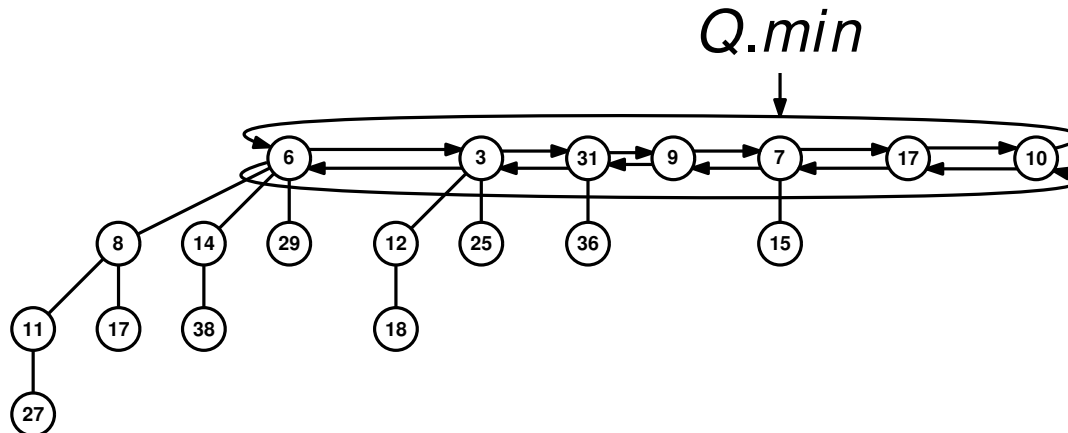




ICS 621: Analysis of Algorithms

Prof. Nodari Sitchinava



Mergeable Priority Queues: Fibonacci Heaps

Heaps

	Binomial
■ MAKE()	$O(1)$
■ INSERT(Q, x)	$O(1)^*$
■ MINIMUM(Q)	$O(1)$
■ EXTRACT-MIN(Q)	$O(\log n)$
■ DECREASE-KEY(Q, x, k)	$O(\log n)$
■ DELETE(Q, x)	$O(\log n)$
■ UNION(Q_1, Q_2)	$O(\log n)$

* Amortized cost

Heaps

	Binomial	Lazy Binomial
■ MAKE()	$O(1)$	$O(1)$
■ INSERT(Q, x)	$O(1)^*$	$O(1)$
■ MINIMUM(Q)	$O(1)$	$O(1)$
■ EXTRACT-MIN(Q)	$O(\log n)$	$O(\log n)^*$
■ DECREASE-KEY(Q, x, k)	$O(\log n)$	$O(\log n)$
■ DELETE(Q, x)	$O(\log n)$	$O(\log n)^*$
■ UNION(Q_1, Q_2)	$O(\log n)$	$O(1)$

* Amortized cost

Heaps

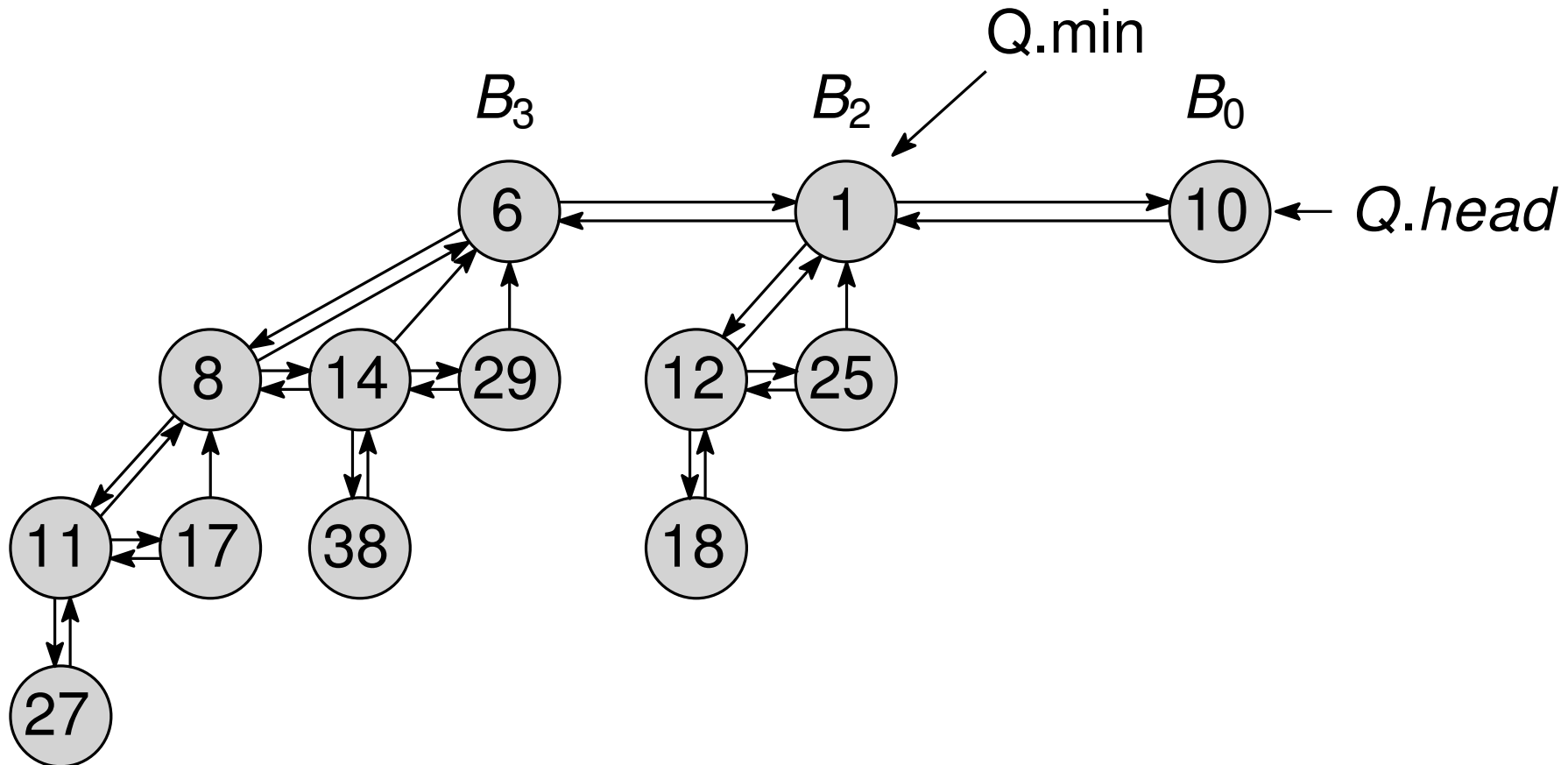
	Binomial	Lazy Binomial	Fibonacci
■ MAKE()	$O(1)$	$O(1)$	$O(1)$
■ INSERT(Q, x)	$O(1)^*$	$O(1)$	$O(1)$
■ MINIMUM(Q)	$O(1)$	$O(1)$	$O(1)$
■ EXTRACT-MIN(Q)	$O(\log n)$	$O(\log n)^*$	$O(\log n)^*$
■ DECREASE-KEY(Q, x, k)	$O(\log n)$	$O(\log n)$	$O(1)^*$
■ DELETE(Q, x)	$O(\log n)$	$O(\log n)^*$	$O(\log n)^*$
■ UNION(Q_1, Q_2)	$O(\log n)$	$O(1)$	$O(1)$

* Amortized cost

Reminder: Binomial Heaps

Collection of heap-ordered binomial trees:

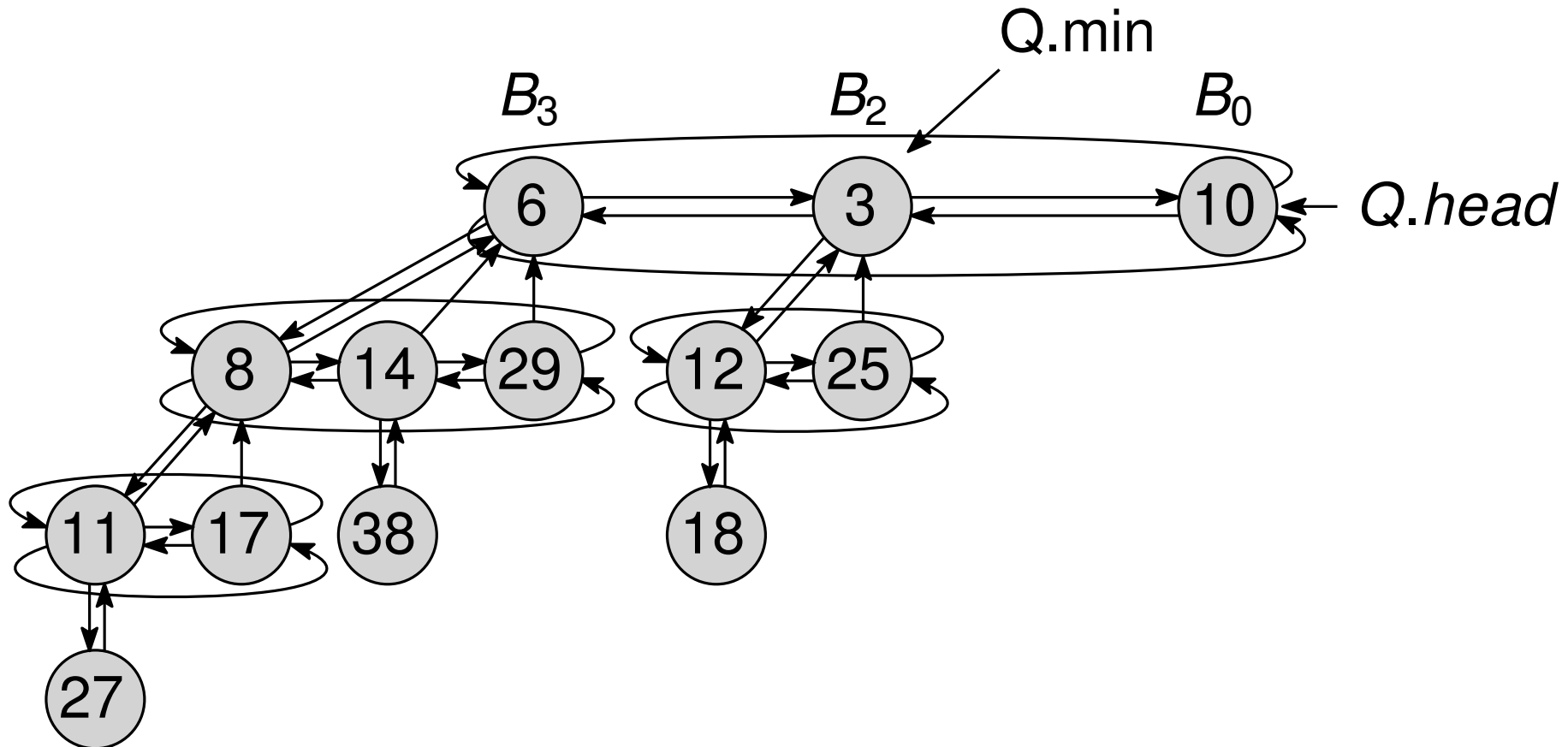
- Each tree is heap-ordered
- At most **one** tree B_k , for $k = 0, 1, 2, \dots, \lfloor \log n \rfloor$



Reminder: Lazy Binomial Heaps

Collection of heap-ordered binomial trees:

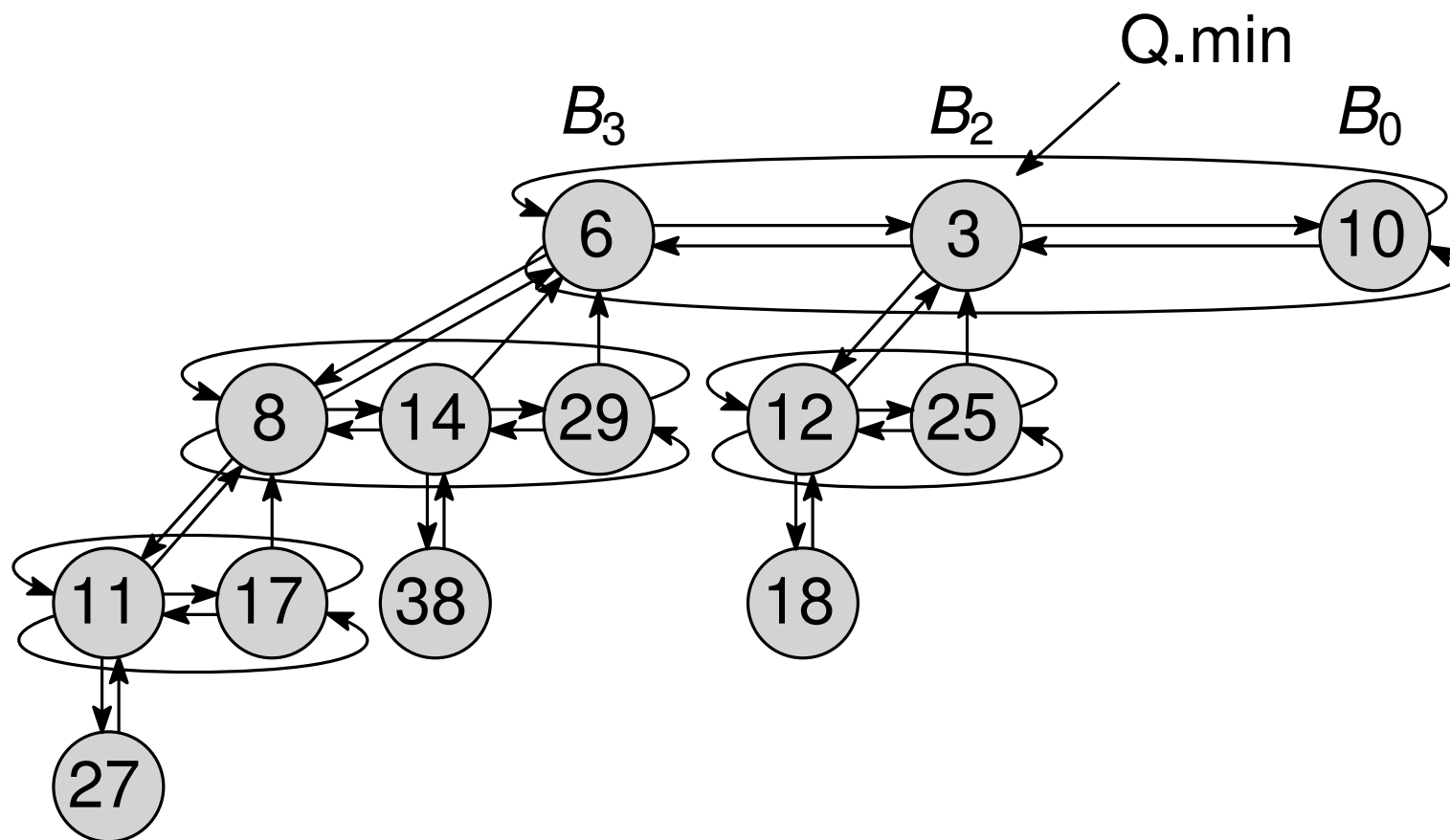
- Each tree is heap-ordered
- *Arbitrary* number of trees in the root list
- Sibling lists are doubly-linked *circular* lists



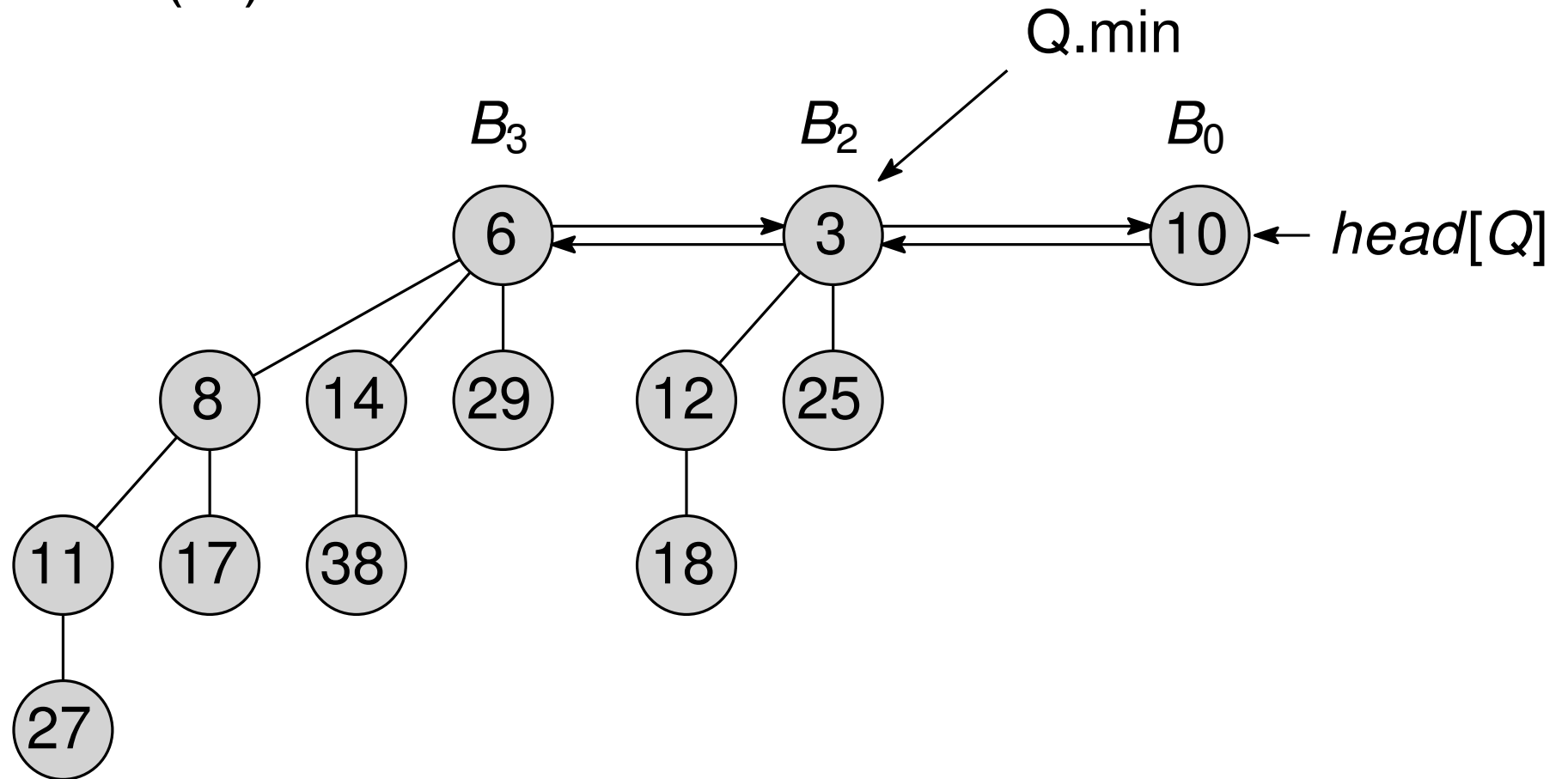
Reminder: Lazy Binomial Heaps

Collection of heap-ordered binomial trees:

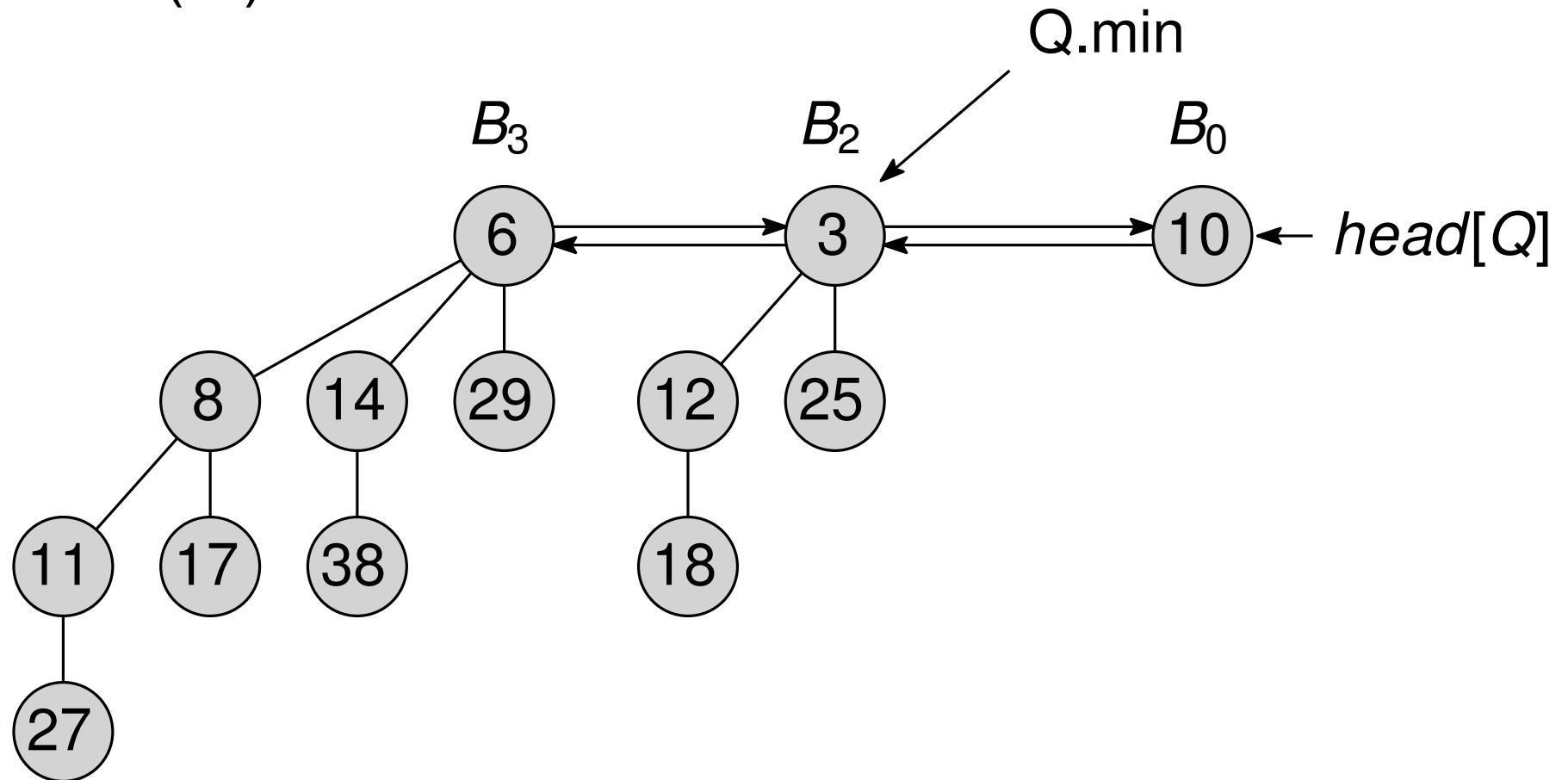
- Each tree is heap-ordered
- *Arbitrary* number of trees in the root list
- Sibling lists are doubly-linked *circular* lists



MINIMUM(Q)

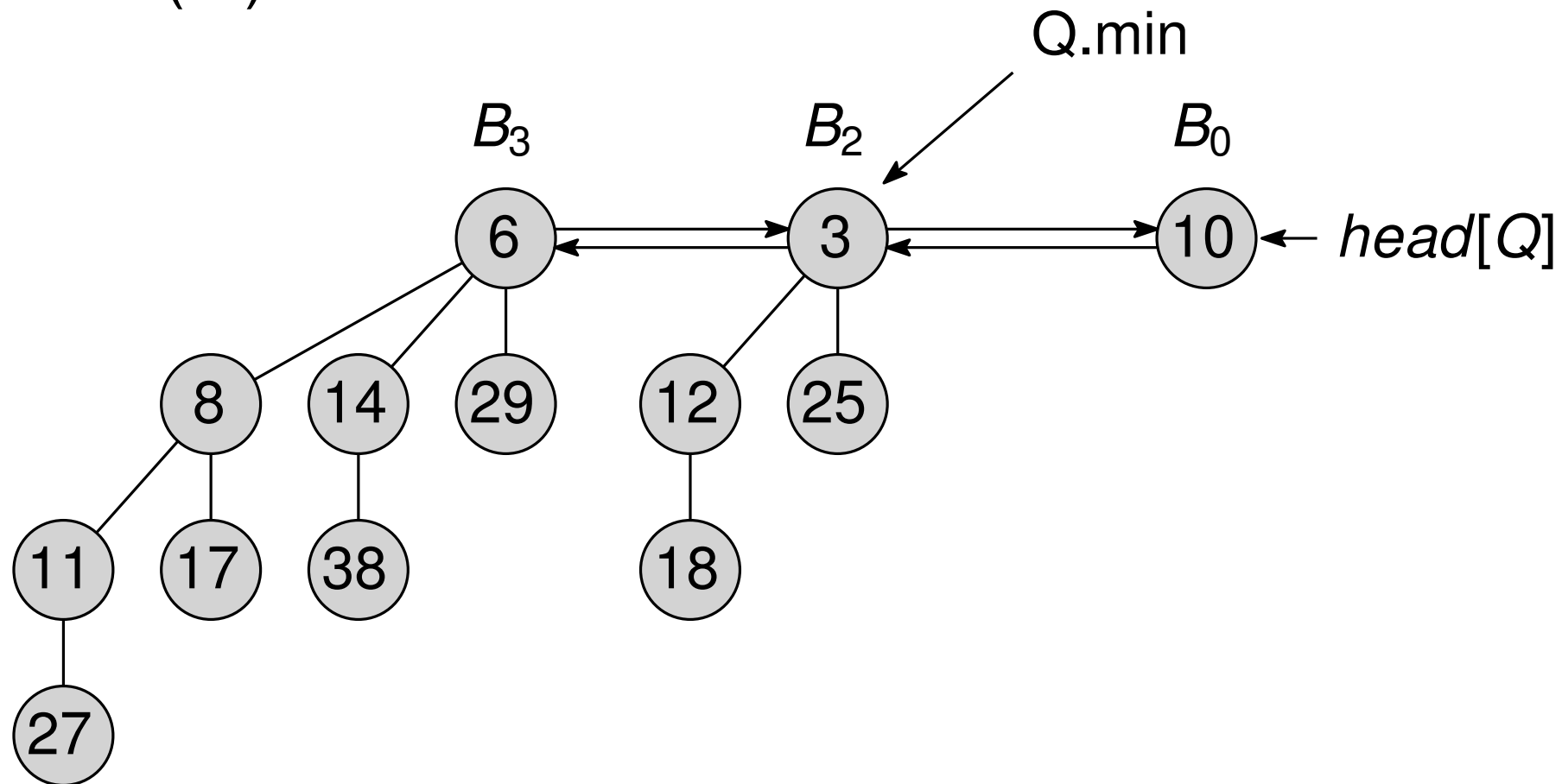


MINIMUM(Q)



function MINIMUM(Q)
return $Q.min$

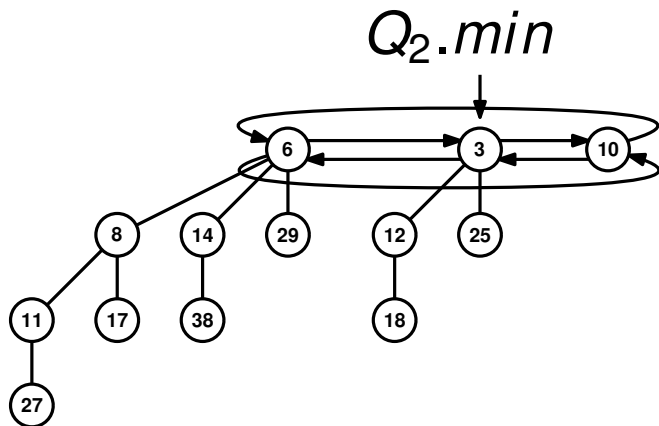
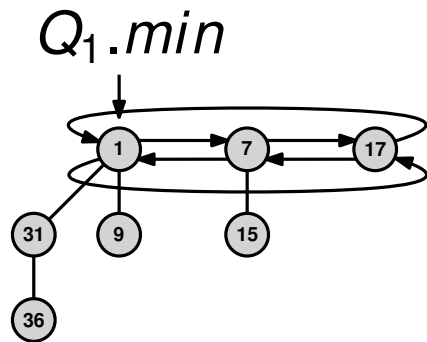
MINIMUM(Q)



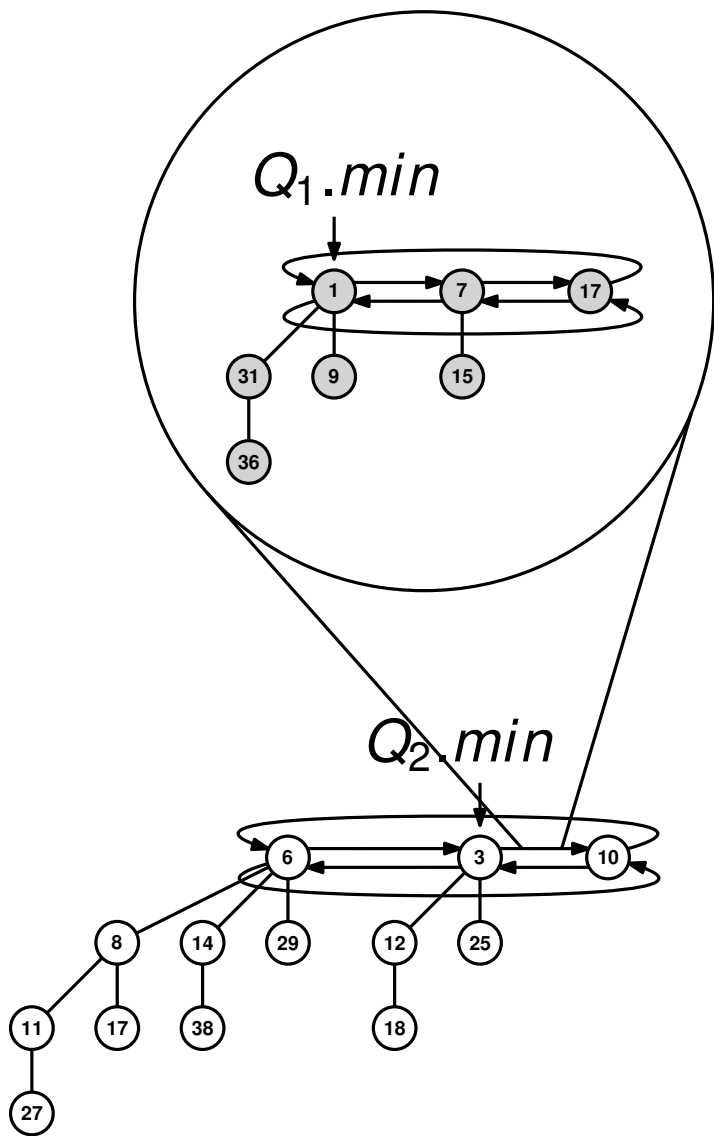
function MINIMUM(Q)
return $Q.min$

$O(1)$ time worst-case

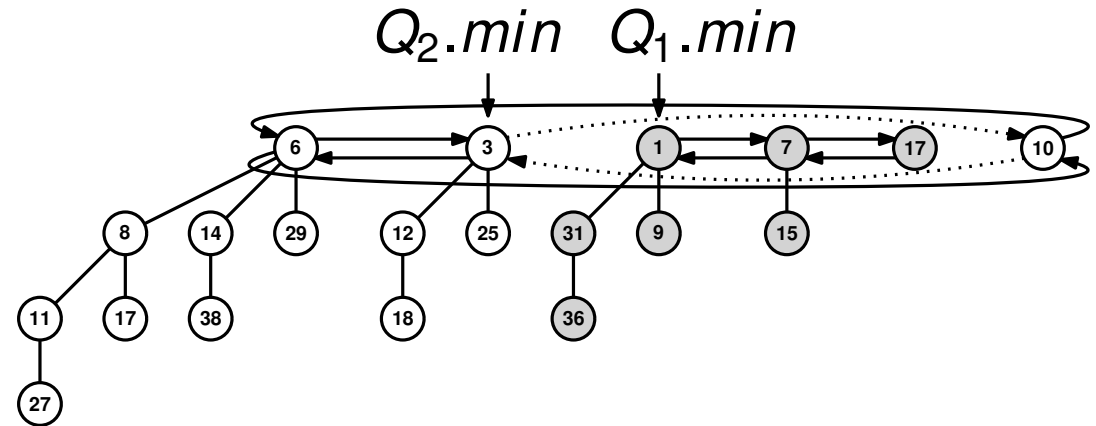
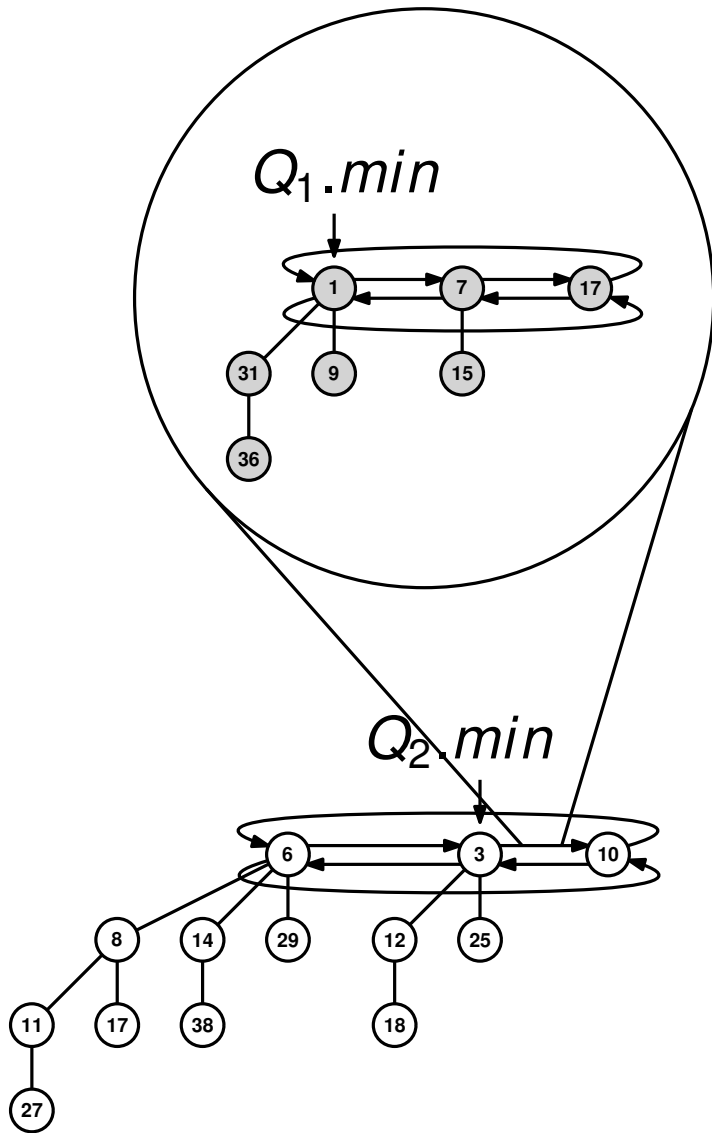
Lazy UNION(Q_1, Q_2)



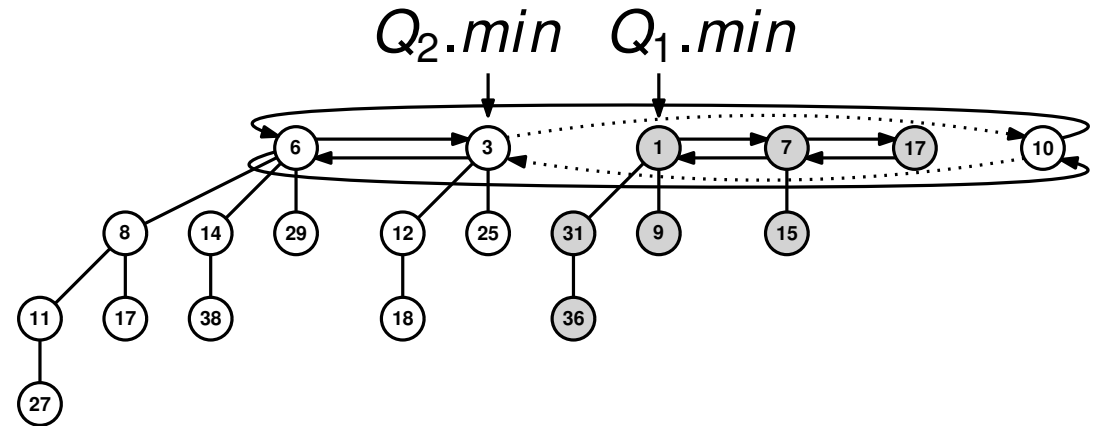
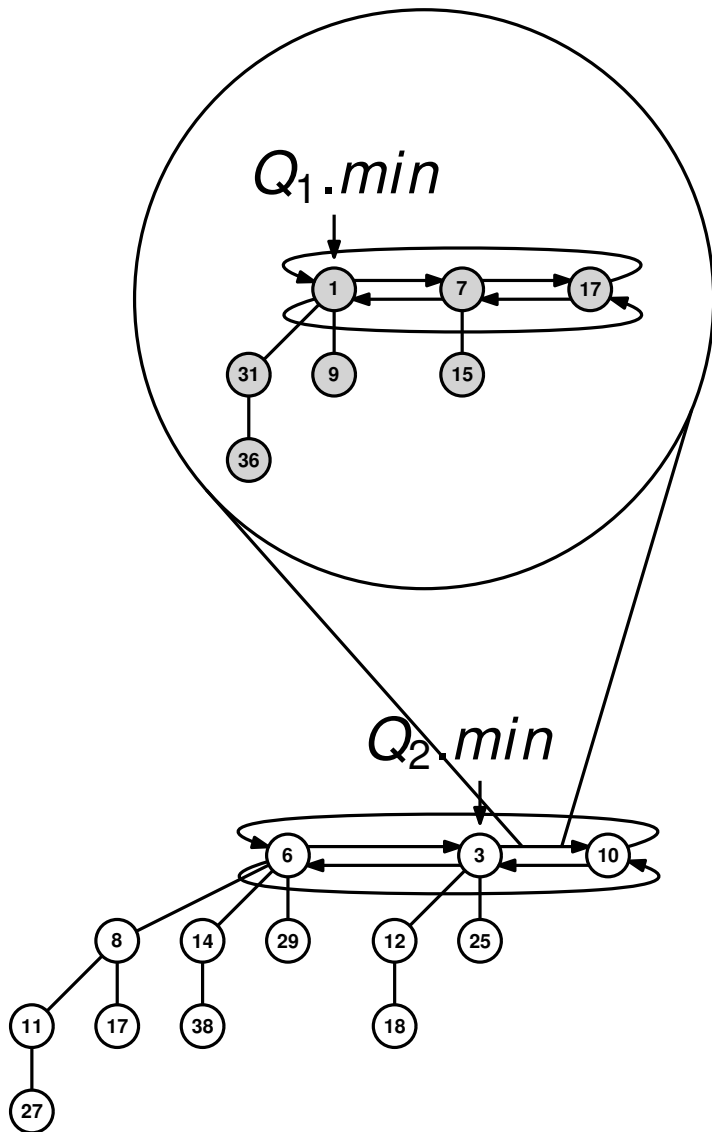
Lazy UNION(Q_1, Q_2)



Lazy UNION(Q_1, Q_2)



Lazy UNION(Q_1, Q_2)



function UNION(Q_1, Q_2)

$L_1 \leftarrow Q_1.min.left$

$R_2 \leftarrow Q_2.min.right$

$L_1.right \leftarrow R_2$

$R_2.left \leftarrow L_1$

$Q_2.min.right \leftarrow Q_1.min$

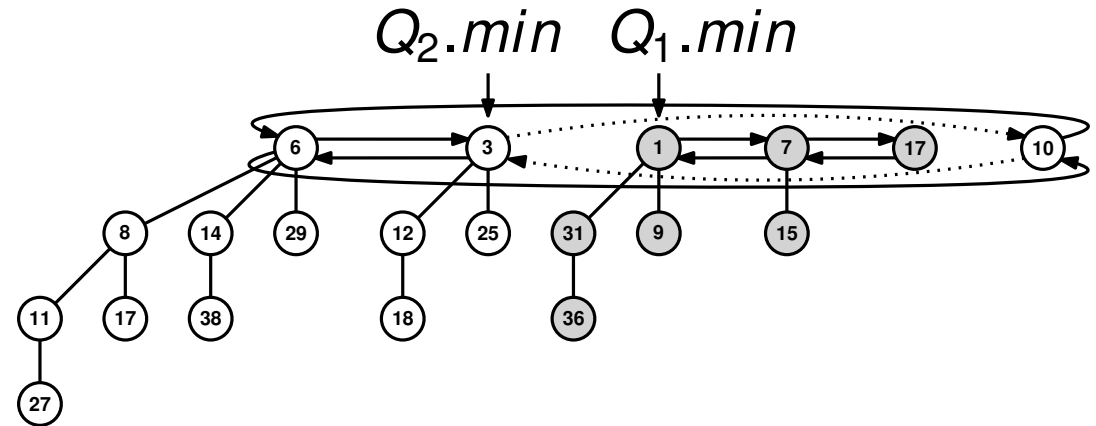
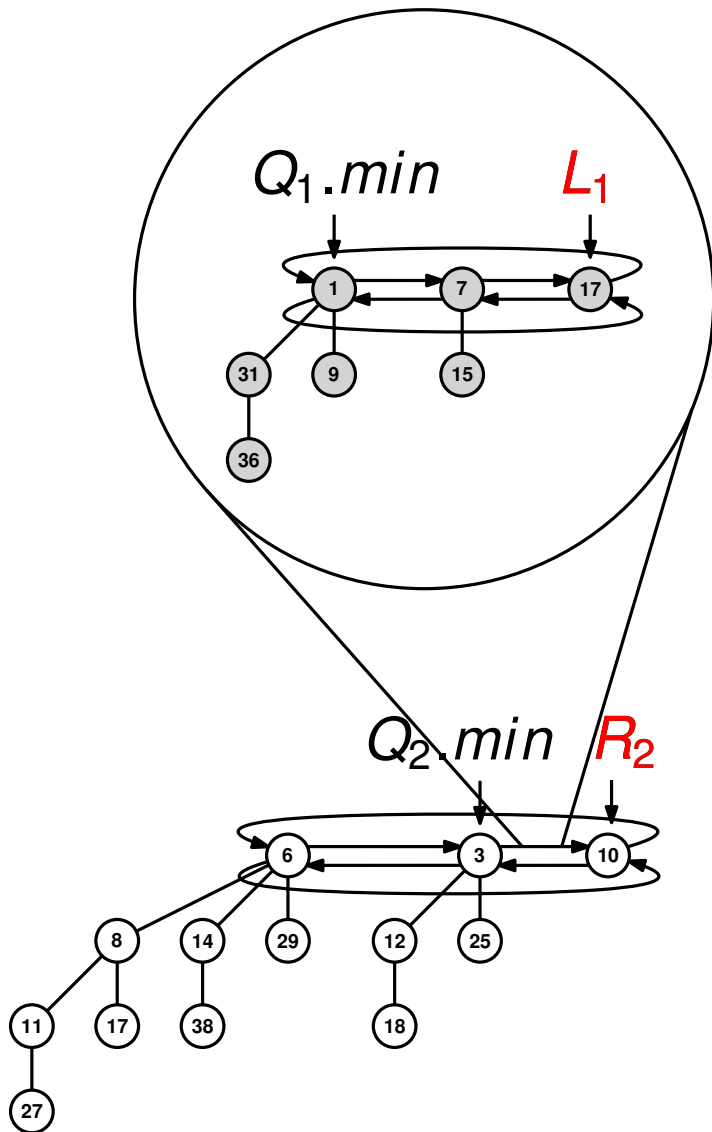
$Q_1.min.left \leftarrow Q_2.min$

if $Q_1.min.key < Q_2.min.key$ **then**

$Q_2.min \leftarrow Q_1.min$

return Q_2

Lazy UNION(Q_1, Q_2)



function UNION(Q_1, Q_2)

→ $L_1 \leftarrow Q_1.min.left$

→ $R_2 \leftarrow Q_2.min.right$

$L_1.right \leftarrow R_2$

$R_2.left \leftarrow L_1$

$Q_2.min.right \leftarrow Q_1.min$

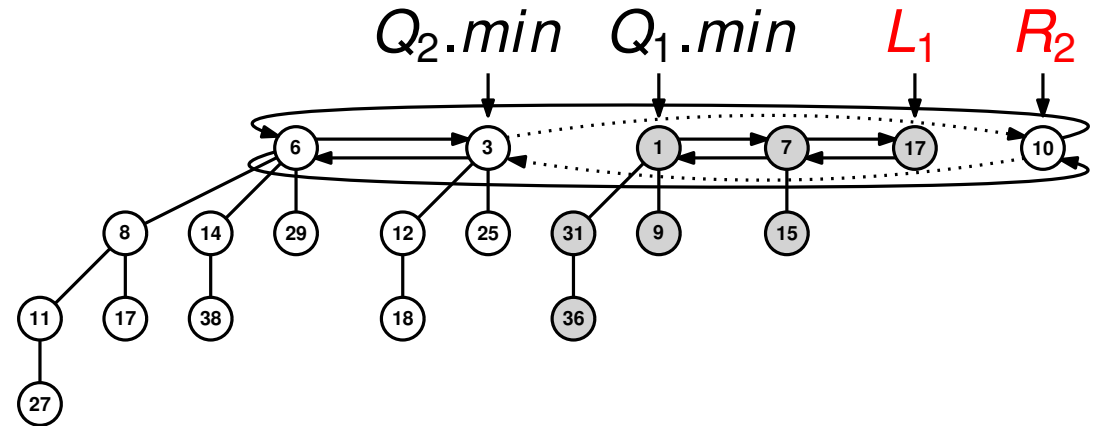
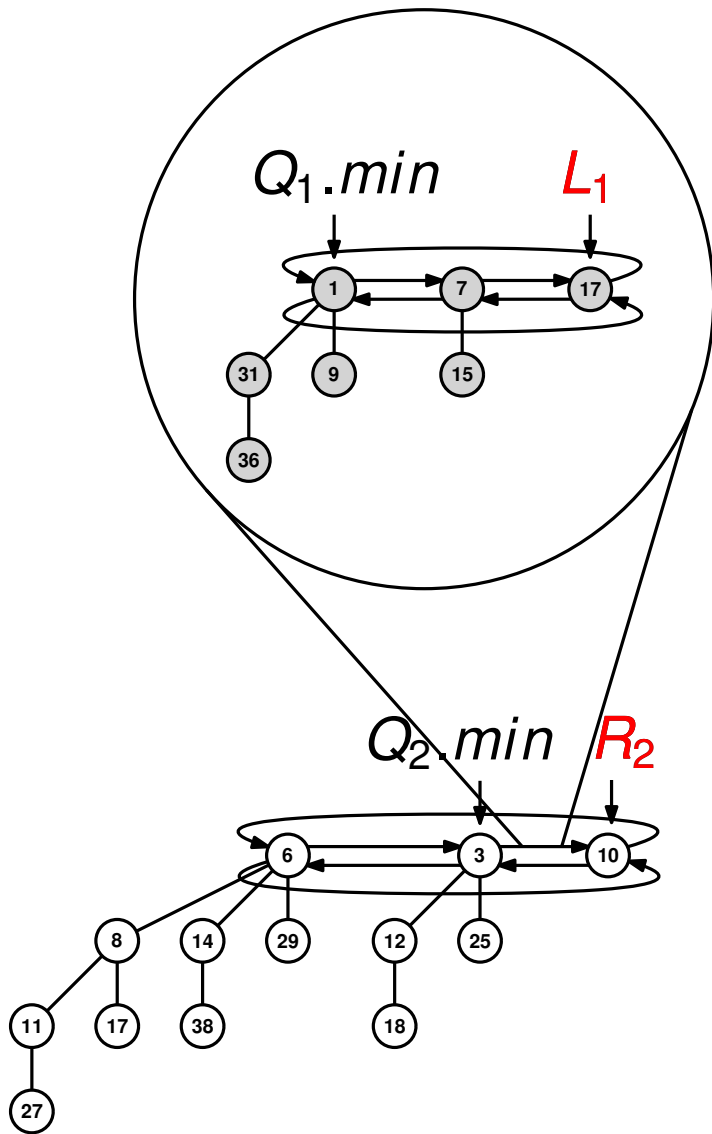
$Q_1.min.left \leftarrow Q_2.min$

if $Q_1.min.key < Q_2.min.key$ **then**

$Q_2.min \leftarrow Q_1.min$

return Q_2

Lazy UNION(Q_1, Q_2)



function UNION(Q_1, Q_2)

→ $L_1 \leftarrow Q_1.min.left$

→ $R_2 \leftarrow Q_2.min.right$

$L_1.right \leftarrow R_2$

$R_2.left \leftarrow L_1$

$Q_2.min.right \leftarrow Q_1.min$

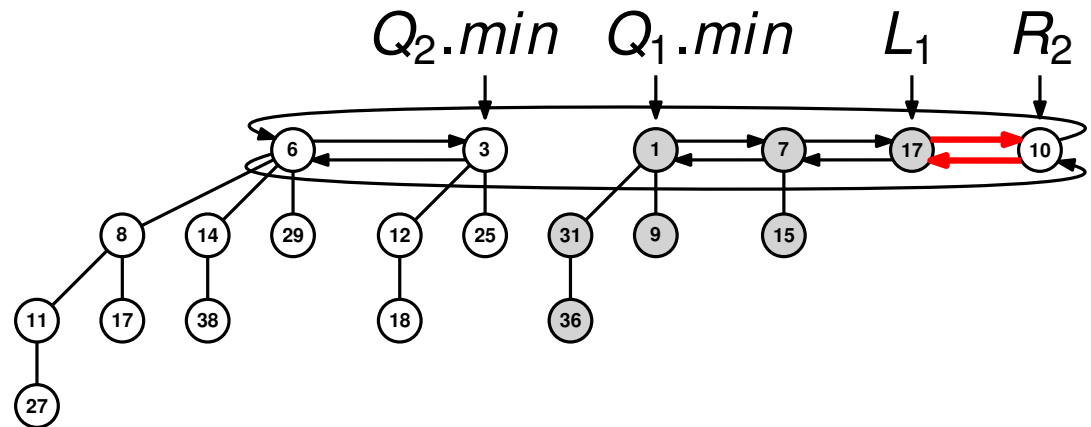
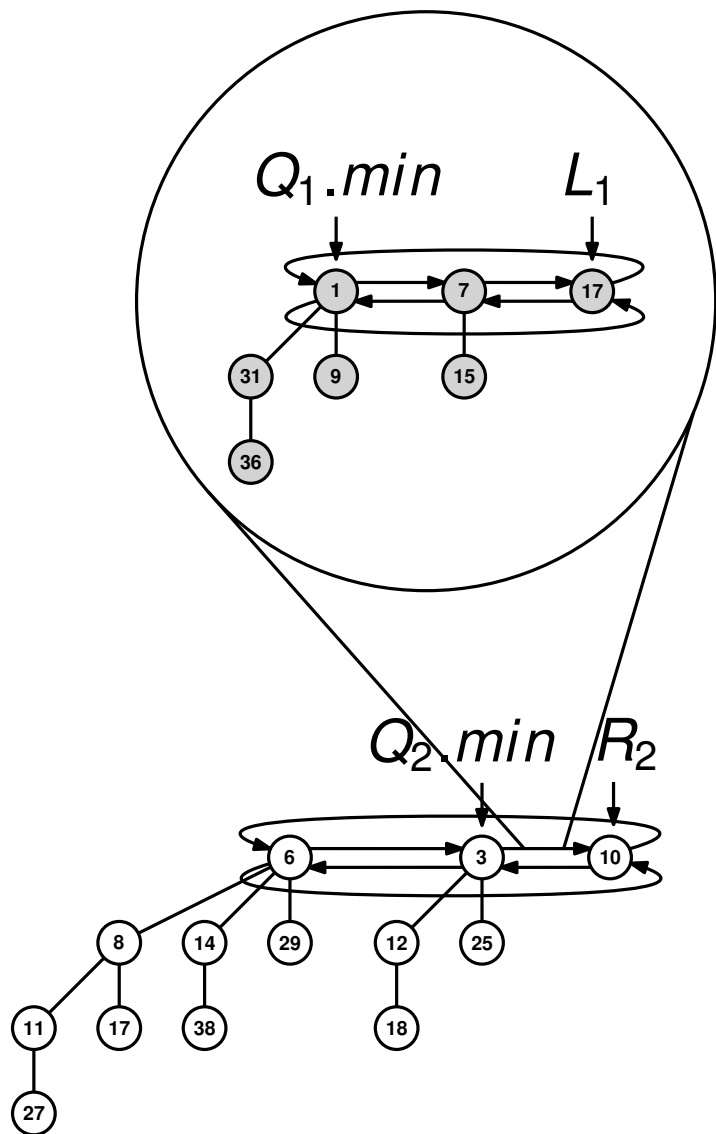
$Q_1.min.left \leftarrow Q_2.min$

if $Q_1.min.key < Q_2.min.key$ **then**

$Q_2.min \leftarrow Q_1.min$

return Q_2

Lazy UNION(Q_1, Q_2)



function UNION(Q_1, Q_2)

$L_1 \leftarrow Q_1.min.left$

$R_2 \leftarrow Q_2.min.right$

$\rightarrow L_1.right \leftarrow R_2$

$\rightarrow R_2.left \leftarrow L_1$

$Q_2.min.right \leftarrow Q_1.min$

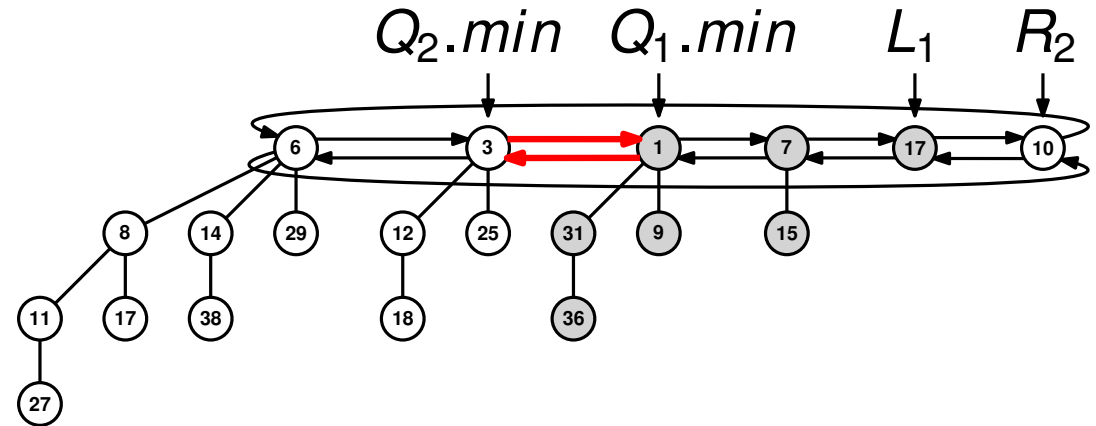
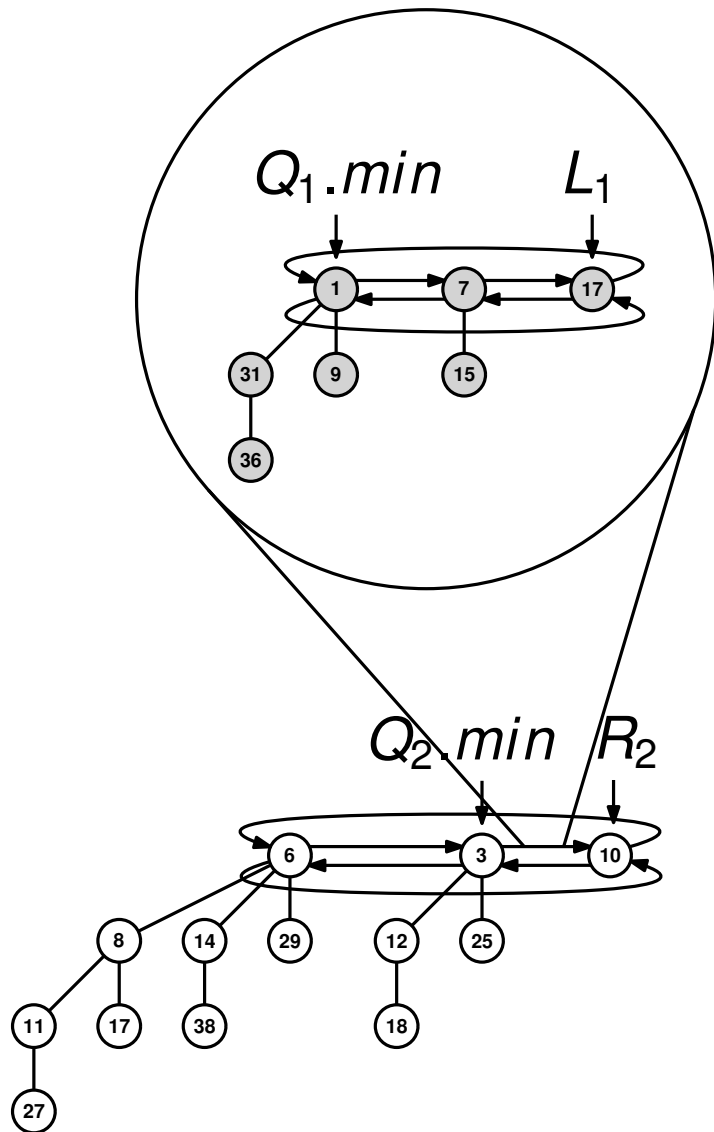
$Q_1.min.left \leftarrow Q_2.min$

if $Q_1.min.key < Q_2.min.key$ **then**

$Q_2.min \leftarrow Q_1.min$

return Q_2

Lazy UNION(Q_1, Q_2)



function UNION(Q_1, Q_2)

$L_1 \leftarrow Q_1.min.left$

$R_2 \leftarrow Q_2.min.right$

$L_1.right \leftarrow R_2$

$R_2.left \leftarrow L_1$

$\rightarrow Q_2.min.right \leftarrow Q_1.min$

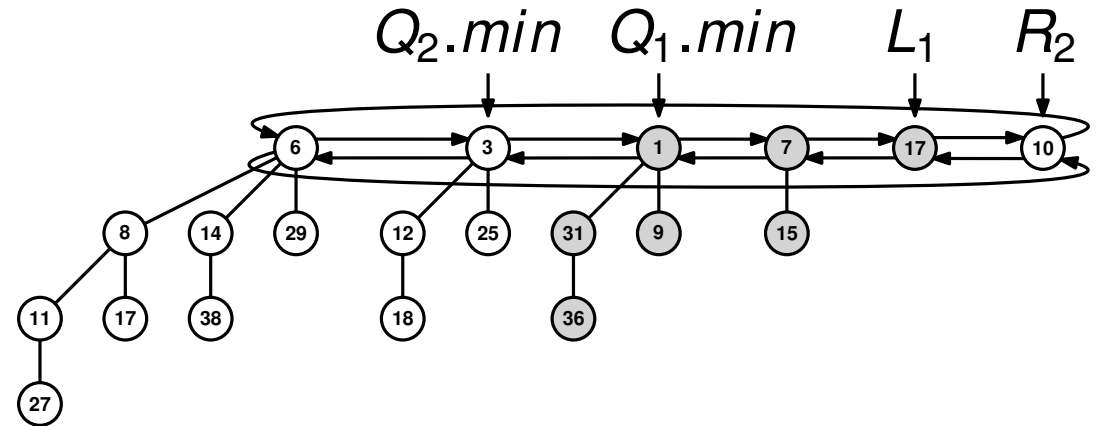
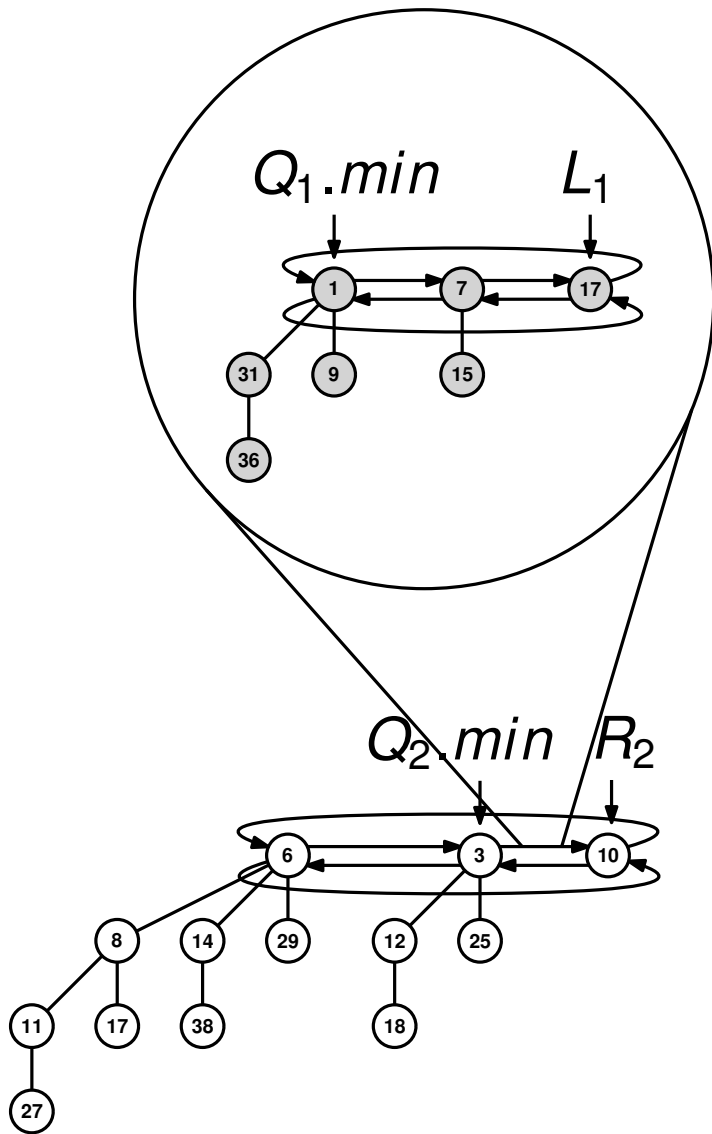
$\rightarrow Q_1.min.left \leftarrow Q_2.min$

if $Q_1.min.key < Q_2.min.key$ **then**

$Q_2.min \leftarrow Q_1.min$

return Q_2

Lazy UNION(Q_1, Q_2)



function UNION(Q_1, Q_2)

$L_1 \leftarrow Q_1.min.left$

$R_2 \leftarrow Q_2.min.right$

$L_1.right \leftarrow R_2$

$R_2.left \leftarrow L_1$

$Q_2.min.right \leftarrow Q_1.min$

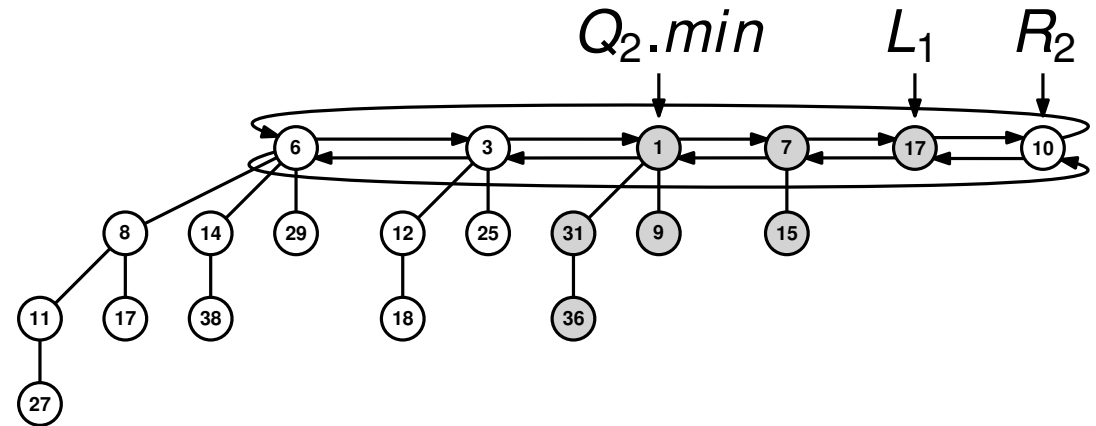
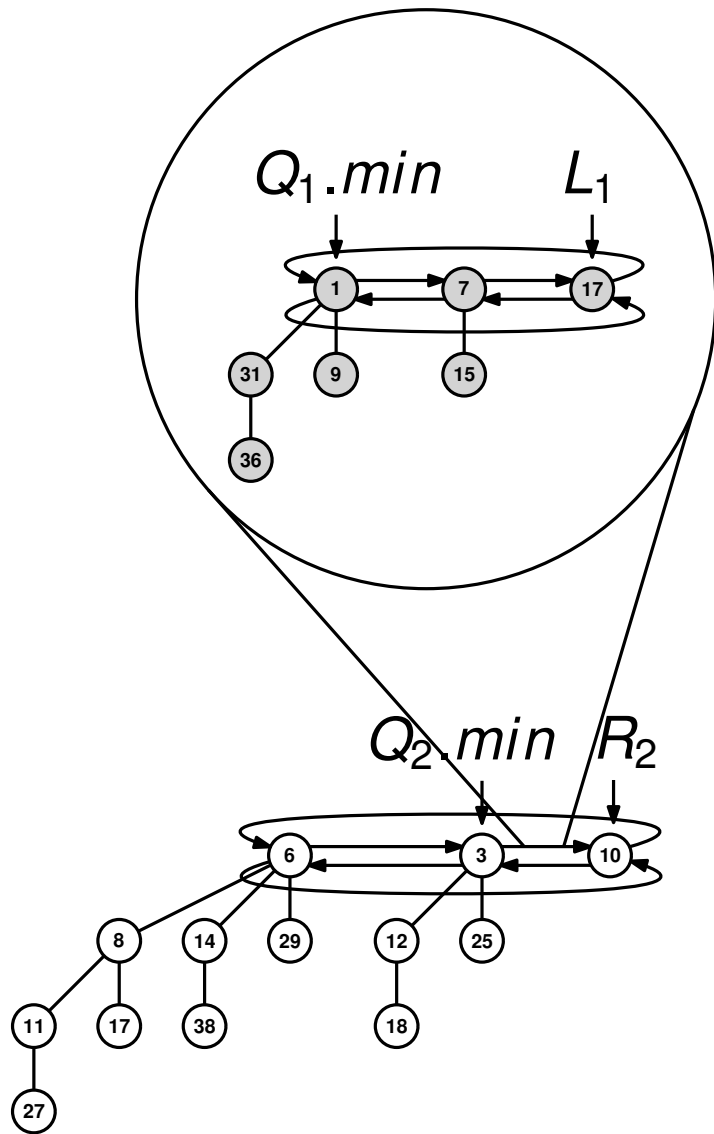
$Q_1.min.left \leftarrow Q_2.min$

→ if $Q_1.min.key < Q_2.min.key$ **then**

→ $Q_2.min \leftarrow Q_1.min$

return Q_2

Lazy UNION(Q_1, Q_2)



function UNION(Q_1, Q_2)

$L_1 \leftarrow Q_1.min.left$

$R_2 \leftarrow Q_2.min.right$

$L_1.right \leftarrow R_2$

$R_2.left \leftarrow L_1$

$Q_2.min.right \leftarrow Q_1.min$

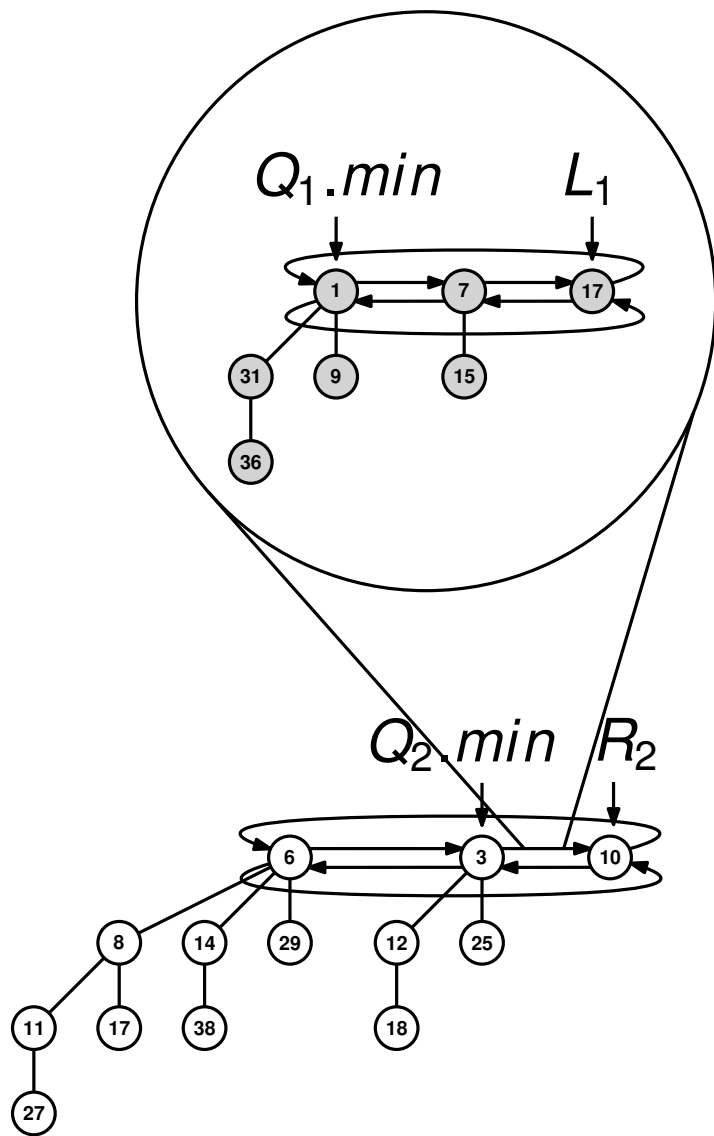
$Q_1.min.left \leftarrow Q_2.min$

→ if $Q_1.min.key < Q_2.min.key$ **then**

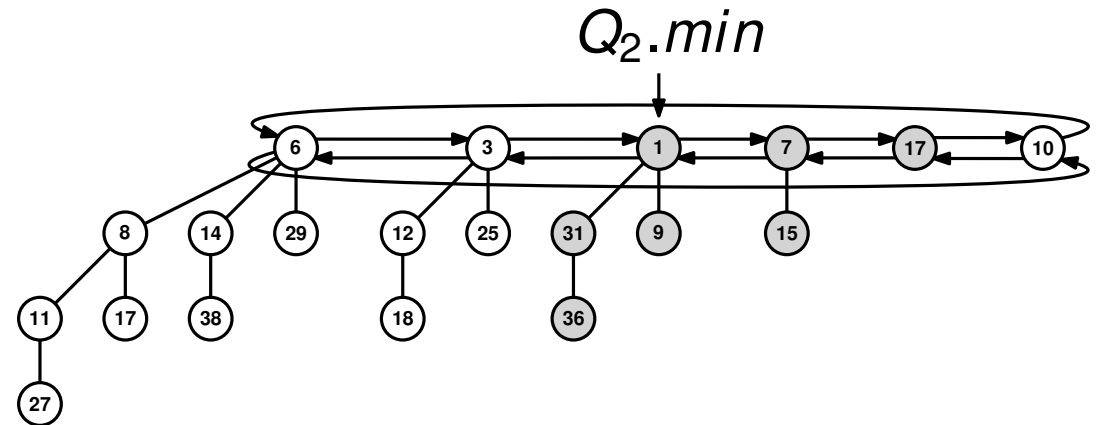
→ $Q_2.min \leftarrow Q_1.min$

return Q_2

Lazy UNION(Q_1, Q_2)



$O(1)$ time worst-case



function UNION(Q_1, Q_2)

$L_1 \leftarrow Q_1.min.left$

$R_2 \leftarrow Q_2.min.right$

$L_1.right \leftarrow R_2$

$R_2.left \leftarrow L_1$

$Q_2.min.right \leftarrow Q_1.min$

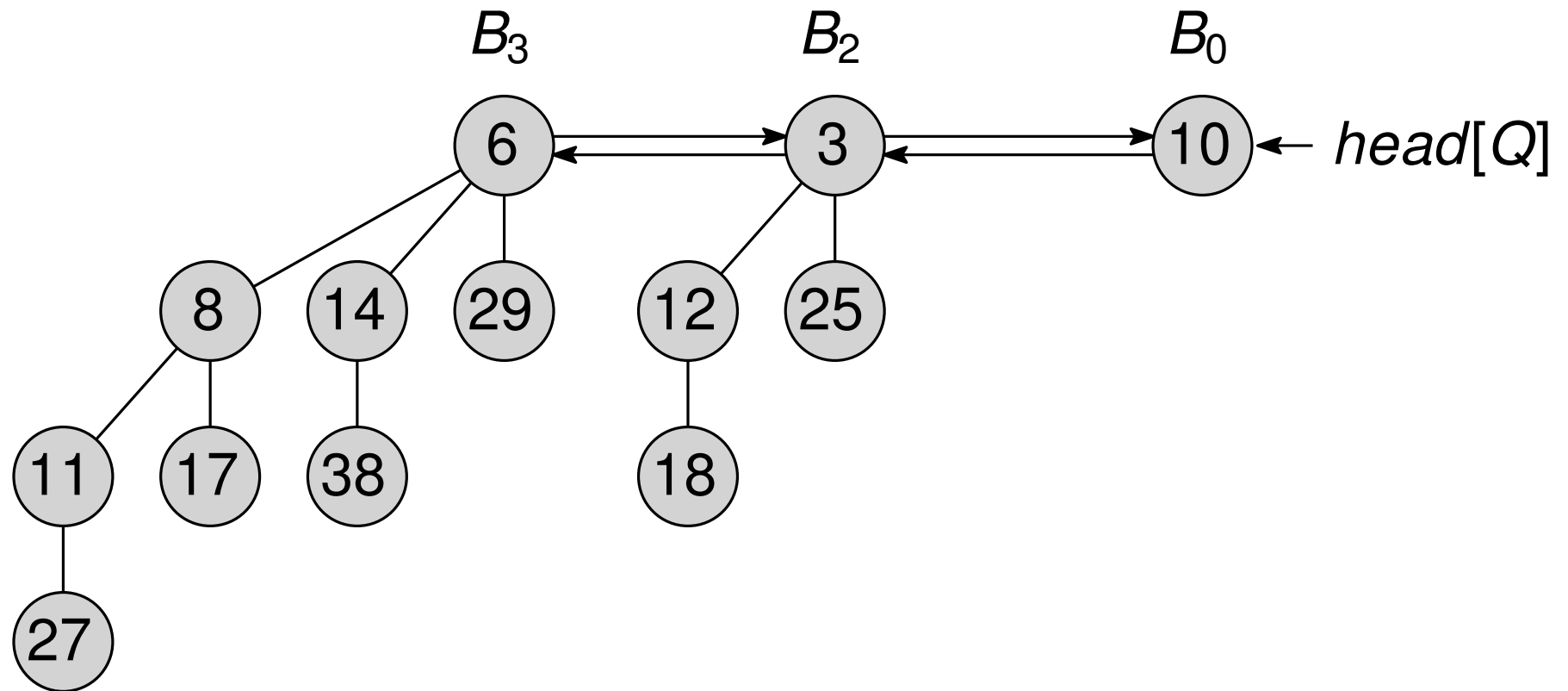
$Q_1.min.left \leftarrow Q_2.min$

if $Q_1.min.key < Q_2.min.key$ **then**

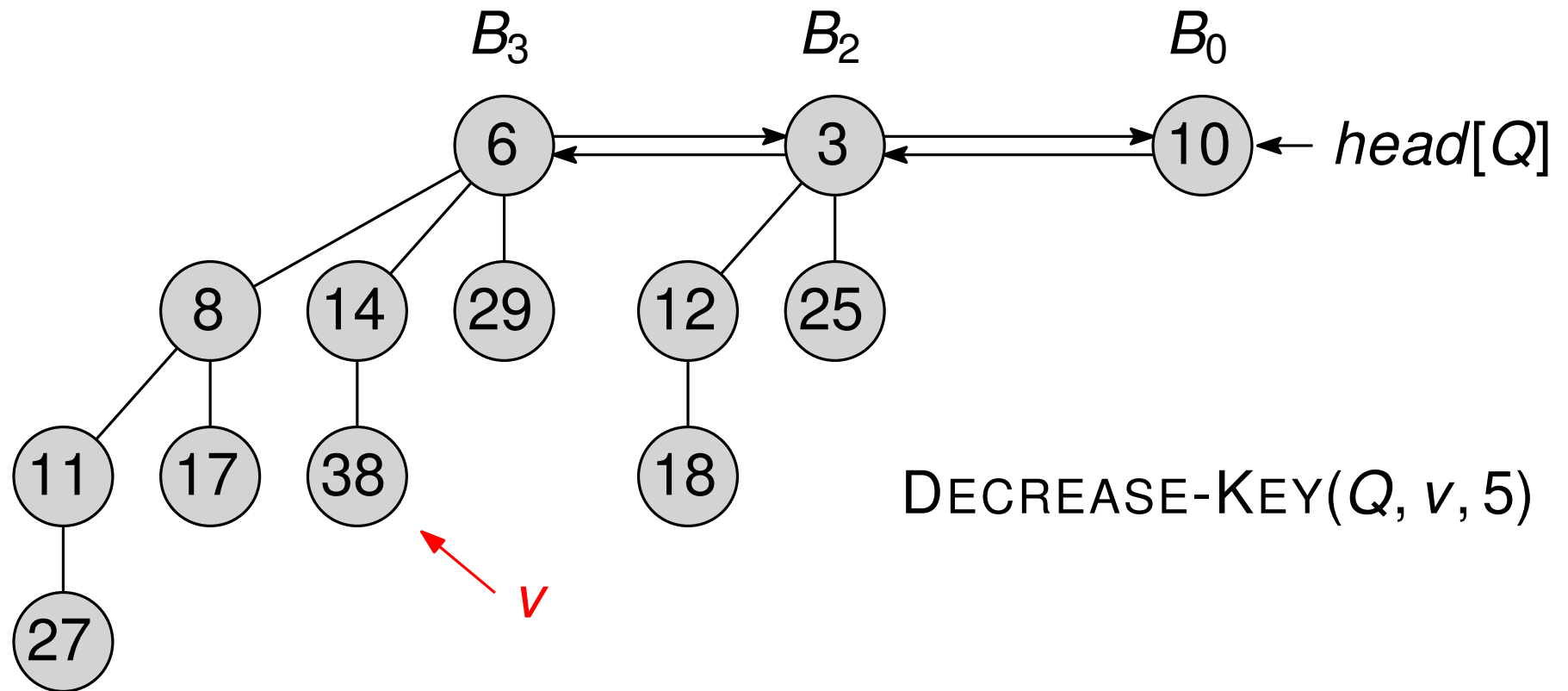
$Q_2.min \leftarrow Q_1.min$

return Q_2

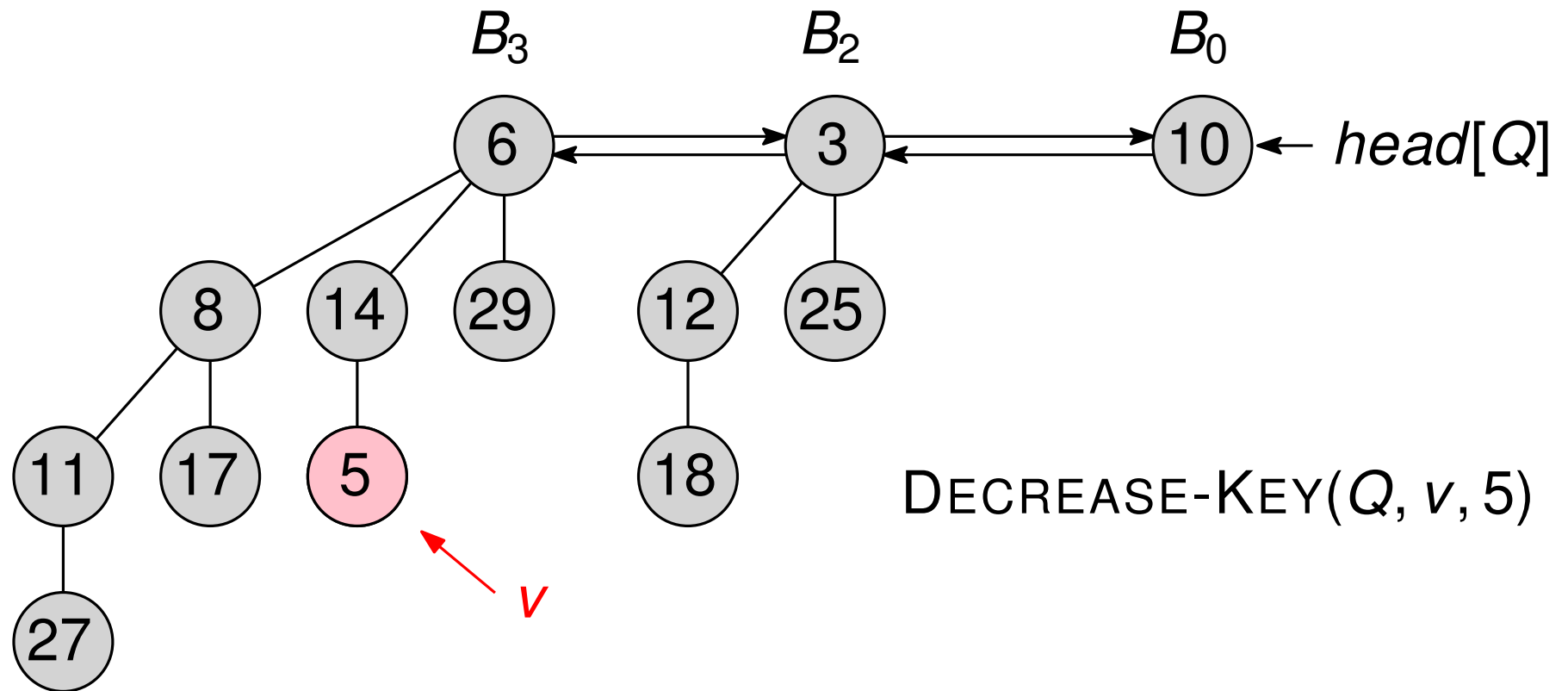
DECREASE-KEY(Q, v, k)



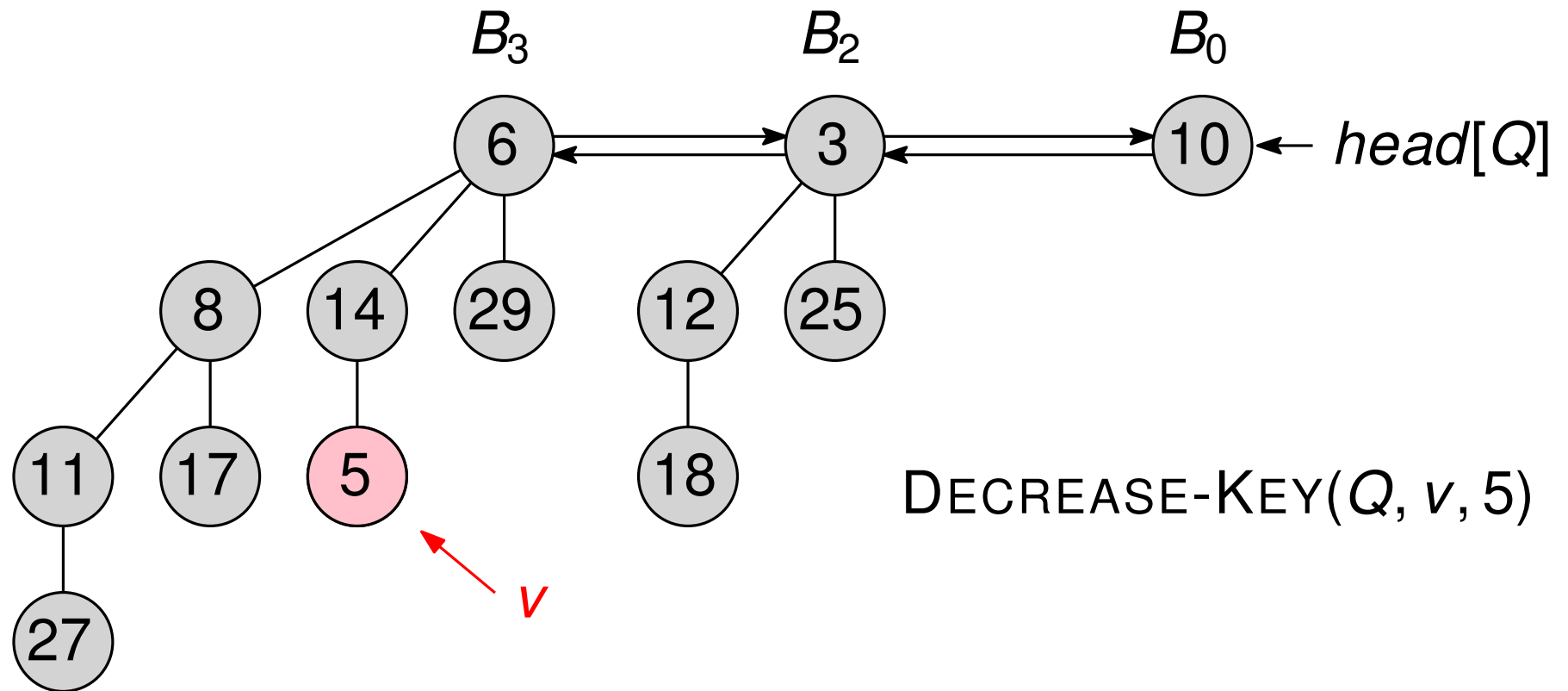
DECREASE-KEY(Q, v, k)



DECREASE-KEY(Q, v, k)

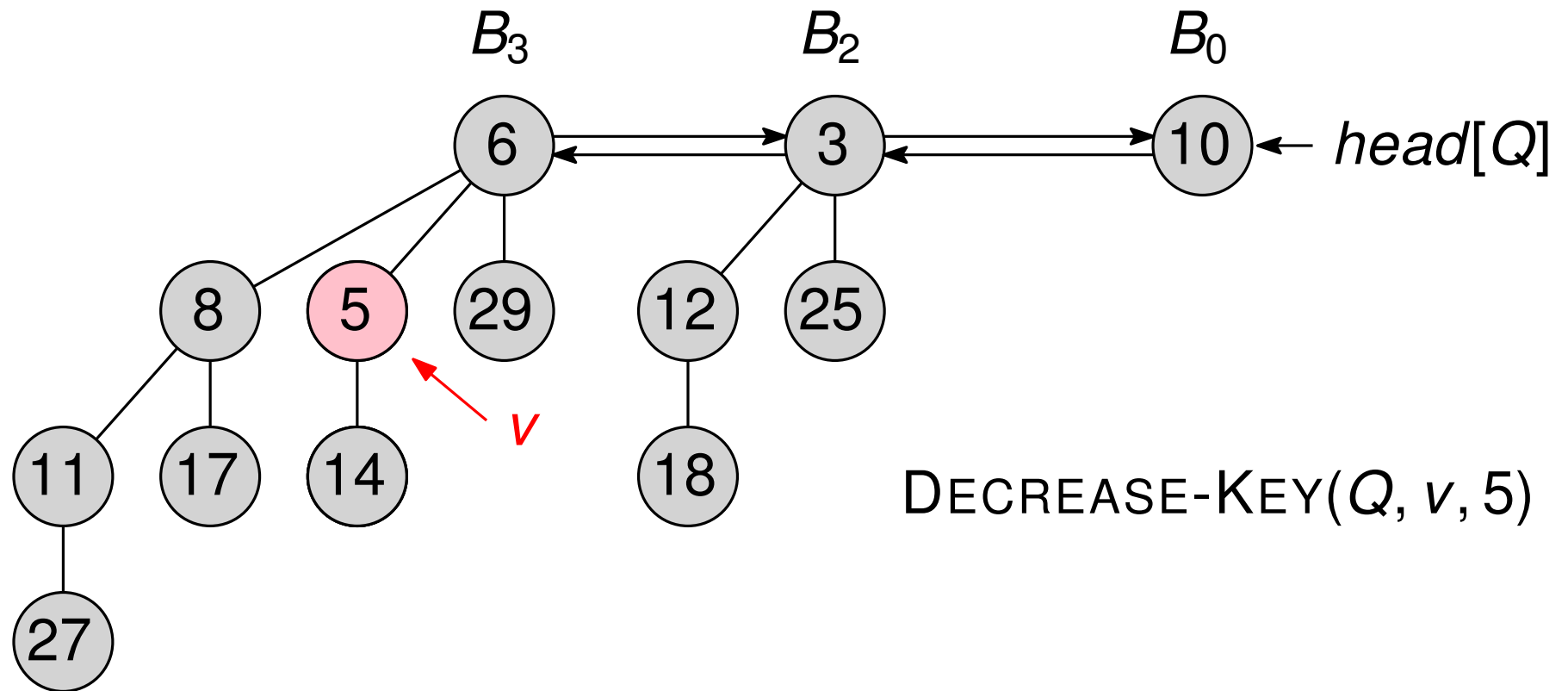


DECREASE-KEY(Q, v, k)



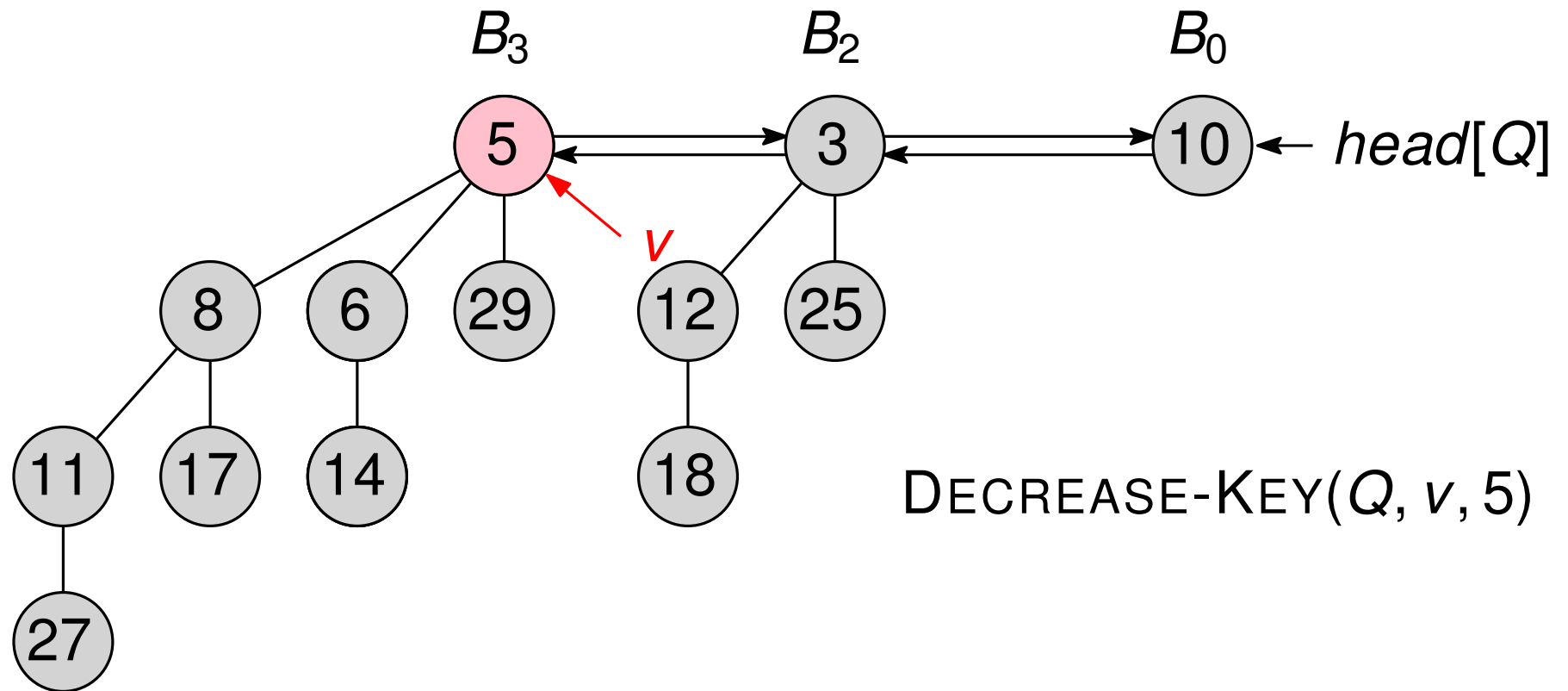
- Fix heap order

DECREASE-KEY(Q, v, k)



- Fix heap order

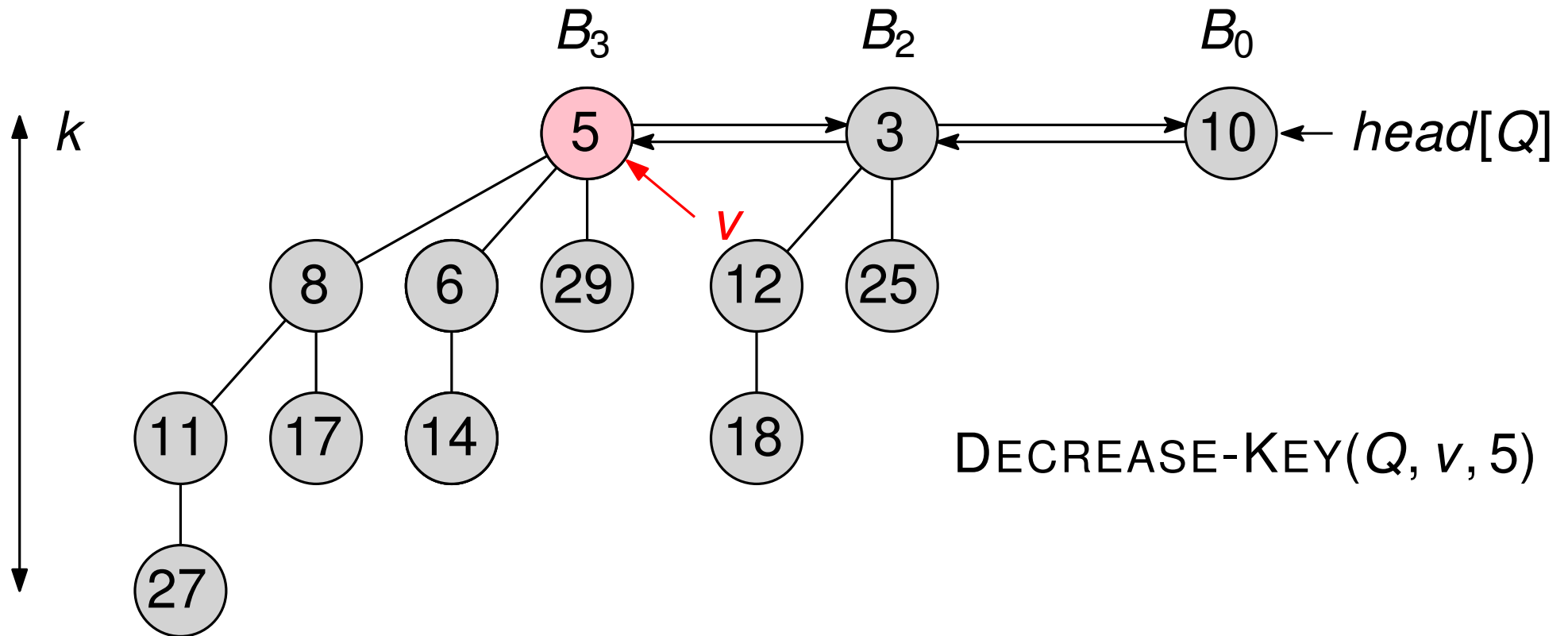
DECREASE-KEY(Q, v, k)



- Fix heap order

DECREASE-KEY(Q, v, k)

Depth of B_k is $k \leq \log n$



DECREASE-KEY($Q, v, 5$)

- Fix heap order

Reminder: EXTRACT-MIN(Q)

function EXTRACT-MIN(Q)

$x = \text{MINIMUM}(Q)$

$Q' = \text{MAKE}()$

$Q'.\text{head} = x.\text{leftchild}$

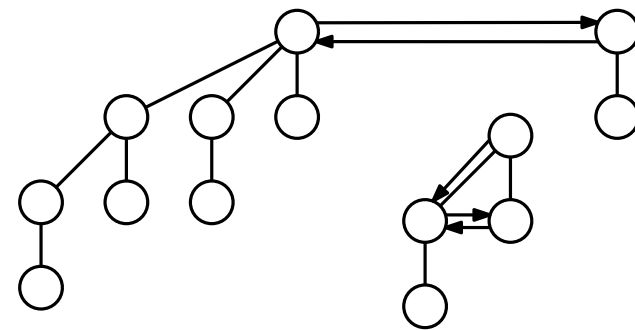
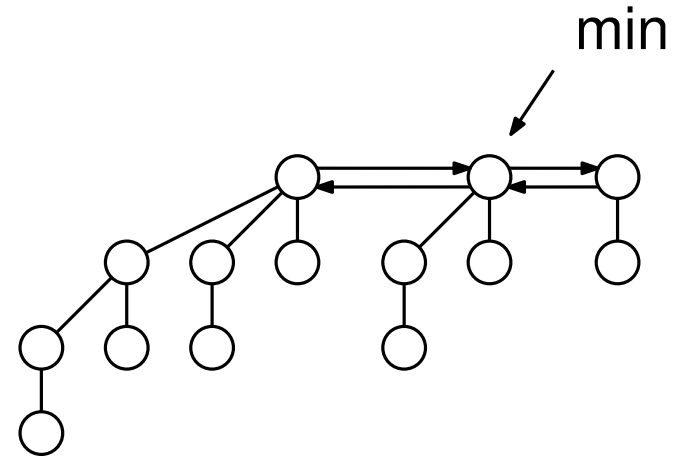
LINKEDLIST-EXTRACT(x)

for each child y of x **do**

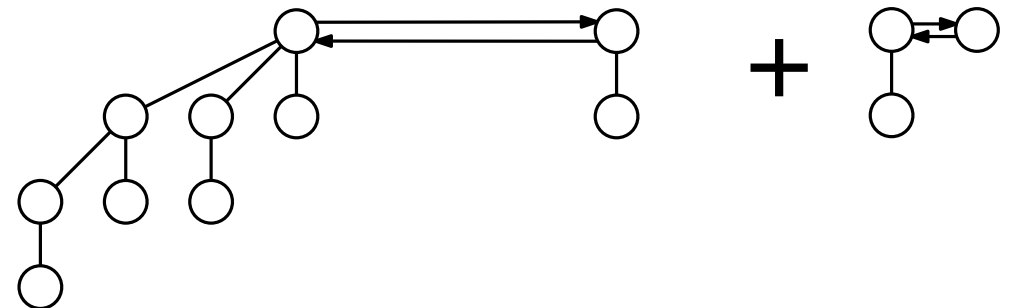
$y.\text{parent} = \text{NIL}$

$Q = \text{UNION}(Q, Q')$

return x



min
○



Analysis: $O(\log n)$

Lazy EXTRACT-MIN(Q)

function EXTRACT-MIN(Q)

$v = \text{MINIMUM}(Q)$

$c = v.\text{child}$

$L_v = v.\text{left}$

$R_v = v.\text{right}$

if $c \neq \text{NIL}$ **then**

$L_c = c.\text{left}$

$L_v.\text{right} = c$

$c.\text{left} = L_v$

$R_v.\text{left} = L_c$

$L_c.\text{right} = R_v$

else

$L_v.\text{right} = R_v$

$R_v.\text{left} = L_v$

$Q.\text{min} = R_v \triangleright$ arbitrary head

CONSOLIDATE(Q)

return v

Lazy EXTRACT-MIN(Q)

function EXTRACT-MIN(Q)

$v = \text{MINIMUM}(Q)$

$c = v.\text{child}$

$L_v = v.\text{left}$

$R_v = v.\text{right}$

if $c \neq \text{NIL}$ **then**

$L_c = c.\text{left}$

$L_v.\text{right} = c$

$c.\text{left} = L_v$

$R_v.\text{left} = L_c$

$L_c.\text{right} = R_v$

else

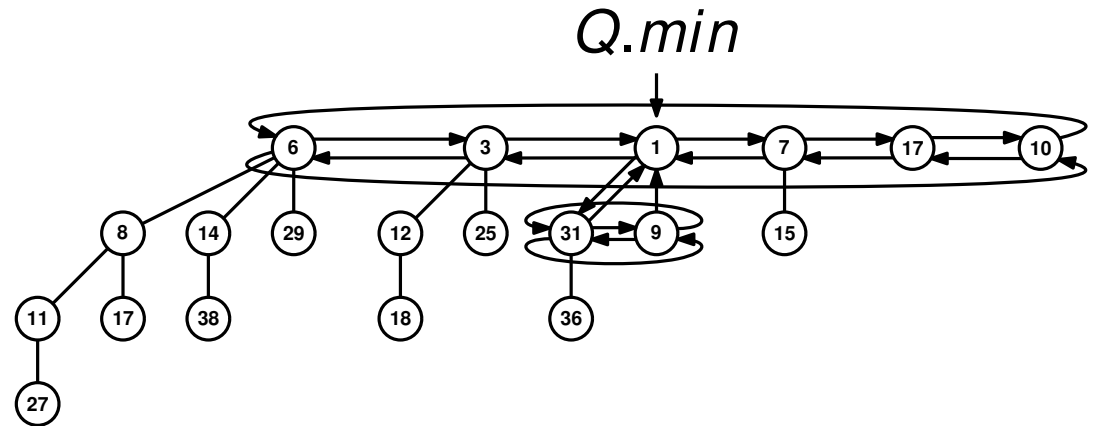
$L_v.\text{right} = R_v$

$R_v.\text{left} = L_v$

$Q.\text{min} = R_v \triangleright$ arbitrary head

CONSOLIDATE(Q)

return v



Lazy EXTRACT-MIN(Q)

function EXTRACT-MIN(Q)

$v = \text{MINIMUM}(Q)$

$c = v.\text{child}$

$L_v = v.\text{left}$

$R_v = v.\text{right}$

if $c \neq \text{NIL}$ **then**

→ $L_c = c.\text{left}$

$L_v.\text{right} = c$

$c.\text{left} = L_v$

$R_v.\text{left} = L_c$

$L_c.\text{right} = R_v$

else

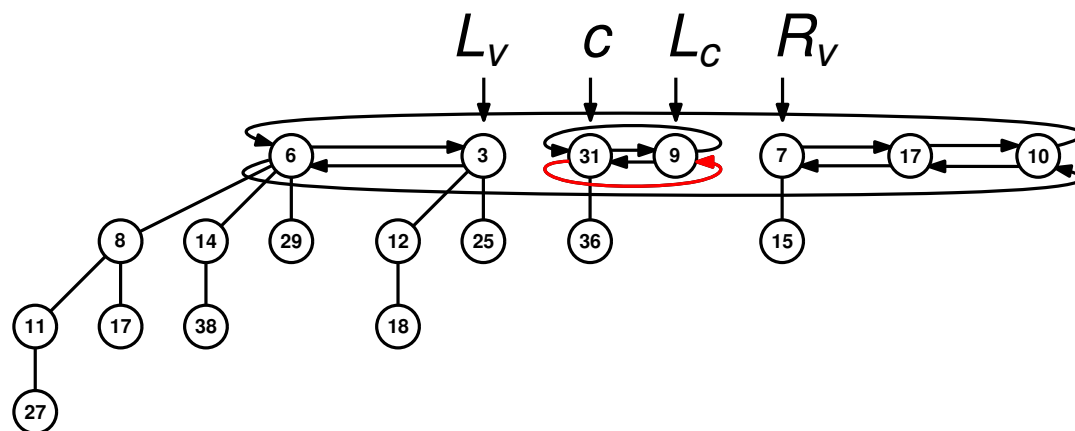
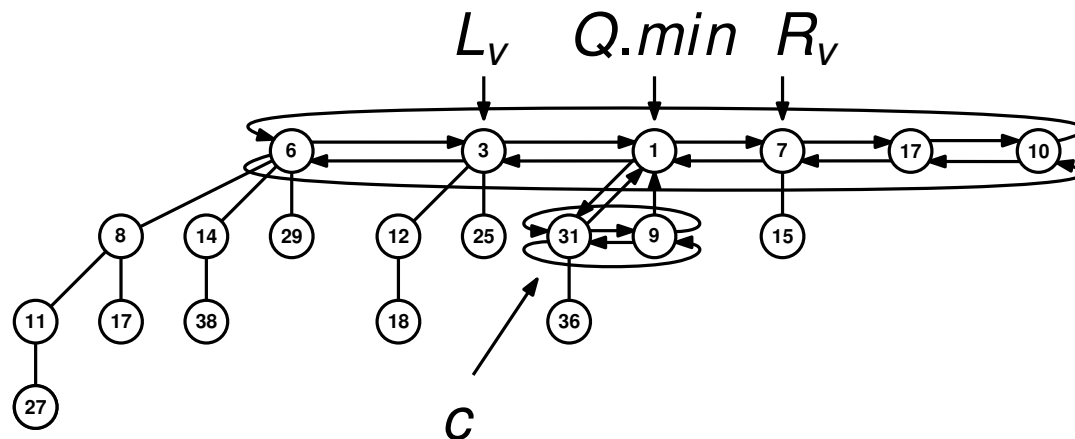
$L_v.\text{right} = R_v$

$R_v.\text{left} = L_v$

$Q.\text{min} = R_v \triangleright$ arbitrary head

CONSOLIDATE(Q)

return v



Lazy EXTRACT-MIN(Q)

function EXTRACT-MIN(Q)

$v = \text{MINIMUM}(Q)$

$c = v.\text{child}$

$L_v = v.\text{left}$

$R_v = v.\text{right}$

if $c \neq \text{NIL}$ **then**

$L_c = c.\text{left}$

→ $L_v.\text{right} = c$

→ $c.\text{left} = L_v$

$R_v.\text{left} = L_c$

$L_c.\text{right} = R_v$

else

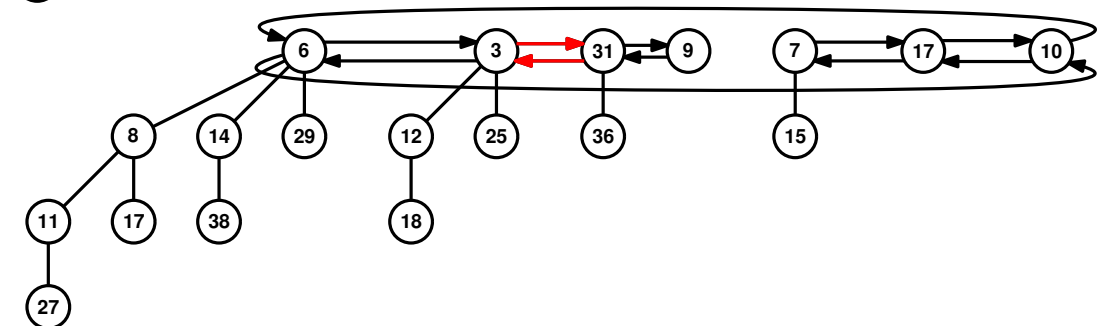
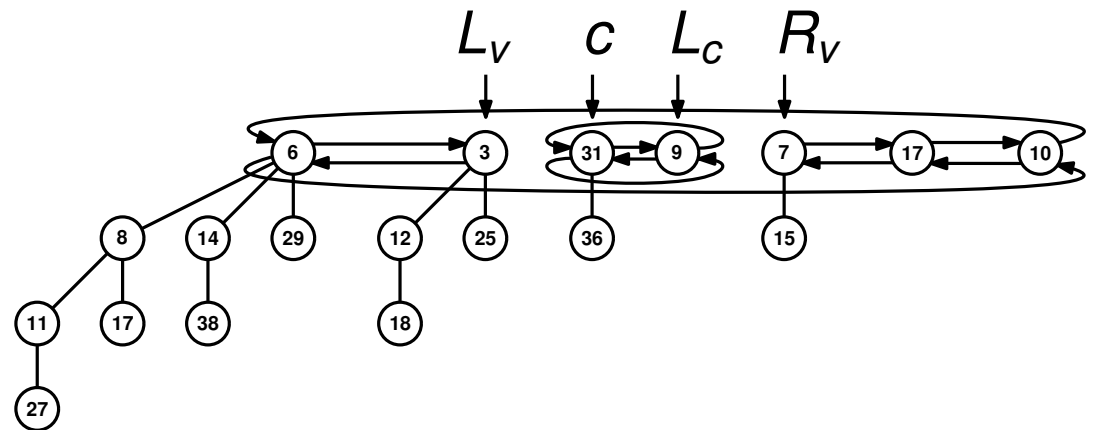
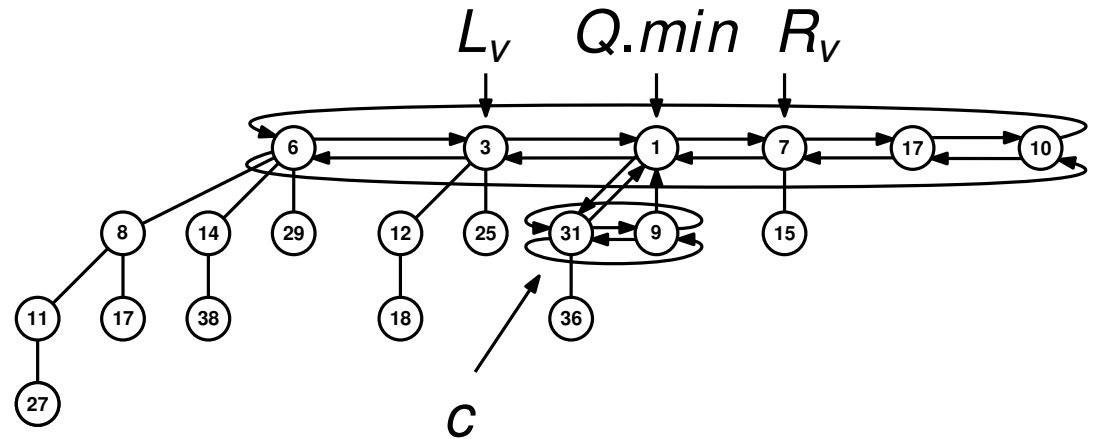
$L_v.\text{right} = R_v$

$R_v.\text{left} = L_v$

$Q.\text{min} = R_v \triangleright$ arbitrary head

CONSOLIDATE(Q)

return v



Lazy EXTRACT-MIN(Q)

function EXTRACT-MIN(Q)

$v = \text{MINIMUM}(Q)$

$c = v.\text{child}$

$L_v = v.\text{left}$

$R_v = v.\text{right}$

if $c \neq \text{NIL}$ **then**

$L_c = c.\text{left}$

$L_v.\text{right} = c$

$c.\text{left} = L_v$

→ $R_v.\text{left} = L_c$

→ $L_c.\text{right} = R_v$

else

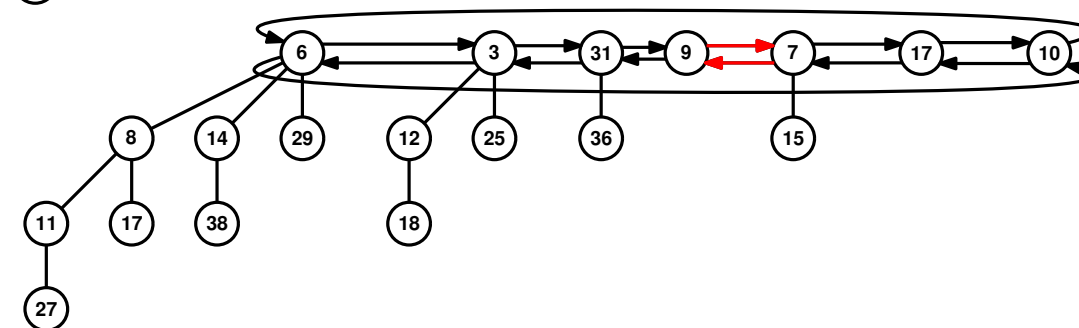
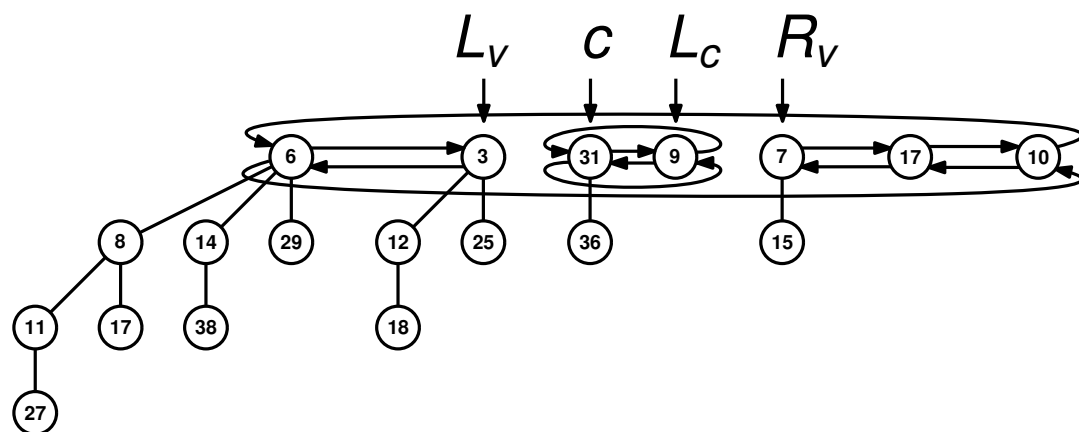
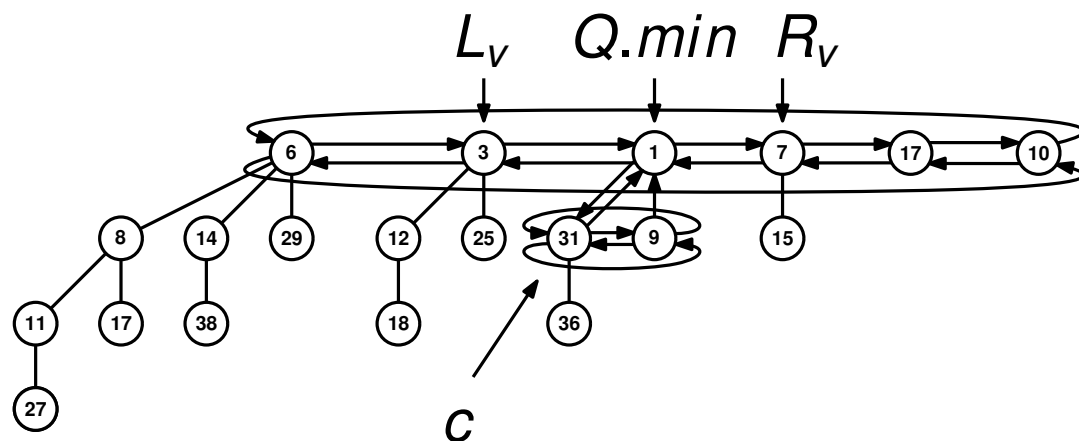
$L_v.\text{right} = R_v$

$R_v.\text{left} = L_v$

$Q.\text{min} = R_v \triangleright$ arbitrary head

CONSOLIDATE(Q)

return v



Lazy EXTRACT-MIN(Q)

function EXTRACT-MIN(Q)

$v = \text{MINIMUM}(Q)$

$c = v.\text{child}$

$L_v = v.\text{left}$

$R_v = v.\text{right}$

if $c \neq \text{NIL}$ **then**

$L_c = c.\text{left}$

$L_v.\text{right} = c$

$c.\text{left} = L_v$

$R_v.\text{left} = L_c$

$L_c.\text{right} = R_v$

else

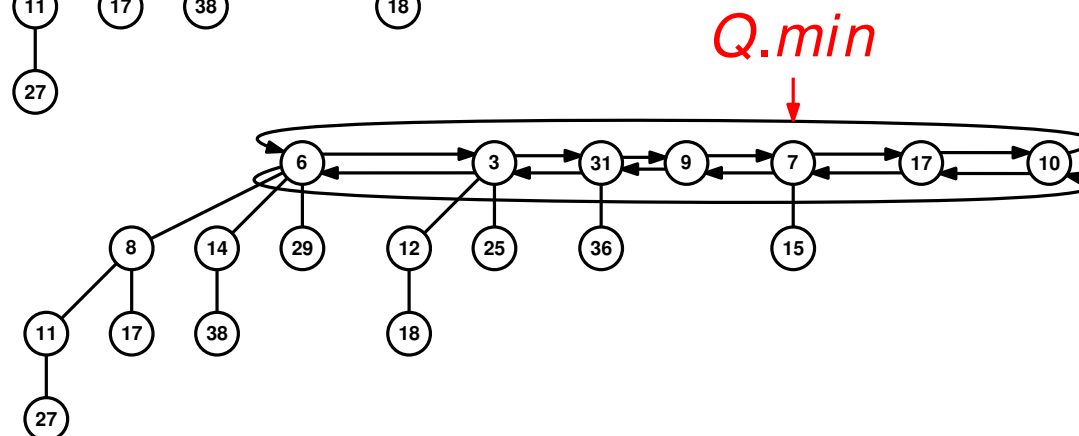
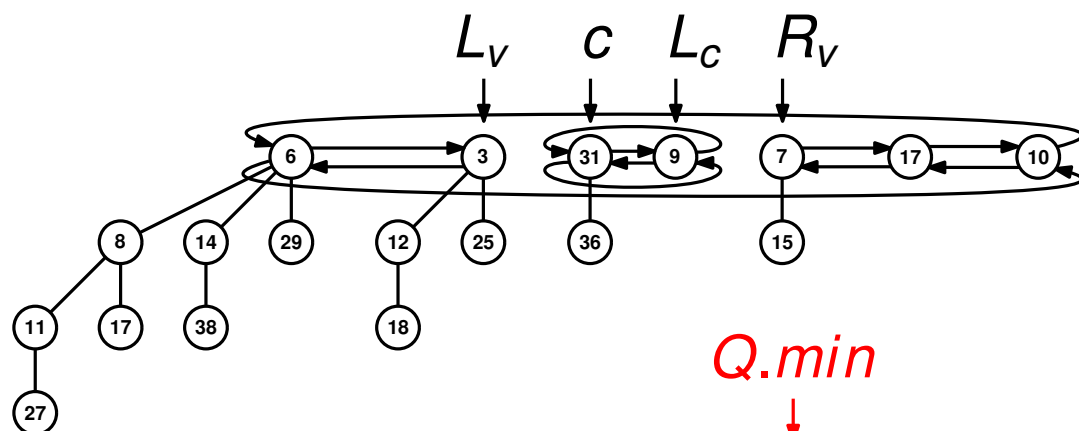
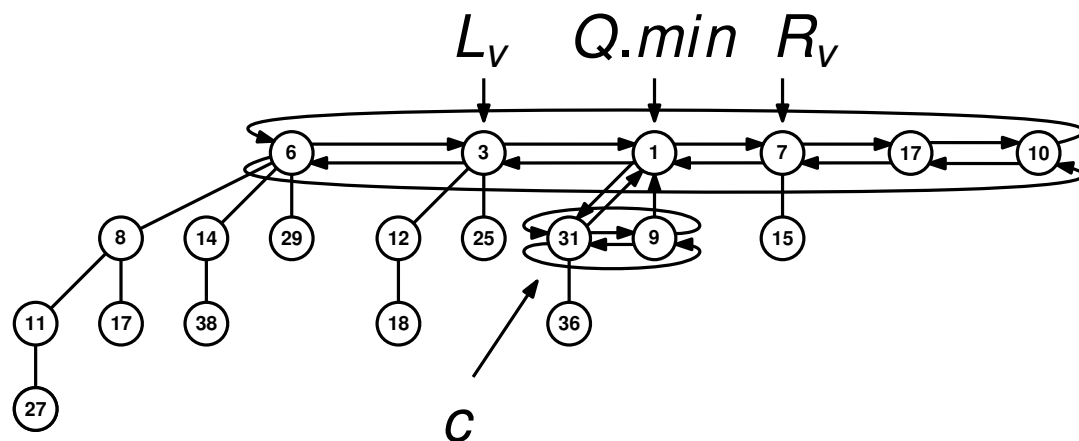
$L_v.\text{right} = R_v$

$R_v.\text{left} = L_v$

$\rightarrow Q.\text{min} = R_v \triangleright$ arbitrary head

CONSOLIDATE(Q)

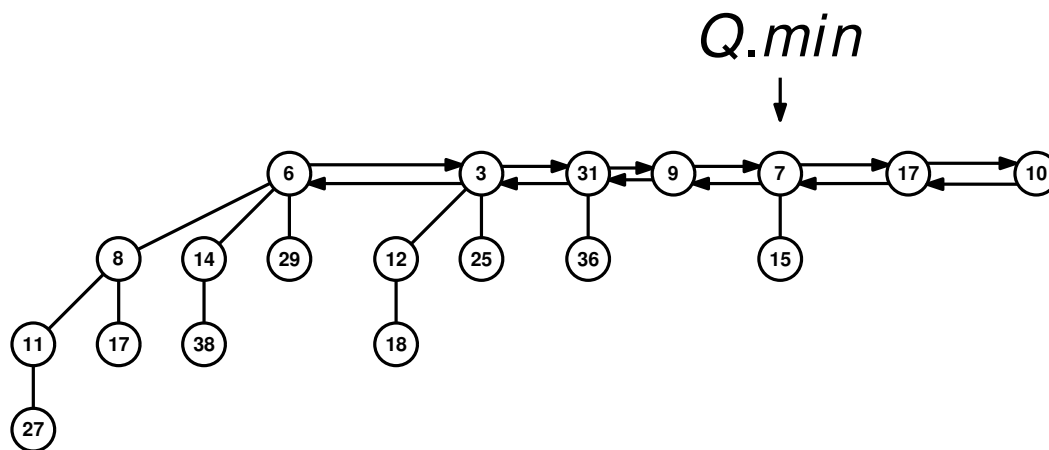
return v



CONSOLIDATE(Q)

Root list:

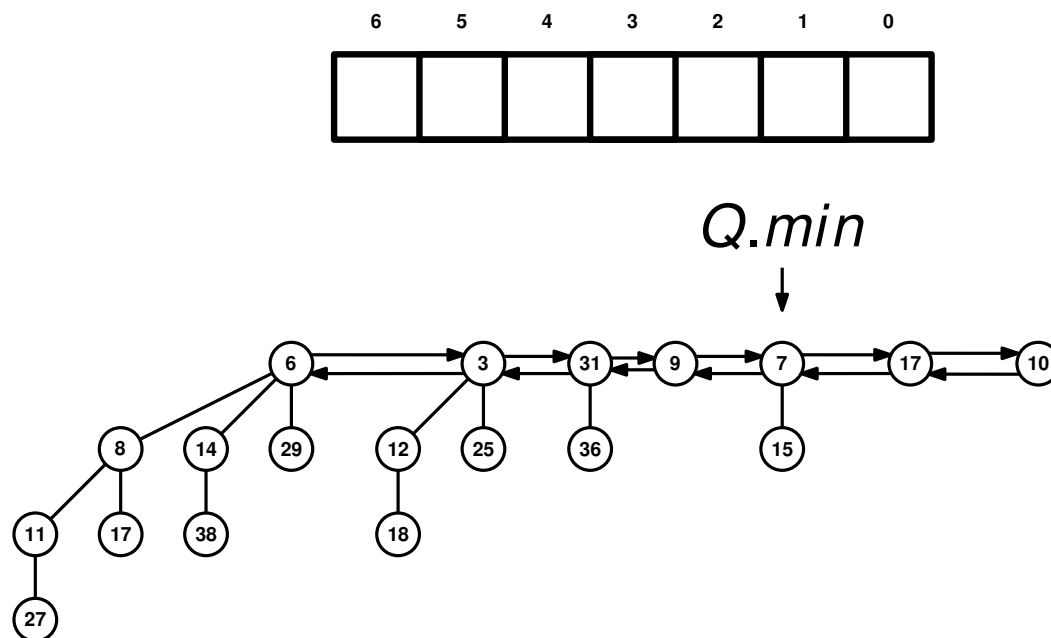
- At most $\log n$ **distinct** tree orders
- Use $\log n$ -sized array with pointers to each tree order



CONSOLIDATE(Q)

Root list:

- At most $\log n$ **distinct** tree orders
- Use $\log n$ -sized array with pointers to each tree order



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

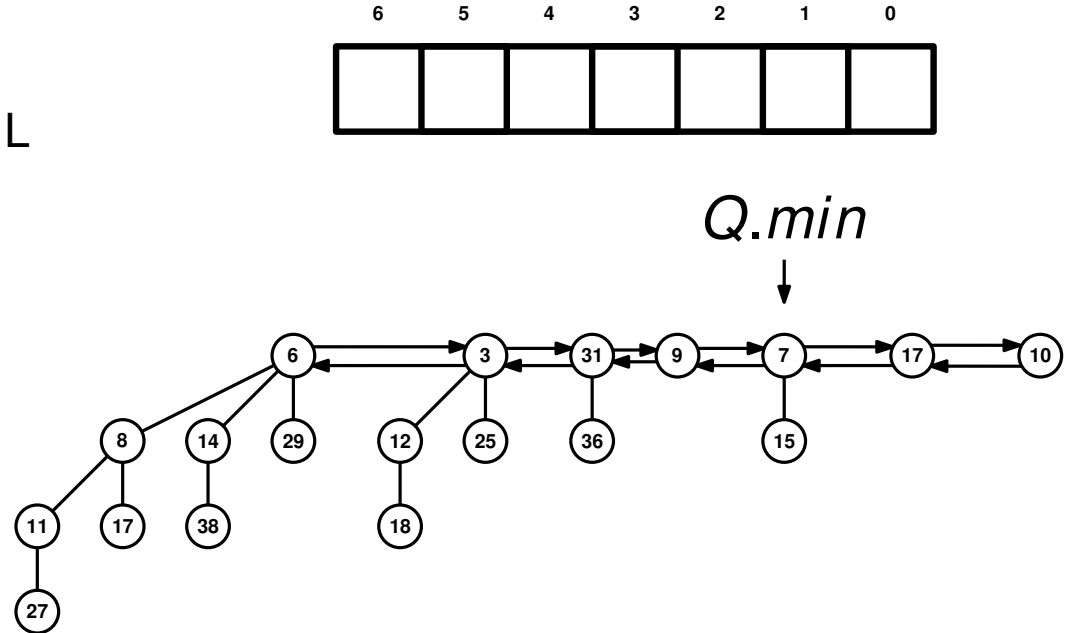
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

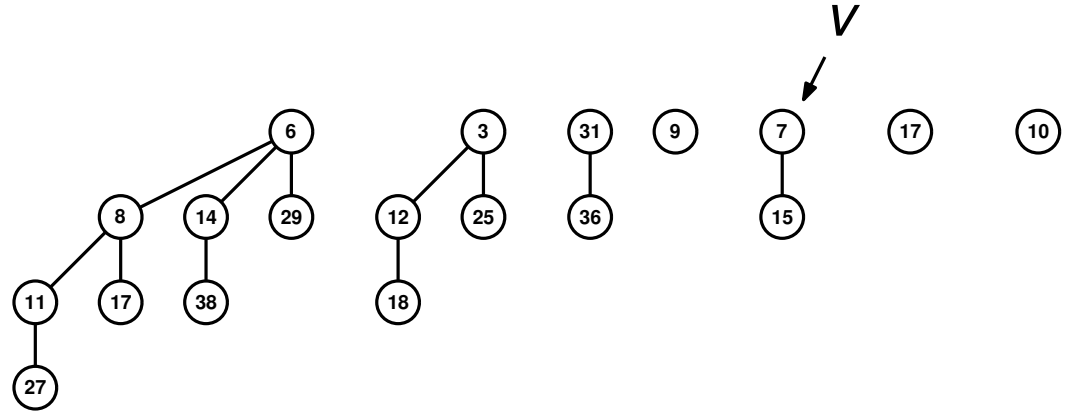
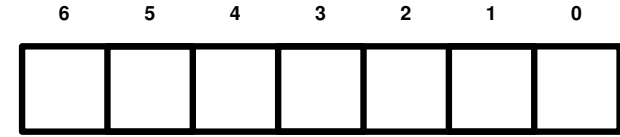
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize log n -sized array A to NIL

for each v in root list **do**

$d = v.degree$

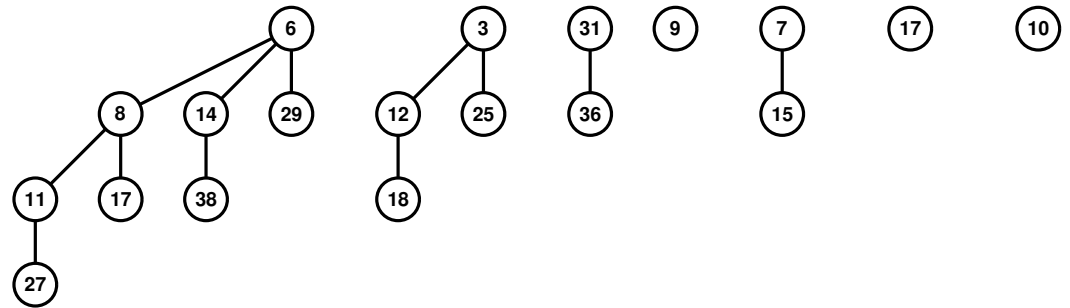
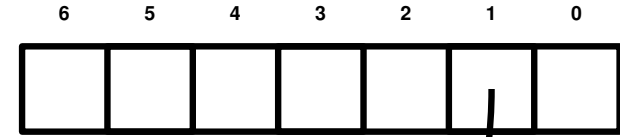
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

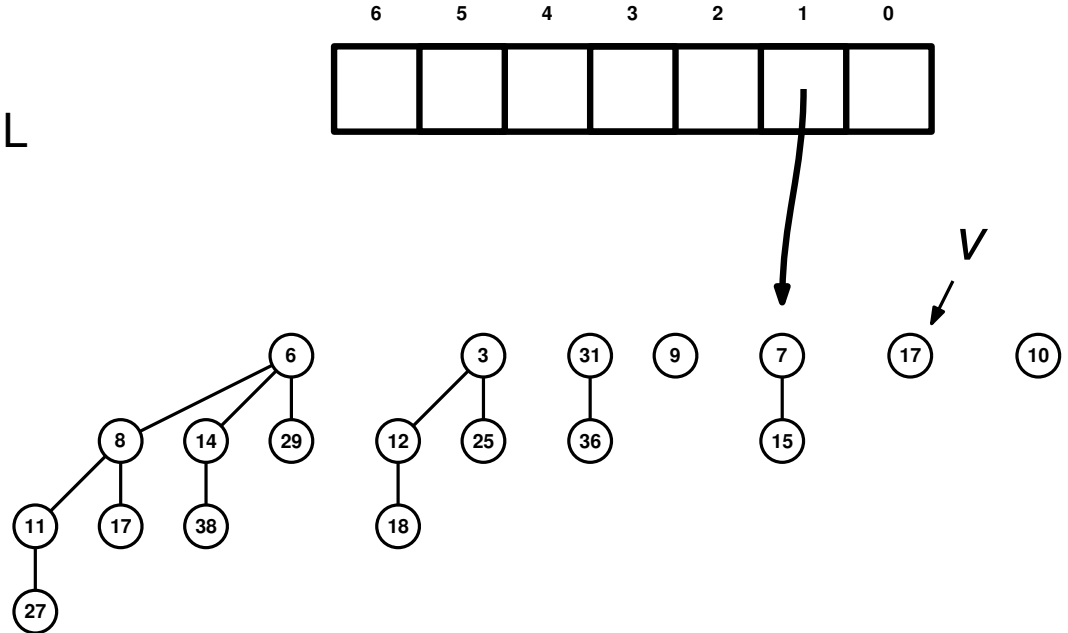
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize log n -sized array A to NIL

for each v in root list **do**

$d = v.degree$

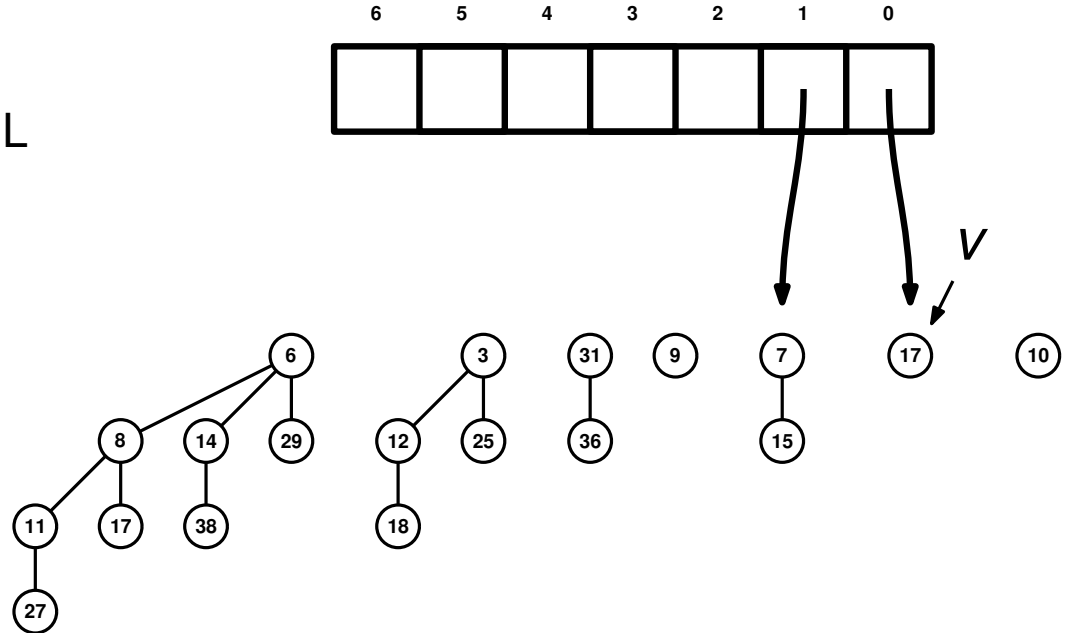
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

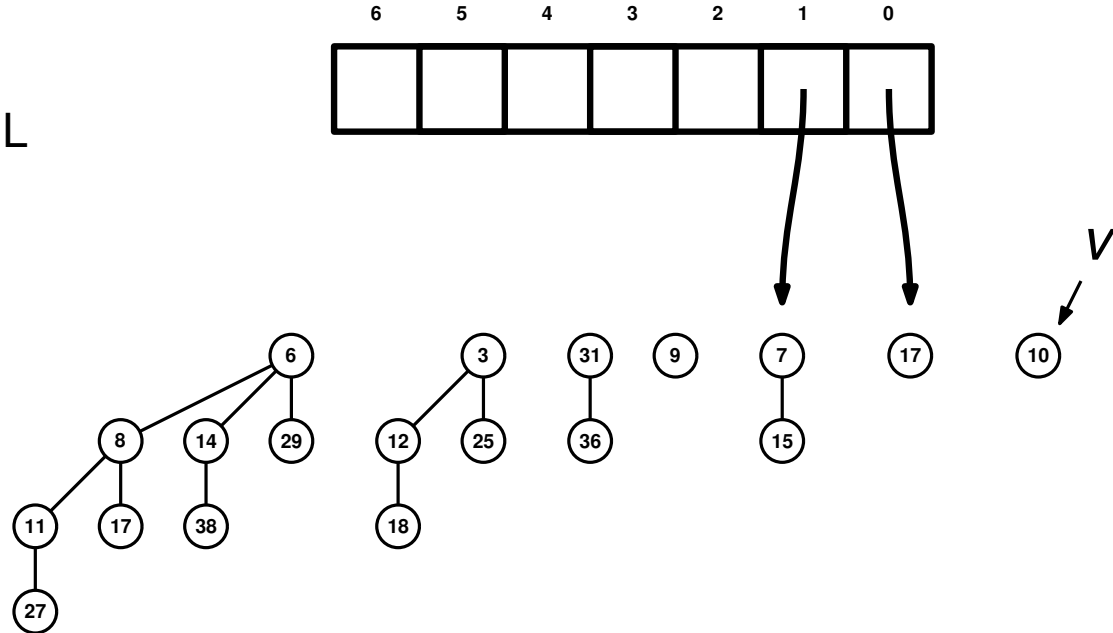
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

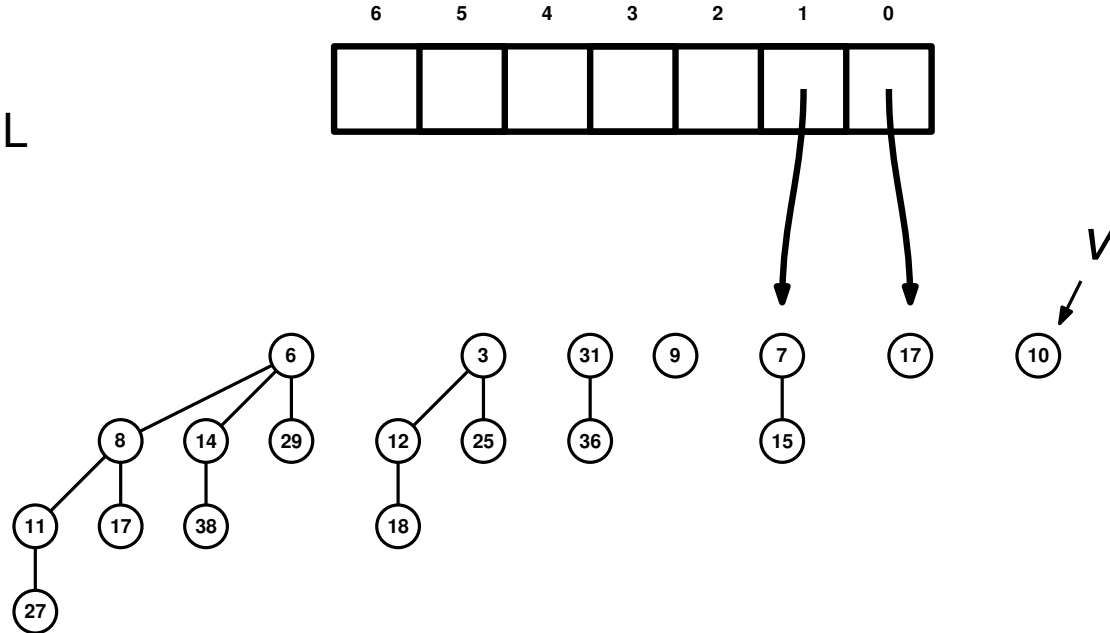
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize log n -sized array A to NIL

for each v in root list **do**

$d = v.degree$

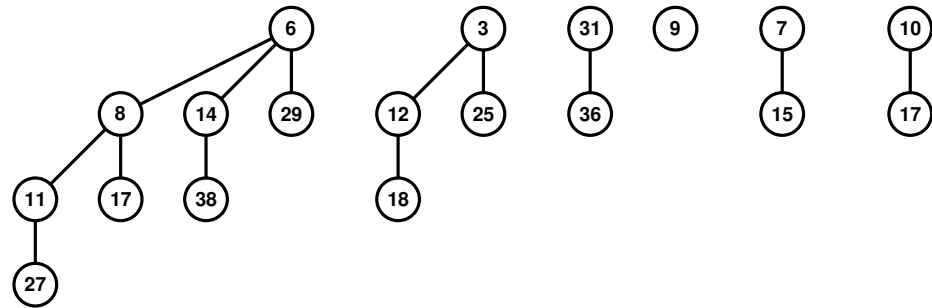
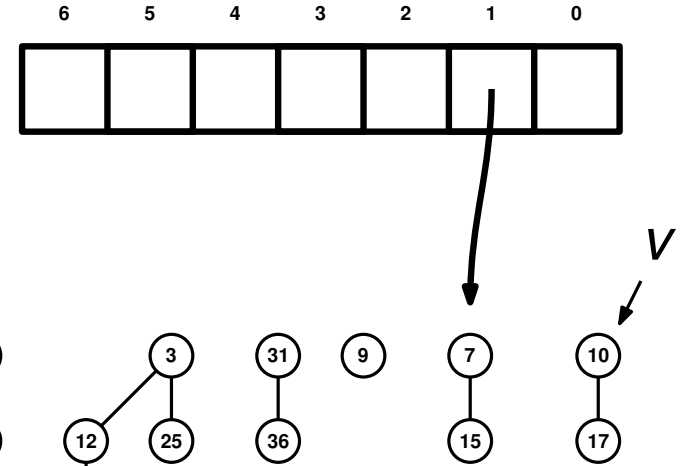
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

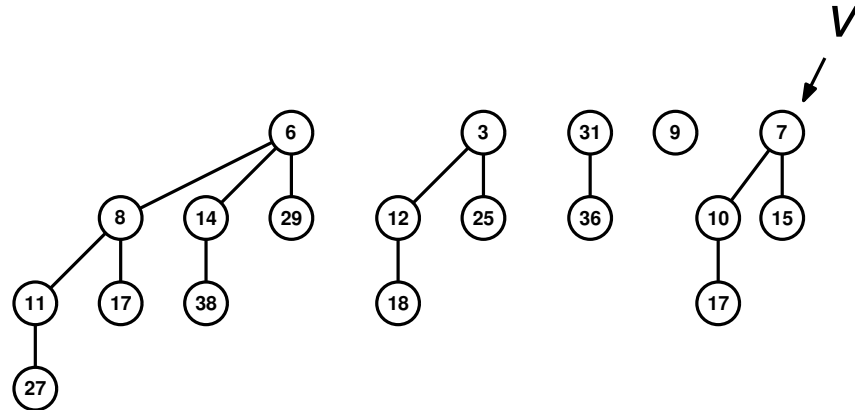
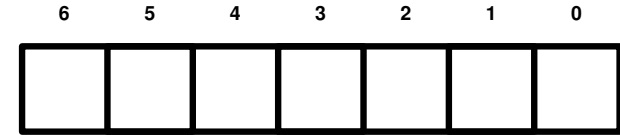
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

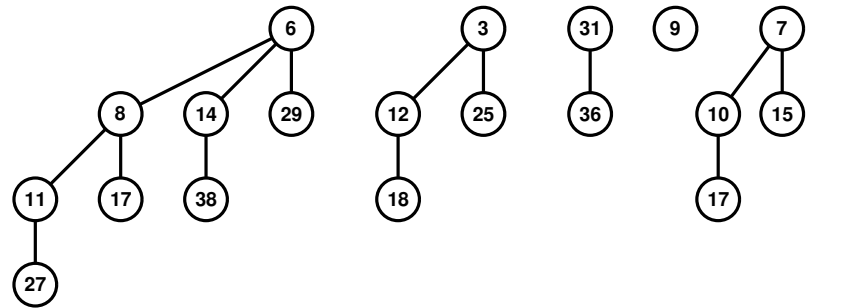
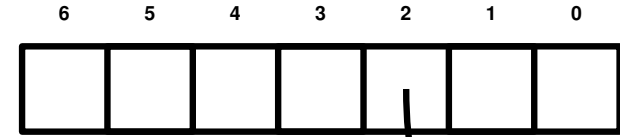
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

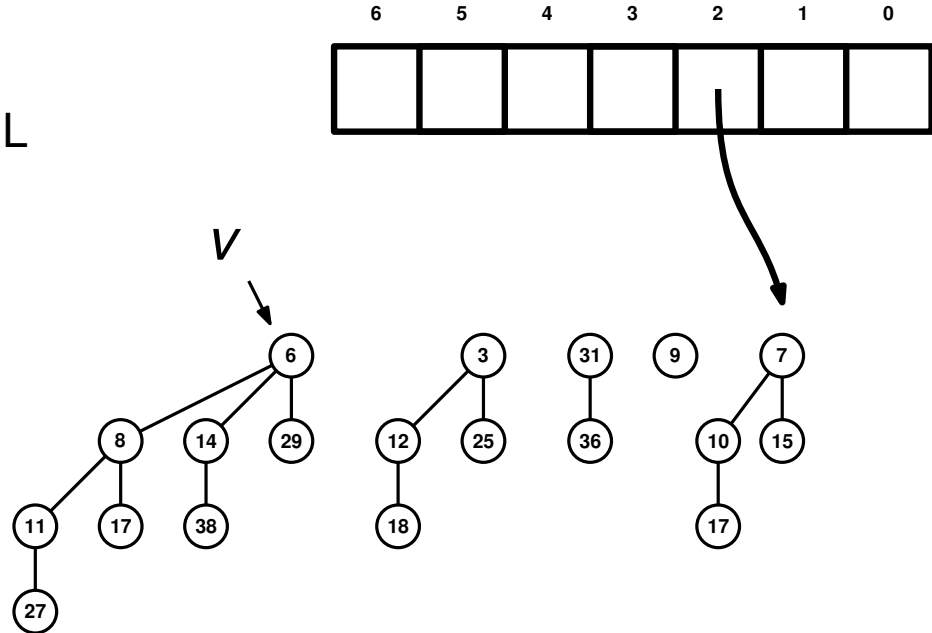
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

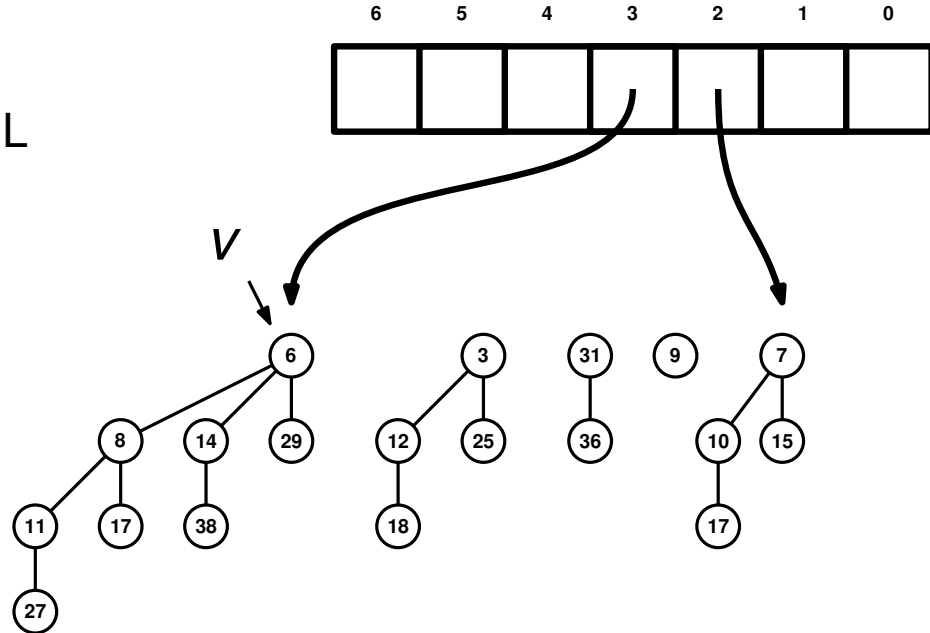
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

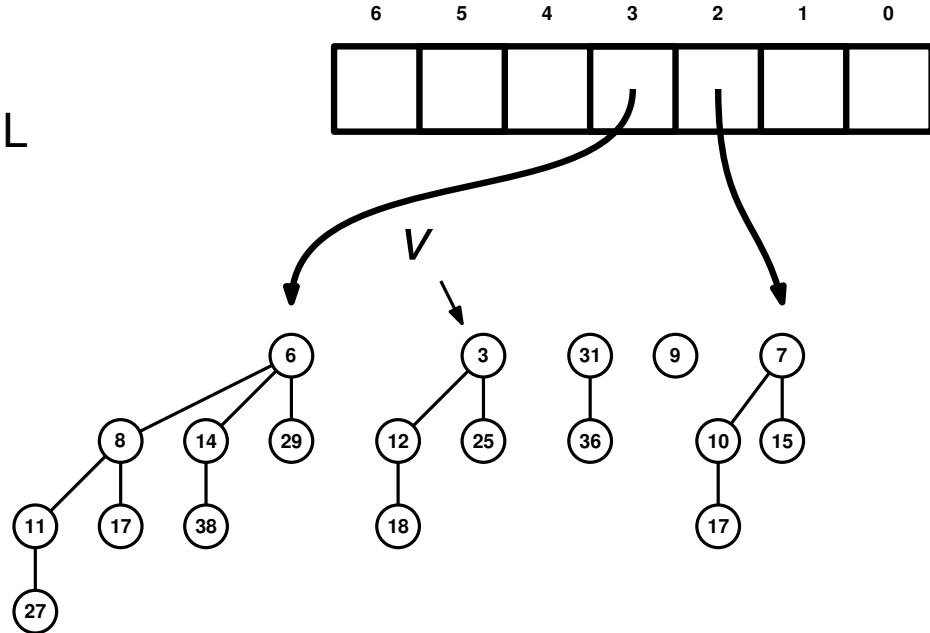
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

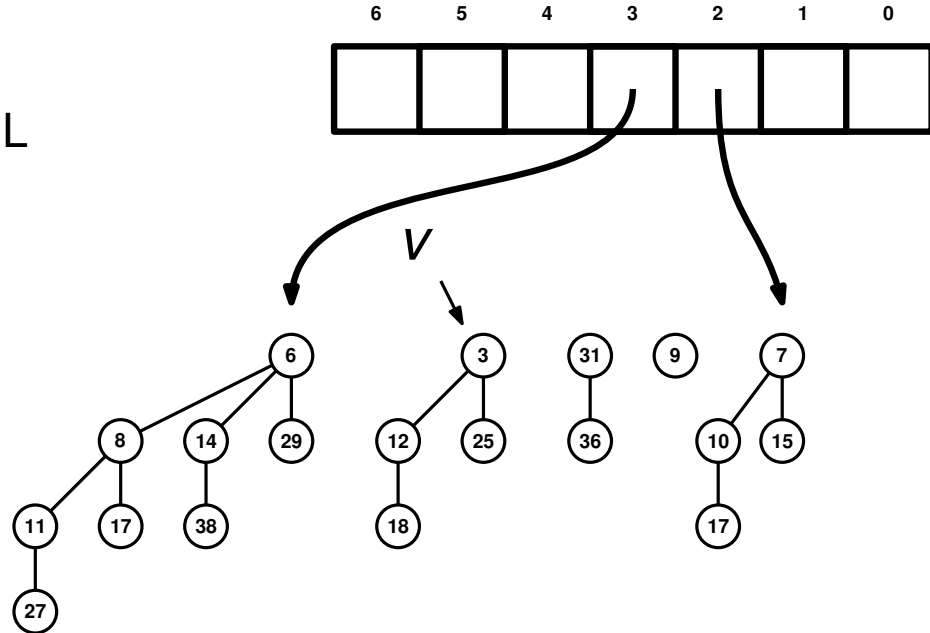
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize log n -sized array A to NIL

for each v in root list **do**

$d = v.degree$

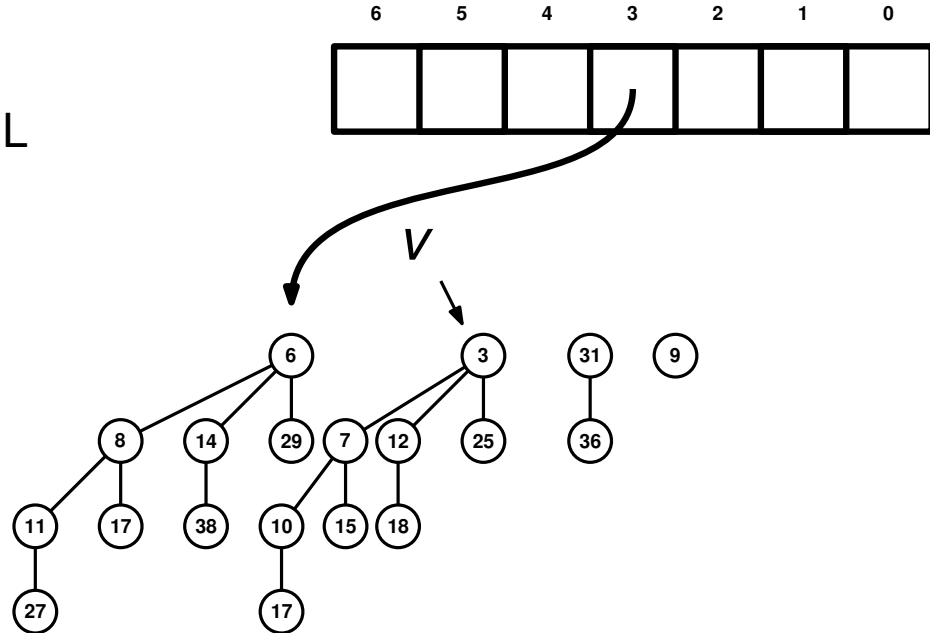
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

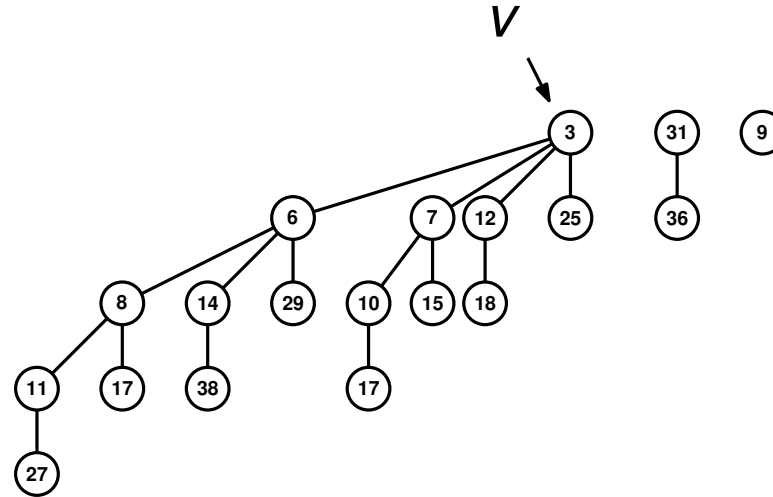
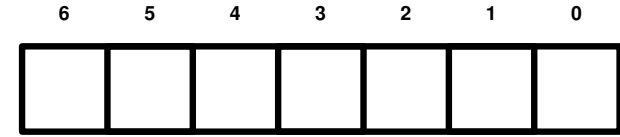
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

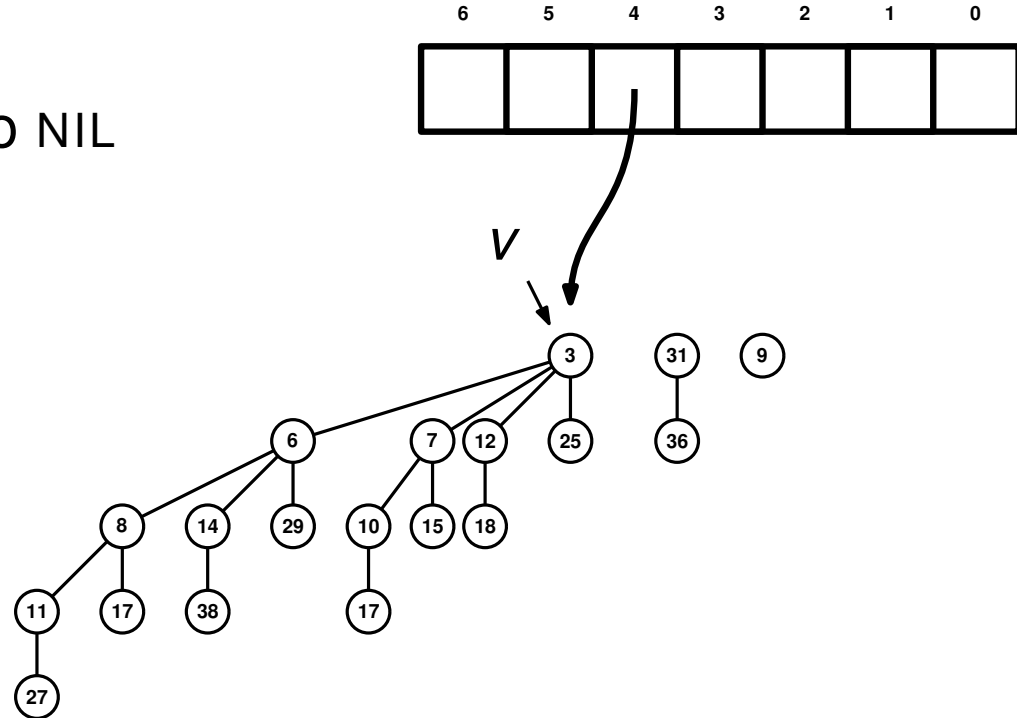
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

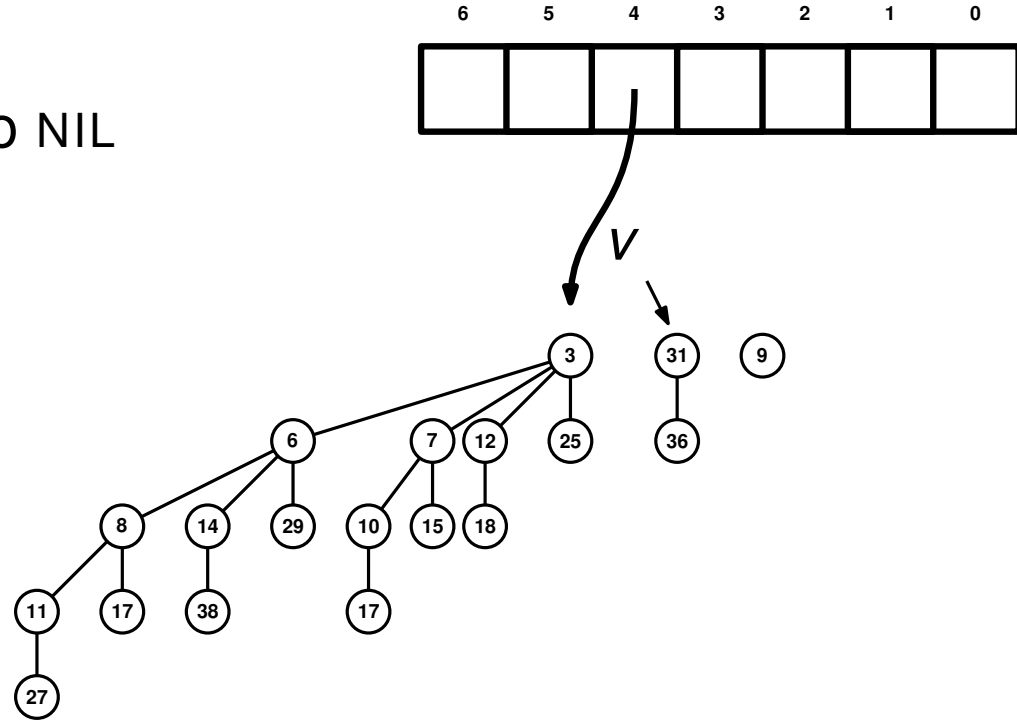
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

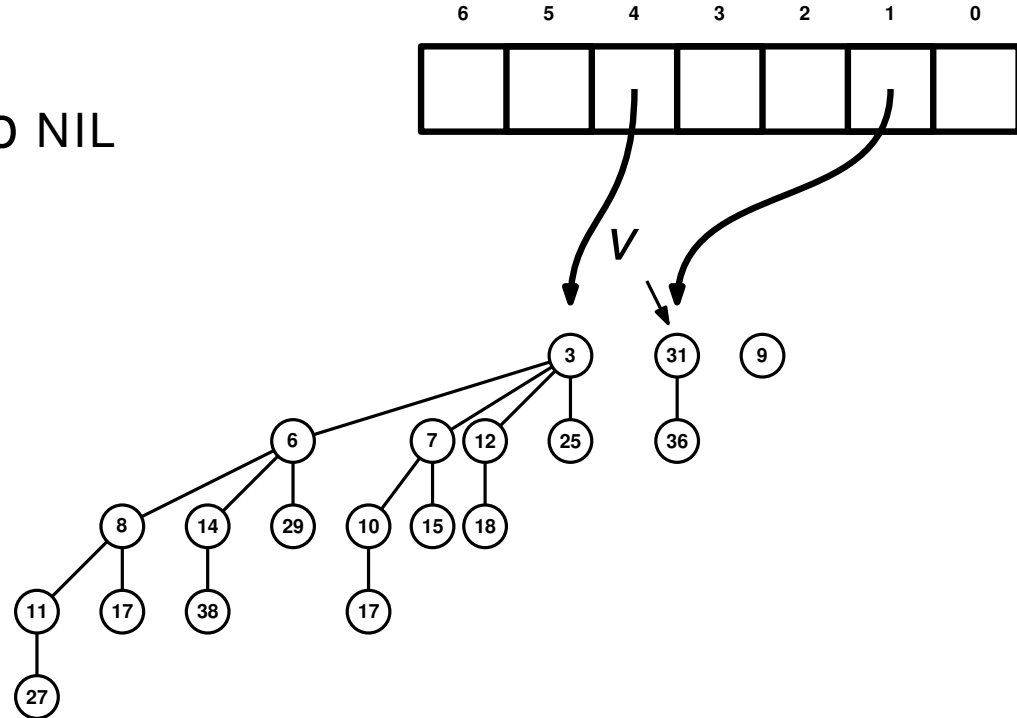
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

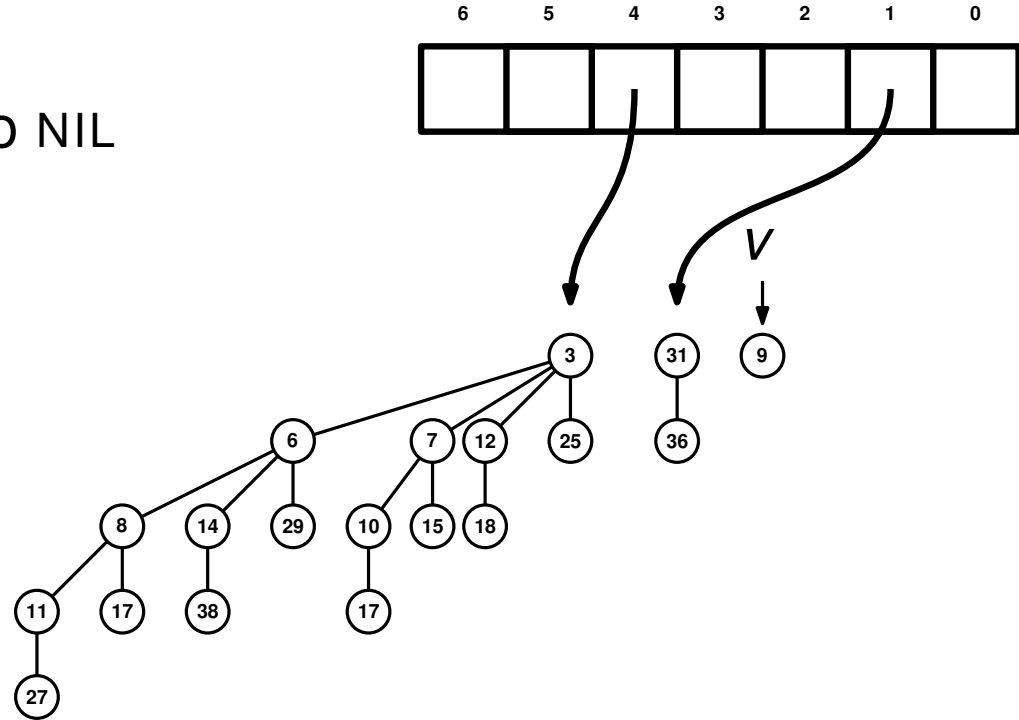
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

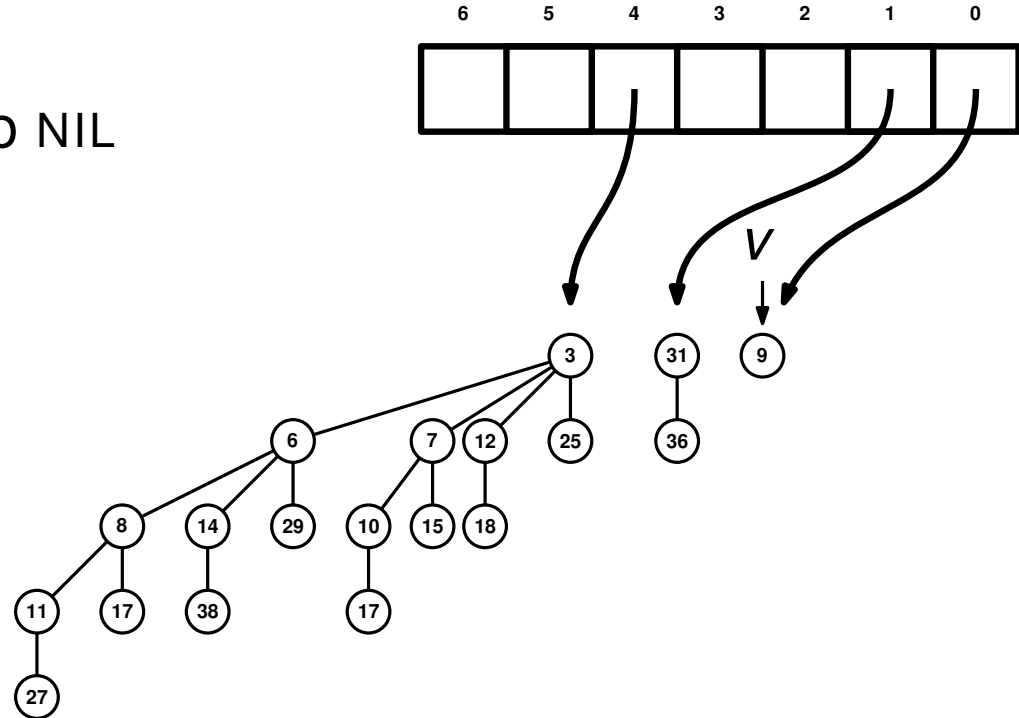
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$

$min = +\infty$

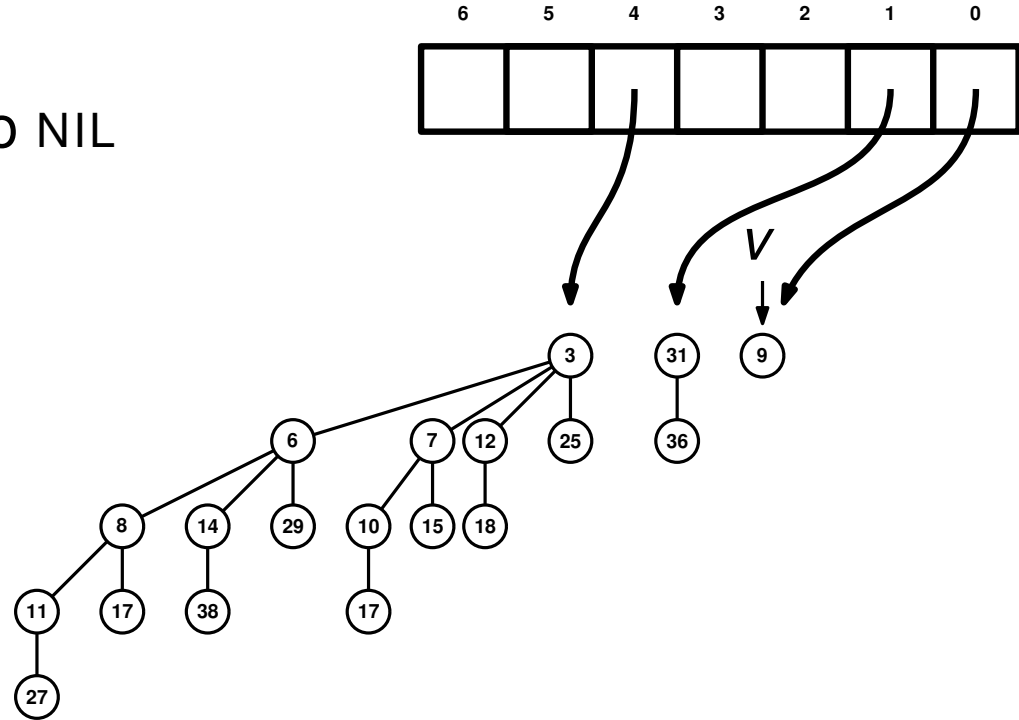
for $i = 0$ **to** $\log n - 1$ **do**

if $A[i] \neq \text{NIL}$ **then**

Add $A[i]$ to the root list

if $A[i].key < min$ **then**

$Q.min \leftarrow A[i]; min = A[i].key$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize log n -sized array A to NIL

for each v in root list **do**

$d = v.degree$

while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$

$min = +\infty$

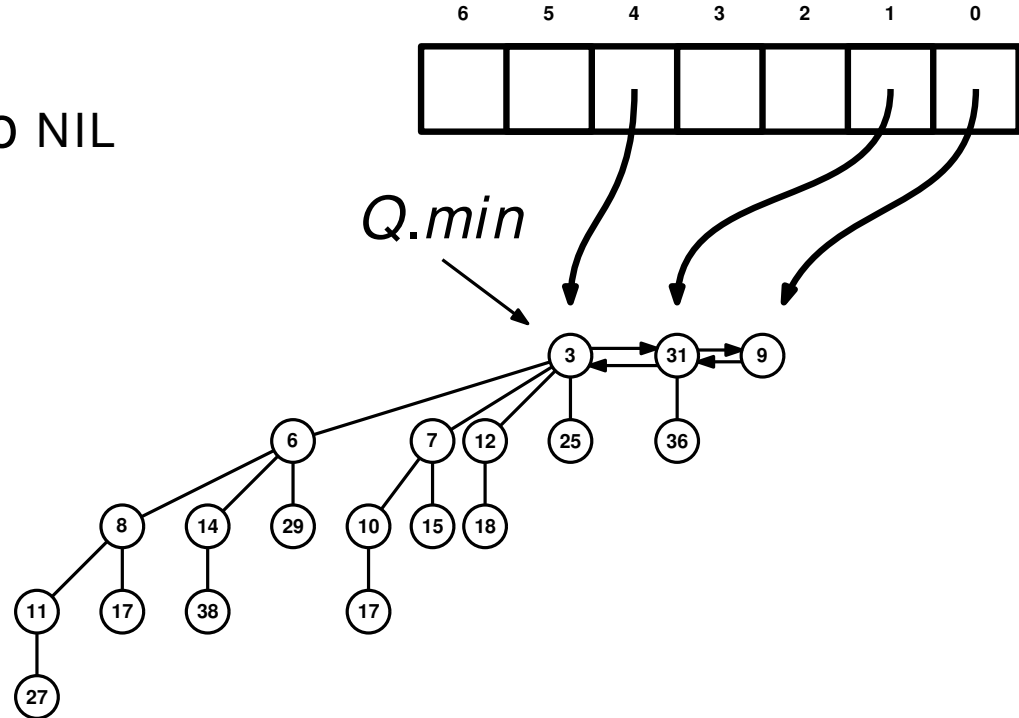
for $i = 0$ **to** $\log n - 1$ **do**

if $A[i] \neq \text{NIL}$ **then**

Add $A[i]$ to the root list

if $A[i].key < min$ **then**

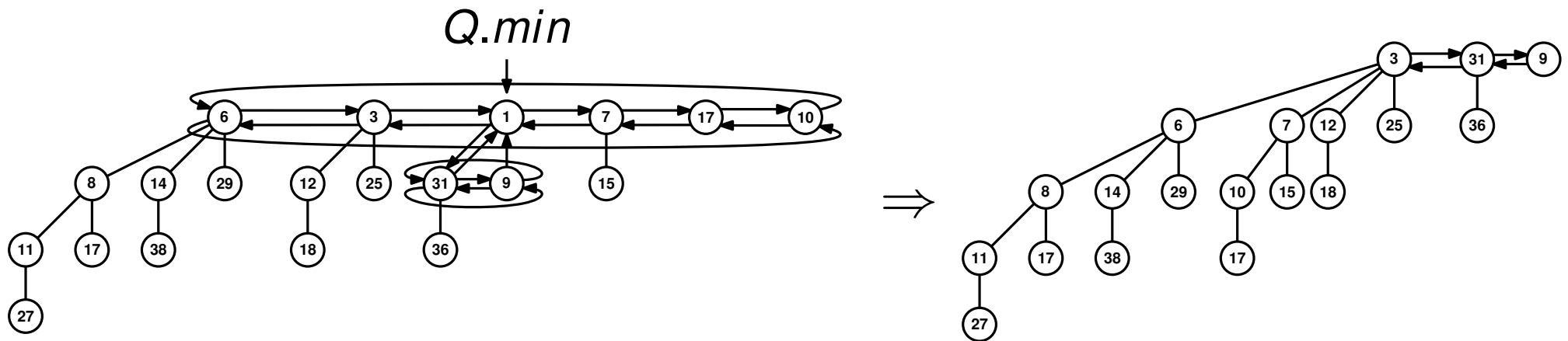
$Q.min \leftarrow A[i]; min = A[i].key$



Lazy EXTRACT-MIN(Q) Analysis

We will use the Potential Method with $\Phi_i = t_i =$ number of trees in the root list after the i -th operation.

Let d be the number of children of the $Q.min$ ($d \leq \log n$)



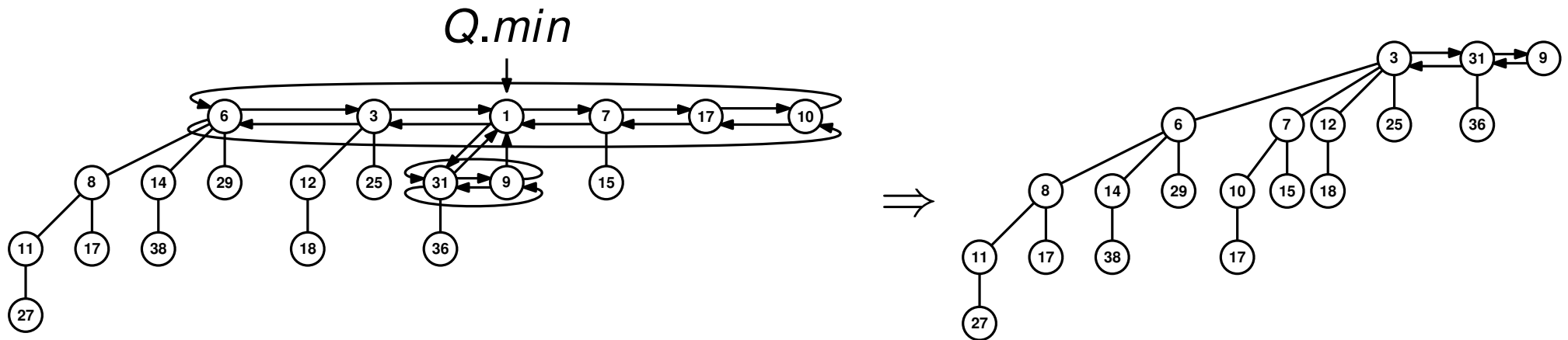
```

function CONSOLIDATE(Q)
  Initialize log n-sized array A to NIL
  for each v in root list do
    d = v.degree
    while A[d] ≠ NIL do
      v = LINK(v, A[d])
      A[d] = NIL
      d = d + 1
    A[d] = v; v.parent = NIL
  min = +∞
  for i = 0 to log n - 1 do
    if A[i] ≠ NIL then
      Add A[i] to the root list
      if A[i].key < min then
        Q.min ← A[i]; min = A[i].key
  
```

Lazy EXTRACT-MIN(Q) Analysis

We will use the Potential Method with $\Phi_i = t_i =$ number of trees in the root list after the i -th operation.

Let d be the number of children of the $Q.min$ ($d \leq \log n$)



- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log n \leq O(1) + t_{i-1} + 2 \log n$
- Change in potential: $\Delta\Phi_i = t_i - t_{i-1}$

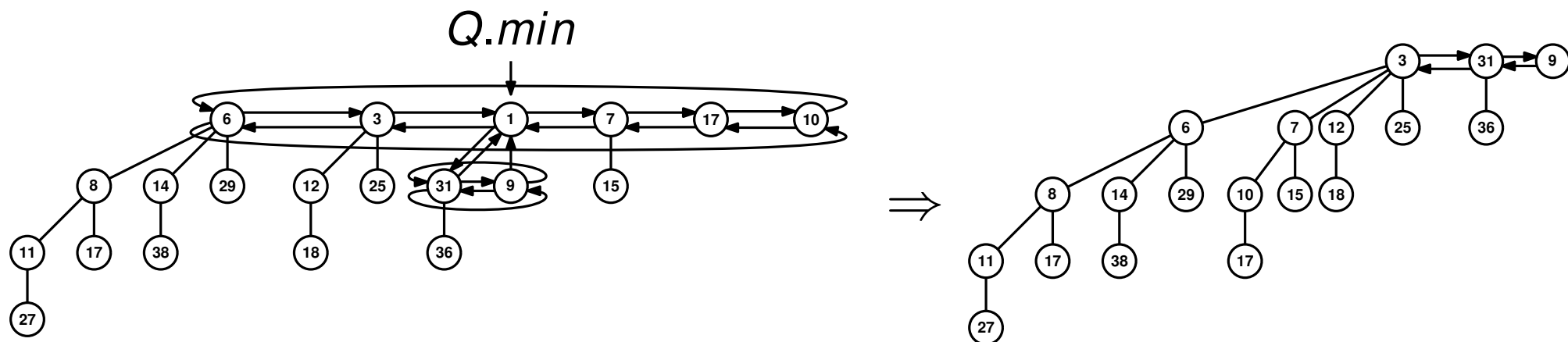
```

function CONSOLIDATE(Q)
  Initialize log n-sized array A to NIL
  for each v in root list do
    d = v.degree
    while A[d] ≠ NIL do
      v = LINK(v, A[d])
      A[d] = NIL
      d = d + 1
    A[d] = v; v.parent = NIL
  min = +∞
  for i = 0 to log n - 1 do
    if A[i] ≠ NIL then
      Add A[i] to the root list
      if A[i].key < min then
        Q.min ← A[i]; min = A[i].key
  
```

Lazy EXTRACT-MIN(Q) Analysis

We will use the Potential Method with $\Phi_i = t_i =$ number of trees in the root list after the i -th operation.

Let d be the number of children of the $Q.min$ ($d \leq \log n$)



- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log n \leq O(1) + t_{i-1} + 2 \log n$
- Change in potential: $\Delta\Phi_i = t_i - t_{i-1}$

$$\hat{c}_i = c_i + \Delta\Phi_i \leq O(1) + 2 \log n + t_i$$

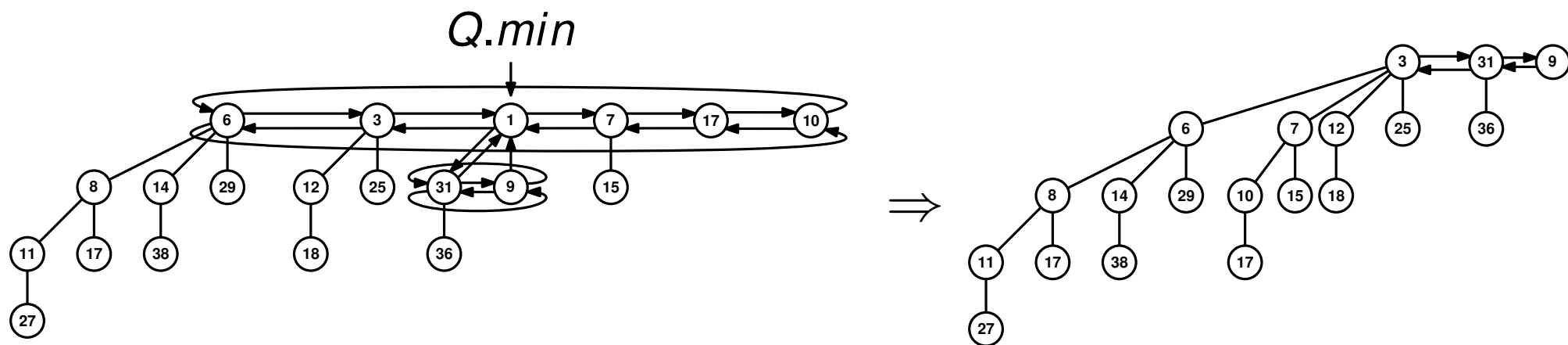
```

function CONSOLIDATE(Q)
  Initialize log n-sized array A to NIL
  for each v in root list do
    d = v.degree
    while A[d] ≠ NIL do
      v = LINK(v, A[d])
      A[d] = NIL
      d = d + 1
    A[d] = v; v.parent = NIL
  min = +∞
  for i = 0 to log n - 1 do
    if A[i] ≠ NIL then
      Add A[i] to the root list
      if A[i].key < min then
        Q.min ← A[i]; min = A[i].key
  
```


Lazy EXTRACT-MIN(Q) Analysis

We will use the Potential Method with $\Phi_i = t_i =$ number of trees in the root list after the i -th operation.

Let d be the number of children of the $Q.min$ ($d \leq \log n$)



- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log n \leq O(1) + t_{i-1} + 2 \log n$
- Change in potential: $\Delta\Phi_i = t_i - t_{i-1}$

$$\hat{c}_i = c_i + \Delta\Phi_i \leq O(1) + 2 \log n + t_i$$

there are $t_i \leq \log n$ trees in the root list after consolidation

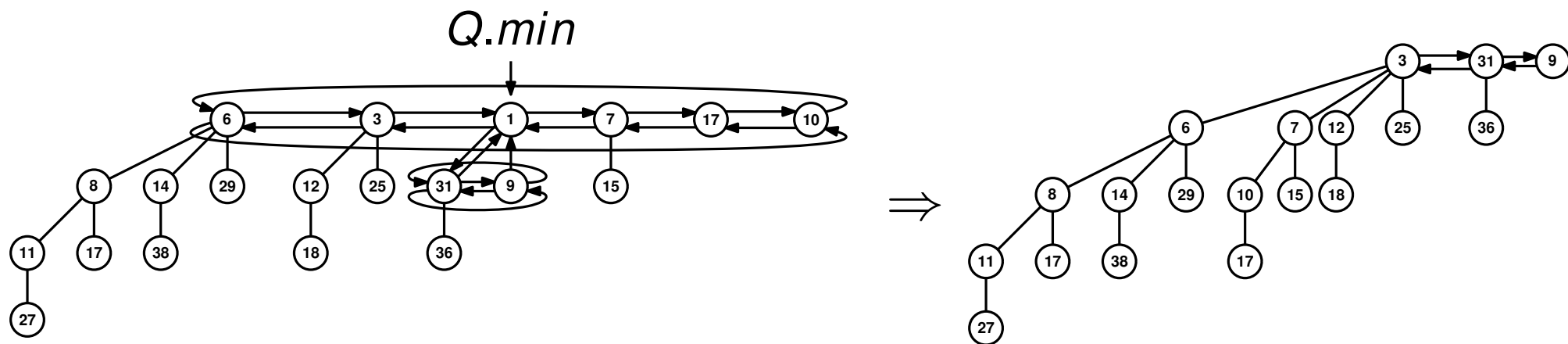
```

function CONSOLIDATE(Q)
  Initialize log n-sized array A to NIL
  for each v in root list do
    d = v.degree
    while A[d] ≠ NIL do
      v = LINK(v, A[d])
      A[d] = NIL
      d = d + 1
    A[d] = v; v.parent = NIL
  min = +∞
  for i = 0 to log n - 1 do
    if A[i] ≠ NIL then
      Add A[i] to the root list
      if A[i].key < min then
        Q.min ← A[i]; min = A[i].key
  
```

Lazy EXTRACT-MIN(Q) Analysis

We will use the Potential Method with $\Phi_i = t_i =$ number of trees in the root list after the i -th operation.

Let d be the number of children of the $Q.min$ ($d \leq \log n$)



- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log n \leq O(1) + t_{i-1} + 2 \log n$
- Change in potential: $\Delta\Phi_i = t_i - t_{i-1}$

$$\hat{c}_i = c_i + \Delta\Phi_i \leq O(1) + 2 \log n + t_i \leq O(1) + 3 \log n = O(\log n)$$

there are $t_i \leq \log n$ trees in the root list after consolidation

```

function CONSOLIDATE(Q)
  Initialize log n-sized array A to NIL
  for each v in root list do
    d = v.degree
    while A[d] ≠ NIL do
      v = LINK(v, A[d])
      A[d] = NIL
      d = d + 1
    A[d] = v; v.parent = NIL
  min = +∞
  for i = 0 to log n - 1 do
    if A[i] ≠ NIL then
      Add A[i] to the root list
      if A[i].key < min then
        Q.min ← A[i]; min = A[i].key
    
```

Heaps

	Binomial	Lazy Binomial	Fibonacci
■ MAKE()	$O(1)$	$O(1)$	$O(1)$
■ INSERT(Q, x)	$O(1)^*$	$O(1)$	$O(1)$
■ MINIMUM(Q)	$O(1)$	$O(1)$	$O(1)$
■ EXTRACT-MIN(Q)	$O(\log n)$	$O(\log n)^*$	$O(\log n)^*$
■ DECREASE-KEY(Q, x, k)	$O(\log n)$	$O(\log n)$	$O(1)^*$
■ DELETE(Q, x)	$O(\log n)$	$O(\log n)^*$	$O(\log n)^*$
■ UNION(Q_1, Q_2)	$O(\log n)$	$O(1)$	$O(1)$

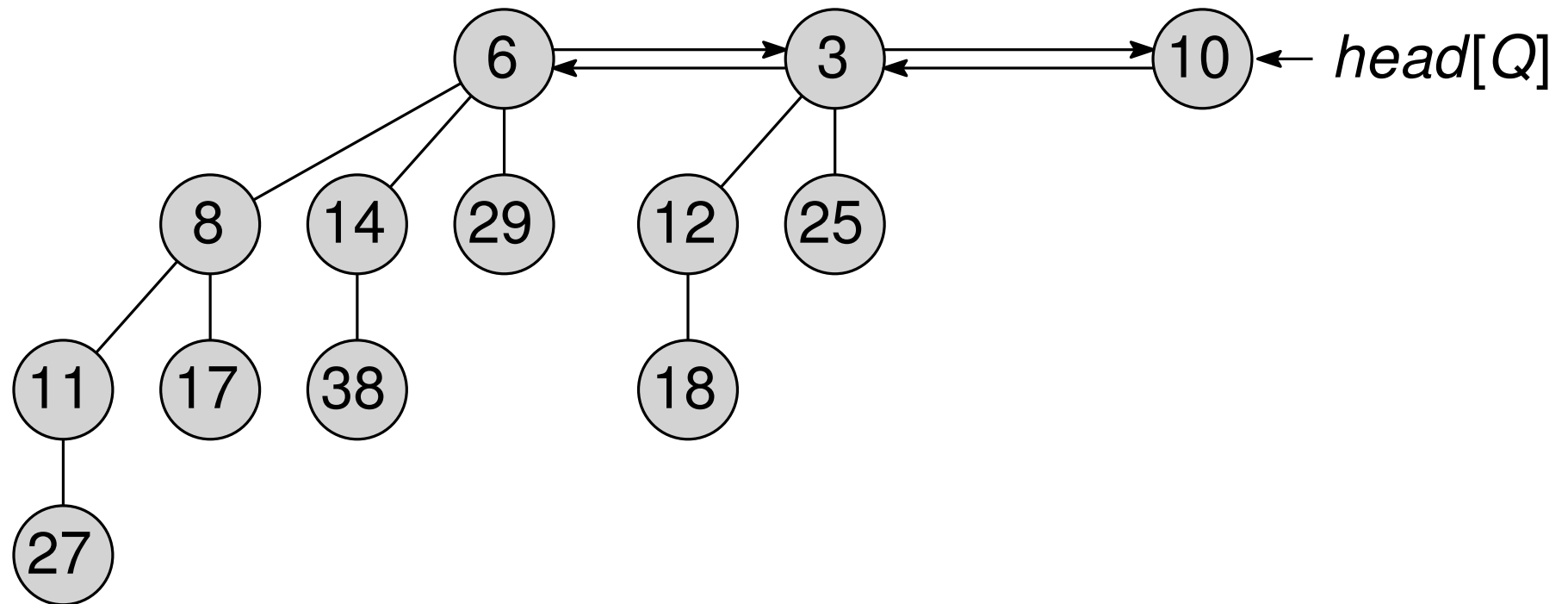
* Amortized cost

Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time

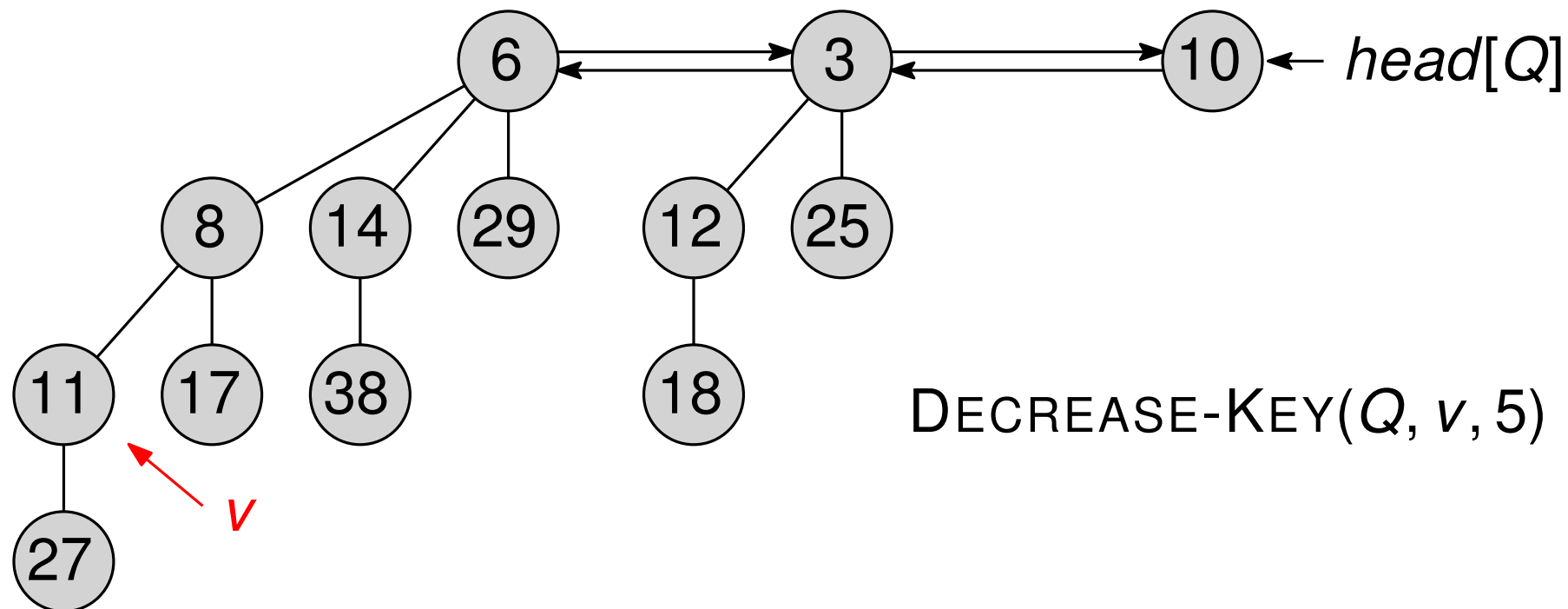
Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time



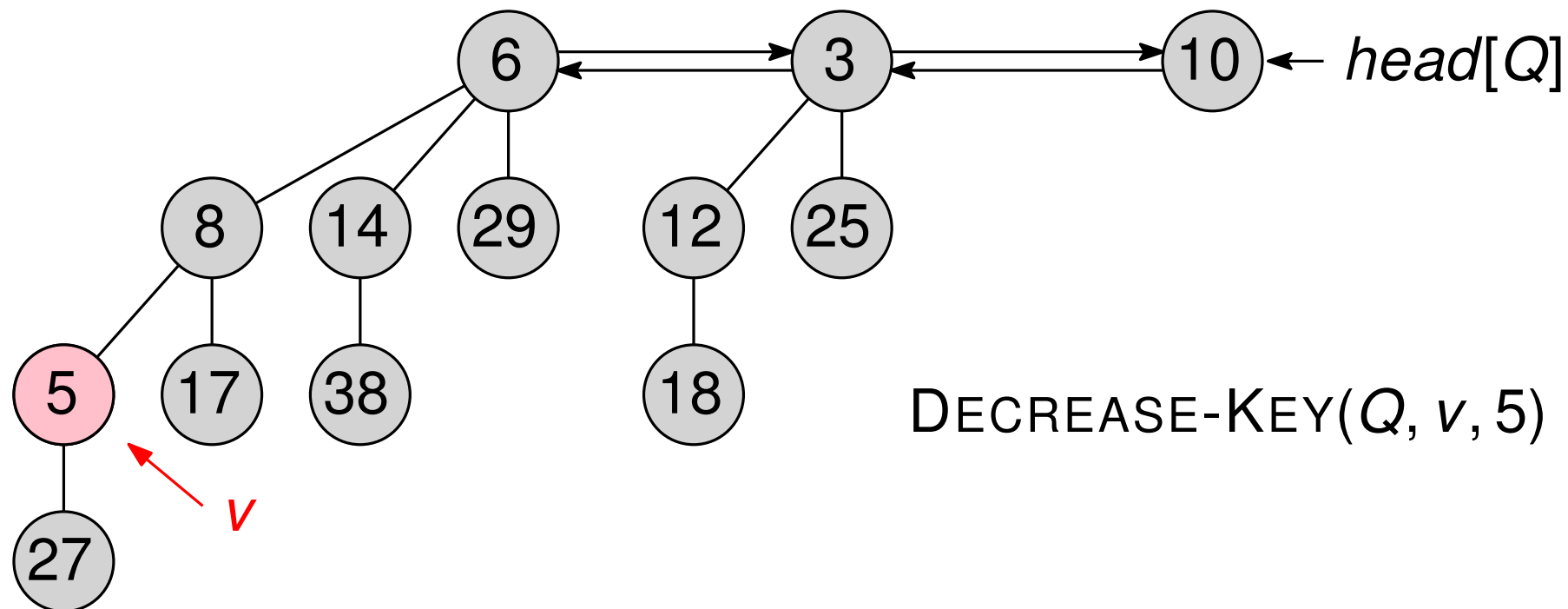
Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time



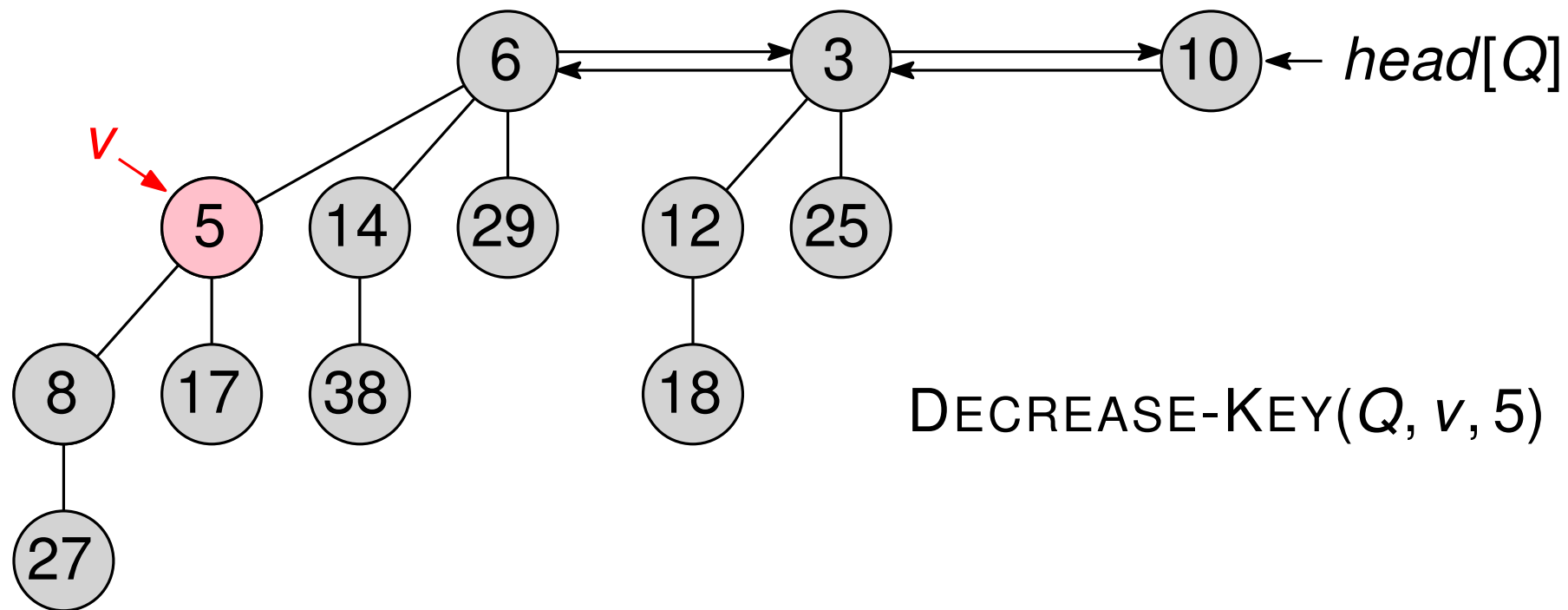
Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time



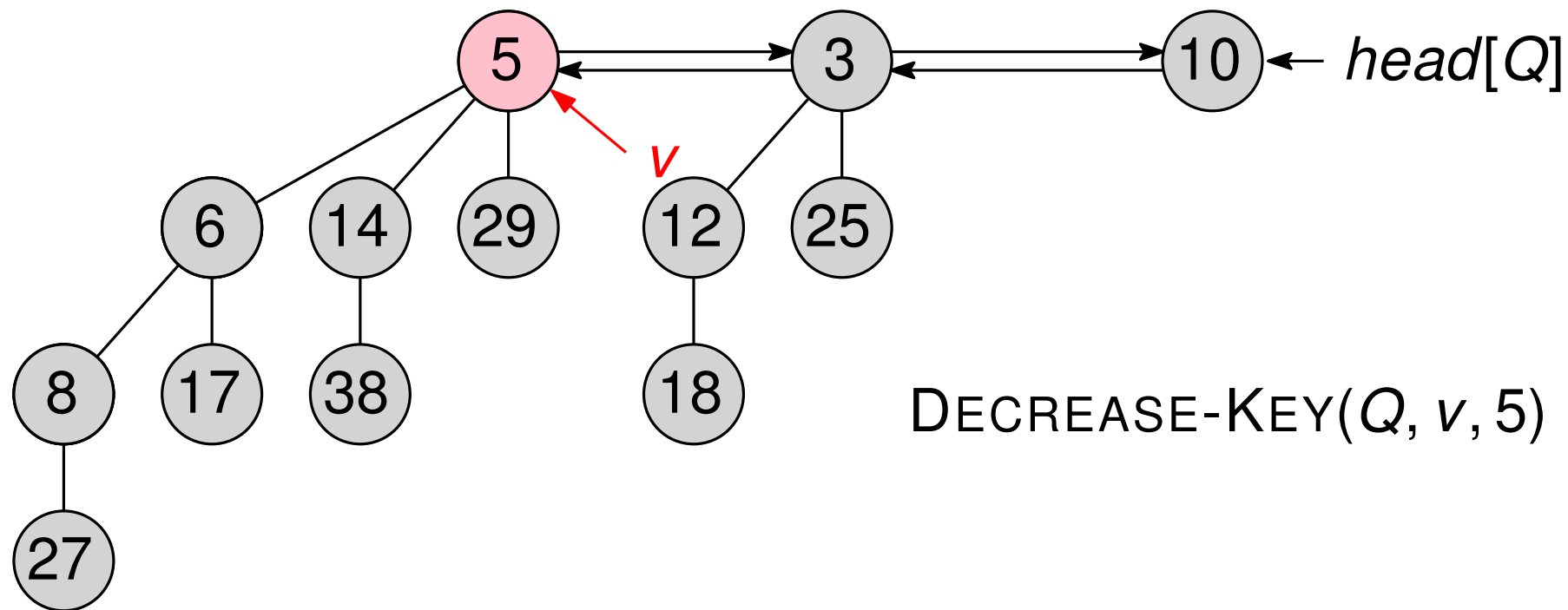
Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time



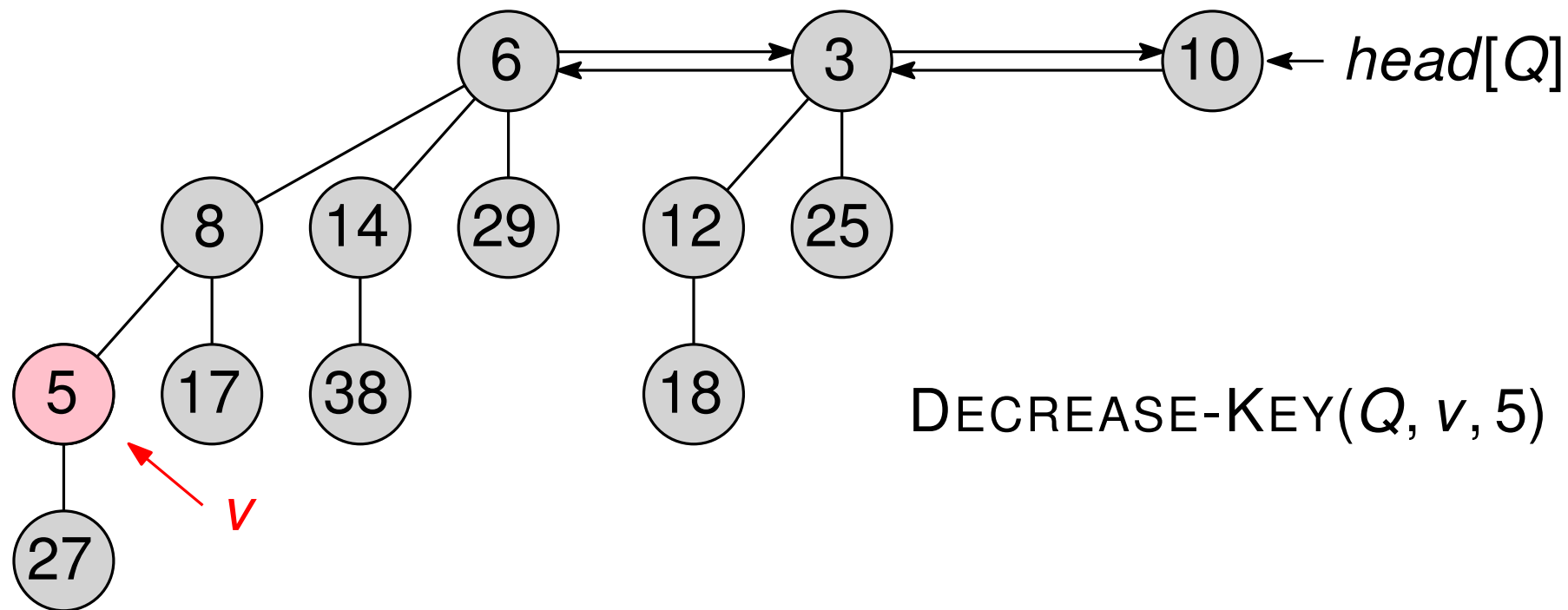
Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time



Fibonacci Heaps

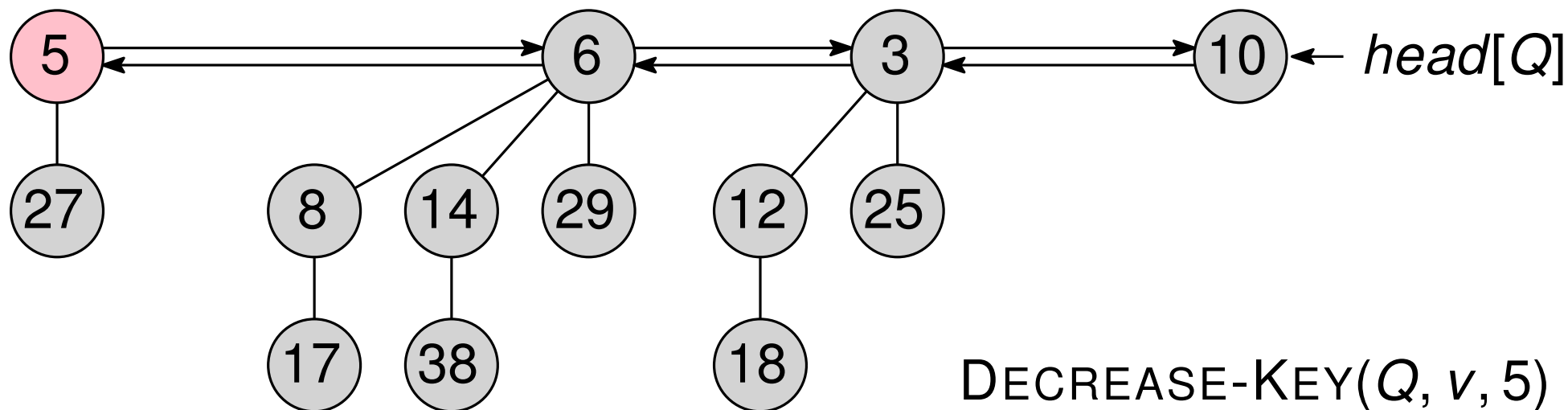
Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time



Idea: If the new key is smaller than parent's, DECREASE-KEY(Q, v, k) will splice-out v and add it to the root list

Fibonacci Heaps

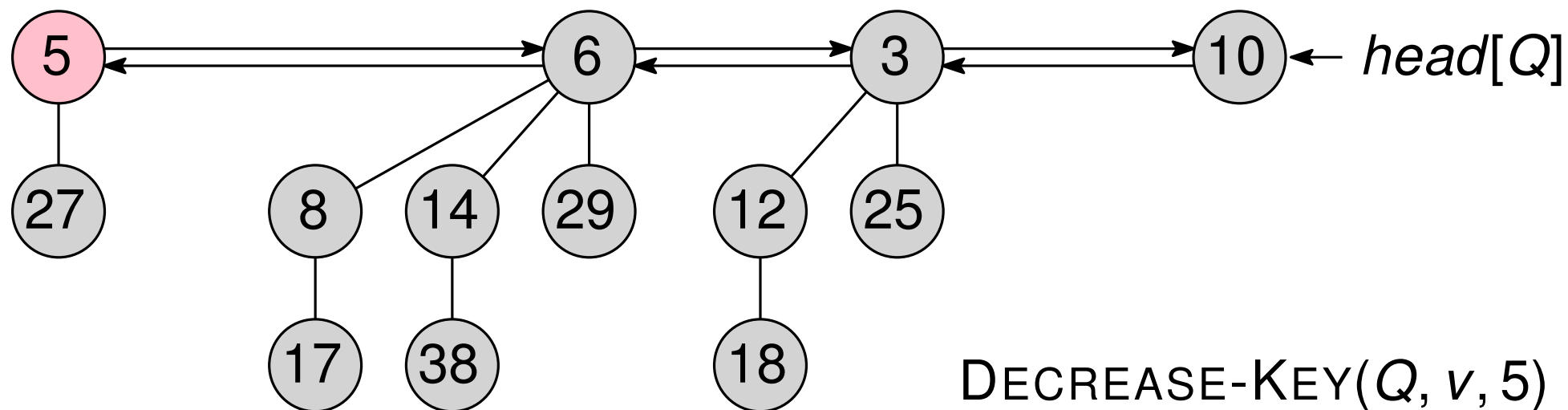
Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time



Idea: If the new key is smaller than parent's, DECREASE-KEY(Q, v, k) will splice-out v and add it to the root list

Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time



Idea: If the new key is smaller than parent's, DECREASE-KEY(Q, v, k) will splice-out v and add it to the root list

Almost!

Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time

Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time

Collection of heap-ordered (binomial) trees:

- Each tree is heap-ordered
- Arbitrary number of trees in the root list
- Sibling lists are doubly-linked circular lists
- Each node v stores
 - A pointer to parent ($v.parent$)
 - A pointer to one (arbitrary) child ($v.child$)

Fibonacci Heaps

Goal: DECREASE-KEY(Q, v, k) in $O(1)$ (amortized) time

Collection of heap-ordered (binomial) trees:

- Each tree is heap-ordered
- Arbitrary number of trees in the root list
- Sibling lists are doubly-linked circular lists
- Each node v stores
 - A pointer to parent ($v.parent$)
 - A pointer to one (arbitrary) child ($v.child$)

Each node v :

- stores number of children ($v.degree$)
- can be “marked” (boolean $v.marked$)
 - If v lost a child since becoming a child of another node
 - If so, will be spliced out and added to root list, next time it loses another child (and will become unmarked)

FIB-DECREASE-KEY(Q, x, k)

function FIB-DECREASE-KEY(Q, v, k)

$v.key = k$

if $v.parent \neq \text{NIL}$ **and** $v.key < v.parent.key$ **then**

 RECURSIVE-CUT($Q, v, v.parent$)

▷ Assert $k < v.key$

FIB-DECREASE-KEY(Q, x, k)

function FIB-DECREASE-KEY(Q, v, k)

▷ Assert $k < v.key$

$v.key = k$

if $v.parent \neq \text{NIL}$ **and** $v.key < v.parent.key$ **then**

 RECURSIVE-CUT($Q, v, v.parent$)

function RECURSIVE-CUT(Q, v, p)

▷ p is the parent of v

 Remove v from child list of p

 Add v to the root list of Q

$v.mark = \text{FALSE}; v.parent = \text{NIL}$

▷ Unmark v

if $p.parent \neq \text{NIL}$ **then**

if $p.mark == \text{FALSE}$ **then**

$p.mark = \text{TRUE}$

▷ p just lost a child, so mark it

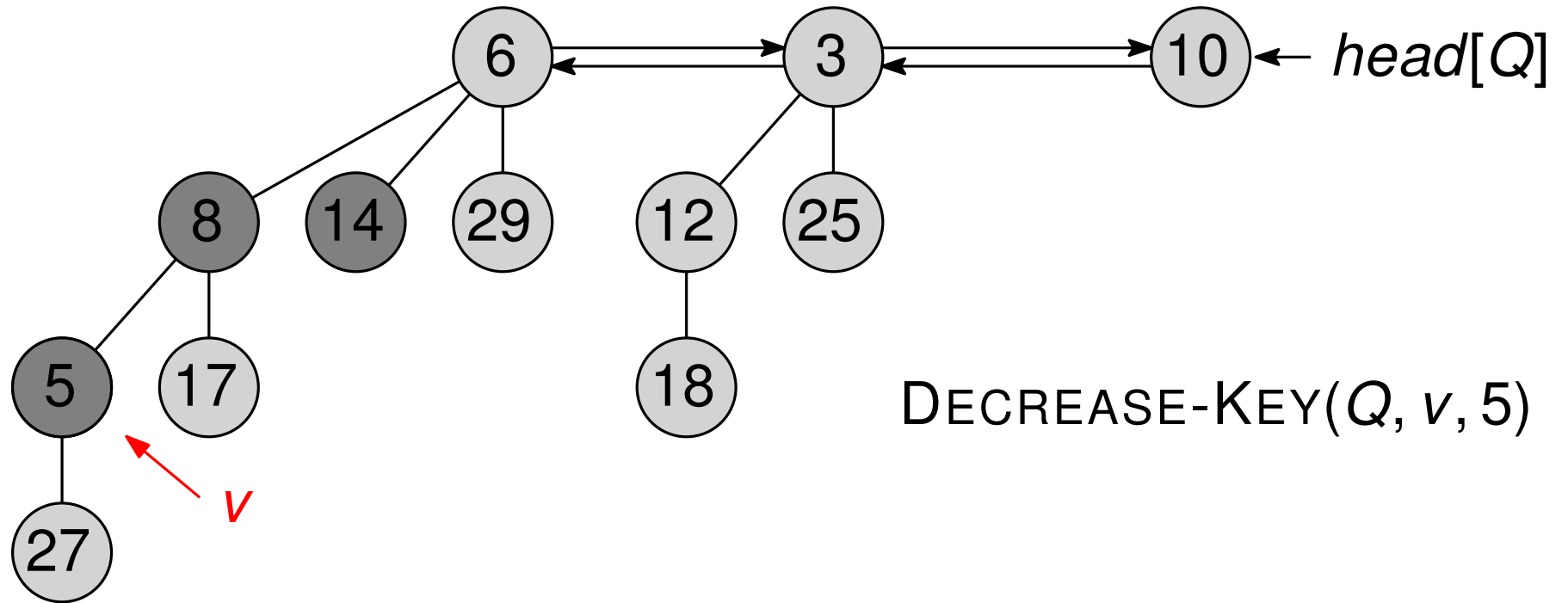
else

▷ p just lost the second child

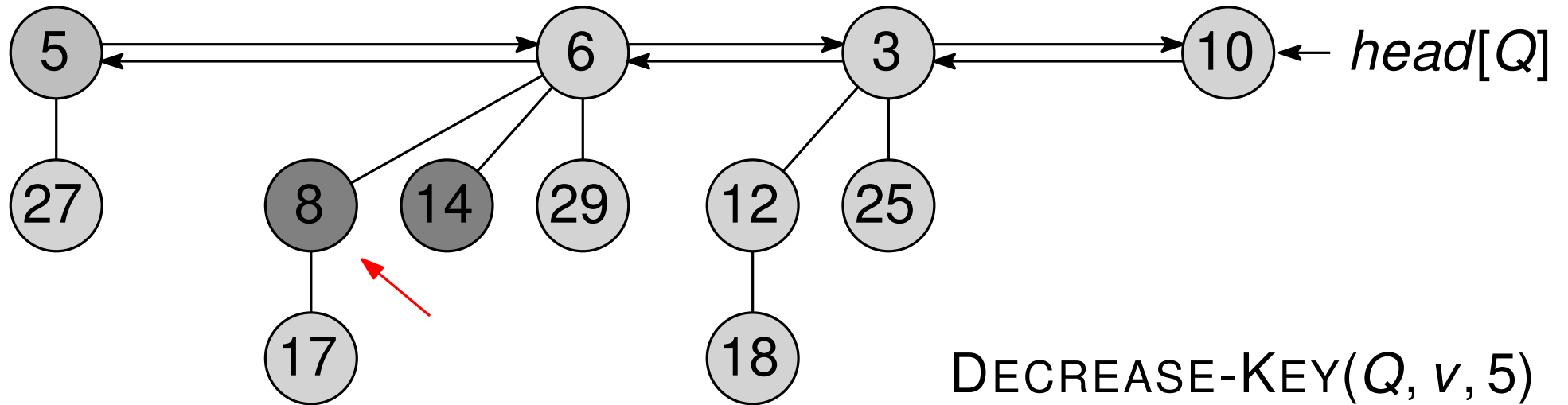
 RECURSIVE-CUT($Q, p, p.parent$)

▷ so add it to the root list too

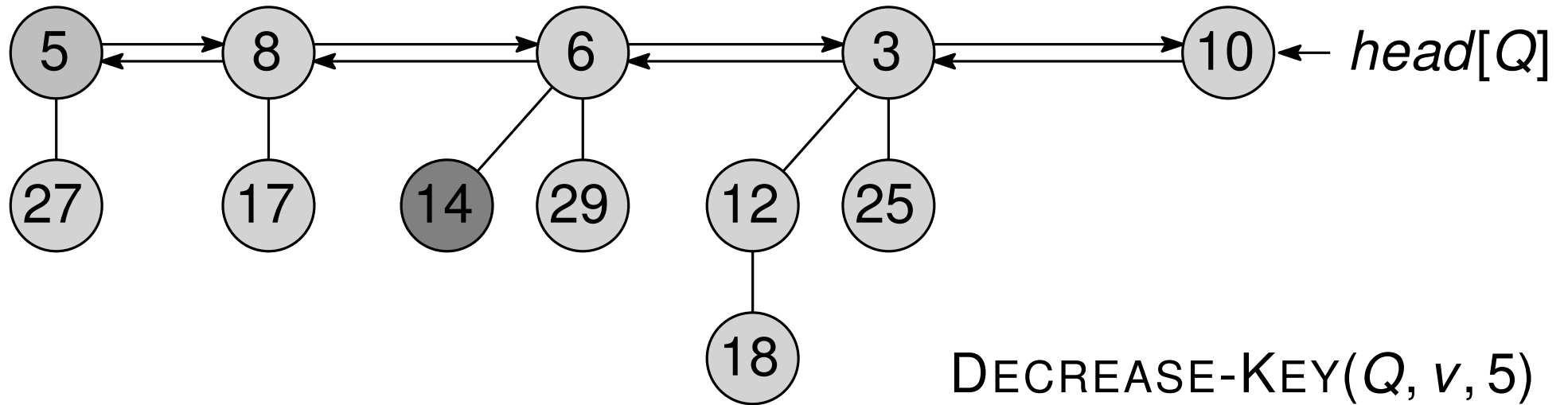
Example: FIB-DECREASE-KEY(Q, v, k)



Example: FIB-DECREASE-KEY(Q, v, k)



Example: FIB-DECREASE-KEY(Q, v, k)



Unmarking Vertices

Each node v :

- stores number of children ($v.degree$)
- can be “marked” (boolean $v.marked$)
 - If v lost a child **since becoming a child of another node**

CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize log n -sized array A to NIL

for each v in root list **do**

$d = v.degree$

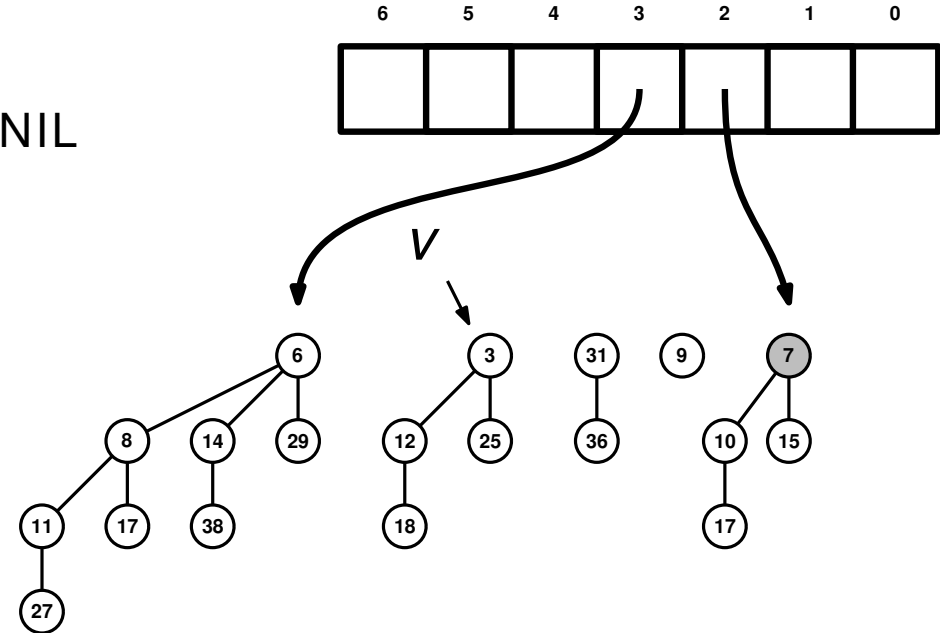
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

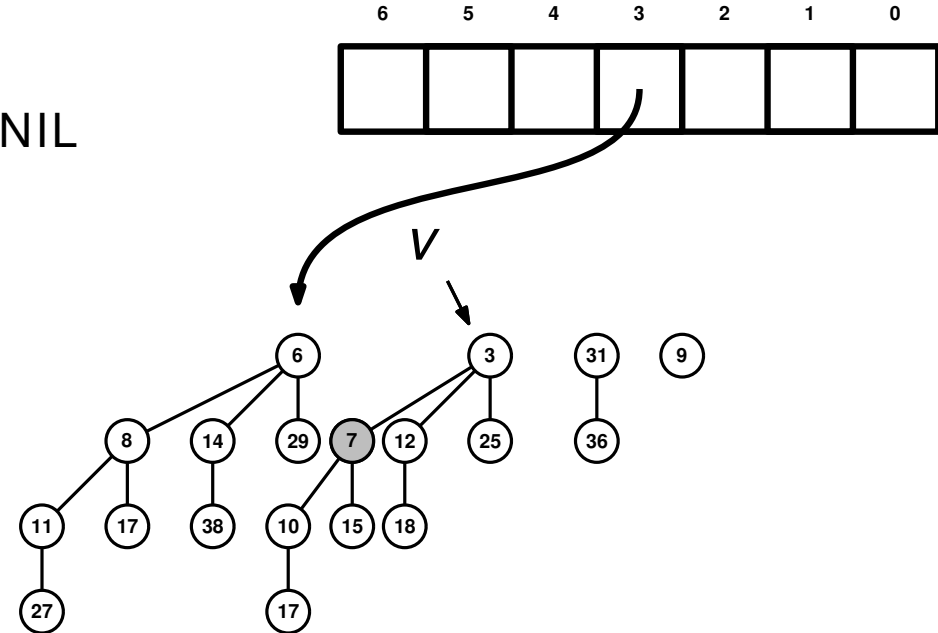
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $\log n$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$

function LINK(v, w)

if $w.key < v.key$ **then**

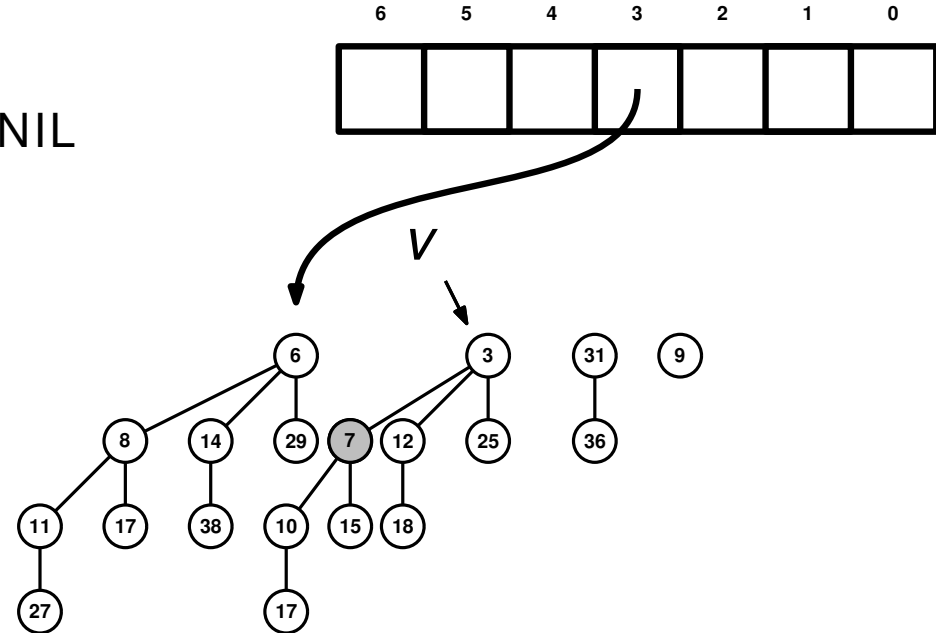
SWAP(v, w)

▷ make sure v is smaller

Add w to the child list of v

$v.degree = v.degree + 1$

$w.marked = \text{FALSE}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize log n -sized array A to NIL

for each v in root list **do**

$d = v.degree$

while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$

function LINK(v, w)

if $w.key < v.key$ **then**

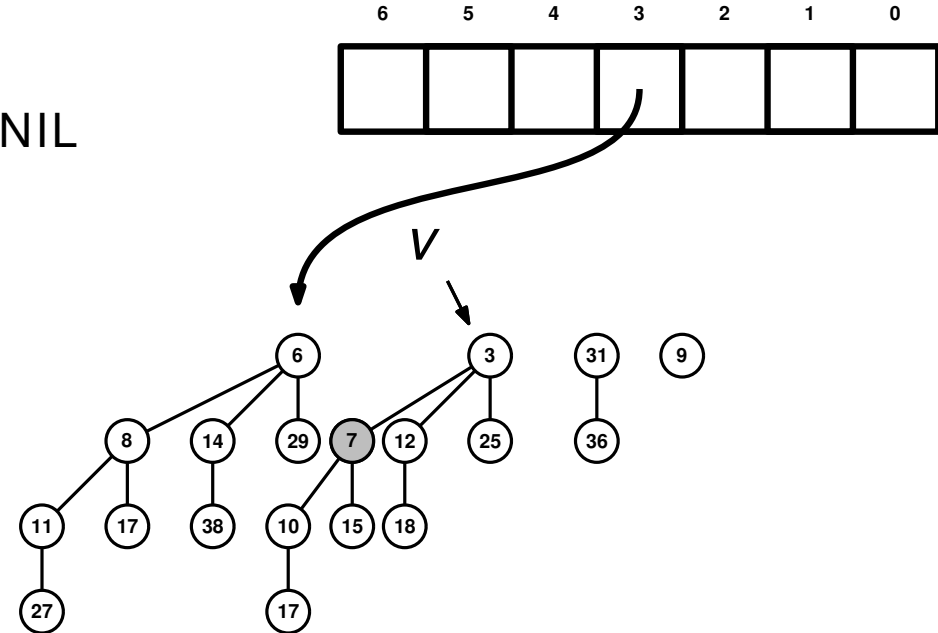
SWAP(v, w)

▷ make sure v is smaller

Add w to the child list of v

$v.degree = v.degree + 1$

→ $w.marked = \text{FALSE}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize log n -sized array A to NIL

for each v in root list **do**

$d = v.degree$

while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$

function LINK(v, w)

if $w.key < v.key$ **then**

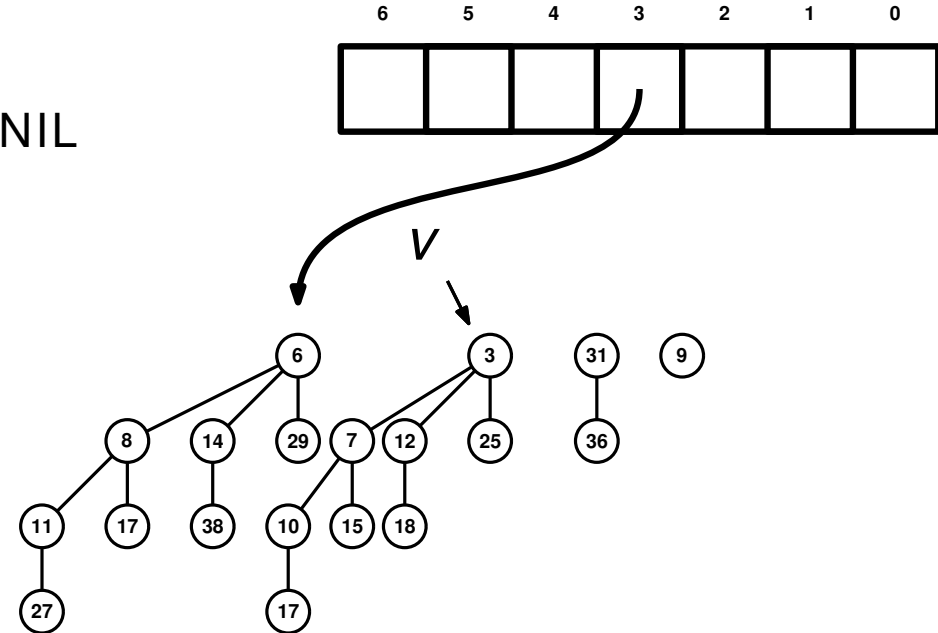
SWAP(v, w)

▷ make sure v is smaller

Add w to the child list of v

$v.degree = v.degree + 1$

→ $w.marked = \text{FALSE}$



FIB-DECREASE-KEY(Q, v, k) Analysis

Potential: $\Phi_i = k(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- $k = \text{constant TBD later}$

function RECURSIVE-CUT(Q, v, p)

Remove v from child list of p

Add v to the root list of Q

$v.mark = \text{FALSE}; v.parent = \text{NIL}$

if $p.parent \neq \text{NIL}$ **then**

if $p.mark == \text{FALSE}$ **then**

$p.mark = \text{TRUE}$

else

 RECURSIVE-CUT($Q, p, p.parent$)

FIB-DECREASE-KEY(Q, v, k) Analysis

Potential: $\Phi_i = k(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- $k = \text{constant TBD later}$

Let t' be the number of trees added to the root list.

```
function RECURSIVE-CUT( $Q, v, p$ )
  Remove  $v$  from child list of  $p$ 
  Add  $v$  to the root list of  $Q$ 
   $v.mark = \text{FALSE}; v.parent = \text{NIL}$ 
  if  $p.parent \neq \text{NIL}$  then
    if  $p.mark == \text{FALSE}$  then
       $p.mark = \text{TRUE}$ 
    else
      RECURSIVE-CUT( $Q, p, p.parent$ )
```

FIB-DECREASE-KEY(Q, v, k) Analysis

Potential: $\Phi_i = k(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- $k = \text{constant TBD later}$

Let t' be the number of trees added to the root list.

- $t_i = t_{i-1} + t'$
- $m_i \leq m_{i-1} - (t' - 1) + 1$

```
function RECURSIVE-CUT( $Q, v, p$ )
  Remove  $v$  from child list of  $p$ 
  Add  $v$  to the root list of  $Q$ 
   $v.mark = \text{FALSE}; v.parent = \text{NIL}$ 
  if  $p.parent \neq \text{NIL}$  then
    if  $p.mark == \text{FALSE}$  then
       $p.mark = \text{TRUE}$ 
    else
      RECURSIVE-CUT( $Q, p, p.parent$ )
```

FIB-DECREASE-KEY(Q, v, k) Analysis

Potential: $\Phi_i = k(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- $k = \text{constant TBD later}$

Let t' be the number of trees added to the root list.

- $t_i = t_{i-1} + t'$
- $m_i \leq m_{i-1} - (t' - 1) + 1$

$$\Delta\Phi = k(t_i - t_{i-1}) + 2k(m_i - m_{i-1}) \leq k(t' + 2 \cdot (2 - t')) = k(4 - t')$$

```
function RECURSIVE-CUT( $Q, v, p$ )
  Remove  $v$  from child list of  $p$ 
  Add  $v$  to the root list of  $Q$ 
   $v.mark = \text{FALSE}; v.parent = \text{NIL}$ 
  if  $p.parent \neq \text{NIL}$  then
    if  $p.mark == \text{FALSE}$  then
       $p.mark = \text{TRUE}$ 
    else
      RECURSIVE-CUT( $Q, p, p.parent$ )
```

FIB-DECREASE-KEY(Q, v, k) Analysis

Potential: $\Phi_i = k(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- $k = \text{constant TBD later}$

Let t' be the number of trees added to the root list.

- $t_i = t_{i-1} + t'$
- $m_i \leq m_{i-1} - (t' - 1) + 1$

$$\Delta\Phi = k(t_i - t_{i-1}) + 2k(m_i - m_{i-1}) \leq k(t' + 2 \cdot (2 - t')) = k(4 - t')$$

Actual cost: $c_i = O(t')$, i.e., $c_i \leq \bar{k} \cdot t'$ for some constant \bar{k}

```
function RECURSIVE-CUT( $Q, v, p$ )
  Remove  $v$  from child list of  $p$ 
  Add  $v$  to the root list of  $Q$ 
   $v.mark = \text{FALSE}; v.parent = \text{NIL}$ 
  if  $p.parent \neq \text{NIL}$  then
    if  $p.mark == \text{FALSE}$  then
       $p.mark = \text{TRUE}$ 
    else
      RECURSIVE-CUT( $Q, p, p.parent$ )
```

FIB-DECREASE-KEY(Q, v, k) Analysis

Potential: $\Phi_i = k(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- $k = \text{constant TBD later}$

Let t' be the number of trees added to the root list.

- $t_i = t_{i-1} + t'$
- $m_i \leq m_{i-1} - (t' - 1) + 1$

```
function RECURSIVE-CUT( $Q, v, p$ )
  Remove  $v$  from child list of  $p$ 
  Add  $v$  to the root list of  $Q$ 
   $v.mark = \text{FALSE}; v.parent = \text{NIL}$ 
  if  $p.parent \neq \text{NIL}$  then
    if  $p.mark == \text{FALSE}$  then
       $p.mark = \text{TRUE}$ 
    else
      RECURSIVE-CUT( $Q, p, p.parent$ )
```

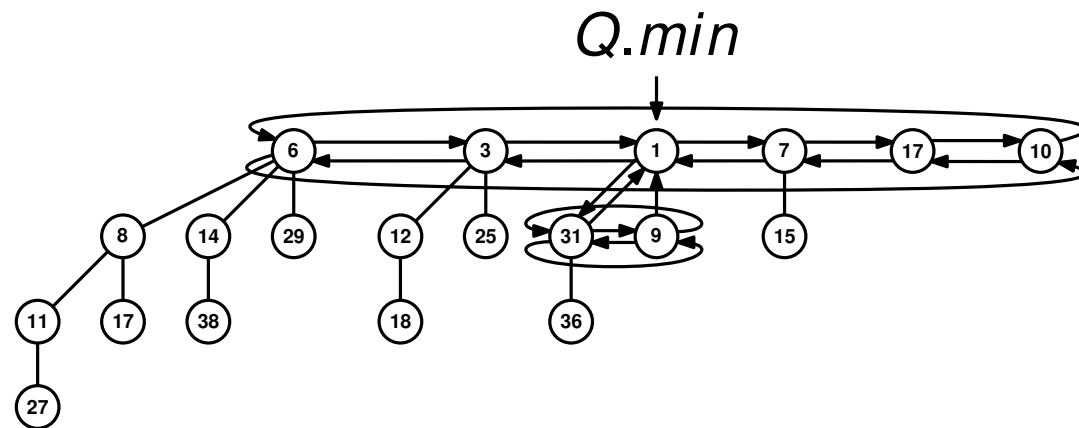
$$\Delta\Phi = k(t_i - t_{i-1}) + 2k(m_i - m_{i-1}) \leq k(t' + 2 \cdot (2 - t')) = k(4 - t')$$

Actual cost: $c_i = O(t')$, i.e., $c_i \leq \bar{k} \cdot t'$ for some constant \bar{k}

By setting $k = \bar{k}$, we get:

$$\hat{c}_i = c_i + \Delta\Phi \leq \bar{k} \cdot t' + \bar{k}(4 - t') = 4\bar{k} = O(1)$$

EXTRACT-MIN(Q)



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

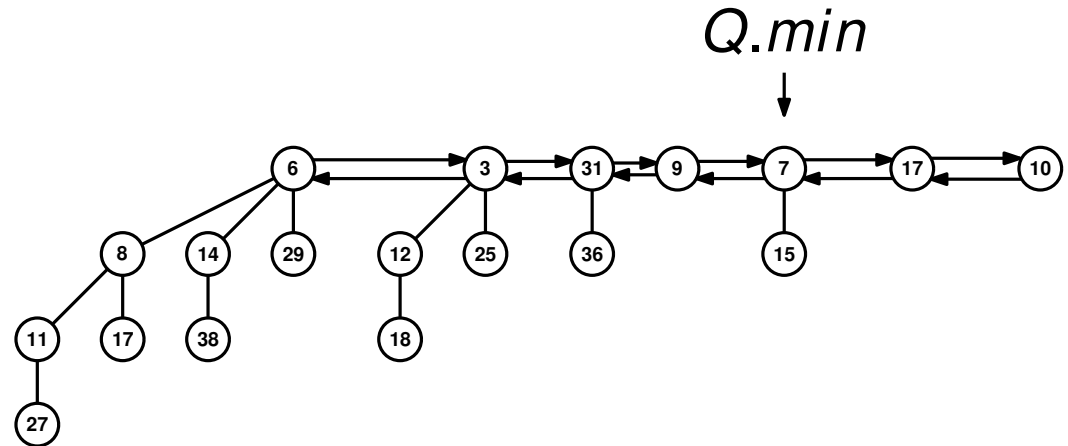
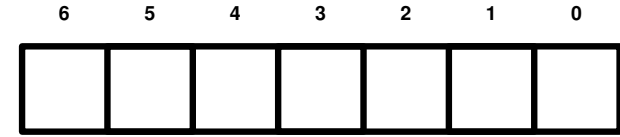
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

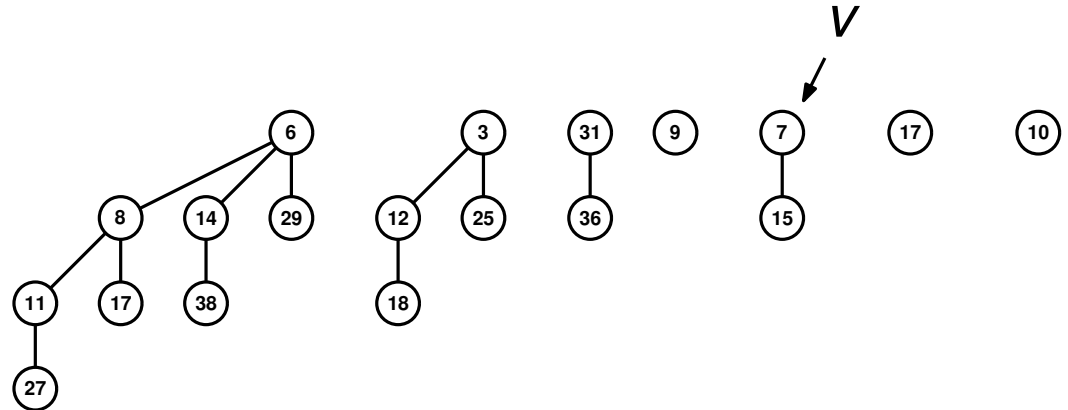
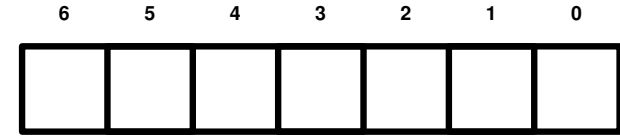
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

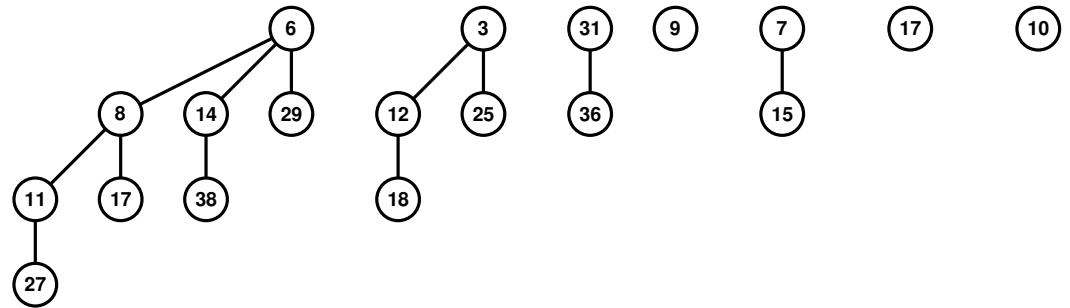
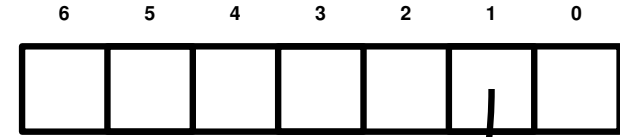
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

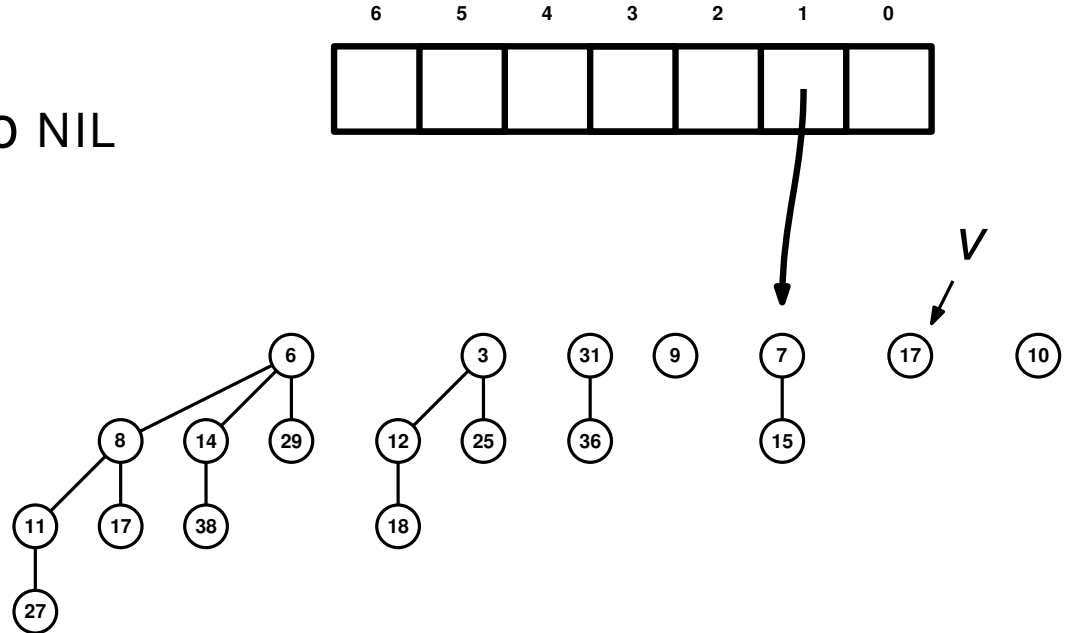
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

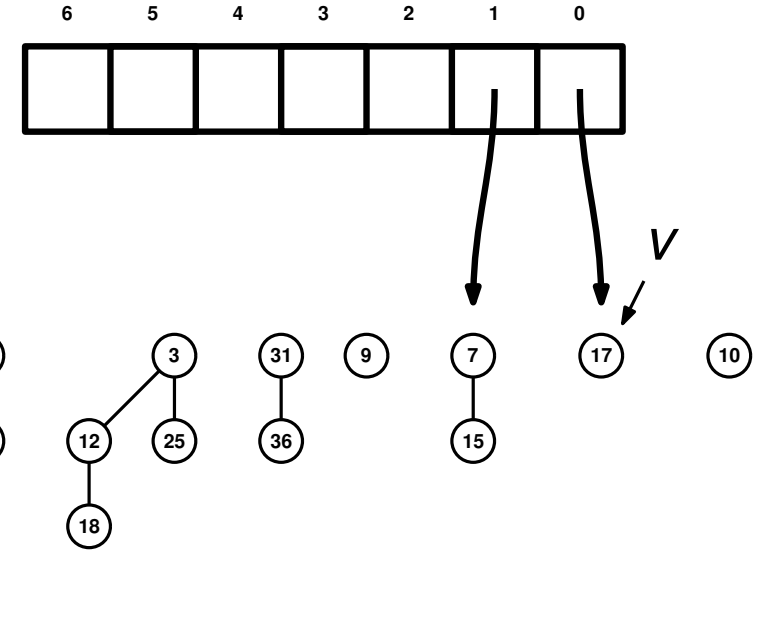
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

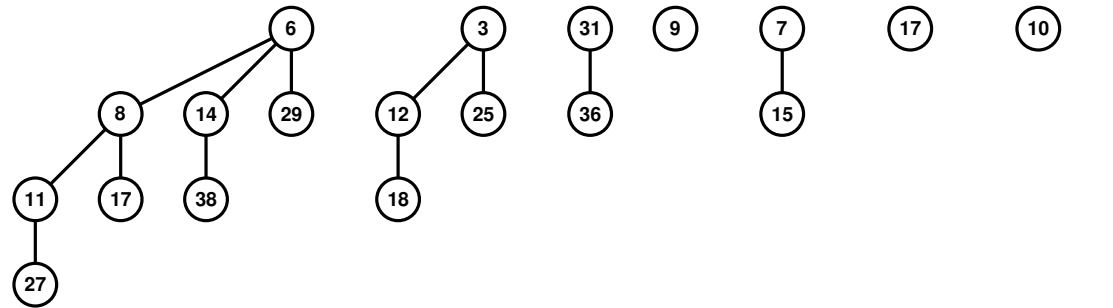
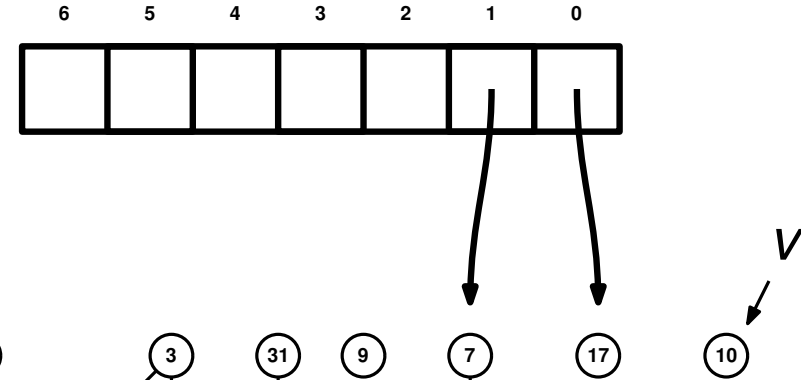
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

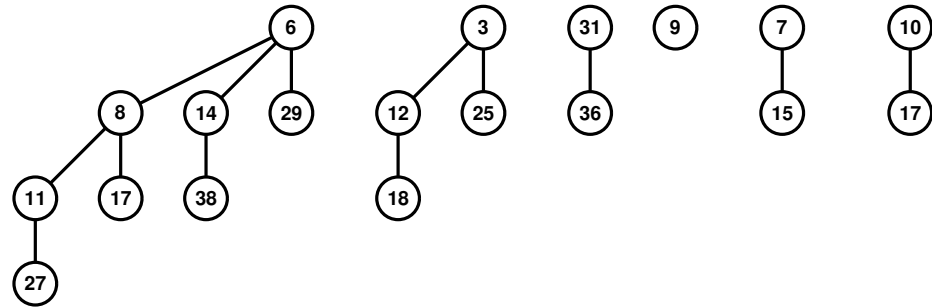
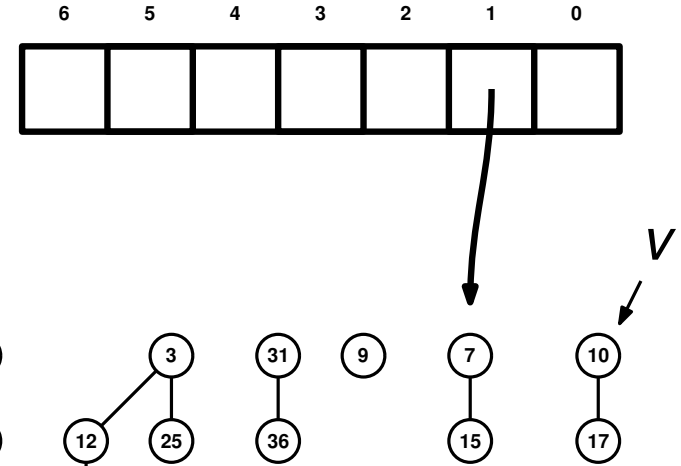
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

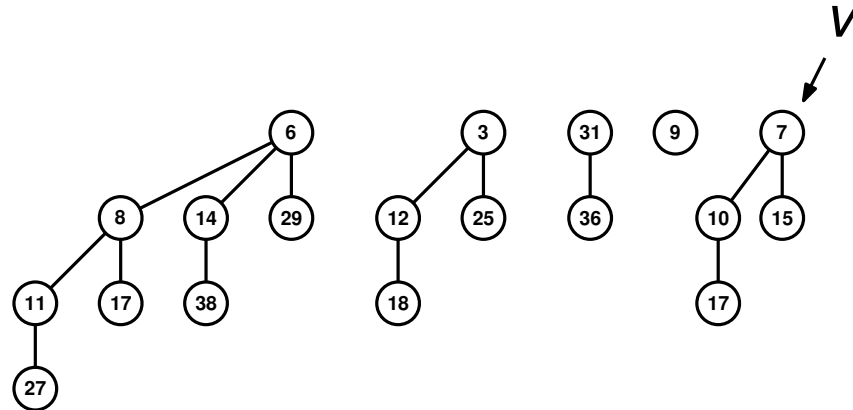
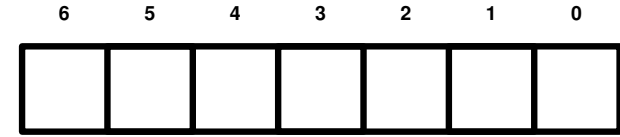
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

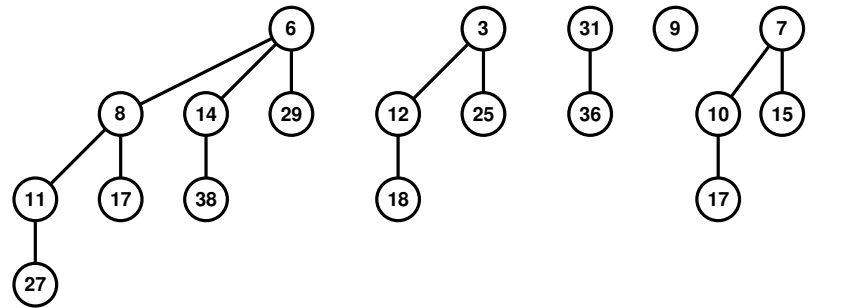
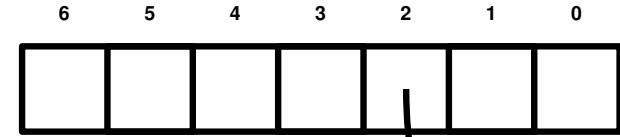
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

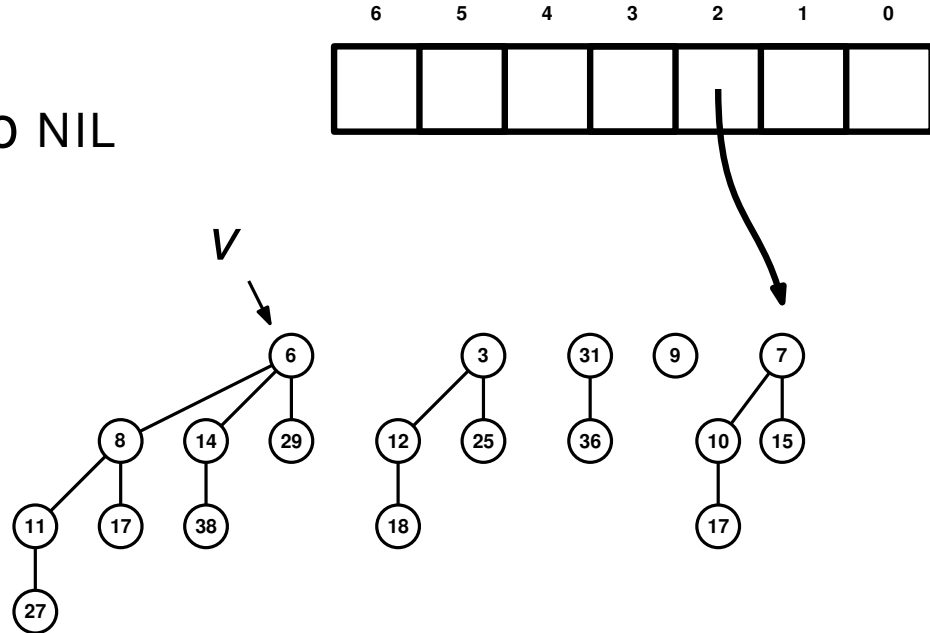
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

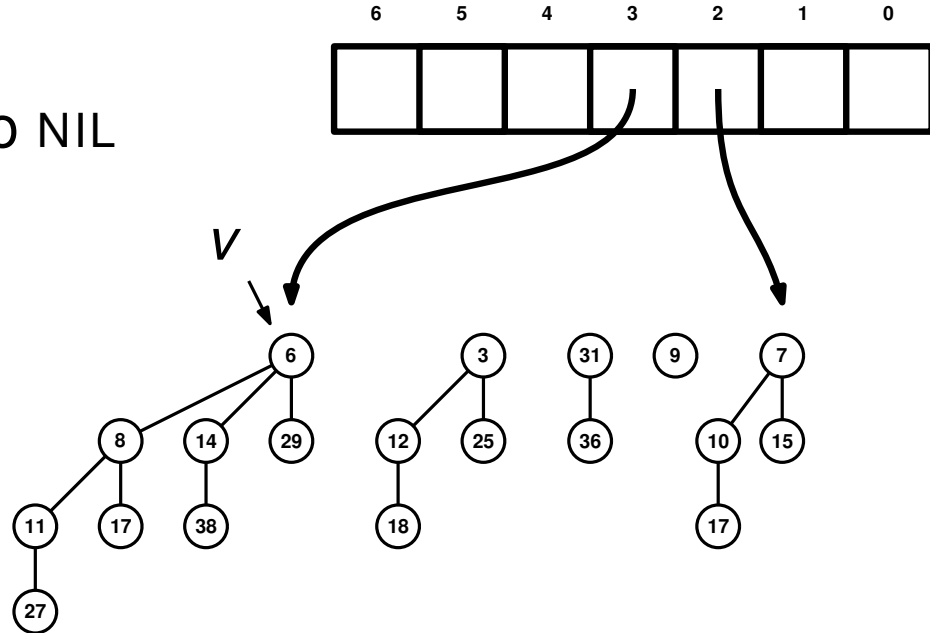
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

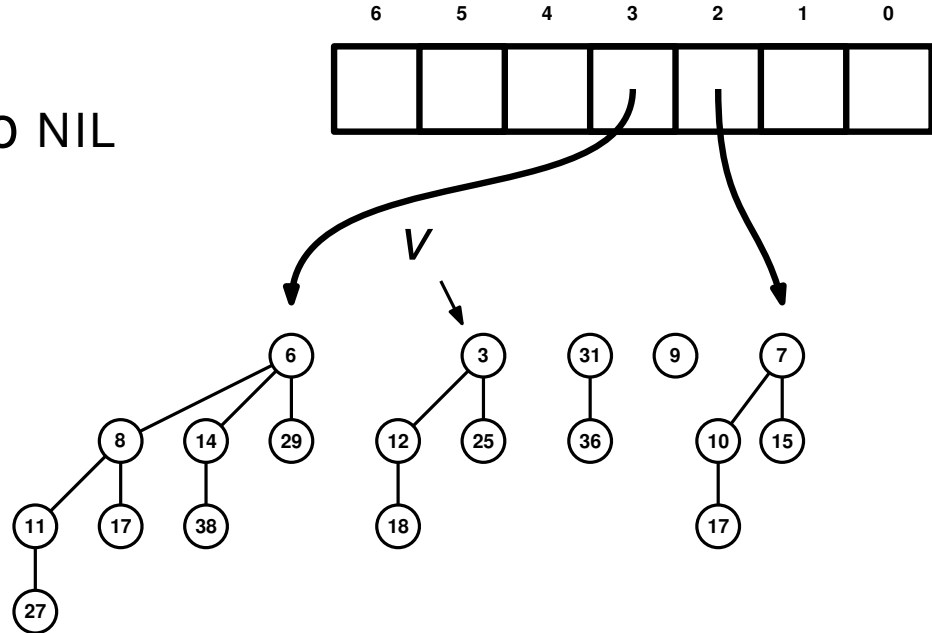
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

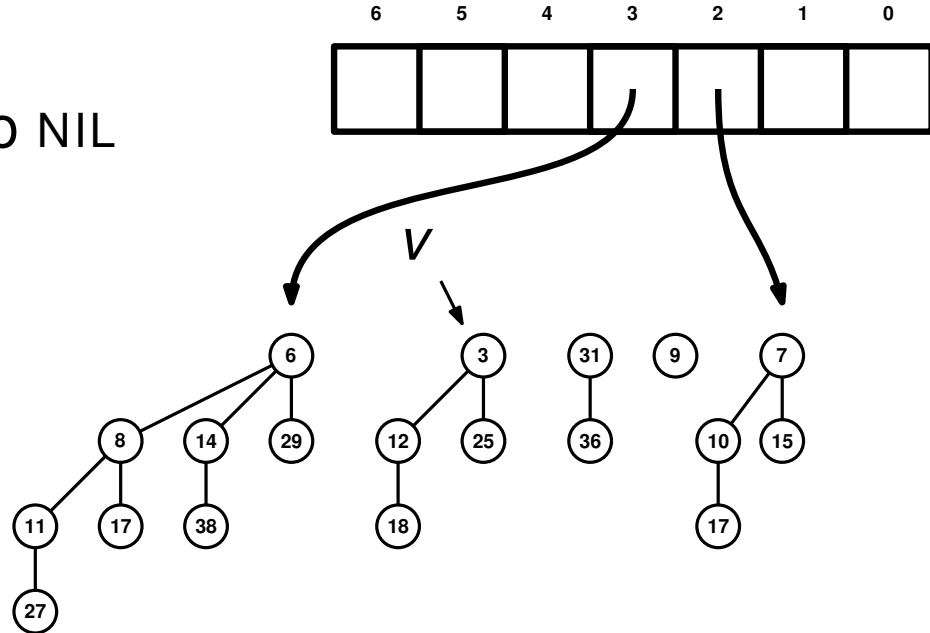
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

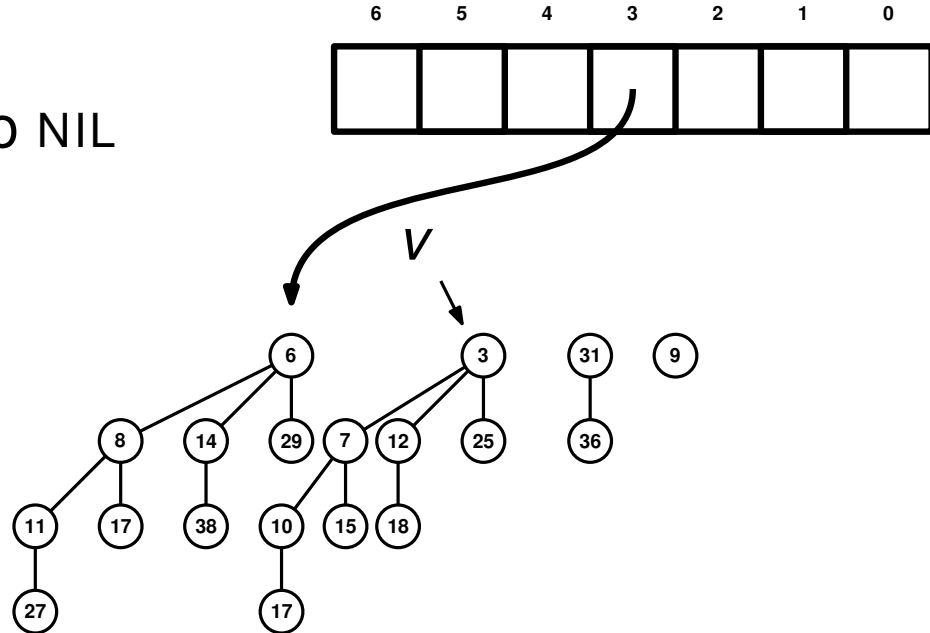
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

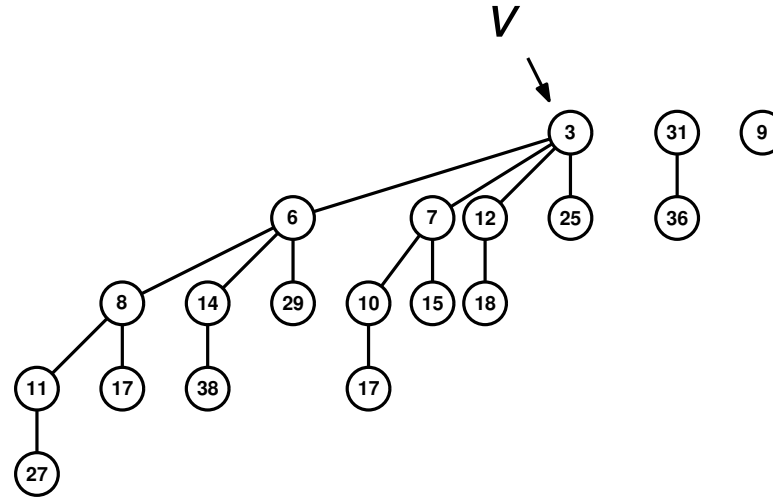
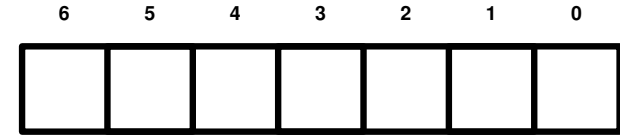
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

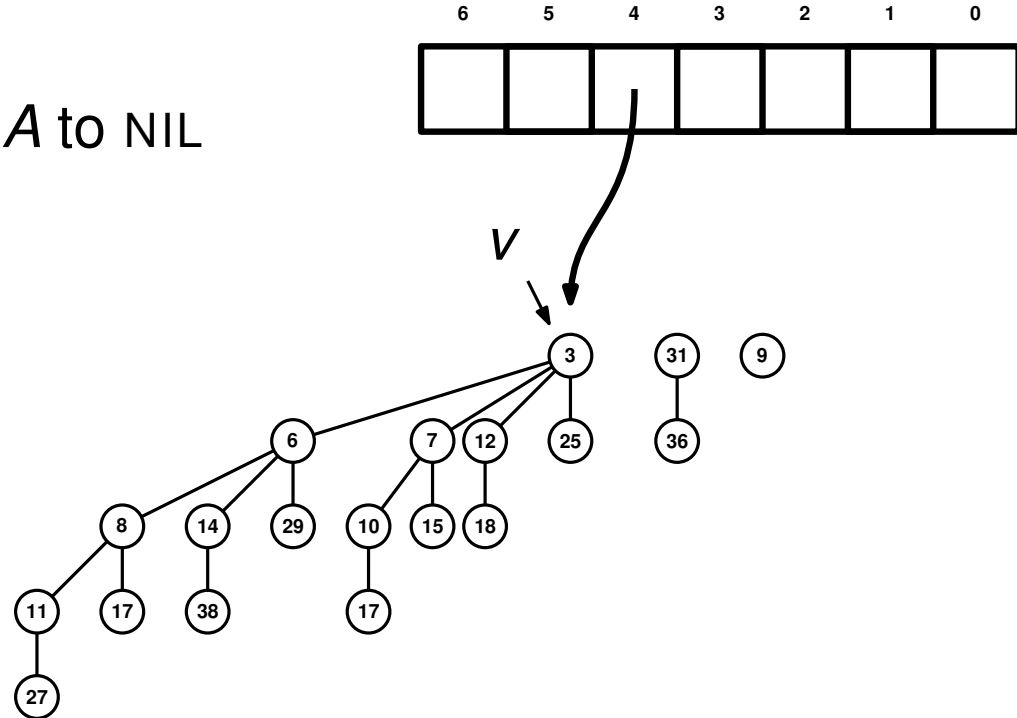
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

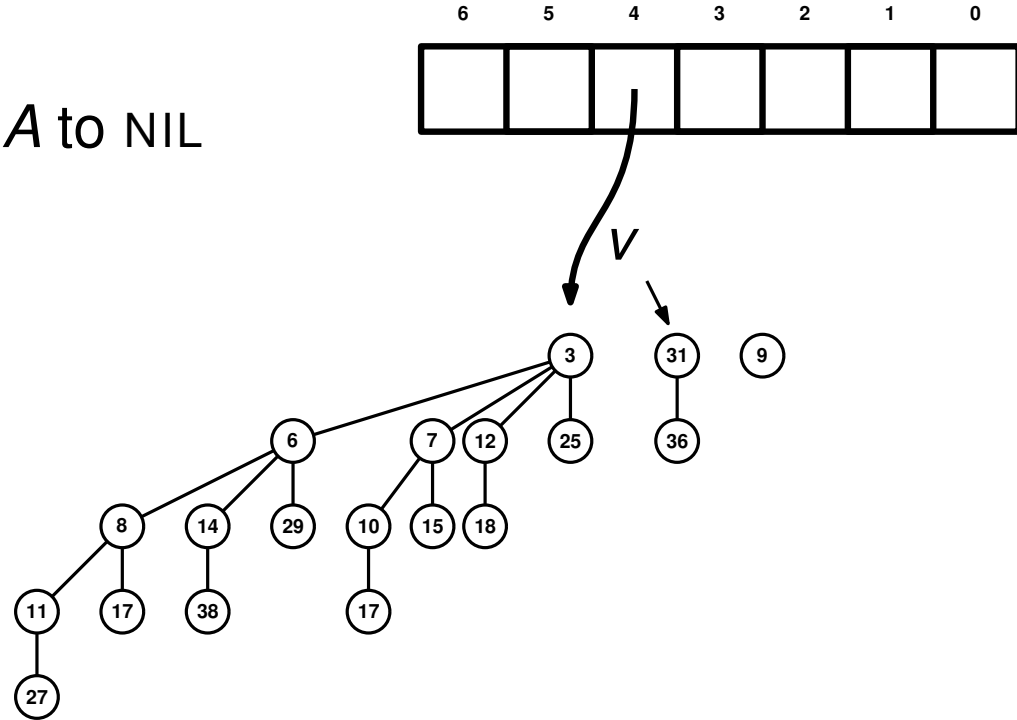
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

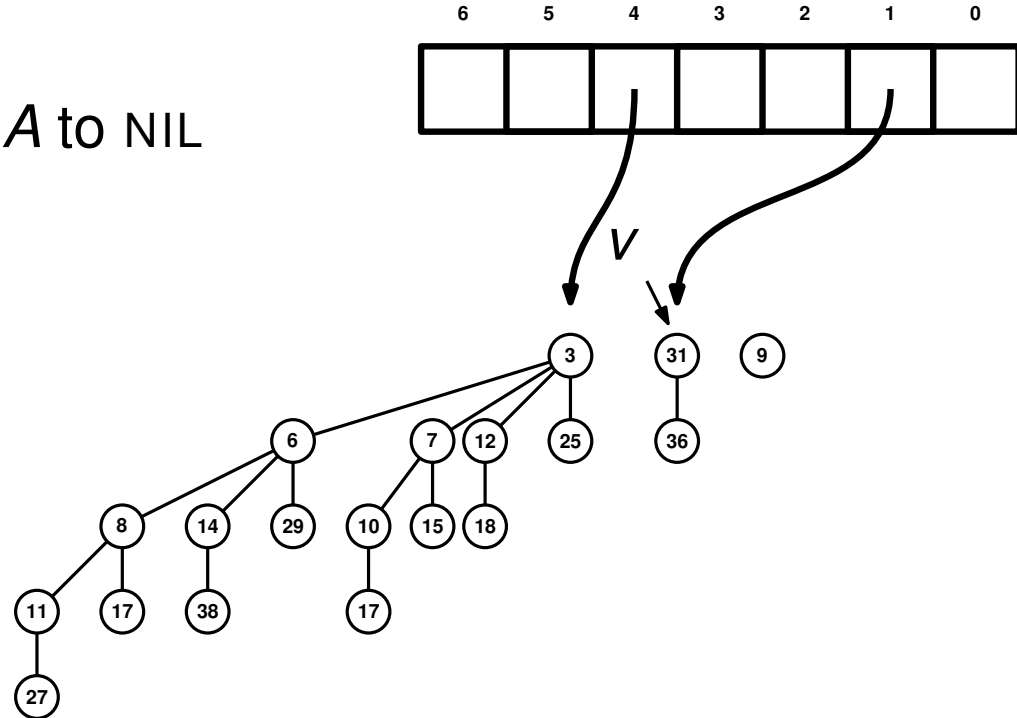
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

→ **for each** v in root list **do**

$d = v.degree$

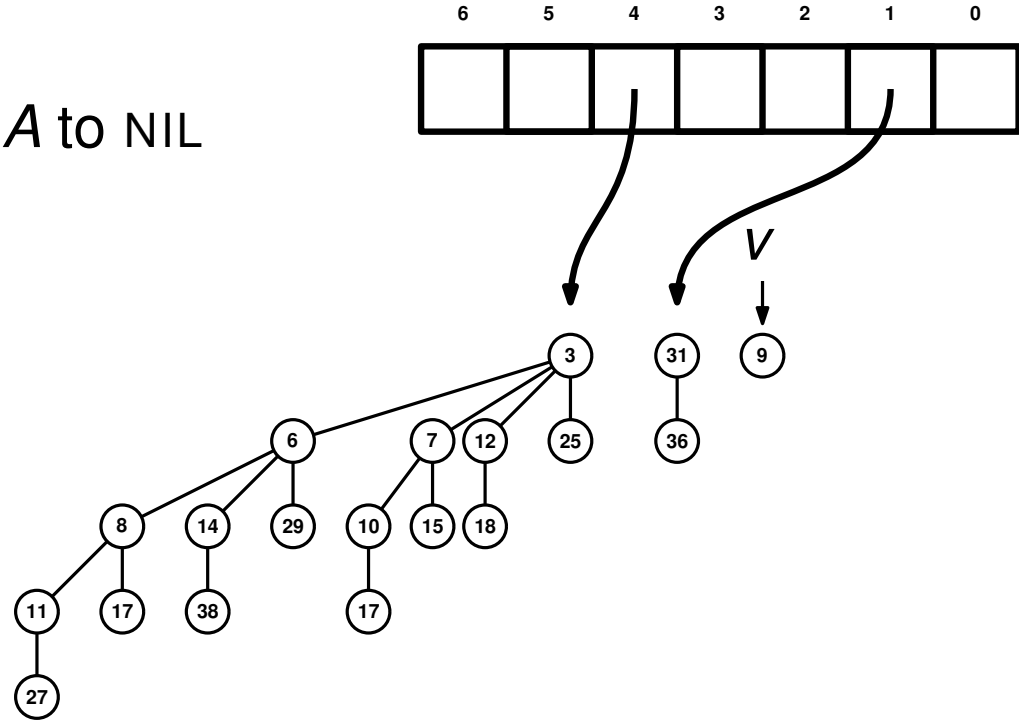
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

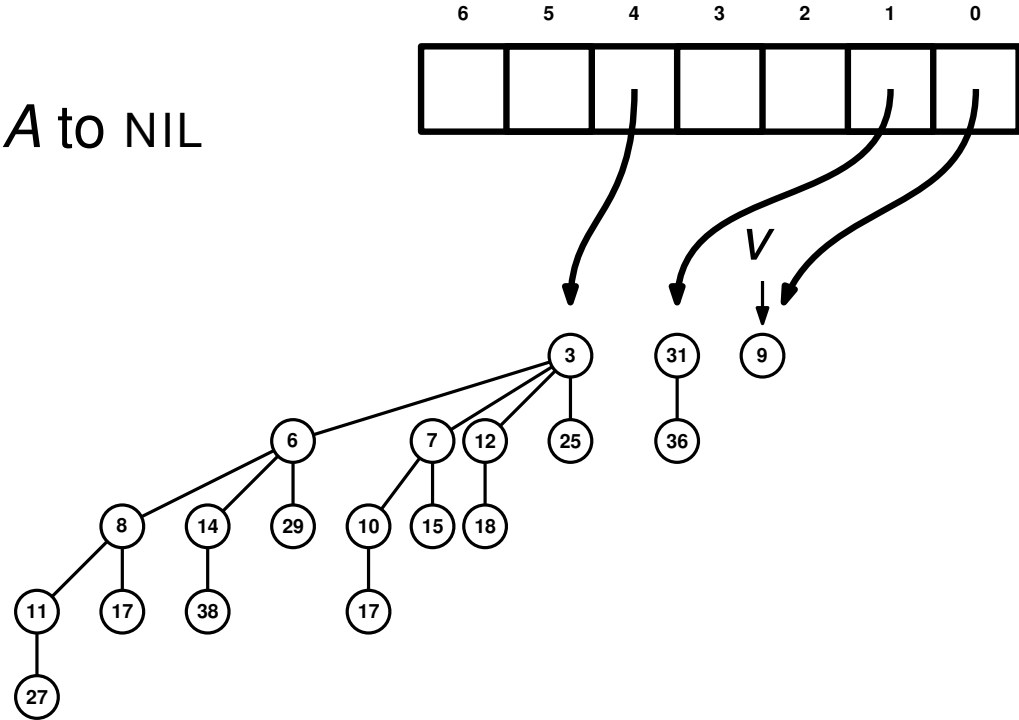
while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$

$min = +\infty$

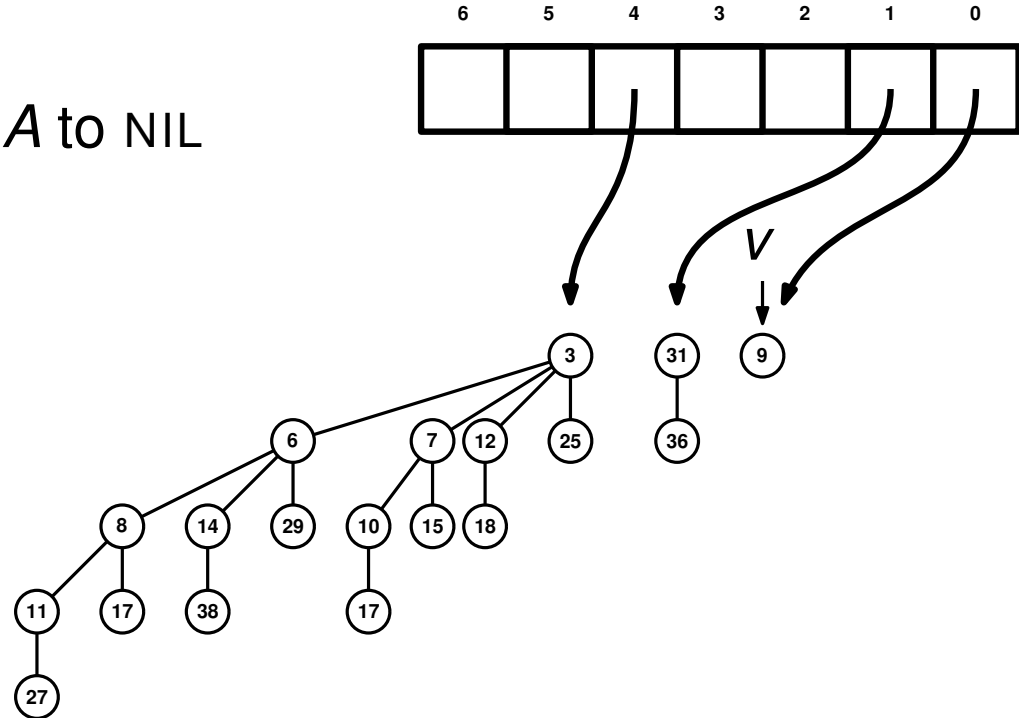
for $i = 0$ to $|A| - 1$ **do**

if $A[i] \neq \text{NIL}$ **then**

Add $A[i]$ to the root list

if $A[i].key < min$ **then**

$Q.min \leftarrow A[i]; min = A[i].key$



CONSOLIDATE(Q)

function CONSOLIDATE(Q)

Initialize $O(\log n)$ -sized array A to NIL

for each v in root list **do**

$d = v.degree$

while $A[d] \neq \text{NIL}$ **do**

$v = \text{LINK}(v, A[d])$

$A[d] = \text{NIL}$

$d = d + 1$

$A[d] = v; v.parent = \text{NIL}$

$min = +\infty$

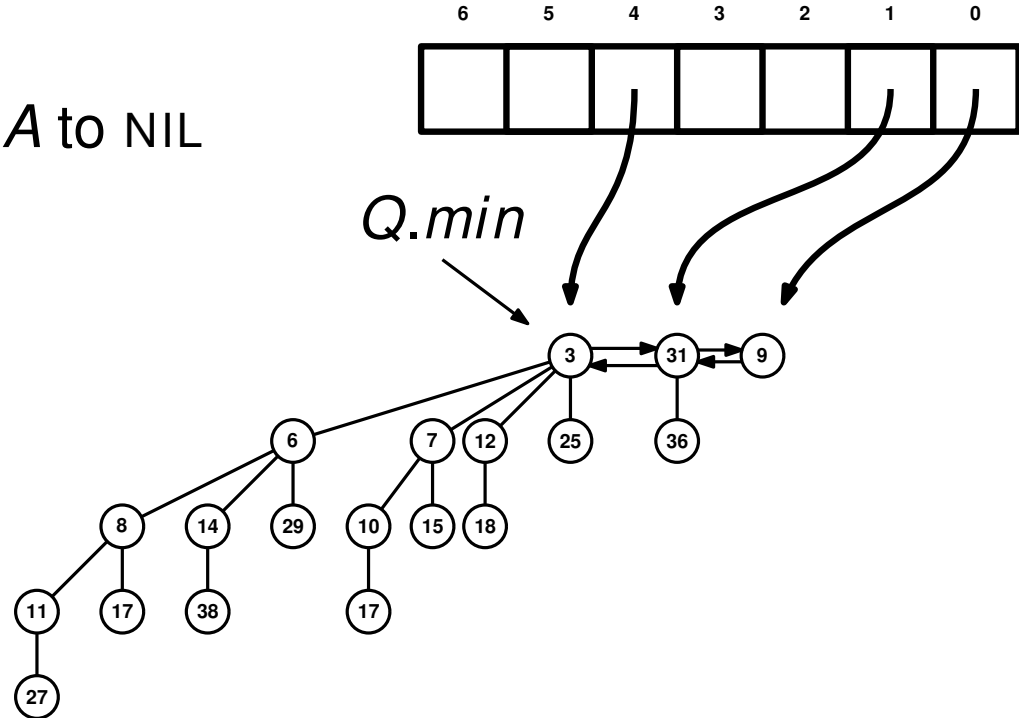
for $i = 0$ to $|A| - 1$ **do**

if $A[i] \neq \text{NIL}$ **then**

Add $A[i]$ to the root list

if $A[i].key < min$ **then**

$Q.min \leftarrow A[i]; min = A[i].key$



Bounding maximum degree

Bounding maximum degree

Lemma 1. *Let $k = v.\text{degree}$. Then for every node v in Fibonacci heap, $\text{SIZE}(v) \geq F_{k+2} \geq \phi^k$, where $\phi = \frac{1+\sqrt{5}}{2}$*

Bounding maximum degree

Lemma 1. *Let $k = v.\text{degree}$. Then for every node v in Fibonacci heap, $\text{SIZE}(v) \geq F_{k+2} \geq \phi^k$, where $\phi = \frac{1+\sqrt{5}}{2}$*

Corollary 1. $k \leq \log_{\phi} n$

Analysis of EXTRACT-MIN(Q)

Potential: $\Phi_i = \bar{k}(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- \bar{k} is a constant

Let d be the number of children of the $Q.min$ ($d \leq \log_\phi n$)

Analysis of EXTRACT-MIN(Q)

Potential: $\Phi_i = \bar{k}(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- \bar{k} is a constant

Let d be the number of children of the $Q.min$ ($d \leq \log_\phi n$)

$$\Delta\Phi_i = \bar{k}(t_i - t_{i-1}) + 2\bar{k}(m_i - m_{i-1}) \leq \bar{k}(t_i - t_{i-1})$$

because $m_i \leq m_{i-1}$



Analysis of EXTRACT-MIN(Q)

Potential: $\Phi_i = \bar{k}(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- \bar{k} is a constant

Let d be the number of children of the $Q.min$ ($d \leq \log_\phi n$)

$$\Delta\Phi_i = \bar{k}(t_i - t_{i-1}) + 2\bar{k}(m_i - m_{i-1}) \leq \bar{k}(t_i - t_{i-1})$$

because $m_i \leq m_{i-1}$

- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log_\phi n \leq O(1) + t_{i-1} + 2 \log_\phi n$

Analysis of EXTRACT-MIN(Q)

Potential: $\Phi_i = \bar{k}(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- \bar{k} is a constant

Let d be the number of children of the $Q.min$ ($d \leq \log_\phi n$)

$$\Delta\Phi_i = \bar{k}(t_i - t_{i-1}) + 2\bar{k}(m_i - m_{i-1}) \leq \bar{k}(t_i - t_{i-1})$$

because $m_i \leq m_{i-1}$

- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log_\phi n \leq O(1) + t_{i-1} + 2 \log_\phi n$

$$\hat{c}_i = c_i + \Delta\Phi_i \leq O(1) + 2 \log_\phi n + \bar{k} \cdot t_i - (\bar{k} - 1) \cdot t_{i-1}$$

Analysis of EXTRACT-MIN(Q)

Potential: $\Phi_i = \bar{k}(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- \bar{k} is a constant

Let d be the number of children of the $Q.min$ ($d \leq \log_\phi n$)

$$\Delta\Phi_i = \bar{k}(t_i - t_{i-1}) + 2\bar{k}(m_i - m_{i-1}) \leq \bar{k}(t_i - t_{i-1})$$

because $m_i \leq m_{i-1}$

- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log_\phi n \leq O(1) + t_{i-1} + 2 \log_\phi n$

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi_i \leq O(1) + 2 \log_\phi n + \bar{k} \cdot t_i - (\bar{k} - 1) \cdot t_{i-1} \\ &\leq O(1) + 2 \log_\phi n + \bar{k} \cdot t_i\end{aligned}$$

for any $\bar{k} \geq 1$

Analysis of EXTRACT-MIN(Q)

Potential: $\Phi_i = \bar{k}(t_i + 2m_i)$

- $t_i = \#$ of root list trees
- $m_i = \#$ of marked nodes
- \bar{k} is a constant

Let d be the number of children of the $Q.min$ ($d \leq \log_\phi n$)

$$\Delta\Phi_i = \bar{k}(t_i - t_{i-1}) + 2\bar{k}(m_i - m_{i-1}) \leq \bar{k}(t_i - t_{i-1})$$

because $m_i \leq m_{i-1}$

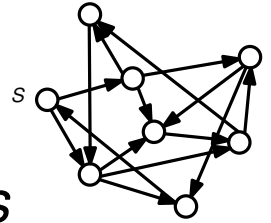
- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log_\phi n \leq O(1) + t_{i-1} + 2 \log_\phi n$

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi_i \leq O(1) + 2 \log_\phi n + \bar{k} \cdot t_i - (\bar{k} - 1) \cdot t_{i-1} \\ &\leq O(1) + 2 \log_\phi n + \bar{k} \cdot t_i \leq O(1) + 2 \log_\phi n + \bar{k} \log_\phi n = O(\log n)\end{aligned}$$

for any $\bar{k} \geq 1$

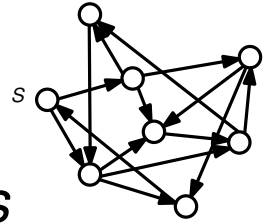
because there are $t_i \leq \log n$ trees in the root list after consolidation

Application: Single Source Shortest Paths



- Input: Graph G with n vertices, m edges with weights, vertex s
- Output: minimum-weight distance from s to every other vertex of G

Application: Single Source Shortest Paths



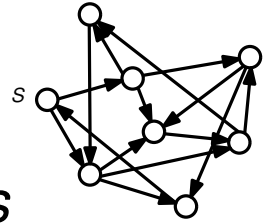
- Input: Graph G with n vertices, m edges with weights, vertex s
- Output: minimum-weight distance from s to every other vertex of G

Dijkstra's Algorithm

Use priority queue using

- n calls to INSERT
- n calls to EXTRACT-MIN
- m calls to DECREASE-KEY

Application: Single Source Shortest Paths



- Input: Graph G with n vertices, m edges with weights, vertex s
- Output: minimum-weight distance from s to every other vertex of G

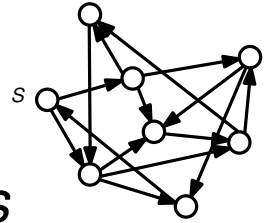
Dijkstra's Algorithm

Use priority queue using

- n calls to INSERT
- n calls to EXTRACT-MIN
- m calls to DECREASE-KEY

	Binary	Binomial	Fibonacci
	$O(1)$	$O(1)^*$	$O(1)$
	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
	$O(\log n)$	$O(\log n)$	$O(1)^*$

Application: Single Source Shortest Paths



- Input: Graph G with n vertices, m edges with weights, vertex s
- Output: minimum-weight distance from s to every other vertex of G

Dijkstra's Algorithm

Use priority queue using

- n calls to INSERT
- n calls to EXTRACT-MIN
- m calls to DECREASE-KEY

Binary

Binomial

Fibonacci

$O(1)$

$O(1)^*$

$O(1)$

$O(\log n)$

$O(\log n)$

$O(\log n)^*$

$O(\log n)$

$O(\log n)$

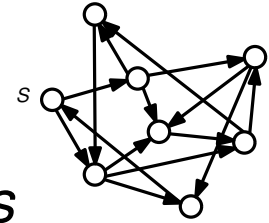
$O(1)^*$

Binary

Binomial

Fibonacci

Application: Single Source Shortest Paths



- Input: Graph G with n vertices, m edges with weights, vertex s
- Output: minimum-weight distance from s to every other vertex of G

Dijkstra's Algorithm

Use priority queue using

- n calls to INSERT
- n calls to EXTRACT-MIN
- m calls to DECREASE-KEY

	Binary	Binomial	Fibonacci
	$O(1)$	$O(1)^*$	$O(1)$
	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
	$O(\log n)$	$O(\log n)$	$O(1)^*$

Binary

$$O(n + n \log n + m \log n) \\ = O(m \log n)$$

Binomial

$$O(n + n \log n + m \log n) \\ = O(m \log n)$$

Fibonacci

$$O(n + m + n \log n) \\ = O(m + n \log n)$$