CSE202 Greedy algorithms II

FIFO

LIFO

Queue

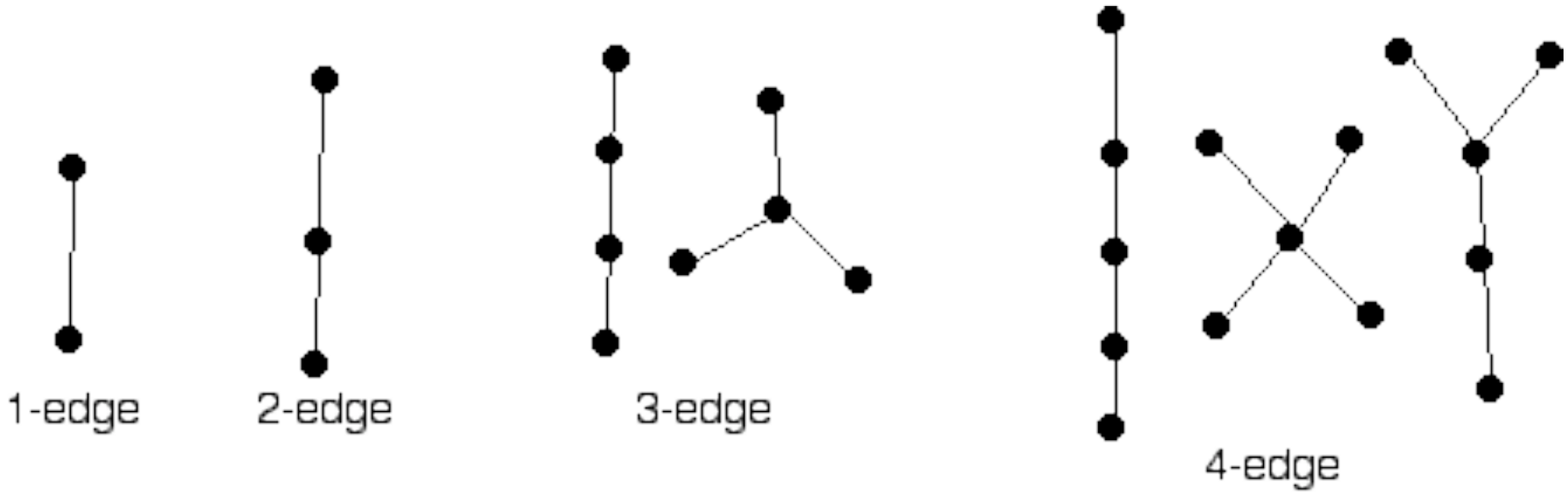
Stack

Announcements

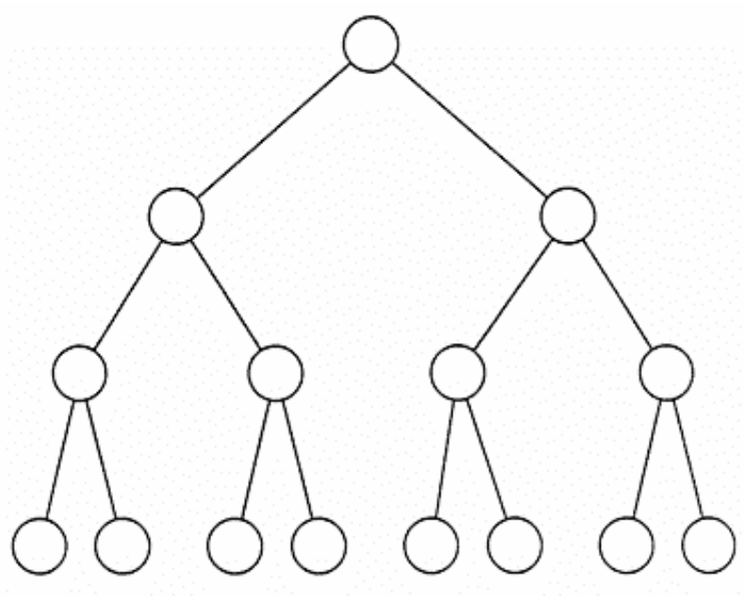
About homework, write your own homework, allowing oral discussion with one fixed partner.

tree

Trees with at most 4 edges



G is a tree on n vertices.

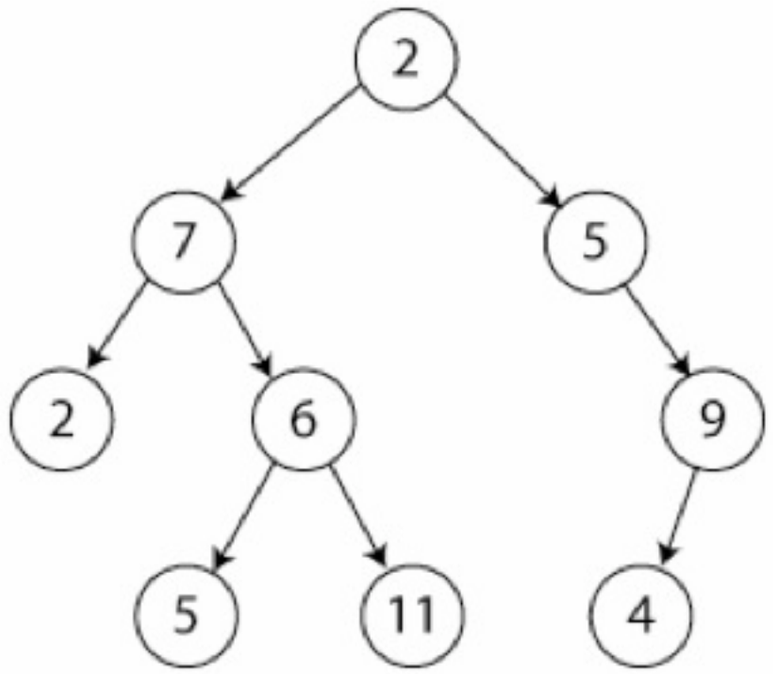


⟺ G is connected with no cycle.

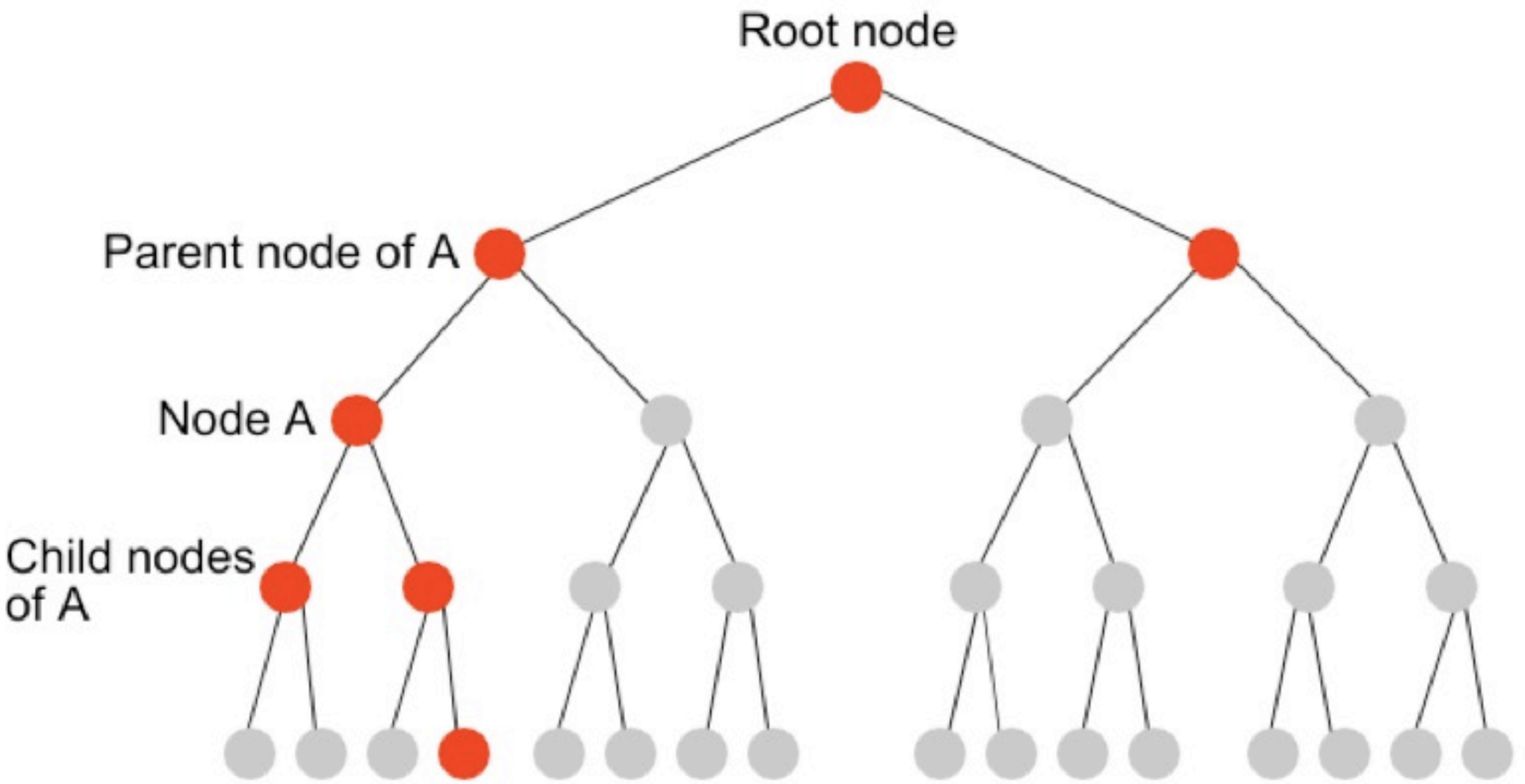
⟺ G is connected with n-1 edges.

⟺ G is formed by adding a leaf to a tree of n-1 vertices.

⟺ There is a unique path between any two vertices.



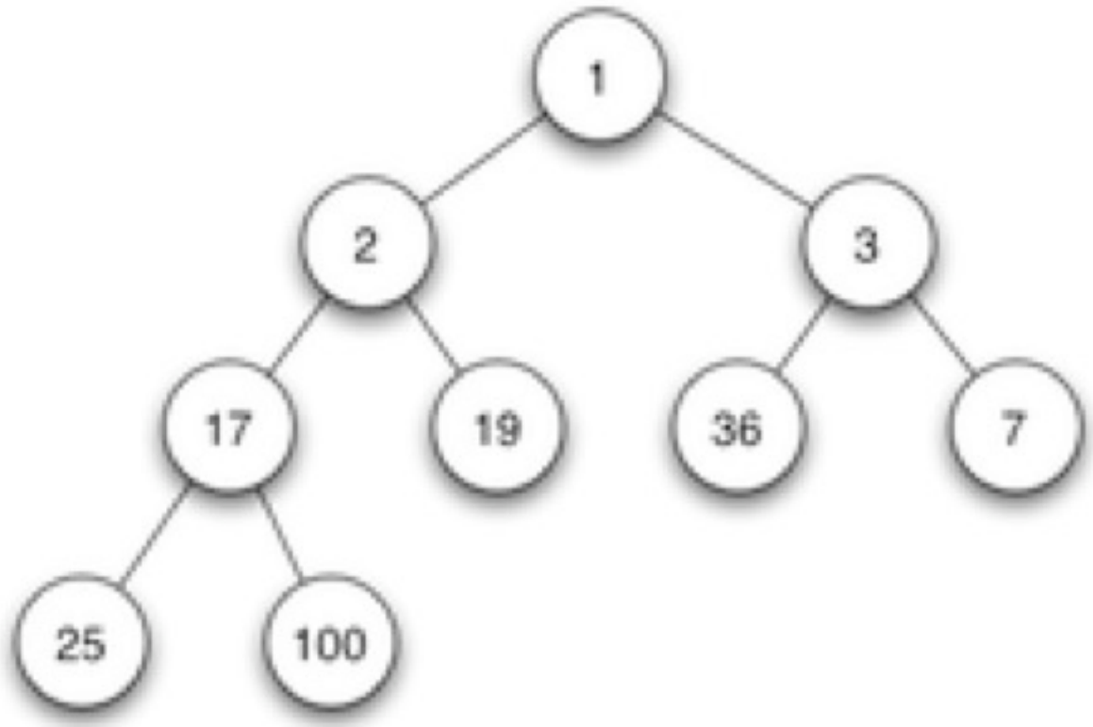
A binary tree



An example of a complete binary tree

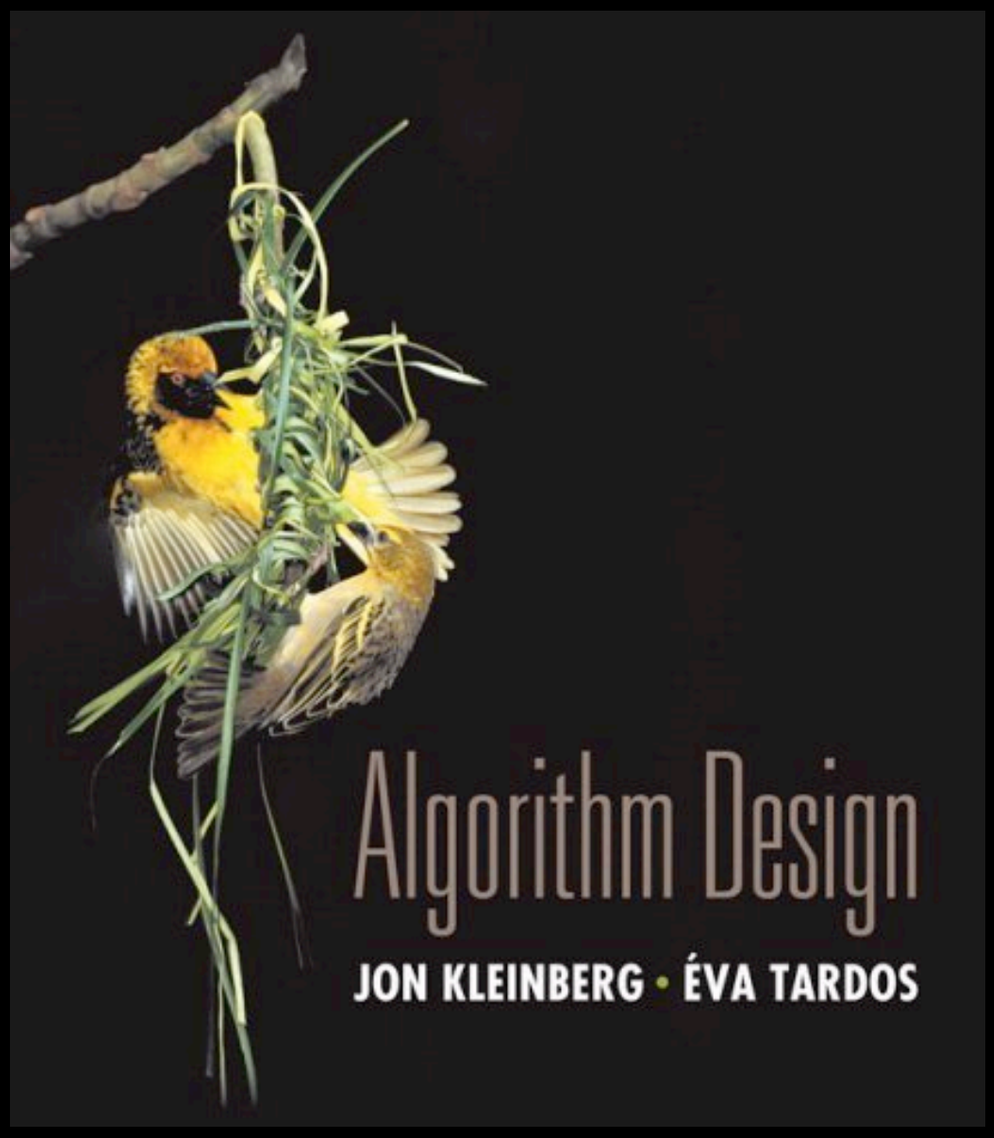
A k-level complete binary tree has ?? vertices.

Binary heap



A binary heap is a useful data structure if you have a collection of objects to which you are adding more objects and need to remove the object with the lowest value.

balanced binary tree



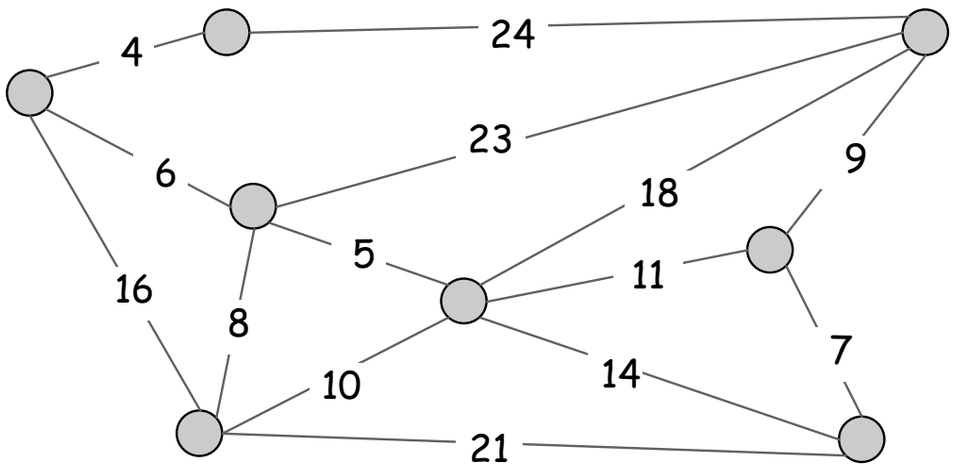
Chapter 4

Greedy Algorithms

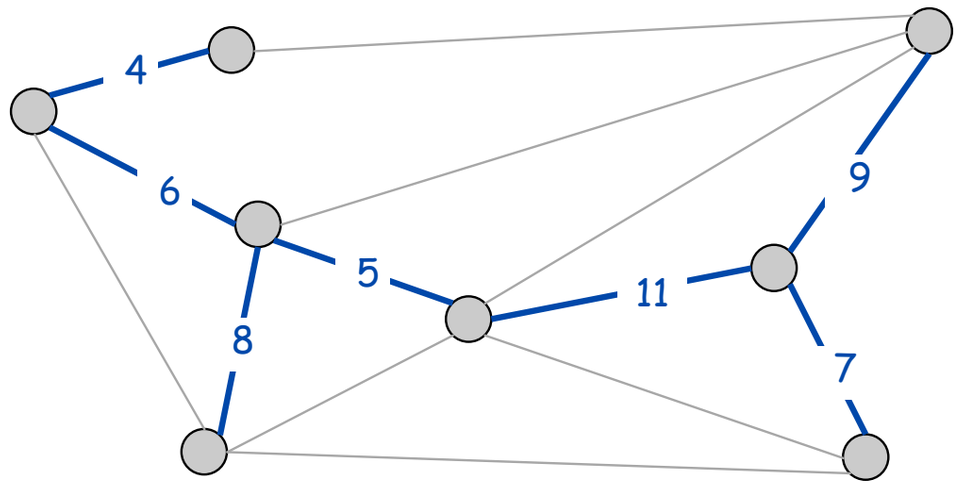
4.5 Minimum Spanning Tree

Minimum Spanning Tree

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



$G = (V, E)$



T, $\sum_{e \in T} c_e = 50$

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.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road

- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree

- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - 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

- Cluster analysis.

Greedy Algorithms

Kruskal's algorithm. Start with $T = \phi$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

Remark. All three algorithms produce an MST.

Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

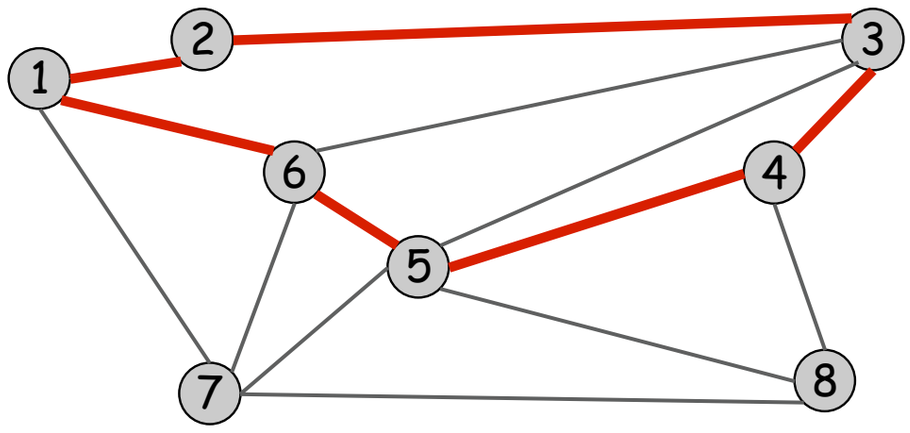
e.g., if all edge costs are integers, perturbing cost of edge e_i by i / n^2

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```
boolean less(i, j) {
   if      (cost(e_i) < cost(e_j)) return true
   else if (cost(e_i) > cost(e_j)) return false
   else if (i < j)                 return true
   else                            return false
}
```

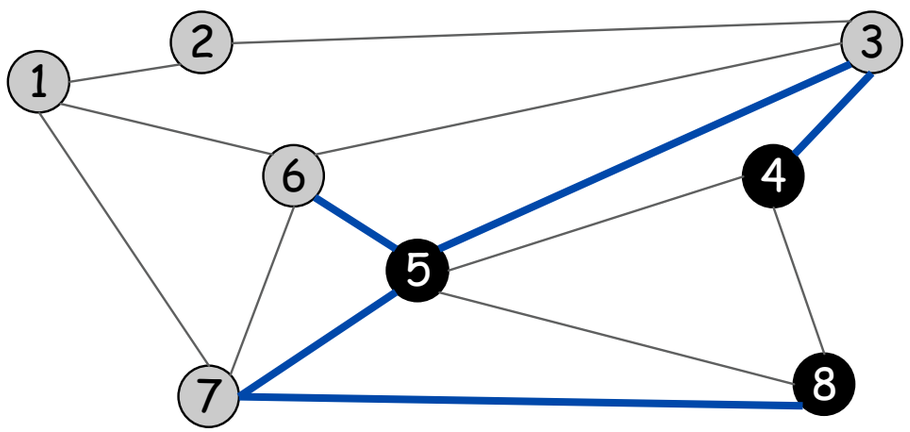
Cycles and Cuts

Cycle. Set of edges the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



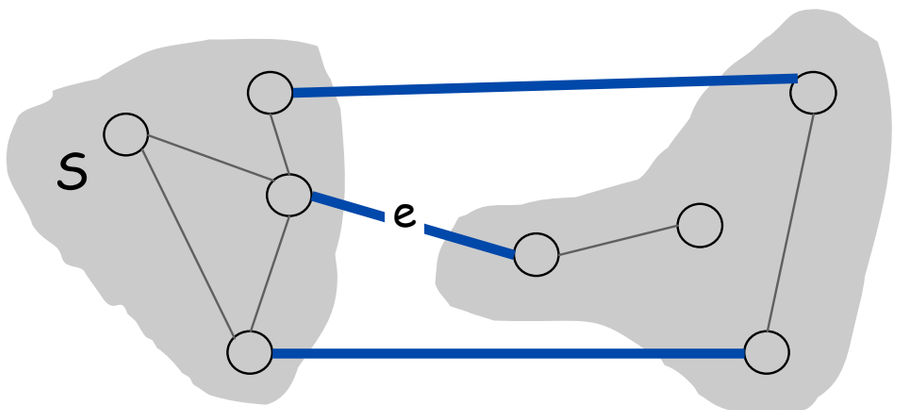
Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

Greedy Algorithms

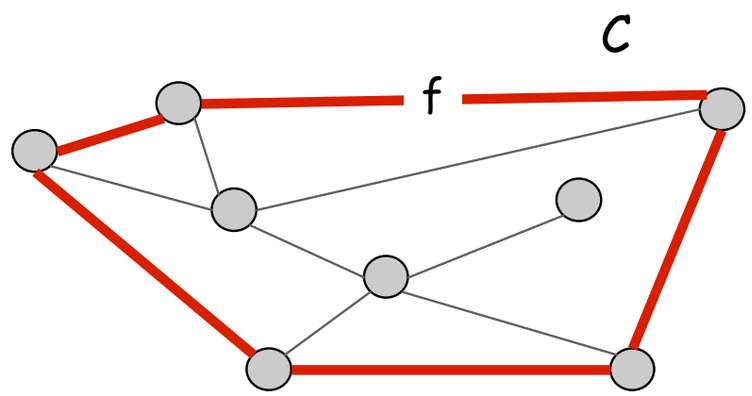
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



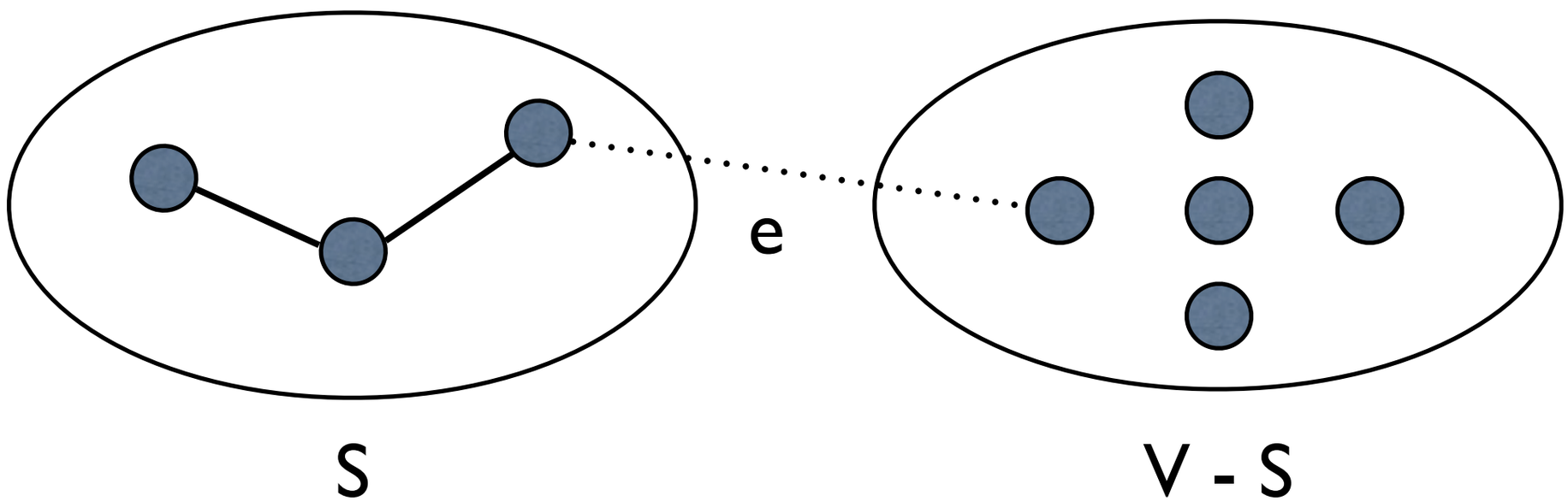
e is in the MST



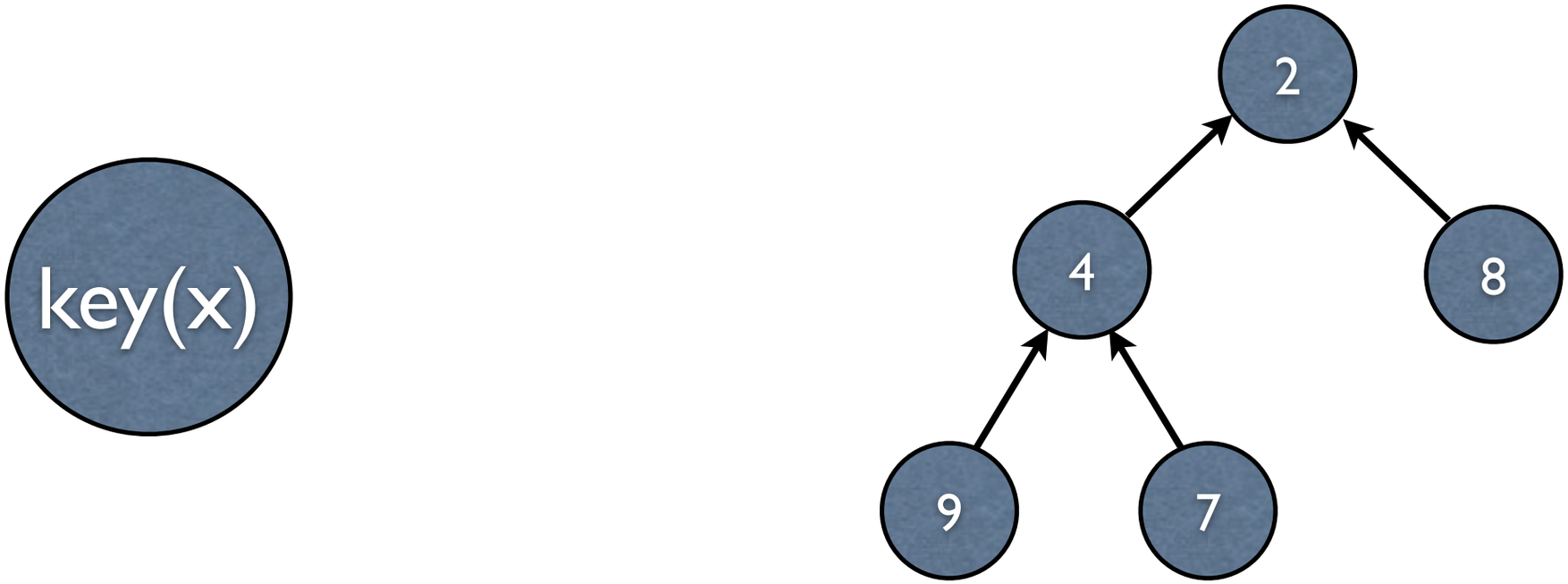
f is not in the MST

Prim's Algorithm

X = { }, S = {r}
Repeat until S has n nodes:
    Pick the **lightest** edge e in the cut (S, V - S)
    Add e to X
    Add v, the end-point of e in V - S to S



Data Structure: Heap

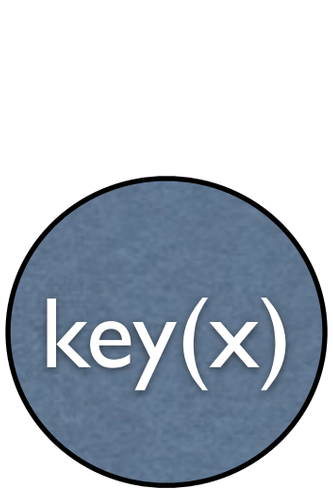


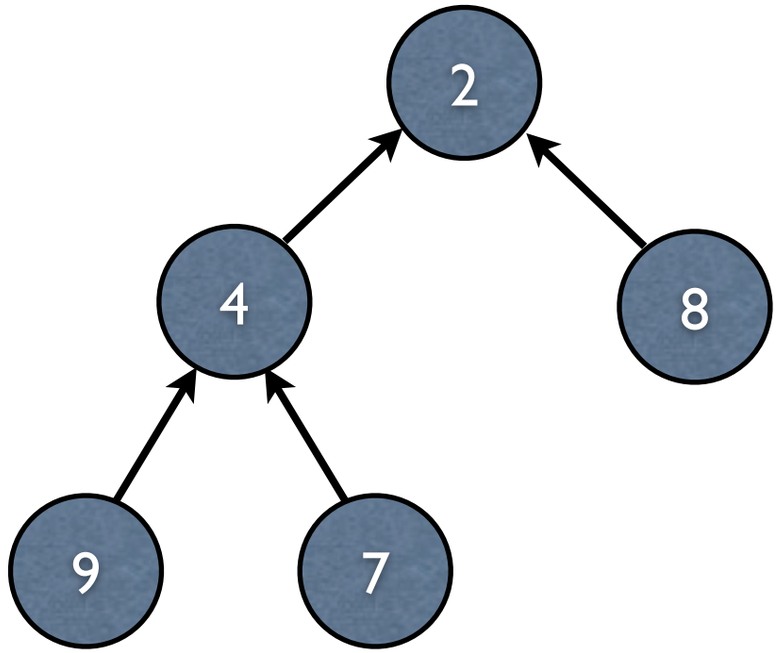
Heap Property: If x is the parent of y, then key(x) <= key(y)

A heap is stored as a **balanced binary tree**

Height = **O(log n)**, where n = # nodes

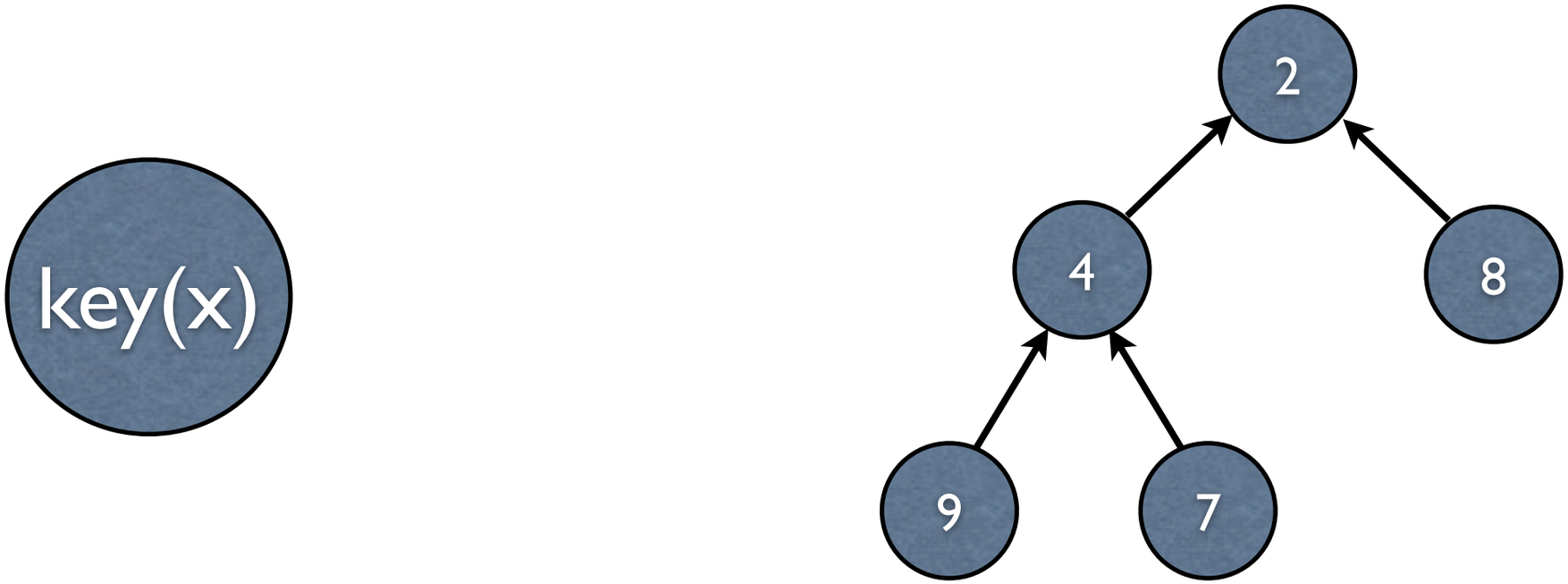
Heap: Reporting the min





Heap Property: If x is the parent of y, then key(x) <= key(y)

Heap: Reporting the min

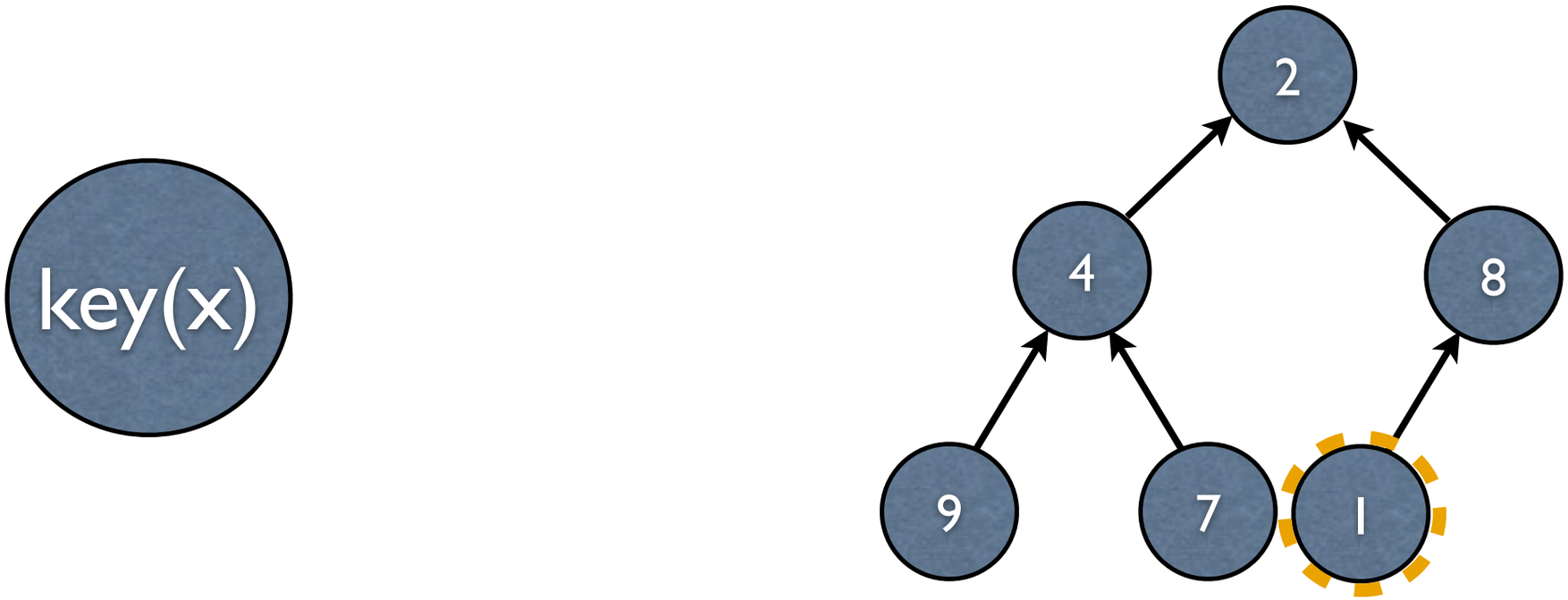


Heap Property: If x is the parent of y, then key(x) <= key(y)

Report the root node

Time = O(1)

Heap: Add an item

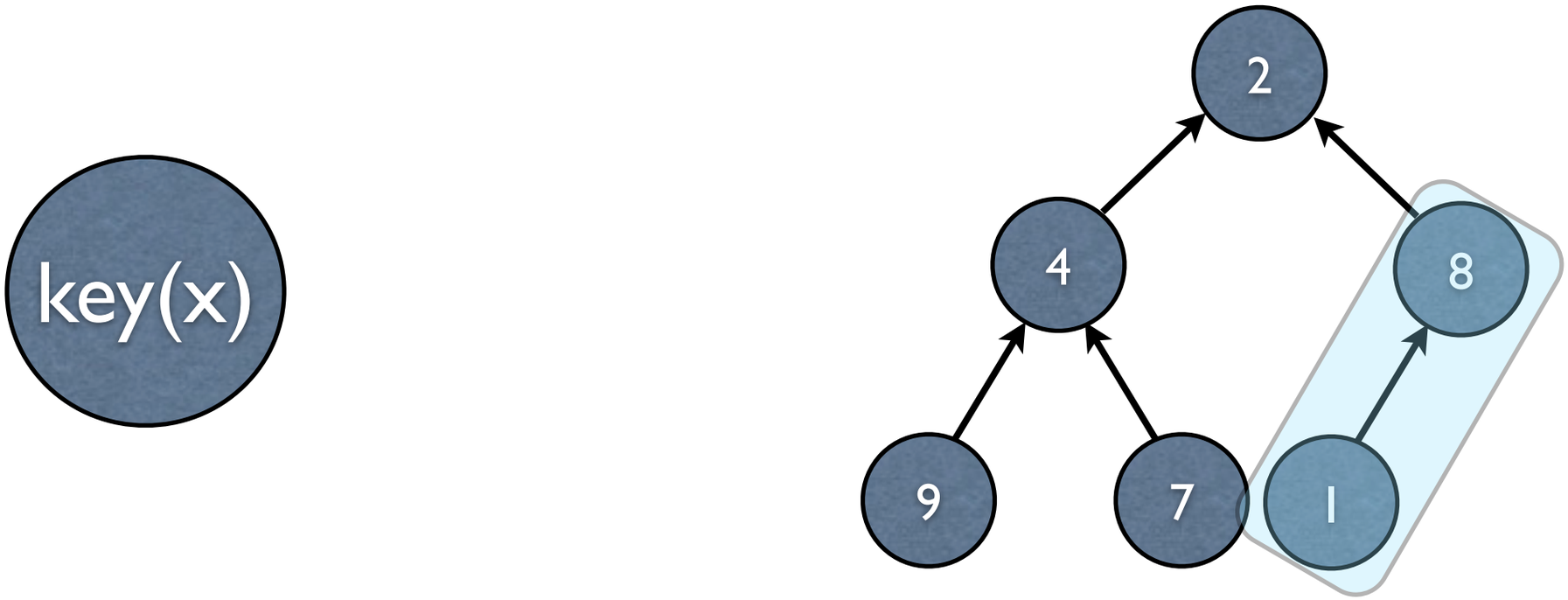


Heap Property: If x is the parent of y, then key(x) <= key(y)

Add item u to the end of the heap

If heap property is violated, swap u with its parent

Heap: Add an item

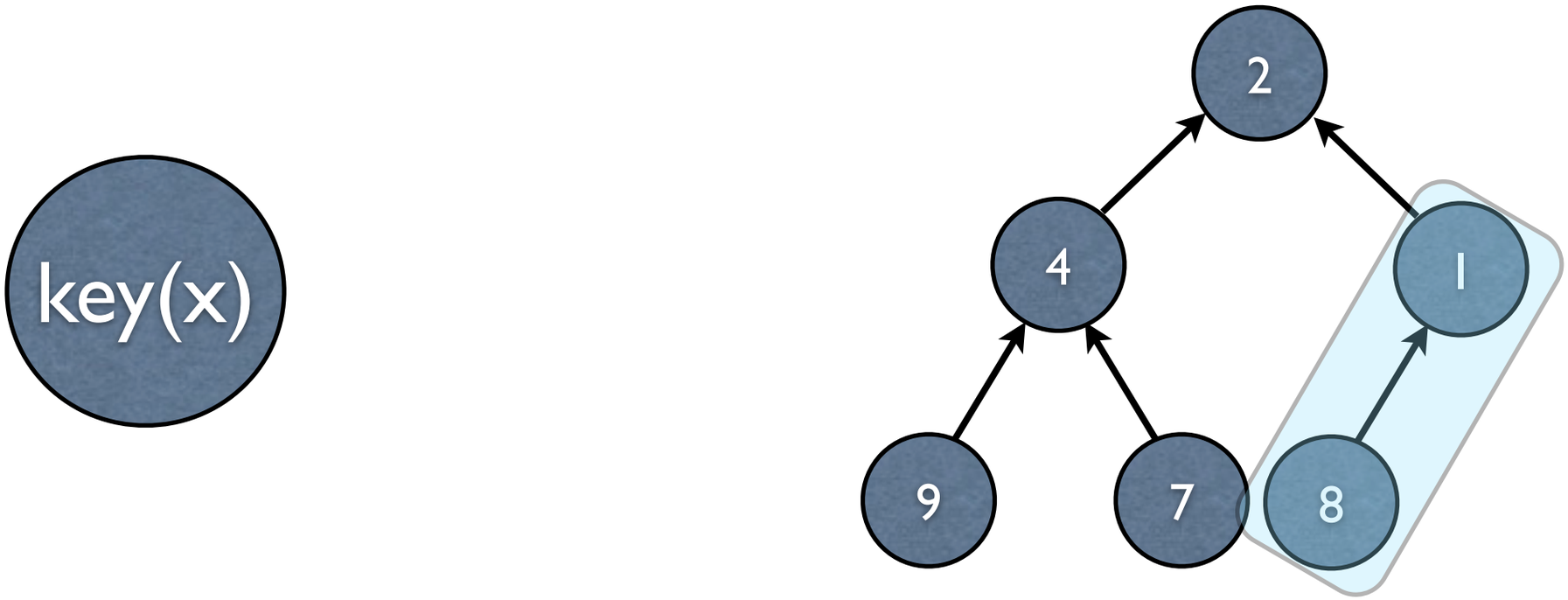


Heap Property: If x is the parent of y, then key(x) <= key(y)

Add item u to the end of the heap

If heap property is violated, swap u with its parent

Heap: Add an item

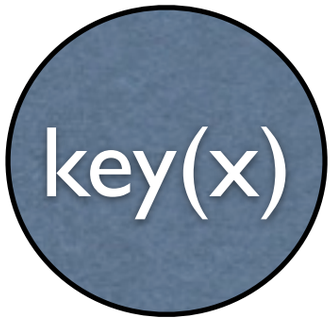


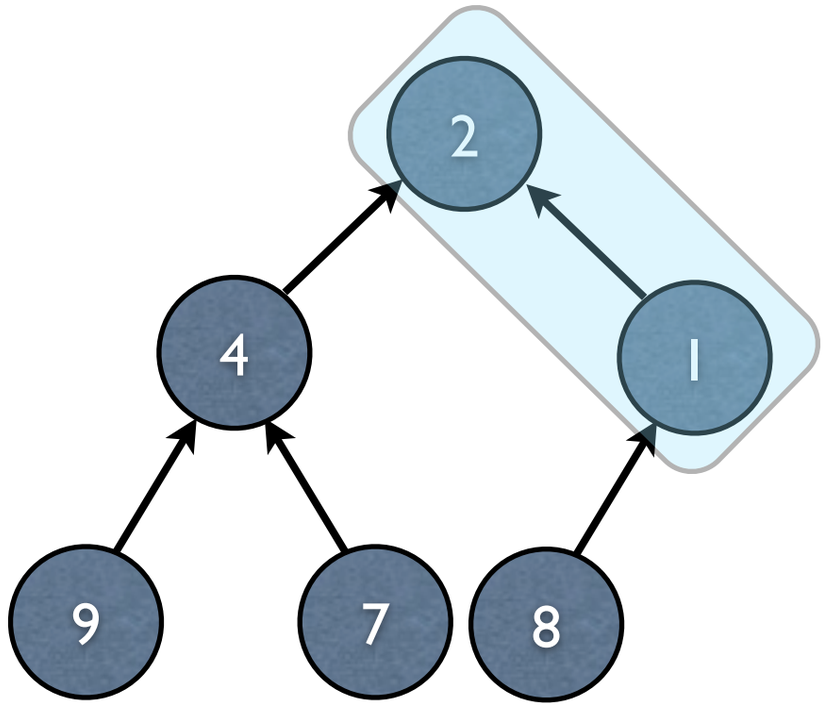
Heap Property: If x is the parent of y, then key(x) <= key(y)

Add item u to the end of the heap

If heap property is violated, swap u with its parent

Heap: Add an item



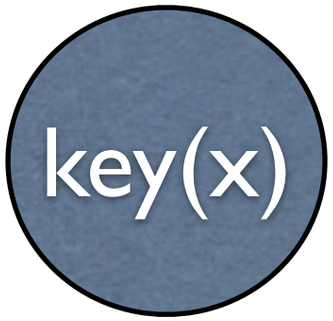


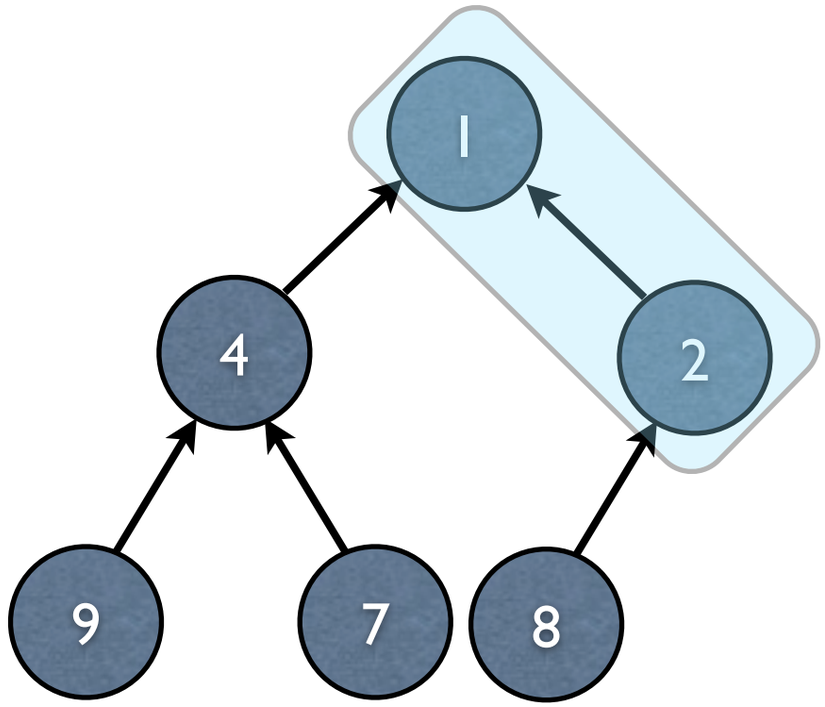
Heap Property: If x is the parent of y, then key(x) <= key(y)

Add item u to the end of the heap

If heap property is violated, swap u with its parent

Heap: Add an item



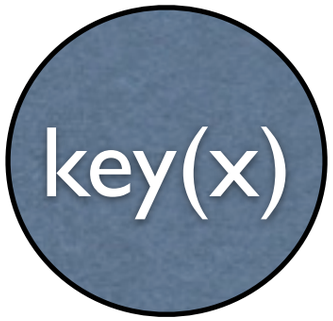


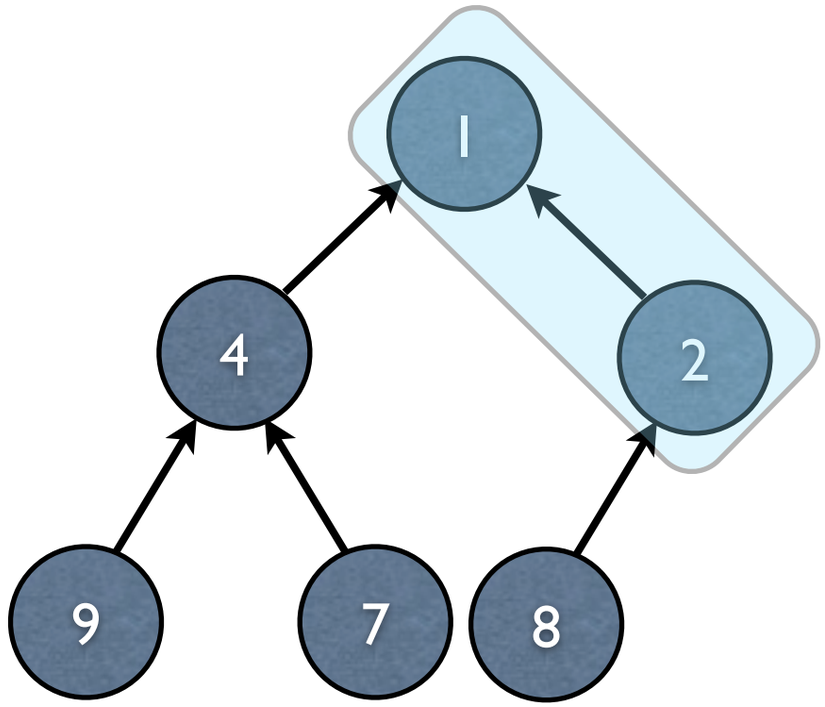
Heap Property: If x is the parent of y, then key(x) <= key(y)

Add item u to the end of the heap

If heap property is violated, swap u with its parent

Heap: Add an item





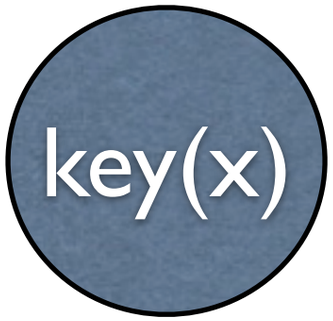
Heap Property: If x is the parent of y, then key(x) <= key(y)

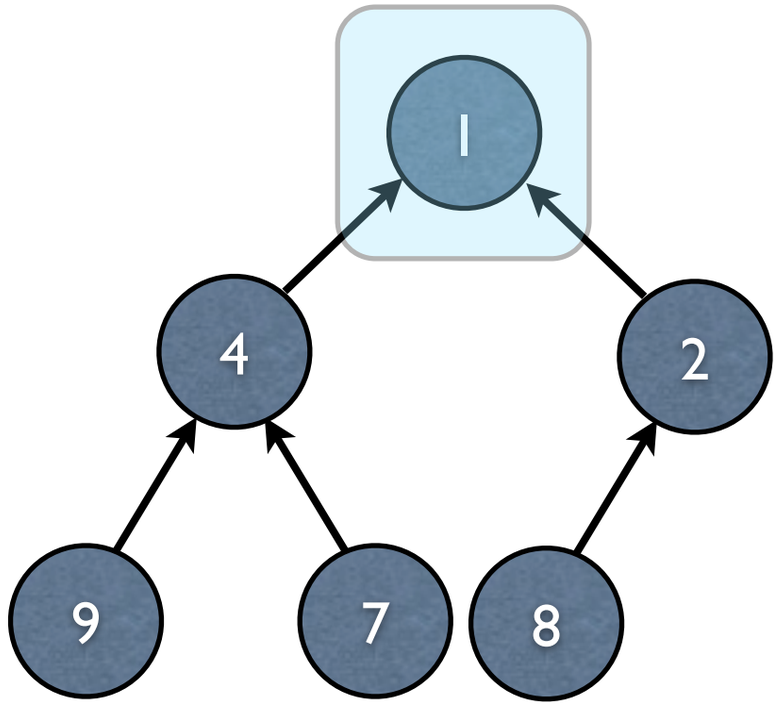
Add item u to the end of the heap

If heap property is violated, swap u with its parent

Time = O(log n)

Heap: Delete an item



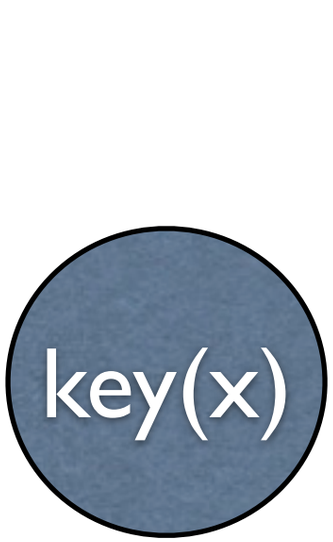


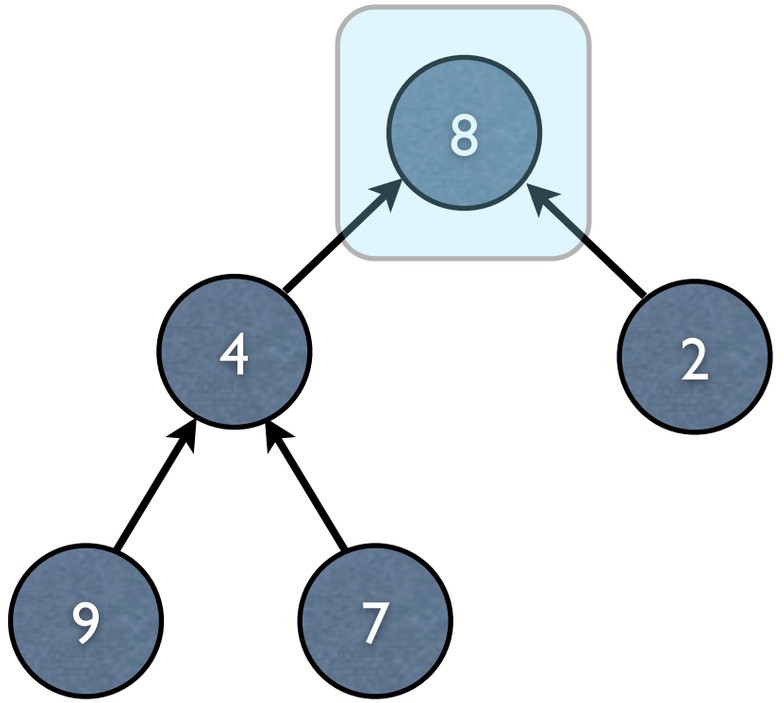
Heap Property: If x is the parent of y, then key(x) <= key(y)

Delete item u

Move v, the last item to u's position

Heap: Delete an item





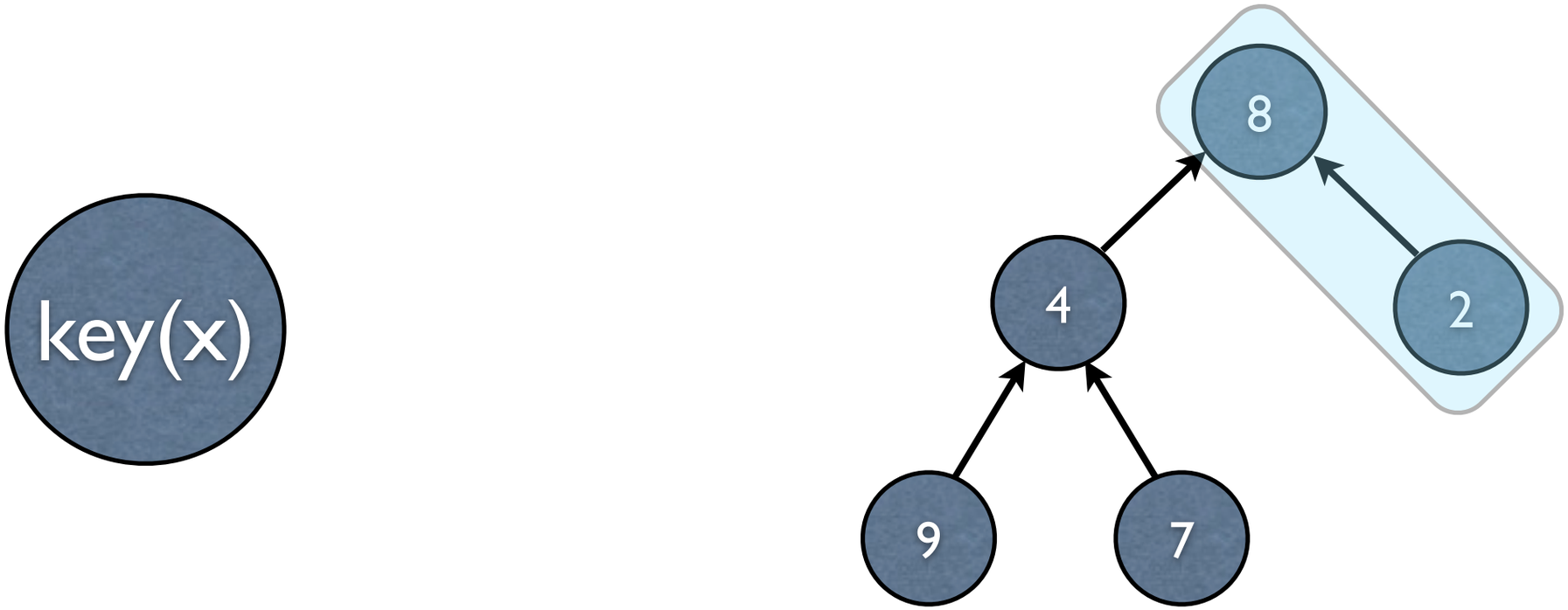
Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

Case 1. key[v] > key[child[v]]

Case 2. key[v] < key[parent[v]]

Heap: Delete an item



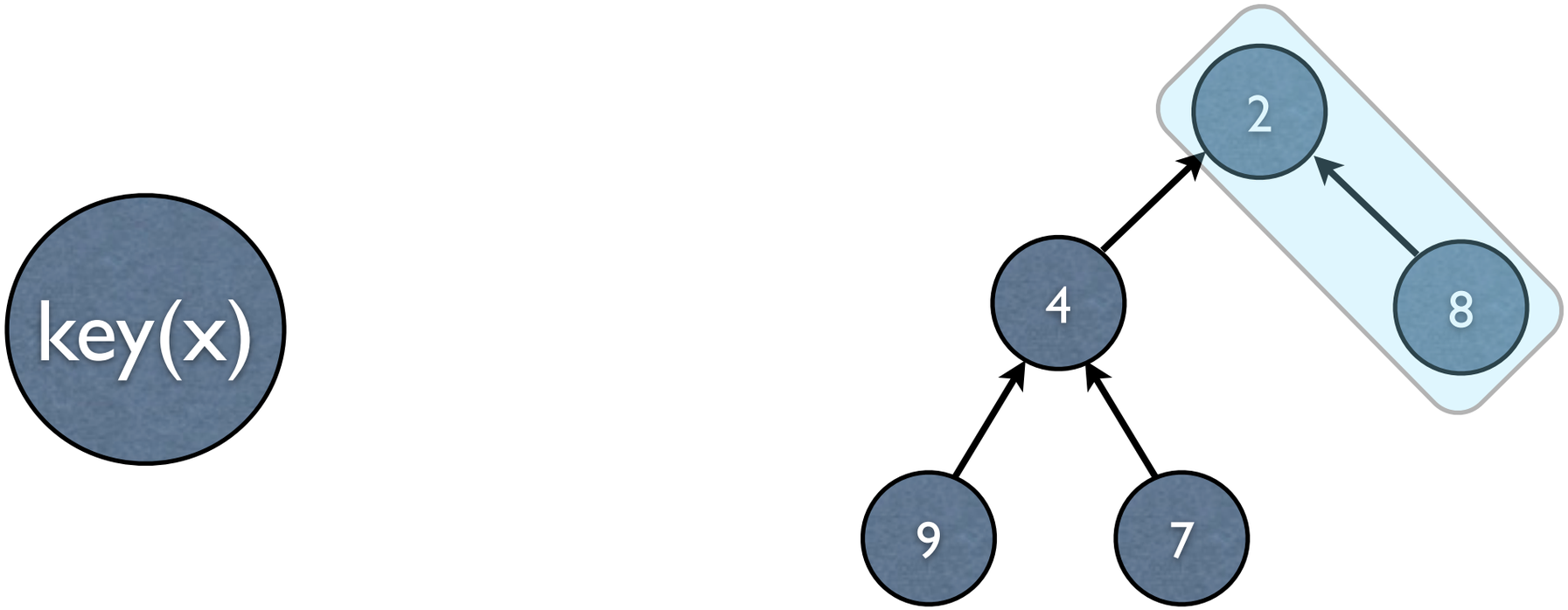
Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

Case 1. key[v] > key[child[v]]

Swap v with its **lowest key** child

Heap: Delete an item



Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

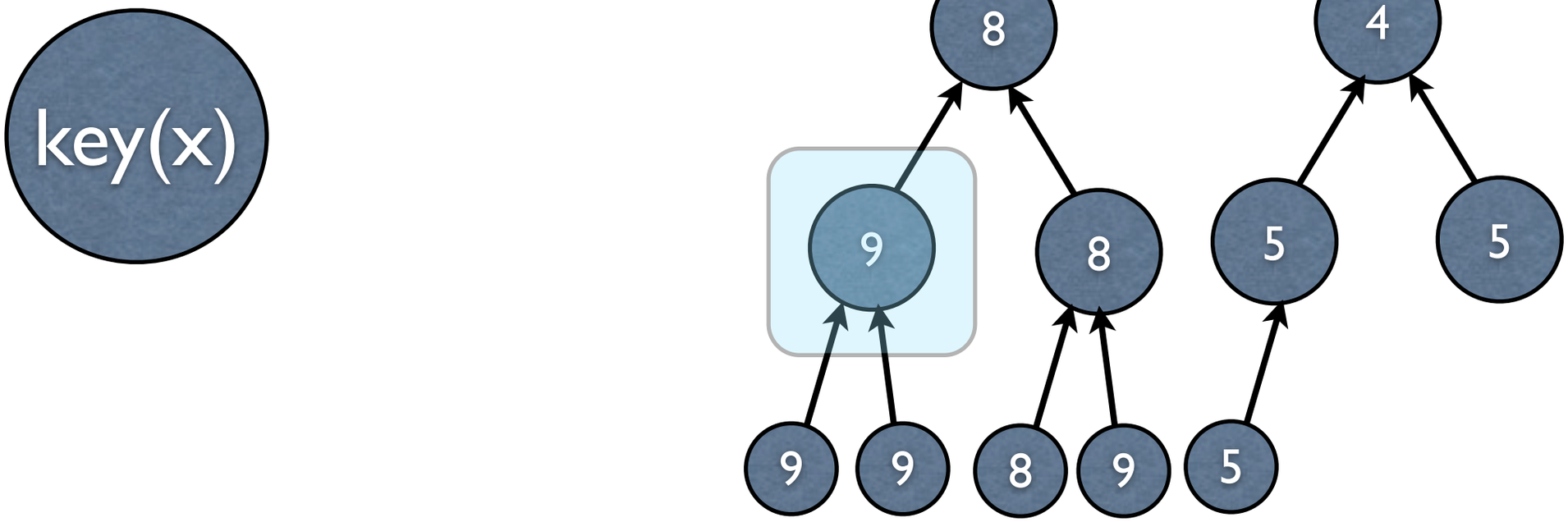
Time = O(log n)

Case 1. key[v] > key[child[v]]

Swap v with its **lowest key** child

Continue until heap property holds

Heap: Delete an item



Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

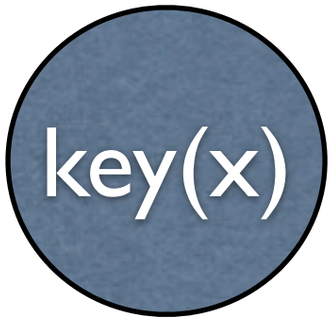
Time = O(log n)

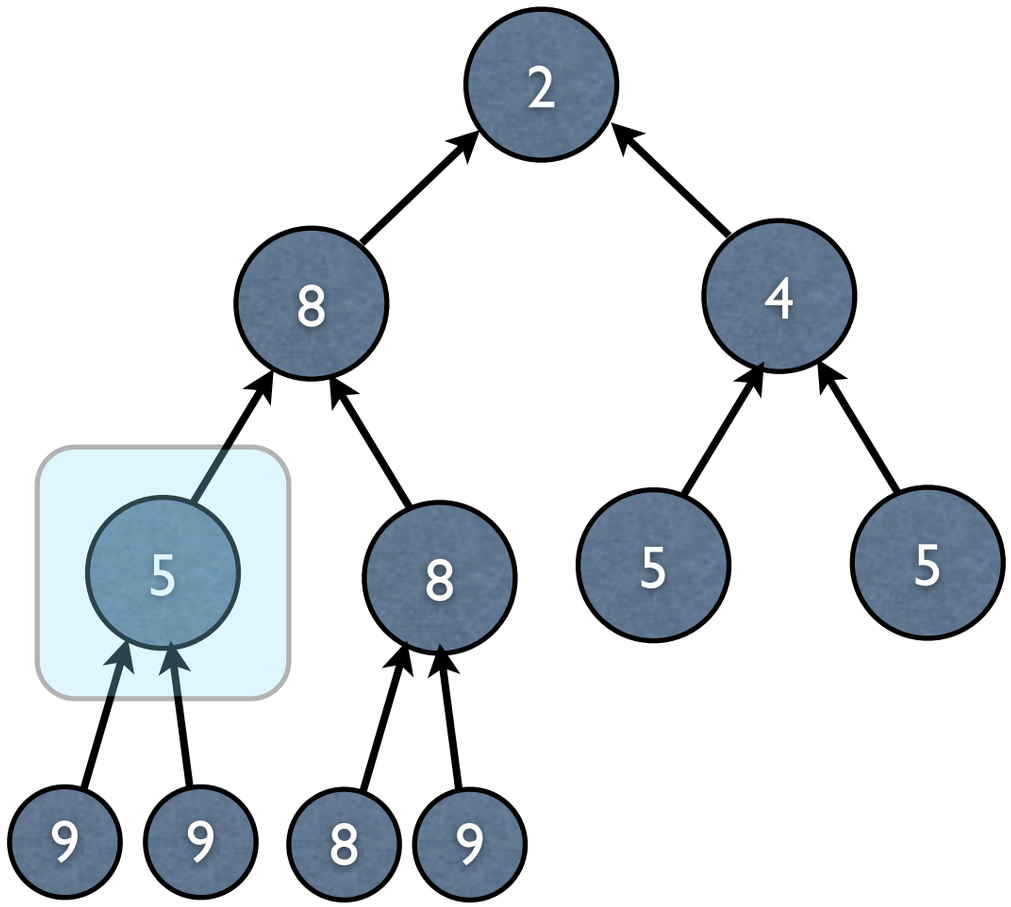
Case 2. key[v] < key[parent[v]]

Swap v with its **parent**

Continue till heap property holds

Heap: Delete an item





Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

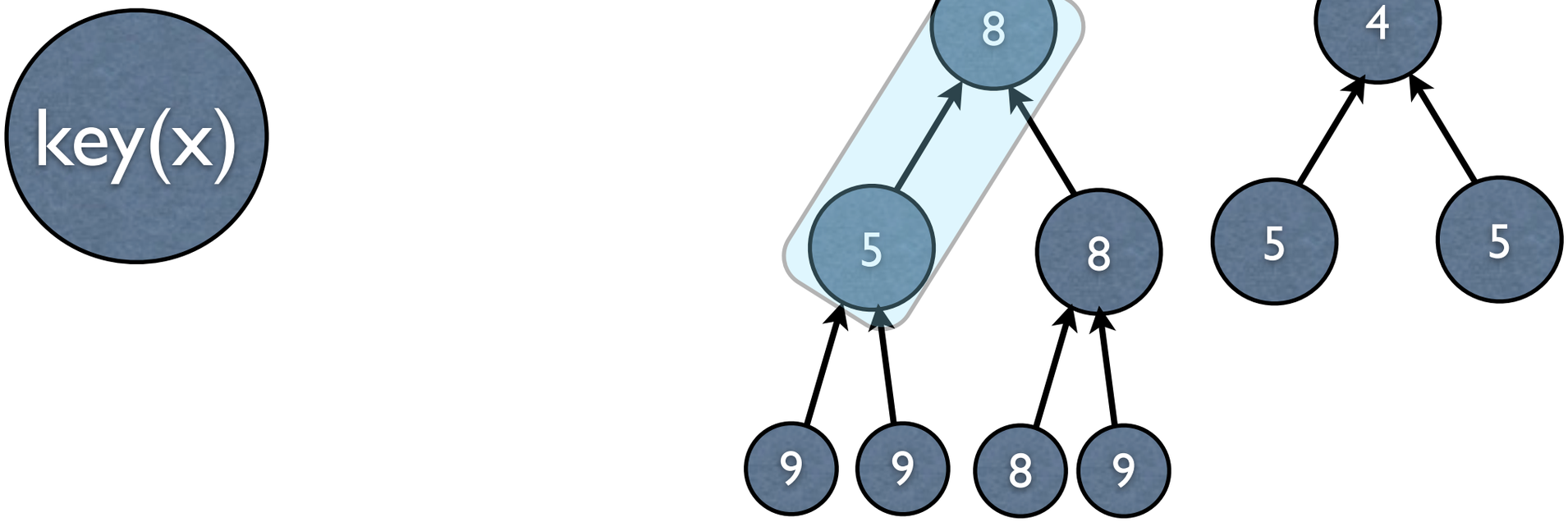
Time = O(log n)

Case 2. key[v] < key[parent[v]]

Swap v with its **parent**

Continue till heap property holds

Heap: Delete an item



Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

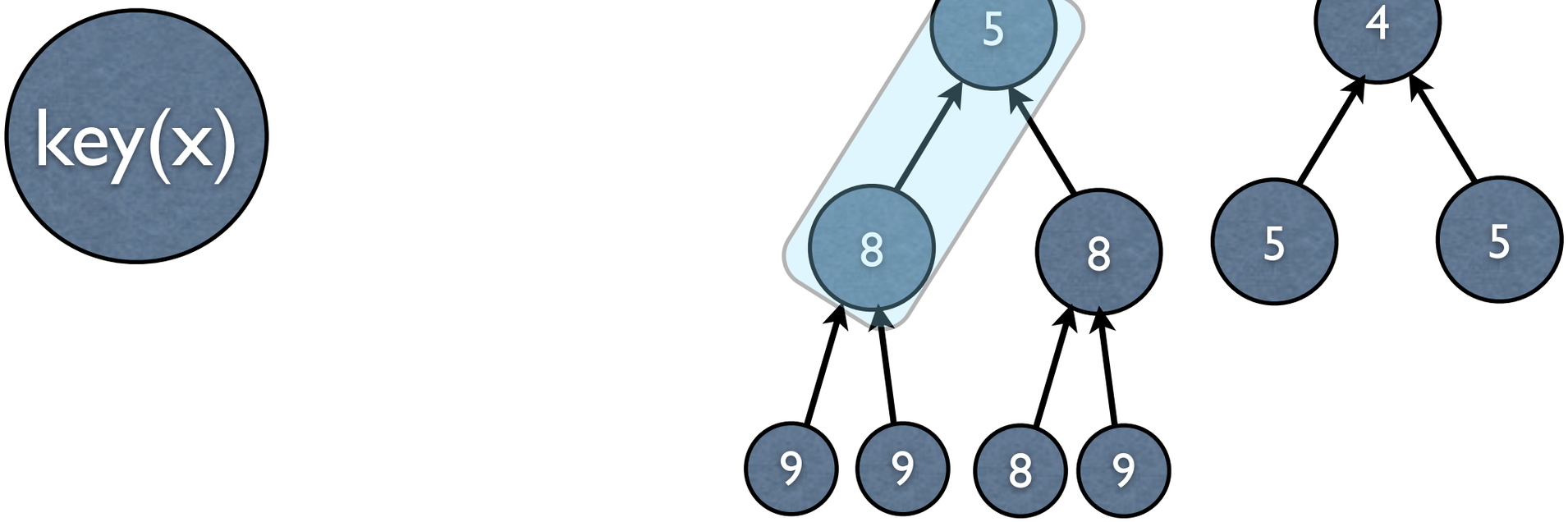
Time = O(log n)

Case 2. key[v] < key[parent[v]]

Swap v with its **parent**

Continue till heap property holds

Heap: Delete an item



Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

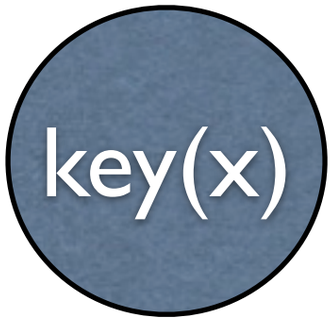
Time = O(log n)

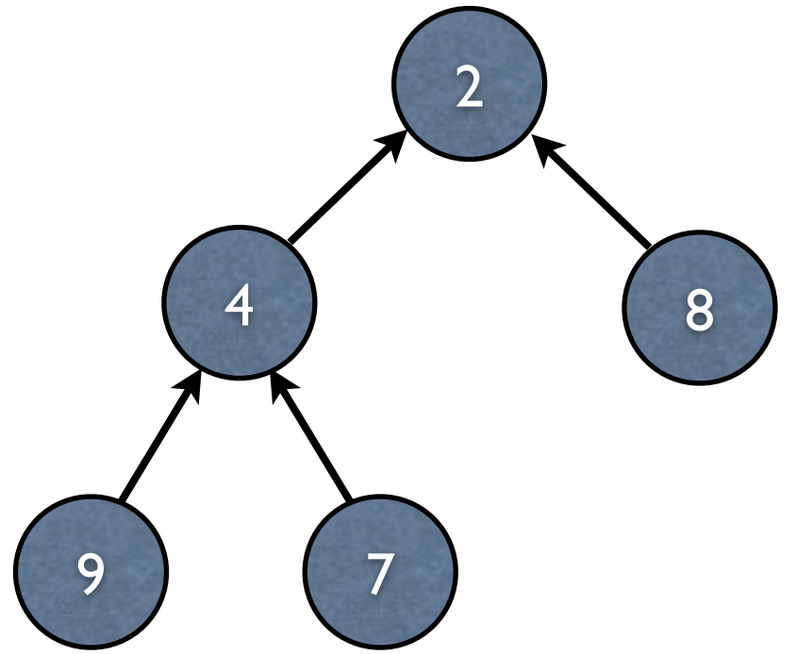
Case 2. key[v] < key[parent[v]]

Swap v with its **parent**

Continue till heap property holds

Summary: Heap





Heap Property: If x is the parent of y, then key(x) <= key(y)

Operations:

Add an element: O(log n)
Delete an element: O(log n)
Report min: O(1)

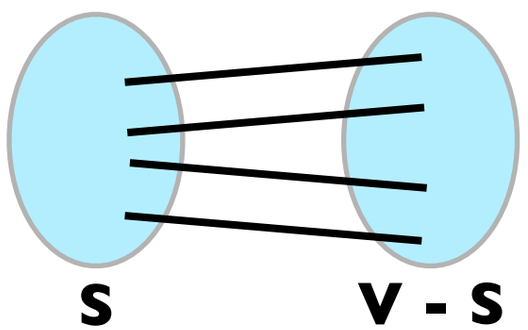
Prim's Algorithm

X = { }, S = {r}
Repeat until S has n nodes:

Pick the **lightest** edge e in the cut (S, V - S)
Add e to X
Add v, the end-point of e in V - S to S

Use a **heap** to store edges between S and V - S
At each step:

1. Pick lightest edge with a report-min
2. Delete all edges b/w v and S from heap
3. Add all edges b/w v and V - S - {v}



Black edges = in heap

Prim's Algorithm

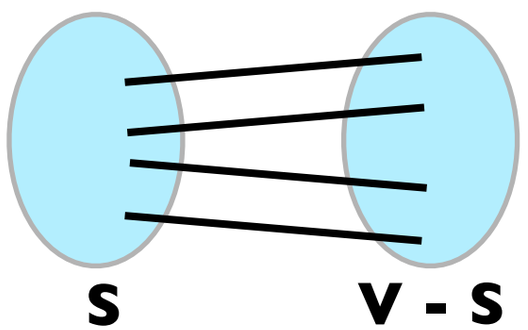
X = { }, S = {r}
Repeat until S has n nodes:

Pick the **lightest** edge e in the cut (S, V - S)
Add e to X
Add v, the end-point of e in V - S to S

Use a **heap** to store edges between S and V - S
At each step:

1. Pick lightest edge with a report-min
2. Delete all edges b/w v and S from heap
3. Add all edges b/w v and V - S - {v}

#edge additions and deletions = O(m) (Why?)
#report mins = O(n)



Black edges = in heap

Prim's Algorithm

X = { }, S = {r}
Repeat until S has n nodes:

Pick the **lightest** edge e in the cut (S, V - S)
Add e to X
Add v, the end-point of e in V - S to S

Use a **heap** to store edges b/w S and V - S
At each step:

1. Pick lightest edge with a report-min
2. Delete all edges b/w v and S from heap
3. Add all edges b/w v and V - S - {v}

#edge additions and deletions = O(m)
#report mins = O(n)

Heap Ops:

Add: O(log n)
Delete: O(log n)
Report min: O(1)

Prim's Algorithm

X = { }, S = {r}
Repeat until S has n nodes:

Pick the **lightest** edge e in the cut (S, V - S)
Add e to X
Add v, the end-point of e in V - S to S

Use a **heap** to store edges b/w S and V - S
At each step:

1. Pick lightest edge with a report-min
2. Delete all edges b/w v and S from heap
3. Add all edges b/w v and V - S - {v}

#edge additions and deletions = O(m)
#report mins = O(n)

Total running time = O(m log n)

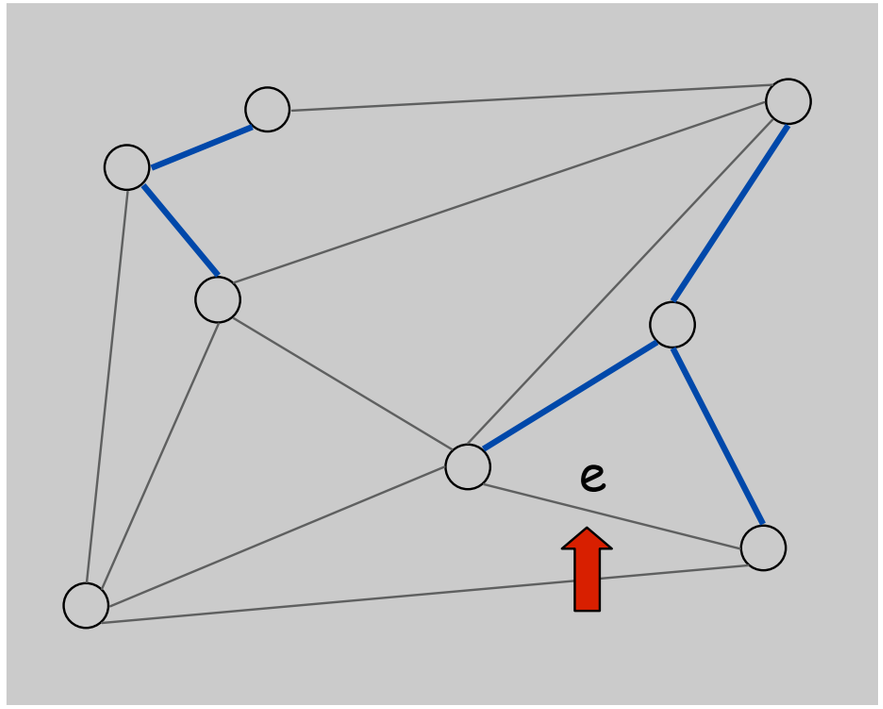
Heap Ops:

Add: O(log n)
Delete: O(log n)
Report min: O(1)

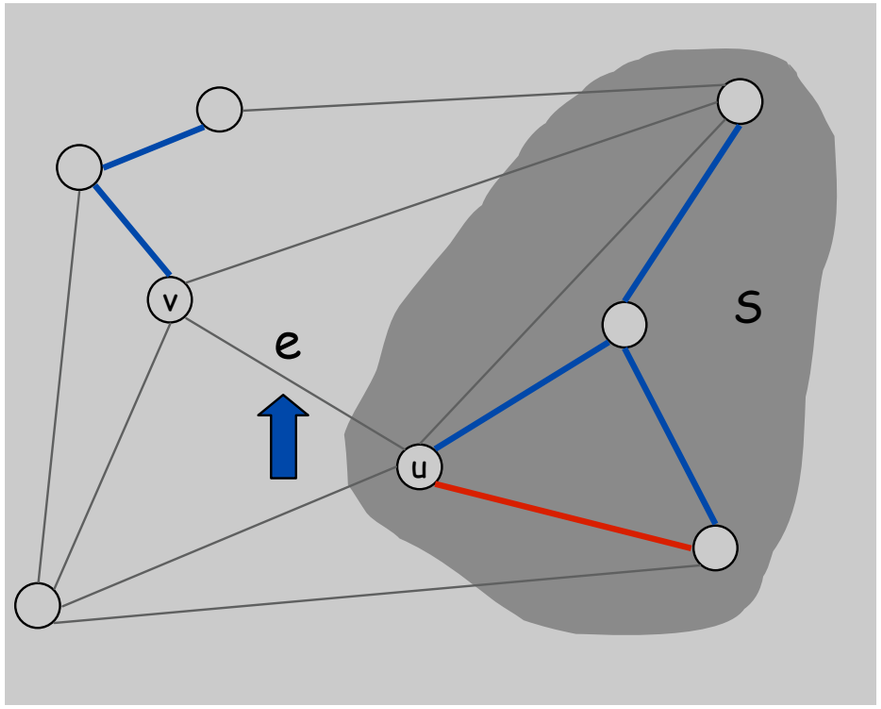
Kruskal's Algorithm

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1



Case 2

Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure.

- Build set T of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \alpha(m, n))$ for union-find.

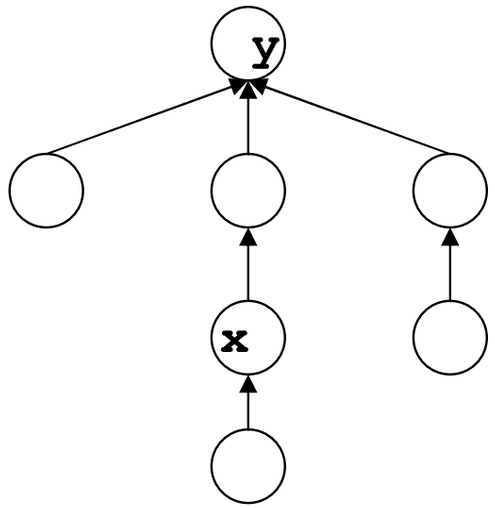
$m \le n^2 \Rightarrow \log m$ is $O(\log n)$

essentially a constant

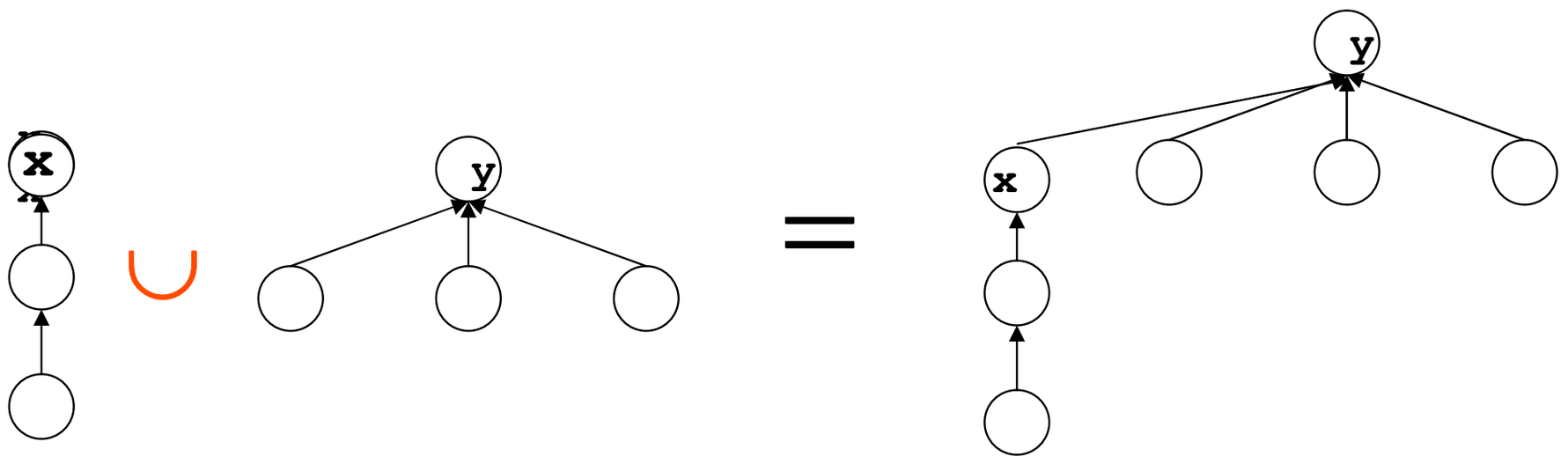
```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m                    are u and v in different connected components?
      (u,v) = ei                     ↙
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }                              ↖ merge two components
   return T
}
```

Union-Find Implementation

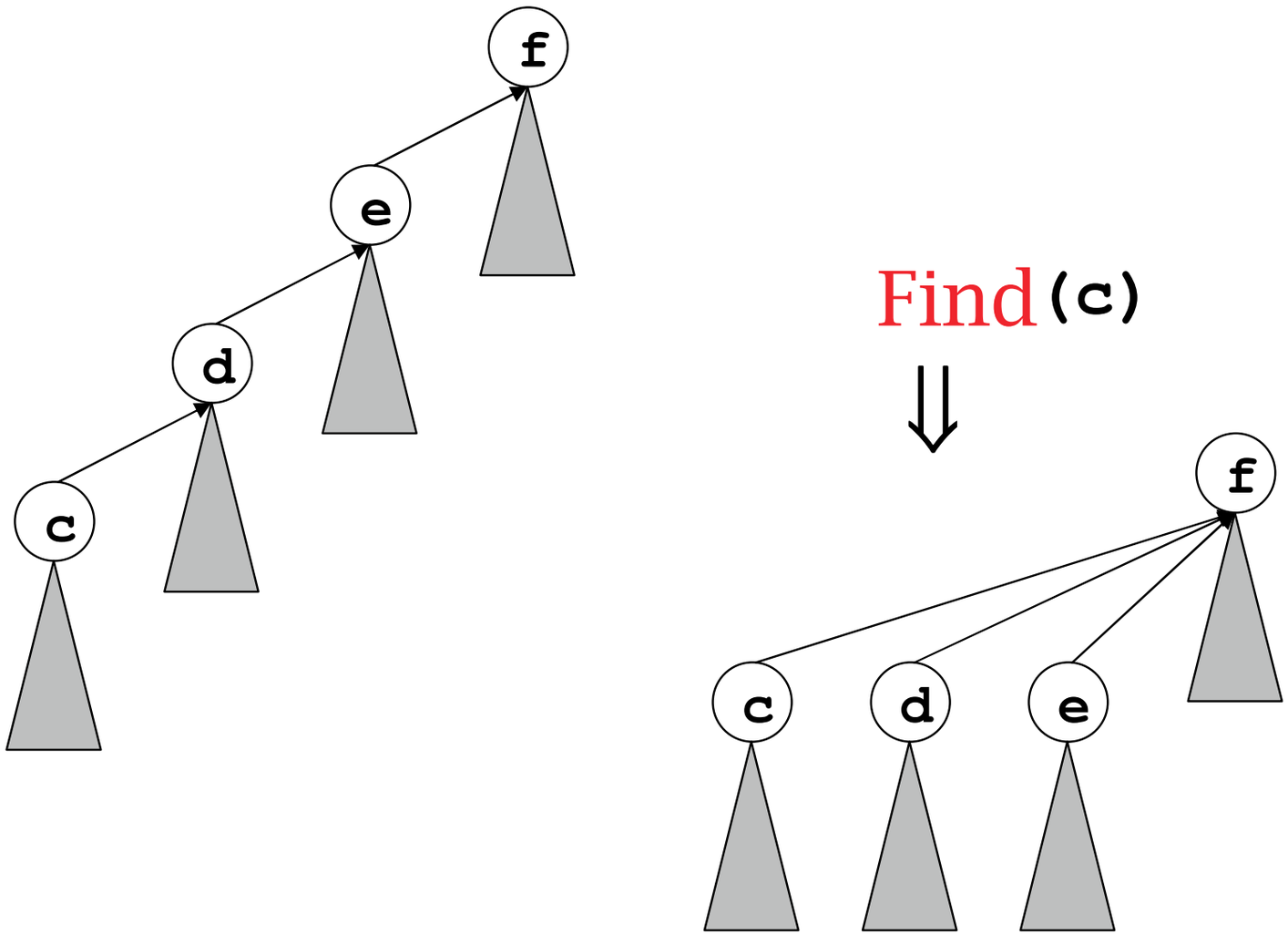
Find(x) -
follow pointers
from x up to root



Union(x,y) - make x a child of y and return y



Path Compression



Summary: Kruskal's Algorithm Running Time

X = { }
For each edge e in **increasing order** of weight:
    If the end-points of e lie in different components in X,
    Add e to X

Sort the edges = O(m log m) = O(m log n)
Add e to X = Union Operation = O(1) + Time(Find)
Check if end-points of e lie in different components = Find Operation

Total time = Sort + O(n) Unions + O(m) Finds = **O(m log n)**
With sorted edges, time = O(n) Unions + O(m) Finds = **O(m log*n)**

The function lg^*n

lg^*n = the number of times we have to take the $\log_2$ of n repeatedly to reach 1

- $lg^* 2 = 1$
- $lg^* 3 = lg^* 4 = lg^* 2^2 = 2$
- $lg^* 16 = lg^* 2^{2^2} = 3$
- $lg^* 65536 = lg^* 2^{2^{2^{2^2}}} = 4$

$\Rightarrow$ $lg^*n \leq 5$ for all practical values of n

Theorem(Tarjan): If

S = a sequence of O(n) Unions and Find-Sets

The worst-case time for S with

- Weighted Unions, and
- Path Compressions

is $O(n \lg^* n)$

$\Rightarrow$ The average time is $O(\lg^* n)$ per operation

Ackerman's function

Define the function $A_i(x)$ inductively by

$A_0(x) = x+1$

$A_{i+1}(x) = A_i(A_i(A_i...(x)))$, where A_i is applied x+1 times.

$A_1(x) = 2x+1$

$A_2(x) = A_1(A_1(A_1(\ldots x))) > 2^{x+1}$

$A_3(x) = A_2(A_2(A_2(\ldots x))) > \underbrace{2\text{^}(2\text{^}(2\text{^}\ldots(x)))}_{x+1}$

The Inverse Ackerman function is

$\alpha(n) = \min\{k: A_k(1) > n\}$

Theorem(Tarjan): Let

S = sequence of $\Omega(n)$ **Unions** and **Find-Sets**

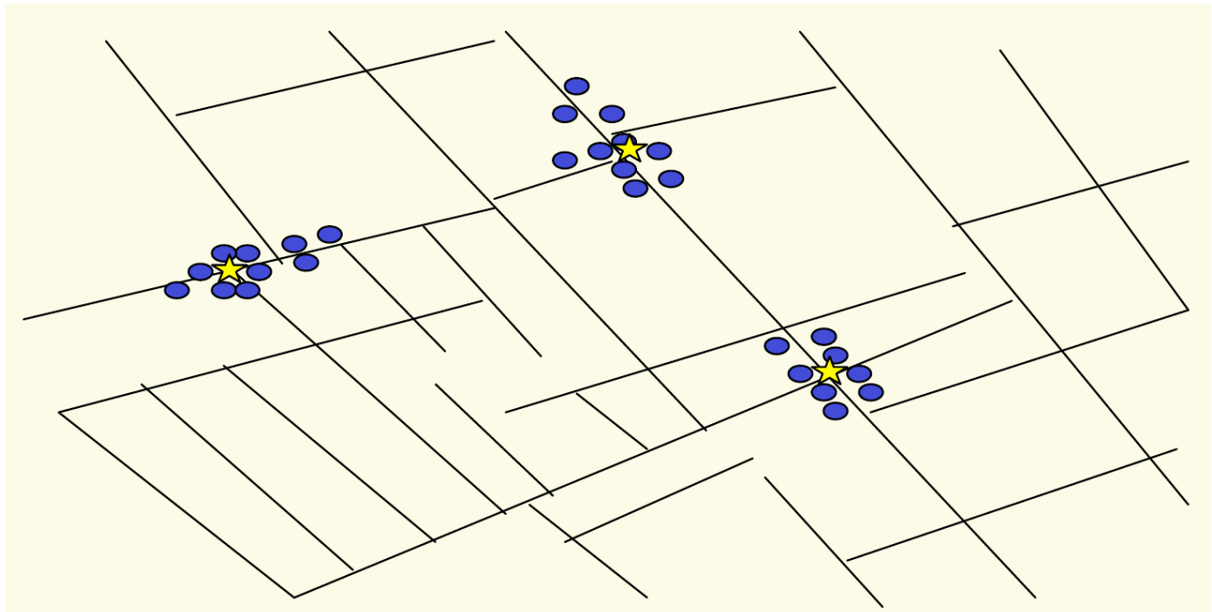
The worst-case time for S with

- Weighted Unions, and
- Path Compressions

is $O(n\alpha(n))$

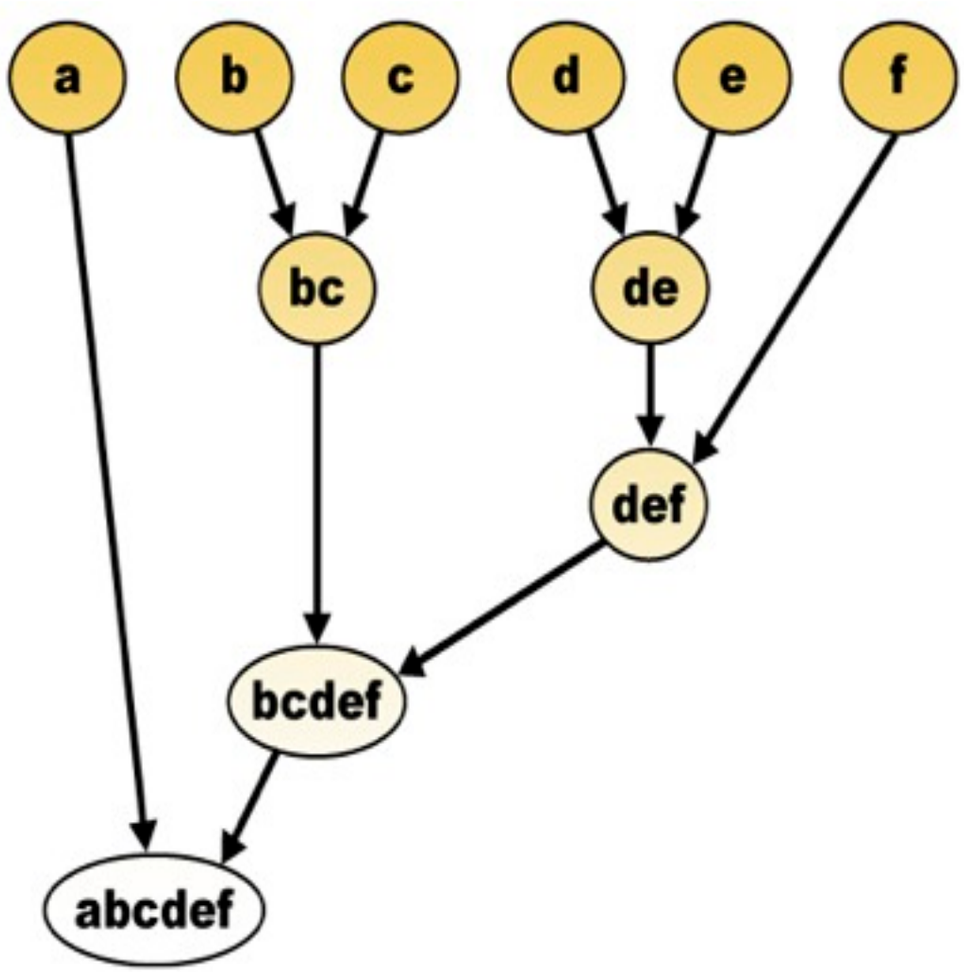
$\Rightarrow$ The average time is $O(\alpha(n))$ steps per operation

4.7 Clustering



Outbreak of cholera deaths in London in 1850s.
Reference: Nina Mishra, HP Labs

Single Linkage Clustering



Problem: Given a set of points, build a hierarchical clustering

Procedure:
Initialize: each node is a cluster
Until we have one cluster:
  Pick two **closest** clusters C, C*
  Merge S = C U C*

Distance between two clusters:
$d(C, C^*) = \min_{x \text{ in } C, y \text{ in } C^*} d(x, y)$

Can you recognize this algorithm?

Clustering

Clustering. Given a set U of n objects labeled p_1, …, p_n, classify into coherent groups.

↑ photos, documents. micro-organisms

Distance function. Numeric value specifying "closeness" of two objects.

↑ number of corresponding pixels whose intensities differ by some threshold

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Clustering of Maximum Spacing

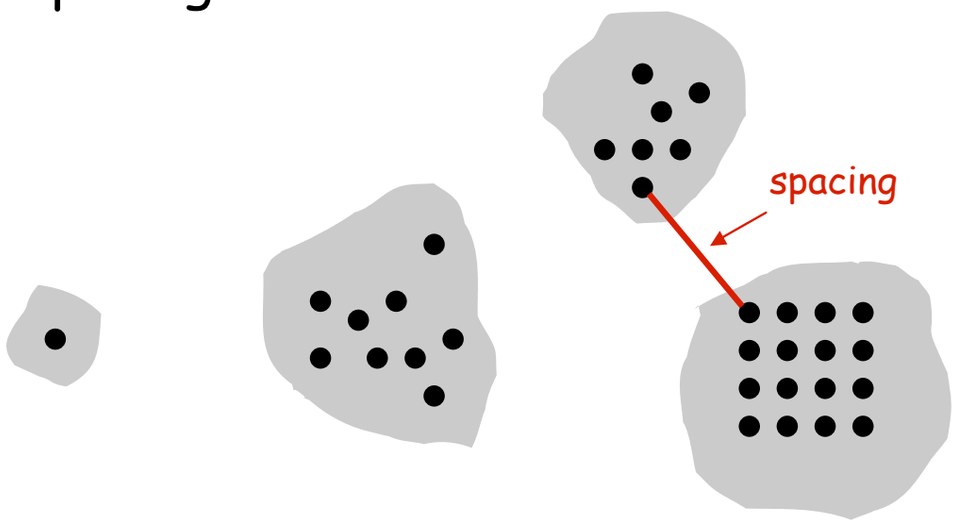
k-clustering. Divide objects into k non-empty groups.

Distance function. Assume it satisfies several natural properties.

- $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
- $d(p_i, p_j) \geq 0$ (nonnegativity)
- $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Spacing. Min distance between any pair of points in different clusters.

Clustering of maximum spacing. Given an integer k, find a k-clustering of maximum spacing.



k = 4

Greedy Clustering Algorithm

Single-link k-clustering algorithm.

- Form a graph on the vertex set U, corresponding to n clusters.
- Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
- Repeat n-k times until there are exactly k clusters.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

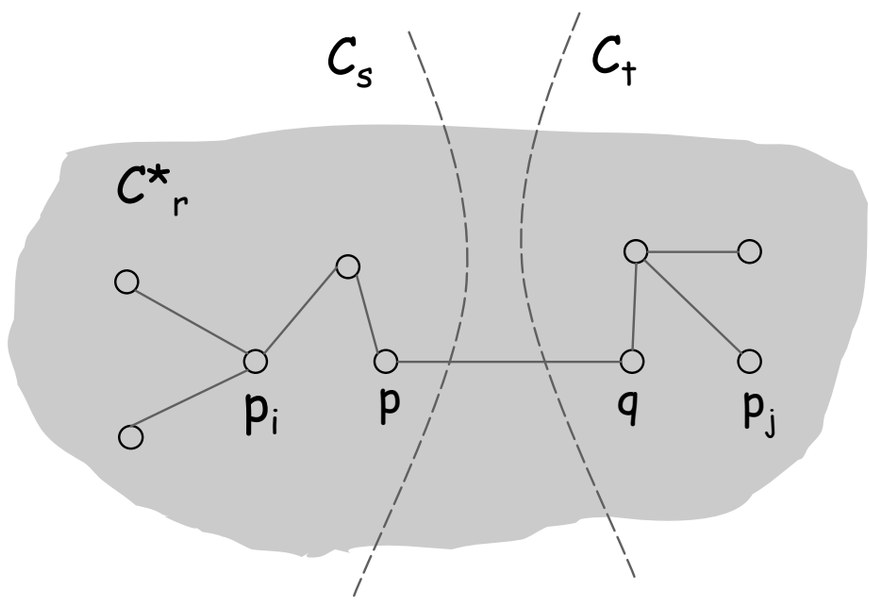
Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.

Greedy Clustering Algorithm: Analysis

Theorem. Let C^* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the k-1 most expensive edges of a MST. C^* is a k-clustering of max spacing.

Pf. Let C denote some other clustering $C_1, \ldots, C_k$.

- The spacing of C^* is the length d^* of the $(k-1)^{st}$ most expensive edge.
- Let p_i, p_j be in the same cluster in C^*, say C^*_r, but different clusters in C, say C_s and C_t.
- Some edge (p, q) on p_i-p_j path in C^*_r spans two different clusters in C.
- All edges on p_i-p_j path have length $\leq d^*$ since Kruskal chose them.
- Spacing of C is $\leq d^*$ since p and q are in different clusters. ▪



Extra Slides

MST Algorithms: Theory

Deterministic comparison based algorithms.

- $O(m \log n)$ [Jarník, Prim, Dijkstra, Kruskal, Boruvka]
- $O(m \log \log n)$. [Cheriton-Tarjan 1976, Yao 1975]
- $O(m \beta(m, n))$. [Fredman-Tarjan 1987]
- $O(m \log \beta(m, n))$. [Gabow-Galil-Spencer-Tarjan 1986]
- $O(m \alpha(m, n))$. [Chazelle 2000]

Holy grail. $O(m)$.

Notable.

- $O(m)$ randomized. [Karger-Klein-Tarjan 1995]
- $O(m)$ verification. [Dixon-Rauch-Tarjan 1992]

Euclidean.

- 2-d: $O(n \log n)$. compute MST of edges in Delaunay
- k-d: $O(k n^2)$. dense Prim