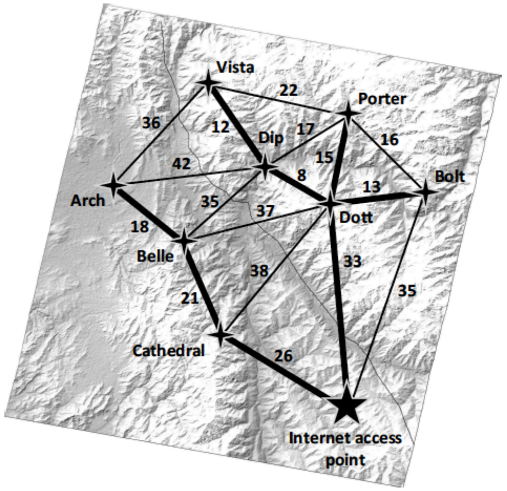
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Minimum Spanning Trees

Saguaros, Saguaro National Monument, 1941. Ansel Adams. U.S. government image. U.S. National Archives and Records Administration.

Application: Connecting a Network

- Suppose the remote mountain country of Vectoria has been given a major grant to install a large Wi-Fi the center of each of its mountain villages.
- Communication cables can run from the main Internet access point to a village tower and cables can also run between pairs of towers.
- The challenge is to interconnect all the towers and the Internet access point as **cheaply** as possible.

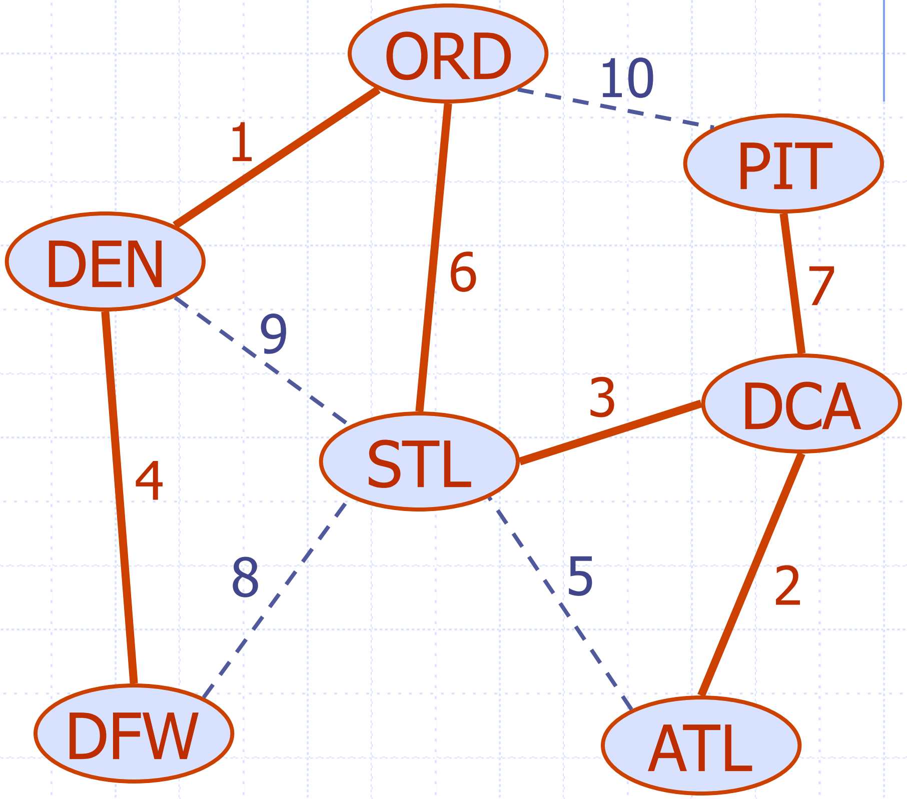


Application: Connecting a Network

- We can model this problem using a graph, **G**, where each vertex in **G** is the location of a Wi-Fi the Internet access point, and an edge in **G** is a possible cable we could run between two such vertices.
- Each edge in **G** could then be given a weight that is equal to the cost of running the cable that that edge represents.
- Thus, we are interested in finding a connected acyclic subgraph of **G** that includes all the vertices of **G** and has minimum total cost.
- Using the language of graph theory, we are interested in finding a **minimum spanning tree (MST)** of **G**.

Minimum Spanning Trees

- Spanning subgraph
  - Subgraph of a graph ***G*** containing all the vertices of ***G***
- Spanning tree
  - Spanning subgraph that is itself a (free) tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation networks



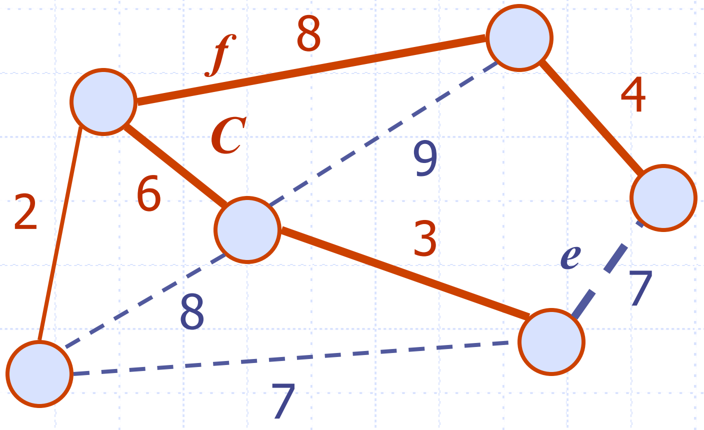
Cycle Property

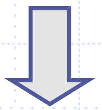
Cycle Property:

- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

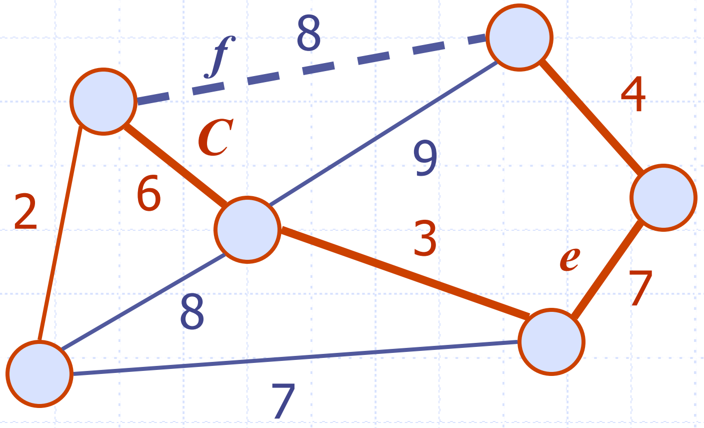
Proof:

- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f





Replacing f with e yields a better spanning tree



Partition Property

Partition Property:

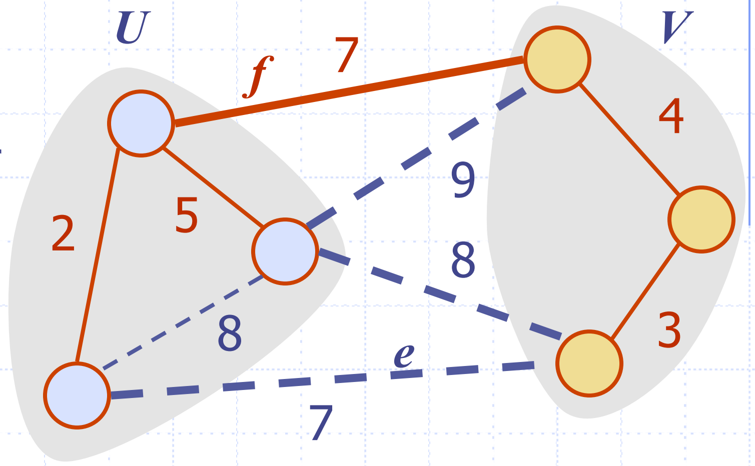
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

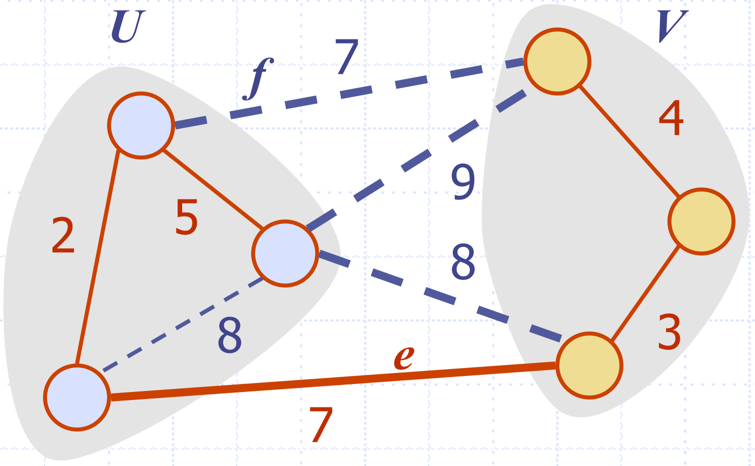
- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property,

$$weight(f) \leq weight(e)$$

- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e



Replacing f with e yields another MST



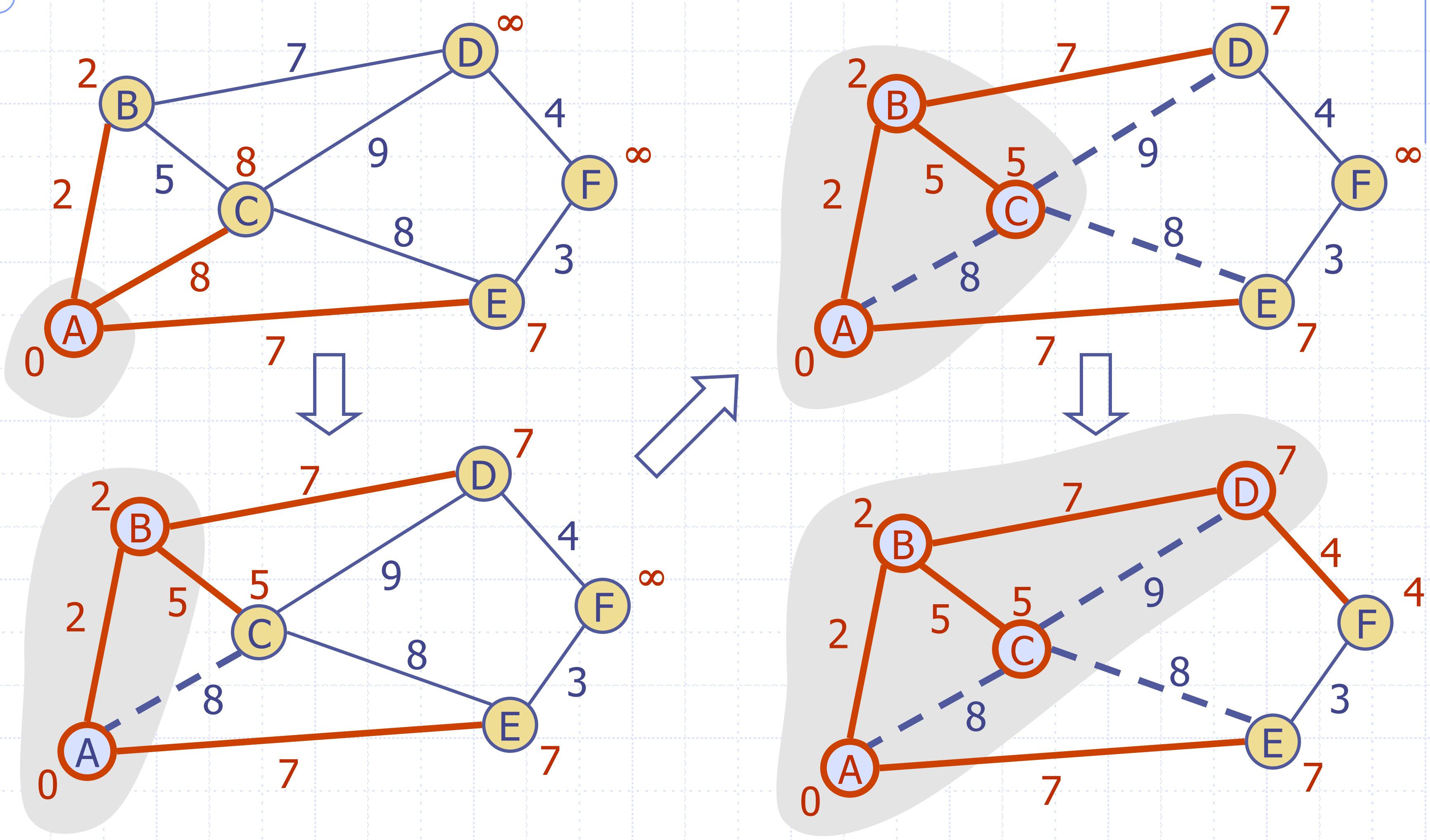
Prim-Jarnik's Algorithm

- Similar to Dijkstra's algorithm
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

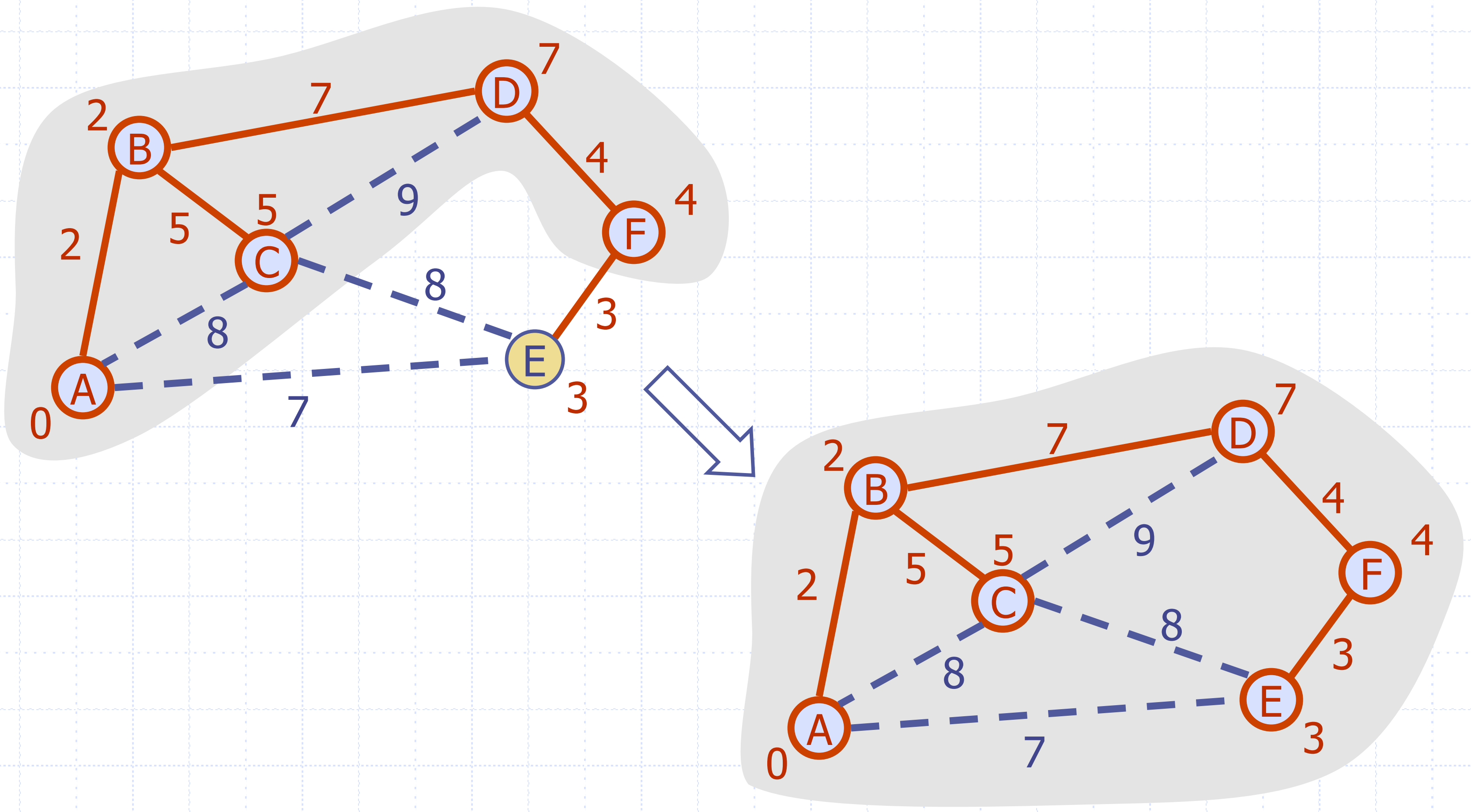
Prim-Jarnik Pseudo-code

```
Algorithm PrimJarníkMST($G$):
  Input: A weighted connected graph $G$ with $n$ vertices and $m$ edges
  Output: A minimum spanning tree $T$ for $G$
   Pick any vertex $v$ of $G$
   $D[v] \leftarrow 0$
   for each vertex $u \neq v$ do
       $D[u] \leftarrow +\infty$
   Initialize $T \leftarrow \emptyset$.
   Initialize a priority queue $Q$ with an item $((u, \text{null}), D[u])$ for each vertex $u$,
   where $(u, \text{null})$ is the element and $D[u]$ is the key.
   while $Q$ is not empty do
       $(u, e) \leftarrow Q$.removeMin()
       Add vertex $u$ and edge $e$ to $T$.
       for each vertex $z$ adjacent to $u$ such that $z$ is in $Q$ do
           // perform the relaxation procedure on edge $(u, z)$
           if $w((u, z)) < D[z]$ then
               $D[z] \leftarrow w((u, z))$
               Change to $(z, (u, z))$ the element of vertex $z$ in $Q$.
               Change to $D[z]$ the key of vertex $z$ in $Q$.
   return the tree $T$
```

Example



Example (contd.)



Analysis

- Graph operations
  - We cycle through the incident edges once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

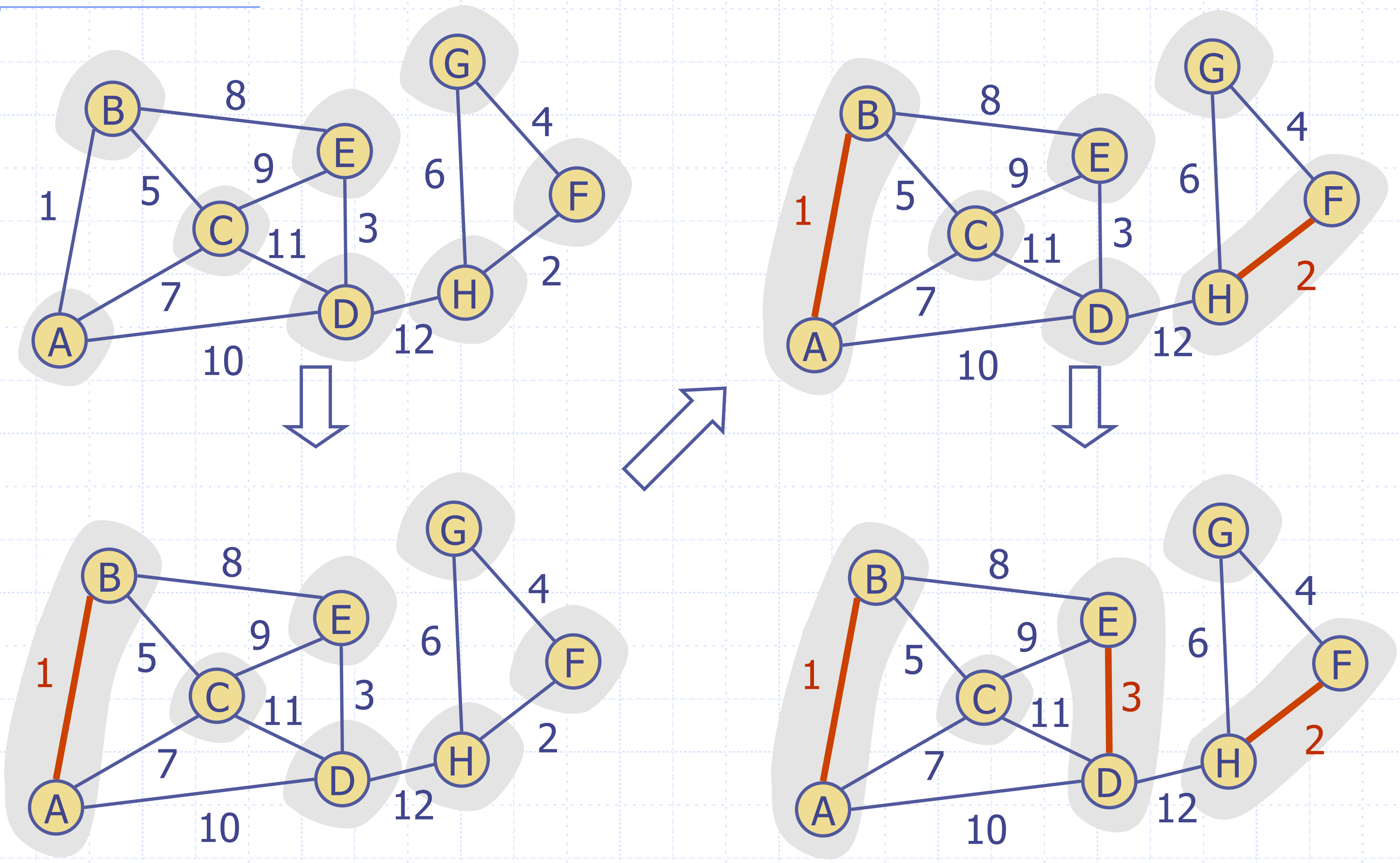
Kruskal's Approach

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge "closest" clusters and their MSTs
- A priority queue stores the edges outside clusters (or you could even sort the edges)
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster and one MST

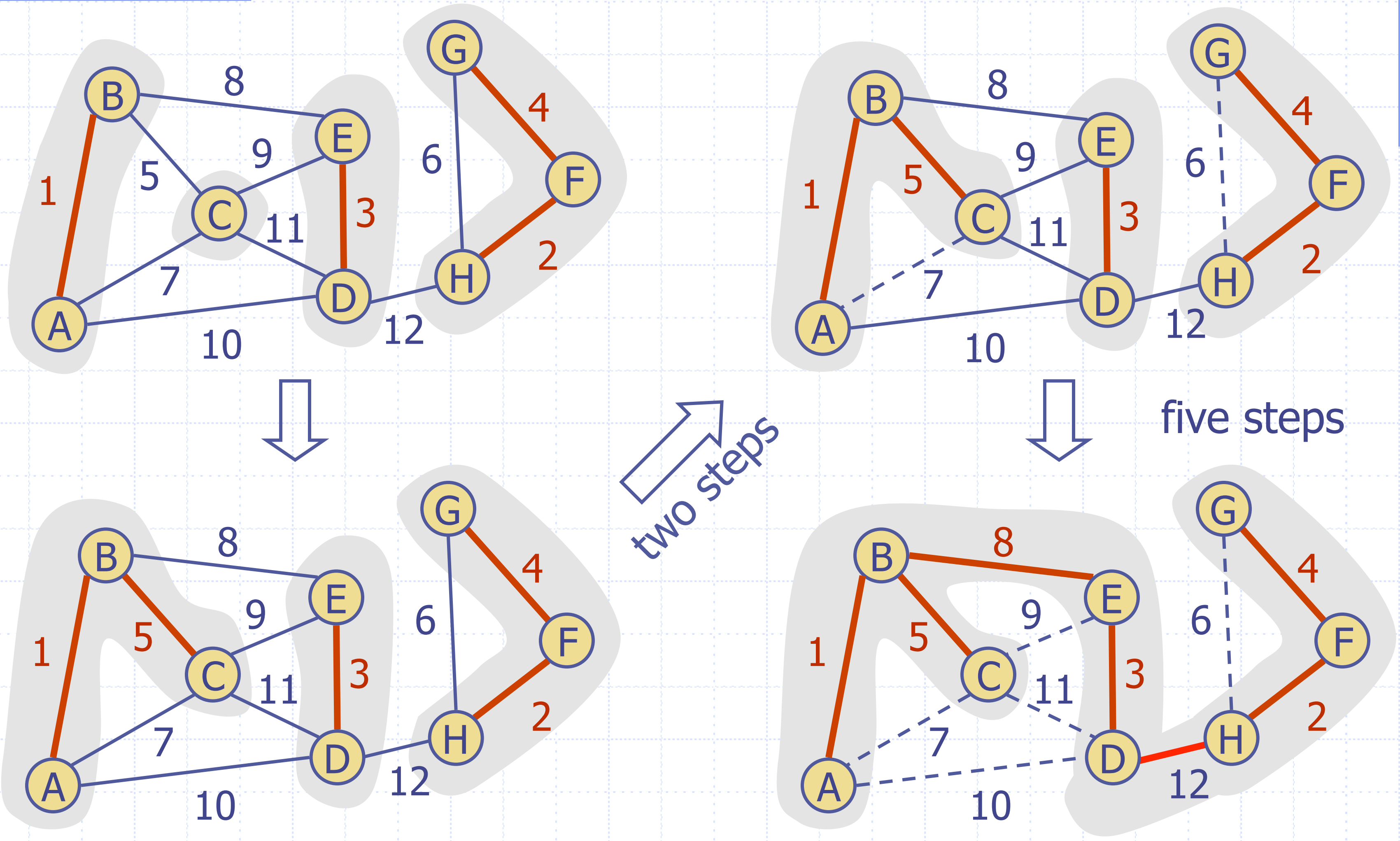
Kruskal’s Algorithm

```
Algorithm KruskalMST($G$):
    Input: A simple connected weighted graph $G$ with $n$ vertices and $m$ edges
    Output: A minimum spanning tree $T$ for $G$
  for each vertex $v$ in $G$ do
      Define an elementary cluster $C(v) \leftarrow \{v\}$.
  Let $Q$ be a priority queue storing the edges in $G$, using edge weights as keys
  $T \leftarrow \emptyset$        // $T$ will ultimately contain the edges of the MST
  while $T$ has fewer than $n - 1$ edges do
      $(u, v) \leftarrow Q$.removeMin()
      Let $C(v)$ be the cluster containing $v$
      Let $C(u)$ be the cluster containing $u$
      if $C(v) \neq C(u)$ then
          Add edge $(v, u)$ to $T$
          Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$
  return tree $T$
```

Example of Kruskal's Algorithm



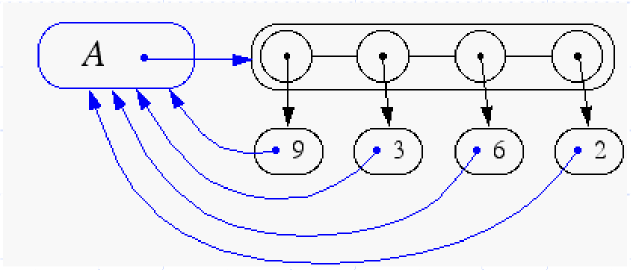
Example (contd.)



Data Structure for Kruskal's Algorithm

- The algorithm maintains a forest of trees
- A priority queue extracts the edges by increasing weight
- An edge is accepted it if connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with operations:
  - makeSet(u): create a set consisting of u
  - find(u): return the set storing u
  - union(A, B): replace sets A and B with their union

List-based Partition

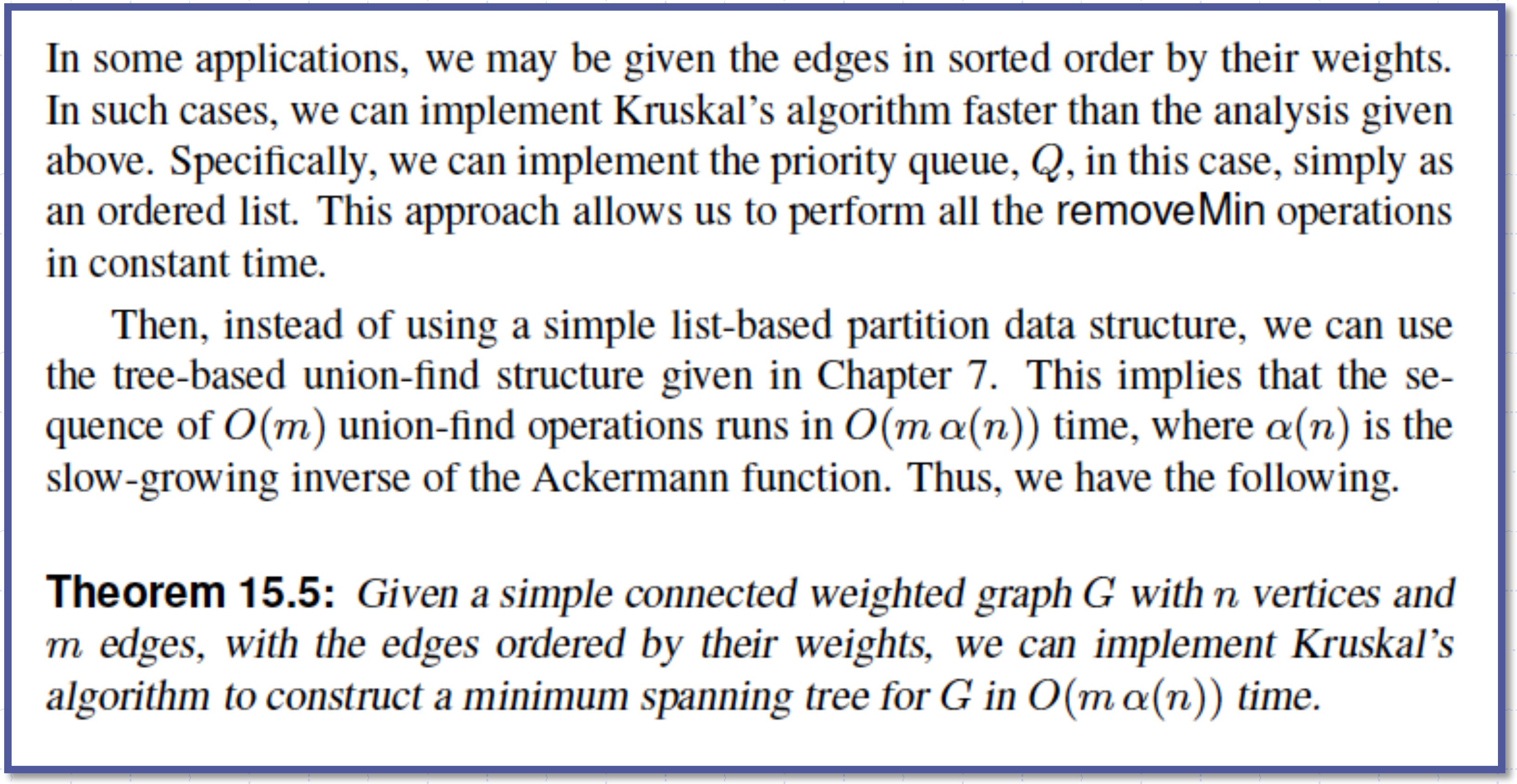


- Each set is stored in a sequence
- Each element has a reference back to the set
  - operation find(u) takes O(1) time, and returns the set of which u is a member.
  - in operation union(A,B), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation union(A,B) is min(|A|, |B|)
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most log n times

Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
  - Cluster merges as unions
  - Cluster locations as finds
- Running time $O((n + m) \log n)$
  - Priority Queue operations: $O(m \log n)$
  - Union-Find operations: $O(n \log n)$

Alternative Implementation



In some applications, we may be given the edges in sorted order by their weights. In such cases, we can implement Kruskal's algorithm faster than the analysis given above. Specifically, we can implement the priority queue, Q, in this case, simply as an ordered list. This approach allows us to perform all the removeMin operations in constant time.

Then, instead of using a simple list-based partition data structure, we can use the tree-based union-find structure given in Chapter 7. This implies that the sequence of $O(m)$ union-find operations runs in $O(m\,\alpha(n))$ time, where $\alpha(n)$ is the slow-growing inverse of the Ackermann function. Thus, we have the following.

Theorem 15.5: *Given a simple connected weighted graph G with n vertices and m edges, with the edges ordered by their weights, we can implement Kruskal's algorithm to construct a minimum spanning tree for G in $O(m\,\alpha(n))$ time.*

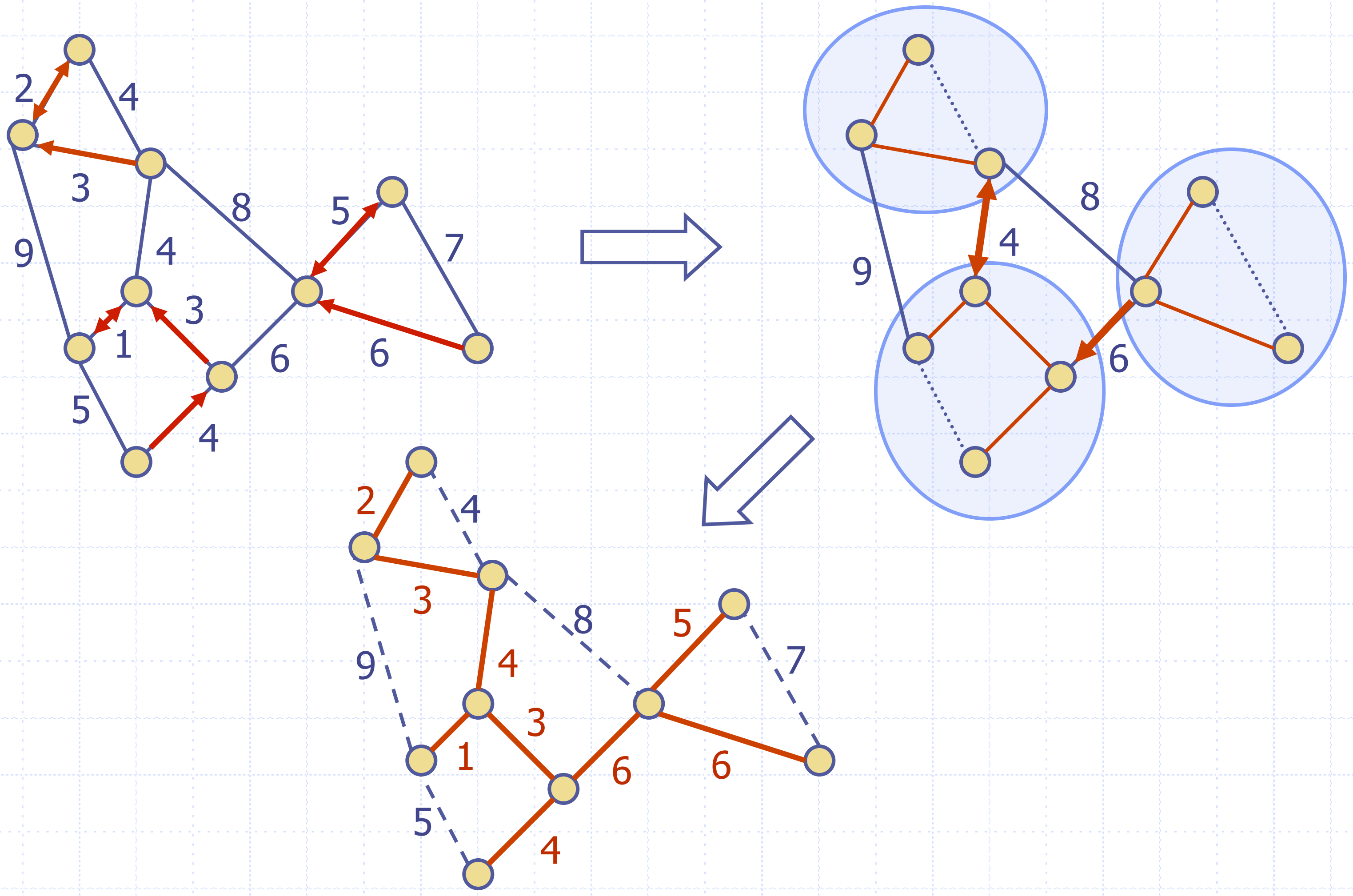
Baruvka’s Algorithm

- Like Kruskal’s Algorithm, Baruvka’s algorithm grows many clusters at once and maintains a forest T
- Each iteration of the while loop halves the number of connected components in forest T
- The running time is $O(m \log n)$

```
Algorithm BaruvkaMST(G)
    $T \leftarrow V$  {just the vertices of $G$}
  while $T$ has fewer than $n - 1$ edges do
    for each connected component $C$ in $T$ do
      Let edge $e$ be the smallest-weight edge from $C$ to another component in $T$
      if $e$ is not already in $T$ then
        Add edge $e$ to $T$
  return $T$
```

Example of Baruvka's Algorithm (animated)

Slide by Matt Stallmann included with permission.


2
4
3
9
4
8
5
7
1
3
6
6
5
4