Algorytmy i struktury danych.

Wykład 6

Krzysztof M. Ocetkiewicz

`Krzysztof.Ocetkiewicz@eti.pg.gda.pl`

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Struktury złączalne

Koszt zamortyzowany

- badamy pesymistyczny czas wykonania n operacji — $T(n)$
- uśredniamy koszt $T(n)$ po wszystkich operacjach — koszt zamortyzowany pojedynczej operacji wynosi $T(n)/n$
- np. tablica “dynamiczna” — po wypełnieniu całej tablicy alokujemy nowe, większe miejsce i przepisujemy wszystkie elementy
- koszt pojedynczego zwiększenia tablicy — $O(k)$, gdzie k to liczba elementów w tablicy

Koszt zamortyzowany

- zaczynamy od rozmiaru początkowego k, wykonujemy n wstawień elementu
- jeżeli powiększamy tablicę o stały rozmiar (co k), wykonamy n/k powiększeń, przepisując $k, 2k, 3k, \ldots n-k$ elementów
- w sumie wykonamy $\frac{n}{2} + \frac{n^2}{2k}$ przypisań, czyli n operacji wymaga $O(n^2)$ kroków (k jest stałą)
- koszt zamortyzowany pojedynczego wstawienia to $O(n^2)/n = O(n)$

Koszt zamortyzowany

- jeżeli powiększamy tablicę dwukrotnie, wykonamy $\log_2 n/k$ powiększeń, przepisując $k, 2k, 4k, ..., n-k$ elementów
- w sumie wykonamy $\leq n + 2n$ przypisań czyli n operacji wymaga $O(n)$ kroków
- koszt zamortyzowany pojedynczego wstawienia to $O(n)/n = O(1)$

Kopce złączalne

- `MakeHeap()` — utworzenie pustego kopca
- `Insert`(H, x) — wstawienie do kopca H klucza x
- `Minimum`(H) — znalezienie najmniejszego klucza w H
- `ExtractMin`(H) — znalezienie i usunięcie z kopca H najmniejszego klucza
- `Union`(H_1, H_2) — utworzenie nowego kopca, zawierającego wszystkie węzły z H_1 i H_2 (kopce te są w wyniku operacji niszczone)
- `DecreaseKey`(H, x, k) — nadanie kluczowi x nowej wartości k w kopcu H, zakłada się, że $k < x$
- `Delete`(H, x) — usunięcie klucza x z kopca H (wymaga wskaźnika na węzeł do usunięcia)

Kopce złączalne

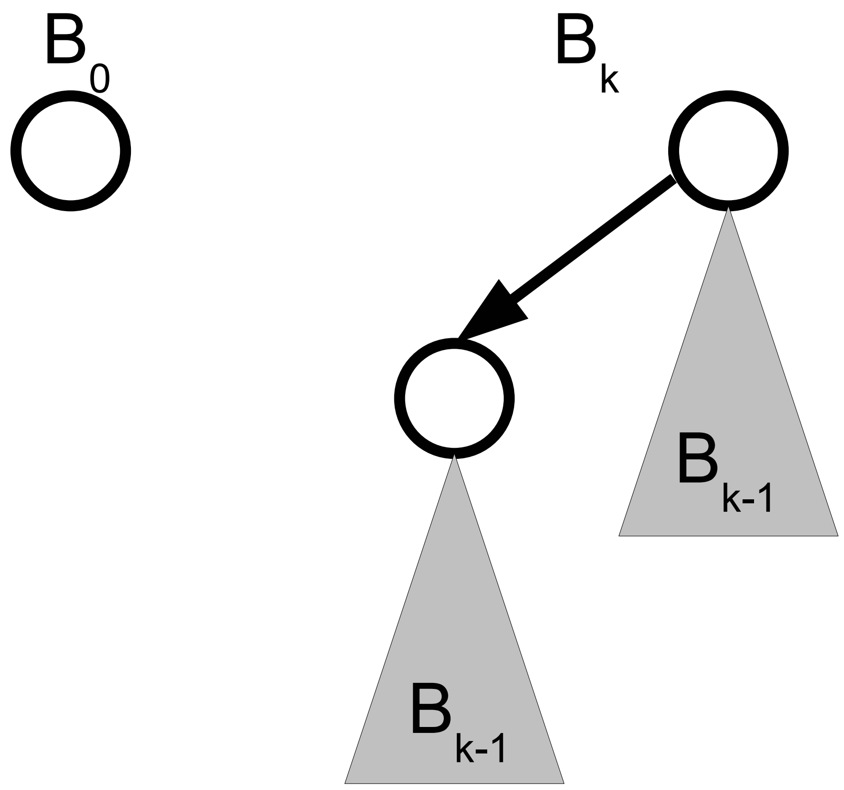
- jeżeli nie musimy łączyć kopców (`Union`), możemy posłużyć się zwykłym kopcem — pozostałe operacje zajmują w nim $O(1)$ lub $O(\log n)$
- łączenie dwóch kopców wymaga $O(n_1 + n_2)$ czasu (połączenie tablic i wykonanie `Heapify` na wynikowej tablicy)
- kopce dwumianowe
- kopce Fibonacciego

Złożoności

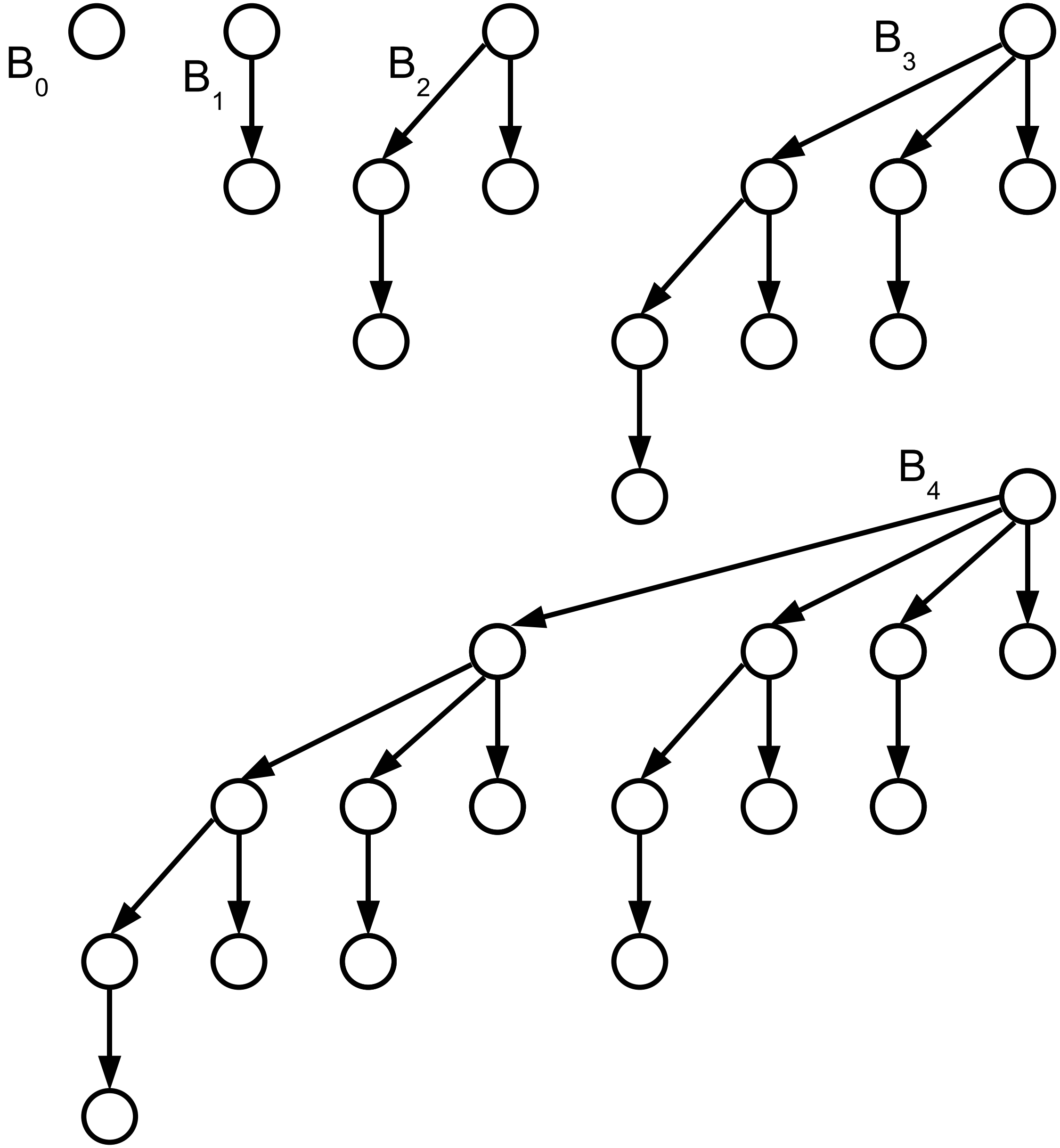
Operacja	kopiec binarny (pesym.)	kopiec dwumianowy (pesym.)	kopiec Fibonacciego (zamort.)	kopiec Fibonacciego (pesym.)
`MakeHeap`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`Insert`	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
`Minimum`	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
`ExtractMin`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
`Union`	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$
`DecreaseKey`	$O(\log n)$	$O(\log n)$	$O(1)$	$O(n)$
`Delete`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

Drzewa dwumianowe

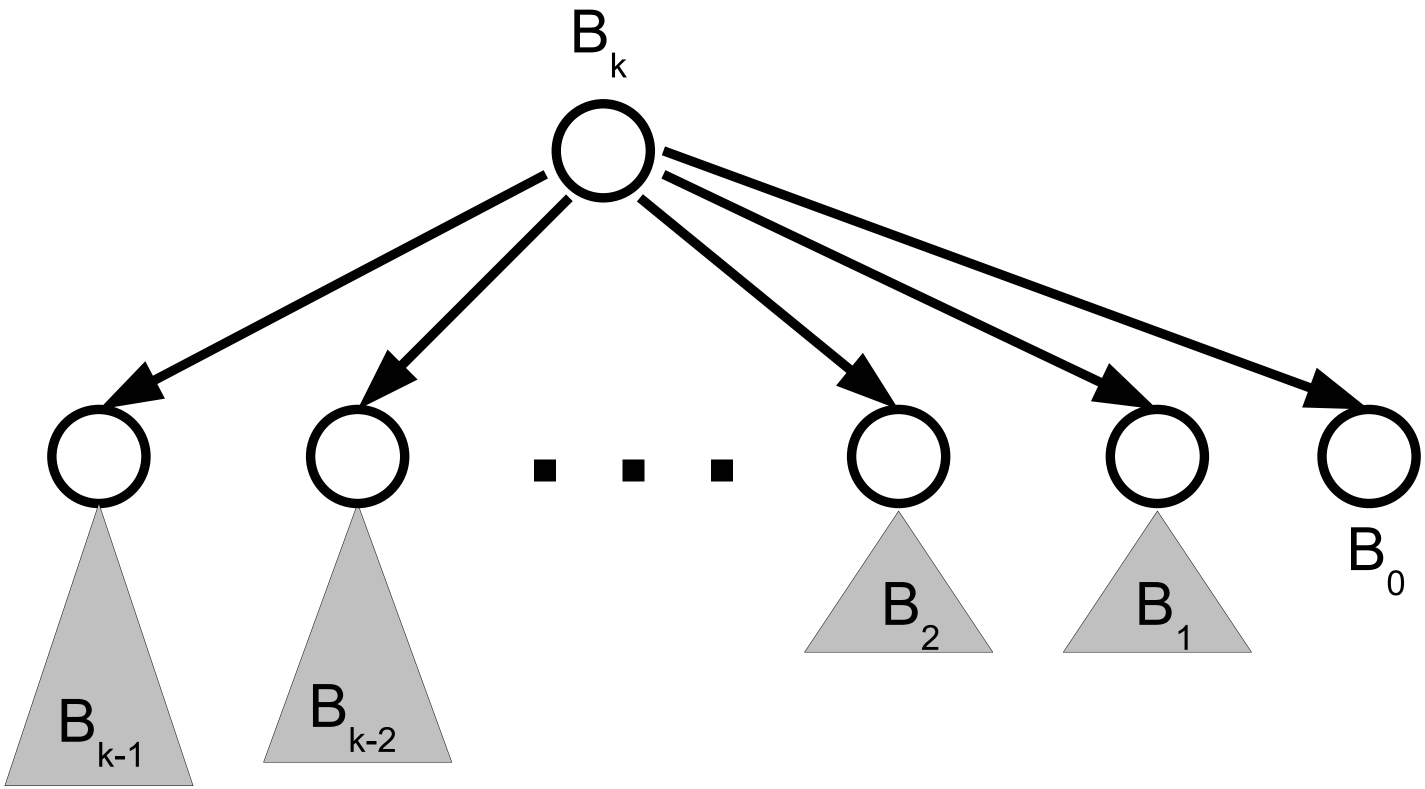
- B_0 to pojedynczy węzeł
- B_k to dwa drzewa B_{k-1}, przy czym korzeń jednego z nich jest skrajnie lewym potomkiem korzenia drugiego drzewa



Drzewa dwumianowe



Drzewa dwumianowe



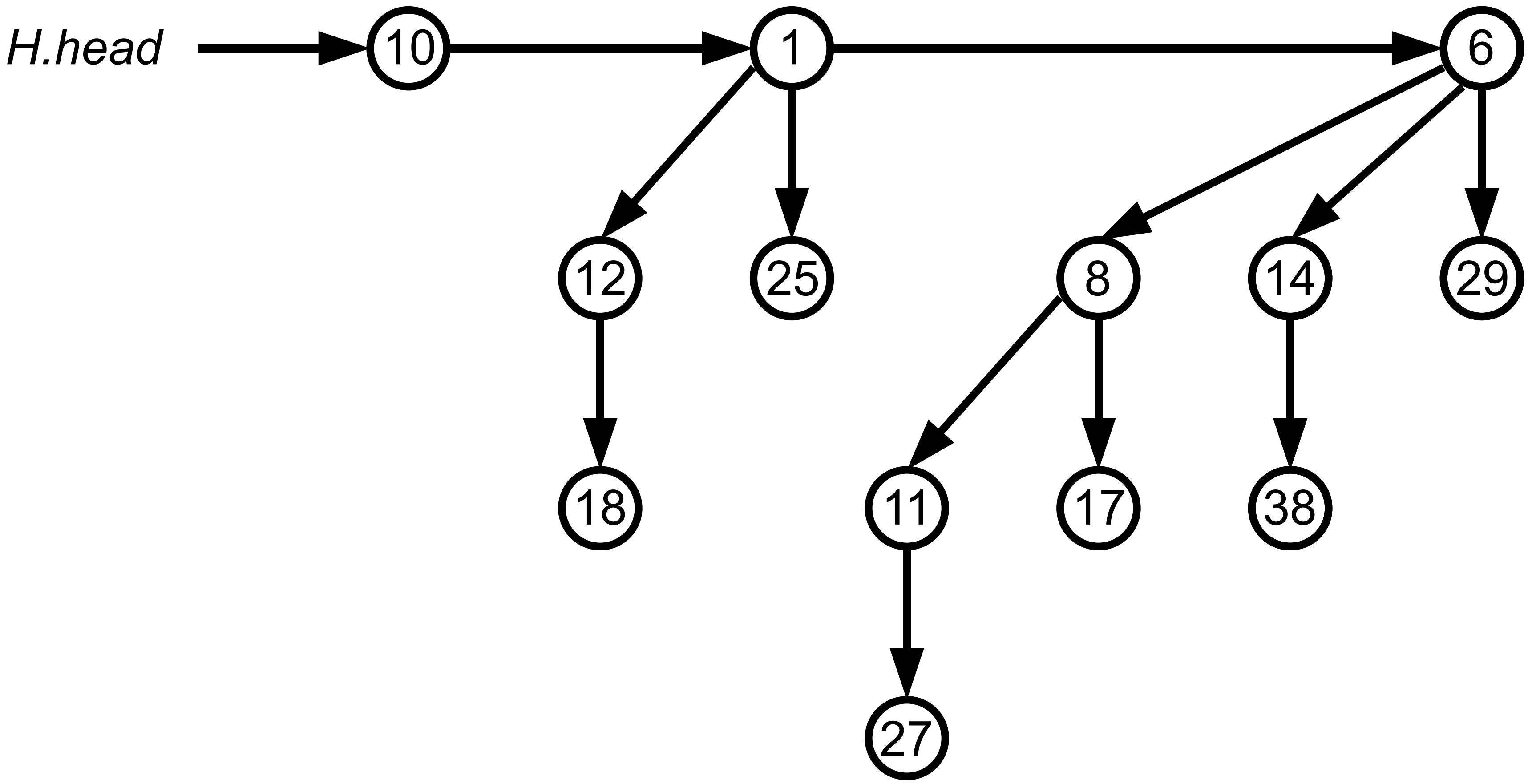
Własności

- w drzewie B_k jest 2^k węzłów (B_k to dwa drzewa B_{k-1})
- wysokość drzewa B_k wynosi k (B_k jest o 1 wyższe niż B_{k-1})
- na głębokości i znajduje się $\binom{k}{i}$ węzłów ($i = 0, 1, \ldots, k$)
- stopień korzenia wynosi k i jest większy od stopnia każdego innego węzła (tworząc B_k dokładamy jednego potomka do korzenia); kolejnymi (od lewej) potomkami korzenia są drzewa $B_{k-1}, B_{k-2}, \ldots, B_0$

Kopiec dwumianowy

- kopiec dwumianowy H jest zbiorem drzew dwumianowych, które mają następujące właściwości
  1. każde drzewo w kopcu jest uporządkowane kopcowo (klucz w rodzicu jest nie większy od kluczy w potomkach)
  2. dla każdego $d \geq 0$ istnieje w H co najwyżej jedno drzewo dwumianowe, którego korzeń ma stopień równy d
- z własności 2 wynika, że kopiec zawierający n węzłów składa się co najwyżej z $\lfloor \log n \rfloor + 1$ drzew dwumianowych

Kopiec dwumianowy



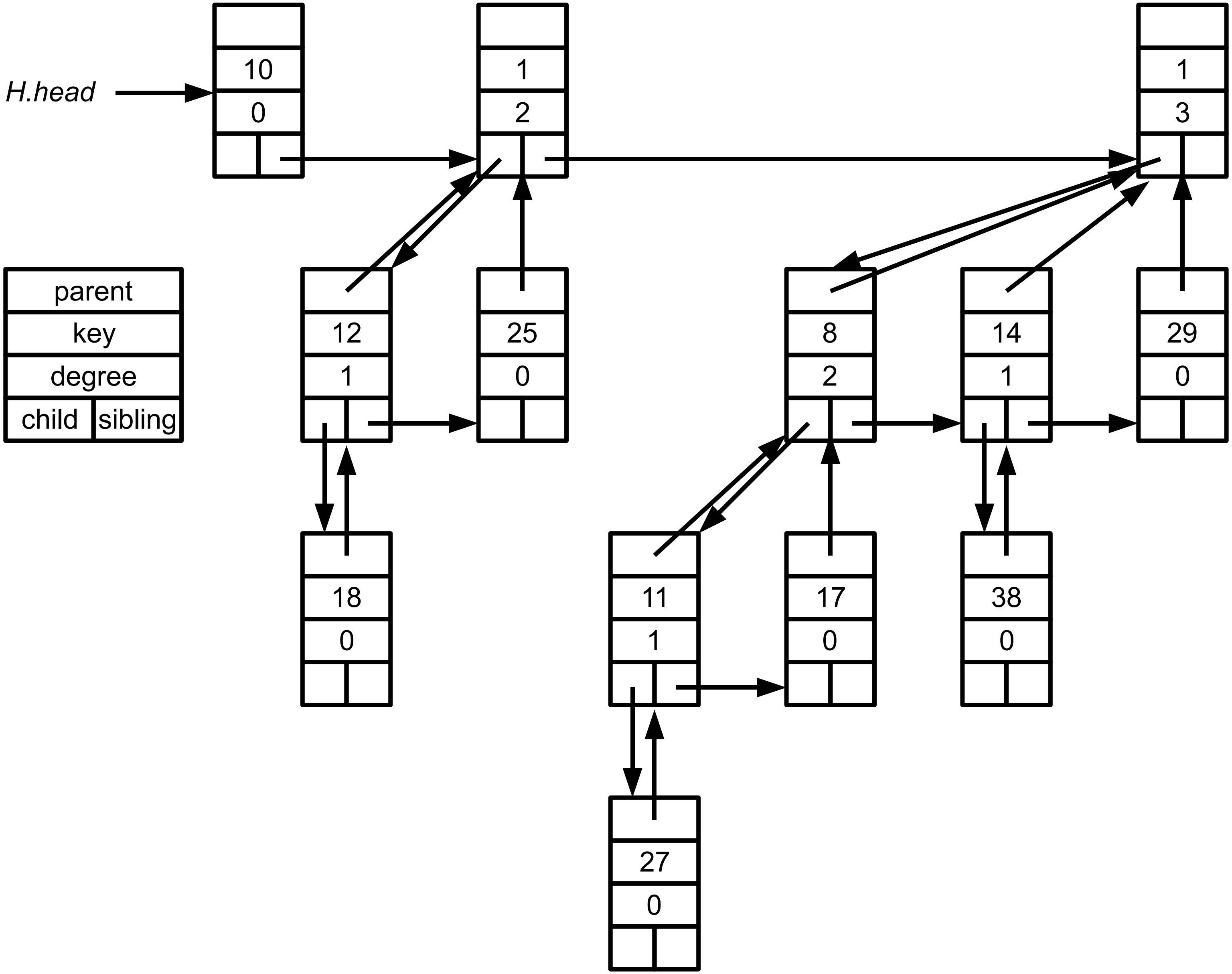
Kopiec dwumianowy

- zazwyczaj implementujemy kopiec dwumianowy przy pomocy drzewa “na lewo syn, na prawo brat”
- każdy węzeł zawiera:
  - key — klucz
  - $parent$ — wskaźnik na rodzica ($NULL$ w korzeniu)
  - $child$ — wskaźnik na pierwszego potomka ($NULL$ w liściach)
  - $sibling$ — wskaźnik na pierwszego brata ($NULL$ w skrajnie prawych potomkach)
  - $degree$ — stopień wierzchołka (liczba potomków)
  - ewentualnie dodatkowe dane powiązane z kluczem

Kopiec dwumianowy

- drzewa składające się na kopiec przechowujemy na liście
- w przypadku korzenia, pole $sibling$ wskazuje na korzeń następnego drzewa
- drzewa występują na liście w kolejności rosnących stopni
- dodatkowo pamiętamy wskaźnik na pierwszy korzeń z listy ($head$)

Kopiec dwumianowy



MakeHeap

- utworzenie nowego kopca wymaga jedynie utworzenia nowego wskaźnika ($head$) przypisania mu wartości $NULL$

`MakeHeap`(H)

1: $H =$ **new** BinomialHeap
2: $H.head = NULL$

Minimum

- minimalny klucz może znajdować się jedynie w jednym z korzeni

`Minimum(`H`)`

$w = NULL$

$t = H.head$

$min = \infty$

while $t \neq NULL$ **do**

    if $t.key < min$ **then**

        $min = t.key$

        $w = t$

    end if

    $t = t.sibling$

end while

return w

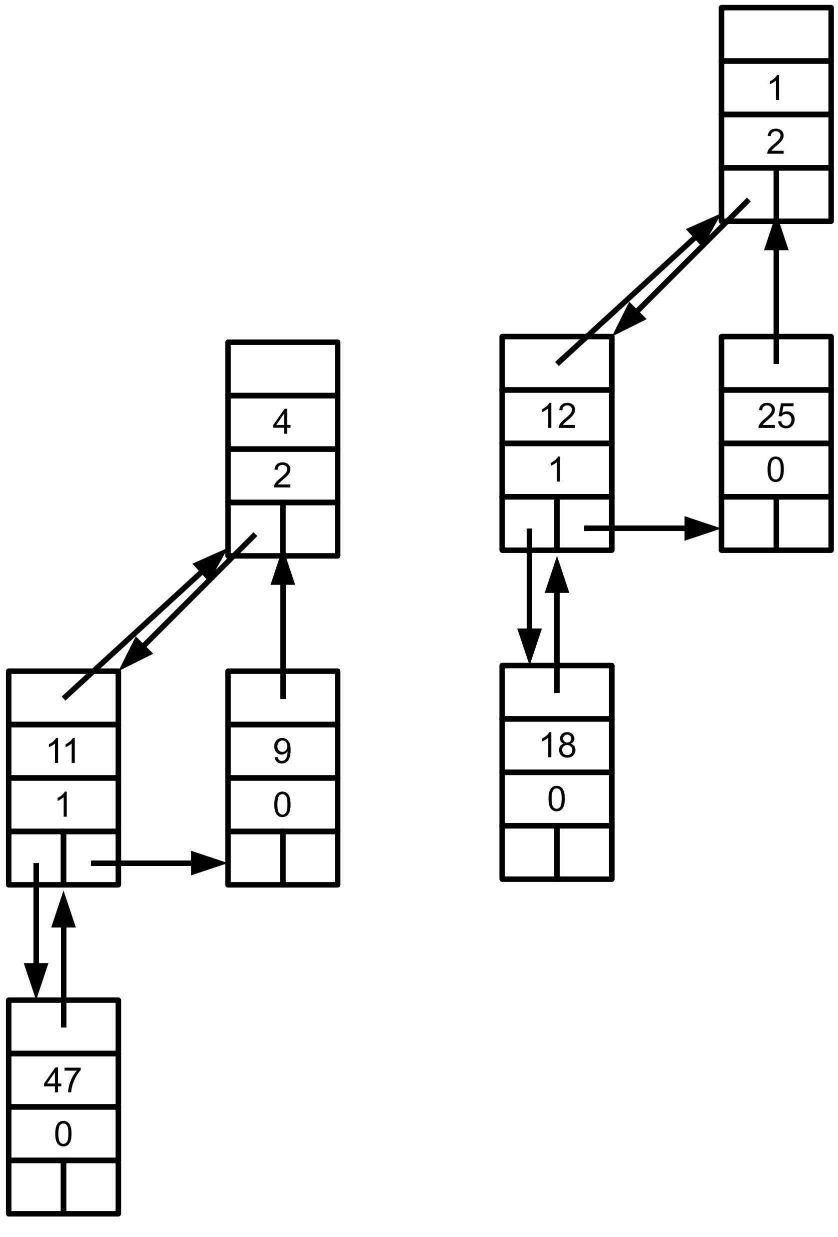
Link

- pomocnicza procedura — dołączenie drzewa B_{k-1} (którego korzeniem jest y) do drugiego drzewa B_{k-1} (o korzeniu z) — z staje się w ten sposób korzeniem drzewa B_k

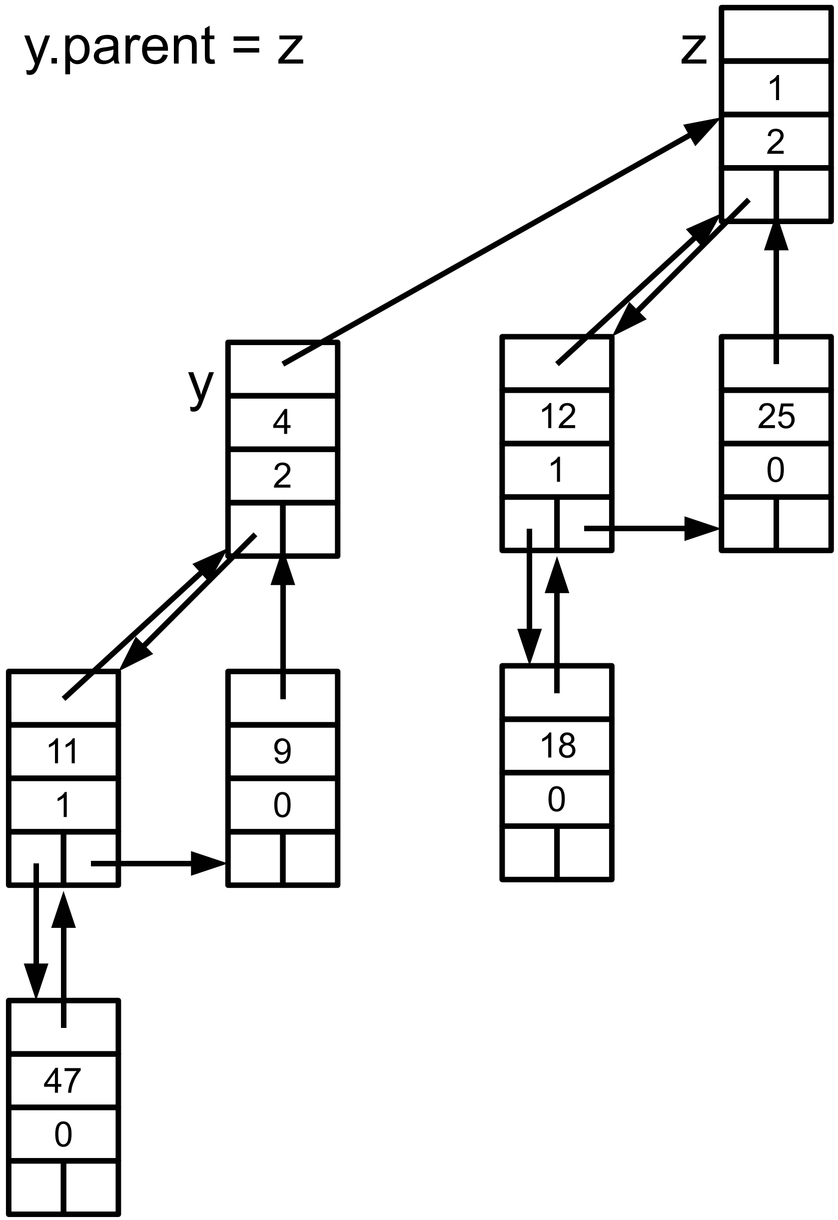
`Link`(y, z)

1: $y.parent = z$
2: $y.sibling = z.child$
3: $z.child = y$
4: $z.degree = z.degree + 1$

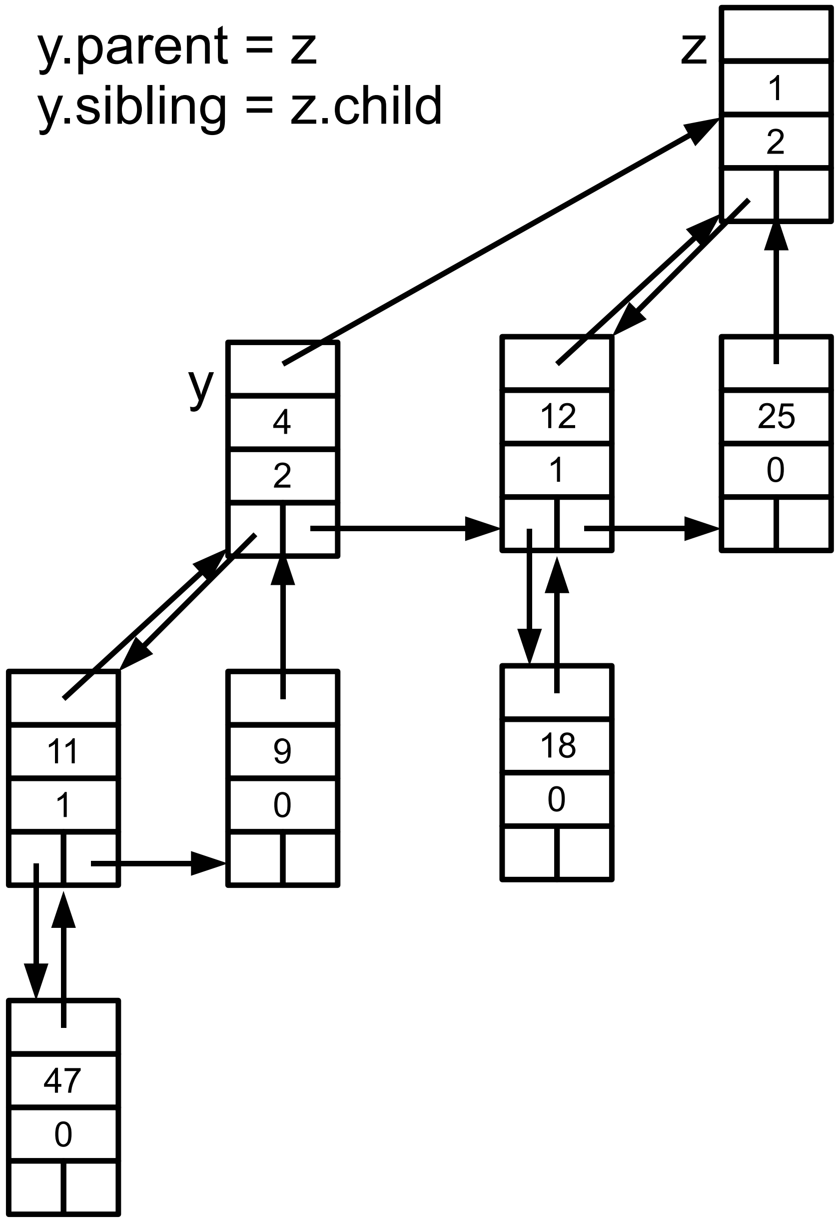
Link



Link

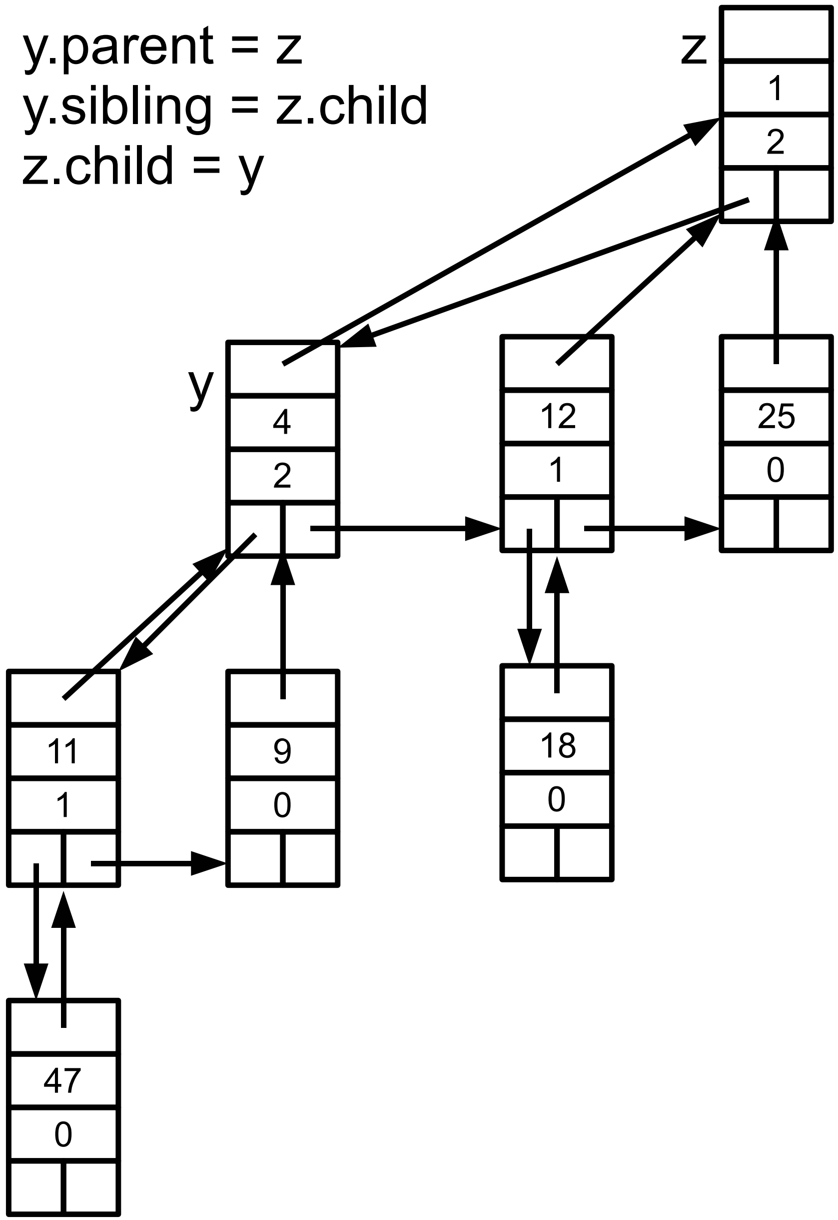


Link



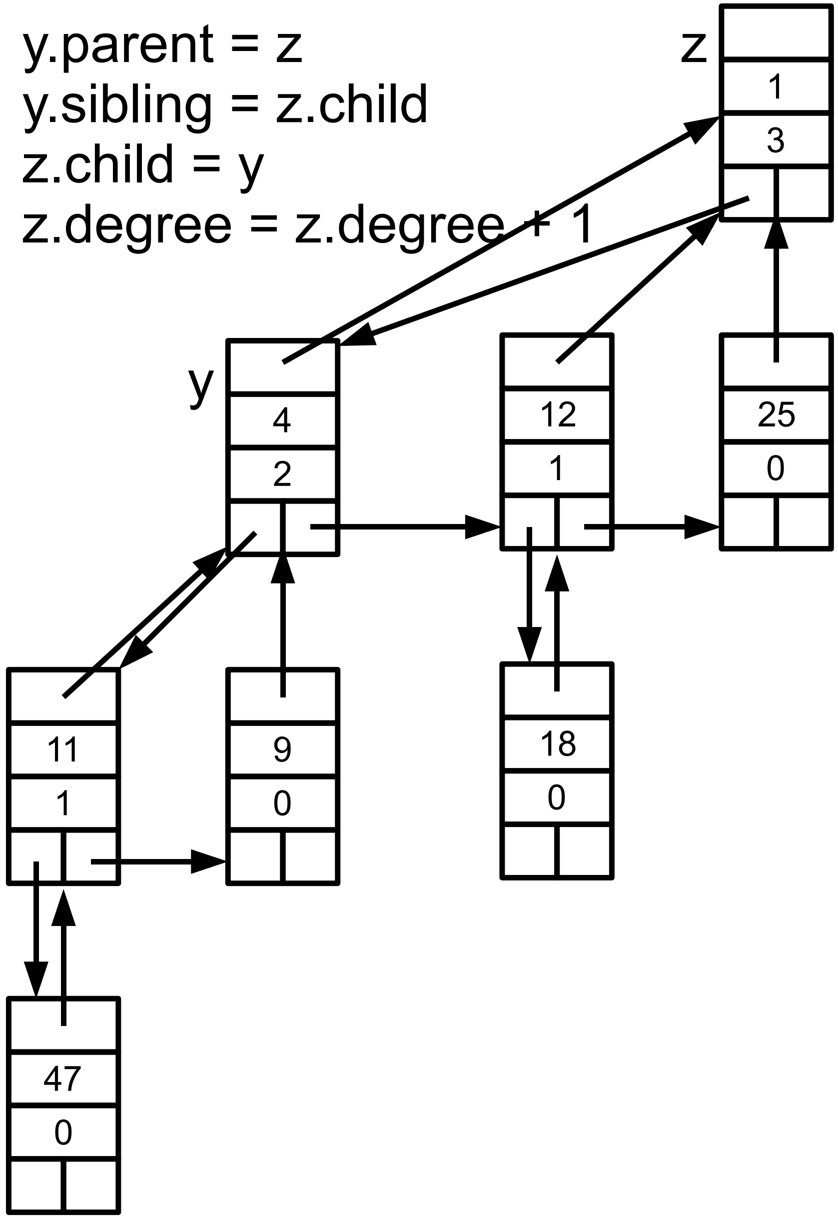
Link

```
y.parent = z
y.sibling = z.child
z.child = y
```



Link

```
y.parent = z
y.sibling = z.child
z.child = y
z.degree = z.degree + 1
```



`Union`

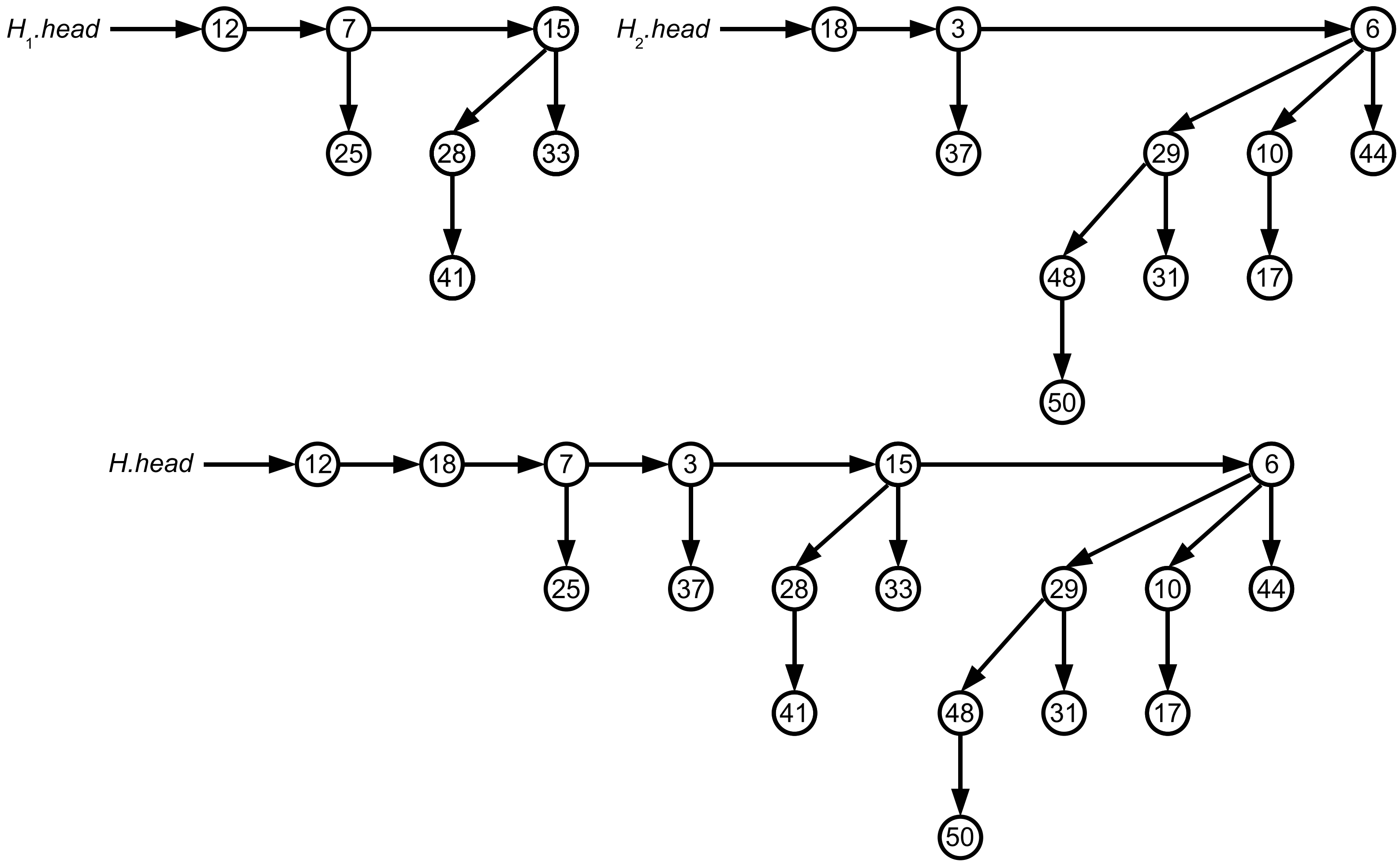
- większość pozostałych operacji na kopcu dwumianowym wykorzystuje procedurę `Union`
- operacja `Union` wykonuje dwie fazy
  - scalenie list korzeni kopców H_1 i H_2 w pojedynczą listę H uporządkowaną niemalejąco według stopni; na liście wynikowej mogą znajeźć się co najwyżej dwa korzenie o tym samym stopniu
  - łączenie korzeni o takim samym stopniu, aż zostanie co najwyżej jeden korzeń dla każdego stopnia

Merge

Merge(H_1, H_2)

$a = H_1.head$
$b = H_2.head$
if $a.degree < b.degree$ **then** $head = a$ **else** $head = b$
$tail = head$
while $a \neq NULL$ i $b \neq NULL$ **do**
    if $a.degree < b.degree$ **then**
        $tail.sibling = a$
        $a = a.sibling$
    else
        $tail.sibling = b$
        $b = b.sibling$
    end if
    $tail = tail.sibling$
end while
if $a \neq NULL$ **then** $tail.sibling = a$
else if $b \neq NULL$ **then** $tail.sibling = b$
else $tail.sibling = NULL$
return $head$

Merge


H1.head
12
7
15
25
28
33
41
H2.head
18
3
6
37
29
10
44
48
31
17
50
H.head
12
18
7
3
15
6
25
37
28
33
29
10
44
41
48
31
17
50

Union

- złożoność operacji `Merge` wynosi $O(m_1 + m_2)$, gdzie m_1 i m_2 to długości list korzeni H_1 i H_2 (a te rosną logarytmicznie względem liczby węzłów)

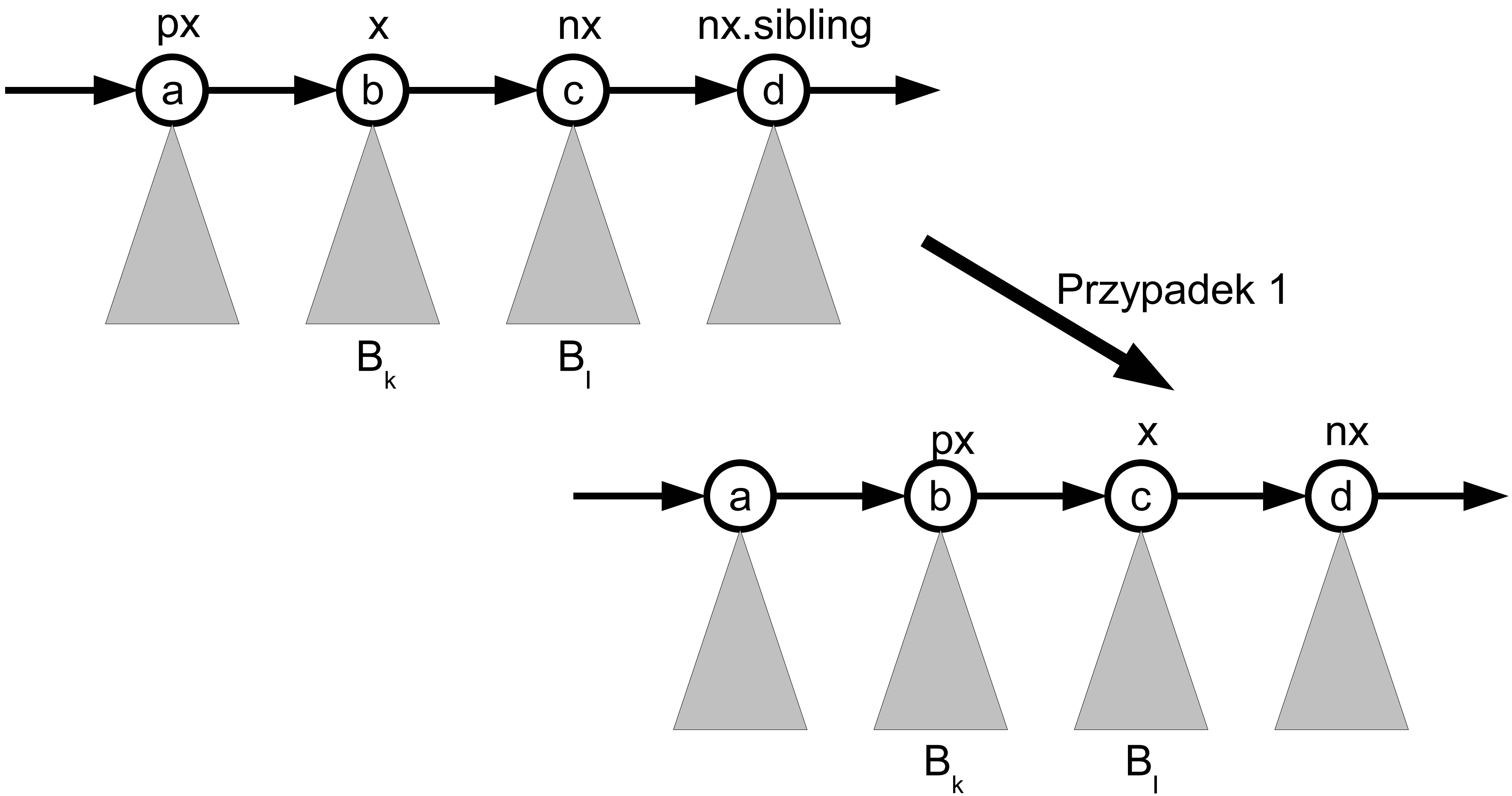
Union

- na początku drugiej fazy mamy listę H zawierającą co najwyżej po dwa drzewa tego samego stopnia
- drzewa o takim samym stopniu będą sąsiadami (gwarantuje to nam operacja `Merge`)
- wykorzystujemy trzy wskaźniki: x — to bieżący korzeń, px to korzeń porzedni oraz nx — korzeń następny ($px.sibling = x$ oraz $x.sibling = nx$)
- w każdym kroku tej fazy może wystąpić jeden z czterech przypadków:

Przypadek 1

- stopień nx jest różny (większy) od stopnia x
- w tym przypadku nie wykonujemy żadnego łącznia
- przesuwamy się do następnego korzenia

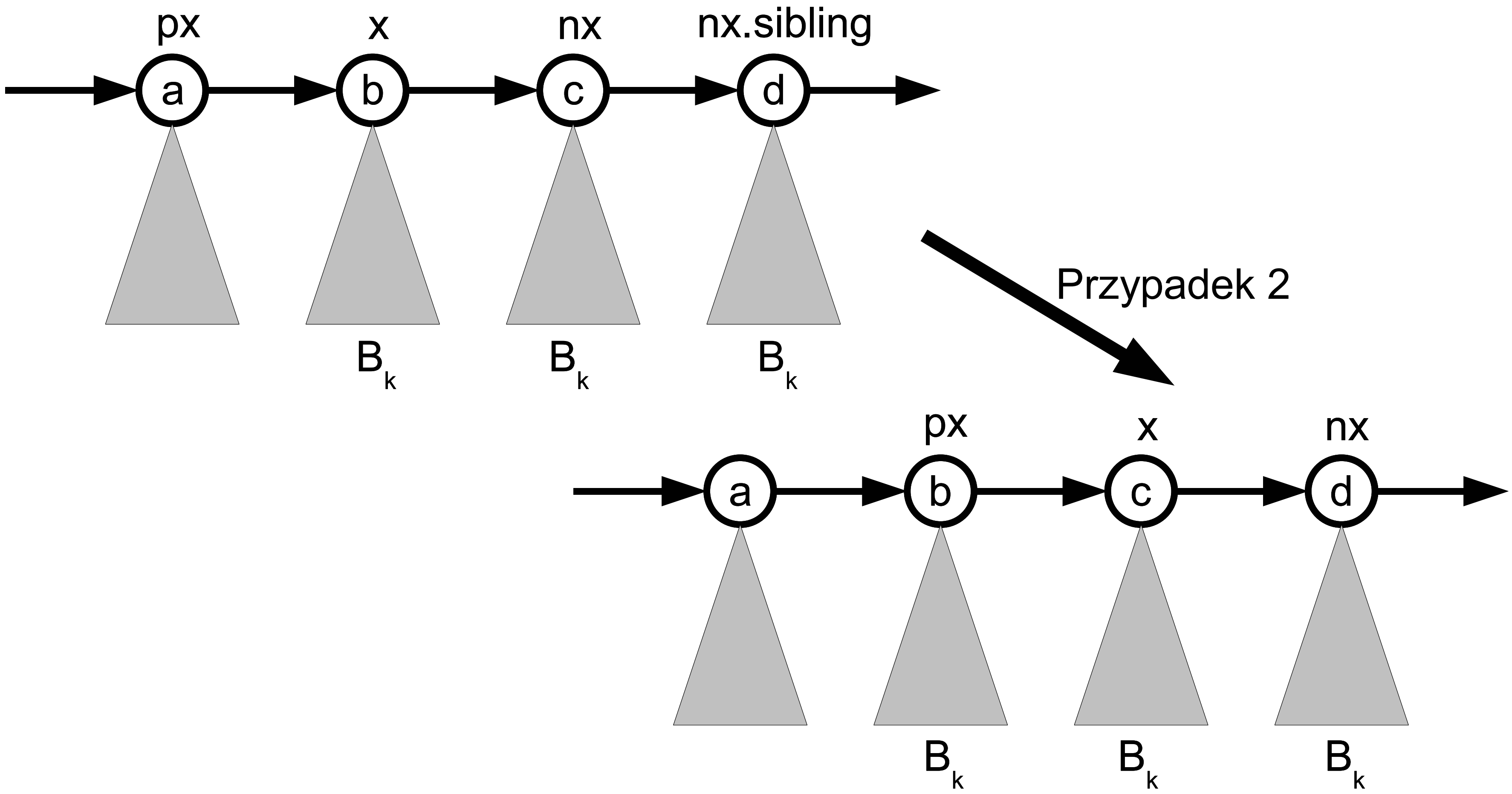
Przypadek 1



Przypadek 2

- x jest pierwszym z trzech korzeni o takich samych stopniach
- taka sytuacja może się zarzyć, gdy w wyniku scalenia list otrzymaliśmy ciąg drzew o stopniach k, k, $k+1$, $k+1$ a następnie w poprzednim kroku połączyliśmy pierwsze dwa drzewa
- postępujemy tak, jak w pierwszym przypadku — przesuwamy się do następnego korzenia
- w następnym kroku połączymy dwa drzewa stopnia $k+1$, więc na liście wynikowej zostanie tylko jedno drzewo stopnia $k+1$

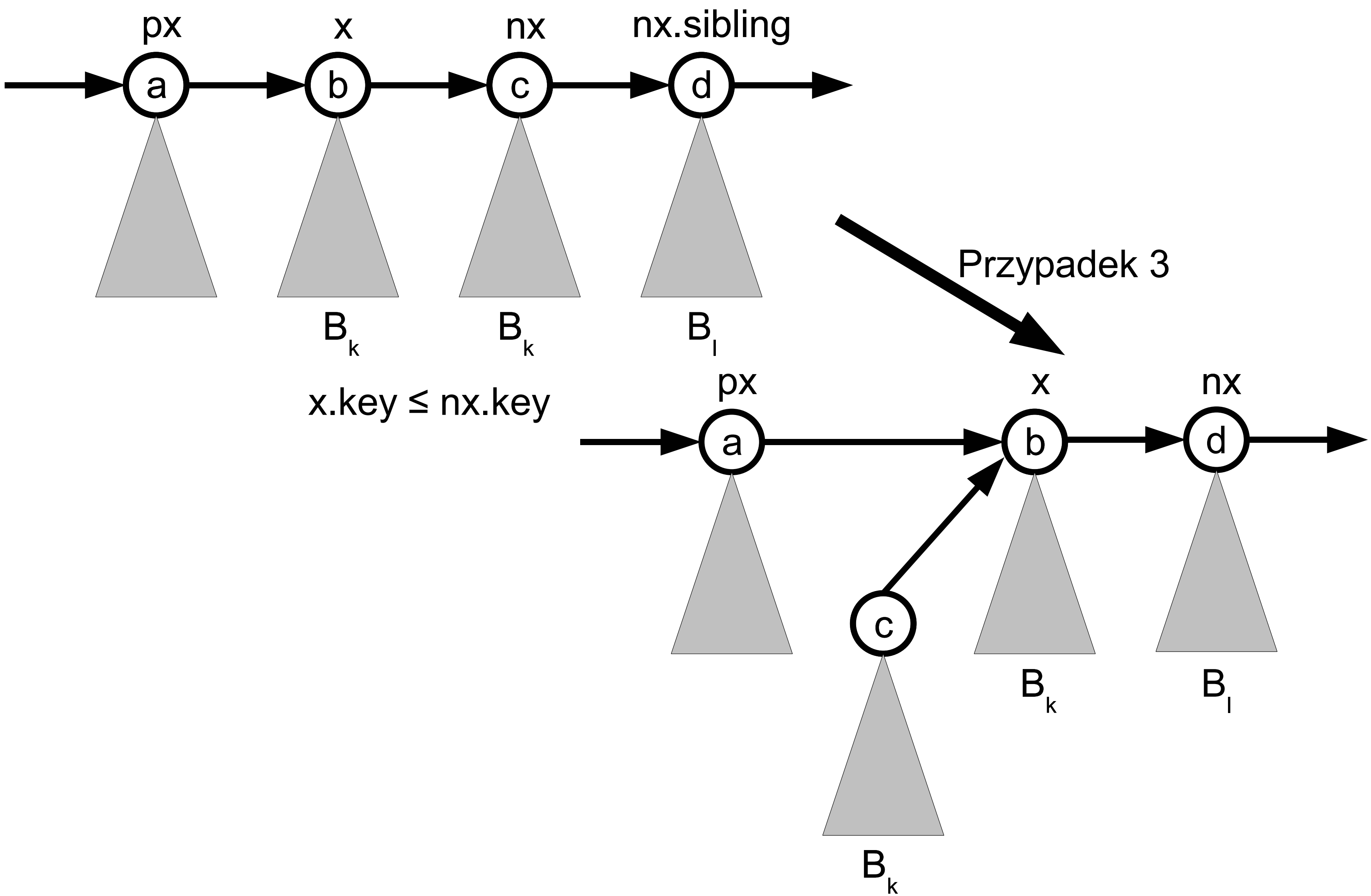
Przypadek 2



Przypadek 3

- x jest pierwszym z dwóch korzeni o takich samych stopniach oraz $x.key \leq nx.key$
- dołączamy drzewo o korzeniu nx do drzewa x
- pozostajemy w węźle x (ale lista korzeni skraca się o jeden korzeń)

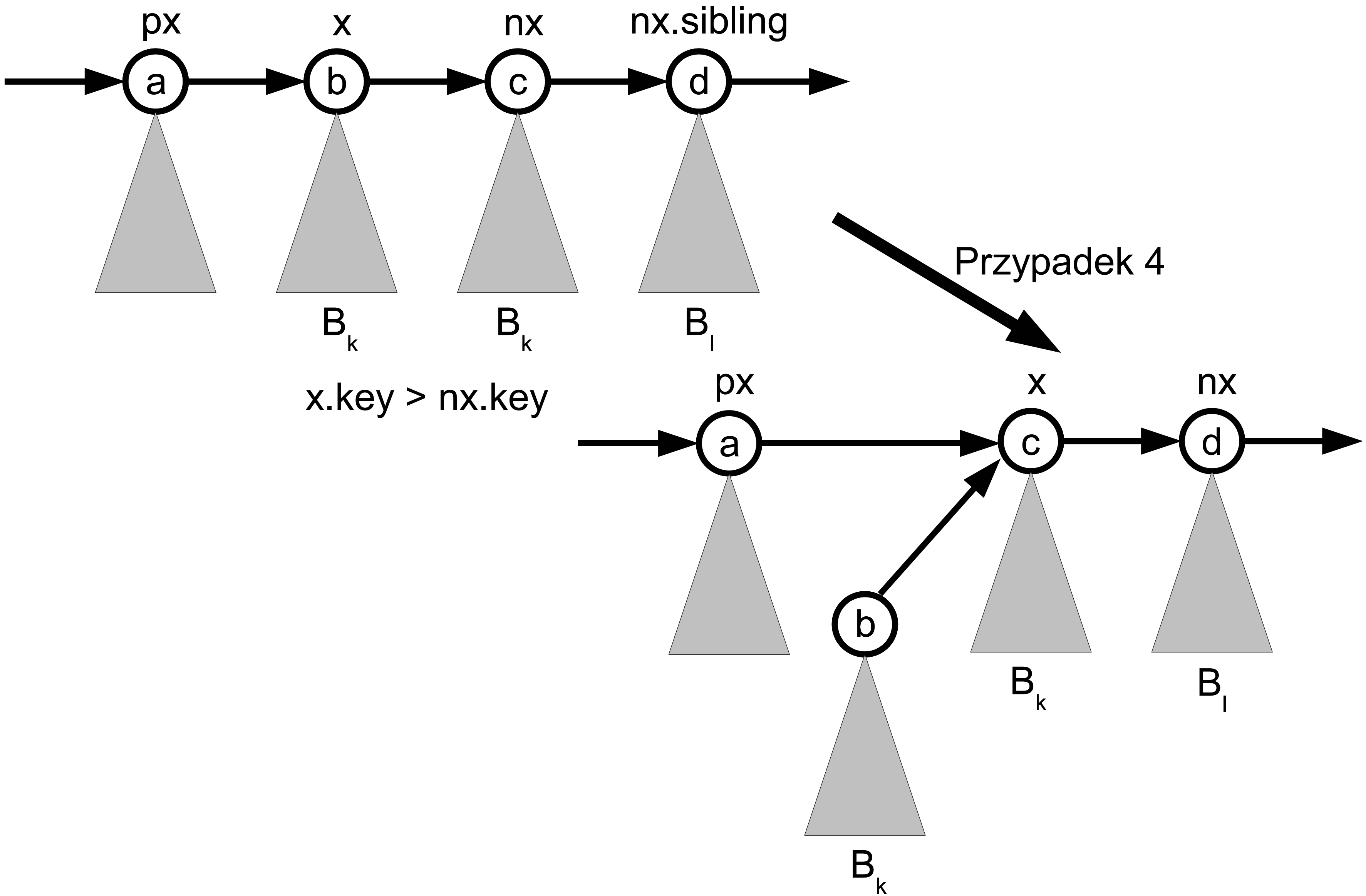
Przypadek 3


px
x
nx
nx.sibling
a
b
c
d
B_k
B_k
B_l
x.key ≤ nx.key
Przypadek 3
px
x
nx
a
b
d
c
B_k
B_l
B_k

Przypadek 4

- x jest pierwszym z dwóch korzeni o takich samych stopniach oraz $x.key > nx.key$
- dołączamy drzewo o korzeniu x do drzewa nx
- bieżącym węzłem staje się nx

Przypadek 4



px
x
nx
nx.sibling
a
b
c
d
B k
B k
B l
Przypadek 4
x.key > nx.key
px
x
nx
a
c
d
b
B k
B l
B k

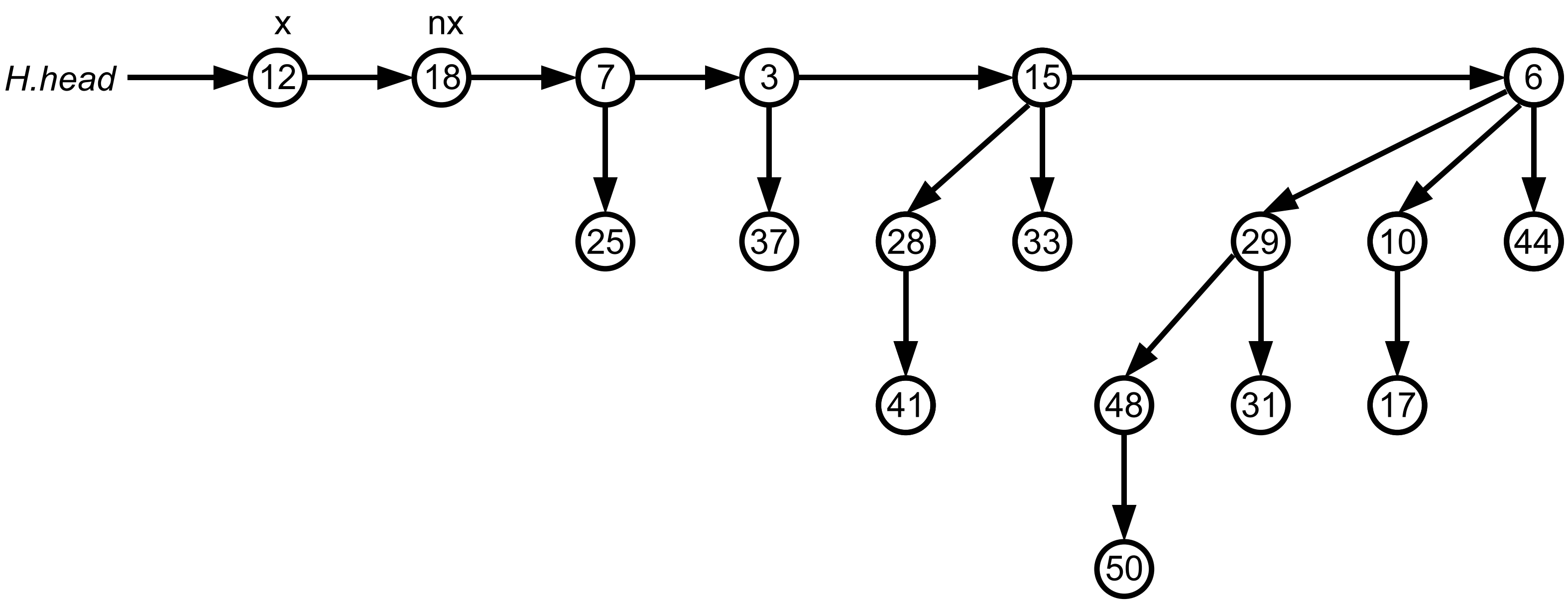
Union

- złożoność drugiej fazy również wynosi $O(\log n)$
- liczba wykonanych kroków jest równa długości listy H — w każdym kroku albo przechodzimy do następnego węzła, albo skracamy o jeden długość listy pozostałej do przejścia
- w każdym kroku wykonujemy stałą liczbę operacji

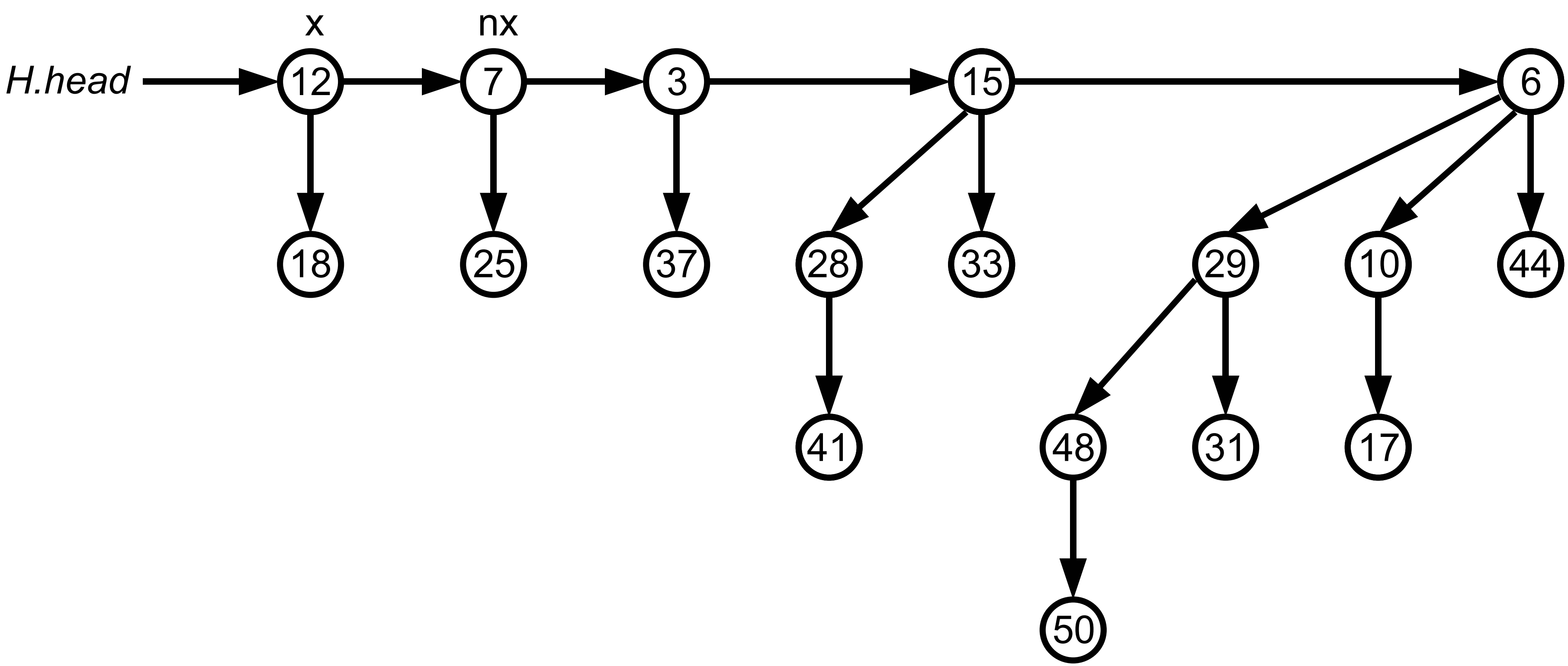
Union

```
Union(H1, H2)
H = MakeHeap()
H.head = HeapMerge(H1, H2)
usuń z pamięci H1 i H2 (pozostawiając jednak listy na które wskazują)
if H.head = NULL then return H
px = NULL
x = H.head
nx = x.sibling
while nx ≠ NULL do
    if x.degree ≠ nx.degree or (nx.sibling ≠ NULL and x.degree = x.sibling.degree) then
        px = x
        x = nx
    else
        if x.key ≤ nx.key then
            x.sibling = nx.sibling
            Link(nx, x)
        else
            if px = NULL then H.head = nx else px.sibling = nx
            Link(x, nx)
            x = nx
        end if
    end if
    nx = x.sibling
end while
return H
```

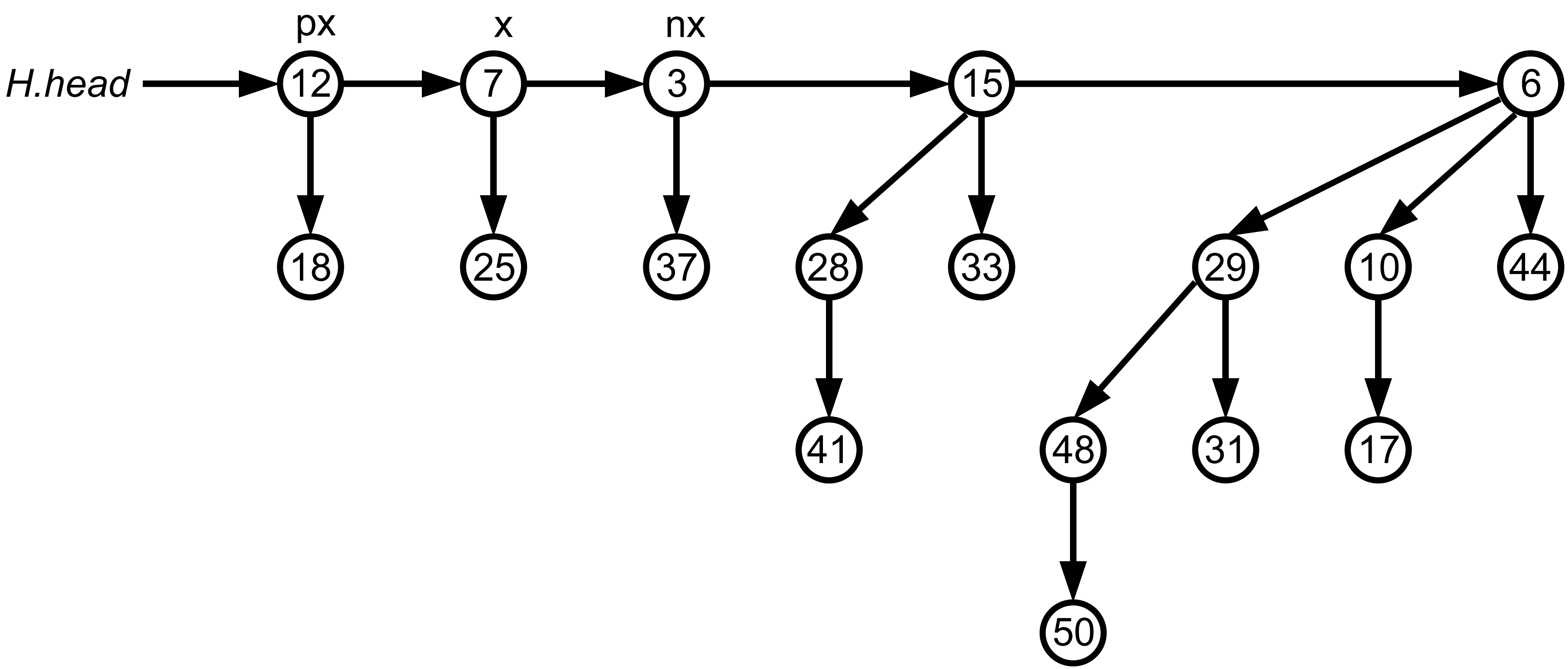
Union



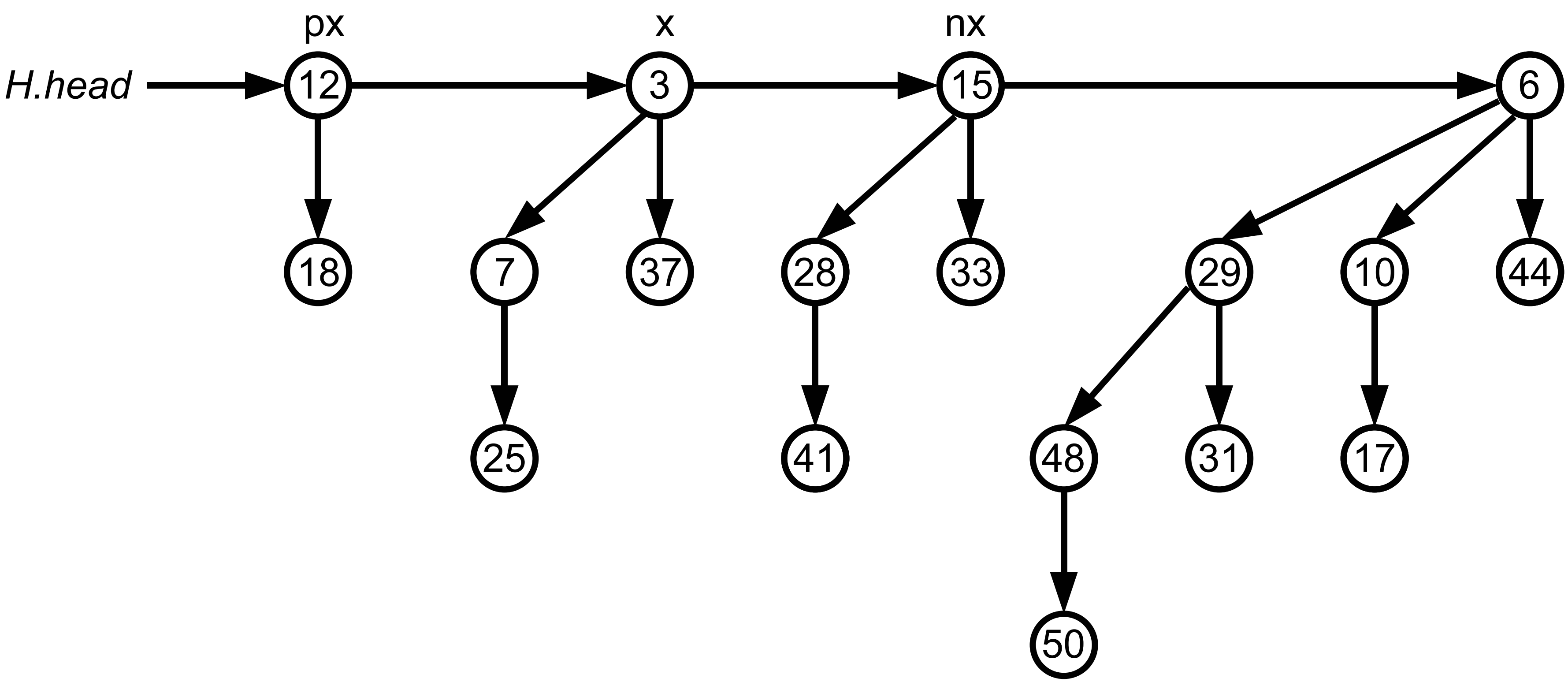
Union



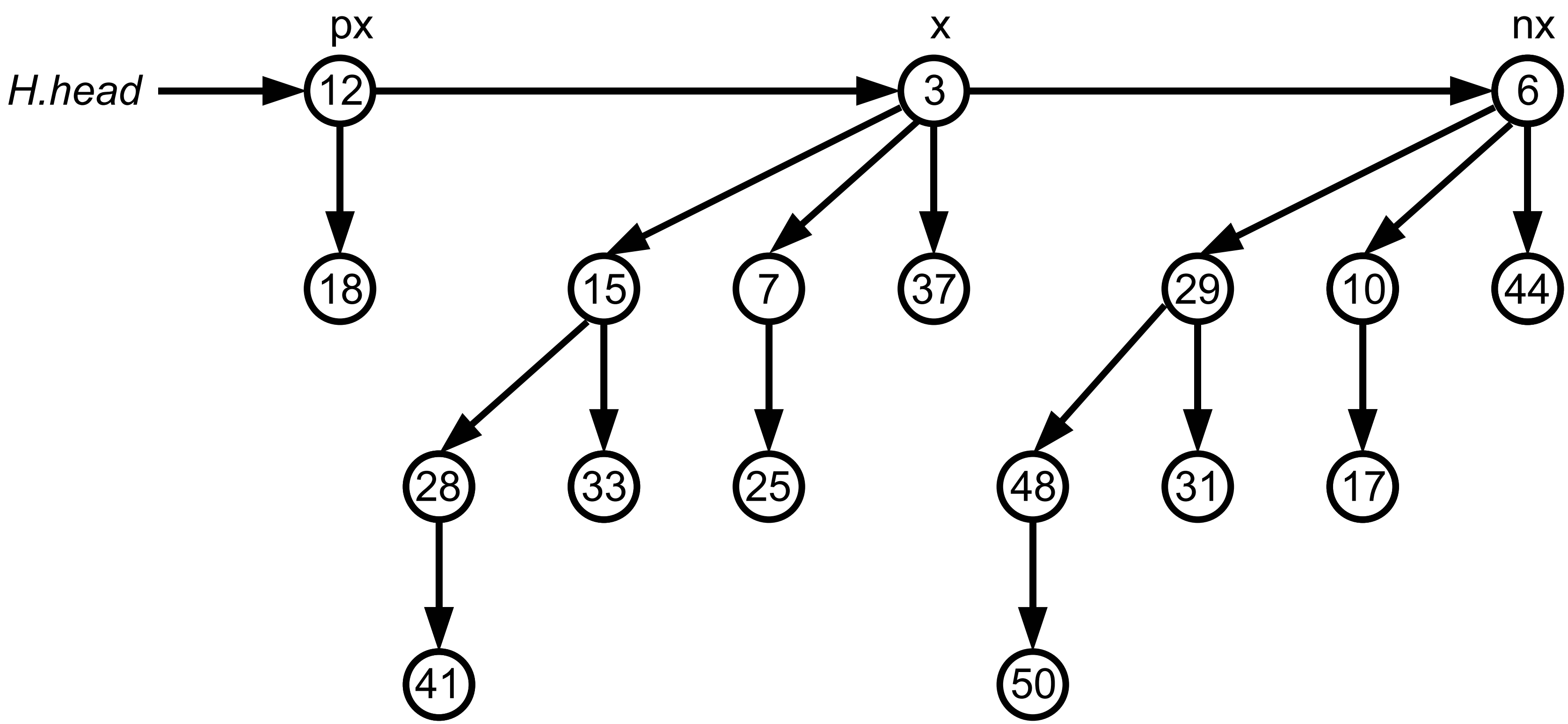
Union



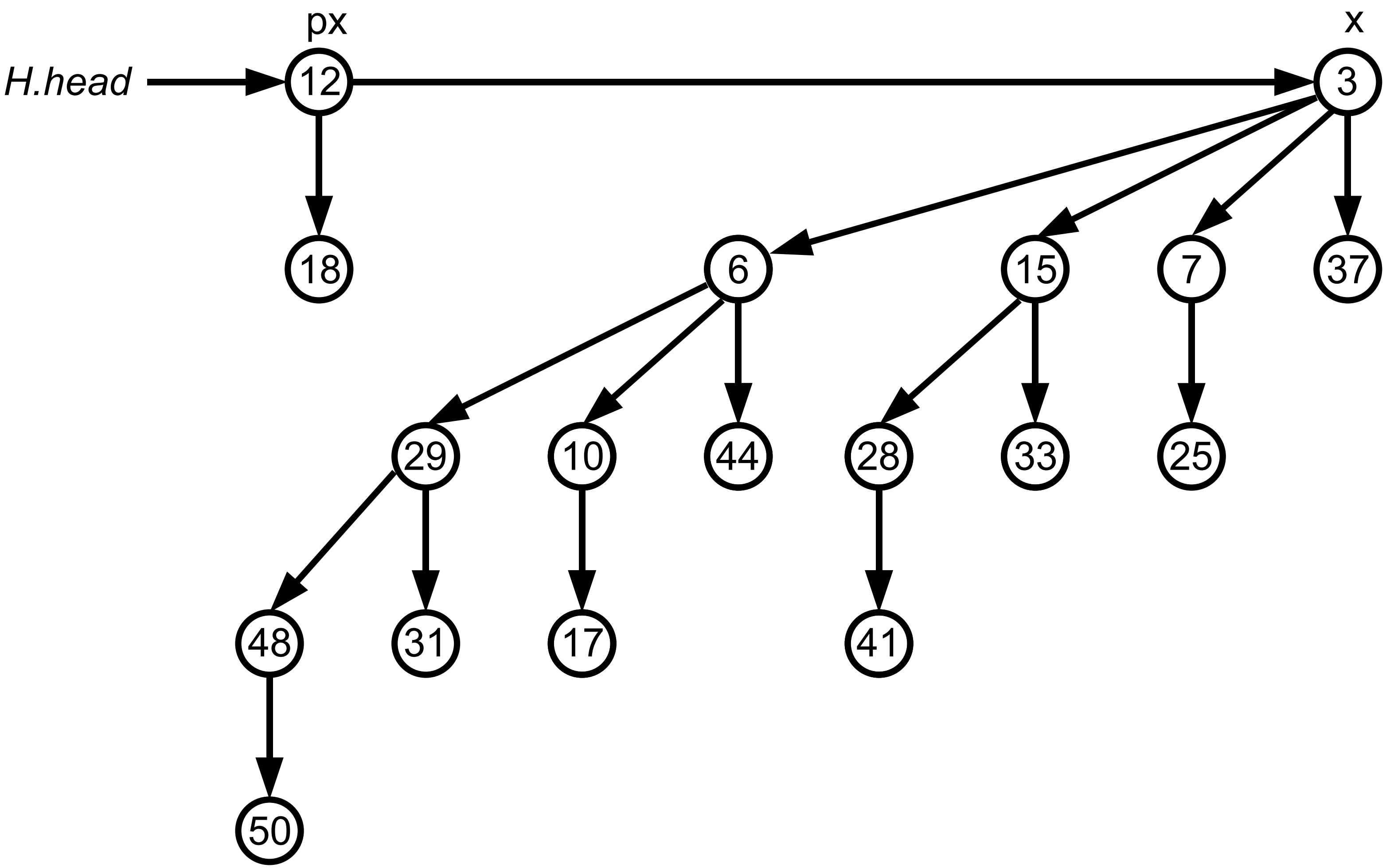
Union



Union



Union



Insert

- wstawienie węzła wykorzystuje operację `Union` — tworzymy kopiec zawierający jeden węzeł a następnie scalamy go z kopcem H
- x jest węzłem do wstawienia ($x.key$ jest już odpowiednio wypełnione)
- wykonamy
  - $O(1)$ — utworzenie kopca, oraz
  - $O(\log n)$ — scalenie kopców

  kroków

Insert

```
Insert(H, x)
  H' = MakeHeap()
  x.parent = NULL
  x.sibling = NULL
  x.child = NULL
  x.degree = 0
  H'.head = x
  H = Union(H, H')
  return H
```

ExtractMin

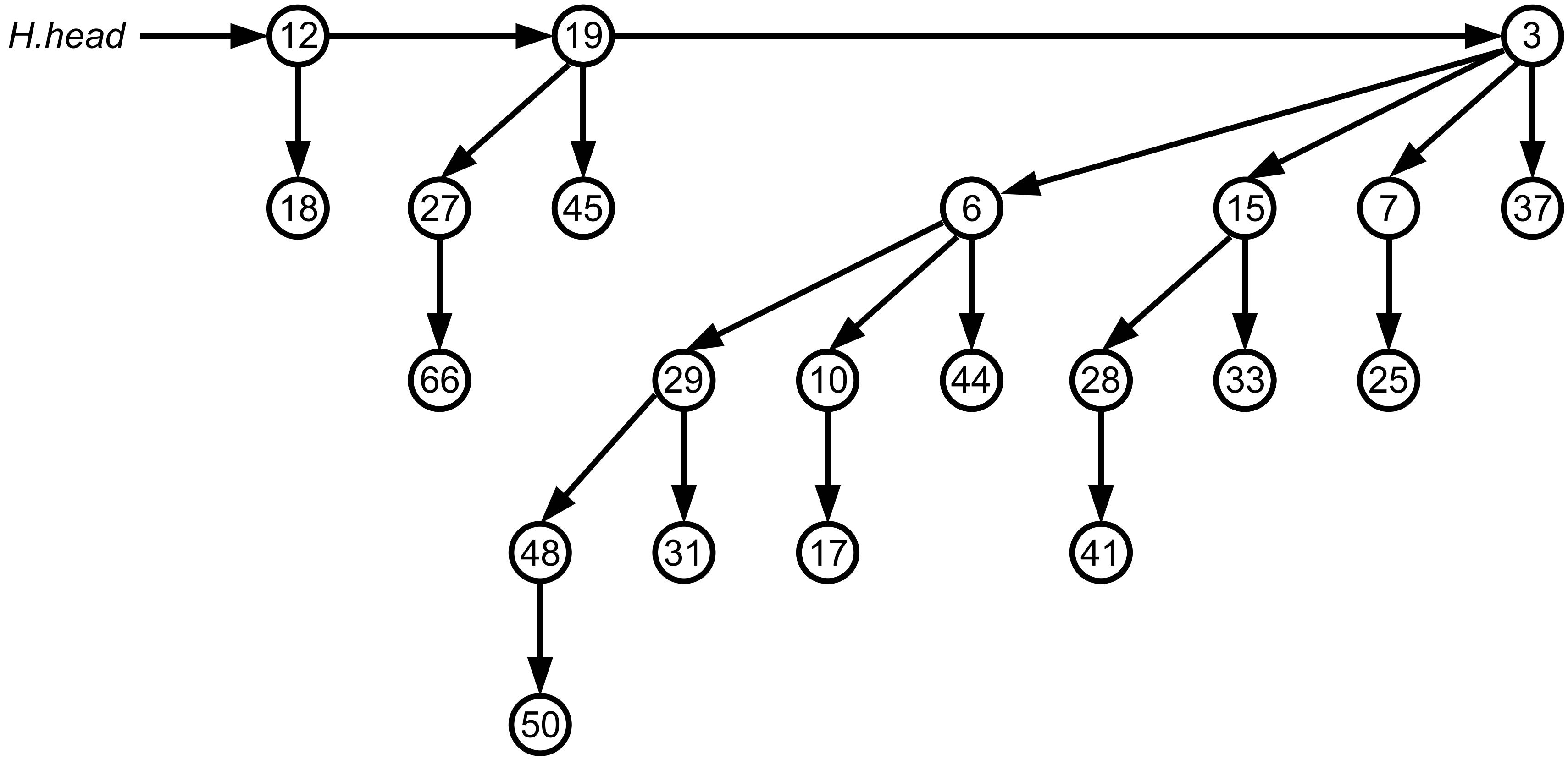
- znajdujemy i usuwamy z listy korzeni w H korzeń o najmniejszym kluczu
- potomkami węzła x są drzewa o stopniach $k, k-1, \ldots, 0$
- odwracamy kolejność potomków węzła x — otrzymujemy kopiec dwumianowy zawierający wszystkie klucze (poza $x.key$) z drzewa o korzeniu w x
- scalamy otrzymany kopiec z kopcem H

`ExtractMin`

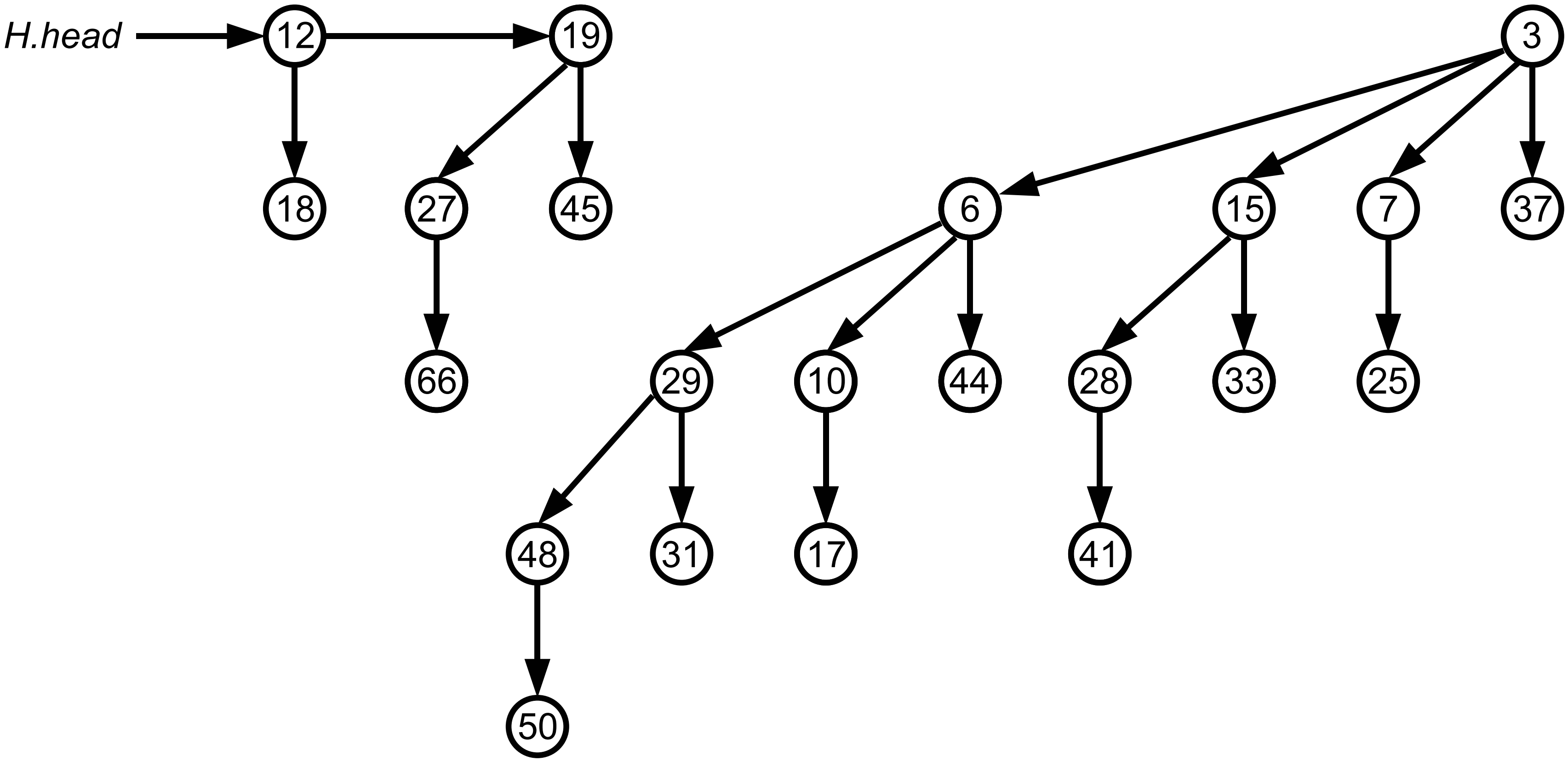
`ExtractMin`(H)

1: znajdź korzeń x z minimalnym kluczem na liście korzeni H
2: usuń x z listy korzeni
3: $H' =$ `MakeHeap`()
4: odwróć kolejność elementów na liście potomków węzła x
5: $H'.head =$ wskaźnik do głowy wynikowej listy
6: $H =$ `Union`(H, H')
7: **return** H

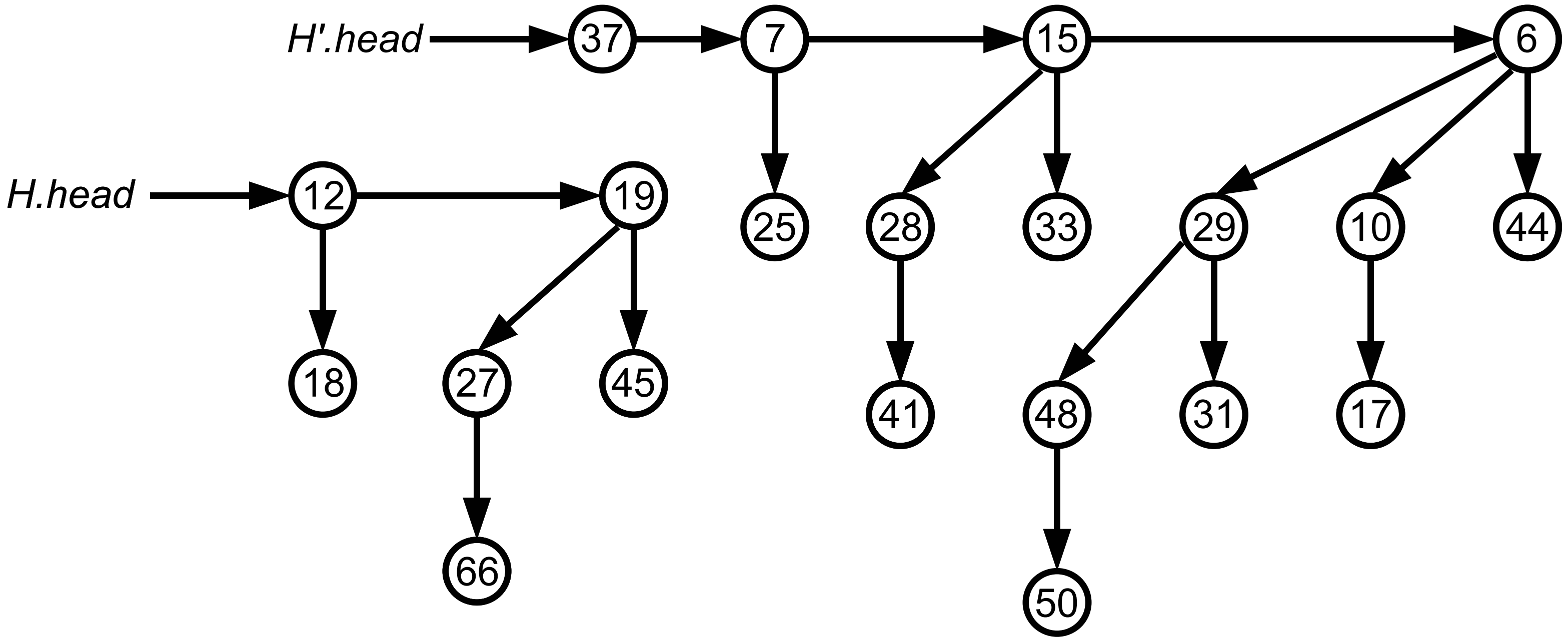
ExtractMin



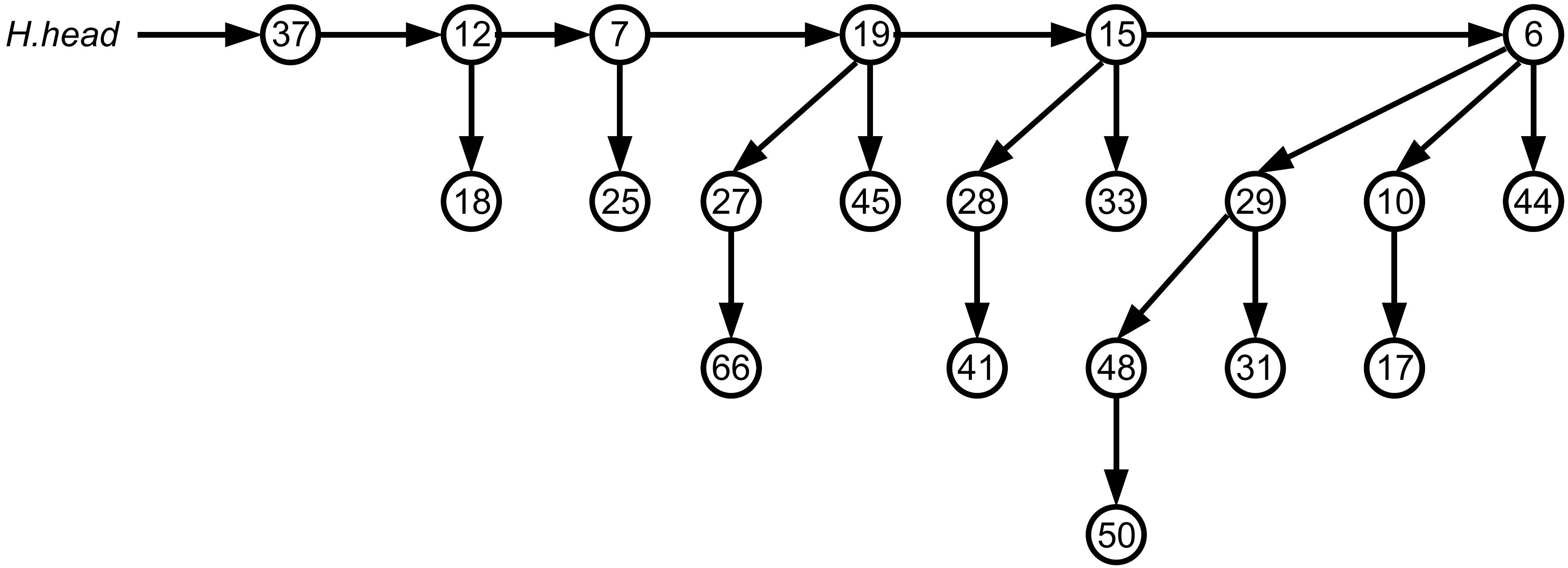
ExtractMin



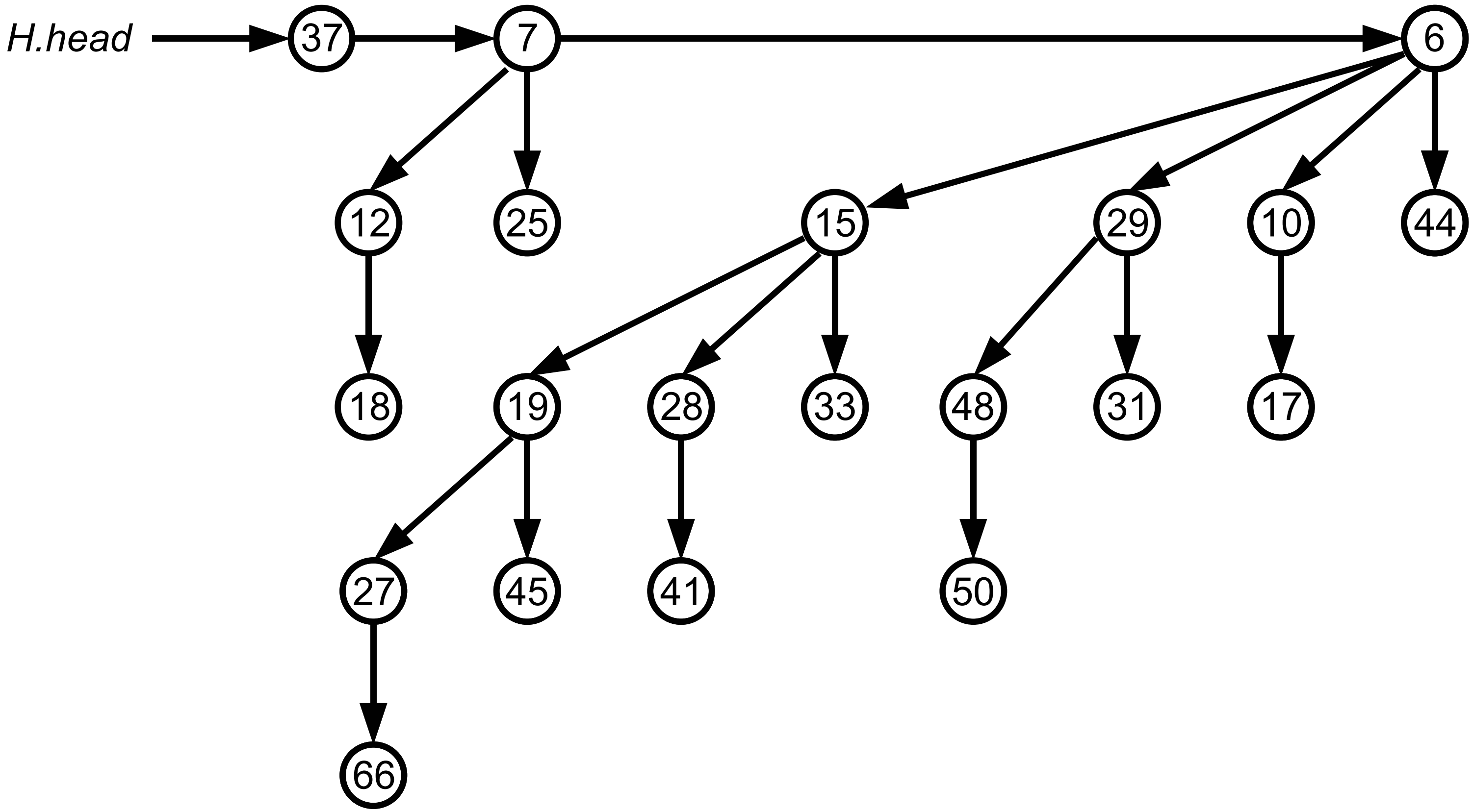
ExtractMin


H'.head
37
7
15
6
H.head
12
19
25
28
33
29
10
44
18
27
45
41
48
31
17
66
50

ExtractMin



ExtractMin



DecreaseKey

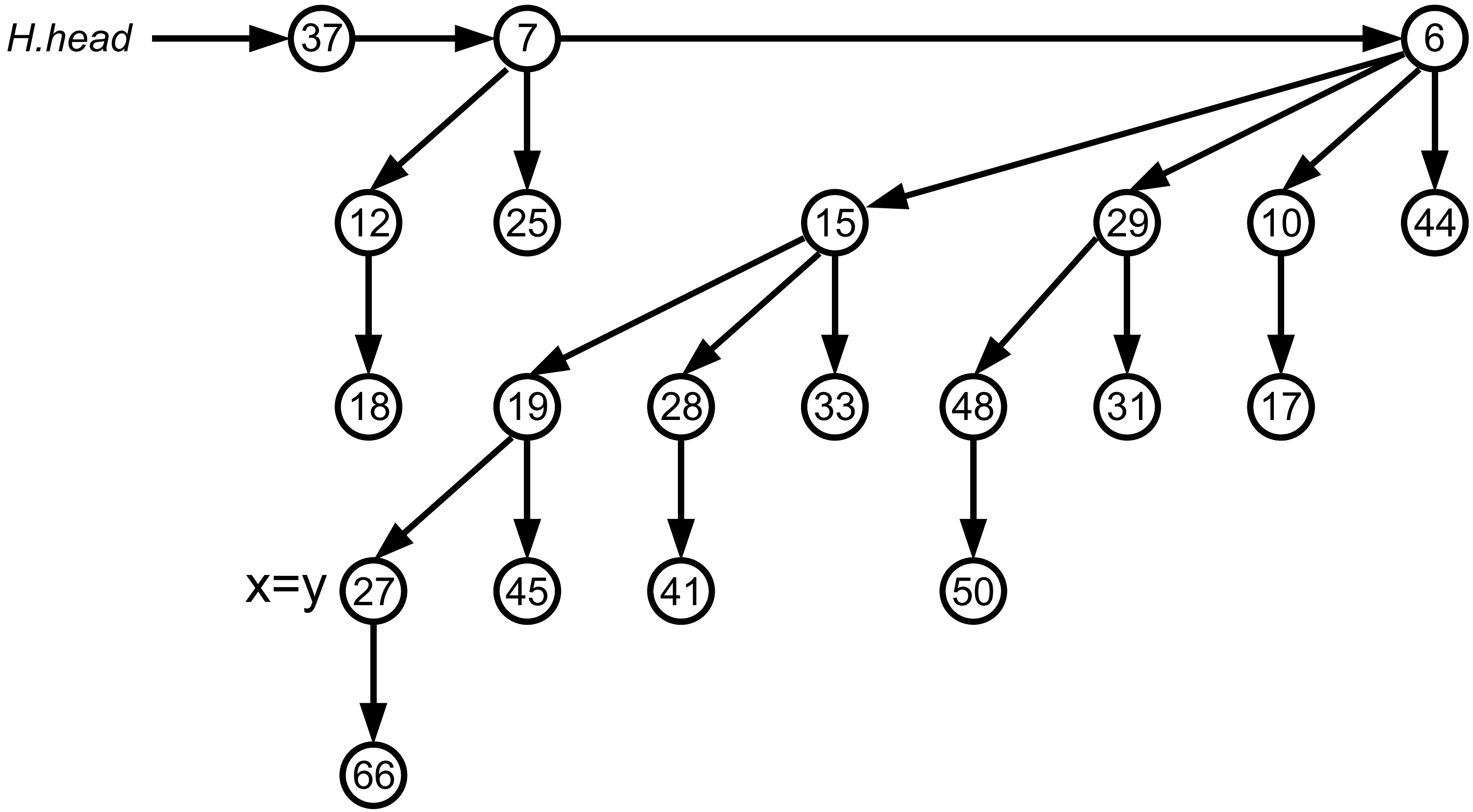
- operacja ta wymaga wskaźnika na węzeł x zawierający klucz, którego wartość należy zmniejszyć
- po zmianie wartości klucza należy przywrócić drzewu zawierającemu x własność kopca (analogicznie jak `Heapify` w zwykłym kopcu)

DecreaseKey

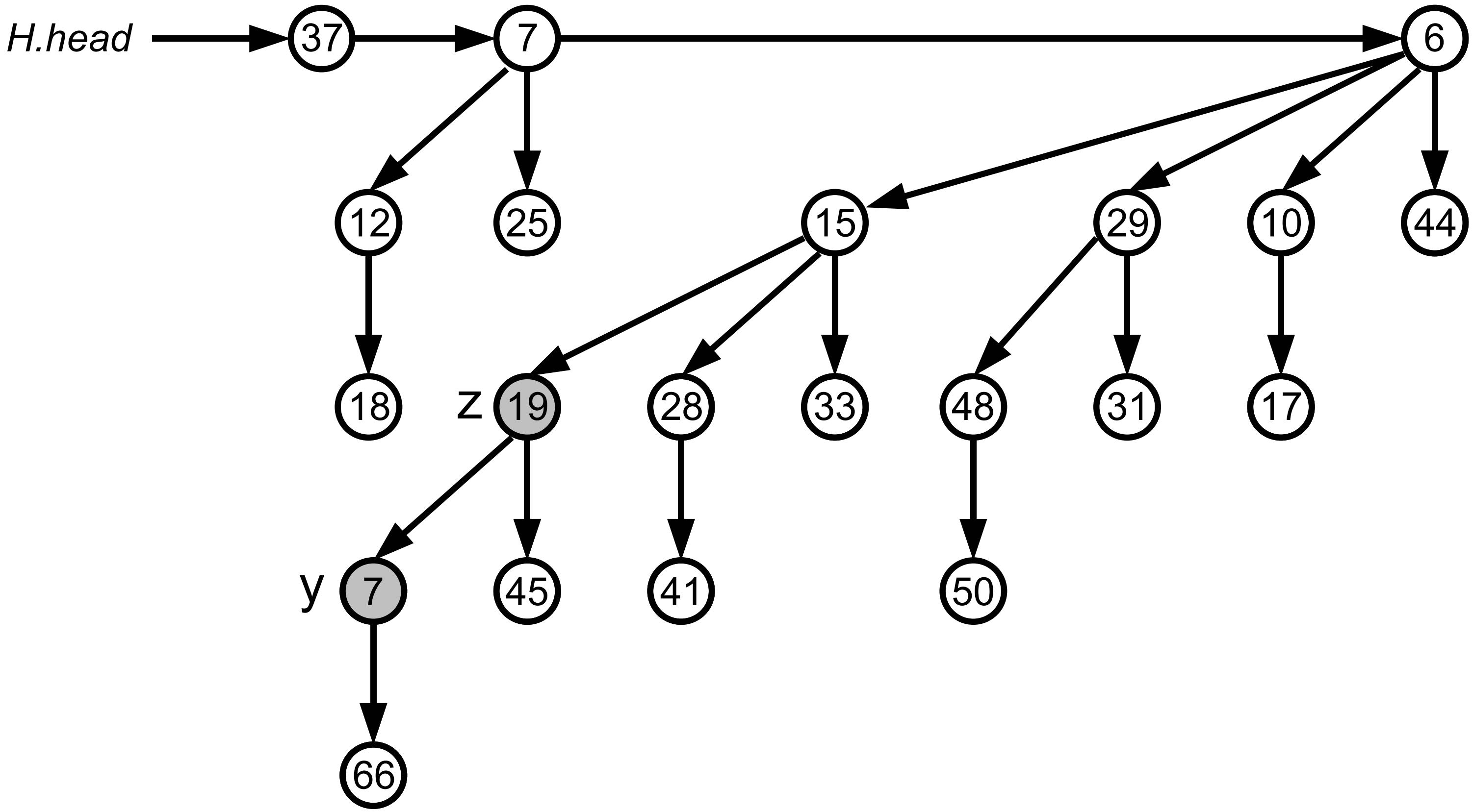
`DecreaseKey`(H, x, k)

if $x.key \leq k$ **then error** “zwiększenie wartości klucza”
$x.key = k$
$y = x$
$z = y.parent$
while $z \neq NULL$ **and** $y.key < z.key$ **do**
    zamień $y.key$ z $z.key$
    zamień wartości dodatkowych danych z węzłów y i z
    $y = z$
    $z = y.parent$
end while

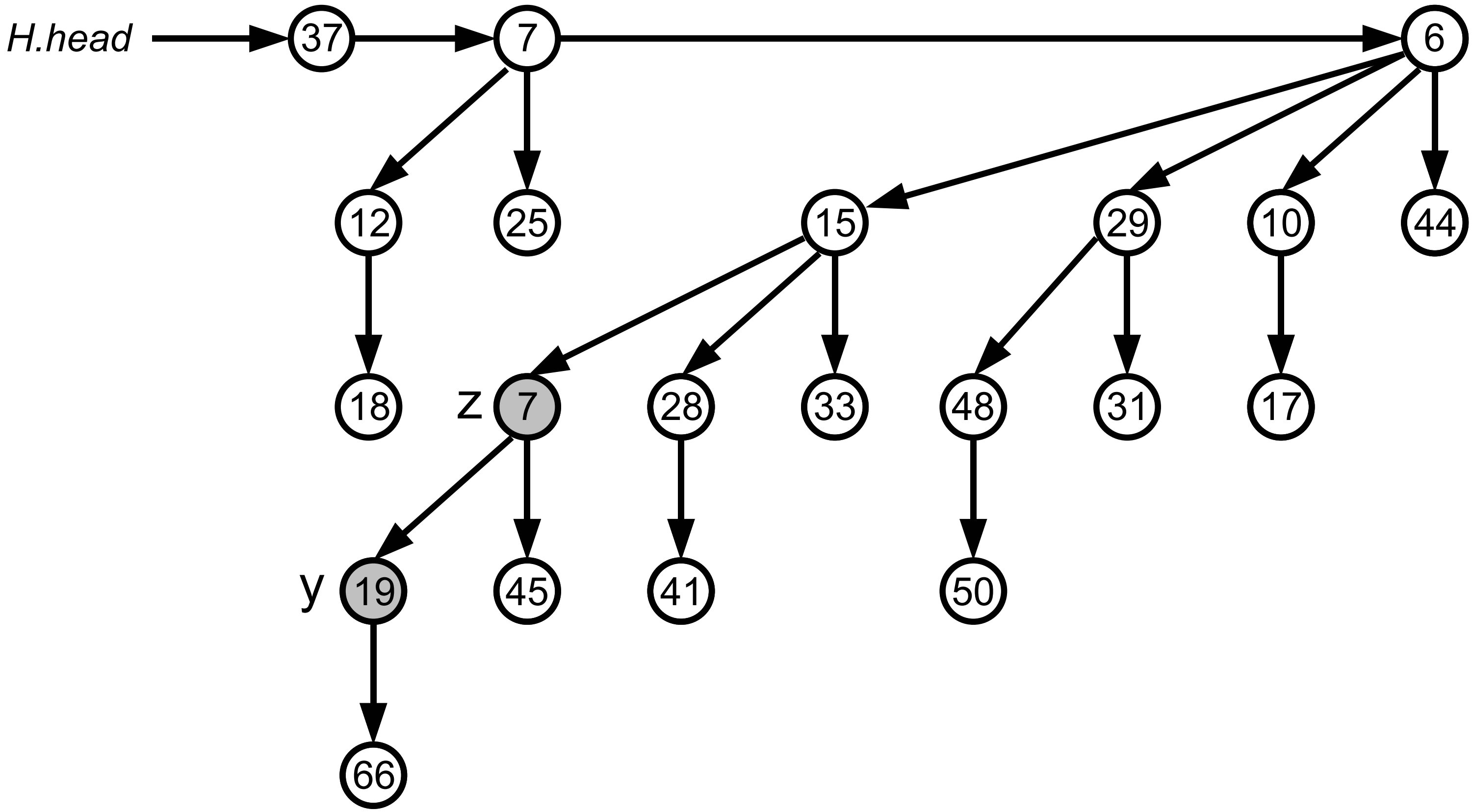
DecreaseKey



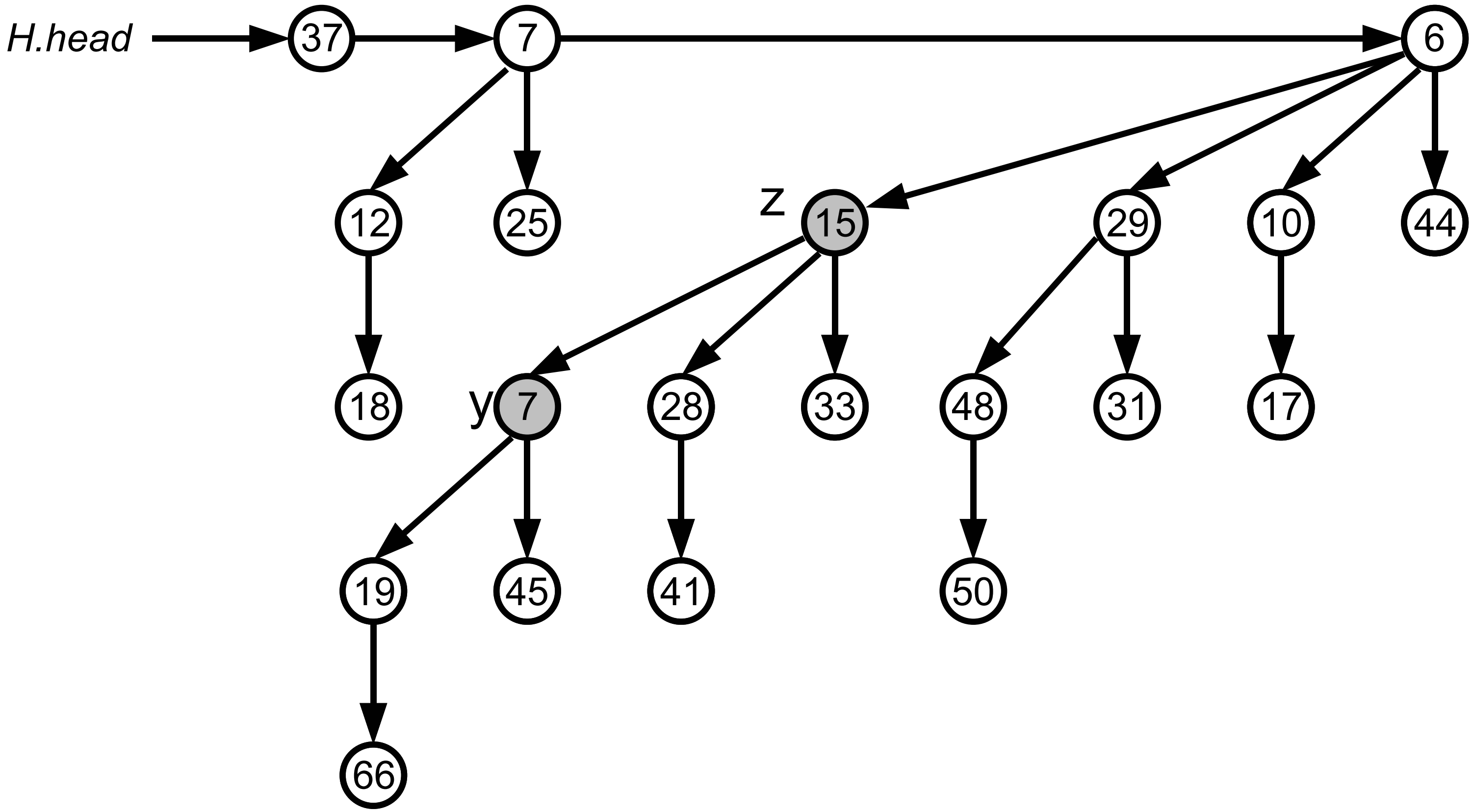
DecreaseKey



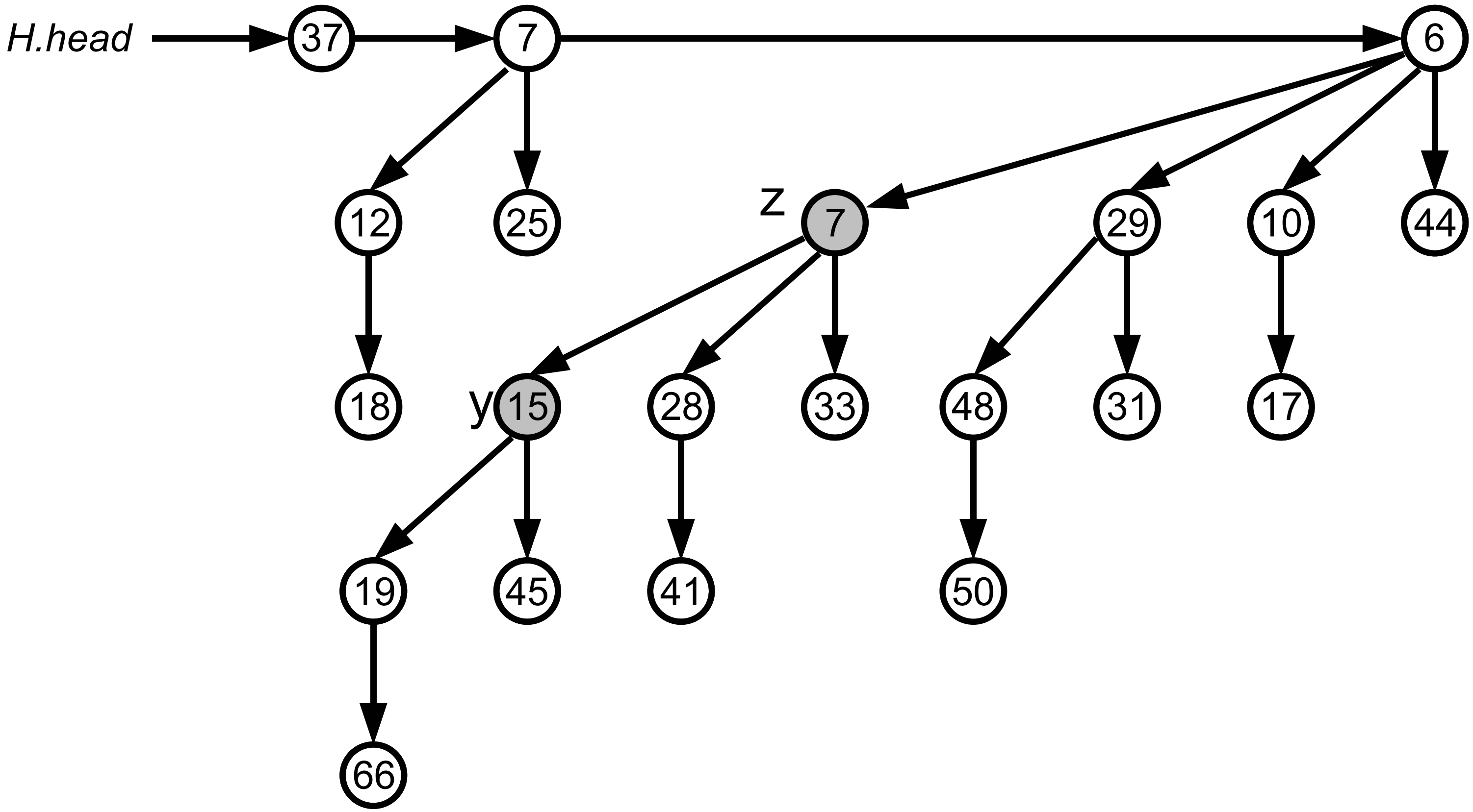
DecreaseKey



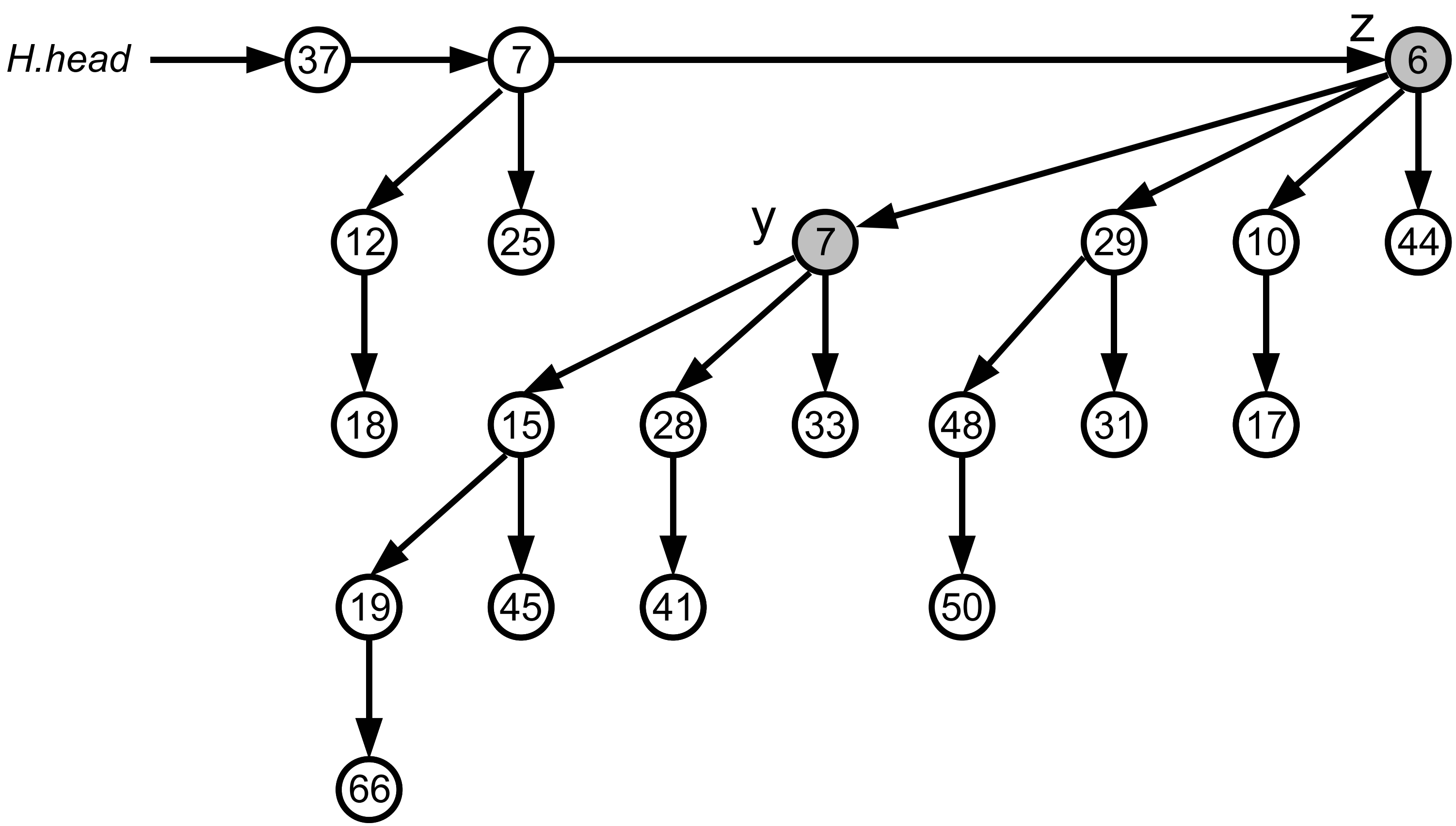
DecreaseKey



DecreaseKey



DecreaseKey



Delete

- również ta operacja wymaga wskaźnika na węzeł x zawierający klucz, który należy usunąć
- zmniejszamy wartość usuwanego klucza do $-\infty$
- usuwamy z kopca najmniejszy klucz (`ExtractMin`)

`Delete`(H, x)

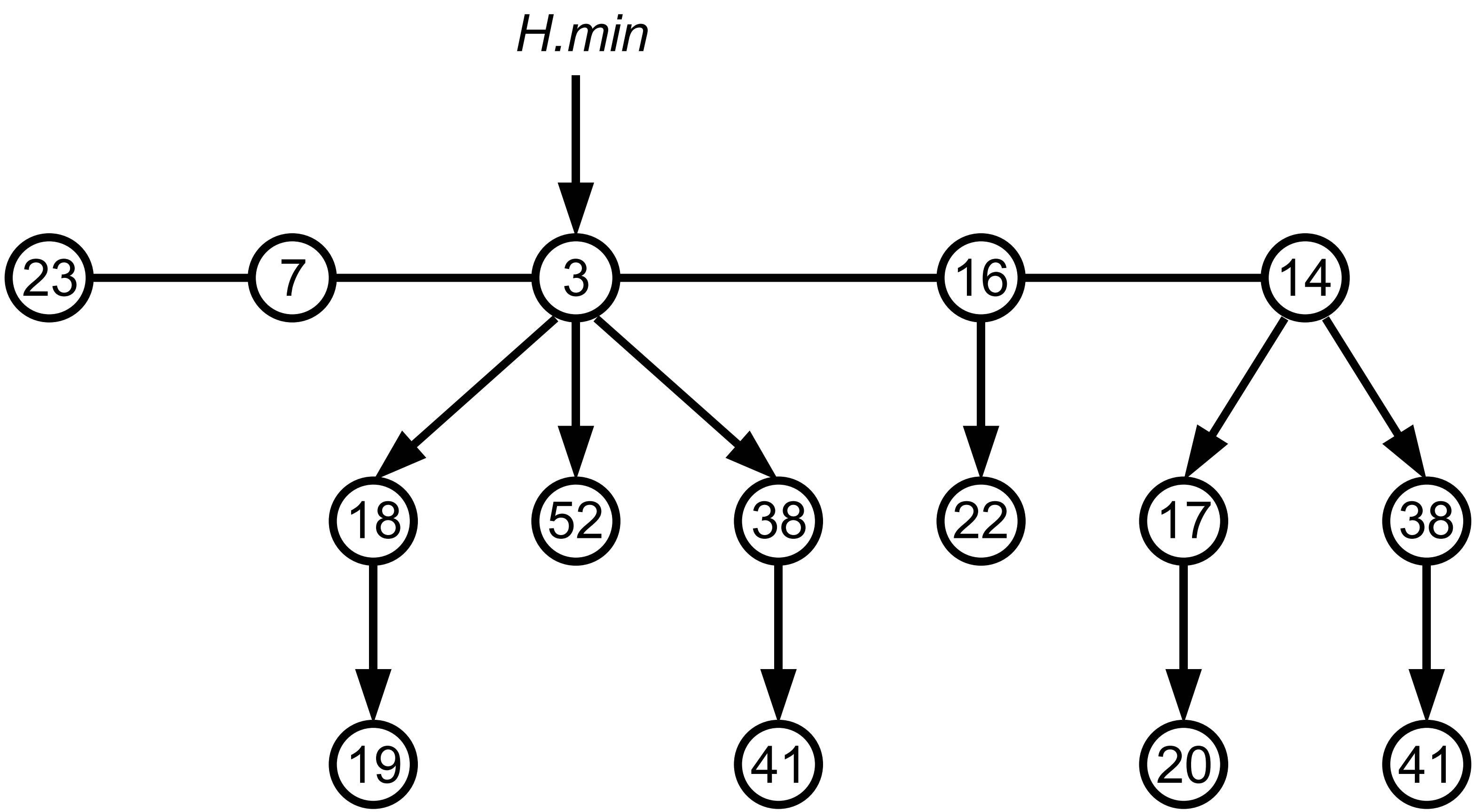
1: `DecreaseKey`(H, x, $-\infty$)
2: `ExtractMin`(H)

Kopce Fibonacciego

Kopiec Fibonacciego

- jest zbiorem drzew, z których każde ma własność kopca
- drzewa nie muszą być drzewami dwumianowymi
- jeżeli ograniczymy się do operacji `MakeHeap`, `Insert`, `Minimum`, `ExtractMin`, `Union` to kopiec Fibonacciego będzie zbiorem “nieuporządkowanych” drzew binarnych (węzły potomne nie muszą być uporządkowane od najwięszego do najmniejszego poddrzewa)
- drzewa są ukorzenione, ale nie są uporządkowane
- każdy węzeł zawiera wskaźnik do któregokolwiek ze swoich potomków
- potomkowie powiązani są cykliczną listą dwukierunkową

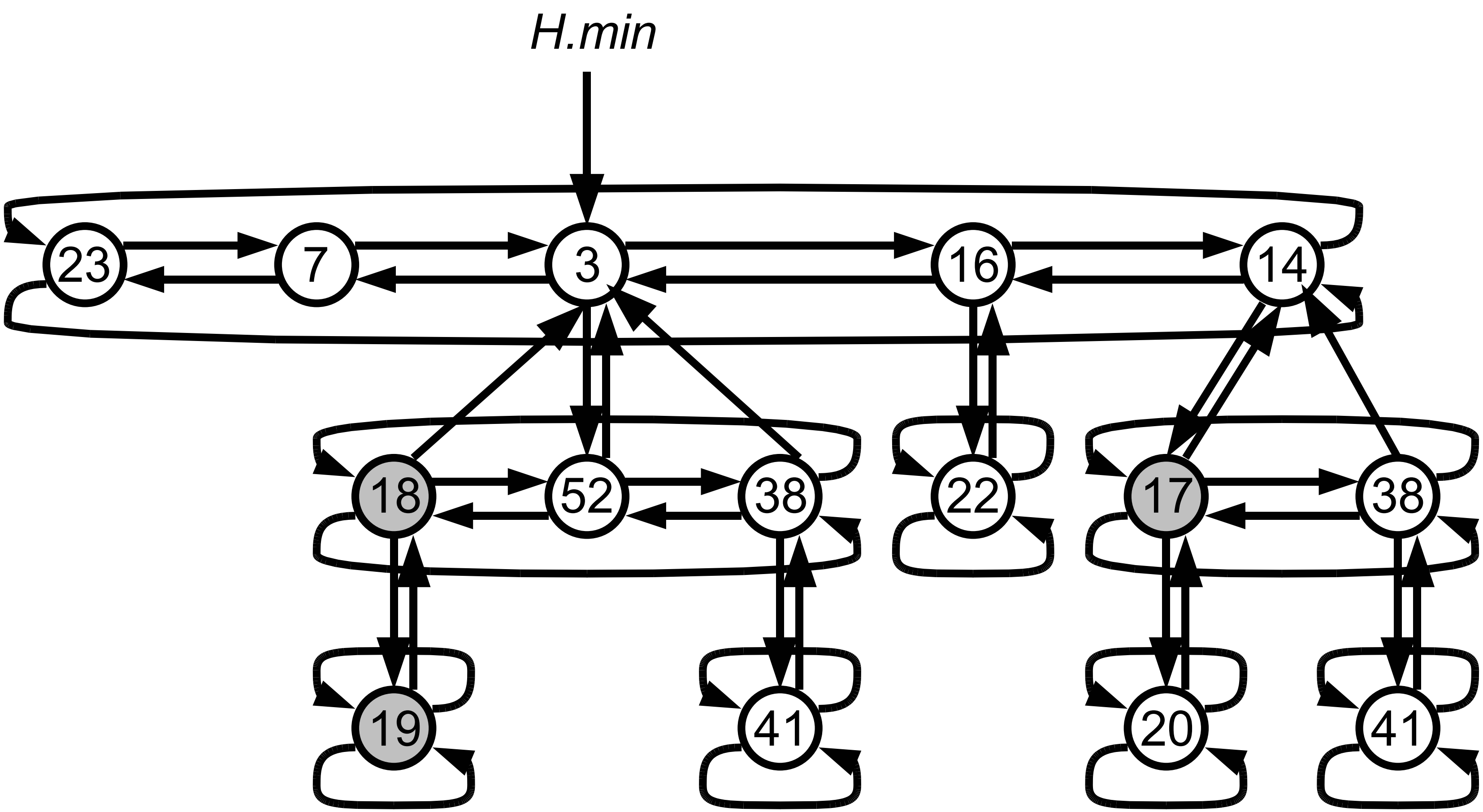
Kopiec Fibonacciego



Kopiec Fibonacciego

- każdy węzeł zawiera:
  - key — klucz
  - $parent$ — wskaźnik na rodzica ($NULL$ w korzeniu)
  - $child$ — wskaźnik na któregokolwiek potomka ($NULL$ w liściach)
  - $left$ — wskaźnik na następnego brata (lub wskaźnik na siebie, gdy brata brak)
  - $right$ — wskaźnik na poprzedniego brata (lub wskaźnik na siebie, gdy brata brak)
  - $degree$ — stopień wierzchołka (liczba potomków)
  - $mark$ — znacznik, mówiący, czy dany węzeł stracił potomka od ostatniej chwili, gdy sam stał się potomkiem innego węzła
  - ewentualnie dodatkowe dane powiązane z kluczem

Kopiec Fibonacciego



Kopiec Fibonacciego

- dostęp do kopca zapewnia wskaźnik min — jest to wskaźnik do korzenia drzewa zawierającego najmniejszy klucz (węzeł minimalny kopca)
- jeżeli kopiec H jest pusty, $H.min = NULL$
- korzenie drzew połączone są przy pomocy wskaźników $left$ i $right$ w dwukierunkową listę cykliczną
- kolejność drzew na liście może być dowolna
- w strukturze H pamiętamy także liczbę węzłów w kopcu (pole $H.n$)

MakeHeap

- utworzenie nowego kopca wymaga jedynie utworzenia nowego obiektu i wypełnienia go odpowiednimi wartościami:

`MakeHeap`(H)

1: $H =$ **new** FibonacciHeap
2: $H.min = NULL$
3: $H.n = 0$

Minimum

- minimalny klucz zawsze jest wskazywany przez pole $H.min$

Minimum(H)

1: **return** $H.min$

Insert

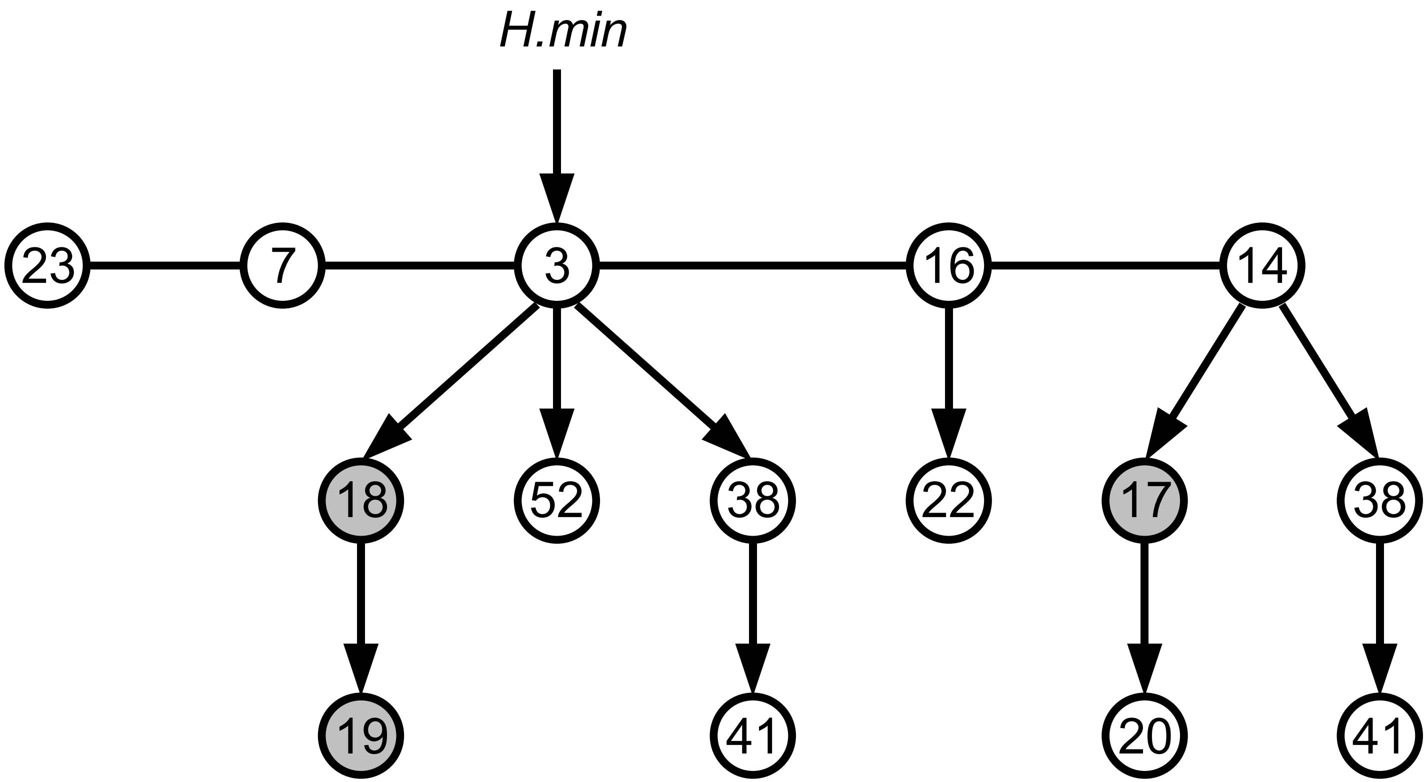
- x jest węzłem do wstawienia ($x.key$ jest już odpowiednio wypełnione)
- dodajemy x do listy korzeni kopca H
- jeżeli trzeba, poprawiamy wskaźnik $H.min$
- zwiększamy licznik węzłów w H
- nie próbujemy sklejać drzew — k kolejnych wstawień wstawi do listy korzeni k jednoelementowych drzew

Insert

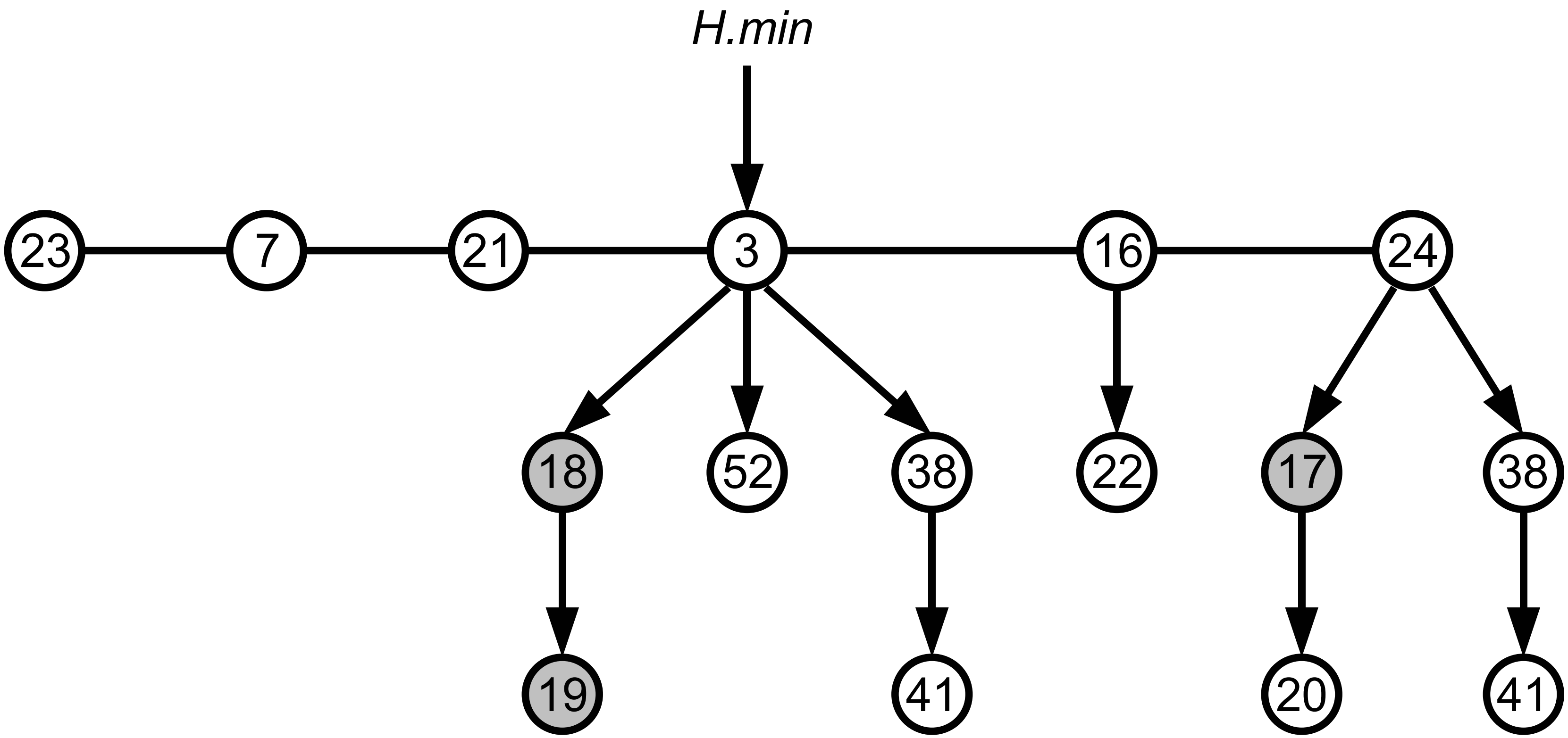
`Insert`(H, x)

$x.degree = 0$
$x.parent = NULL$
$x.left = x$
$x.right = x$
$x.child = NULL$
$x.mark = FALSE$
połącz jednoelementową listę x z listą korzeni H
if $H.min = NULL$ **or** $x.key < H.min.key$ **then**
    $H.min = x$
end if
$H.n = H.n + 1$

Insert



Insert



Union

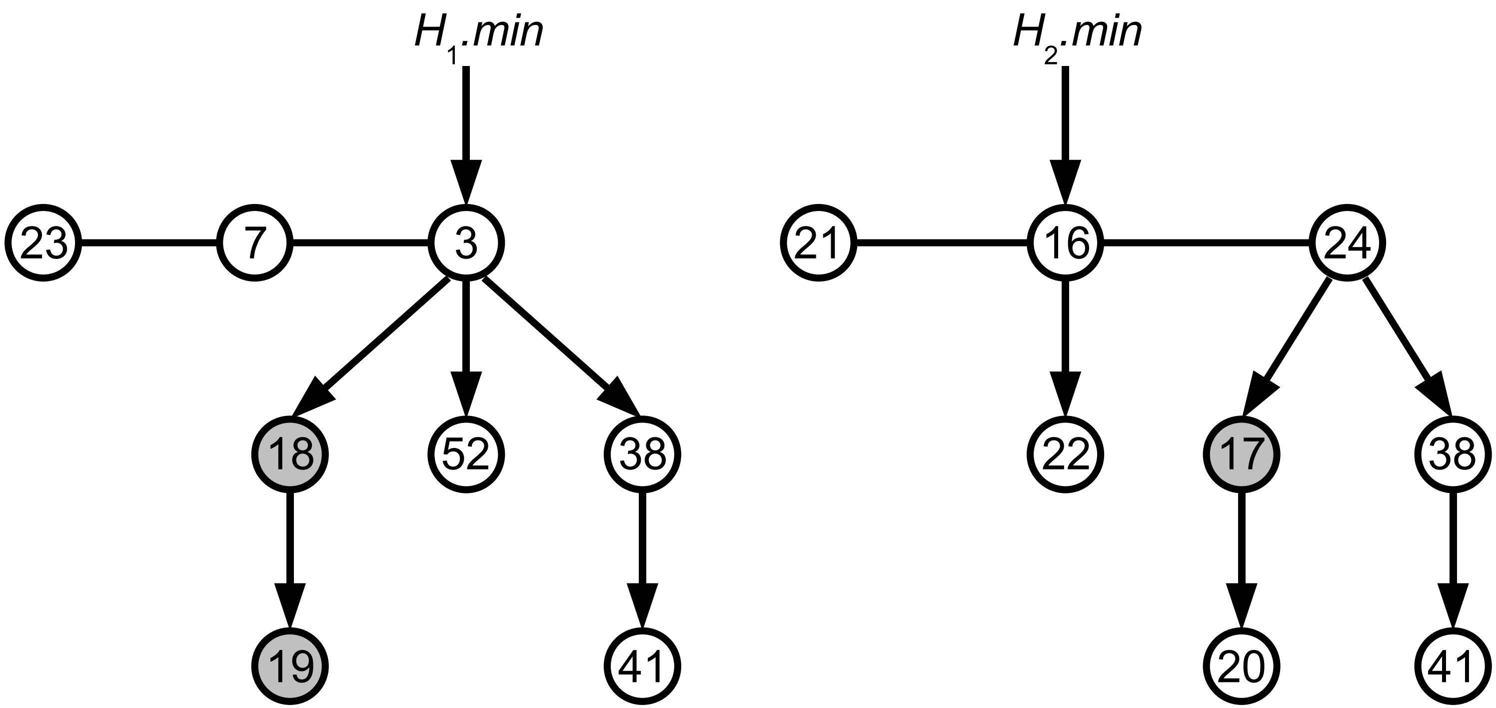
- operacja scalenia kopców jest podobna do wstawiania pojedynczego klucza (tyle, że wstawiamy listę dłuższą niż jeden element)
- ponownie sklejanie drzew odkładamy na później

Union

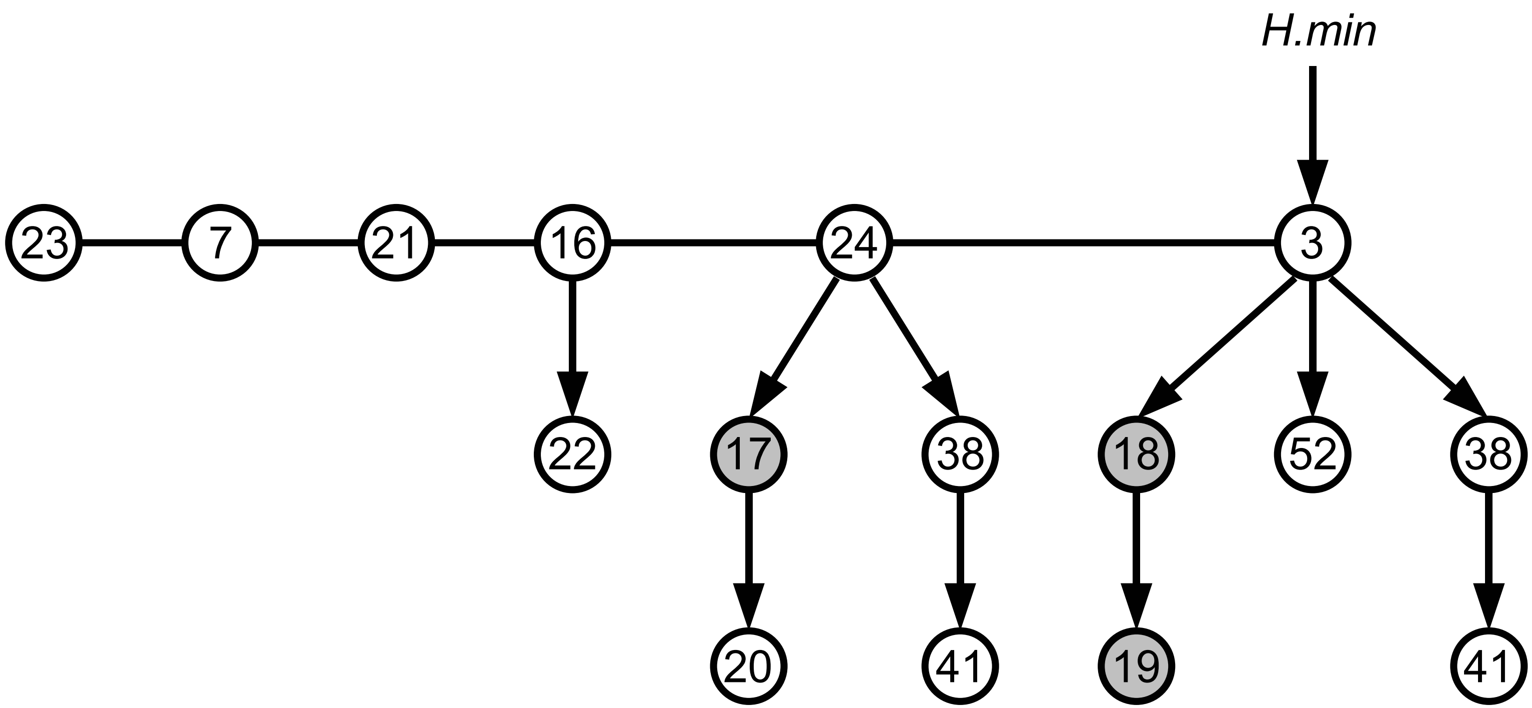
`Union`(H_1, H_2)

$H =$ `MakeHeap`()
$H.min = H_1.min$
sklej listę korzeni H_2 z listą korzeni H_1
if $H_1.min = NULL$ **or** ($H_2.min \neq NULL$ **and** $H_2.min < H_1.min$) **then**
    $H.min = H_2.min$
end if
$H.n = H_1.n + H_2.n$
zwolnij pamięć przydzieloną obiektom H_1 i H_2 (ale nie zwalniaj ich list korzeni)
return H

Union



Union



ExtractMin

- w tej operacji wykonywana jest praca okładana we wstawianiu i scalaniu
- korzystamy z pomocniczej procedury `Consolidate`, która skraca listę korzeni
- `Consolidate` z kolei korzysta z pomocniczej procedury `Link`, łączącej dwa drzewa o takim samym stopniu

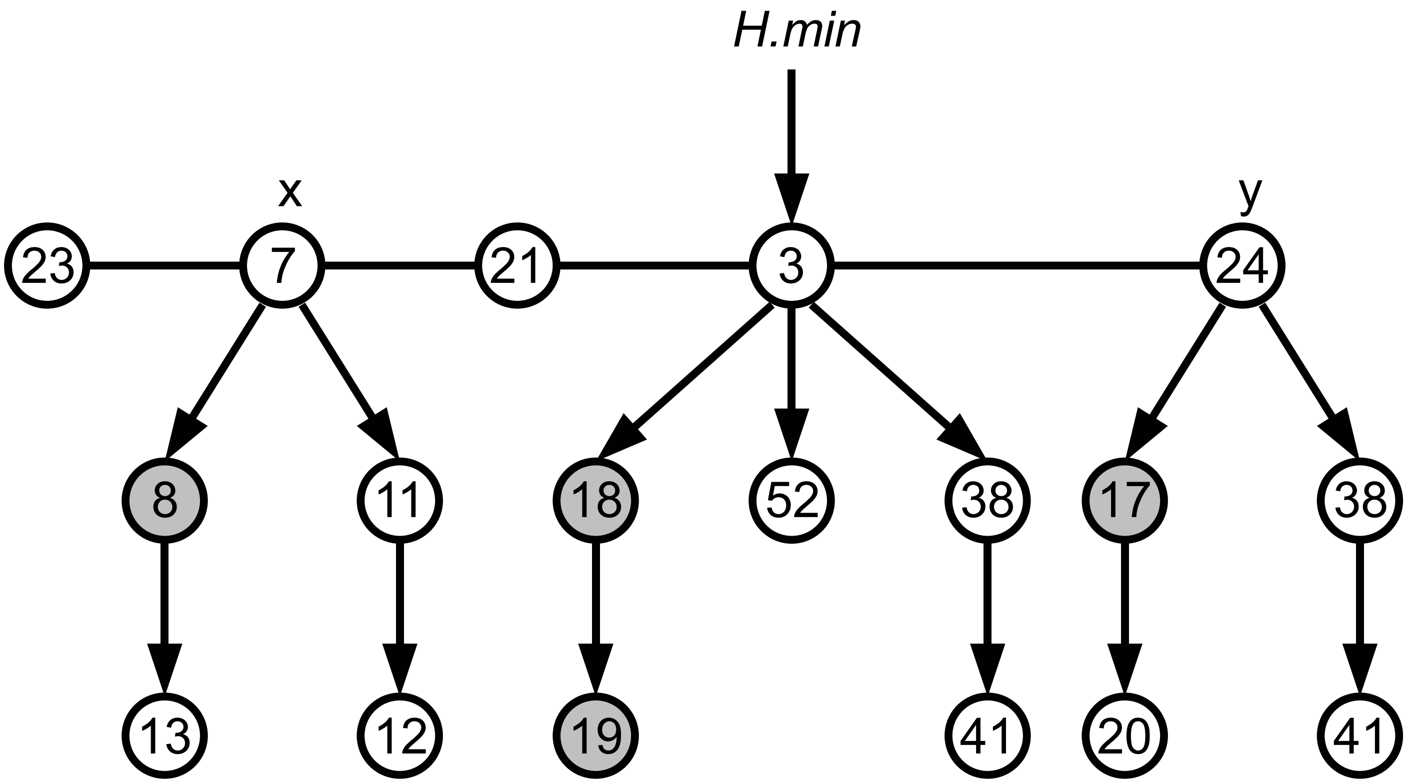
`Link`

- pomocnicza procedura — usunięcie y z listy korzeni i dołączenie go do potomków x (y przestaje być zaznaczony)

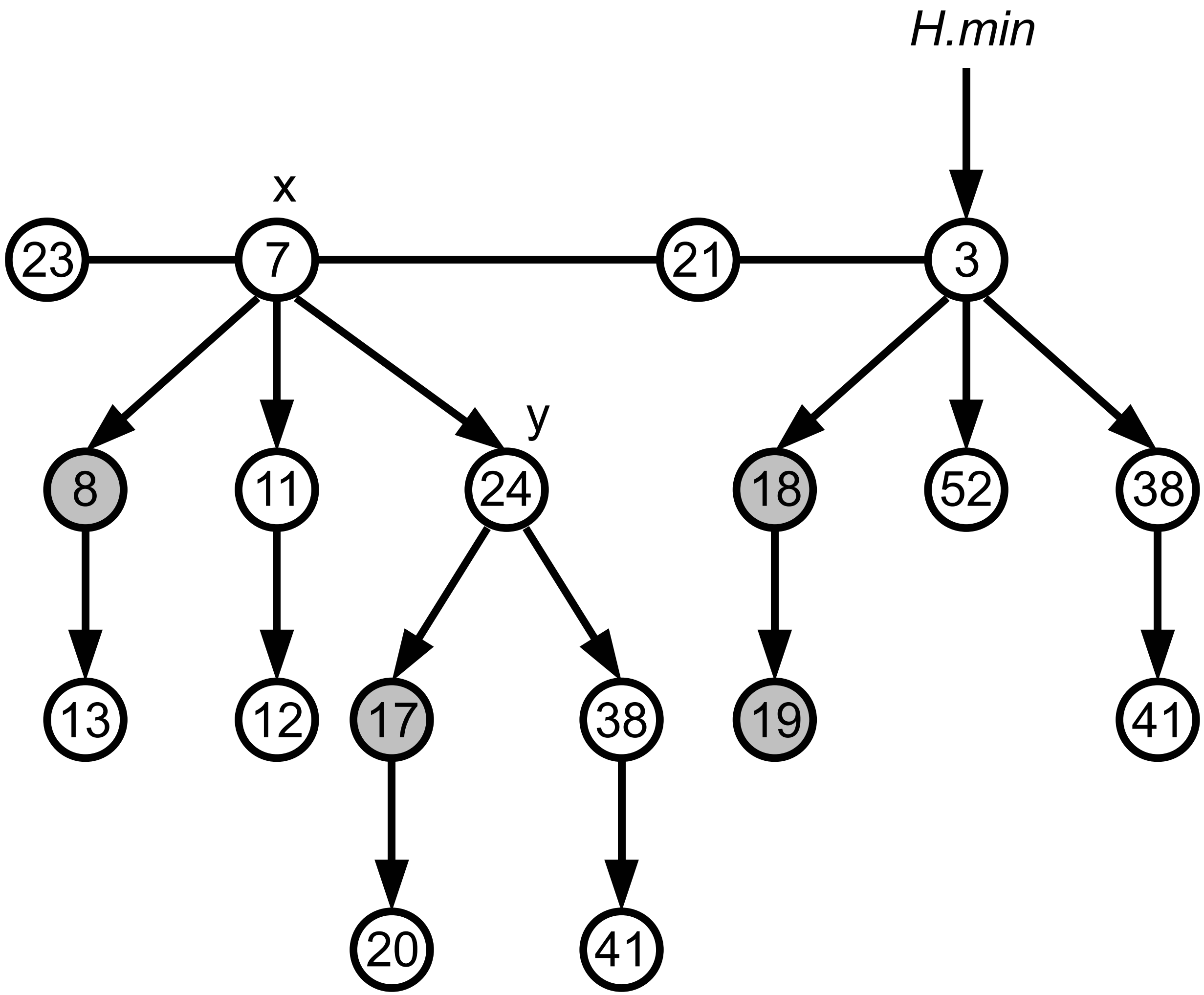
`Link`(H, y, x)

1: usuń y z listy korzeni H
2: wstaw y do listy potomków x
3: $x.degree = x.degree + 1$
4: $y.mark = FALSE$

Link



Link



`Consolidate`

- pomocnicza procedura — skracanie listy korzeni
- dopóki każdy korzeń na liście nie ma innego stopnia:
  - znadź na liście korzeni dwa węzły x i y tego samego stopnia, przy czym $x.key \leq y.key$
  - dołącz y do x, wykorzystując procedurę `Link`
- wykorzystujemy pomocniczą tablicę A — jeżeli $A[i] = y$, to y jest korzeniem takim, że $y.degree = i$ ($i = 0, \ldots, \lfloor \log n \rfloor + 1$)

Consolidate

- tablica A przechowuje wskaźniki do “znanych” drzew o danych stopniach (co najwyżej po jednym na dany stopnień)
- pobierając kolejne drzewo z listy korzeni sprawdzamy, czy znamy już drzewo o takim stopniu
- jeżeli tak, to łączymy te drzewa — otrzymujemy drzewo o stopniu o jeden większym
- sprawdzamy, czy otrzymane drzewo znów możemy połączyć ze znanym drzewem (jeżeli znamy drzewo o takim samym stopniu), itd.

Consolidate

`Consolidate`(H)

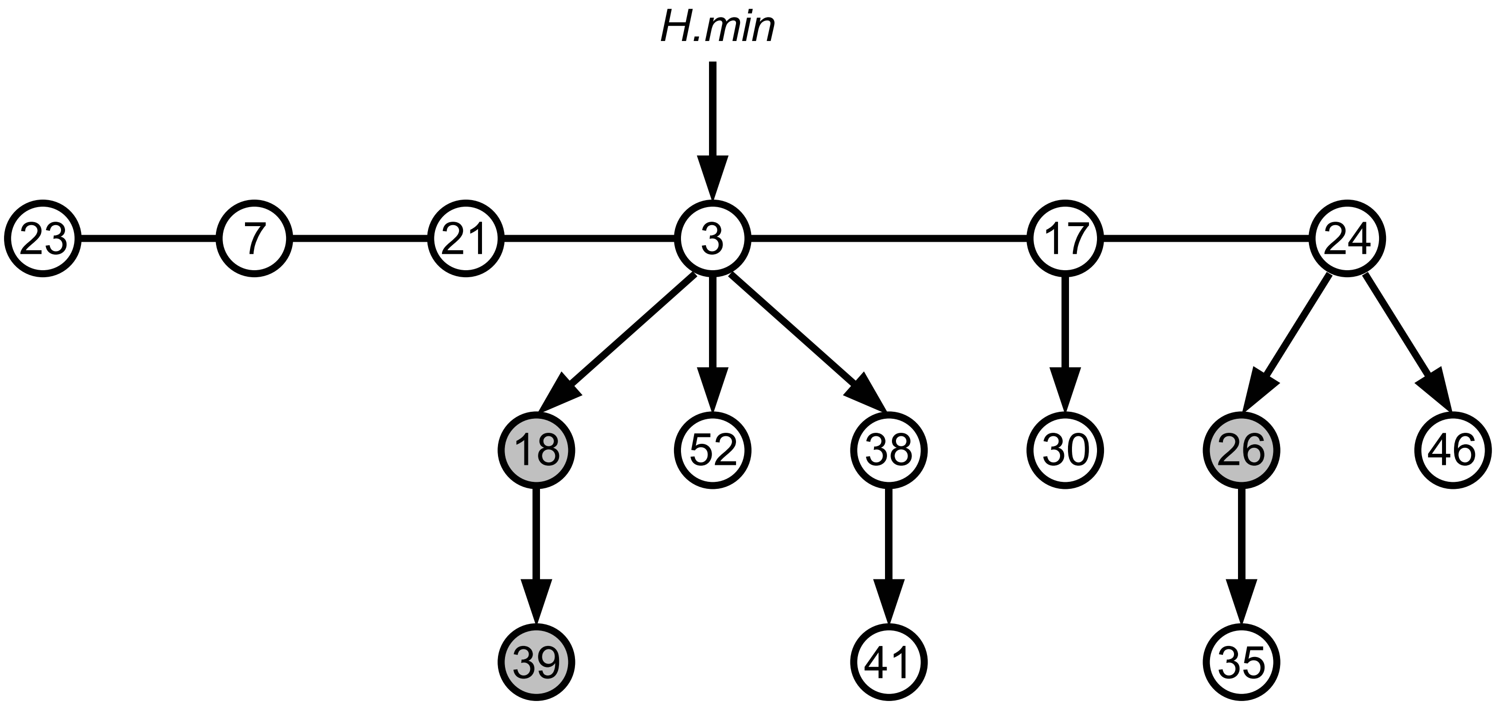
for $i = 0, \ldots, \lfloor \log H.n \rfloor + 1$ **do** $A[i] = NULL$
for każdy węzeł w na liście korzeni H **do**
    $x = w$
    $d = x.degree$
    while $A[d] \neq NULL$ **do**
        $y = A[d]$
        if $x.key > y.key$ **then** zamień x z y
        `Link`(H, y, x)
        $A[d] = NULL$
        $d = d + 1$
    end while
    $A[d] = x$
end for
$H.min = NULL$
for $i = 0, \ldots, \lfloor \log n \rfloor + 1$ **do**
    if $A[i] \neq NULL$ **then**
        dodaj $A[i]$ do listy korzeni H
        if $H.min = NULL$ **or** $A[i].key < H.min.key$ **then** $H.min = A[i]$
    end if
end for

ExtractMin

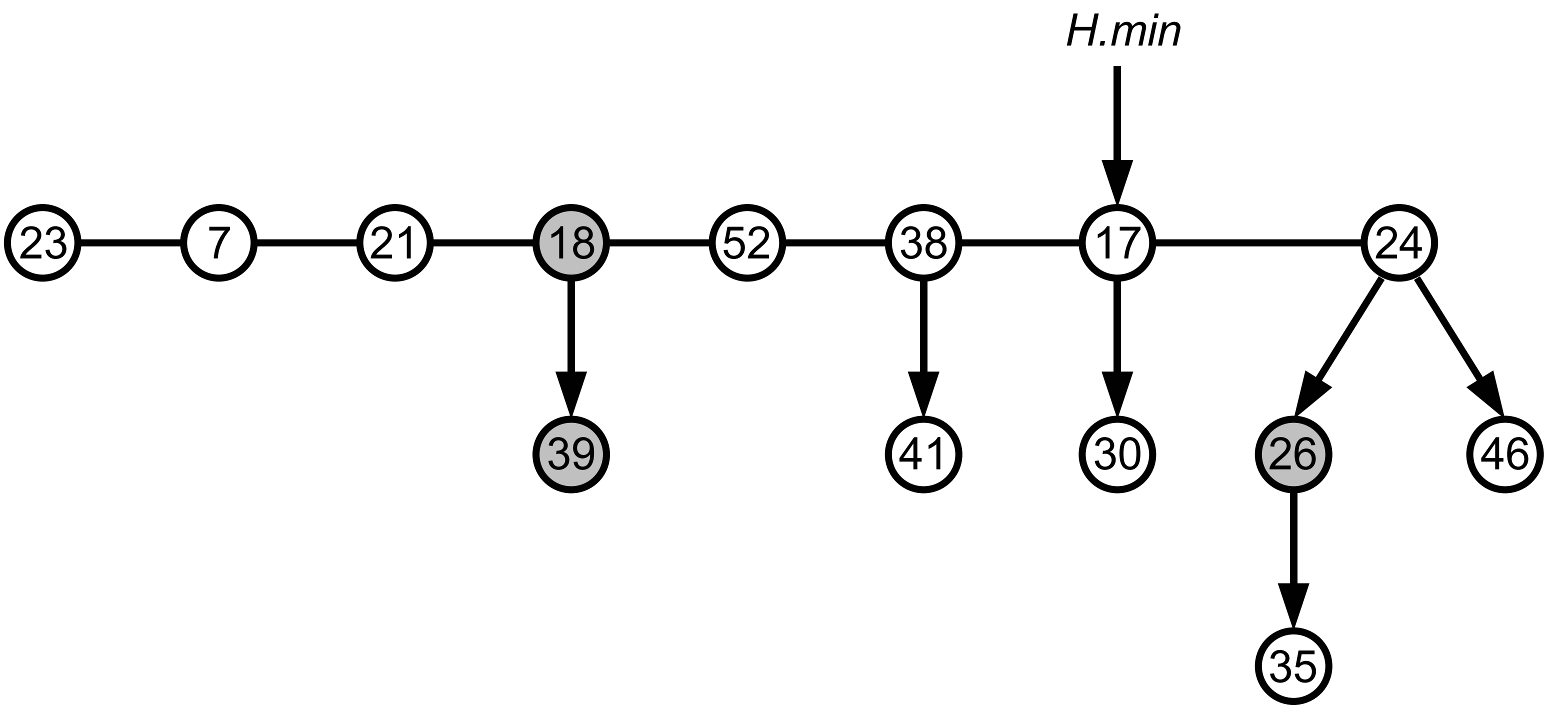
`ExtractMin`(H)

$z = H.min$
if $z \neq NULL$ **then**
    for każdy potomek x węzła z **do**
        dodaj x do listy korzeni H
        $x.parent = NULL$
    end for
    $zr = z.right$
    usuń z z listy korzeni H
    if $z = zr$ **then**
        $H.min = NULL$
    else
        $H.min = zr$
        `Consolidate`(H)
    end if
    $H.n = H.n - 1$
end if
return z

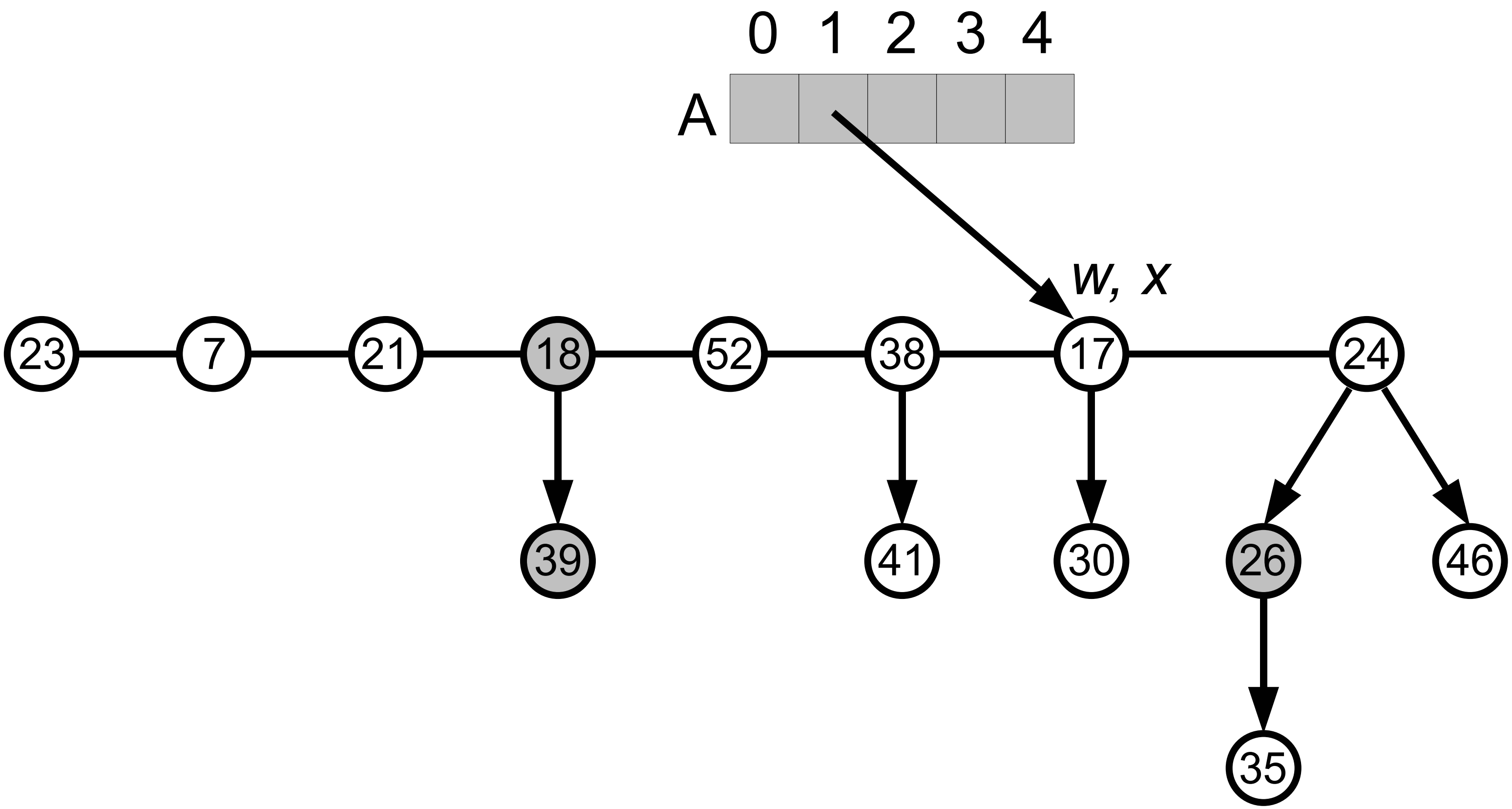
ExtractMin



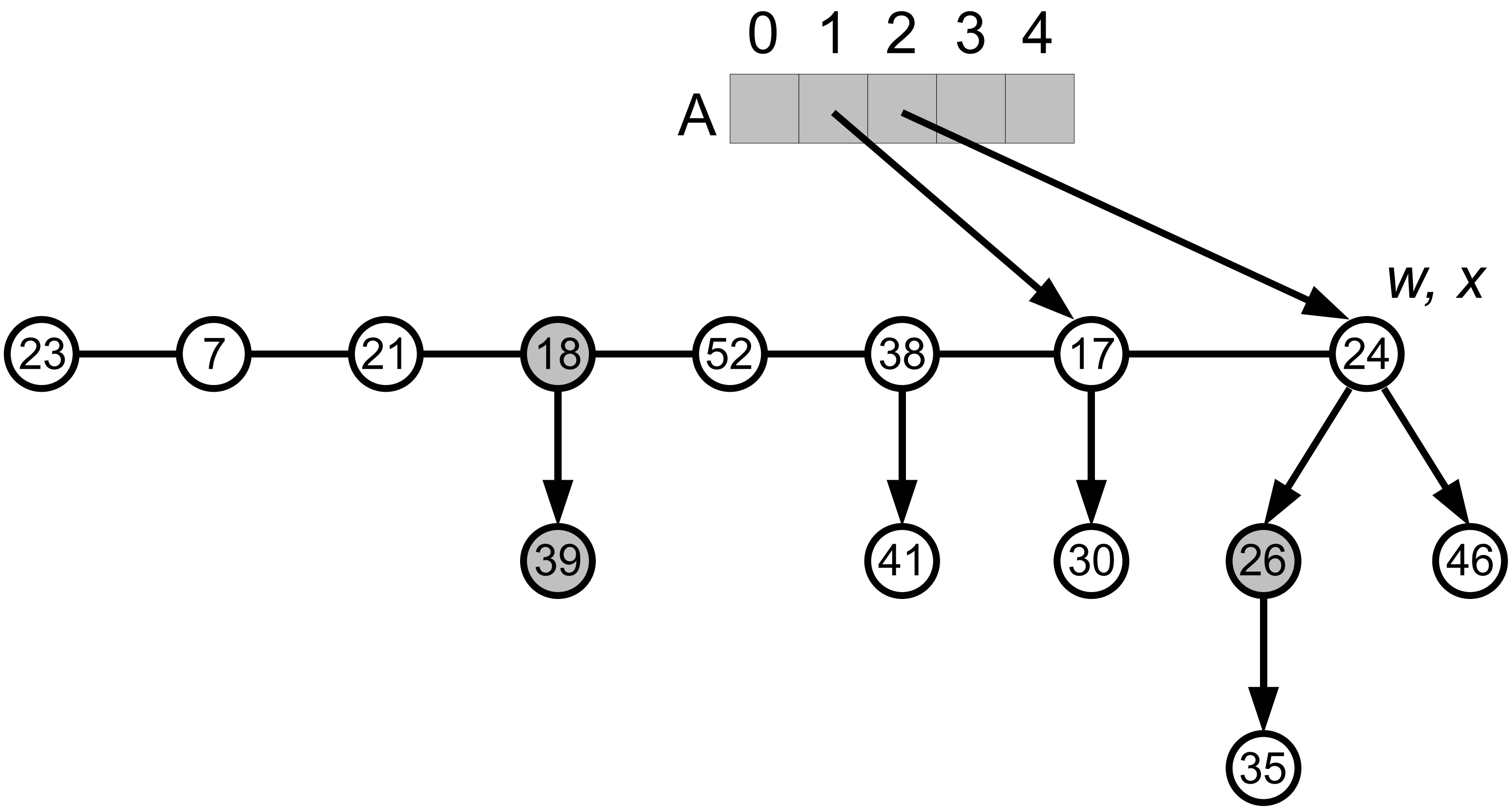
ExtractMin



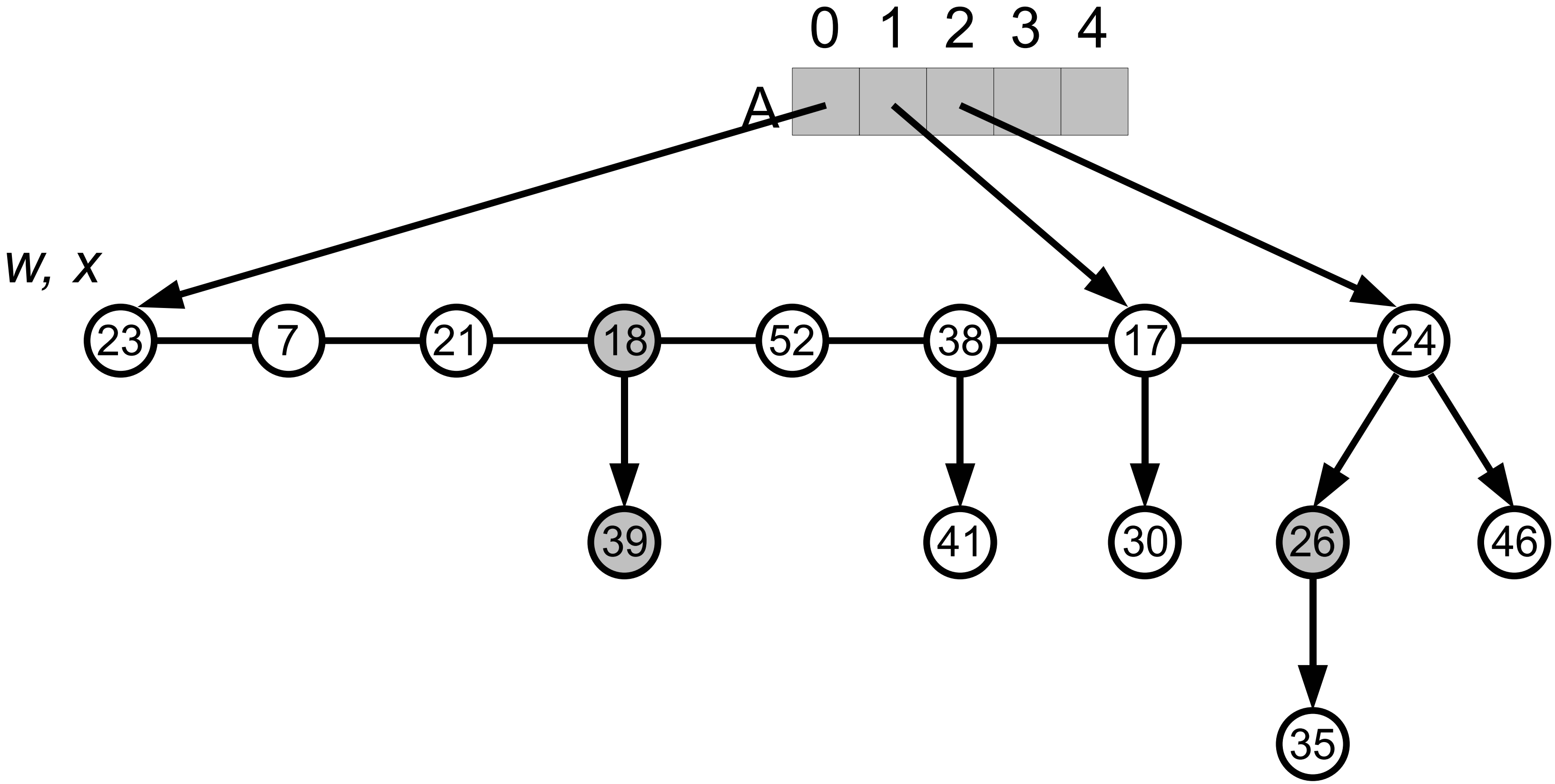
ExtractMin



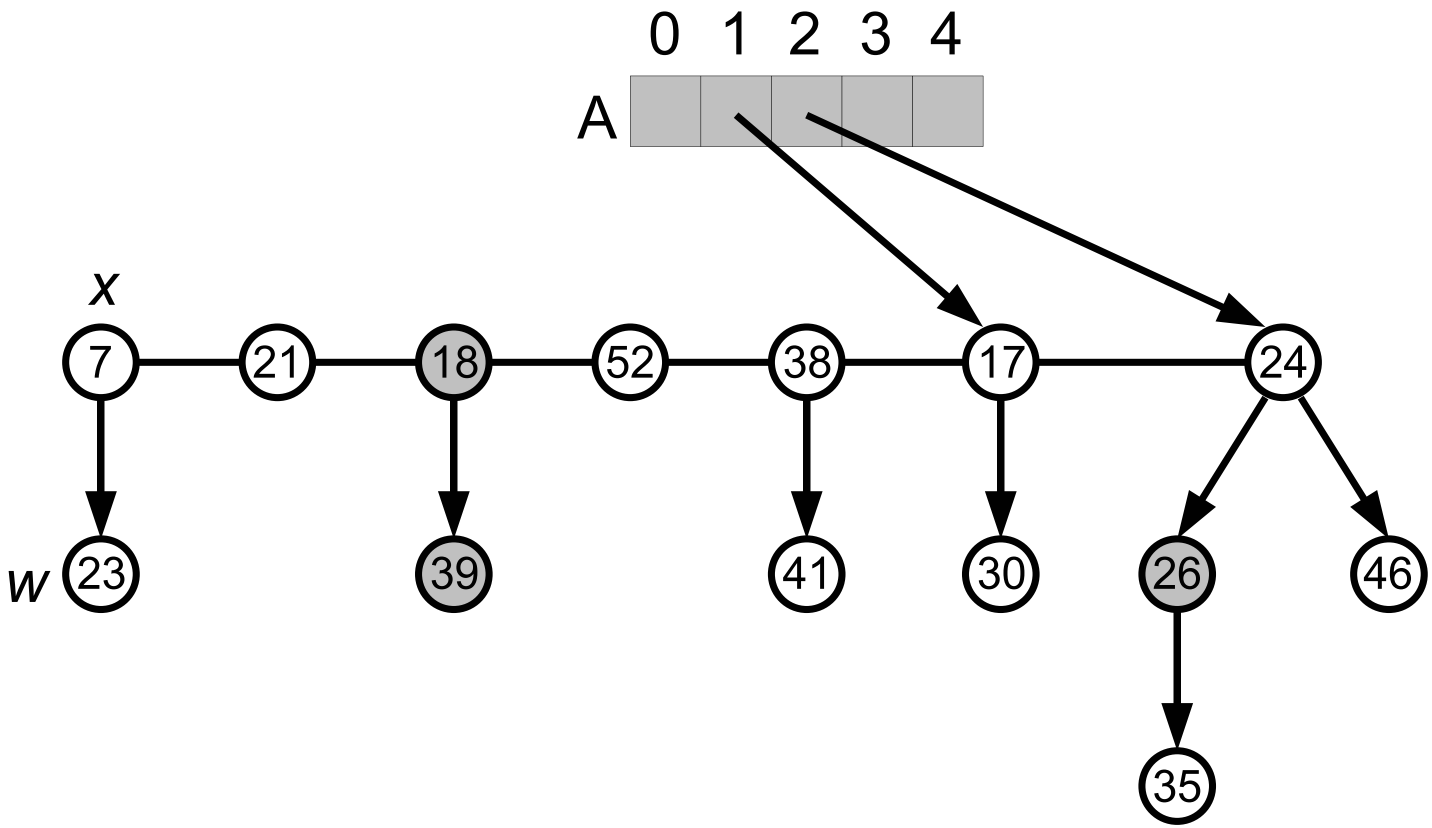
ExtractMin



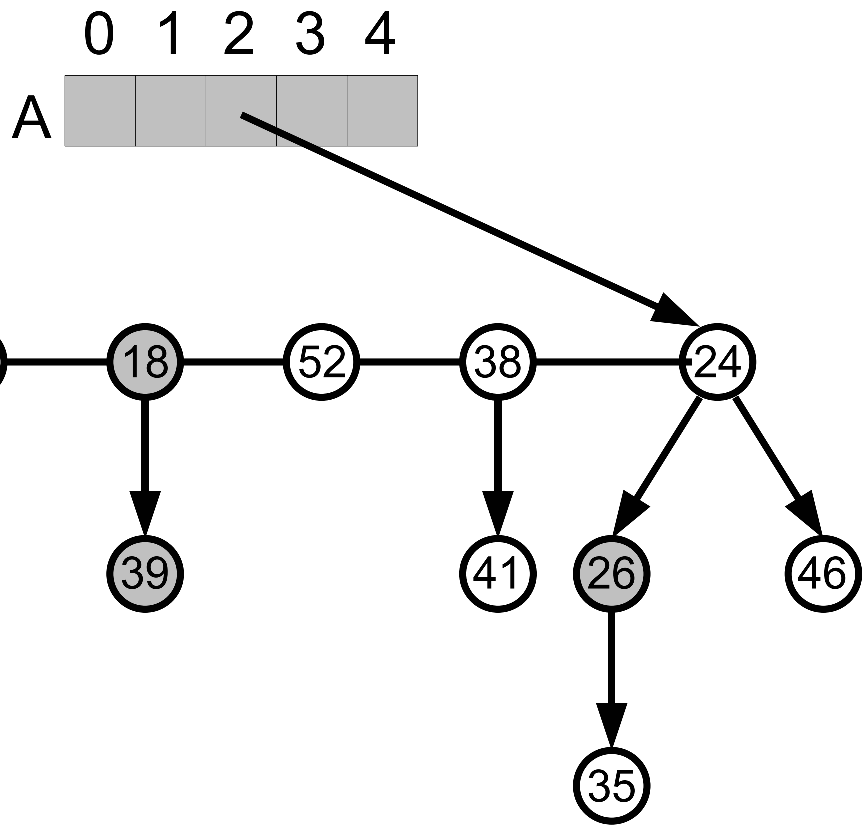
ExtractMin



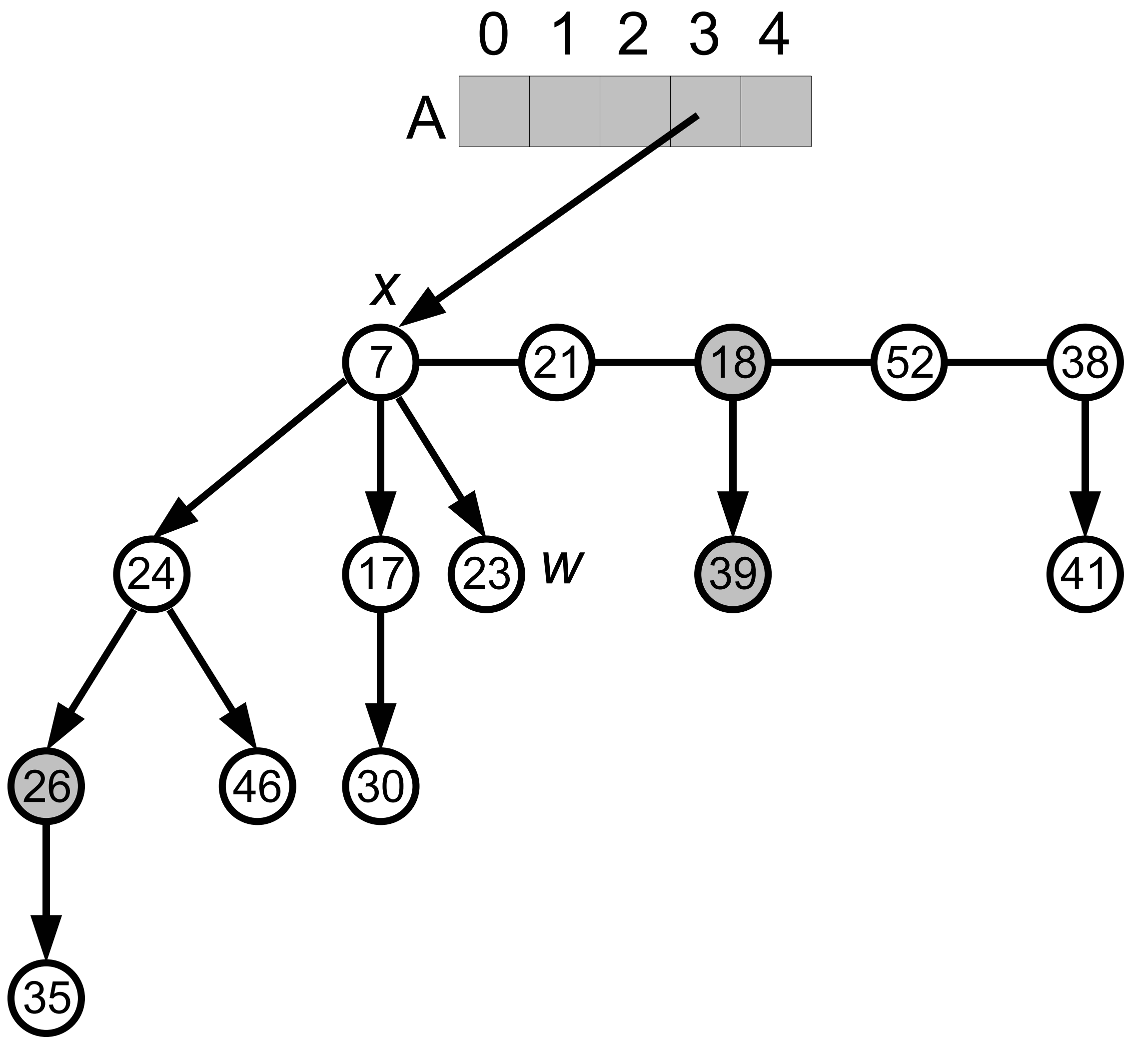
ExtractMin



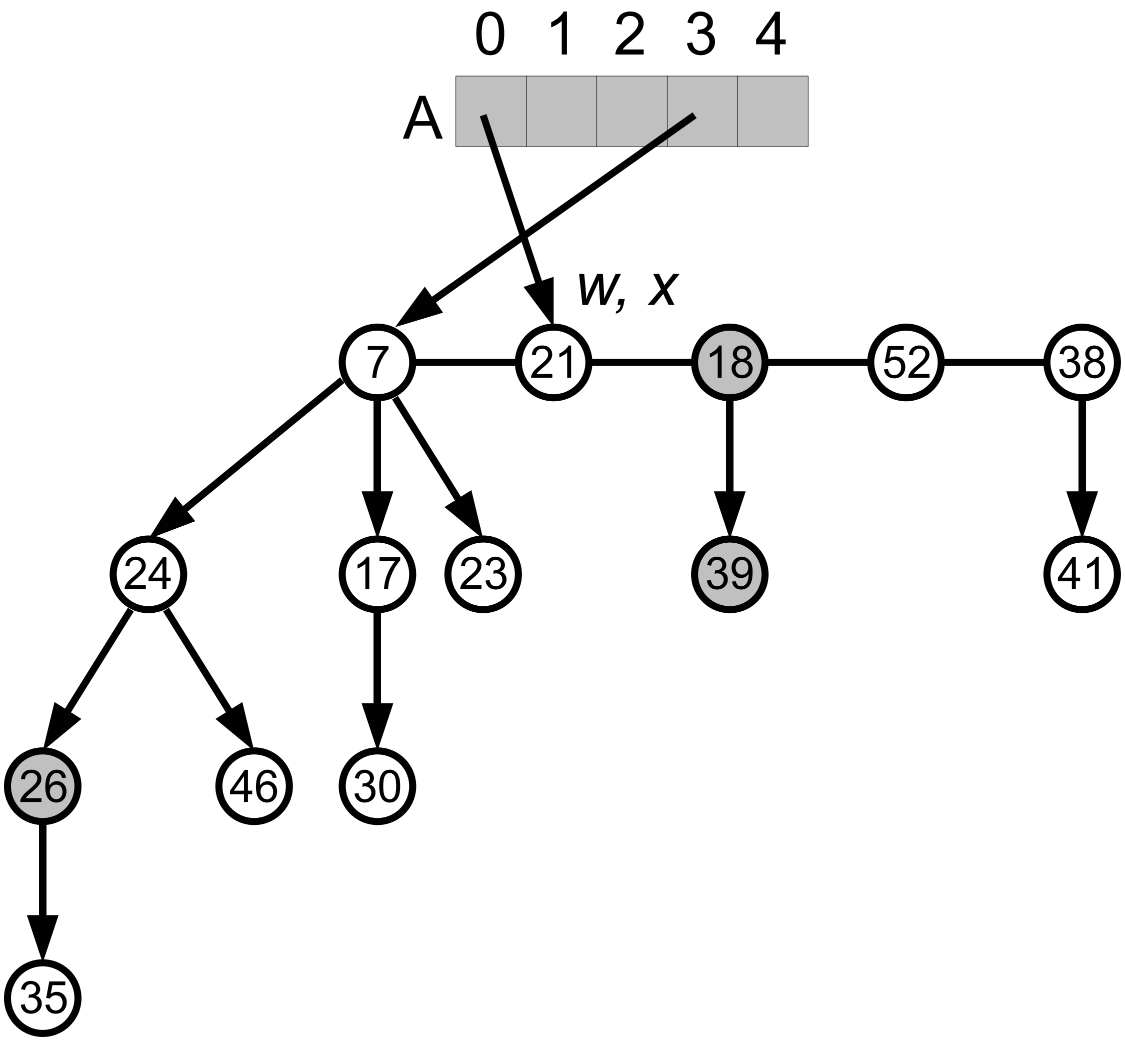
ExtractMin



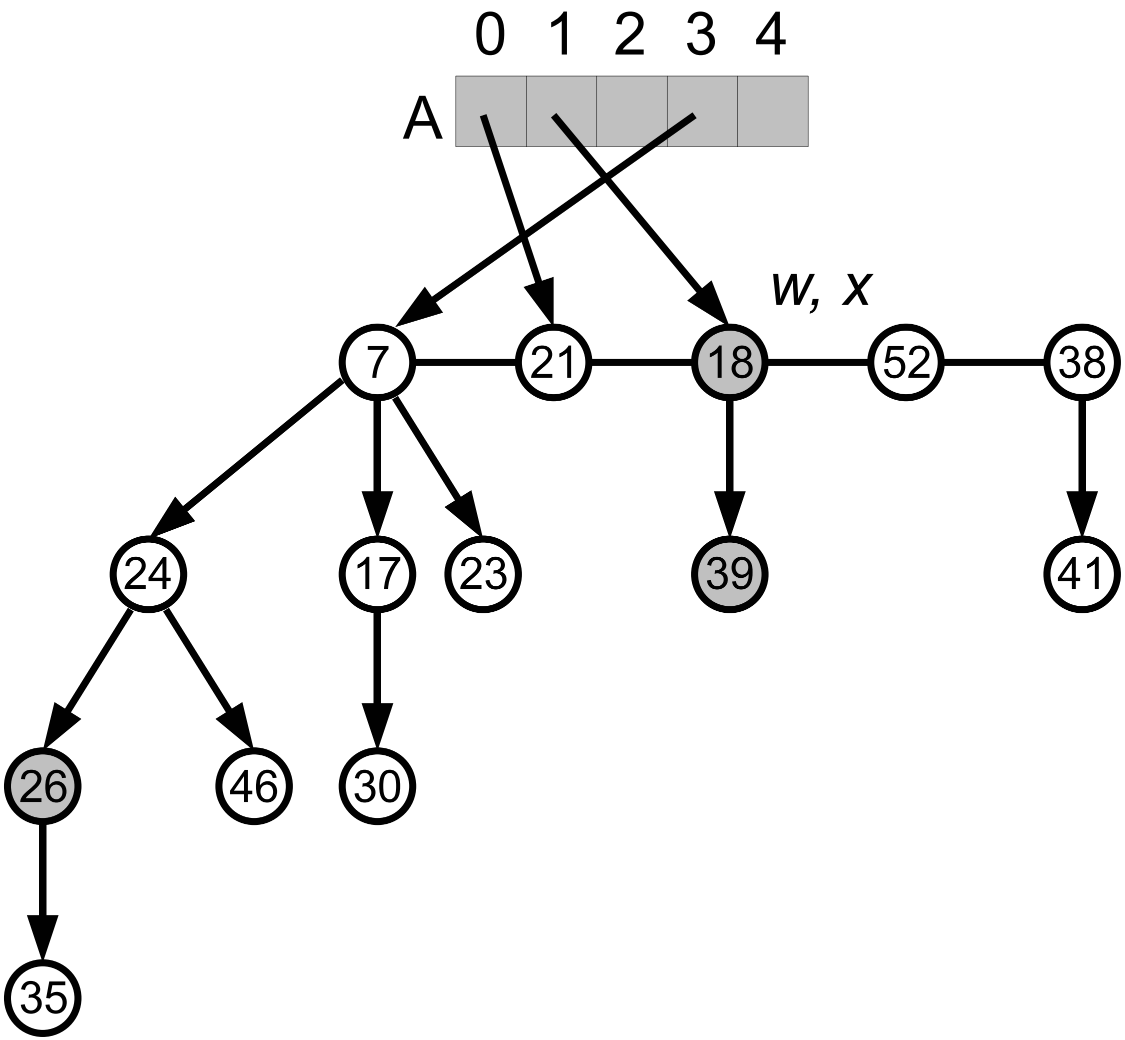
ExtractMin



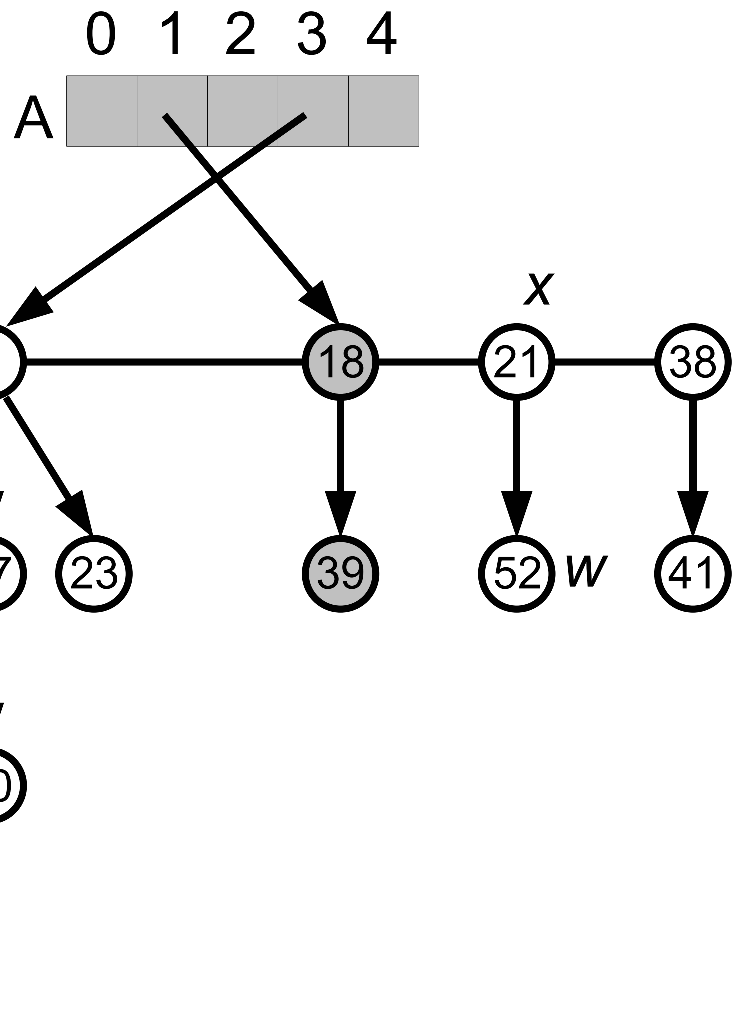
ExtractMin



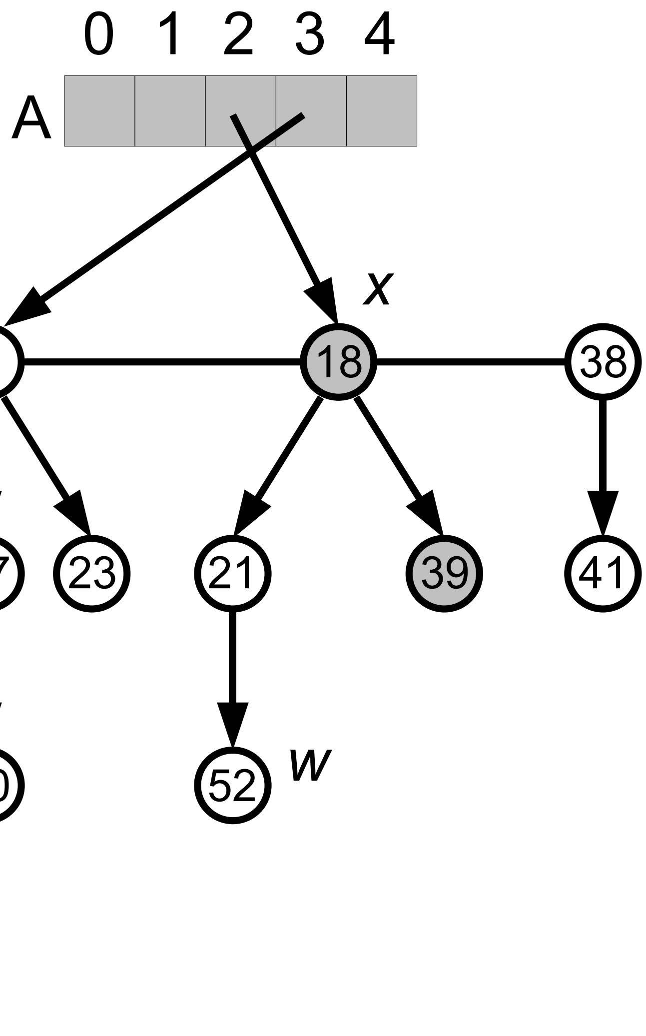
ExtractMin



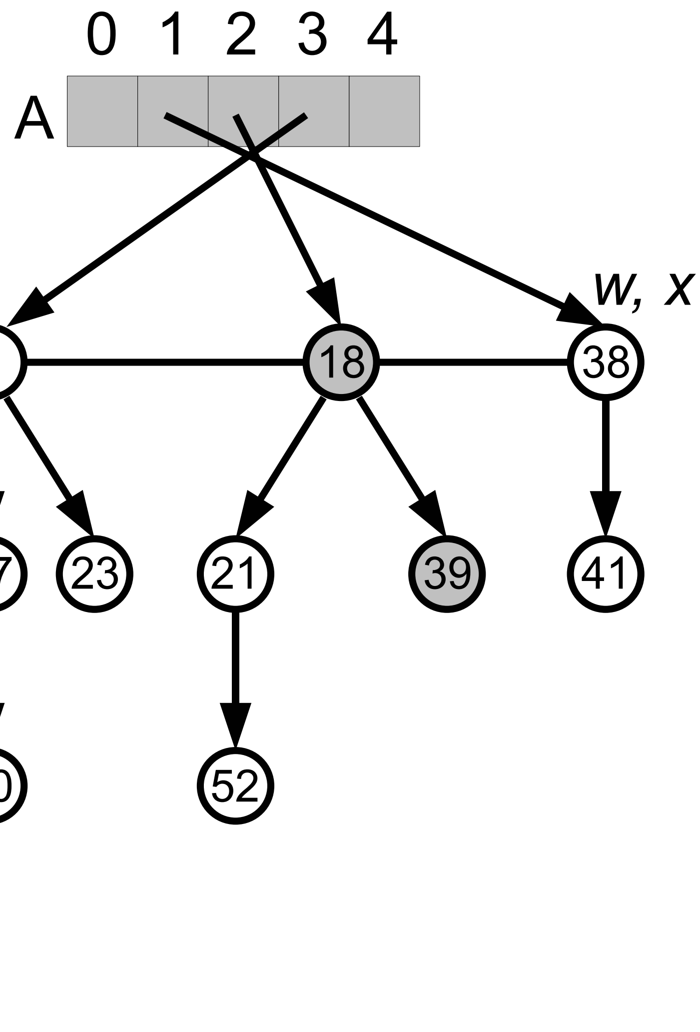
ExtractMin



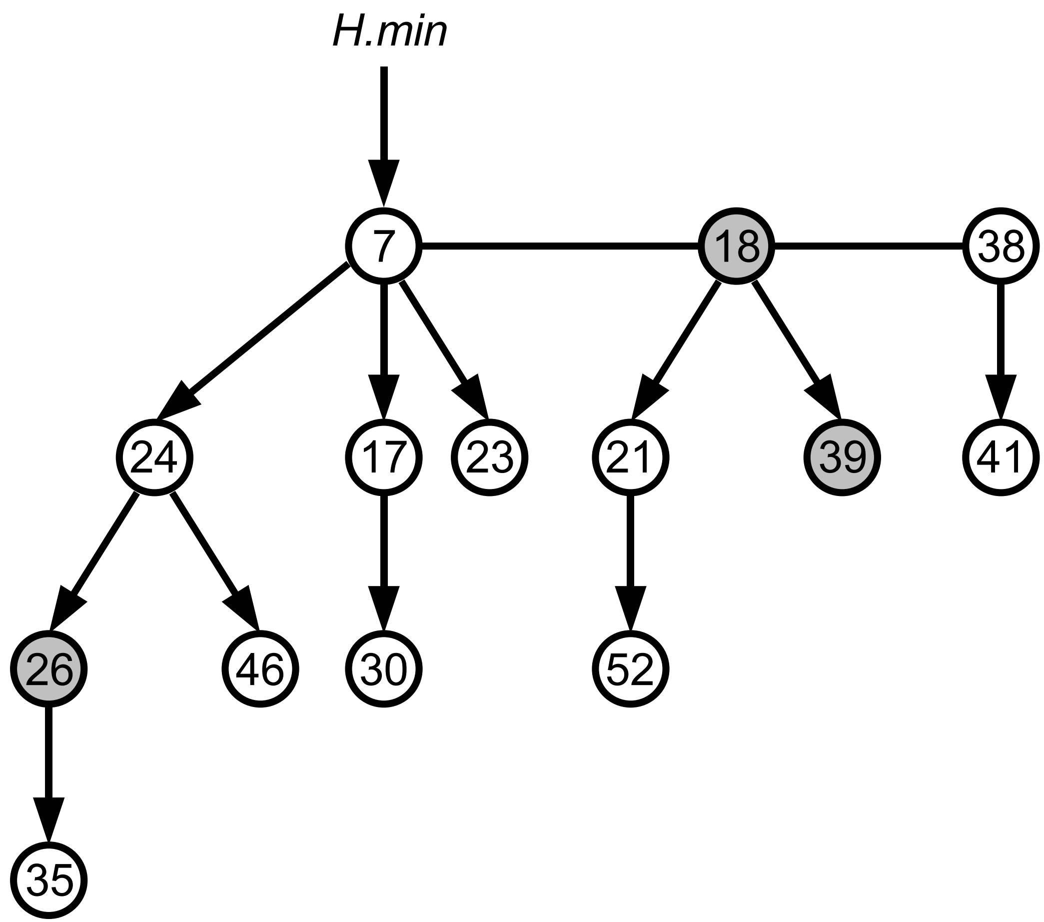
ExtractMin



ExtractMin



ExtractMin



DecreaseKey

- operacja ta wymaga wskaźnika na węzeł x zawierający klucz, którego wartość należy zmniejszyć
- jeżeli x jest korzeniem, lub $x.parent$ ma nie większy klucz niż x, nie ma potrzeby zmian w strukturze kopca
- w przeciwnym wypadku, odłączamy węzeł od rodzica, i dołączamy go do listy korzeni

DecreaseKey

- niech $p = x.parent$; jeżeli x jest drugim odłączanym potomkiem p od ostatniego dołączenia p od innego węzła (pole $p.mark = TRUE$) odcinamy p od jego rodzica i dołączamy p do listy korzeni
- odcięcie może się “propagować” w górę
- odcinanie węzła od rodzica w momencie utraty drugiego potomka ma na celu utrzymanie górnego oszacowania na stopień dowolnego węzła w n-węzłowym kopcu $D(n) \leq \lfloor \log_\phi n \rfloor$, gdzie $\phi = \frac{1+\sqrt{5}}{2}$

DecreaseKey

`DecreaseKey`(H, x, k)

if $x.key \leq k$ **then error** “zwiększenie wartości klucza”
$x.key = k$
$y = x.parent$
if $y \neq NULL$ **and** $x.key < y.key$ **then**
    `Cut`(H, x, y)
    `CascadingCut`(H, y)
end if
if $x.key < H.min.key$ **then**
    $H.min = x$
end if

Cut

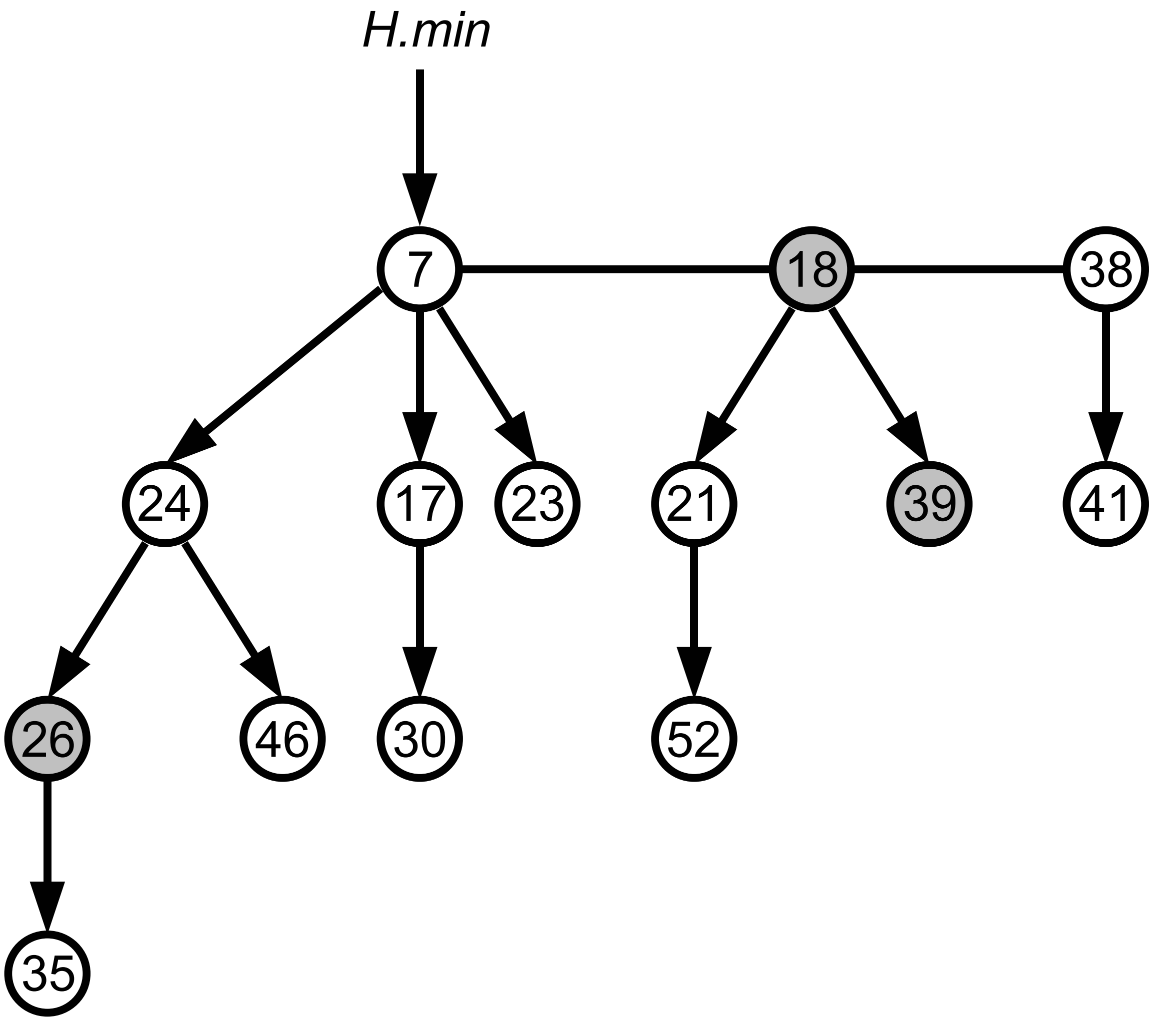
`Cut`(H, x, y)

1: usuń x z listy potomków y i zmniejsz $y.degree$ o 1
2: dodaj x do listy korzeni H
3: $x.parent = NULL$
4: $x.mark = FALSE$

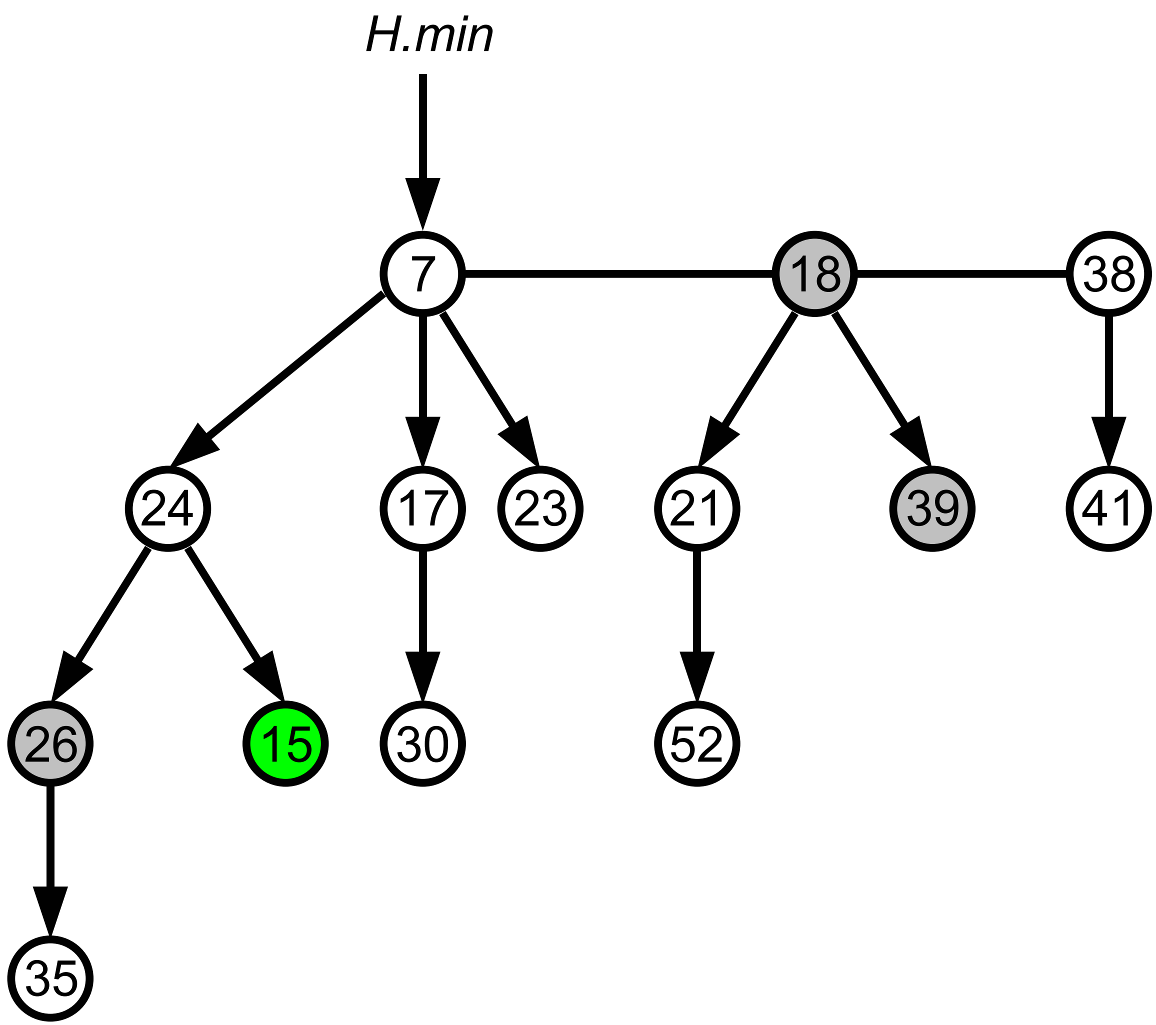
CascadingCut

```
CascadingCut(H, y)
  z = y.parent
  if z ≠ NULL then
      if y.mark = FALSE then
          y.mark = TRUE
      else
          Cut(H, y, z)
          CascadingCut(H, z)
      end if
  end if
```

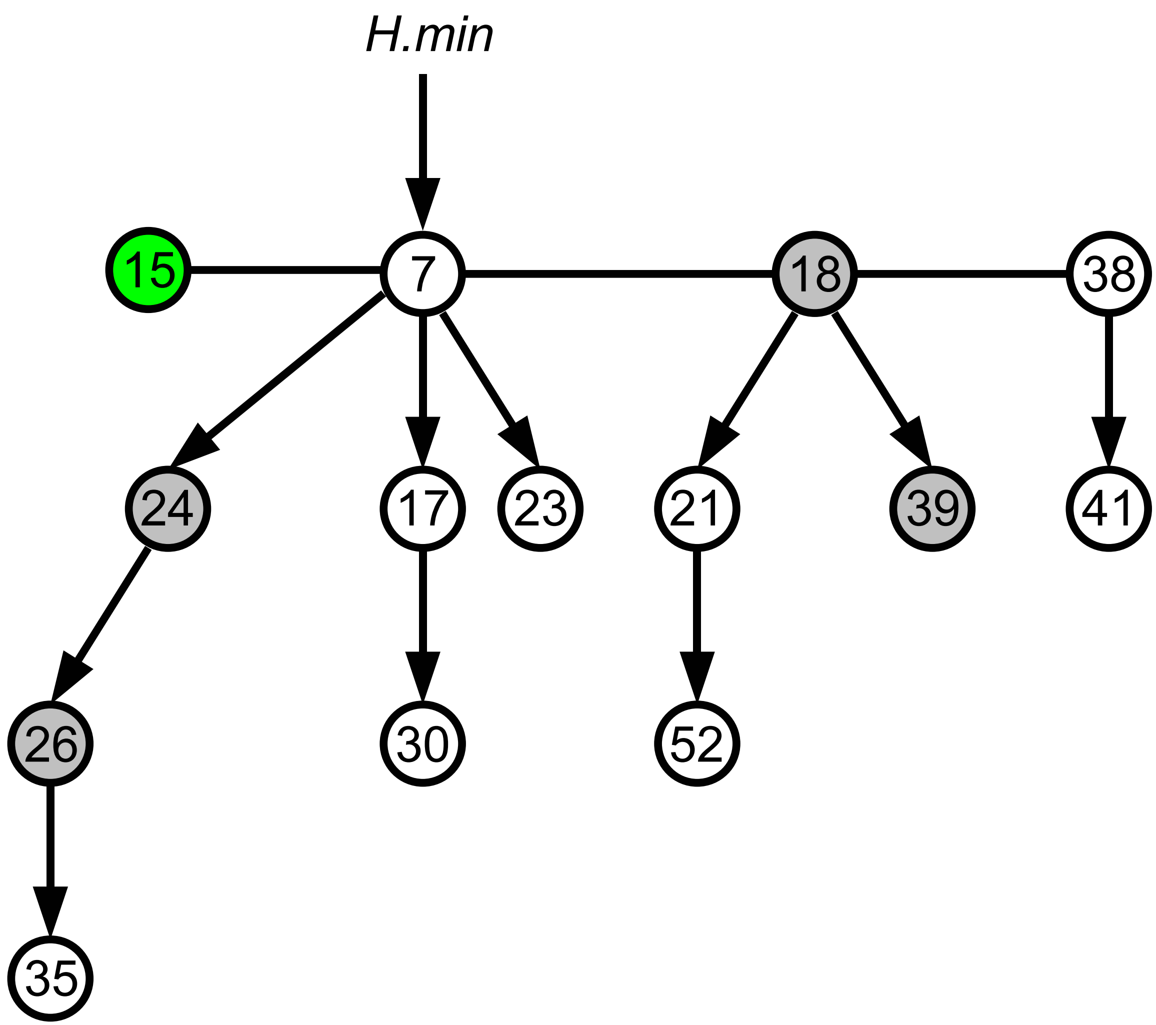
DecreaseKey



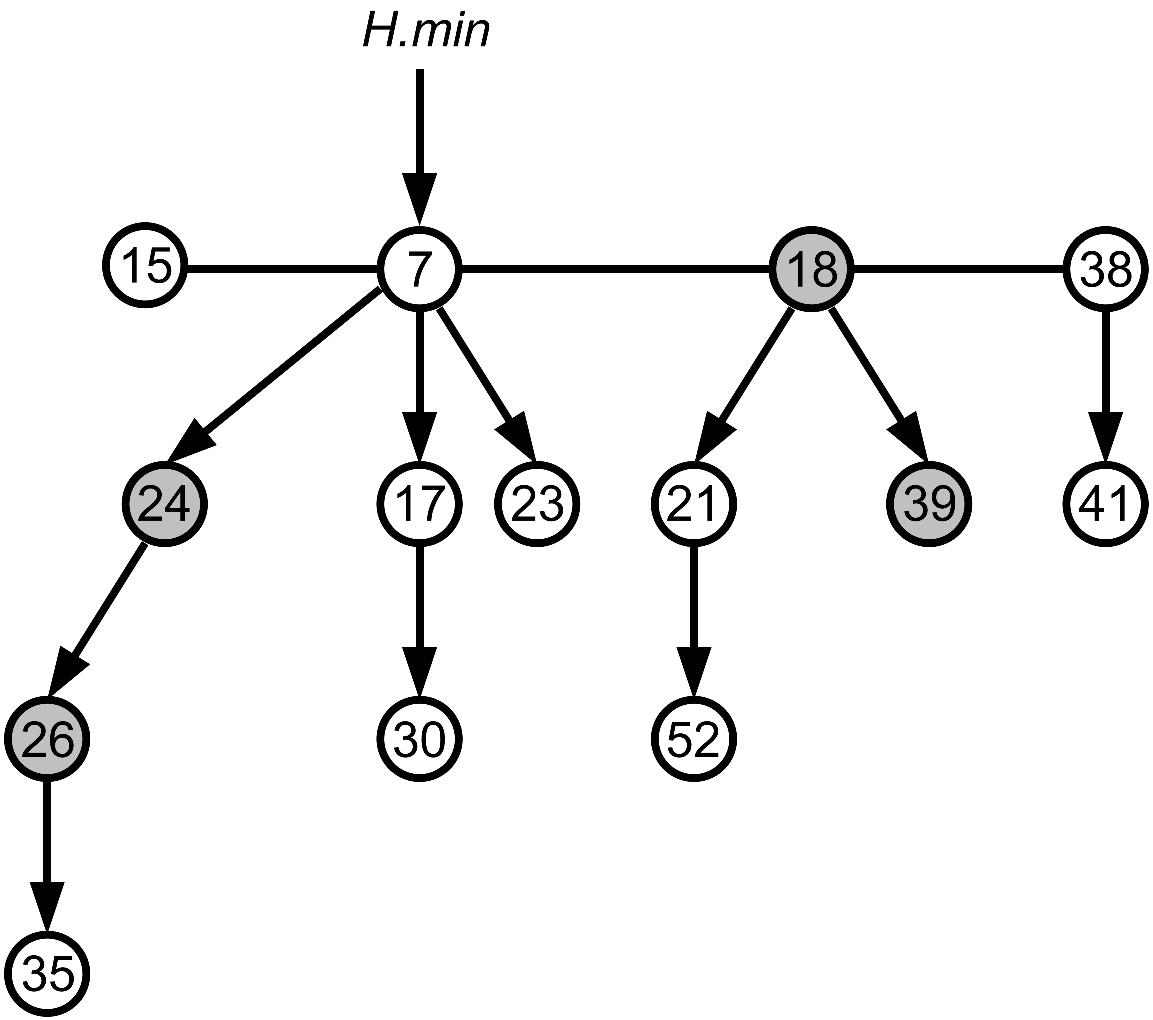
DecreaseKey



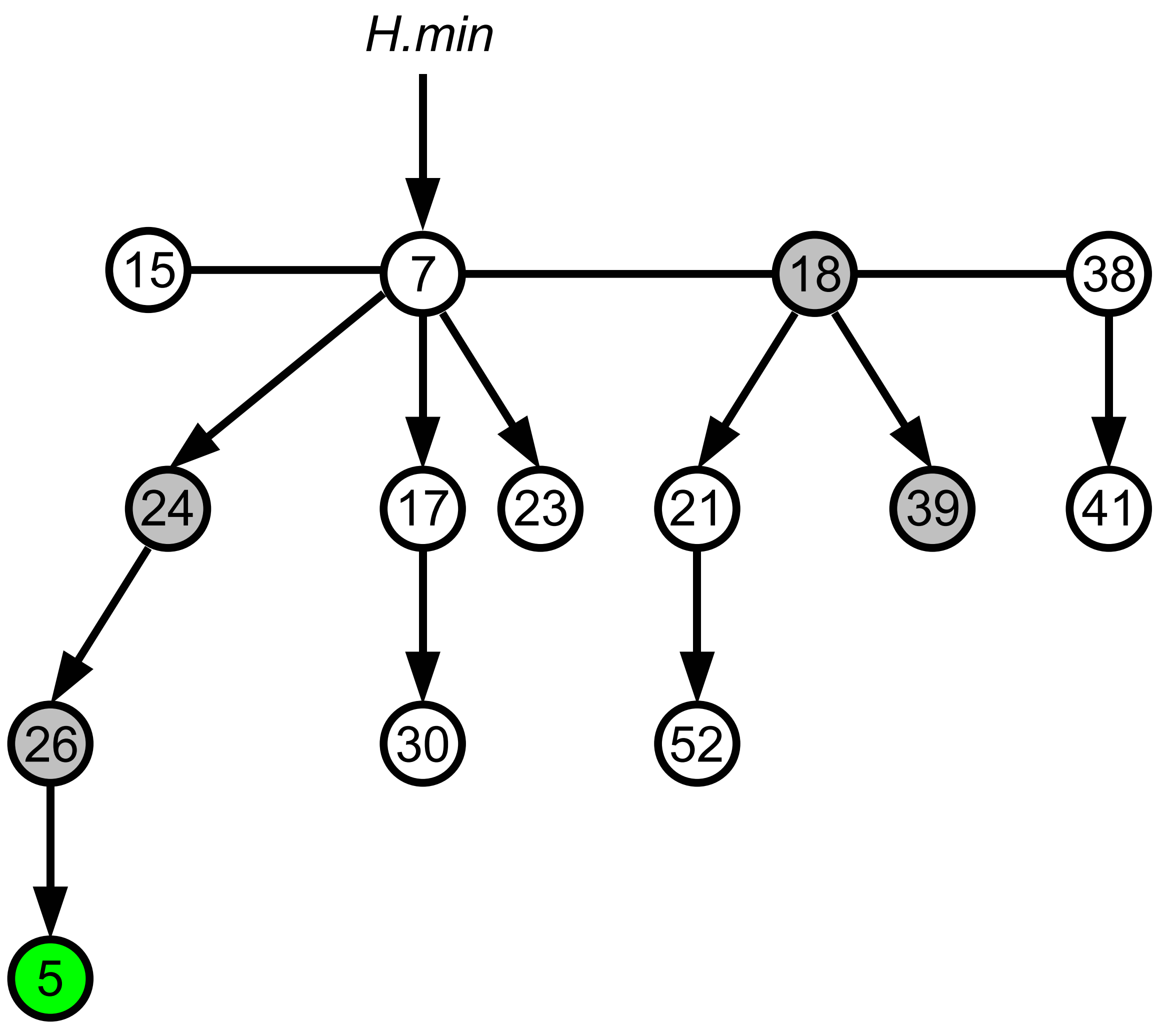
DecreaseKey



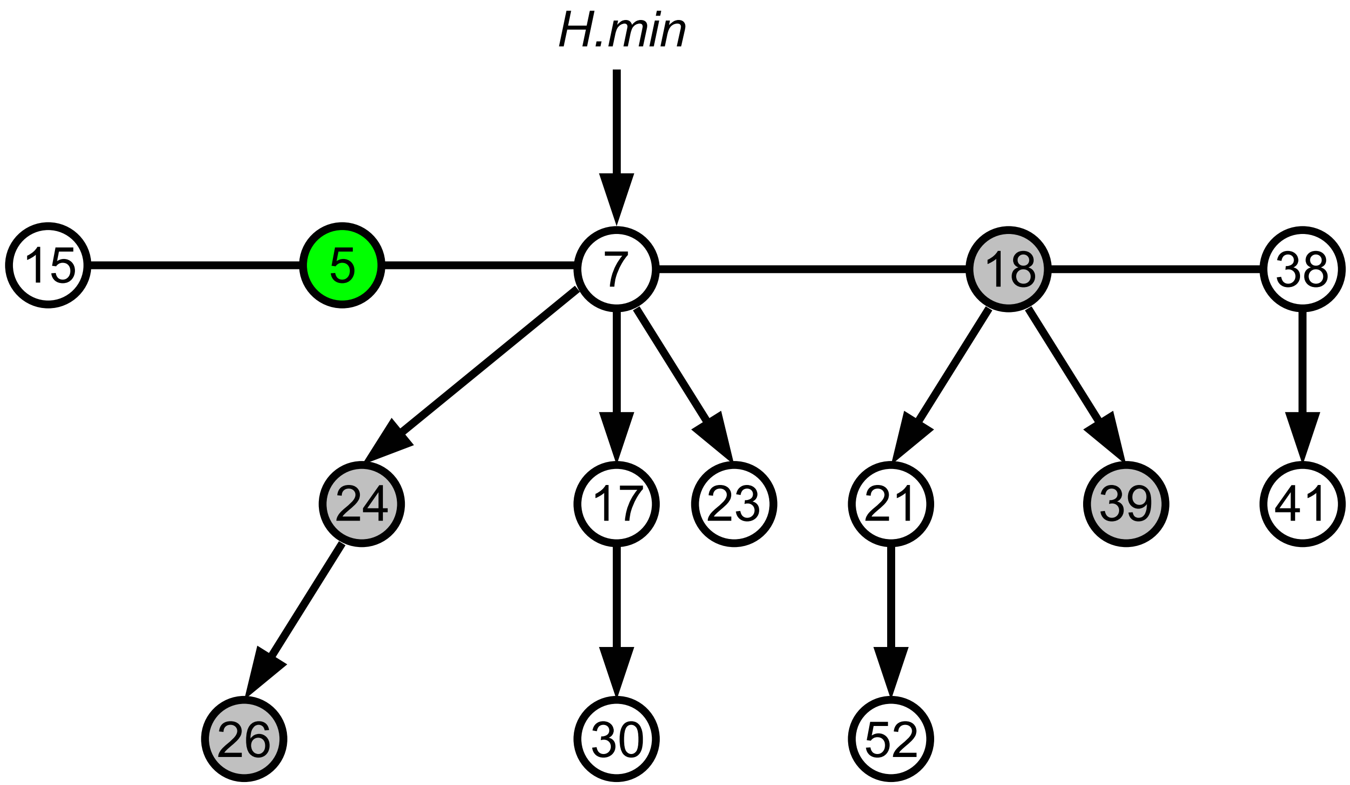
DecreaseKey



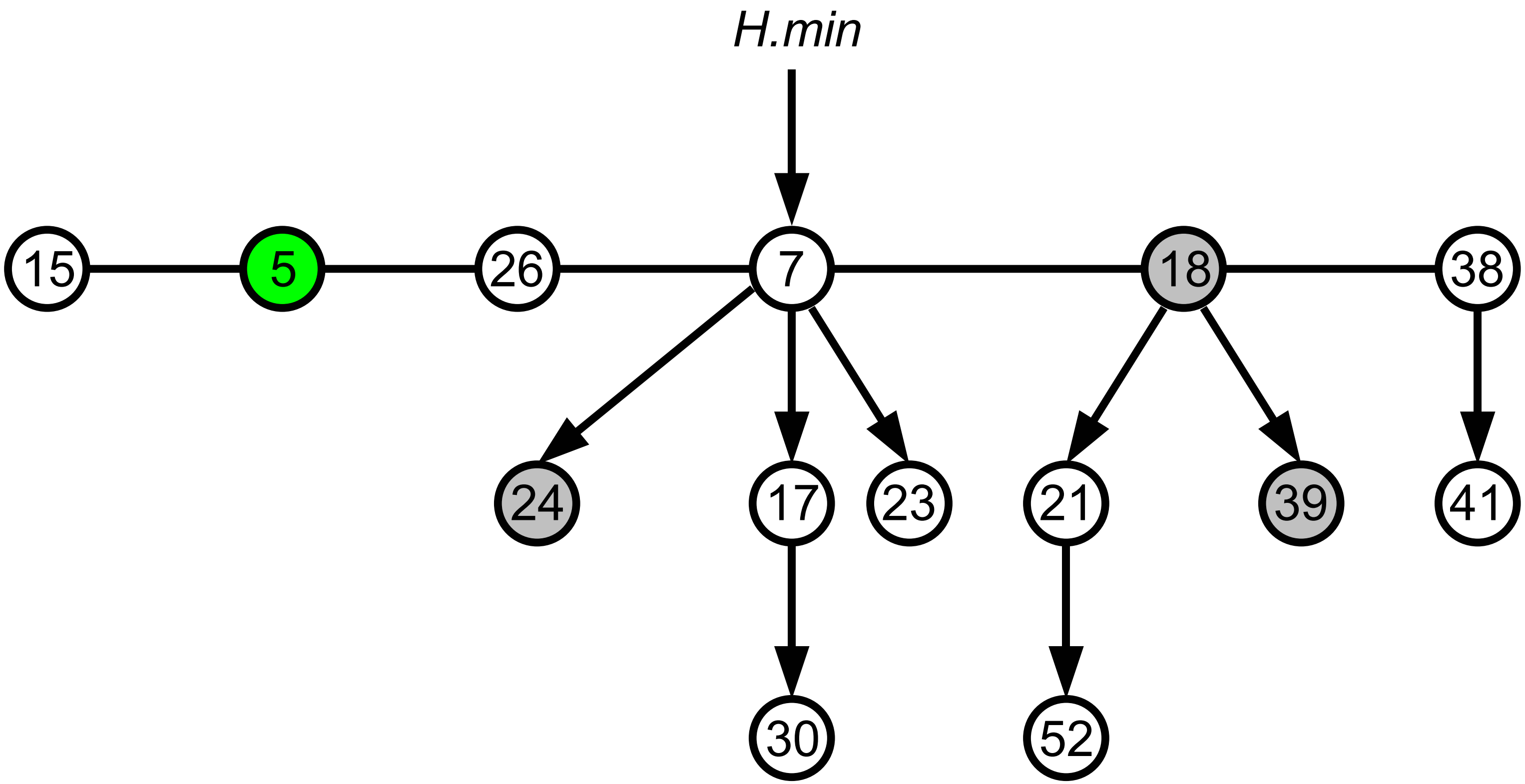
DecreaseKey



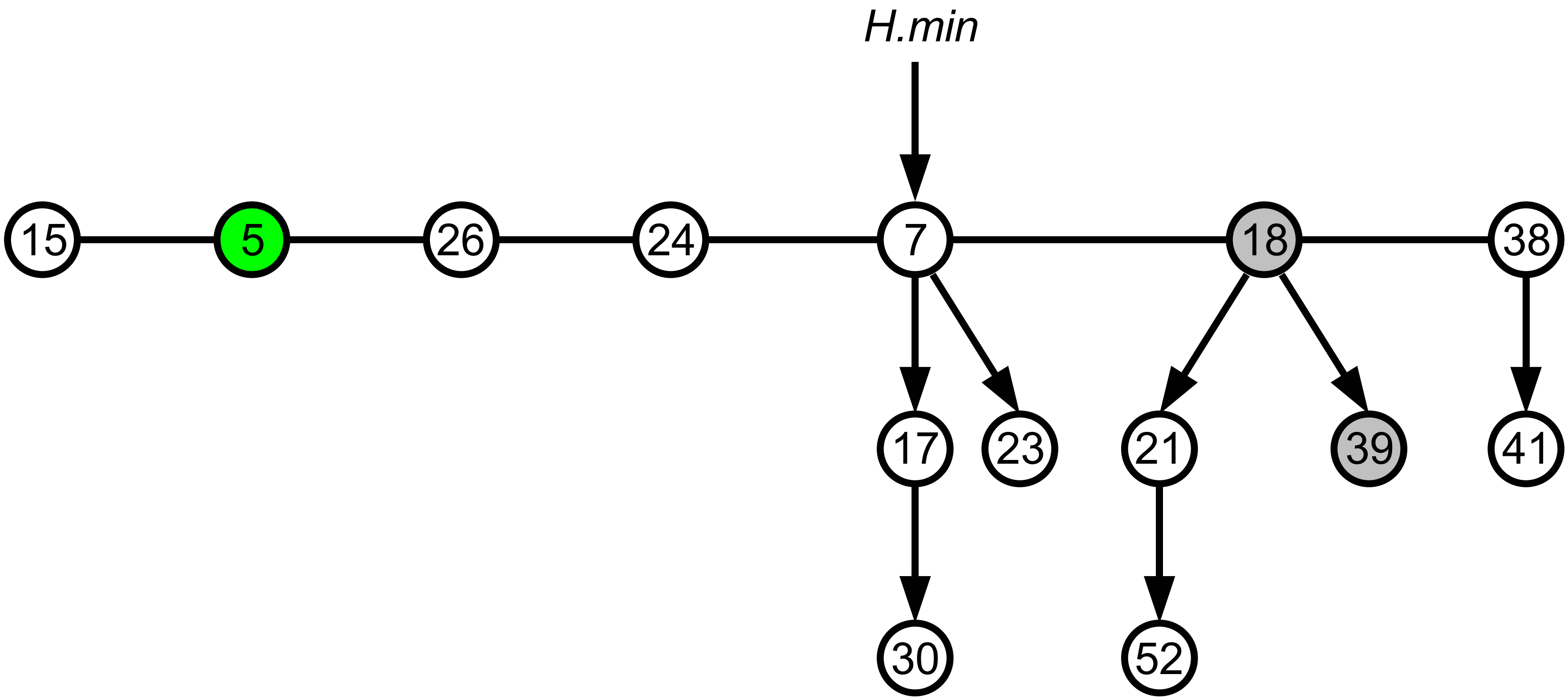
DecreaseKey



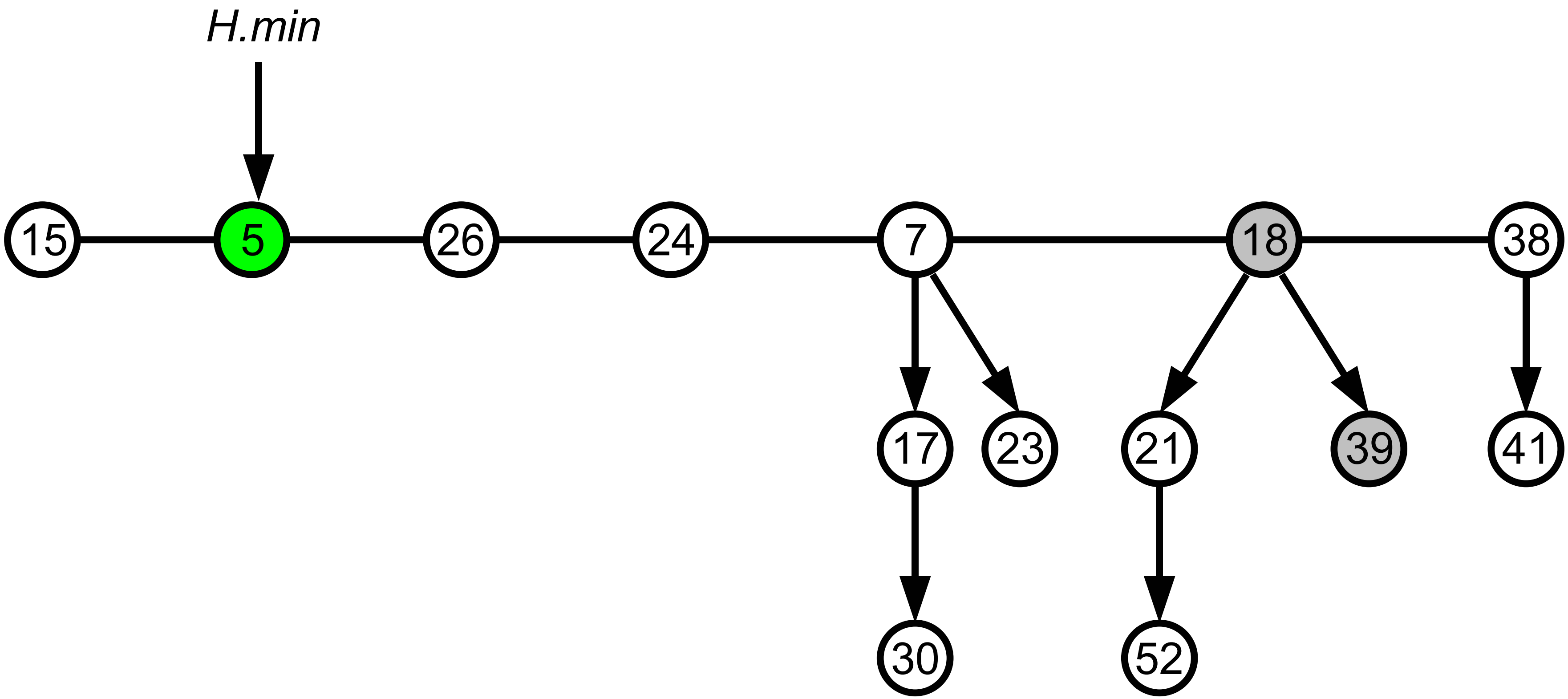
DecreaseKey



DecreaseKey



DecreaseKey



Delete

- również ta operacja wymaga wskaźnika na węzeł x zawierający klucz, który należy usunąć
- zmniejszamy wartość usuwanego klucza do $-\infty$
- usuwamy z kopca najmniejszy klucz (`ExtractMin`)

`Delete`(H, x)

1: `DecreaseKey`(H, x, $-\infty$)
2: `ExtractMin`(H)

Zbiory rozłączne

Zbiory rozłączne

- mamy pewną ilość elementów pogrupowanych w pewną ilość zbiorów rozłącznych
- chcemy wiedzieć, do którego zbioru należy element
- chcemy mieć możliwość łączenia dwóch zbiorów

Reprezentant

- do identyfikacji zbioru wykorzystujemy reprezentanta — jest to wyróżniony element w zbiorze
- sprawdzenie, czy elementy x i y należą do tego samego zbioru wykonujemy porównując reprezentanta zbioru, do którego należy x z reprezentantem zbioru zawierającego y
- zazwyczaj nie ma znaczenia, którego elementu ze zbioru użyjemy
- ważne jest natomiast, aby ponowne zapytanie o reprezentanta zwróciło taką samą odpowiedź (o ile zbiór się w tym czasie nie zmieniał)

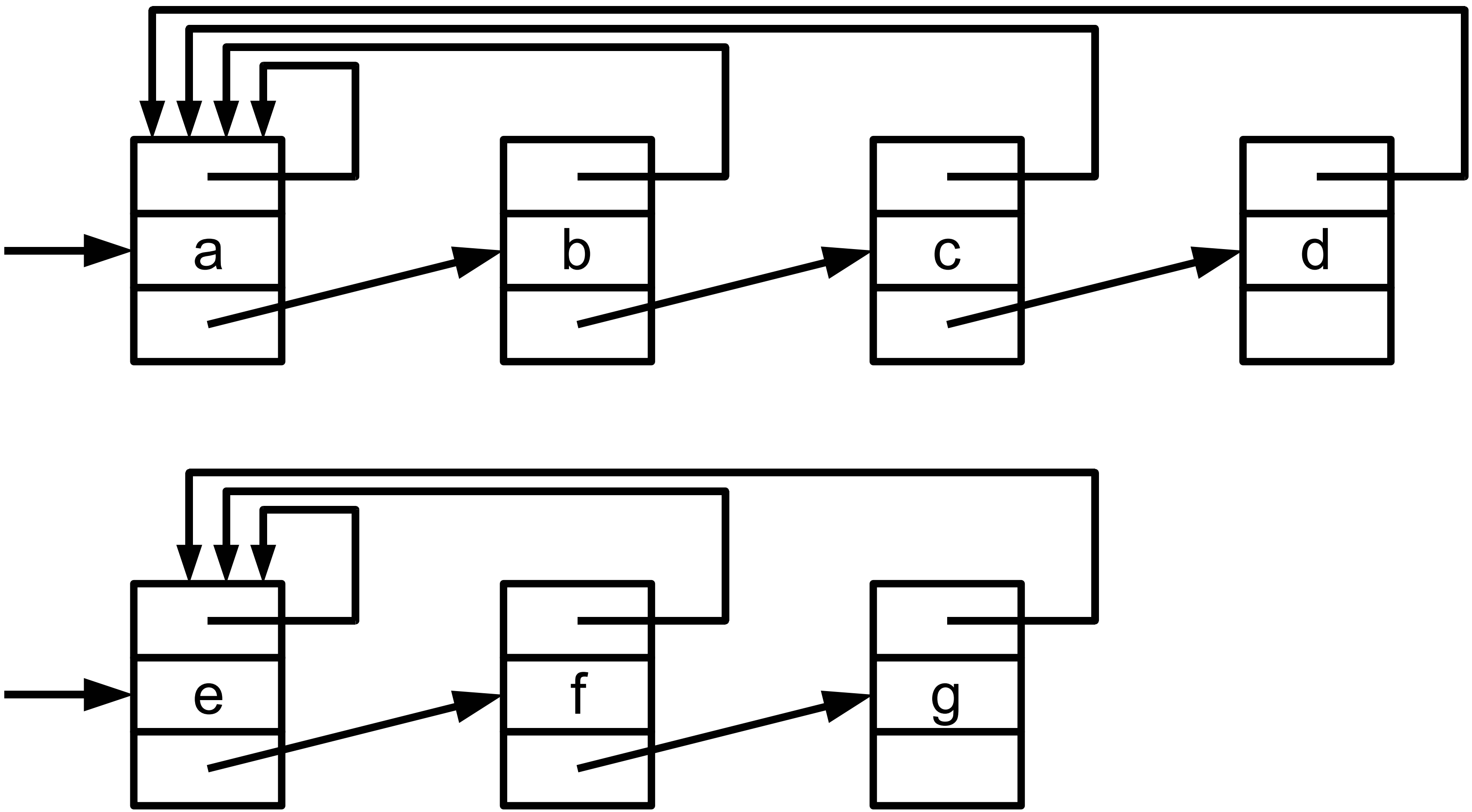
Zbiory rozłączne

- `MakeSet`(x) — utworzenie jednoelementowego zbioru, zawierającego x
- `Union`(x, y) — połączenie zbioru zawierającego x ze zbiorem zwawierającym y, otrzymujemy nowy zbiór (zbiory wejściowe przestają samodzielnie istnieć); nowym reprezentantem może być którykolwiek element z połączonego zbioru
- `FindSet`(x) — znalezienie reprezentanta zbioru zawierającego x

Reprezentacja listowa

- każdy zbiór reprezentujemy za pomocą listy
- reprezentantem zbioru (listy) jest jej pierwszy element (głowa)
- każdy węzeł posiada następujące pola:
  - element zbioru
  - wskaźnik na następny węzeł
  - wskaźnik na pierwszy element listy (reprezentanta)

Reprezentacja listowa



`MakeSet`

- operacja `MakeSet` wymaga tylko utworzenia jednoelmentowej listy

`MakeSet`(x)

1: $L =$ **new** ListNode
2: $L.repr = L$
3: $L.next = NULL$
4: $L.elem = x$
5: **return** L

FindSet

- operacja `FindSet` zwraca zawartość wskaźnika $repr$ węzła

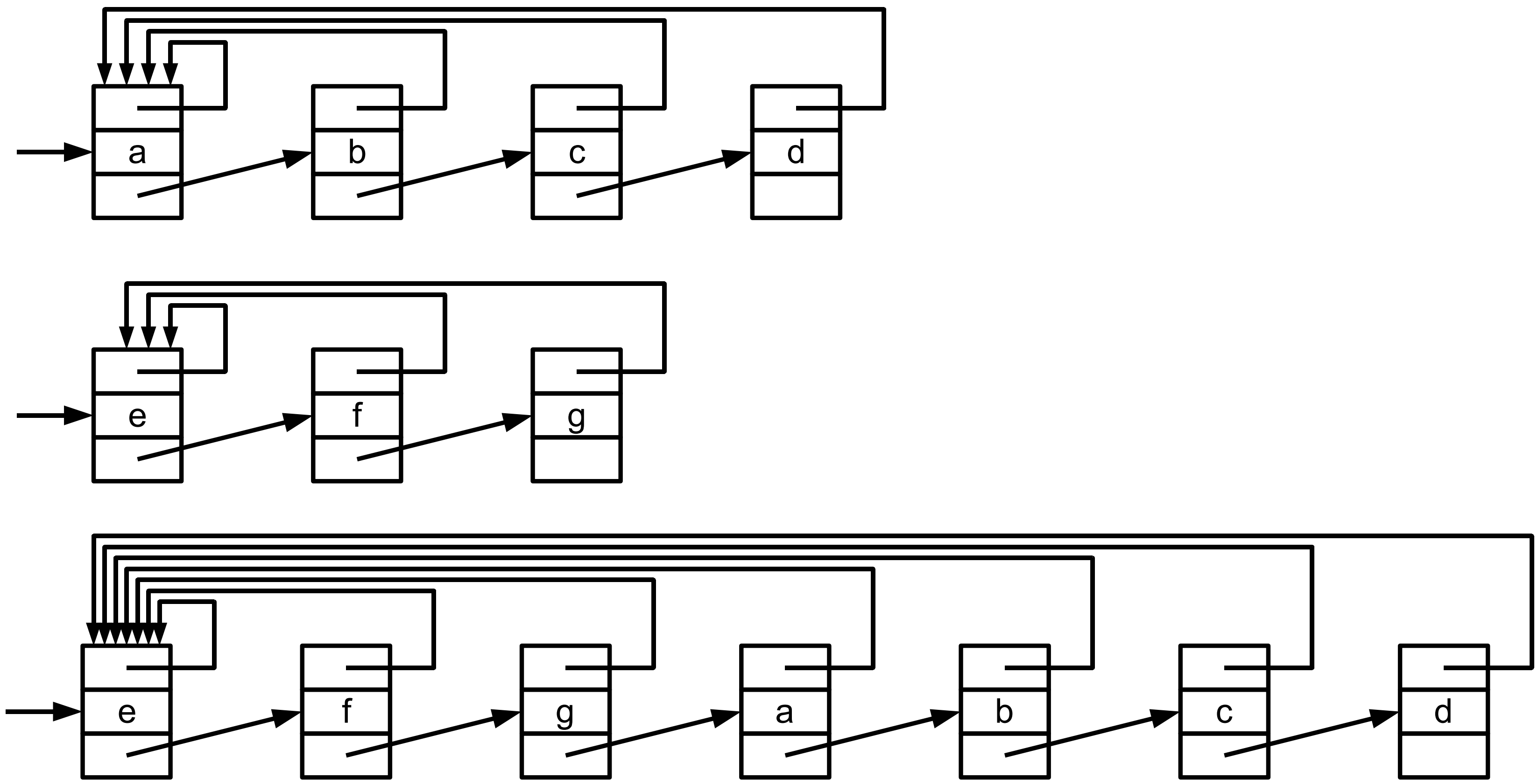
`FindSet`(x)

1: **return** $x.repr$

`Union`

- najprostsza implementacja procedury `Union` — dołączamy listę z elementem x na koniec listy z elementem y
- musimy jeszcze uaktualnić wskaźniki na reprezentanta we wszystkich węzłach należących do listy zbioru zawierającego x
- jest to kosztowne — zajmuje czas liniowy względem długości listy

Reprezentacja listowa



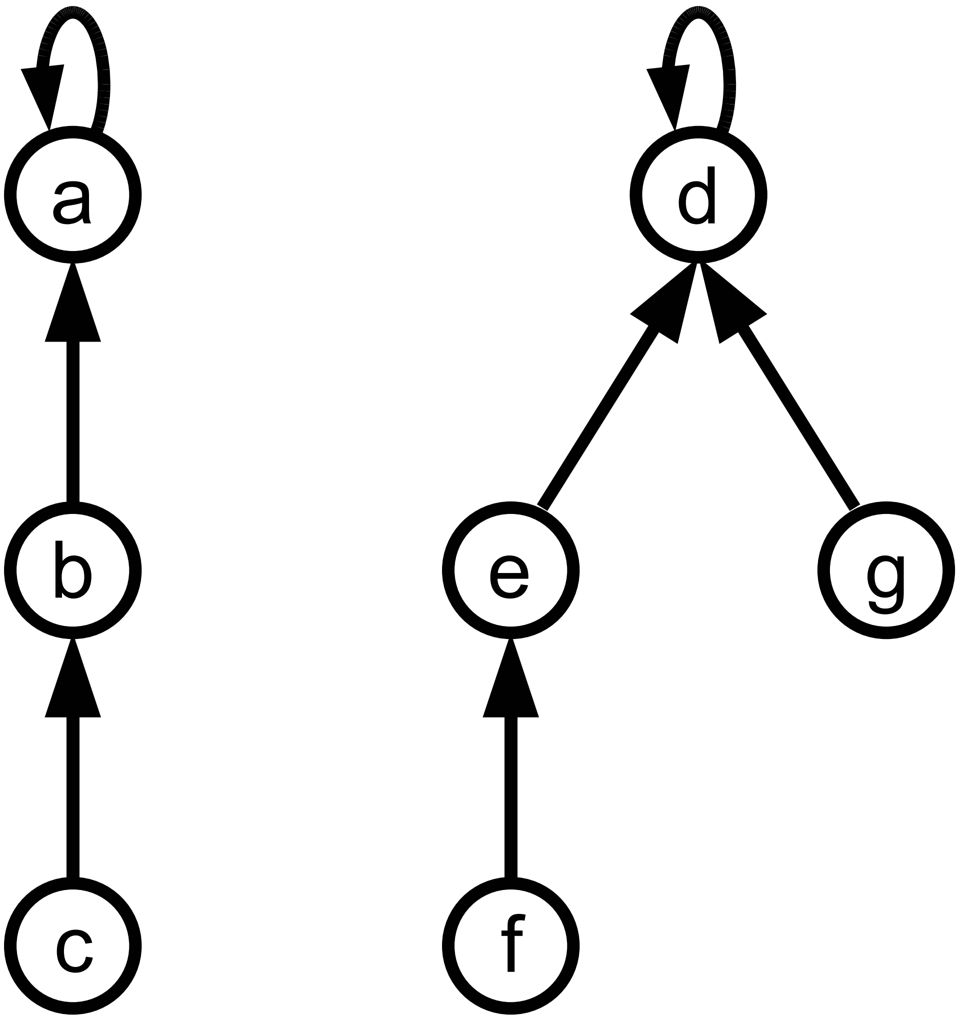
Heurystyka z wyważaniem

- efektywność procedury `Union` można poprawić pamiętając w każdym reprezentancie długość listy, którą reprezentuje (uaktualnianie jej nie jest kosztowne)
- łącząc dwa zbiory dołączamy zawsze krótszą listę na koniec dłuższej (rozstrzygając remisy w dowolny sposób)
- operacja nadal zajmuje czas liniowy względem długości listy, jednak można pokazać, że ciąg m operacji `MakeSet`, `Union` i `FindSet`, spośród których n to operacje `MakeSet` (operacji `Union` może być zatem co najwyżej $n-1$) zajmuje $O(m + n \log n)$ czasu

Lasy zbiorów rozłącznych

- w tej implementacji przedstawiamy zbiory przy pomocy drzew ukorzenionych
- korzeń drzewa jest jego reprezentantem
- wszystkie nasze zbiory tworzą zatem las
- każdy węzeł zawiera element oraz wskaźnik na rodzica
- rodzicem reprezentanta jest on sam

Lasy zbiorów rozłącznych



MakeSet

- operacja `MakeSet` wymaga tylko utworzenia jednoelmentowego drzewa

`MakeSet`(x)

1: $T =$ **new** TreeNode
2: $T.parent = T$
3: $T.elem = x$
4: **return** T

FindSet

- reprezentanta zbioru do którego należy x znajdujemy, przechodząc do rodzica tak długo, aż dojdziemy do korzenia

FindSet(x)

1: $y = x$
2: **while** $y.parent \neq y$ **do**
3: $\quad y = y.parent$
4: **end while**
5: **return** x

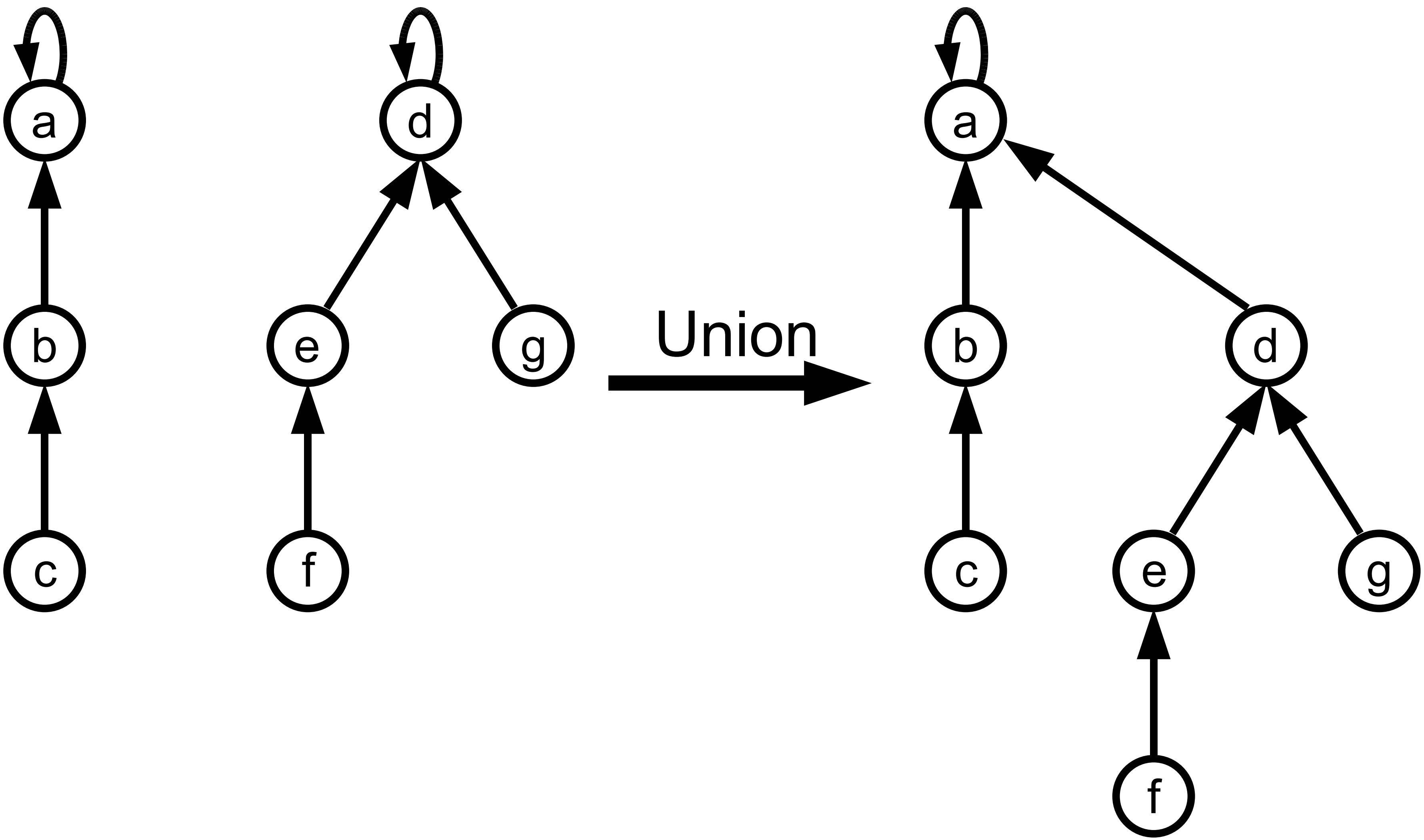
`Union`

- najprostsza implementacja procedury `Union` polega na modyfikacji rodzica reprezentanta jednego ze zbiorów

`Union`(x, y)

1: $rx =$ `FindSet`(x)
2: $ry =$ `FindSet`(y)
3: $ry.parent = rx$

Lasy zbiorów rozłącznych



`Union`

- przedstawiona implementacja procedury `Union` nie ma przewagi nad prostszą wersją operacji `Union` na listach
- jej efektywność można jednak znacząco poprawić wykorzystując dwie heurystyki:
  - łączenie według rangi
  - kompresję ścieżek

Łączenie według rangi

- zasada działania jest podobna do heurystyki z wyważaniem
- do drzewa większego dołączamy mniejsze (nigdy na odwrót)
- jednak zamiast ilości elementów w zbiorze, w korzeniu pamiętamy jedynie górne ograniczenie na wysokość korzenia (jego rangę)
- korzeń o mniejszej randze dołączamy do korzenia o randze większej

Łączenie według rangi

```
MakeSet(x)
  T = new TreeNode
  T.parent = T
  T.elem = x
  T.rank = 0
  return T
```

Łączenie według rangi

`Union`(x, y)

$rx =$ `FindSet`(x)
$ry =$ `FindSet`(y)
if $rx.rank > ry.rank$ **then**
    $ry.parent = rx$
else
    $rx.parent = ry$
    if $rx.rank = ry.rank$ **then** $ry.rank = ry.rank + 1$
end if

Kompresja ścieżki

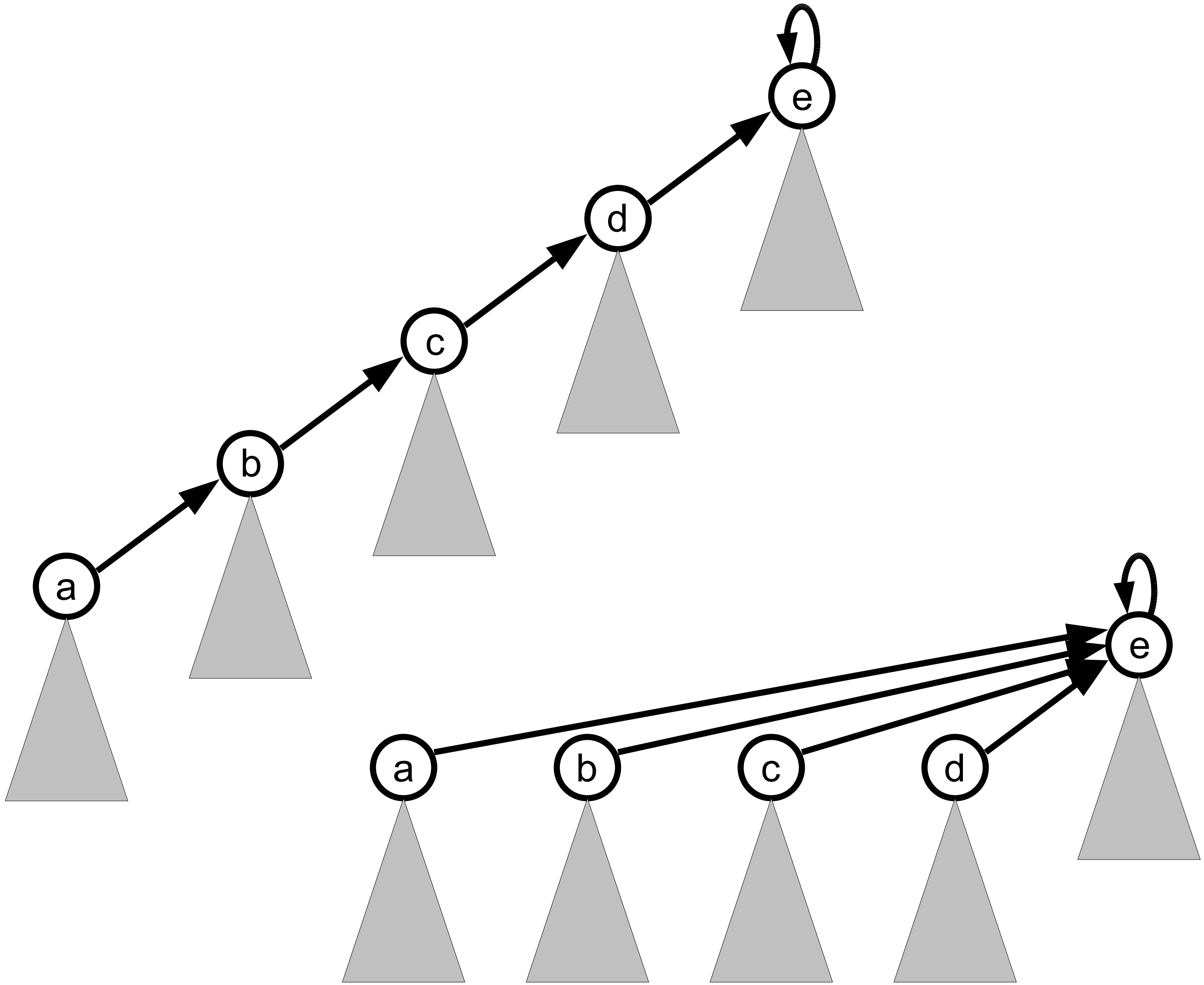
- kompresję ścieżki wykonujemy w trakcie działania procedury `FindSet`
- szukając reprezentanta zbioru zawierającego węzeł x, przechodzimy do rodzica, dopóki nie dojdziemy do korzenia
- po znalezieniu reprezentanta wracamy po scieżce, którą przeszliśmy, każdemu z węzłów ustawiając jako nowego rodzica reprezentanta

Kompresja ścieżki

`FindSet`(x)

if $x.parent \neq x$ **then**
    $x.parent =$ `FindSet`($x.parent$)
end if
return $x.parent$

Kompresja ścieżki



Lasy zbiorów rozłącznych

- wykorzystując te dwie heurystyki otrzymujemy asymptotycznie najefektywniejszą strukturę do reprezentacji zbiorów rozłącznych
- ciąg m operacji `MakeSet`, `Union` i `FindSet`, spośród których n to operacje `MakeSet` zajmuje $O(m\alpha(m,n))$ gdzie $\alpha(m,n)$ jest bardzo wolno rosnącą odwrotnością funkcji Ackremanna

Lasy zbiorów rozłącznych

- w każdym wyobrażalnym zastosowaniu struktury danych dla zbiorów rozłącznych $\alpha(m, n) \leq 4$ — możemy zatem potraktować ten czas jako liniowy względem m
- samo łączenie według rangi daje czas $O(m \log n)$
- sama kompresja ścieżki daje czas $O(f \log_{(1+f/n)} n)$ gdy $f \geq n$ lub $O(n + f \log n)$ gdy $f < n$, gdzie f to liczba operacji `FindSet`

Zastosowania zbiorów rozłącznych

- wyszukiwanie spójnych składowych w grafie
- sprawdzanie, czy dodanie krawędzi do grafu utworzy cykl
- wyznaczanie minimalnych drzew spinających

Koniec

Dziękuję za uwagę.