Greedy Algorithms: Shortest Path

Greedy: Recap

- Greedy algorithms
    - Interval scheduling and minimizing lateness
    - Prove optimality using exchange argument
- Greedy algorithm on undirected graphs:
    - Minimum spanning tree problem
    - **Cut property** and **cycle property**
    - Prims: $O(m \log n)$ time using priority queue (min heap)
    - Kruskal's: $O(m \log n)$ time using union-find
- Last lecture on Greedy:
    - Shortest paths in weighted **directed** graphs

Cycle Property: MST

Lemma (Cycle Property). For any cycle C in G, its highest cost edge e is in no MST of G.

Proof. (By contradiction)

Suppose a MST T contains e.

- Main Idea: Construct another spanning tree $T' = T - \{e\} \cup \{e'\}$ with weight less than T ($\Rightarrow\!\Leftarrow$)
- How to find such an e'?
- **Take-home exercise**: finish the proof of this lemma

Cycle Property: MST

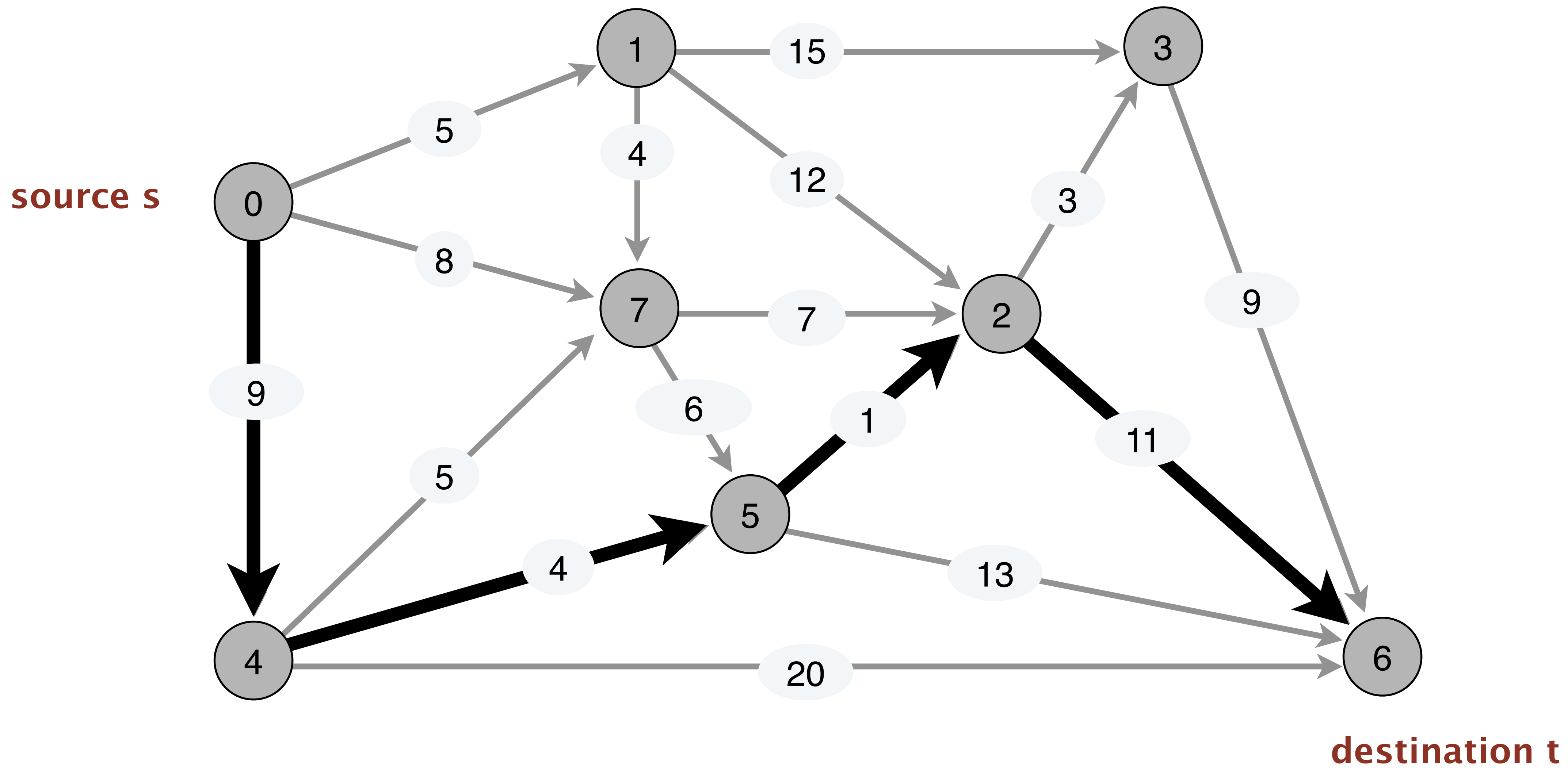
Lemma (Cycle Property). For any cycle C in G, its highest cost edge e is in no MST of G.

Proof. (By contradiction)

Suppose a MST T contains the heaviest edge $e = (u, v) \in C$.

- Removing e from T breaks the tree into two components: $S, V - S$ such that $u \in S$ and $v \in V - S$
- The cycle C in the graph must have another edge f going from S to $V - S$
- Adding f back to connect the components gives us another MST T'
- $w(T') < w(T)$, why?
- $\Rightarrow \Leftarrow$ ∎

Shortest Paths in Weighted Graph



length of path = 9 + 4 + 1 + 11 = 25

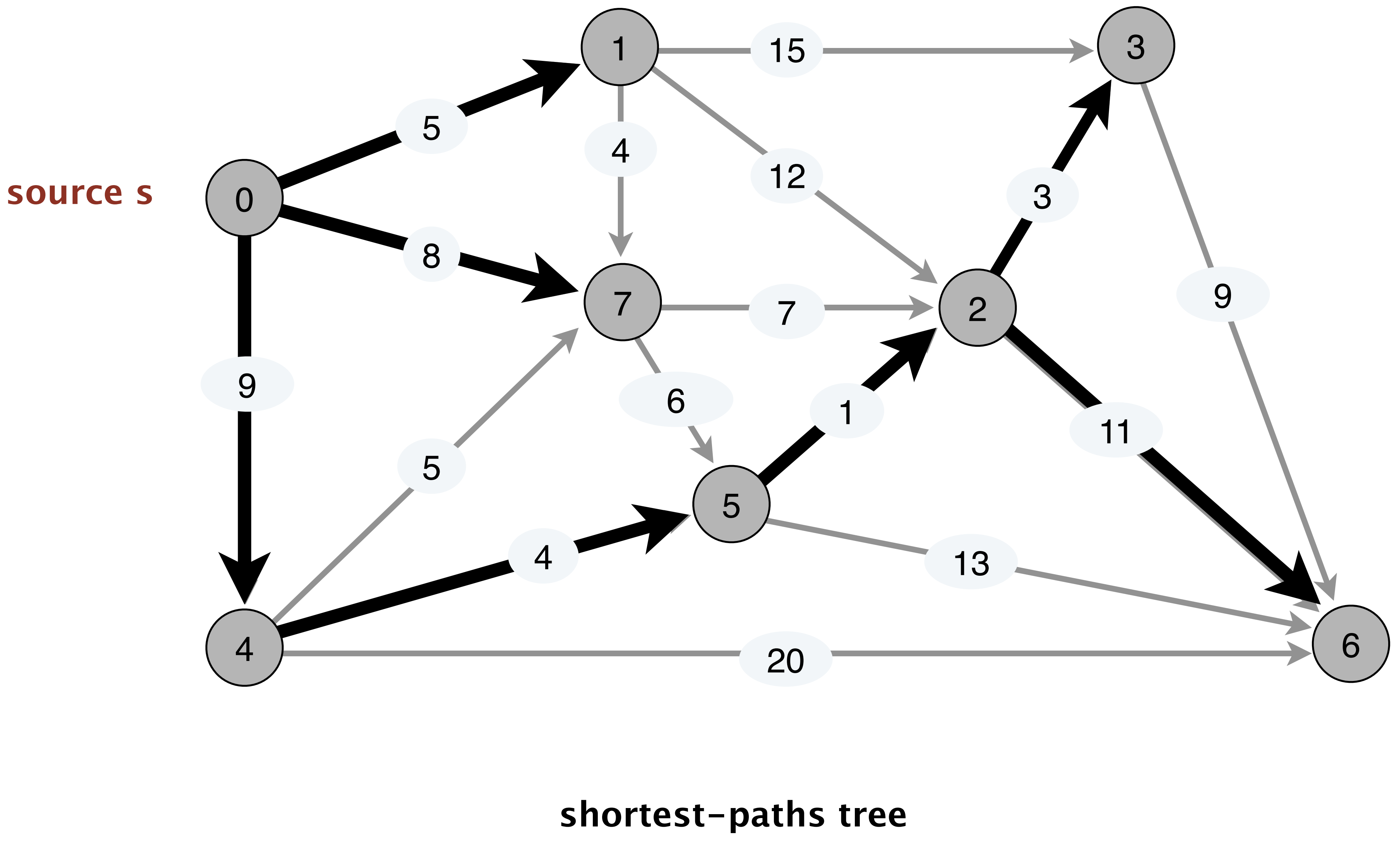
Shortest Paths in Weighted Graph

Problem. Given a directed graph $G = (V, E)$ with positive edge weights: that is, each edge $e \in E$ has a positive weight $w(e)$ and vertices s and t, find the shortest path from s to t.

The shortest path from s to t in a weighted graph is a path P from s to t (or a s-t path) with minimum weight $w(P) = \sum_{e \in P} w(P)$.

Single-Source Shortest Path

Assumption. There exists a path from s to every node in the graph.



Single-Source Shortest Path

Problem. Given a directed graph $G = (V, E)$ with positive edge weights w_e for each $e \in E$ and a source $s \in V$, find a shortest directed path from s to every other vertex in V.

Quick quiz. Which of these changes to edge weights on a graph does not affect the shortest paths?

A. Adding 17

B. Multiplying 17

C. None of the above

Shortest Paths Applications

- Map routing
- Robot navigation
- Texture mapping
- Typesetting in LaTeX.
- Urban traffic planning.
- Scheduling, routing of operators
- Network routing protocols (OSPF, BGP, RIP)
- It is so important that we will revisit shortest paths when we study dynamic programming!

Dijkstra’s Algorithm

Computes the shortest path from s to all vertices

Dijkstra’s algorithm has the following key components

- It evolves a tree, rooted at s, of shortest paths to the vertices closest to s
- It keeps a conservative estimate (that is, over-estimate) $d(u)$ of the shortest path length to vertices u not yet in the tree
- It selects the next vertex to add to the tree based on lowest estimate **(Greedy: choose locally best next move)**

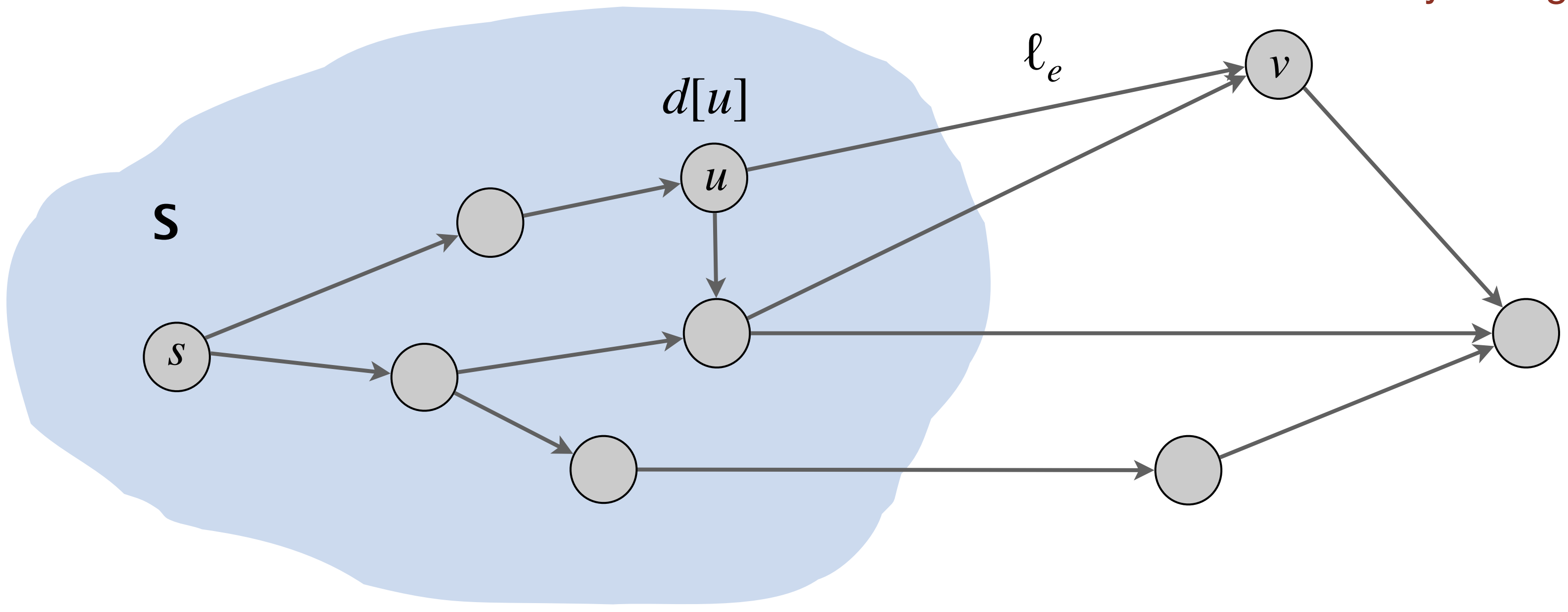
Dijkstra's Algorithm

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \rightsquigarrow u$ path.

- Initialize $S \leftarrow \{s\}$, $d[s] \leftarrow 0$.
- Repeatedly add unexplored node $v \notin S$ which minimizes

$$\min_{e=(u,v):u \in S} \boxed{d[u] + \ell_e}$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$



Dijkstra’s Demo

Dijkstra’s Algorithm

```
procedure Dijkstra(G, s)    # G = (V,E) is connected
   S = {s}; d[s] ← 0         # initialize
   for all neighbors v of s:
       d[v] ← w(s, v)
   for all non-neighbors v of s:
       d[v] ← ∞
   while S not equal to V:
       select v from V - S with minimum d[v]
       Add v to S
       for each neighbor u of v in V - S:
            if d[v] + w(v, u) < d[u]: # better path
                d[u] = d[v] + w(v, u) # update
```

Every time a node is added to S, we update the estimate of all its neighbors!

Dijkstra’s Algorithm (with Tree)

```
procedure Dijkstra(G, s)    # G = (V,E) is connected
    T = ∅; S = {s}; d[s] ← 0        # initialize
    for all neighbors v of s:
        d[v] ← w(s, v); pred[v] ← s
    for all non-neighbors v of s:
       d[v] ← ∞
    while S not equal to V:
          select v from V - S with minimum d[v]
          Add v to S;  add (v, pred[v]) to T
          for each neighbor u of v in V - S:
              if d[v] + w(v, u) < d[u]:   # better path
                  d[u] = d[v] + w(v, u)
                  pred[u] ← v
```

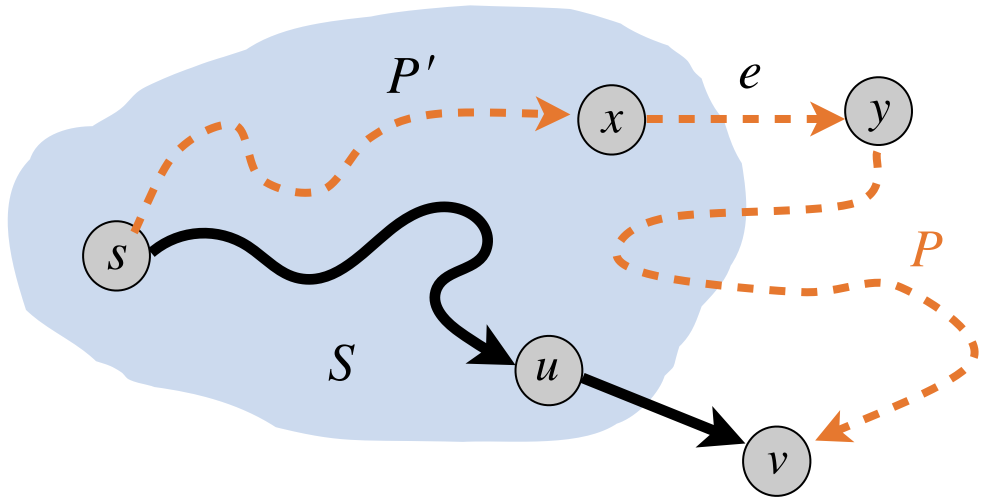
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$.

We now grow S to size $k + 1$: let v be next node added to S

- Goal is to show that $d[v]$ is the length of the shortest s-v path
- One way to show this:
  - Consider an arbitrary path s-v path P in the graph
  - Sufficient to show $w(P) \geq d[v]$



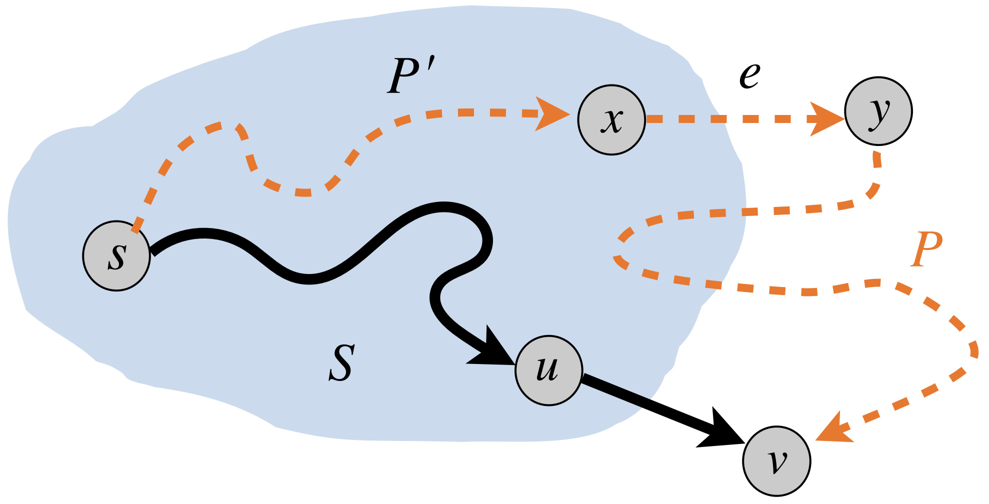
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$].

Inductive Step: Adding node v to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (why?)



Non-negative weights

$$w(P) \geq w(P') + w_e$$

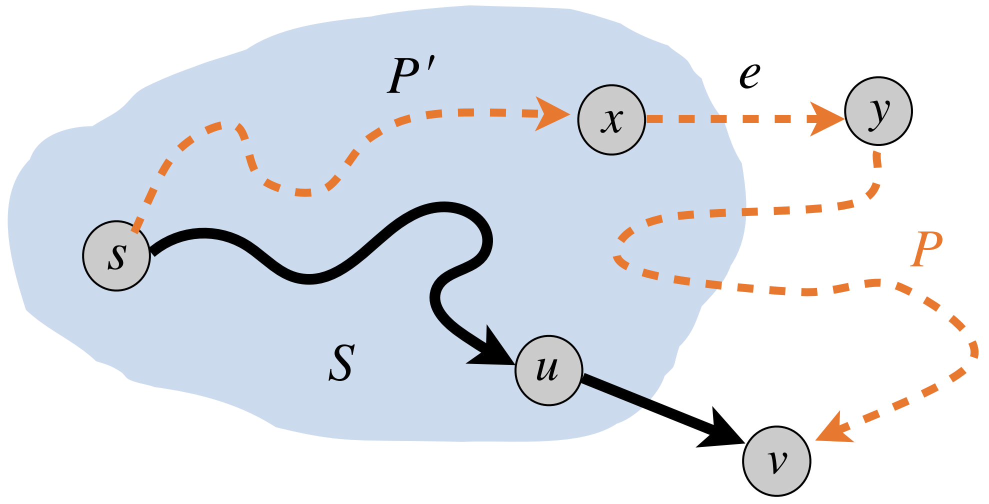
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$].

Inductive Step: Adding node v to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



Inductive Hypothesis

$$w(P) \geq w(P') + w_e \geq d[x] + w_e$$

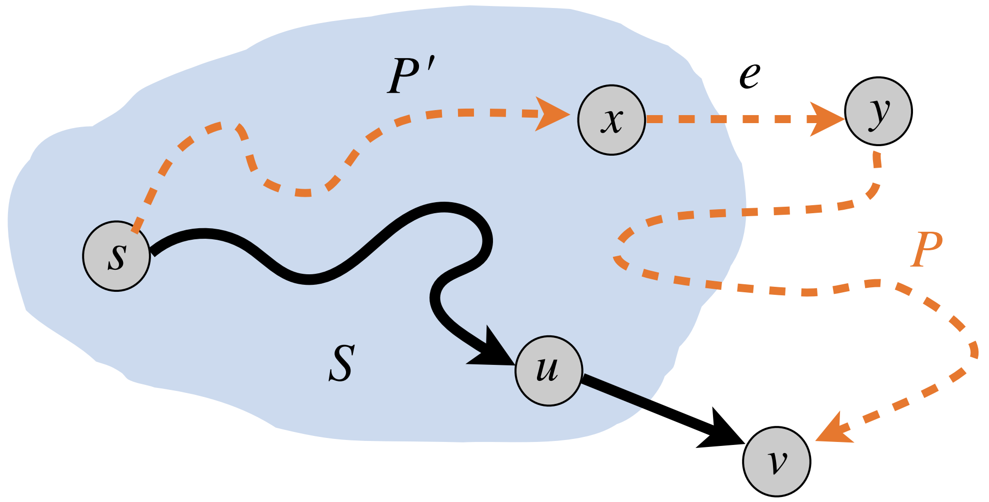
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$].

Inductive Step: Adding node v to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



When x was added to S, $d[y]$ was updated

$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y]$$

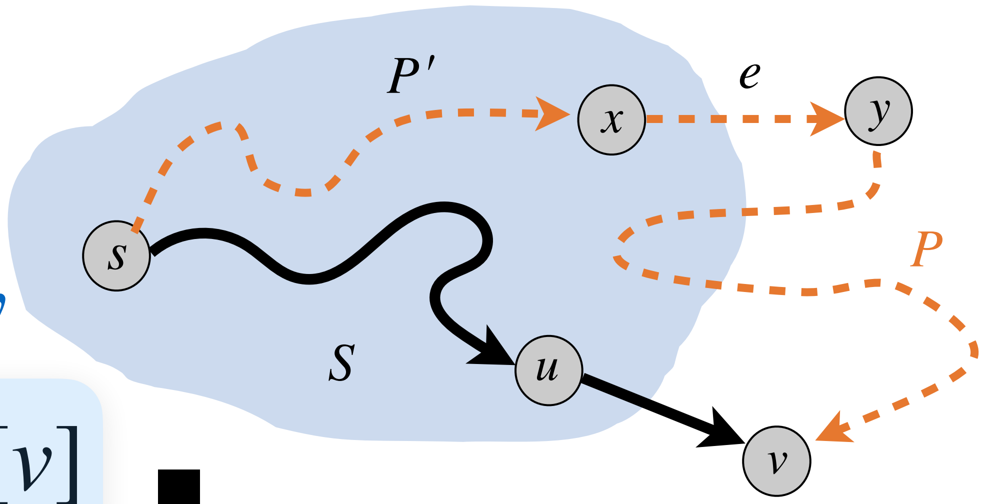
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$].

Inductive Step: Adding node v to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)

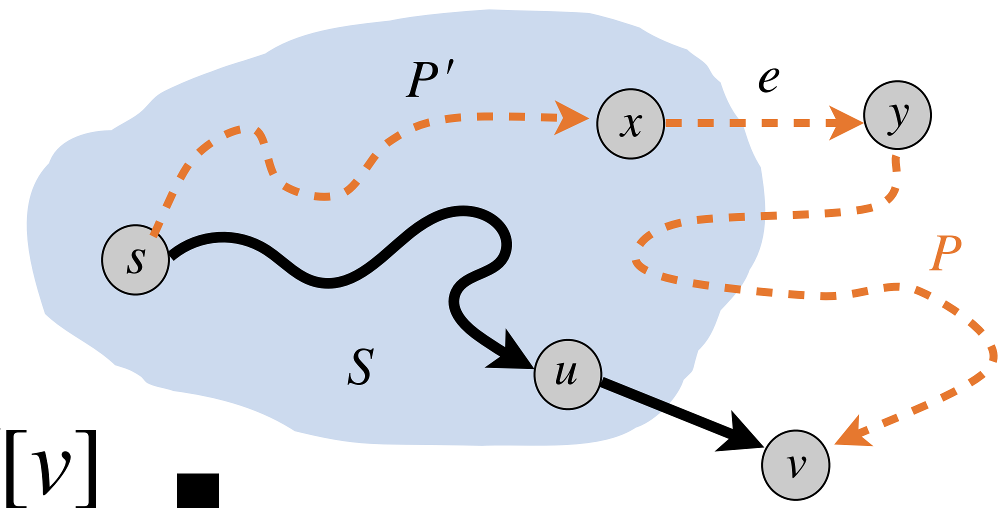


Dijkstra chose to add v instead of y

$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v] \quad \blacksquare$$

Negative Weights

- Dijkstra's only works for non-negative edge weights
- Where does the argument below break down if weights are negative?
- How do we handle negative weights?
  - We will study a dynamic-programming based approach
  - Called Bellman-Ford algorithm



$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v] \quad \blacksquare$$

Implementation & Running Time

- We perform the while loop once for each vertex: n iterations
- Within each iteration:
    - **Main step.** Find & delete unvisited vertex v with min $d[v]$ among all v that are neighbors of elements in S
        - We could do this in $O(m)$ by checking all edges
    - Update $d[u]$ for each neighbor of v: $O(\text{outdegree}(v))$
- Other bookkeeping:
    - Maintain set of visited S and unvisited vertices $V - S$:
    - Maintain a tree of edges $(v, (\text{pred}[v])$
- Naively we can implement in $O(nm)$ time

Extract Min

Decrease Key

Priority Queue:
Binary Heap (Review)

CS136 Review: Priority Queue

Managing such a set typically involves the following operations on S

- **Insert**. Insert a new element into S
- **Delete**. Delete an element from S
- **ExtractMin.** Retrieve highest priority element in S

Priorities are encoded as a ‘key’ value

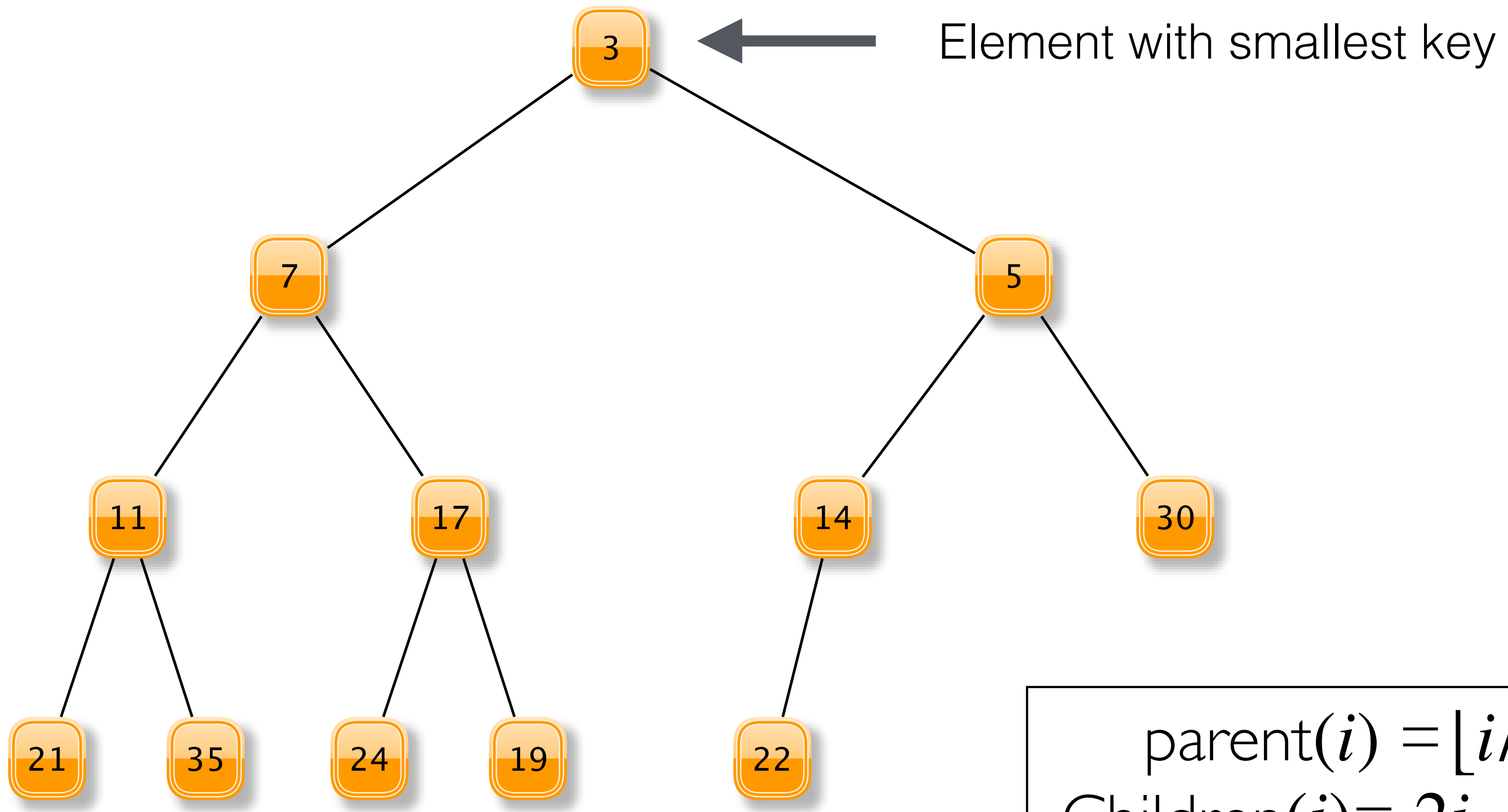
Typically: higher priority <—> lower key value

Heap as Priority Queue. Combines tree structure with array access

- Heap-ordered complete binary tree (implicit)
- Insert and delete: $O(\log n)$ time (‘tree’ traversal & moves)
- **Extract min.** Delete item with minimum key value: $O(\log n)$

Heap Example

Heap property: For every element v, at node i, the element w at i's parent satisfies key$(w) \leq$ key(v)



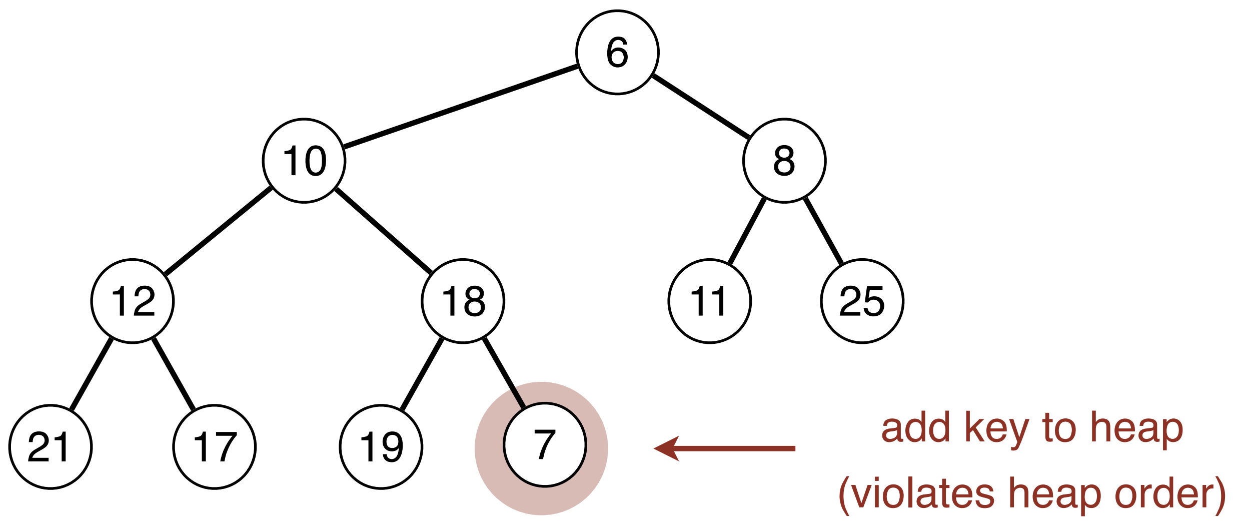
parent$(i) = \lfloor i/2 \rfloor$.
Children$(i) = 2i,\ 2i+1$

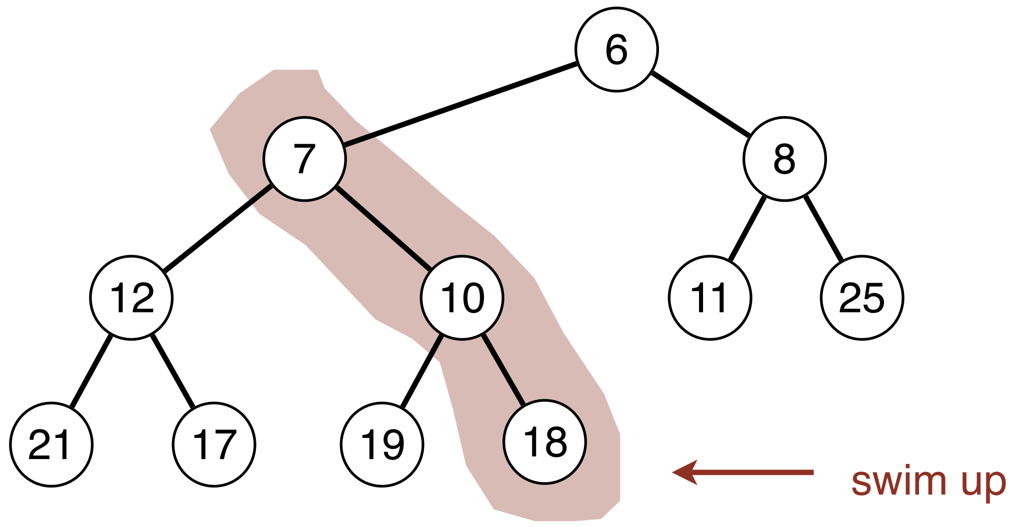
H

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	3	7	5	11	17	14	30	21	35	24	19	22	-	-	-

Binary Heap: Insert

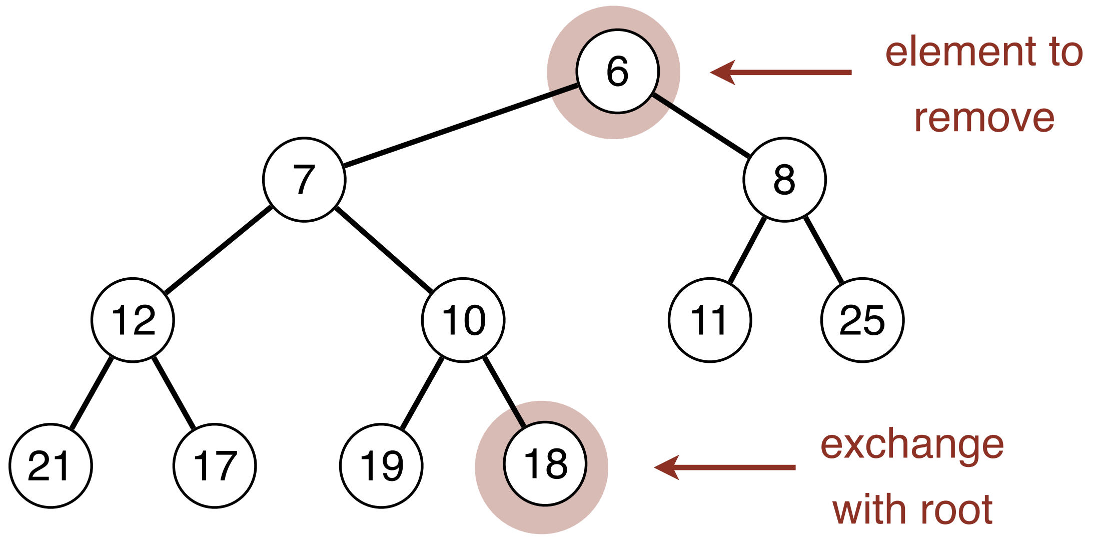
Insert. Add element in new node at end; repeatedly exchange new element with element in its parent until heap order is restored.

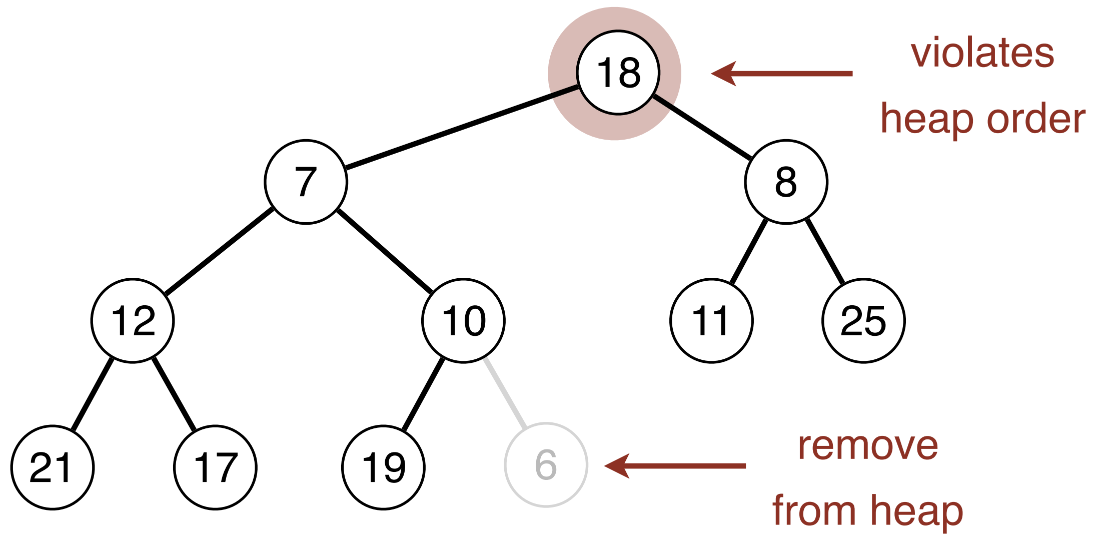


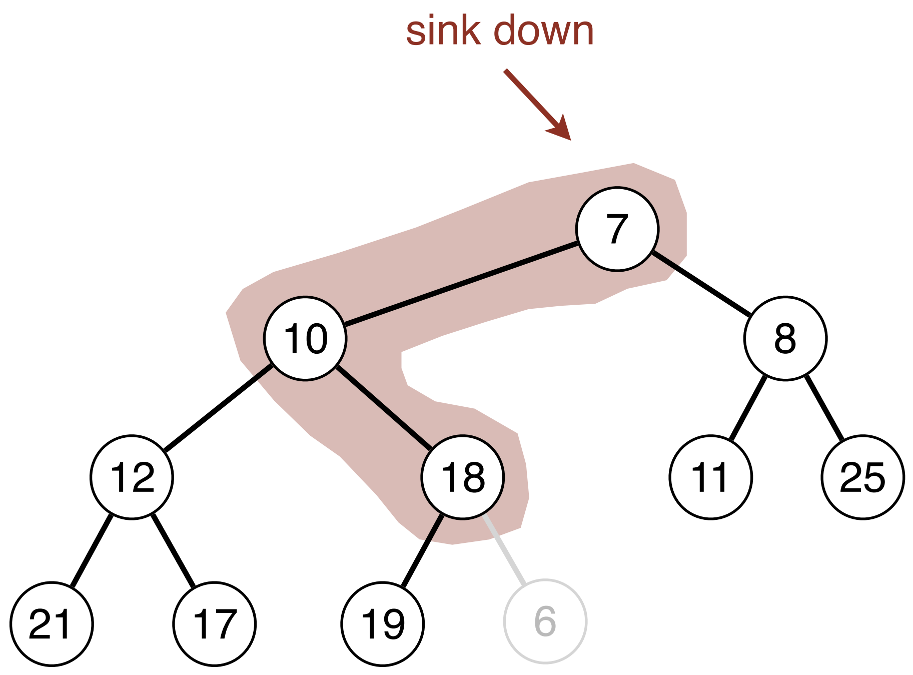


Binary Heap: Extract Min

Extract min. Exchange element in root with last node; repeatedly exchange element with its smaller child until heap order is restored.





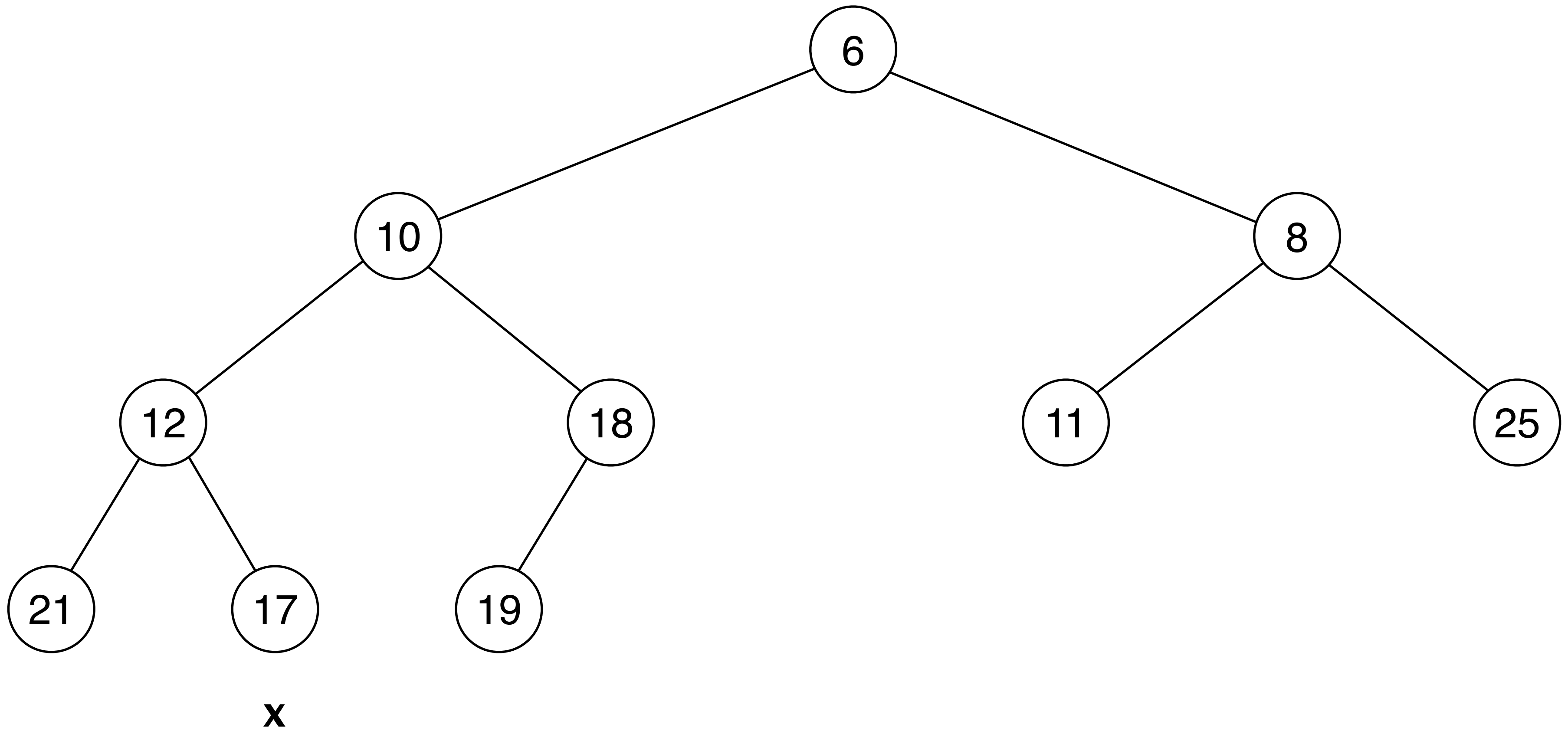


Binary Heap: Decrease Key

Decrease Key. Given the index of a node in the heap, repeatedly exchange element with its parent until heap order is restored.

Need a mapping from the identity of a node to its index

decrease key of node x to 11



Decrease/Change Key

How to to update priorities (perform decrease-key) in the priority queue efficiently?

- Recall vertices are represented by $1, \ldots, n$
- Maintain an array `PQIndex[1..n]` that holds the index of each vertex v in the priority queue
- **(Decrease-min)** If we update $d[u]$ for some u, we then heapify-up from u's location in the PQ to restore heap property
- Every time we swap two heap elements, we update `PQIndex` for the two vertices

PQ Summary

Given a priority with n items in it, we can perform the following operations efficiently:

- Insert and deletes: $O(\log n)$
- Extract min: $O(\log n)$
    - $O(1)$ to just find minimum, $O(\log n)$ to delete and maintain heap property
- Decrease key: $O(\log n)$

Back to Dijkstra's Analysis

Running Time Analysis

Running Time: Traversal of S (each edge visited at most once)

  - $O(n \log n + m)$
  - Why the $O(\log n)$?
  - n deleteMin operations from PQ to select next vertex $O(n \log n)$

- Construction of T: time proportional to its size: $O(n)$
- Creation of priority queue: $O(n)$
- At most one decrease-key for each edge: $O(m \log n)$

Total time: $O((n + m)\log n)$. This is $O(m \log n)$ if G is connected

Space: $O(n + m)$

What About Undirected Graphs

How to solve the single-source shortest paths problem in undirected graphs with positive edge lengths?

(a) Replace each undirected edge with two antiparallel edges of same length and run Dijkstra's algorithm on the resulting digraph

(b) Modify Dijkstra's algorithms so that when it processes node u, it consider all edges incident to u (instead of edges leaving u)

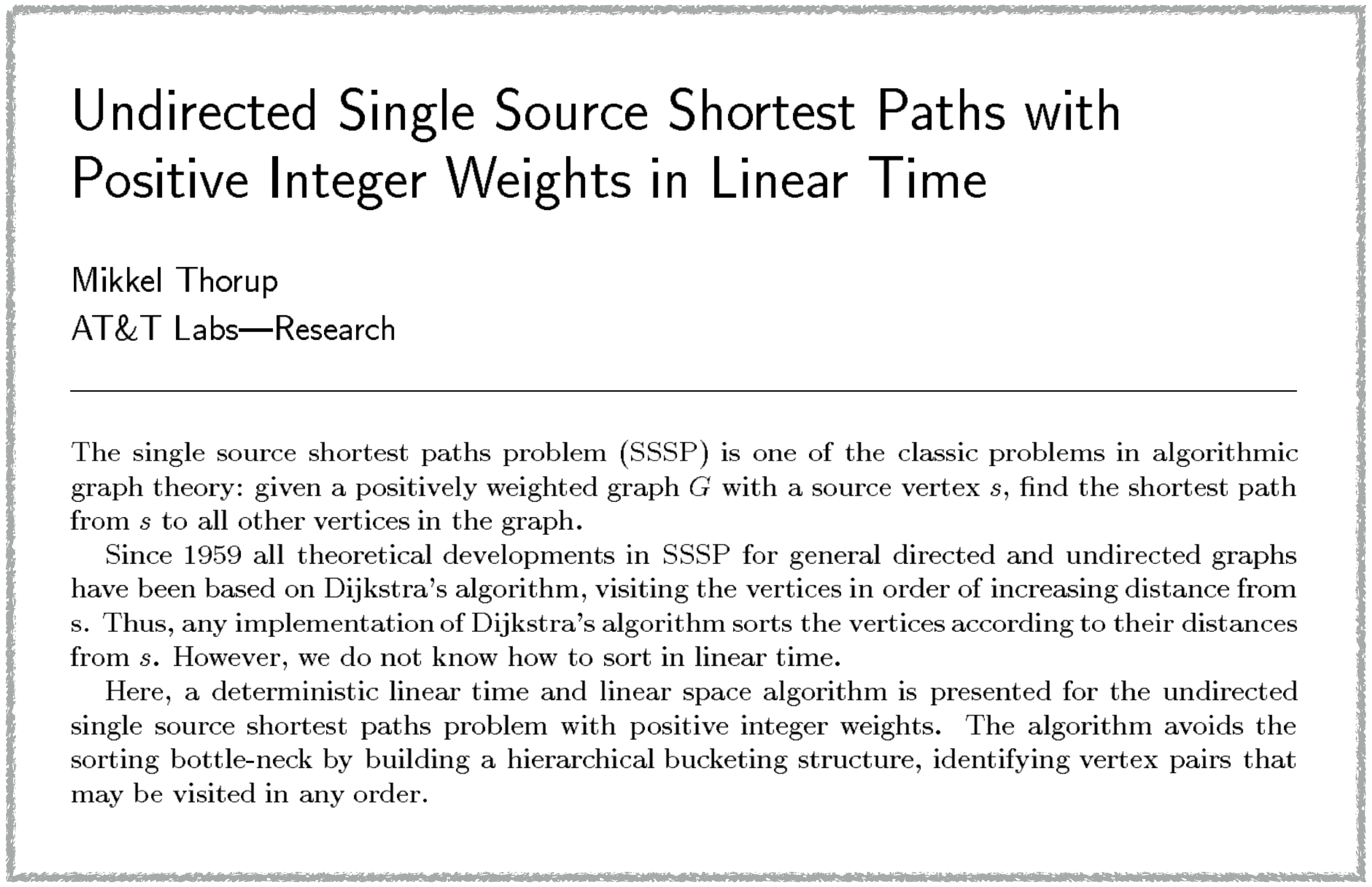
(c) Either A or B

(d) Neither A nor B

Shortest Path in Linear Time

[Thorup 1999] Can solve single-source shortest paths problem in undirected graphs with positive integer edge lengths in $O(m)$ time.

Remark. Does not explore vertices in increasing distance from s



Undirected Single Source Shortest Paths with Positive Integer Weights in Linear Time

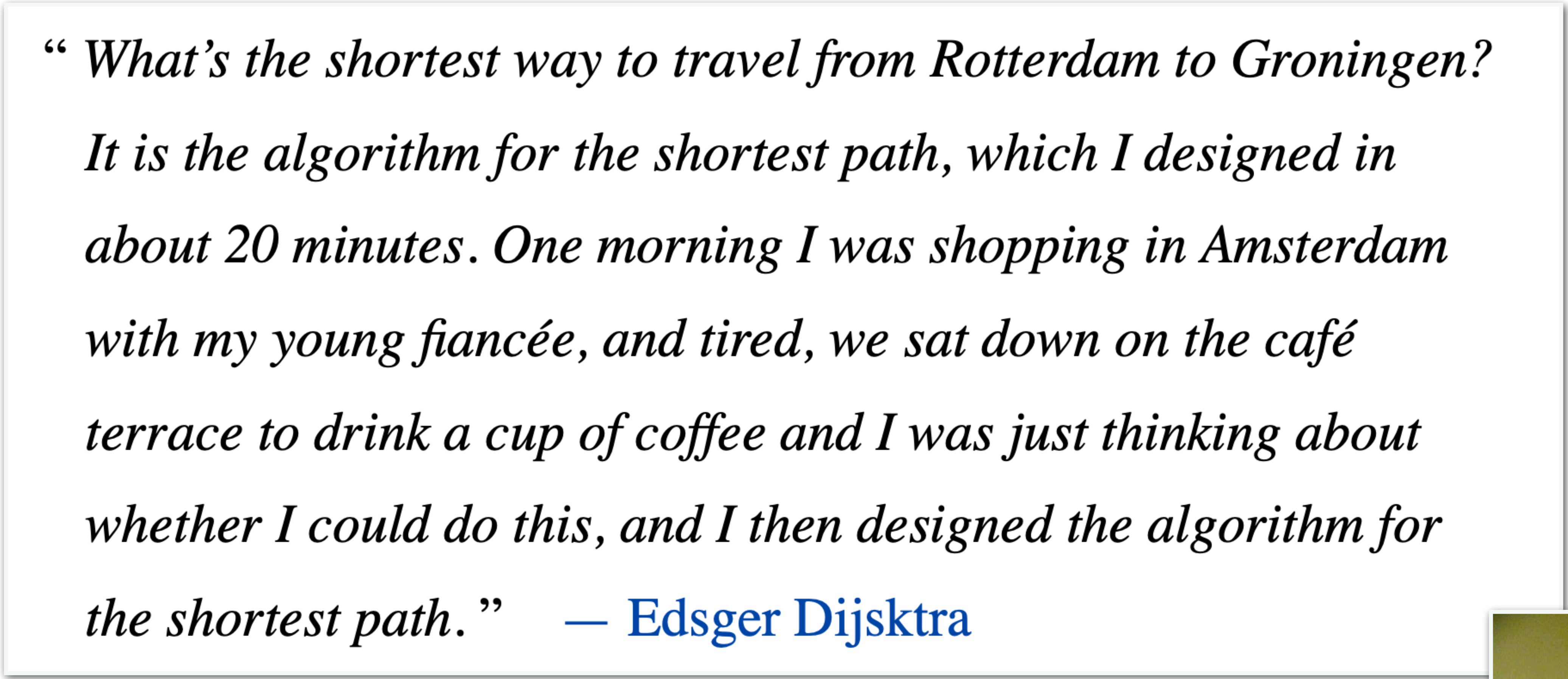
Mikkel Thorup
AT&T Labs—Research

The single source shortest paths problem (SSSP) is one of the classic problems in algorithmic graph theory: given a positively weighted graph G with a source vertex s, find the shortest path from s to all other vertices in the graph.

Since 1959 all theoretical developments in SSSP for general directed and undirected graphs have been based on Dijkstra's algorithm, visiting the vertices in order of increasing distance from s. Thus, any implementation of Dijkstra's algorithm sorts the vertices according to their distances from s. However, we do not know how to sort in linear time.

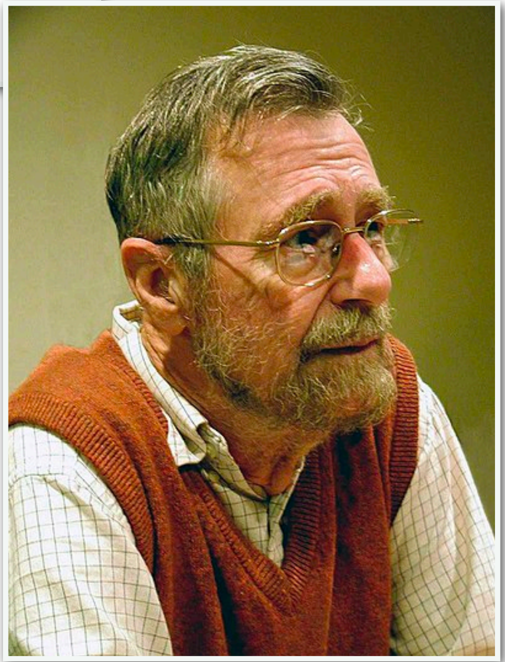
Here, a deterministic linear time and linear space algorithm is presented for the undirected single source shortest paths problem with positive integer weights. The algorithm avoids the sorting bottle-neck by building a hierarchical bucketing structure, identifying vertex pairs that may be visited in any order.

Edsger Dijkstra (1930-2002)



" What's the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. 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 was just thinking about whether I could do this, and I then designed the algorithm for the shortest path." — Edsger Dijsktra

- Shortest-path algorithm was actually discovered independently (around 1956) by a bunch of different people (read Jeff Erickson's description and Strigler's law in CS). *"Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-SeitzDantzig-Dijkstra-Minty-Whiting-Hillier algorithm"*



Recap: Greedy Algorithms

- Scheduling non-conflicting jobs (intervals)
    - Earliest finish-times first
- Scheduling with deadlines to maximize lateness of jobs
    - Earliest-deadline first
- Exchange argument to prove correctness
- Minimum spanning trees: greedily pick edges
    - Cut property: essentially a non-local exchange argument
    - Prims, Kruskals: correctness from cut property
    - Union find data structure
- Djisktra's shortest path: greedily find paths

Next Algorithmic Design Paradigm: Divide and Conquer (Recursion)

Now: Warm Up
A Fun Algorithmic Problem!

Whose solution has a useful algorithmic design trick !

Exercise: House Finding Problem

- Suppose we live in a straight line world
- You need to find your friends house
- You know that it is at most n steps away from your current location (say 0)
- But you don’t know what n is
- And you don’t know if their house is on your left or on your right
- How fast can you find their house?

House Finding: First Strategy

- Initialize $k = 1$
- Suppose you start at point 0)
- While house not found:
    - Walk from 0 to $-k$
    - Walk back to k
    - Walk back to 0
    - $k \leftarrow k + 1$

How good is this?

- $4\ (1 + 2 + 3 + \ldots + n)$
- $O(n^2)$

Can we do better?

House Finding: Better Strategy

- Initialize $k = 1$
- Suppose you start at point 0)
- While house not found:
    - Walk from 0 to $-k$
    - Walk back to k
    - Walk back to 0
    - $k \leftarrow 2k$

Double our guess! (Called **repeated doubling**)

How good is this?

- $4\ (1 + 2 + 4 + 8 + \ldots 2^{\ell})$ where $n = 2^{\ell}$ (assumption)
- $O(n)$ (We have a geometric series (bounded by largest term)

Acknowledgments

- The pictures in these slides are taken from
    - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
    - Jeff Erickson’s Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)