CS3000: Algorithms & Data
Jonathan Ullman

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

Oct 26, 2018

Announcements

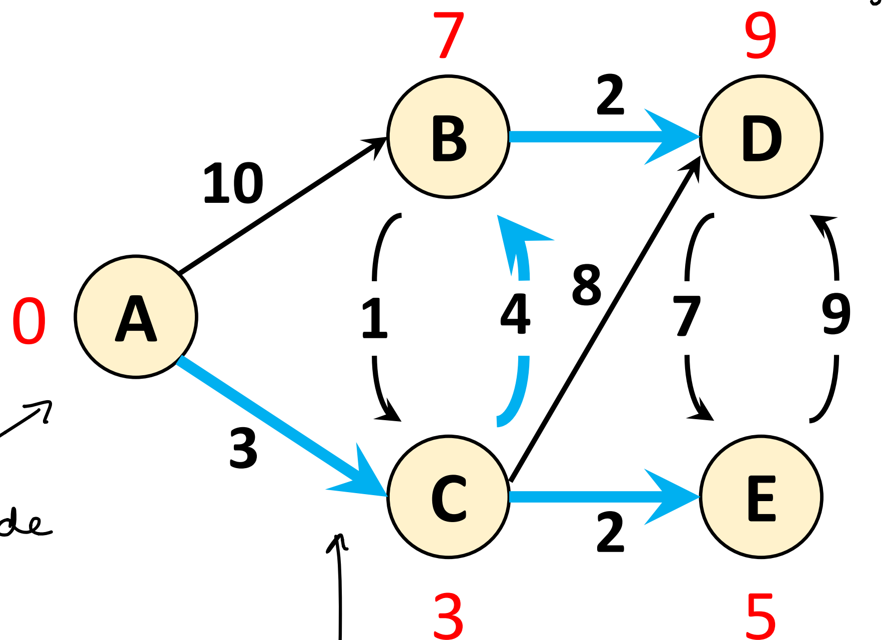
- HW5 due tonight
- HW6 out now, due 11/2

Shortest Paths

Given: • $G = (V, E, \{\ell(e)\})$

• source node $s \in V$

Find shortest path from s to all $u \in V$



source node

blue edges: shortest path tree

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

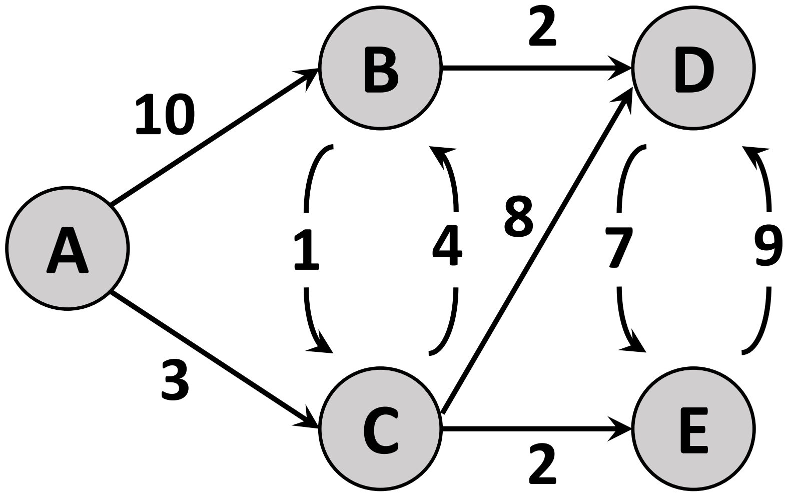
- The length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge lengths

- 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

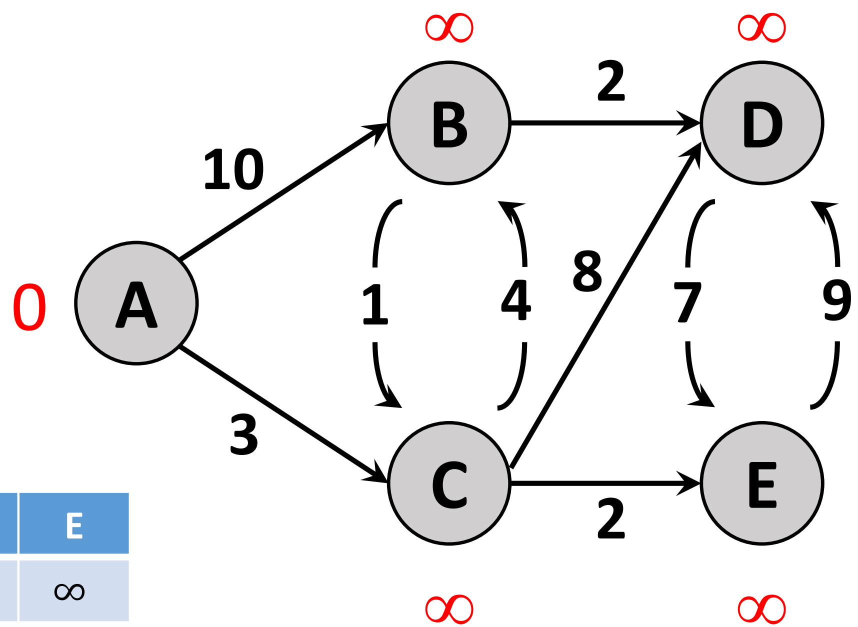
- **Dijkstra's Shortest Path Algorithm** is a modification of BFS for non-negatively weighted graphs
- **Informal Version:**
  - **Maintain a set S of explored nodes** (Initially empty)
  - **Maintain an upper bound on distance** (Initially $d(s)=0$, $d(u)=\infty$)
    - If u is explored, then we know $d(u)$ (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**
  - **Repeat until we're done**

Dijkstra’s Algorithm: Demo



Dijkstra's Algorithm: Demo

Initialize

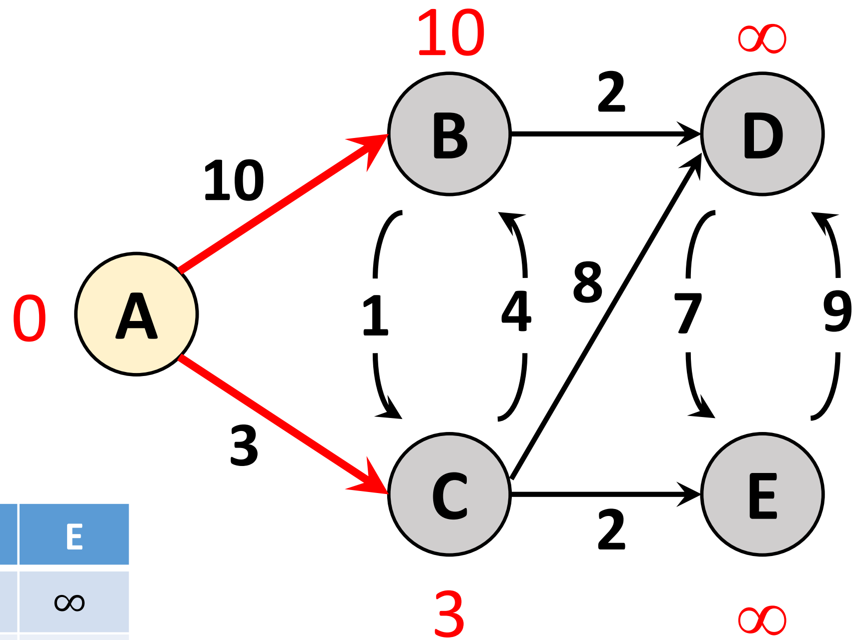


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

$$S = \{\}$$

Dijkstra's Algorithm: Demo

Explore A

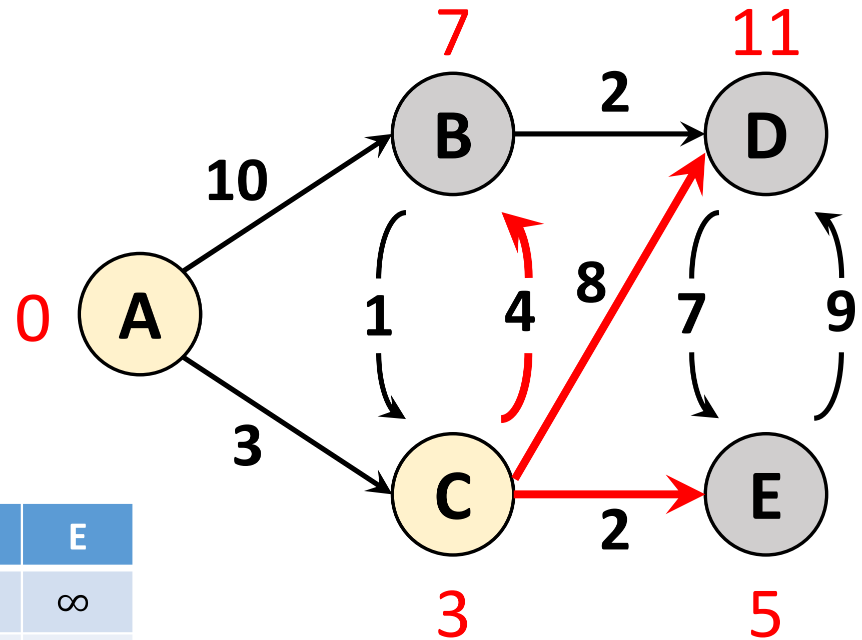


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞

$$S = \{A\}$$

Dijkstra's Algorithm: Demo

Explore C

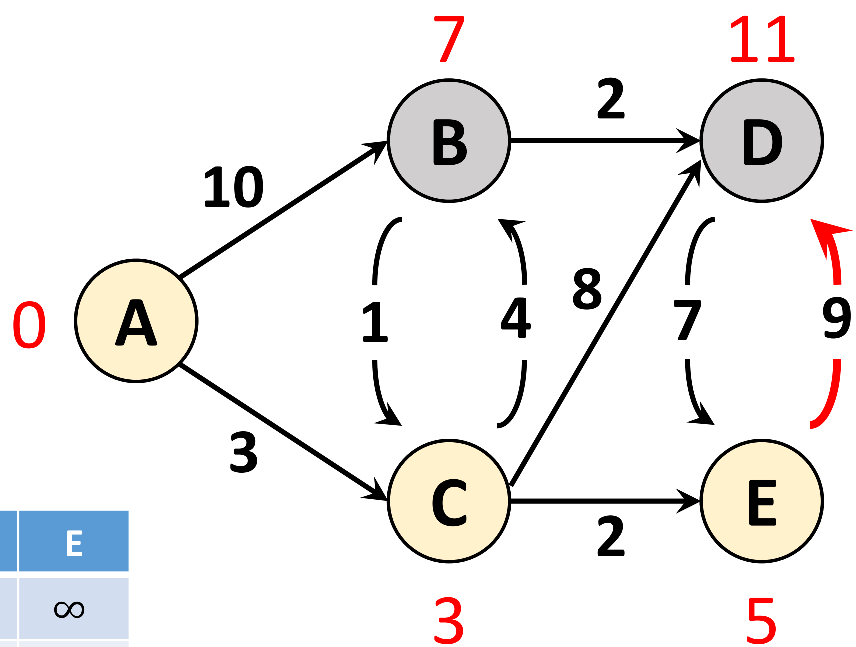


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5

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

Dijkstra's Algorithm: Demo

Explore E

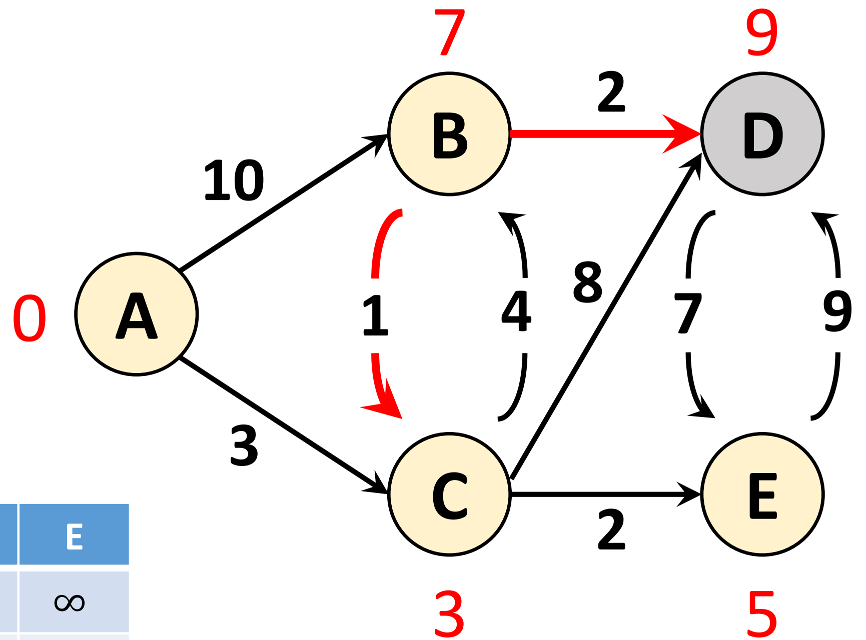


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5

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

Dijkstra's Algorithm: Demo

Explore B

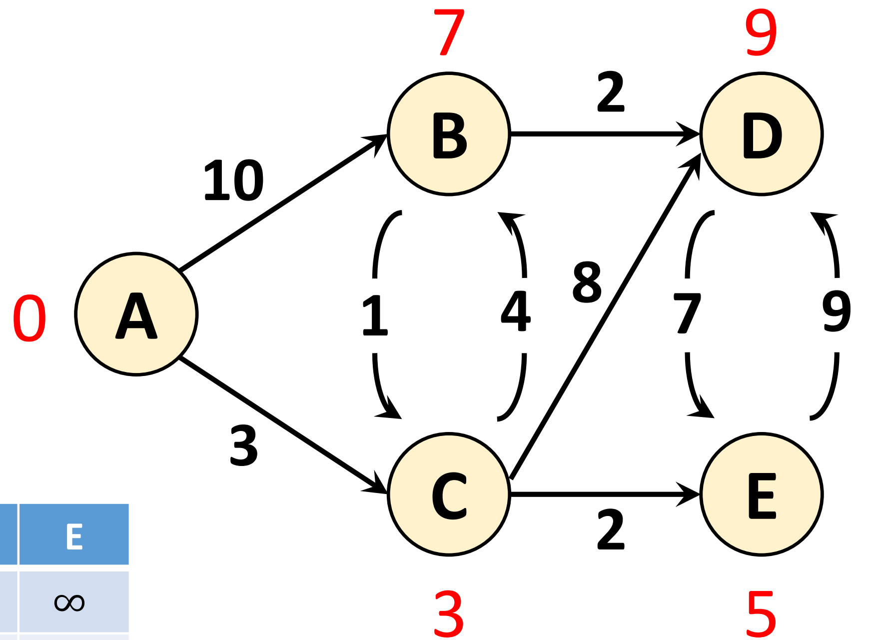


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

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

Dijkstra's Algorithm: Demo

Don't need to explore D

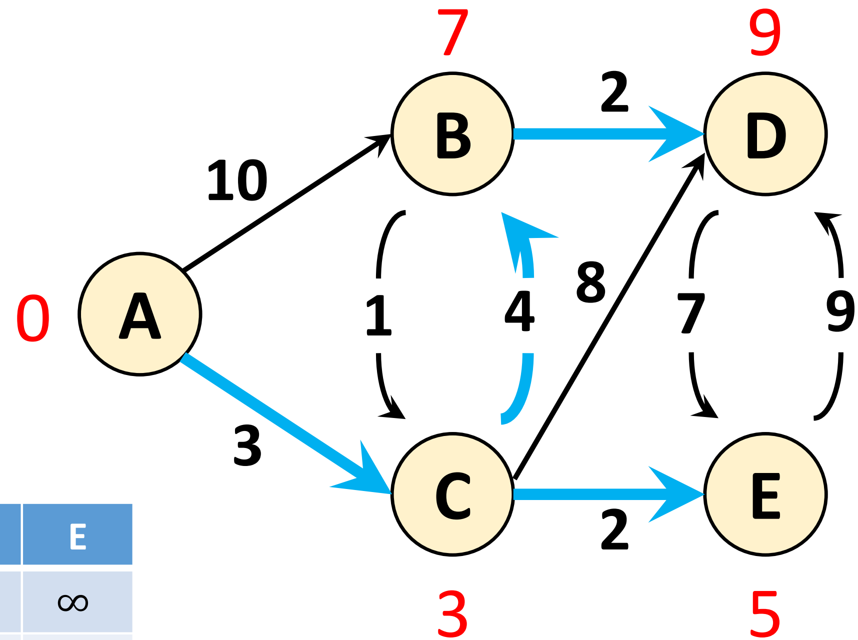


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

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

Dijkstra’s Algorithm: Demo

Maintain parent pointers so we can find the shortest paths



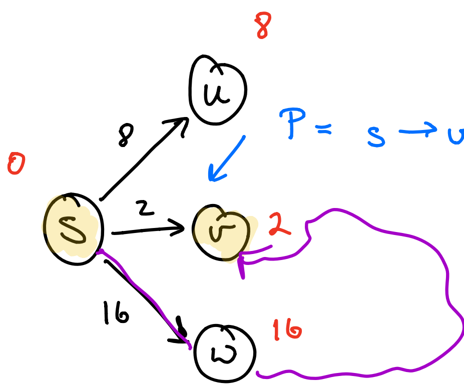
	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

Correctness of Dijkstra

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

Quite trivial

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



$P = s \to v \quad \ell(P) = \ell(s \to v)$

$P' = s \to w \rightsquigarrow v$

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

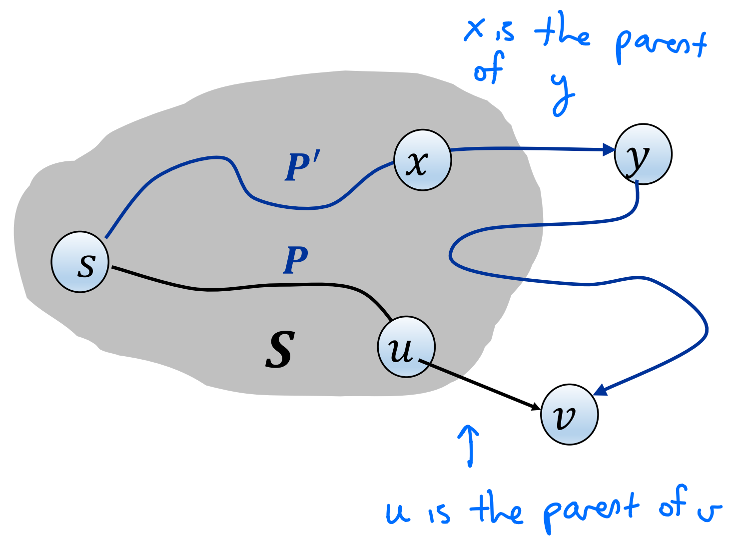
Correctness of Dijkstra

current guess of the distance when we explore i^{th} node

- **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 1st and 2nd nodes

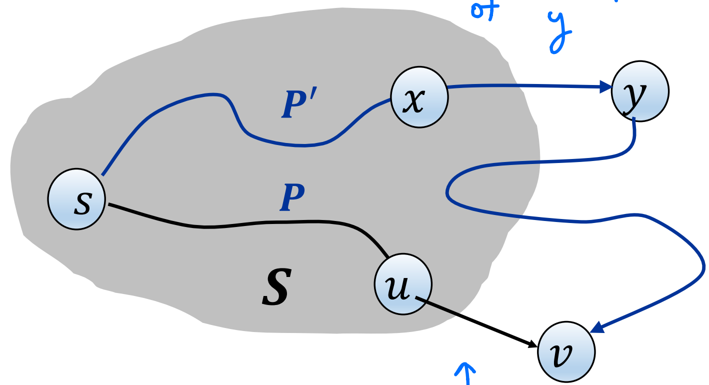
Correctness of Dijkstra



- **Invariant:** after we explore the i-th node, $d_i(v)$ is correct for every $v \in S$
- **Proof:**
  - There is some path P of length $\ell(P) = d_i(v)$
  - Consider any other path P' from s to v
    - P' leaves S using some edge $x \to y$
    - The part of P' from s to x is a shortest path, has length $d_i(x)$

Correctness of Dijkstra

x is the parent of y



u is the parent of v

- **Invariant:** after we explore the i-th node, $d_i(v)$ is correct for every $v \in S$
- **Proof:**
  - $\ell(P') \geq (\text{distance to } x) + \ell(x \to y)$

    $= d_i(x) + \ell(x \to y)$

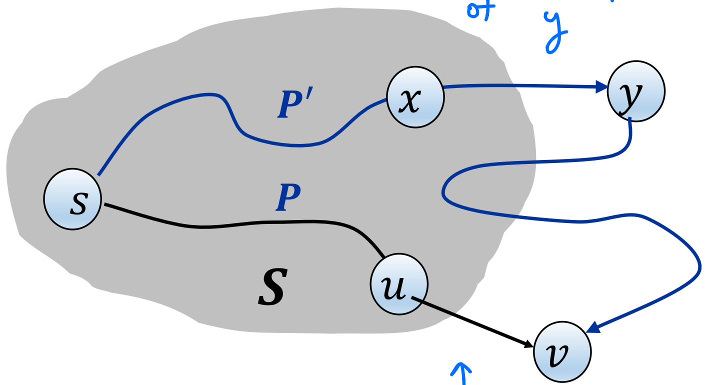
    $\geq d_i(y)$ [Because x was explored]

    $\geq d_i(v)$ [Because we explored v, not y]

    $\geq d(s,v)$ [Because d_i is an upper bound]

Correctness of Dijkstra

x is the parent of y



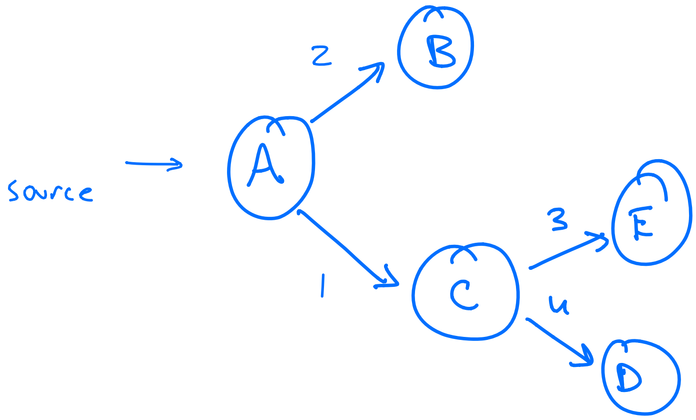
u is the parent of v

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

• $\ell(P') \geq \ell(P)$

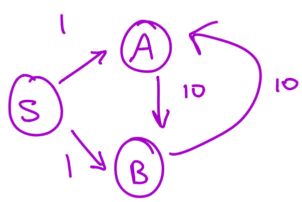
$\therefore$ P is a shortest path

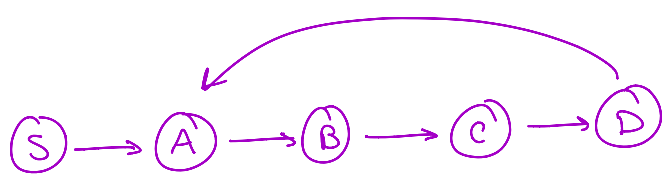
$\therefore$ $d_i(v) = d(s, v)$ $\square$



	A	B	C	D	E
d_0	0	∞	∞	∞	∞
d_1	0	2	$\boxed{1}$	∞	∞
d_2	0	$\boxed{2}$	1	5	4

- There is no relationship btw consecutive nodes that we explore





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

- 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
    - Lookup current distance
    - Possibly decrease distance

interact with each other

Priority Queues / Heaps

Priority Queues

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

  keys = nodes
  values = distances
  } in Dijkstra

- 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

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

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

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

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

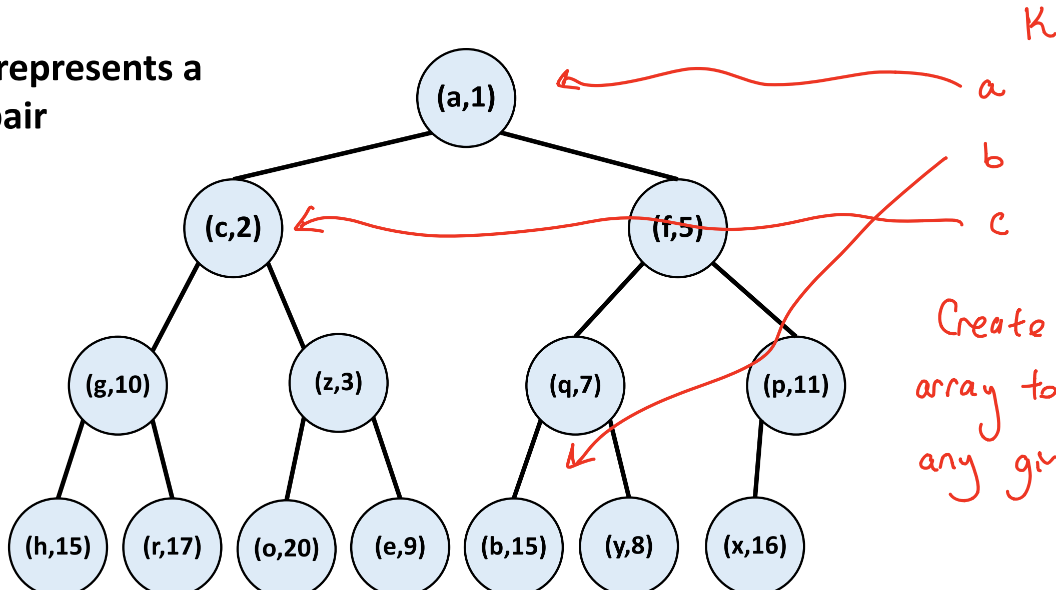
  - Insert takes O(1) time
  - ExtractMin, DecreaseKey take O(n) time

- **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)

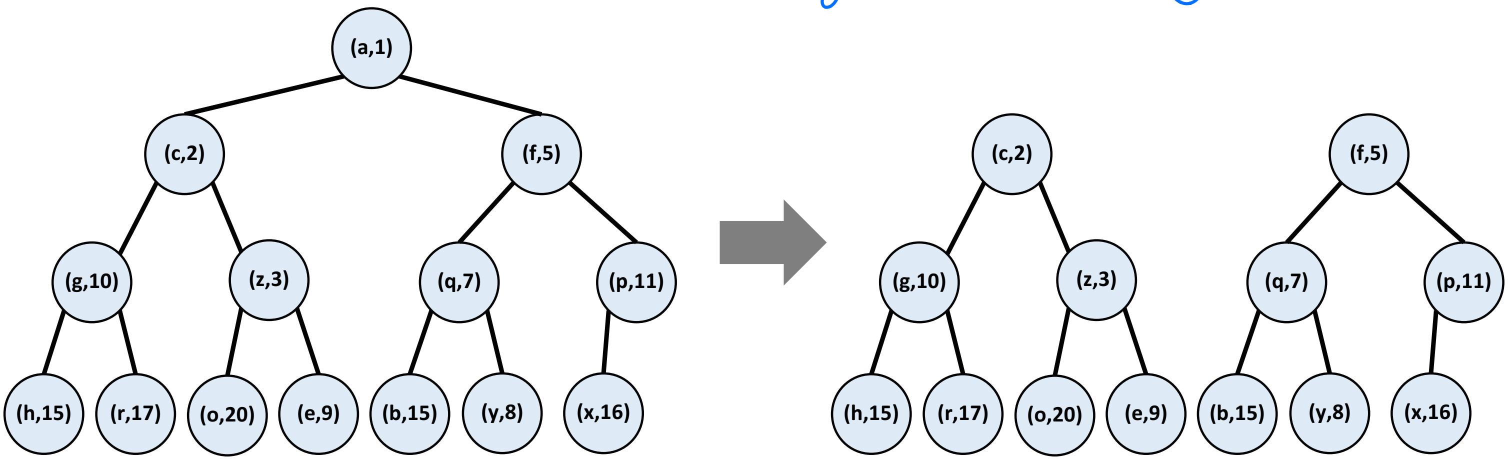
Each node represents a key-value pair



Create an array to locate any given key

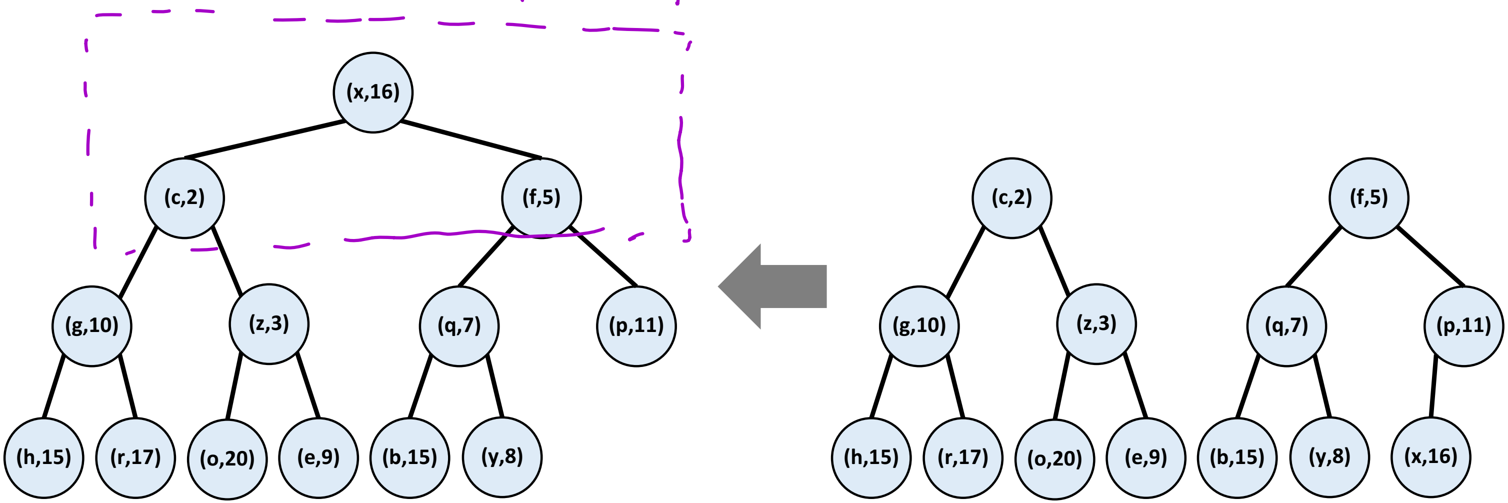
Implementing ExtractMin

- If we delete the min, we get two binary trees
- Need to get back to being a tree



Implementing ExtractMin

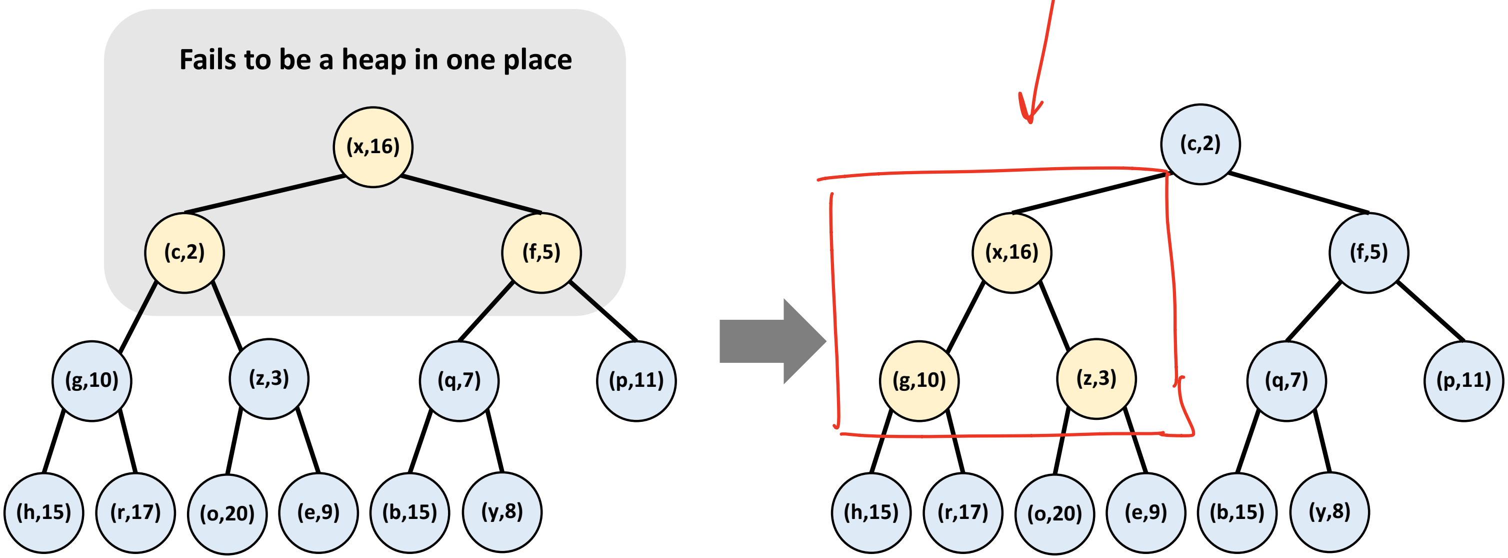
not in heap order



everything but the root satisfies heap order

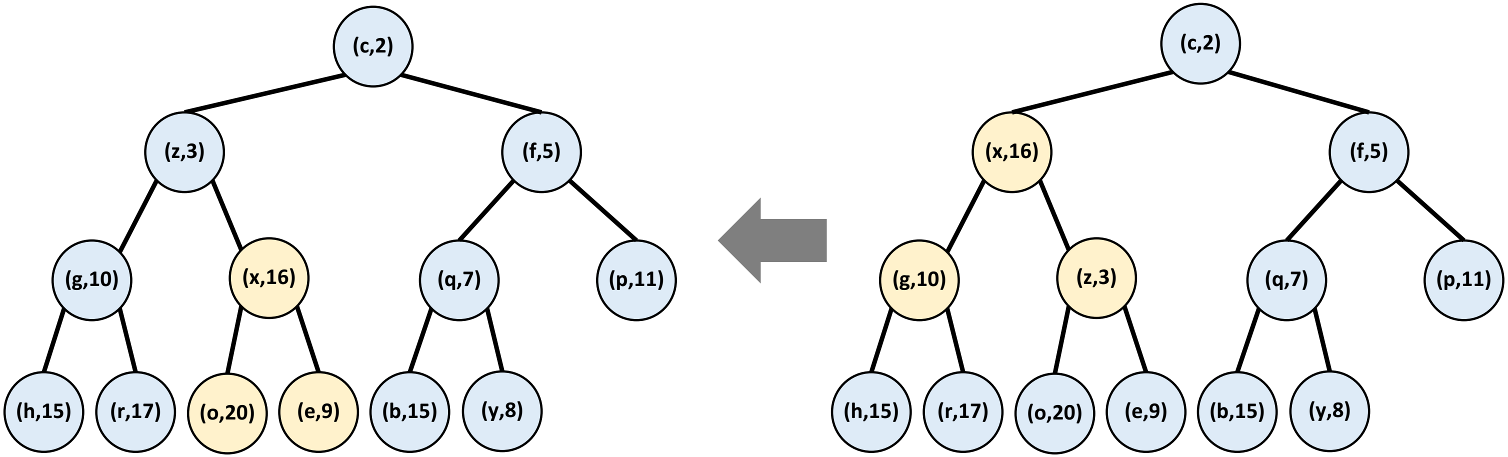
Implementing ExtractMin

we pushed the problem down a level

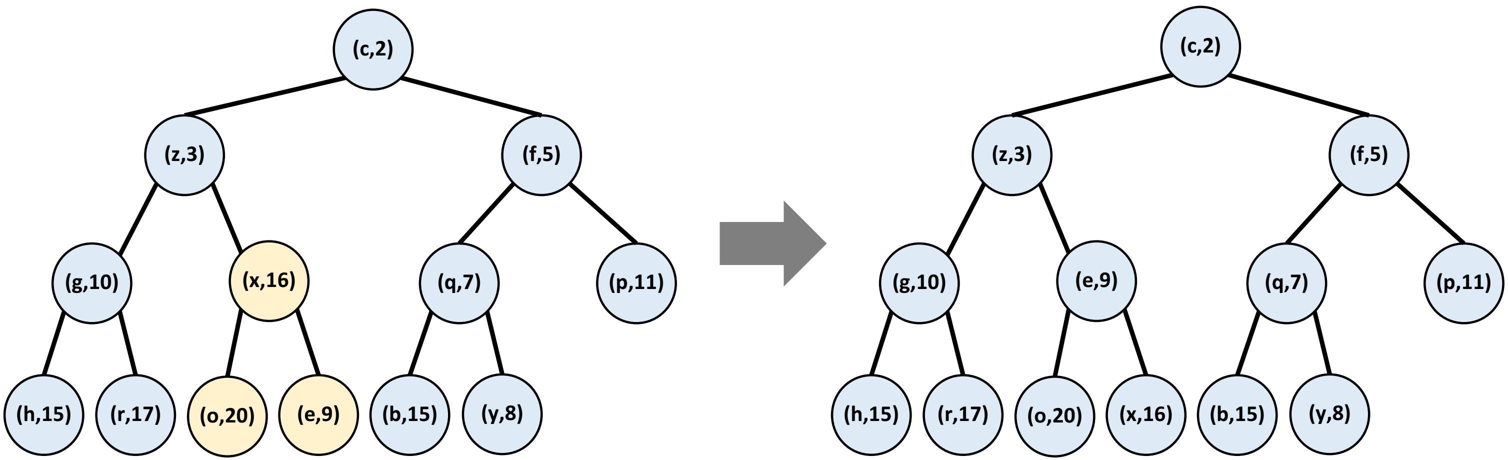


- If there are n key-value pairs, the # of levels is $O(\log n)$

Implementing ExtractMin



Implementing ExtractMin



After $O(\log n)$ swaps, the tree is in heap order again!

Implementing ExtractMin

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

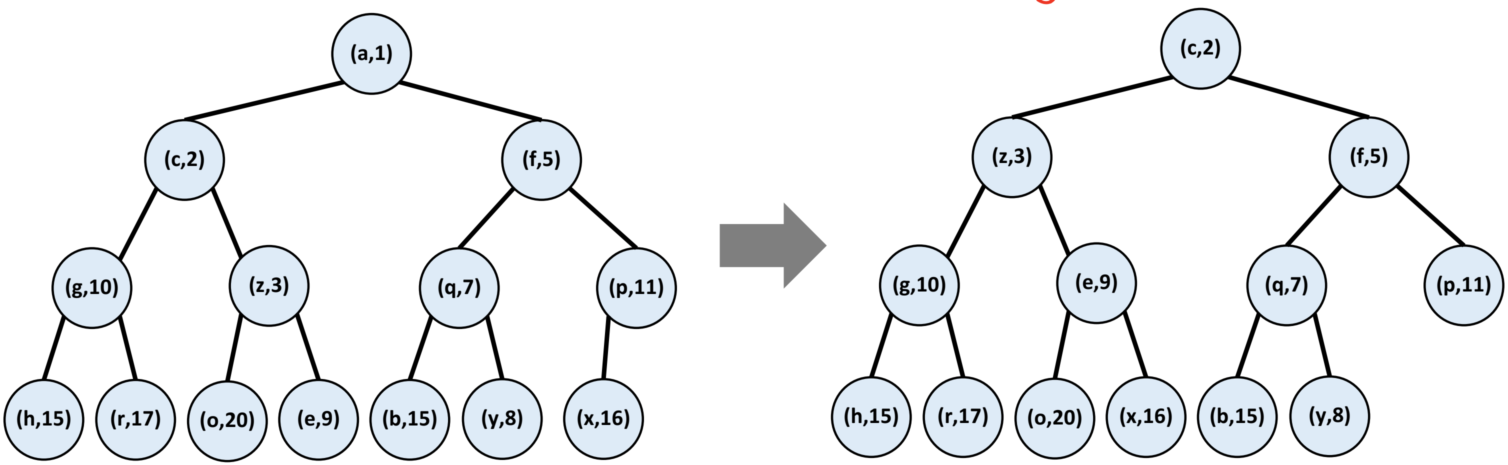
$O(1)$

$O(1)$

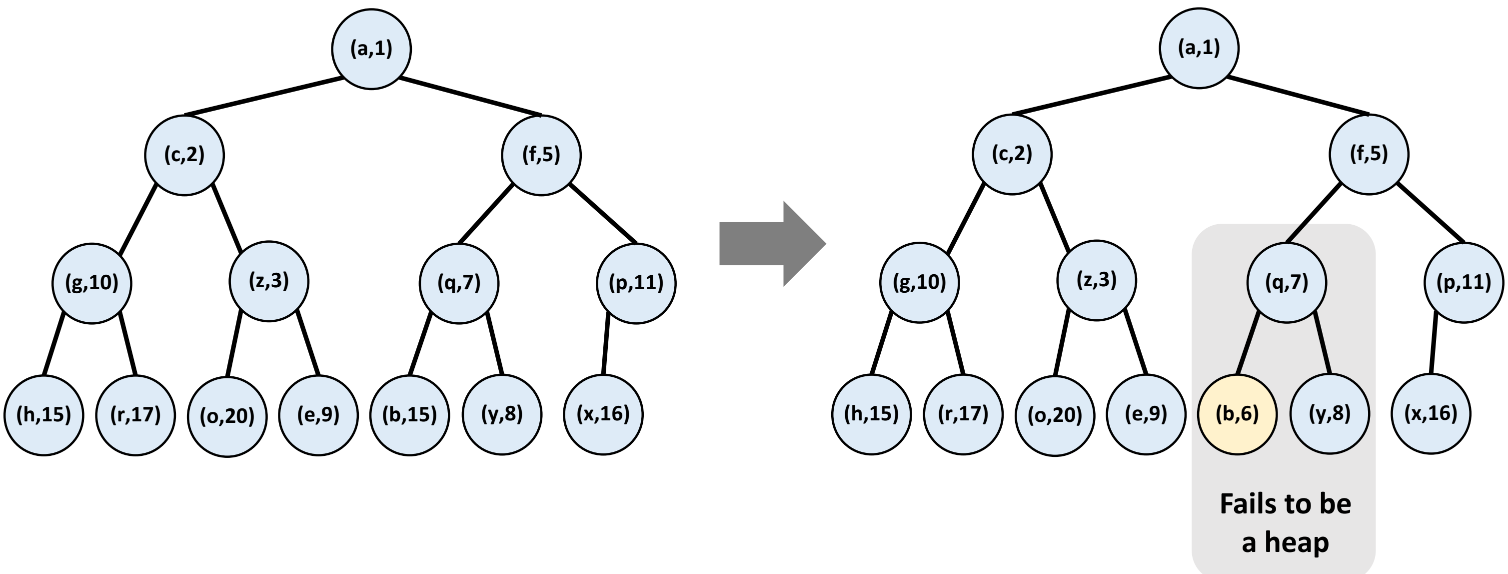
$O(1)$ time per swap

$O(\log n)$ swaps

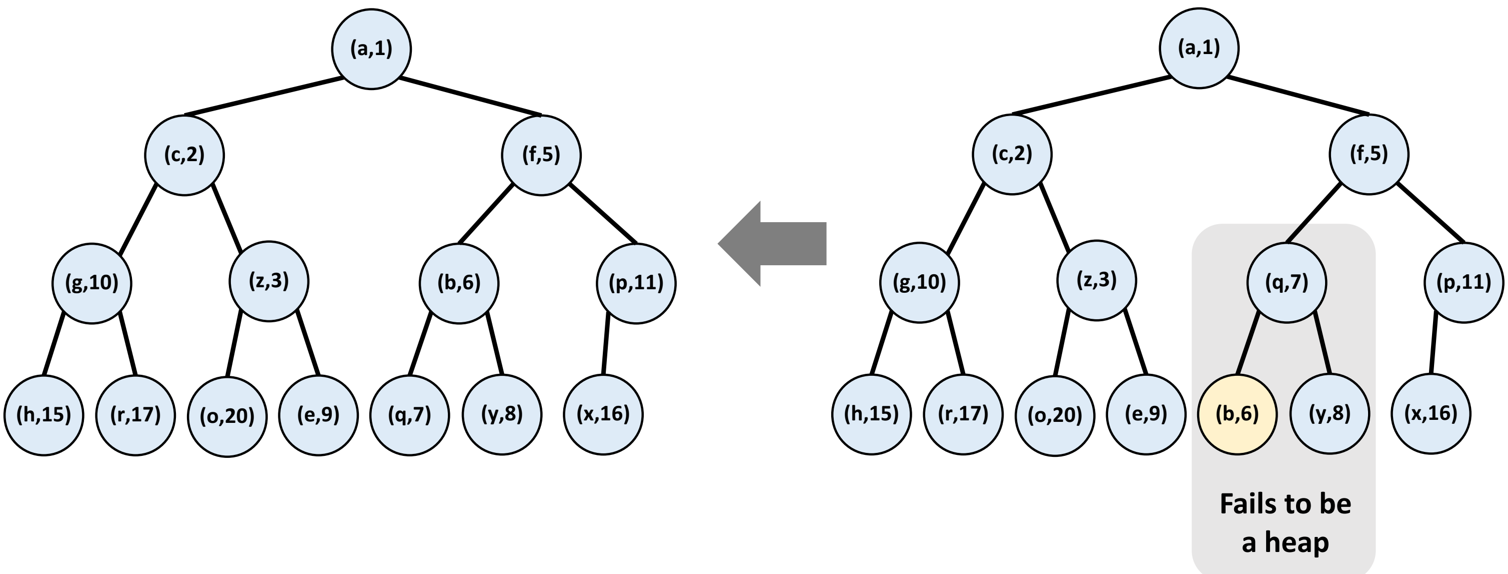
$O(\log n)$ time



Implementing DecreaseKey



Implementing DecreaseKey



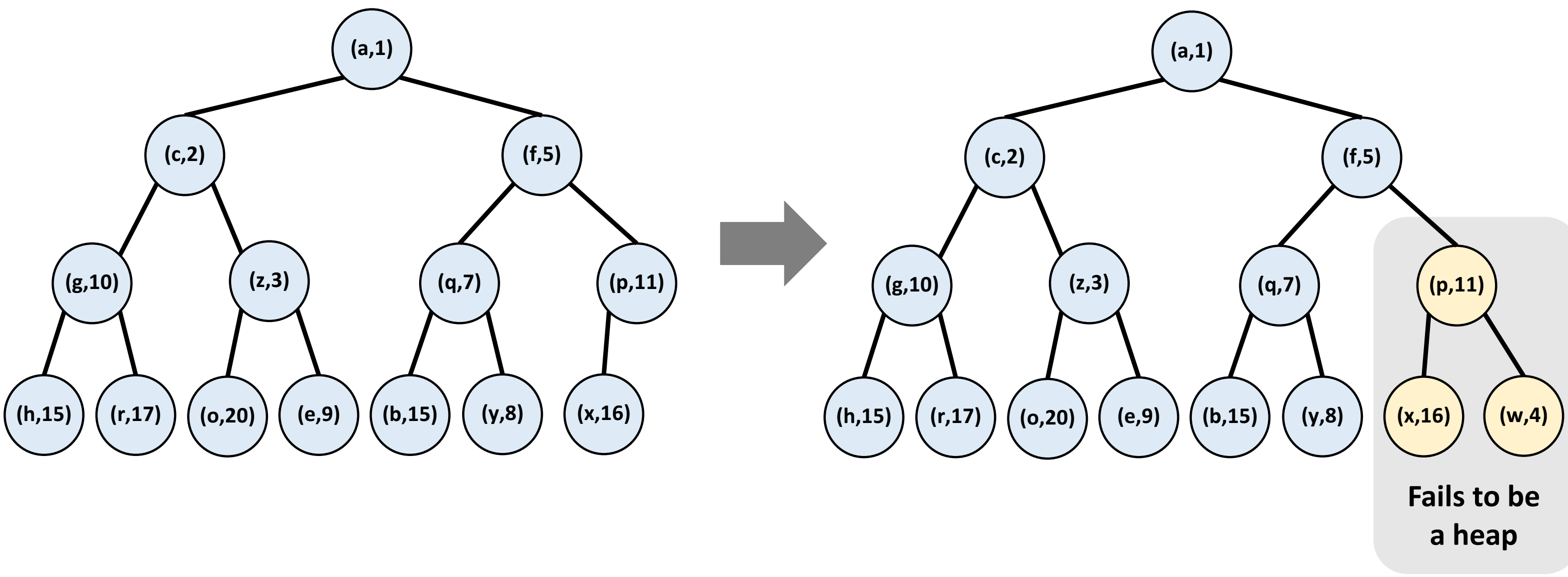
Implementing DecreaseKey

- Two steps:
  - Change the key $O(1)$ time
  - Repair the heap-order (heapify up) $O(1)$ time per swap

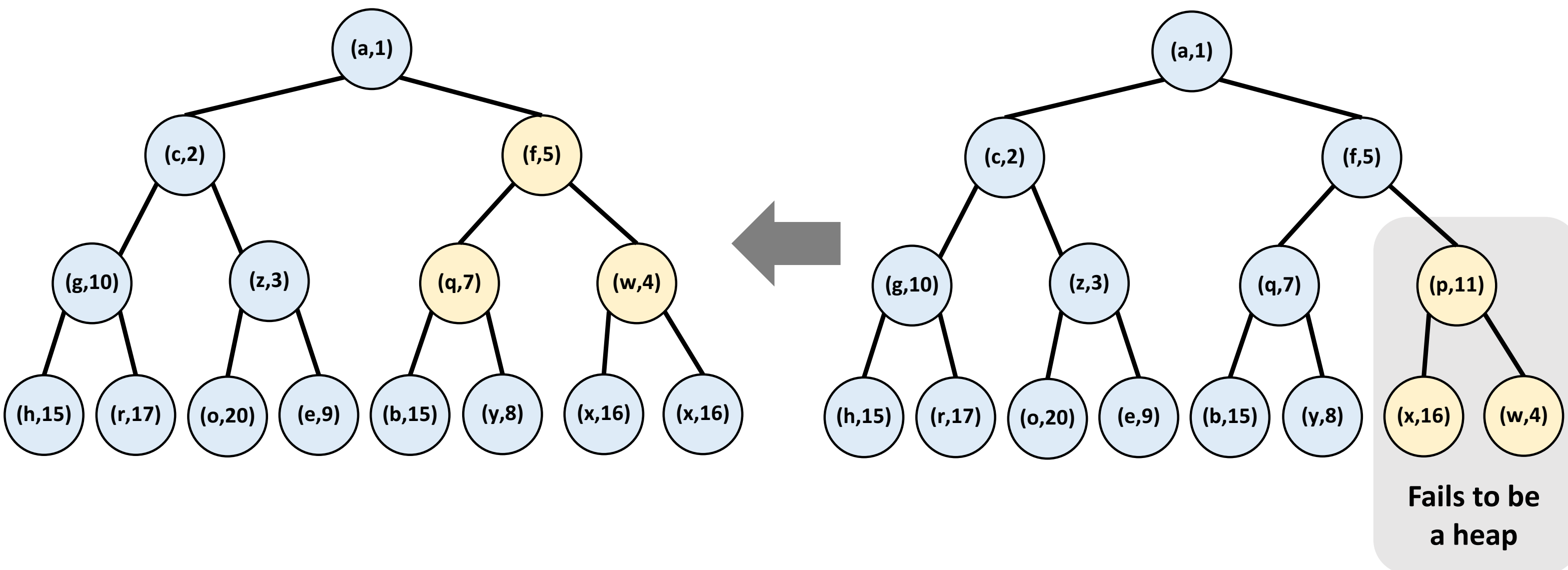
$\times\ O(\log n)$ swaps

$O(\log n)$ time

Implementing Insert



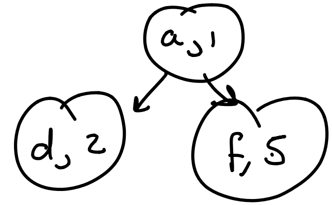
Implementing Insert



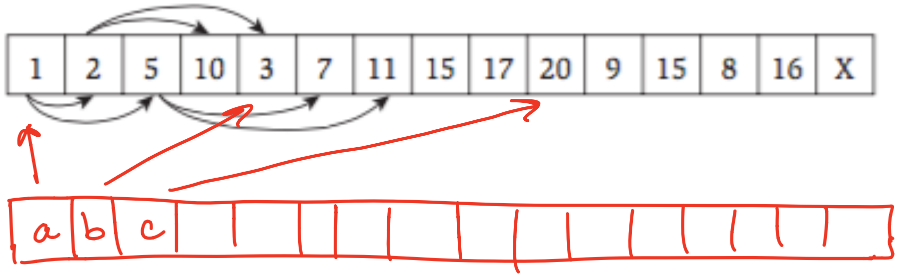
Implementing Insert

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

Implementation Using Arrays

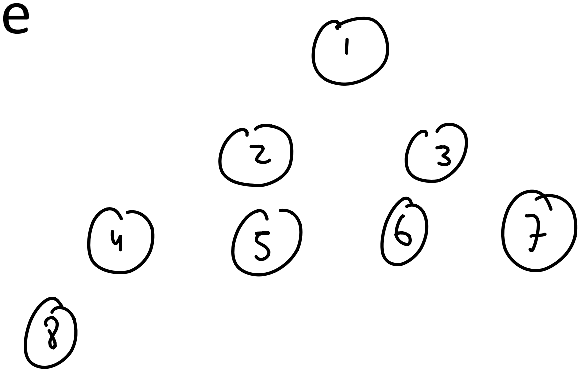


Array V



Array K

- 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

  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)
```

n insertions

$O(n \log n)$

n iterations

$O(\log n)$ time

deg(u) decrease key operations

$O(\deg(u) \cdot \log n)$

Total Time: $\sum_u O(\log n) + O(\deg(u) \cdot \log n) = O((m+n) \log n)$

$= O(m \log n)$

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: $O(m \log n)$
  - With negatively weighted edges: $O(mn)$ time
- **Compare to BFS**