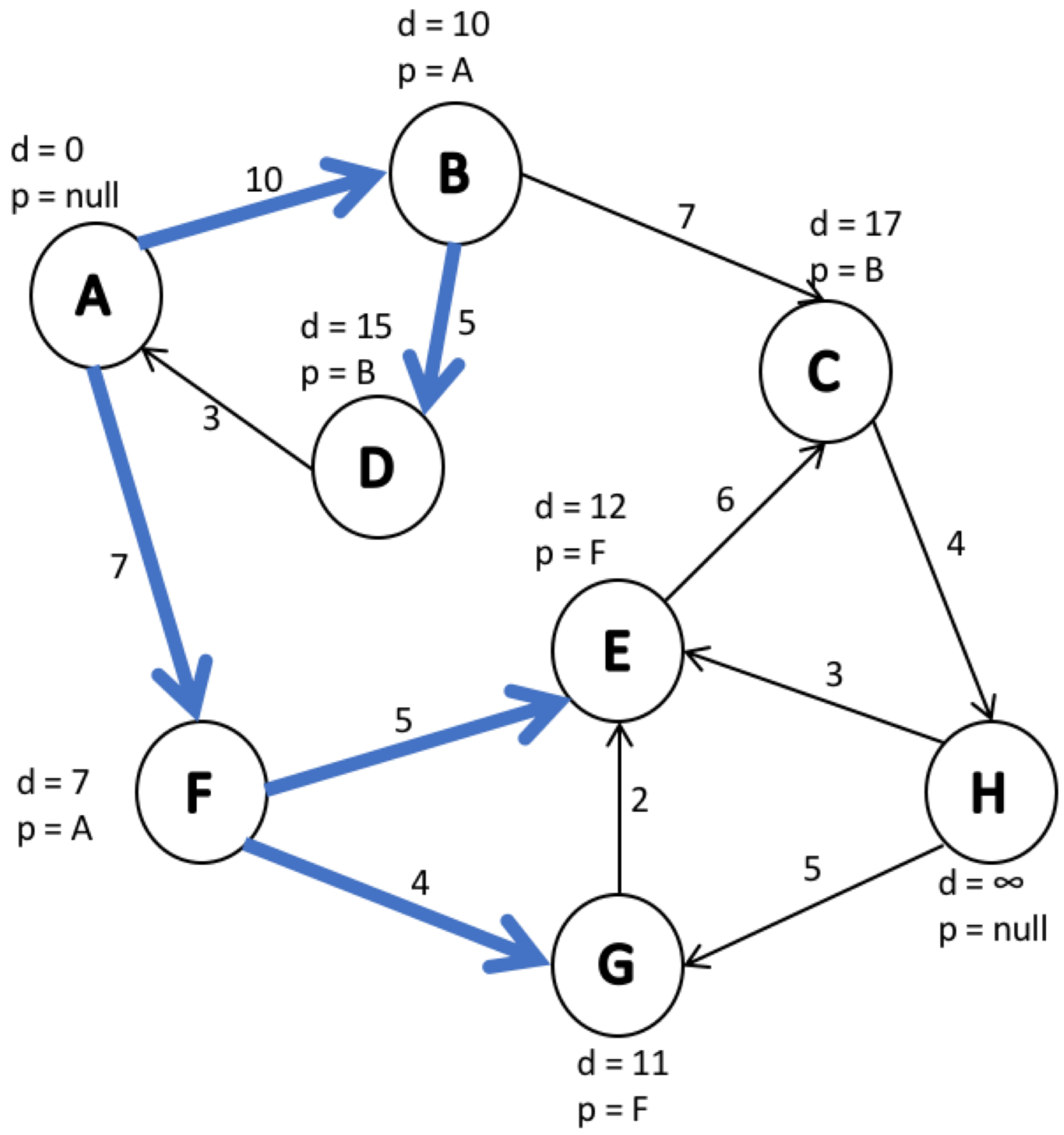
V	d	p
~~A~~	~~0~~	~~null~~
~~B~~	~~10~~	~~A~~
C	17	B
D	15	B
E	12	F
~~F~~	~~7~~	~~A~~
~~G~~	~~11~~	~~F~~
H	∞	null

Pop and update (nothing was updated)



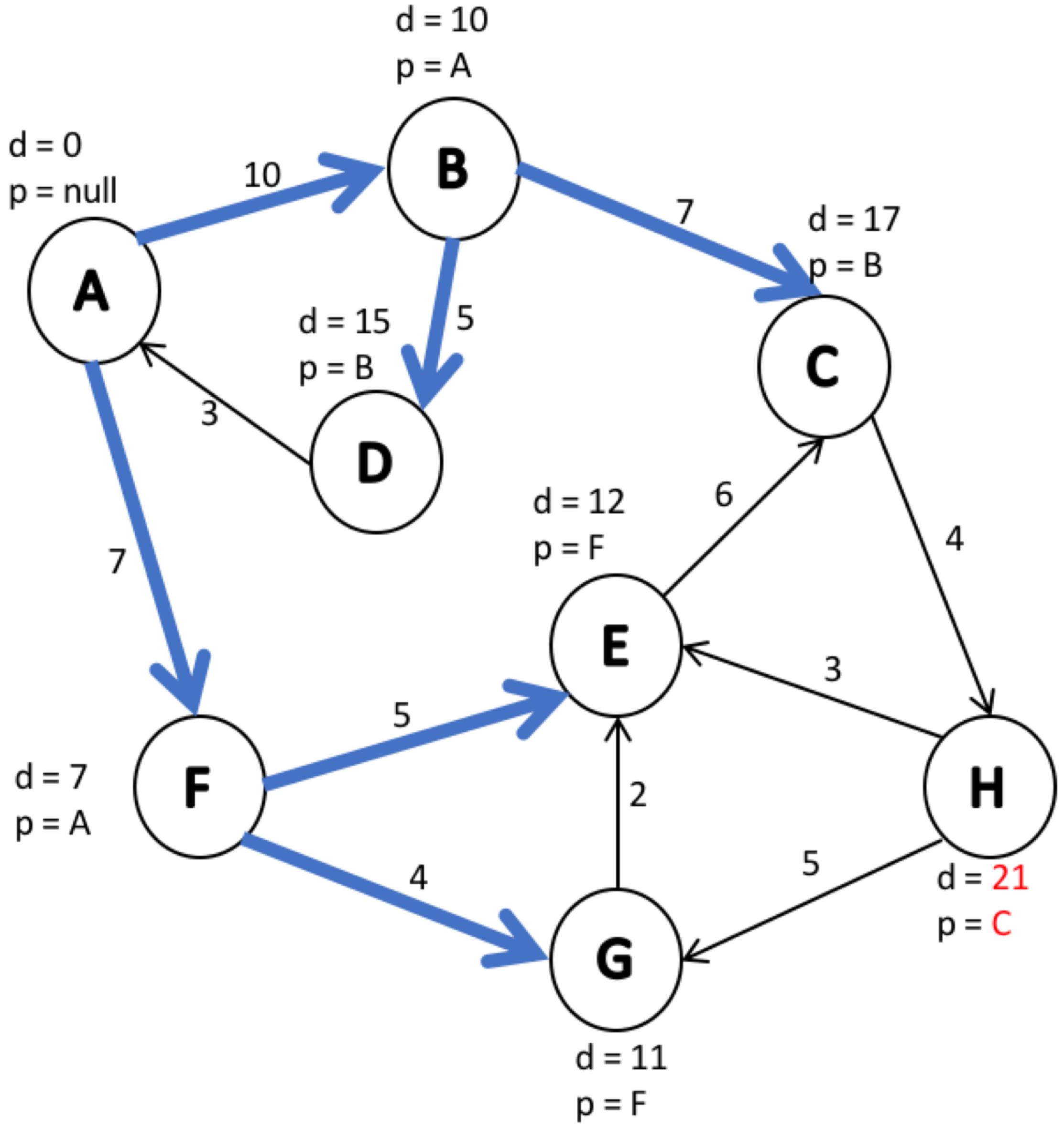
V	d	p
~~A~~	~~0~~	~~null~~
~~B~~	~~10~~	~~A~~
C	17	B
D	15	B
~~E~~	~~12~~	~~F~~
~~F~~	~~7~~	~~A~~
~~G~~	~~11~~	~~F~~
H	∞	null

Add the edge, pop D and update (nothing was updated)



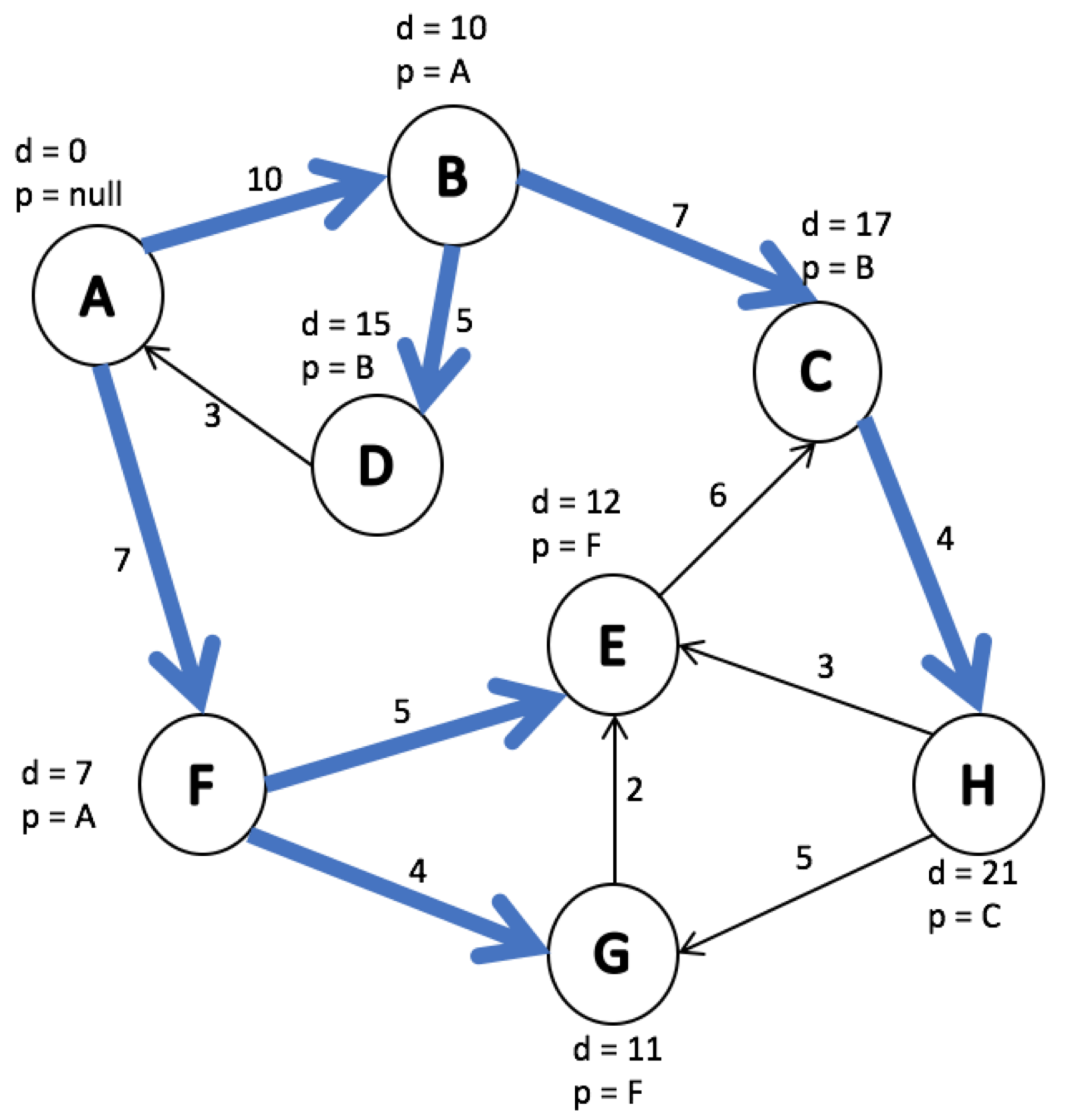
V	d	p
~~A~~	~~0~~	~~null~~
~~B~~	~~10~~	~~A~~
C	17	B
~~D~~	~~15~~	~~B~~
~~E~~	~~12~~	~~F~~
~~F~~	~~7~~	~~A~~
~~G~~	~~11~~	~~F~~
H	∞	null

Add the edge, pop C and update

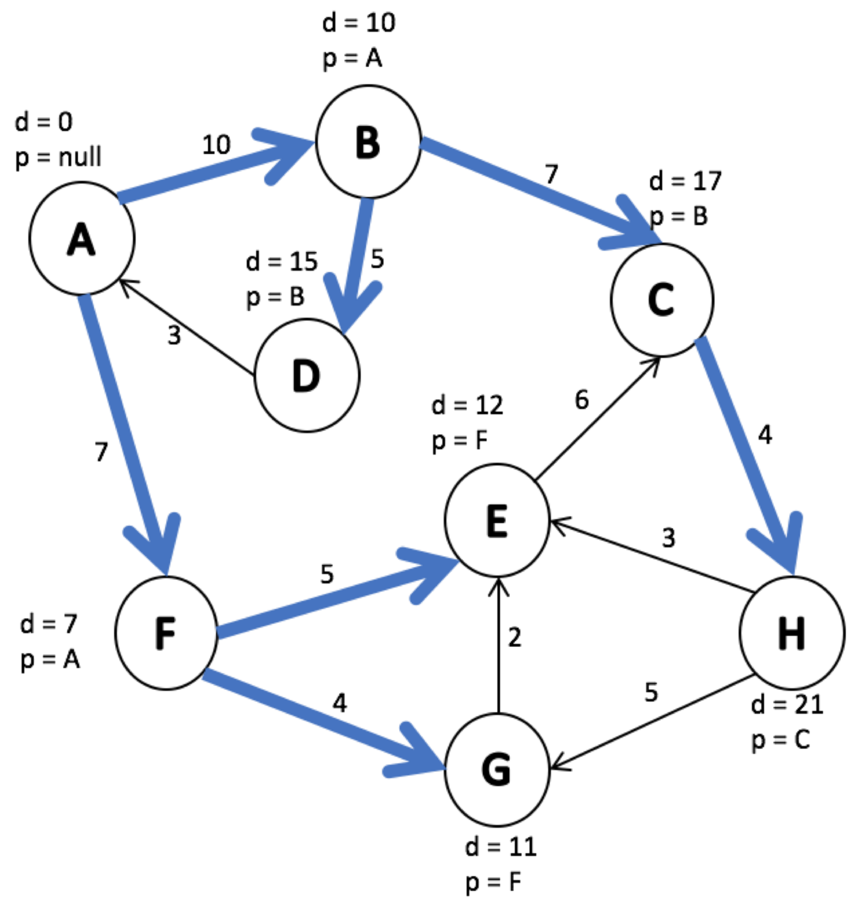


V	d	p
~~A~~	~~0~~	~~null~~
~~B~~	~~10~~	~~A~~
~~C~~	~~17~~	~~B~~
~~D~~	~~15~~	~~B~~
~~E~~	~~12~~	~~F~~
~~F~~	~~7~~	~~A~~
~~G~~	~~11~~	~~F~~
H	21	C

Add the edge from C to H and pop H. heap becomes empty



V	d	p
A	0	null
B	10	A
C	17	B
D	15	B
E	12	F
F	7	A
G	11	F
H	21	C



V	d	p
A	0	null
B	10	A
C	17	B
D	15	B
E	12	F
F	7	A
G	11	F
H	21	C

What is the path from A to H?

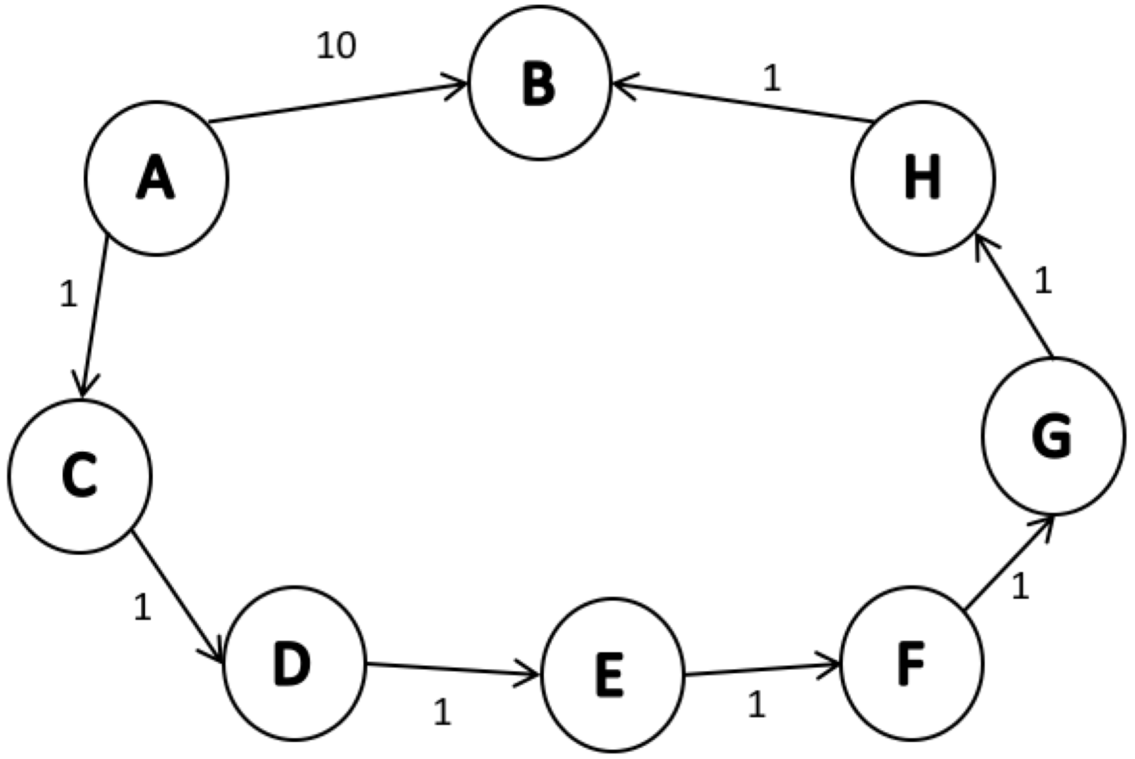
Start at H and trace back the predecessor nodes → A-B-C-H

V	d	p
A	0	null
B	10	A
C	17	B
D	15	B
E	12	F
F	7	A
G	11	F
H	21	C

The shortest path from A to H is 21.

The time to find this information is O(1).

If there is no path to a particular vertex, we will have infinity as distance.



If we want to get the most direct path instead of the shortest path, we can adjust edge weights.
For example, we can add 1 to all edges. In that case, path A-C-D-E-F-G-H-B will be of length 14, while path A-B will be 11 and Dijkstra would pick A-B.

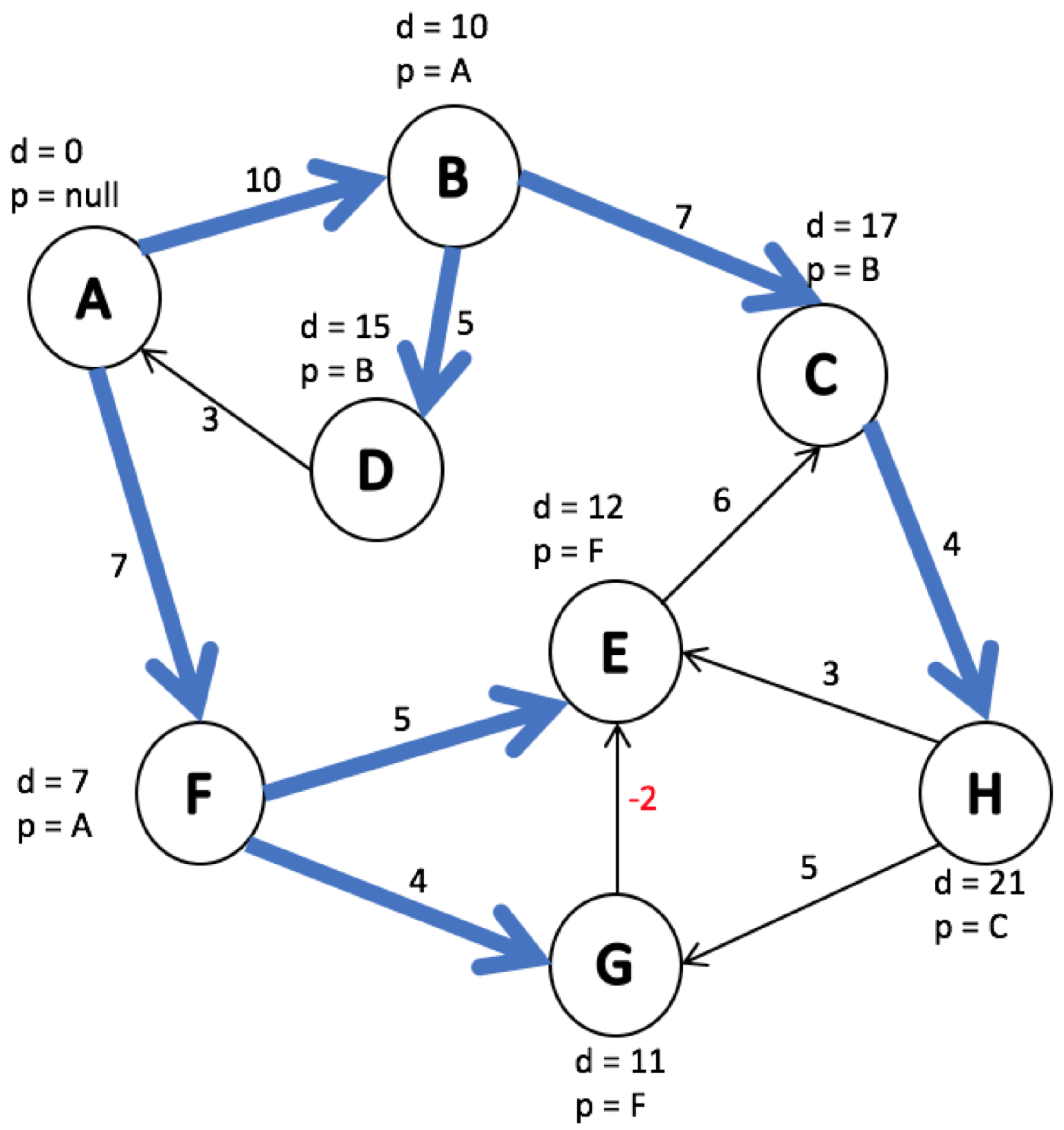
The shortest path will be A-C-D-E-F-G-H-B instead of A-B because the first path has length 7 and the second path has length 10.

When there is a tie in path lengths, it is up to us to decide how we want to handle that.

Can Dijkstra's algorithm handle undirected graphs?
Yes, it can. It will not go back or in loop because that will increase the path length.

Can Dijkstra's algorithm handle graph with negative cycles?
No, because negative weight cycle doesn't have defined shortest path. We can always find a shorter path which leads to negative infinity.

Dijkstra's algorithm can handle graphs with negative edges, but no negative cycles - it will finish, there will be no infinite loop. However, <u>it will not produce the shortest path.</u>



V	d	p
A	0	null
B	10	A
C	17	B
D	15	B
E	12	F
F	7	A
G	11	F
H	21	C

Running time of Dijkstra’s algorithm

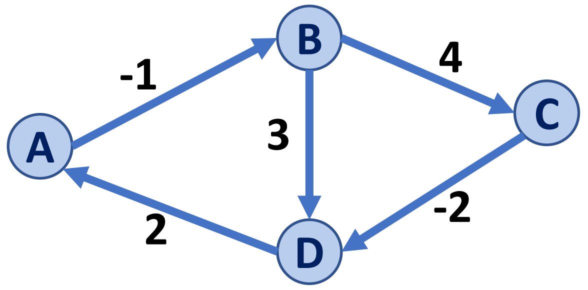
Remember, we built Dijkstra’s algorithm on top of Prim’s algorithm.

We only added two lines of code which take O(1).

Therefore, Dijkstra’s running time is the same as Prim’s.

Floyd-Warshall Algorithm

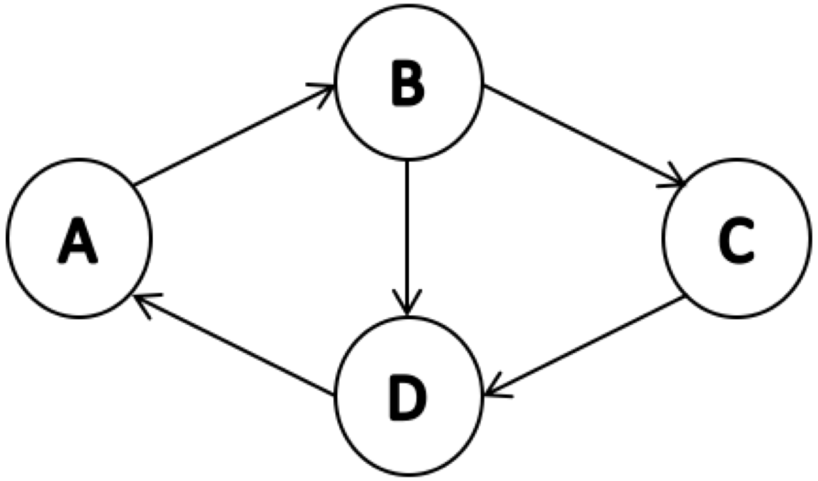
Floyd-Warshall's Algorithm is an alterative to Dijkstra in the presence of negative-weight edges (not negative weight cycles).



```
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]
```

Algorithm setup:

- Maintain a table (matrix) that has the shortest known paths between vertices.
- Initialize the table with three possible values:
- self edges to 0
- edges by edge weights
- unknown paths to infinity

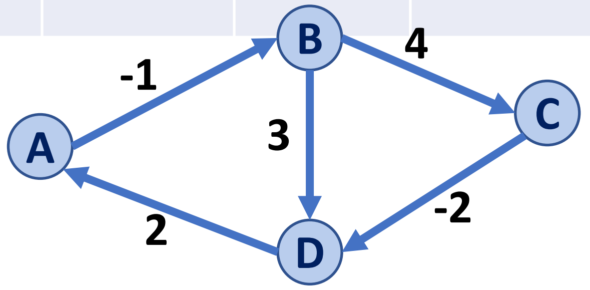


	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	∞	∞	0

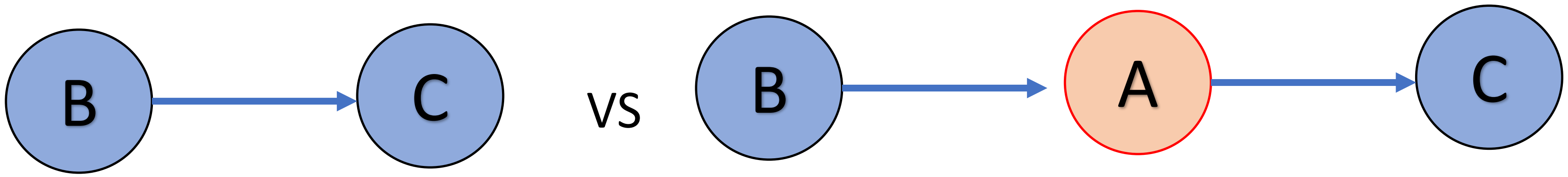
Floyd-Warshall Algorithm

```
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      foreach (Vertex k : G):
        if d[u, v] > d[u, k] + d[k, v]:
          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	∞	∞	0



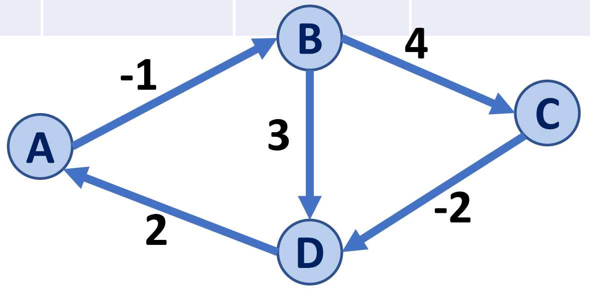
Can we add a vertex in between to vertices to make the distance shorter.



Floyd-Warshall Algorithm

```
foreach (Vertex u : G):
  foreach (Vertex v : G):
    foreach (Vertex k : G):
      if d[u, v] > d[u, k] + d[k, v]:
        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	∞	∞	0



Let us consider k=A:

$B \rightarrow C$ 4 vs. $B \rightarrow A \rightarrow C$ $+\infty$

$B \rightarrow D$ 3 vs. $B \rightarrow A \rightarrow D$ $+\infty$

$C \rightarrow B$ $+\infty$ vs. $C \rightarrow A \rightarrow B$ $+\infty$

$C \rightarrow D$ -2 vs. $C \rightarrow A \rightarrow D$ $+\infty$

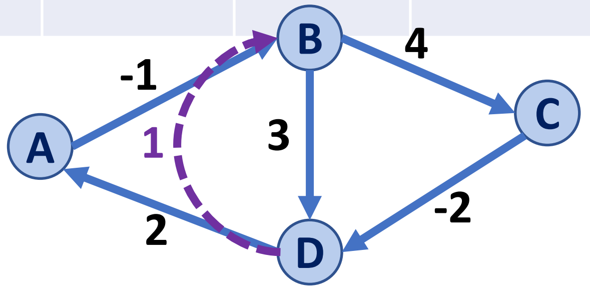
$D \rightarrow B$ $+\infty$ vs. $D \rightarrow A \rightarrow B$ 2+(-1) = 1

$D \rightarrow C$ $+\infty$ vs. $D \rightarrow A \rightarrow C$ $+\infty$

Floyd-Warshall Algorithm

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

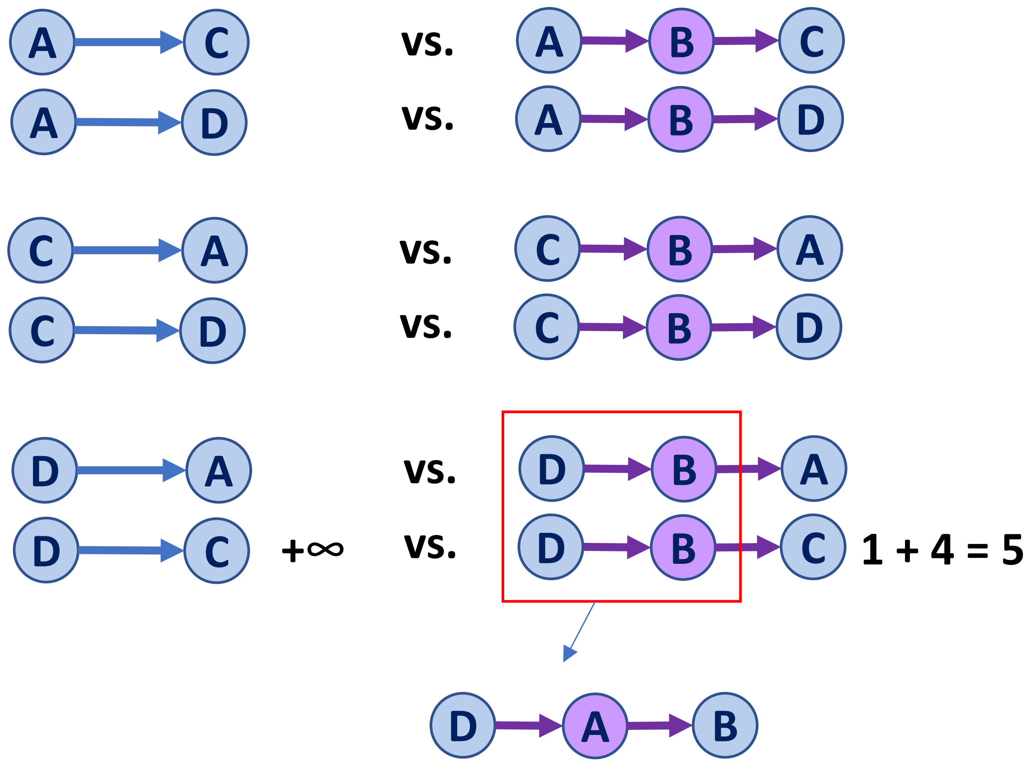


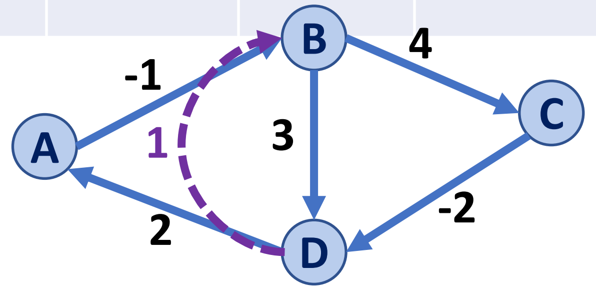
Floyd-Warshall Algorithm

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





This edge does not actually gets created. Values in the matrix saves information about updated path values.

Floyd-Warshall Algorithm

Running time:

$$O(n^3)$$

Floyd-Warshall's algorithm explores all possible paths to determine the shortest path. If we explored all possible paths with Dijkstra's algorithm, the running time would have been much worse than n^3.

When Tree has a cycle…

Good Luck on exam!

Thank you for amazing artworks!