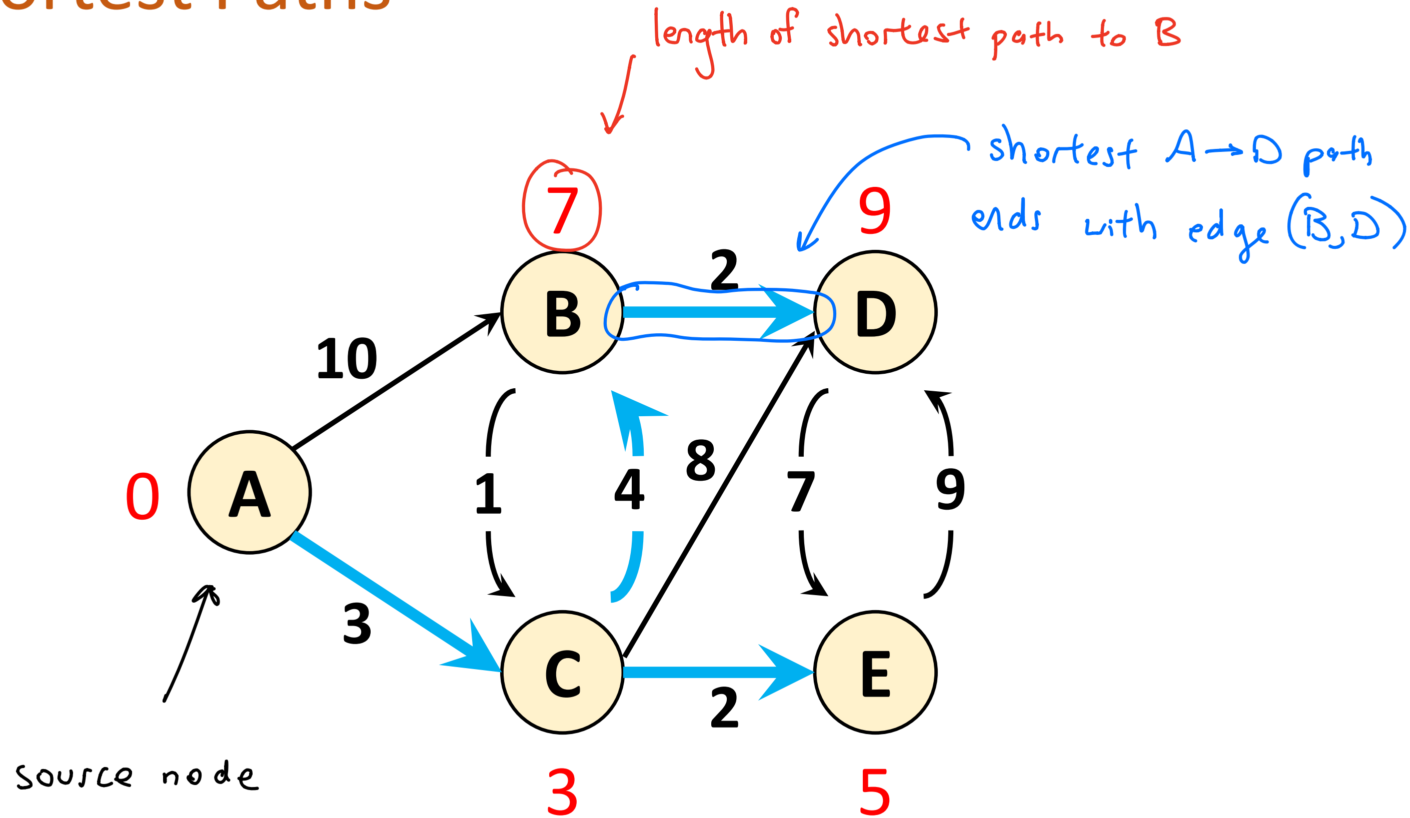
# CS3000: Algorithms & Data
# Jonathan Ullman

Lecture 14:
- Finish Dijkstra's Algorithm
- Bellman-Ford

Oct 26, 2018

Announcements
- HW5 due tonight
- HW6 out now, due 11/2

# Shortest Paths



# 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

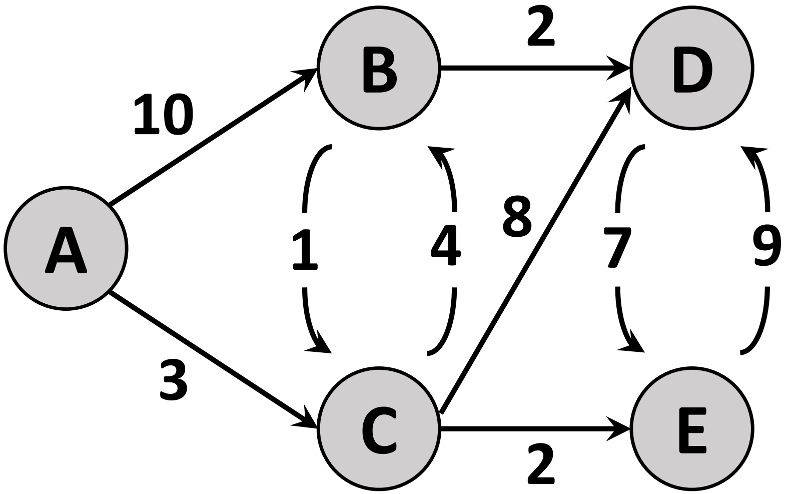
- The length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge lengths

- 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** $t \in V$

# Dijkstra's Algorithm

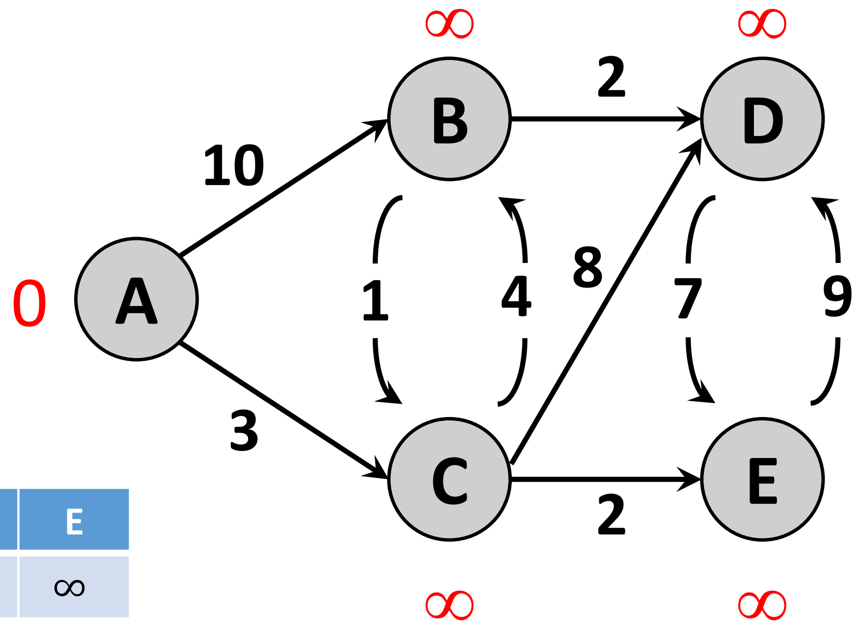
- **Dijkstra's Shortest Path Algorithm** is a modification of BFS for non-negatively weighted graphs
- **Informal Version:**
  - **Maintain a set $S$ of explored nodes** (Initially empty)
  - **Maintain an upper bound on distance** (Initially $d(s)=0$, $d(u)=\infty$)
    - If $u$ is explored, then we know $d(u)$ (Key Invariant)
    - If $u$ is explored, and $(u, v)$ is an edge, then we know $d(v) \leq d(u) + \ell(u, v)$
  - **Explore the "closest" unexplored node** ] Maintains invariant
  - **Repeat until we're done**

# Dijkstra’s Algorithm: Demo



# Dijkstra's Algorithm: Demo

**Initialize**

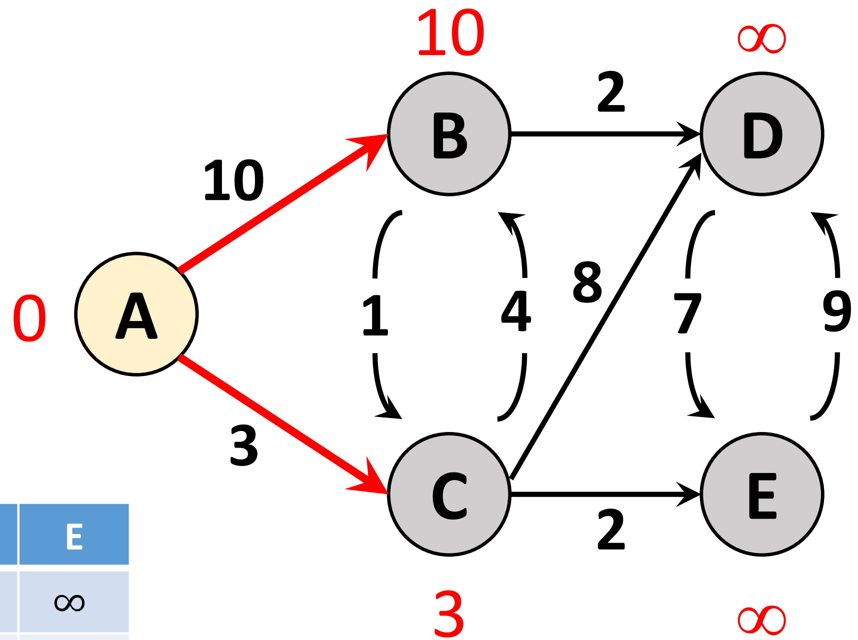



| | A | B | C | D | E |
|---|---|---|---|---|---|
| $d_0(u)$ | **0** | ∞ | ∞ | ∞ | ∞ |

$$S = \{\}$$

# Dijkstra's Algorithm: Demo

**Explore A**

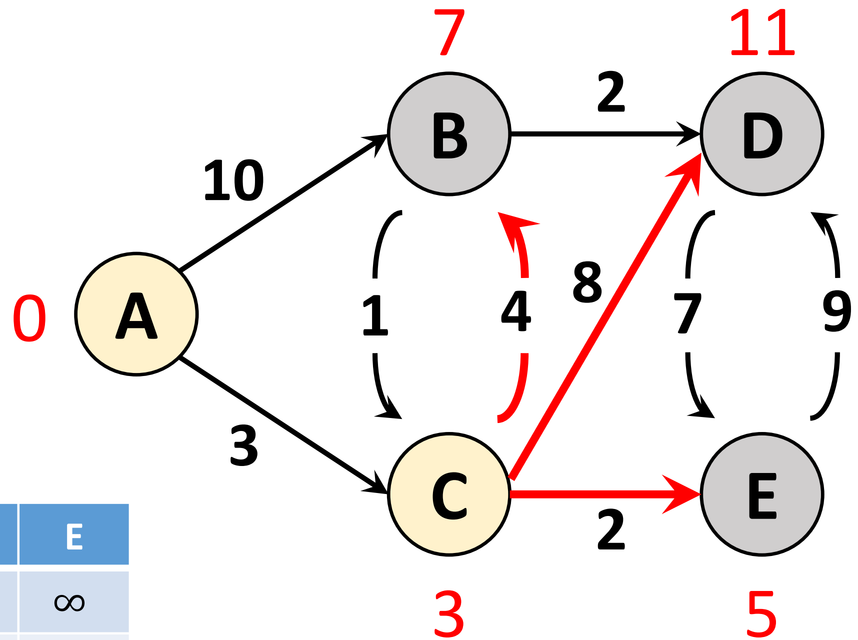



| | A | B | C | D | E |
|---|---|---|---|---|---|
| $d_0(u)$ | **0** | ∞ | ∞ | ∞ | ∞ |
| $d_1(u)$ | **0** | **10** | **3** | ∞ | ∞ |

$$S = \{A\}$$

# Dijkstra's Algorithm: Demo

**Explore C**

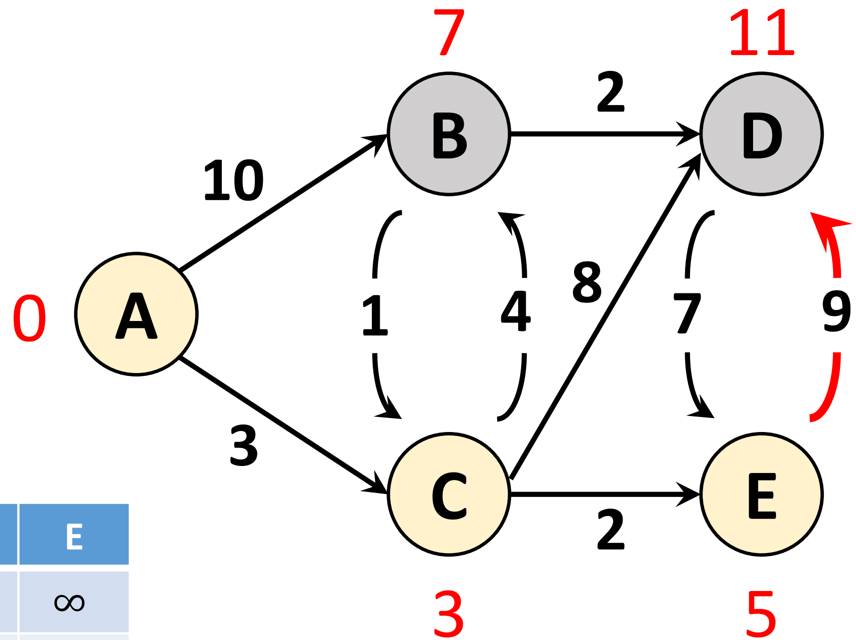



| | A | B | C | D | E |
|---|---|---|---|---|---|
| $d_0(u)$ | 0 | ∞ | ∞ | ∞ | ∞ |
| $d_1(u)$ | 0 | 10 | 3 | ∞ | ∞ |
| $d_2(u)$ | 0 | 7 | 3 | 11 | 5 |

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

# Dijkstra's Algorithm: Demo

**Explore E**

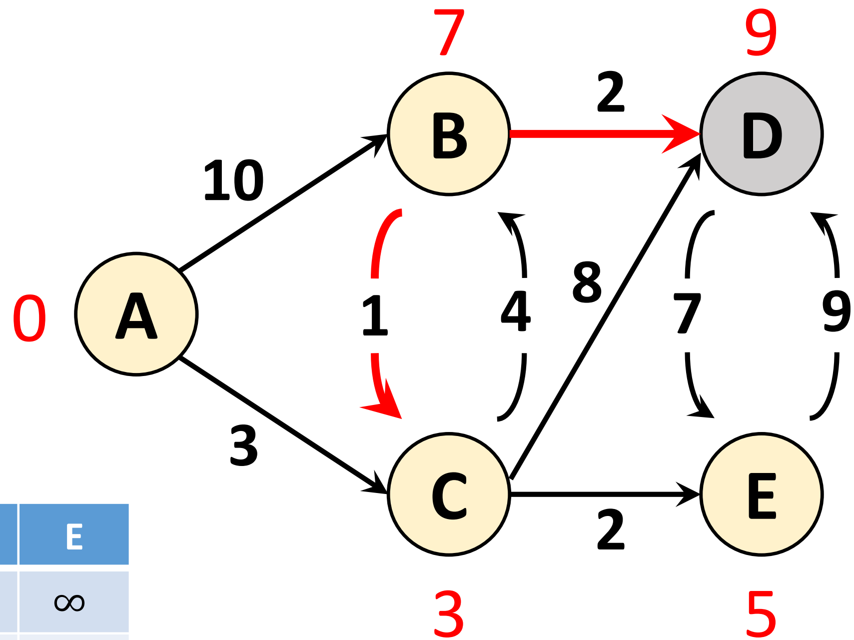



| | A | B | C | D | E |
|---|---|---|---|---|---|
| $d_0(u)$ | 0 | ∞ | ∞ | ∞ | ∞ |
| $d_1(u)$ | 0 | 10 | 3 | ∞ | ∞ |
| $d_2(u)$ | 0 | 7 | 3 | 11 | 5 |
| $d_3(u)$ | 0 | 7 | 3 | 11 | 5 |

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

# Dijkstra's Algorithm: Demo

**Explore B**

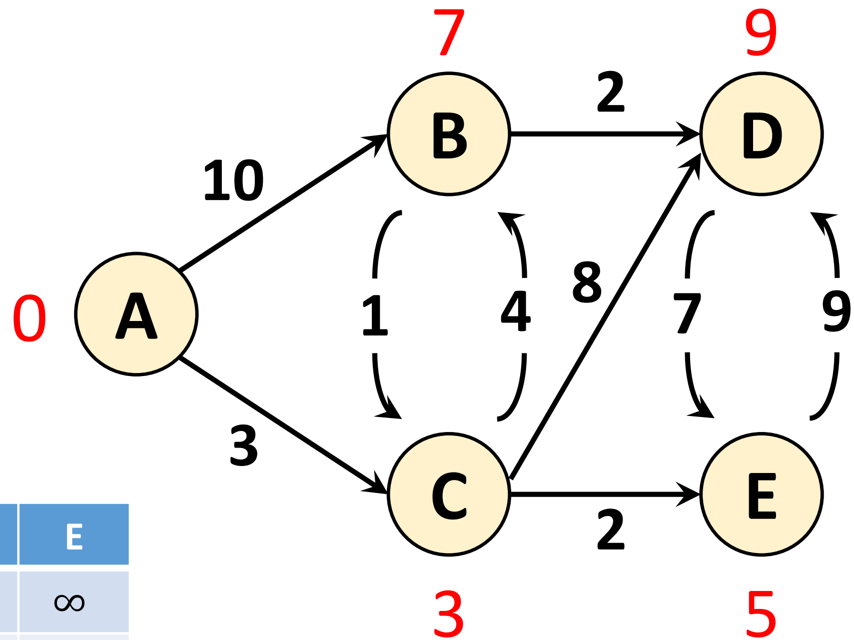



| | A | B | C | D | E |
|---|---|---|---|---|---|
| $d_0(u)$ | 0 | ∞ | ∞ | ∞ | ∞ |
| $d_1(u)$ | 0 | 10 | 3 | ∞ | ∞ |
| $d_2(u)$ | 0 | 7 | 3 | 11 | 5 |
| $d_3(u)$ | 0 | 7 | 3 | 11 | 5 |
| $d_4(u)$ | 0 | 7 | 3 | 9 | 5 |

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

# Dijkstra's Algorithm: Demo

**Don't need to explore D**

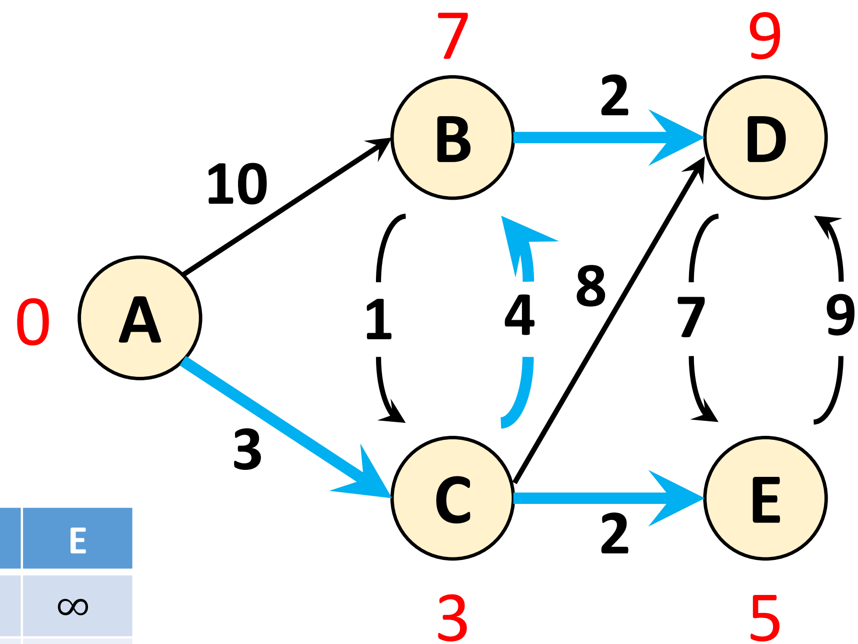



| | A | B | C | D | E |
|---|---|---|---|---|---|
| $d_0(u)$ | 0 | ∞ | ∞ | ∞ | ∞ |
| $d_1(u)$ | 0 | 10 | 3 | ∞ | ∞ |
| $d_2(u)$ | 0 | 7 | 3 | 11 | 5 |
| $d_3(u)$ | 0 | 7 | 3 | 11 | 5 |
| $d_4(u)$ | 0 | 7 | 3 | 9 | 5 |

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

# Dijkstra's Algorithm: Demo

**Maintain parent pointers so we can find the shortest paths**



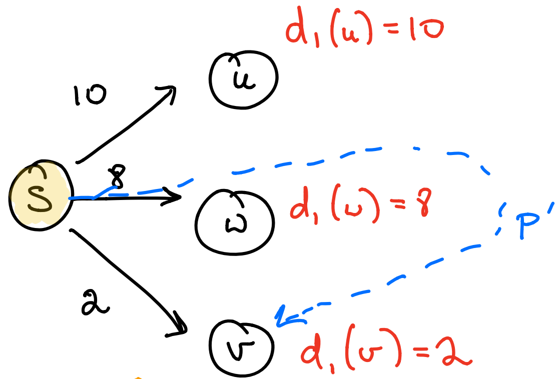

| | A | B | C | D | E |
|---|---|---|---|---|---|
| $d_0(u)$ | 0 | ∞ | ∞ | ∞ | ∞ |
| $d_1(u)$ | 0 | 10 | 3 | ∞ | ∞ |
| $d_2(u)$ | 0 | 7 | 3 | 11 | 5 |
| $d_3(u)$ | 0 | 7 | 3 | 11 | 5 |
| $d_4(u)$ | 0 | 7 | 3 | 9 | 5 |

# Correctness of Dijkstra

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

Quite a trivial stmt

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




explore v 2nd

- $d(v) \le 2$
- To prove: length of $P' \geq 2$

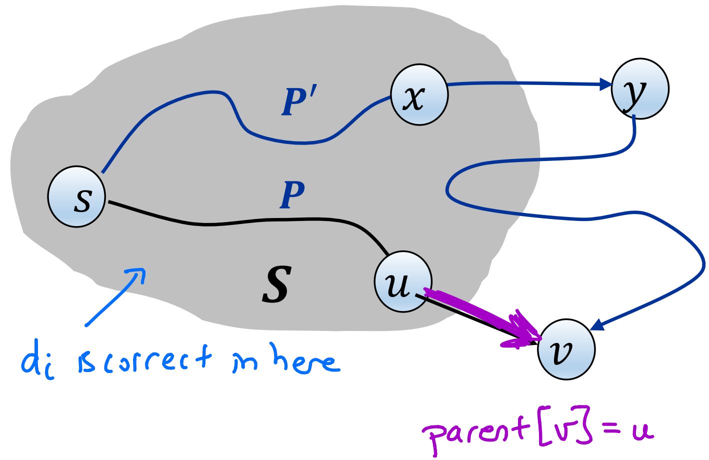
$$\ell(P') \geq \ell(s \to w)$$
$$\geq \ell(s \to v)$$
$$= d(v)$$

# 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

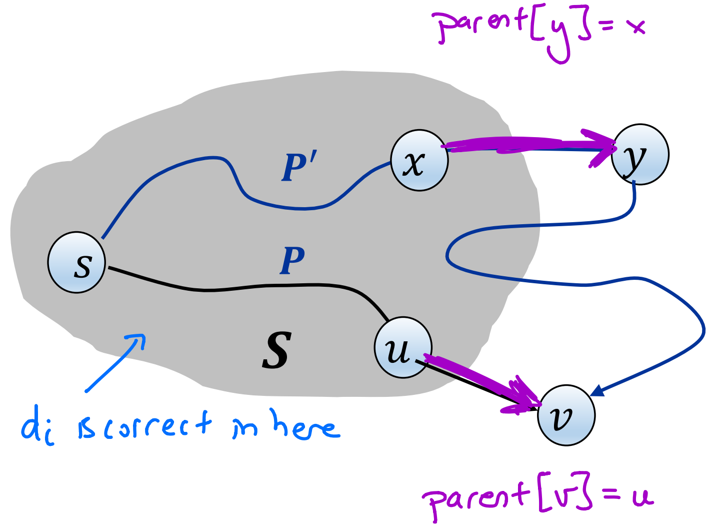
# Correctness of Dijkstra




- **Invariant:** after we explore the i-th node, $d_i(v)$ is correct for every $v \in S$
- **Proof:**
  - $\ell(P) = d_i(v)$
  - Consider any other path $P'$. We'll show $\ell(P') \geq \ell(P)$
  - $P'$ can be written as $P_{s,x} + (x,y) + P_{y,v}$

    ($P_{s,x}$) might as be the shortest path

# Correctness of Dijkstra




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

$$\ell(P') = \ell(P'_{s,x}) + \ell(x \to y) + \ell(P'_{y,v})$$

$$\geq \ell(P'_{s,x}) + \ell(x \to y) \qquad [\ell(e) \geq 0]$$

$$\geq d(s,x) + \ell(x \to y) \qquad [\text{def of distance}]$$

$$= d_i(x) + \ell(x \to y) \qquad [\text{By invariant + } x \in S]$$

$$\geq d_i(y) \qquad [\text{Because } x \text{ explored + } d_i \text{ goes down}]$$

$\geq d_i(v)$ [Because $y \notin S$, but we chose to explore $v$]

$\ell(P) = d_i(v)$ and for every path $P'$ from $s$ to $v$,

$\ell(P') \geq d_i(v)$

$\therefore$ $P$ is a shortest path and $d(s,v) = d_i(v)$

Suppose $x$ is the $j^{th}$ node explored for $j < i$

- $d_j(y) \leq d_j(x) + \ell(x \to y)$
- $d_i(y) \leq d_j(y)$
- $d_i(x) = d_j(x)$

$\therefore$ $d_i(y) \leq d_i(x) + \ell(u \to v)$

# 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 Naively

Total Time:

$$\sum_{u \in V} O(n) + O(\deg(u)+1)$$

$$= O(n^2 + m)$$

- Need to explore all $n$ nodes
- Each exploration requires:
  - ① Finding the unexplored node $u$ with smallest distance
  - ② Updating the distance for each neighbor of $u$
    - 2a Lookup current distance
    - 2b Possibly decrease distance

Bottleneck is finding the min

① Takes $O(n)$ time to find minimum distance node

② For each of the $\deg(u)$ neighbors

2a $O(1)$ time to lookup

2b $O(1)$ time to decrease

# Priority Queues / Heaps

# Priority Queues

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

keys = nodes u

values = d[u]

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

if $(d[v] > d[u] + \ell(u \to v))$

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

$u \leftarrow \arg\min_{w \in Q} d[w]$

# 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 |

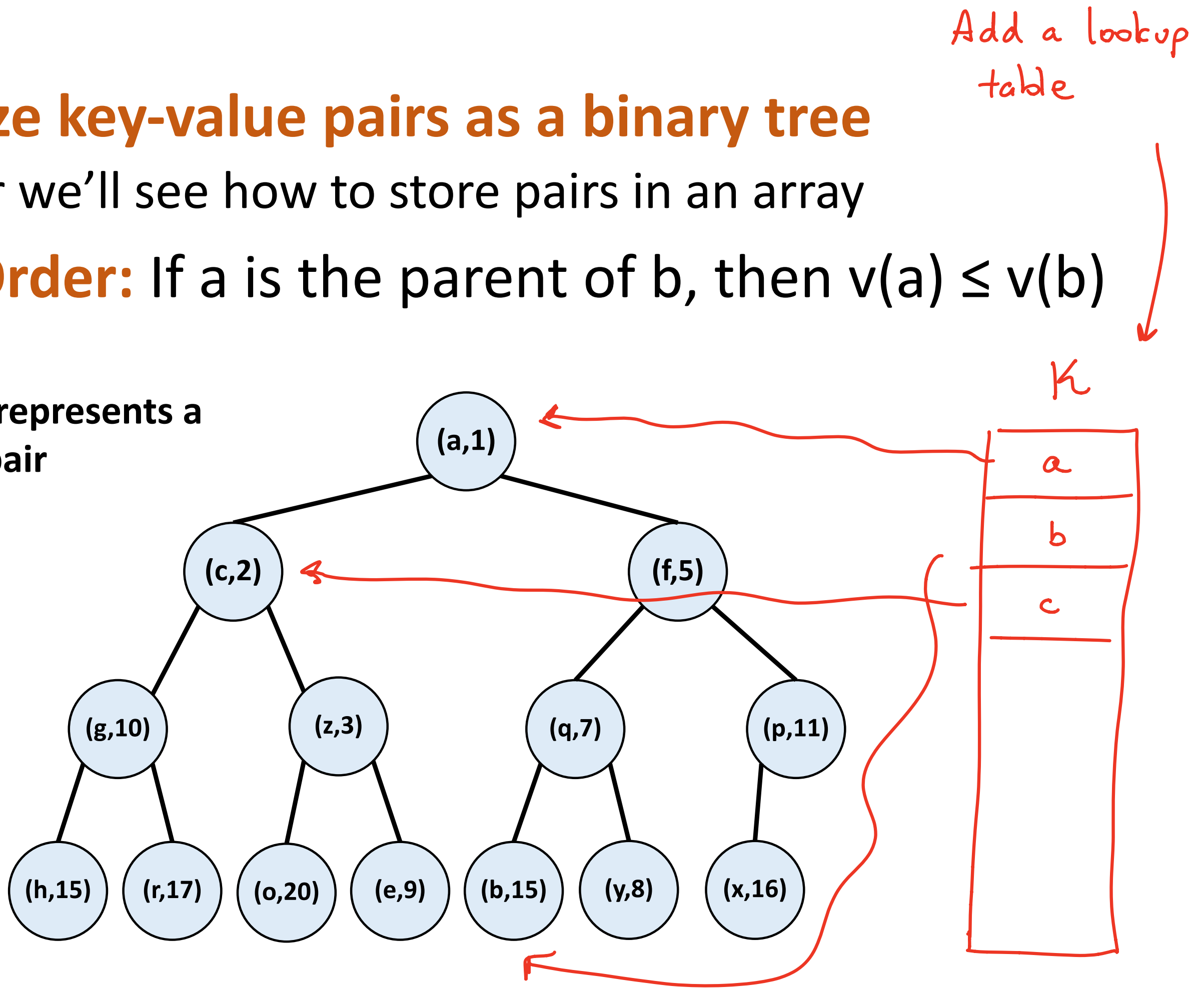
  - Insert takes O(1) time
  - ExtractMin, DecreaseKey take O(n) time

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

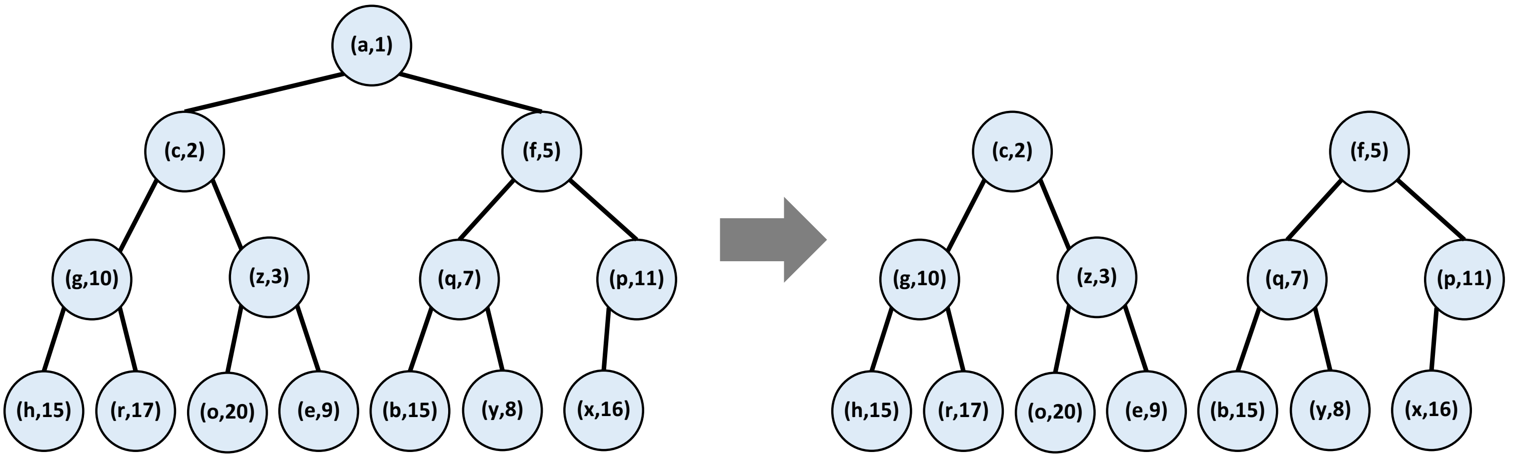
# Heaps

- **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)

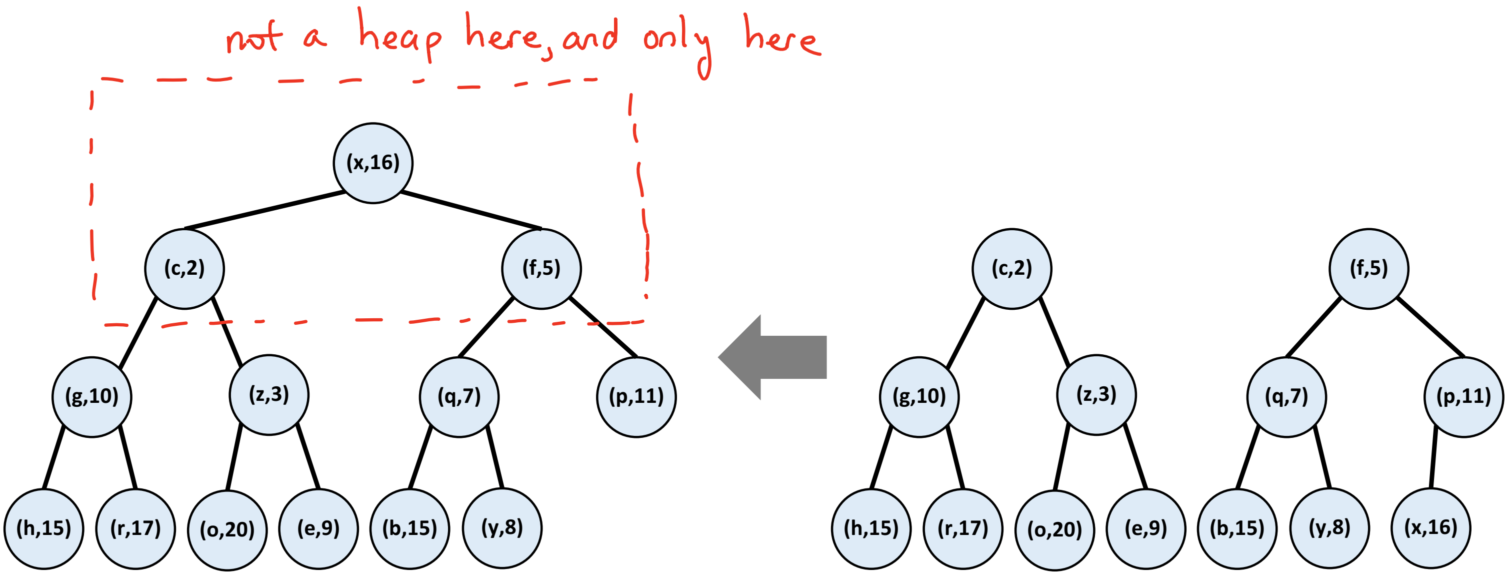
**Each node represents a key-value pair**



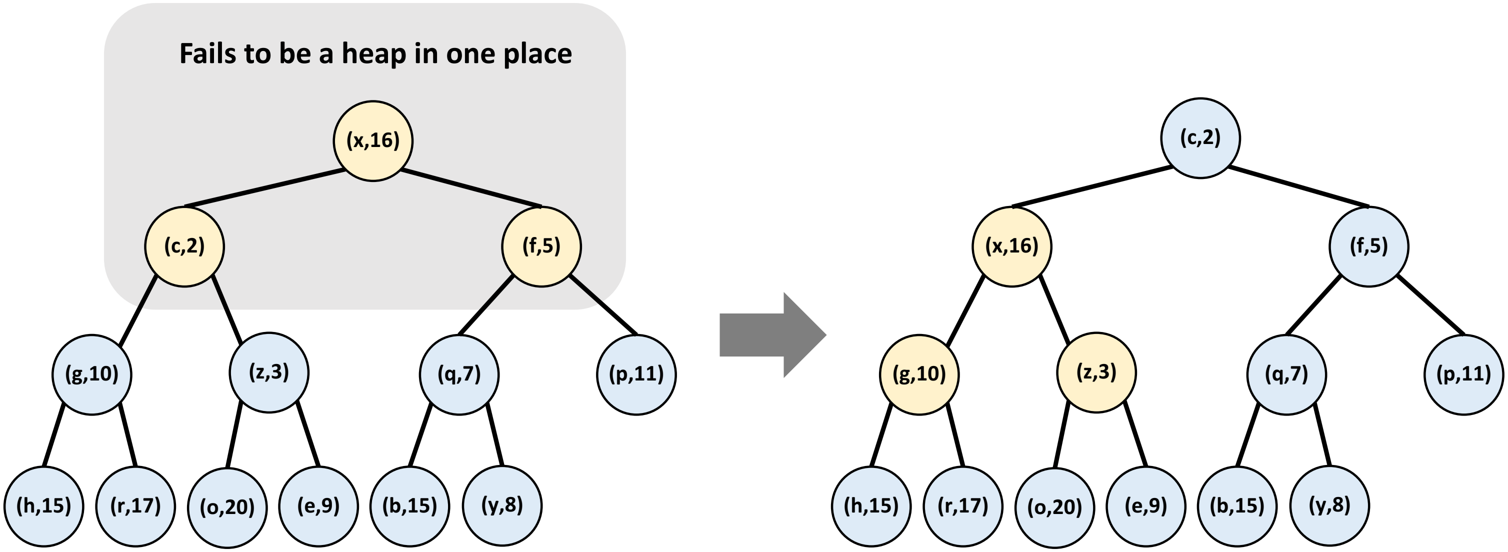
# Implementing ExtractMin



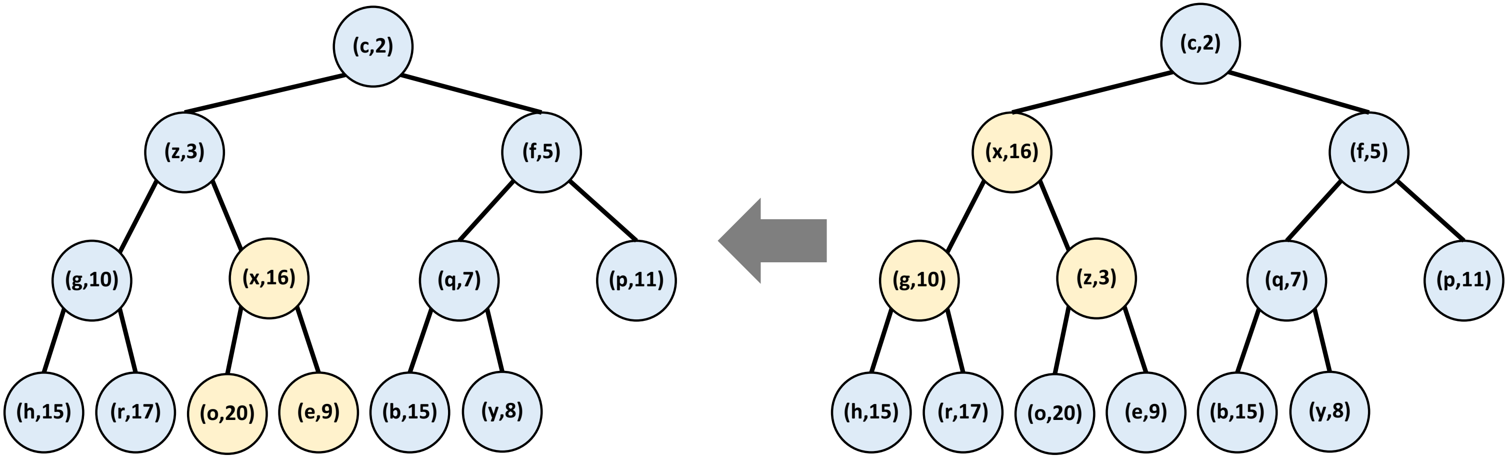
# Implementing ExtractMin



# Implementing ExtractMin

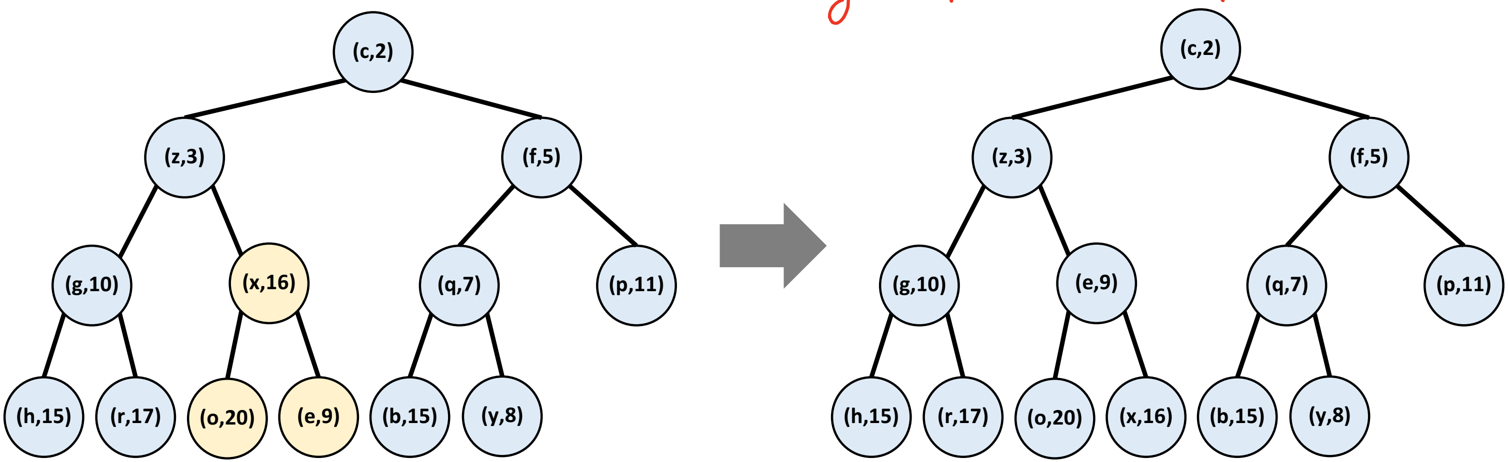


# Implementing ExtractMin



# Implementing ExtractMin

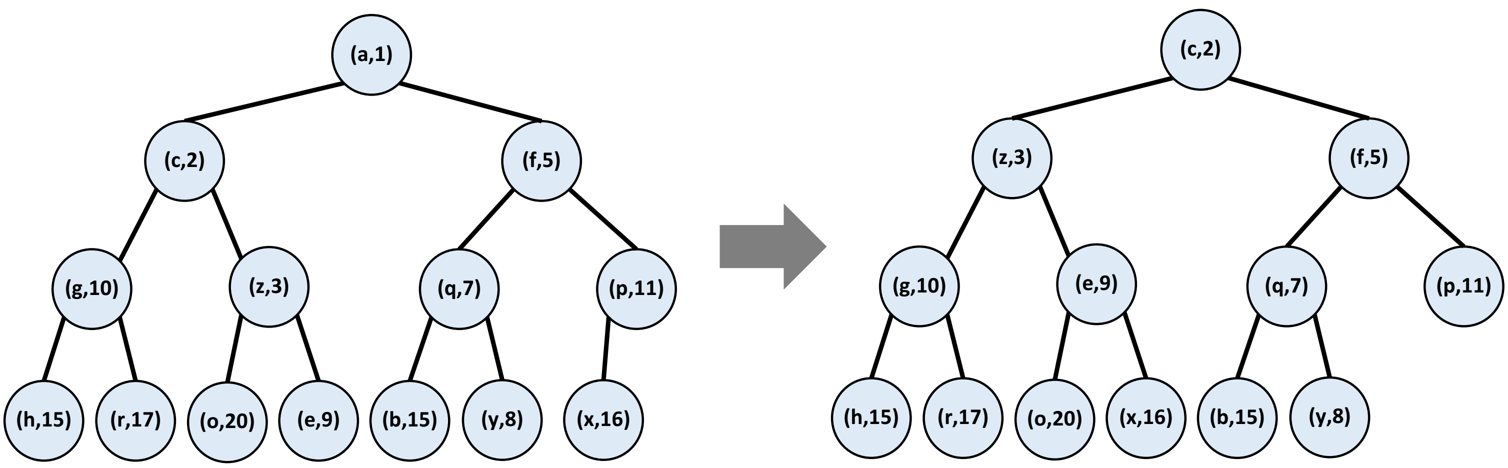
- For any triple, we can fix the heap property in $O(1)$ time
- Any swap lowers the problem one level



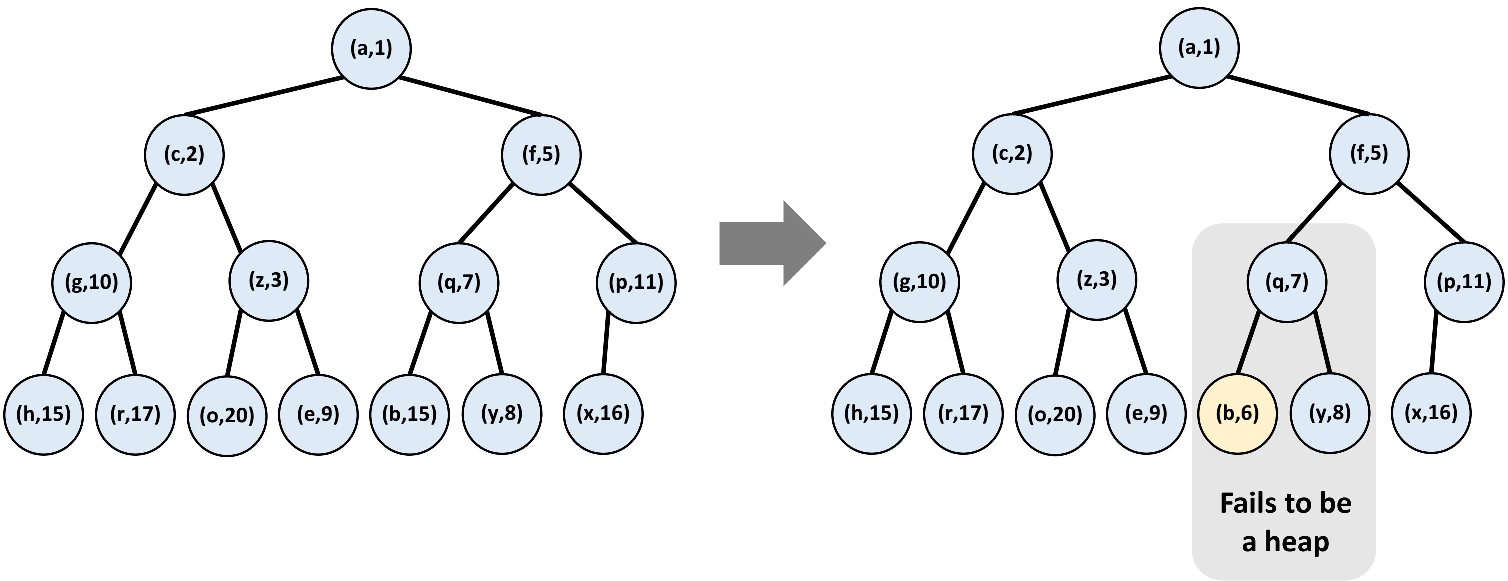

- Only $\lceil \log_2(n+1) \rceil = O(\log n)$ levels!

# Implementing ExtractMin

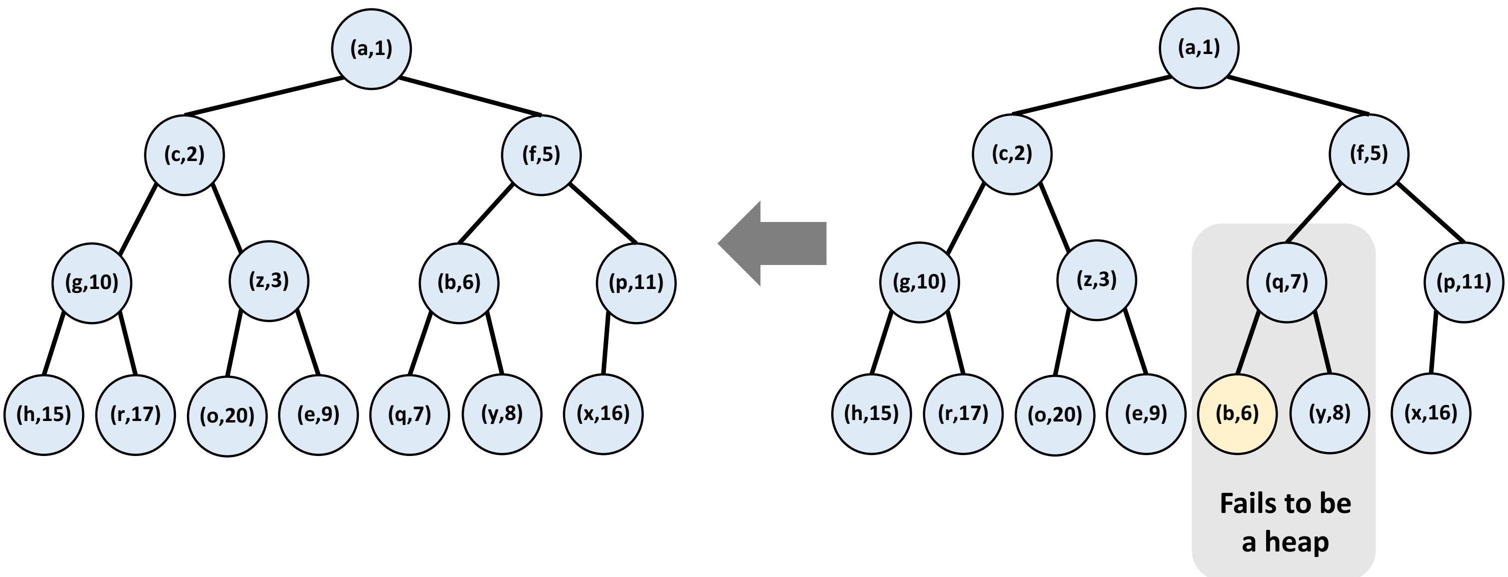
- Three steps:
  - Pull the minimum from the root O(1)
  - Move the last element to the root O(1)
  - Repair the heap-order (heapify down) O(log n)



# Implementing DecreaseKey



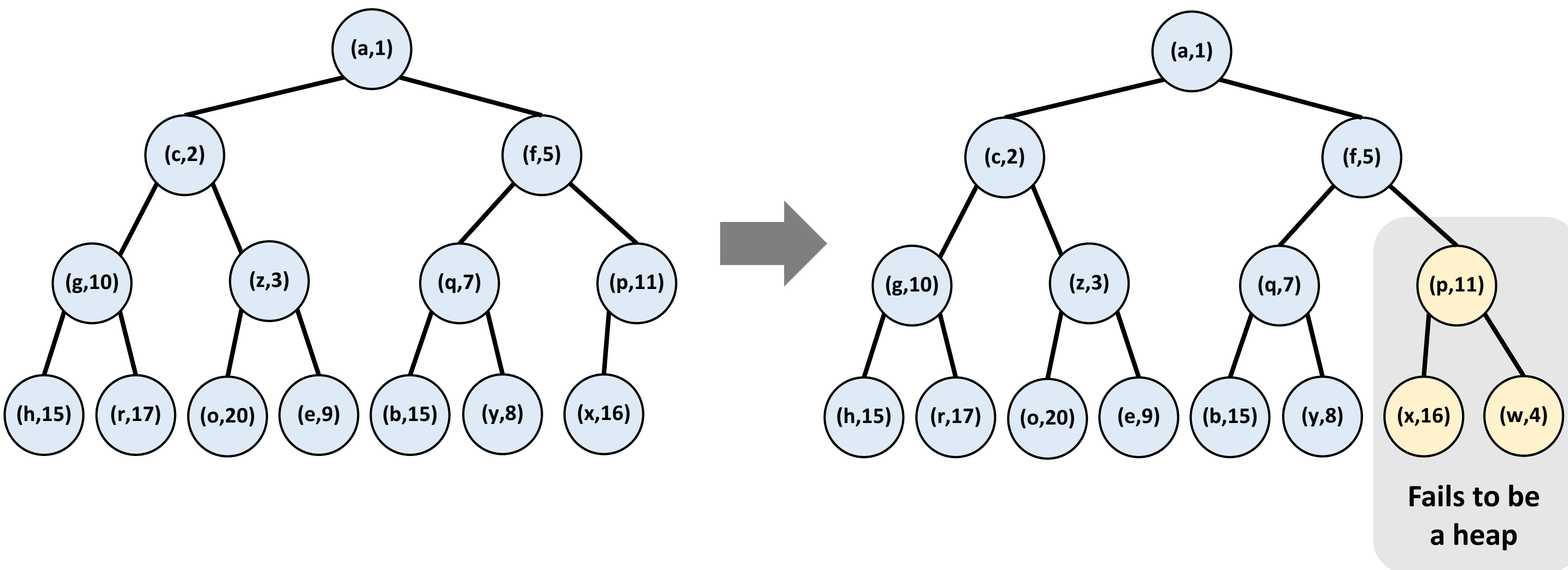
# Implementing DecreaseKey



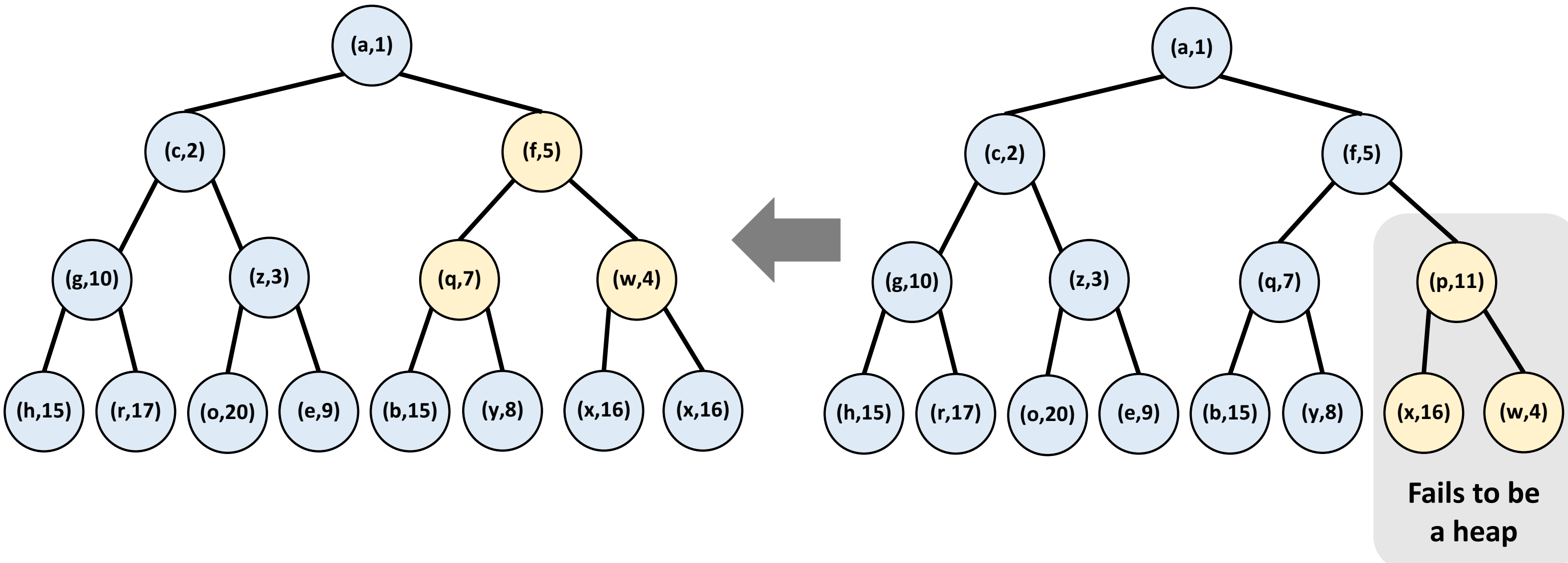
# Implementing DecreaseKey

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

# Implementing Insert



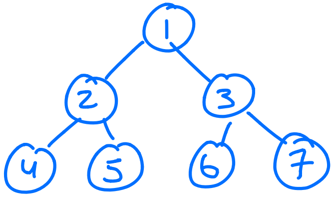
# Implementing Insert

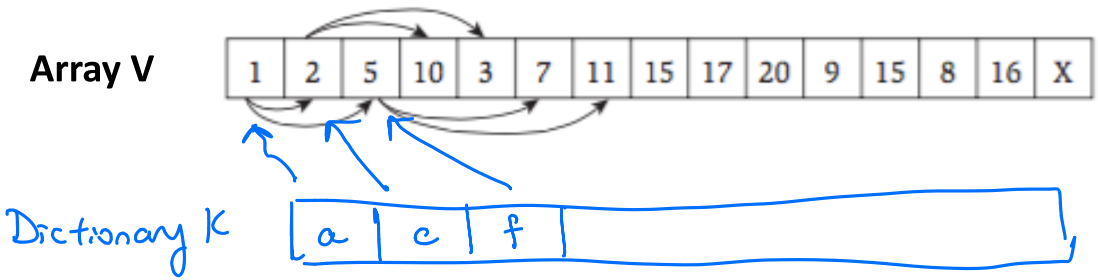


# Implementing Insert

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

# Implementation Using Arrays




- 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 in the binary tree
  - LeftChild(i) = 2i
  - RightChild(i) = 2i+1
  - Parent(i) = $\lfloor i/2 \rfloor$

# 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
                                   O(n) time

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

    For ((u,v) in E): ← Loop deg(u) times
      d[v] ← Lookup(Q,v) ← O(1)
      If (d[v] > d[u] + ℓ(u,v)):
        DecreaseKey(Q,v,d[u] + ℓ(u,v)) ← O(log n)
        parent[v] ← u

  Return (d, parent)
```

$O(n) + \sum_{u \in V} O(\log n) + O(\log n \cdot deg(u))$

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

# 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: $O(m \log n)$
  - For negative weight edges, Bellman-Ford takes $O(mn)$ time

- **Compare to BFS**