CS3000: Algorithms & Data
Jonathan Ullman

Lecture 11:

- Shortest Paths: BFS, Start Dijkstra

Feb 24, 2020

Shortest Paths:
Breadth-First Search

Exploring a Graph

- **Problem:** Is there a path from s to t?
- **Idea:** Explore all nodes reachable from s.

- Two different search techniques:
  - **Depth-First Search:** follow a path until you get stuck, then go back
  - **Breadth-First Search:** explore all nearby nodes before moving on to farther away nodes
    - Finds the shortest path from s to t!

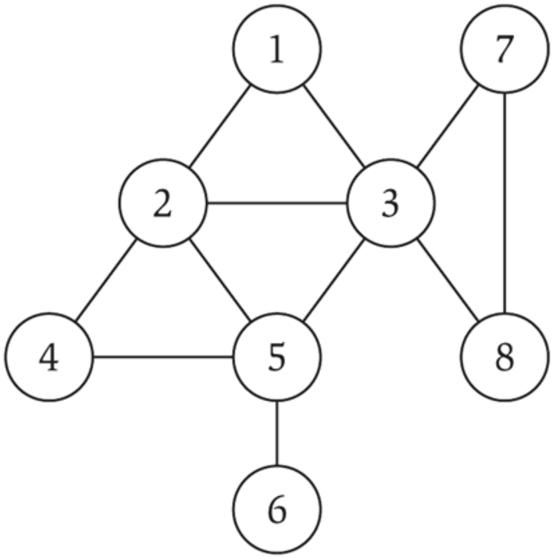
Breadth-First Search (BFS)

- **Informal Description:** start at s, find neighbors of s, find neighbors of neighbors of s, and so on…

- BFS Tree:
  - $L_0 = \{s\}$
  - $L_1 =$ all neighbors of L_0
  - $L_2 =$ all neighbors of L_1 that are not in L_0, L_1
  - $L_3 =$ all neighbors of L_2 that are not in L_0, L_1, L_2
  - …
  - $L_d =$ all neighbors of L_{d-1} that are not in $L_0, \ldots, L_{d-1}$
  - Stop when L_{d+1} is empty

Example

- BFS this graph from $\boldsymbol{s = 1}$



Breadth-First Search Implementation

```
BFS(G = (V,E), s):
  Let explored[v] ← false ∀v, explored[s] ← true
  Let layer[v] ← ∞ ∀v, layer[s] ← 0
  Let parent[v] ←⊥ ∀v
  Let i ← 0, L_0 = {s}, T ← ∅

  While (L_i is not empty):
    Initialize new layer L_{i+1}
    For (u in L_i):
      For ((u,v) in E):
        If (explored[v] = false):
          explored[v] ← true,
          layer[v] ← i+1
          parent[v] ← u
          Add v to L_{i+1}
    i ← i+1
```

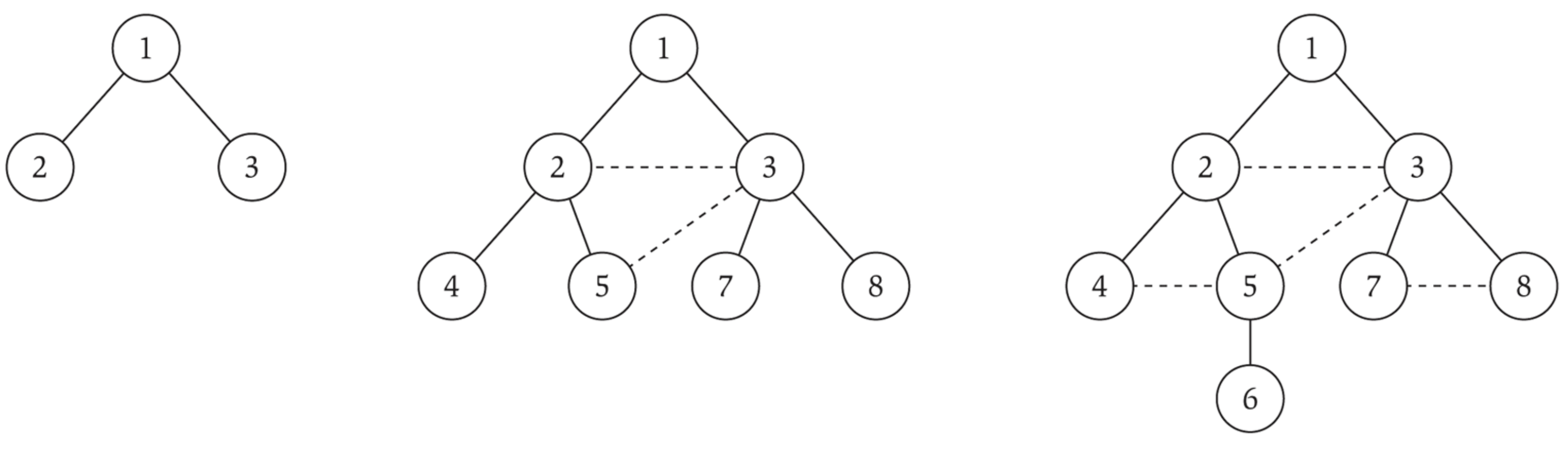
BFS Running Time (Adjacency List)

```
BFS(G = (V,E), s):
  Let explored[v] ← false ∀v, explored[s] ← true
  Let layer[v] ← ∞ ∀v, layer[s] ← 0
  Let parent[v] ←⊥ ∀v
  Let i ← 0, L0 = {s}, T ← ∅

  While (Li is not empty):
    Initialize new layer Li+1
    For (u in Li):
      For ((u,v) in E):
        If (explored[v] = false):
          explored[v] ← true,
          layer[v] ← i+1
          parent[v] ← u
          Add v to Li+1
    i ← i+1
```

Shortest Paths via BFS

- **Definition:** the distance between s, t is the number of edges on the shortest path from s to t
- **Thm:** BFS finds distances from s to other nodes
  - L_i contains all nodes at distance i from s

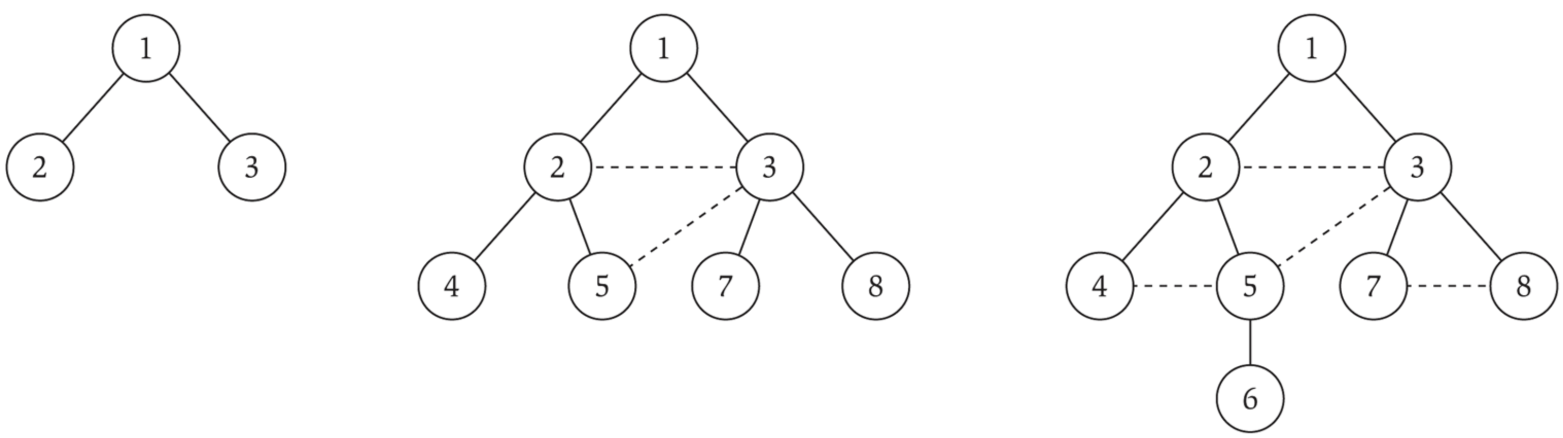


Shortest Paths via BFS

- **Definition:** the distance between s, t is the number of edges on the shortest path from s to t
- **Thm:** BFS finds distances from s to other nodes
  - L_i contains all nodes at distance i from s

Shortest Paths via BFS

- **Definition:** the distance between s, t is the number of edges on the shortest path from s to t
- **Thm:** BFS finds distances from s to other nodes and the tree edges give the shortest s to t path
  - Can find distances and shortest path tree in time $O(n+m)$… then can find a shortest path in time $O(n)$

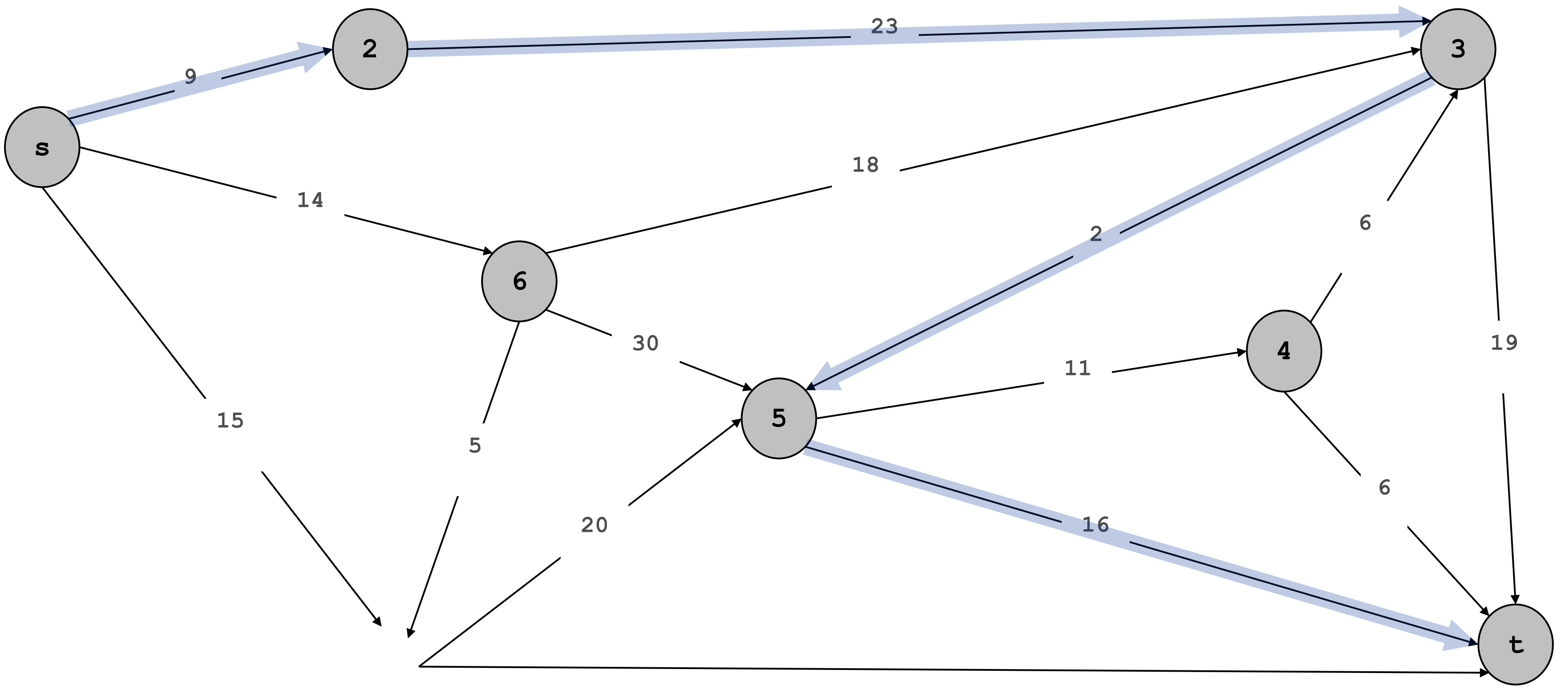


Shortest Paths via BFS

- **Definition:** the distance between s, t is the number of edges on the shortest path from s to t
- **Thm:** BFS finds distances from s to other nodes and the tree edges give the shortest s to t path
  - Can find distances and shortest path tree in time $O(n + m)$… then can find a shortest path in time $O(n)$

Shortest Paths: Dijkstra

Navigation



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/lengths/capacities
  - Can be directed or undirected

- **Today:**
  - Directed graphs (one-way streets)
  - Strongly connected (there is always some path)
  - Non-negative edge lengths ($\ell(e) \geq 0$)

Shortest Paths

- The length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge lengths

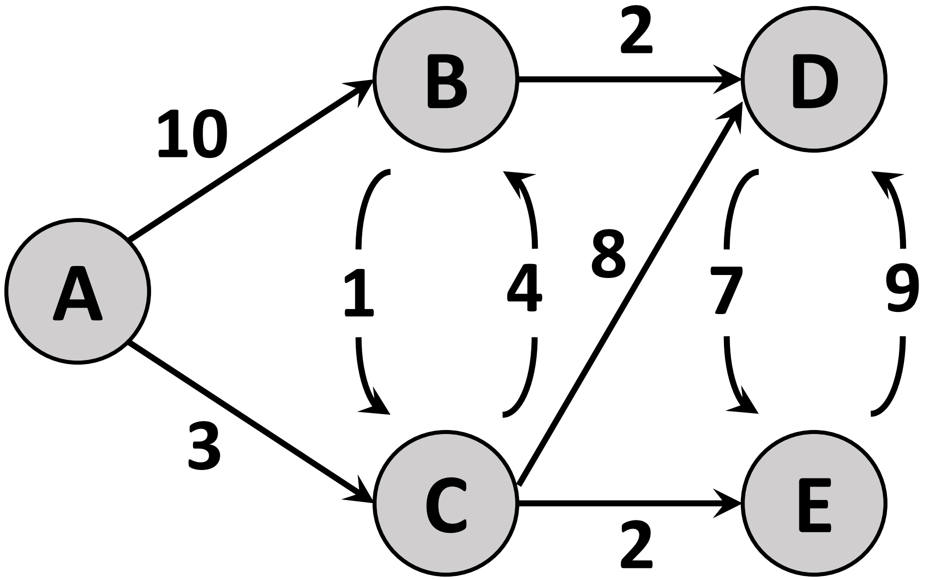
- The distance $d(s, t)$ is the length of the shortest path from s to t
- **Shortest Path:** given nodes $s, t \in V$, find the shortest path from s to t
- **Single-Source Shortest Paths:** given a node $s \in V$, find the shortest paths from s to **every** $t \in V$

Structure of Shortest Paths

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

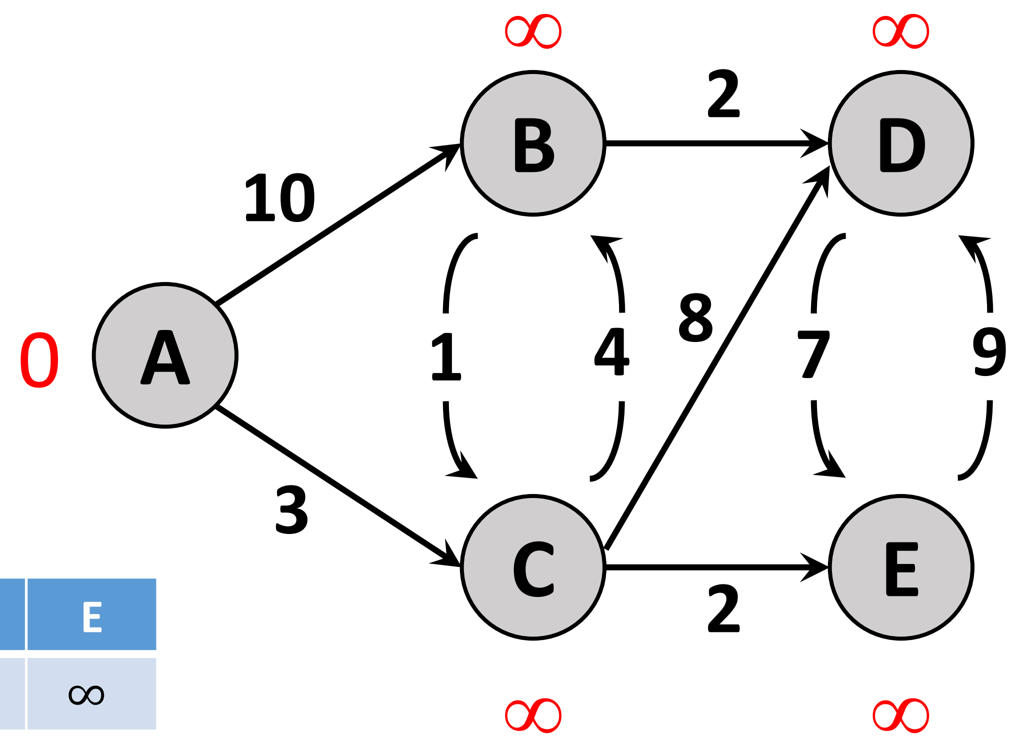
- If $(u, v) \in E$, and $d(s, v) = d(s, u) + \ell(u, v)$ then there is a shortest $s \rightsquigarrow v$-path ending with (u, v)

Dijkstra’s Algorithm: Demo



Dijkstra's Algorithm: Demo

Initialize

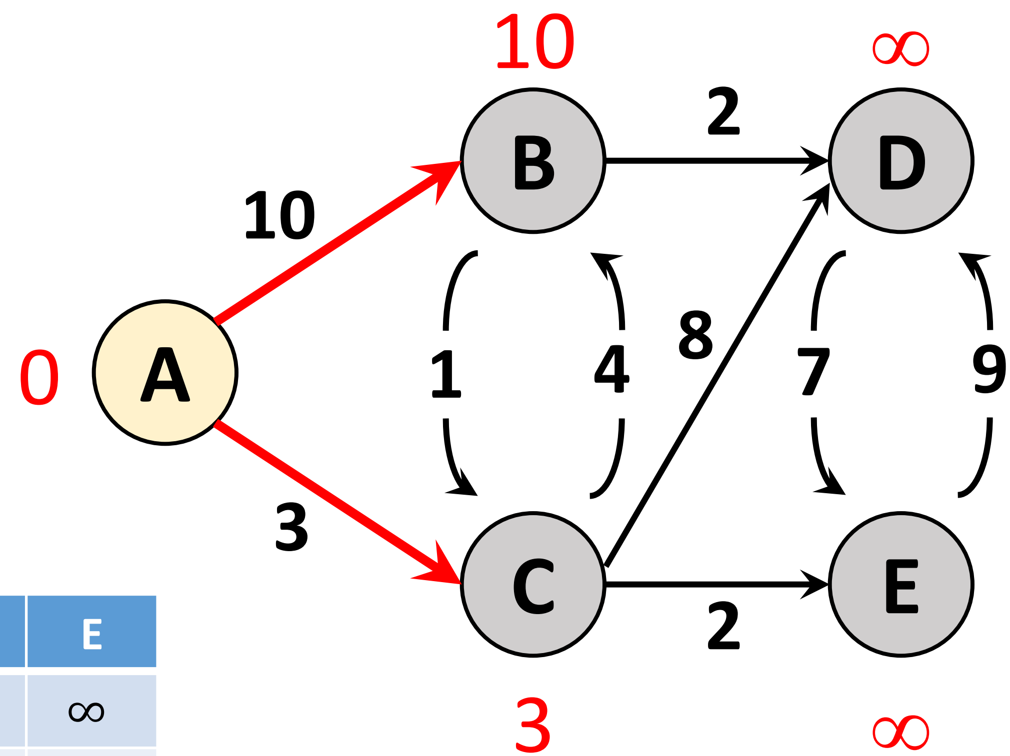


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

$$S = \{\}$$

Dijkstra's Algorithm: Demo

Explore A

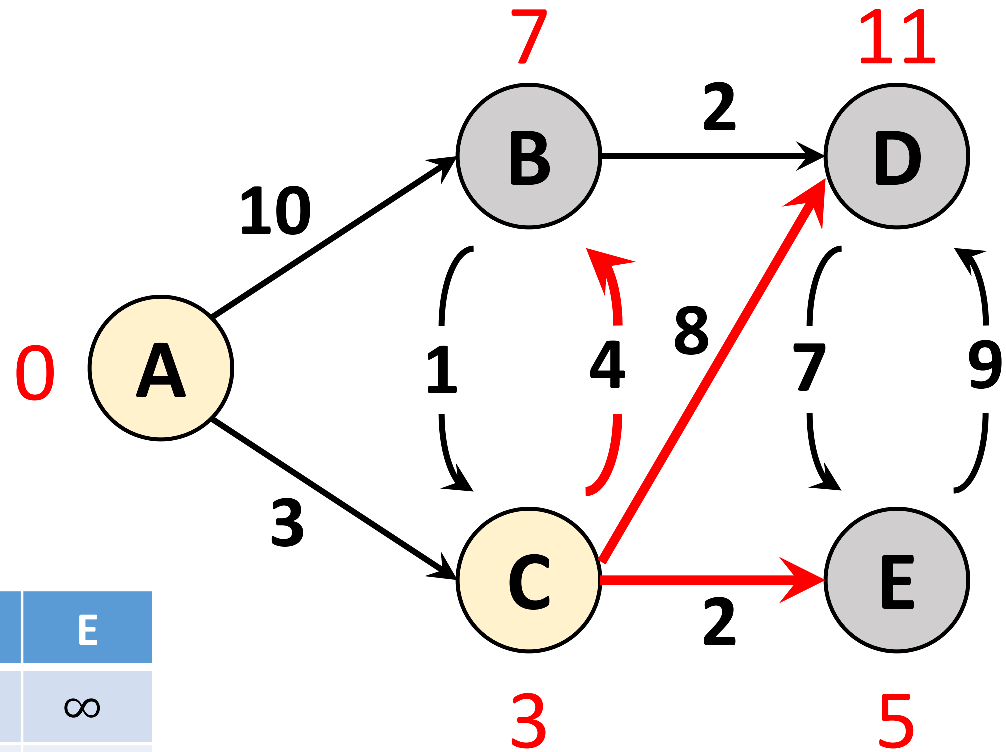


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞

$$S = \{A\}$$

Dijkstra's Algorithm: Demo

Explore C

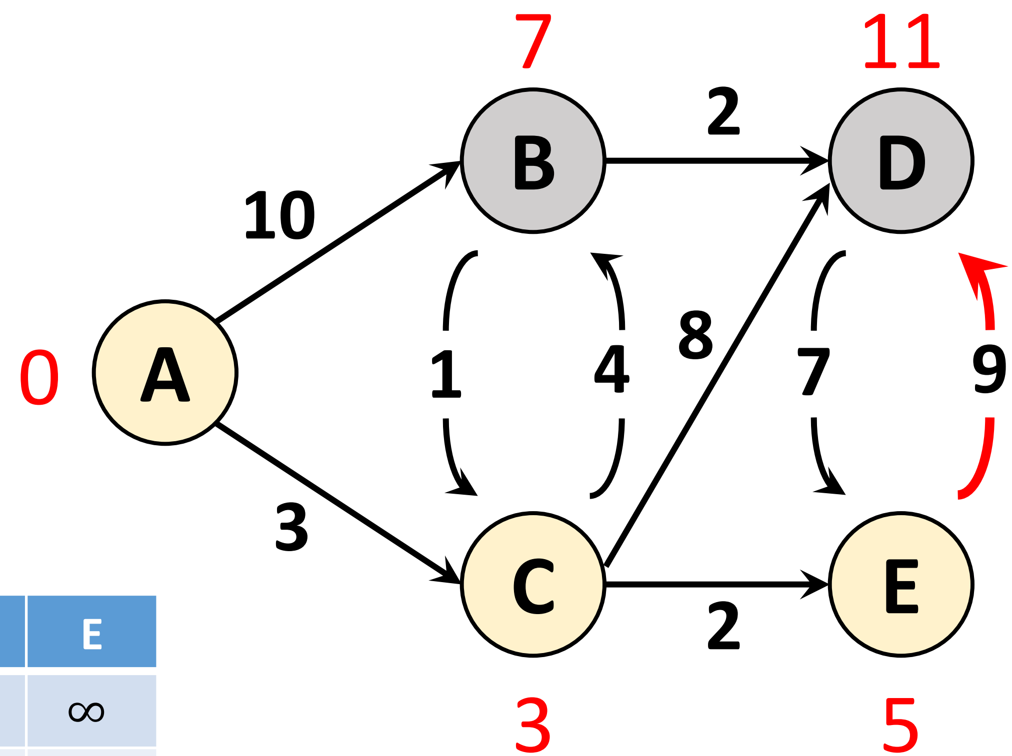


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞
$d_2(u)$	**0**	**7**	**3**	**11**	**5**

$$S = \{A, C\}$$

Dijkstra's Algorithm: Demo

Explore E

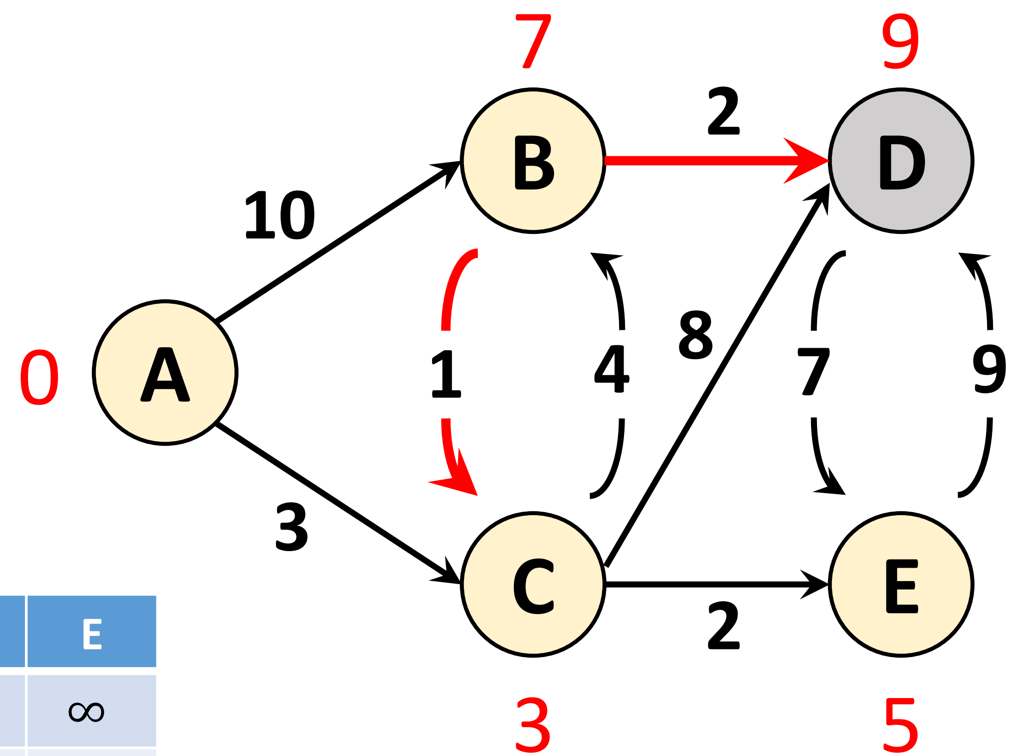


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5

$$S = \{A, C, E\}$$

Dijkstra’s Algorithm: Demo

Explore B

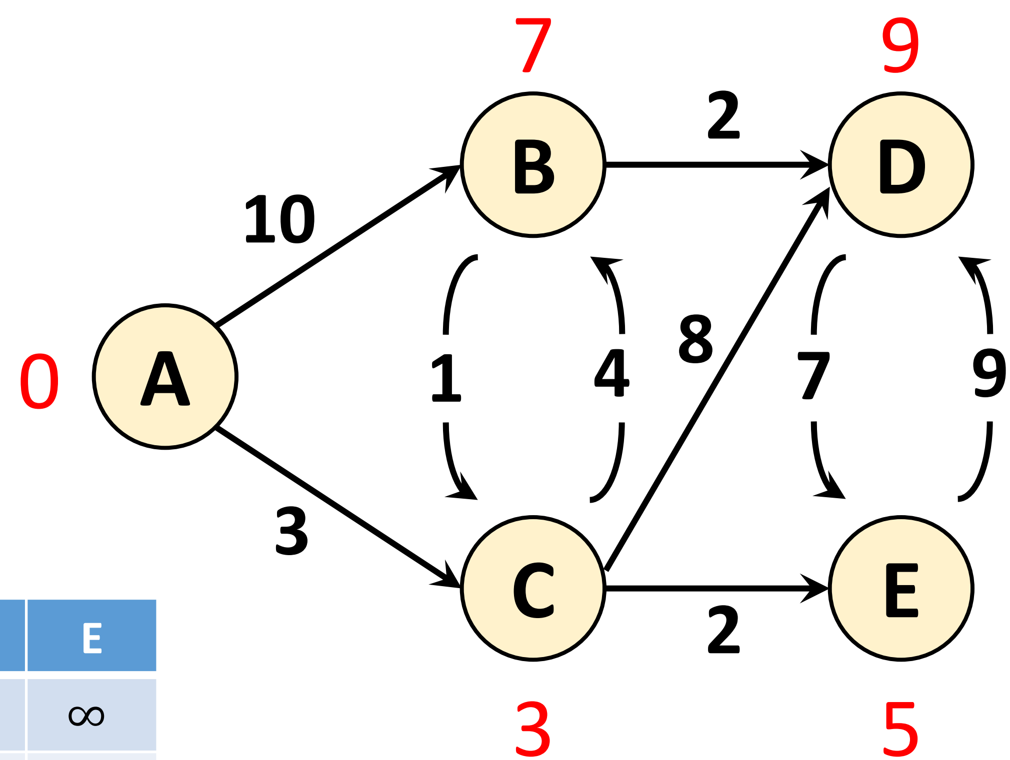


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$S = \{A, C, E, B\}$$

Dijkstra's Algorithm: Demo

Don't need to explore D

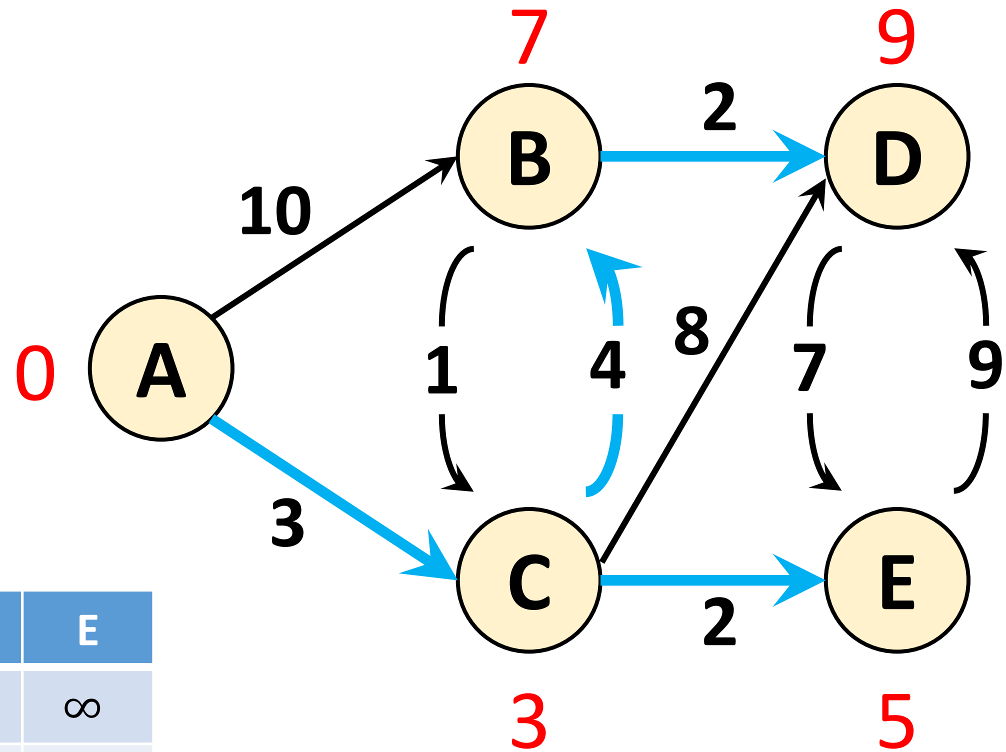


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$S = \{A, C, E, B, D\}$$

Dijkstra's Algorithm: Demo

Maintain parent pointers so we can find the shortest paths



	A	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**

Correctness of Dijkstra

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

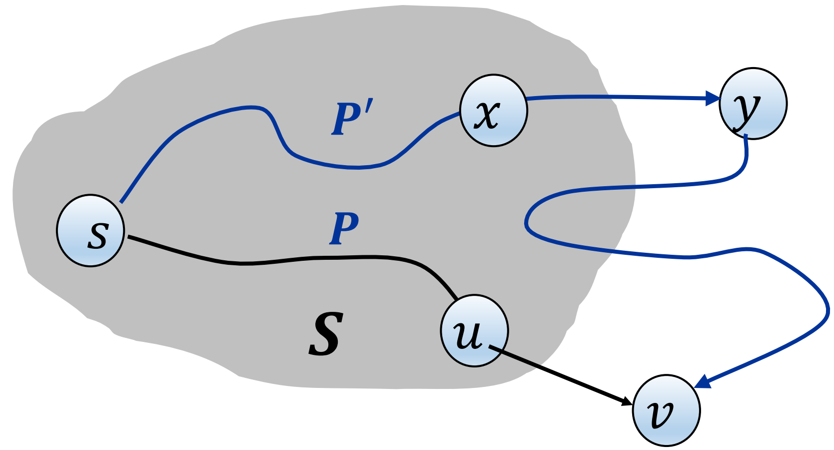
- **Warmup 1:** after exploring the first node v, $d_1(v)$ is the correct distance

Correctness of Dijkstra

- **Invariant:** after we explore the i-th node, $d_i(v)$ is correct for every $v \in S$

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

Correctness of Dijkstra



- **Invariant:** after we explore the i-th node, $d_i(v)$ is correct for every $v \in S$
- **Proof:**

Implementing Dijkstra

```
Dijkstra(G = (V,E,{ℓ(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 ((u,v) in E):
      If (d[v] > d[u] + ℓ(u,v)):
        d[v] ←d[u] + ℓ(u,v)
        parent[v] ←u

  Return (d, parent)
```

Implementing Dijkstra (Naïvely)

- Need to explore all n nodes
- Each exploration requires:
  - Finding the unexplored node u with smallest distance
  - Updating the distance for each neighbor of u

Priority Queues / Heaps

Priority Queues

- Need a data structure Q to hold key-value pairs

- 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

- **Naïve approach:** linked lists

Key	a	c	e	h	b	g	k	d	f
Value	11	12	2	36	4	20	42	10	8

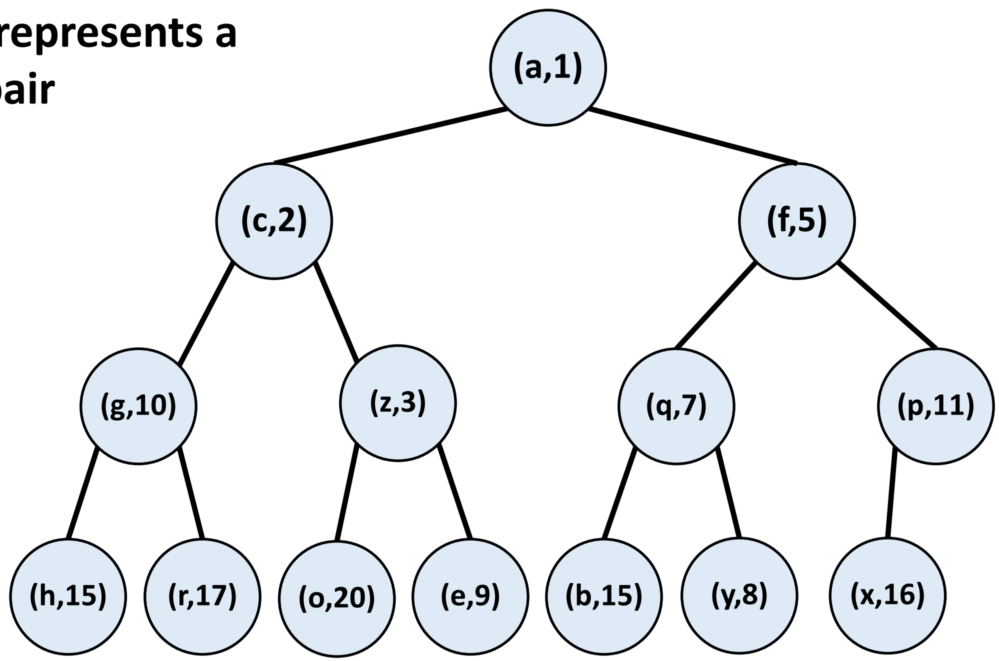
  - Insert takes O(1) time
  - ExtractMin, DecreaseKey take O(n) time

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

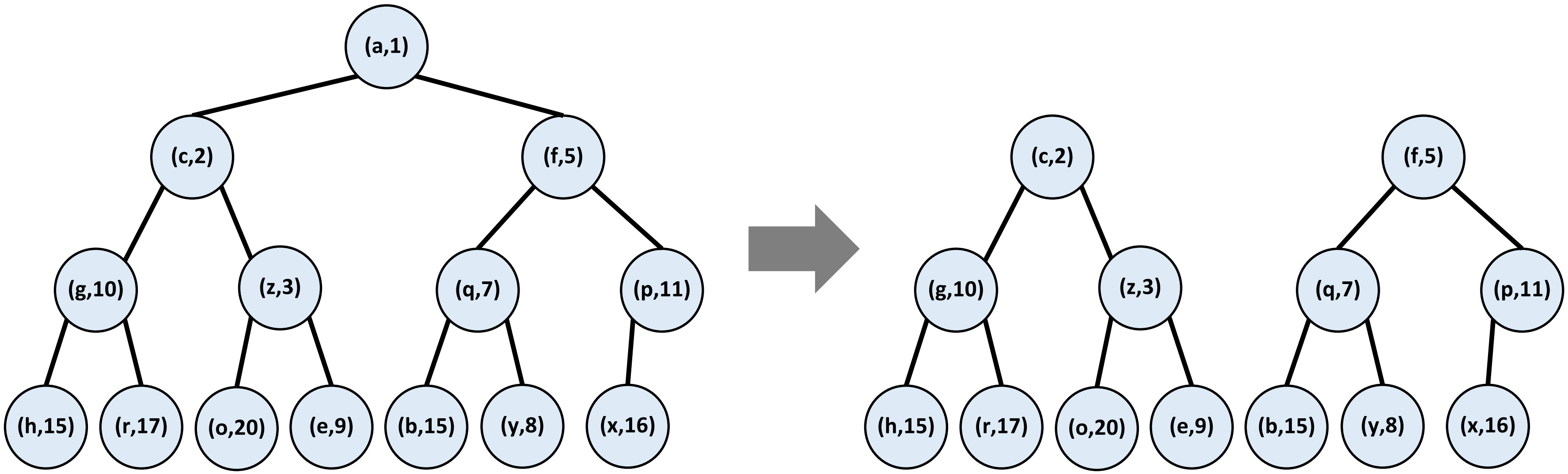
Heaps

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

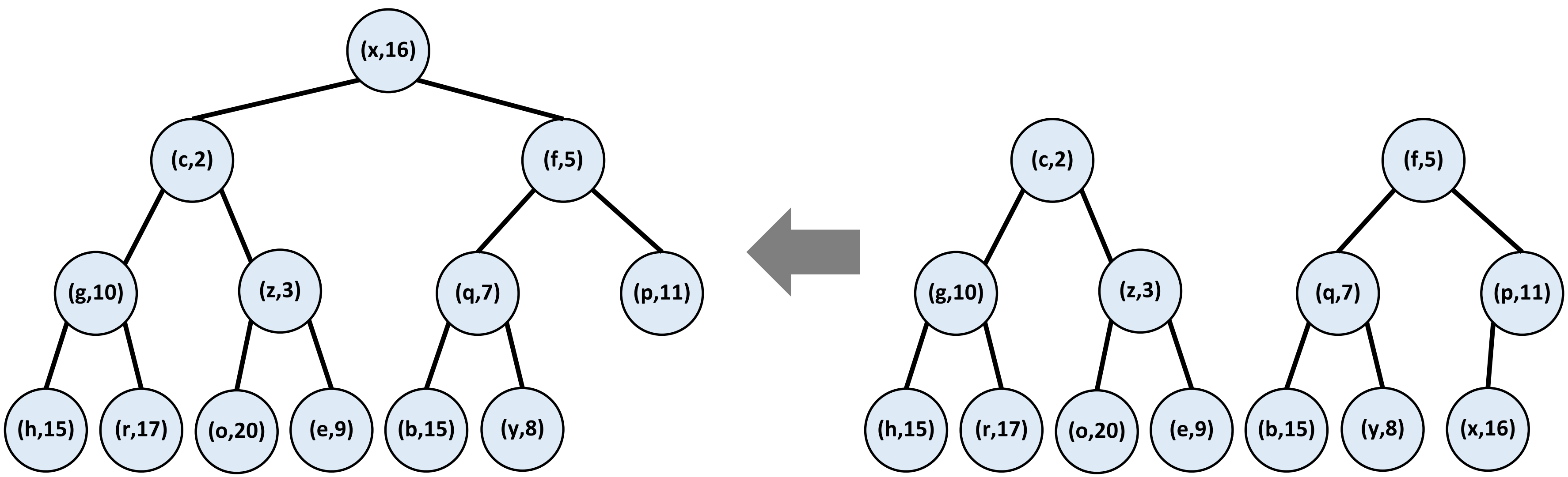
Each node represents a key-value pair



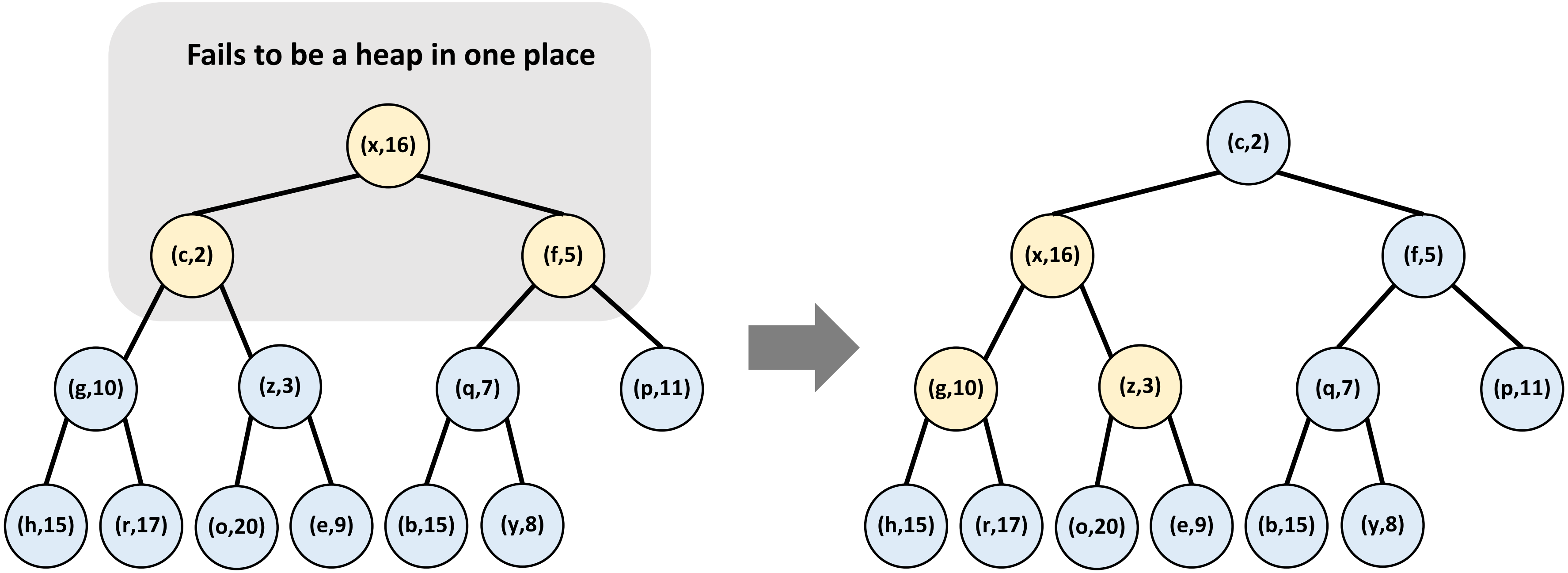
Implementing ExtractMin



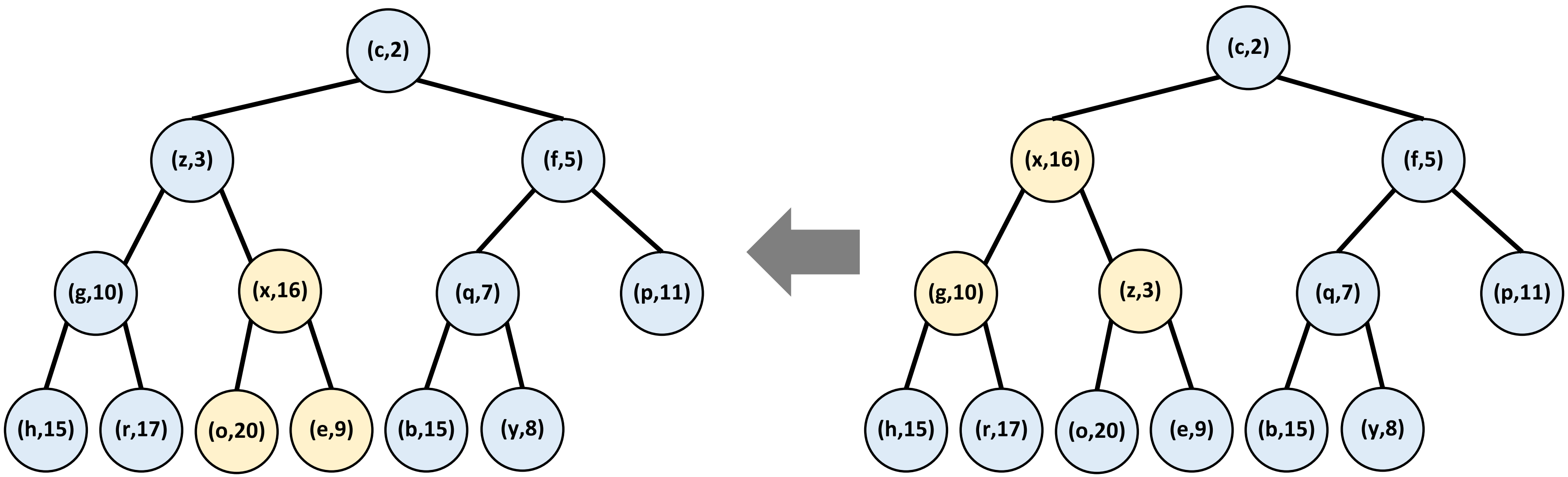
Implementing ExtractMin



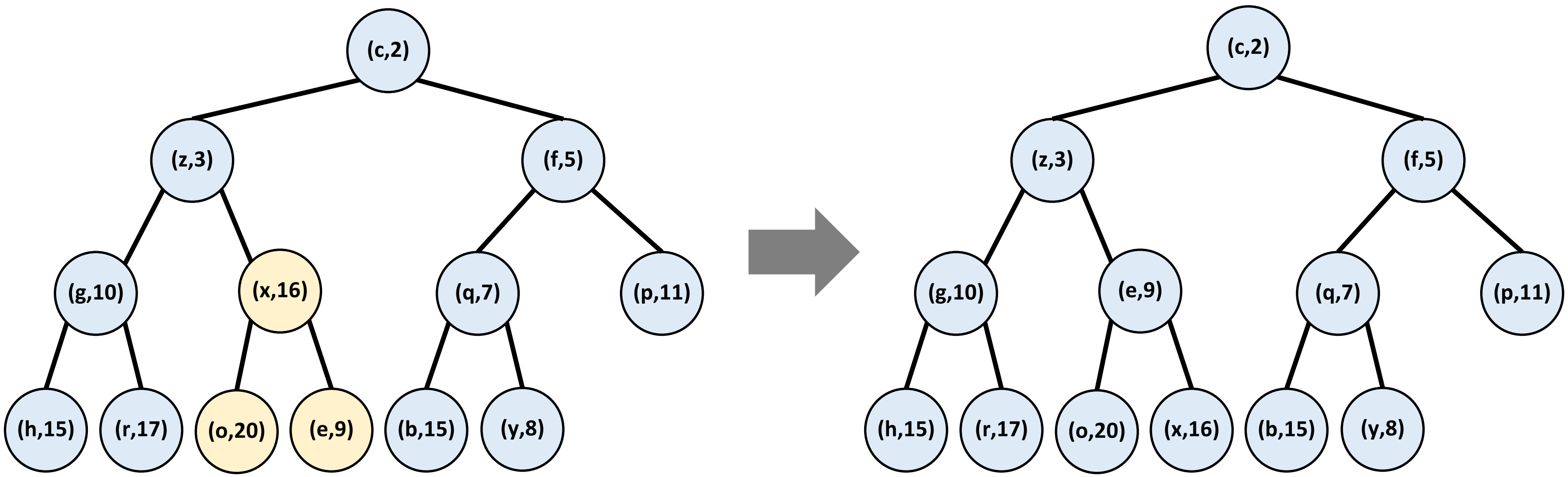
Implementing ExtractMin



Implementing ExtractMin

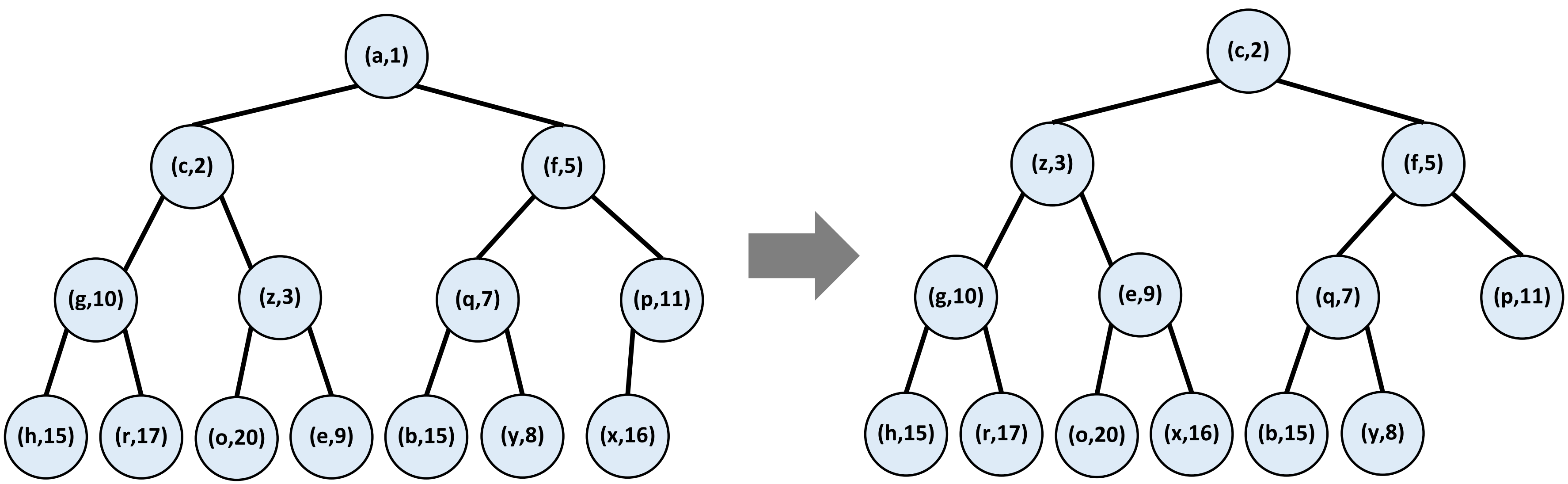


Implementing ExtractMin

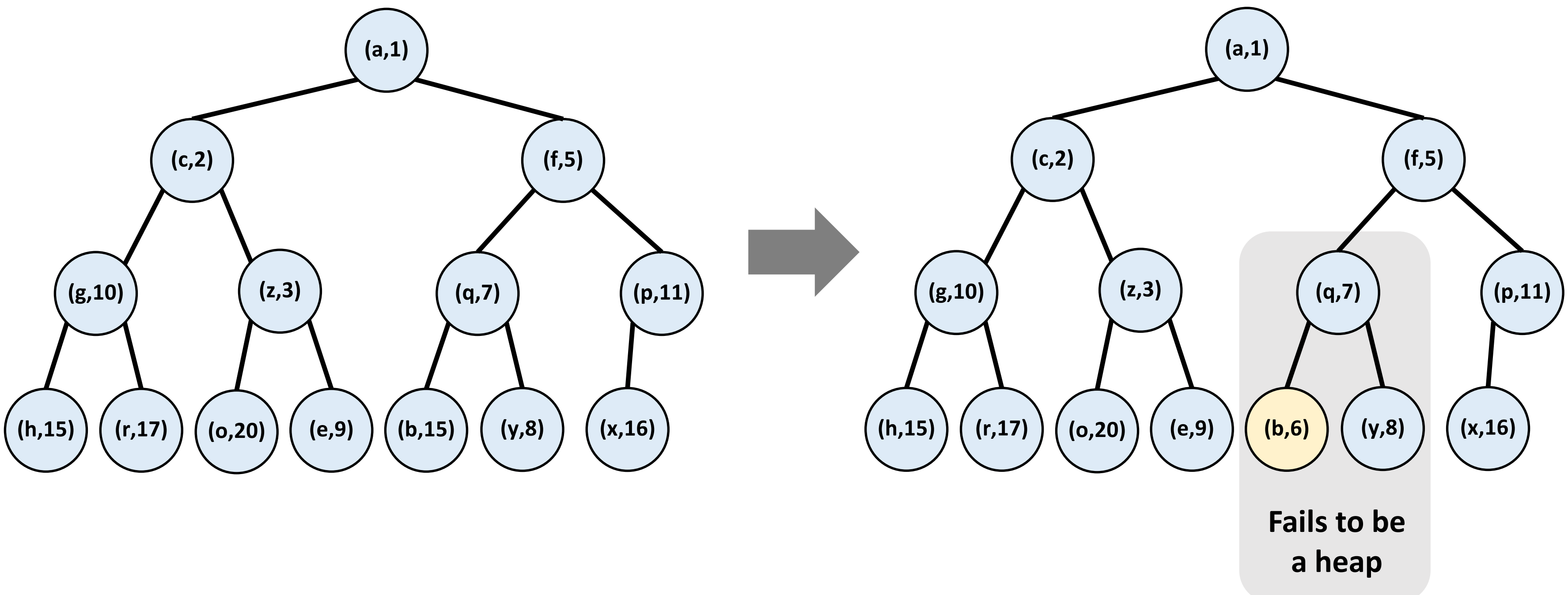


Implementing ExtractMin

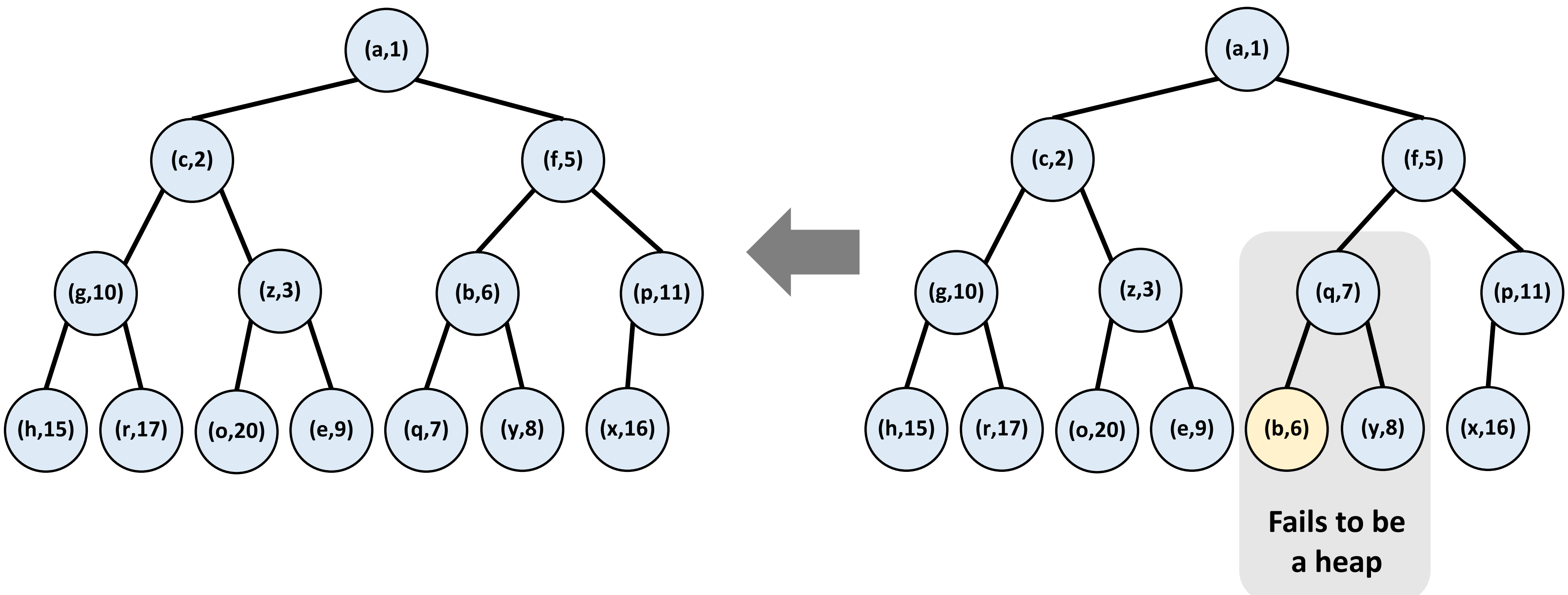
- Three steps:
  - Pull the minimum from the root
  - Move the last element to the root
  - Repair the heap-order (heapify down)



Implementing DecreaseKey



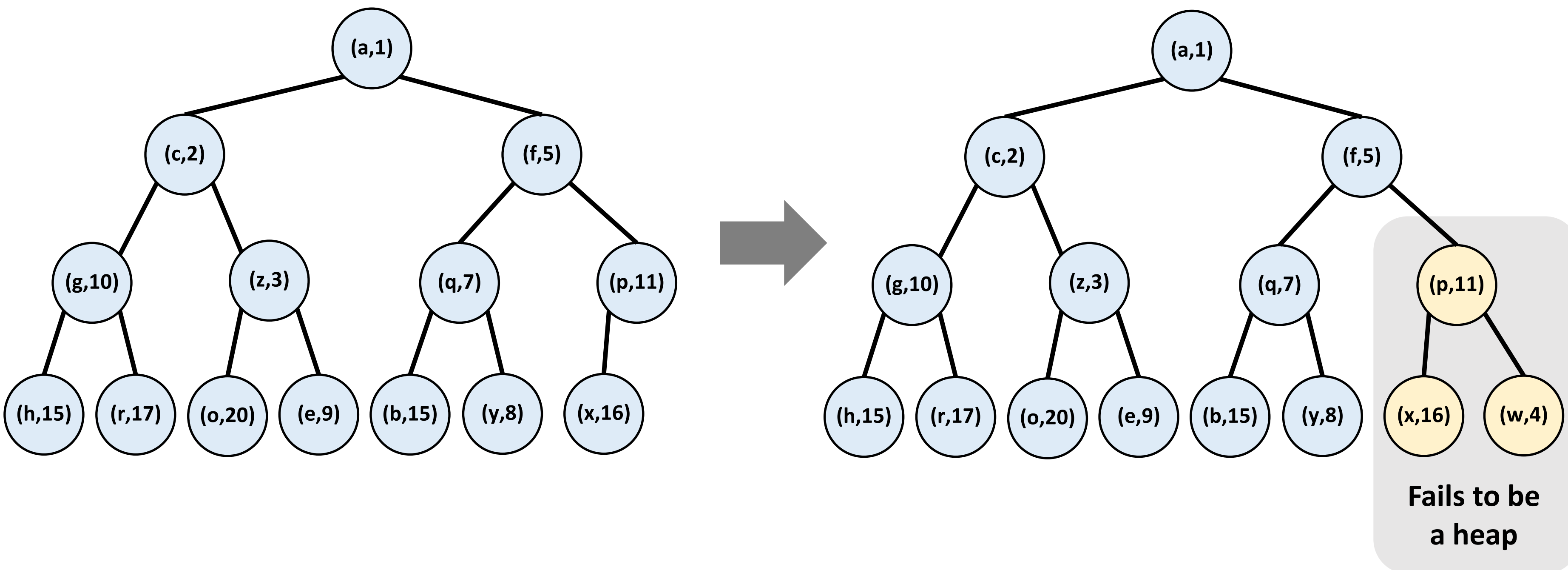
Implementing DecreaseKey



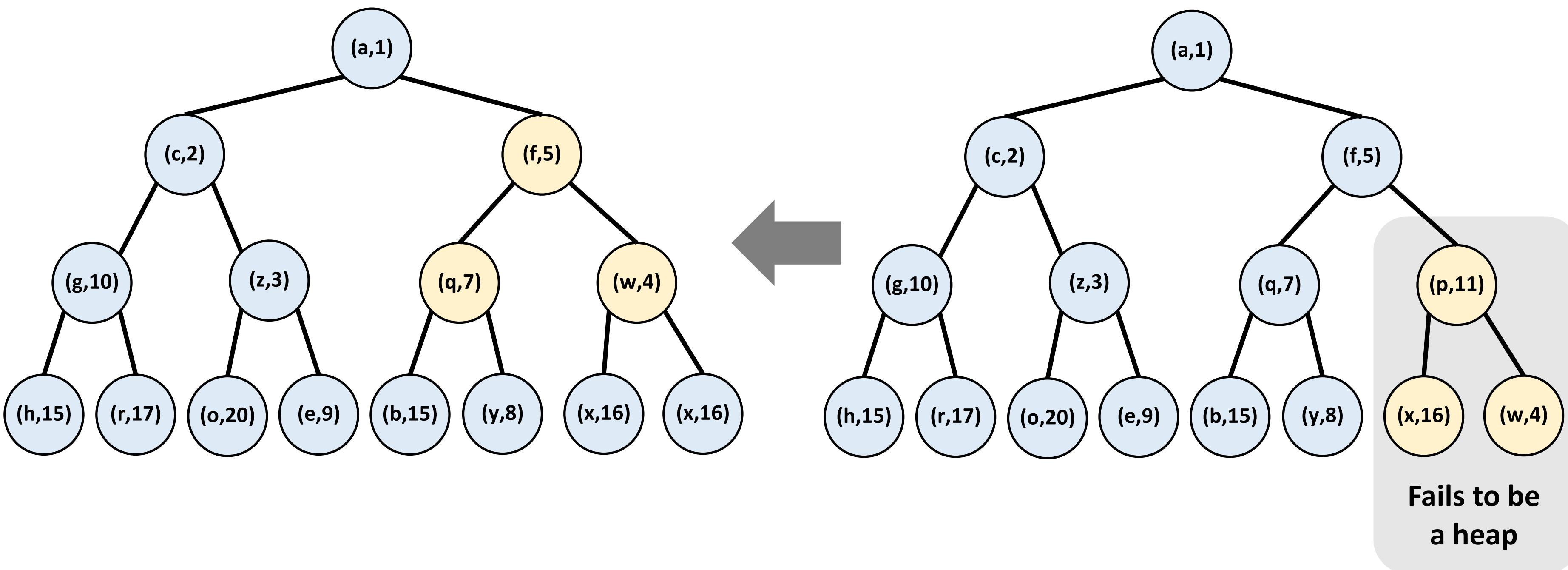
Implementing DecreaseKey

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

Implementing Insert



Implementing Insert



Implementing Insert

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

Implementation Using Arrays

Array V

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

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 ((u,v) in E):
      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)
```

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: $O(m \log n)$

- **Compare to BFS**