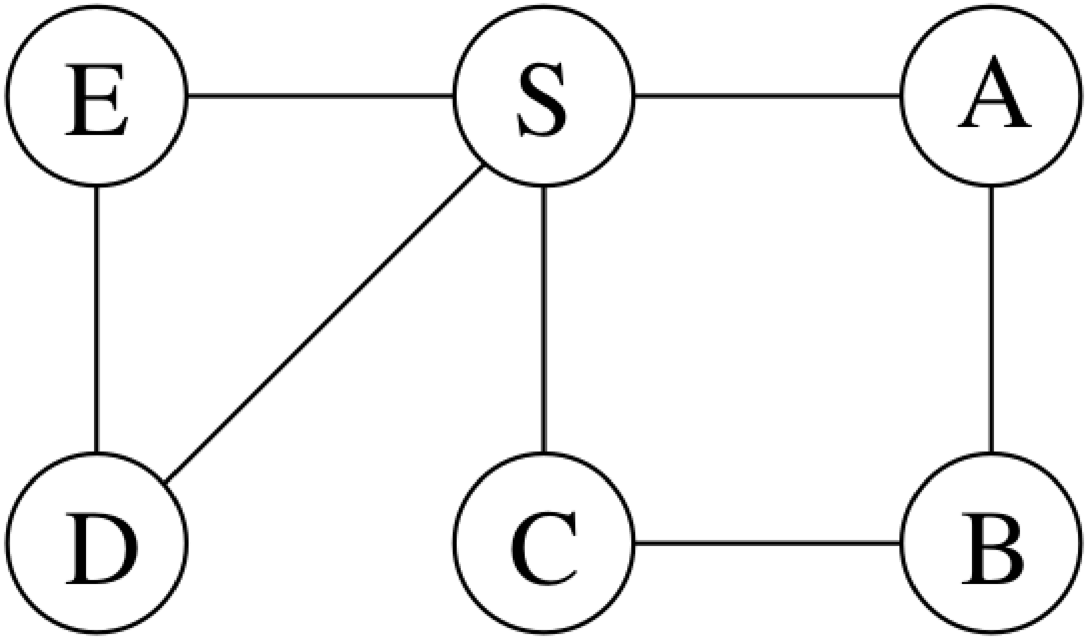
CS 344

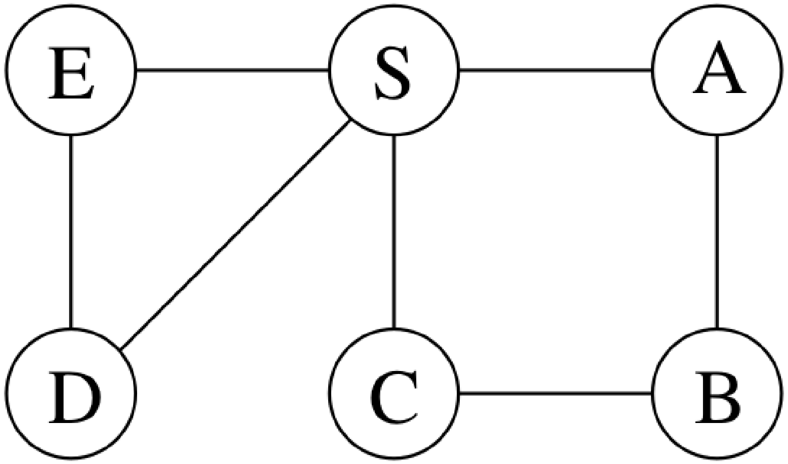
LECTURE 8

GRAPHS, PATHS, AND BFS

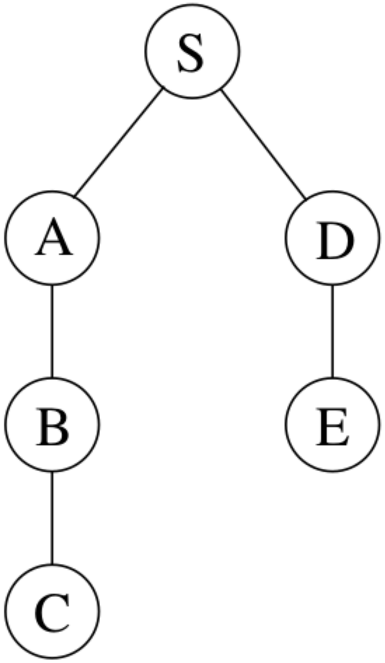
Suppose we have a graph, and do DFS search from S:



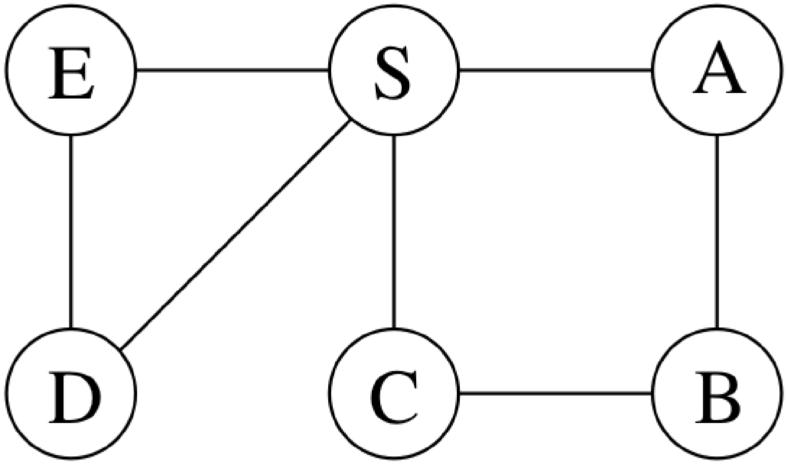
We should get the following tree:



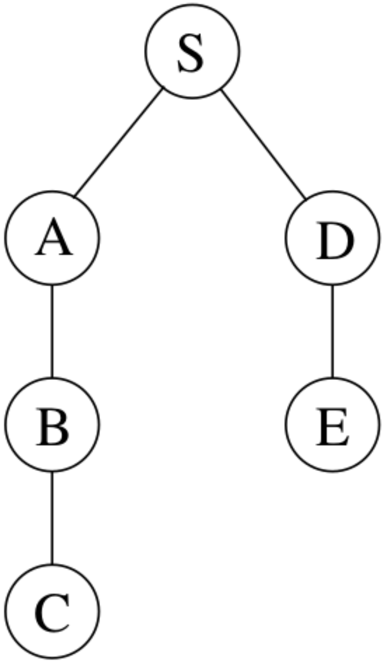
n



We should get the following tree:

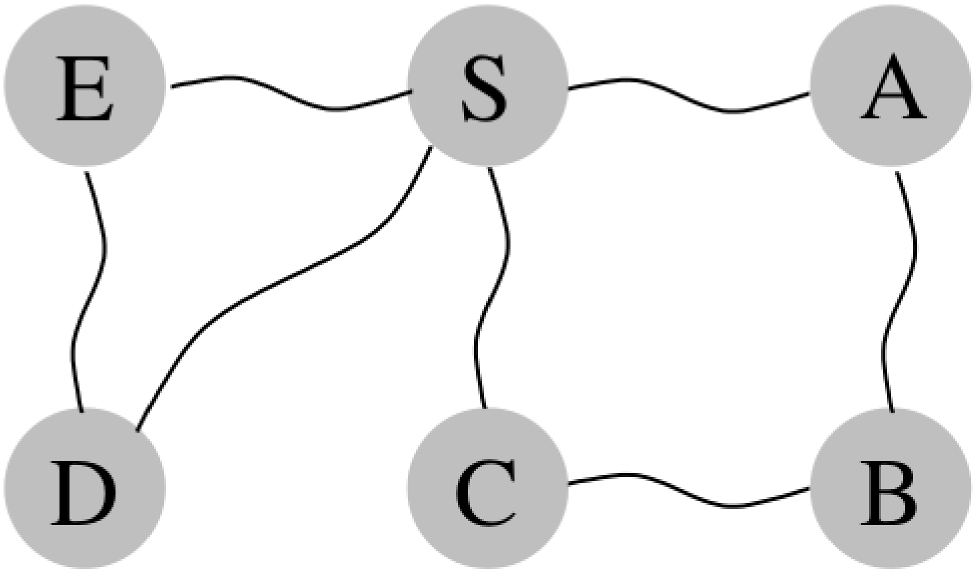


n

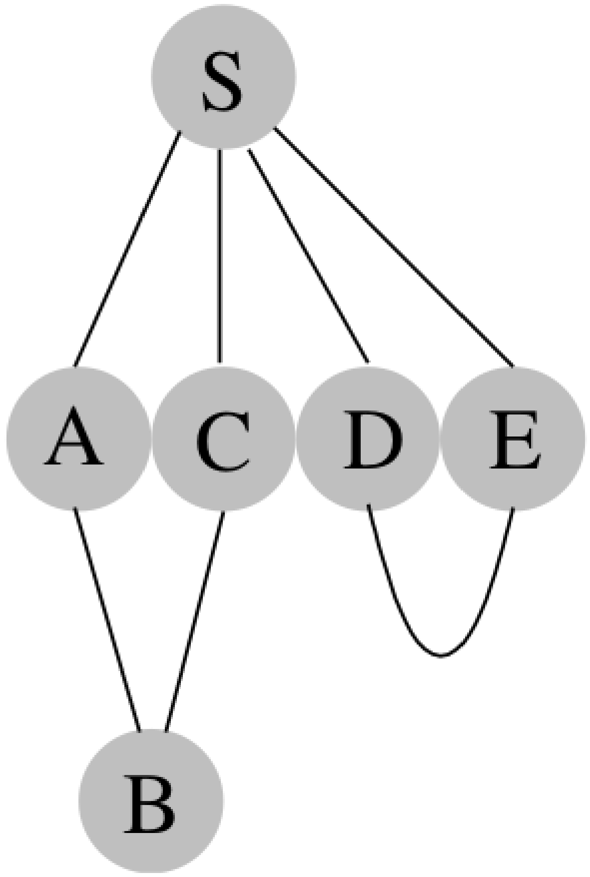


But if we were interested in getting to C, this gives a rather inefficient route of $S \to A \to B \to C$ instead of $S \to C$

Imagine the graph as a physical set of marbles connected by string:



When we pick up the graph by S, we see immediately how to get to C:

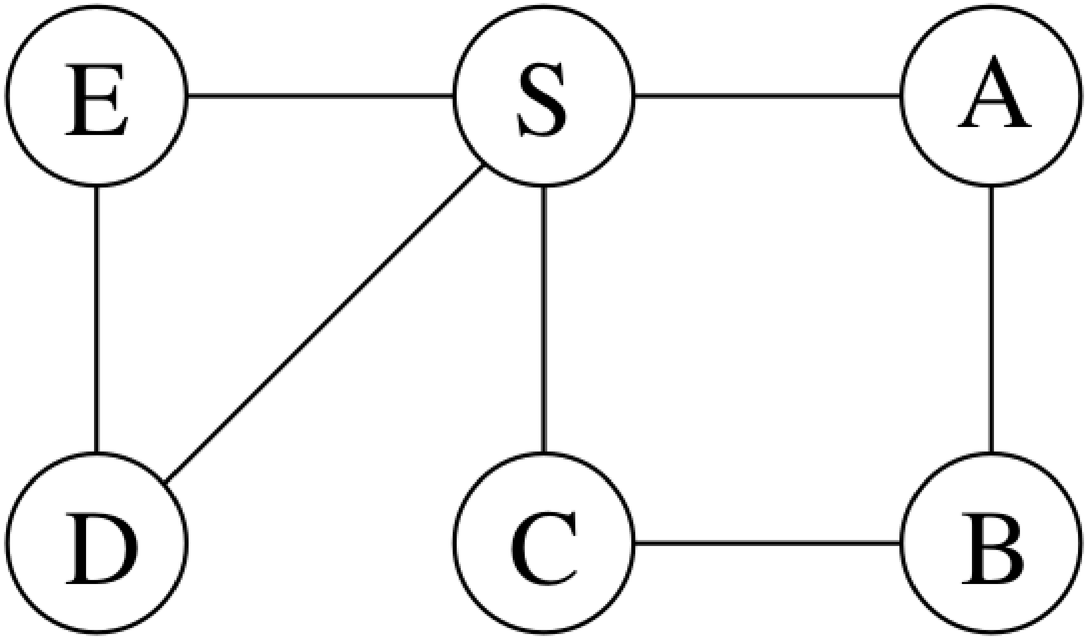


BREADTH-FIRST SEARCH (BFS)

- Proceed layer by layer
- Find layer $d + 1$ by scanning neighbors of layer d

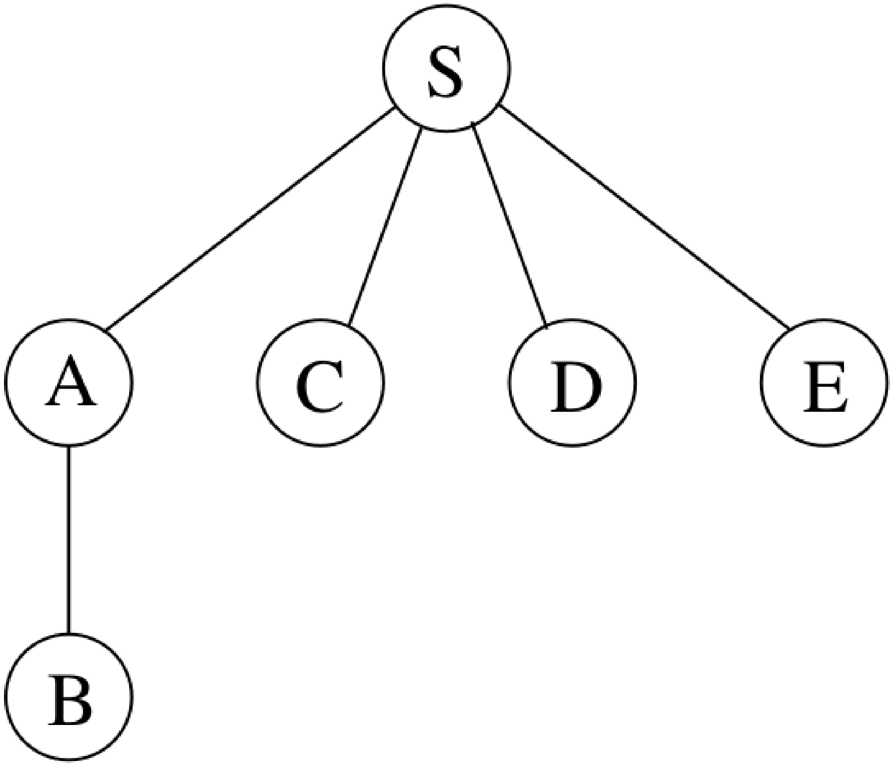
```
procedure bfs(G, s)
Input:    Graph G = (V, E), directed or undirected; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
   u = eject(Q)
   for all edges (u, v) ∈ E:
      if dist(v) = ∞:
         inject(Q, v)
         dist(v) = dist(u) + 1
```

Let's run BFS on our graph, starting at S:



Let's run BFS on our graph, starting at S:

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



IS BFS CORRECT?

That is, for each d, at some point:

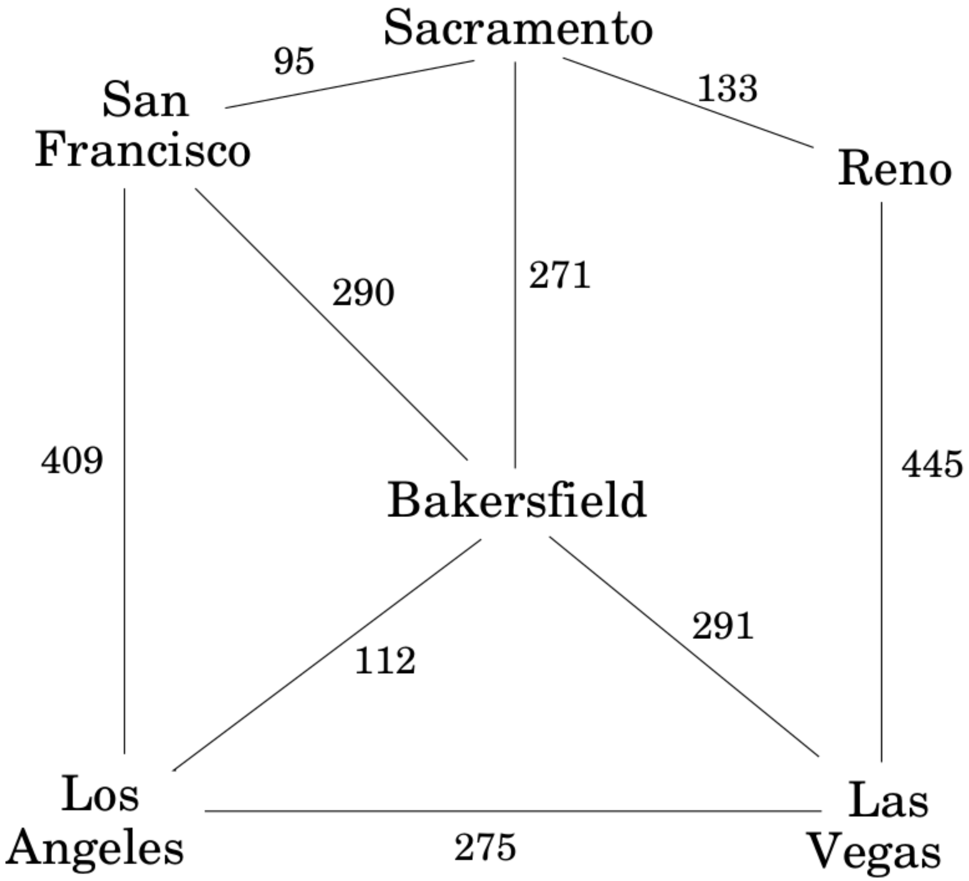
- all nodes with distance $\leq d$ are correctly set
- all other nodes have distance set to ∞
- the queue contains exactly the nodes at distance d

BFS RUNNING TIME

- Each vertex put onto queue once
- Examine each edge once (directed) or twice (undirected)

$$O(|V| + |E|)$$

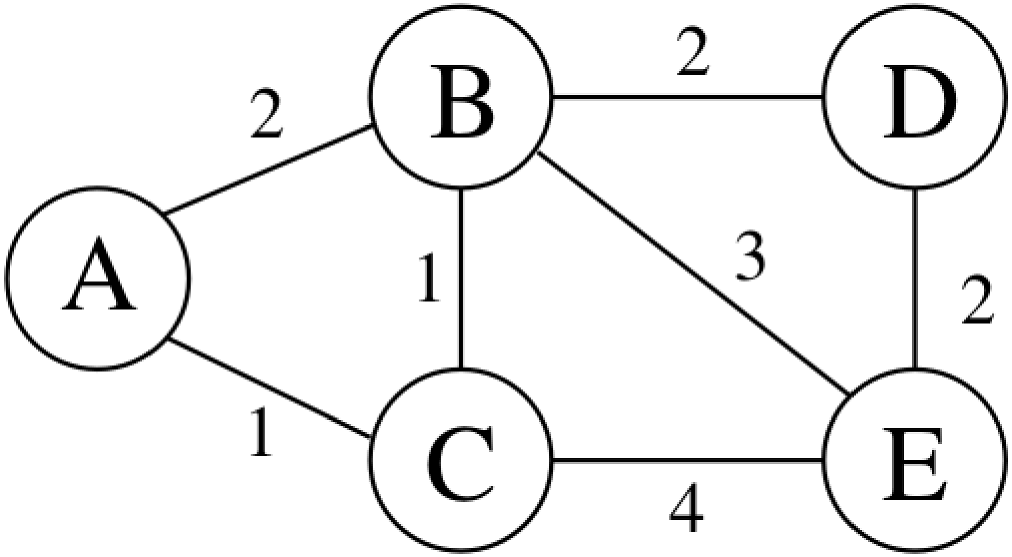
Edges are often weighted with some kind of length:



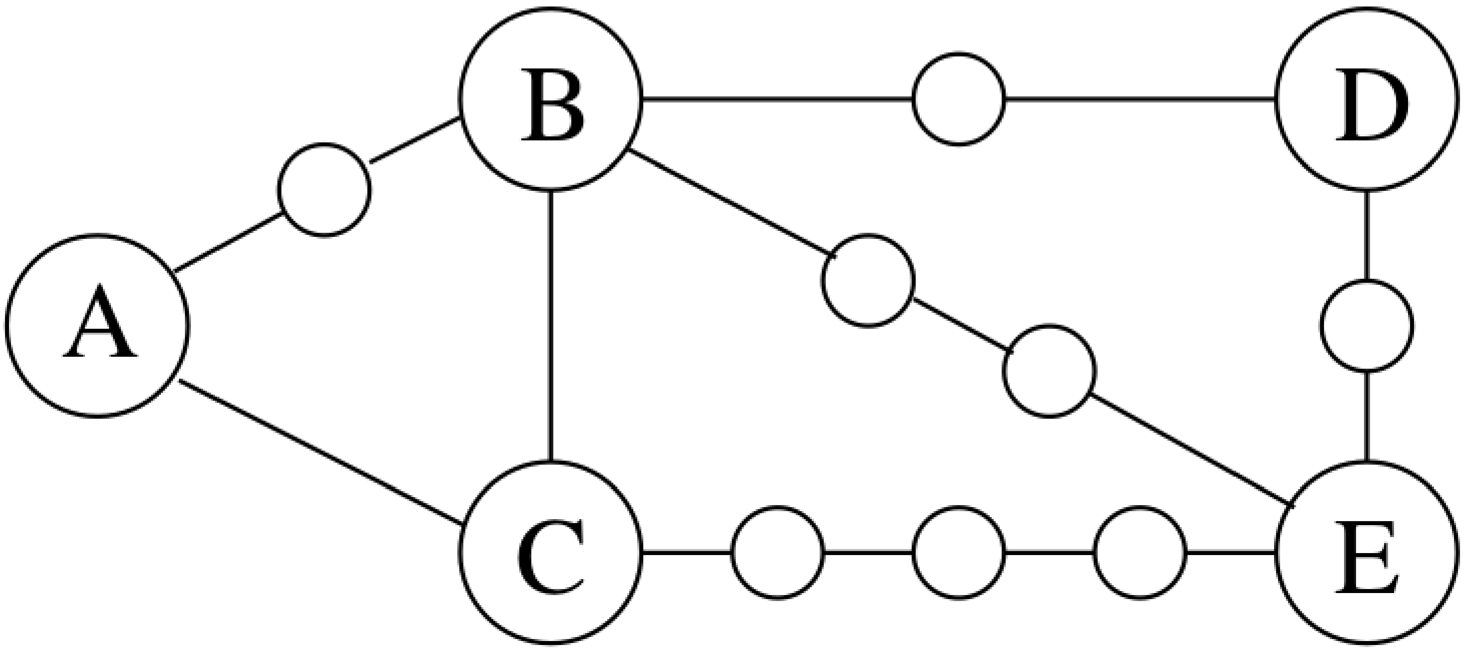
We could compute shortest paths using BFS:

- Break edges into unit lengths with dummy nodes!

Suppose this is our graph:

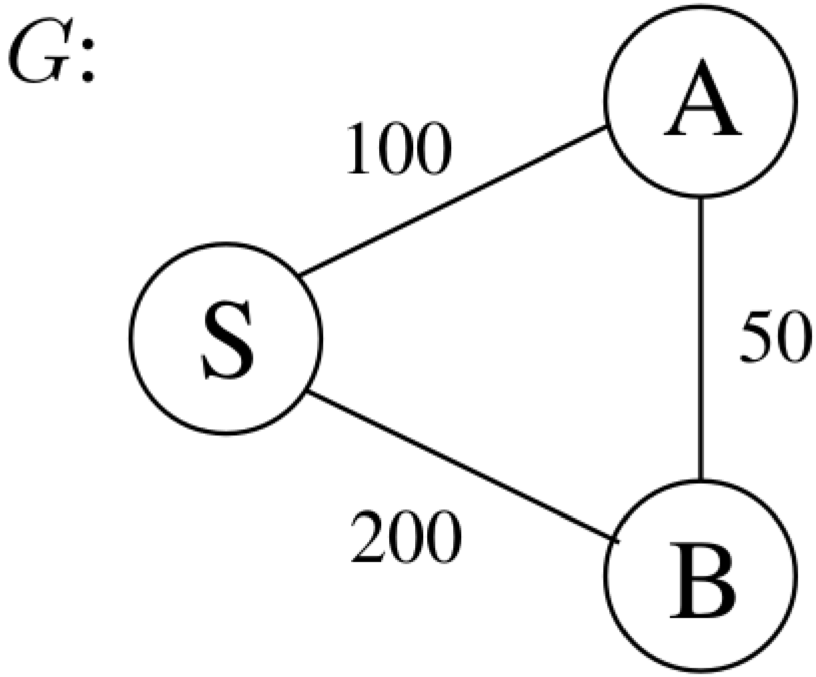


We can break up each edge based on its length:

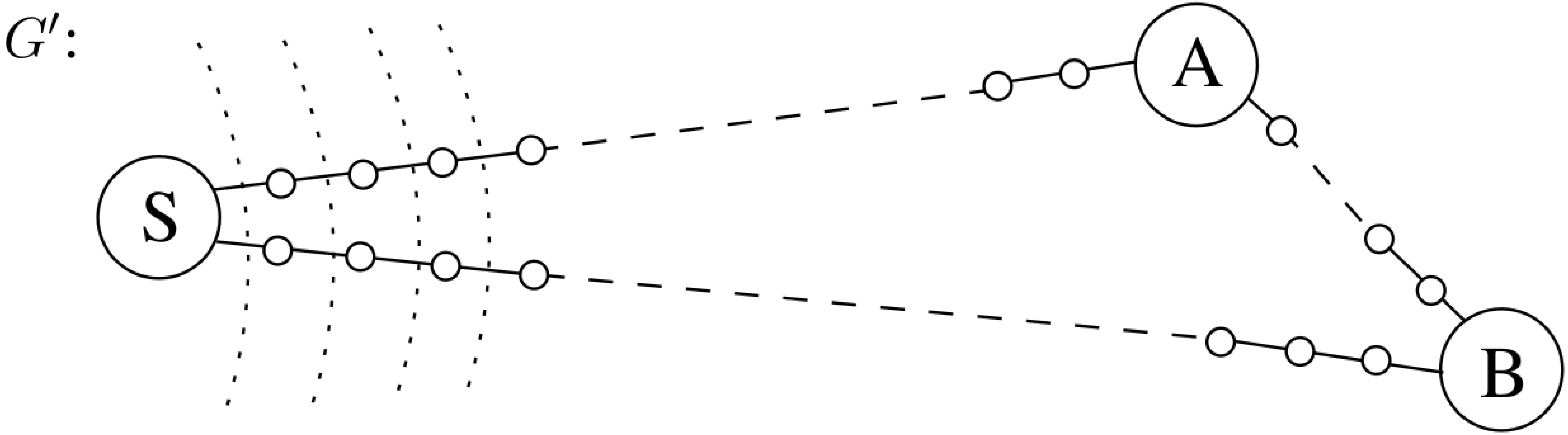


And then run BFS as usual

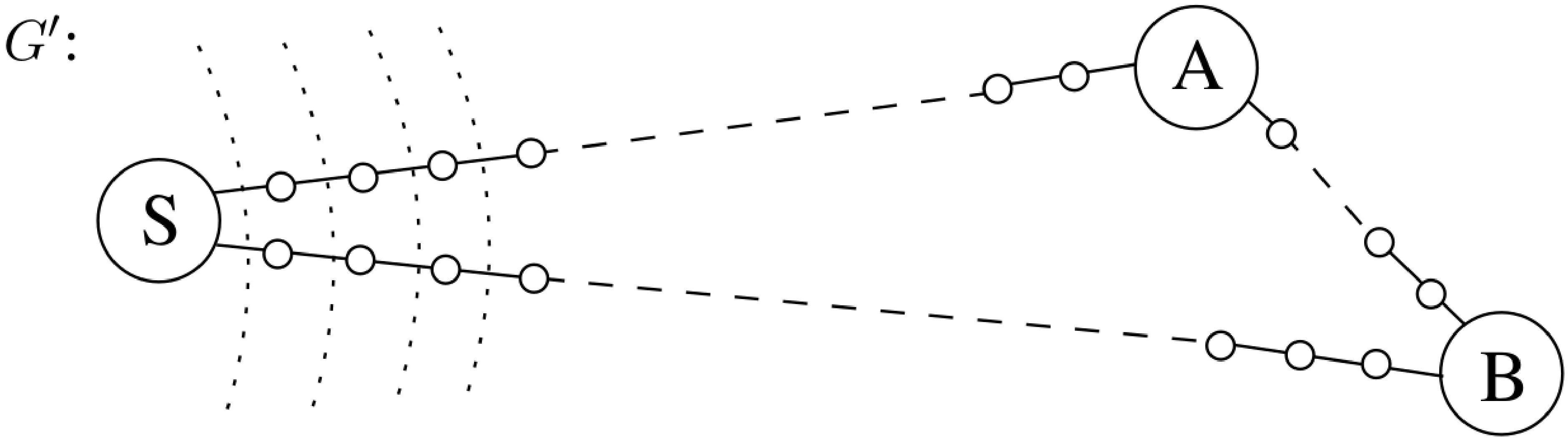
But what if this was our graph?



Then BFS is a slow, boring process:



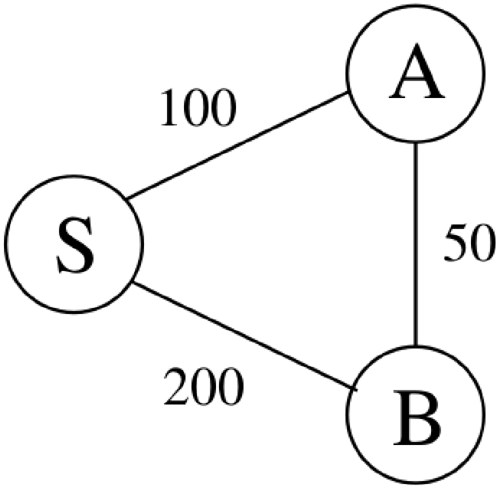
Then BFS is a slow, boring process:



What if we could tell BFS to wake us up when it gets to the interesting part?

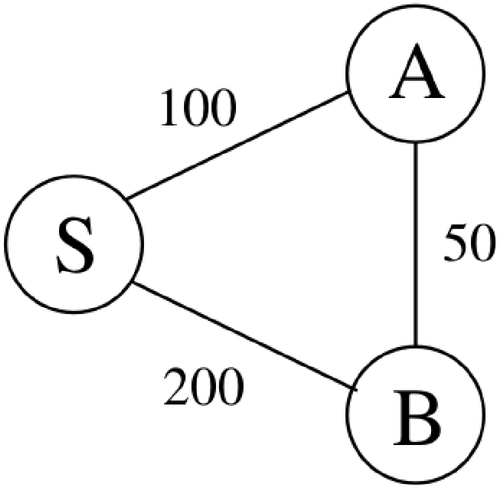
Wake us up when it gets to the interesting part:

G:



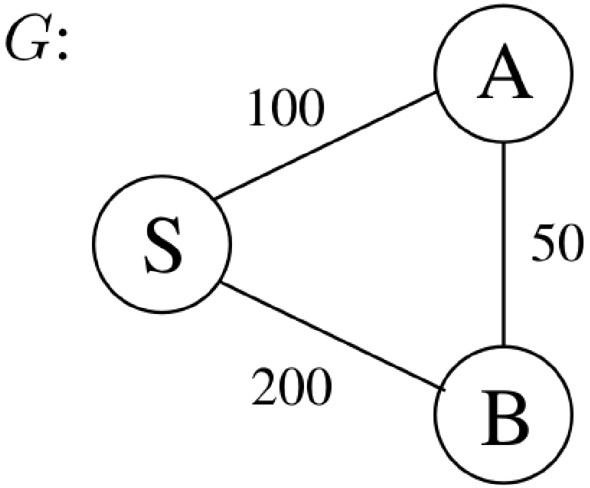
Wake us up when it gets to the interesting part:

G:



- ETAs for A and B are 100 and 200

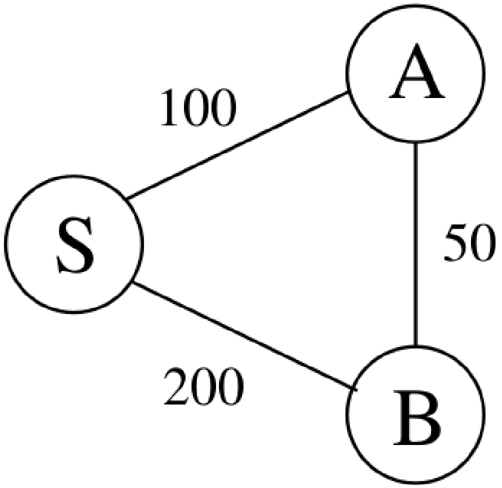
Wake us up when it gets to the interesting part:



- ETAs for A and B are 100 and 200
- Set alarms for 100 and 200

Wake us up when it gets to the interesting part:

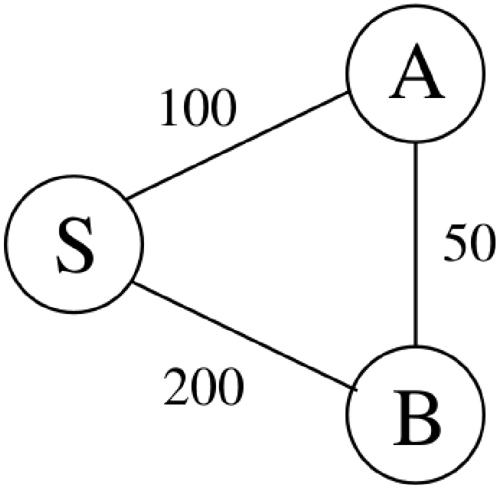
G:



- ETAs for A and B are 100 and 200
- Set alarms for 100 and 200
- Wake up at time 100

Wake us up when it gets to the interesting part:

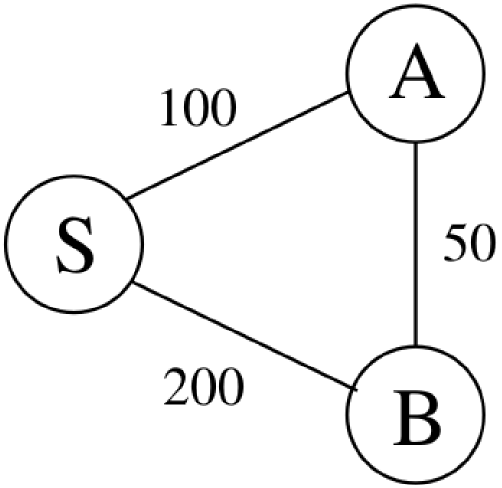
G:



- ETAs for A and B are 100 and 200
- Set alarms for 100 and 200
- Wake up at time 100
- New ETA for B is 150

Wake us up when it gets to the interesting part:

G:



- ETAs for A and B are 100 and 200
- Set alarms for 100 and 200
- Wake up at time 100
- New ETA for B is 150
- Change B's alarm to 150

Wake us up when it gets to the interesting part:

Wake us up when it gets to the interesting part:

- Set alarm for node s at time 0

Wake us up when it gets to the interesting part:

- Set alarm for node s at time 0
- Repeat until no more alarms:

Wake us up when it gets to the interesting part:

- Set alarm for node s at time 0
- Repeat until no more alarms:

  Say the next alarm goes off at time T for node u

Wake us up when it gets to the interesting part:

- Set alarm for node s at time 0
- Repeat until no more alarms:

  Say the next alarm goes off at time T for node u

  - Then the distance from s to u is T

Wake us up when it gets to the interesting part:

- Set alarm for node s at time 0
- Repeat until no more alarms:

  Say the next alarm goes off at time T for node u

  - Then the distance from s to u is T
  - For each neighbor v of u:

Wake us up when it gets to the interesting part:

- Set alarm for node s at time 0
- Repeat until no more alarms:

  Say the next alarm goes off at time T for node u

  - Then the distance from s to u is T
  - For each neighbor v of u:
    - If there's no alarm for v, set one for $T + l(u, v)$

Wake us up when it gets to the interesting part:

- Set alarm for node s at time 0
- Repeat until no more alarms:

  Say the next alarm goes off at time T for node u

  - Then the distance from s to u is T
  - For each neighbor v of u:
    - If there's no alarm for v, set one for $T + l(u, v)$
    - If v's alarm is later than that, reduce it to this

This is essentially Dijkstra's algorithm!

(This is essentially Dijkstra)

DIJKSTRA'S ALGORITHM

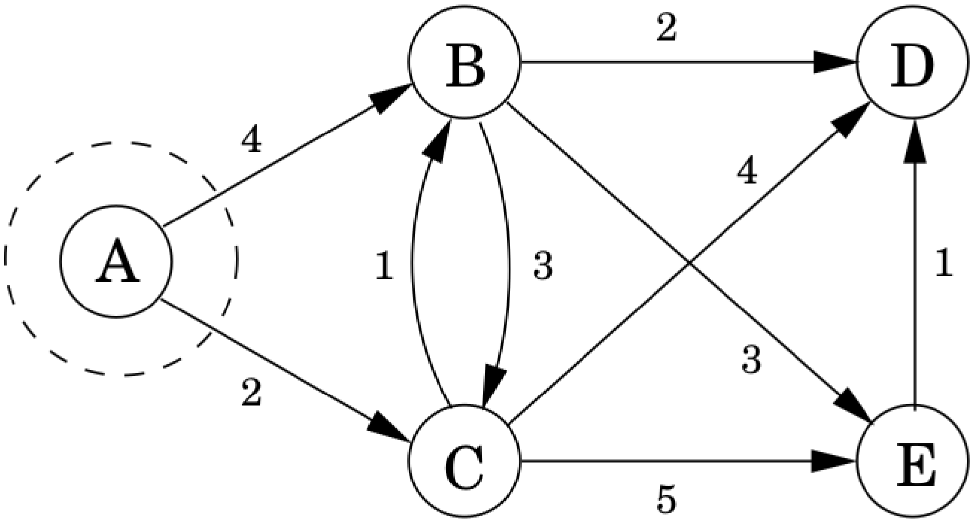
Given a graph G and a starting vertex s,
find shortest paths to all reachable vertices

DIJKSTRA'S ALGORITHM

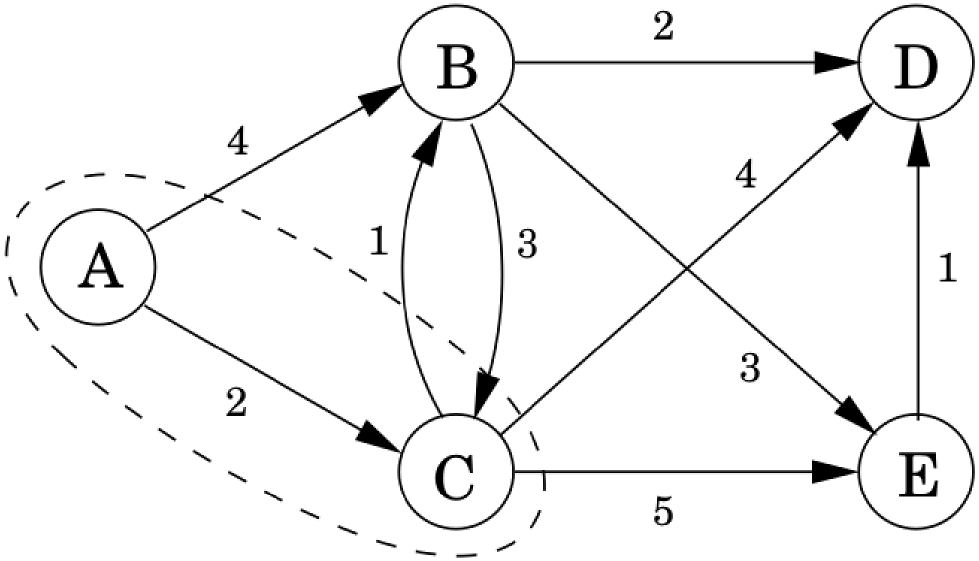
We need to use a **priority queue** with these operations:

- insert
- decrease-key
- delete-min
- make-queue

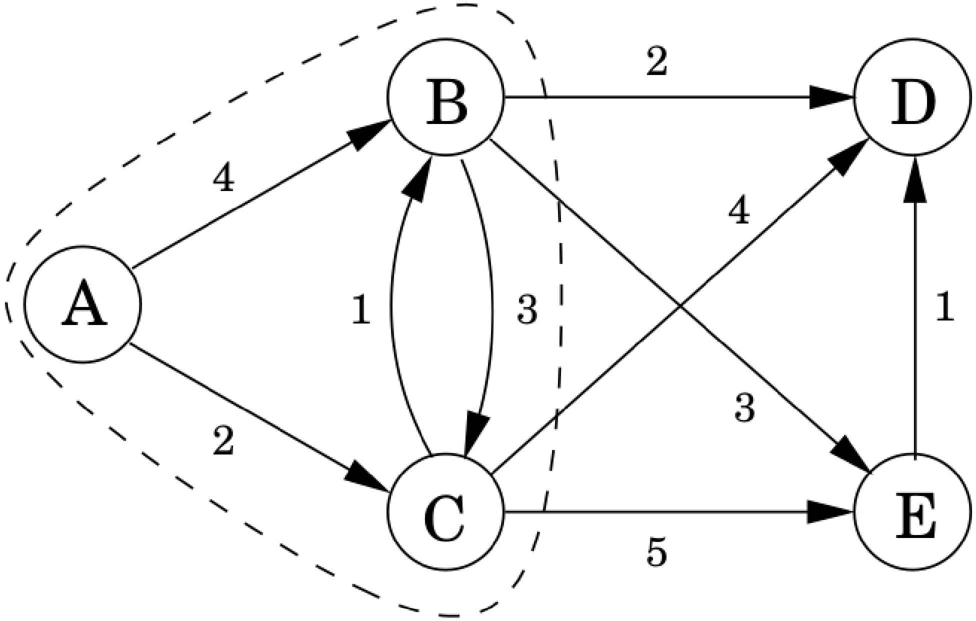
```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil
dist(s) = 0

H = makequeue(V)  (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u, v) ∈ E:
      if dist(v) > dist(u) + l(u, v):
         dist(v) = dist(u) + l(u, v)
         prev(v) = u
         decreasekey(H, v)
```



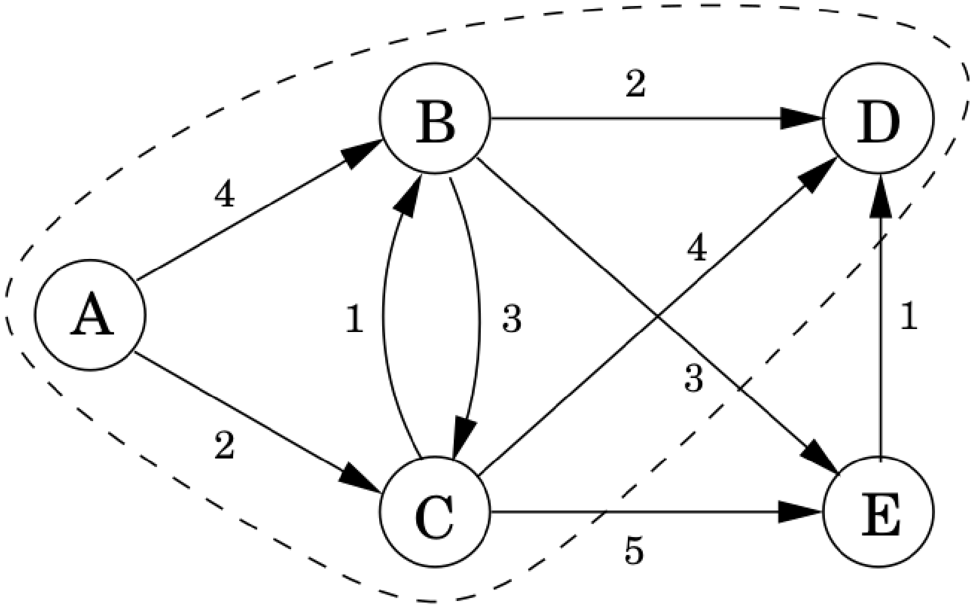
A: 0	**D**: ∞
B: 4	**E**: ∞
C: 2	



A: 0	**D**: 6
B: 3	**E**: 7
C: 2	

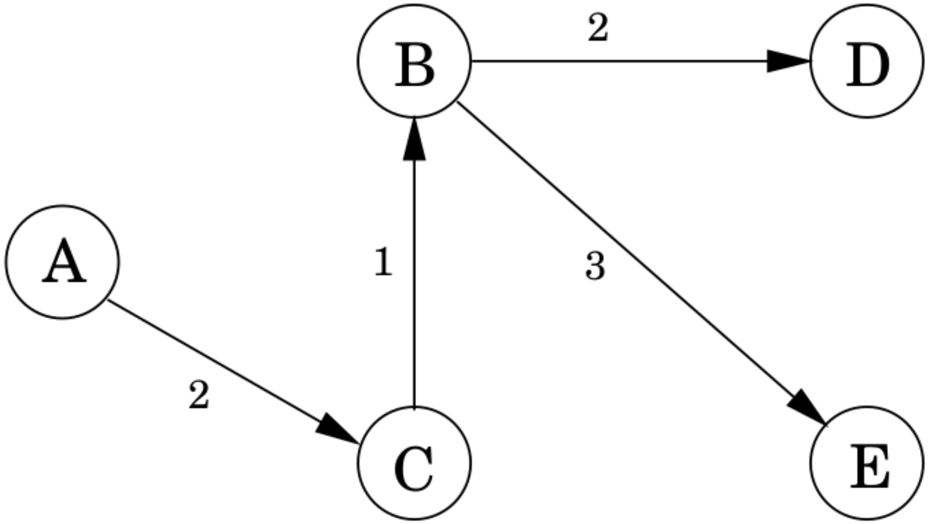


A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	**D**: 5
B: 3	**E**: 6
C: 2	

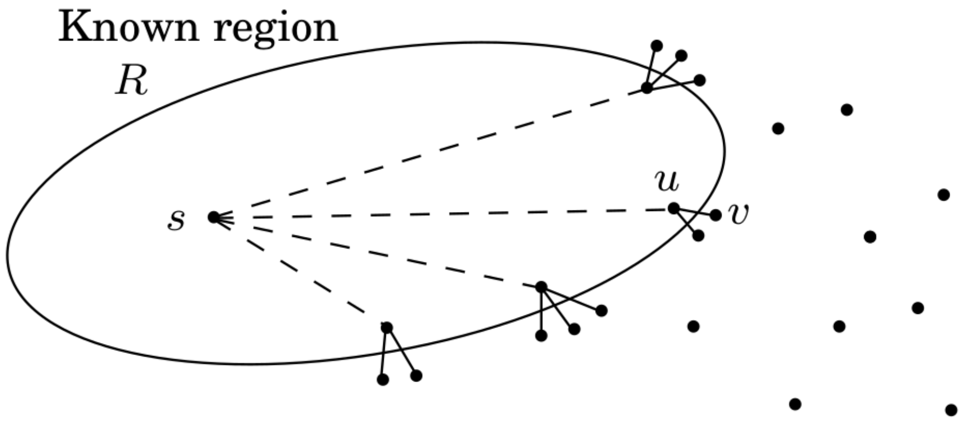
Finally, we get this tree for paths from A:



Dijkstra's algorithm is basically just BFS:

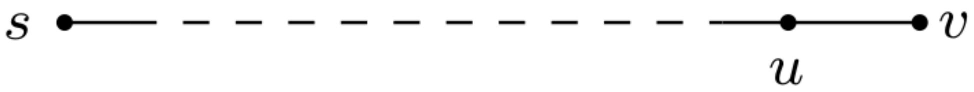
- Instead of a regular queue,
- use a priority queue to account for lengths

Another way of looking at this:



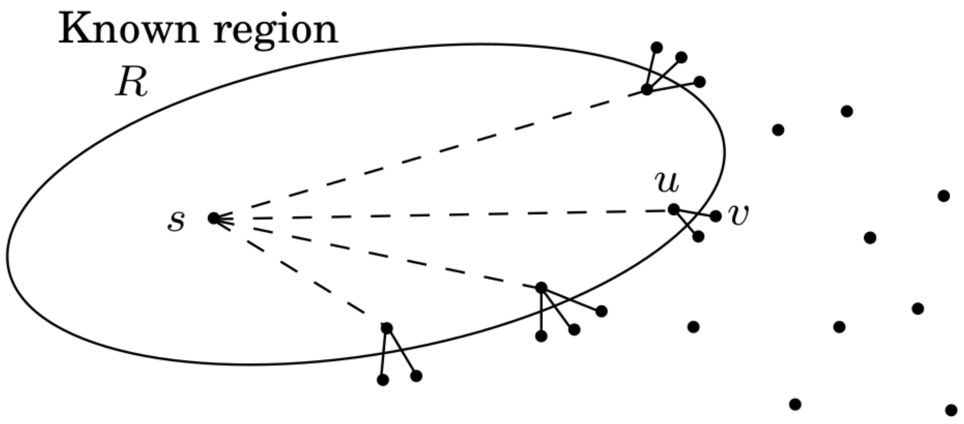
- Start at s
- Grow the "known region" R to larger distances
- Next node added to R is the one closest to s

How do we find the next node v outside R?



- Consider shortest path $s \to \cdots \to u \to v$
- u closer to s than v
- $u \in R$
- So $s \to v$ path extends a currently known path by one edge

How do we find the next node v outside R?



- Try all single-edge extentions, find the shortest path
- Its endpoint is v

That is, v is the node outside R where

$\mathrm{distance}(s, u) + l(u, v)$ is minimized, for all $u \in R$

```
Initialize dist(s) to 0, other dist(·) values to ∞
R = { } (the ''known region'')
while R ≠ V:
    Pick the node v ∉ R with smallest dist(·)
    Add v to R
    for all edges (v, z) ∈ E:
        if dist(z) > dist(v) + l(v, z):
            dist(z) = dist(v) + l(v, z)
```

DIJKSTRA'S RUNNING TIME

- makequeue: at most $|V|$ insert operations
- $|V|$ deletemin operations
- $|V| + |E|$ insert/decreasekey operations

Time depends on implementation, but if we use a binary heap:

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

How can we implement the priority queue?

- Array
- Binary heap
- d-ary heap
- Fibonacci heap

ARRAY

- insert, decreasekey: $O(1)$
- deletemin: $O(n)$

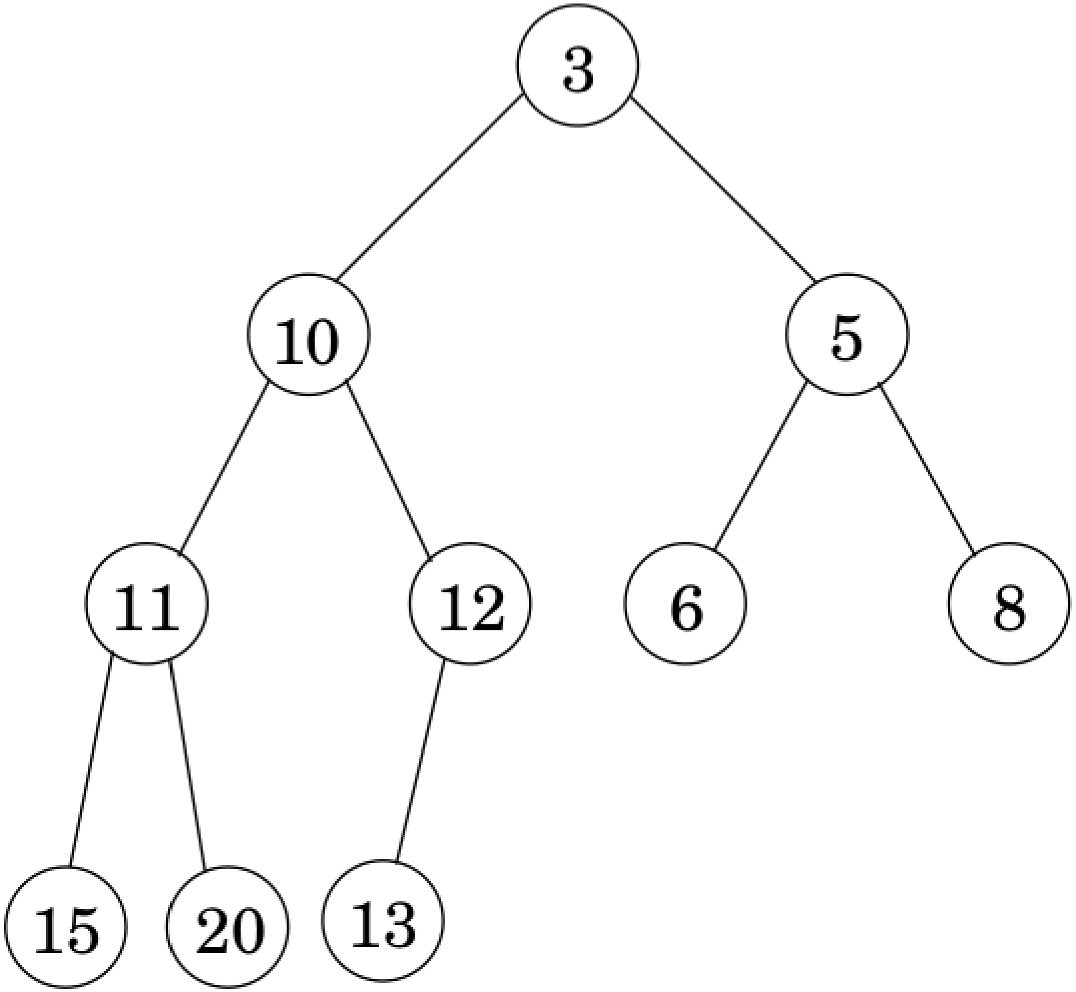
BINARY HEAP

- Complete binary tree (except last level)
- Key value $\leq$ that of its children

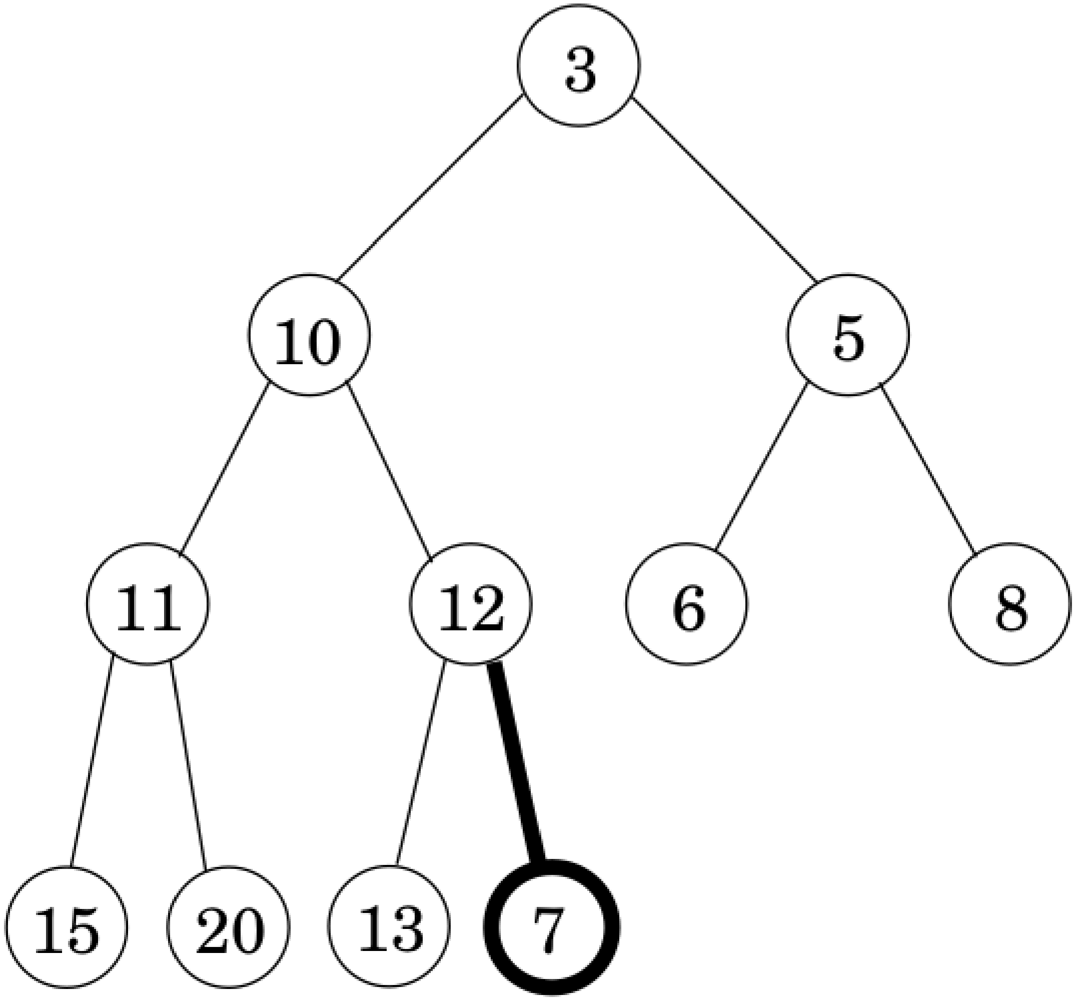
BINARY HEAP

- insert:
  - place node at bottom of tree and "bubble up"
- decreasekey:
  - "bubble up" (it's already in the tree)
- deletemin:
  - remove (and return) root
  - move last element to root
  - "sift down"

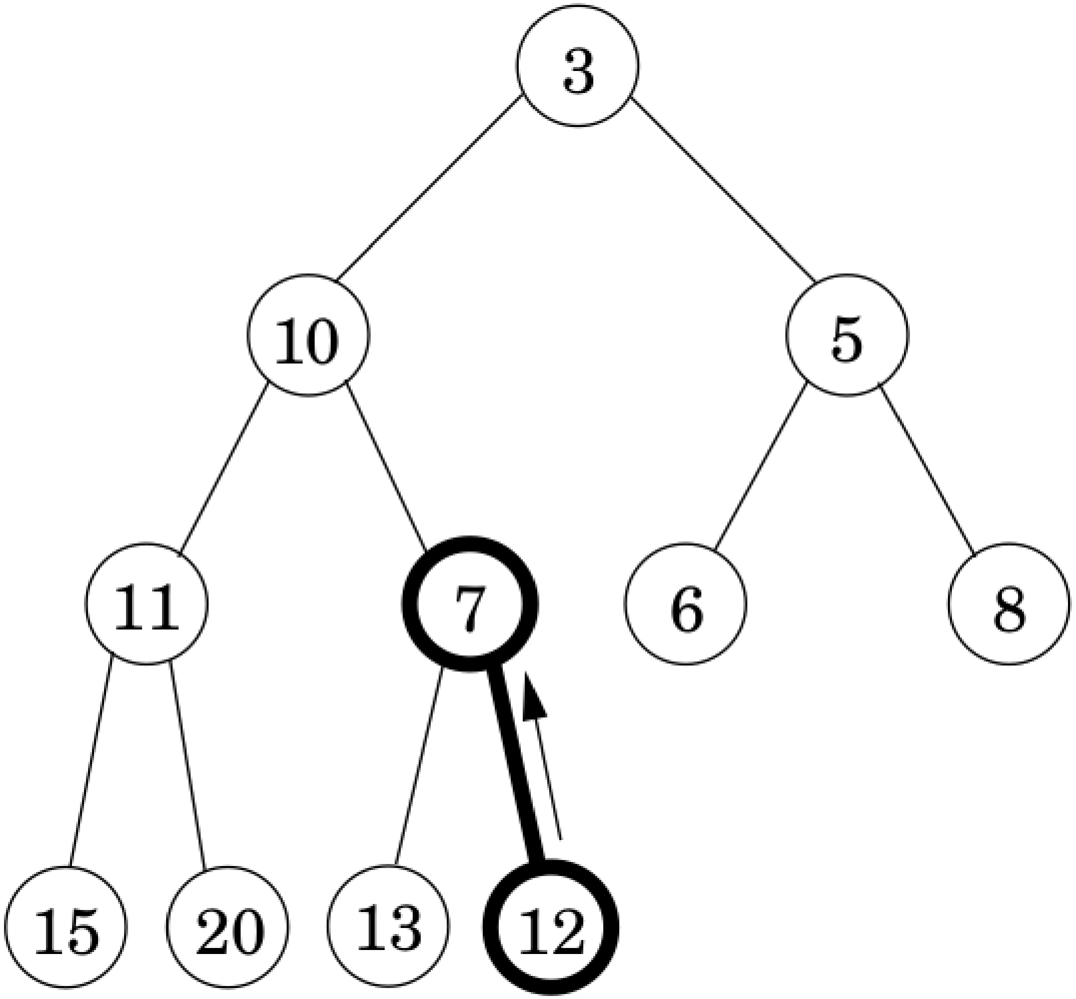
Let's say we have this initial heap:



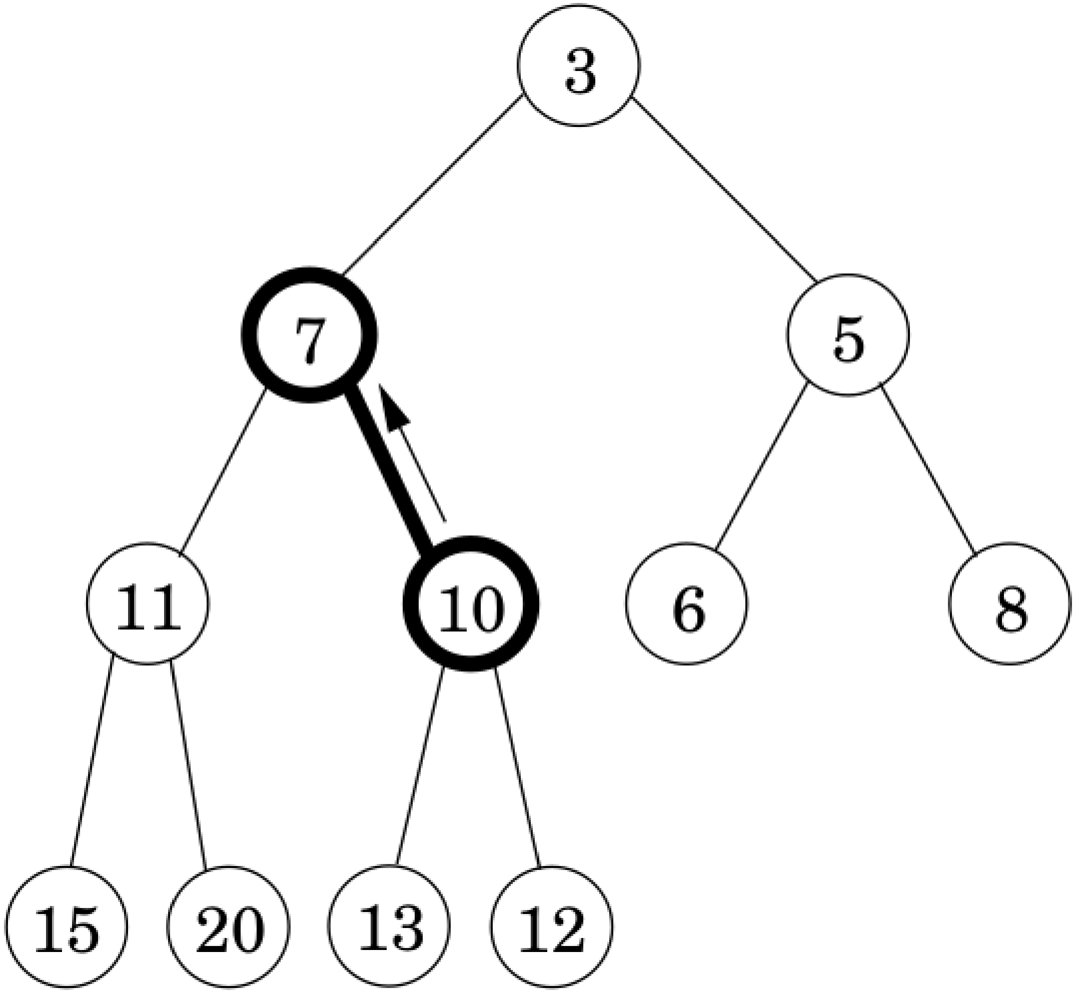
We insert 7:



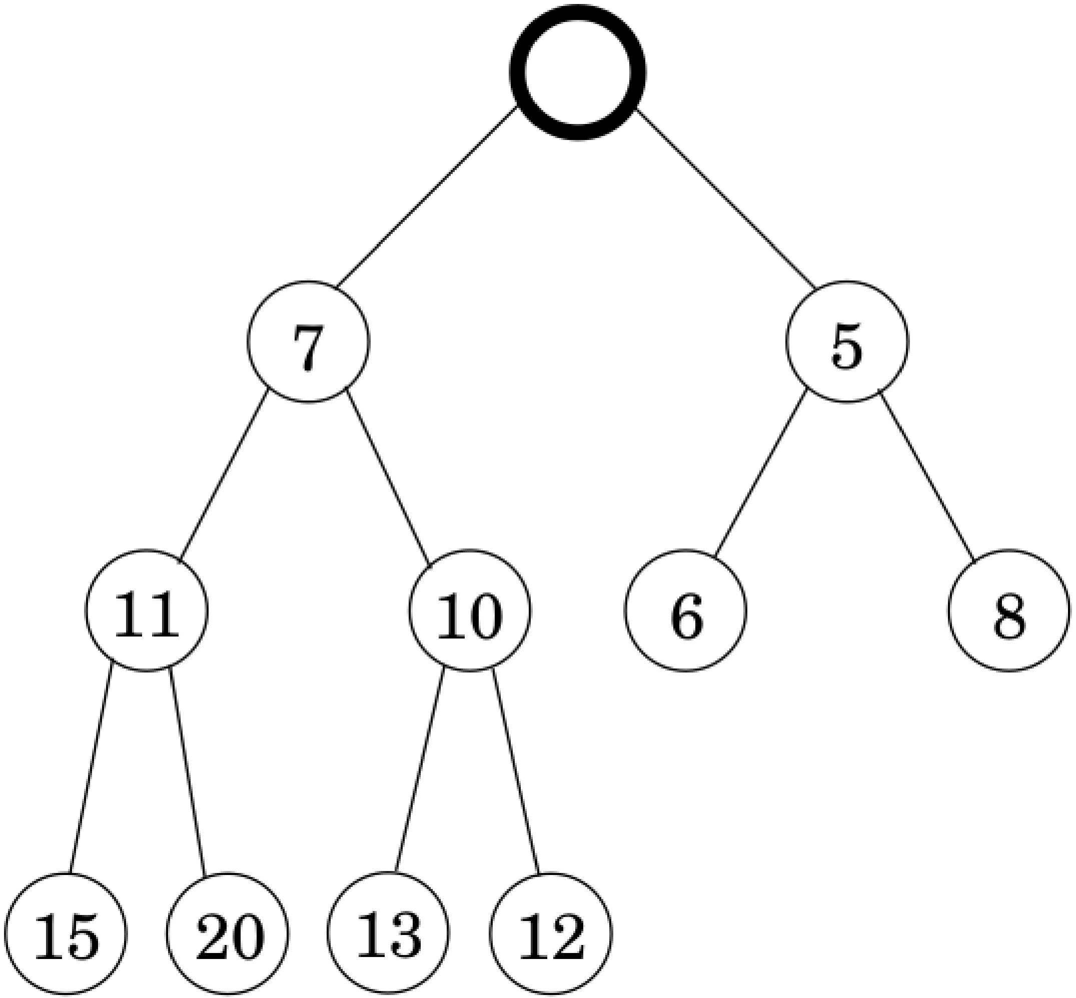
Bubble up:



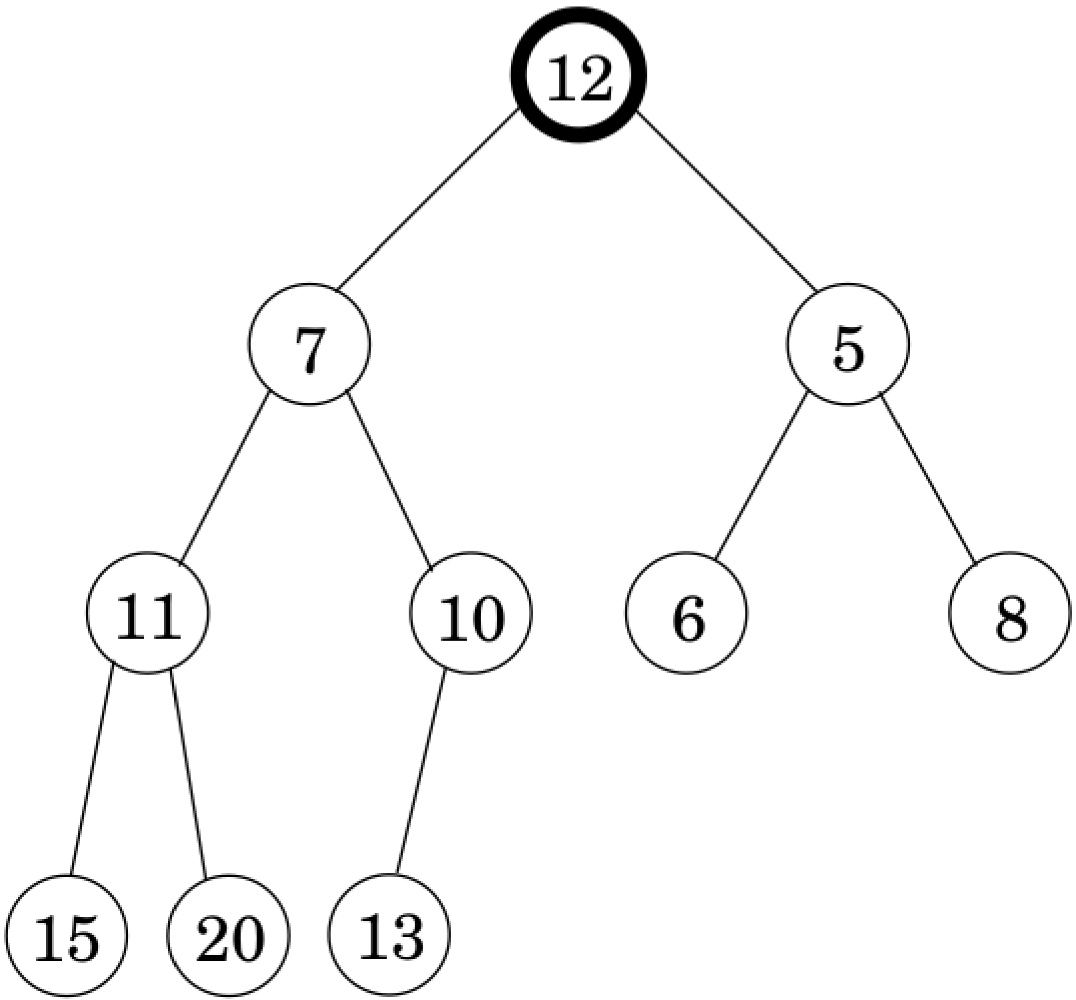
Bubble up:



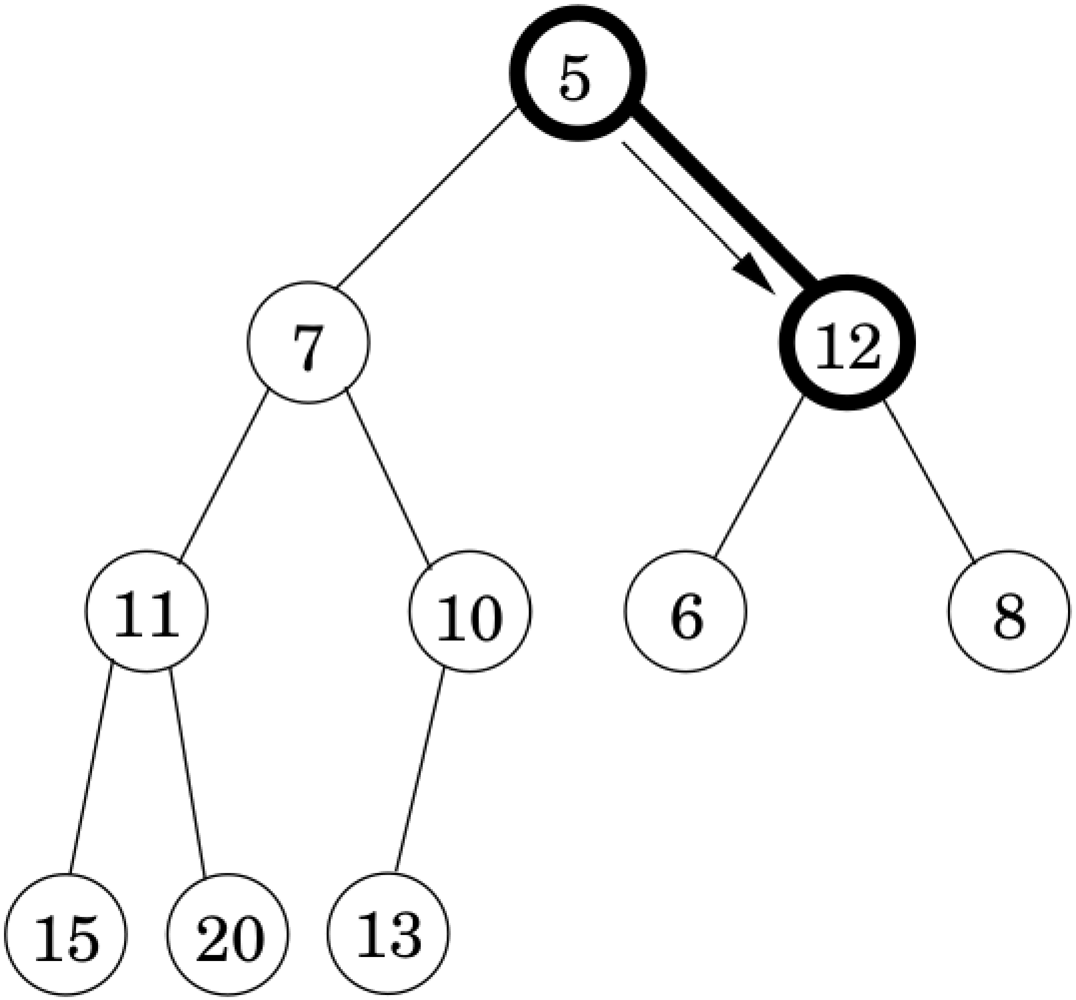
Now let's run delete-min:



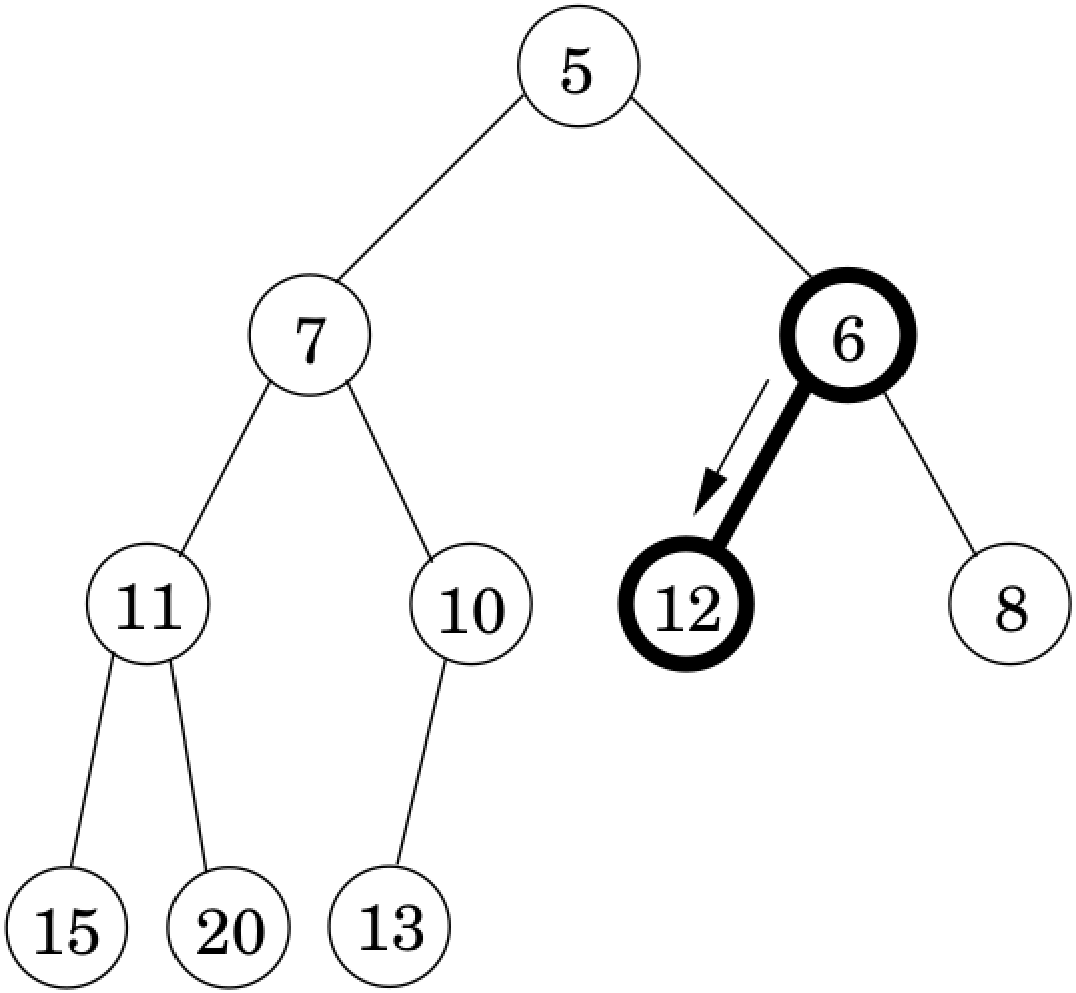
Move last node to root:



Sift down:



Sift down:



d-ARY HEAP

- Like binary heap, but with d children for each node
- Height reduces to

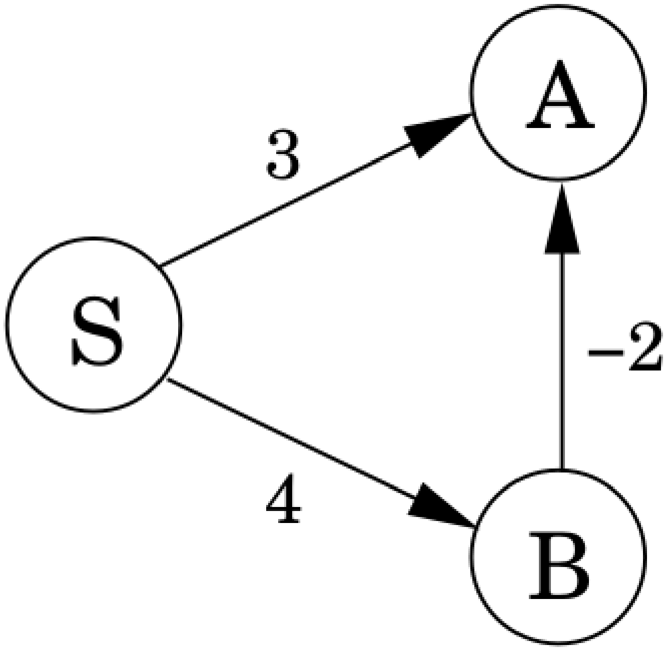
$$\Theta(\log_d n) = \Theta((\log n)/(\log d))$$

- Insert/decreasekey slightly faster
- Deletemin slightly slower

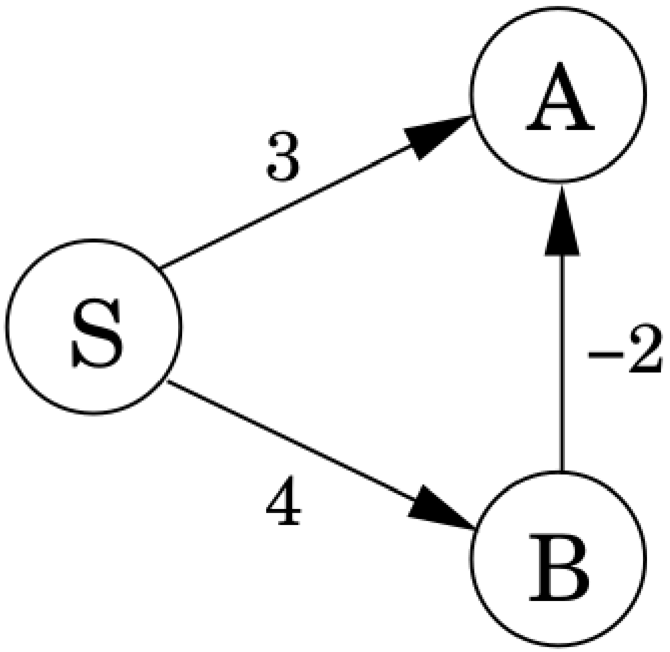
Which implementation is best?

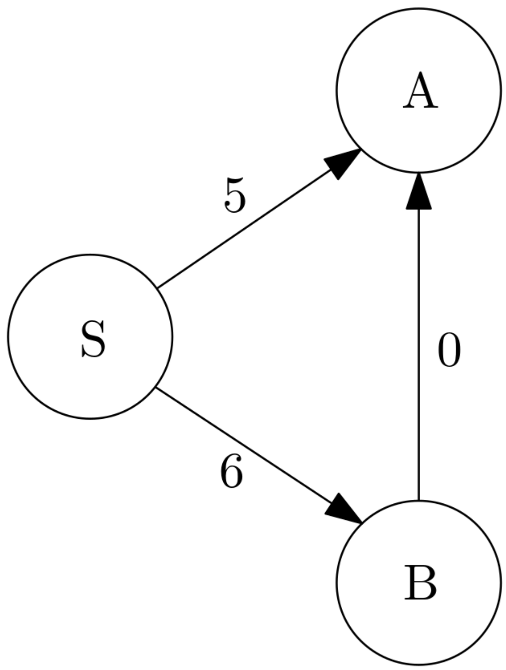
Implementation	`deletemin`	`insert`/ `decreasekey`	$\lvert V\rvert \times$ `deletemin` $+$ $(\lvert V\rvert + \lvert E\rvert) \times$ `insert`
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert + \lvert E\rvert) \log \lvert V\rvert)$
d**-ary heap**	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O((\lvert V\rvert \cdot d + \lvert E\rvert) \frac{\log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ **(amortized)**	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$

Suppose now that the graph has negative edges:



Can we just shift everything into the positive range?





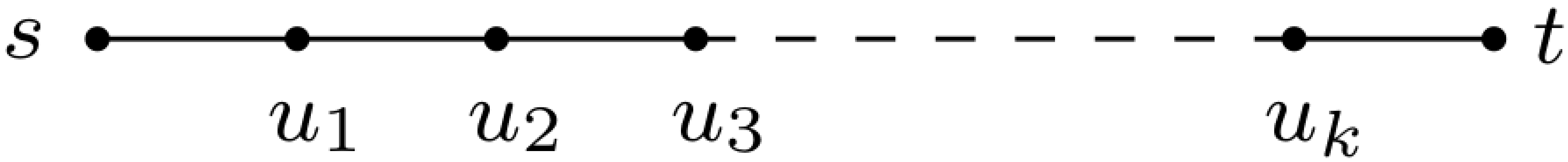
Note that the distances in Dijkstra's algorithm are always $\leq$ the true distance.

$$\underline{\texttt{procedure update}}((u,v) \in E)$$
$$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + l(u,v)\}$$

- If u is the second-last node in shortest path to v, gives exact distance
- Cannot set dist(v) smaller than the true distance

Dijkstra's algorithm can be seen as a sequence of these updates

Consider the shortest path from s to some node t:

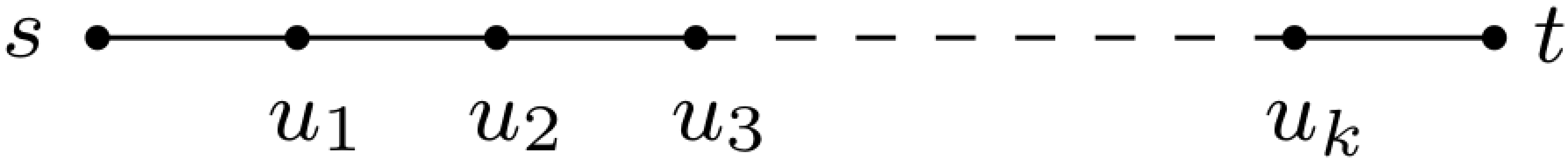


- This path has at most $|V| - 1$ edges
- If we did updates in this order:

$$(s, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_k, t)$$

we would get the correct distance to t.

Consider the shortest path from s to some node t:



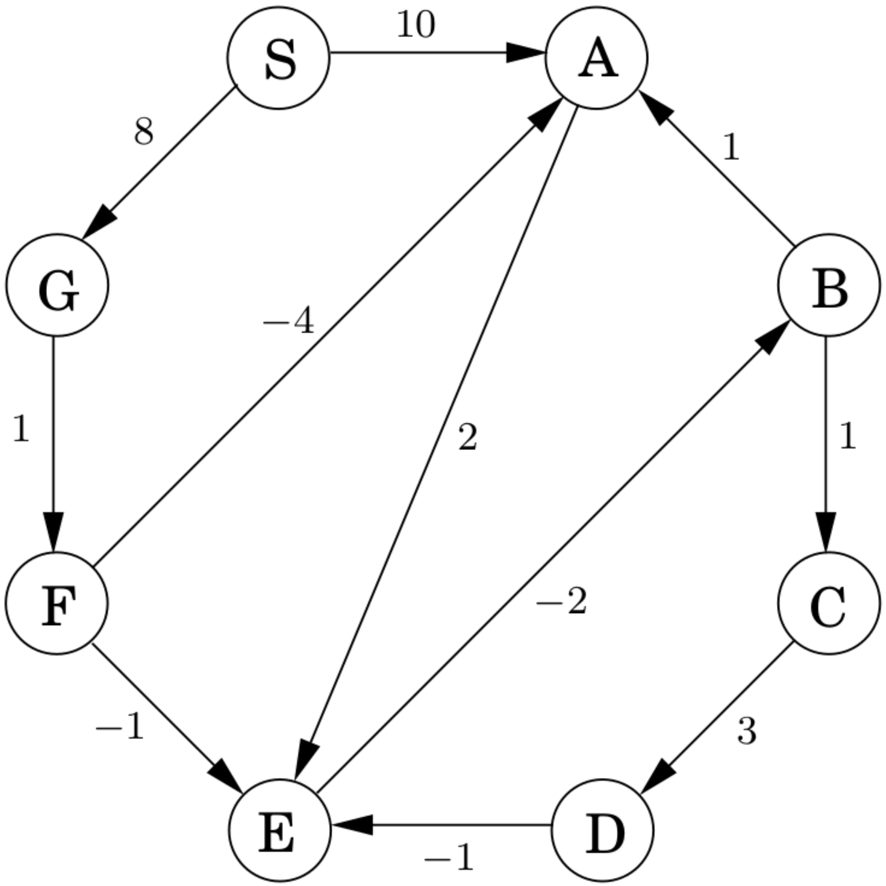
- How can we update the right edges in the right order, without already knowing the shortest paths?
- Just update all edges $|V| - 1$ times!

$\Rightarrow$ Bellman-Ford algorithm, $O(|V| \cdot |E|)$.

Bellman-Ford algorithm:

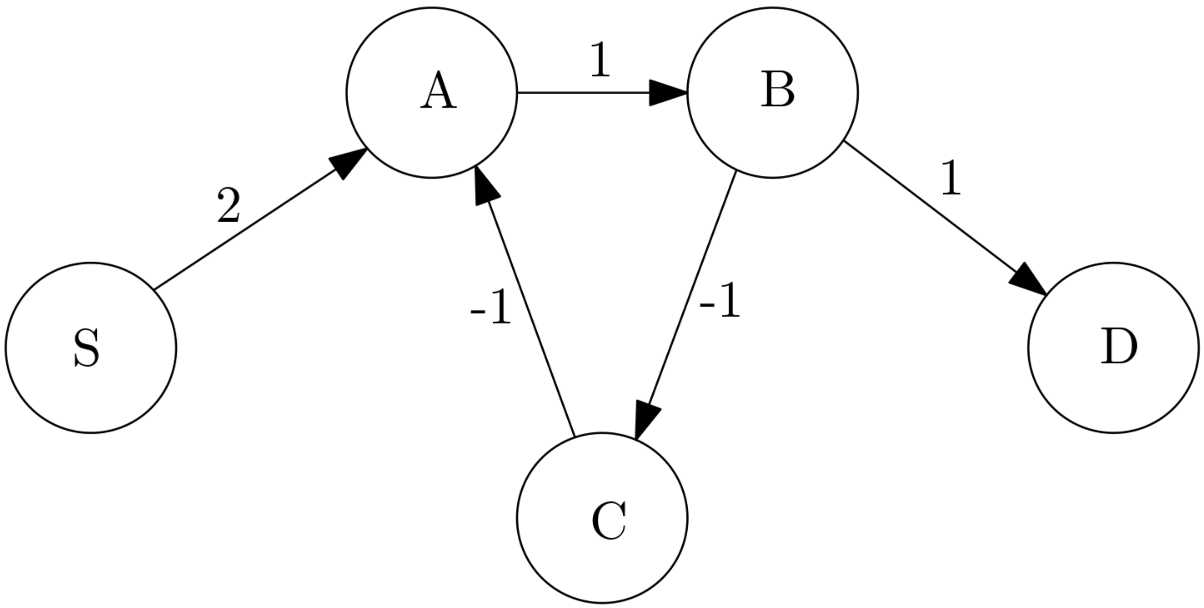
```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)
```

Bellman-Ford algorithm:



	Iteration							
Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

What if there is a negative cycle?



Two kinds of graphs that cannot have negative cycles:

- graphs with no negative edges
- dags

SHORTEST PATHS IN DAGS

Key idea:

- In any path, the vertices appear in increasing linearized order

SHORTEST PATHS IN DAGS

- Linearize the dag using DFS
- Visit the vertices in sorted order
- Update all outgoing edges

<u>procedure dag-shortest-paths</u>(G, l, s)

Input: Dag $G = (V, E)$;
edge lengths $\{l_e : e \in E\}$; vertex $s \in V$

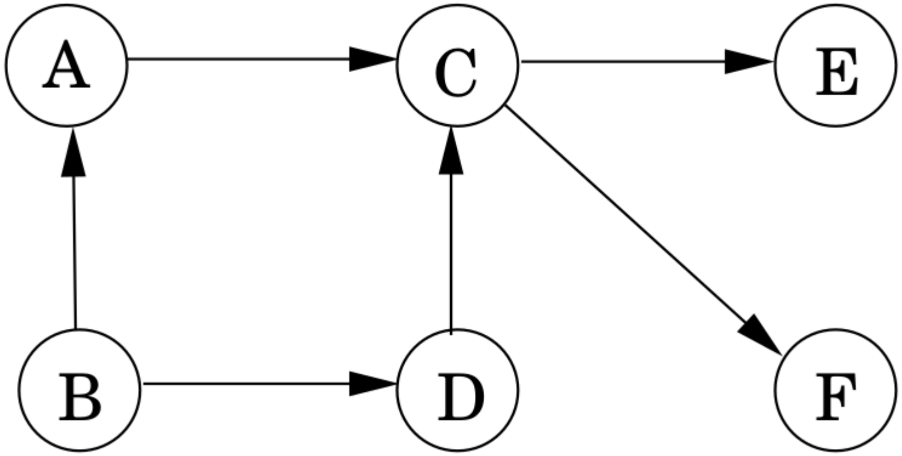
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u.

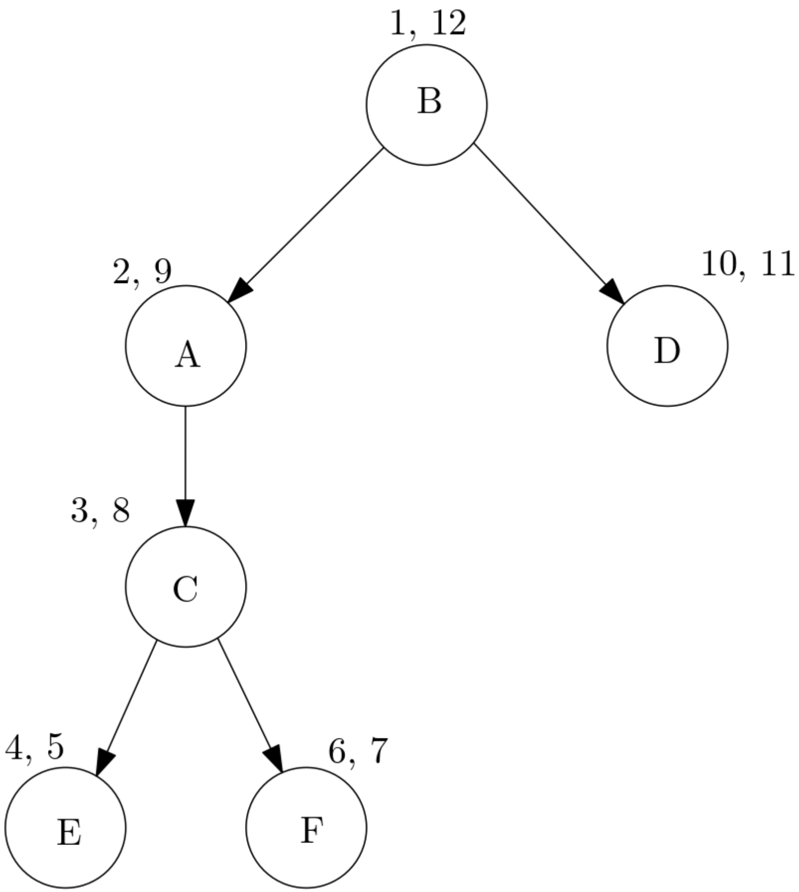
for all $u \in V$:
    dist$(u) = \infty$
    prev$(u) =$ nil

dist$(s) = 0$
Linearize G
for each $u \in V$, in linearized order:
    for all edges $(u, v) \in E$:
        update(u, v)

Recap: How can we linearize a dag algorithmically?

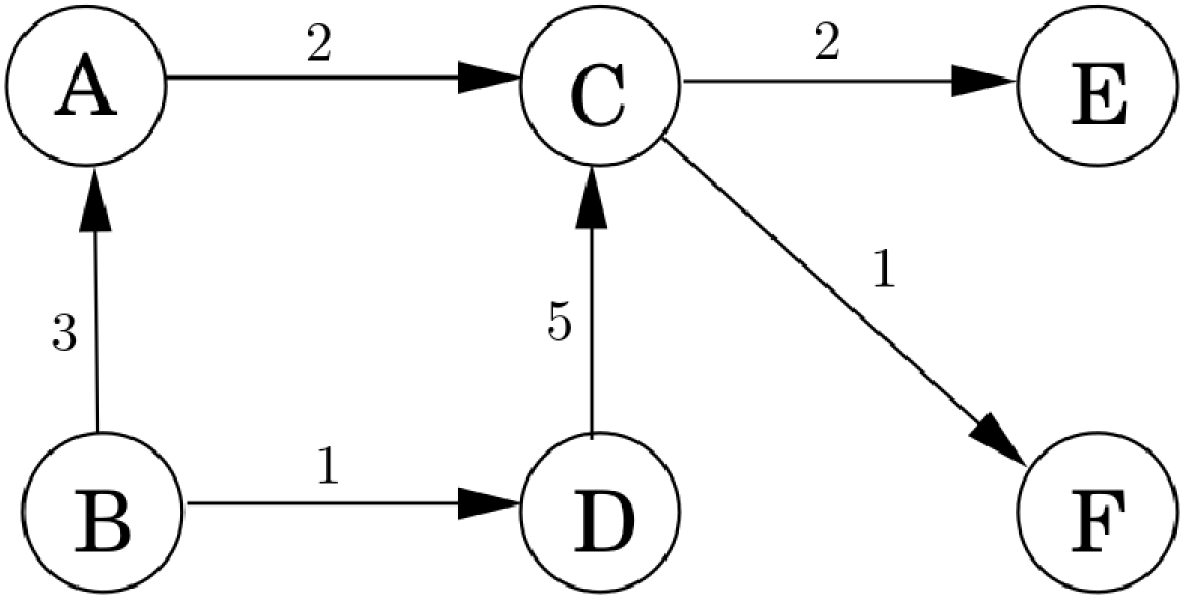
- List nodes in decreasing order of post numbers



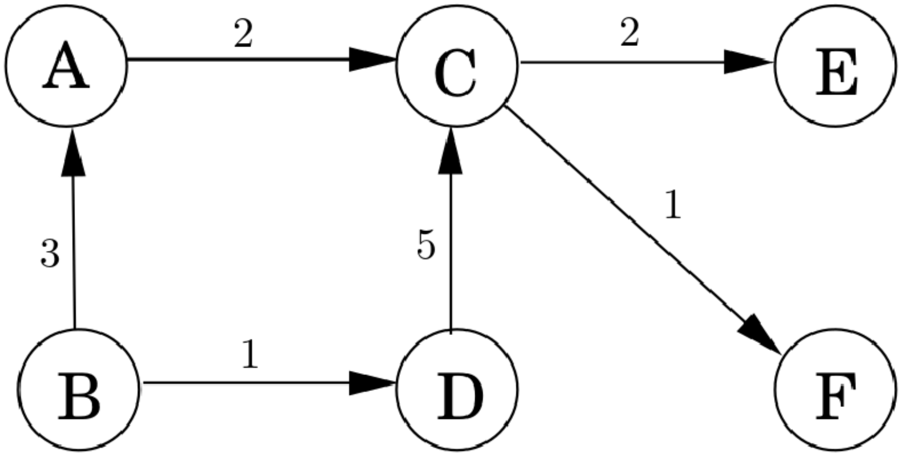


Order: B, D, A, C, F, E

Now let's add some lengths:

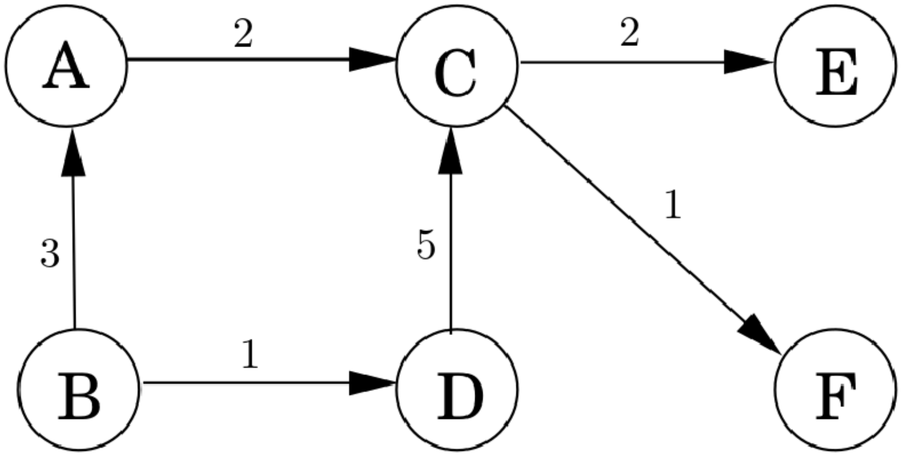


Order: B, D, A, C, F, E



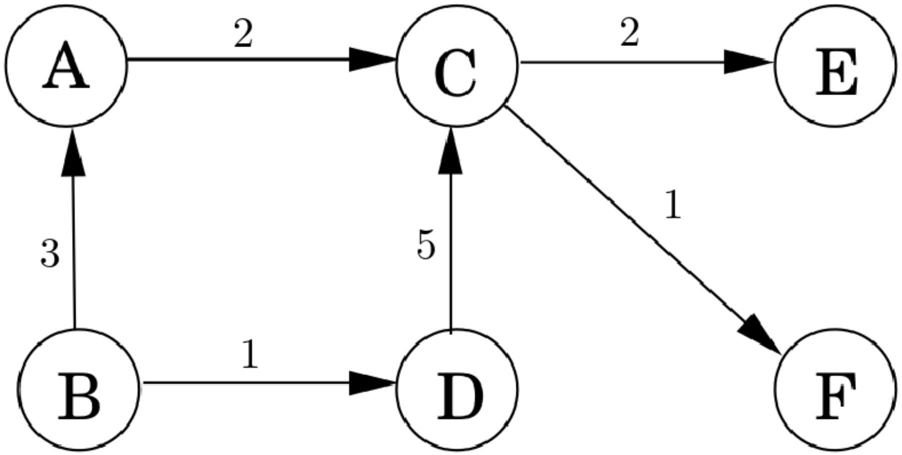
$$B, D, A, C, F, E$$

	0
A	∞
B	0
C	∞
D	∞
E	∞
F	∞



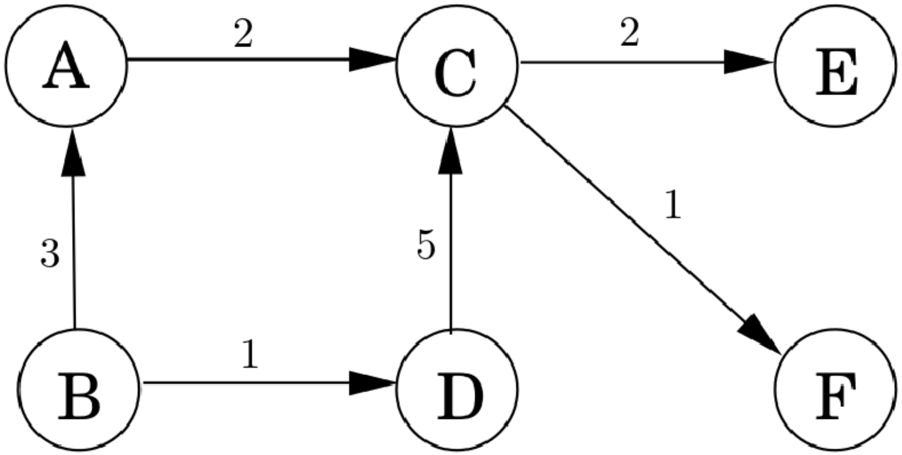
$$B, D, A, C, F, E$$

	0	1
A	∞	3
B	0	0
C	∞	∞
D	∞	1
E	∞	∞
F	∞	∞



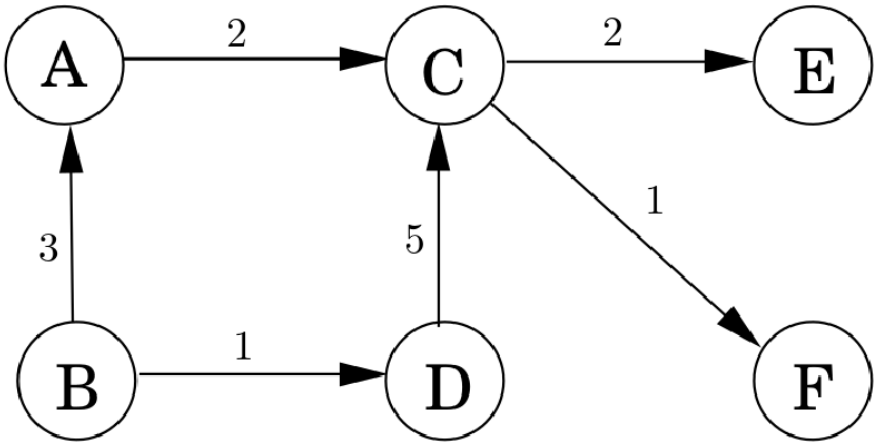
$$B, D, A, C, F, E$$

	0	1	2
A	∞	3	3
B	0	0	0
C	∞	∞	6
D	∞	1	1
E	∞	∞	∞
F	∞	∞	∞



$$B, D, A, C, F, E$$

	0	1	2	3
A	∞	3	3	3
B	0	0	0	0
C	∞	∞	6	5
D	∞	1	1	1
E	∞	∞	∞	∞
F	∞	∞	∞	∞



$$B, D, A, C, F, E$$

	0	1	2	3	4
A	∞	3	3	3	3
B	0	0	0	0	0
C	∞	∞	6	5	5
D	∞	1	1	1	1
E	∞	∞	∞	∞	7
F	∞	∞	∞	∞	6