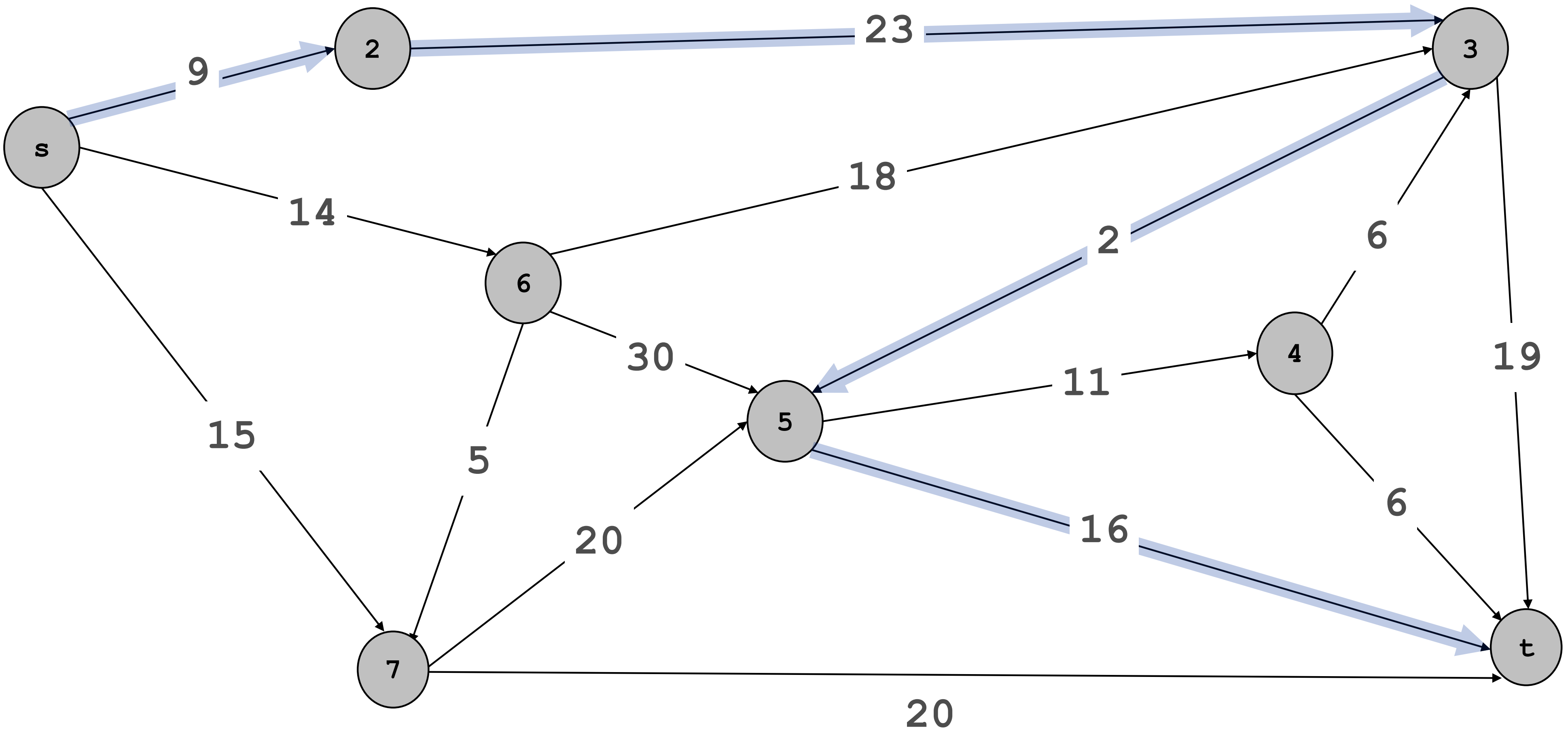
Graph Optimization

a. Dijkstra’s Algorithm

Weighted Graphs



Weighted Graphs

- **Definition:** A weighted graph $G = (V, E, \{w(e)\})$
  - V is the set of vertices
  - $E \subseteq V \times V$ is the set of edges
  - $w(e) \in \mathbb{R}$ are edge weights
  - Can be directed or undirected

- **Today:**
  - Directed graphs (one-way streets)
  - Non-negative edge weights ($w(e) \geq 0$)

Shortest Paths

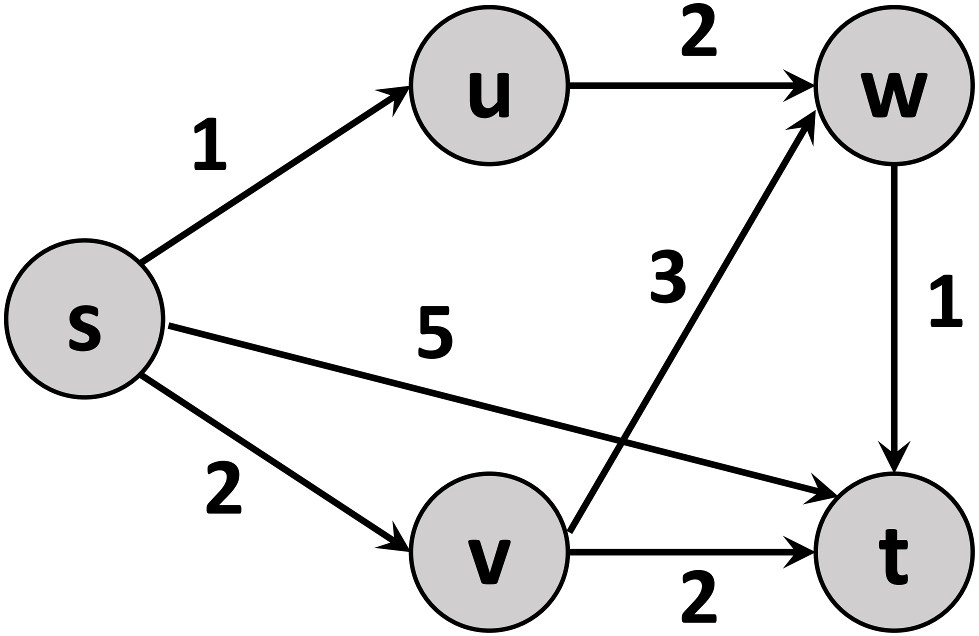
- In weighted graphs, the length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of its edge weights:

$$w((v_1,v_2)) + w((v_2,v_3)) + \cdots + w((v_{k-1},v_k)).$$

- The distance $d(s,t)$ is the length of the shortest path from s to t
- **Shortest Path:** given nodes $s,t \in V$, find the shortest path from s to t
- ***Single-Source Shortest Paths:*** *given a node $s \in V$, find the shortest paths from s to **every** $t \in V$*
- **All-Pairs Shortest Paths:** find the shortest path between every $(s,t) \in V$

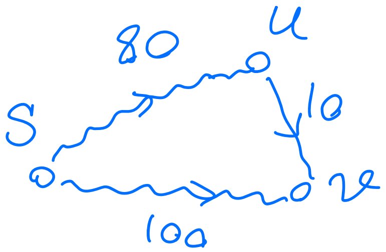
Distance

- In weighted graphs, the length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge weights:
- The distance $d(s,t)$ is the length of the shortest path from s to t

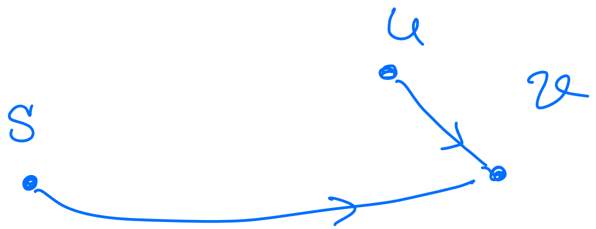


$d(s,t)$ = 4

Structure of Shortest Paths



- If $(u, v) \in E$, then $d(s, v) \leq d(s, u) + w(u, v)$ for every node $s \in V$



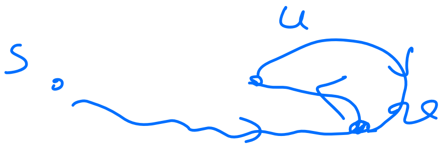
Suppose that $d(s,v) > d(s,u) + w(u,v)$.

But then the path $\underbrace{s \rightsquigarrow u}_{\text{shortest path from } s \text{ to } u}, v$ should be shorter than $s \rightsquigarrow v$, contradiction.

- If $(u, v) \in E$, and $d(s, v) = d(s, u) + w(u, v)$ then there is a shortest $s \rightsquigarrow v$-path ending with (u, v)

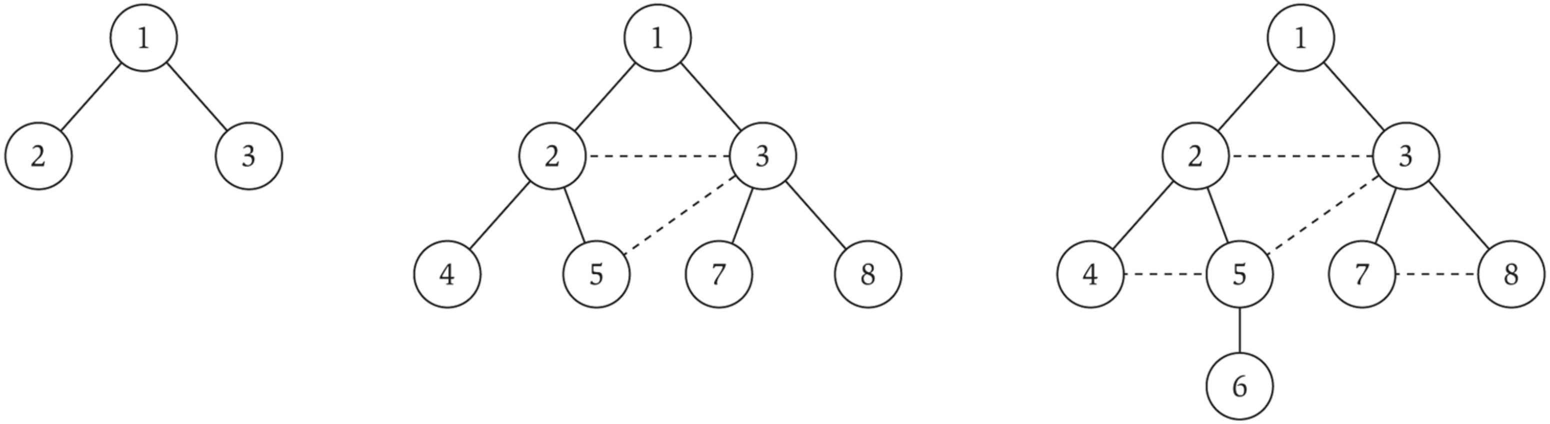
Take the shortest path $s \rightsquigarrow u$, and add the edge (u,v) at the end.

This is a path from s to v of length $d(s,v)$, which is the shortest.



Compare to BFS

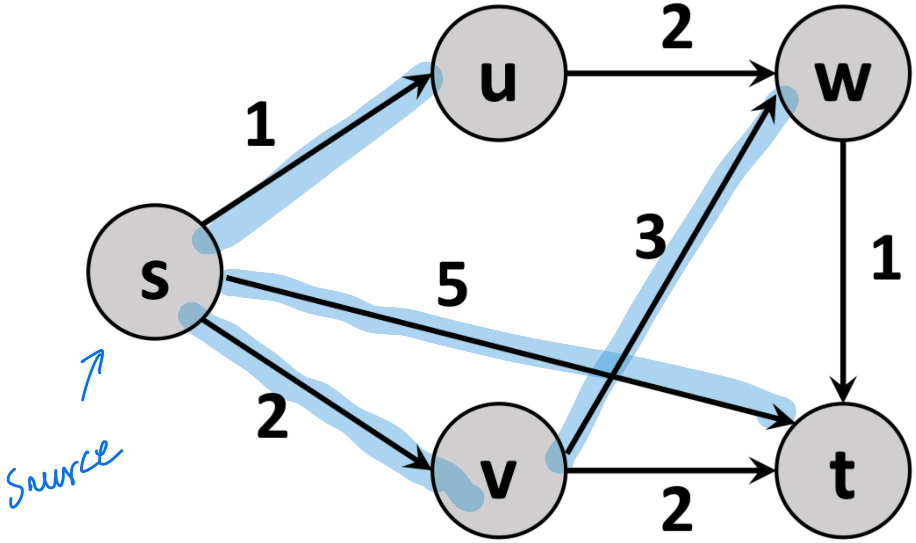
- **Thm.:** BFS finds distances from s to other nodes in unweighted graphs
  - L_i contains all nodes at distance i from s
  - Nodes not in any layer are not reachable from s



- **Question:** Does running a BFS from *s* **always** solve the SSSP problem on weighted graphs?

Compare to BFS

Question: Does running a BFS from *s* **always** solve the SSSP problem on weighted graphs?



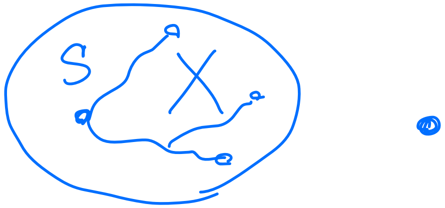
Dijkstra’s Algorithm

- **Dijkstra’s Shortest Path Algorithm** is a modification of BFS for non-negatively weighted graphs

“One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I [...] designed the algorithm for the shortest path [...] It was a twenty-minute invention. [...] I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame.” - Dijkstra

Edsger W. Dijkstra
1930-2002

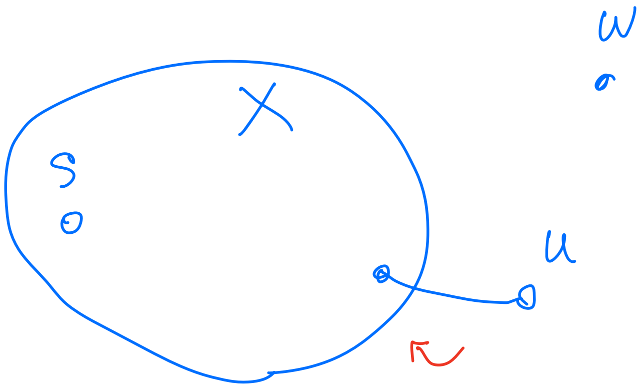
Dijkstra's Algorithm



- **Dijkstra's Shortest Path Algorithm** is a modification of BFS for non-negatively weighted graphs
- **Informal Version:**
  - **Maintain a set X of explored nodes**
  - **Maintain an upper bound on distance for all unexplored nodes**
    - If u is explored, then we know $d(s,u)$ (from the source *s*) (Key Invariant)
    - If u is explored, and (u,v) is an edge, then we know $d(s,v) \leq \ d(s,u) + w(u,v)$
  - **Explore the "closest" unexplored node**
  - **Repeat until we're done**

Dijkstra's Algorithm

- **Explore the "closest" unexplored node**
  - The unexplored node with the smallest upper bound on its distance
  - Tighten (lower) its out-neighbors' upper bounds (when possible)



when we add the closest vertex u to X,

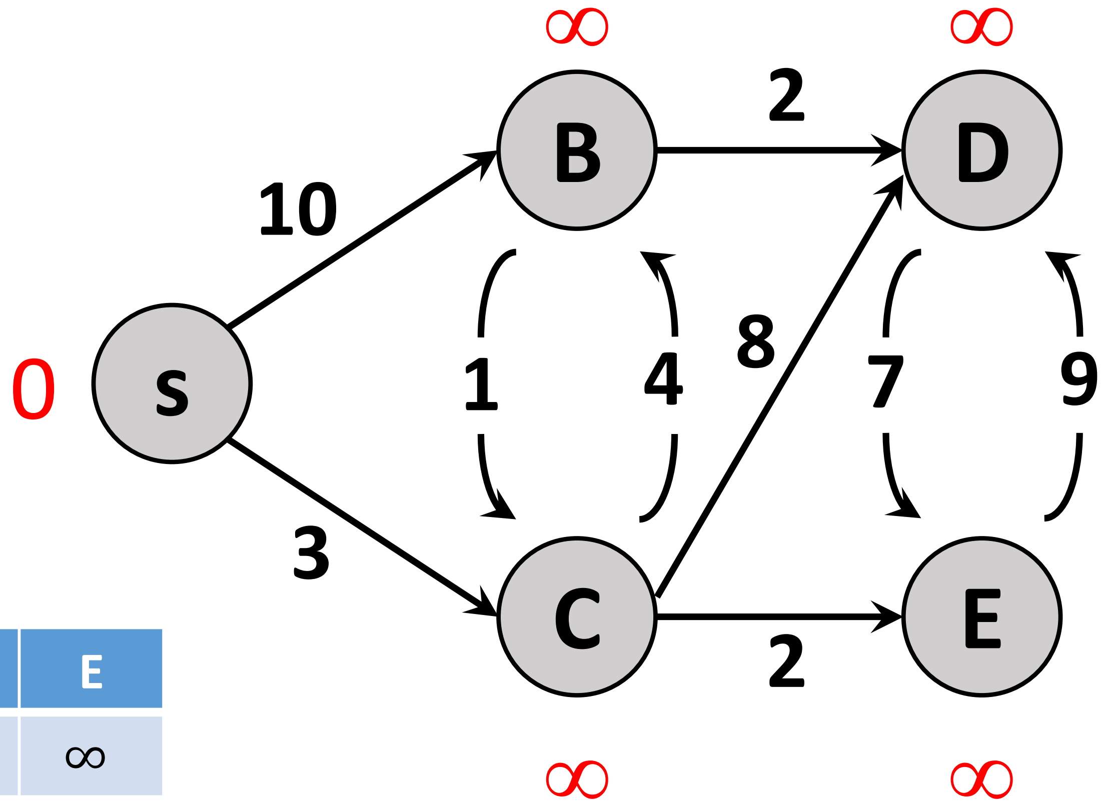
for every other unexplored node w, we tighten the upperbound to:

such that (u,w) is an edge

$d(s,w) \leftarrow \min\{d(s,w),\ d(s,u) + w(u,w)\}$

Dijkstra’s Algorithm: Demo

Initialize



	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞

$X = \{\ s\ \}$

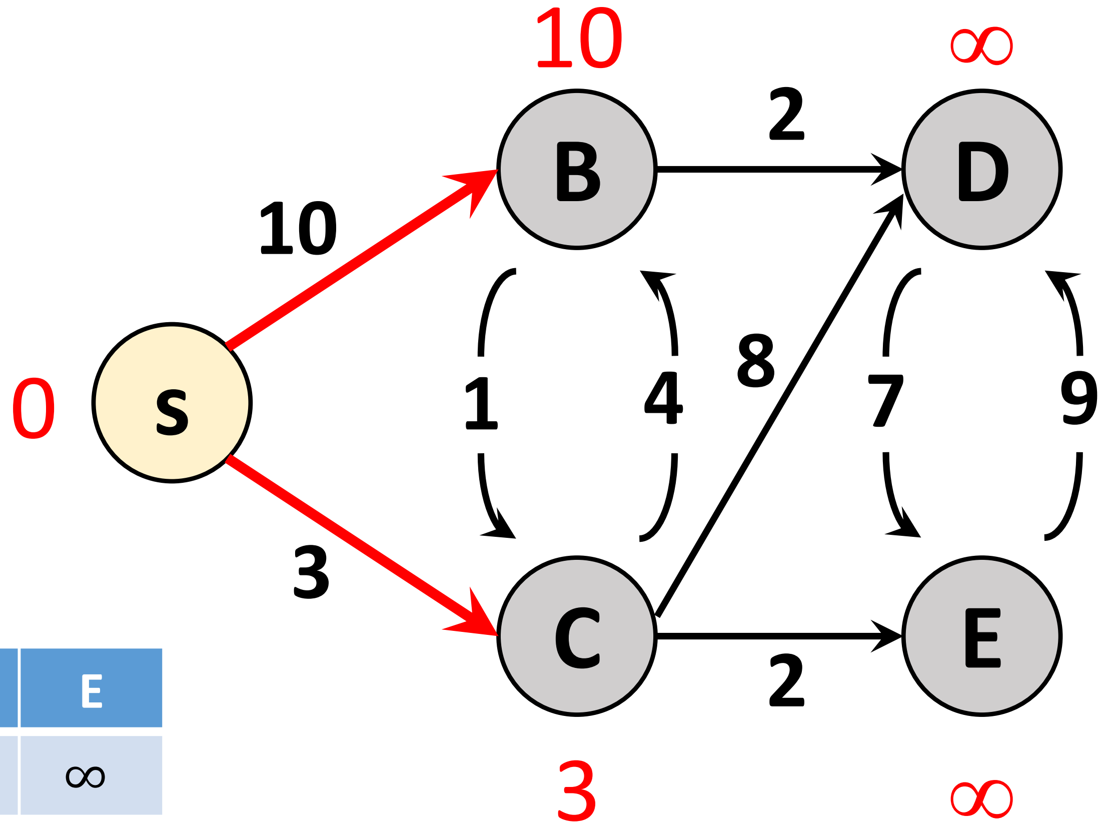
Dijkstra’s Algorithm

- **Explore the “closest” unexplored node**
  - The unexplored node with the smallest upper bound on its distance
  - Tighten its out-neighbors’ upper bounds (if we can)

	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

Dijkstra’s Algorithm: Demo

Explore *s*

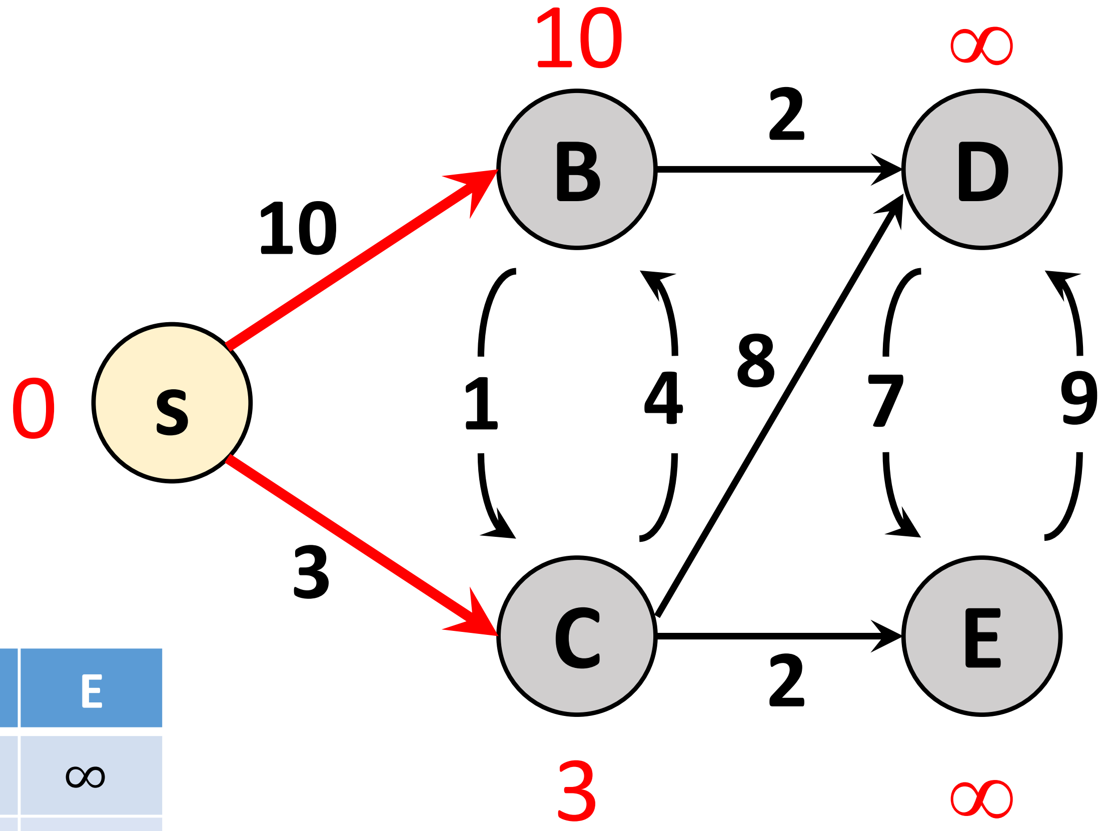


	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

$$X = \{ \, s \qquad\qquad \}$$

Dijkstra’s Algorithm: Demo

Explore *s*

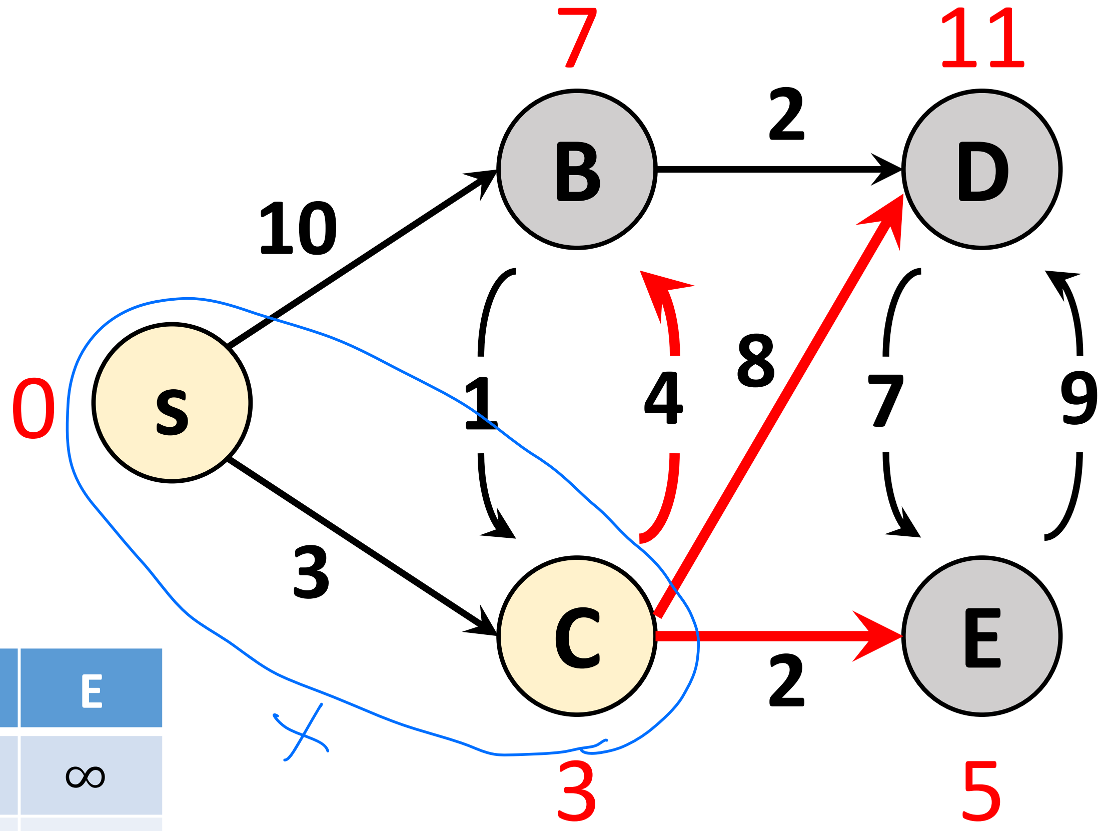


	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞

$X = \{ s \qquad\qquad \}$

Dijkstra's Algorithm: Demo

Explore C

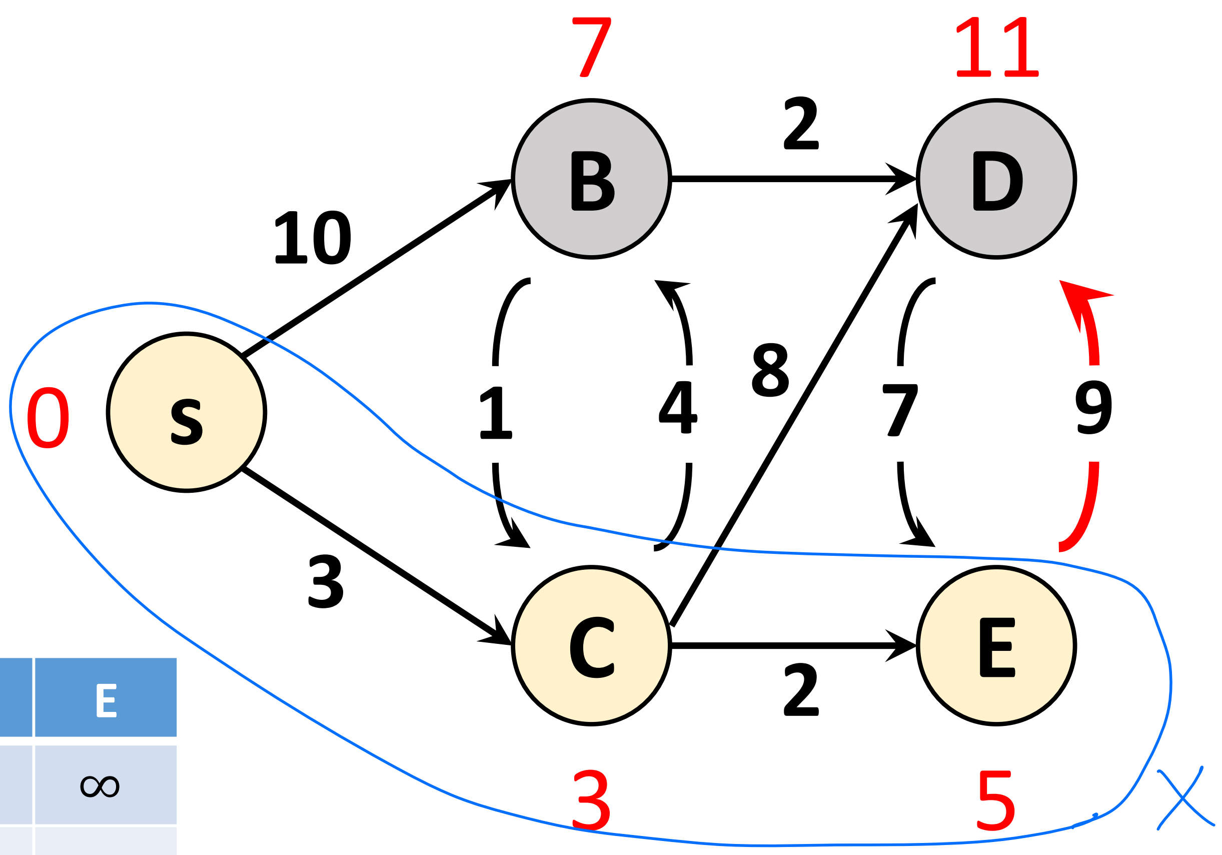


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5

$$X = \{s, C \qquad\qquad \}$$

Dijkstra’s Algorithm: Demo

Explore E

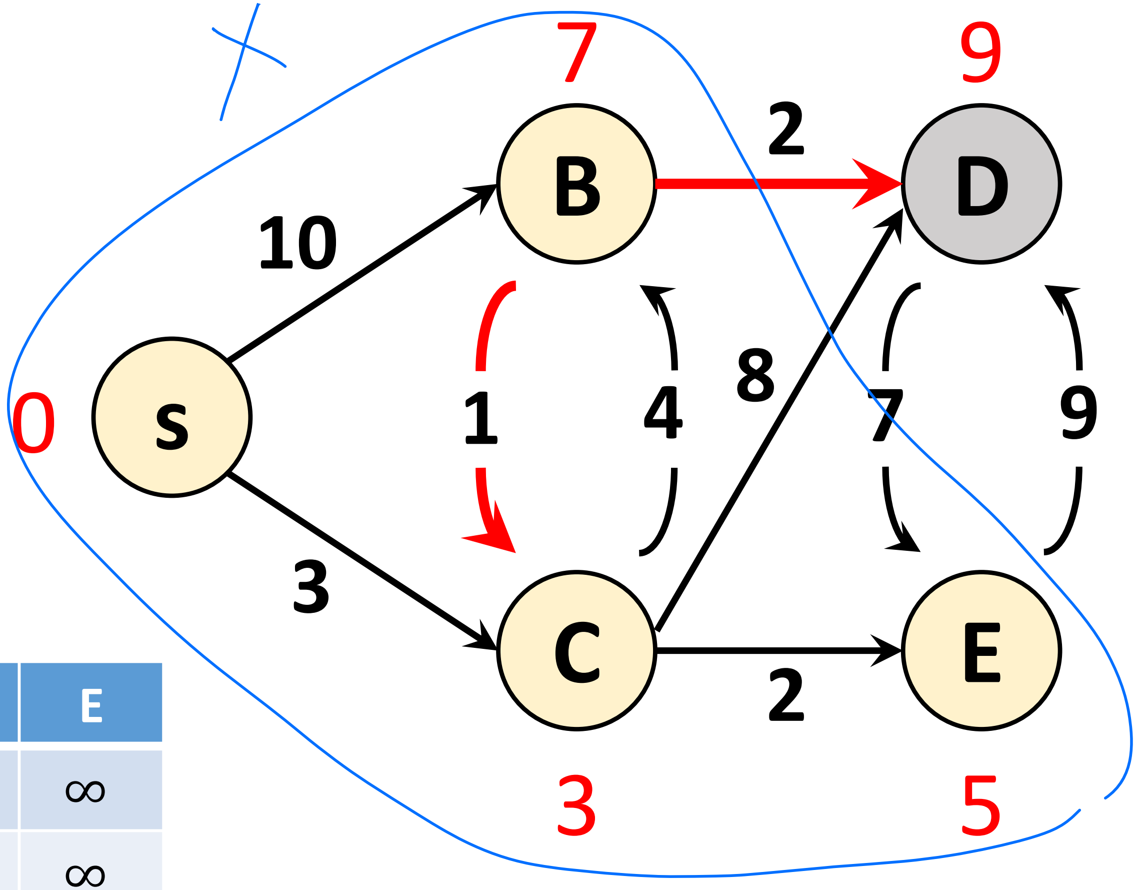


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5

$X = \{s, C, E \qquad\qquad \}$

Dijkstra's Algorithm: Demo

Explore B

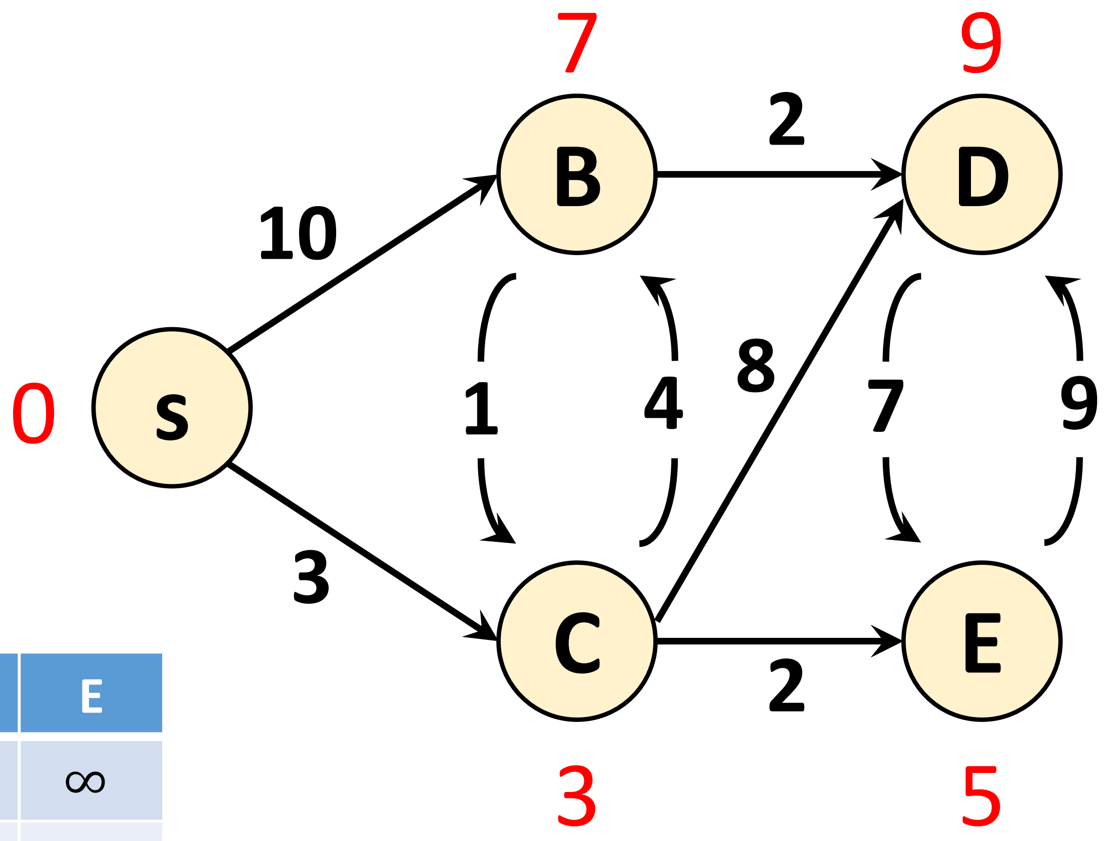


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$X = \{s, C, E, B \qquad\qquad \}$$

Dijkstra’s Algorithm: Demo

Don’t need to explore D

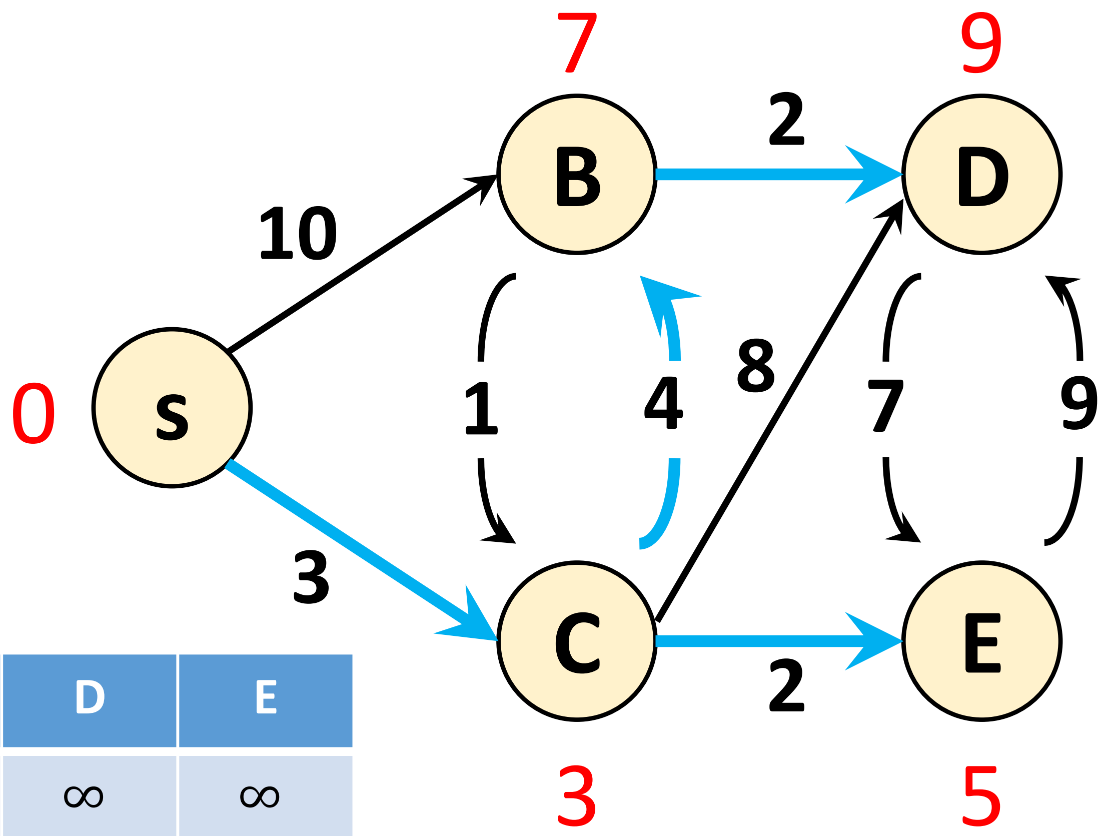


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$X = \{s, C, E, B, D\}$$

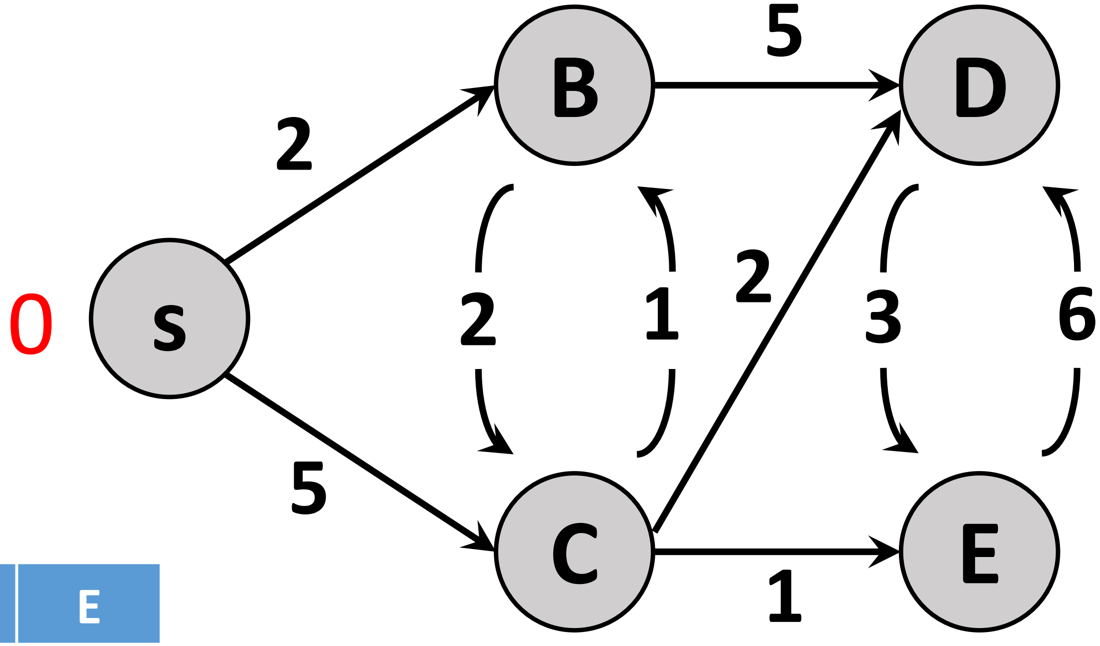
Dijkstra’s Algorithm: Demo

Maintain parent pointers so we can find the shortest paths



	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

Dijkstra’s Algorithm: Practice



	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

$X = \{ \qquad\qquad \}$

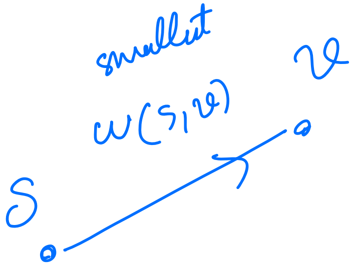
Correctness of Dijkstra

	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

- **Warmup 0:** initially, $d_0(s)$ is the correct distance $d(s,s)$

✓ because all weights are positive, $d(s,s) = 0$.

- **Warmup 1:** before we explore the second node v, $d_1(v)$ is the correct distance $d(s,v)$



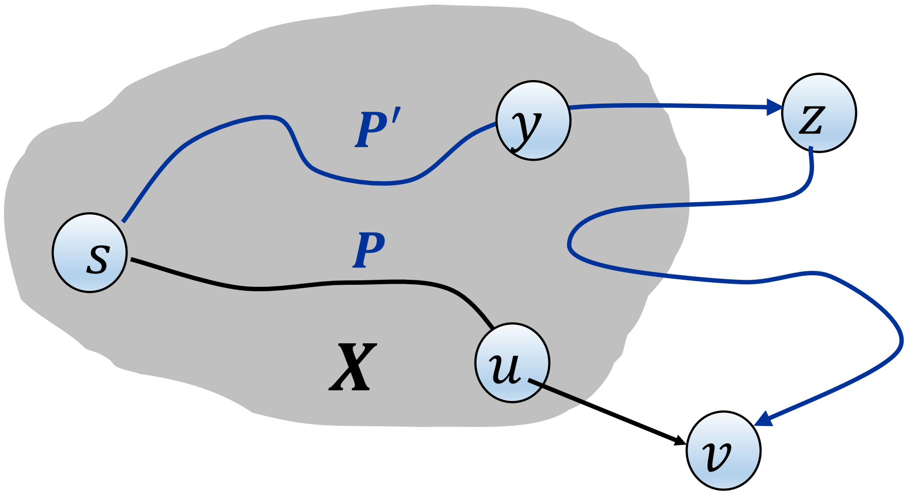
Proof by contradiction: Suppose there is a shorter path $s \to u \to \dots \to v$. Since $w(s,u) \geq w(s,v)$ and all edges have positive weights, this path can't be shorter.

Correctness of Dijkstra

- **Invariant:** before we explore the k-th node *v*, $d_{k-1}(v)$ is correct (tight)
  - $d_{k-1}(v)$: upper bound on distance from *s* to *v* before exploring *k*-th node/after exploring *k-1*-th node

- We just argued the invariant holds before we've explored the 1st and 2nd nodes

Correctness of Dijkstra



- **Invariant:** before we explore the k-th node *v*, $d_{k-1}(v)$ is correct
- **Proof:**

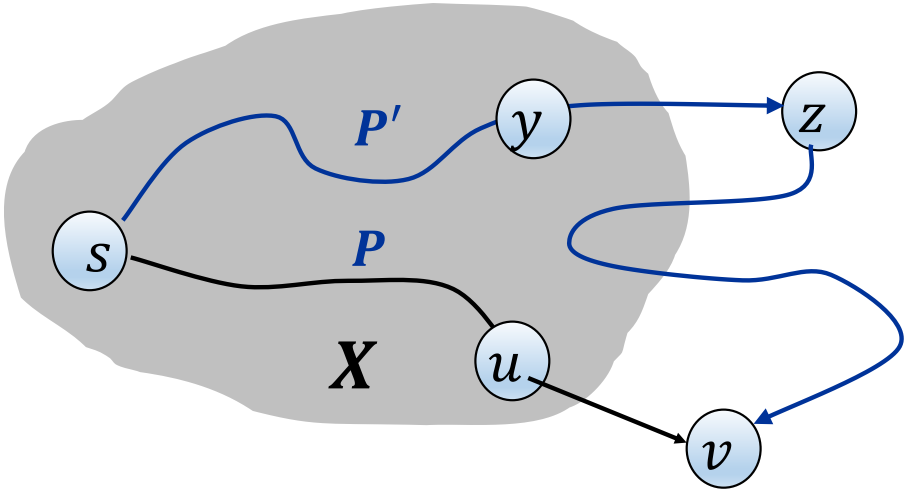
Suppose that $d_{k-1}(v)$ is not correct. That means there is another path P from s to v of length shorter than $d_{k-1}(v)$. Consider two cases:

1) the path starts from s, never leaves X, except for the last vertex v. Take u to be the last vertex of P before v. The moment that we add u to X, we should update $d(v)$ to $d(s,u)+w(u,v)$, Contradiction.

2) the path P reaches another vertex Z outside X (i.e., Z unexplored) before reaching v. Note that $d_{k-1}(z) = d(s,z)$ because the moment we add y to X, we update $d(z)$ to $d(s,y)+w(y,z)$. But since $d_{k-1}(v) \leq d_{k-1}(z)$, the path that goes from s to z and then to v cannot have shorter length than $d\;(v)$

contradiction.

Correctness of Dijkstra



- **Invariant:** before we explore the k-th node v, $d_{k-1}(v)$ is correct
- **Proof:**

Implementing Dijkstra Naively

initialization

```
Dijkstra(G = (V,E,{w(e)},s):
  d[s] ← 0, d[u] ← ∞ for every u != s
  parent[u] ←⊥ for every u
  Q ← V                 // Q holds the unexplored nodes
```

$\Theta(n^2)$ overall

```
  While (Q is not empty):
    $u \leftarrow \operatorname{argmin}_{w \in Q} d[w]$     //Find closest unexplored
    Remove $u$ from Q
```

$O(m)$ time overall

every edge (u,v) is only explored once when u leaves Q.

```
    // Update the neighbors of u
    For (v in out[u]):
      If (d[v] > d[u] + w(u,v)):
        d[v] ← d[u] + w(u,v)
        parent[v] ← u

  Return (d, parent)
```

overall takes $O(n^2+m)$ time.

Priority Queues / Heaps

Priority Queues

- Need a data structure Q to hold key-value pairs so that we can quickly find closest unexplored node
  - Keys = vertices that are unexplored
  - Values = the upper bound $d(v)$ for v

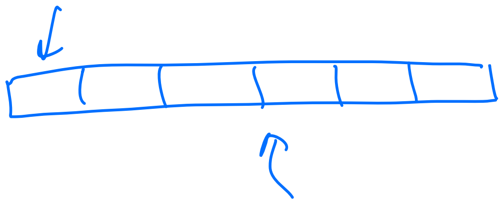
$(\underset{\overset{\uparrow}{\text{key}}}{v}, d(v))$

- Need to support the following operations
  - Insert(Q,k,v): add a new key-value pair
  - Lookup(Q,k): return the value of some key
  - ExtractMin(Q): identify the key with the smallest value
  - DecreaseKey(Q,k,v): reduce the value of some key

Priority Queues

{ a : 10,
b : 15,
c : 20 }

- **Naïve approach:** dictionary
  - Insert, DecreaseKey, Lookup take O(1) time
  - ExtractMin takes O(n) time

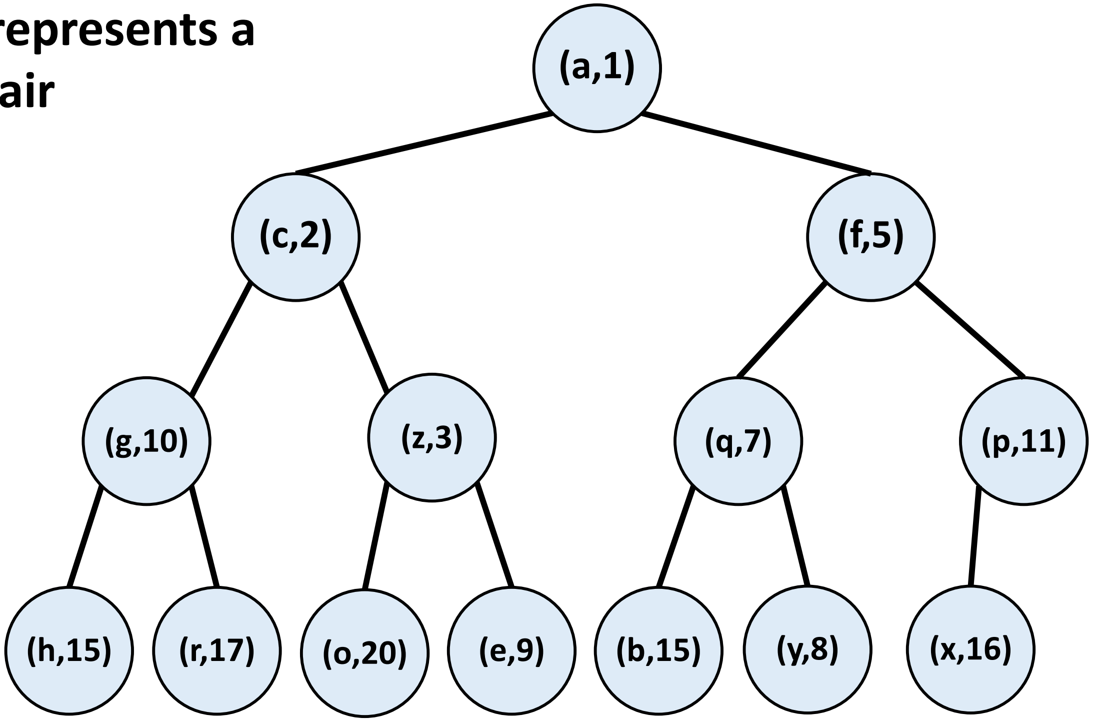


- **(Binary) Heaps:** implement all operations in O(log n) time where n is the number of keys

Heaps

- **Organize key-value pairs as a complete binary tree**
  - Later we'll see how to store pairs in an array
- **Heap Order:** If a is the parent of b, then val(a) ≤ val(b)

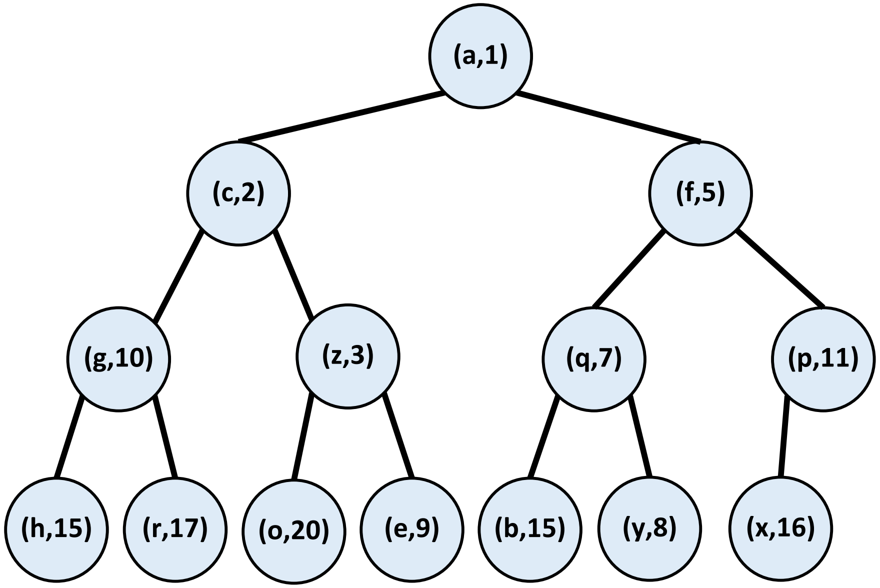
Each node represents a key-value pair



(key, value)

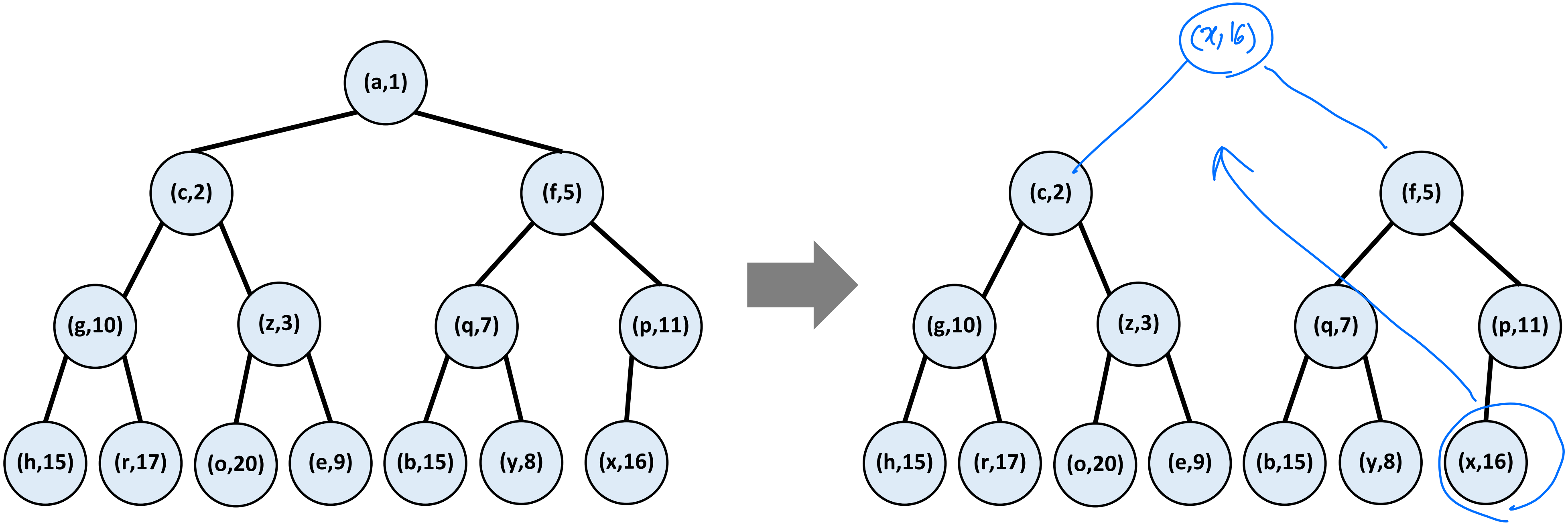
Implementing ExtractMin

Where is the min?



Implementing ExtractMin

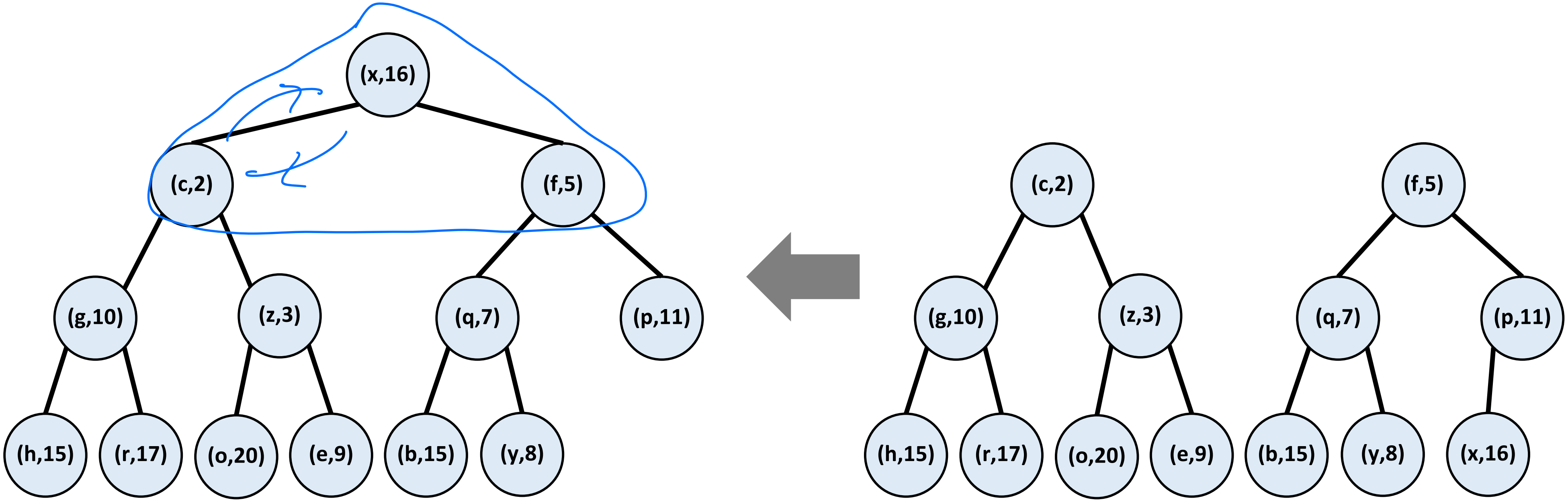
- **If we delete the min, we get two binary trees**
- **We need to get back to having one tree**



- **Idea:** move the last element up to the root

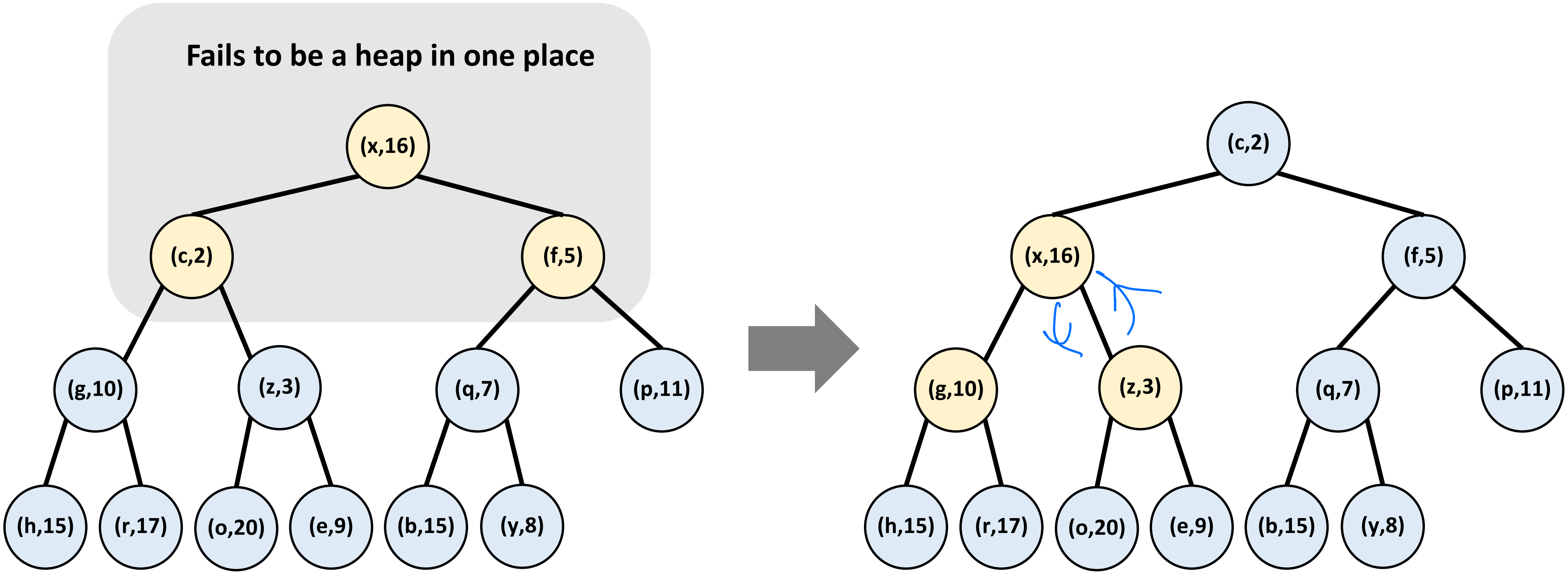
Implementing ExtractMin

- **Problem?** The root does not have value smaller than its children



- **Idea:** Swap the root with its child with smallest value

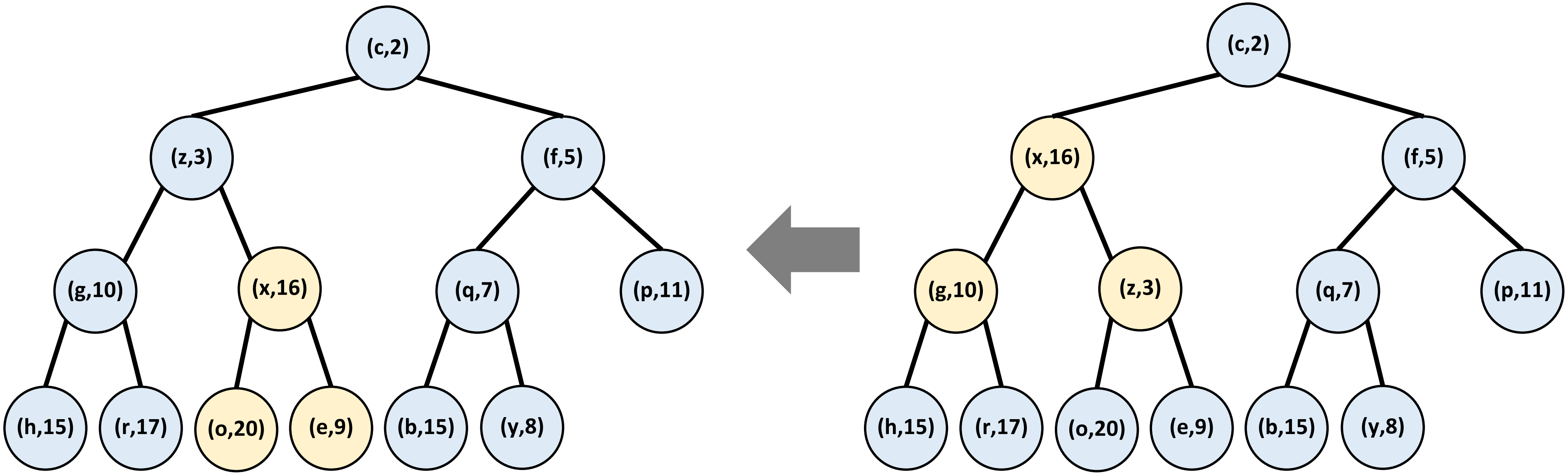
Implementing ExtractMin


Fails to be a heap in one place

- **# of levels:** $O(\lg n)$

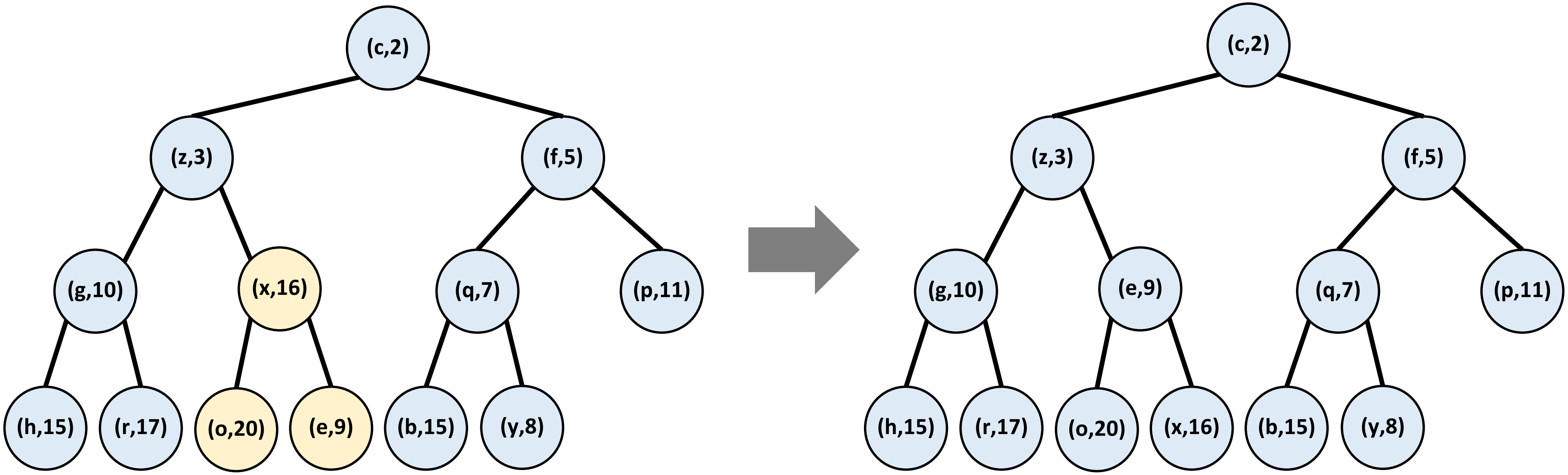
Implementing ExtractMin

- **Idea:**

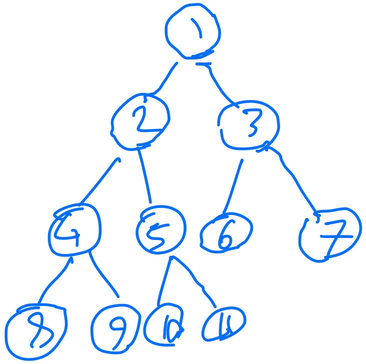


Implementing ExtractMin

- **Observation:**

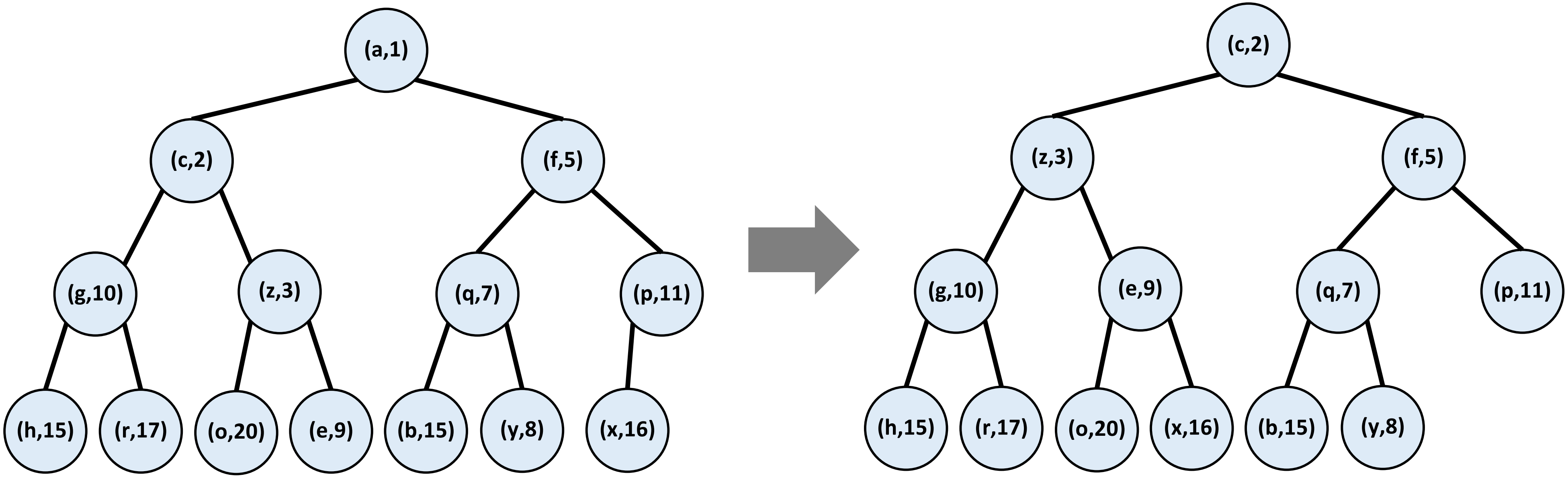


Implementing ExtractMin

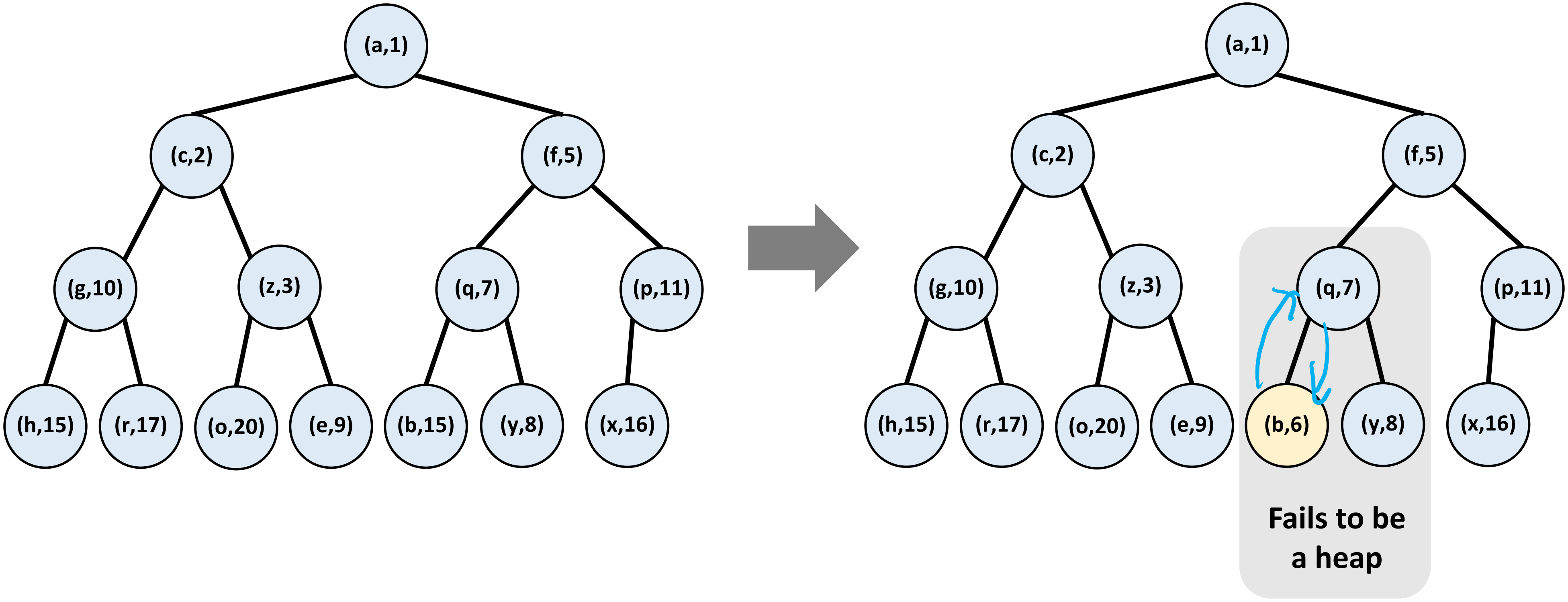


- Three steps:
  - Pull the minimum from the root
  - Move the last element to the root
  - Repair the heap-order (heapify down)

from the last level

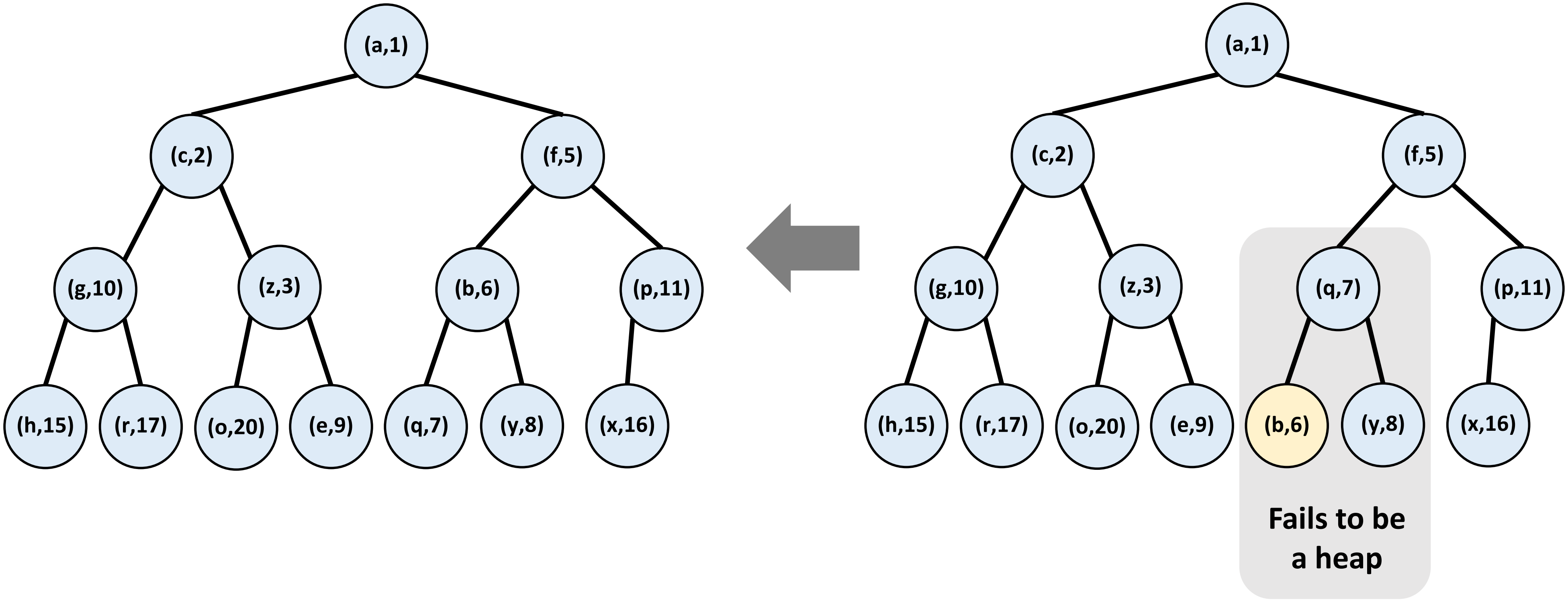


Implementing DecreaseKey



Fails to be a heap

Implementing DecreaseKey

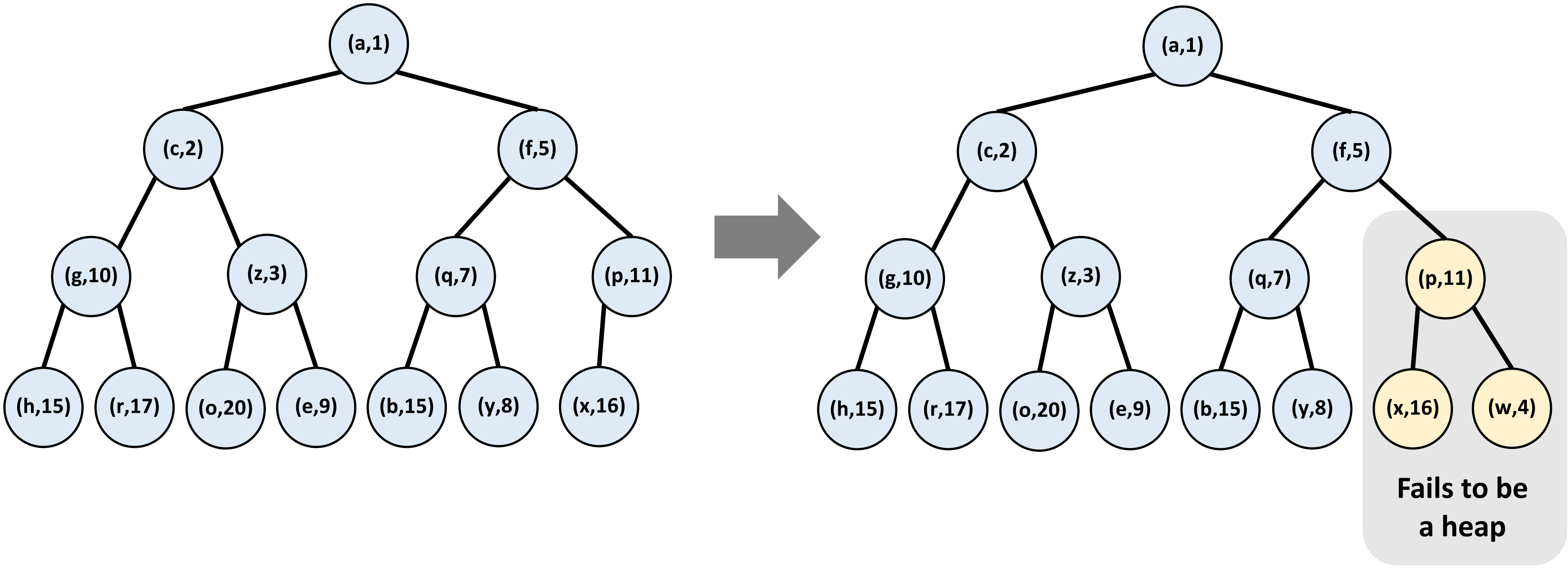


Fails to be a heap

Implementing DecreaseKey

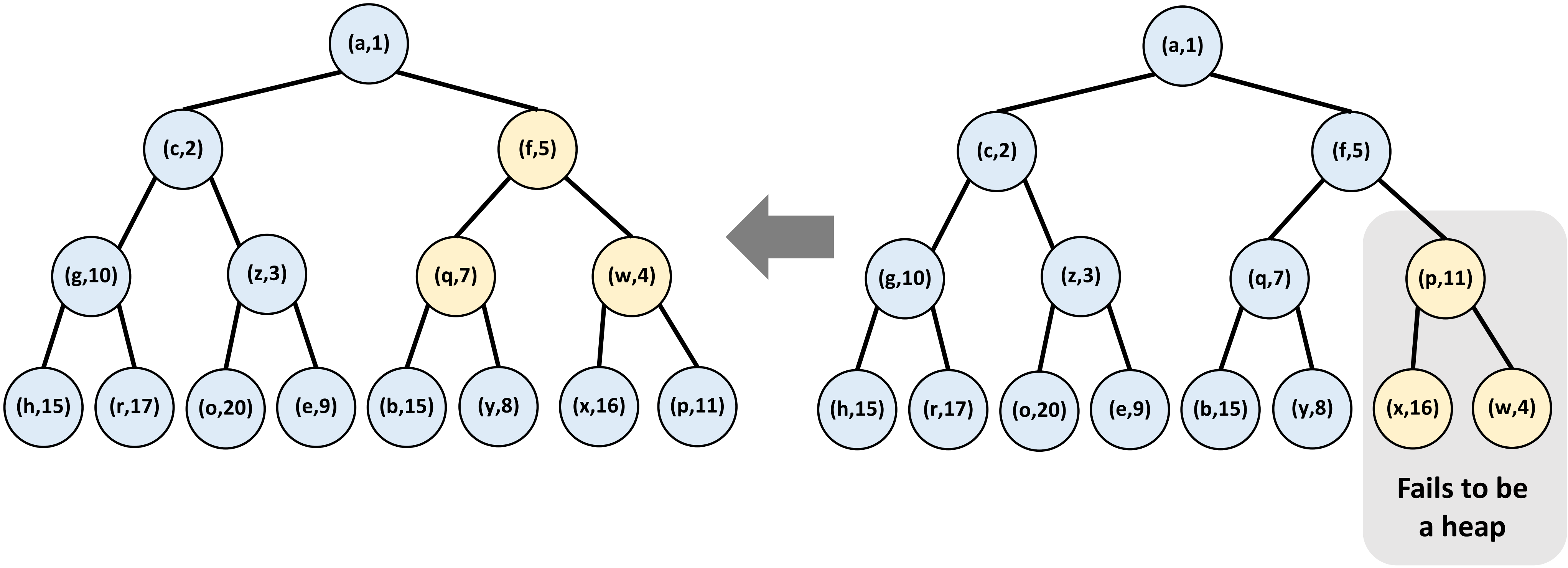
- Two steps:
  - Change the key
  - Repair the heap-order (heapify up)

Implementing Insert



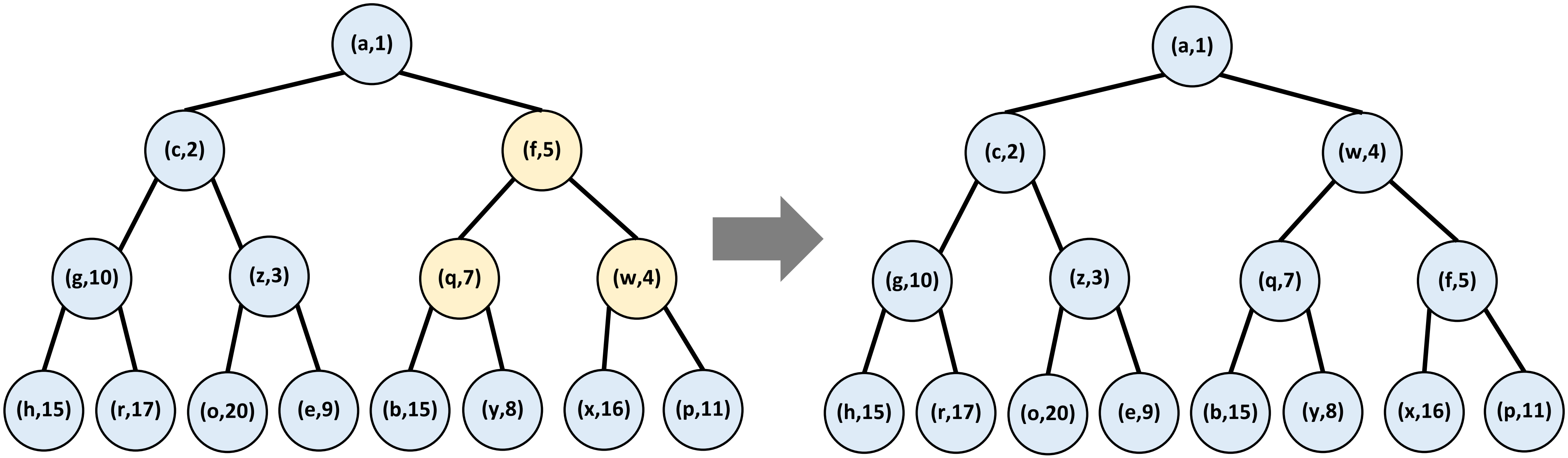
Fails to be a heap

Implementing Insert



Fails to be a heap

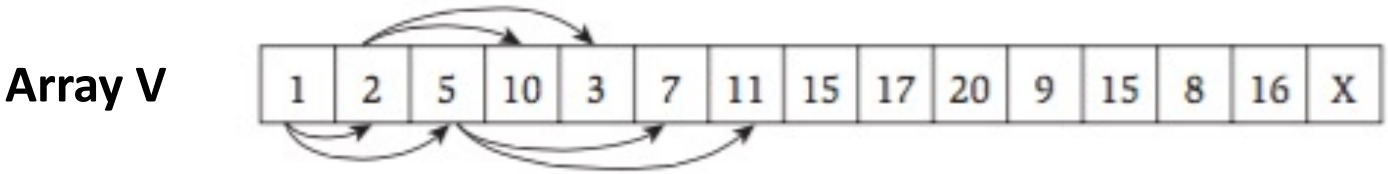
Implementing Insert



Implementing Insert

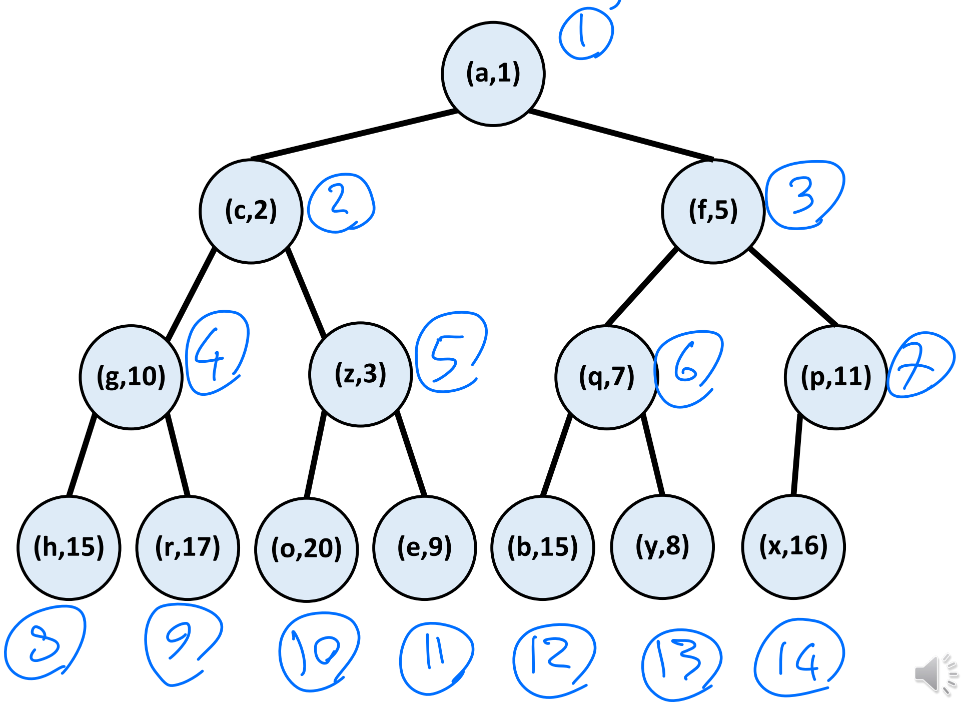
- Two steps:
  - Put the new key in the last location
  - Repair the heap-order (heapify up)

Implementation Using Arrays



- Maintain an array V holding the values
- Maintain a dictionary K mapping keys to ~~values~~ location in the heap
  - Can find the value (**lookup**) for a given key in $O(1)$ time
- For any node i in the binary tree
  - LeftChild(i) = 2i
  - RightChild(i) = 2i+1
  - Parent(i) = ⌊i/2⌋

{
a : 1
c : 2
f : 3
}



Binary Heaps

- **Heapify:**
  - O(1) time to fix a single triple
  - With n keys, might have to fix O(log n) triples
  - Total time to heapify is O(log n)

- **Lookup** takes O(1) time
- **ExtractMin** takes O(log n) time
- **DecreaseKey** takes O(log n) time
- **Insert** takes O(log n) time

Implementing Dijkstra with Heaps

```
Dijkstra(G = (V,E,{ℓ(e)}, s):
  Let Q be a new heap
  Let parent[u] ← ⊥ for every u
  Insert(Q,s,0), Insert(Q,u, ∞) for every u != s

  While (Q is not empty):
    (u,d[u]) ← ExtractMin(Q)    O(lg n) time

    For (v in out[u]):
      d[v] ← Lookup(Q,v)
      If (d[v] > d[u] + ℓ(u,v)):
        DecreaseKey(Q,v,d[u] + ℓ(u,v))
        parent[v] ← u

  Return (d, parent)
```

$O(n \lg n)$ total

$O(m \lg n)$

$O(n + m \lg n)$

Alg1: $O(n^2 + m)$ Alg2: $O(n + m \lg n)$

Dijkstra Summary:

- **Dijkstra's Algorithm** solves **single-source shortest paths** in non-negatively weighted graphs
  - Algorithm can fail if edge weights are negative!

- **Implementation:**
  - A **priority queue** supports all necessary operations
  - Implement priority queues using **binary heaps**
  - Overall running time of Dijkstra: $O(m \log n)$
  - Can do even better using Fibonacci heaps!