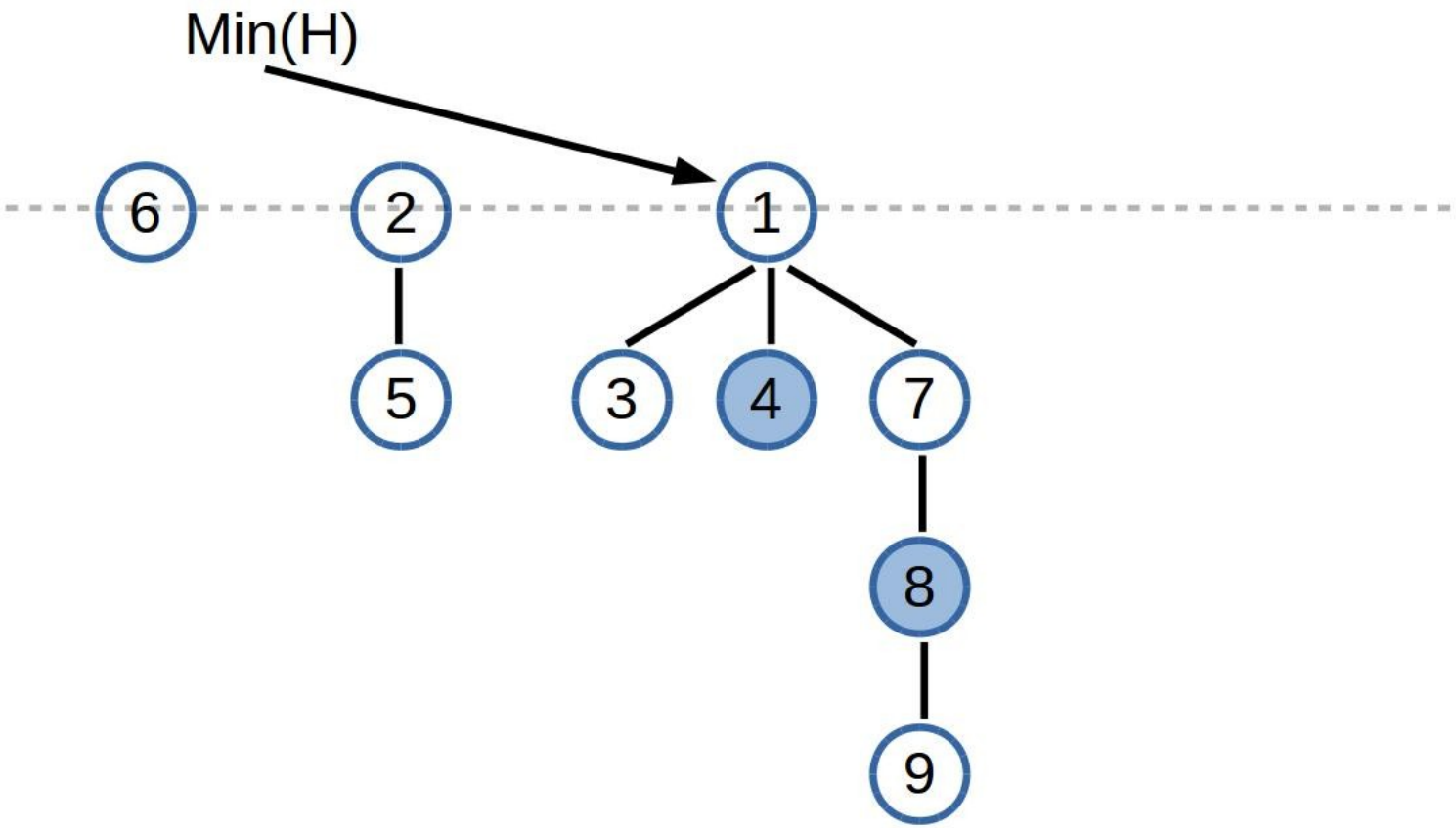
Algorithmen & Komplexität

Fibonacci-Heaps


Min(H)
6
2
1
5
3
4
7
8
9

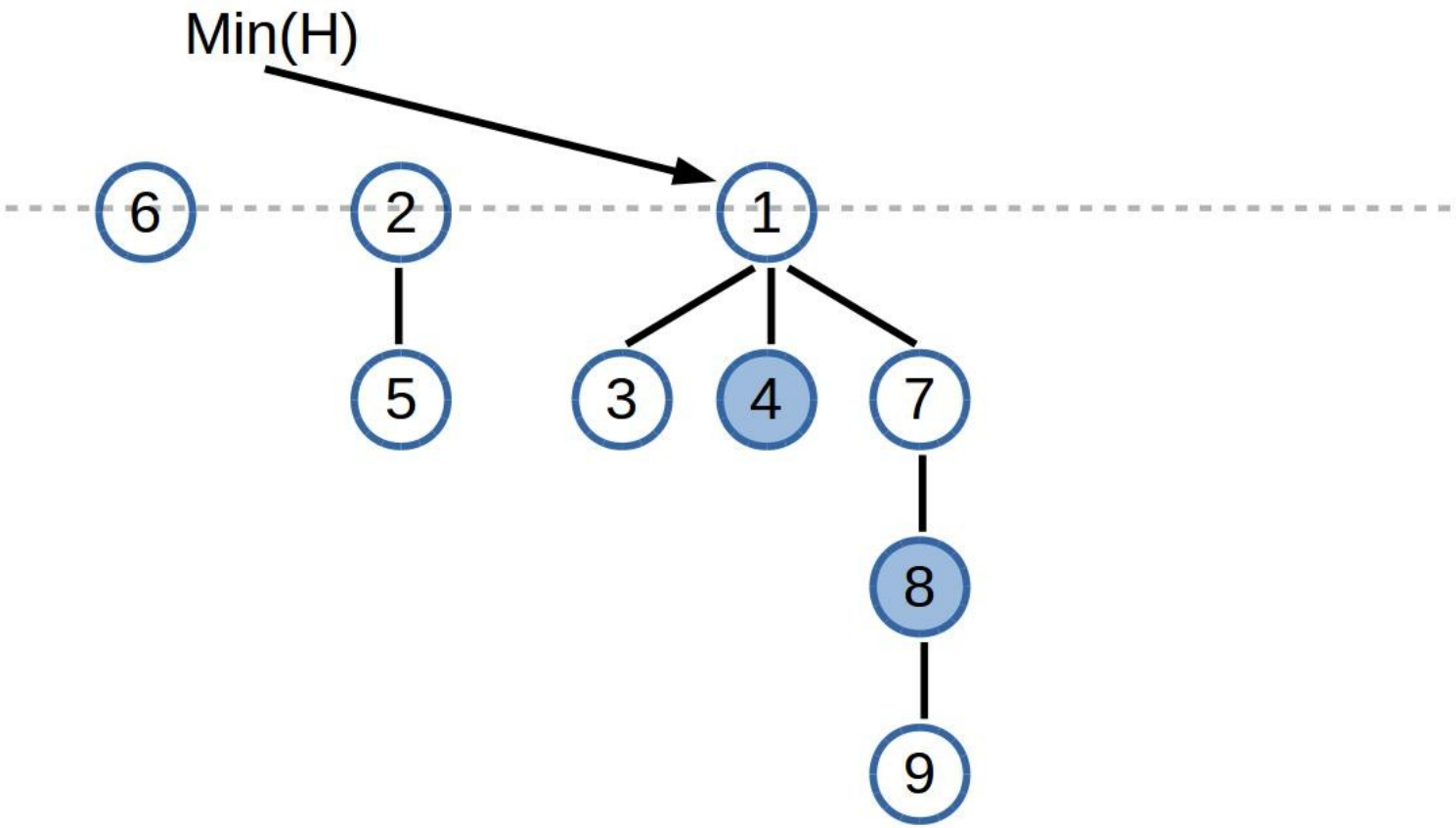
4.4 Fibonacci-Heaps

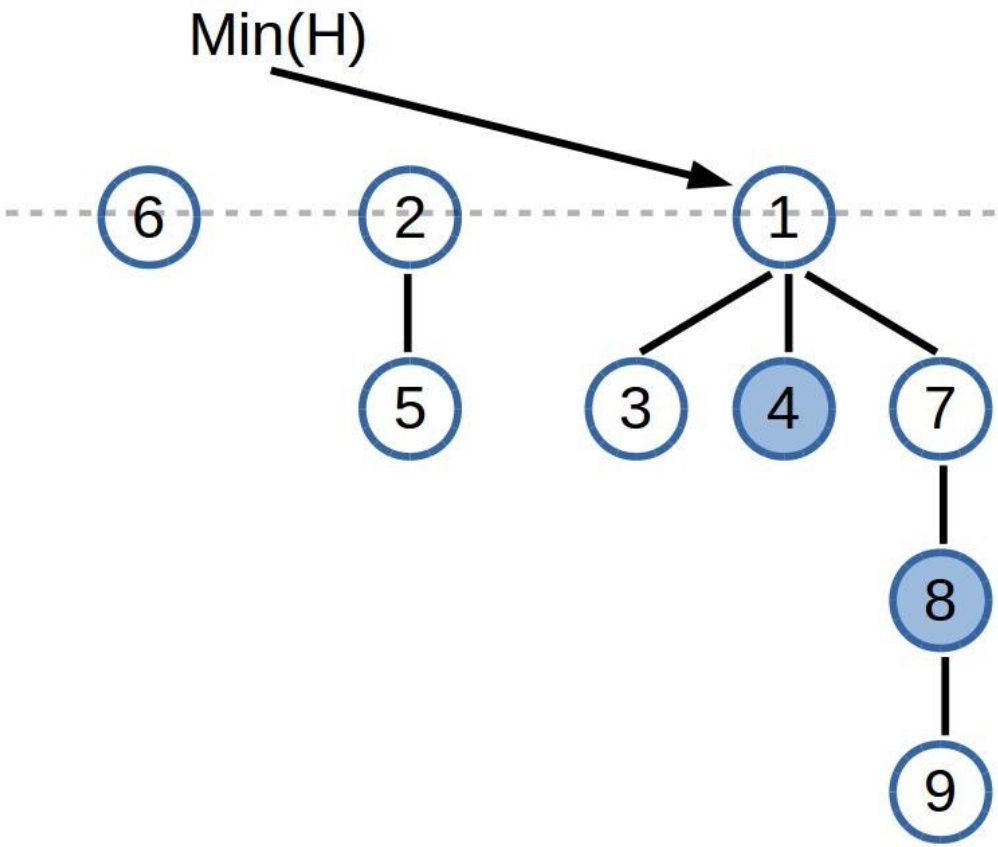
Datenstruktur, die die folgenden drei Operationen unterstützt:

Insert:	Füge ein neues Element hinzu
ExtractMin:	gibt das Element mit dem kleinsten Schlüssel zurück und löscht es aus der Datenstruktur
DecreaseKey:	ändert den Schlüssel eines Elementes zu einem neuen (kleineren) Wert

Fibonacci-Heaps: Eigenschaften

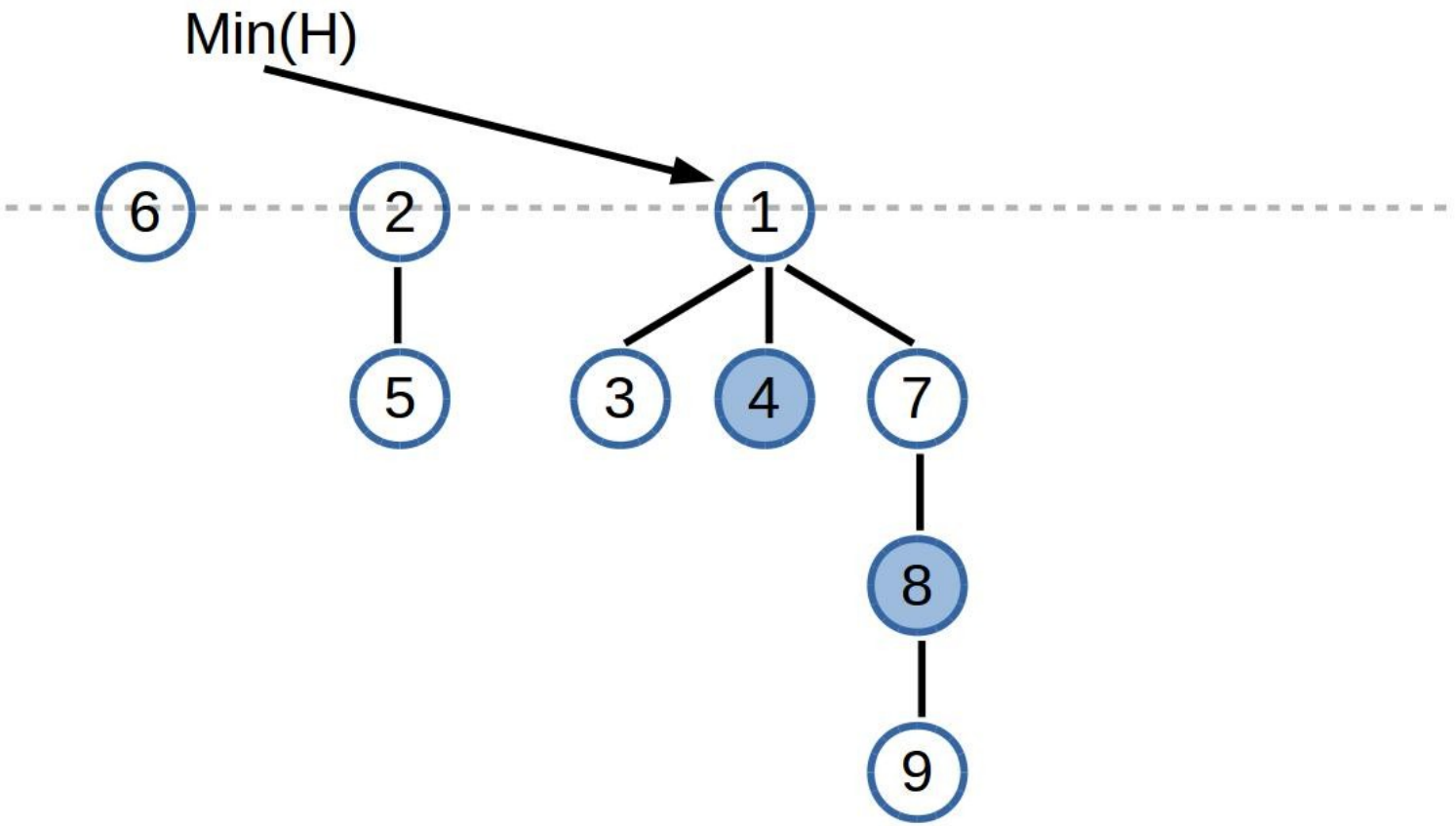
Satz:
Beginnend mit einem leeren Heap können k Operationen `Insert`, `Decrease-Key` und `Extract-Min` in Zeit $\mathcal{O}(k + \ell \cdot \log n)$ ausgeführt werden, wobei ℓ die Anzahl von `Extract-Min` Operationen ist und n die maximale Anzahl von Elementen bezeichnet, die zu irgend einem Zeitpunkt im Heap enthalten sind.


Min(H)
6
2
1
5
3
4
7
8
9

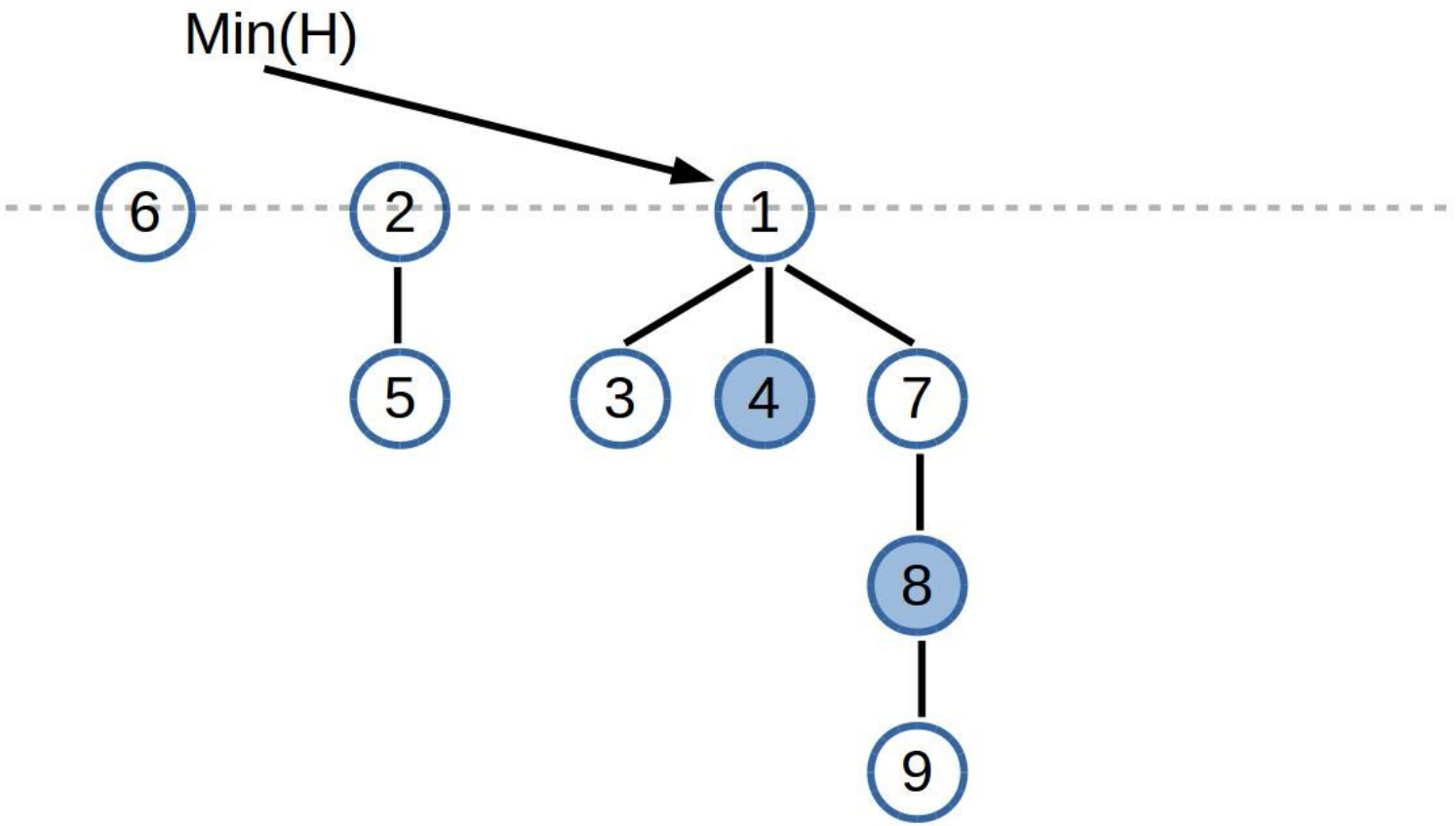


Knoten x enthält:

- key[x]
- parent[x]
- child[x]
- left[x]
- right[x]
- rank[x]
- marked[x]



Min(H) zeigt auf den Knoten mit dem kleinsten Schlüssel.



$key[x] \leq key[child[x]]$

Unterstützte Operationen

- Insert(H,x)
- Extract-Min(H)
- Decrease-Key(H,x,k)

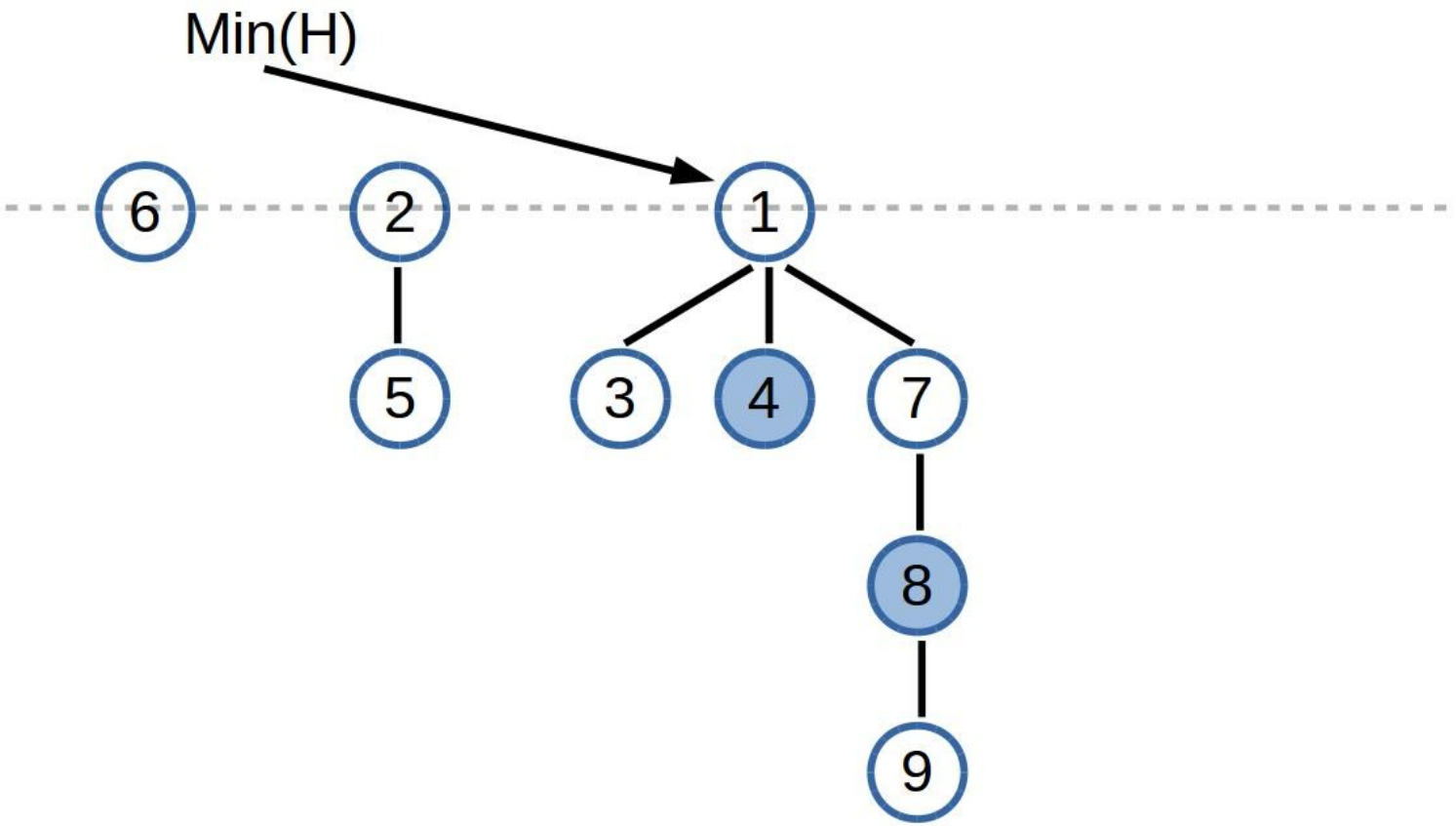
Insert

Insert(H,x):

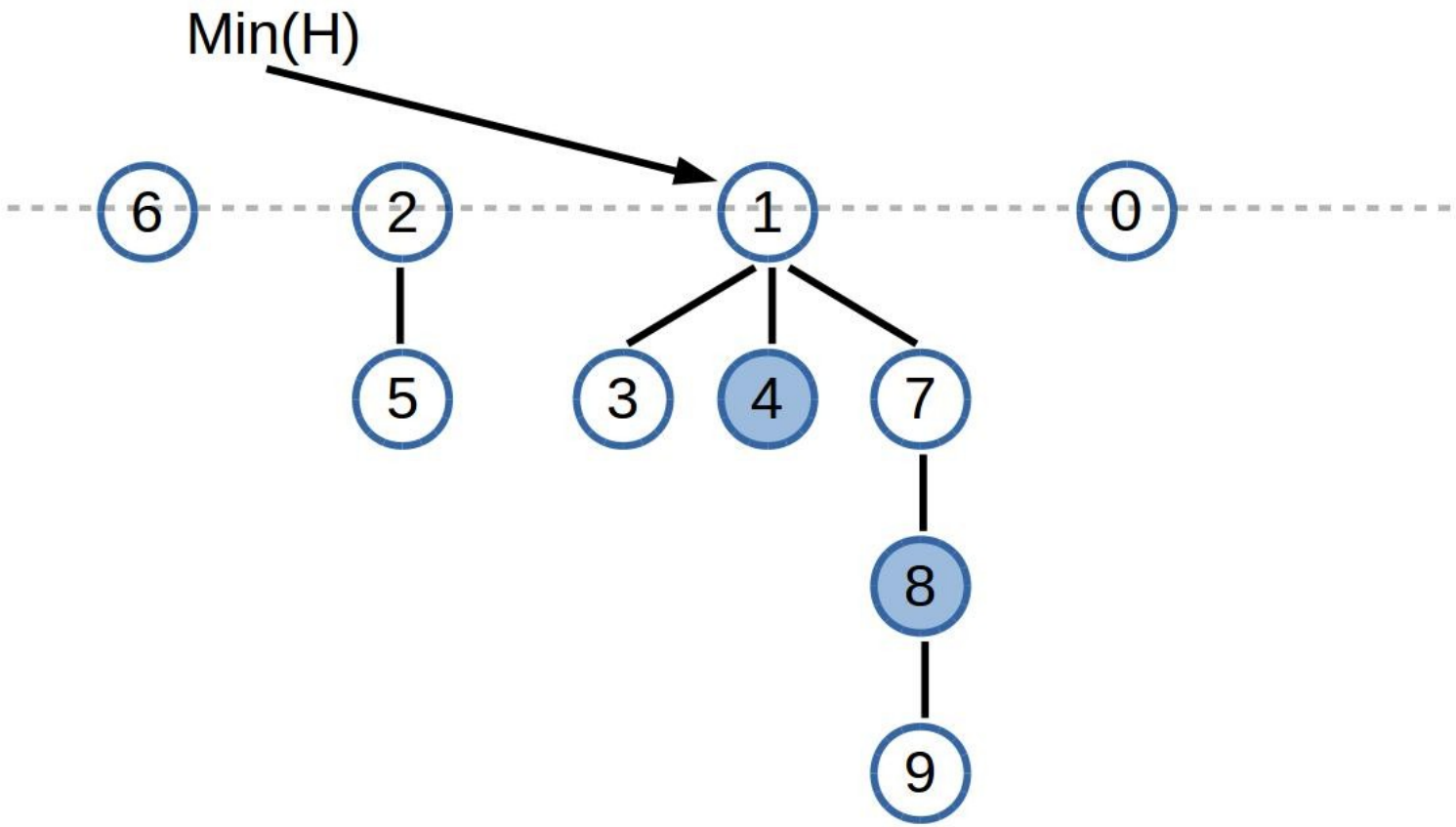
- Knoten x wird in die Wurzelliste eingefügt.
- Zeiger min[H] wird aktualisiert.

Insert(H,0)

- 0 in die Wurzelliste einfügen !

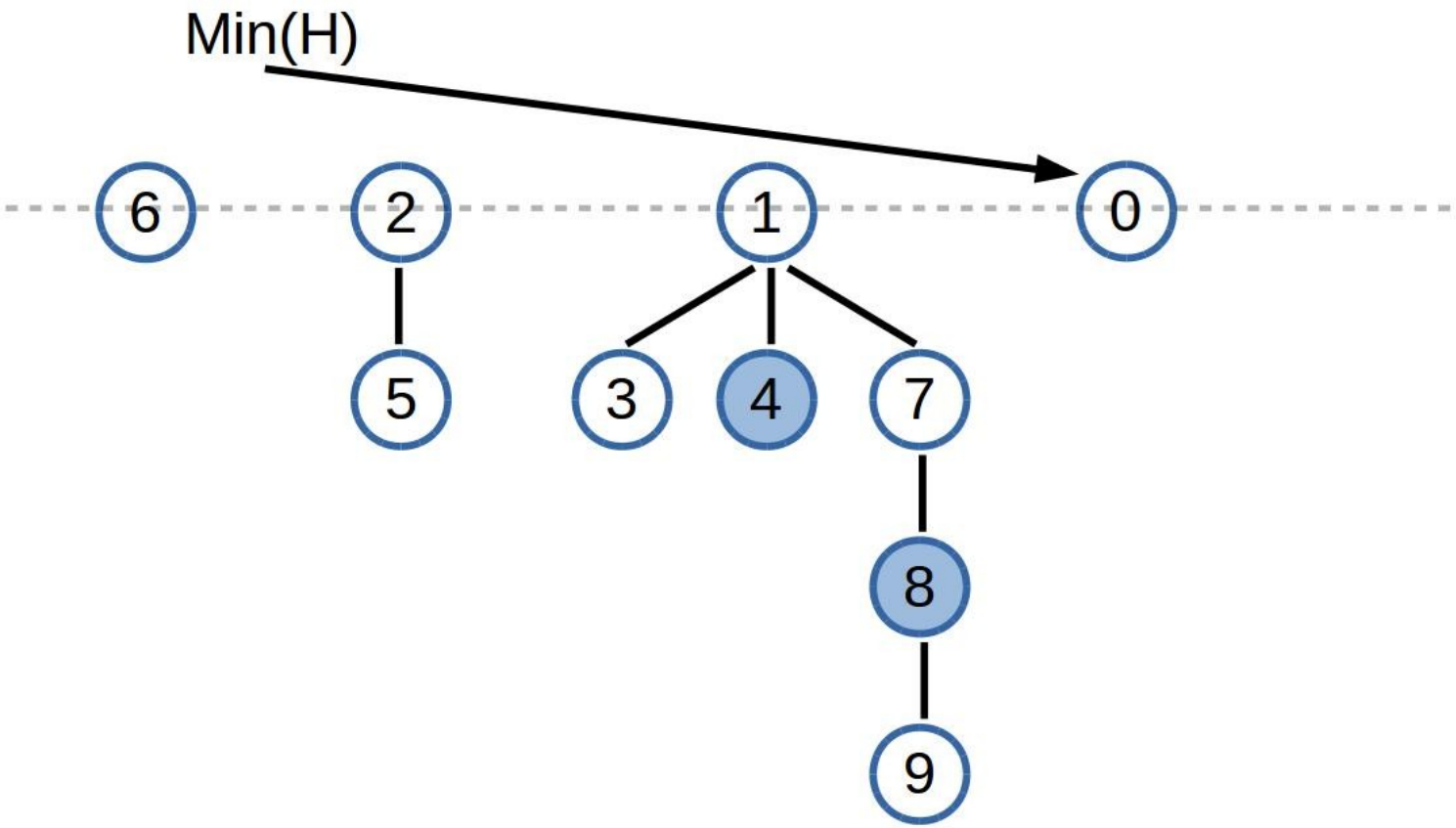


Insert(H,0)



Insert(H,0)

- Aktualisiere Min(H) !



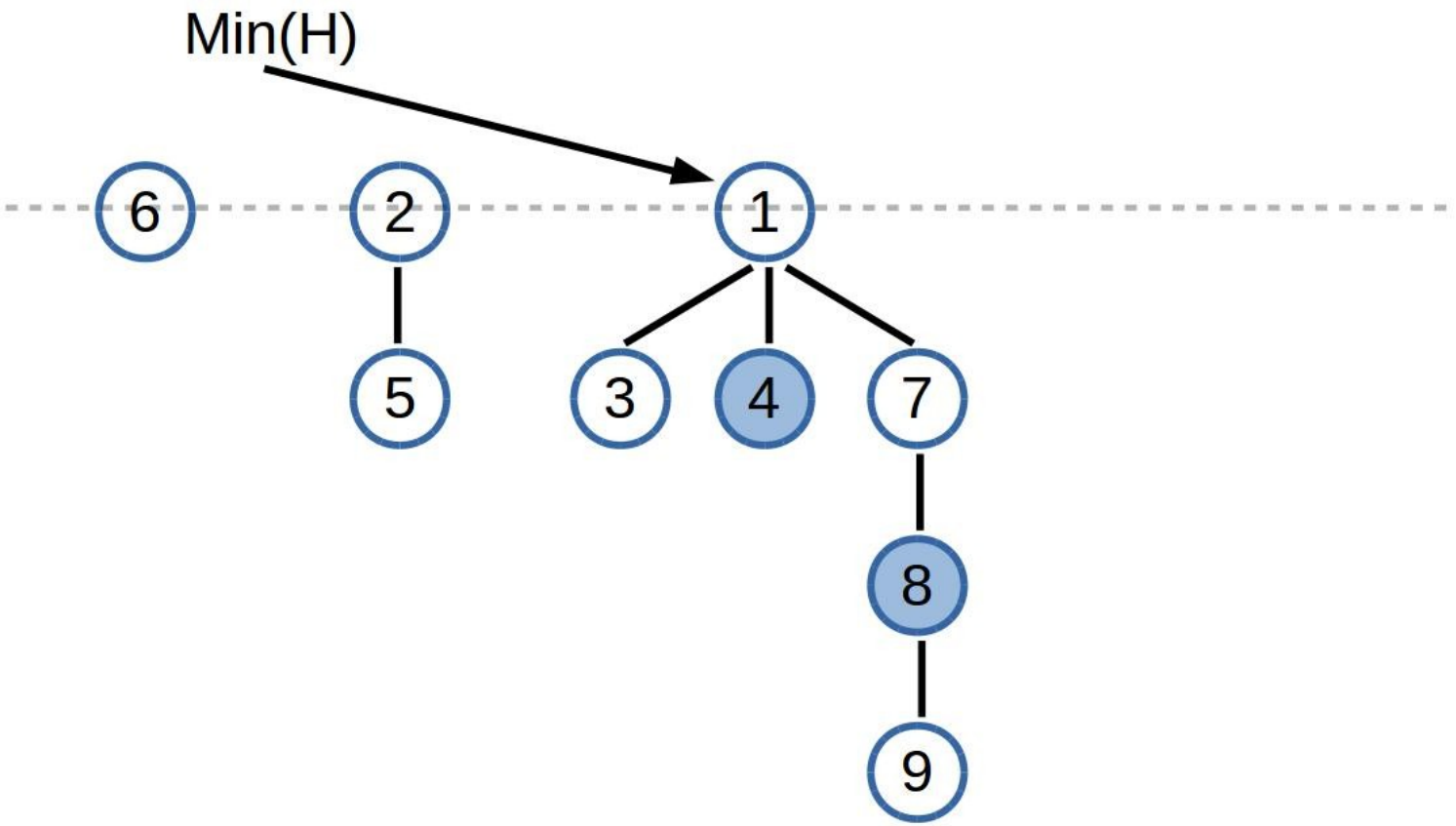
ExtractMin

ExtractMin(H):

- Der Knoten auf den min[H] zeigt wird entfernt.
- Alle Kinder dieses Knotens werden in die Wurzelliste eingefügt.
- Datenstruktur wird mit **Consolidate(H)** aufgeräumt:

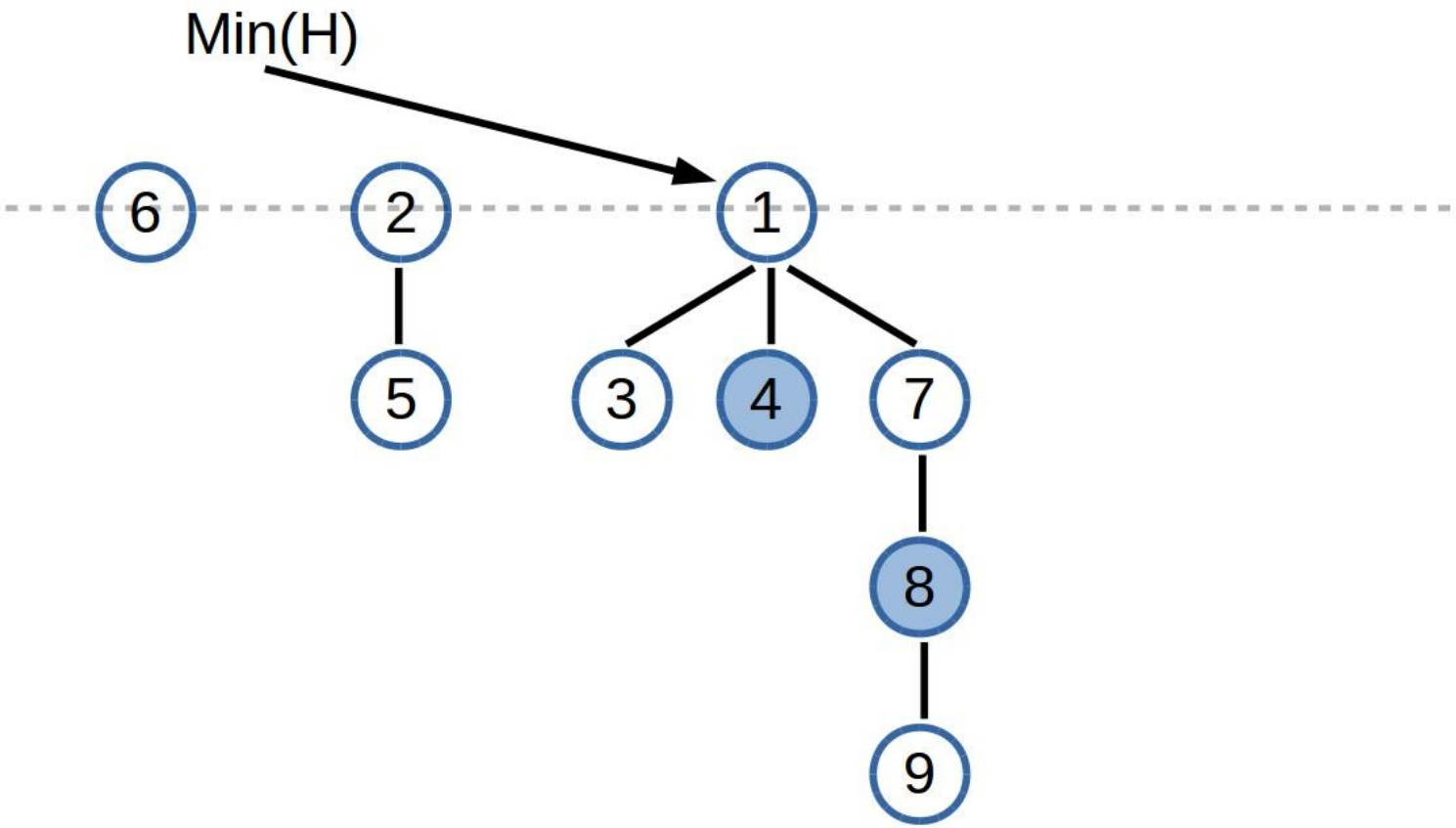
  → Wurzelliste soll keine zwei Knoten mit dem gleichen Rang (Rang = Anzahl Kinder) enthalten

Extract-Min(H)



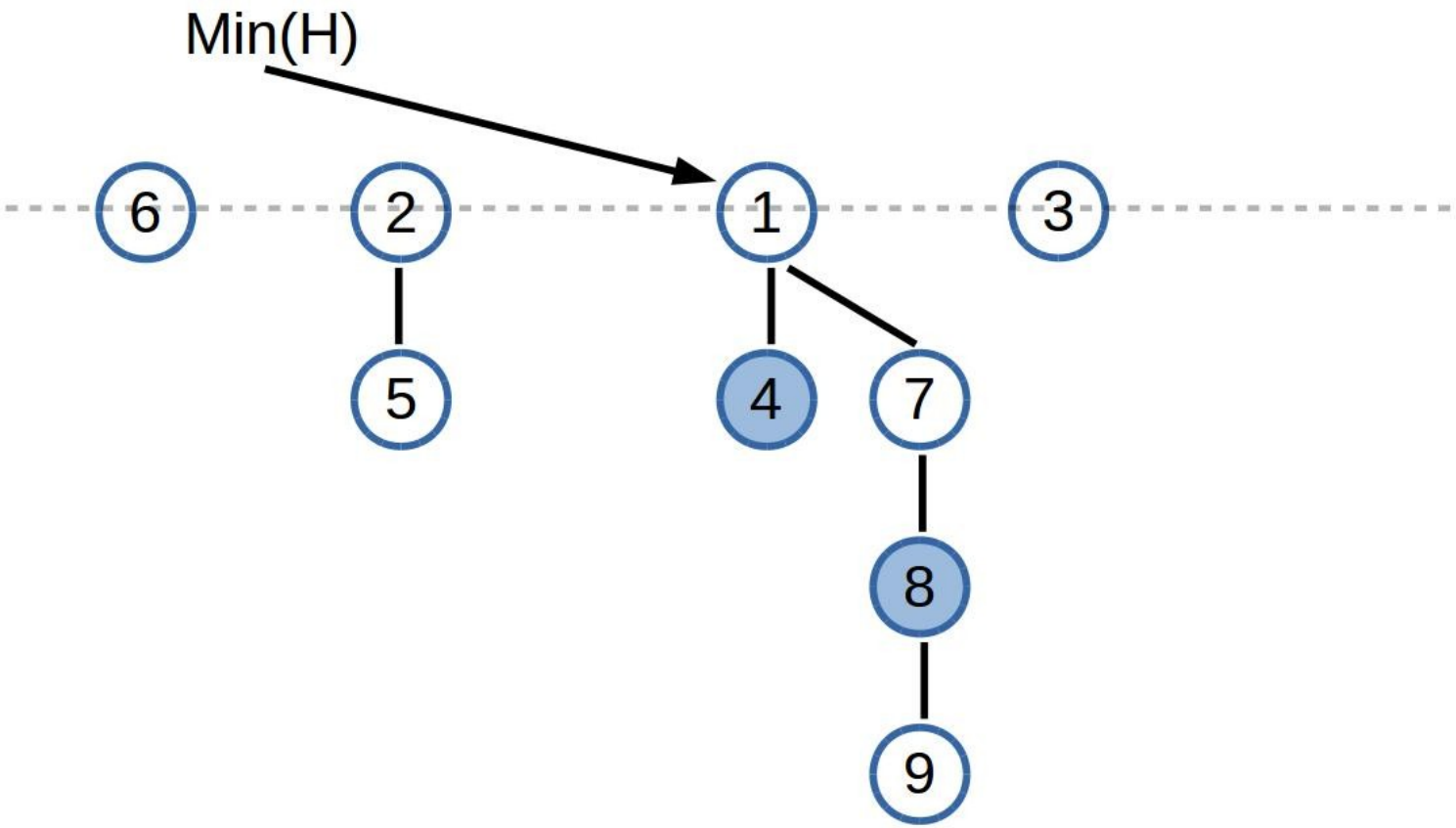
Extract-Min(H)

-Füge alle Kinder in die Wurzelliste ein!



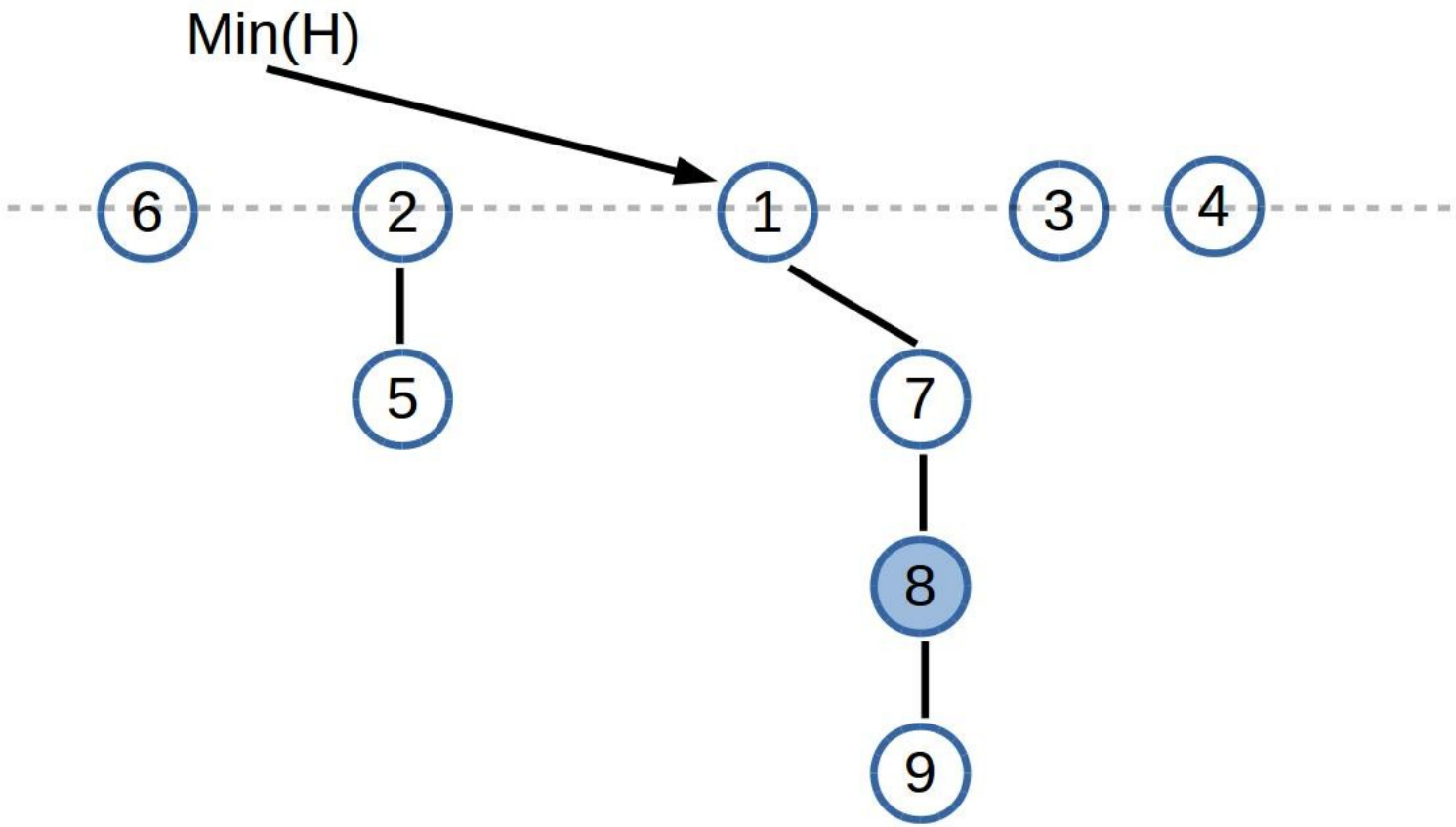
Extract-Min(H)

-Füge alle Kinder in die Wurzelliste ein!



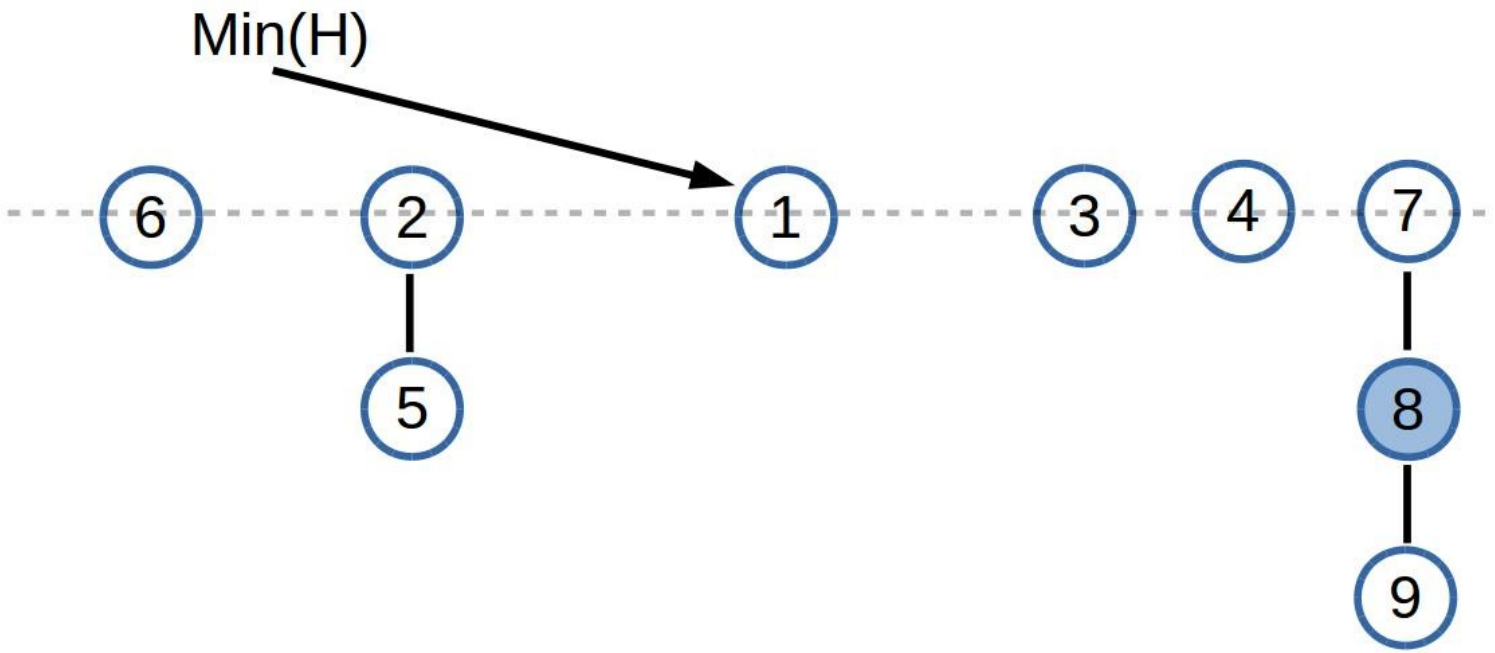
Extract-Min(H)

-Füge alle Kinder in die Wurzelliste ein!

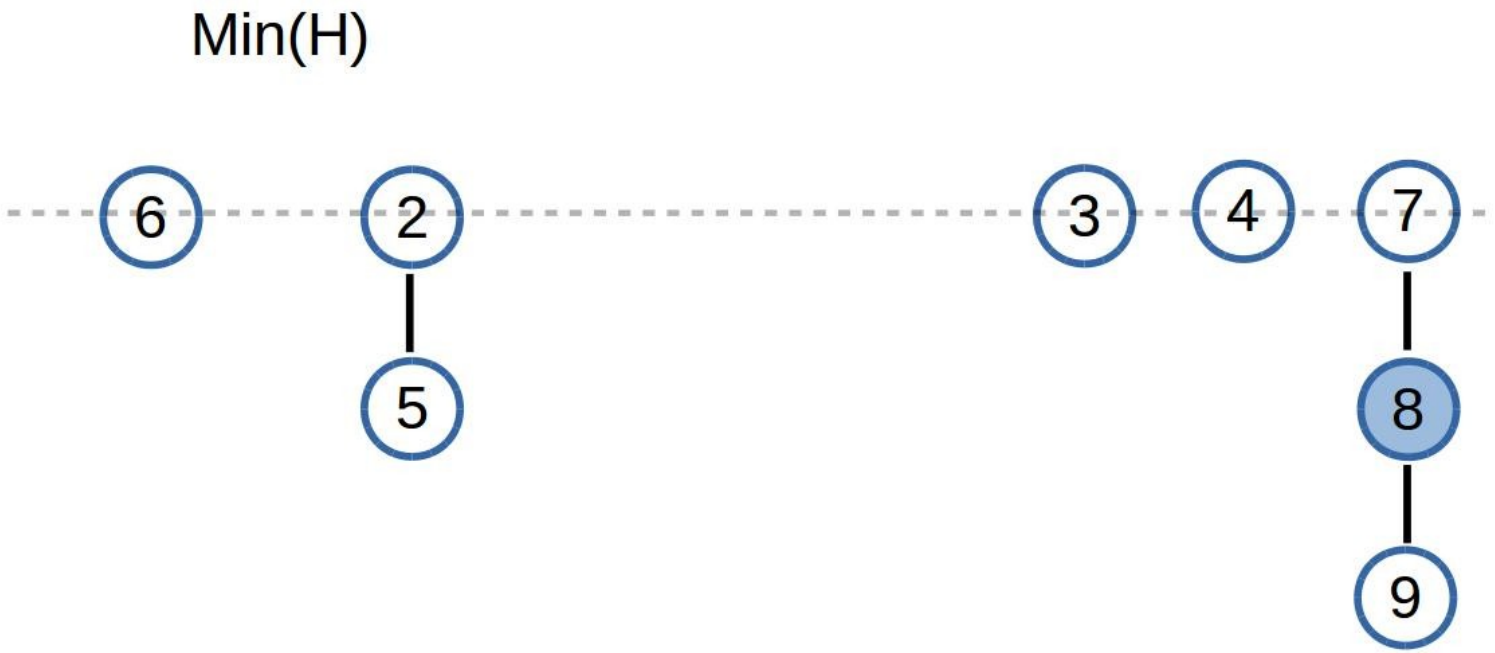


Extract-Min(H)

-Entferne Min(H) !

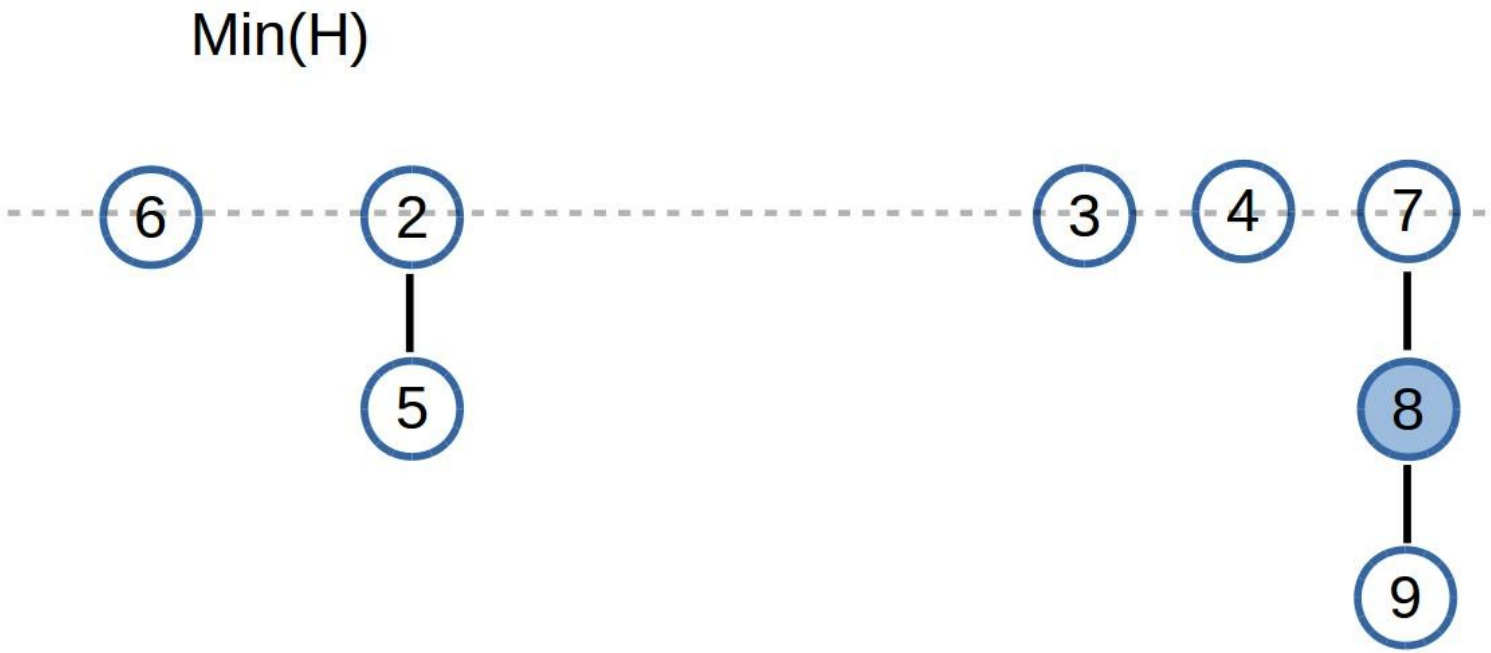


Extract-Min(H) -Consolidate !



Consolidate(H)

-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat



Consolidate(H)

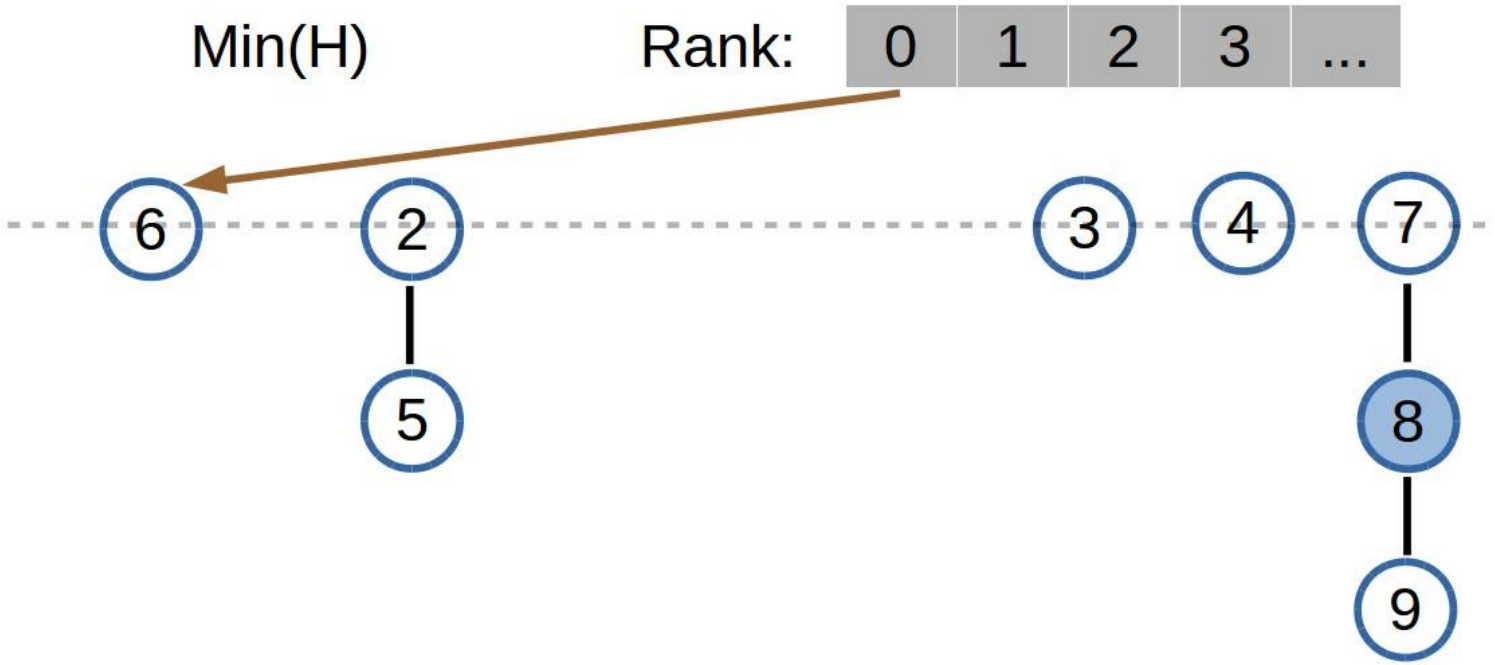
-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat

Min(H)

Rank:	0	1	2	3	...

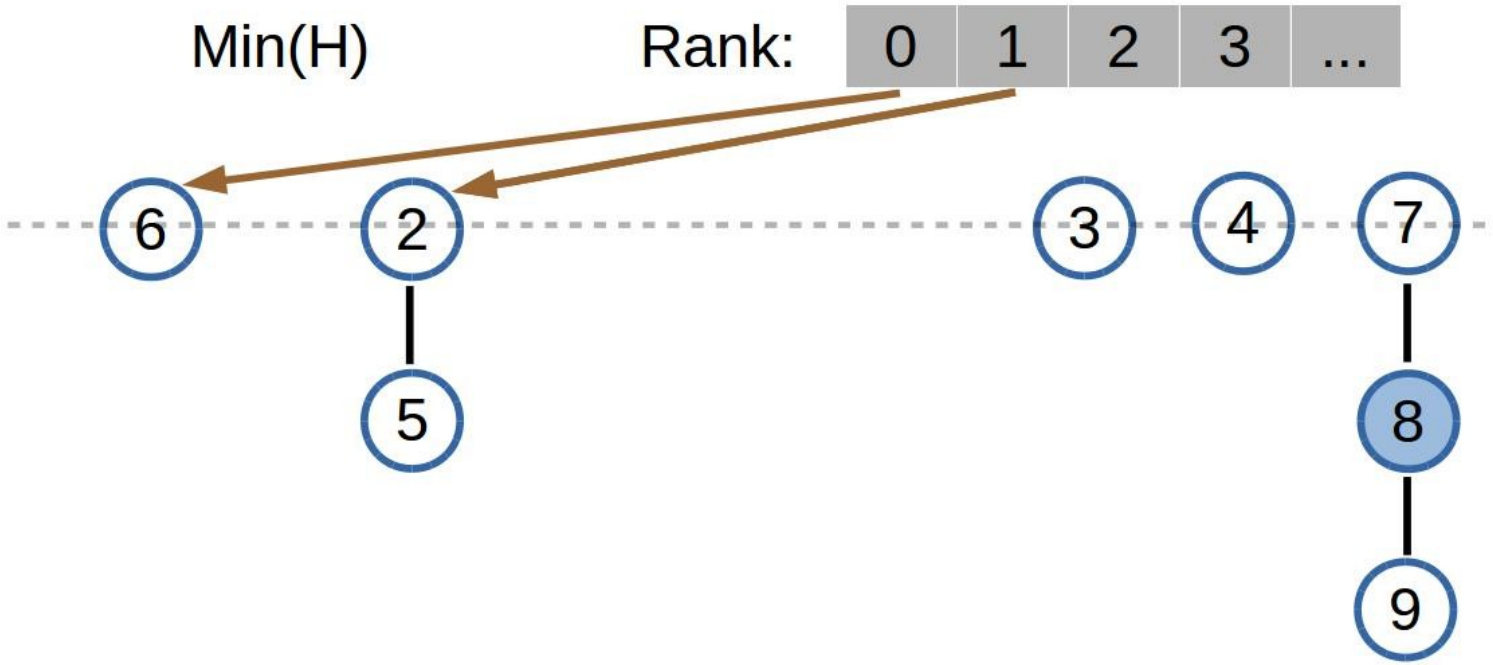
Consolidate(H)

-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat



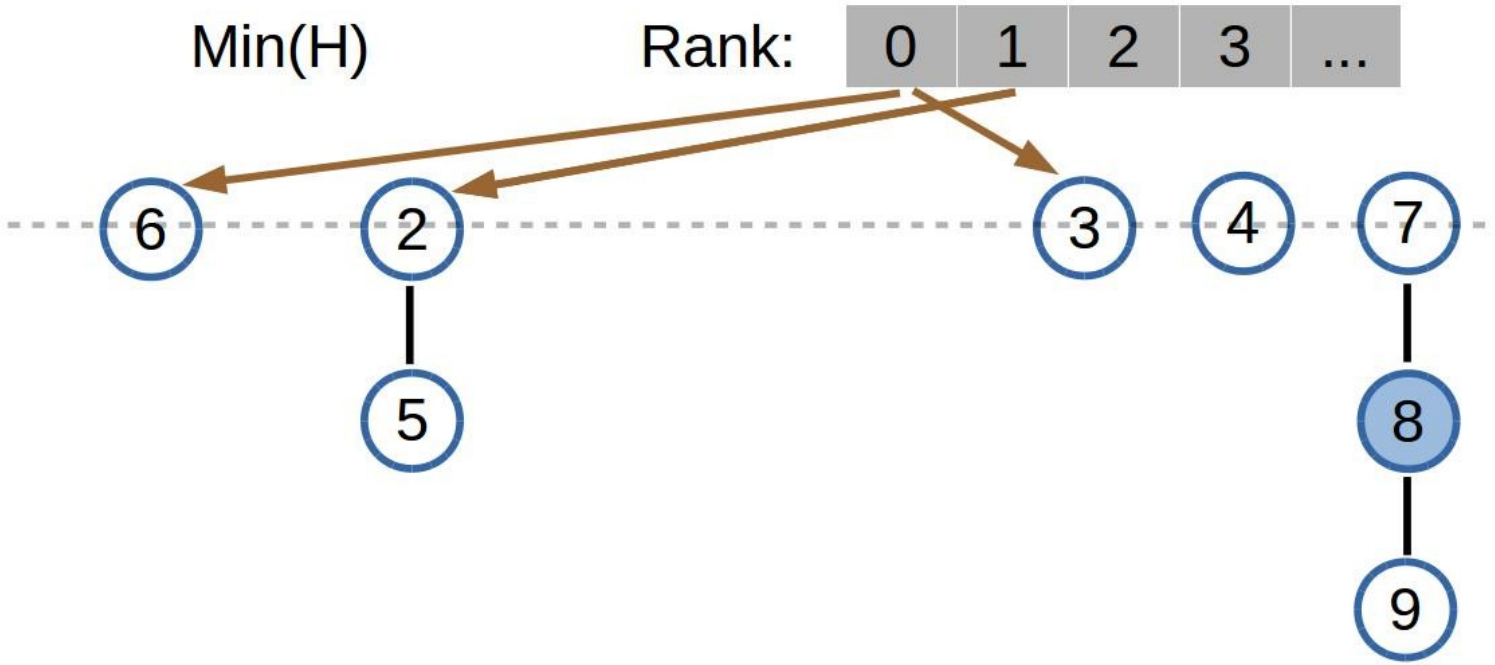
Consolidate(H)

-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat



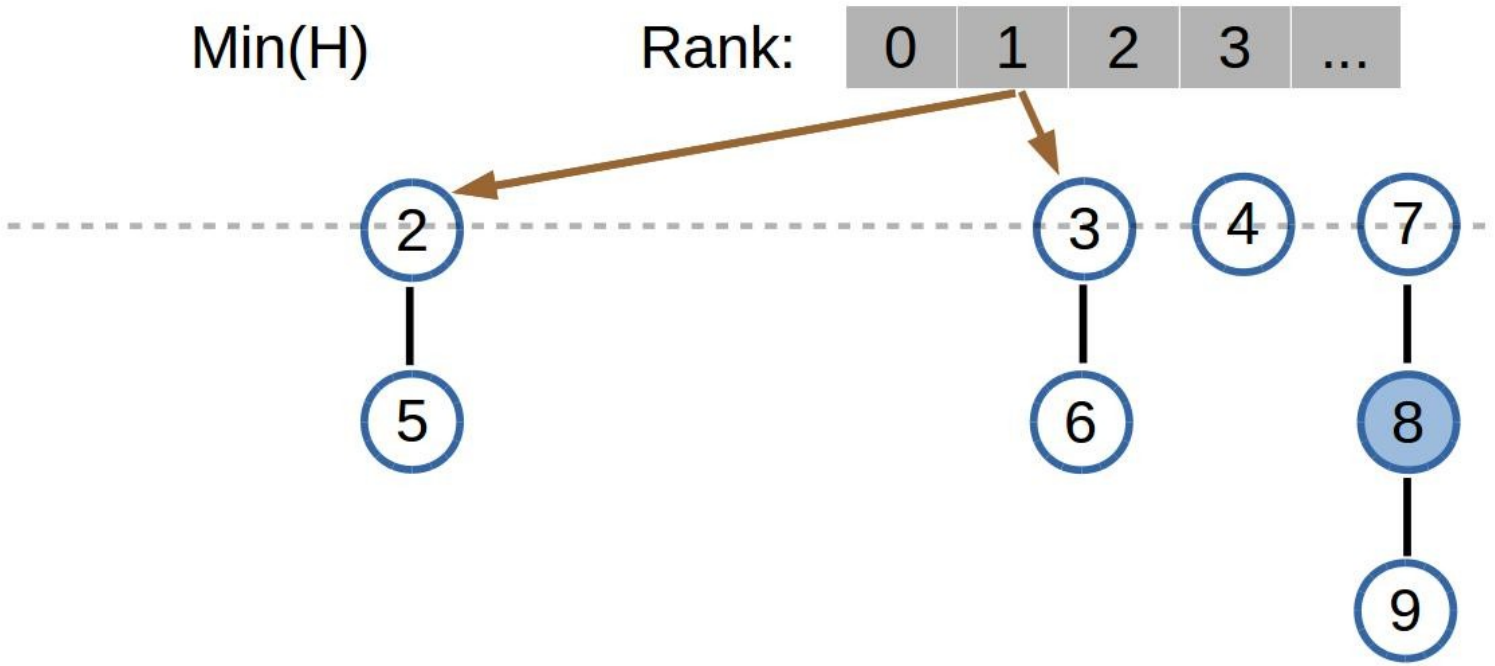
Consolidate(H)

-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat



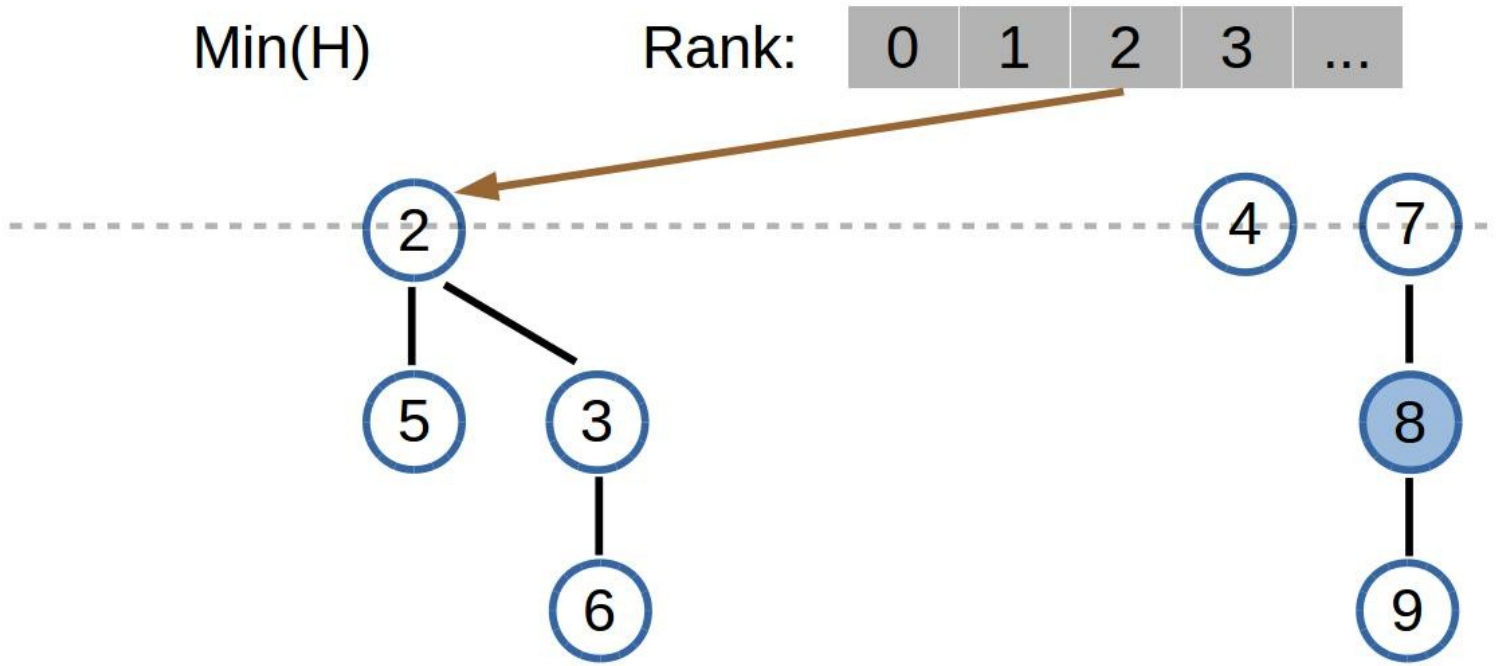
Consolidate(H)

-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat



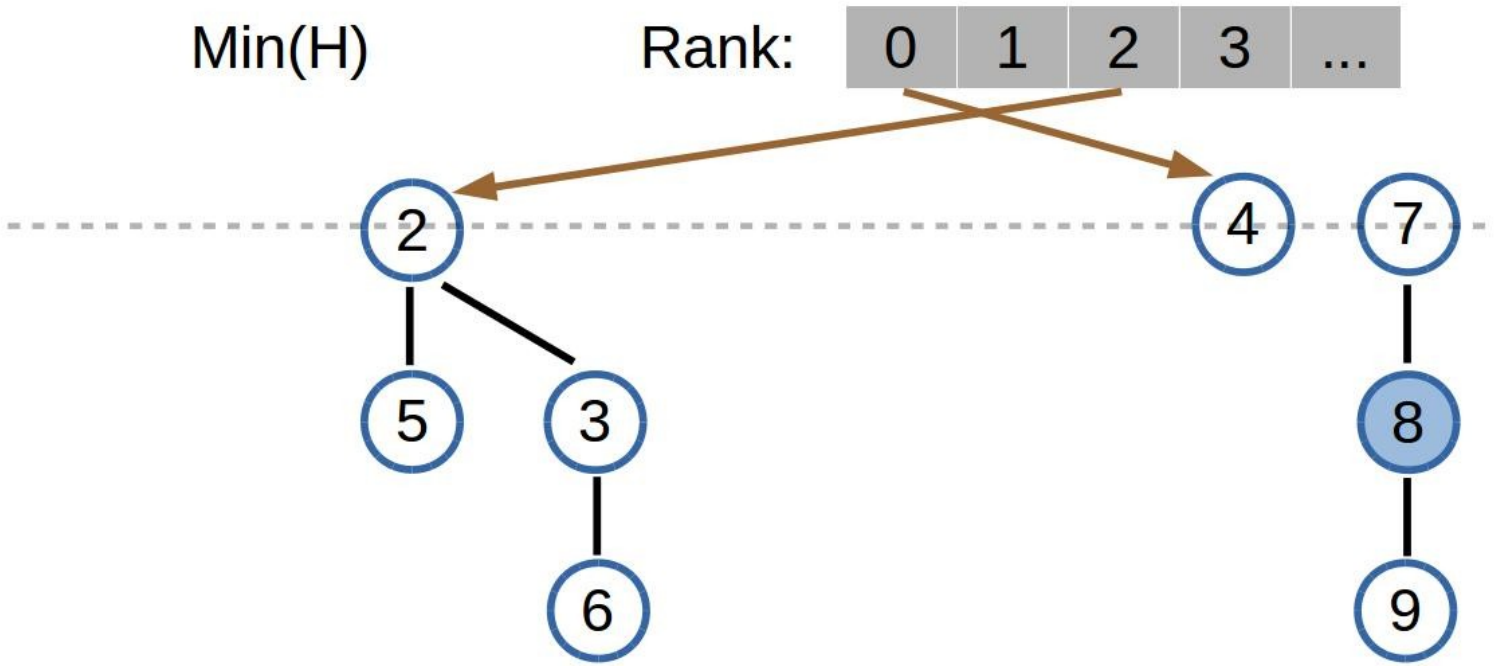
Consolidate(H)

-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat



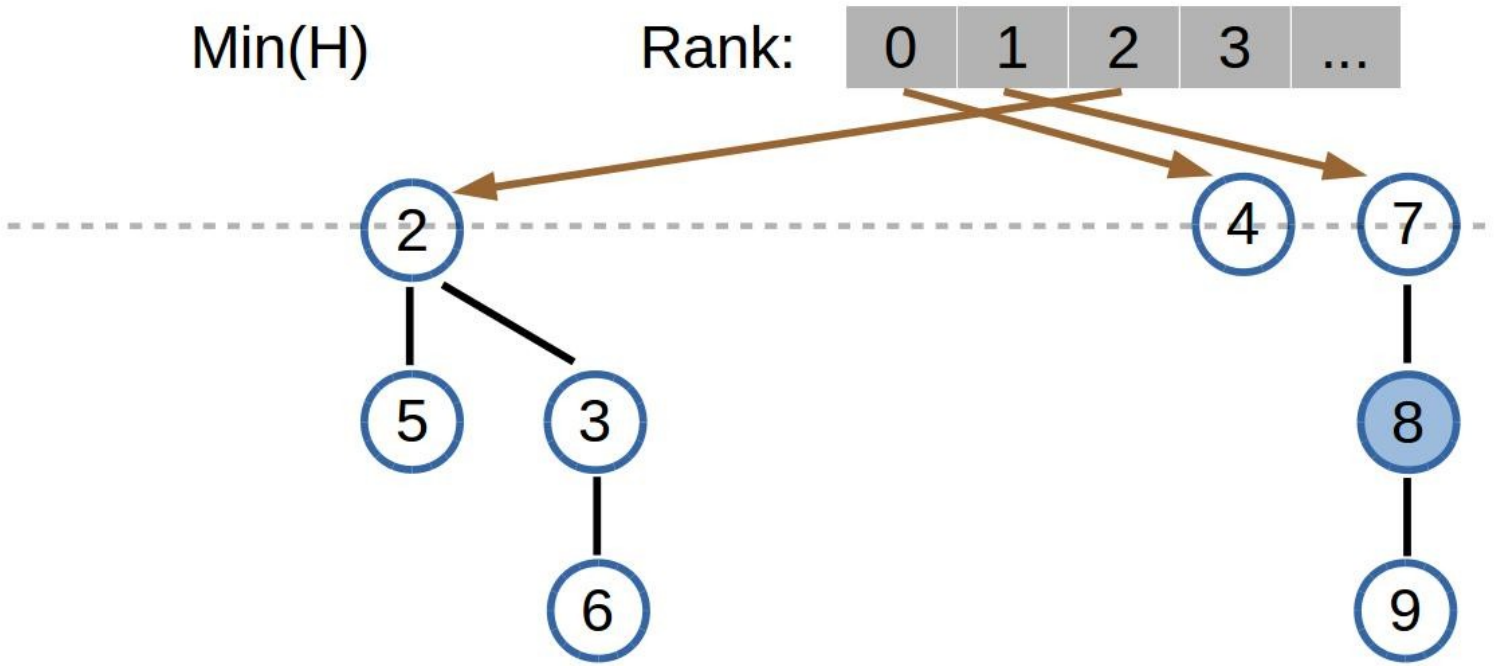
Consolidate(H)

-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat

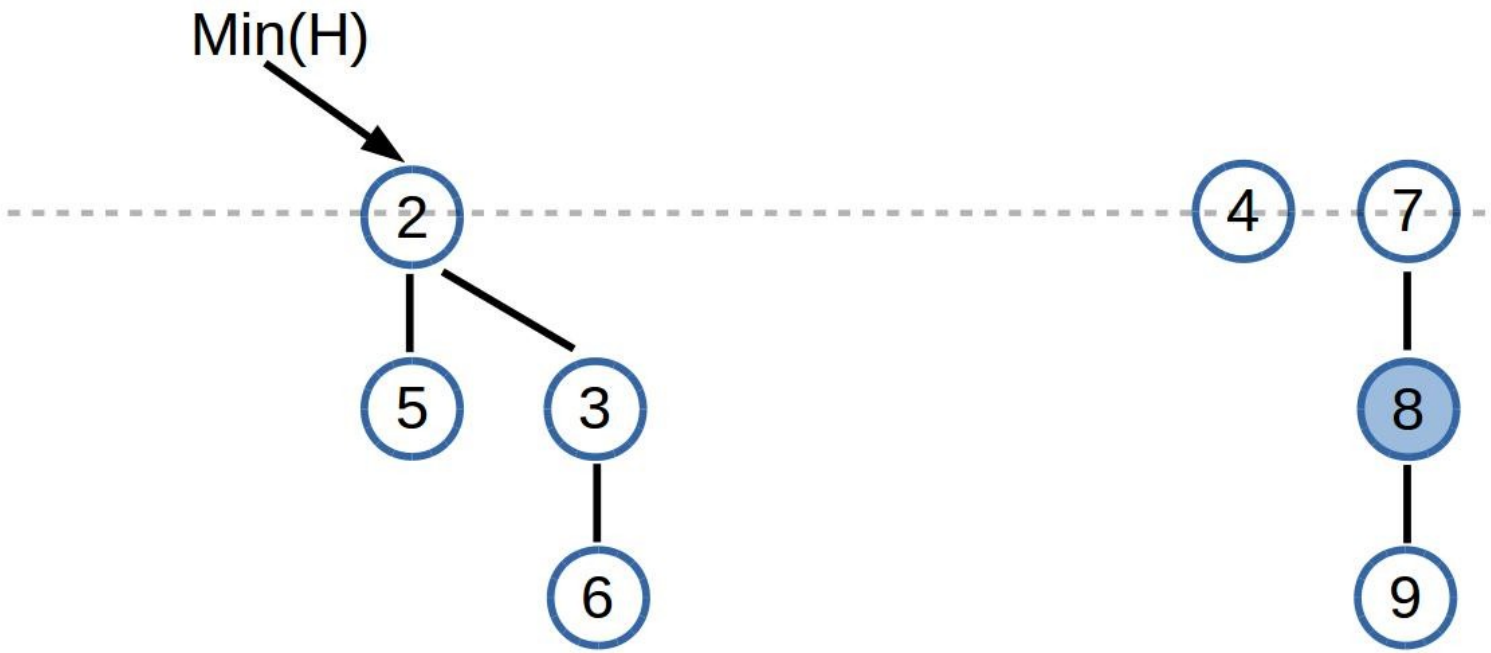


Consolidate(H)

-Vereinige Bäume, bis es in der Wurzelliste keine zwei Knoten von gleichem Rank hat



Consolidate(H) -Aktualisiere Min(H) !



Decrease-Key

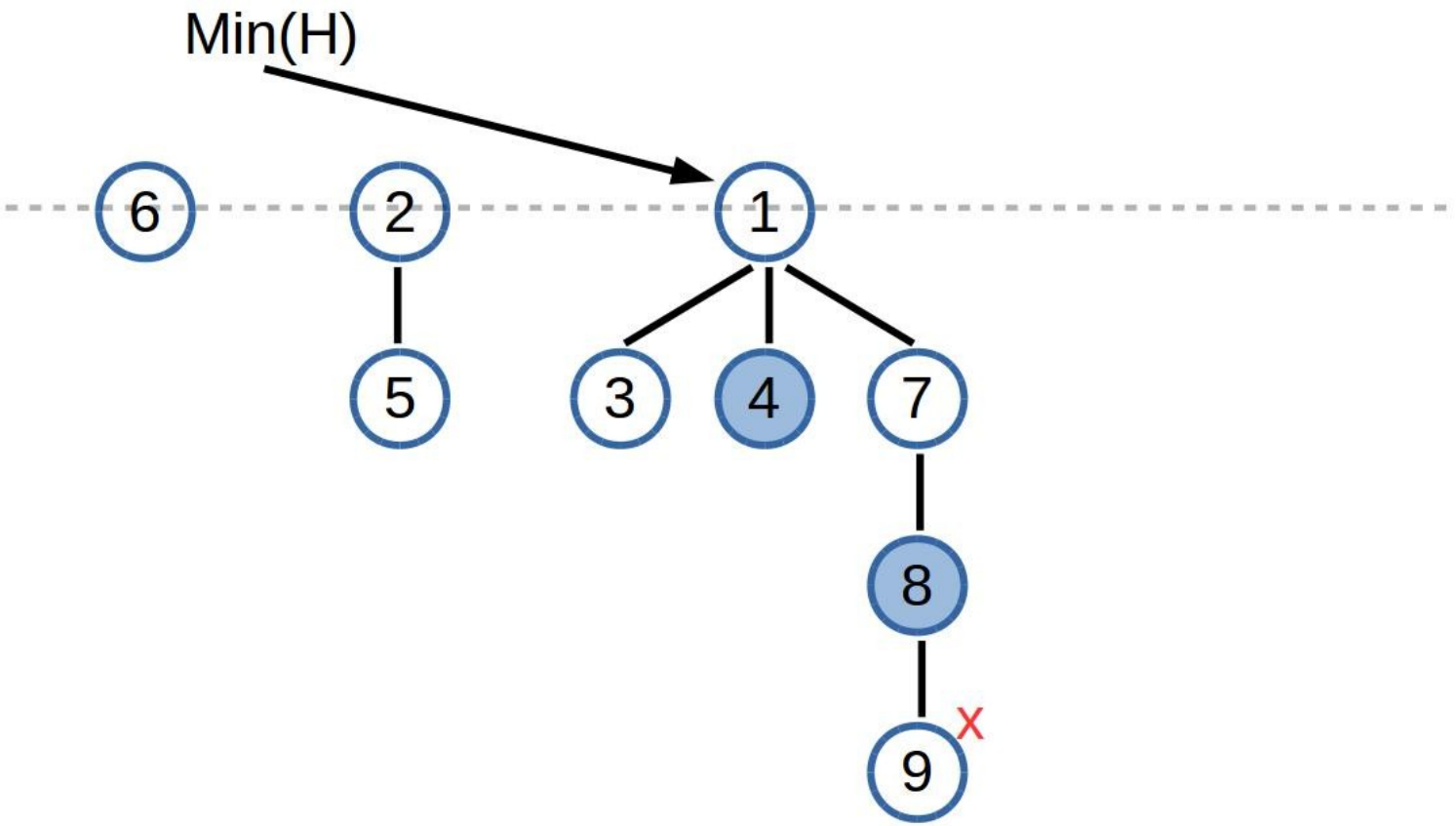
Decrease-Key(H,x,k):

- Setze key[**x**]=k. Falls **x** nun einen kleineneren Schlüssel als sein Elternknoten hat, mache das folgende:
- Füge **x** mit dem neuen Wert in die Wurzelliste ein.
- Betrachte Elternknoten **y** von **x**:

  - Falls **y** nicht markiert, markiere **y**.
  - Falls **y** markiert, schneide auch diesen Knoten von seinem Elternknoten ab und füge ihn in die Wurzelliste ein, usw.

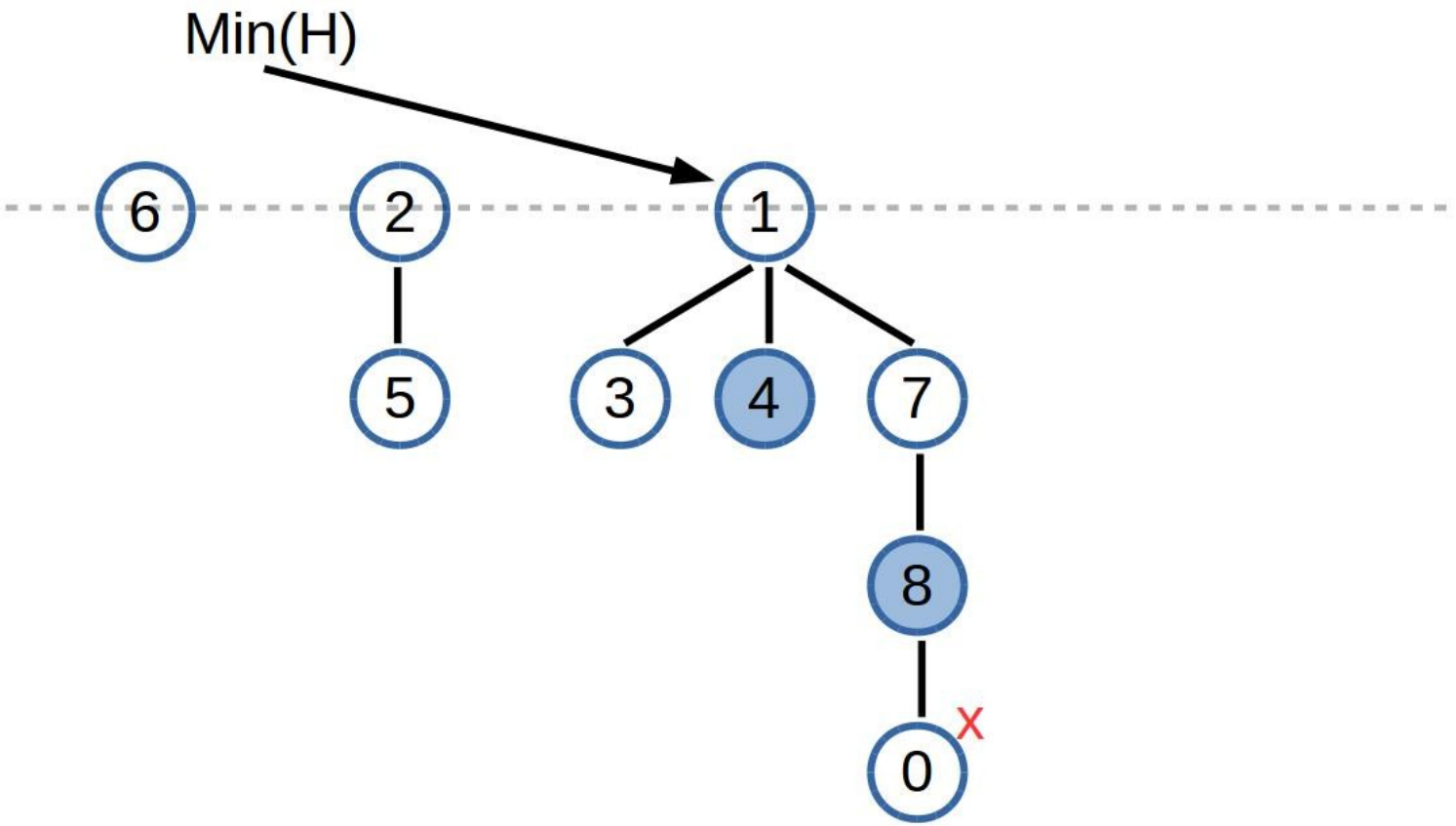
  → Cascading-Cuts

Decrease-Key(H,x,0)



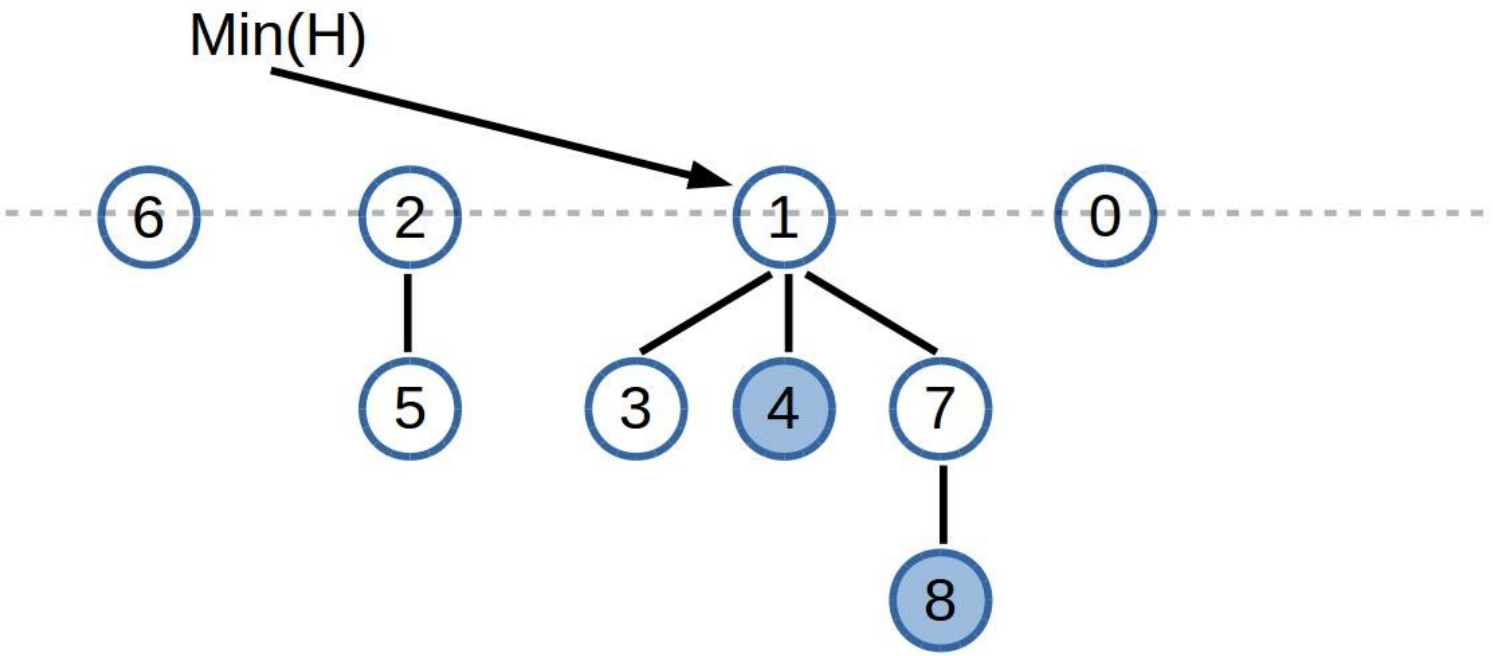
Decrease-Key(H,x,0)

-key[x]=0
-x in die Wurzelliste einfügen



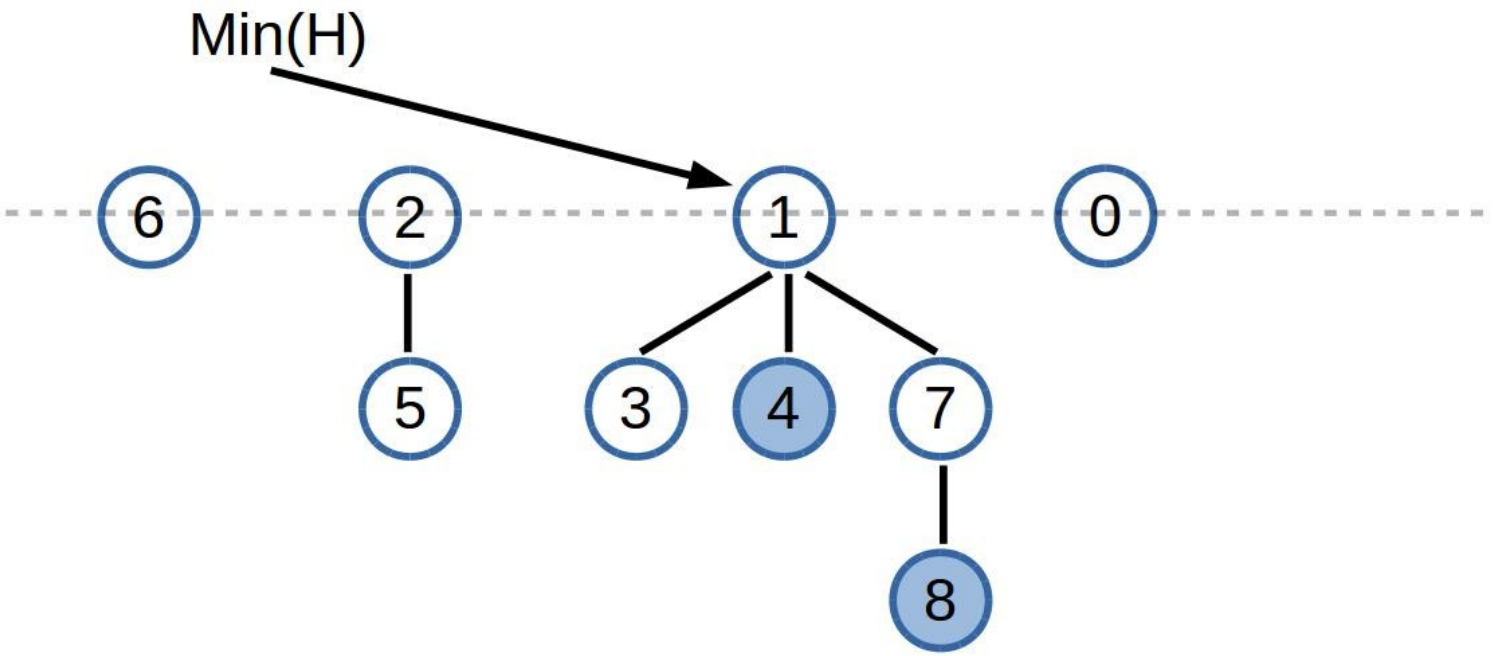
Decrease-Key(H,x,0)

-key[x]=0
-x in die Wurzelliste einfügen



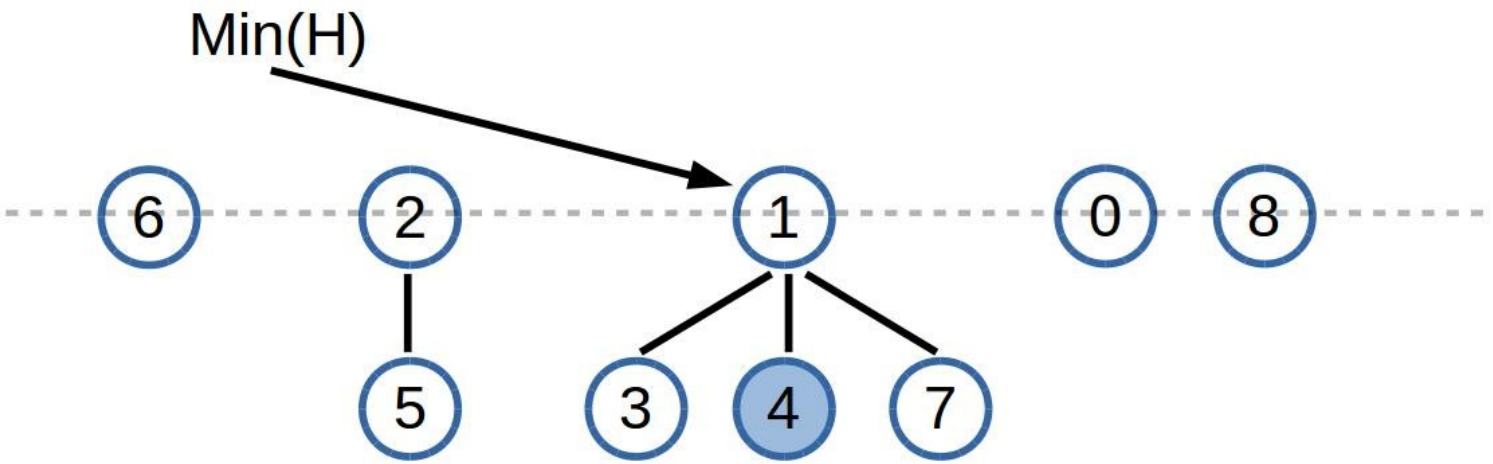
Decrease-Key(H,x,0)

-**if** Elternknoten ist markiert,
Ihn in die Wurzelliste einfügen



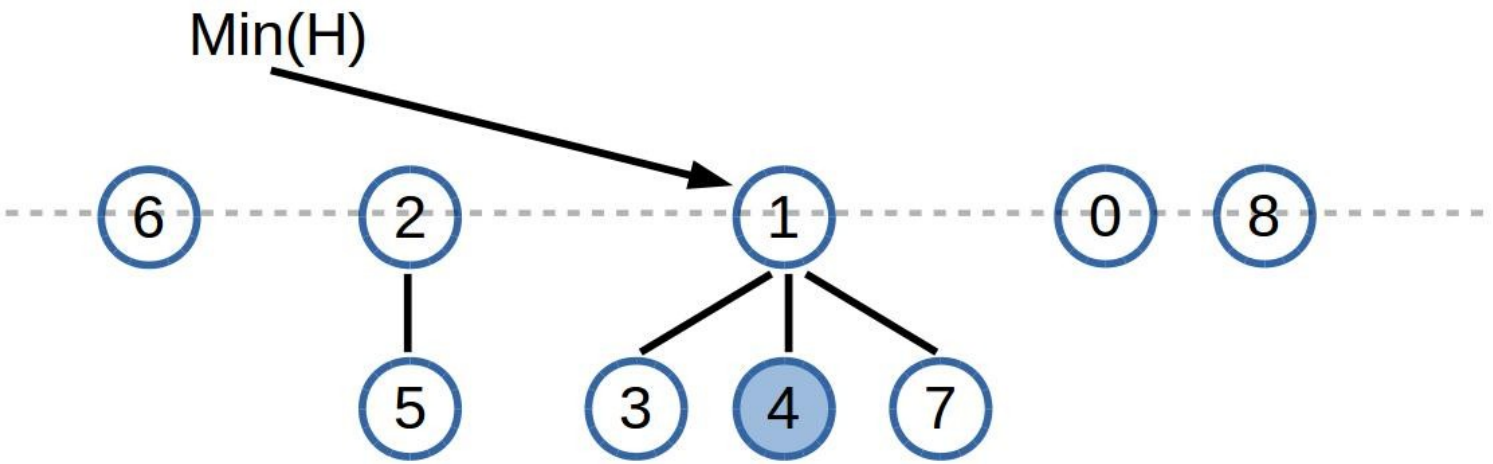
Decrease-Key(H,x,0)

-**if** Elternknoten ist markiert,
Ihn in die Wurzelliste einfügen



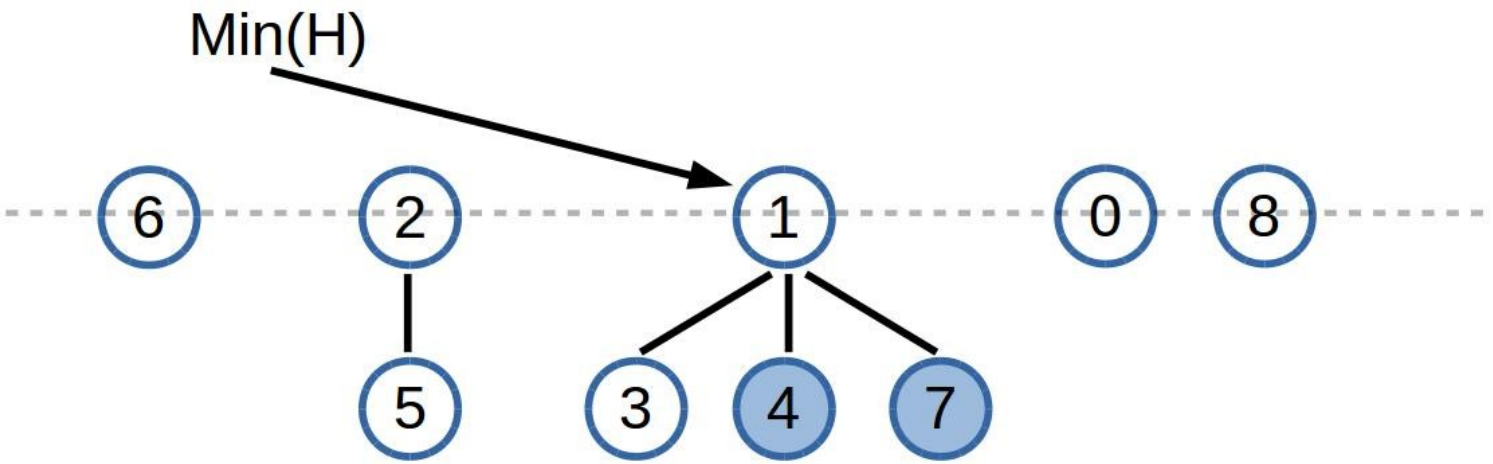
Decrease-Key(H,x,0)

-**if** Elternknoten ist nicht markiert, markiere ihn



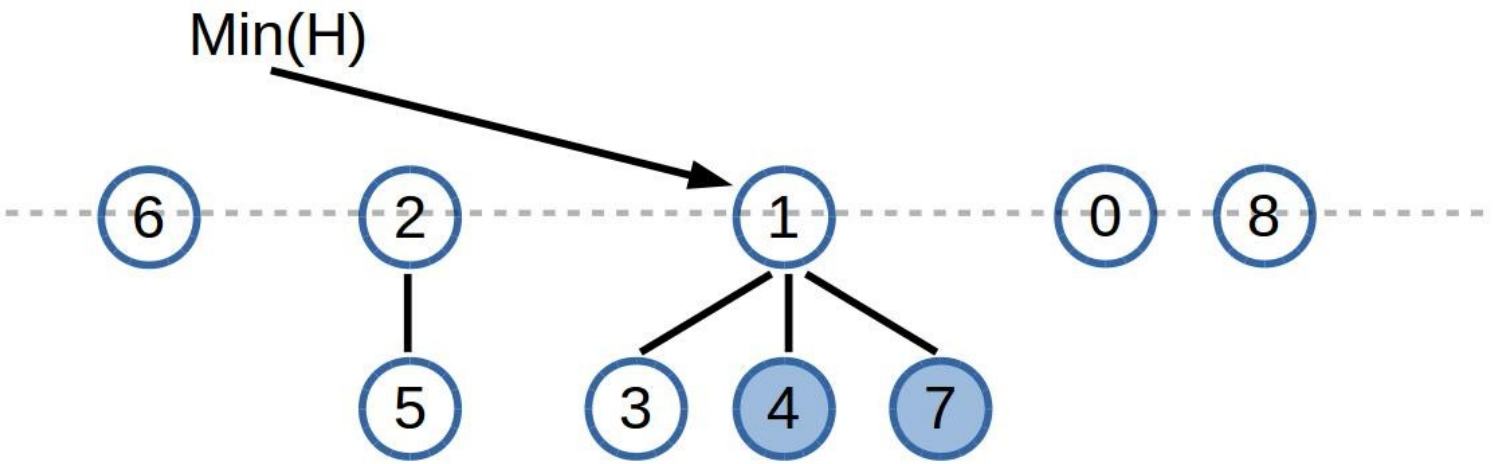
Decrease-Key(H,x,0)

-**if** Elternknoten ist nicht markiert, markiere ihn

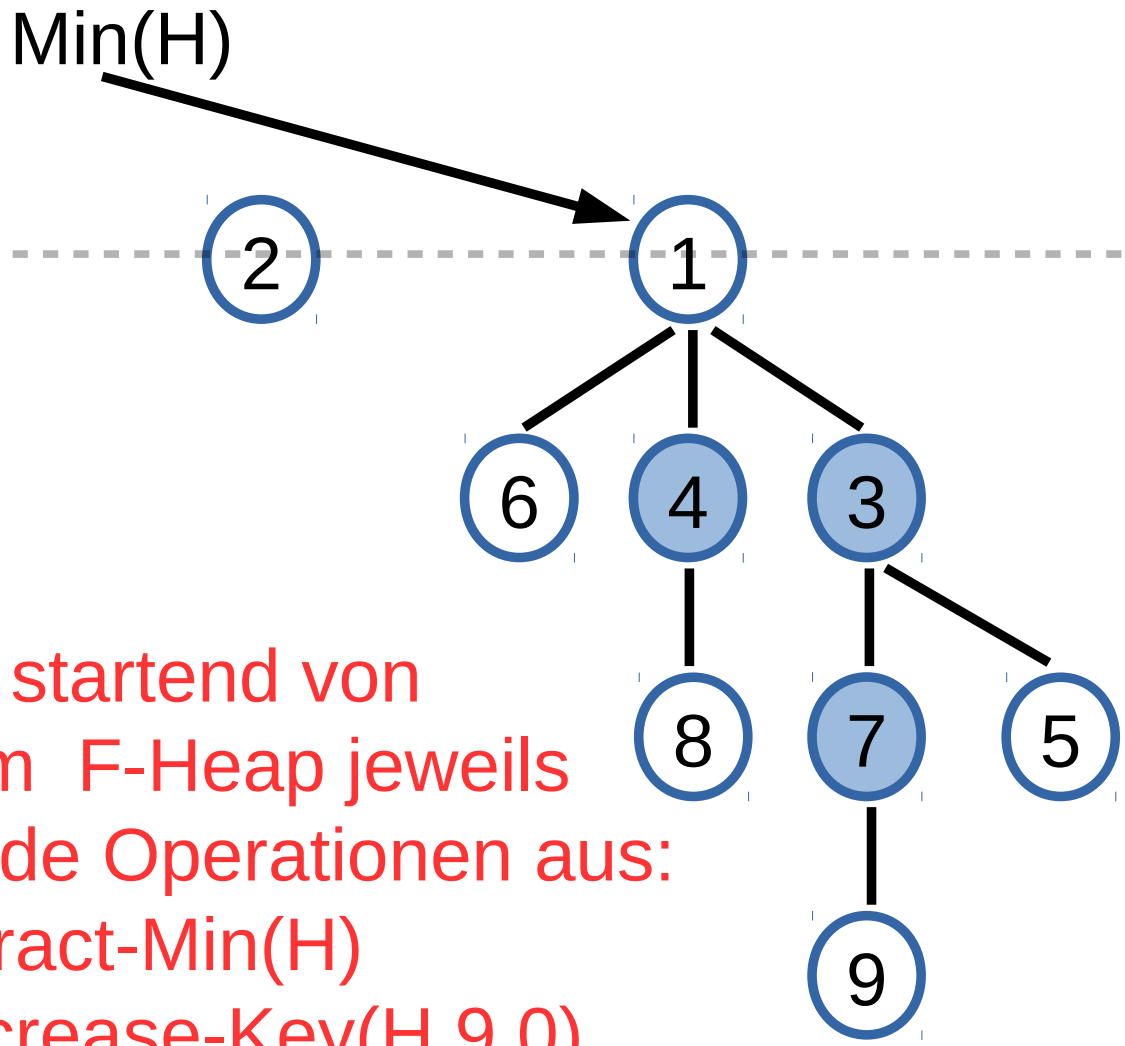


Decrease-Key(H,x,0)

-**if** Elternknoten ist nicht markiert, markiere ihn



2-Minuten Übung



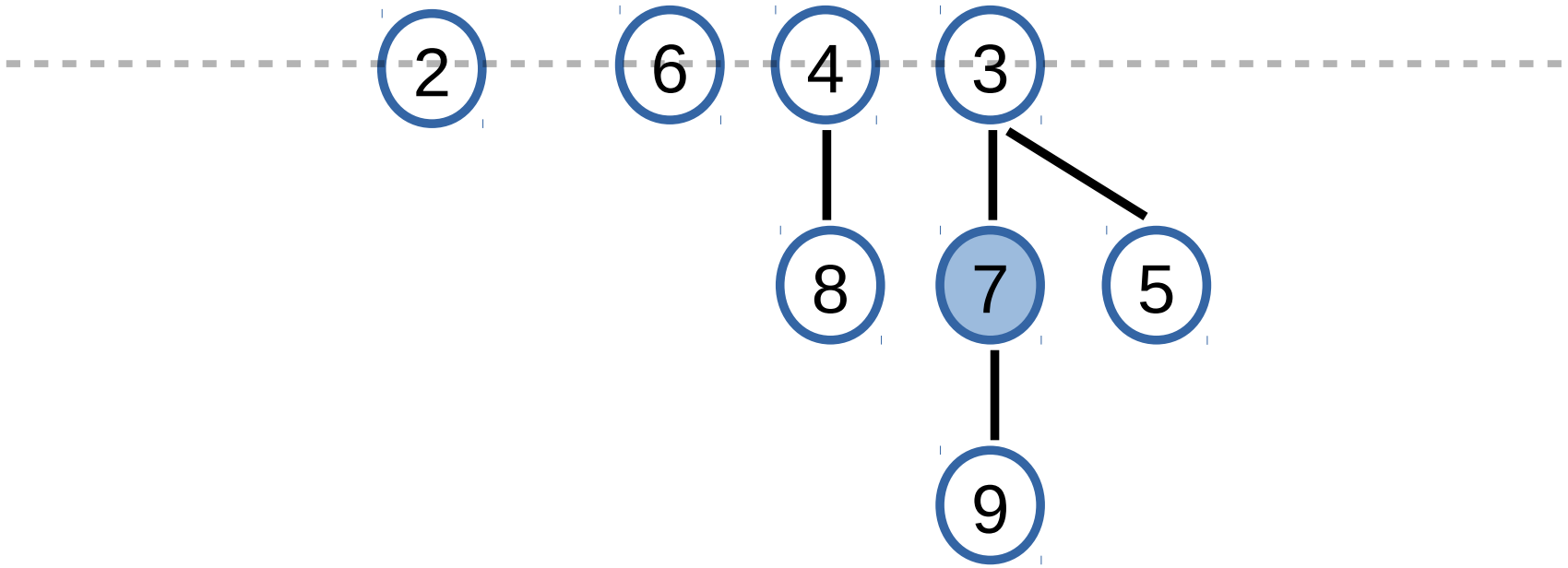
Führe startend von diesem F-Heap jeweils folgende Operationen aus:

a) Extract-Min(H)

b) Decrease-Key(H,9,0)

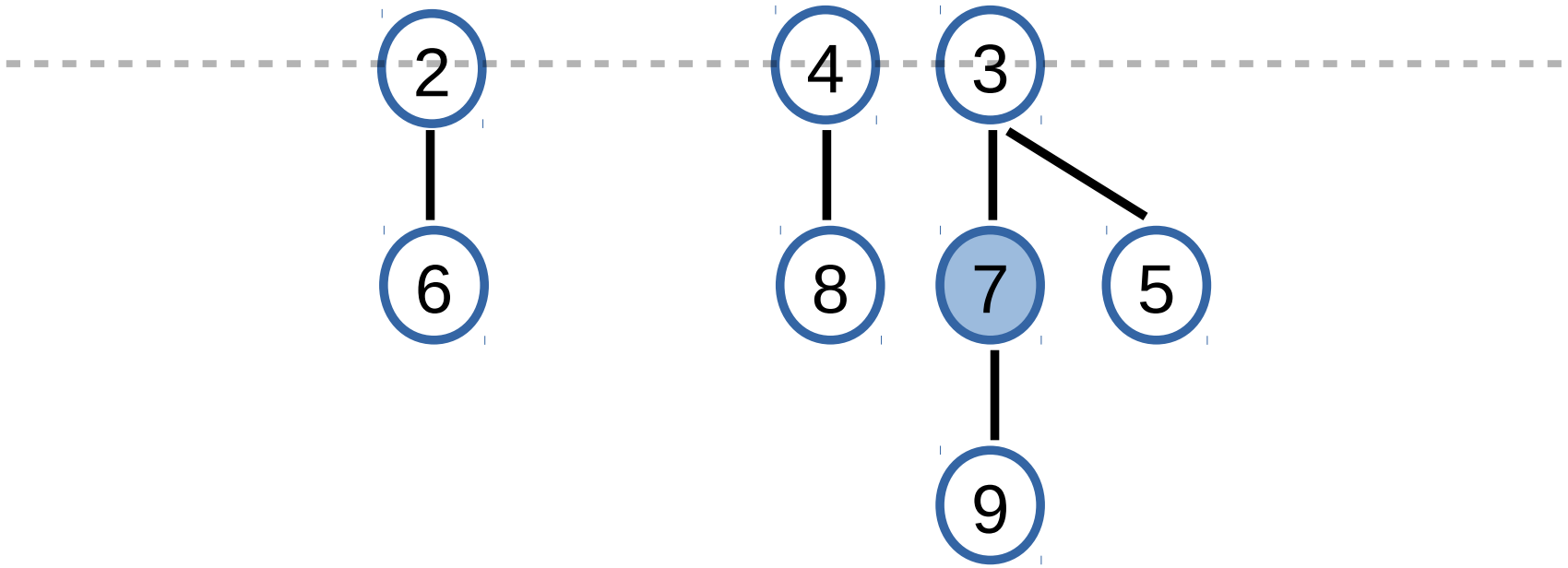
Extract-Min

Min(H)



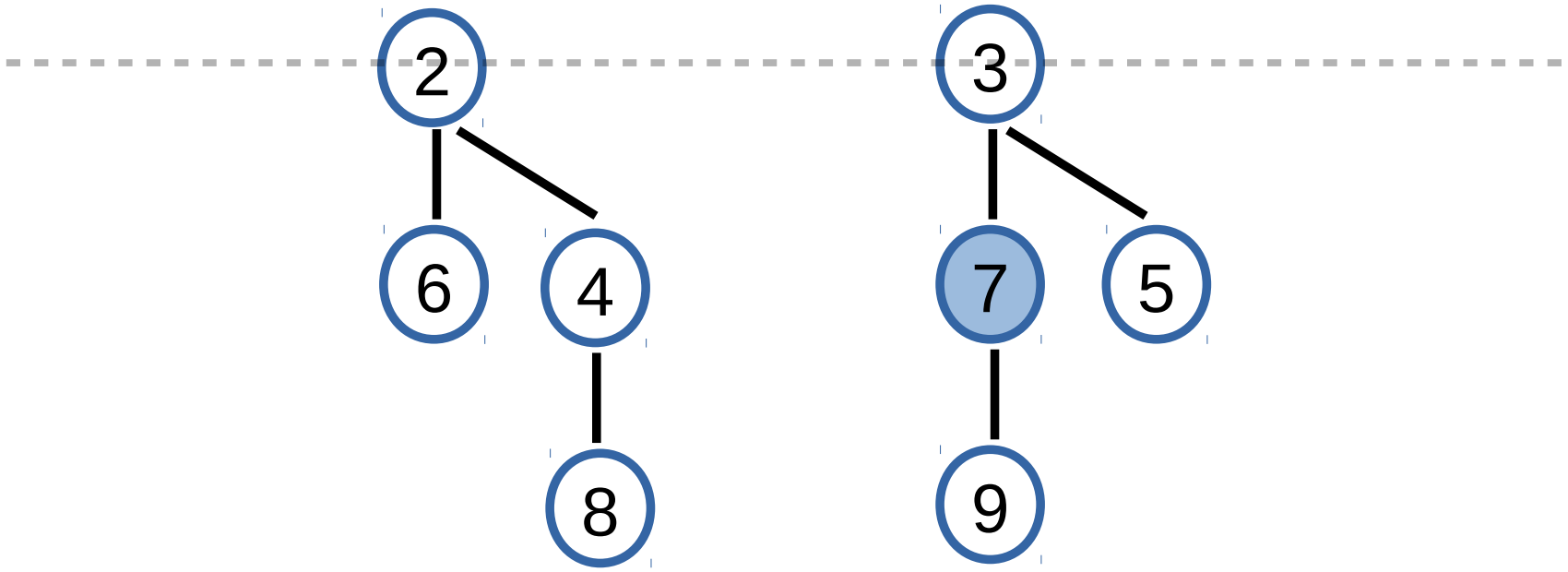
Extract-Min

Min(H)



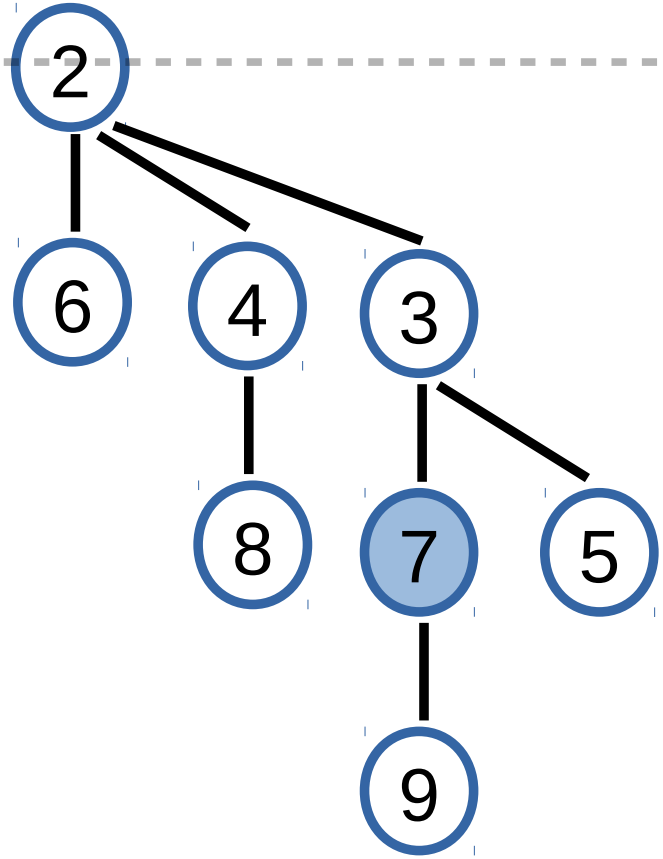
Extract-Min

Min(H)

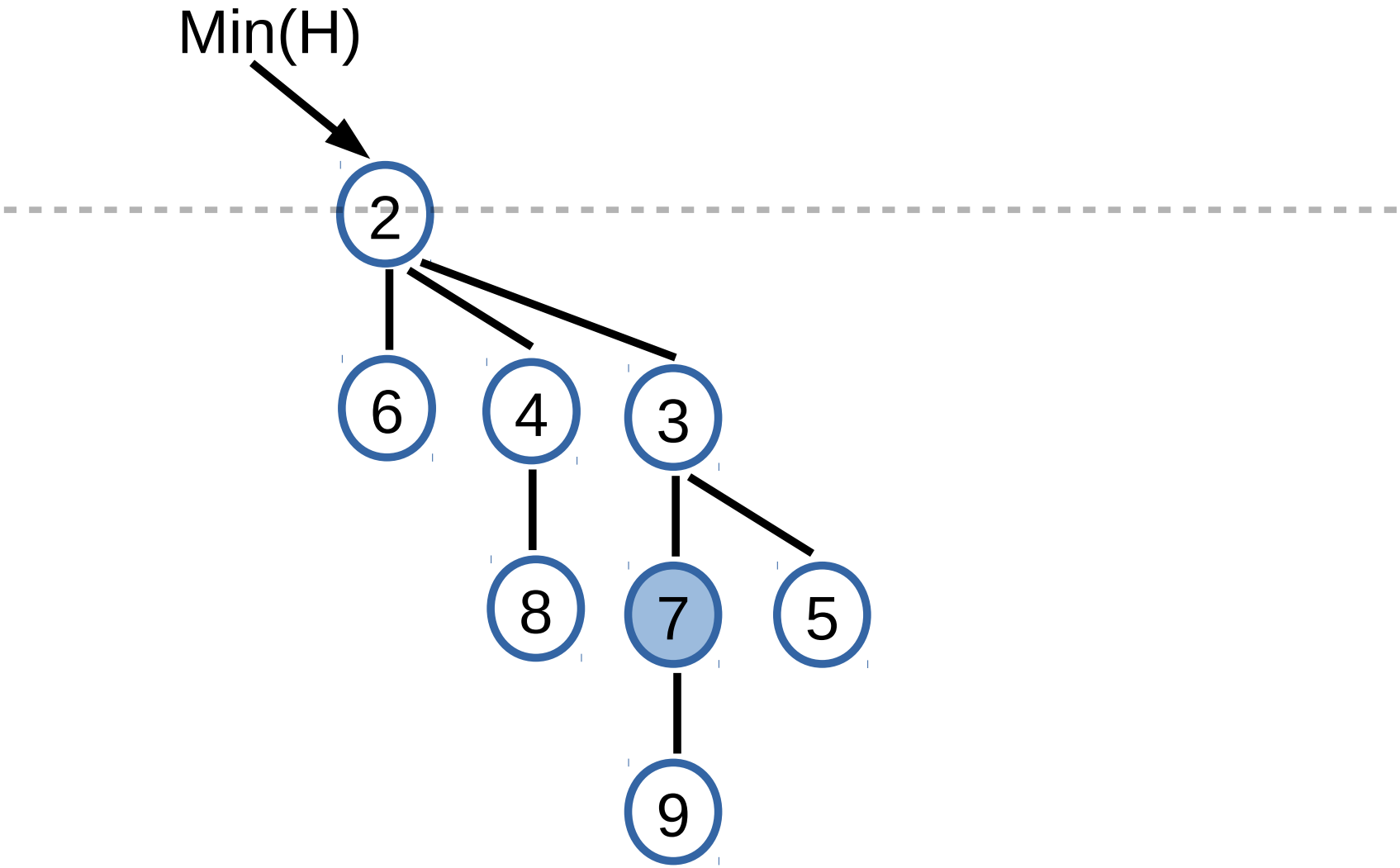


Extract-Min

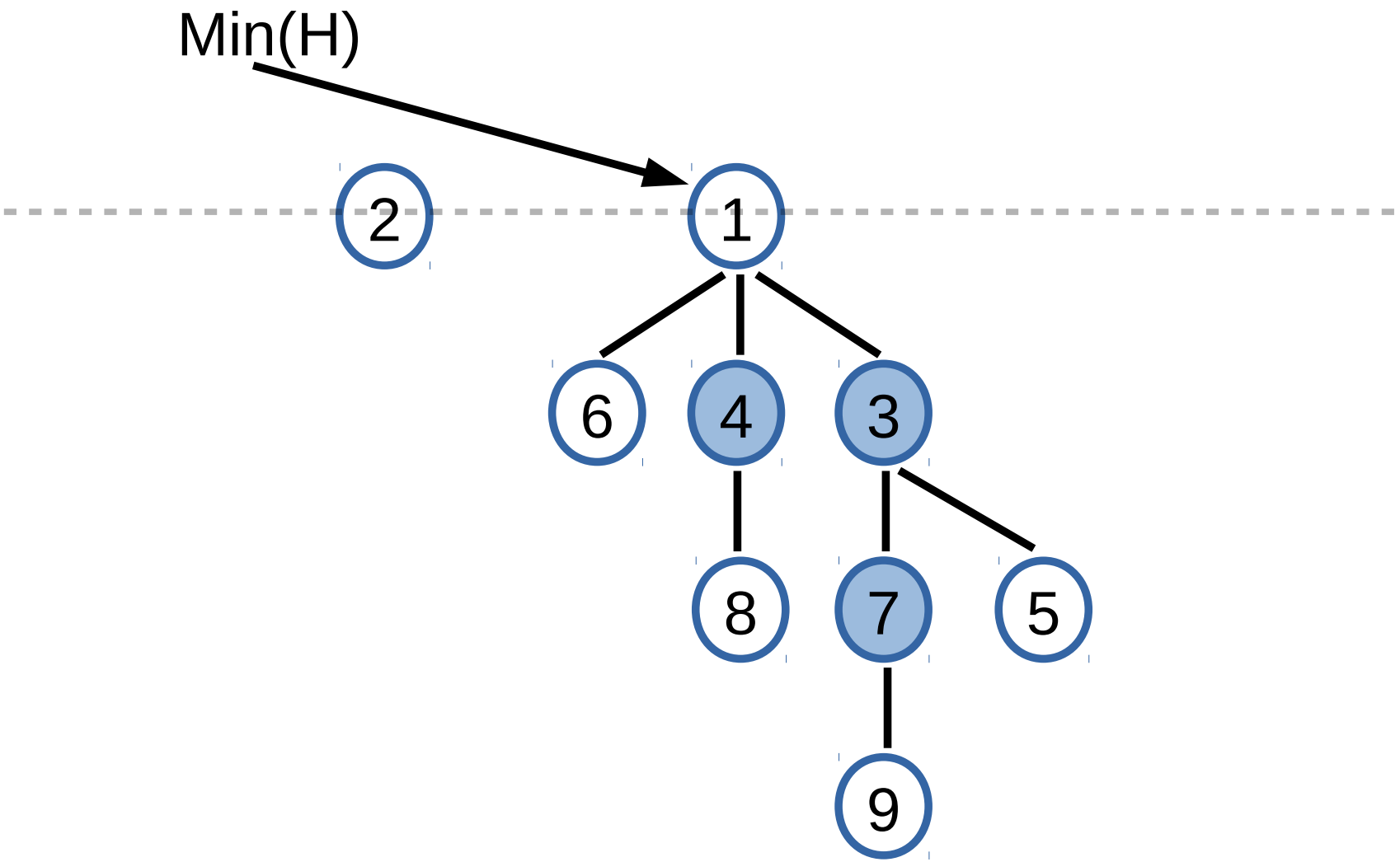
Min(H)



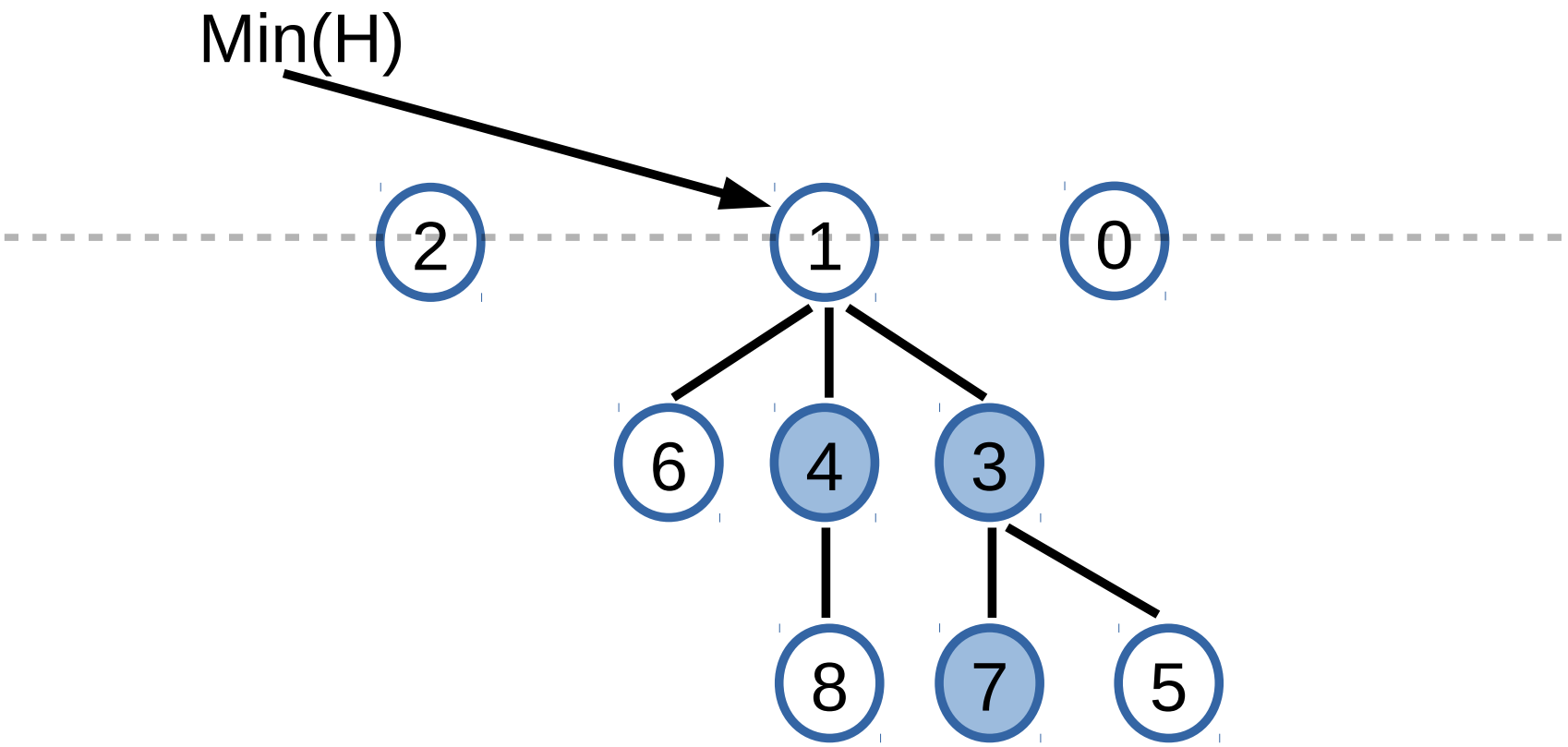
Extract-Min



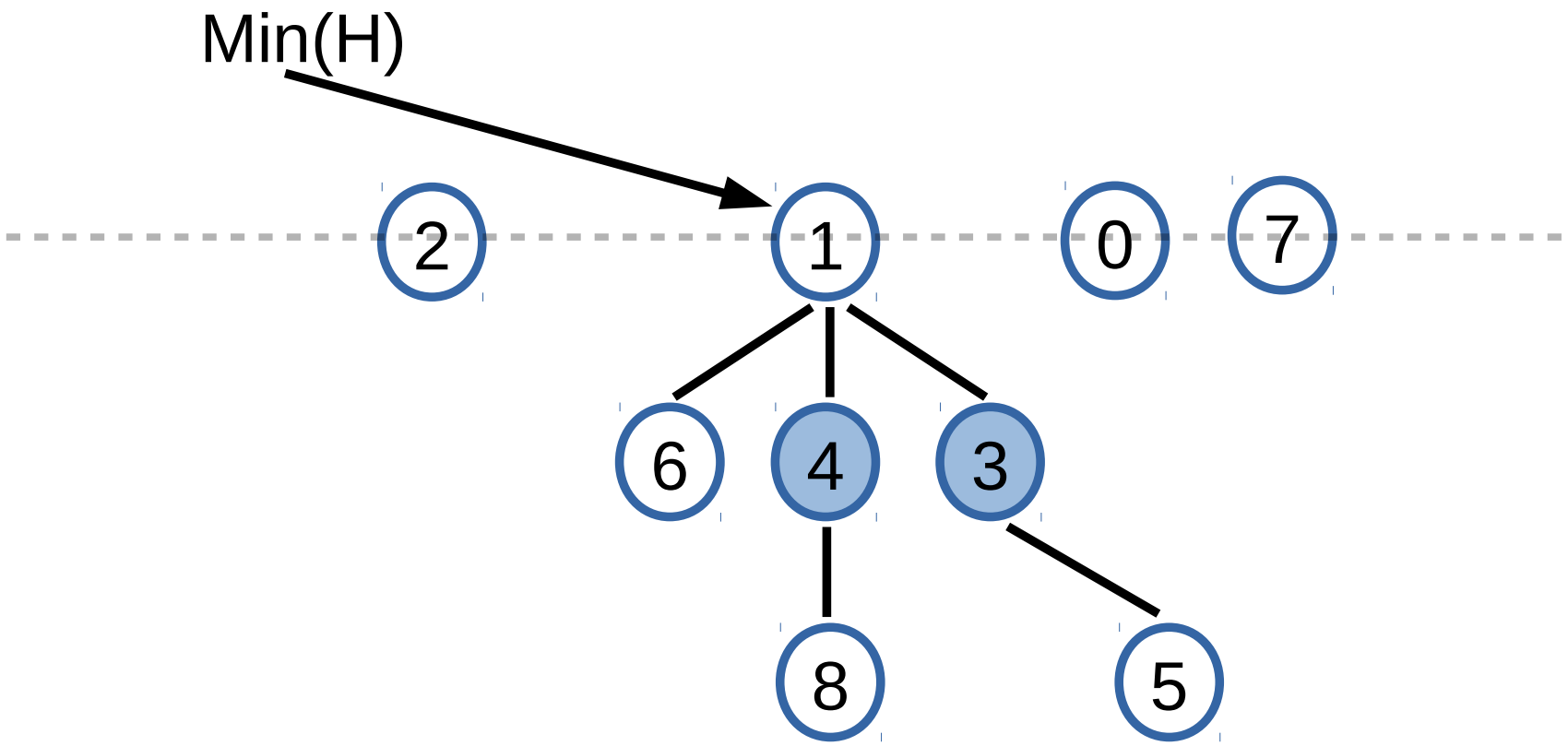
Decrease-Key(H,9,0)



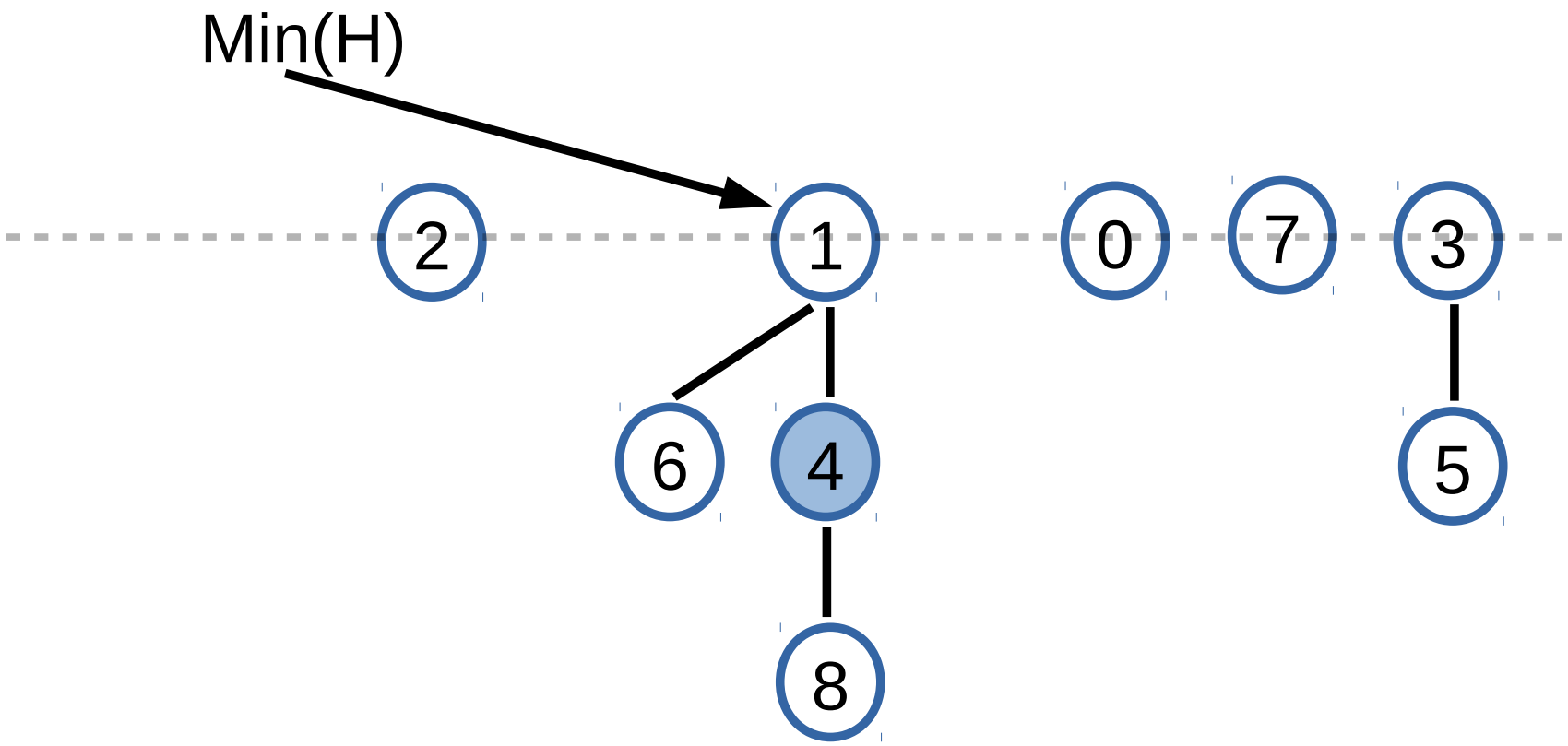
Decrease-Key(H,9,0)



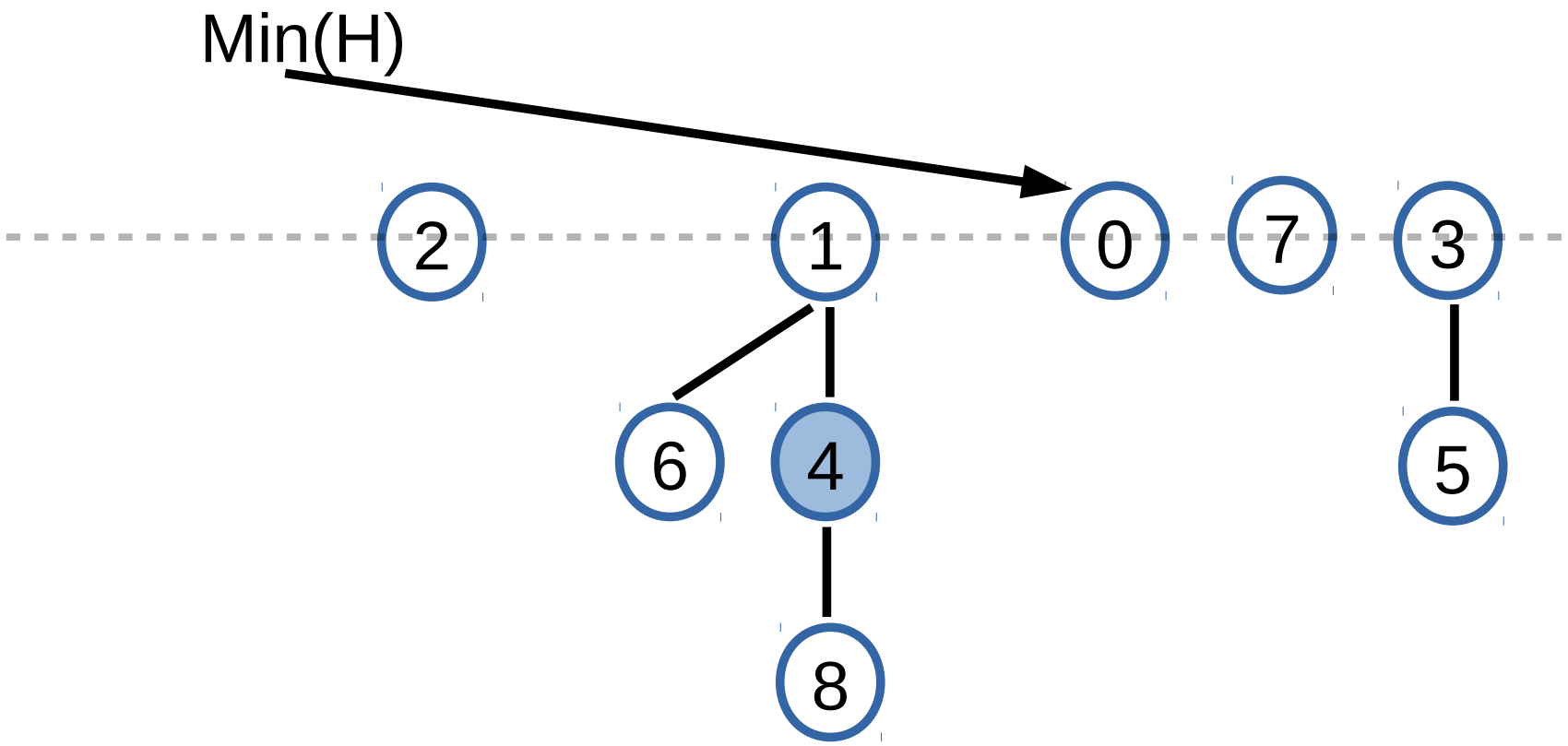
Decrease-Key(H,9,0)



Decrease-Key(H,9,0)



Decrease-Key(H,9,0)



Fibonacci-Heaps: Eigenschaften

Satz:
Beginnend mit einem leeren Heap können k Operationen `Insert`, `Decrease-Key` und `Extract-Min` in Zeit $\mathcal{O}(k + \ell \cdot \log n)$ ausgeführt werden, wobei ℓ die Anzahl von `Extract-Min` Operationen ist und n die maximale Anzahl von Elementen bezeichnet, die zu irgend einem Zeitpunkt im Heap enthalten sind.

Beweisidee

(1) Für jeden F-Heap H mit n Elementen gilt:

$$rank[x] \leq \log_{3/2}(n) \quad \text{für alle } x \in H$$

(2) Verwende amortisierte Analyse um zu zeigen:

Bezeichnet a(i) die amortisierte Laufzeit der i-ten Operation, so gilt:

$$a(i) = \begin{cases} O(1) & \text{i-te Operation Insert oder DecreaseKey} \\ O(\log n) & \text{i-te Operation ExtractMin} \end{cases}$$

Beweis (1)

Für jeden F-Heap H mit n Elementen gilt:

$$\text{rank}[x] \leq \log_{3/2}(n) \quad \text{für alle } x \in H$$

Beweis (2)

Für die amortisierte Laufzeit a(i) der i-ten Operation gilt:

$$a(i) = \begin{cases} O(1) & \text{i-te Operation Insert oder DecreaseKey} \\ O(\log n) & \text{i-te Operation ExtractMin} \end{cases}$$

Wir setzen: (C geeignete Konstante, s.u.)

Φ(i) := 2C · #markierter Knoten + C · #Knoten in Wurzelliste

Beweis (2)

Insert:

- $t(i) - t(i-1) \leq C$
- $m(i) = m(i-1)$
- $r(i) = r(i-1) + 1$
- $\Phi(i) - \Phi(i-1) = C$

$$a(i) = t(i) - t(i-1) + \Phi(i) - \Phi(i-1)$$
$$\leq C + C = O(1)$$

Beweis (2)

Decrease-Key:

Sei x die Anzahl Abschneide-Operationen

- $t(i) - t(i-1) \leq C\,x + C$
- $m(i) \leq m(i-1) - (x-1) + 1$
- $r(i) = r(i-1) + x$
- $\Phi(i) - \Phi(i-1) \leq -C\,x + 4\,C$

$$a(i) = t(i) - t(i-1) + \Phi(i) - \Phi(i-1)$$
$$\leq 5\,C = O(1)$$

Beweis (2)

Extract-Min:

Sei x die Anzahl Bäume, die in Consolidate zusammen gehängt werden.

- $t(i) - t(i-1) \leq C (\log_{3/2}(n) + \log_{3/2}(n) + x)$
- $m(i) \leq m(i-1)$
- $r(i) \leq r(i-1) + \log_{3/2}(n) - x$
- $\Phi(i) - \Phi(i-1) \leq C (\log_{3/2}(n) - x)$

$$a(i) = t(i) - t(i-1) + \Phi(i) - \Phi(i-1)$$
$$\leq 2 C \log_{3/2}(n) = O(\log_{3/2}(n))$$

Ende

Algorithmen & Datenstrukturen

Wiederholung

- Beispiele effizienter Algorithmen
  - Sortieren, Median, Matrixmult., Pfade, Spannbaum
- Grundlegende Paradigmen für Algorithmenentwurf
  - Divide & Conquer
  - Dynamische Programmierung
  - Greedy Algorithmen
- Datenstrukturen
  - Suchbäume
  - Union-Find-Strukturen
  - Priority Queues
  - Wörterbücher