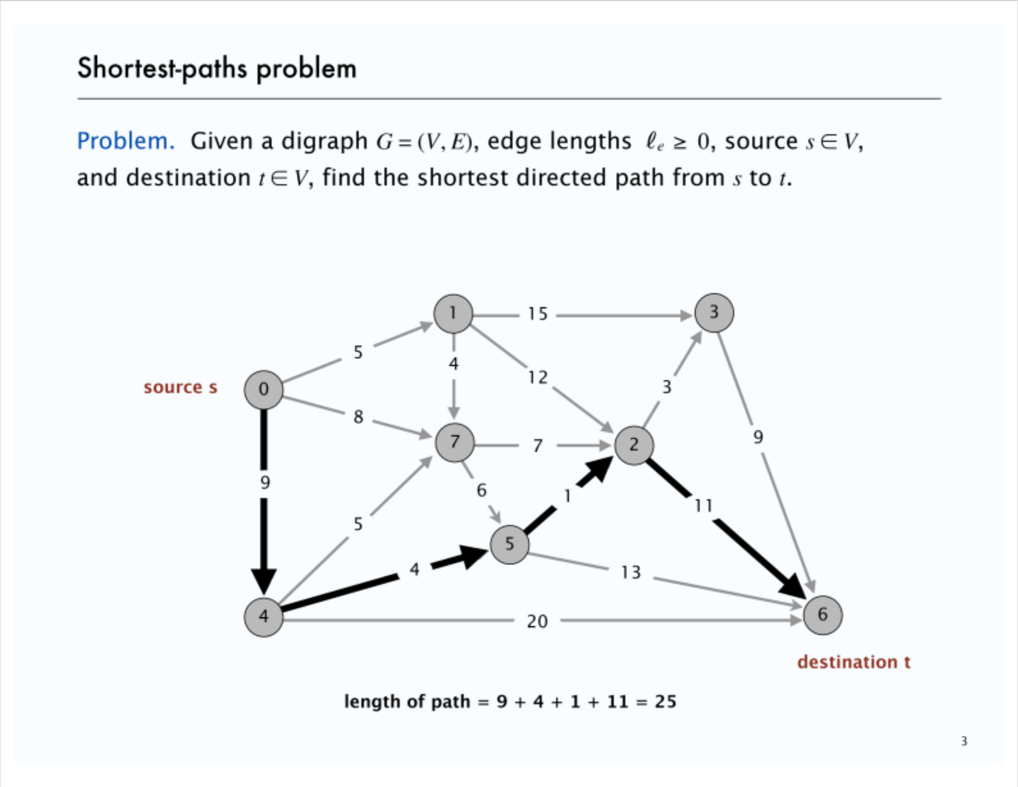
Lecture 5

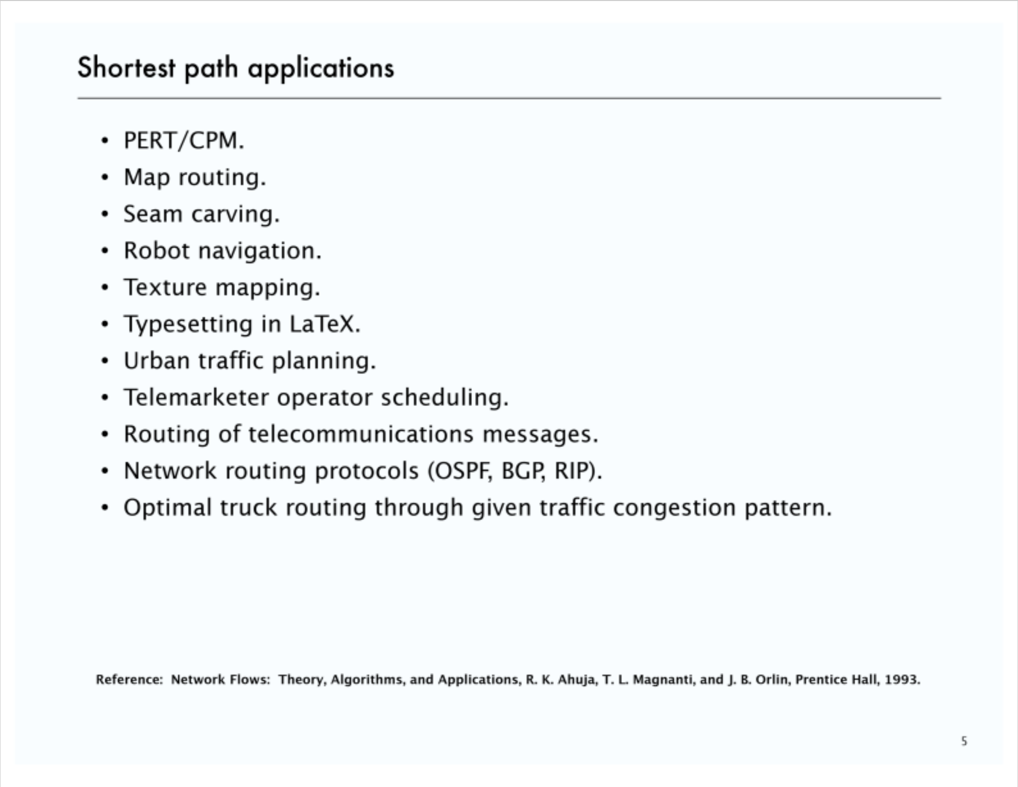
Wednesday, May 17, 2017 12:37 AM



Shortest-paths problem

Problem. Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, and destination $t \in V$, find the shortest directed path from s to t.

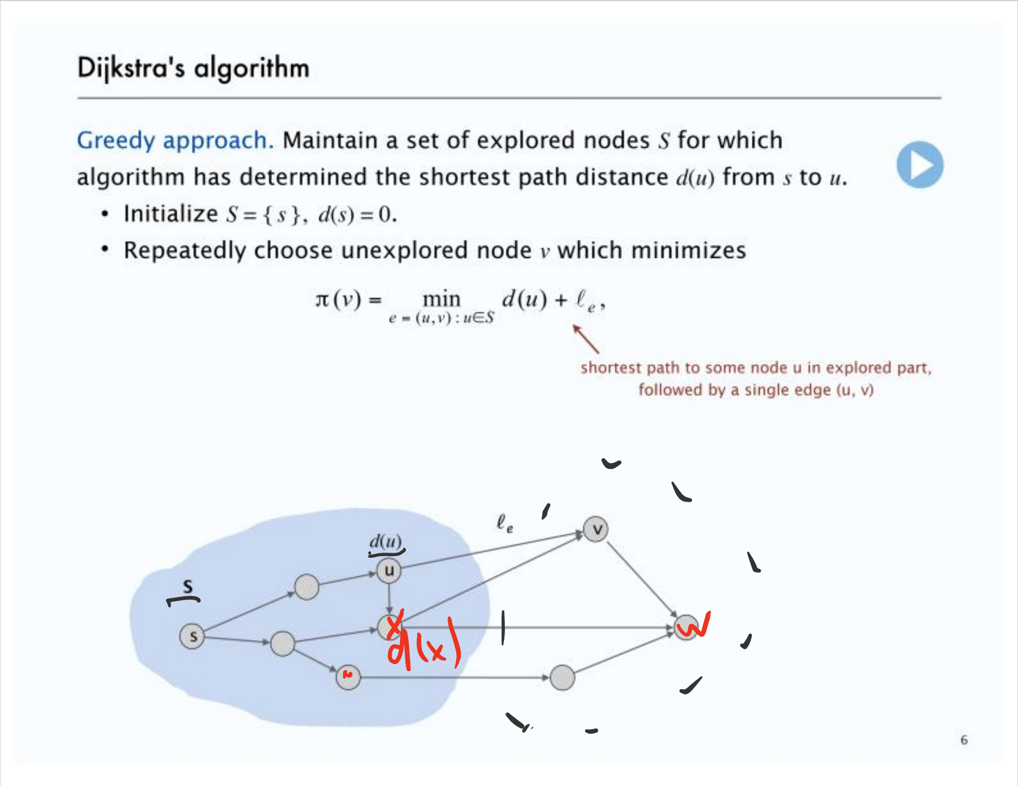
length of path = 9 + 4 + 1 + 11 = 25



Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in LaTeX.
- Urban traffic planning.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.



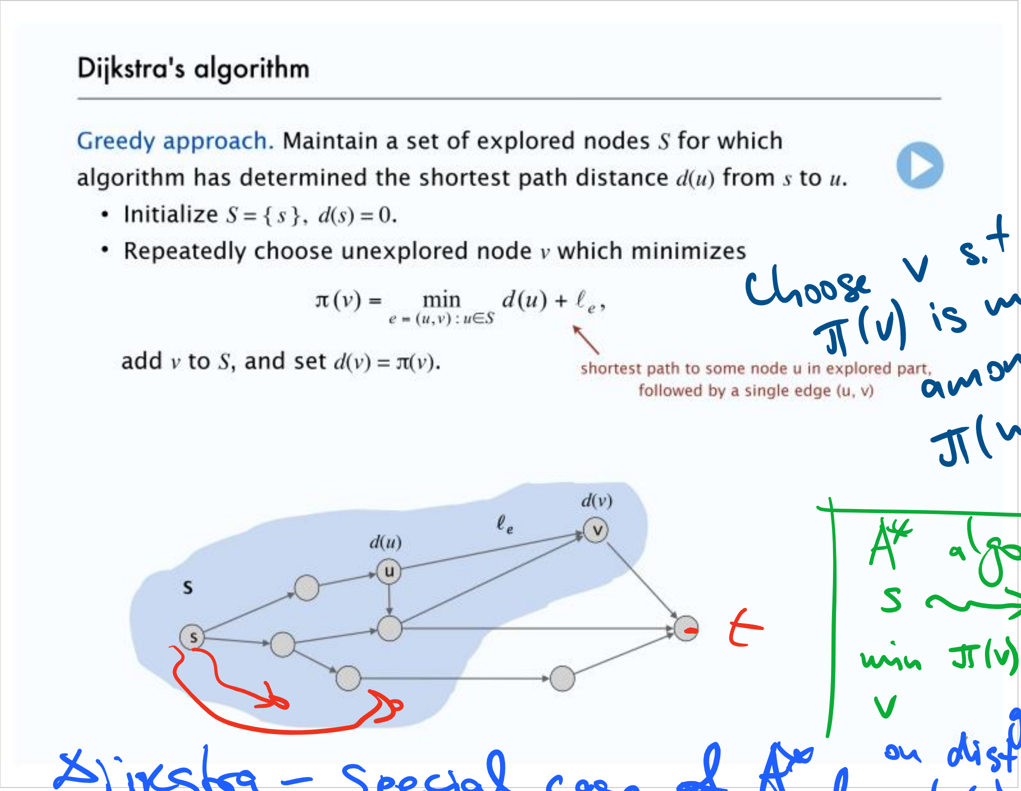
Dijkstra's algorithm

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined the shortest path distance $d(u)$ from s to u.

- Initialize $S = \{ s \}$, $d(s) = 0$.
- Repeatedly choose unexplored node v which minimizes

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

shortest path to some node u in explored part, followed by a single edge (u, v)



Dijkstra's algorithm

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined the shortest path distance $d(u)$ from s to u.

- Initialize $S = \{ s \}$, $d(s) = 0$.
- Repeatedly choose unexplored node v which minimizes

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

add v to S, and set $d(v) = \pi(v)$.

shortest path to some node u in explored part, followed by a single edge (u, v)

Choose v s.t $\pi(v)$ is min among all $\pi(w)$'s

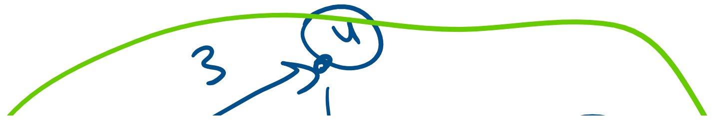
A^* algo

$s \rightsquigarrow t$

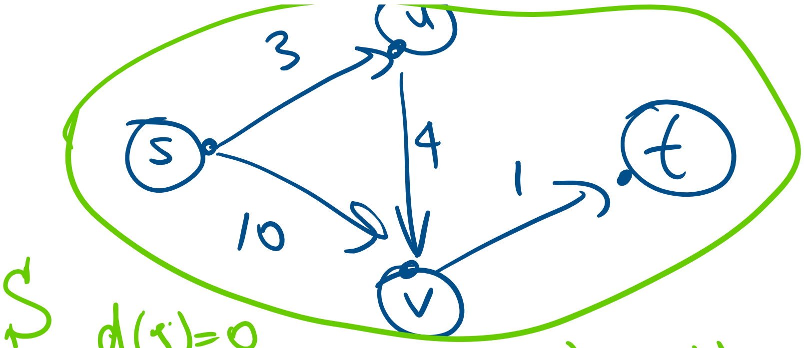
$\min_{v} \pi(v) + h(v)$

guess?? on dist(v,t)

Dijkstra = Special case of A^* for $h(v)=0$

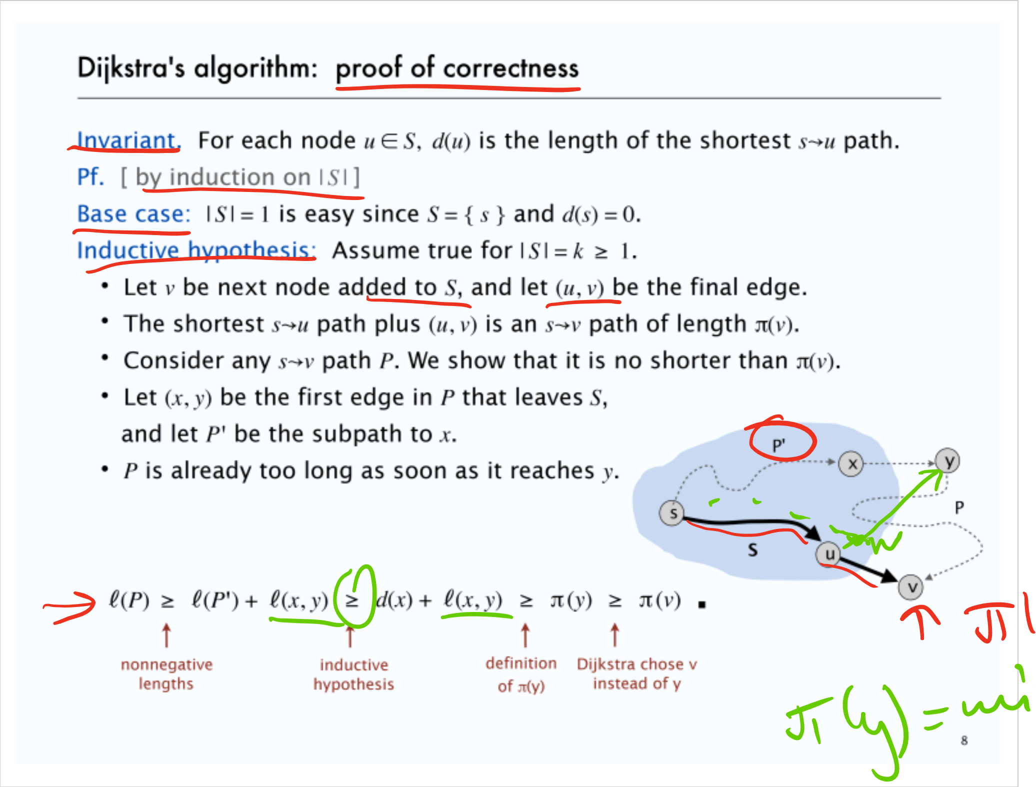


$\pi(t) = d(v)+1$



$\pi(t) = d(v) + 1$
$= 7 + 1 = 8$

$d(s) = 0$
$d(u) = \pi(u) = 3$; $d(v) = \pi(v) = 7$; $d(t) = 8$ ✓



Dijkstra's algorithm: proof of correctness

Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest $s \rightsquigarrow u$ path.

Pf. [by induction on $|S|$]

Base case: $|S| = 1$ is easy since $S = \{ s \}$ and $d(s) = 0$.

Inductive hypothesis: Assume true for $|S| = k \ge 1$.

- Let v be next node added to S, and let (u, v) be the final edge.
- The shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
- Consider any $s \rightsquigarrow v$ path P. We show that it is no shorter than $\pi(v)$.
- Let (x, y) be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it reaches y.

$$\ell(P) \ge \ell(P') + \ell(x, y) \ge d(x) + \ell(x, y) \ge \pi(y) \ge \pi(v) \ \blacksquare$$

nonnegative lengths

inductive hypothesis

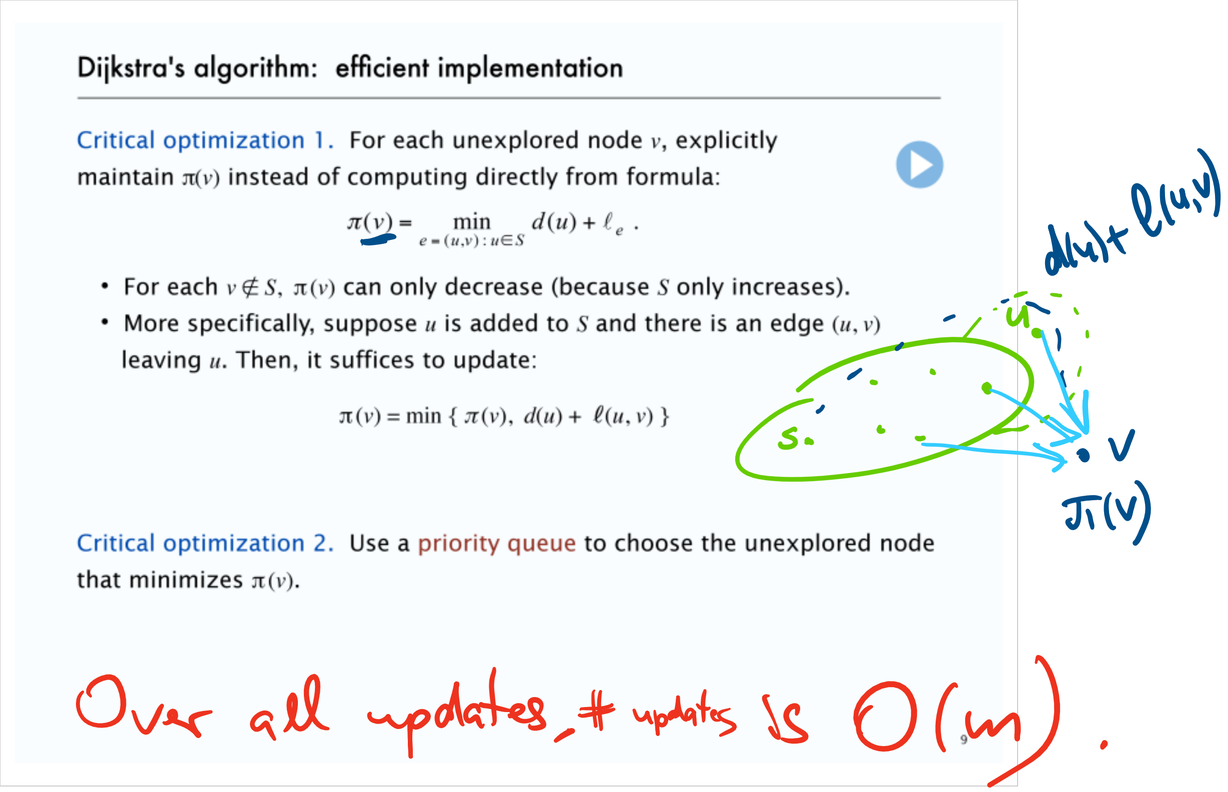
definition of $\pi(y)$

Dijkstra chose v instead of y

possibly $(y = v)$

$\pi(v) \overset{?}{=} d(v)$

$\pi(y) = \min\{ d(x) + \ell(x,y),$
$d(w) + \ell(w,y) \}$



Dijkstra's algorithm: efficient implementation

Critical optimization 1. For each unexplored node v, explicitly maintain $\pi(v)$ instead of computing directly from formula:

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e .$$

- For each $v \notin S$, $\pi(v)$ can only decrease (because S only increases).
- More specifically, suppose u is added to S and there is an edge (u, v) leaving u. Then, it suffices to update:

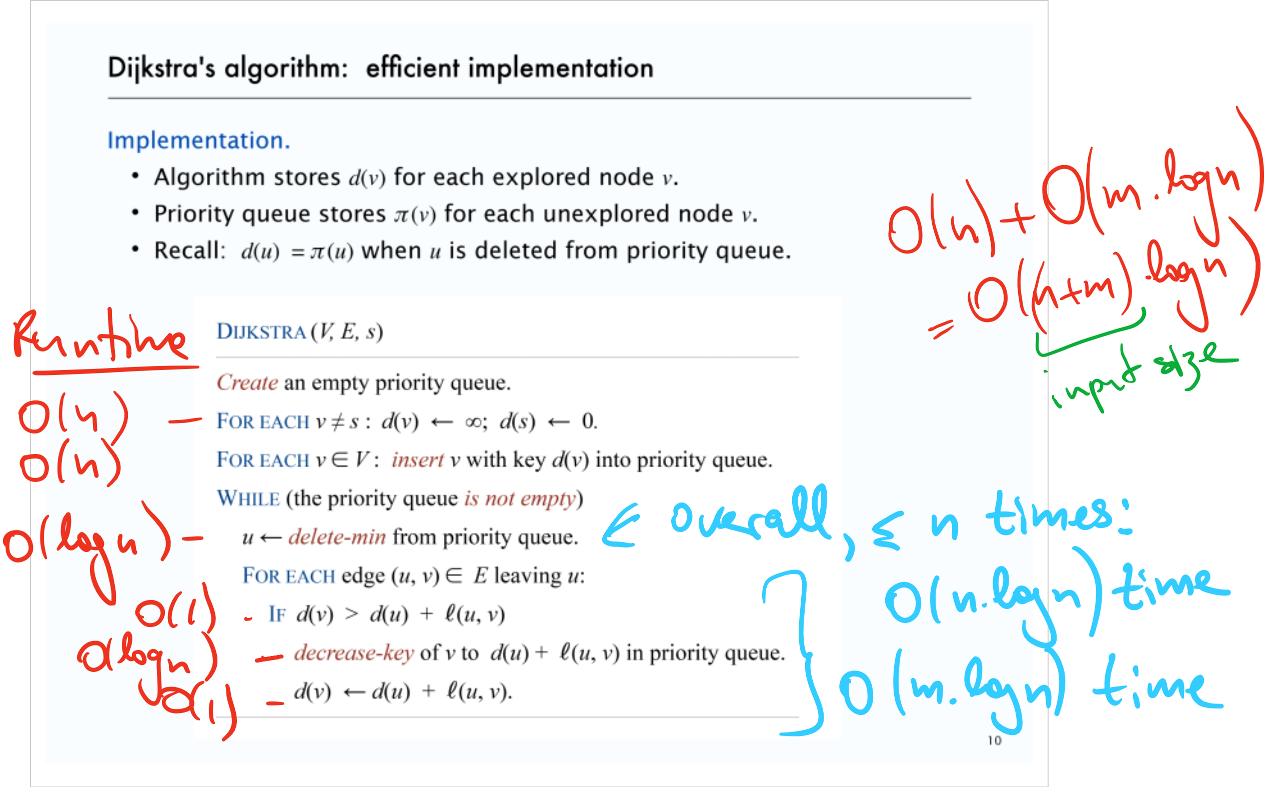
$$\pi(v) = \min \{ \pi(v),\ d(u) + \ell(u, v) \}$$

Critical optimization 2. Use a priority queue to choose the unexplored node that minimizes $\pi(v)$.

$d(u) + \ell(u,v)$

$\pi(v)$

Over all updates, # updates is $O(m)$.



Dijkstra's algorithm: efficient implementation

Implementation.

- Algorithm stores $d(v)$ for each explored node v.
- Priority queue stores $\pi(v)$ for each unexplored node v.
- Recall: $d(u) = \pi(u)$ when u is deleted from priority queue.

```
DIJKSTRA (V, E, s)
Create an empty priority queue.
FOR EACH $v \neq s$ : $d(v) \leftarrow \infty$; $d(s) \leftarrow 0$.
FOR EACH $v \in V$ : insert $v$ with key $d(v)$ into priority queue.
WHILE (the priority queue is not empty)
    $u \leftarrow$ delete-min from priority queue.
    FOR EACH edge $(u, v) \in E$ leaving $u$:
        IF $d(v) > d(u) + \ell(u, v)$
            decrease-key of $v$ to $d(u) + \ell(u, v)$ in priority queue.
            $d(v) \leftarrow d(u) + \ell(u, v)$.
```

Runtime

$O(n)$ – FOR EACH $v \neq s$

$O(n)$ – FOR EACH $v \in V$

$O(\log n)$ – delete-min

$O(1)$ – IF

$O(\log n)$ – decrease-key

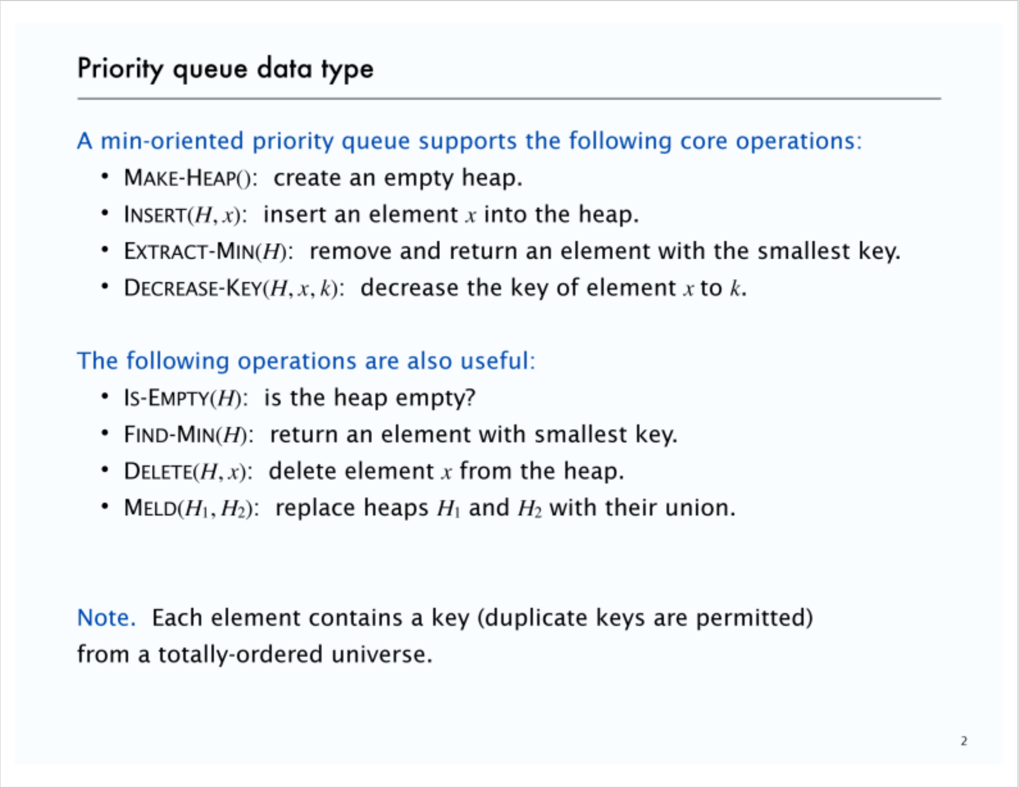
$O(1)$ – $d(v) \leftarrow d(u) + \ell(u, v)$

← overall, ≤ n times: $O(n \cdot \log n)$ time

} $O(m \cdot \log n)$ time

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

$= O(\underbrace{(n+m)}_{\text{input size}} \cdot \log n)$



Priority queue data type

A min-oriented priority queue supports the following core operations:

- MAKE-HEAP(): create an empty heap.
- INSERT(H, x): insert an element x into the heap.
- EXTRACT-MIN(H): remove and return an element with the smallest key.
- DECREASE-KEY(H, x, k): decrease the key of element x to k.

The following operations are also useful:

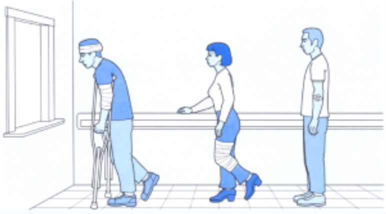
- IS-EMPTY(H): is the heap empty?
- FIND-MIN(H): return an element with smallest key.
- DELETE(H, x): delete element x from the heap.
- MELD(H_1, H_2): replace heaps H_1 and H_2 with their union.

Note. Each element contains a key (duplicate keys are permitted) from a totally-ordered universe.

Priority queue applications

Applications.

- A* search.
- Heapsort.
- Online median.
- Huffman encoding.
- Prim's MST algorithm.
- Discrete event-driven simulation.
- Network bandwidth management.
- Dijkstra's shortest-paths algorithm.
- …

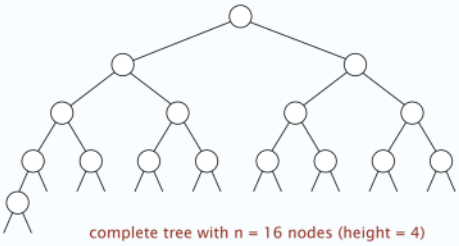


http://younginc.site11.com/source/5895/fos0092.html

Complete binary tree

Binary tree. Empty or node with links to two disjoint binary trees.

Complete tree. Perfectly balanced, except for bottom level.



complete tree with n = 16 nodes (height = 4)

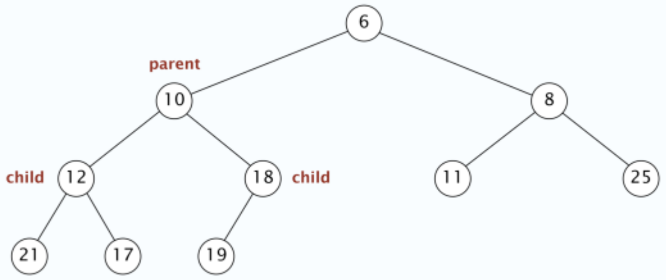
Property. Height of complete binary tree with n nodes is $\lfloor \log_2 n \rfloor$.

Pf. Height increases (by 1) only when n is a power of 2. ▪

Binary heap

Binary heap. Heap-ordered complete binary tree.

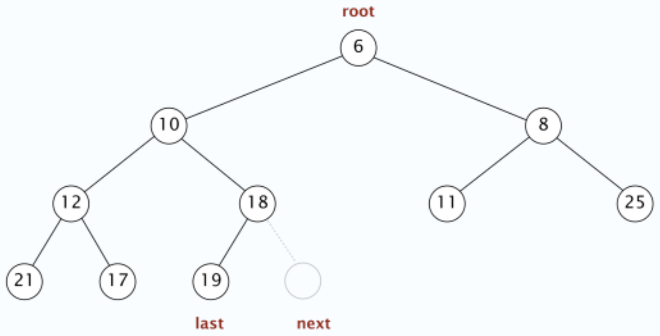
Heap-ordered tree. For each child, the key in child ≥ key in parent.



Explicit binary heap

Pointer representation. Each node has a pointer to parent and two children.

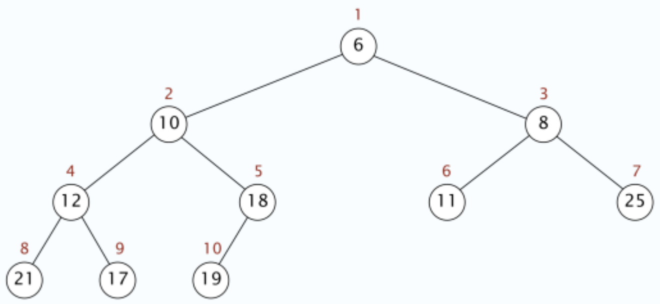
- Maintain number of elements n.
- Maintain pointer to root node.
- Can find pointer to last node or next node in $O(\log n)$ time.



Implicit binary heap

Array representation. Indices start at 1.

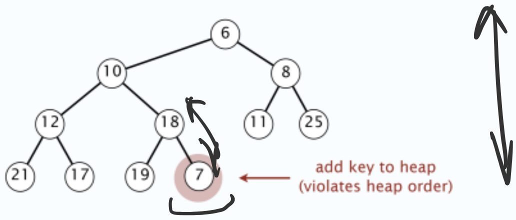
- Take nodes in level order.
- Parent of node at k is at $\lfloor k/2 \rfloor$.
- Children of node at k are at $2k$ and $2k+1$.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
6	10	8	12	18	11	25	21	17	19						

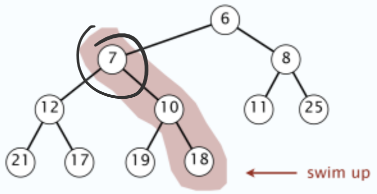
Binary heap: insert

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



add key to heap (violates heap order)

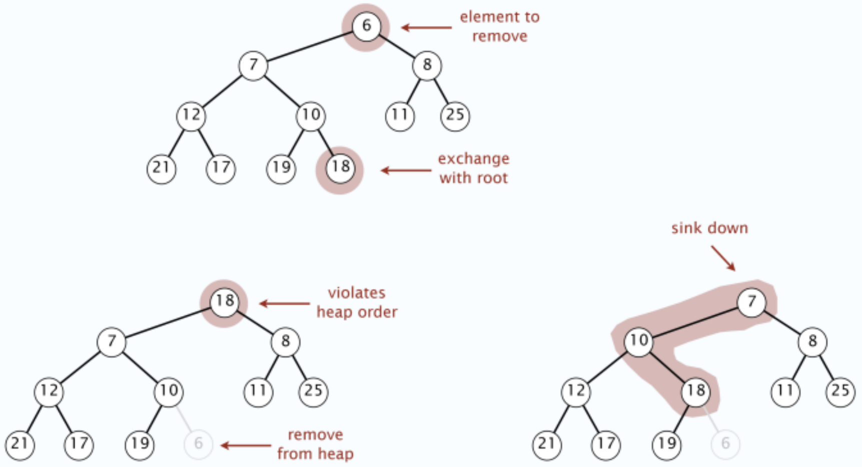
log n



swim up

Binary heap: extract the minimum

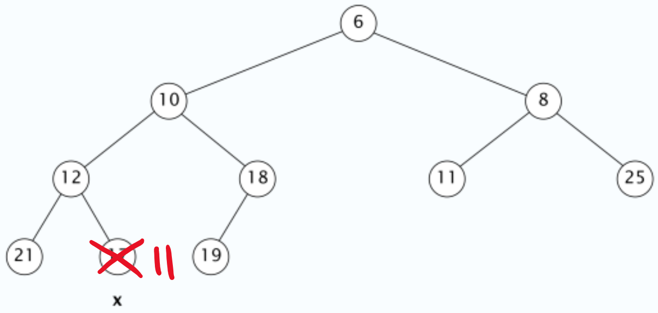
Extract min. Exchange element in root node with last node; repeatedly exchange element in root with its smaller child until heap order is restored.



Binary heap: decrease key

Decrease key. Given a handle to node, repeatedly exchange element with its parent until heap order is restored.

decrease key of node x to 11



Binary heap: analysis

Theorem. In an implicit binary heap, any sequence of m Insert, Extract-Min, and Decrease-Key operations with n Insert operations takes $O(m \log n)$ time.

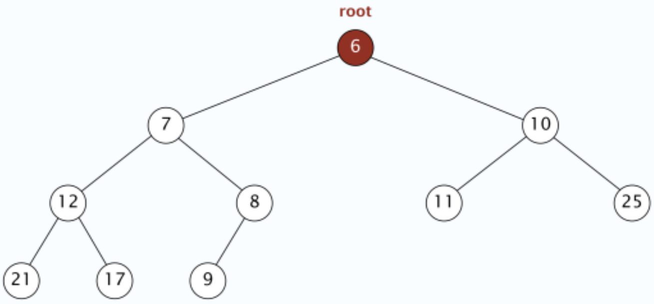
Pf.

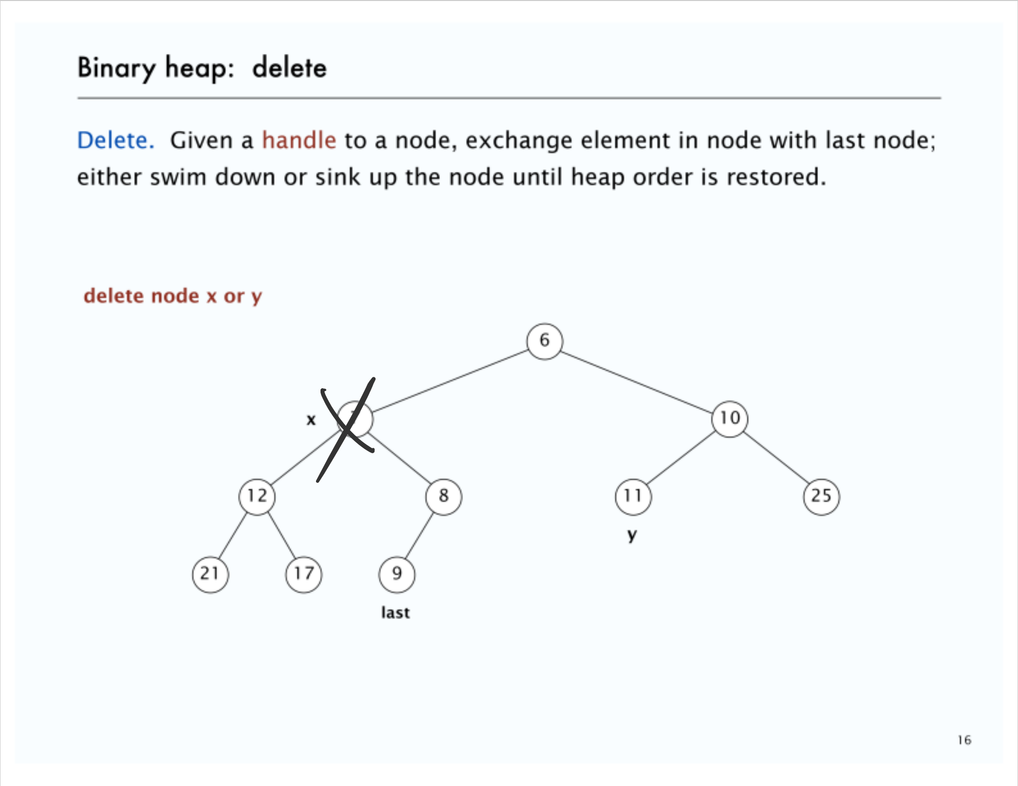
- Each heap op touches nodes only on a path from the root to a leaf; the height of the tree is at most $\log_2 n$.
- The total cost of expanding and contracting the arrays is $O(n)$. ▪

Theorem. In an explicit binary heap with n nodes, the operations Insert, Decrease-Key, and Extract-Min take $O(\log n)$ time in the worst case.

Binary heap: find-min

Find the minimum. Return element in the root node.

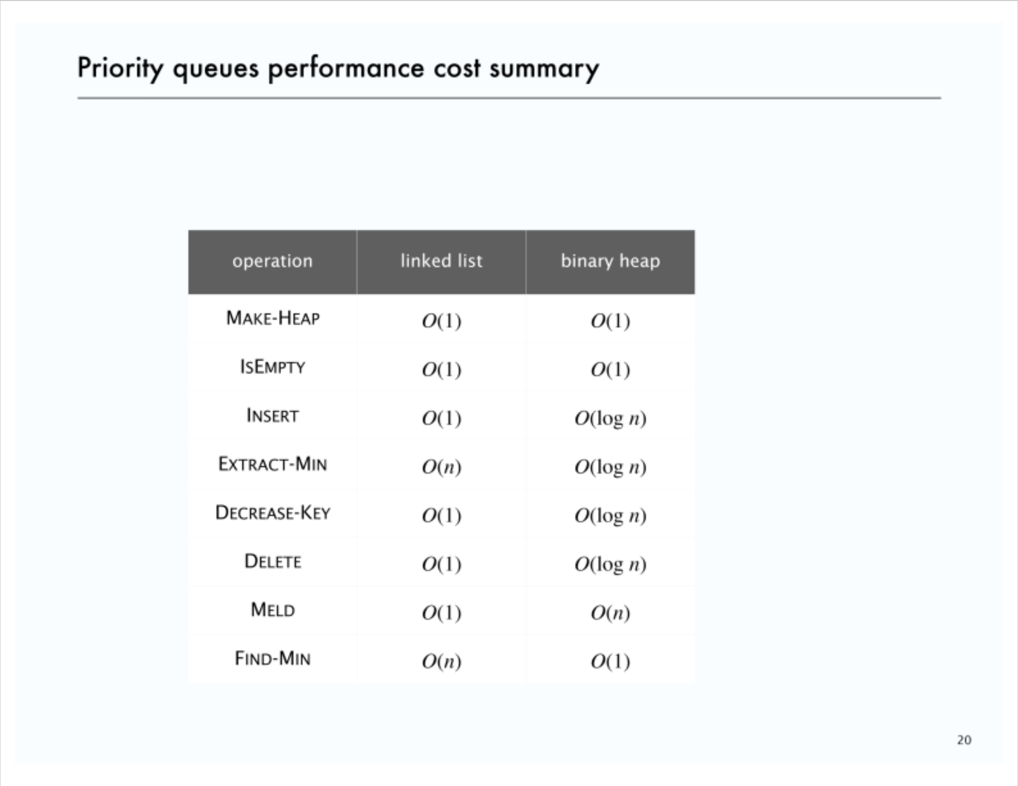




Binary heap: delete

Delete. Given a handle to a node, exchange element in node with last node; either swim down or sink up the node until heap order is restored.

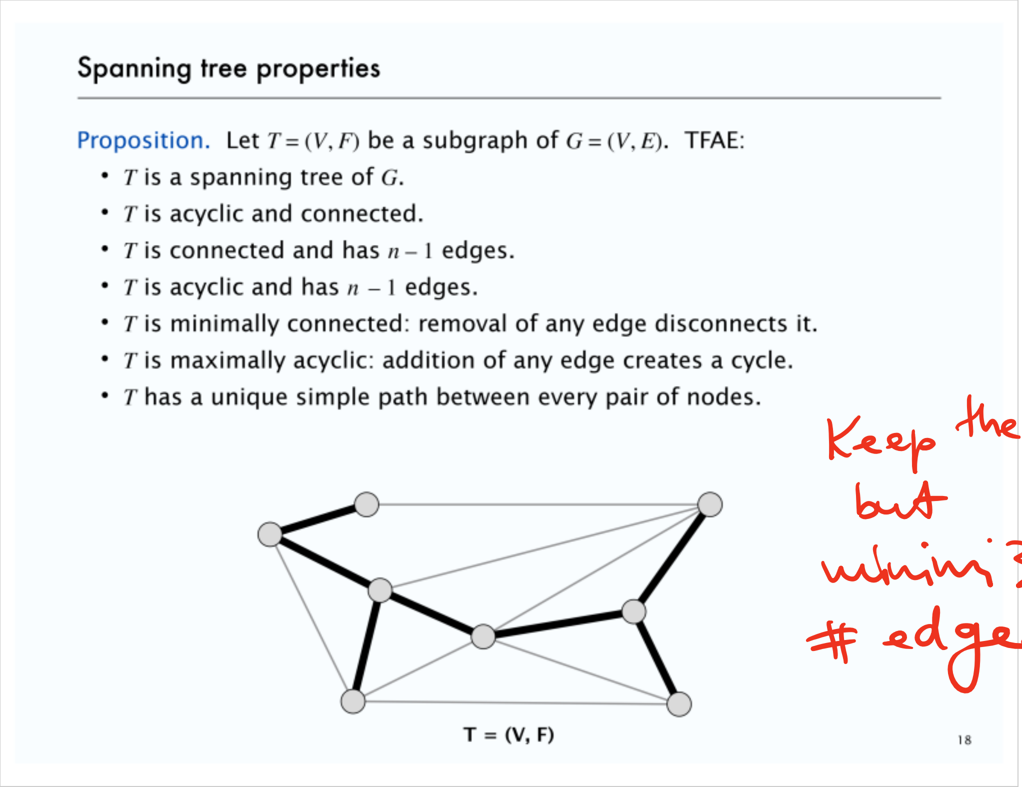
delete node x or y



Priority queues performance cost summary

operation	linked list	binary heap
MAKE-HEAP	$O(1)$	$O(1)$
ISEMPTY	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$
EXTRACT-MIN	$O(n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$
DELETE	$O(1)$	$O(\log n)$
MELD	$O(1)$	$O(n)$
FIND-MIN	$O(n)$	$O(1)$

Min Spanning Tree (MST)



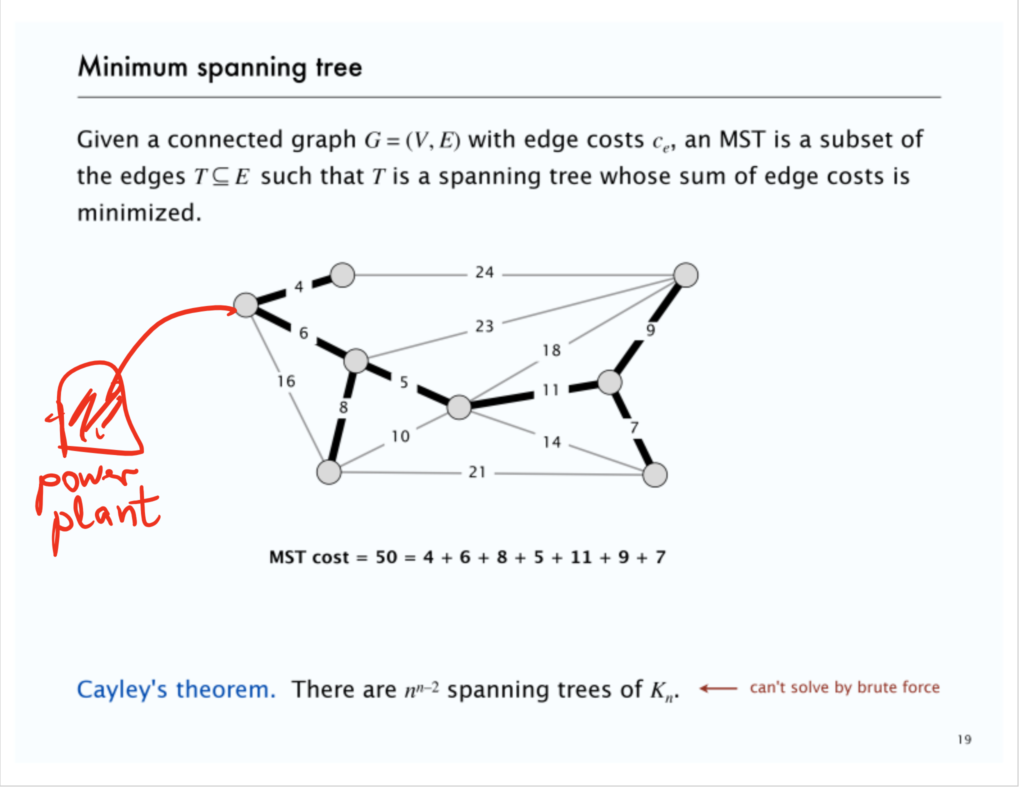
Spanning tree properties

Proposition. Let $T = (V, F)$ be a subgraph of $G = (V, E)$. TFAE:

- T is a spanning tree of G.
- T is acyclic and connected.
- T is connected and has $n - 1$ edges.
- T is acyclic and has $n - 1$ edges.
- T is minimally connected: removal of any edge disconnects it.
- T is maximally acyclic: addition of any edge creates a cycle.
- T has a unique simple path between every pair of nodes.

T = (V, F)

Keep the connectivity
but
minimize
edges



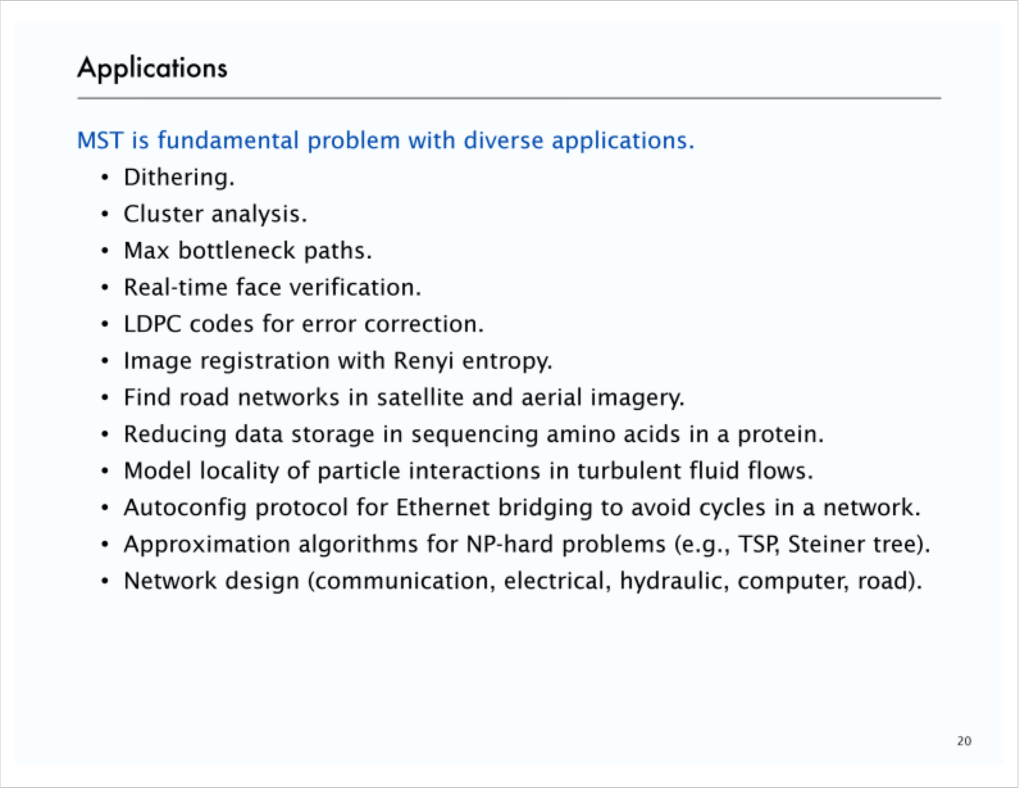
Minimum spanning tree

Given a connected graph $G = (V, E)$ with edge costs c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge costs is minimized.

power plant

MST cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

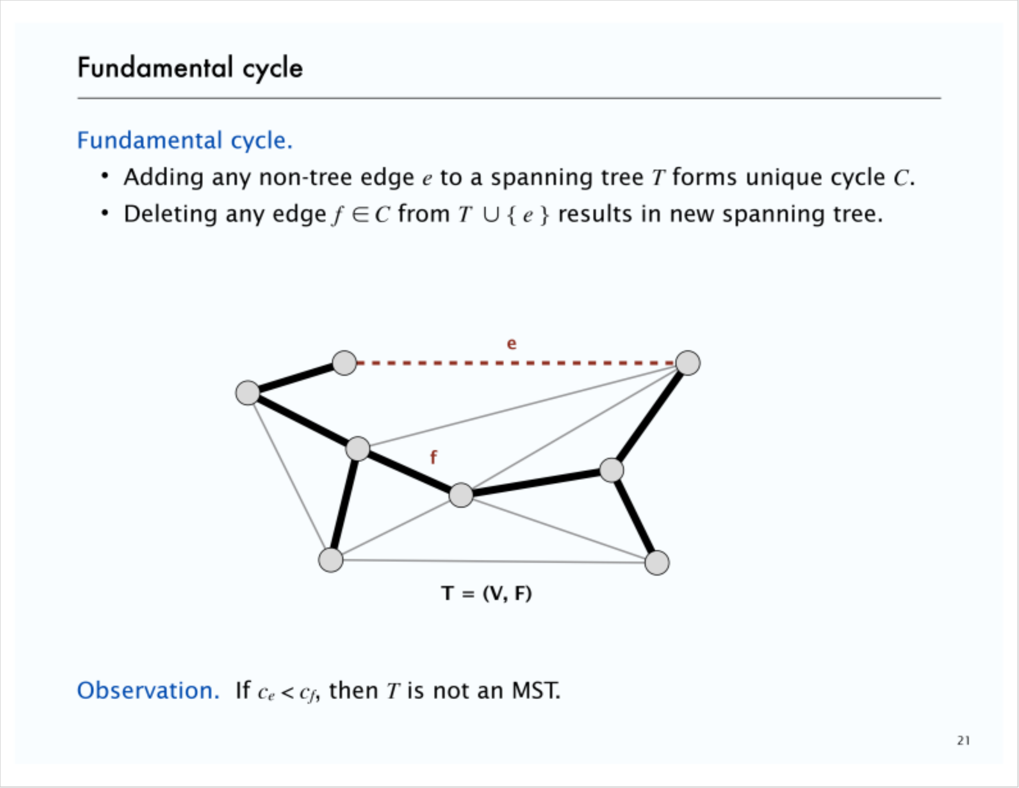
Cayley's theorem. There are n^{n-2} spanning trees of K_n. ← can't solve by brute force



Applications

MST is fundamental problem with diverse applications.

- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, computer, road).



Fundamental cycle

Fundamental cycle.

- Adding any non-tree edge e to a spanning tree T forms unique cycle C.
- Deleting any edge $f \in C$ from $T \cup \{ e \}$ results in new spanning tree.

T = (V, F)

Observation. If $c_e < c_f$, then T is not an MST.

Kruskal's Algorithm: G=(V,E), edge costs cost(e_i)

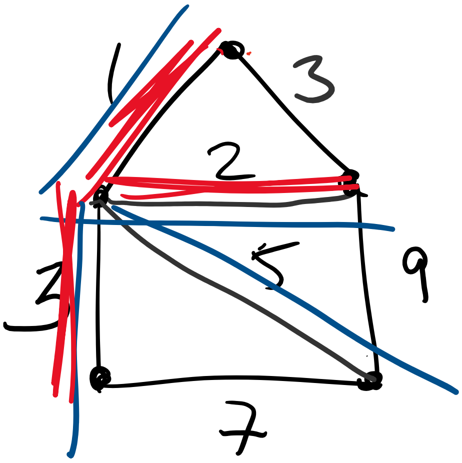
```
sort  $e_1, e_2, \dots, e_m$  so that  $cost(e_1) \le cost(e_2) \le \cdots \le cost(e_m)$

$T := \emptyset$

for $i = 1$ to $m$
    if $T \cup \{e_i\}$ has no cycle  then $T := T \cup \{e_i\}$
    endif
endfor

return $T$
```

Example:



cost = 11

T_{opt}

Correctness of Kruskal's algo:

$T_0 = \emptyset$

T_i = T after iteration i

Need to prove: T_m is MST.

Proof strategy:

Argue that each T_i can be

Argue that each T_i can be extended to some MST, using some of the edges $e_{i+1}, e_{i+2}, \dots, e_m$.

Defn: T_i is promising if there exists some MST T_{opt} s.t.

$$T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, \dots, e_m\}.$$

Claim: $\forall\, i$, T_i is promising.

$\Rightarrow$ $\underline{T_m}$ is promising

$\Rightarrow$ $\exists$ MST $\underline{T_{opt}}$: $\underbrace{T_m \subseteq T_{opt} \subseteq T_m \cup \emptyset}_{\Updownarrow \; T_m = T_{opt}}$

$\Rightarrow$ T_m is MST.

Pf of Claim: (by induction on i)

Base case: $i = 0$. $T_0 = \emptyset$ is promising: $\underline{T_0 \subseteq \underset{\uparrow \, \text{some MST}}{T_{opt}} \subseteq T_0 \cup \{e_1, \dots, e_m\}}$

Induction step: Assume T_i is promising ($i \geq 0$). $T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, \dots, e_m\} \Leftarrow$

Prove T_{i+1} is promising.

Case 1: $e_{i+1} \notin T_{i+1}$. $e_{i+1} \notin T_{opt}$

$$\underbrace{T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{\underline{e_{i+2}}, \dots, e_m\}}$$

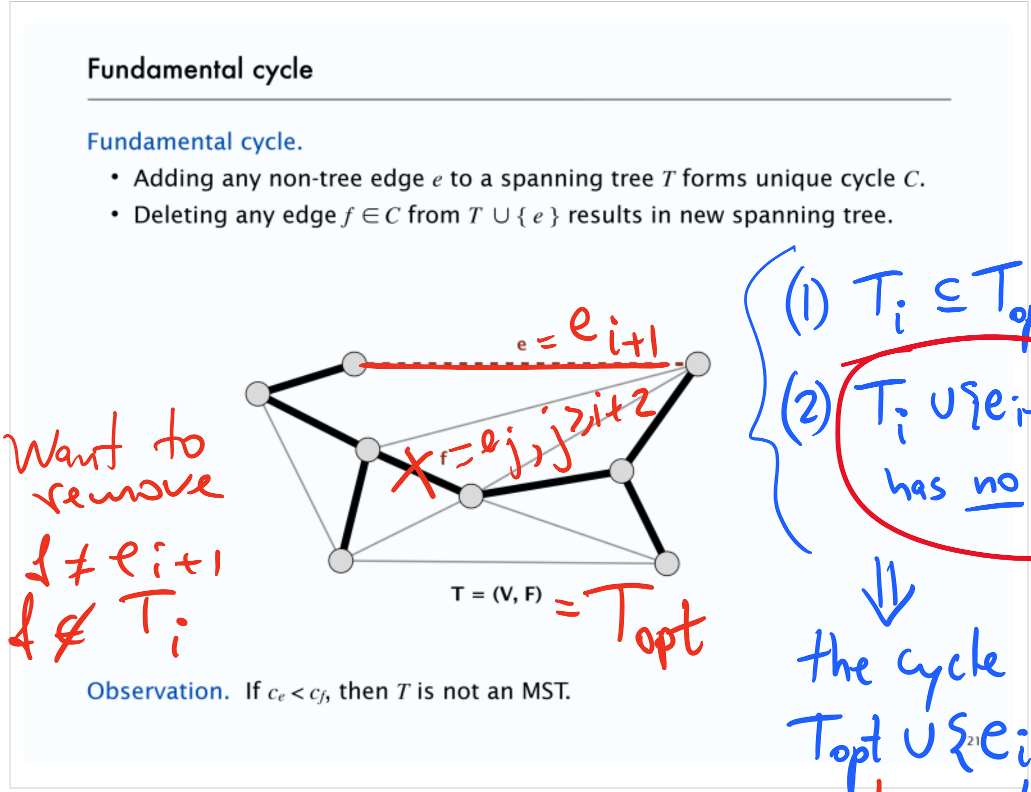
Case 2: $e_{i+1} \in T_{i+1}$

Case 2.1: $e_{i+1} \in T_{opt}$. ✓

Case 2.2: $e_{i+1} \notin T_{opt}$.

$T_i \subseteq T_{opt}$, $T_i \cup \{e_{i+1}\}$ has cycle

$\Rightarrow T_{opt} \cup \{e_{i+1}\}$ has a cycle



Fundamental cycle

Fundamental cycle.

- Adding any non-tree edge e to a spanning tree T forms unique cycle C.
- Deleting any edge $f \in C$ from $T \cup \{ e \}$ results in new spanning tree.

Want to remove

$f \neq e_{i+1}$

$f \notin T_i$

Observation. If $c_e < c_f$, then T is not an MST.

21

(1) $T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+2}, \dots, e_m\}$

(2) $T_i \cup \{e_{i+1}\}$ has no cycle

$\Downarrow$

the cycle in $T_{opt} \cup \{e_{i+1}\}$ must contain

$f = e_j$

$j > i+1$

cost(e_j)

Some e_j, $j > i+1 \Rightarrow$ [illegible] $\geq \text{cost}(e_{i+1})$

Define $T'_{opt} = T_{opt} \cup \{e_{i+1}\} - \{e_j\}$.

(1) T'_{opt} is a spanning tree.

(2) $\text{cost}(T'_{opt}) = \text{cost}(T_{opt}) + \underbrace{c(e_{i+1}) - c(e_j)}_{\leq 0}$

by the edge ordering

$\leq \text{cost}(T_{opt})$.

Finally, observe

$$T_{i+1} \subseteq T'_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \dots, e_m\}.$$

So T_{i+1} is promising. $\square$

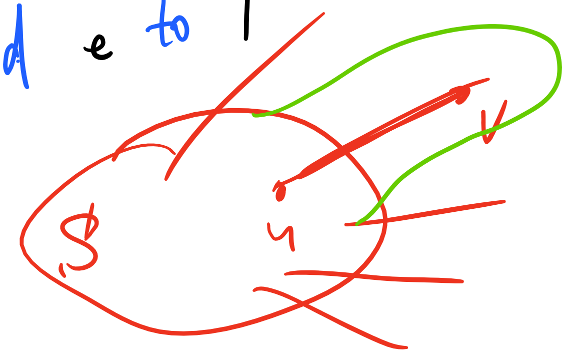
Prim's algorithm: $G = (V, E)$

$s \in V$; $S = \{s\}$; $T = \emptyset$

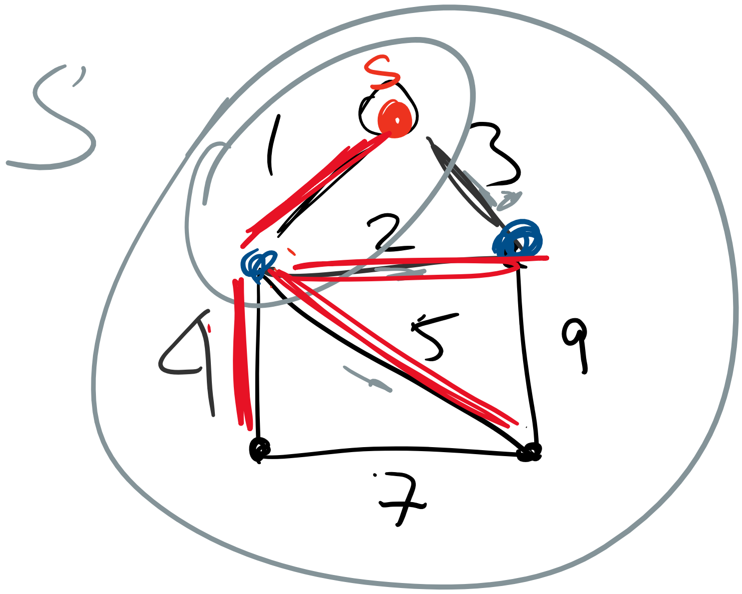
repeat for $|V| - 1$ steps

    find $e = (u, v) \in E$ with $u \in S$, $v \in \bar{S}$

```
    find $e=(u,v) \in E$ with $u \in S$,
        of min cost
    & add v to S ; add e to T
endrepeat
return T
```



Ex:



Correctness of Prim's Algo

Using promising

S_i, T_i = S, T after iteration i

iteration i

Def: (S_i, T_i) is promising if can be extended to some MST, using edges with at most one endpoint in S_i $\boxed{S_i}$!

Claim: $\forall i$, (S_i, T_i) is promising

Cor: $(S_{n-1}, \underline{T_{n-1}})$ is MST

Proof of Claim: By induction on i.

Base case: $i=0$ $\quad \underset{\{s\}}{S_0}, \underset{\emptyset}{T_0}$: (S_0, T_0) is promising ✓

Ind. Step: Assume i. (S_i, T_i) is promising

$\Downarrow$

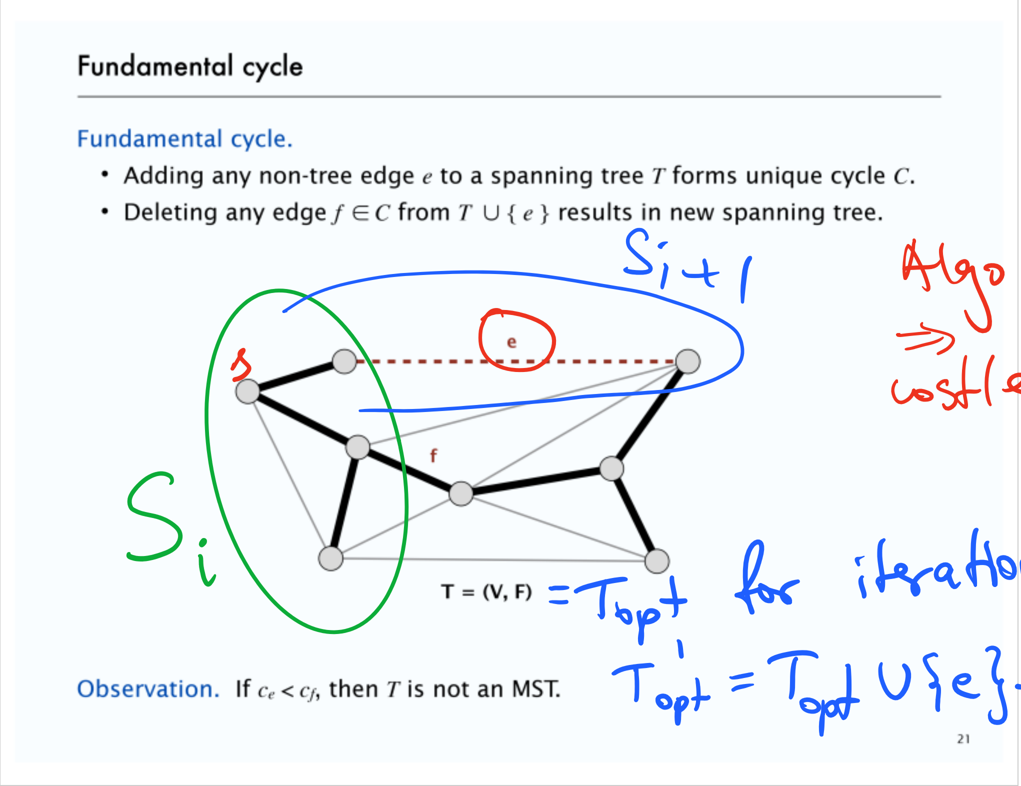
$\exists$ MST T_{opt} extending T_i

Prove $i+1$.

Edge $\underline{e}$ is added to T_i to form T_{i+1}.

Case 1: $e \in T_{opt}$. ✓

Case 2: $e \notin T_{opt}$


Fundamental cycle

Fundamental cycle.

- Adding any non-tree edge e to a spanning tree T forms unique cycle C.
- Deleting any edge $f \in C$ from $T \cup \{ e \}$ results in new spanning tree.

$= T_{opt}$ for iteration i

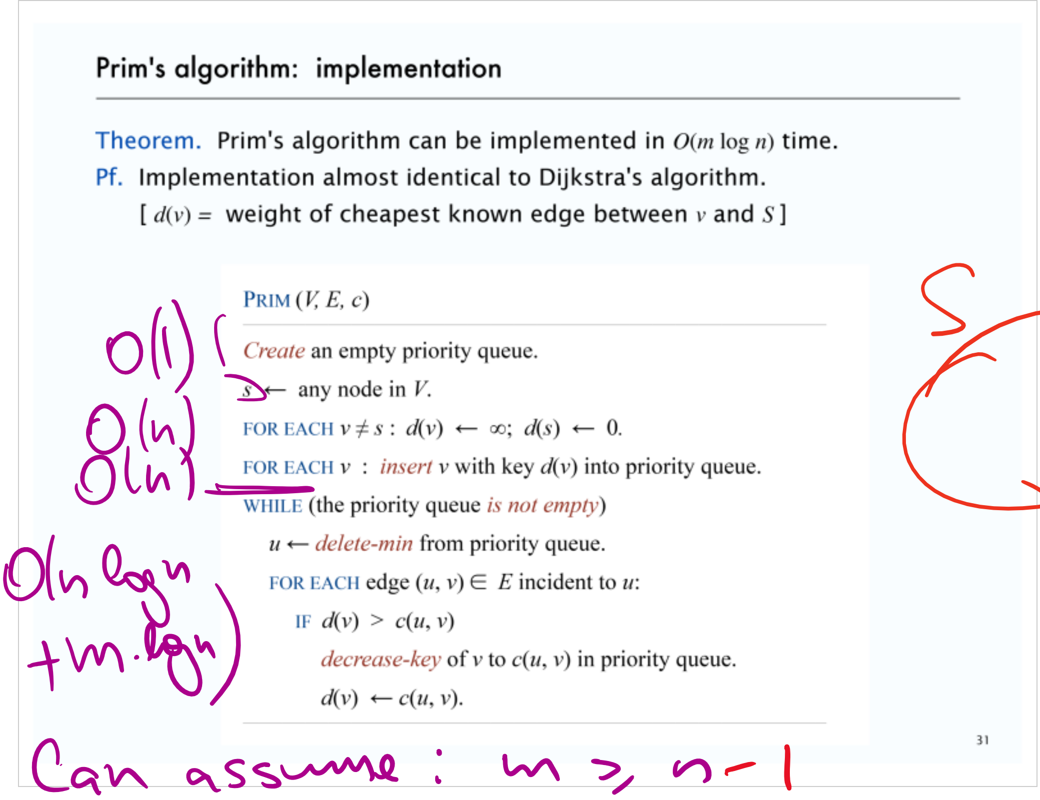
$T'_{opt} = T_{opt} \cup \{e\} - \{f\}$

Observation. If $c_e < c_f$, then T is not an MST.

Algo picked e

$\Rightarrow$

$\text{cost}(e) \underset{=}{\leq} \text{cost}(f)$


Prim's algorithm: implementation

Theorem. Prim's algorithm can be implemented in $O(m \log n)$ time.

Pf. Implementation almost identical to Dijkstra's algorithm.

[$d(v)$ = weight of cheapest known edge between v and S]

```
PRIM (V, E, c)
Create an empty priority queue.                              O(1)
s ← any node in V.
FOR EACH v ≠ s : d(v) ← ∞; d(s) ← 0.                          O(n)
FOR EACH v : insert v with key d(v) into priority queue.     O(n)
WHILE (the priority queue is not empty)                      O(n log n
    u ← delete-min from priority queue.                      + m.log n)
    FOR EACH edge (u, v) ∈ E incident to u:
        IF d(v) > c(u, v)
            decrease-key of v to c(u, v) in priority queue.
            d(v) ← c(u, v).
```

Can assume : $m \geq n-1$

S

u

v

$d(v) := \min \{d(v), c(u,v)\}$

Kruskal's algorithm: implementation

Theorem. Kruskal's algorithm can be implemented in $O(m \log m)$ time.

- Sort edges by weight.
- Use union-find data structure to dynamically maintain connected components.

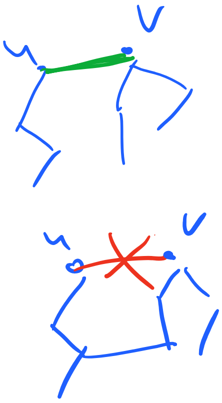
$O(m \log m)$!

```
KRUSKAL (V, E, c)

SORT m edges by weight so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$S \leftarrow \phi$
FOREACH $v \in V$: MAKESET($v$).
FOR $i = 1$ TO $m$
    $(u, v) \leftarrow e_i$
    IF FINDSET($u$) $\neq$ FINDSET($v$)    ← are u and v in same component?
        $S \leftarrow S \cup \{ e_i \}$
        UNION($u$, $v$).    ← make u and v in same component
RETURN $S$
```

adding u—v doesn't create a cycle

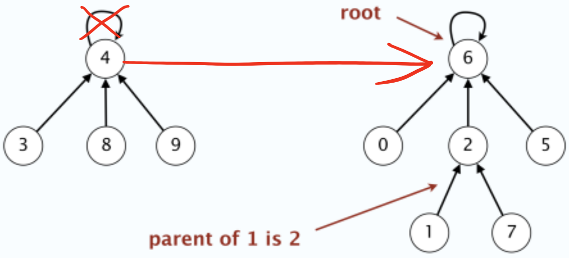
take time $O(\log m)$



Disjoint-sets data structure

Representation. Represent each set as a tree of elements.

- Each element has a parent pointer in the tree.
- The root serves as the canonical element.
- FIND(x). Find the root of the tree containing x.
- UNION(x, y). Make the root of one tree point to root of other tree.



root

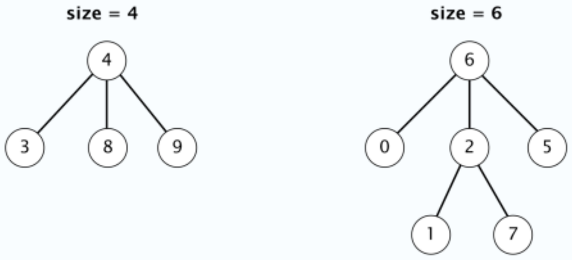
parent of 1 is 2

Note. For brevity, we suppress arrows and self loops in figures.

Link-by-size

Link-by-size. Maintain a subtree count for each node, initially 1.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

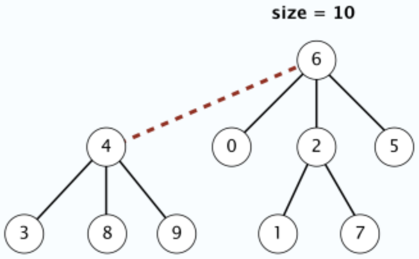
union(7, 3)



Link-by-size

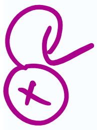
Link-by-size. Maintain a subtree count for each node, initially 1.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

union(7, 3)



Link-by-size

Link-by-size. Maintain a subtree count for each node, initially 1.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).



```
MAKE-SET (x)
    parent(x) ← x.
    size(x) ← 1.
```

```
FIND (x)
    WHILE (x ≠ parent(x))
        x ← parent(x).
    RETURN x.
```

```
UNION-BY-SIZE (x, y)
    r ← FIND (x).
    s ← FIND (y).
    IF (r = s) RETURN.
    ELSE IF (size(r) > size(s))
        parent(s) ← r.
        size(r) ← size(r) + size(s).
    ELSE
        parent(r) ← s.
        size(s) ← size(r) + size(s).
```

Link-by-size: analysis

Property. Using link-by-size, for every root node r: $size(r) \ge 2^{height(r)}$.

height(r) ≤ log size(r)

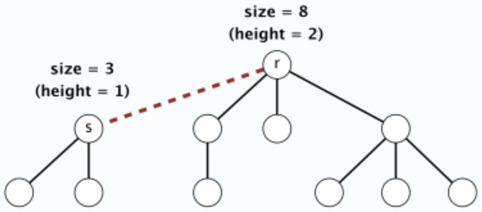
Pf. [by induction on number of links]

v f

- Base case: singleton tree has size 1 and height 0. ✓
- Inductive hypothesis: assume true after first i links.
- Tree rooted at r changes only when a smaller tree rooted at s is linked into r.
- Case 1. [$height(r) > height(s)$]

$$\begin{aligned} size'(r) &\ge size(r) \\ &\ge 2^{height(r)} \quad \text{← inductive hypothesis} \\ &= 2^{height'(r)}. \end{aligned}$$

after merge



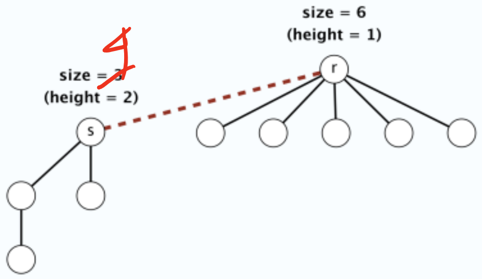
Link-by-size: analysis

Property. Using link-by-size, for every root node r, $\boxed{size(r) \geq 2^{height(r)}}$.

Pf. [by induction on number of links]

- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first i links.
- Tree rooted at r changes only when a smaller tree rooted at s is linked into r.
- Case 2. [$height(r) \leq height(s)$]

$$\begin{aligned} size'(r) &= size(r) + size(s) \\ &\geq 2\, size(s) && \leftarrow \text{link-by-size} : \quad size(s) \leq size(r) \\ &\geq 2 \cdot 2^{height(s)} && \leftarrow \text{inductive hypothesis} \\ &= 2^{height(s)+1} \\ &= 2^{height'(r)}. \quad ▪ \end{aligned}$$



Link-by-size: analysis

Theorem. Using link-by-size, any UNION or FIND operations takes $O(\log n)$ time in the worst case, where n is the number of elements.

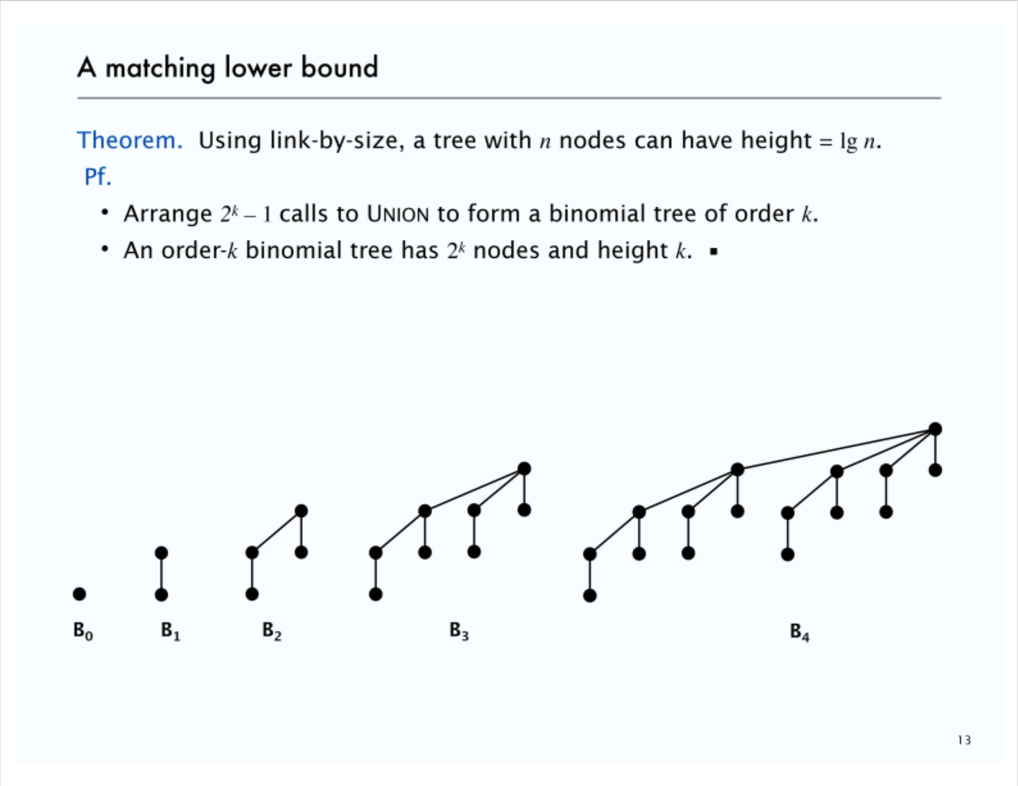
Pf.

- The running time of each operation is bounded by the tree height.
- By the previous property, the height is $\leq \lfloor \lg n \rfloor$. ▪

↑
$\lg n = \log_2 n$

$$size(r) \geq 2^{height(r)}$$

$$\boxed{\log size(r) \geq height(r)}$$



A matching lower bound

Theorem. Using link-by-size, a tree with n nodes can have height $= \lg n$.

Pf.

- Arrange $2^k - 1$ calls to UNION to form a binomial tree of order k.
- An order-k binomial tree has 2^k nodes and height k. ▪