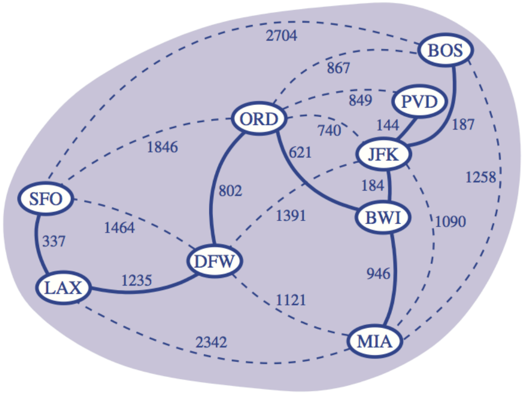
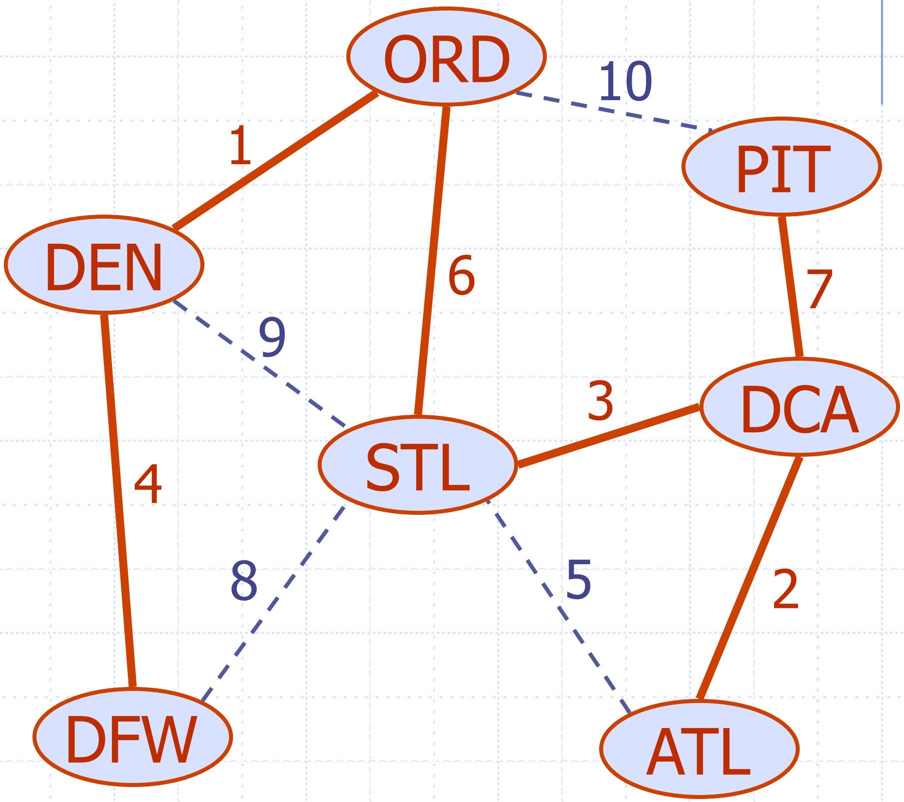
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Minimum Spanning Trees



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



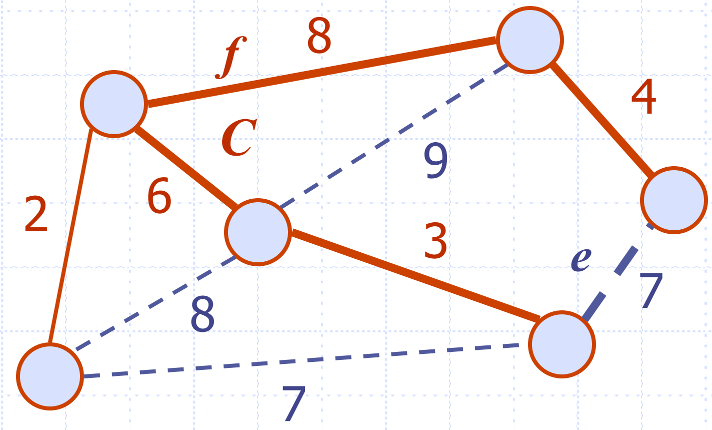
Cycle Property

Cycle Property:

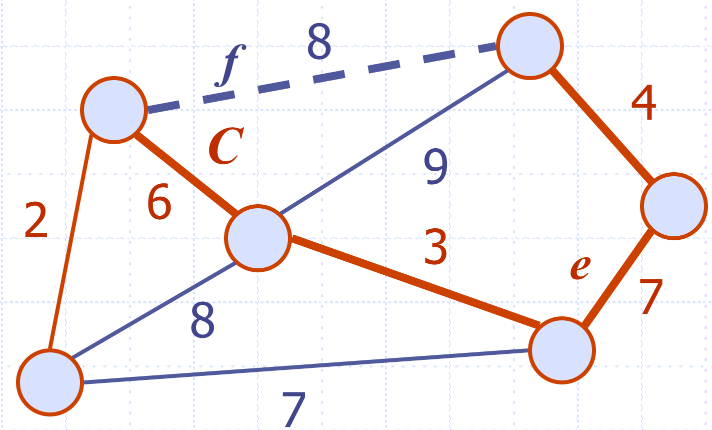
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \le weight(e)$

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:

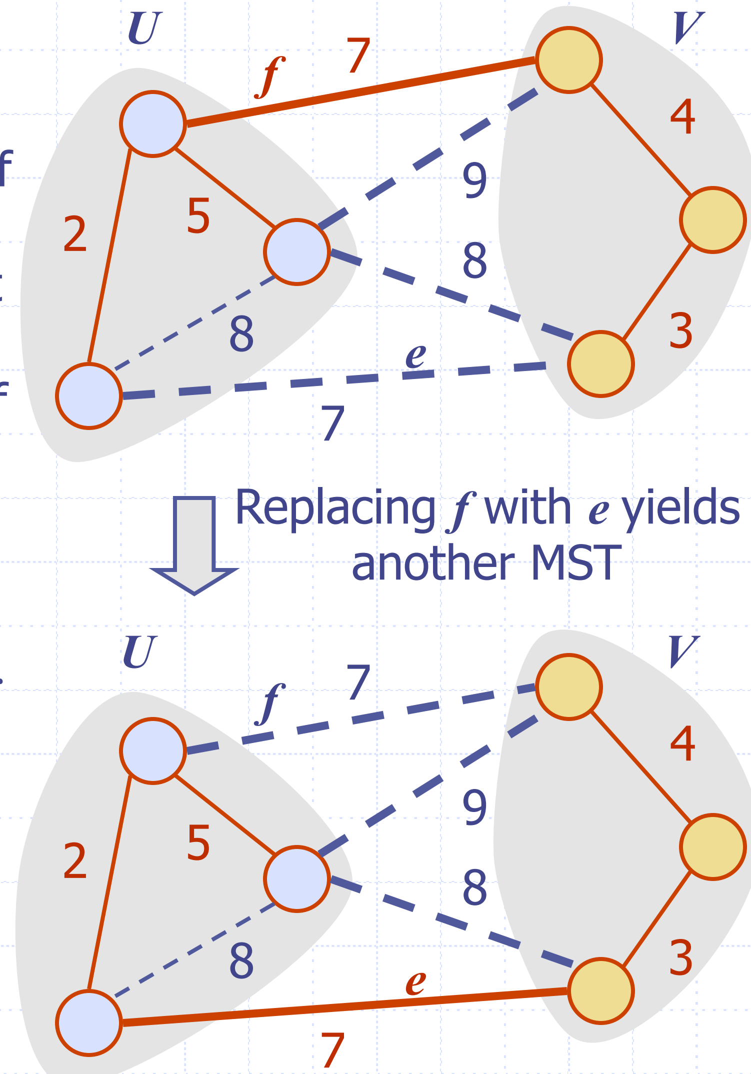
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

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



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

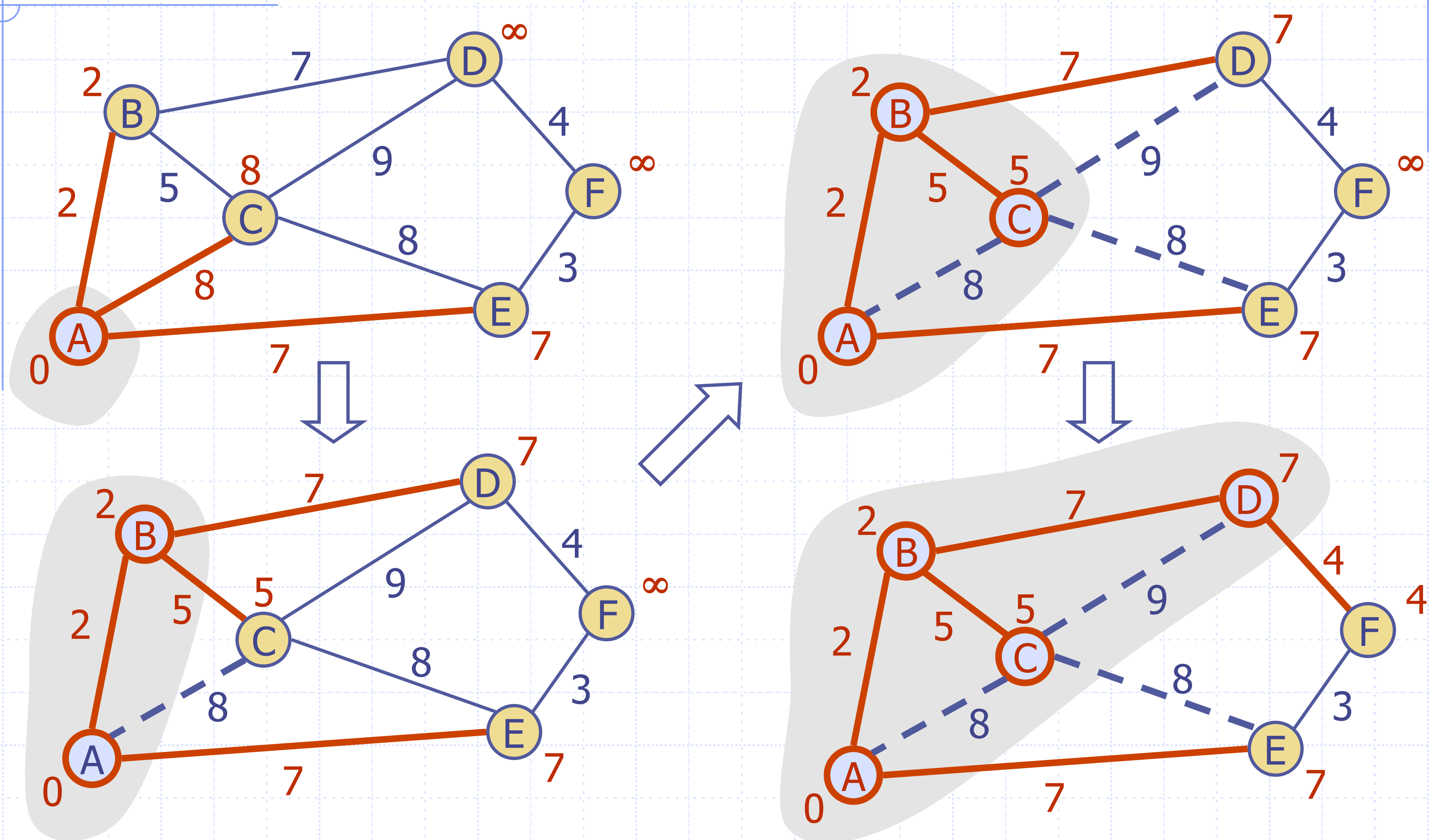
Prim-Jarnik Pseudo-code

Algorithm PrimJarnik(G):

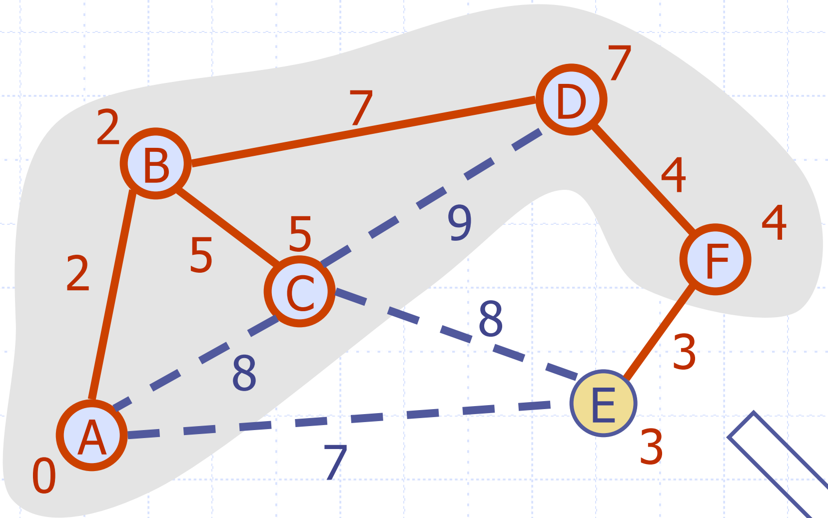
  Input: An undirected, weighted, connected graph G with n vertices and m edges
  Output: A minimum spanning tree T for G

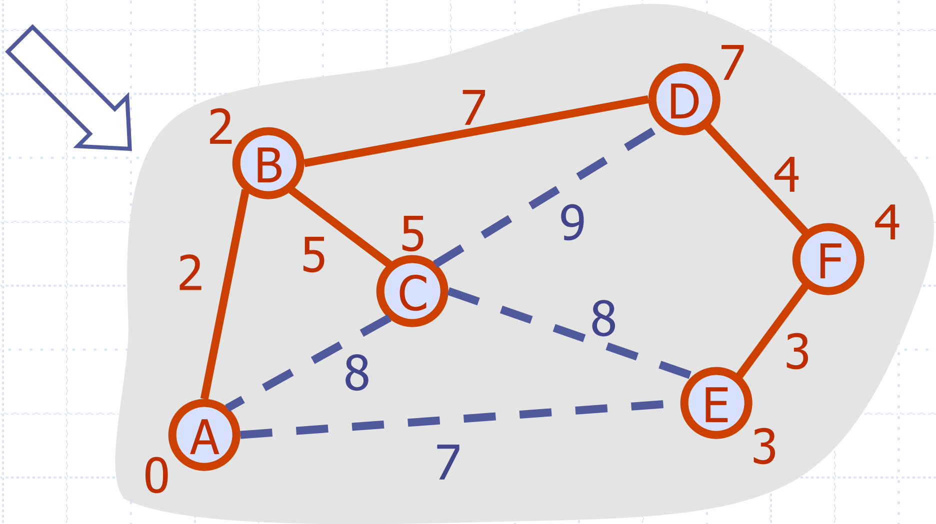
Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
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

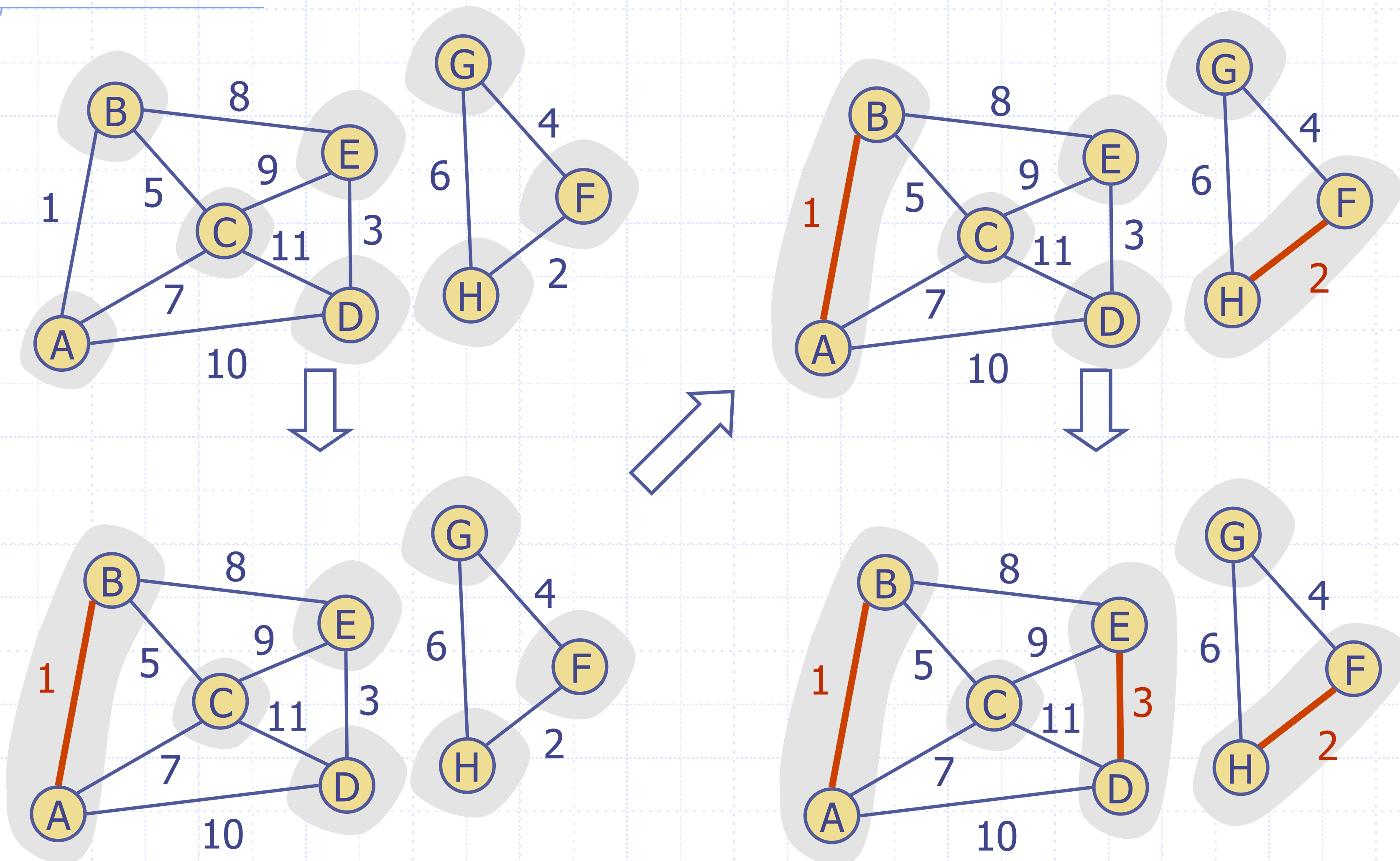
- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge "closest" clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster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

Kruskal's Algorithm

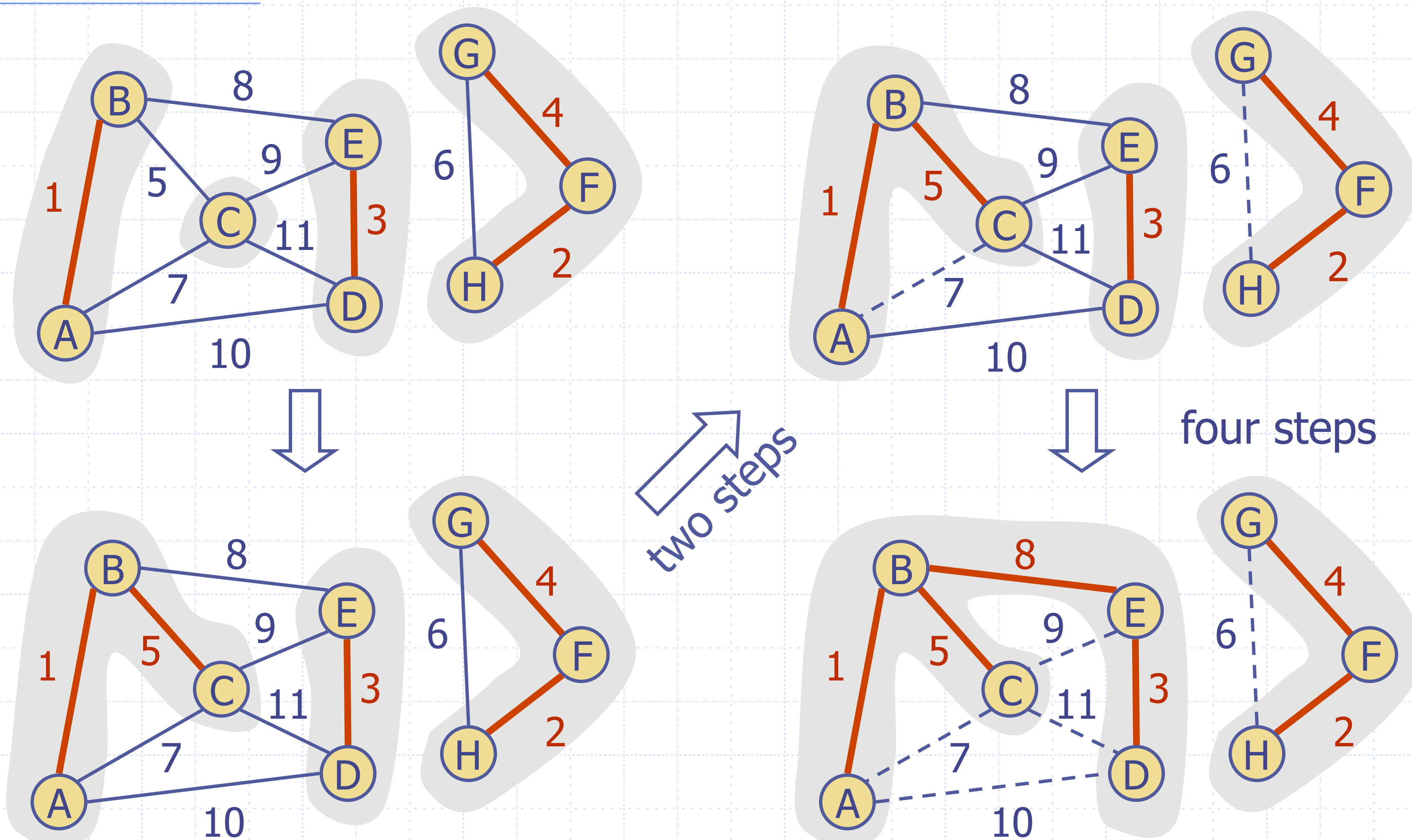
Algorithm Kruskal(G):
   Input: A simple connected weighted graph G with n vertices and m edges
   Output: A minimum spanning tree T for G

```
for each vertex v in G do
    Define an elementary cluster C(v) = {v}.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
T = ∅                    {T will ultimately contain the edges of the MST}
while T has fewer than n − 1 edges do
    (u,v) = value returned by Q.remove_min()
    Let C(u) be the cluster containing u, and let C(v) be the cluster containing v.
    if C(u) ≠ C(v) then
        Add edge (u,v) to T.
        Merge C(u) and C(v) into one cluster.
return tree T
```

Example



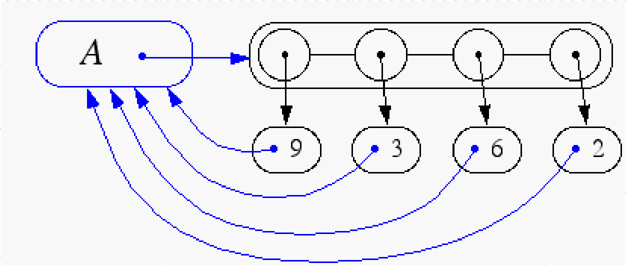
Example (contd.)



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: $O(n \log n)$

Java Implementation

```
/** Computes a minimum spanning tree of graph g using Kruskal's algorithm. */
public static <V> PositionalList<Edge<Integer>> MST(Graph<V,Integer> g) {
  // tree is where we will store result as it is computed
  PositionalList<Edge<Integer>> tree = new LinkedPositionalList<>();
  // pq entries are edges of graph, with weights as keys
  PriorityQueue<Integer, Edge<Integer>> pq = new HeapPriorityQueue<>();
  // union-find forest of components of the graph
  Partition<Vertex<V>> forest = new Partition<>();
  // map each vertex to the forest position
  Map<Vertex<V>,Position<Vertex<V>>> positions = new ProbeHashMap<>();

  for (Vertex<V> v : g.vertices())
    positions.put(v, forest.makeGroup(v));

  for (Edge<Integer> e : g.edges())
    pq.insert(e.getElement(), e);

```

Java Implementation, 2

```
int size = g.numVertices( );
// while tree not spanning and unprocessed edges remain...
while (tree.size( ) != size − 1 && !pq.isEmpty( )) {
  Entry<Integer, Edge<Integer>> entry = pq.removeMin( );
  Edge<Integer> edge = entry.getValue( );
  Vertex<V>[ ] endpoints = g.endVertices(edge);
  Position<Vertex<V>> a = forest.find(positions.get(endpoints[0]));
  Position<Vertex<V>> b = forest.find(positions.get(endpoints[1]));
  if (a != b) {
    tree.addLast(edge);
    forest.union(a,b);
  }
}

return tree;
}
```

Baruvka's Algorithm (Exercise)

- 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 ← 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.