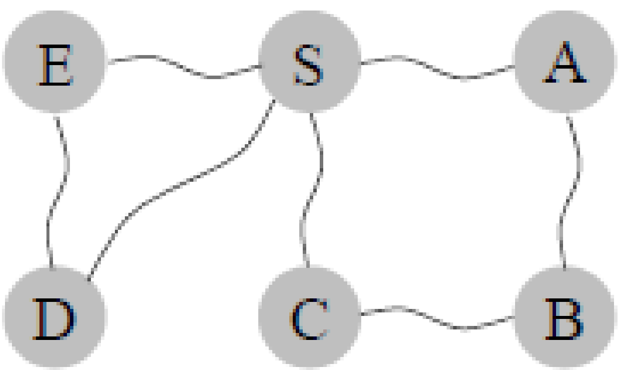
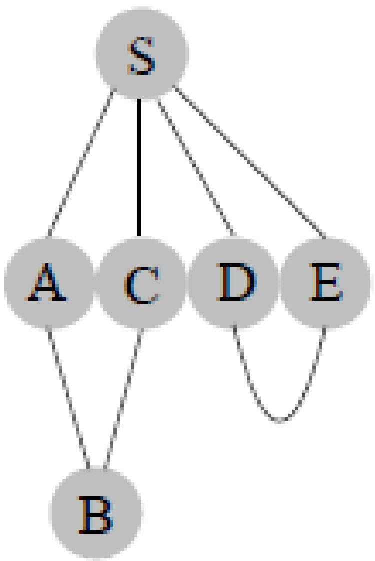
Paths in graphs

Distance in graphs

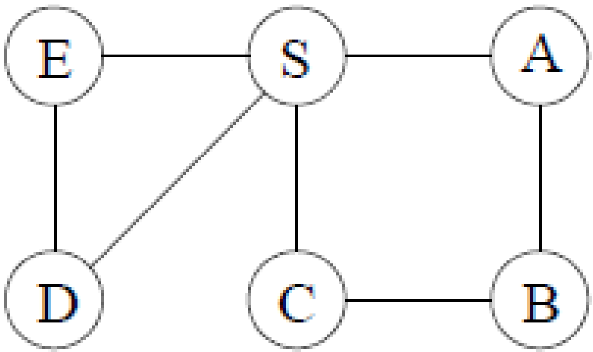
- The *distance* between two nodes is the length of the shortest path between them.
- How do we find the shortest paths from *s* to all other vertices?

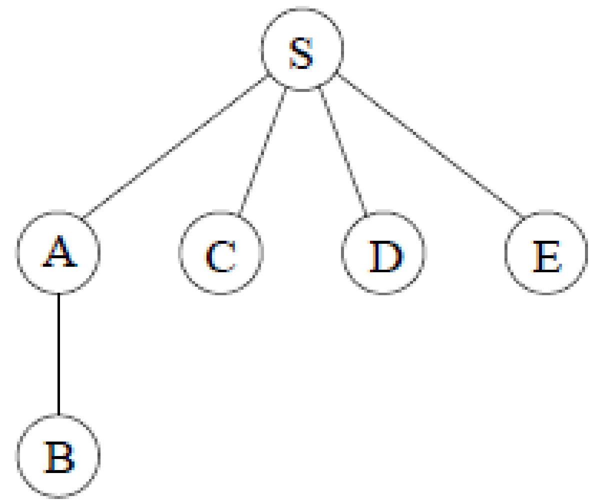




Breadth-first search

```
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
```





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	$[\,]$

Correctness

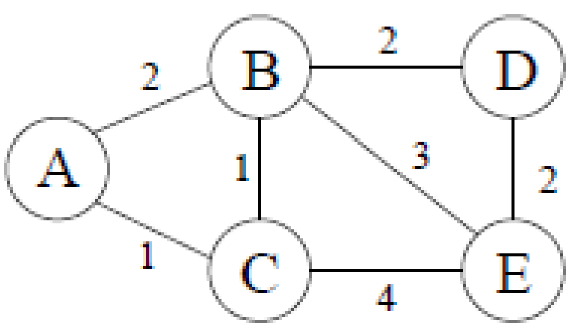
- Use induction
- For each $d = 0, 1, 2, \ldots$, there is a moment at which
  - (1) all nodes at distance $\leq d$ from s have their distances correctly set
  - (2) all other nodes have their distances set to ∞
  - (3) the queue contains exactly the nodes at distance d.

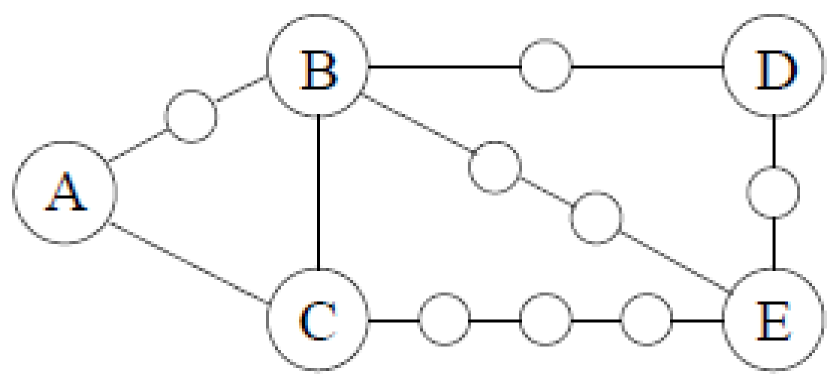
Analysis

- Each vertex is put on the queue exactly once $\rightarrow 2|V|$ queue operations
- **for** loop looks at each edge once (in directed graphs) or twice (in undirected graphs) $\rightarrow O(|E|)$ time
- $O(|V| + |E|)$

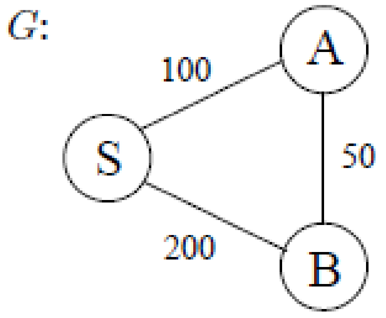
Weighted graphs

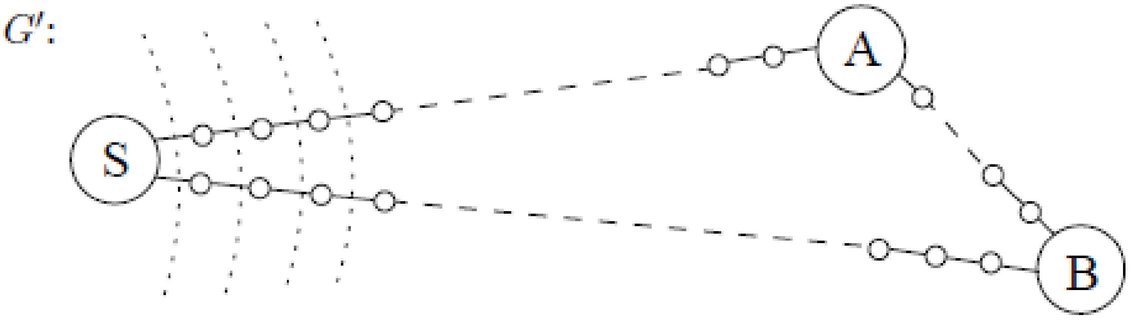
- Breadth-first search finds shortest paths in any graph whose edges have unit length.
- Can we adapt it to a more general graph $G = (V, E)$ whose edge lengths are *positive integers?*





- Set an alarm – estimated times of arrival
  - ex) initially, A : T=100, B : T=200
- At T=100, reset the alarm for B as T=150





Algorithm

- Set an alarm clock for node s at time 0.
- Repeat until there are no more alarms:

  Say the next alarm goes off at time T, for node u. Then:

  1) The distance from s to u is T.

  2) For each neighbor v of u in G:

     - If there is no alarm yet for v,
       set one for time $T + l(u, v)$.
     - If v's alarm is set for later than $T + l(u, v)$,
       then reset it to this earlier time.

Priority queue

Data structure supporting the following operations :

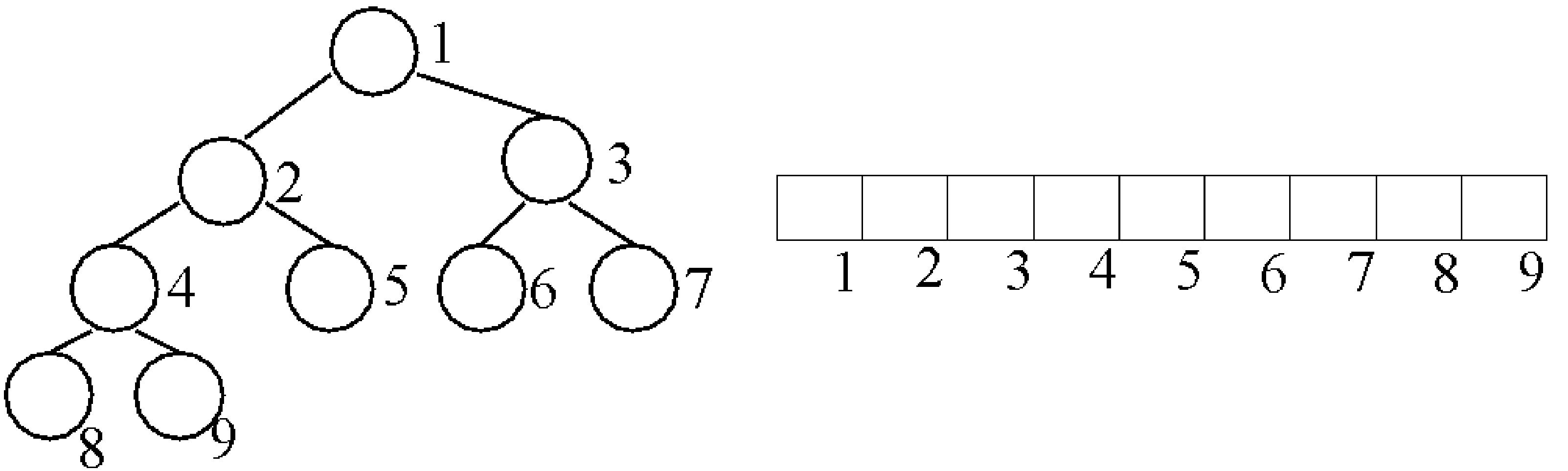
- *Insert.* Add a new element to the set.
- *Decrease-key.* Accommodate the decrease in key value of a particular element
- *Delete-min.* Return the element with the smallest key, and remove it from the set.
- *Make-queue.* Build a priority queue out of the given elements, with the given key values. (In many implementations, this is faster than inserting the elements one by one.)

Binary heap

- Complete Binary Tree - All levels are completely filled except possibly the lowest, which is filled from the left up to a point.
- The value of each node $\leq$ value of its children. (min-heap)

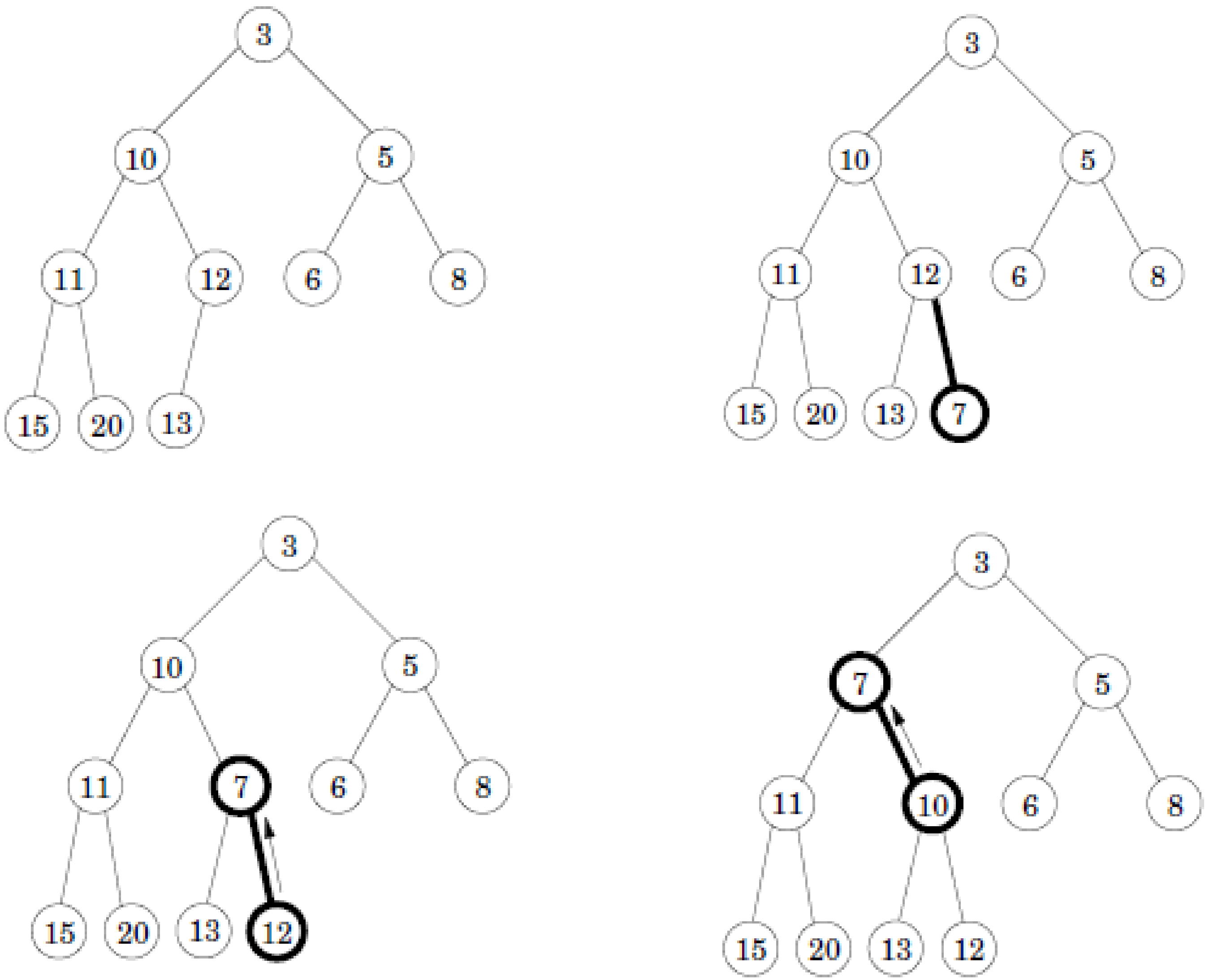
Heap data structure

Complete binary tree implemented by an array

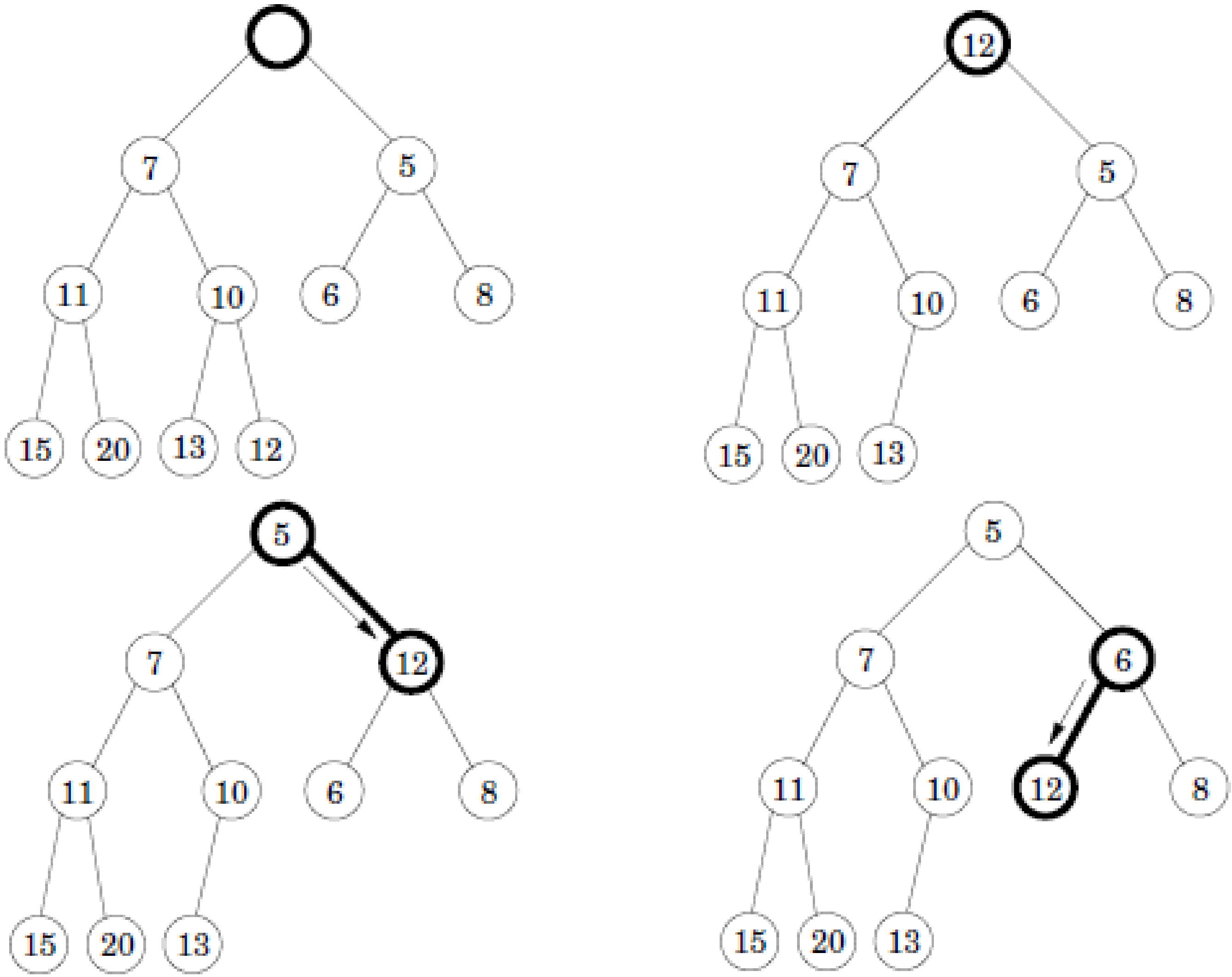


- The root is stored in $A[1]$
- The parent of $A[i]$ is $A[\lfloor \frac{i}{2} \rfloor]$.
- The left child of $A[i]$ is $A[2 \cdot i]$.
- The right child of $A[i]$ is $A[2 \cdot i + 1]$.
- The node in the far right of the bottom level is stored in $A[n]$.
- If $2i + 1 > n$, then the node does not have a right child.

Insert

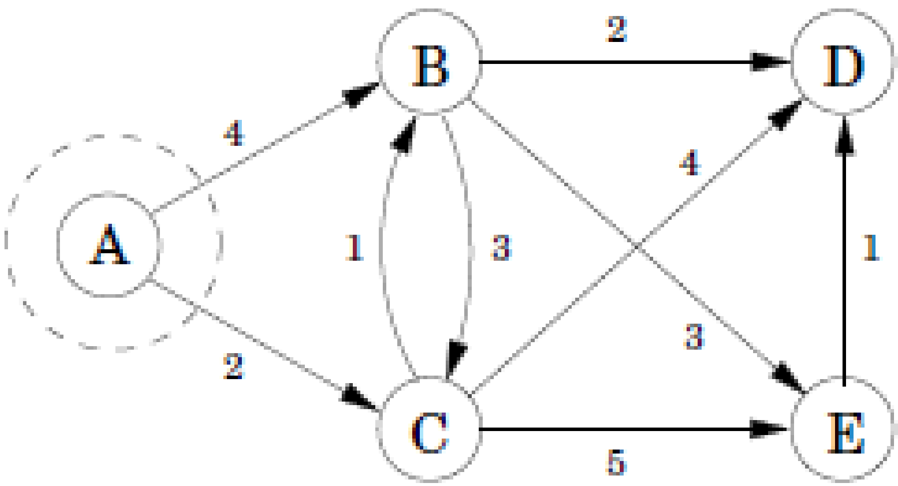

3
10
5
11
12
6
8
15
20
13
3
10
5
11
12
6
8
15
20
13
7
3
10
5
11
7
6
8
15
20
13
12
3
7
5
11
10
6
8
15
20
13
12

Delete-min

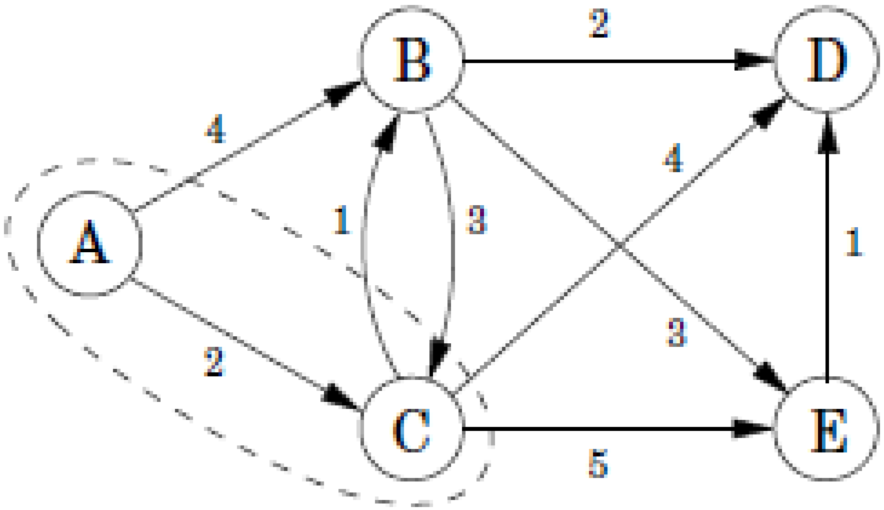

7
5
11
10
6
8
15
20
13
12
12
7
5
11
10
6
8
15
20
13
5
7
12
11
10
6
8
15
20
13
5
7
6
11
10
12
8
15
20
13

Dijkstra's algorithm

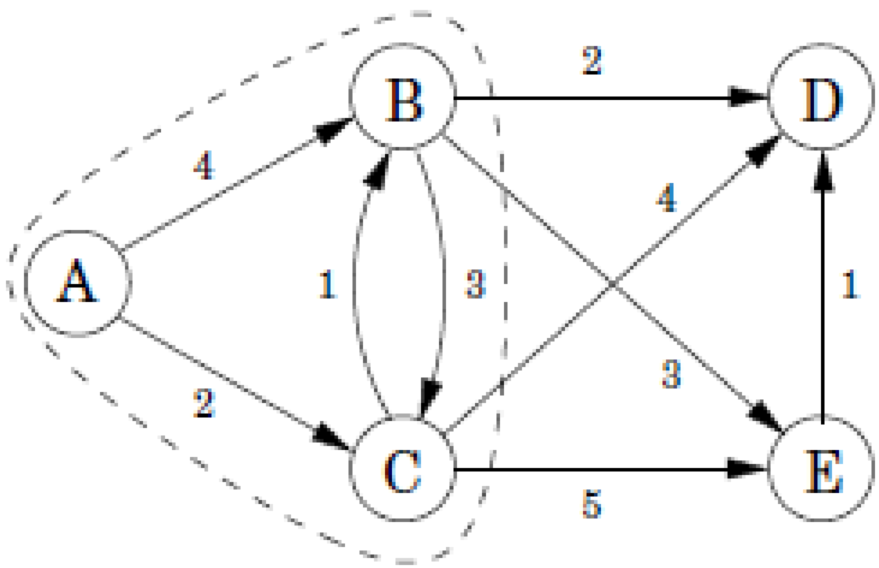
```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

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)
```



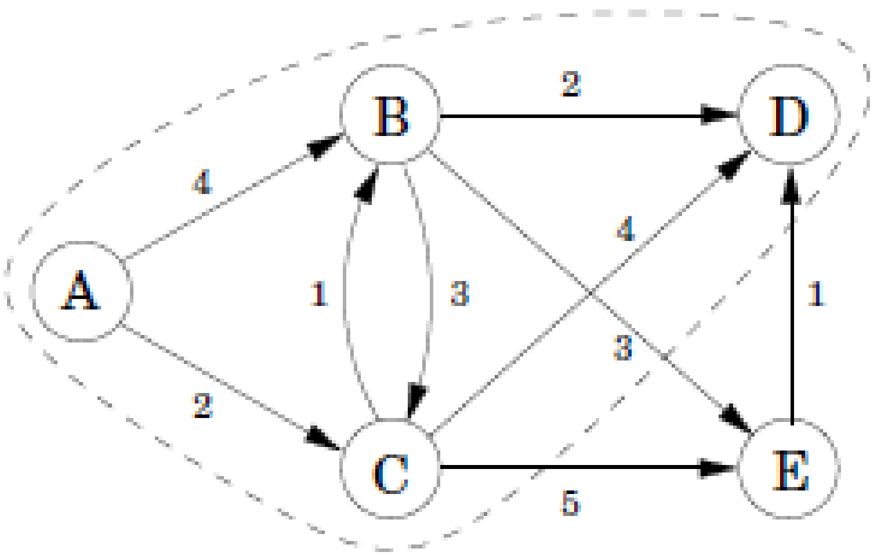
A: 0 B: 4 C: 2	D: ∞ E: ∞



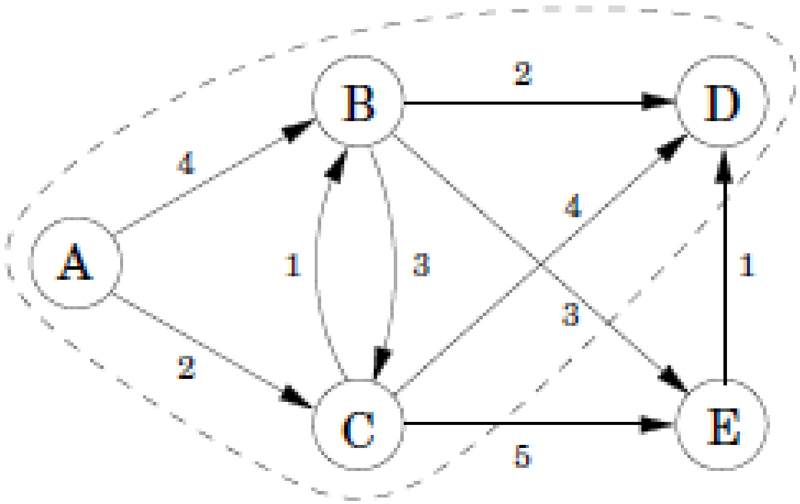
A: 0	D: 6
B: 3	E: 7
C: 2	



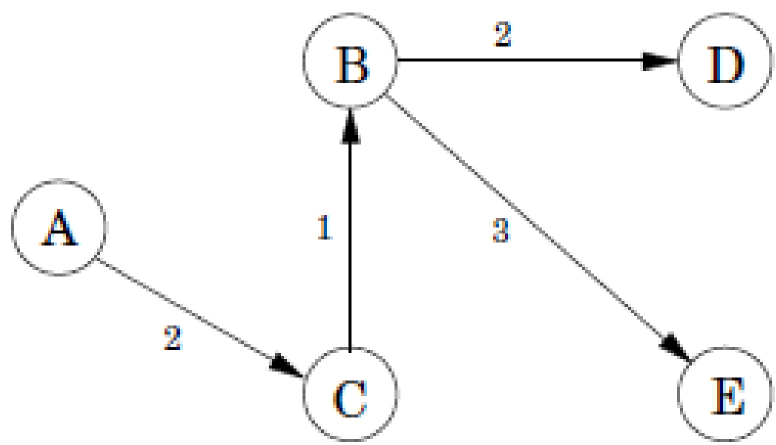
A: 0	D: 5
B: 3	E: 6
C: 2	



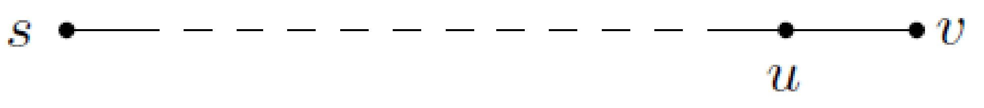
A: 0	D: 5
B: 3	E: 6
C: 2	



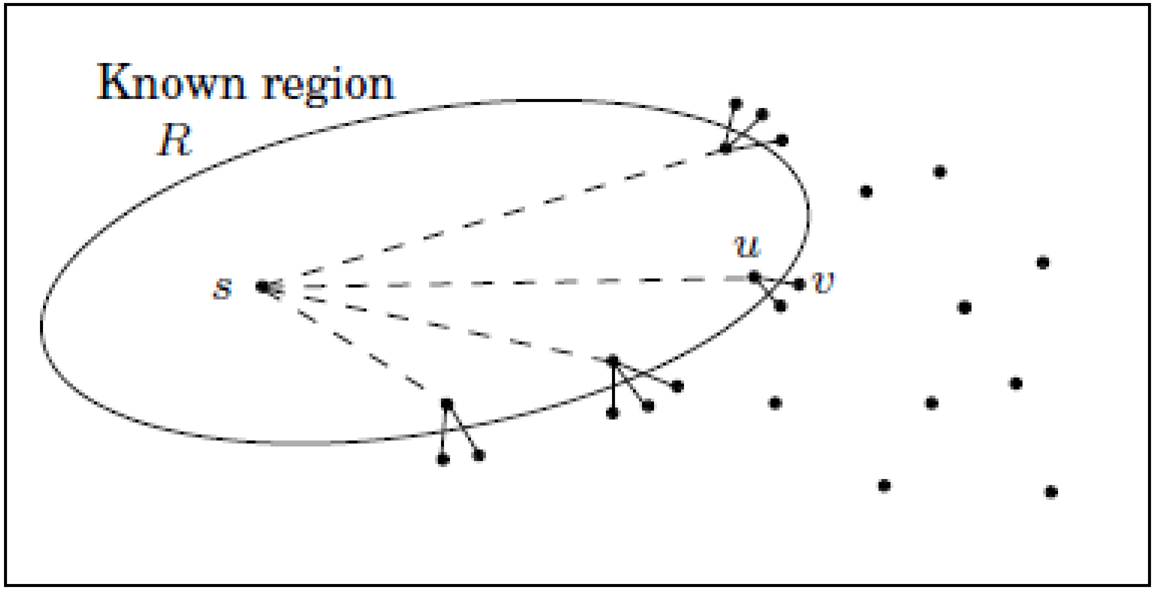
A: 0	D: 5
B: 3	E: 6
C: 2	



- Starting from *s*, we expand the region *R* of the graph where shortest paths are known.
- What is the next vertex *v* to add to *R*?
  - the node outside *R* that is *closest* to *s*
- Consider the shortest path from *s* to *v*. Let *u* be the node before *v* on this path.



- Since all edge lengths are positive, *u* must be closer to *s* than *v* is.
- Thus, *u* is in *R*. (Since *v* is the closest node to *s* outside *R*.)
- So, the shortest path from *s* to *v* is a *known shortest path extended by a single edge*.
- *v* is the node outside *R* for which the smallest value of distance(*s*,*u*)+*l*(*u*,*v*) is attained, as *u* ranges over *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)
```

Correctness

- Use induction.
- At the end of each iteration of the while loop, the following conditions hold:
  - (1) there is a value d such that all nodes in R are at distance $\leq d$ from s and all nodes outside R are at distance $\geq d$ from s
  - (2) for every node u, the value dist(u) is the length of the shortest path from s to u whose intermediate nodes are constrained to be in R (if no such path exists, the value is ∞).

```
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)
```

Analysis

- $|V|$ deletemin operations
- $|V| + |E|$ insert/decreasekey operations

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

Update

- We can consider Dijkstra's algorithm as performing a sequence of the following update procedure.

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

- This *update operation* uses the fact that the distance to *v* cannot be more than the distance to *u* + *l*(*u*, *v*).
- **property 1** It gives the correct distance to *v* in the particular case where *u* is the second-last node in the shortest path to *v*, and dist(*u*) is correctly set.
- **property 2** It will never make dist(*v*) too small, and in this sense it is *safe*. For instance, extra update's can't hurt.

Property 2 : Initializing dist(s) = 0 and dist(v) = ∞ for all $v \in V - \{s\}$ establishes dist(v) $\geq$ distance to v for all $v \in V$.
This invariant is maintained over any sequence of **update**'s.

Pf) Suppose not.

Let v be the first vertex for which dist(v) < distance to v, and

let u be the vertex that caused dist(v) to change : dist(v) = dist(u) + $l(u, v)$.

Then, dist(v)	$<$	distance to v	(supposition)
	$\leq$	distance to u + distance from u to v	(triangle inequality)
	$\leq$	distance to u + $l(u, v)$	(shortest path $\leq$ specific path)
	$\leq$	dist(u) + $l(u, v)$	(v is first violation)

Contradiction.

Property 1: Let u be v's predecessor on a shortest path from s to v. Then, if dist(u) = distance to u, after **update**(u, v), dist(v) = distance to v.

Pf) distance to v = distance to u + $l(u, v)$.

Suppose dist(v) > distance to v before **update**.

(Otherwise, by **property 2**, dist(v) = distance to v.)

Then, dist(v) > dist(u) + $l(u, v)$,

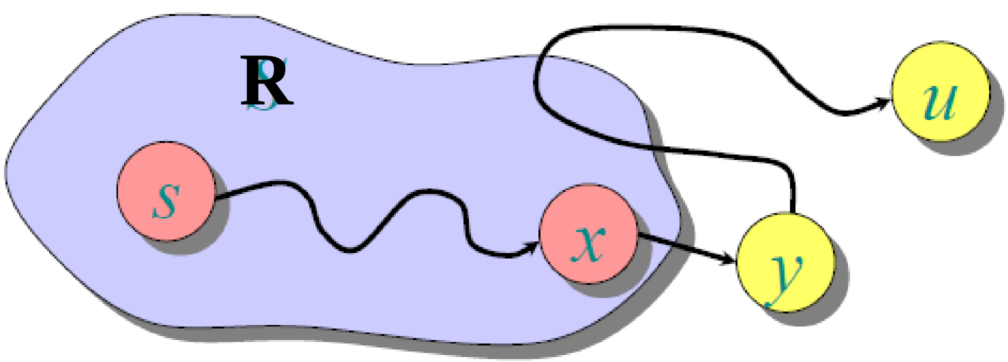
because dist(v) > distance to v = distance to u + $l(u, v)$ = dist(u) + $l(u, v)$.

By **update**(u, v), dist(v) = dist(u) + $l(u, v)$ = distance to v.

Theorem : Dijkstra's algorithm terminates with dist(v) = distance to v for all $v \in V$.

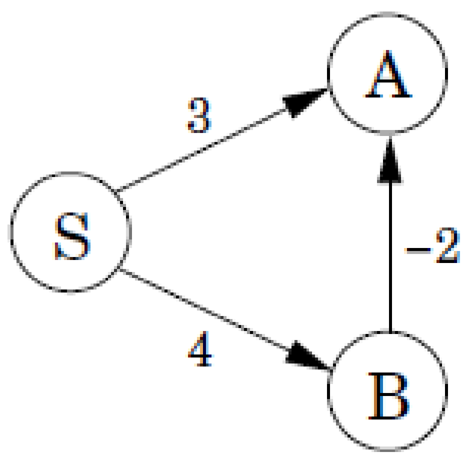
Proof.

- It suffices to show that dist(v) = distance to v for every $v \in V$ when v is added to R.
- Suppose u is the first vertex added to R for which dist(u) > distance to u.
- Let y be the first vertex in $V - R$ along a shortest path from s to u, and let x be its predecessor.
- Since u is the first vertex violating the claimed invariant, we have dist(x) = distance to x.
- When x was added to R, we update the edge (x, y), which implies that dist(y)= distance to $y \leq$ distance to u < dist(u).
- But, dist(u) $\leq$ dist(y) by our choice of u. Contradiction.



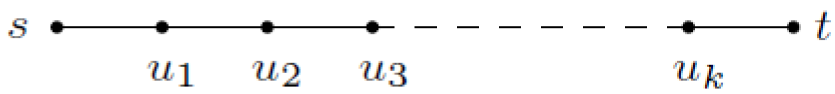
Negative edges

- Dijkstra's algorithm works in part because the shortest path from the starting point *s* to any node *v* must pass exclusively through nodes that are closer than *v*.
- This no longer holds when edge lengths can be negative.



Negative edges

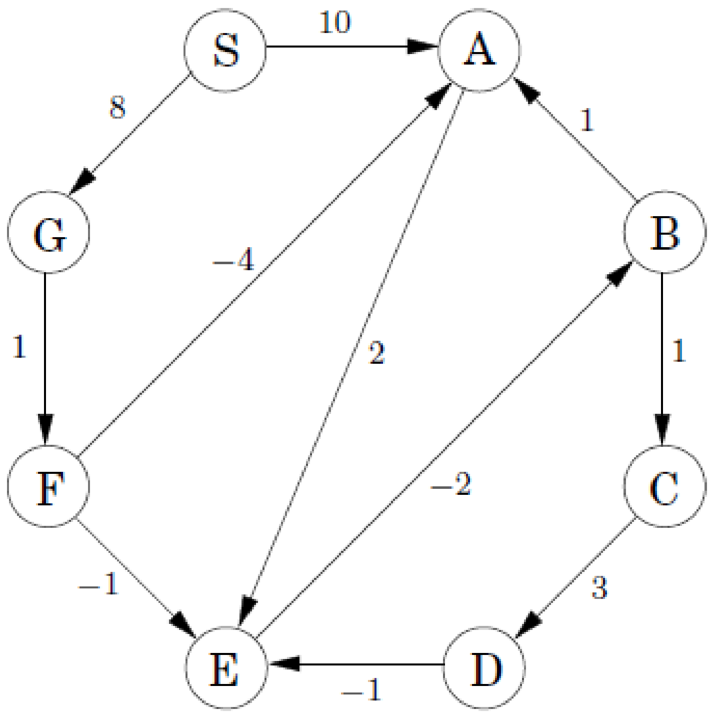
- What is a correct sequence of updates with negative edges?
- Consider a shortest path from *s* to *t* :



- This path can have at most |*V*|-1 edges.
- If the sequence of updates performed includes (s,u_1), (u_1, u_2), (u_2, u_3) … (u_k, t), in that order *(though not necessarily consecutively)*,then by **property 1** the distance to *t* will be correctly computed.
- Simply update *all the edges* |*V*|- 1 times!

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)
```

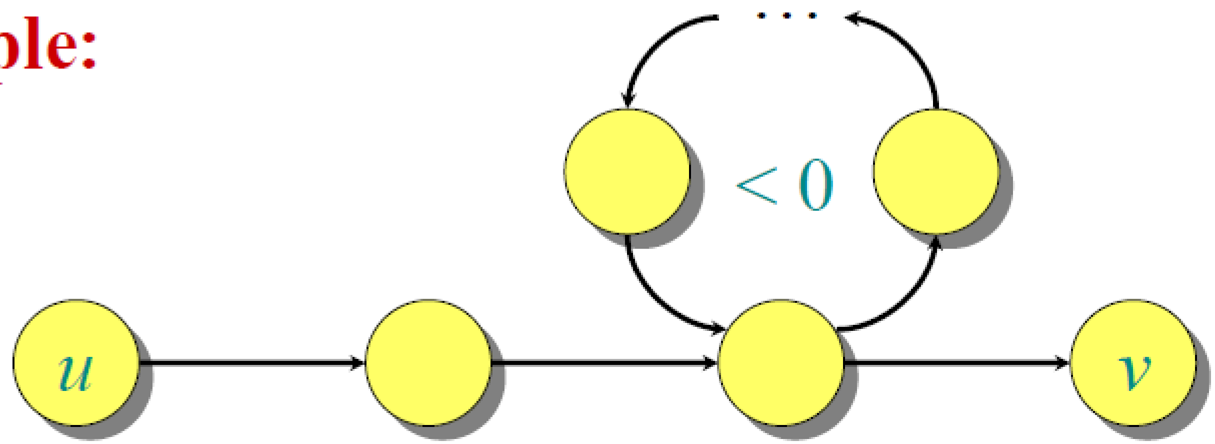


	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Negative cycles

- If a graph contains a negative-weight cycle, some shortest paths may not exist.
- Instead of stopping after $|V|$- 1 iterations, perform one extra round.
- There is a negative cycle if and only if some dist value is reduced during this final round.

Example:



Shortest paths in dags

- We need to perform a sequence of updates that includes every shortest path as a subsequence.
- In any path of a dag, the vertices appear in increasing linearized order.

```
procedure dag-shortest-paths(G, l, s)
Input:    Dag G = (V, E);
          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
Linearize G
for each u ∈ V, in linearized order:
   for all edges (u, v) ∈ E:
      update(u, v)
```