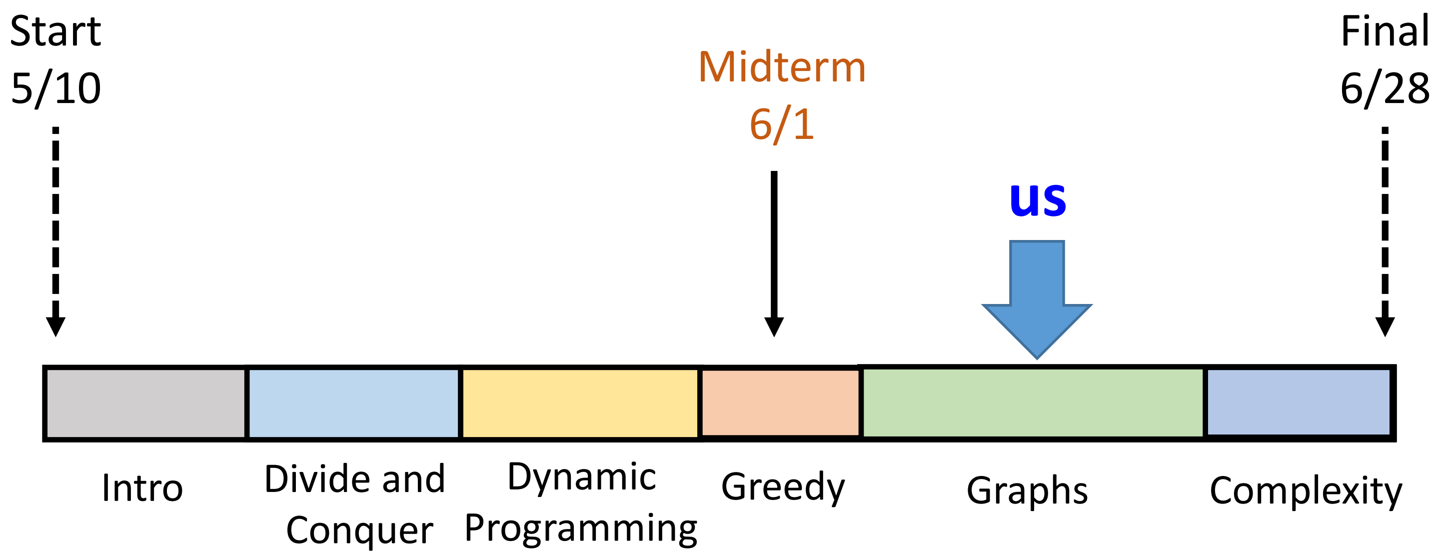
HW4 - due 6/12
Q4 - out 6/11 due M 6/14 @ NOON

CS3000: Algorithms & Data
Drew van der Poel

Lecture 18
- Dijkstra's

June 10, 2021

Outline



Last class: Graphs: Strongly Connected Components, Dijkstra's

Next class: Graphs: Bellman-Ford

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:

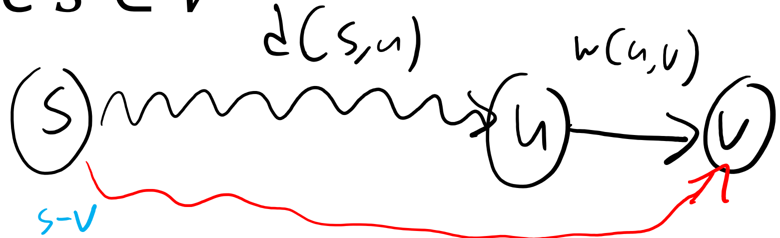
$$\ell(P) = \sum_{e \in P} w(e)$$

- 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 $(s,t) \in V$

Structure of Shortest Paths

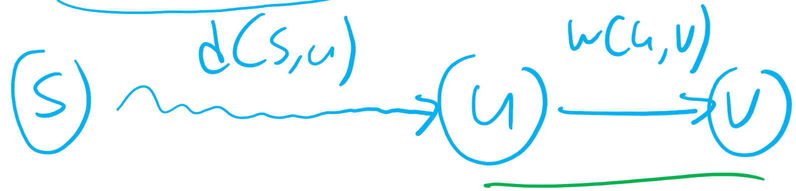
- If $(u, v) \in E$, then $d(s, v) \le d(s, u) + w(u, v)$ for every node $s \in V$

$\exists$ a path from s to v w/ (u,v) as the final edge of length $d(s,u) + w(u,v)$



The shortest s-v path can't be any longer than this

- 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)



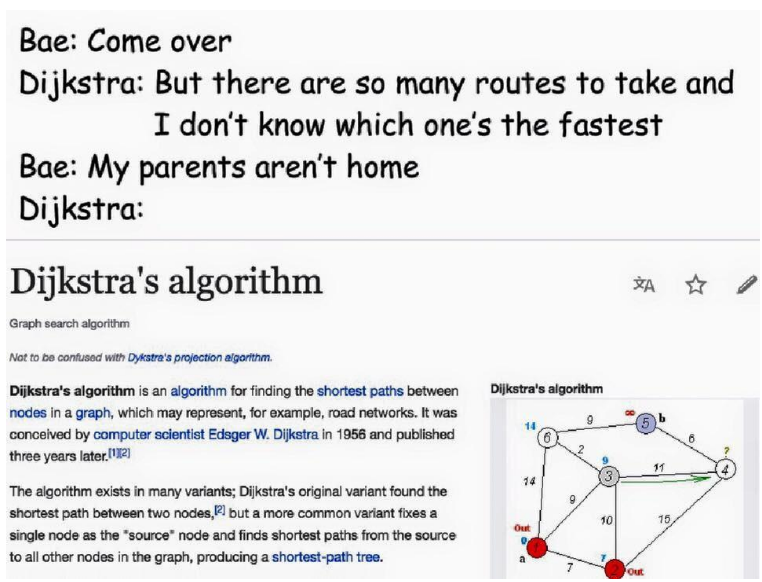
No s-v path is shorter than this one

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**

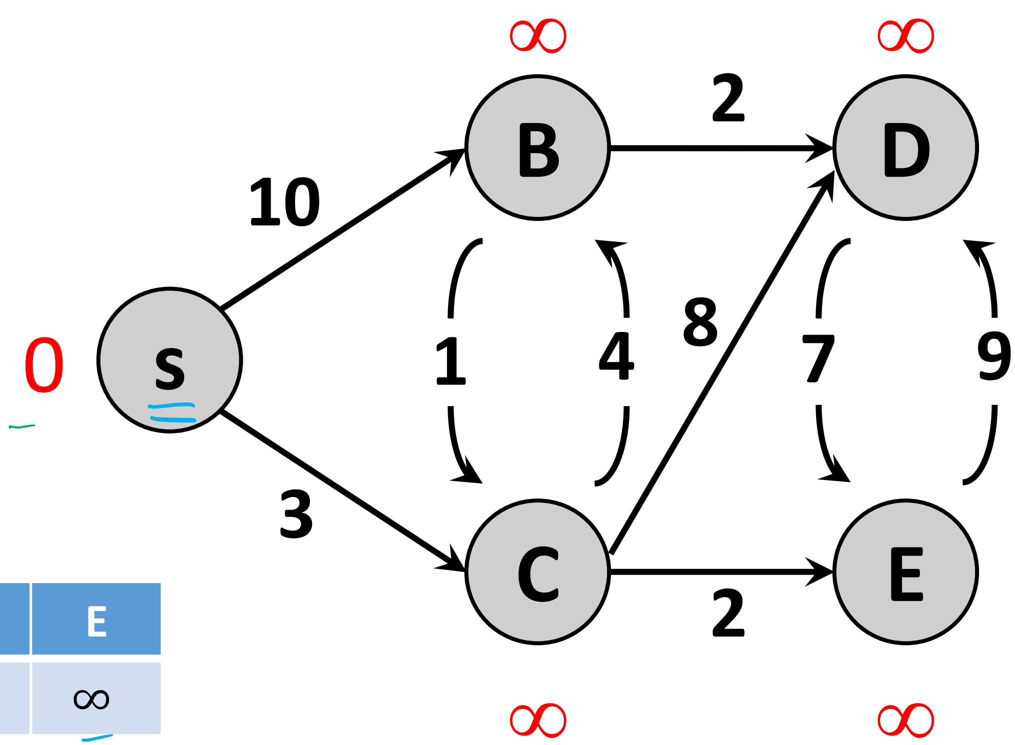
Dijkstra's Algorithm

- **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)



Dijkstra's Algorithm: Demo

Initialize



	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞

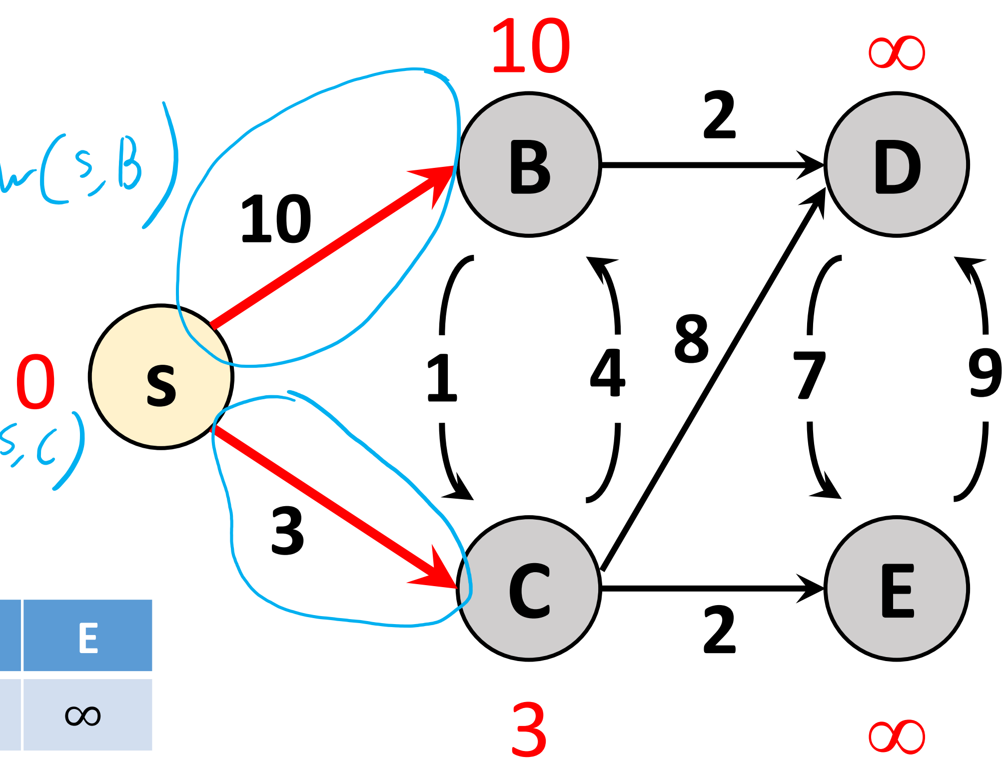
$$X = \{ \qquad \}$$

Dijkstra's Algorithm: Demo

Explore *s*

$$d(s,B) \le d(s,s) + w(s,B)$$
$$\le 0 + 10$$

$$d(s,C) \le d(s,s) + w(s,C)$$
$$\le 0 + 3$$



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

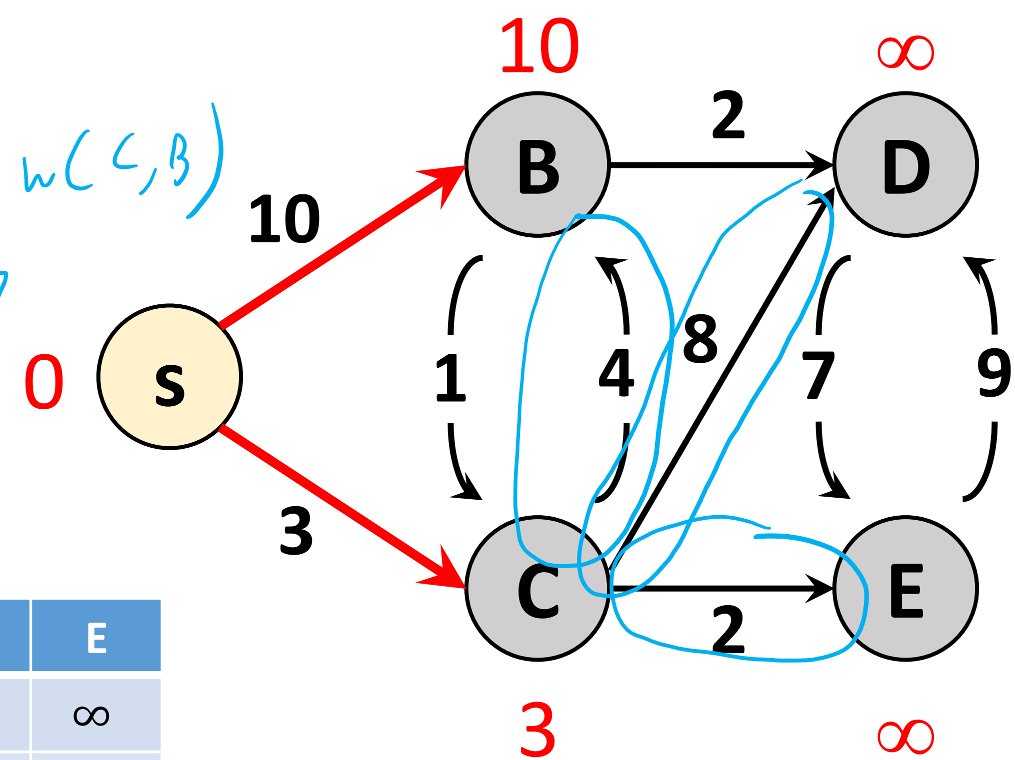
$X = \{ s, C \}$

Dijkstra's Algorithm: Demo

Explore *s*

$d(s,B) \leq d(s,C) + w(C,B)$

$3 + 4 = 7$



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

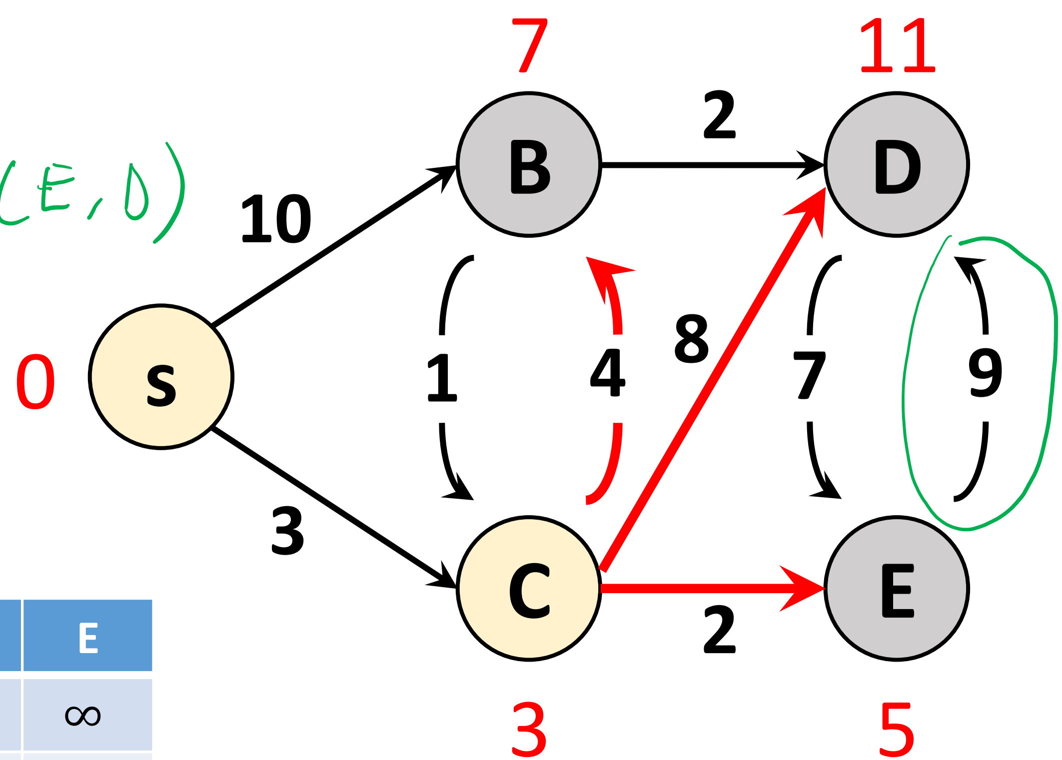
$X = \{ s, C, E \}$

Dijkstra's Algorithm: Demo

Explore C

$d(s,D) \le d(s,E) + w(E,D)$

$\le 5 + 9$

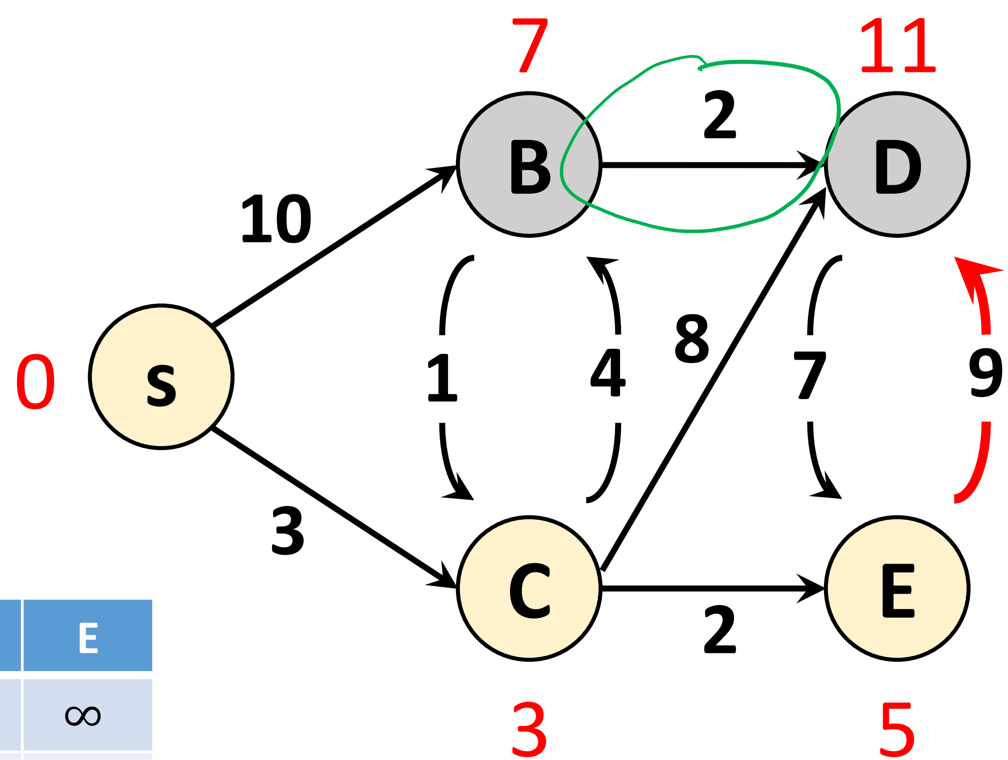


	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞
$d_2(u)$	**0**	**7**	**3**	**11**	$\boxed{\textbf{5}}$
		7		11	

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

Dijkstra's Algorithm: Demo

Explore E



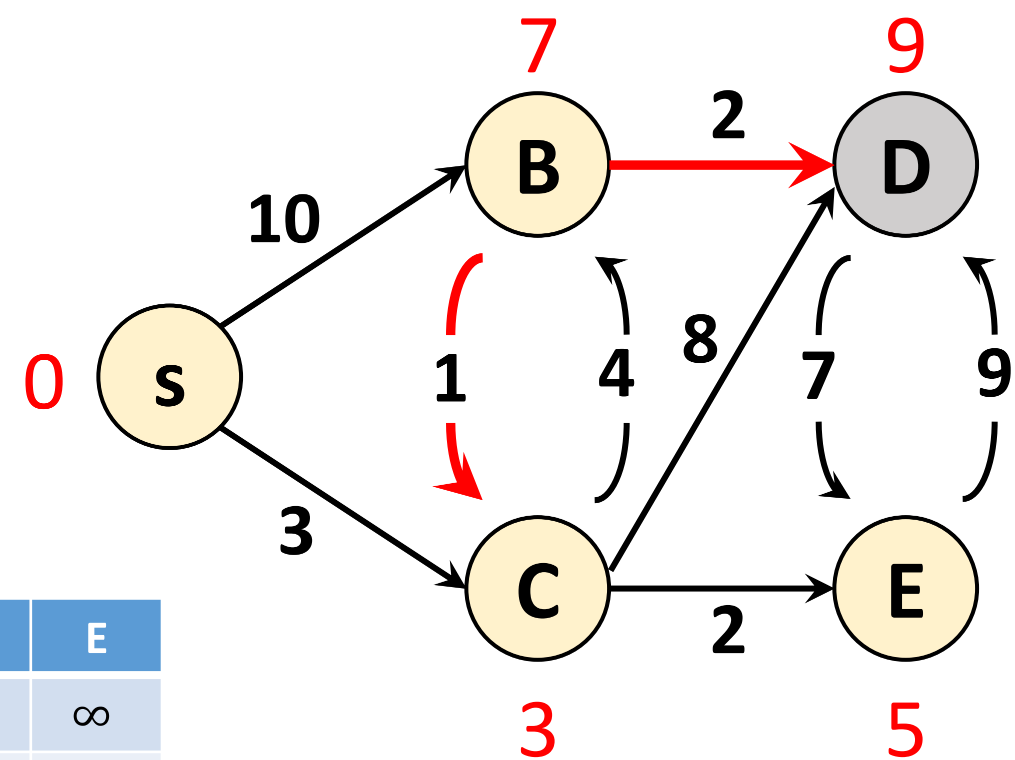
	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5

9

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

Dijkstra's Algorithm: Demo

Explore B

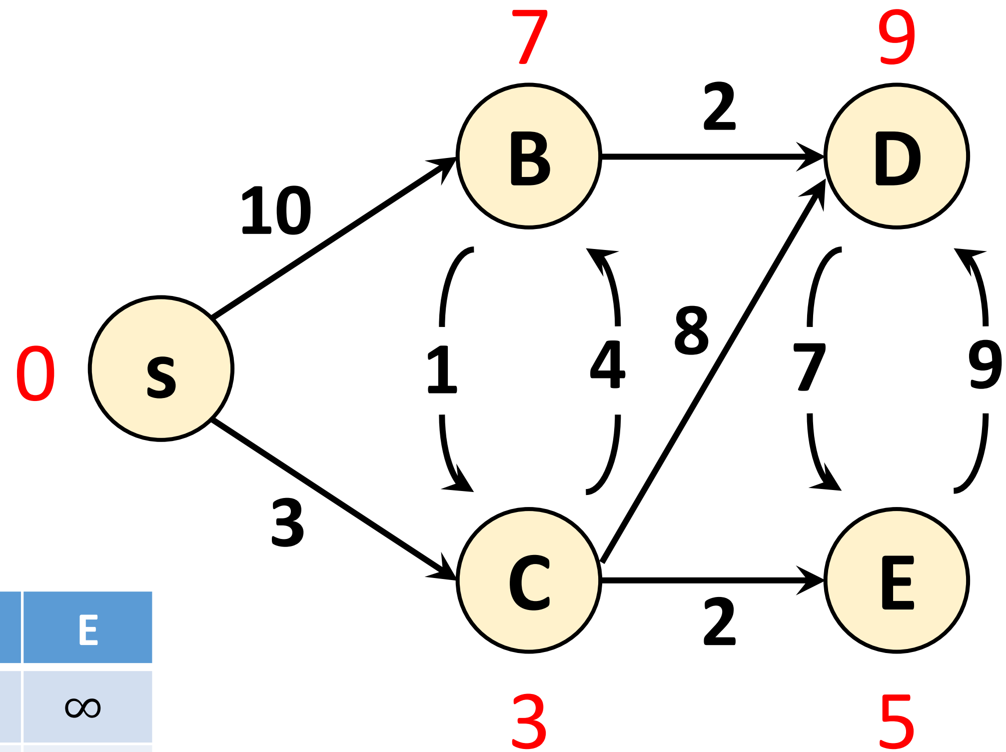


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

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

Dijkstra's Algorithm: Demo

Don't need to explore D

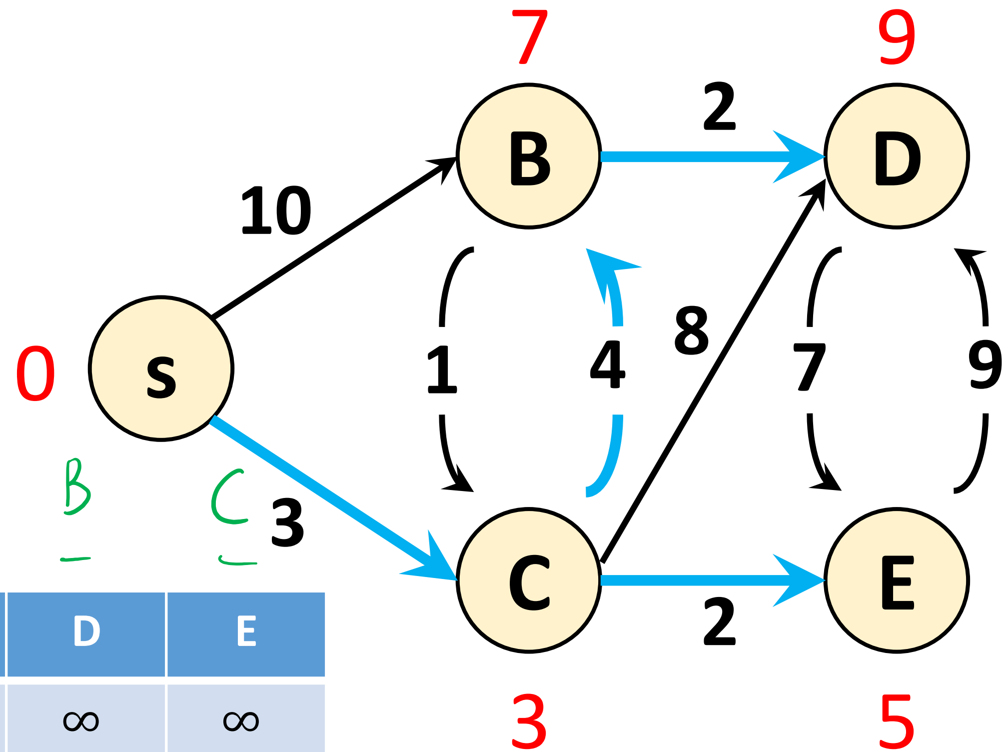


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

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

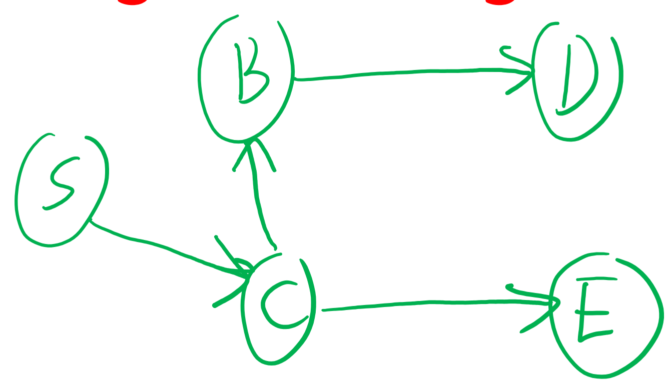
Dijkstra's Algorithm: Demo

Maintain parent pointers so we can find the shortest paths

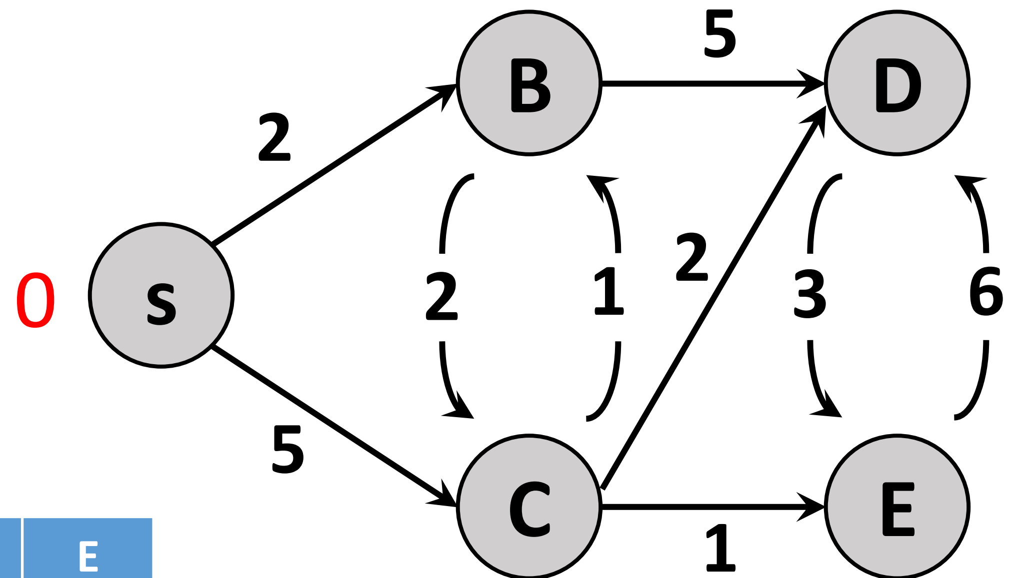


Parent – C S B C

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



Dijkstra's Algorithm: Practice



	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

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

Implementing Dijkstra Naively

d[i] – upper bound on distance from s to i

```
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

  While (Q is not empty):
    u ← argmin_{w∈Q} d[w]      //Find closest unexplored
    Remove u from 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)
```

O(n)

O(n) times

O(n) each

O(n²) total

O(1)

O(out-deg(u)) each

O(m) Total

O(1)

$\sum_{u \in V}$ out-deg[u] = m

Total: $O(n^2 + m)$

$= O(n^2)$

Priority Queues / Heaps

Priority queue : heap :: list : array

data structure

Priority Queues

- Need a data structure Q to hold key-value pairs so that we can quickly find closest unexplored node
  - Keys = nodes
  - Values = distance upper bounds
- Need to support the following operations
  - Insert(Q,k,v): add a new key-value pair ← Initialization
  - Lookup(Q,k): return the value of some key

    if $d[v] > d[u] + w(u,v)$

  - ExtractMin(Q): identify the key with the smallest value

    $u \leftarrow \operatorname{argmin}_{w \in Q} (d[w])$

  - DecreaseKey(Q,k,v): reduce the value of some key

    $d[v] \leftarrow d[u] + w(u,v)$

Priority Queues

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

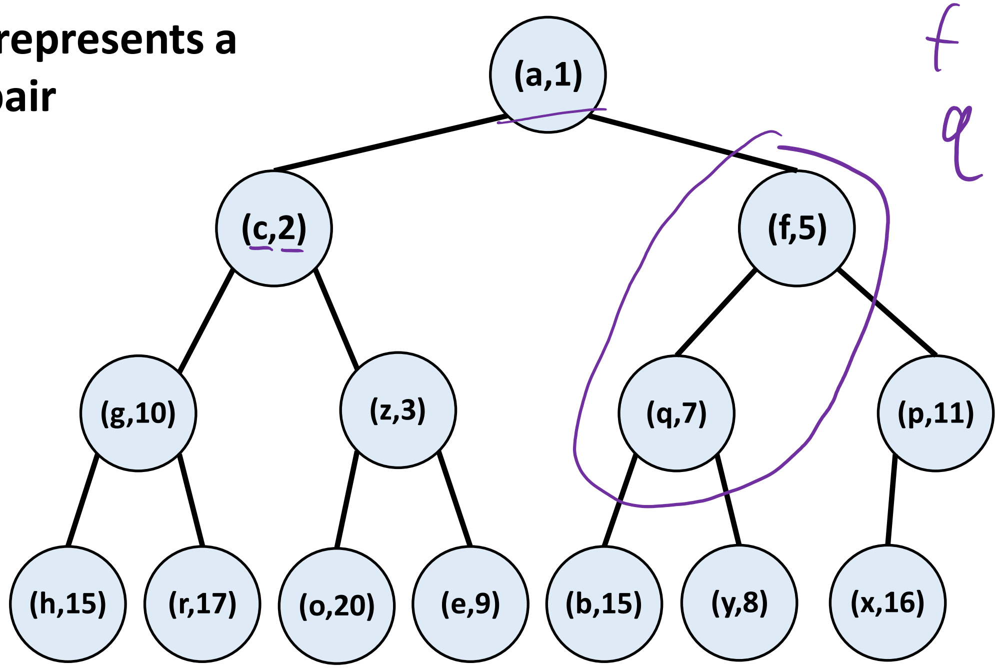
- **(Binary) Heaps:** implement all operations in O(log n) time where n is the number of keys

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)



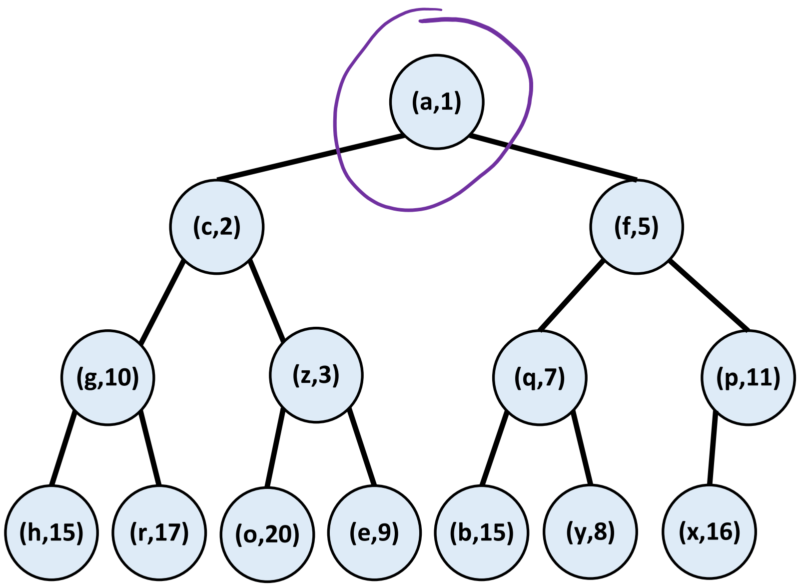
f is parent of q

val(f) = 5

$\geq$ 7 = val(q)

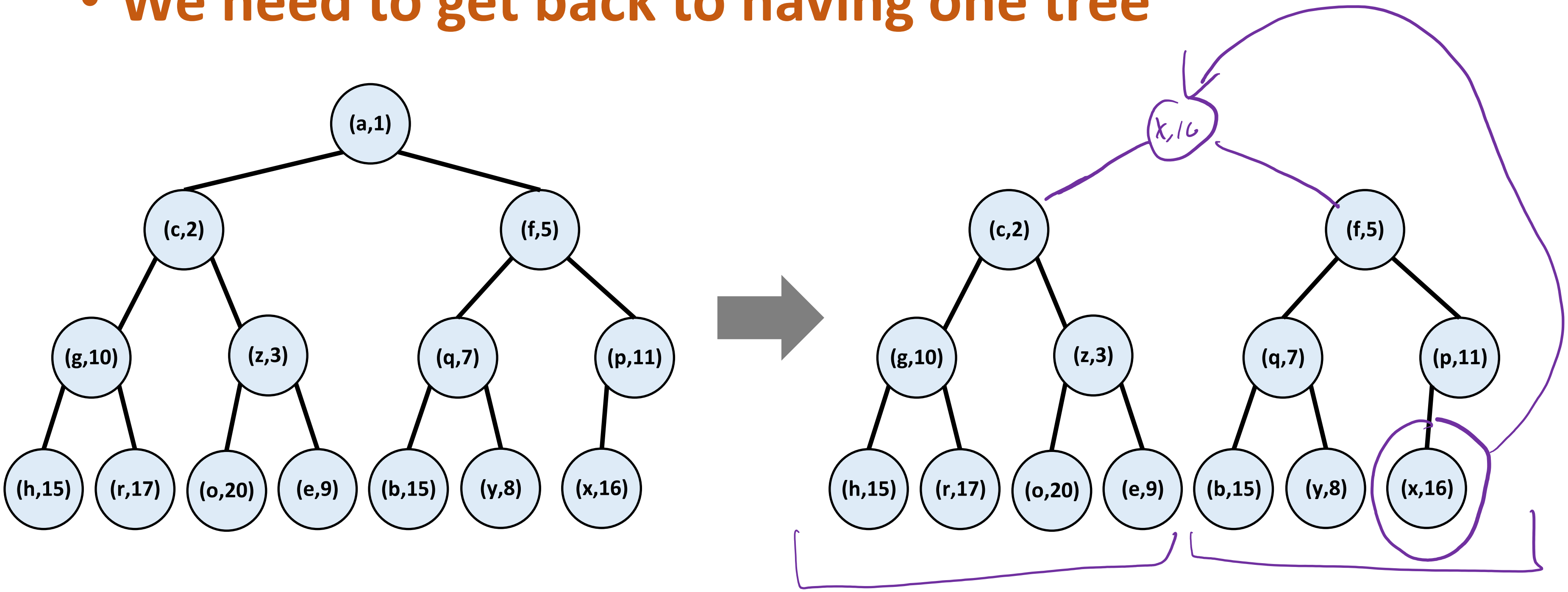
Implementing ExtractMin

Where is the min? root



Implementing ExtractMin

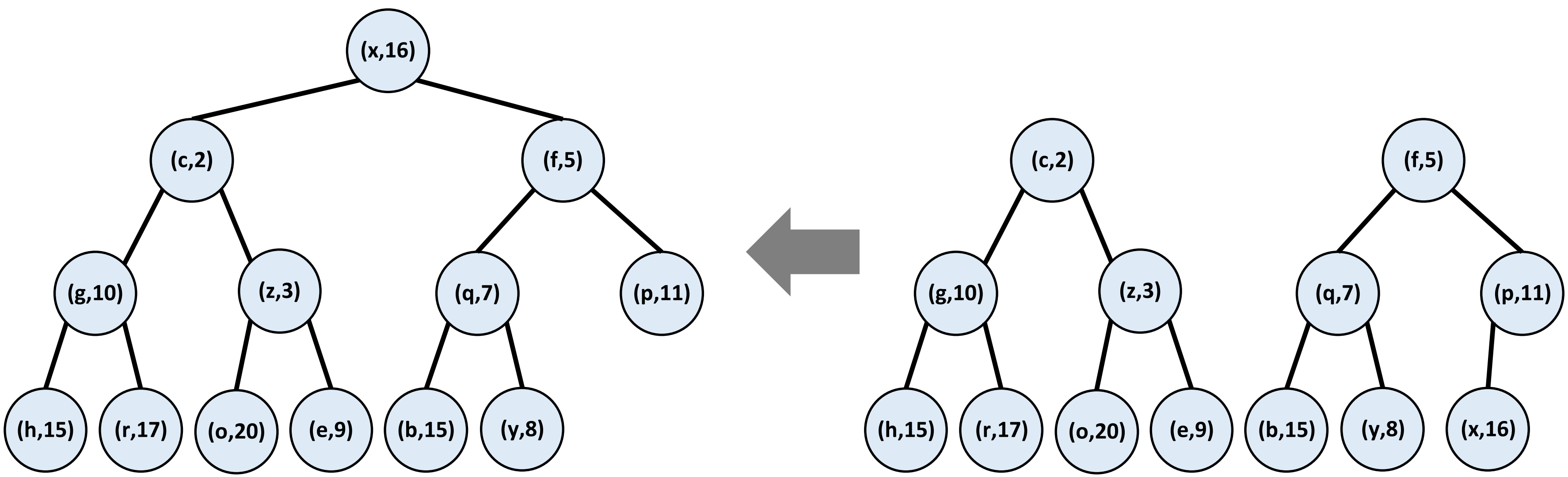
- **If we delete the min, we get two binary trees**
- **We need to get back to having one tree**



- **Idea:** take the last leaf and place @ root

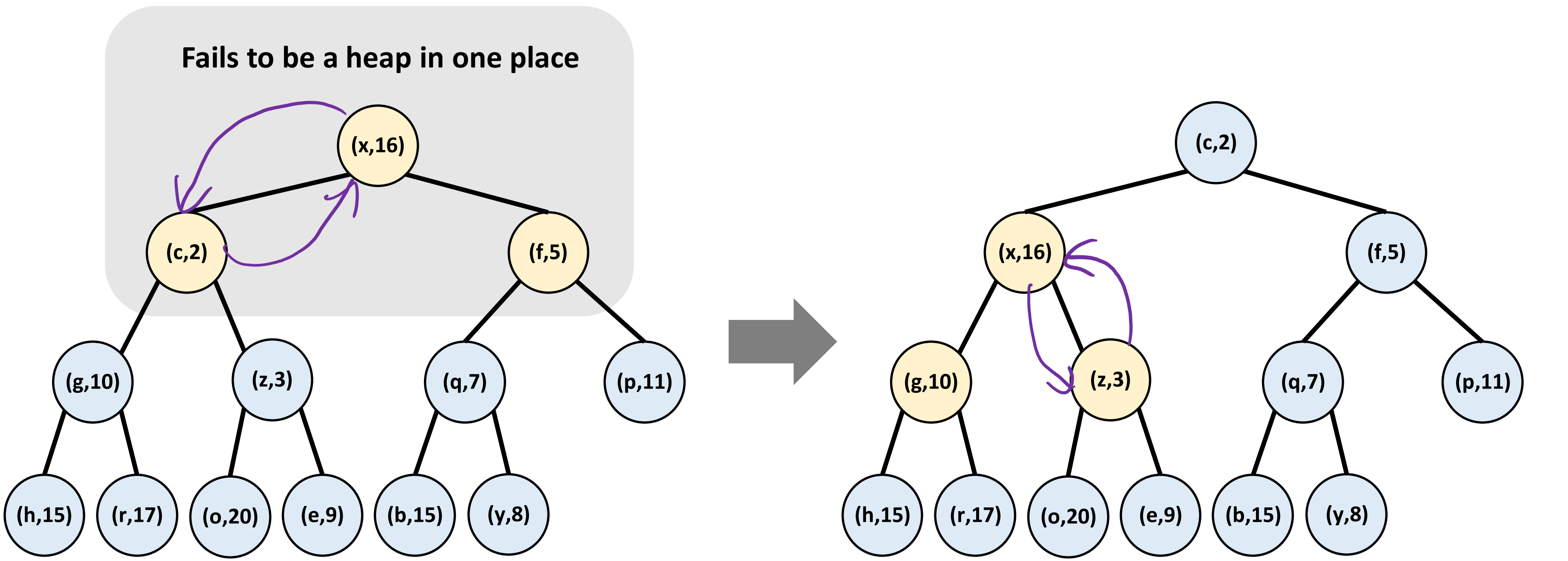
Implementing ExtractMin

- **Problem?** heap order is not observed!



- **Idea:** Swap it w/ smaller child

Implementing ExtractMin



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

$n = 13$

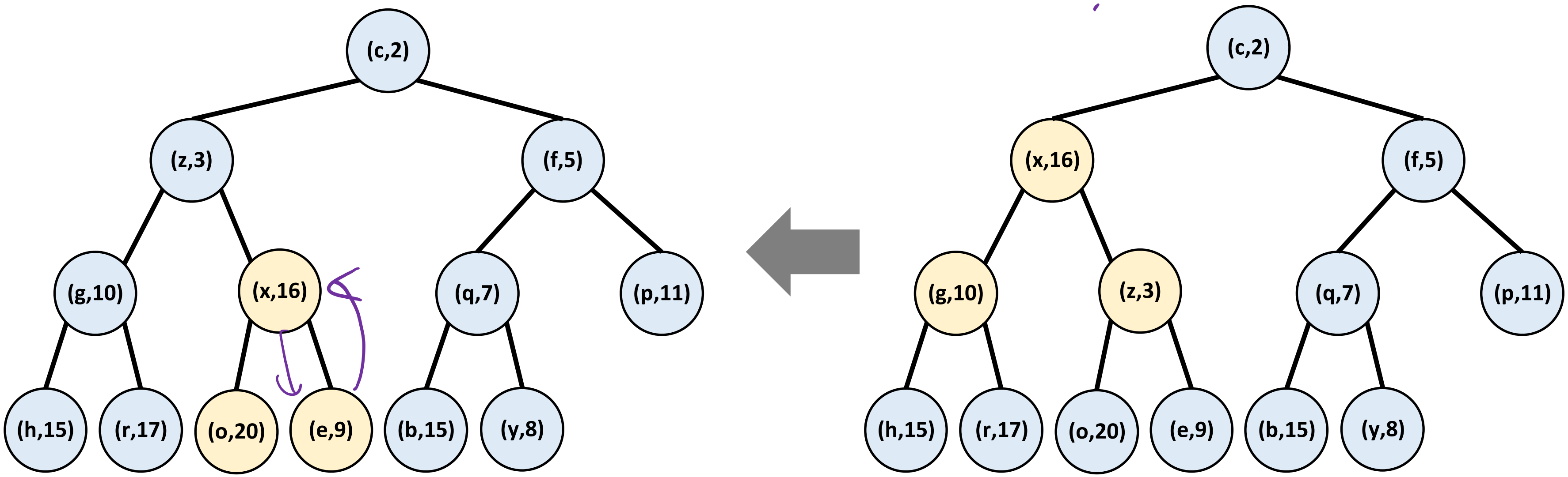
$\log_2 13 = 3.X$

# of levels $= 4$

$\rightarrow \lfloor \log_2 n \rfloor + 1$

Implementing ExtractMin

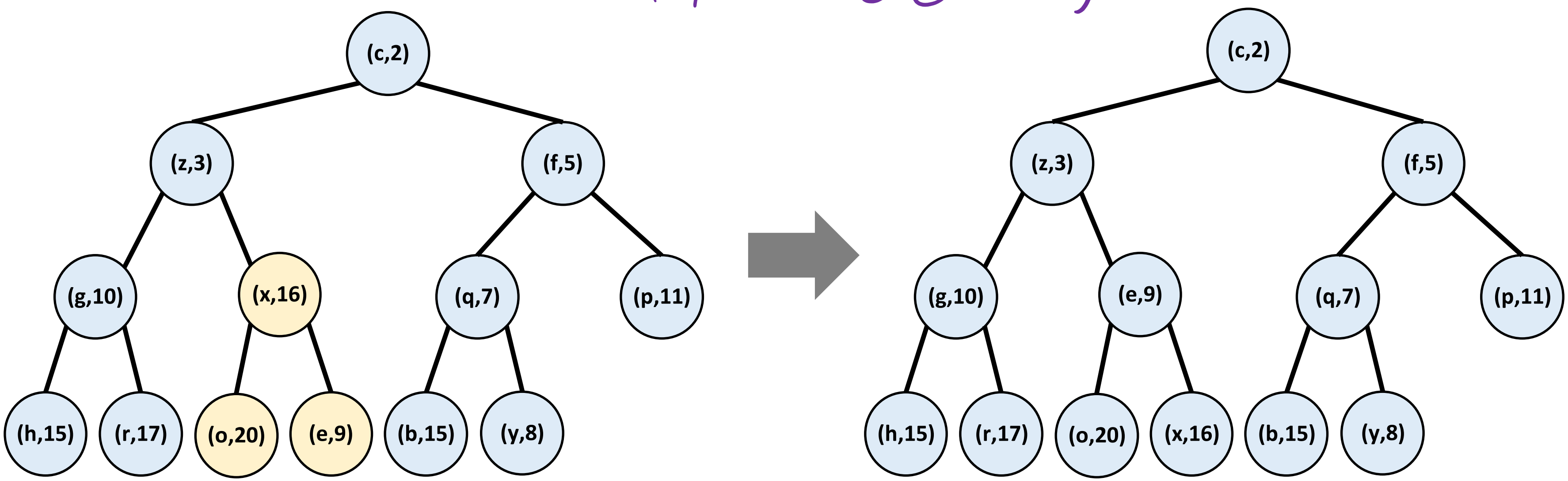
- **Idea:** Keep on pushing down problematic node!



Implementing ExtractMin

- **Observation:**

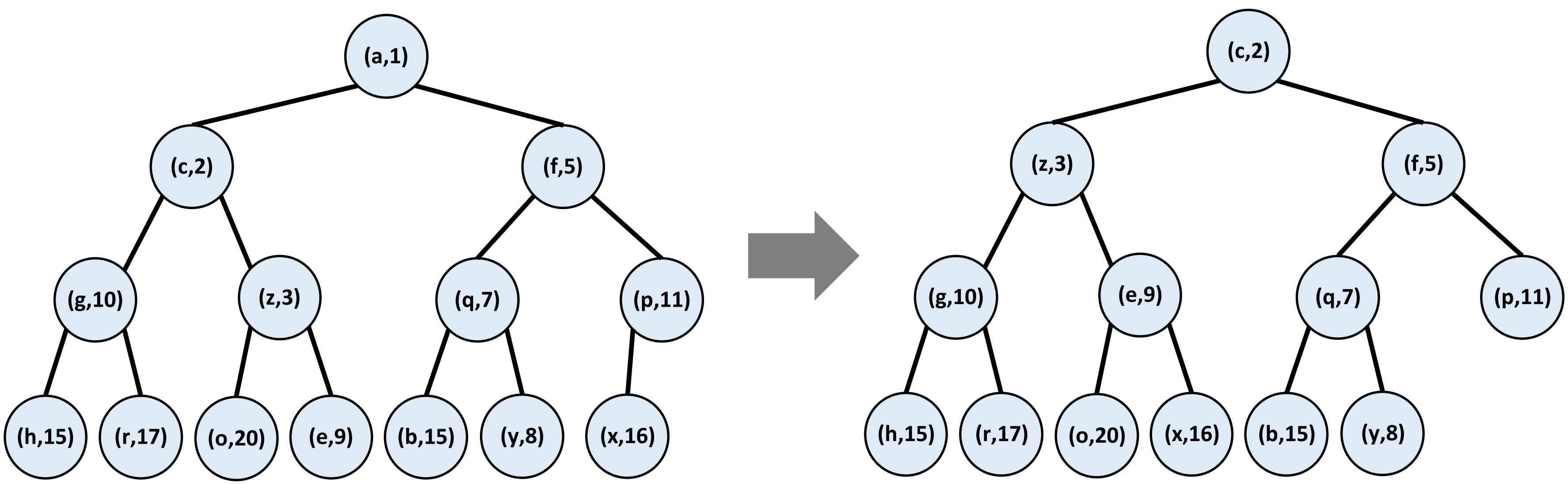
restore heap order after O(log n)



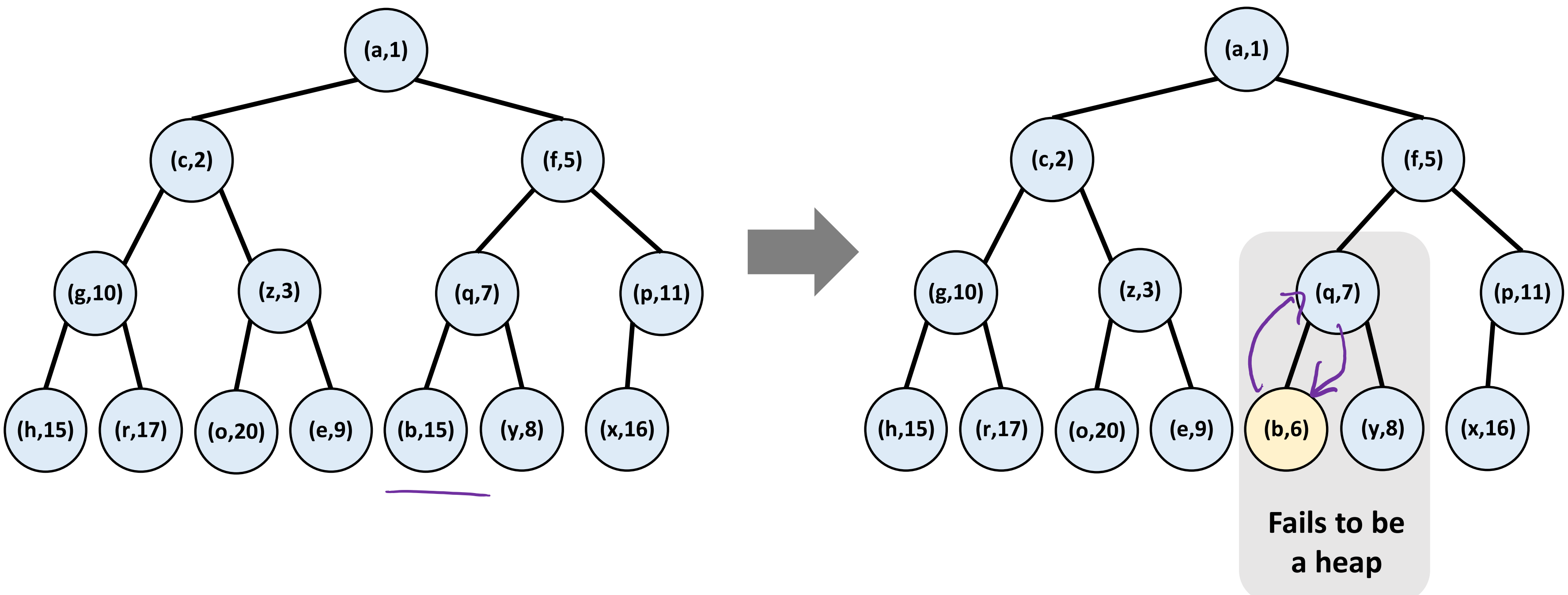
heap!

Implementing ExtractMin

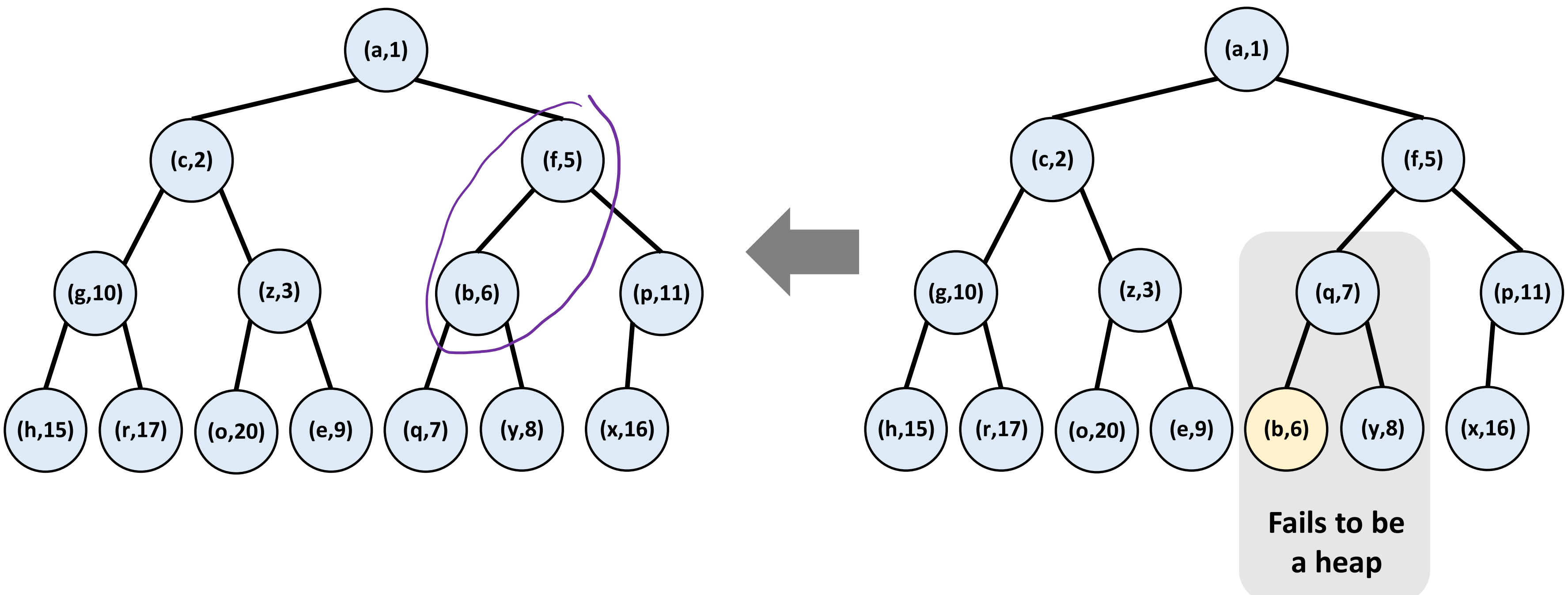
- Three steps:
  - Pull the minimum from the root $O(1)$
  - Move the last element to the root $O(1)$
  - Repair the heap-order (heapify down) $O(\log n)$



Implementing DecreaseKey



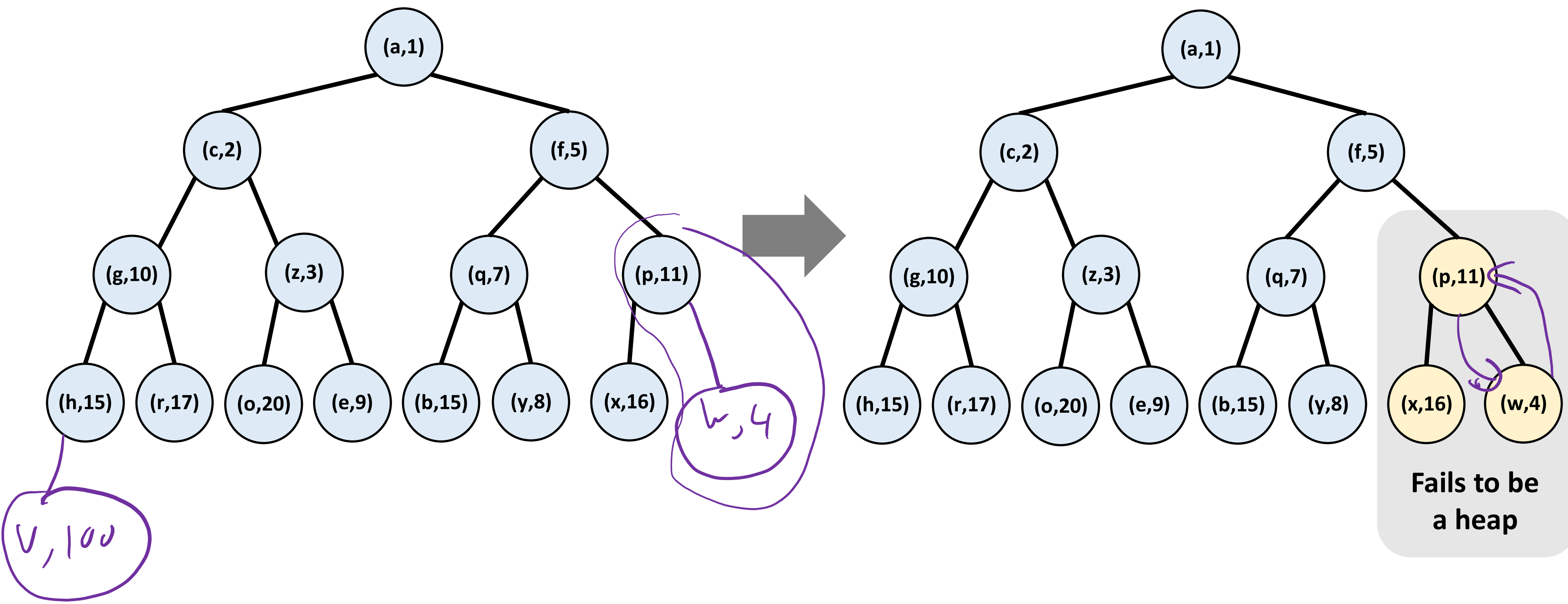
Implementing DecreaseKey



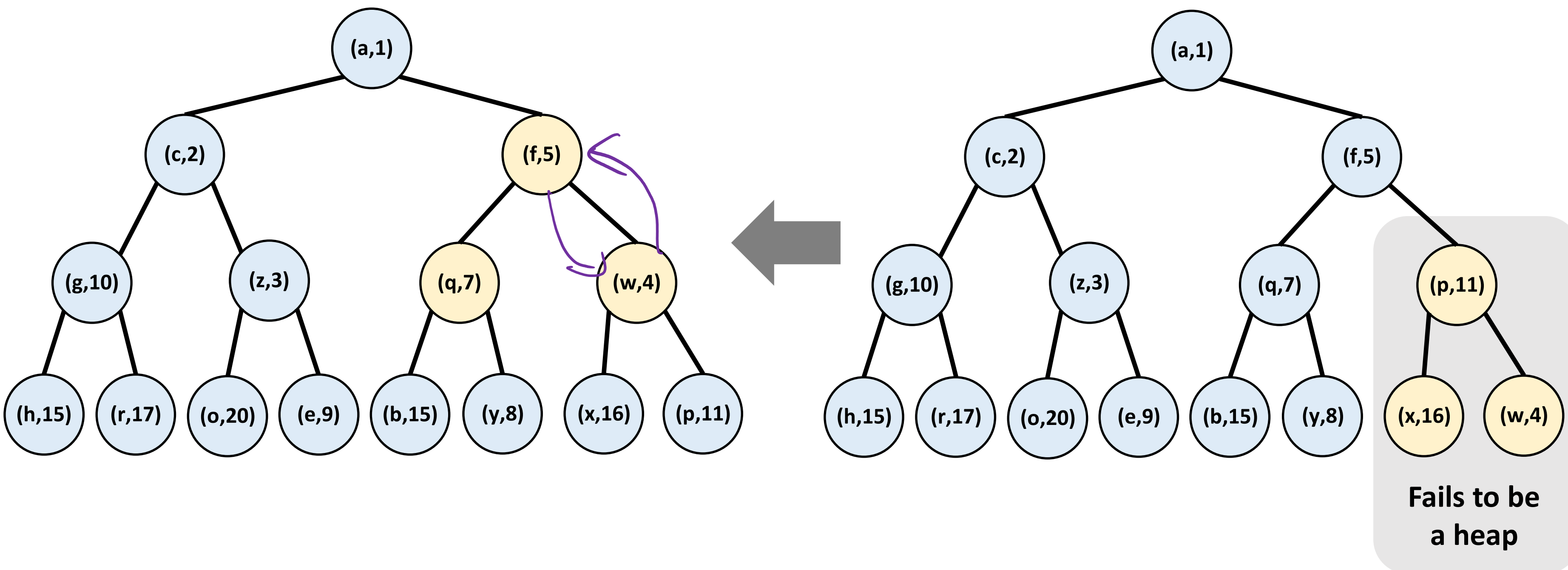
Implementing DecreaseKey

- Two steps:
  - Change the key $O(1)$
  - Repair the heap-order (heapify up) $O(\log n)$

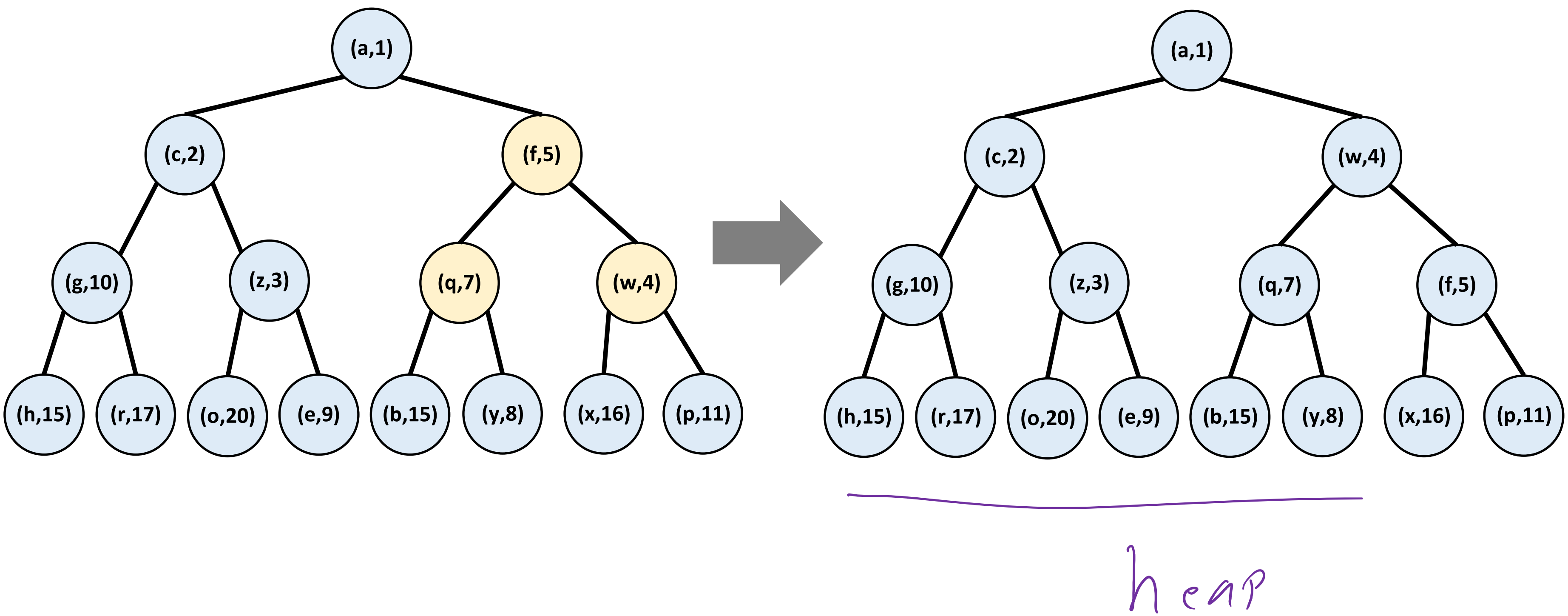
Implementing Insert



Implementing Insert



Implementing Insert



Implementing Insert

- Two steps:
  - Put the new key in the last location $O(1)$
  - Repair the heap-order (heapify up) $O(\log n)$

Implementation Using Arrays

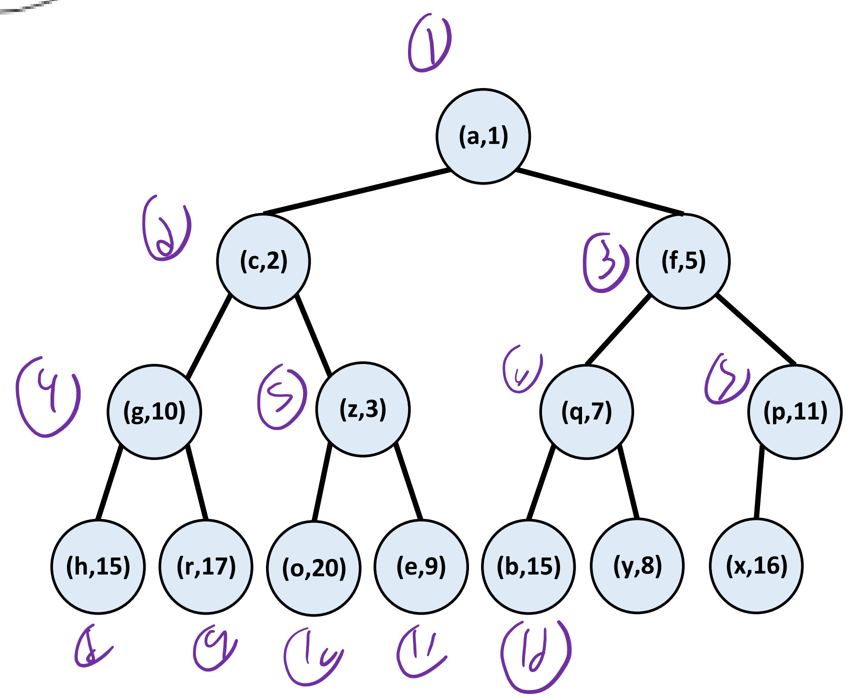
Array V

1	2	5	10	3	7	11	15	17	20	9	15	8	16	X

1 2 3

K

A : 1
B : 12
C : 2
⋮



- Maintain an array V holding the values
- Maintain a dictionary K mapping keys to values
  - Can find the value (**lookup**) for a given key in $O(1)$ time

Implementation Using Arrays

Array V

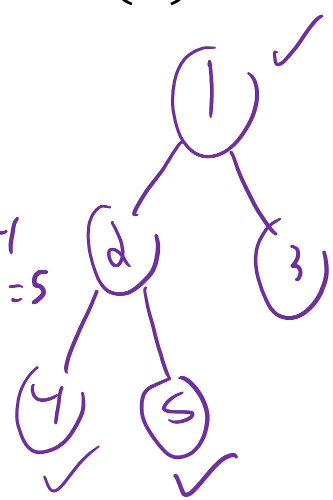
- Maintain an array V holding the values
- Maintain a dictionary K mapping keys to values
  - Can find the value for a given key in $O(1)$ time
- For any node i in the binary tree
  - LeftChild(i) = 2i
  - RightChild(i) = 2i+1
  - Parent(i) = ⌊i/2⌋

$i = 2$

$LC(2) = 2 \cdot 2 = 4$

$RC(2) = 2 \cdot 2 + 1 = 5$

$P(2) = \left\lfloor \frac{2}{2} \right\rfloor = 1$



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)

    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)

O(n)

n loops

each: O(log n)
Total: O(n log n)

O(out-deg(u))

each:
total: O(m)

each: O(1)
total: O(m)

each: O(log n)
total: O(m log n)

Total R/T:

$O(n \log n + m \log n)$

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!

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 $d(s,s)$

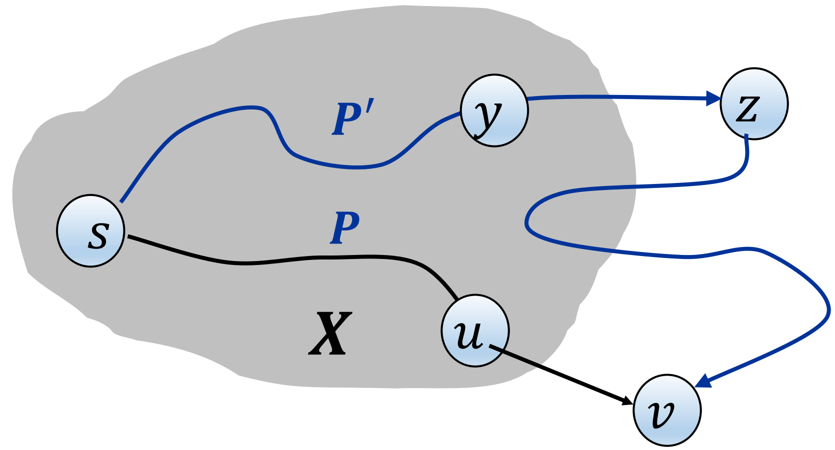
- **Warmup 1:** before we explore the second node v, $d_1(v)$ is the correct distance $d(s,v)$

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

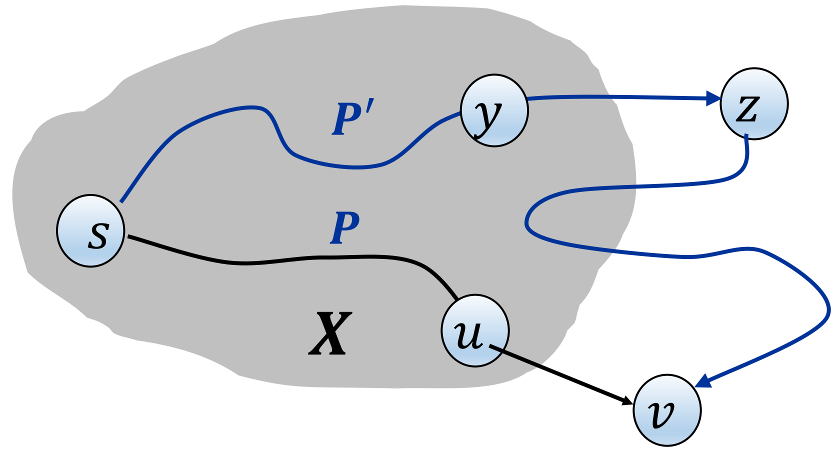
- We just argued the invariant holds before we've explored the 1^{st} and 2^{nd} nodes

Correctness of Dijkstra



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

Correctness of Dijkstra



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