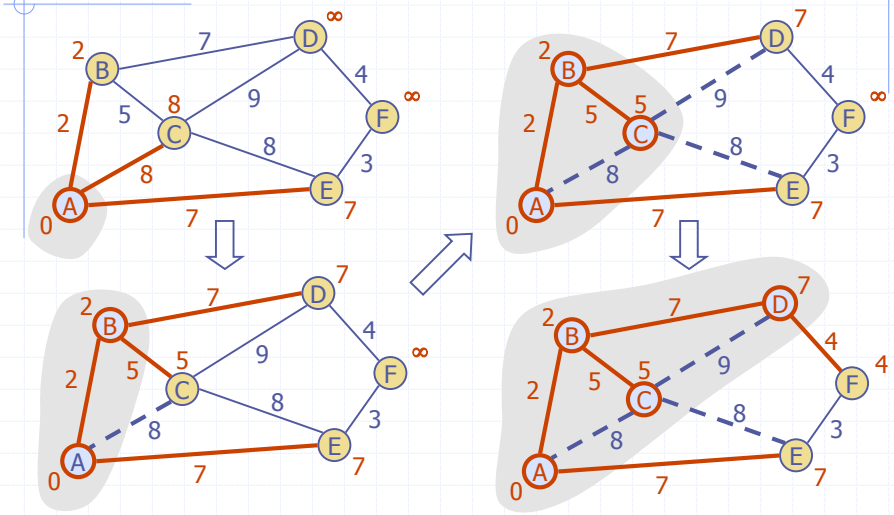
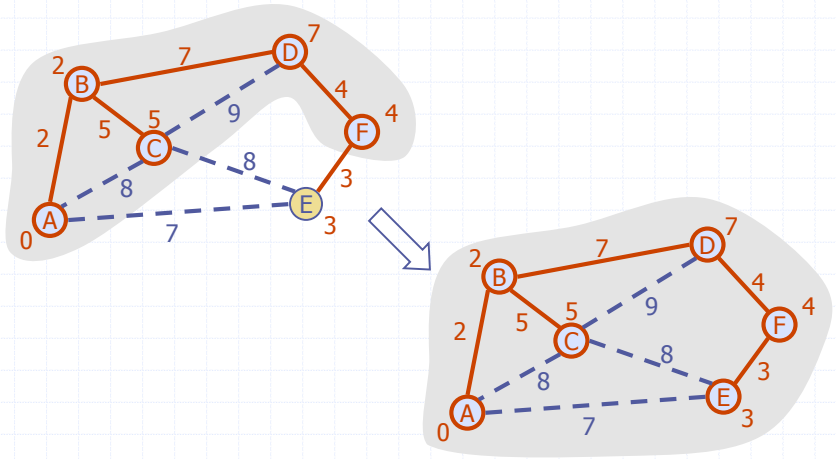
Example



Example (contd.)



Dijkstra vs. Prim-Jarnik

```
Algorithm DijkstraShortestPaths(G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(getLocator(z),r)
```

```
Algorithm PrimJarnikMST(G)
  Q ← new heap-based priority queue
  s ← a vertex of G
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
      z ← G.opposite(u,e)
      r ← weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(getLocator(z),r)
```

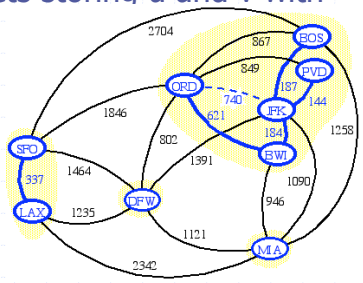

Kruskal's Algorithm (§ 12.7.1)

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree T which is our MST

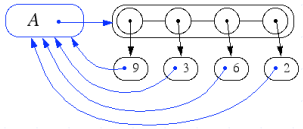
```
Algorithm KruskalMST(G)
  for each vertex V in G do
    define a Cloud(v) of ← {v}
  let Q be a priority queue.
  Insert all edges into Q using their
weights as the key
  T ← ∅
  while T has fewer than n-1 edges do
    edge e = T.removeMin()
    Let u, v be the endpoints of e
    if Cloud(v) ≠ Cloud(u) then
      Add edge e to T
      Merge Cloud(v) and Cloud(u)
  return T
```

Data Structure for Kruskal Algortihm (§ 10.6.2)

- The algorithm maintains a forest of trees
- 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 the operations:
  - **find**(u): return the set storing u
  - **union**(u,v): replace the sets storing u and v with their union



Representation of a 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(u,v), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation union(u,v) is $\min(n_u,n_v)$, where n_u and n_v are the sizes of the sets storing u and v
- 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

- A partition-based version of Kruskal's Algorithm performs cloud merges as unions and tests as finds.

```
Algorithm Kruskal(G):
  Input: A weighted graph G.
  Output: An MST T for G.
Let P be a partition of the vertices of G, where each vertex forms a separate set.
Let Q be a priority queue storing the edges of G, sorted by their weights
Let T be an initially-empty tree
while Q is not empty do
  (u,v) ← Q.removeMinElement()
  if P.find(u) != P.find(v) then
      Add (u,v) to T
      P.union(u,v)
return T
```

Running time: O((n+m)log n)