Minimum Spanning Trees

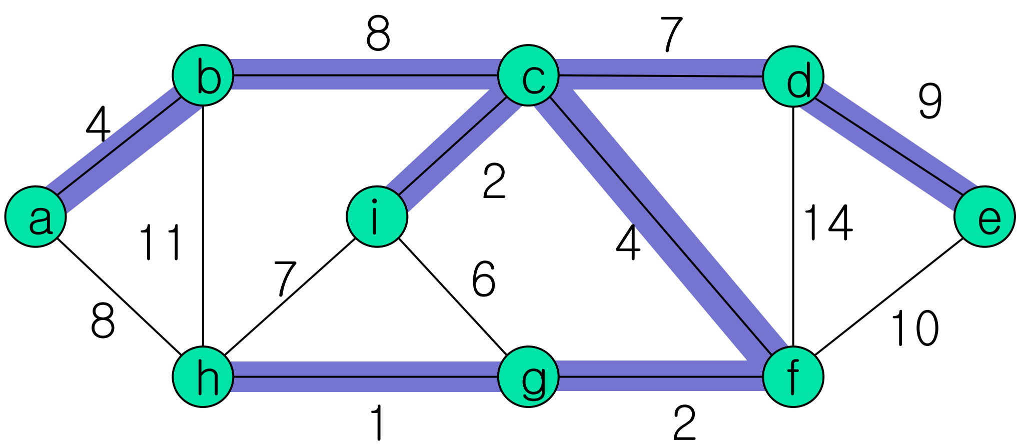
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Minimum Spanning Tree

- In the design of electronic circuitry, it is often necessary to make the pins of several components electrically equivalent by wiring together.
- To interconnect a set of n pins, we can use an arrangement of n-1 pins.
- Of all such arrangements, the one using the least amount of wire is the most desirable.

Minimum Spanning Tree

- Given a connected, undirected, weighted graph
- Find a spanning tree using edges that minimizes the total weight



Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Minimum Spanning Tree

- We can model this wiring problem with a connected, undirected graph $G=(V,E)$, where
  - V is the set of pins
  - E is the set of possible interconnections between pair of pins
  - A weight $w(u,v)$ for each edge $(u,v)\in E$ that specifying the cost to connect u and v
- Find an acyclic subset $T\subseteq E$ that connects all of the vertices and whose total weight
  $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized.
- Since T is acyclic and connects all the vertices, we call a minimum spanning tree.

Minimum Spanning Tree

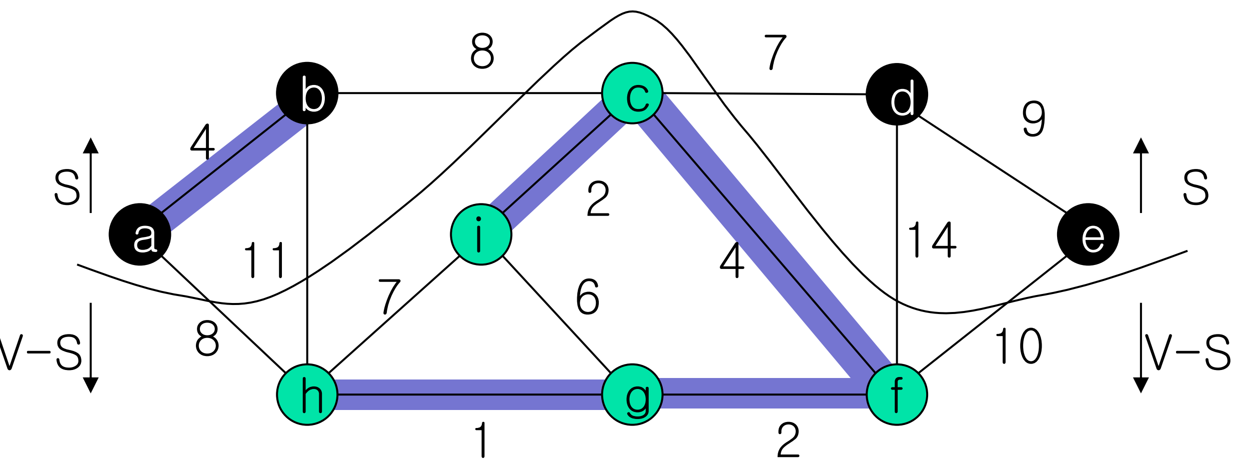
GENERIC-MST(G, w)

1. A = Ø
2. **while** A does not form a spanning tree
3.   find an edge (u,v) that is safe for A
4.   A = A U {(u,v)}
5. **return** A

Minimum Spanning Tree

- If A U {(u,v)} is also a subset of a minimum spanning tree, we call the edge (u,v) a safe edge.
- A cut (S, V-S) of an undirected graph G=(V,E) is a partition of V.
- An edge (u,v)∈E crosses the cut (S, V-S) if one of its endpoints is in S and the other is in V-S.
- A cut respects a set A of edges if no edge in A crosses the cut.
- An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

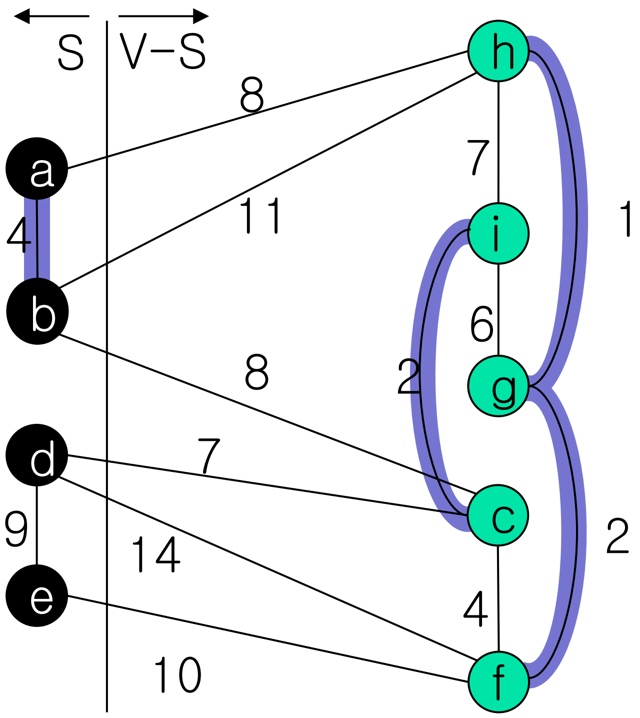
One Way of Viewing a Cut (S, V-S)



- Black vertices are in the set S, and green vertices are in V-S.
- The edge (d,c) is the unique light edge crossing the cut.

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Another Way of Viewing a Cut (S, V-S)



Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

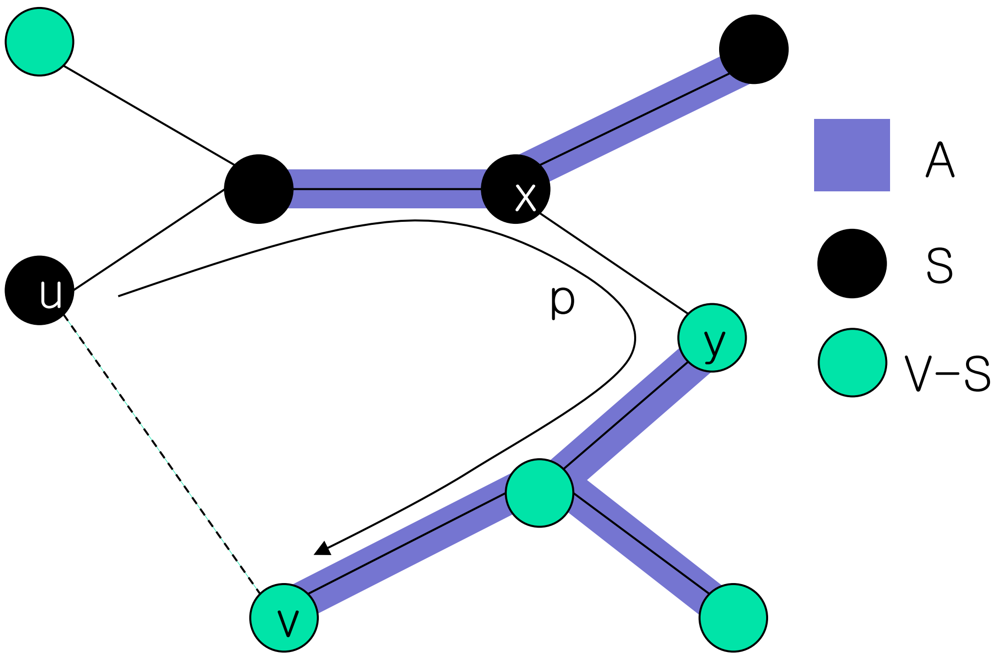
Theorem 23.1

- Let $G=(V,E)$ be a connected undirected graph with a real-valued weight function w defined on E.
- Let A be a subset of E that is included in some minimum spanning tree for G.
- Let $(S,V-S)$ be any cut of G that respects A and let (u,v) be a light edge crossing $(S,V-S)$.
- Then, the edge (u,v) is safe for A

Theorem 23.1 (Proof)

- Let T be a minimum spanning tree that includes A
- Assume that T does not contain the light edge (u,v)
- Construct another minimum spanning tree T' that include A U {(u,v)} by using a cut-and-paste technique, thereby showing that (u,v) is a safe edge for A

Theorem 23.1 (Proof)



- The edge (x, y) is an edge on the unique simple path p from u to v in T and the edges in A are shaded.

Theorem 23.1 (Proof)

- The light edge (u,v) forms a cycle with the edge on the simple path p from u to v in T.
- Since u and v are on opposite sides of the cut (S, V-S), there is at least one edge in T on the simple path p that also crosses the cut. Let (x,y) be any such edge.
- The edge (x, y) is not in A, since the cut respects A.
- Since (x,y) is on the unique path from u to v in T, removing (x,y) breaks T into two components.
- Adding (u,v) re-connects them to form a new spanning tree T' = T – {(x,y)} U{(u,v)}.

Theorem 23.1 (Proof)

- We next show that $T' = T - \{(x,y)\} \cup \{(u,v)\}$ is a minimum spanning tree.
- Since (u, u) is a light edge crossing (S, V-S) and (x, y) also crosses this cut, we have $w(u,v) \leq w(x, y)$ resulting that $w(T') = w(T) - w(x,y) + w(u,v) \leq w(T)$.
- But $w(T) \leq w(T')$, since T is a minimum spanning tree.
- Thus, T' must be a minimum spanning tree.
- Because $A \subseteq T'$ and $A \cup \{(u,v)\} \subseteq T'$ where T' is a minimum spanning tree, (u,v) is safe for A.

Understanding of GENERIC-MST

- As the method proceeds, the set A is always acyclic; otherwise, a minimum spanning tree including A would contain a cycle, which is a contradiction.
- At any point in the execution, the graph $G_A=(V, A)$ is a forest, and each of the connected components of G_A is a tree.
- Some of the trees may contain just one vertex, as is the case, for example, when the method begins: A is empty and the forest contains |V| trees, one for each vertex.
- Moreover, any safe edge (u,v) for A connects distinct components of G_A, since $A \cup \{(u,v)\}$ must be acyclic.

Understanding of GENERIC-MST

- The **while** loop in lines 2 - 4 of GENERIC-MST executes |V| - 1 times because it finds one of the |V|-1 edges of a minimum spanning tree in each iteration.
- Initially, when A = Ø, there are |V| trees in G_A, and each iteration reduces that number by 1.
- When the forest contains only a single tree, the method terminates.

Corollary 23.2

- Let $G=(V,E)$ be a connected, undirected graph with a real-valued weight function w defined on E.
- Let A be a subset of E that is included in some minimum spanning tree for G.
- Let $C=(V_C,E_C)$ be a connected component (tree) in the forest $G_A=(V,A)$.
- If (u,v) is a light edge connecting C to some other component in G_A, then (u,v) is safe for A

Corollary 23.2 (Proof)

- The cut (V_C, $V-V_C$) respects A and (u,v) is a light edge for this cut.
- Thus, (u,v) is safe for A.

Kruskal's and Prim's Algorithms

- They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST.
- In Kruskal's algorithm,
  - The set A is a forest whose vertices are all those of the given graph.
  - The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.
- In Prim's algorithm,
  - The set A forms a single tree.
  - The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

Kruskal's Algorithm

- A greedy algorithm since **at each step it adds to the forest an edge of least possible weight.**
- Find a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight.
- Let two trees C_1 and C_2 are connected by (u,v).
- Since (u,v) must be a light edge connecting C_1 to some other tree, Corollary 23.2 implies that (u,v) is a safe edge for C_1.

Implementation of Kruskal's Algorithm

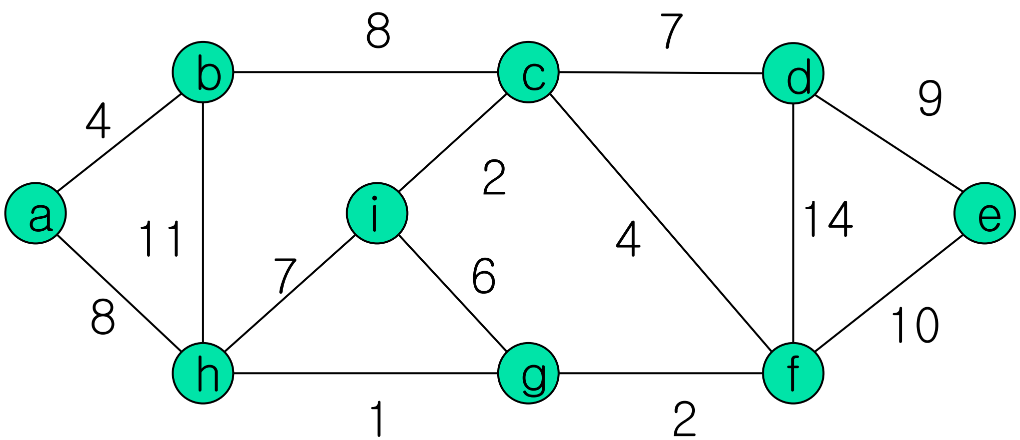
- It uses a disjoint-set data structure to maintain several disjoint sets of elements.
- Each set contains the vertices in one tree of the current forest.
- The operation FIND-SET(u) returns a representative element from the set that contains u.
- Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v).
- To combine trees, Kruskal's algorithm calls the UNION procedure.

Kruskal's Algorithm

```
MST-KRUSKAL(G,w)
    A = ∅
    for each v ∈ G.V
        Make-Set(v)
    sort the edges of G.E into nondecreasing order
    by weight w
    for each edge (u,v) ∈ G.E in sorted order
        if Find-Set(u) ≠ Find-Set(v)
                A = A U {{u,v}}
                Union(u,v)
    return A
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

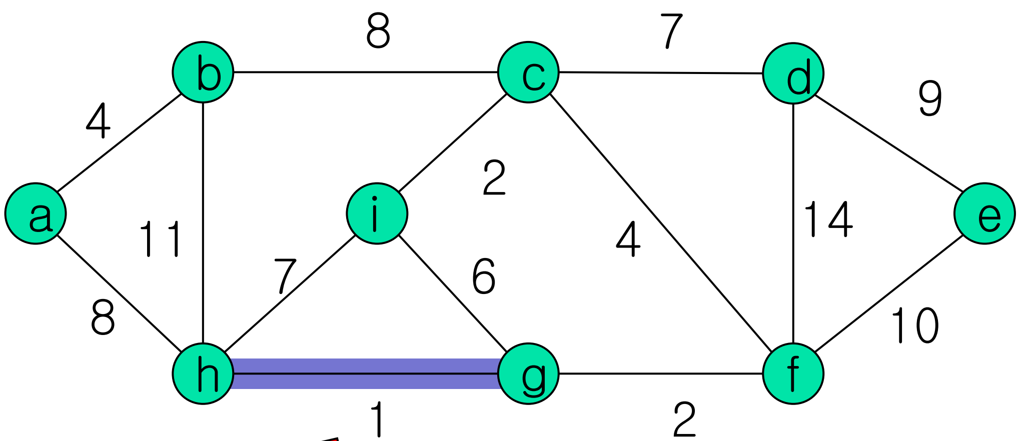
Kruskal’s Algorithm




```
5.      for each edge (u,v) ∈ G.E in sorted order
6.          if Find-Set(u) ≠ Find-Set(v)
7.              A = A U {{u,v}}
8.              Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

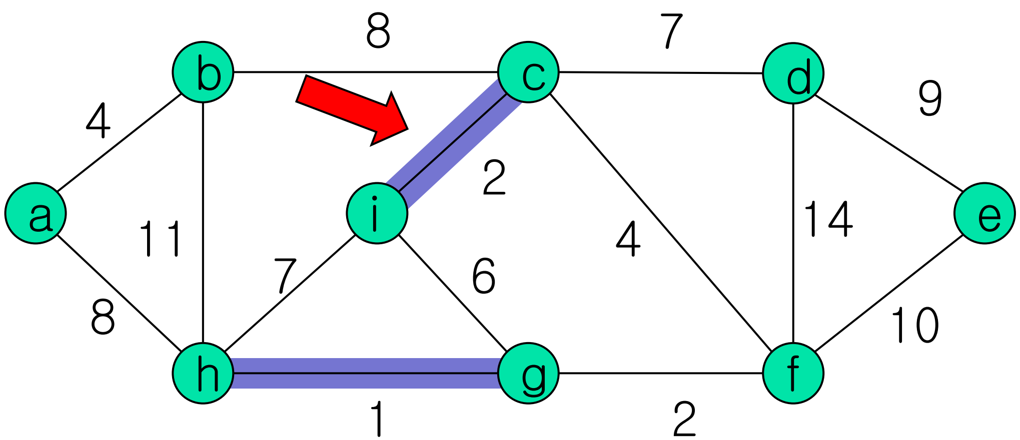
Kruskal’s Algorithm




```
for each edge (u,v) ∈ G.E in sorted order
    if Find-Set(u) ≠ Find-Set(v)
        A = A U {{u,v}}
        Union(u,v)
```

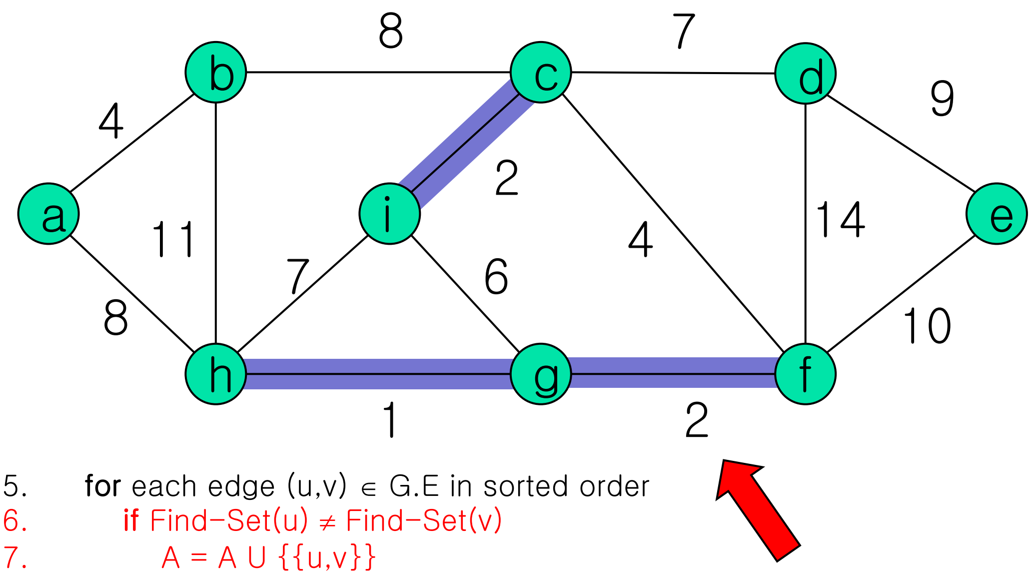
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm



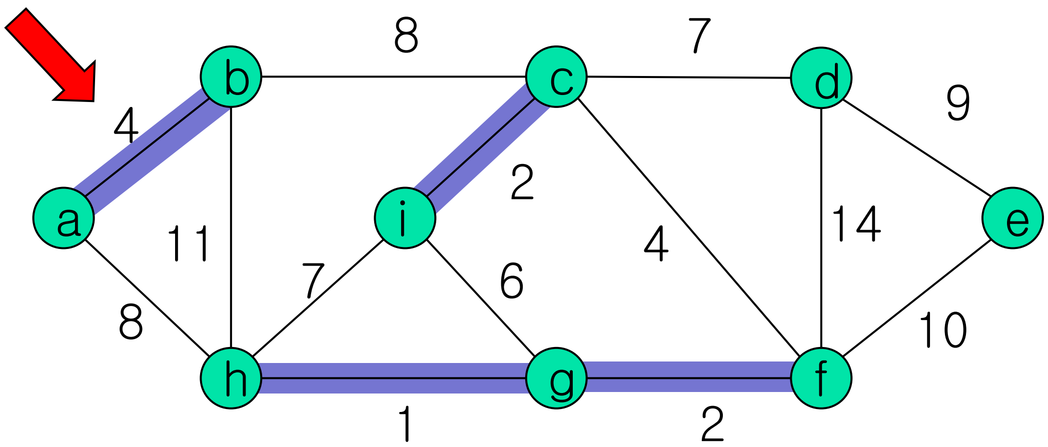
5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Kruskal's Algorithm



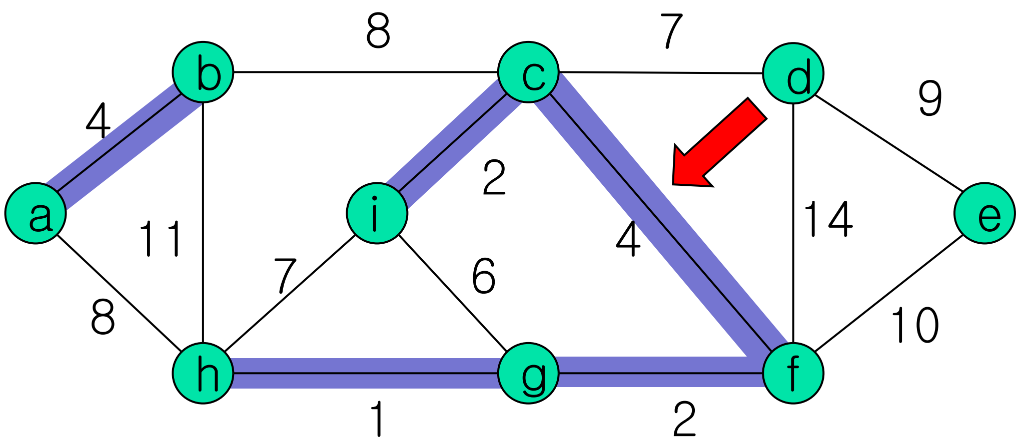
5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Kruskal's Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

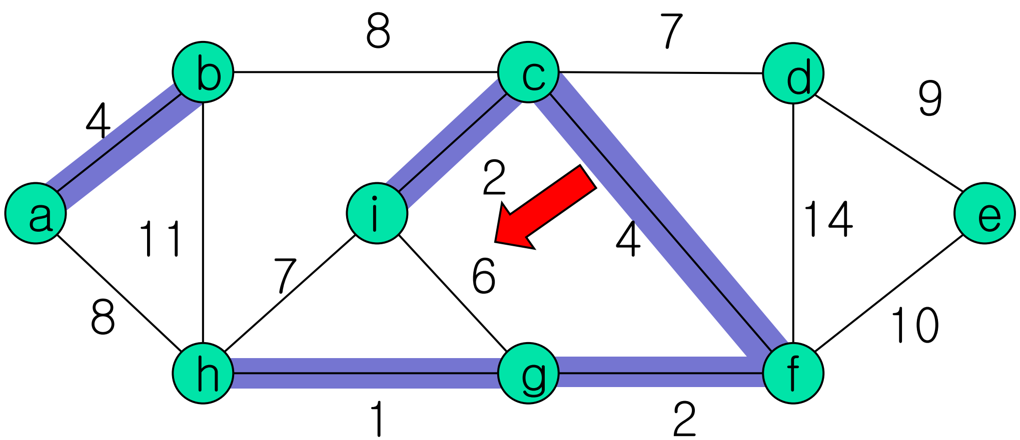
Kruskal's Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

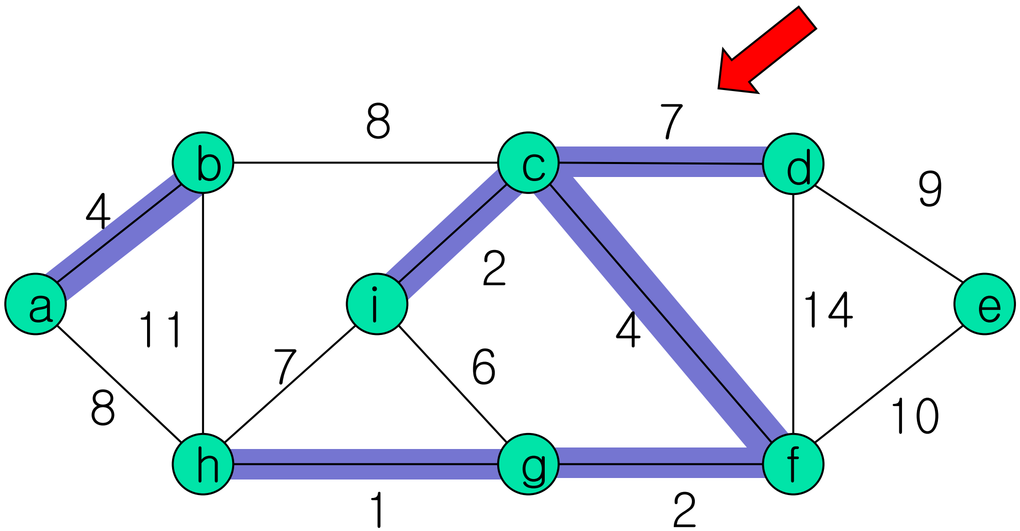
Kruskal's Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

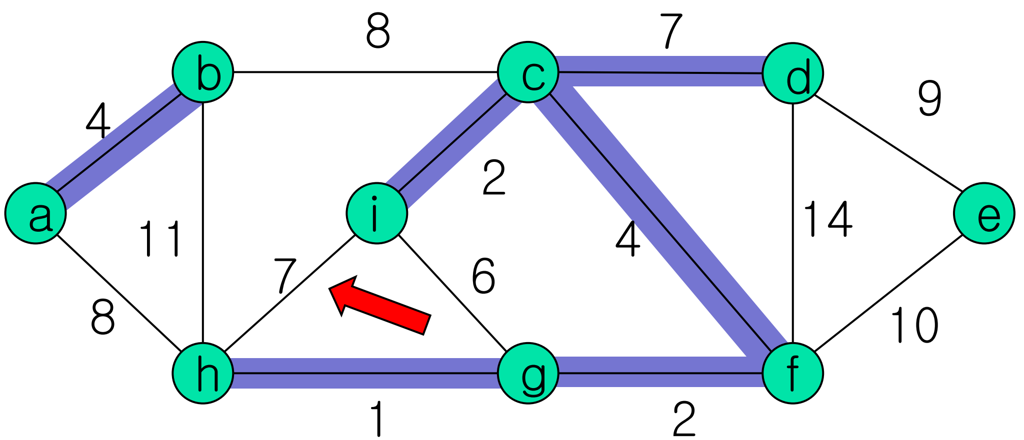
Kruskal's Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

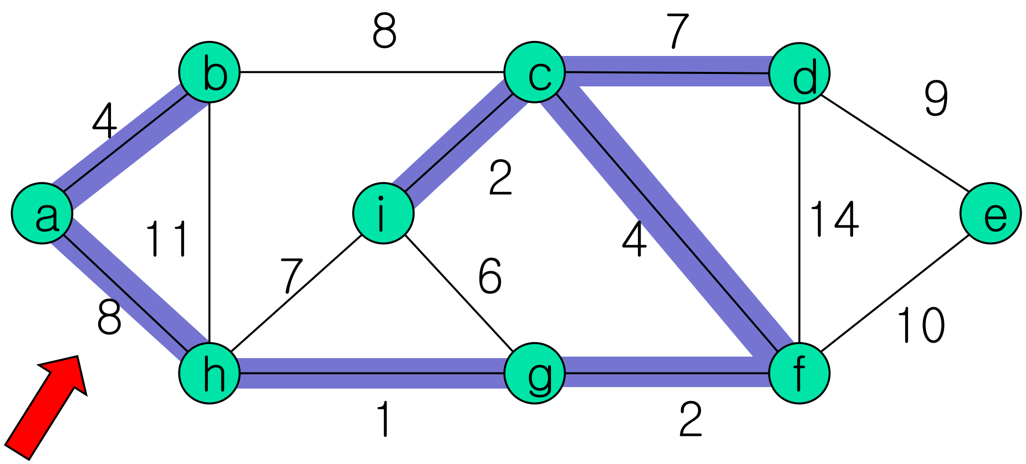
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

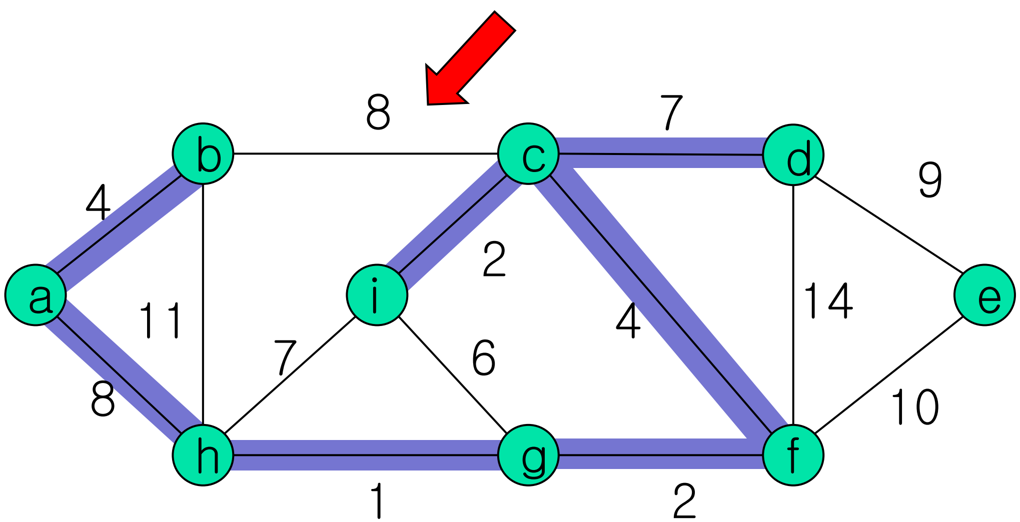
Kruskal's Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

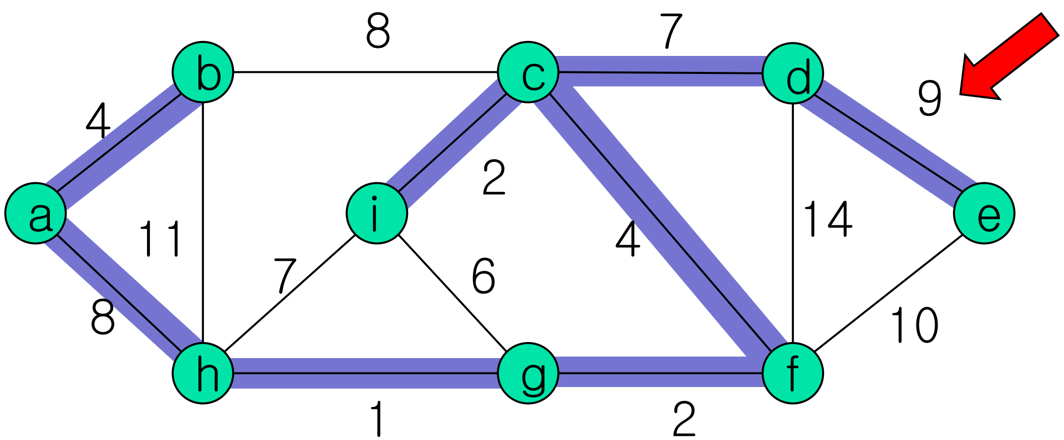
Kruskal's Algorithm



5. **for** each edge (u,v) ∊ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

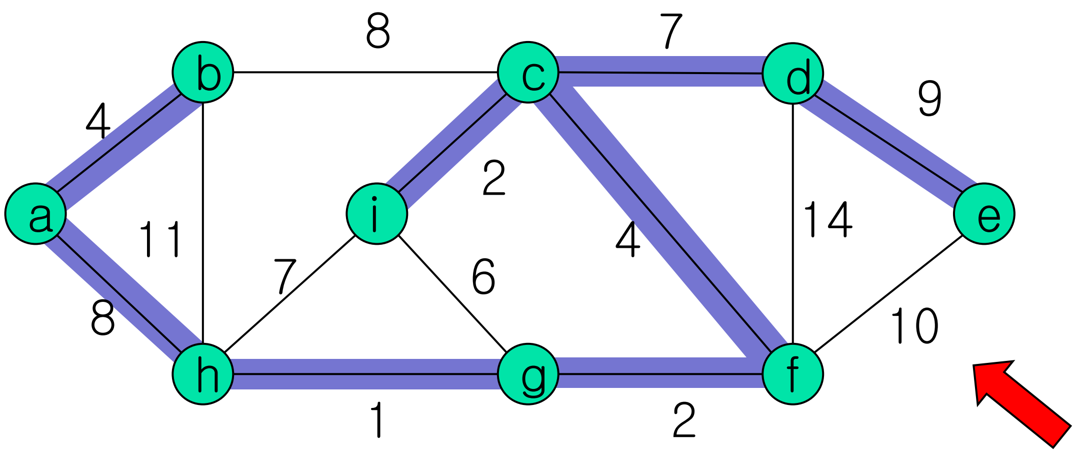
Kruskal's Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

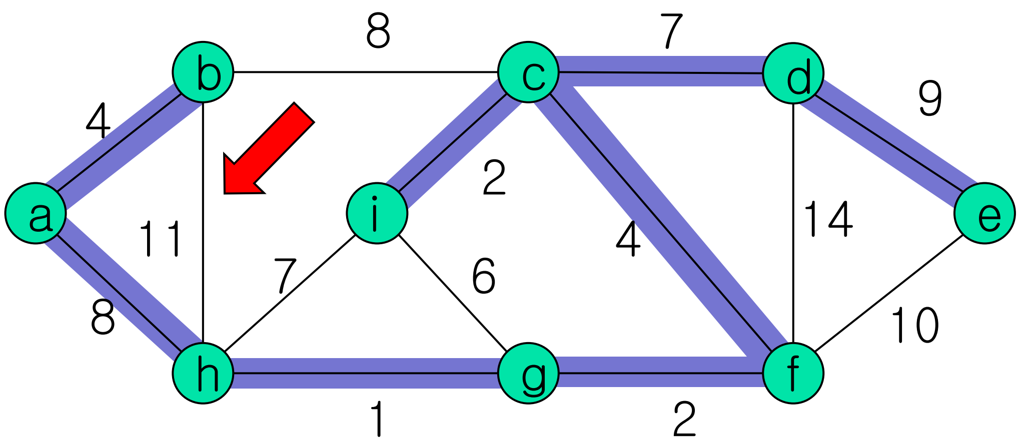
Kruskal's Algorithm




```
for each edge (u,v) ∈ G.E in sorted order
    if Find-Set(u) ≠ Find-Set(v)
        A = A U {{u,v}}
        Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

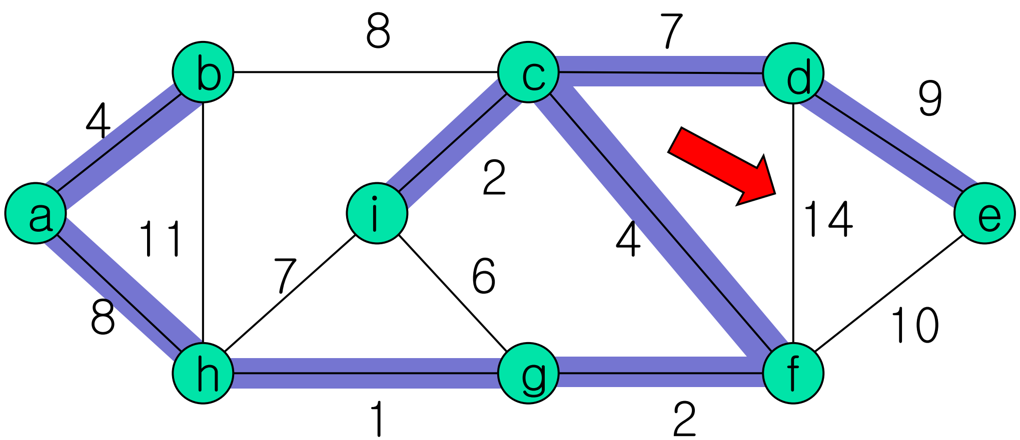
Kruskal’s Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

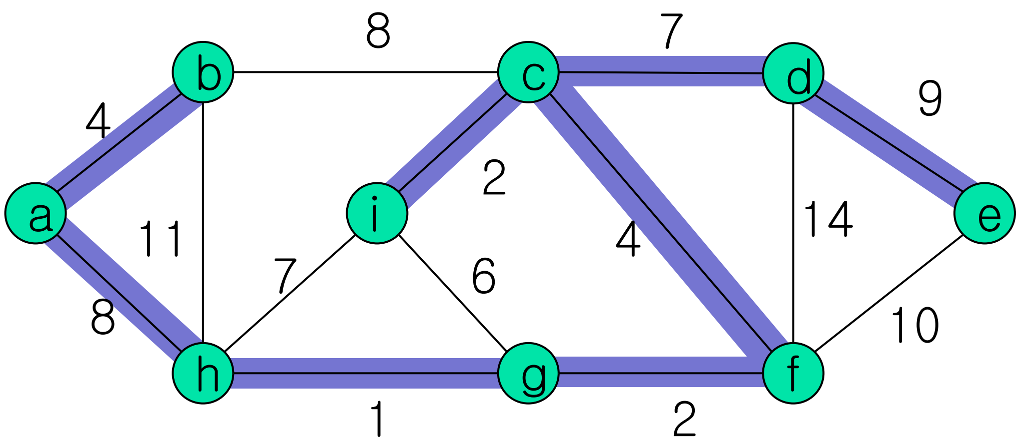
Kruskal's Algorithm



5. **for** each edge (u,v) ∈ G.E in sorted order
6. **if** Find-Set(u) ≠ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm




```
for each edge (u,v) ∈ G.E in sorted order
    if Find-Set(u) ≠ Find-Set(v)
        A = A U {{u,v}}
        Union(u,v)
return A
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal’s Algorithm

```
MST-KRUSKAL(G,w)
    A = ∅
    for each v ∈ G.V
        Make-Set(v)                                   O(V) Make-Set() calls
    sort the edges of G.E into nondecreasing order
    by weight w                                       O(E lg E)
    for each edge (u,v) ∈ G.E in sorted order
        if Find-Set(u) ≠ Find-Set(v)                  O(E) Find-Set() calls
                A = A U {{u,v}}
                Union(u,v)                            O(V) Union() calls
    return A
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Running Time of Kruskal's Algorithm

- Use the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics (Section 21.3).
- Sorting the edges in line 4 is $O(|E| \lg |E|)$.
- The disjoint-set operations takes $O((|V|+|E|)\ \alpha(|V|))$ time, where α is the very slowly growing function (Section 21.4).
  - The **for** loop (lines 2–3) performs $|V|$ MAKE-SET operations.
  - The **for** loop (lines 5–8) performs $O(|E|)$ FIND-SET and UNION operations.
- Since G is connected, we have $|E| \geq |V|-1$, the disjoint-set operations take $O(|E|\alpha(|V|))$ time.
- Moreover, since $\alpha(|V|) = O(\lg |V|) = O(\lg |E|)$, Kruskal's algorithm takes $O(|E| \lg |E|)$ time.
- Observing that $|E| < |V|^2 \Rightarrow \lg |E| = O(\lg V)$, the total running time of Kruskal's algorithm becomes $O(E \lg V)$.

Running Time of Kruskal's Algorithm

O(|V|) Make−Set() calls
O(|E|) Find−Set() calls
O(|V|) Union() calls

- In a nut shell,
  - Sort edges: $O(|E| \lg |E|)$
  - Disjoint-set operations
    - $O(|V|+|E|)$ operations $\Rightarrow O((|V|+|E|)\ \alpha(|V|))$ time
    - $|E| \geq |V|-1 \Rightarrow O(E|\ \alpha(|V|))$ time
    - Since $\alpha(n)$ can be upper bounded by the height of the tree,
      $\alpha(|V|)=O(\lg |V|)=O(\lg |E|)$.
- Thus, the total running time of Kruskal's algorithm is is $O(|E| \lg |E|)$
- By observing that $|E| < |V|^2 \Rightarrow \lg |E| = O(\lg |V|)$, it becomes $O(|E| \lg |V|)$.

Prim’s Algorithm

- Special case of the generic minimum-spanning-tree method.
- A greedy algorithm since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree’s weight.
- The edges in the set A always form a single tree.
- Each step adds to the tree A a light edge that connects A to an isolated vertex – one on which no edge of A is incident.
- By Corollary 23.2, this rule adds only edges that are safe for A

Implementation of Prim's algorithm

- Input is a connected Graph G and the root r of the minimum spanning tree.
- During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.
- For each v, v.key is the minimum weight of any edge connecting v to a vertex in the tree.
- By convention, v.key = ∞ if there is no such edge.
- The attribute v.π names the parent of v in the tree.
- The algorithm maintains the set A from Generic-MST as $A=\{(v, v.\pi):v\in V-\{r\}-Q\}$.
- It terminates when the min-priority queue Q is empty.

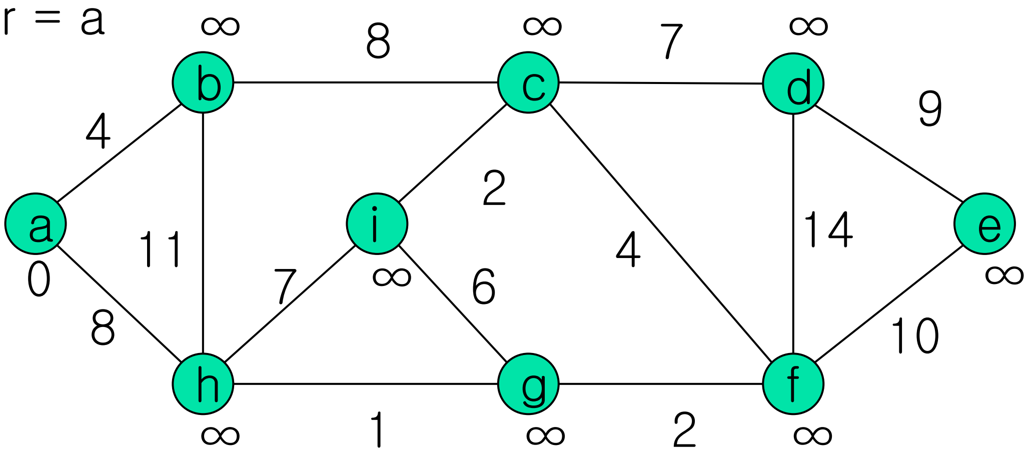
Prim's Algorithm

```
MST-PRIM(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π=NIL
    r.key = 0
    Q = G.V
    while Q ≠ Ø
        u = Extract-Min(Q)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u,v) < v.key
                v.π = u
                v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Prim's Algorithm

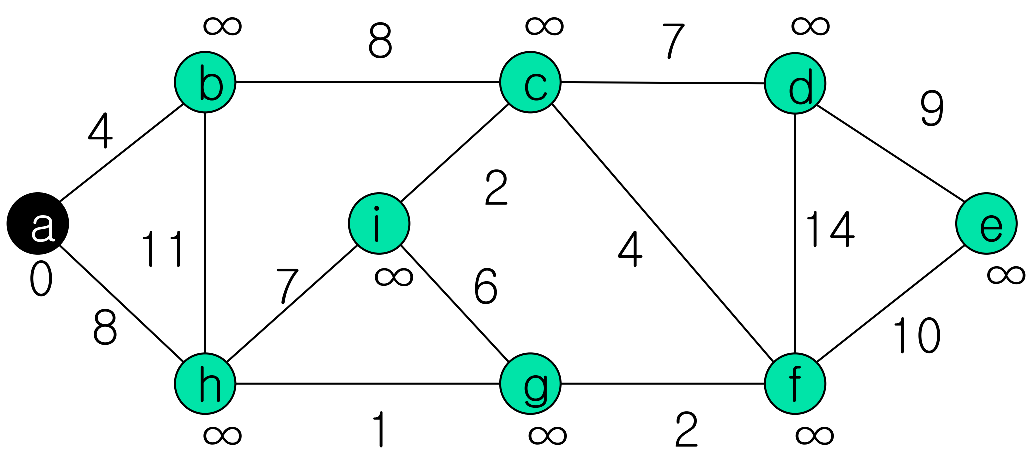
r = a



1. **for** each u ∈ G.V
2. u.key = ∞
3. u.π=NIL
4. r.key = 0
5. Q = G.V

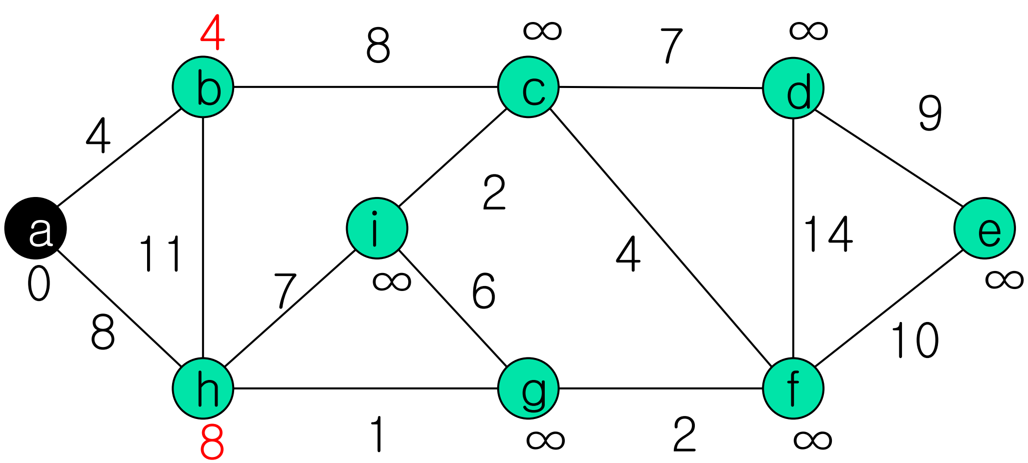
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Prim's Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

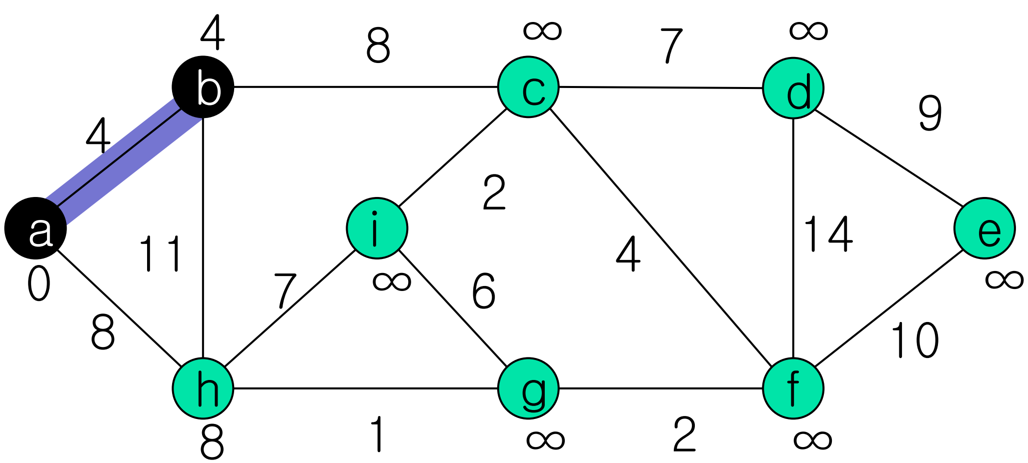
Prim's Algorithm




```
7.        u = Extract-Min(Q)
8.           for each v ∈ G.Adj[u]
9.              if v ∈ Q and w(u,v) < v.key
10.                v.π = u
11.                v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

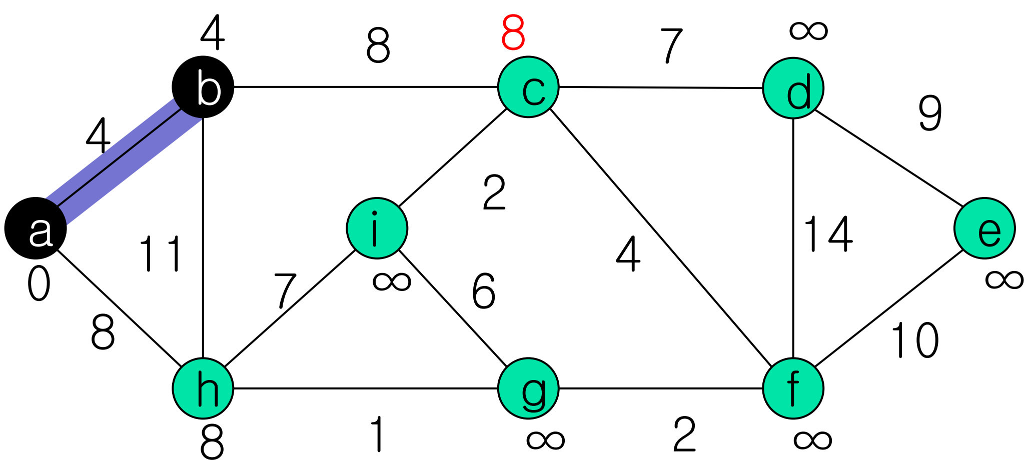
Prim's Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

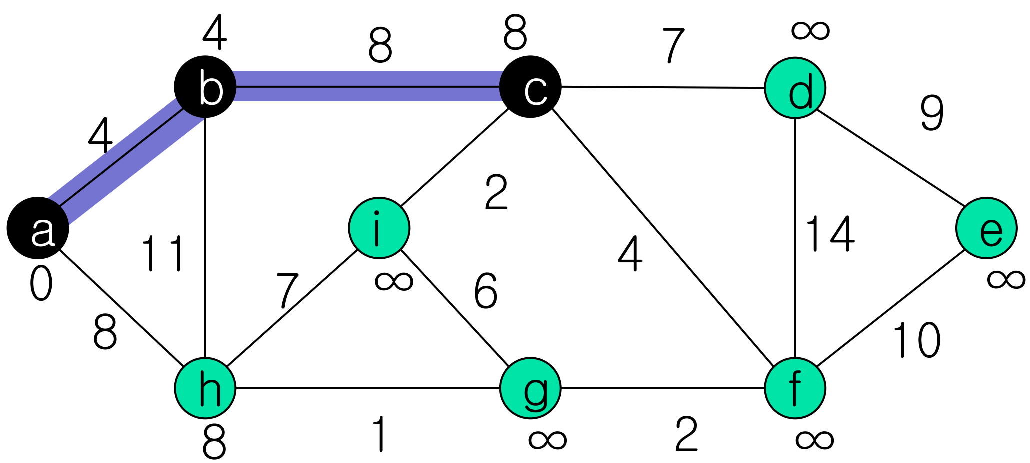
Prim's Algorithm




```
u = Extract-Min(Q)
   for each v ∈ G.Adj[u]
      if v ∈ Q and w(u,v) < v.key
         v.π = u
         v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

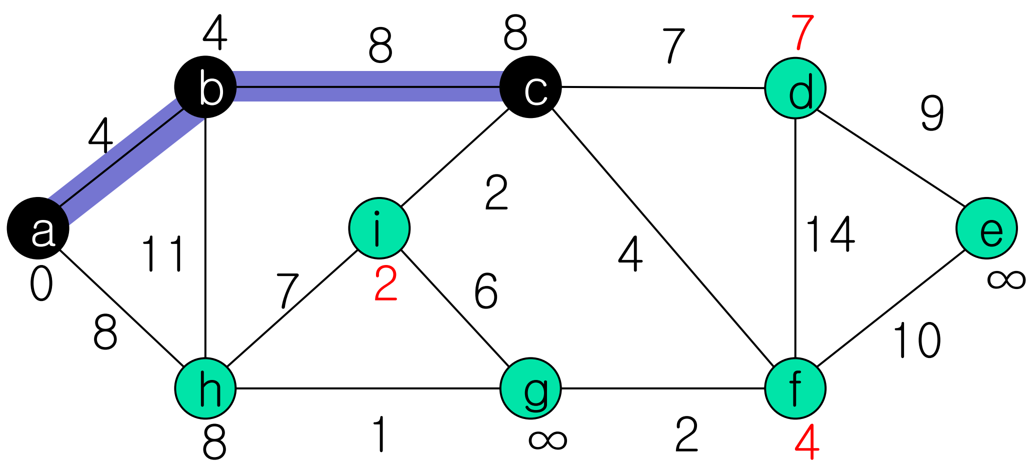
Prim’s Algorithm



7. u = Extract-Min(Q)
8. **for** each v ∈ G.Adj[u]
9. **if** v ∈ Q and w(u,v) < v.key
10. v.π = u
11. v.key = w(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

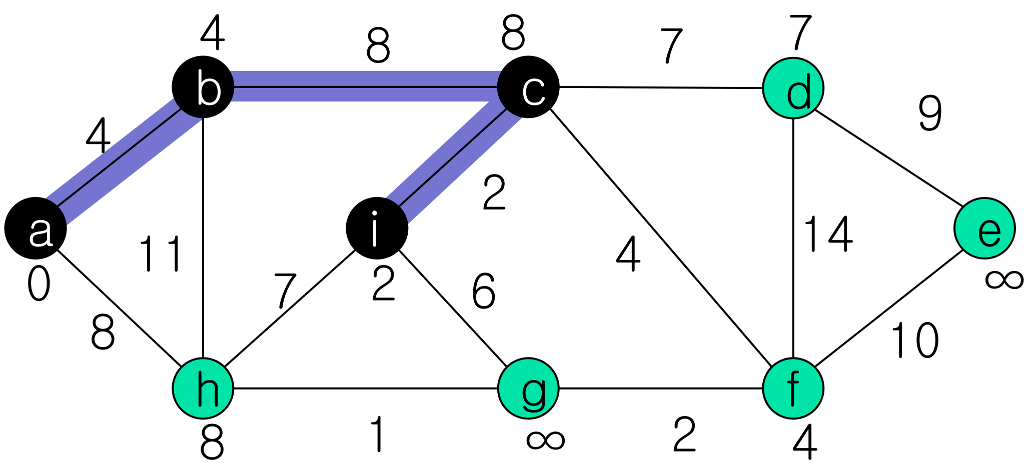
Prim's Algorithm



7. u = Extract-Min(Q)
8. **for** each v ∈ G.Adj[u]
9. **if** v ∈ Q and w(u,v) < v.key
10. v.π = u
11. v.key = w(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

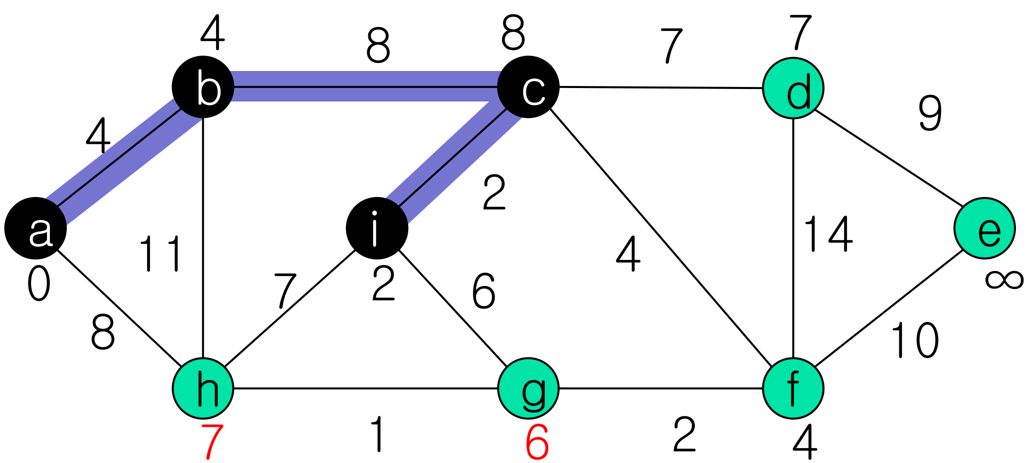
Prim's Algorithm



7. u = Extract-Min(Q)
8. **for** each v ∈ G.Adj[u]
9. **if** v ∈ Q and w(u,v) < v.key
10. v.π = u
11. v.key = w(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

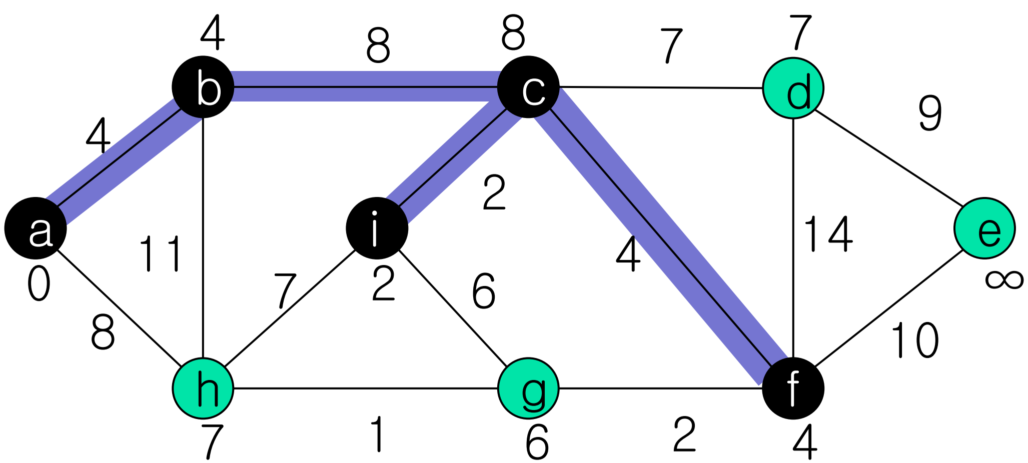
Prim’s Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

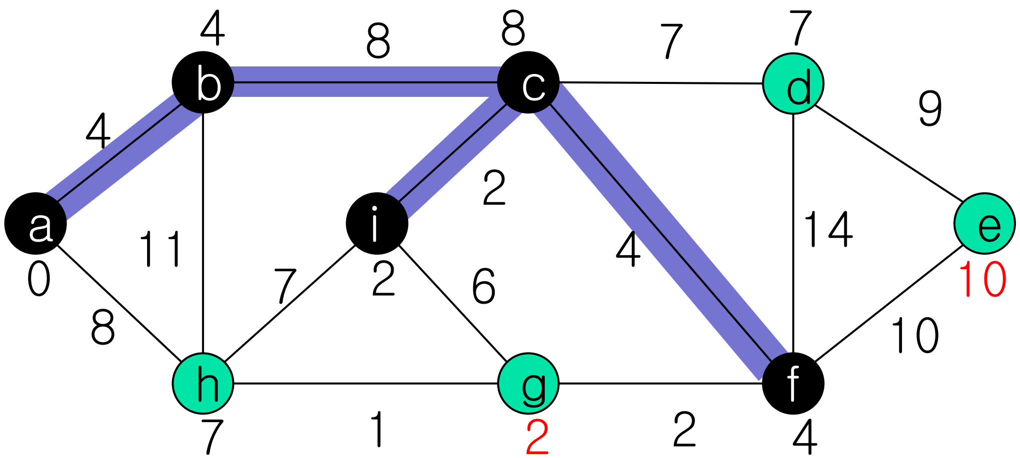
Prim’s Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

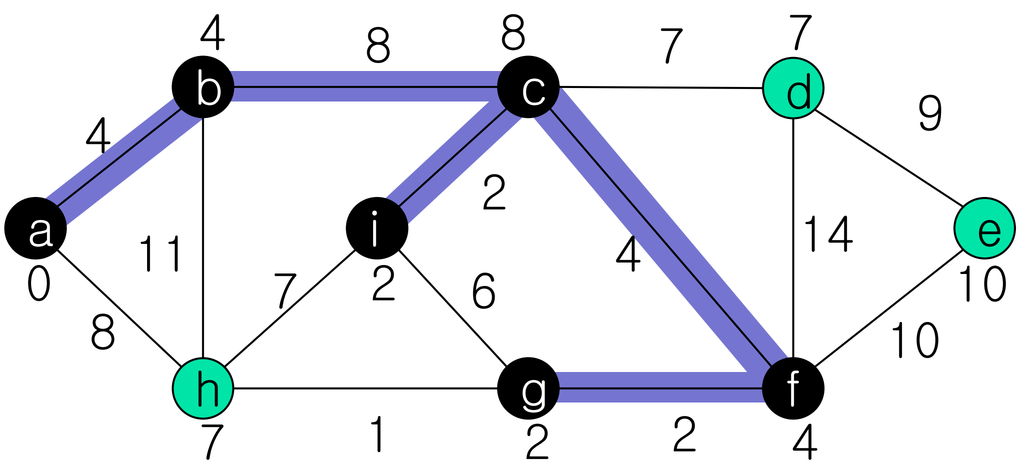
Prim's Algorithm




```
u = Extract-Min(Q)
   for each v ∈ G.Adj[u]
      if v ∈ Q and w(u,v) < v.key
         v.π = u
         v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

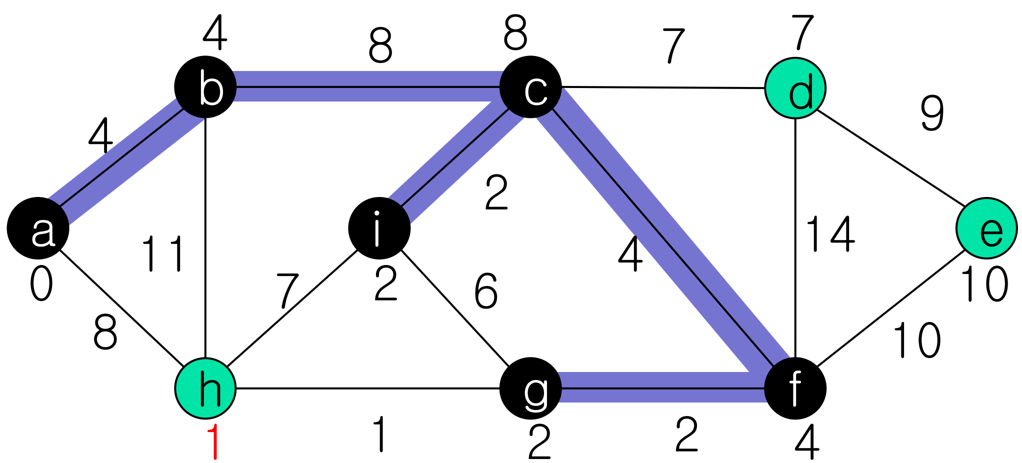
Prim’s Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

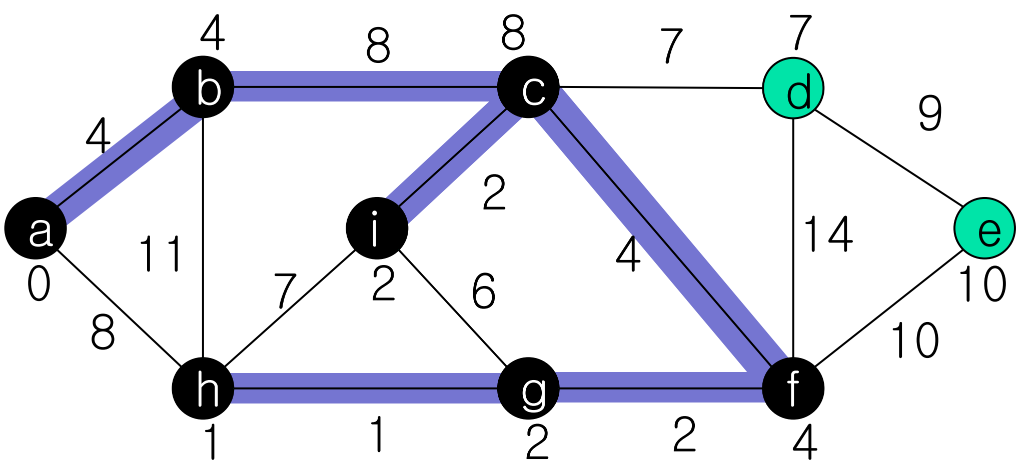
Prim's Algorithm




```
7.       u = Extract-Min(Q)
8.          for each v ∈ G.Adj[u]
9.             if v ∈ Q and w(u,v) < v.key
10.               v.π = u
11.               v.key = w(u,v)
```

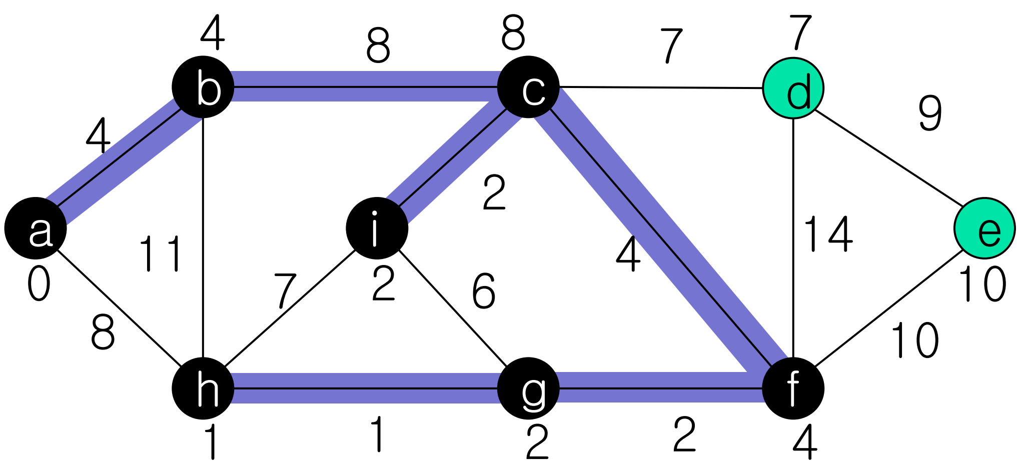
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Prim’s Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

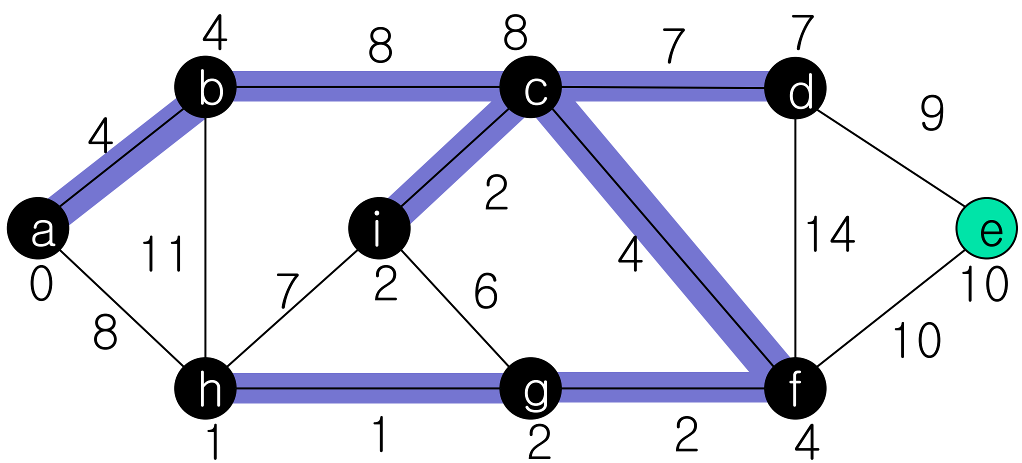
Prim’s Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

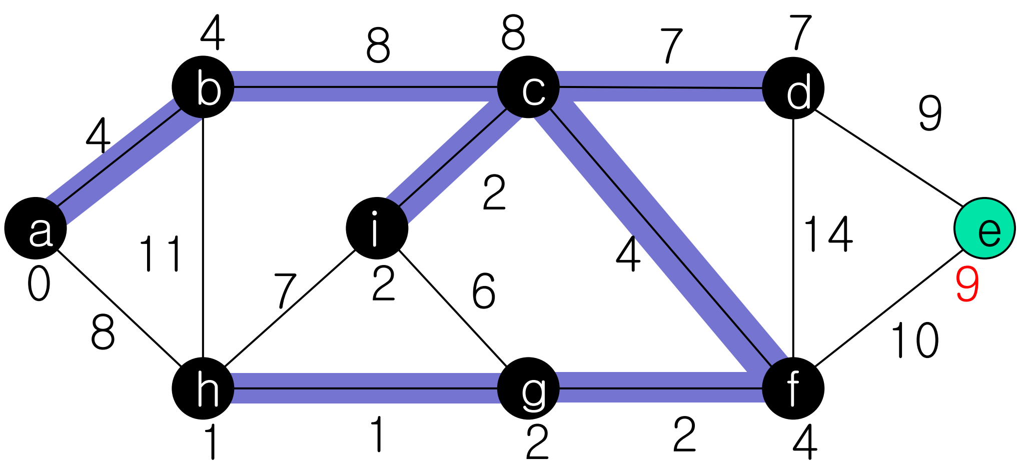
Prim's Algorithm



7. u = Extract-Min(Q)
8. **for** each v ∈ G.Adj[u]
9. **if** v ∈ Q and w(u,v) < v.key
10. v.π = u
11. v.key = w(u,v)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

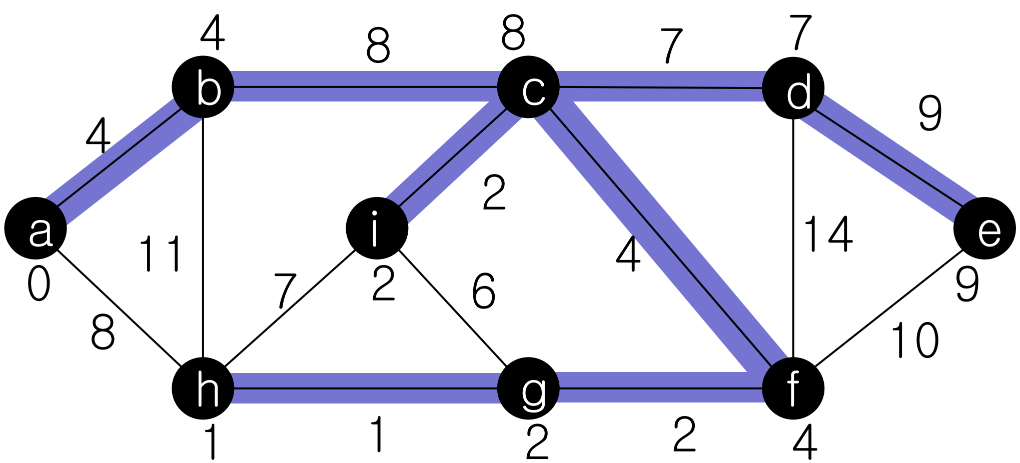
Prim’s Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

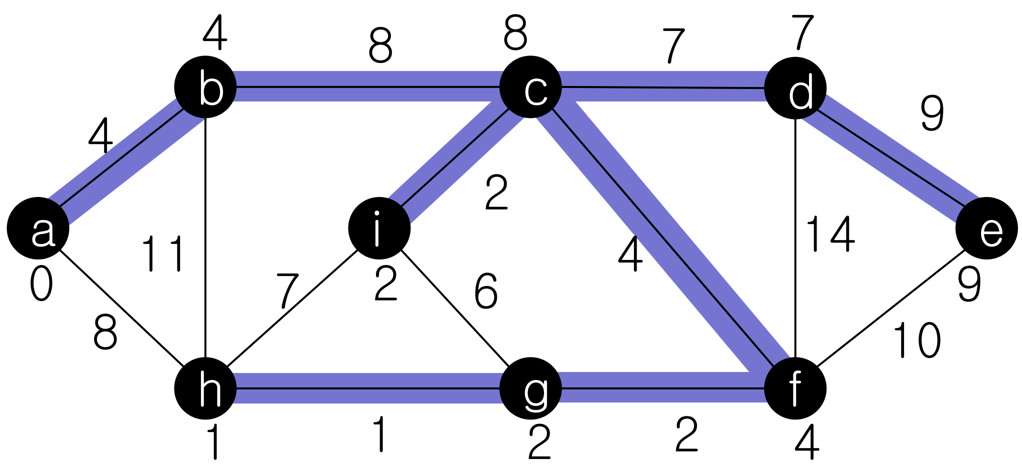
Prim’s Algorithm




```
u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Prim's Algorithm



6. **while** Q ≠ ∅
7. u = Extract-Min(Q)

Running Time of Prim's Algorithm

- The running time of Prim's algorithm depends on how we implement the min-priority queue Q.
- If we implement Q as a binary min-heap,
  - EXTRACT-MIN takes O(lg |V|) time.
  - DECREASE-KEY takes O(lg |V|) time.
- If we implement Q as a simple array,
  - EXTRACT-MIN takes O(|V|) time.
  - DECREASE-KEY O(1) time.
- If we implement Q as a Fibonacci heap,
  - EXTRACT-MIN takes O(lg V|) amortized time.
  - DECREASE-KEY O(1) amortized time.

Prim's Algorithm

```
MST-PRIM(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π=NIL
    r.key = 0
    Q = G.V
    while Q ≠ Ø
        u = Extract-Min(Q)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u,v) < v.key
                v.π = u
                v.key = w(u,v)
```

$O(|V|)$

Q: Implement as a binary min-heap

$O(|V| \lg |V|)$

$O(|E| \lg |V|)$
DECREASE-KEY O(E) times

Prim's Algorithm

```
MST-PRIM(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π=NIL
    r.key = 0
    Q = G.V
    while Q ≠ Ø
        u = Extract-Min(Q)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u,v) < v.key
                v.π = u
                v.key = w(u,v)
```

$O(|V|)$

Q: Implement as an array

$O(|V|^2)$

$O(|E|)$

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Prim’s Algorithm

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π=NIL
r.key = 0
Q = G.V
while Q ≠ Ø
    u = Extract-Min(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

O(|V|)

Q: Implement as a Fibonacci mean-heap

O(|V| lg |V|)

O(|E|)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Minimum-Cost Spanning Trees

- Cost of a spanning tree
  - Sum of the costs (weights) of the edges in the spanning tree
- Min-cost spanning tree
  - A spanning tree of least cost
- Greedy method
  - At each stage, make the best decision possible at the time
    - Based on either a least cost or a highest profit criterion
  - Make sure the decision will result in a feasible solution
    - Satisfy the constraints of the problem
- To construct min-cost spanning trees
  - Best decision : least-cost
  - Constraints
    - Use only edges within the graph
    - Use exactly n-1 edges
    - May not use edges that produce a cycle

Kruskal's Algorithm

- Procedure
  - Build a min-cost spanning tree T by adding edges to T one at a time
  - Select edges for inclusion in T in nondecreasing order of their cost
  - Edge is added to T if it does not form a cycle

Kruskal's Algorithm (Cont.)

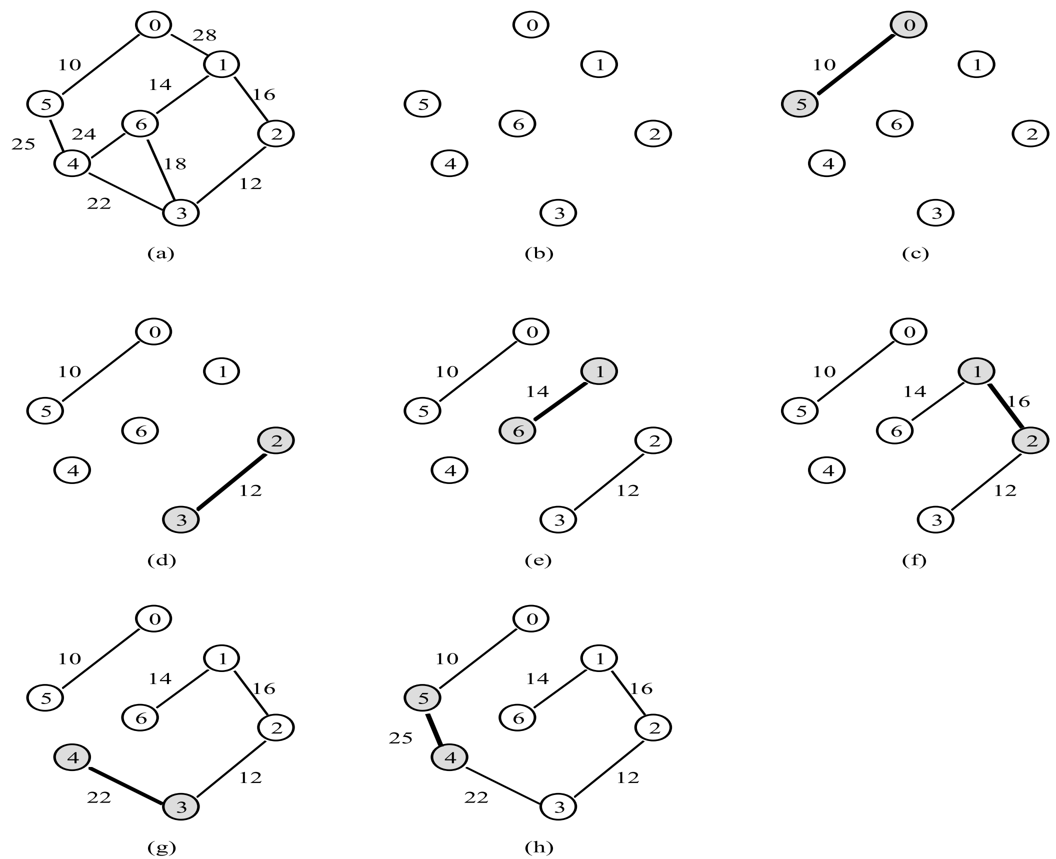


Figure 6.23 : Stages in Kruskal's algorithm

Kruskal's Algorithm (Cont.)

```
T = Φ;
while( (T contains less than n-1 edges) && (E not empty) ) {
      choose an edge (v,w) from E of the lowest cost;
      delete (v,w) from E;
      if( (v,w) does not create a cycle in T ) add (v,w) to T;
      else discard (v,w);
}
if (T contains fewer than n-1 edge) cout << “no spanning tree” << endl;
```

Program 6.6: Kruskal’s algorithm

- Time Complexity
  - When we use a min heap to determine the lowest cost edge , O(eloge)

Prim’s Algorithm

- Property
  - At all times during the algorithm the set of selected edges forms a tree
- Procedure
  - Begin with a tree T that contains a single vertex
  - Add a least-cost edge (u,v) to T such that $T \cup \{(u,v)\}$ is also a tree
  - Repeat until T contains n-1 edges

Prim's Algorithm (Cont.)

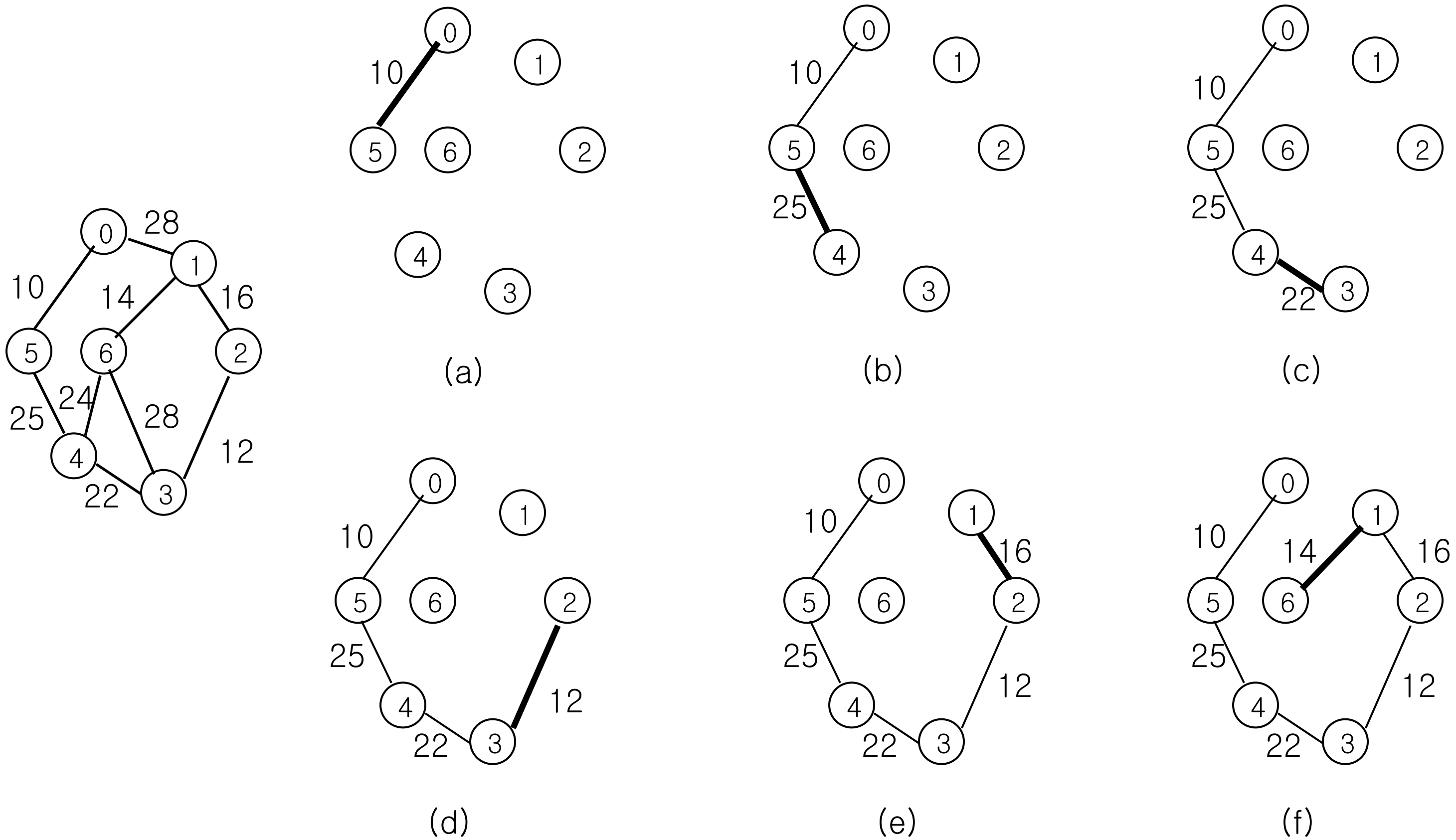


Figure 6.24: Stages in Prim's algorithm

Prim's Algorithm (Cont.)

```
// Assume that G has at least one vertex.
TV = { 0 };  // start with vertex 0 and no edges
for(T = Φ; T contains fewer than n-1 edges; add (u,v) to T)
{
    Let (u,v) be a least-cost edge such that u ∈ TV and v !∈ TV;
    if(there is no such edge) break;
    add v to TV;
}
if(T contains fewer than n-1 edges) cout << "no spanning tree" << endl;
```

Program 6.7: Prim's algorithm

Fibonacci Heaps

Introduction to Data Structures

Kyuseok Shim

ECE, SNU.

Priority Queues Summary

Operation	Linked List	Binary Heap	Fibonacci Heap †
make-heap	1	1	1
is-empty	1	1	1
insert	1	log n	1
delete-min	n	log n	log n
decrease-key	n	log n	1
delete	n	log n	log n
union	1	n	1
find-min	n	1	1

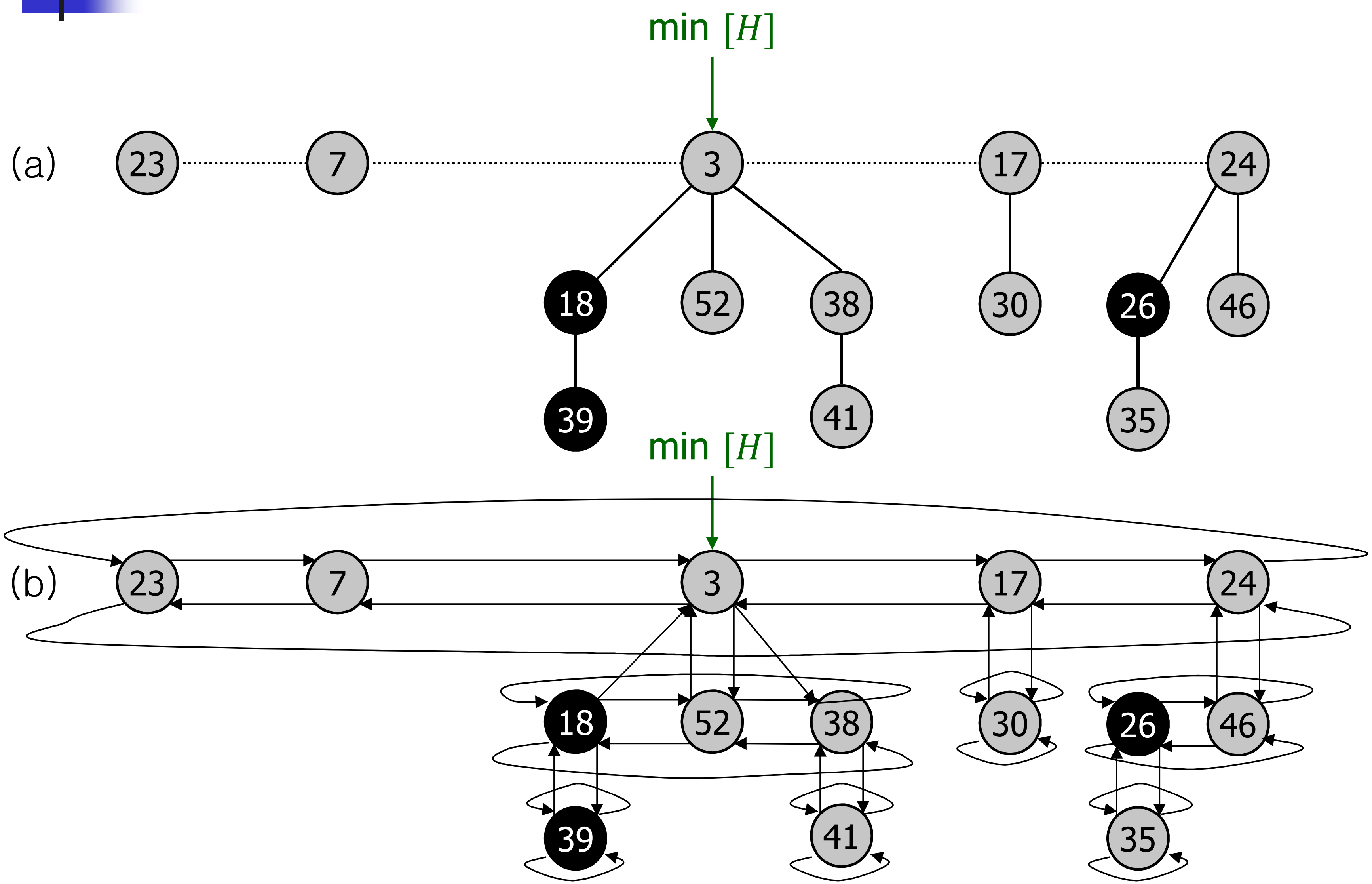
n = number of elements in priority queue

† amortized

Fibonacci Heaps

- Fibonacci heap history. Fredman and Tarjan (1986)
  - Ingenious data structure and analysis.
  - Original motivation - *V insert, V delete-min, E decrease-key*
    - Improve Dijkstra's shortest path algorithm from $O(E \log V)$ to $O(E + V \log V)$.
    - Also improve MST(minimum spanning tree) algorithm.
- Basic idea
  - Similar to binomial heaps, but less rigid structure.
  - Fibonacci heap: lazily defer consolidation until next *delete-min*.
  - Decrease-key and union run in O(1) time.

Structure of Fibonacci Heaps

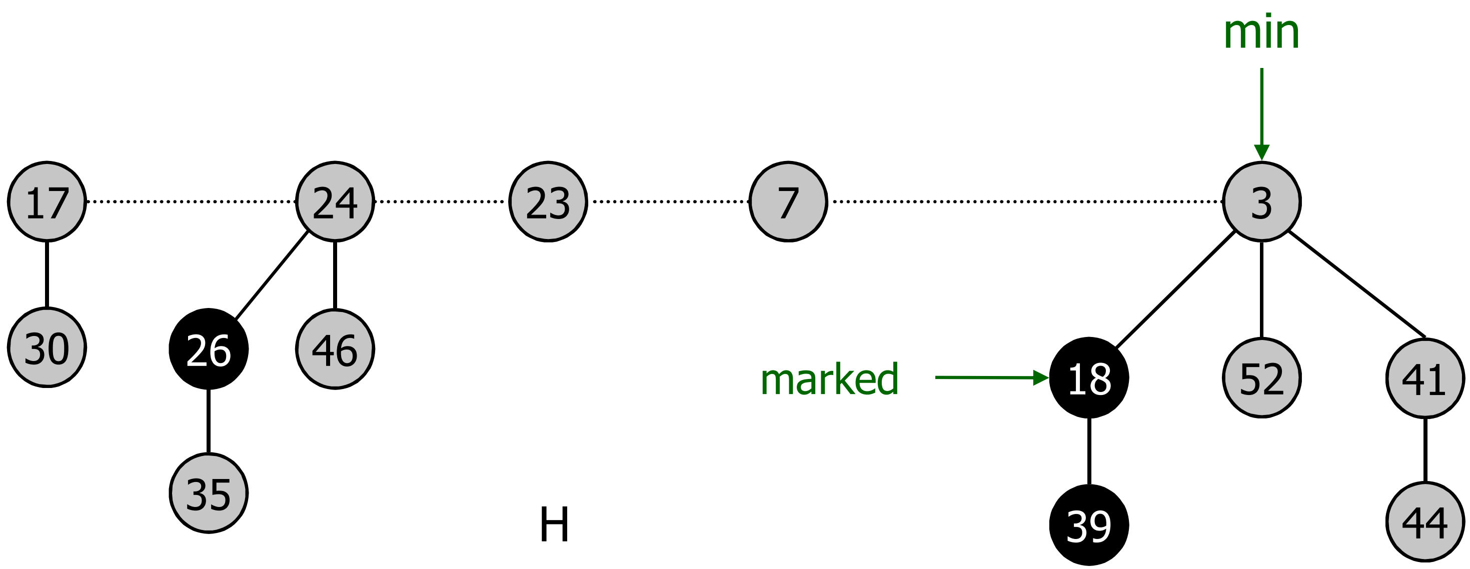


Fibonacci Heaps

- Advantages of using circular doubly linked lists
  - Inserting a node in to any location takes O(1) time.
  - Removing a node from anywhere takes O(1) time.
  - Concatenating two such lists into one circular doubly linked list takes O(1) time.

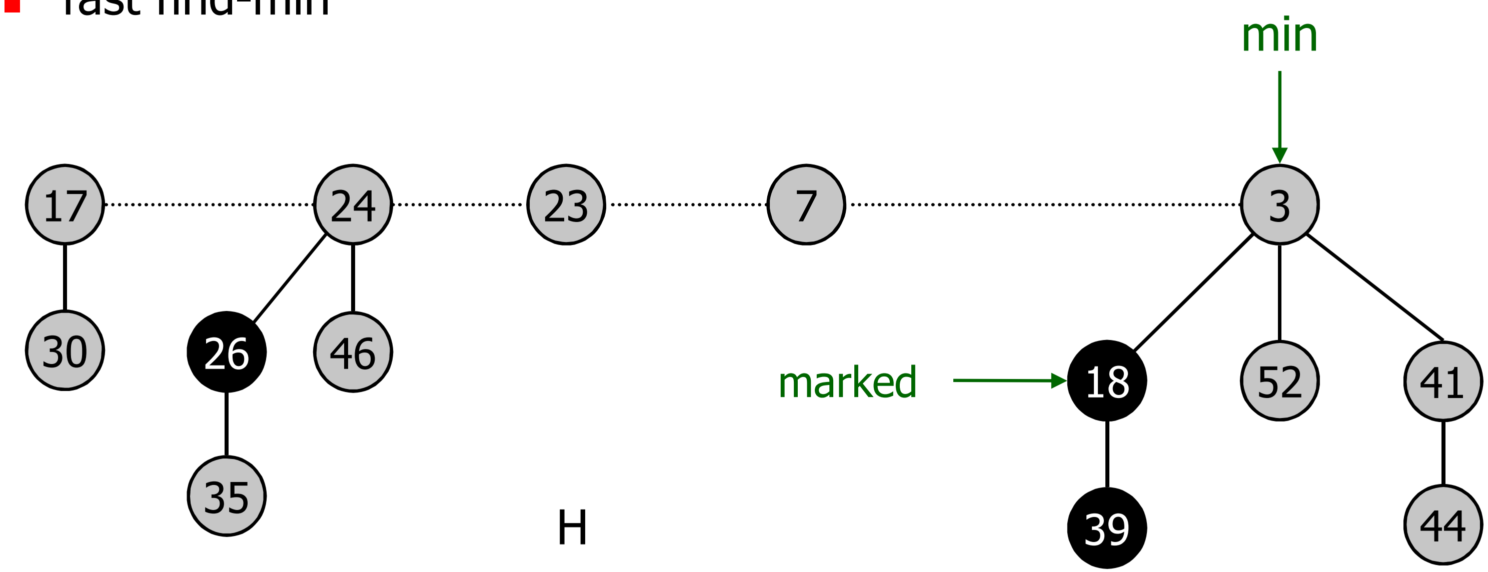
Fibonacci Heaps: Structure

- Fibonacci heap.
  - Set of heap-ordered trees.
  - Maintain pointer to minimum element.
  - Set of marked nodes.



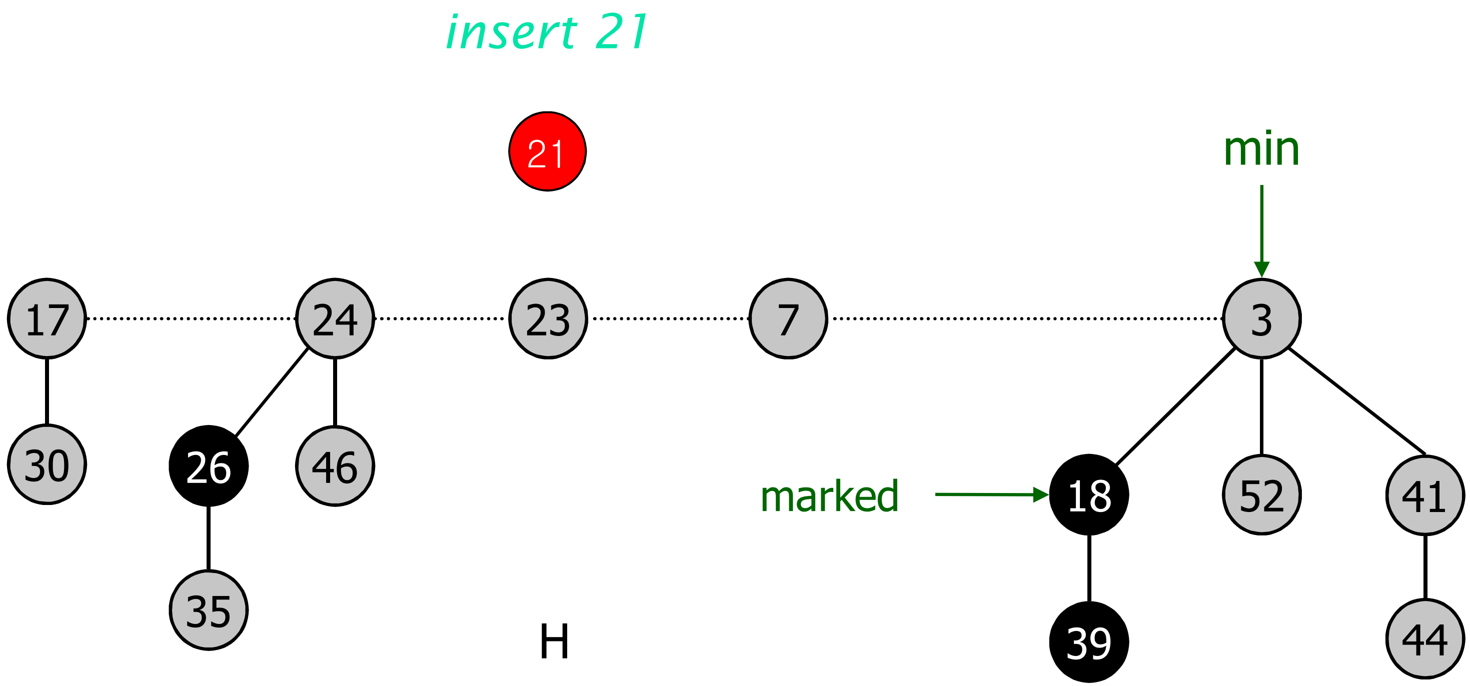
Fibonacci Heaps: Implementation

- Represent trees using left-child, right sibling pointers and circular, doubly linked list.
  - can quickly splice off subtrees
- Roots of trees connected with circular doubly linked list.
  - fast union
- Pointer to root of tree with min element.
  - fast find-min



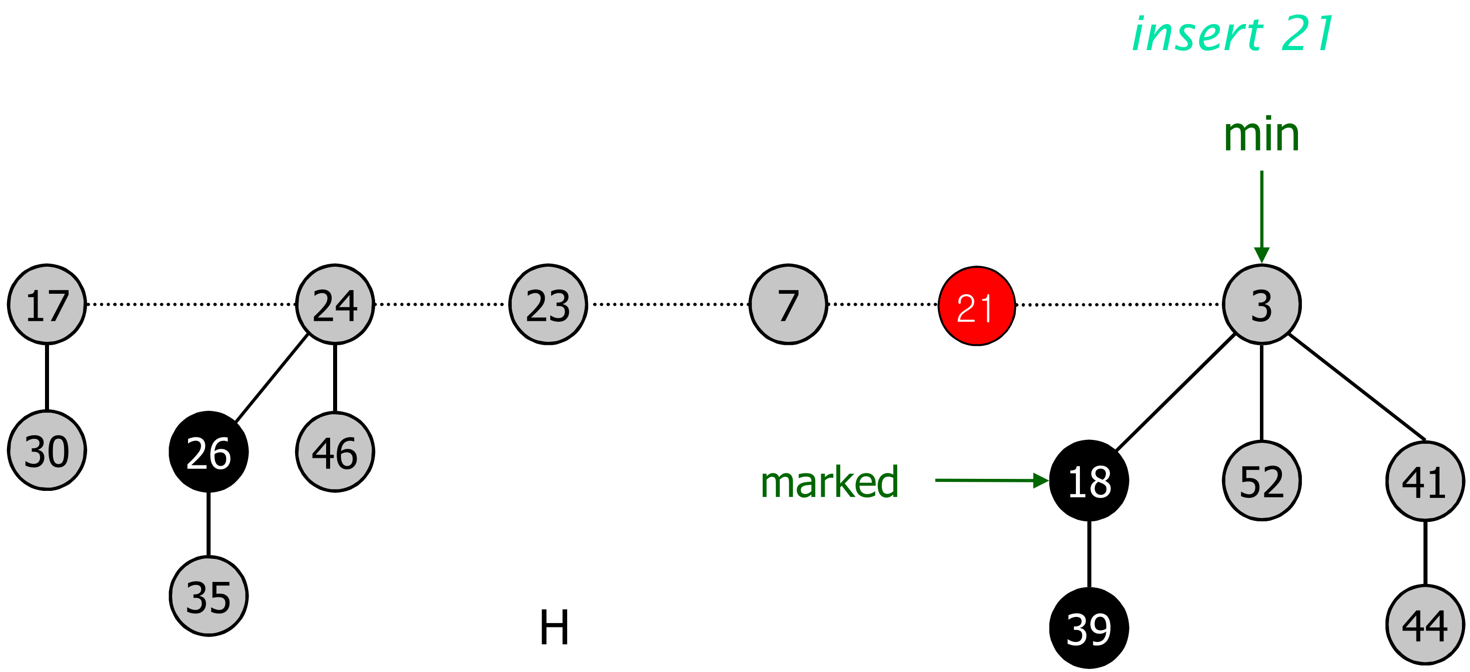
Fibonacci Heaps: Insert

- Insert.
  - Create a new singleton tree.
  - Add to left of min pointer.
  - Update min pointer.



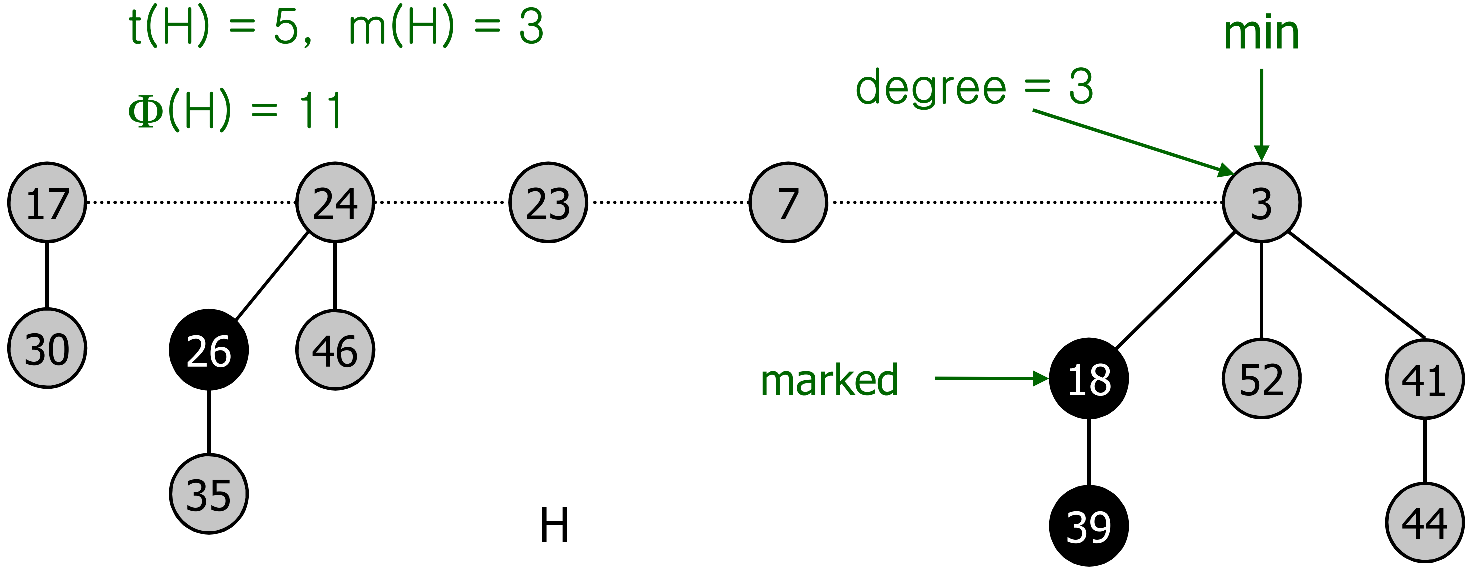
Fibonacci Heaps: Insert

- Insert.
  - Create a new singleton tree.
  - Add to left of min pointer.
  - Update min pointer.



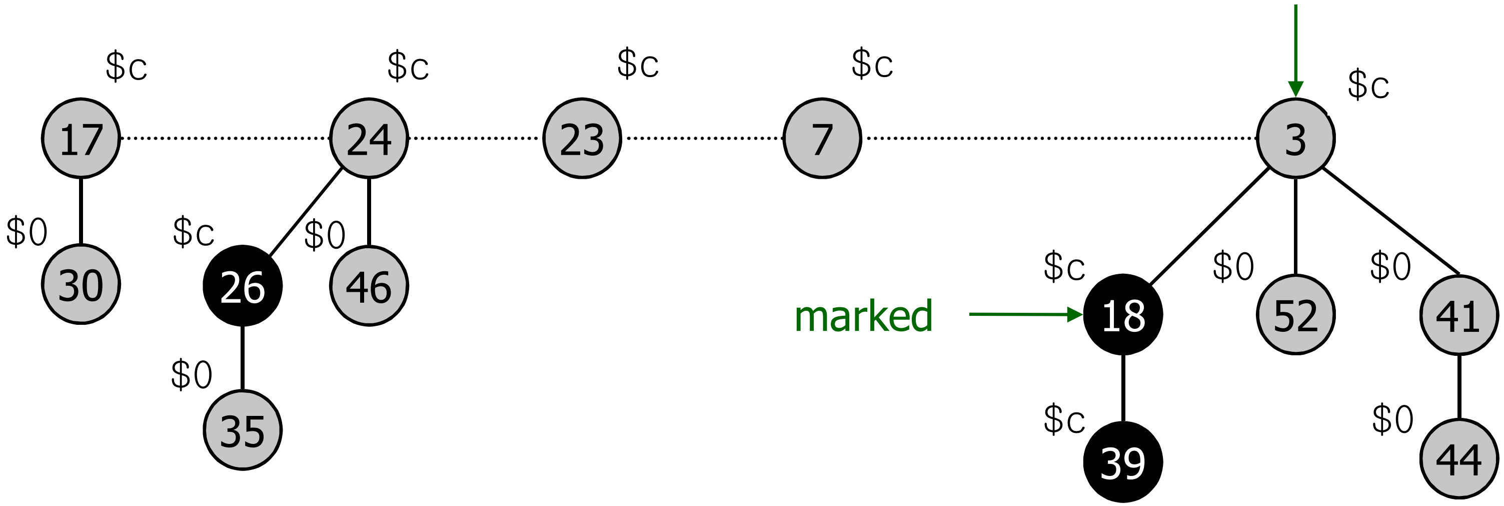
Fibonacci Heaps: Potential Function

- $\Phi(H) = t(H) + 2m(H)$
  - D(n) = max degree of any node in Fibonacci heap with n nodes.
  - Mark[x] = mark of node x (black or gray).
  - t(H) = number of trees in heap *H*.
  - m(H) = number of marked nodes in heap *H*.



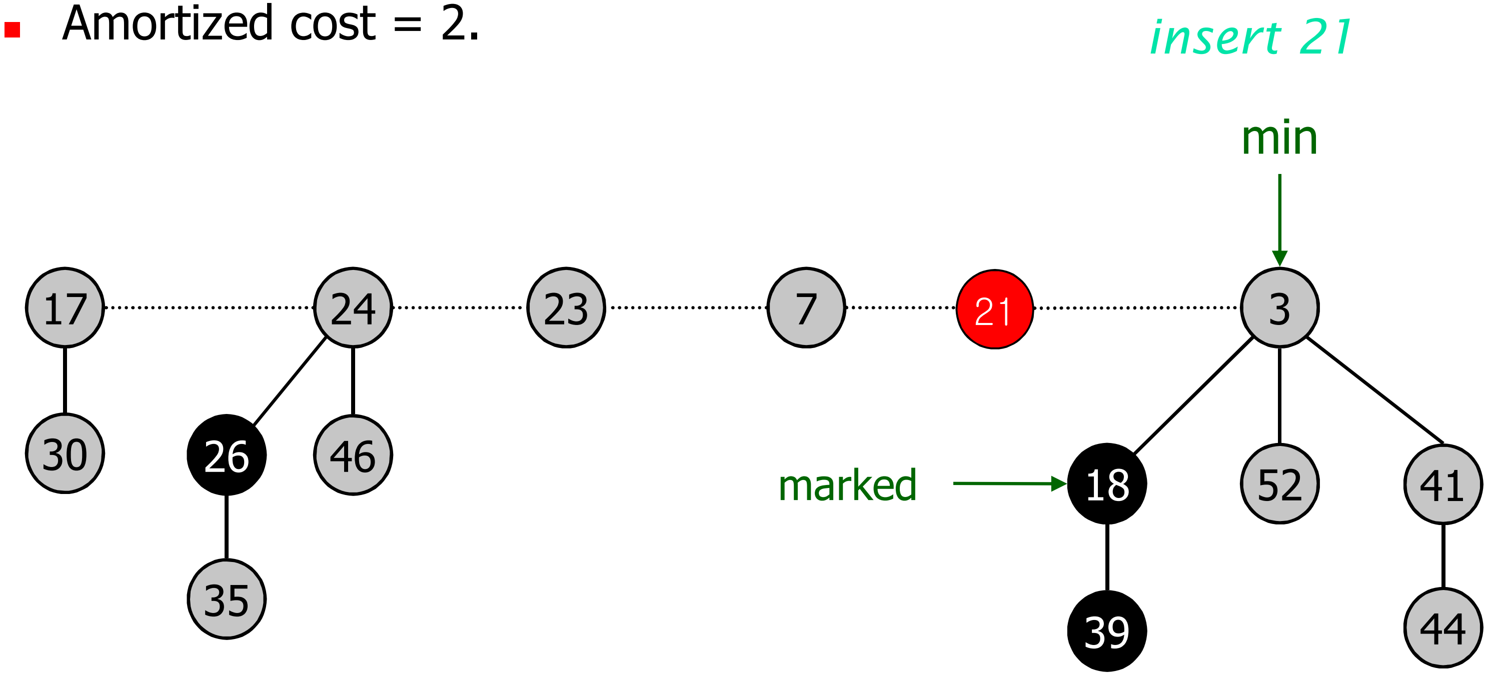
Fibonacci Heaps: Potential Function

- Intuition
  - Root node들은 $c credit을 가짐.
  - Marked node들은 $c credit을 가짐.
  - 나머지 internal node들은 $0 credit을 가짐.
  - Node가 root list로 올라갈 때에는 $c credit을 붙여주고 marked node이면 $c credit을 unmark하는데 사용한다.
  - 따라서, 모든 root node들은 $c credit을 갖게 됨.



Fibonacci Heaps: Insert

- $\Phi(H) = t(H) + 2m(H)$
  - The initial value of $\Phi(H) = 0$.
- Amortized cost. O(1)
  - Actual cost = 1.
  - Change in potential = +1.
  - Amortized cost = 2.

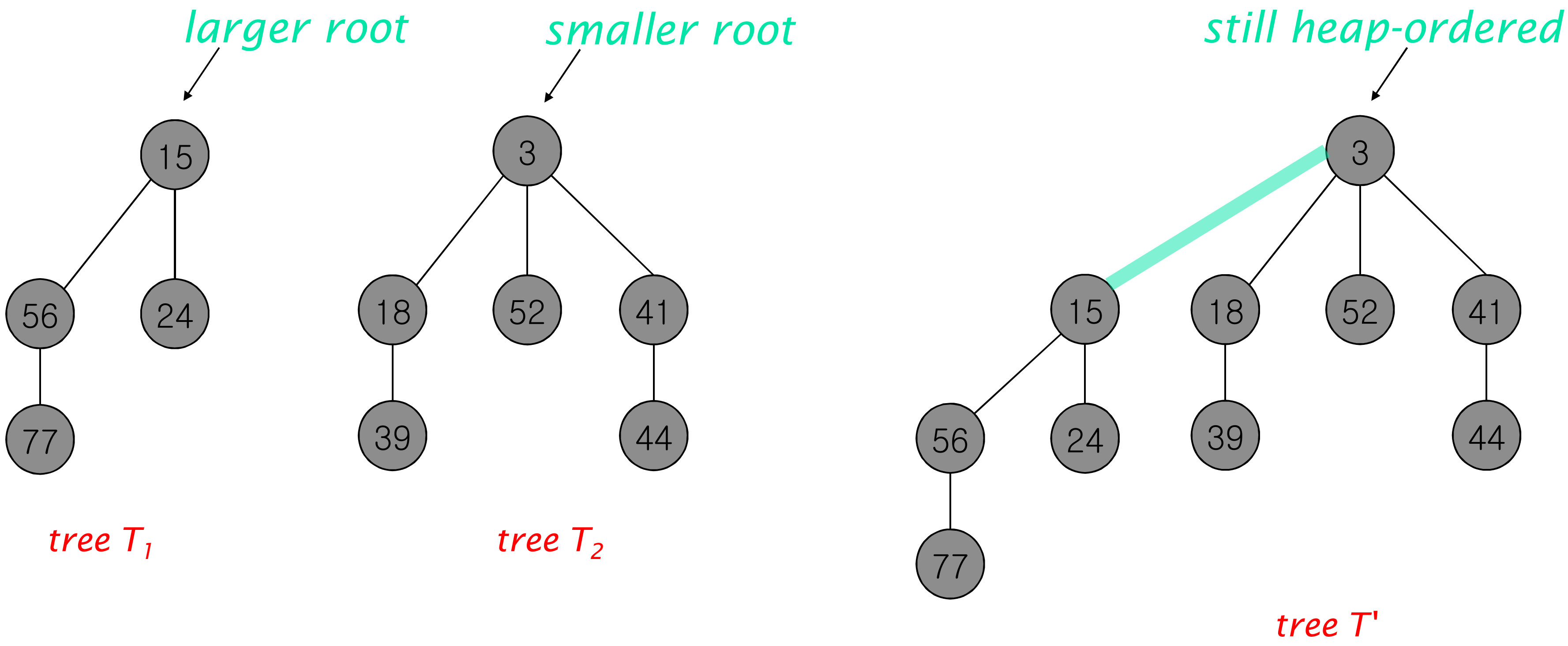


Fibonacci Heaps: Insert Analysis

- Notation.
  - D(n) = max degree of any node in Fibonacci heap with n nodes.
  - t(H) = number of trees in heap *H*.
  - m(H) = number of marked nodes in heap *H*.
  - Φ(H) = t(H) + 2m(H).
- $\Delta\Phi$(H) = 1
  - Before extracting minimum node
    - Φ(H) = t(H) + 2m(H).
  - After extracting minimum node, t(H') $\leq$ D(n) + 1 since no two trees have same degree
    - Φ(H') = t(H) + 1 + 2m(H)
- Amortized cost. O(1)
  - $\hat{c_i} = c_i + \Phi_i - \Phi_{i-1} = 1 + (t(H) + 1 + 2m(H)) - (t(H) + 2m(H)) = 2$

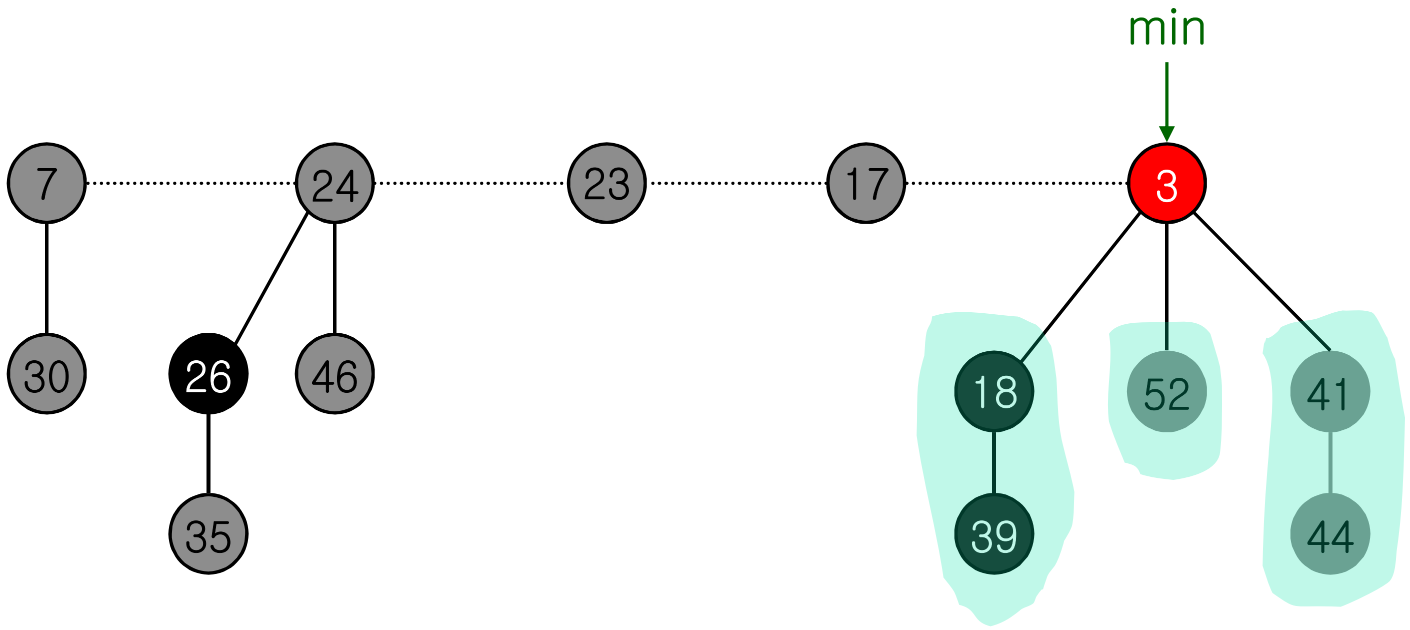
Fibonacci Heaps: Delete

- Linking operation.
  - Make larger root be a child of smaller root.



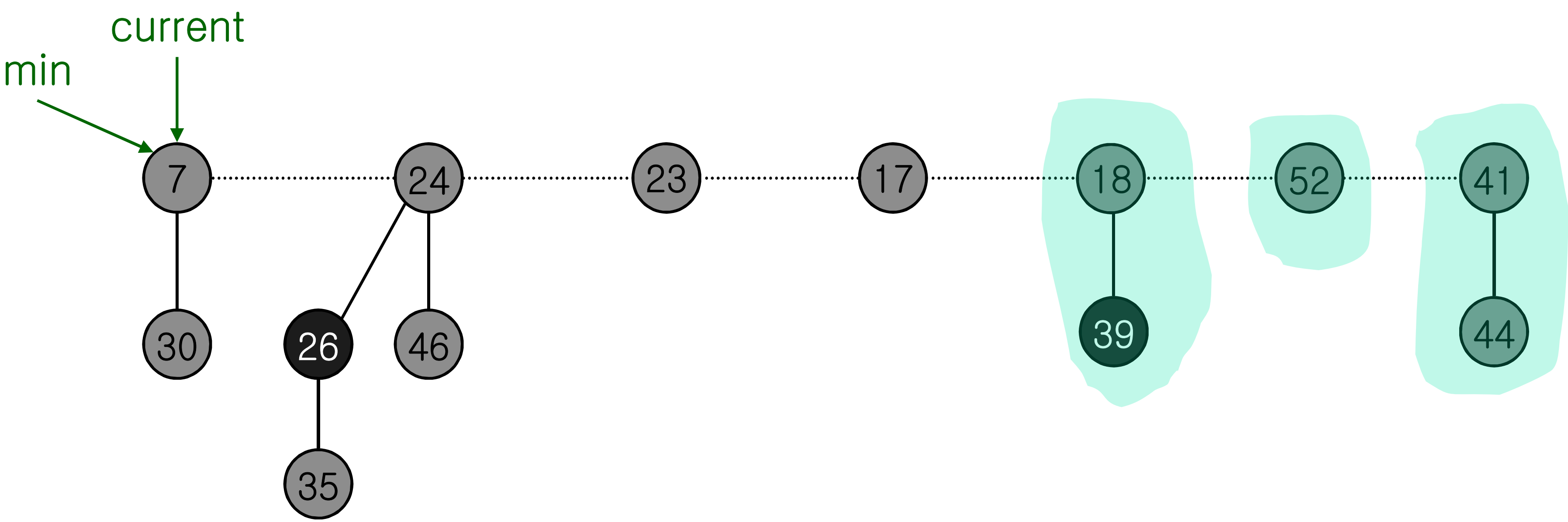
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



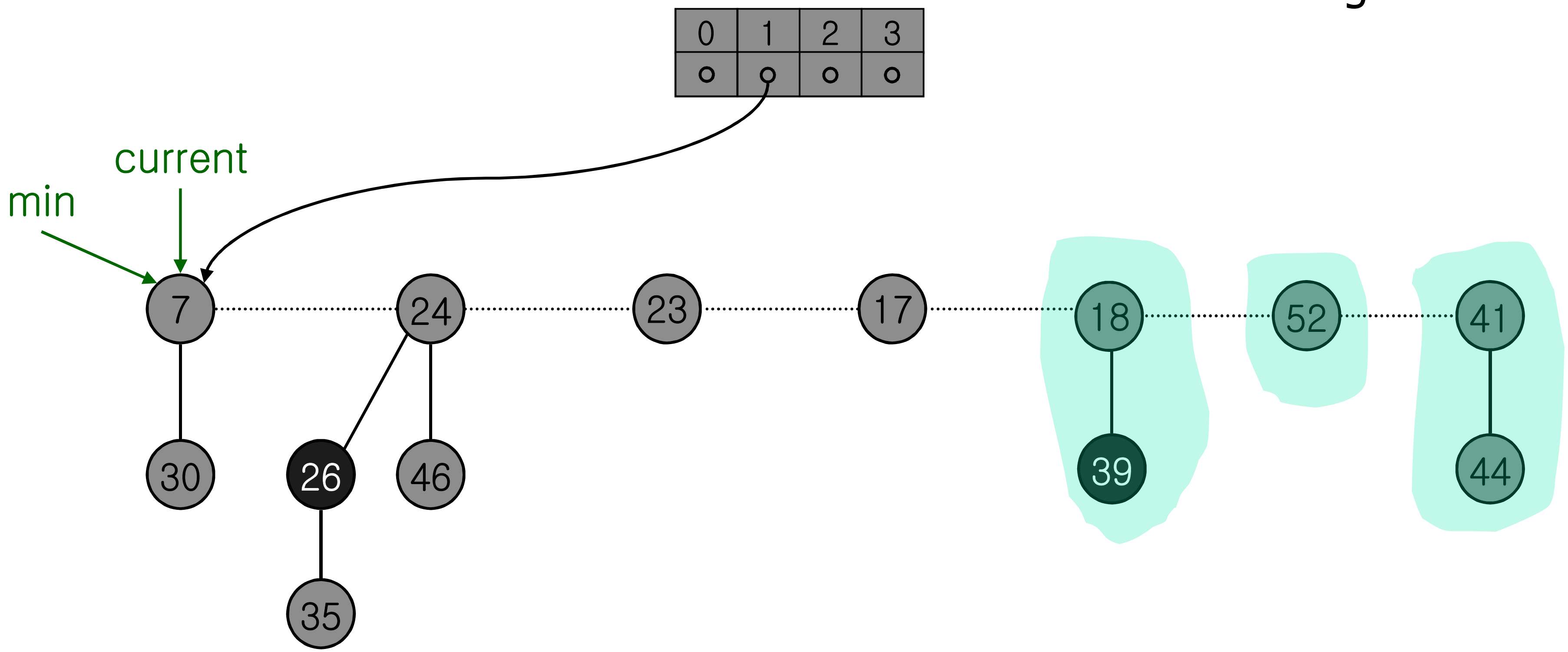
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



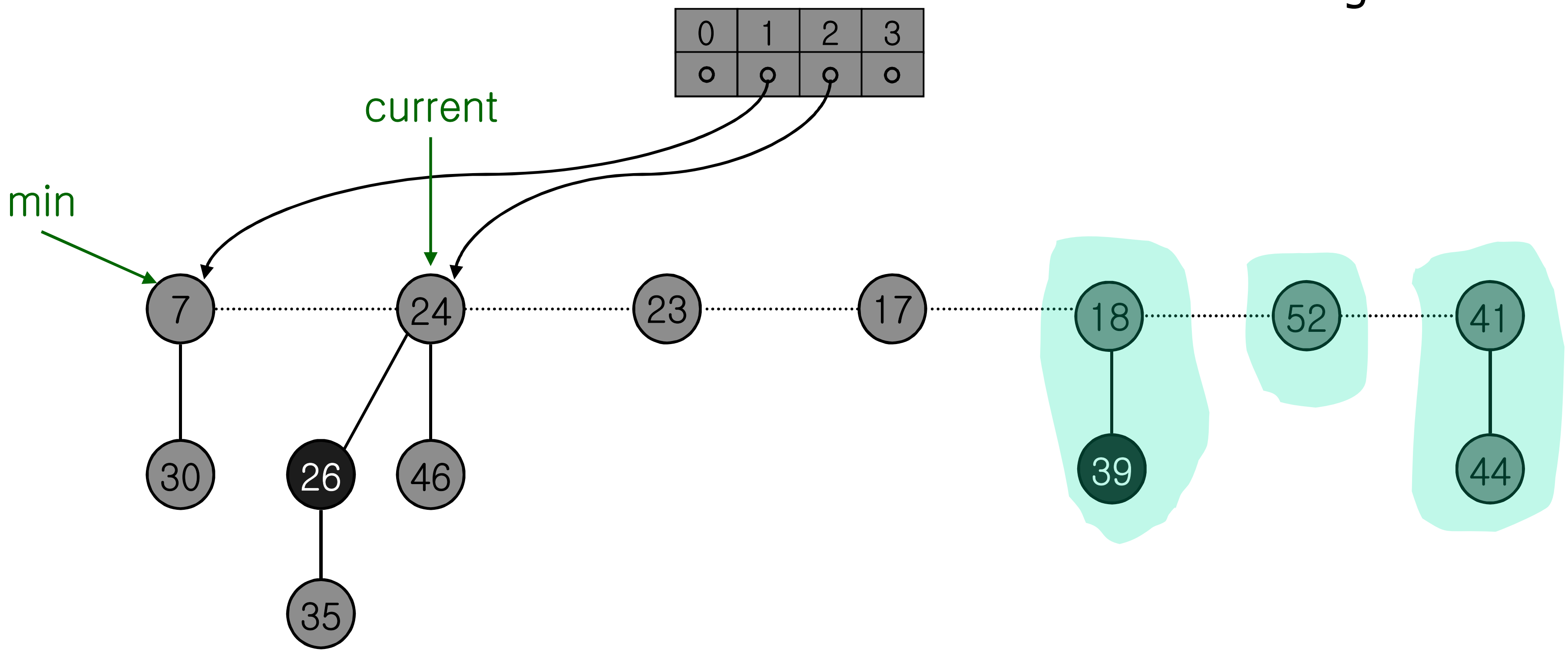
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



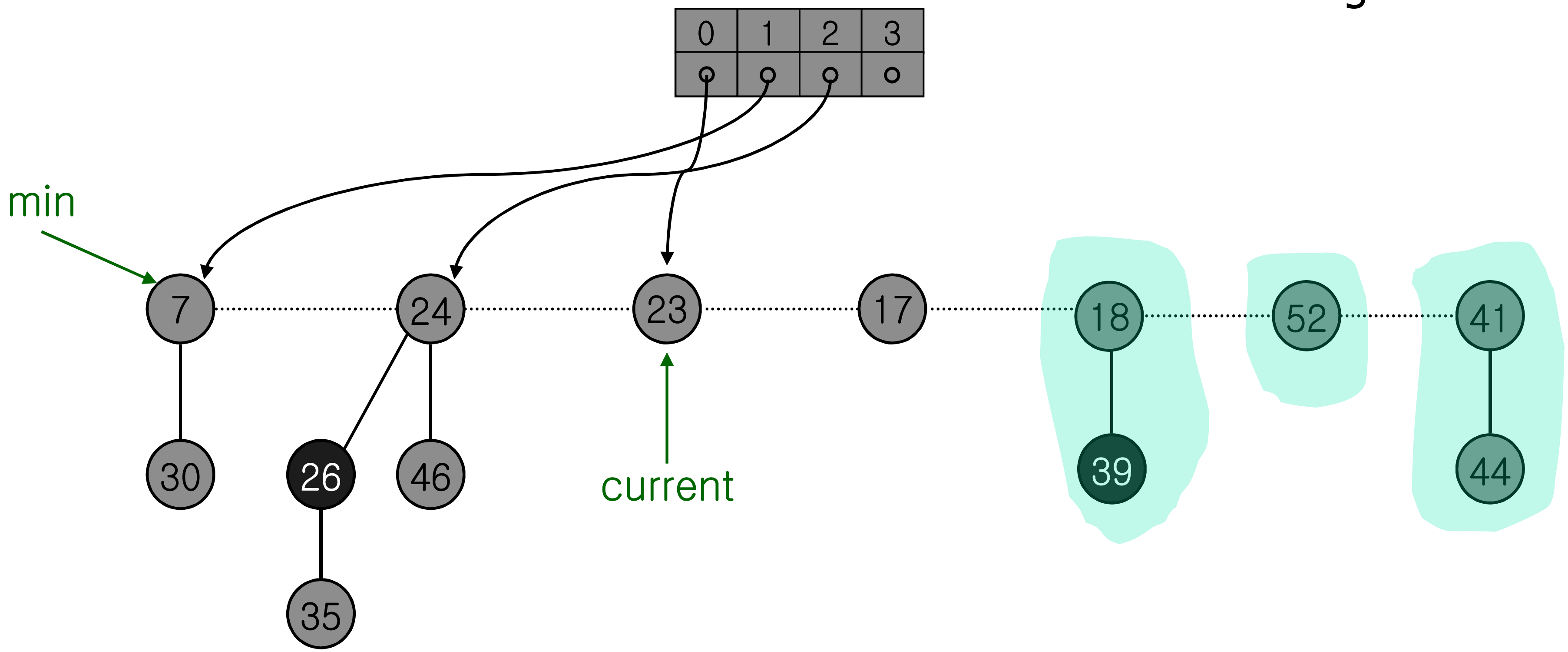
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



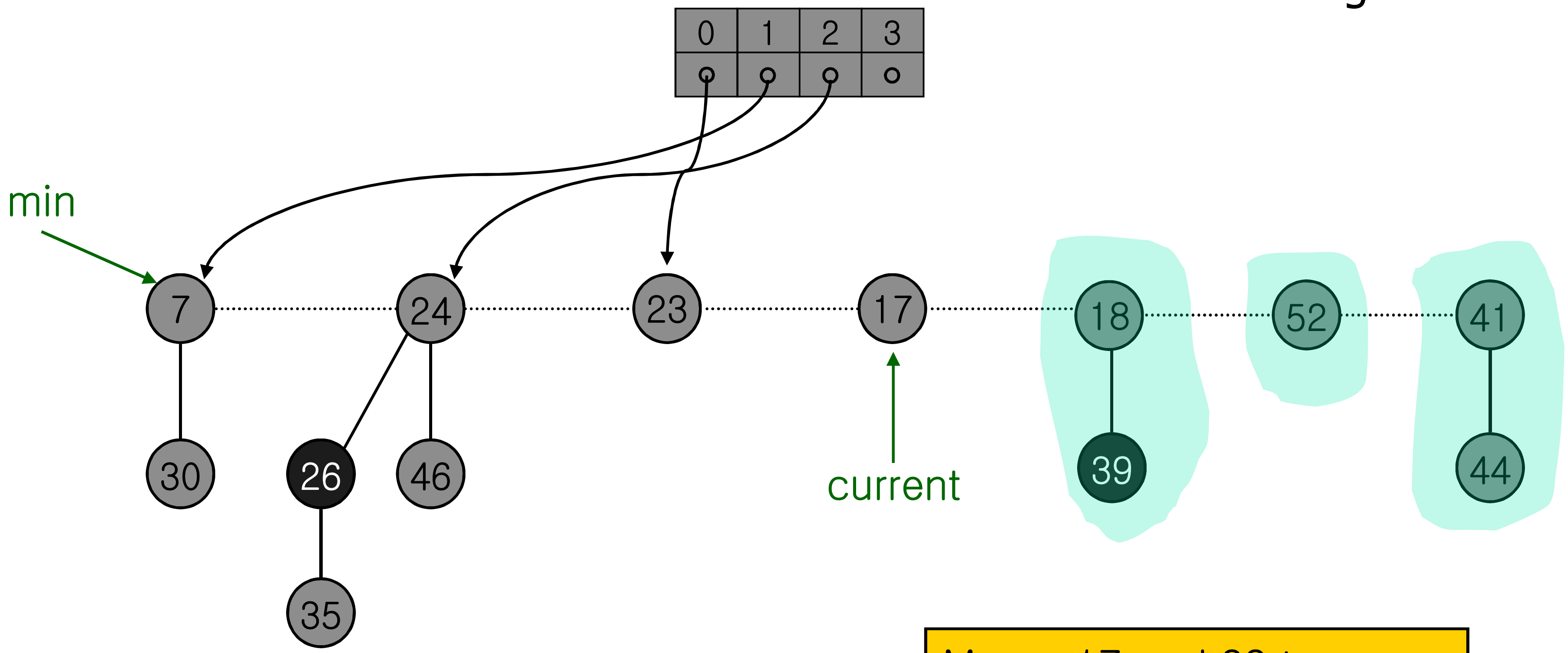
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



Fibonacci Heaps: Delete

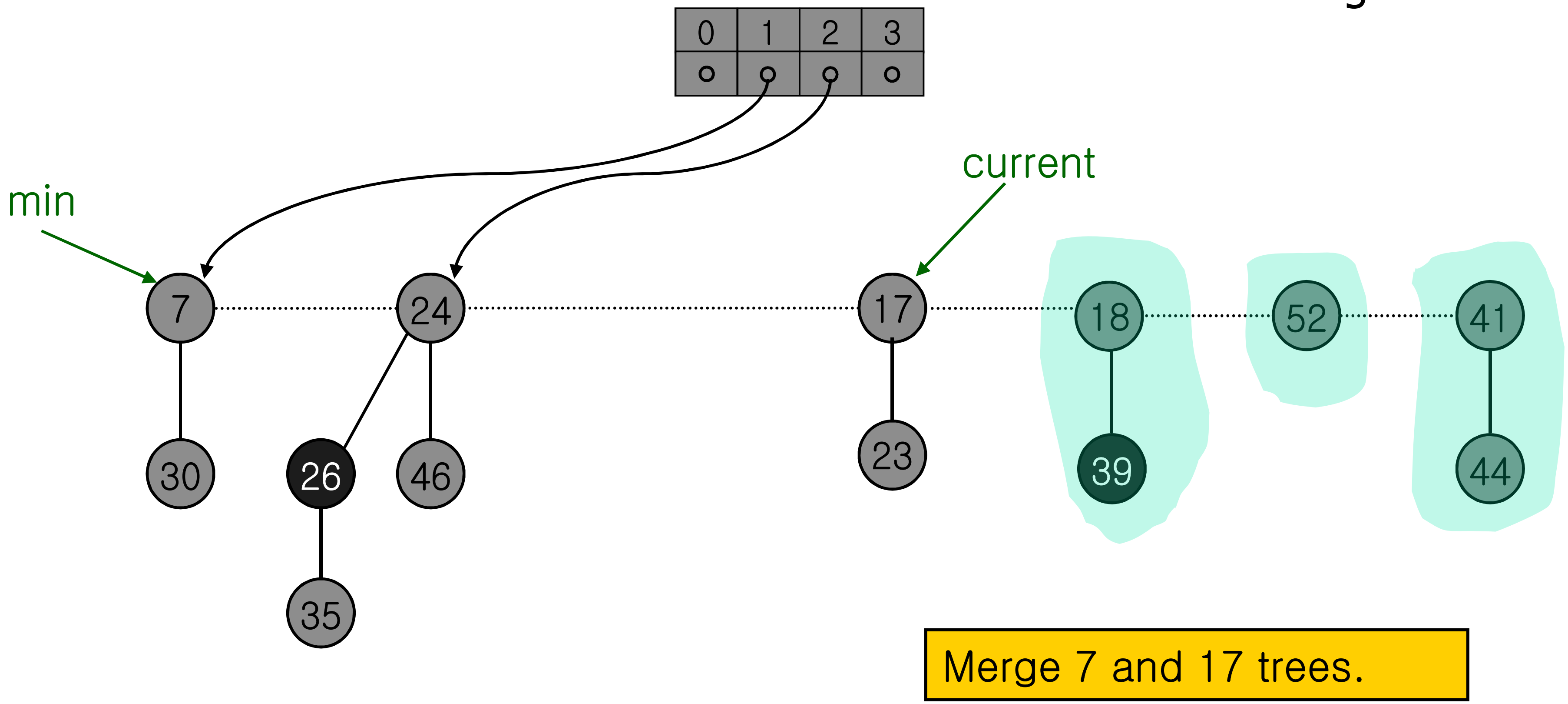
- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



Merge 17 and 23 trees.

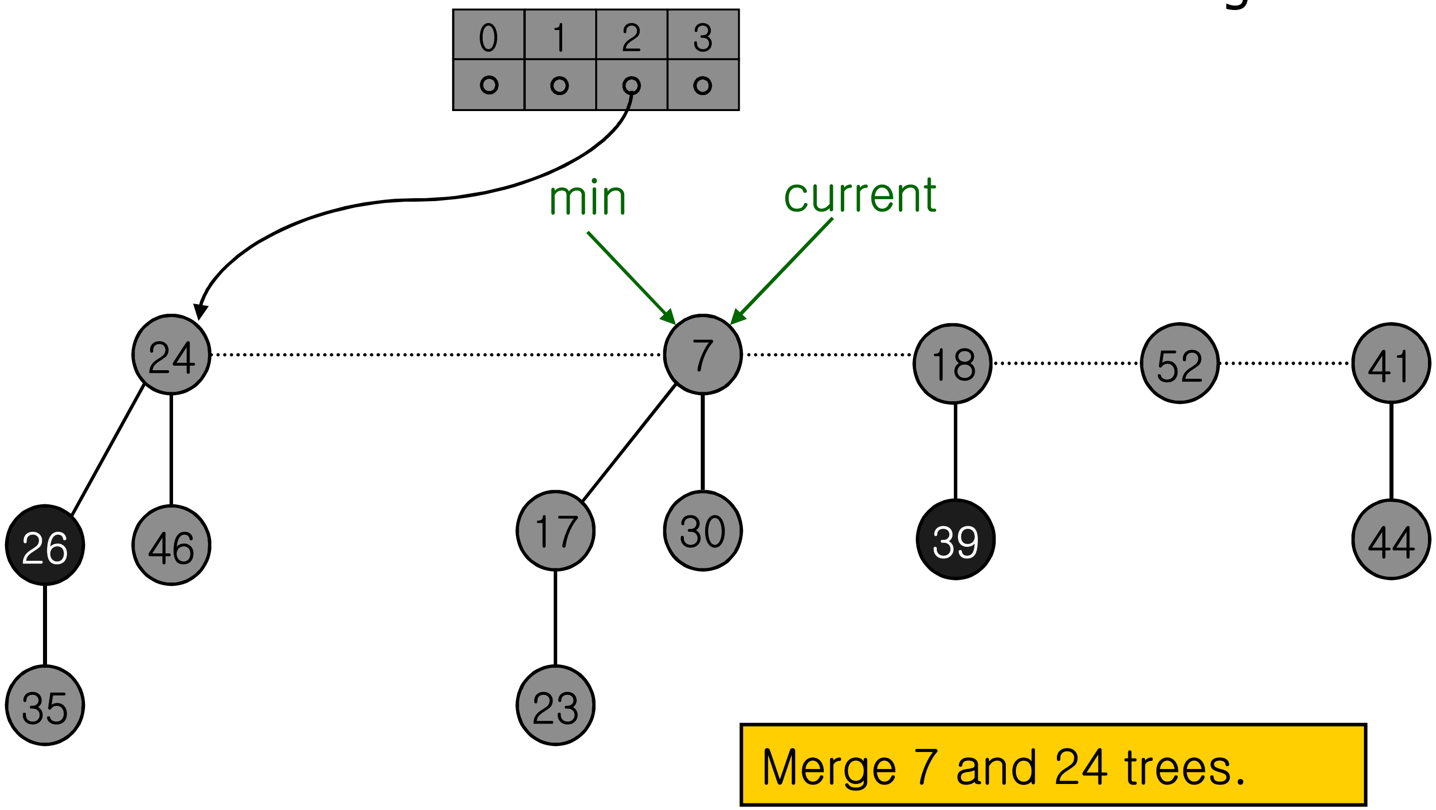
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



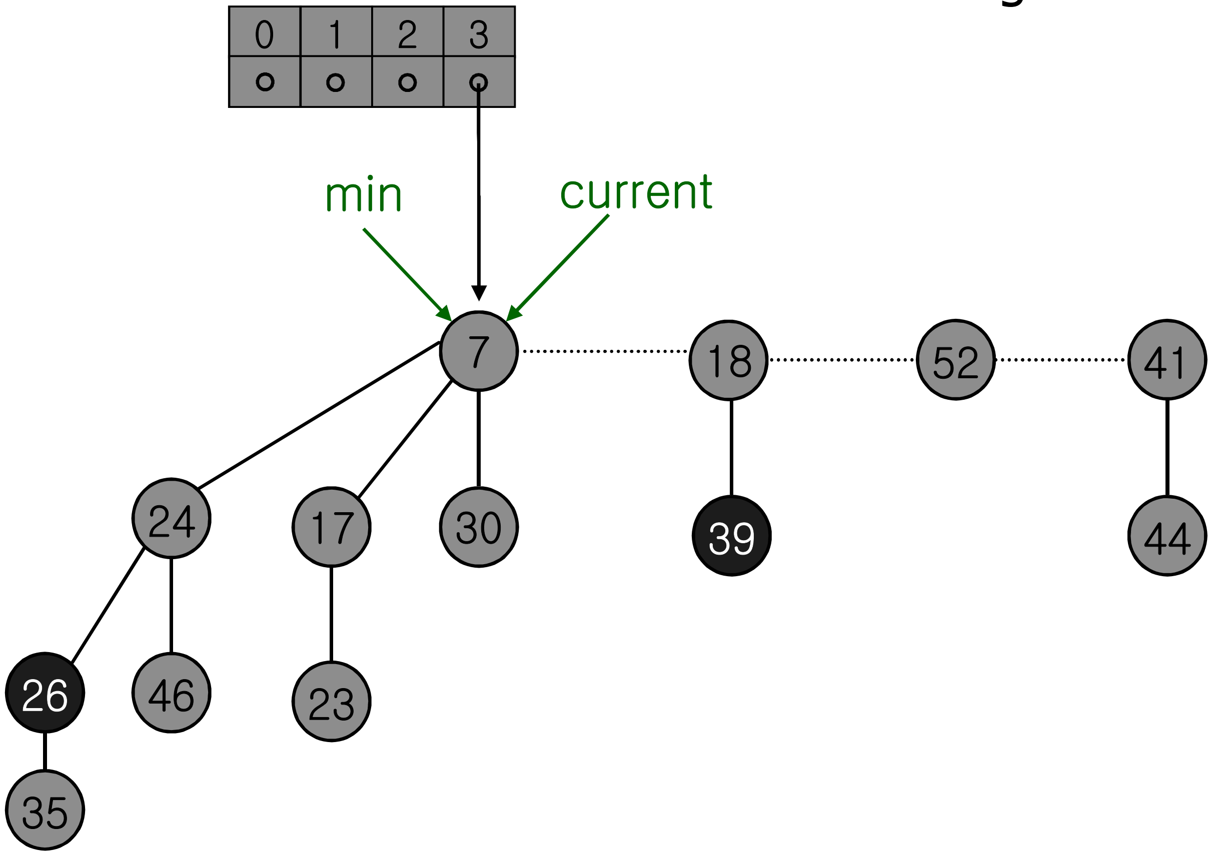
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



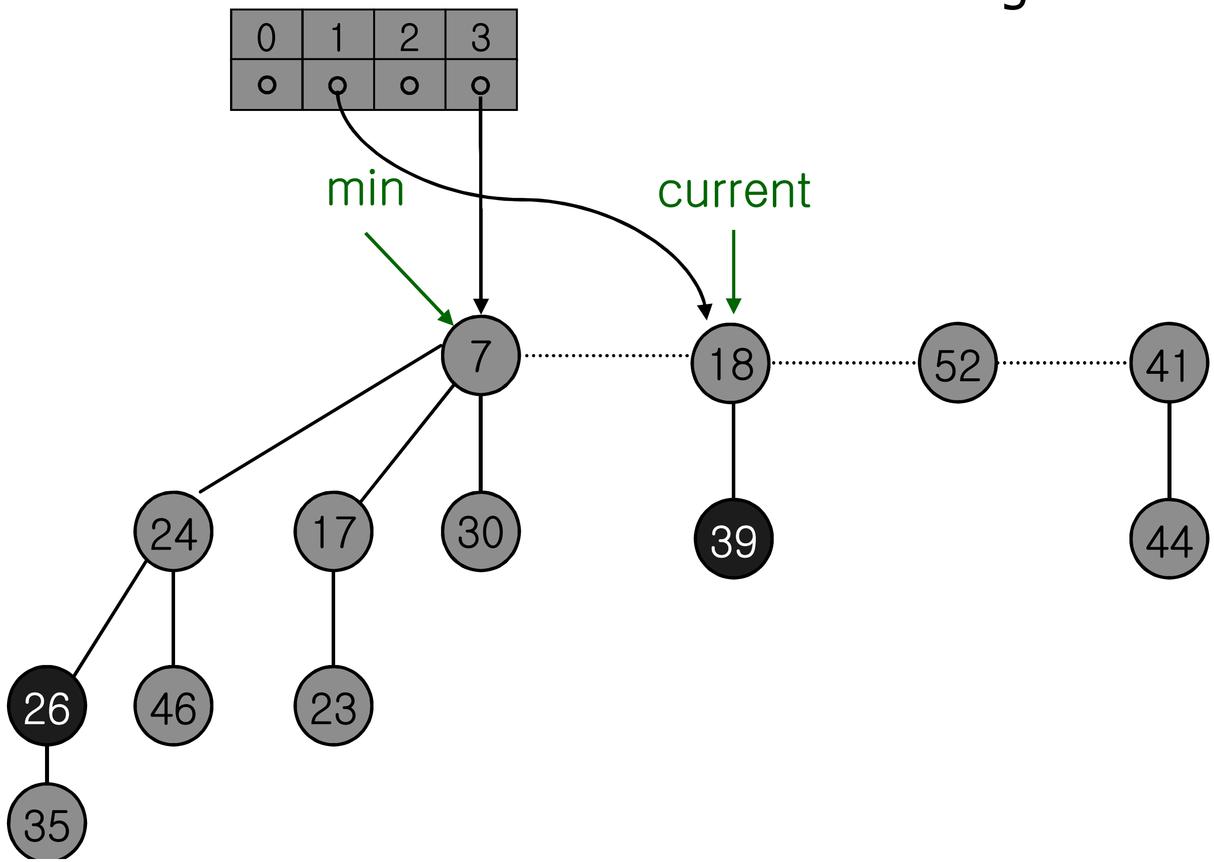
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



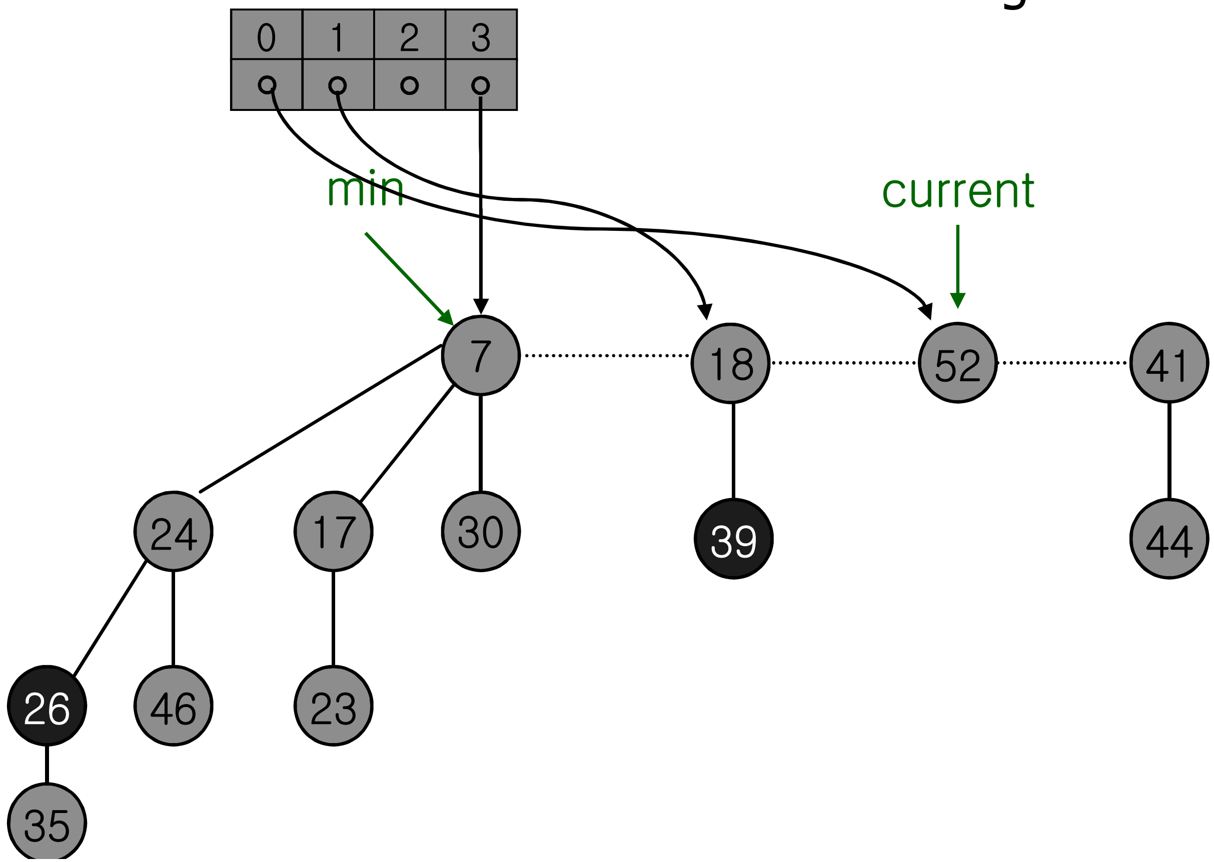
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



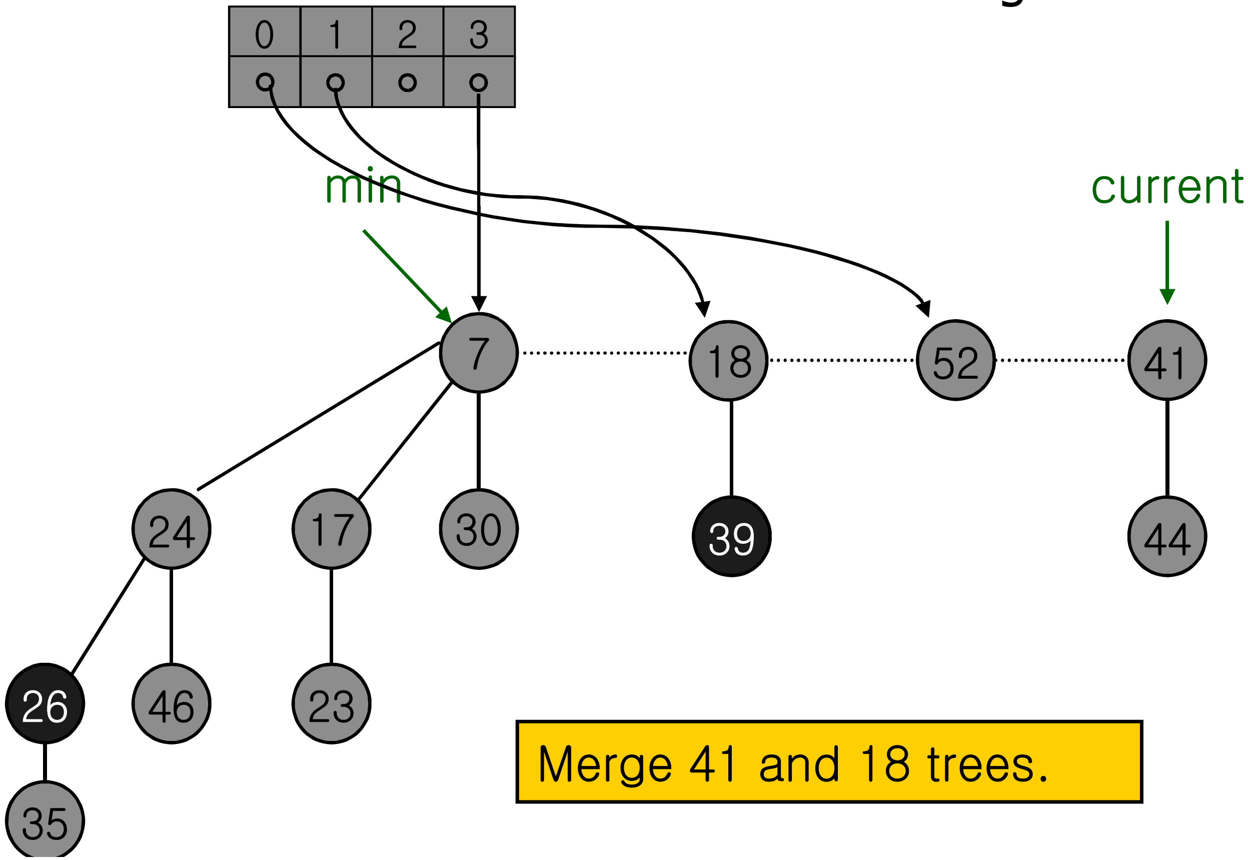
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



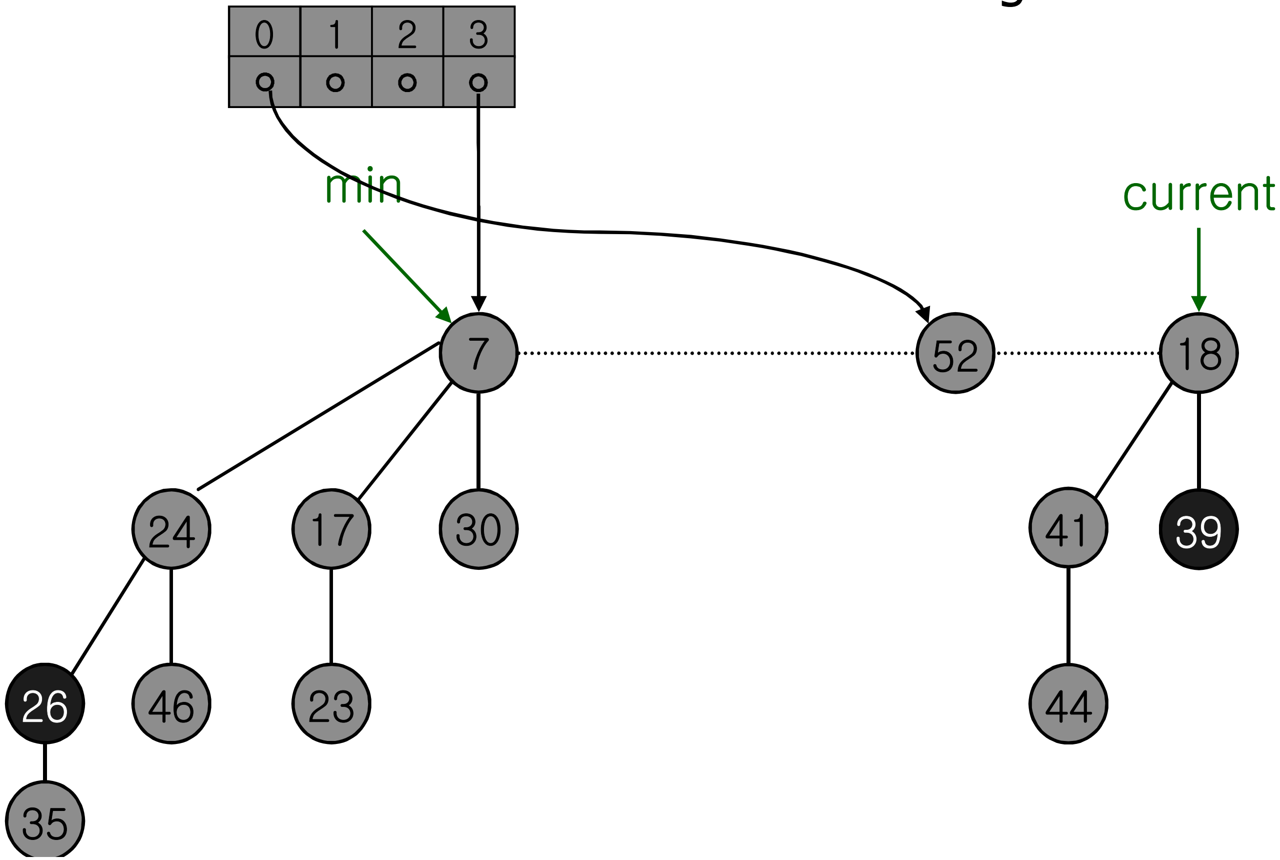
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



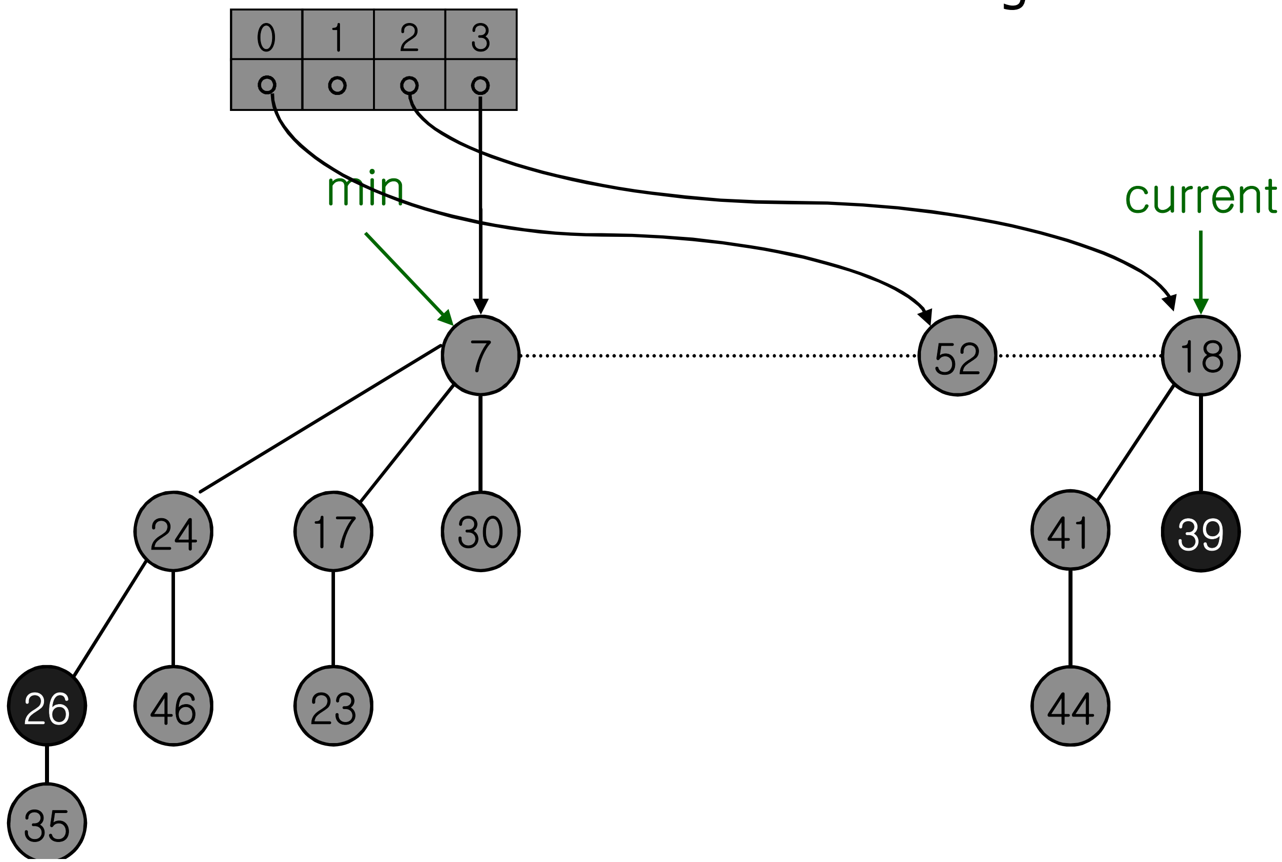
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



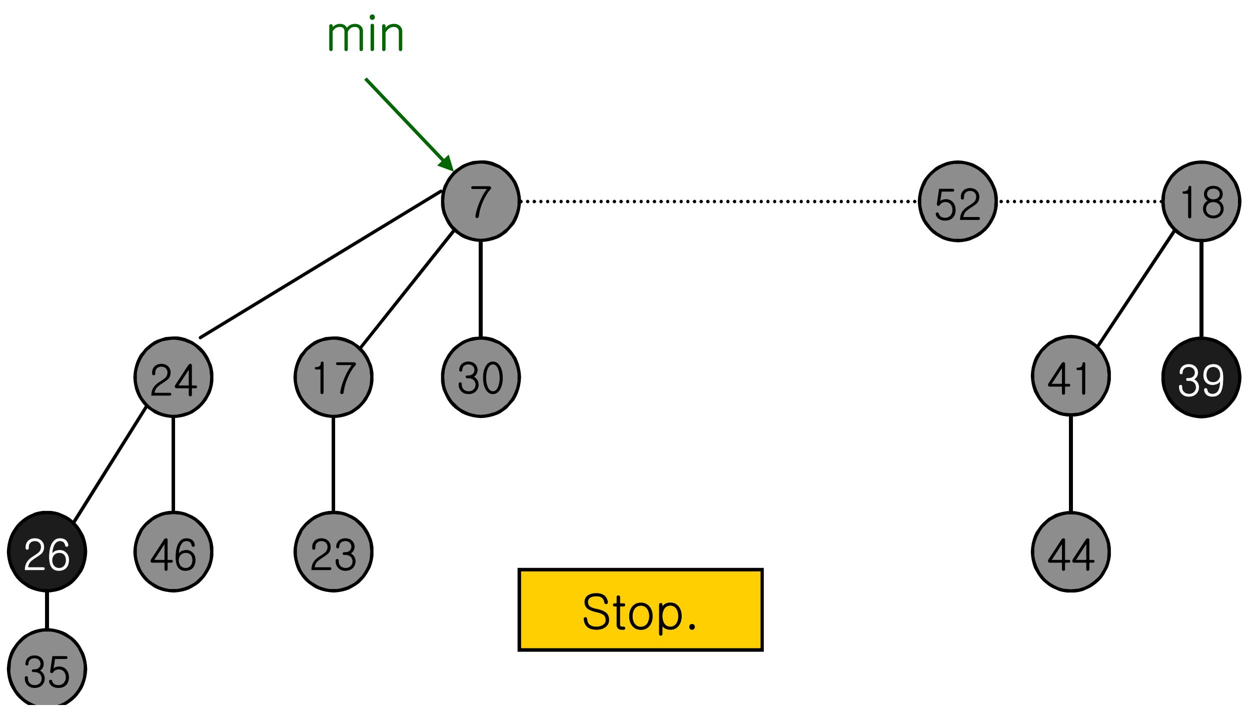
Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



Fibonacci Heaps: Delete

- Delete min.
  - Delete min and concatenate its children into root list.
  - Consolidate trees so that no two roots have same degree.



Fibonacci Heaps: Delete Min Analysis

- Notation.
  - D(n) = max degree of any node in Fibonacci heap with n nodes.
  - t(H) = number of trees in heap *H*.
  - m(H) = number of marked nodes in heap *H*.
  - Φ(H) = t(H) + 2m(H).
- Actual cost. 2D(n) + t(H)
  - D(n) + 1 work adding min's children into root list and updating min.
    - at most D(n) children of min node
  - D(n) + t(H) - 1 work consolidating trees.
    - work is proportional to size of root list since number of roots decreases by one after each merging
    - D(n) + t(H) - 1 root nodes at beginning of consolidation

Fibonacci Heaps: Delete Min Analysis

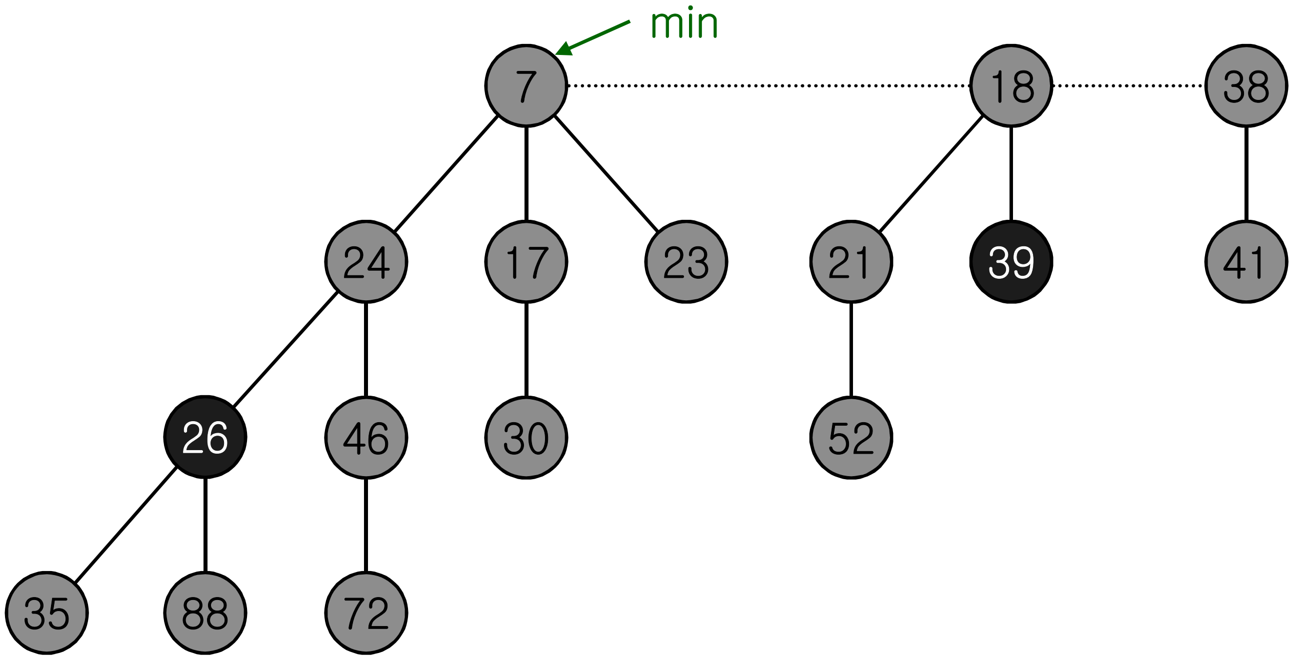
- Notation.
  - D(n) = max degree of any node in Fibonacci heap with n nodes.
  - t(H) = number of trees in heap *H*.
  - m(H) = number of marked nodes in heap *H*.
  - $\Phi(H)$ = t(H) + 2m(H).
- $\Delta\Phi(H)$ = D(n) + 1 - t(H)
  - Before extracting minimum node
    - $\Phi(H)$ = t(H) + 2m(H).
  - After extracting minimum node, t(H') $\leq$ D(n) + 1 since no two trees have same degree
    - $\Phi(H')$ = D(n) + 1 + 2m(H)
- Amortized cost. O(D(n))
  - $\hat{c_i} = c_i + \Phi_{\mathrm{i}} - \Phi_{\mathrm{i-1}}$ = (2D(n) + t(H)) + (D(n) + 1 + 2m(H)) – (t(H) + 2m(H)) = 3D(n)+1=O(D(n))

Fibonacci Heaps: Delete Min Analysis

- Is amortized cost of O(D(n)) good?
  - Yes, if only Insert, Delete-min, and Union operations supported.
    - Fibonacci heap contains only binomial trees since we only merge trees of equal root degree
    - This implies $D(n) \leq \lfloor \log_2 N \rfloor$
  - Yes, if we support Decrease-key in clever way.
    - we'll show that $D(n) \leq \lfloor \log_\phi N \rfloor$, where ϕ is golden ratio
    - $\phi^2 = 1 + \phi$
    - $\phi = (1 + \sqrt{5}) / 2 = 1.618\ldots$
    - limiting ratio between successive Fibonacci numbers!

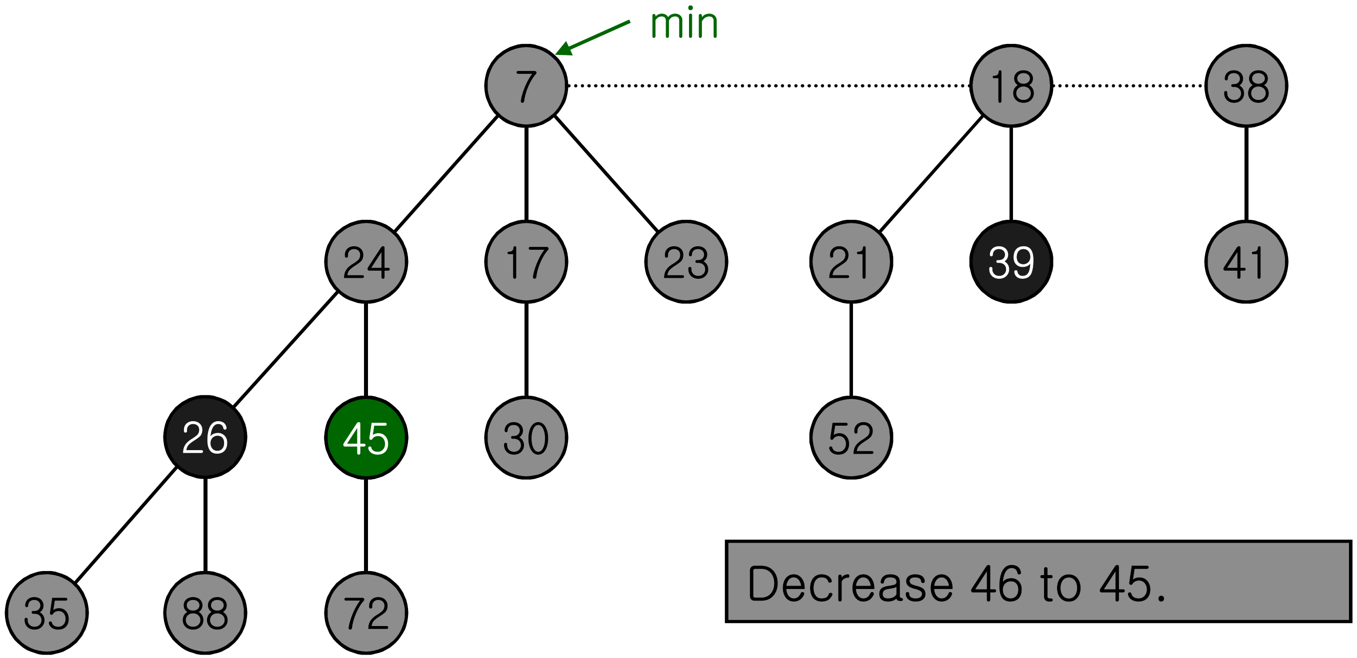
Fibonacci Heaps: Decrease Key

- Mark.
  - Indicate whether node x has lost a child since the last time x was made the child of another node.
  - Newly created nodes are unmarked.
  - Whenever a node is made the child of another node, it is unmarked.



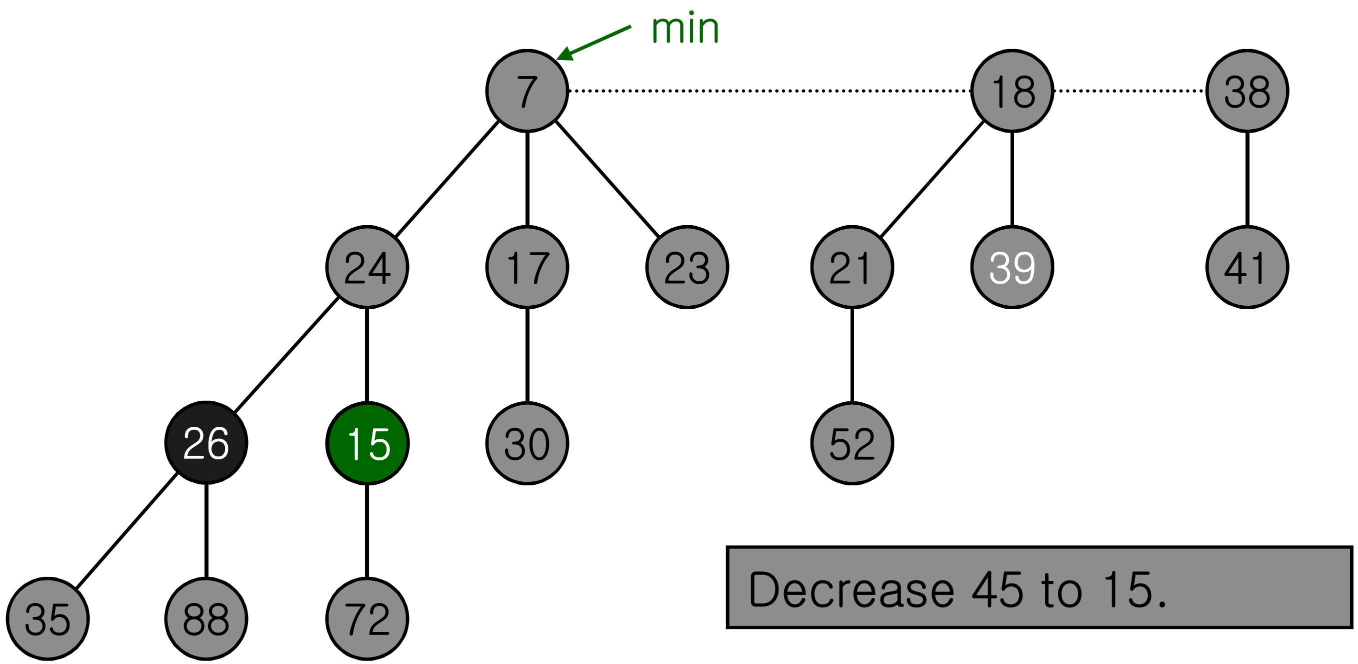
Fibonacci Heaps: Decrease Key

- Decrease key of element x to k.
  - Case 0: min-heap property not violated.
    - Decrease key of x to k
    - Change heap min pointer if necessary



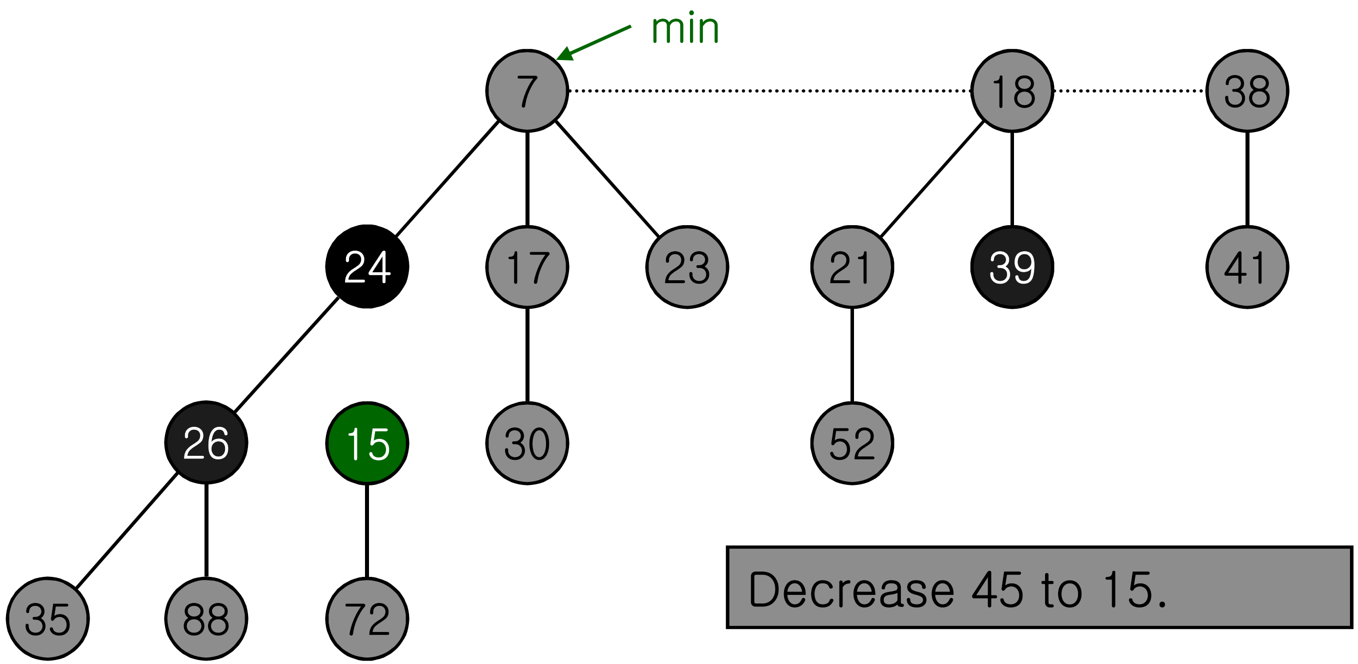
Fibonacci Heaps: Decrease Key

- Decrease key of element x to k.
  - Case 1: parent of x is unmarked.
    - decrease key of x to k
    - cut off link between x and its parent
    - mark parent
    - add tree rooted at x to root list, updating heap min pointer



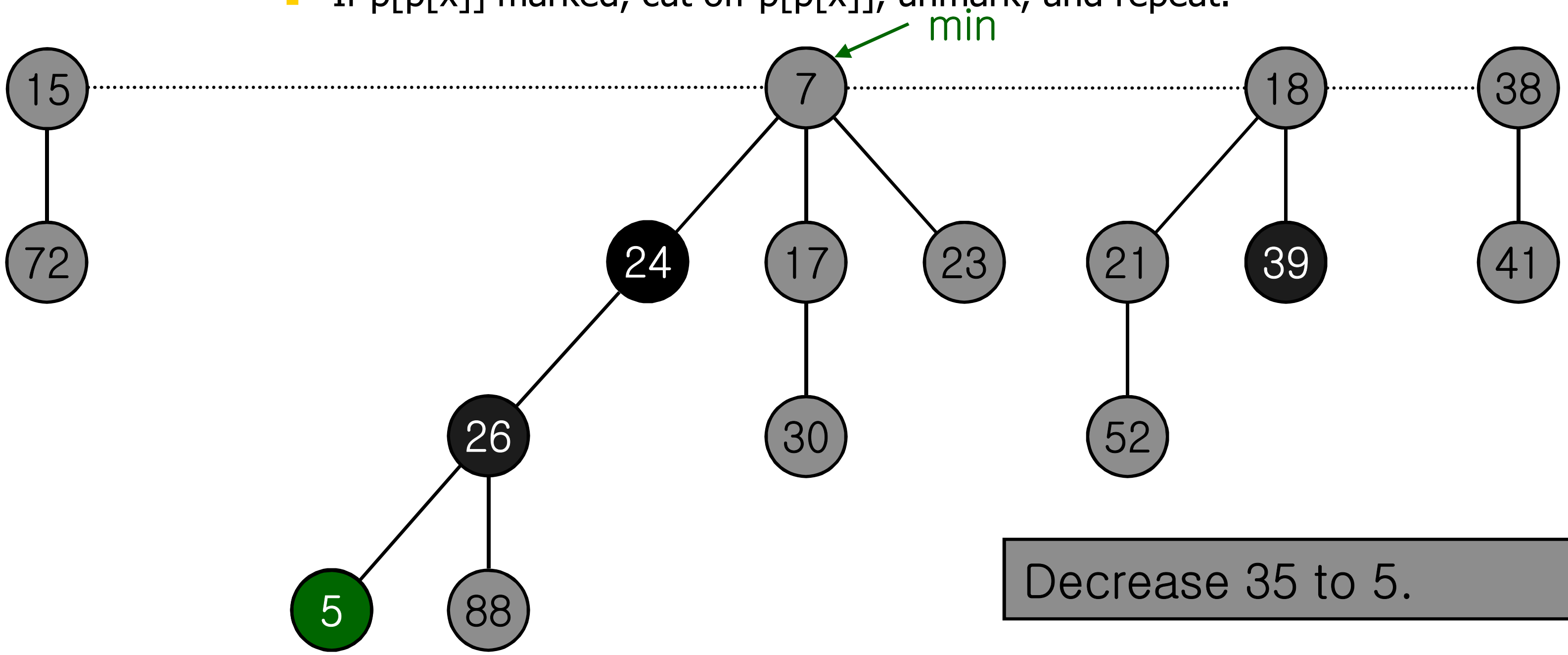
Fibonacci Heaps: Decrease Key

- Decrease key of element x to k.
  - Case 1: parent of x is unmarked.
    - decrease key of x to k
    - cut off link between x and its parent
    - mark parent
    - add tree rooted at x to root list, updating heap min pointer



Fibonacci Heaps: Decrease Key

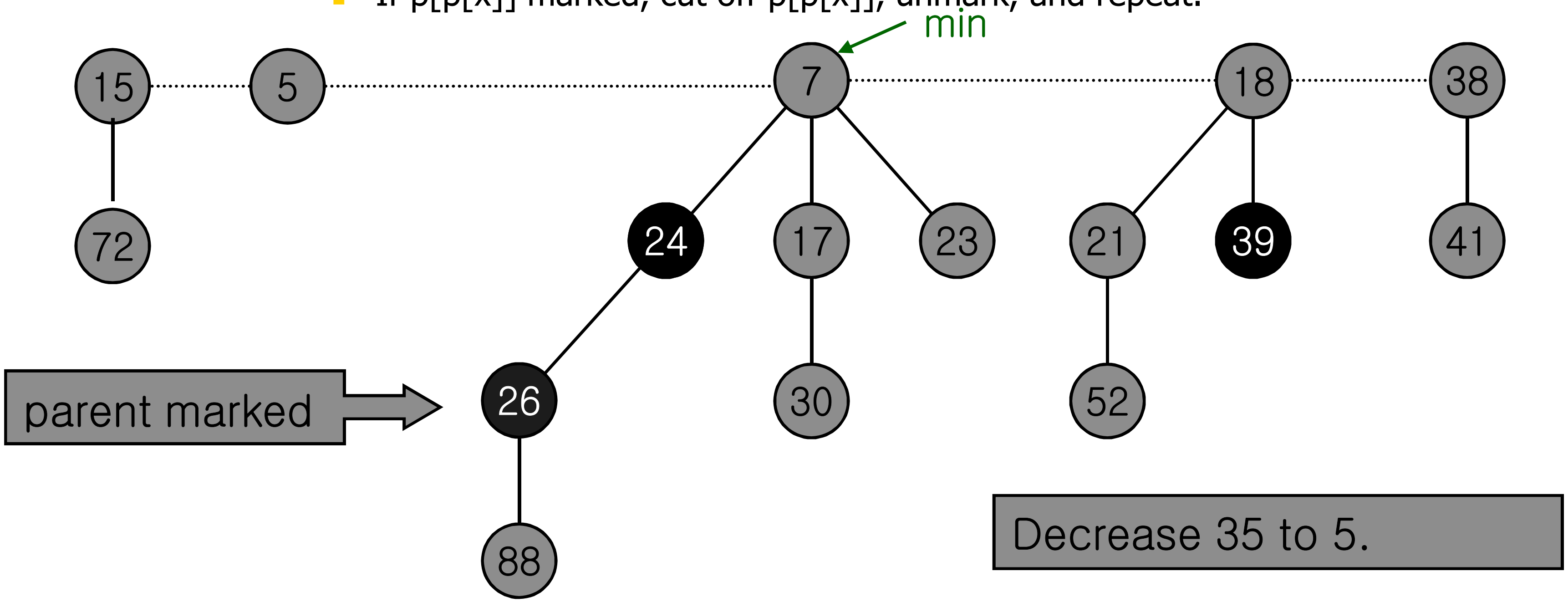
- Decrease key of element x to k.
  - Case 2: parent of x is marked.
    - decrease key of x to k
    - cut off link between x and its parent p[x], and add x to root list
    - cut off link between p[x] and p[p[x]], add p[x] to root list
      - If p[p[x]] unmarked, then mark it.
      - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



Decrease 35 to 5.

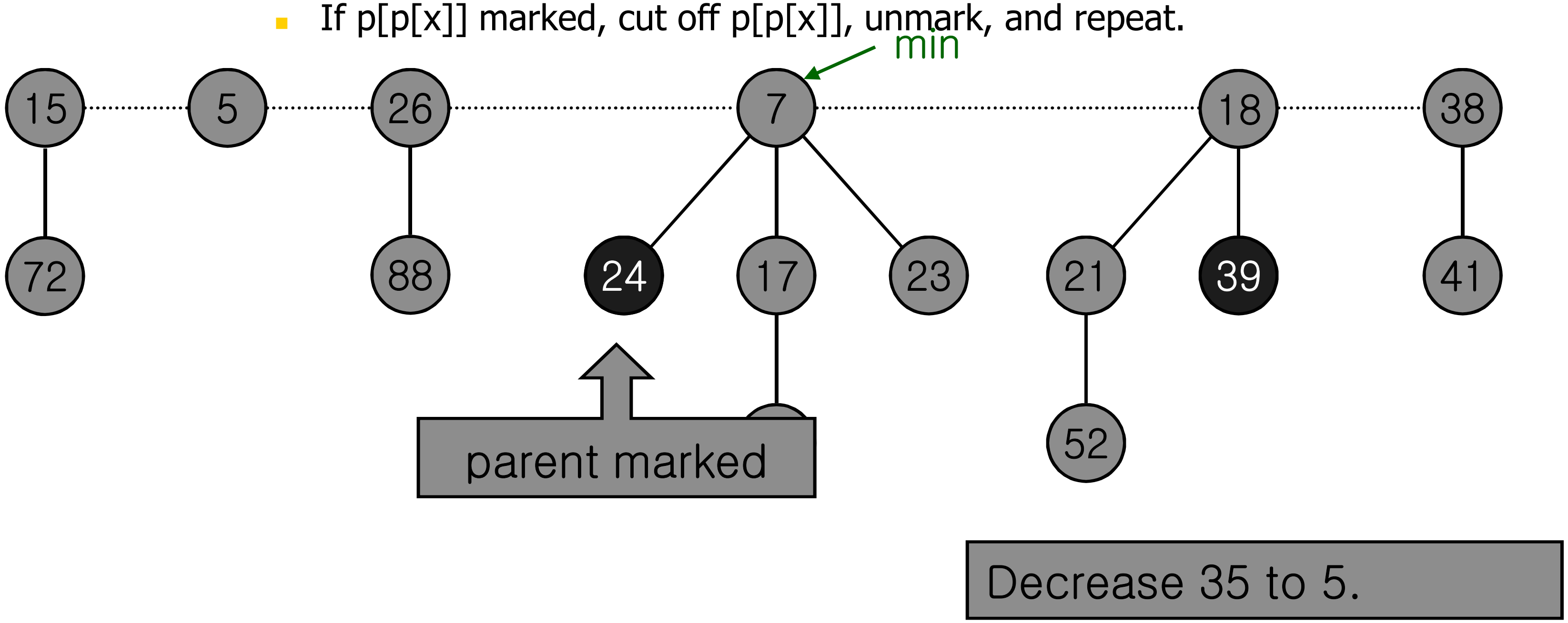
Fibonacci Heaps: Decrease Key

- Decrease key of element x to k.
  - Case 2: parent of x is marked.
    - decrease key of x to k
    - cut off link between x and its parent p[x], and add x to root list
    - cut off link between p[x] and p[p[x]], add p[x] to root list
      - If p[p[x]] unmarked, then mark it.
      - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



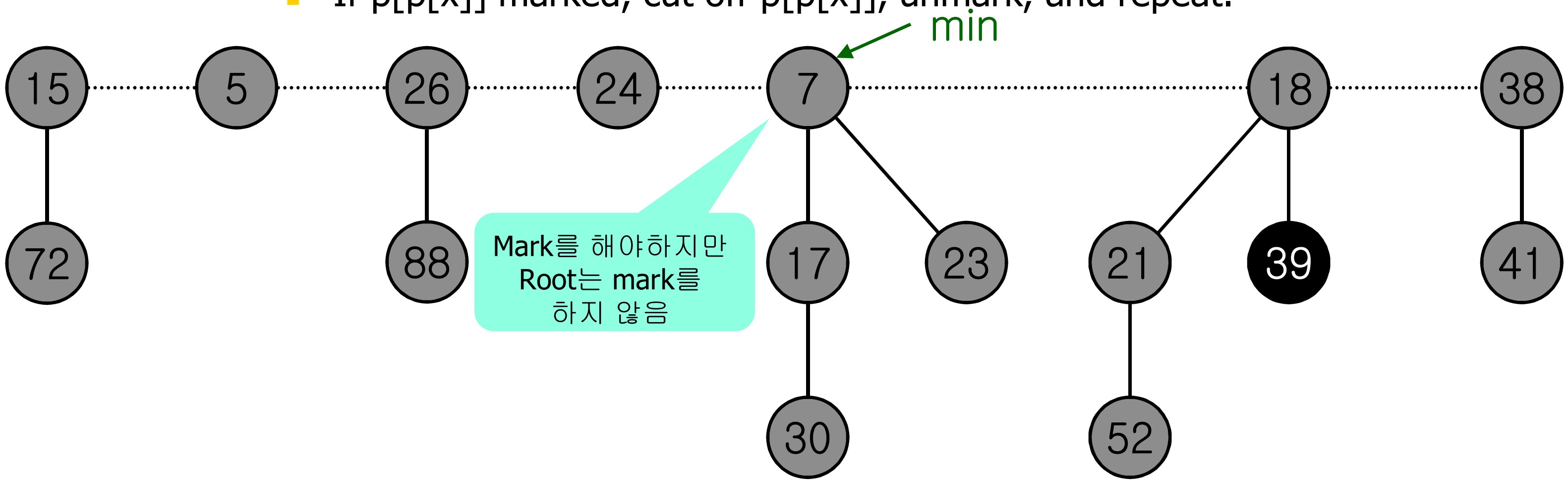
Fibonacci Heaps: Decrease Key

- Decrease key of element x to k.
  - Case 2: parent of x is marked.
    - decrease key of x to k
    - cut off link between x and its parent p[x], and add x to root list
    - cut off link between p[x] and p[p[x]], add p[x] to root list
      - If p[p[x]] unmarked, then mark it.
      - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



Fibonacci Heaps: Decrease Key

- Decrease key of element x to k.
  - Case 2: parent of x is marked.
    - decrease key of x to k
    - cut off link between x and its parent p[x], and add x to root list
    - cut off link between p[x] and p[p[x]], add p[x] to root list
      - If p[p[x]] unmarked, then mark it.
      - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



Decrease 35 to 5.

Fibonacci Heaps: Decrease Key Analysis

- Notation.
  - D(n) = max degree of any node in Fibonacci heap with n nodes.
  - t(H) = number of trees in heap *H*.
  - m(H) = number of marked nodes in heap *H*.
  - Φ(H) = t(H) + 2m(H).
  - c = delete될 노드와 그에 대한 mark된 ancestor의 개수
- Actual cost. c
  - O(1) time for decrease key.
  - O(1) time for each of c cascading cuts, plus reinserting in root list.

Fibonacci Heaps: Decrease Key Analysis

- Notation.
  - D(n) = max degree of any node in Fibonacci heap with n nodes.
  - t(H) = number of trees in heap *H*.
  - m(H) = number of marked nodes in heap *H*.
  - Φ(H) = t(H) + 2m(H).
  - c = delete될 노드와 그에 대한 mark된 ancestor의 개수
- $\Delta\Phi$(H) = - c + 4 = 4 - c
  - Before decreasing a node
    - t(H) + 2m(H).
  - After decreasing a node,
    - t(H') = t(H) + c
    - m(H') = m(H) – c +2
    - Each cascading cut unmarks a node
    - Last cascading cut could potentially mark a node
- Amortized cost. O(1)
  - $\hat{c_i} = c_i + \Phi_i - \Phi_{i-1} = c + (t(H) + c + 2(m(H) - c + 2)) - (t(H) + c + 2m(H))) = 4$

Fibonacci Heaps: Delete

- Delete node x.
  - Decrease key of x to $-\infty$.
  - Delete min element in heap.

- Amortized cost. $O(D(n))$
  - $O(1)$ for decrease-key.
  - $O(D(n))$ for delete-min.

    where $D(n)$ = max degree of any node in Fibonacci heap.