

# Algorytmy i struktury danych

## Struktury złączalne, zbiory rozłączne

Krzysztof M. Ocetkiewicz  
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

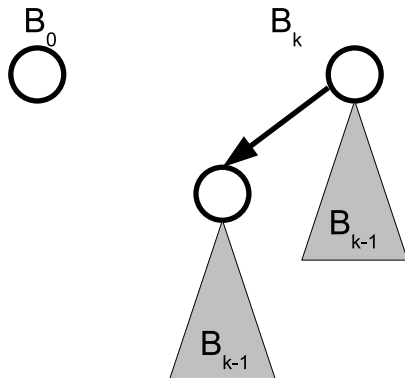
- $\text{MakeHeap}()$  — utworzenie pustego kopca
- $\text{Insert}(H, x)$  — wstawienie do kopca  $H$  klucza  $x$
- $\text{Minimum}(H)$  — znalezienie najmniejszego klucza w  $H$
- $\text{ExtractMin}(H)$  — znalezienie i usunięcie z kopca  $H$  najmniejszego klucza
- $\text{Union}(H_1, H_2)$  — utworzenie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

- 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

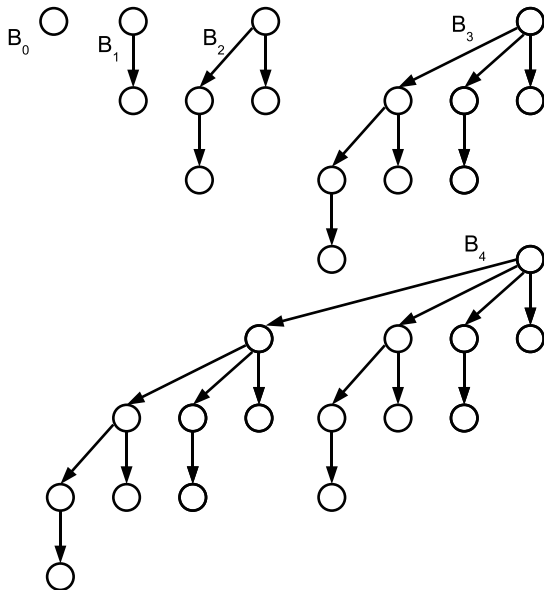
Operacja	kopiec	kopiec	kopiec	
	binarny (pesym.)	dwumianowy (pesym.)	Fibonacciego (zamort.)	(pesym.)
MakeHeap	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Minimum	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
ExtractMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Union	$O(n)$	$O(\log n)$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

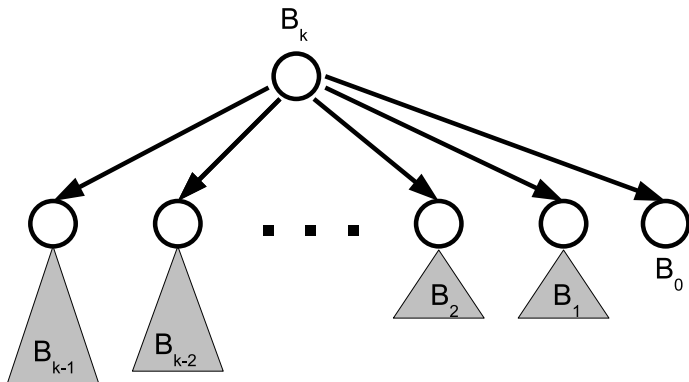
- $B_0$  to pojedynczy węzeł
- $B_k$  to dwa drzewa  $B_{k-1}$ , przy czym korzeń jednego z nich jest skrajnie lewym potomkiem korzenia drugiego drzewa



# Drzewa dwumianowe



# Drzewa dwumianowe

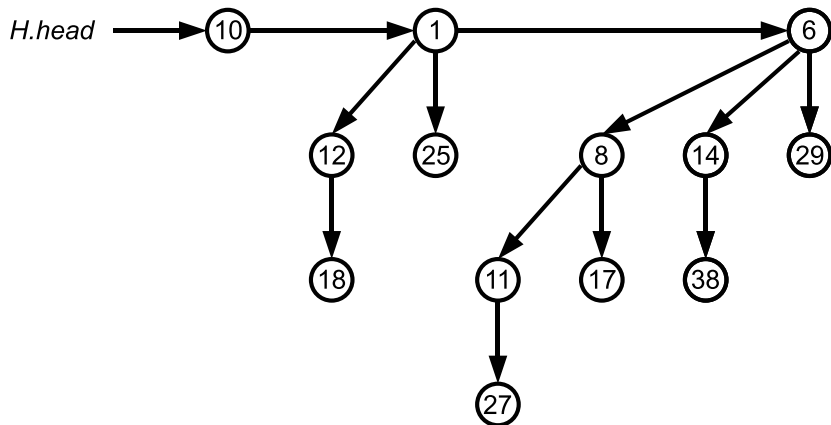


- w drzewie  $B_k$  jest  $2^k$  węzłów ( $B_k$  to dwa drzewa  $B_{k-1}$ )
- wysokość drzewa  $B_k$  wynosi  $k$  ( $B_k$  jest o 1 wyższe niż  $B_{k-1}$ )
- na głębokości  $i$  znajduje się  $\binom{k}{i}$  węzłów ( $i = 0, 1, \dots, k$ )
- stopień korzenia wynosi  $k$  i jest większy od stopnia każdego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



- 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 z co najwyżej  $\lfloor \log n \rfloor + 1$  drzew dwumianowych

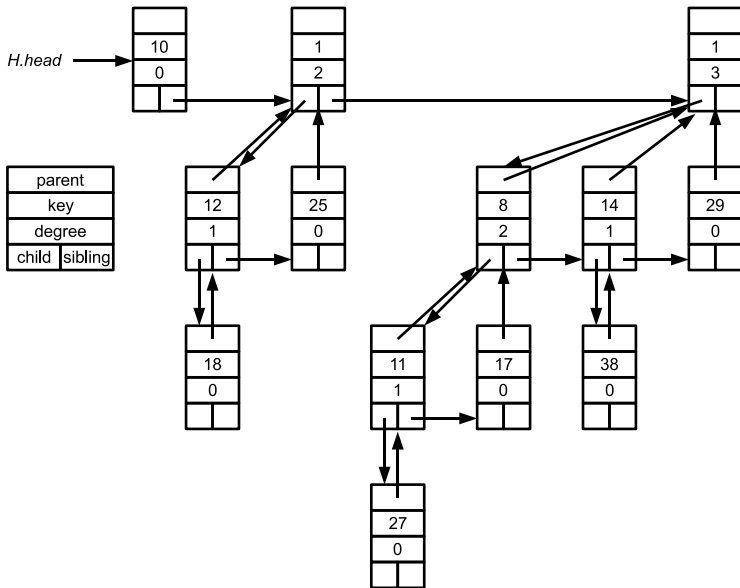
# Kopiec dwumianowy



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

- 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



- utworzenie nowego kopca wymaga jedynie utworzenia nowego wskaźnika (*head*) przypisania mu wartości *nullptr*

MakeHeap(*H*)

- 1: *H* = **new** BinomialHeap
- 2: *H.head* = *nullptr*

- minimalny klucz może znajdować się jedynie w jednym z korzeni

Minimum( $H$ )

```
1:  $w = \text{nullptr}$ 
2:  $t = H.\text{head}$ 
3:  $\text{min} = \infty$ 
4: while  $t \neq \text{nullptr}$  do
5:     if  $t.\text{key} < \text{min}$  then
6:          $\text{min} = t.\text{key}$ 
7:          $w = t$ 
8:     end if
9:      $t = t.\text{sibling}$ 
10: end while
11: return  $w$ 
```

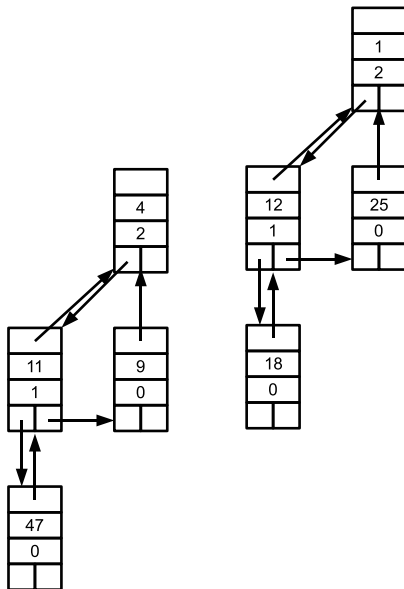
- pomocnicza procedura — dołączenie drzewa  $B_{k-1}$  (którego korzeniem jest  $y$ ) do drugiego drzewa  $B_{k-1}$  (o korzeniu  $z$ ) —  $z$  staje się w ten sposób korzeniem drzewa  $B_k$

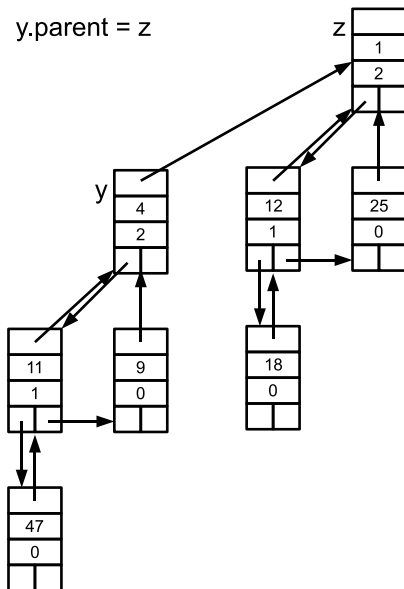
Link( $y, z$ )

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

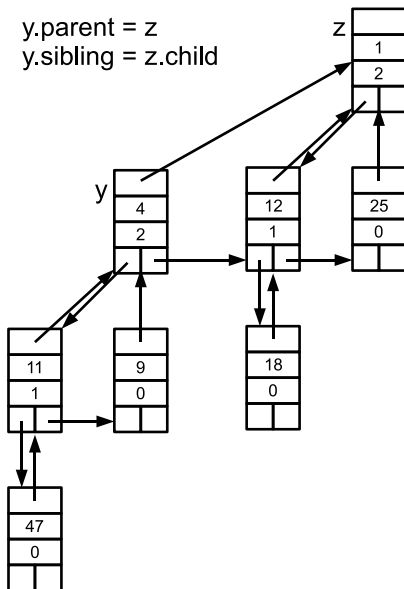


# Link

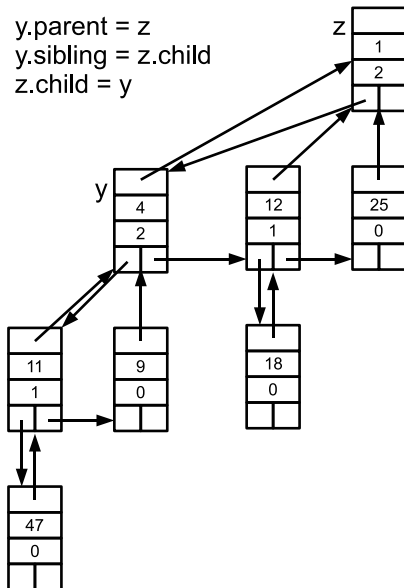




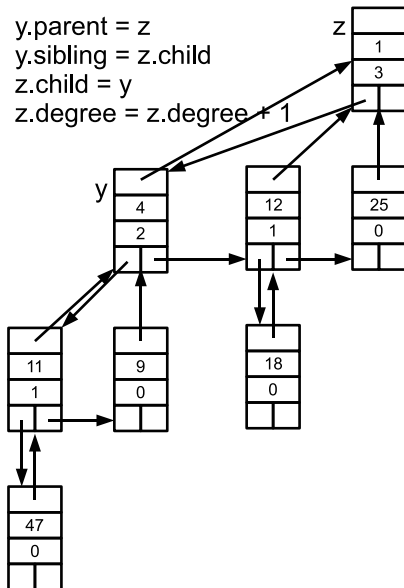
# Link



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



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



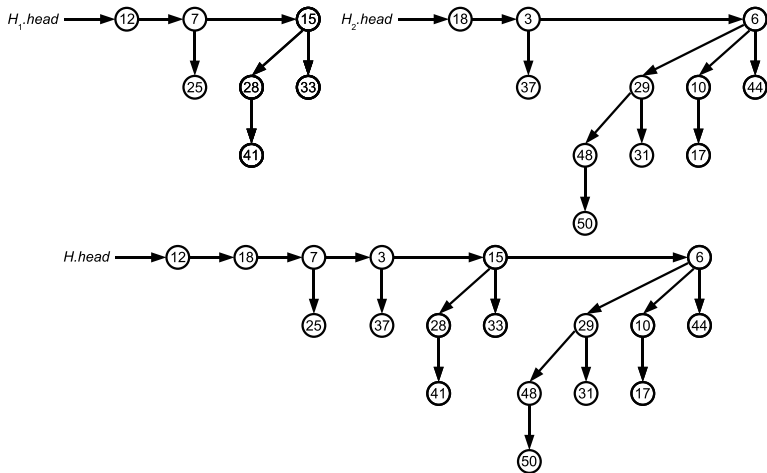
- większość pozostałych operacji na kopcu dwumianowym wykorzystuje procedurę `Union`
- operacja `Union` wykonuje dwie fazy
  - scalenie list korzeni kopców  $H_1$  i  $H_2$  w pojedynczą listę  $H$  uporządkowaną niemalejąco według stopni; na liście wynikowej mogą znaleźć się co najwyżej dwa korzenie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

# Merge

Merge( $H_1, H_2$ )

```
1:  $a = H_1.head$ 
2:  $b = H_2.head$ 
3: if  $a.degree < b.degree$  then  $head = a$  else  $head = b$ 
4:  $tail = head$ 
5: while  $a \neq nullptr$  or  $b \neq nullptr$  do
6:     if  $a.degree < b.degree$  then
7:          $tail.sibling = a$ 
8:          $a = a.sibling$ 
9:     else
10:         $tail.sibling = b$ 
11:         $b = b.sibling$ 
12:    end if
13:     $tail = tail.sibling$ 
14: end while
15: if  $a \neq nullptr$  then  $tail.sibling = a$ 
16: else if  $b \neq nullptr$  then  $tail.sibling = b$ 
17: else  $tail.sibling = nullptr$ 
18: return  $head$ 
```

# Merge





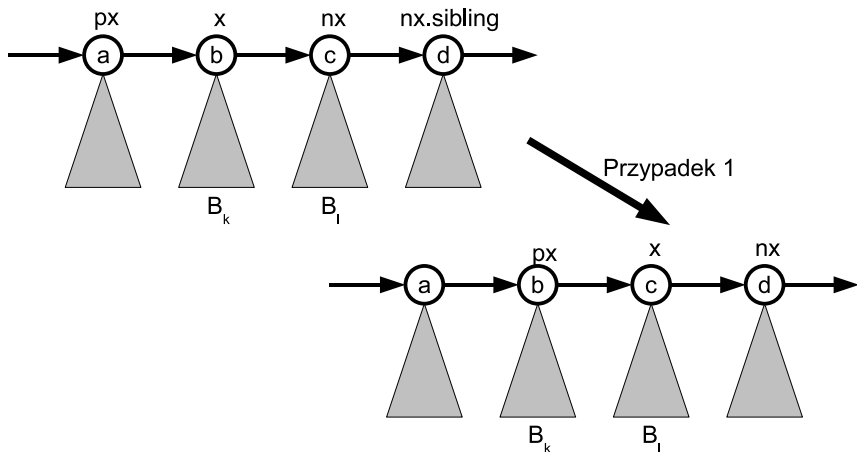
- złożoność operacji Merge wynosi  $O(m_1 + m_2)$ , gdzie  $m_1$  i  $m_2$  to długości list korzeni  $H_1$  i  $H_2$  (a te rosną logarytmicznie względem liczby węzłów)

- 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 poprzedni 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 łączenia
- przesuujemy się do następnego korzenia

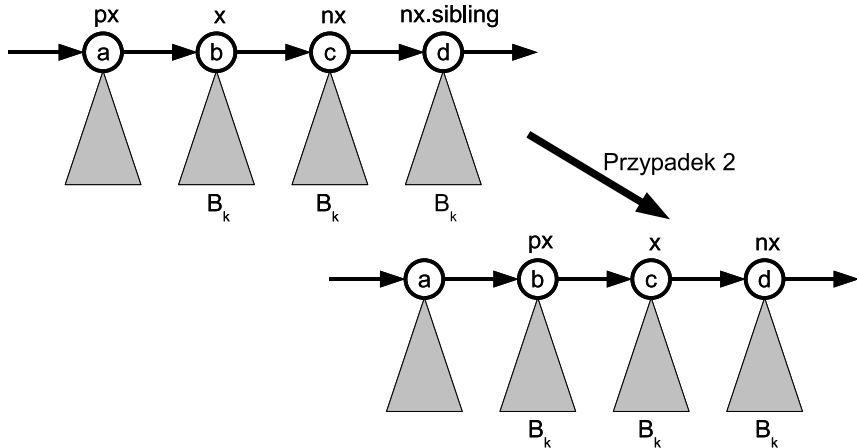
# Przypadek 1



## Przypadek 2

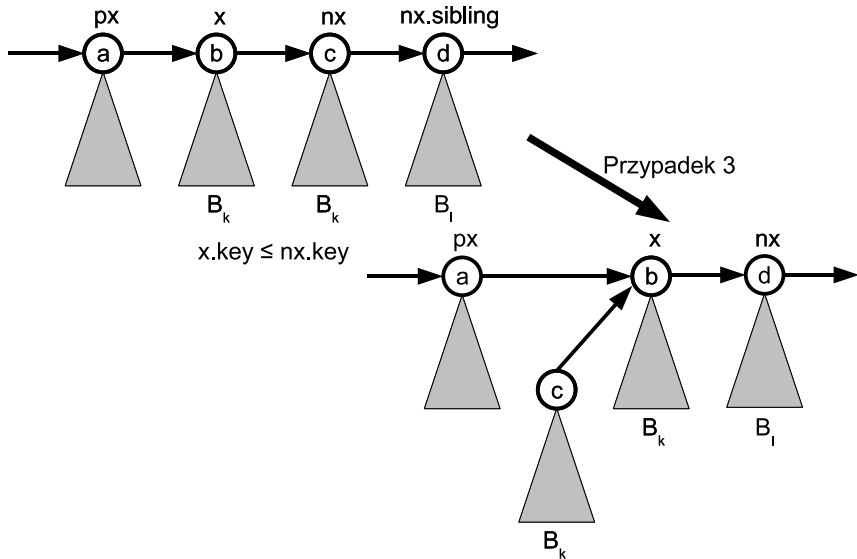
- $x$  jest pierwszym z trzech korzeni o takich samych stopniach
- taka sytuacja może się zdarzyć, gdy w wyniku scalenia list otrzymaliśmy ciąg drzew o stopniach  $k, k, k + 1, k + 1$  a następnie połączyliśmy pierwsze dwa drzewa
- postępujemy tak, jak w pierwszym przypadku — przesuwamy się do następnego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

# Przypadek 2



- $x$  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

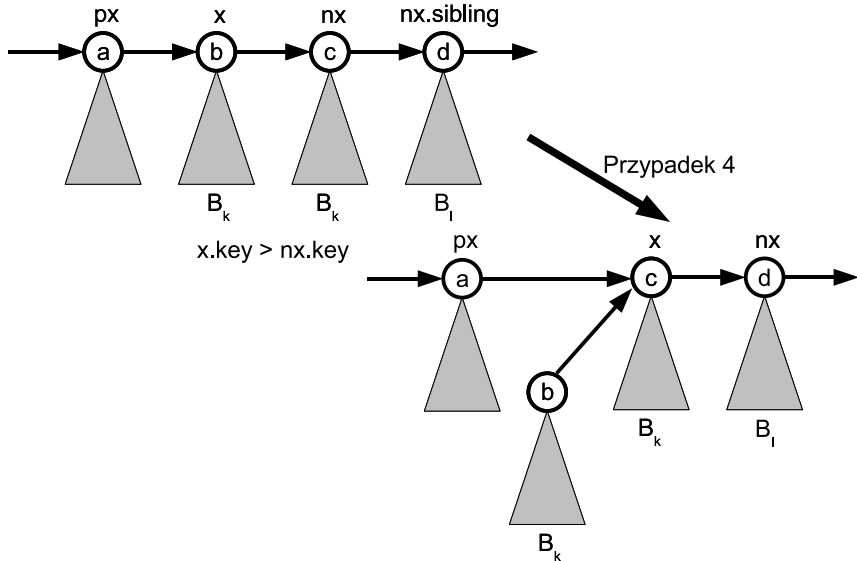




## Przypadek 4

- $x$  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



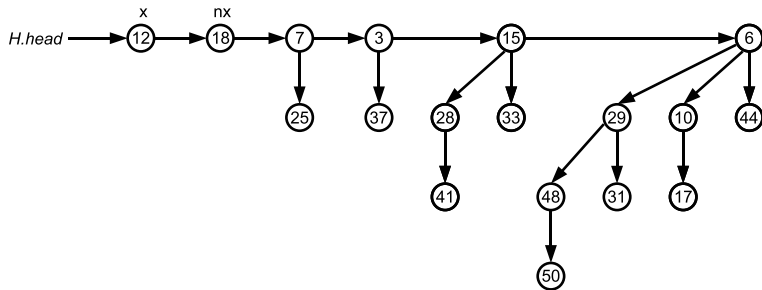
- złożoność drugiej fazy również wynosi  $O(\log n)$
- liczba wykonanych kroków jest równa długości listy  $H$  — w każdym kroku albo przechodzimy do następnego węzła, albo skracamy o jeden długość listy pozostałej do przejścia
- w każdym kroku wykonujemy stałą liczbę operacji

# Union

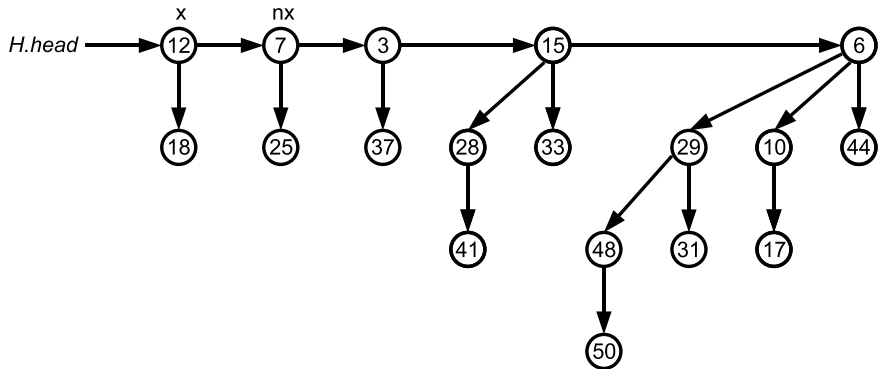
Union( $H_1, H_2$ )

```
1:  $H = \text{MakeHeap}()$ 
2:  $H.\text{head} = \text{HeapMerge}(H_1, H_2)$ 
3: usuń z pamięci  $H_1$  i  $H_2$  (pozostawiając jednak listy na które wskazują)
4: if  $H.\text{head} = \text{nullptr}$  then return  $H$ 
5:  $px = \text{nullptr}$ 
6:  $x = H.\text{head}$ 
7:  $nx = x.\text{sibling}$ 
8: while  $nx \neq \text{nullptr}$  do
9:     if  $x.\text{degree} \neq nx.\text{degree}$  or ( $nx.\text{sibling} \neq \text{nullptr}$  and  $x.\text{degree} = x.\text{sibling}.\text{degree}$ ) then
10:          $px = x$ 
11:          $x = nx$ 
12:     else
13:         if  $x.\text{key} \leq nx.\text{key}$  then
14:              $x.\text{sibling} = nx.\text{sibling}$ 
15:              $\text{Link}(nx, x)$ 
16:         else
17:             if  $px = \text{nullptr}$  then  $H.\text{head} = nx$  else  $px.\text{sibling} = nx$ 
18:              $\text{Link}(x, nx)$ 
19:              $x = nx$ 
20:         end if
21:     end if
22:      $nx = x.\text{sibling}$ 
23: end while
24: return  $H$ 
```

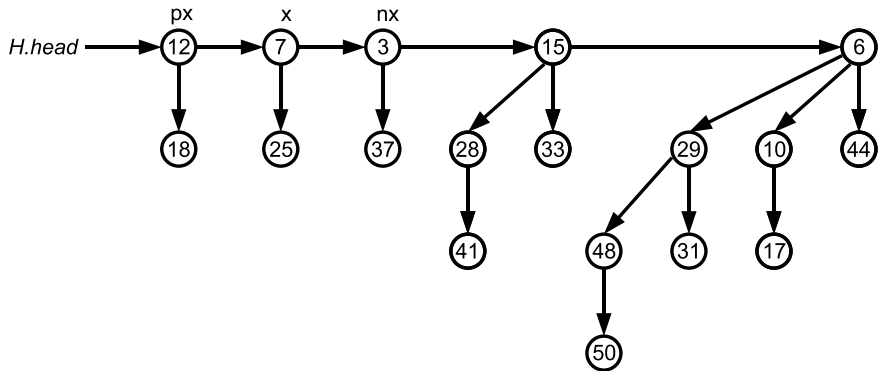
# Union

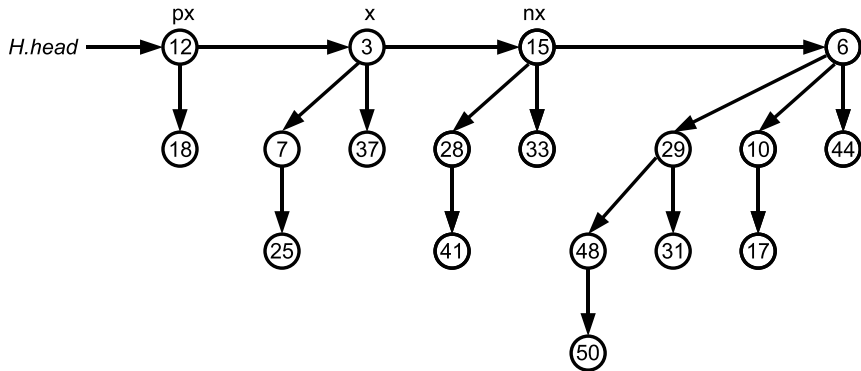


# Union



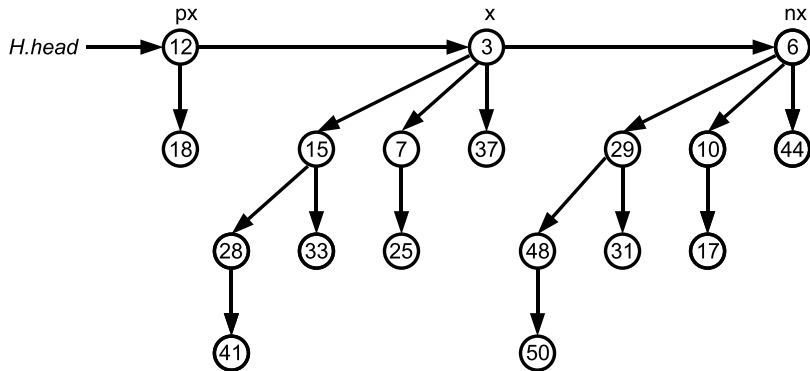
# Union



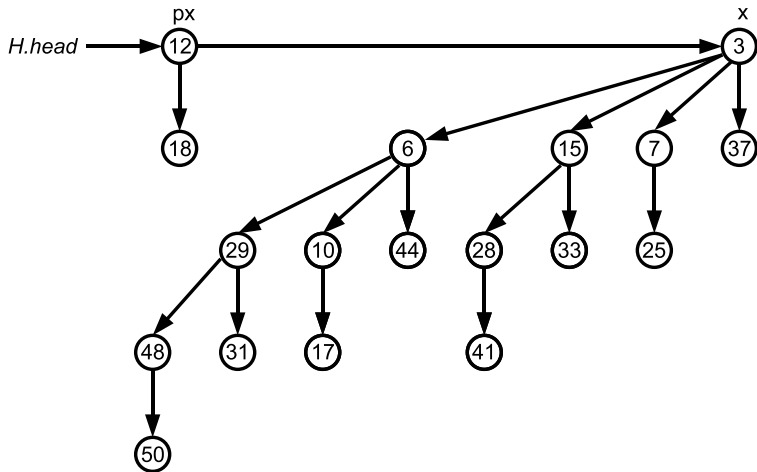




# Union



# Union



- 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( $H, x$ )

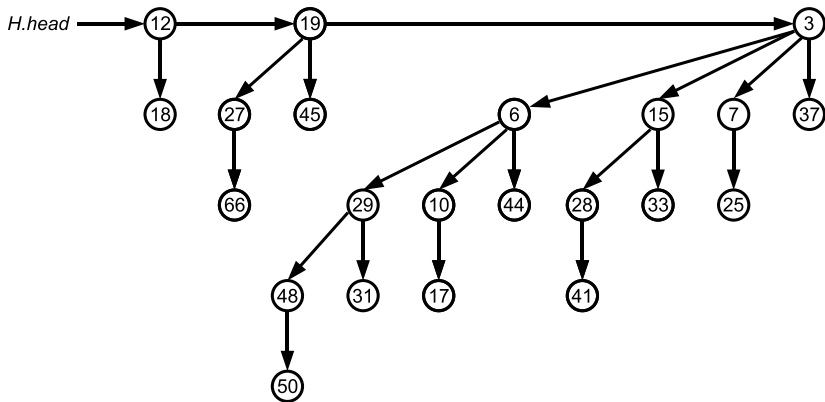
- 1:  $H' = \text{MakeHeap}()$
- 2:  $x.\text{parent} = \text{nullptr}$
- 3:  $x.\text{sibling} = \text{nullptr}$
- 4:  $x.\text{child} = \text{nullptr}$
- 5:  $x.\text{degree} = 0$
- 6:  $H'.\text{head} = x$
- 7:  $H = \text{Union}(H, H')$
- 8: **return**  $H$

- znajdujemy i usuwamy z listy korzeni w  $H$  korzeń o najmniejszym kluczu
- potomkami węzła  $x$  są drzewa o stopniach  $k, k - 1, \dots, 0$
- 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  $H$

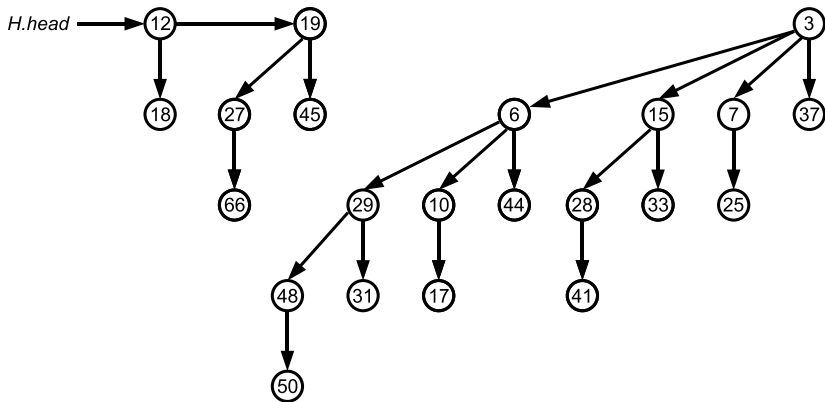
## ExtractMin( $H$ )

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

# ExtractMin

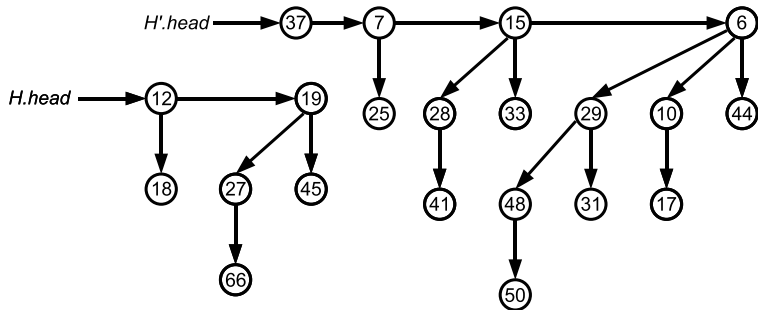


# ExtractMin

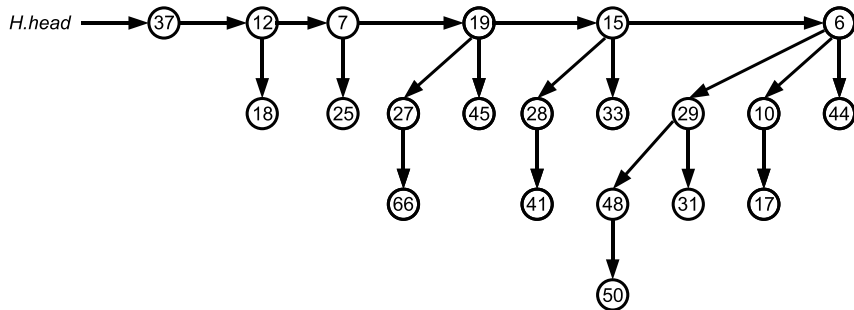




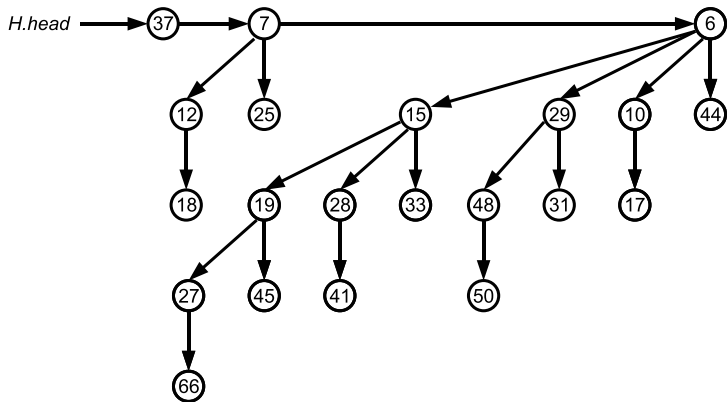
# ExtractMin



# ExtractMin



# ExtractMin



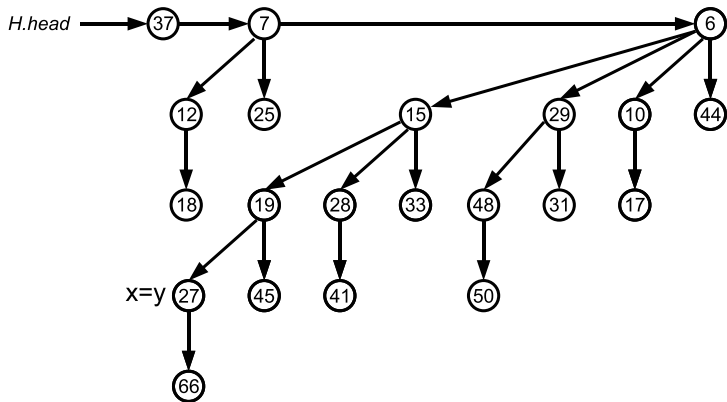
- operacja ta wymaga wskaźnika na węzeł  $x$  zawierający klucz, którego wartość należy zmniejszyć
- po zmianie wartości klucza należy przywrócić drzewu zawierającemu  $x$  własność kopca (analogicznie jak `Heapify` w zwykłym kopcu)

# DecreaseKey

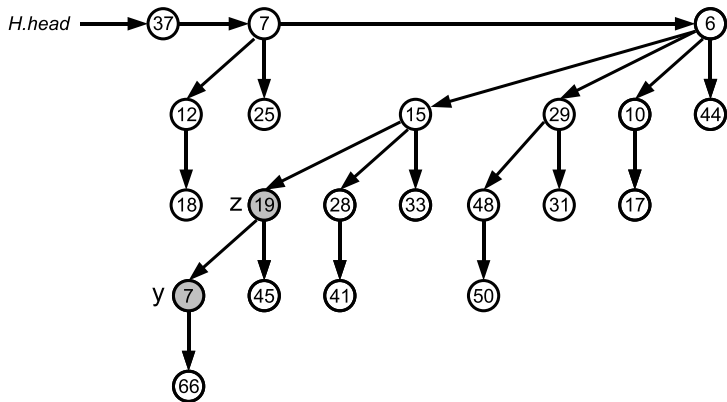
DecreaseKey( $H, x, k$ )

- 1: **if**  $x.key \leq k$  **then error** “zwiększenie wartości klucza”
- 2:  $x.key = k$
- 3:  $y = x$
- 4:  $z = y.parent$
- 5: **while**  $z \neq nullptr$  **and**  $y.key < z.key$  **do**
- 6:     zamień  $y.key$  z  $z.key$
- 7:     zamień wartości dodatkowych danych z węzłów  $y$  i  $z$
- 8:      $y = z$
- 9:      $z = y.parent$
- 10: **end while**

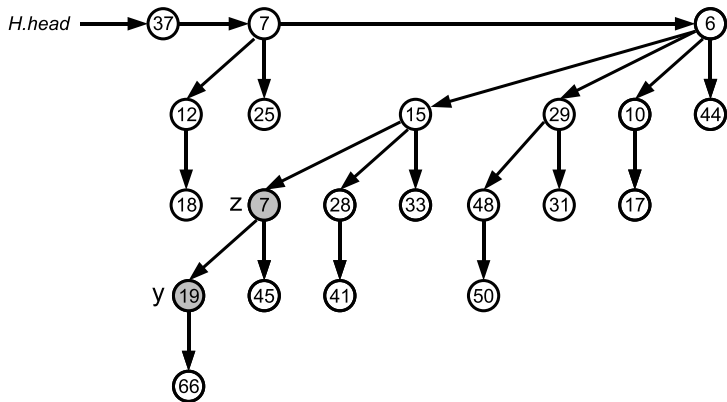
# DecreaseKey



# DecreaseKey

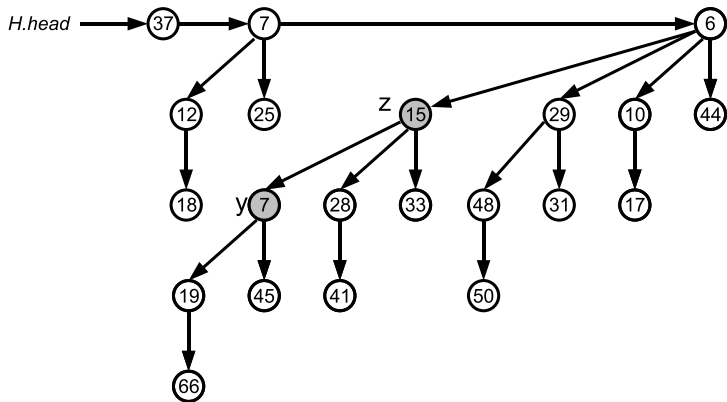


# DecreaseKey

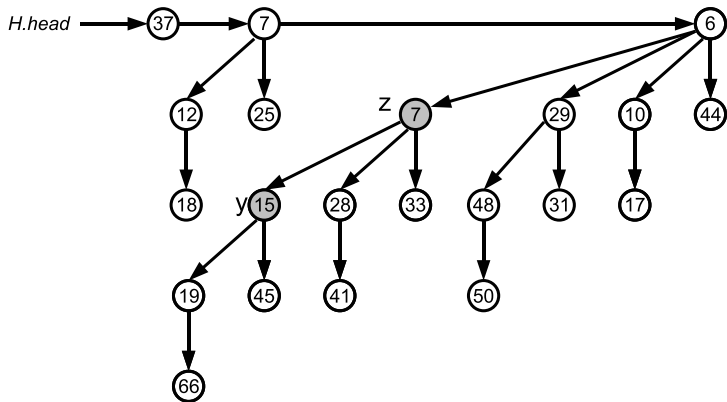




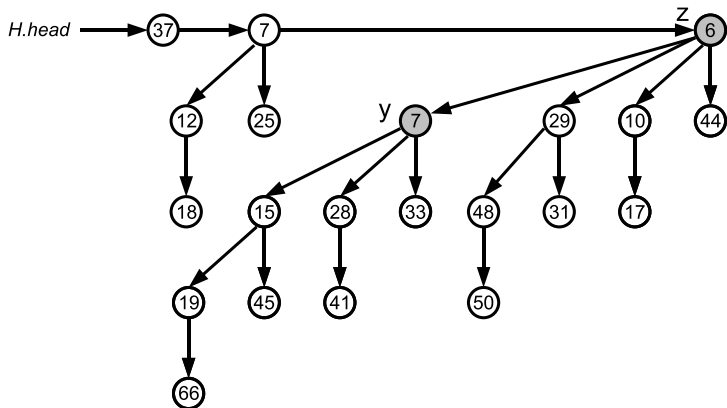
# DecreaseKey



# DecreaseKey



# DecreaseKey



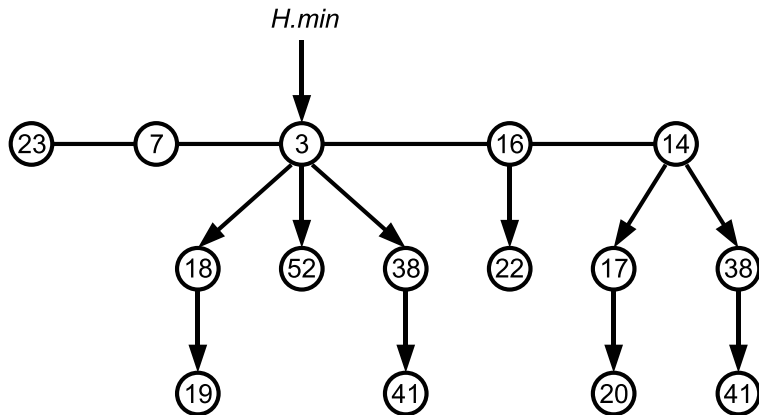
- również ta operacja wymaga wskaźnika na węzeł  $x$  zawierający klucz, który należy usunąć
- zmniejszamy wartość usuwanego klucza do  $-\infty$
- usuwamy z kopca najmniejszy klucz (`ExtractMin`)

`Delete( $H, x$ )`

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

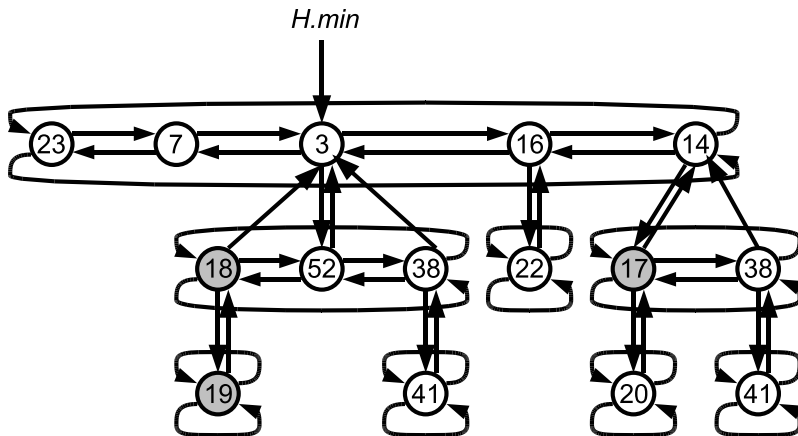
- jest zbiorem drzew, z których każde ma własność kopca
- drzewa nie muszą być drzewami dwumianowymi
- jeżeli ograniczymy się do operacji MakeHeap, Insert, Minimum, ExtractMin, Union to kopiec Fibonacciego będzie zbiorem “nieuporządkowanych” drzew binarnych (węzły potomne nie muszą być uporządkowane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

# Kopiec Fibonacciego



- każdy węzeł zawiera:
  - *key* — klucz
  - *parent* — wskaźnik na rodzica (*nullptr* w korzeniu)
  - *child* — wskaźnik na któregokolwiek potomka (*nullptr* 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





- dostęp do kopca zapewnia wskaźnik *min* — jest to wskaźnik do korzenia drzewa zawierającego najmniejszy klucz (węzeł minimalny kopca)
- jeżeli kopiec *H* jest pusty,  $H.min = \text{nullptr}$
- 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  $H.n$ )

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

MakeHeap( $H$ )

- 1:  $H = \mathbf{new}$  FibonacciHeap
- 2:  $H.min = \mathit{nullptr}$
- 3:  $H.n = 0$

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

Minimum(*H*)

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

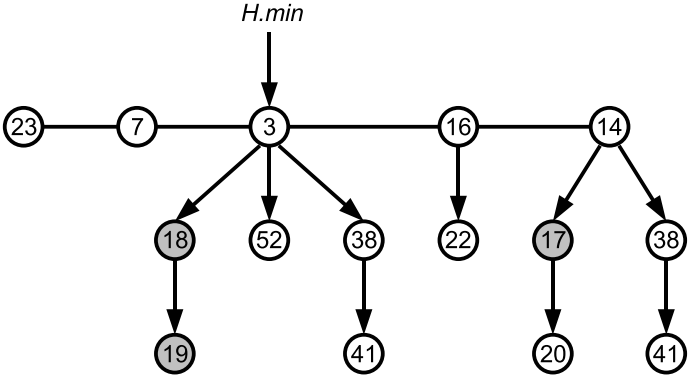
- $x$  jest węzłem do wstawienia ( $x.key$  jest już odpowiednio wypełnione)
- dodajemy  $x$  do listy korzeni kopca  $H$
- jeżeli trzeba, poprawiamy wskaźnik  $H.min$
- zwiększamy licznik węzłów w  $H$
- nie próbujemy sklejać drzew —  $k$  kolejnych wstawień wstawi do listy korzeni  $k$  jednoelementowych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

# Insert

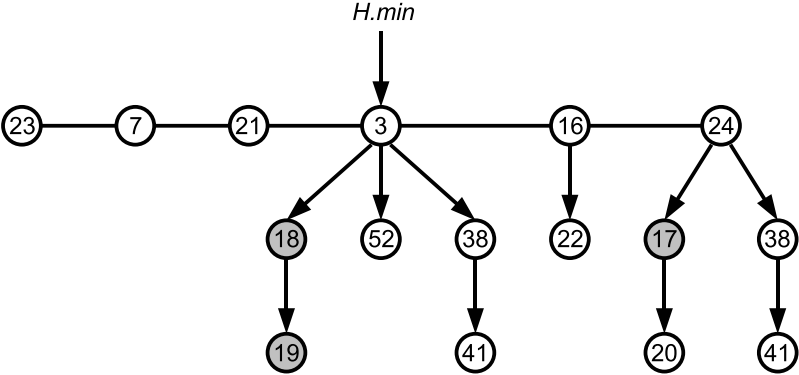
Insert( $H, x$ )

- 1:  $x.degree = 0$
- 2:  $x.parent = nullptr$
- 3:  $x.left = x$
- 4:  $x.right = x$
- 5:  $x.child = nullptr$
- 6:  $x.mark = FALSE$
- 7: połącz jednoelementową listę  $x$  z listą korzeni  $H$
- 8: **if**  $H.min = nullptr$  **or**  $x.key < H.min.key$  **then**
- 9:      $H.min = x$
- 10: **end if**
- 11:  $H.n = H.n + 1$

# Insert



# Insert



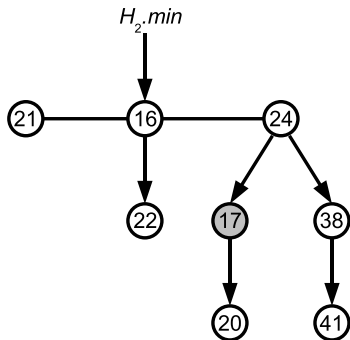
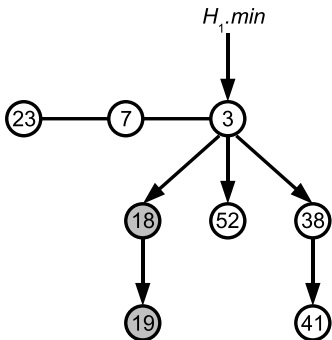
- operacja scalenia kopców jest podobna do wstawiania pojedynczego klucza (tyle, że wstawiamy listę dłuższą niż jeden element)
- ponownie sklejanie drzew odkładamy na później



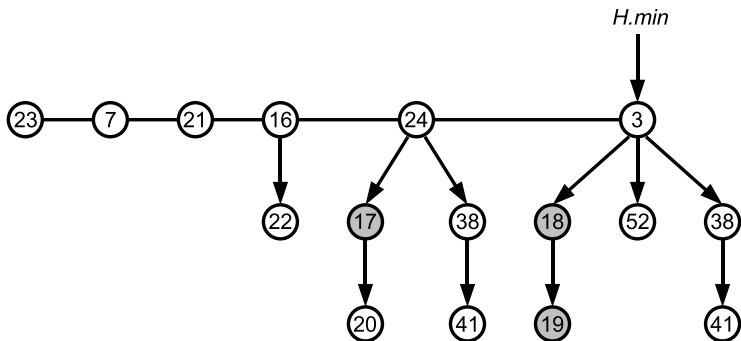
Union( $H_1, H_2$ )

- 1:  $H = \text{MakeHeap}()$
- 2:  $H.min = H_1.min$
- 3: sklej listę korzeni  $H_2$  z listą korzeni  $H_1$
- 4: **if**  $H_1.min = \text{nullptr}$  **or**  
     $(H_2.min \neq \text{nullptr} \text{ and } H_2.min < H_1.min)$  **then**
- 5:      $H.min = H_2.min$
- 6: **end if**
- 7:  $H.n = H_1.n + H_2.n$
- 8: zwolnij pamięć przydzieloną obiektom  $H_1$  i  $H_2$   
    (ale nie zwalniasz ich list korzeni)
- 9: **return**  $H$

# Union



# Union

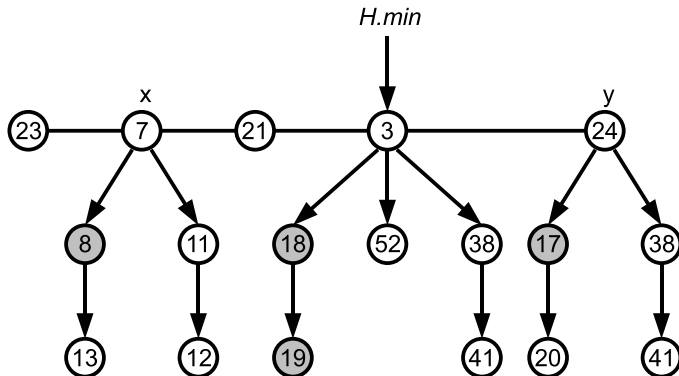


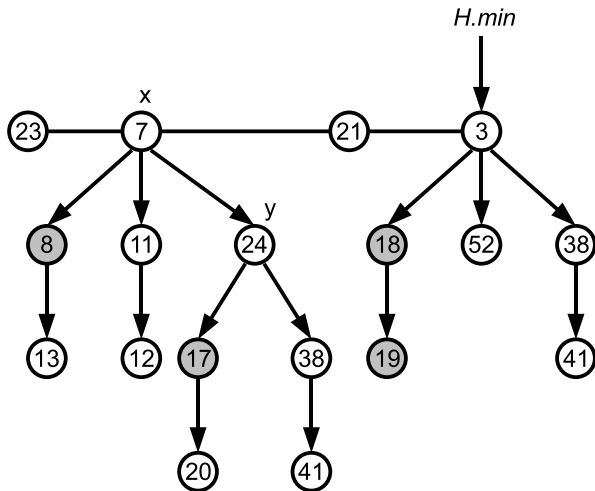
- w tej operacji wykonywana jest praca okładana we wstawianiu i scalaniu
- korzystamy z pomocniczej procedury Consolidate, która skraca listę korzeni
- Consolidate z kolei korzysta z pomocniczej procedury Link, łączącej dwa drzewa o takim samym stopniu

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

$\text{Link}(H, y, x)$

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





- pomocnicza procedura — skracanie listy korzeni
- dopóki każdy korzeń na liście nie ma innego stopnia:
  - znajdź 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  $y.degree = i$  ( $i = 0, \dots, \lfloor \log n \rfloor + 1$ )



- tablica  $A$  przechowuje wskaźniki do “znanych” drzew o danych stopniach (co najwyżej po jednym na dany stopień)
- 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( $H$ )

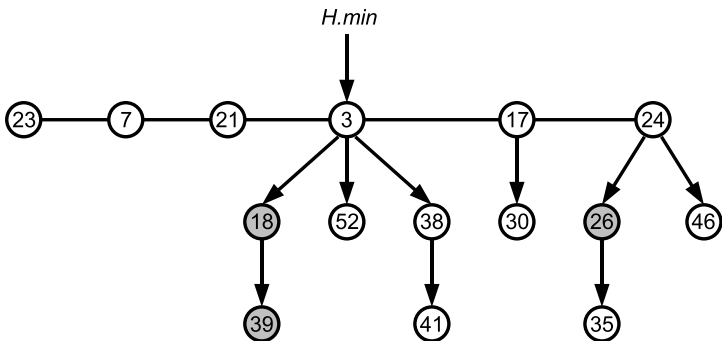
```
1: for  $i = 0, \dots, \lfloor \log H.n \rfloor + 1$  do  $A[i] = \text{nullptr}$ 
2: for każdy węzeł  $w$  na liście korzeni  $H$  do
3:      $x = w$ 
4:      $d = x.\text{degree}$ 
5:     while  $A[d] \neq \text{nullptr}$  do
6:          $y = A[d]$ 
7:         if  $x.\text{key} > y.\text{key}$  then zamień  $x$  z  $y$ 
8:          $\text{Link}(H, y, x)$ 
9:          $A[d] = \text{nullptr}$ 
10:         $d = d + 1$ 
11:    end while
12:     $A[d] = x$ 
13: end for
14:  $H.\text{min} = \text{nullptr}$ 
15: for  $i = 0, \dots, \lfloor \log n \rfloor + 1$  do
16:     if  $A[i] \neq \text{nullptr}$  then
17:         dodaj  $A[i]$  do listy korzeni  $H$ 
18:         if  $H.\text{min} = \text{nullptr}$  or  $A[i].\text{key} < H.\text{min}.\text{key}$  then  $H.\text{min} = A[i]$ 
19:     end if
20: end for
```

# ExtractMin

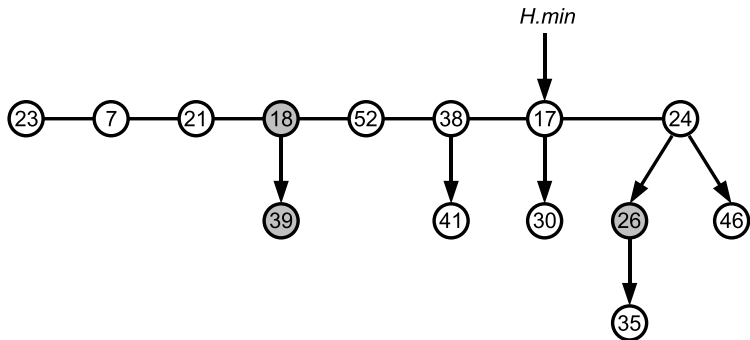
ExtractMin( $H$ )

```
1:  $z = H.min$ 
2: if  $z \neq nullptr$  then
3:     for każdy potomek  $x$  węzła  $z$  do
4:         dodaj  $x$  do listy korzeni  $H$ 
5:          $x.parent = nullptr$ 
6:     end for
7:      $zr = z.right$ 
8:     usuń  $z$  z listy korzeni  $H$ 
9:     if  $z = zr$  then
10:         $H.min = nullptr$ 
11:    else
12:         $H.min = zr$ 
13:        Consolidate( $H$ )
14:    end if
15:     $H.n = H.n - 1$ 
16: end if
17: return  $z$ 
```

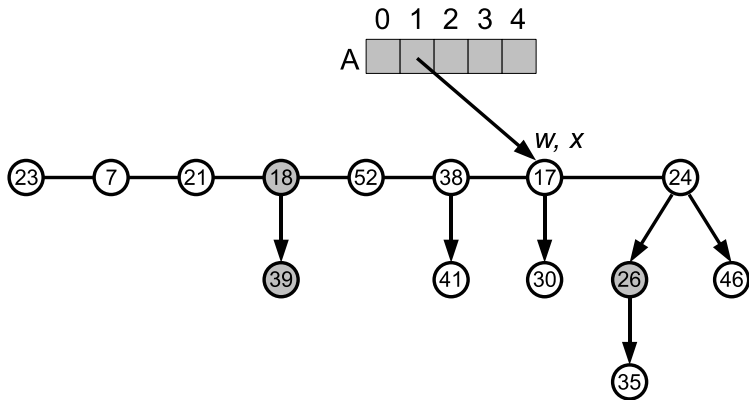
# ExtractMin



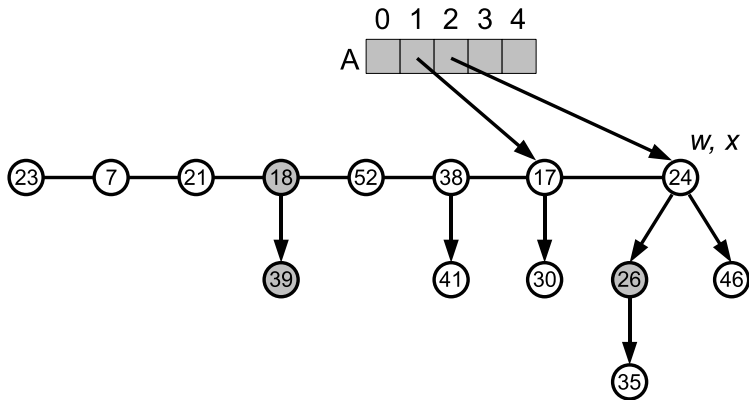
# ExtractMin



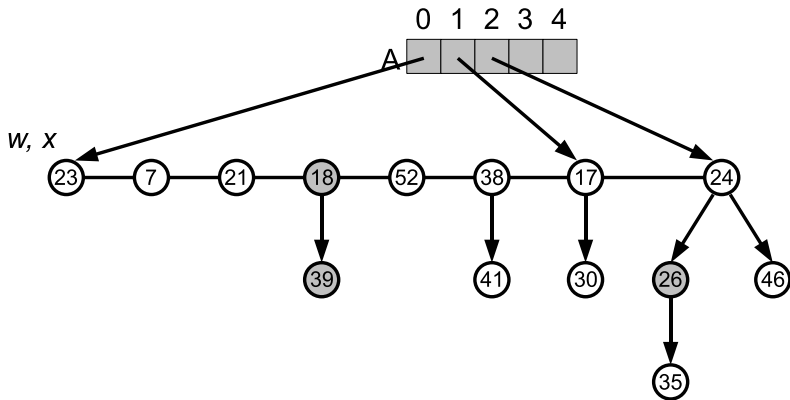
# ExtractMin



# ExtractMin

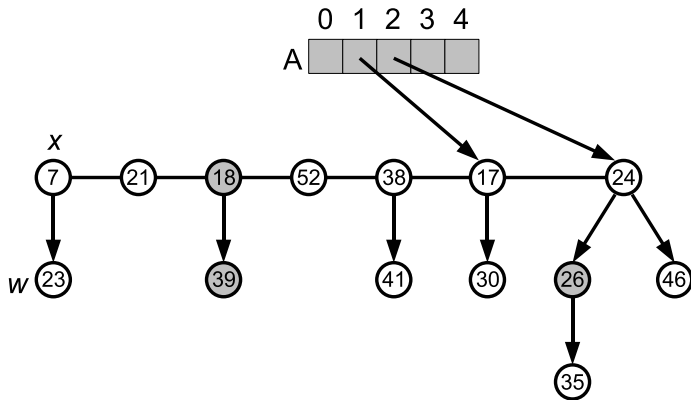


# ExtractMin

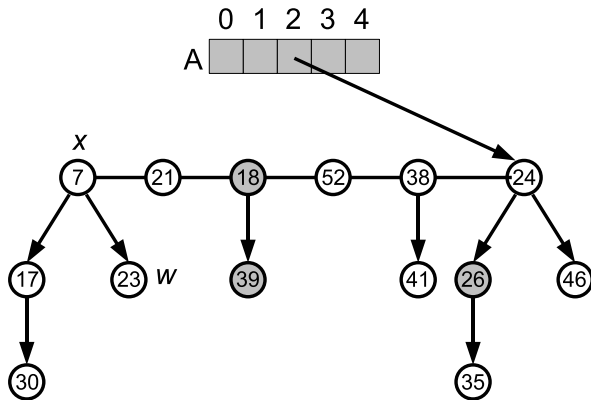




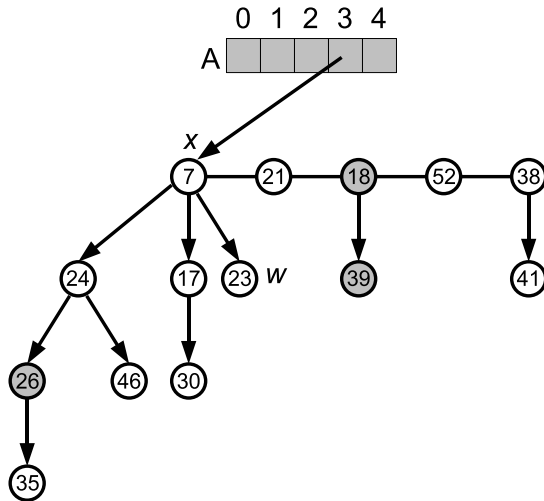
# ExtractMin



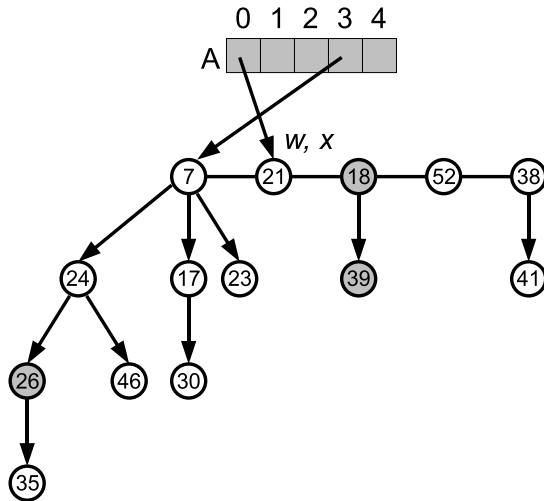
# ExtractMin



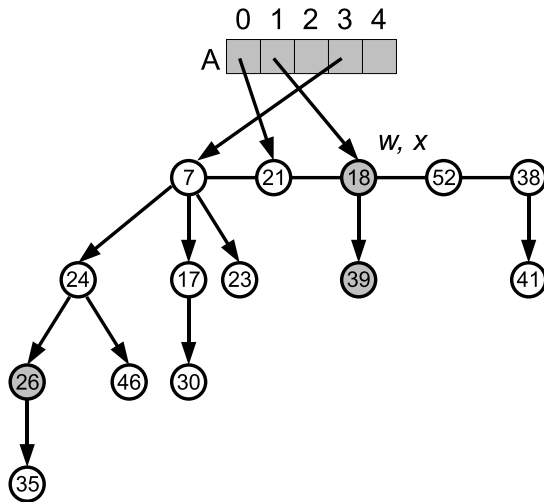
# ExtractMin



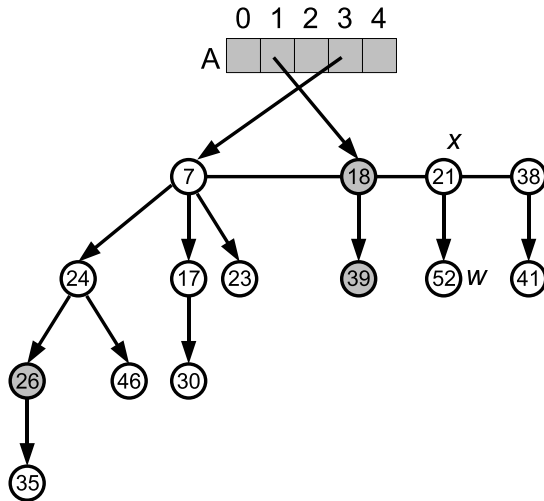
# ExtractMin



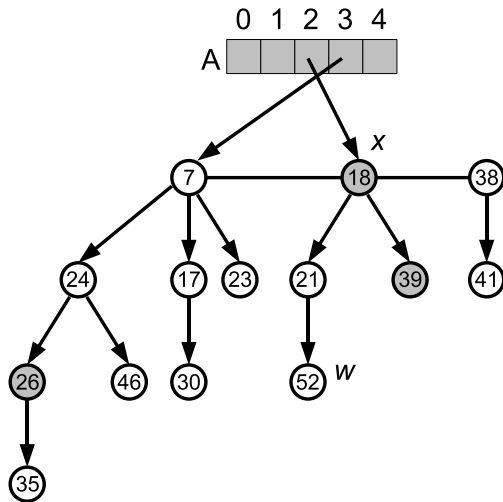
# ExtractMin



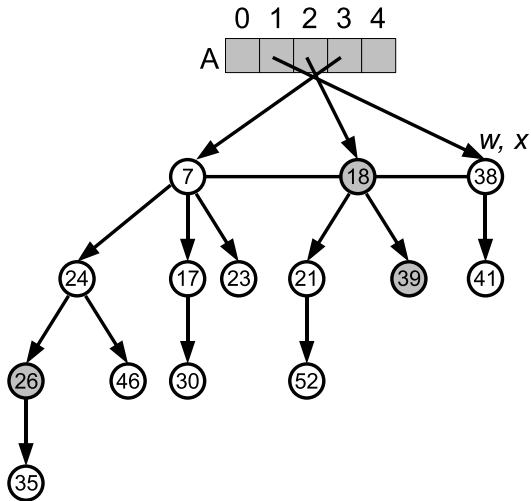
# ExtractMin



# ExtractMin

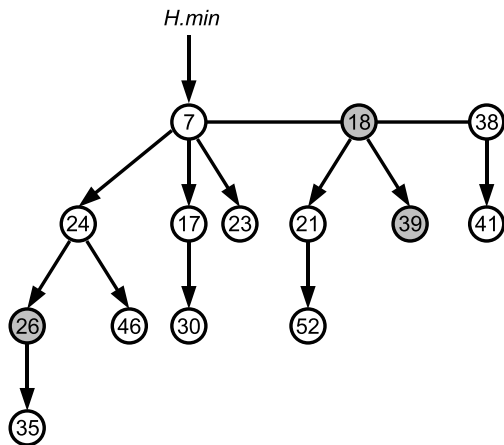


# ExtractMin





# ExtractMin



- 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

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

# DecreaseKey

DecreaseKey( $H, x, k$ )

- 1: **if**  $x.key \leq k$  **then error** “zwiększenie wartości klucza”
- 2:  $x.key = k$
- 3:  $y = x.parent$
- 4: **if**  $y \neq nullptr$  **and**  $x.key < y.key$  **then**
- 5:     Cut( $H, x, y$ )
- 6:     CascadingCut( $H, y$ )
- 7: **end if**
- 8: **if**  $x.key < H.min.key$  **then**
- 9:      $H.min = x$
- 10: **end if**

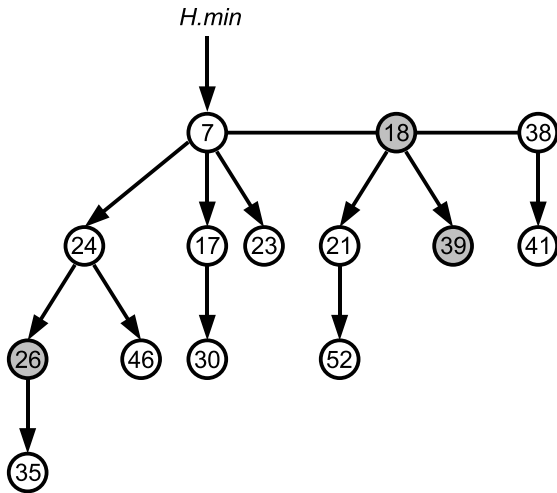
$\text{Cut}(H, x, y)$

- 1: usuń  $x$  z listy potomków  $y$  i zmniejsz  $y.degree$  o 1
- 2: dodaj  $x$  do listy korzeni  $H$
- 3:  $x.parent = \text{nullptr}$
- 4:  $x.mark = \text{FALSE}$

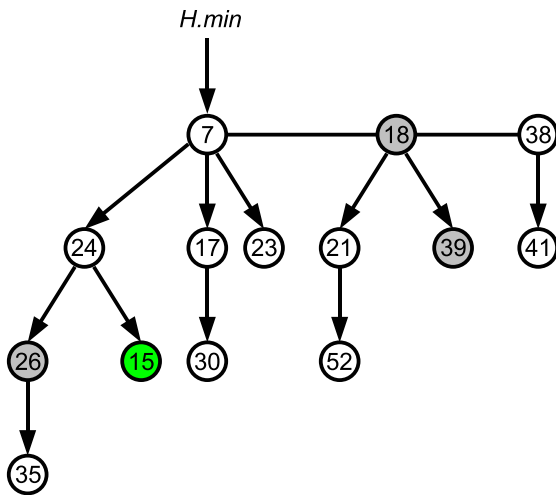
CascadingCut( $H, y$ )

```
1:  $z = y.parent$   
2: if  $z \neq nullptr$  then  
3:     if  $y.mark = FALSE$  then  
4:          $y.mark = TRUE$   
5:     else  
6:         Cut( $H, y, z$ )  
7:         CascadingCut( $H, z$ )  
8:     end if  
9: end if
```

# DecreaseKey

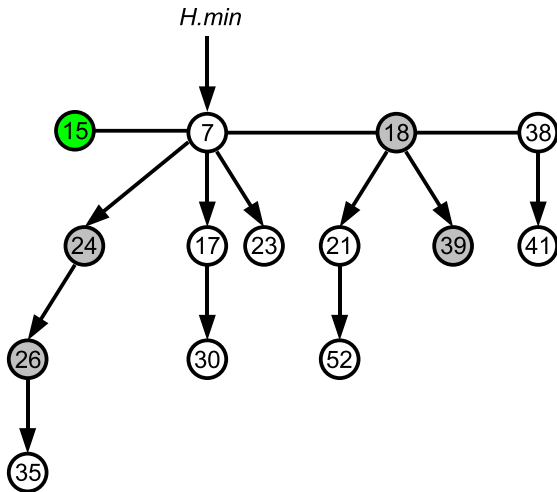


# DecreaseKey

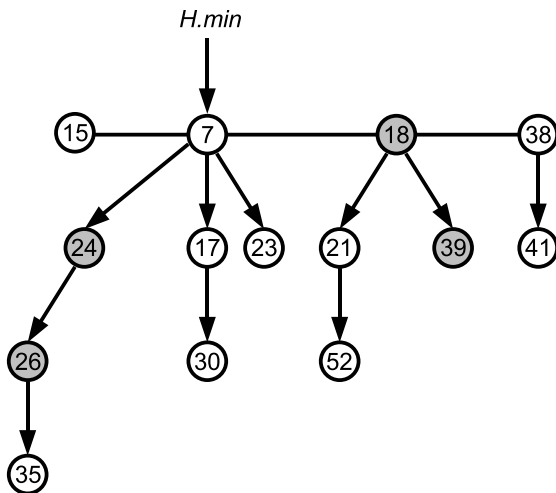




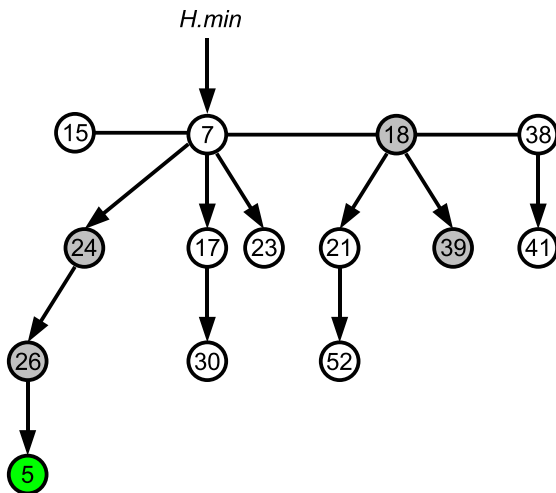
# DecreaseKey



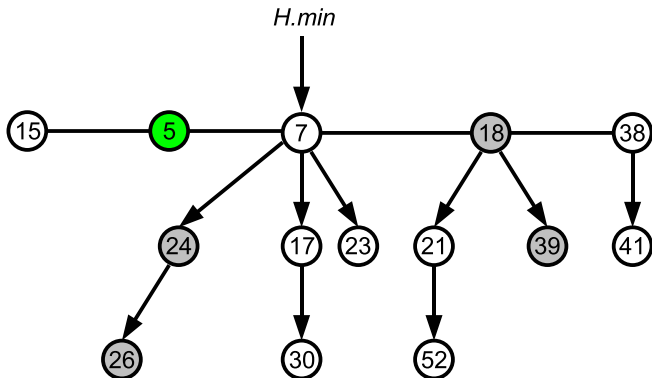
# DecreaseKey



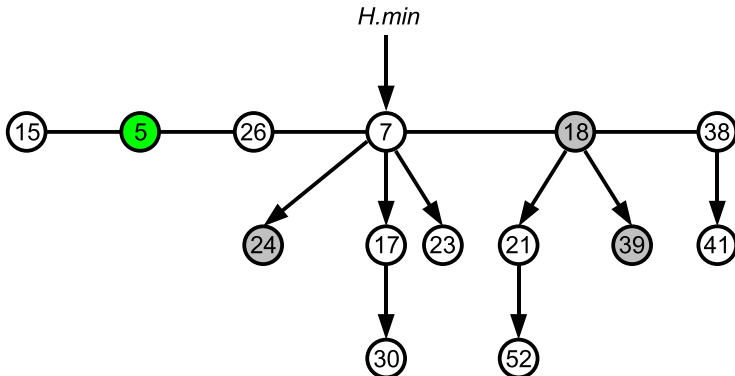
# DecreaseKey



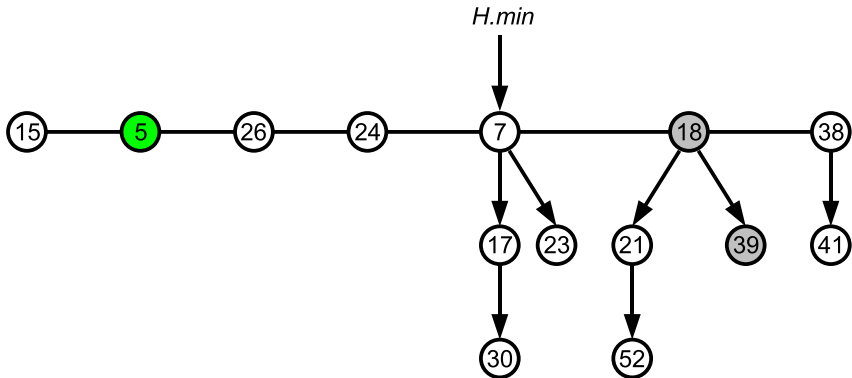
# DecreaseKey



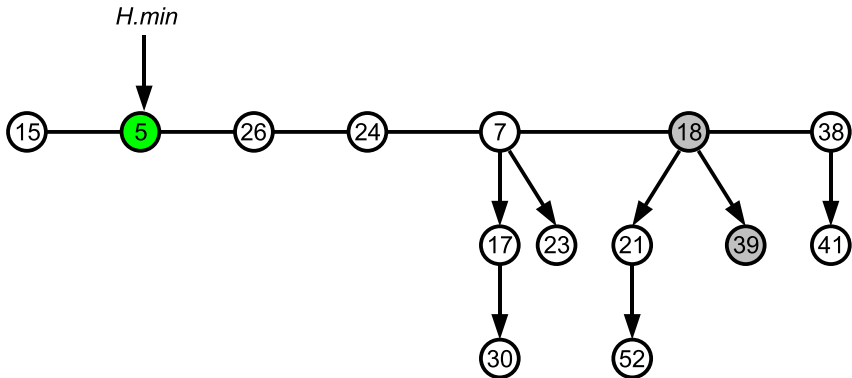
# DecreaseKey



# DecreaseKey



# DecreaseKey



- 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( $H$ )`



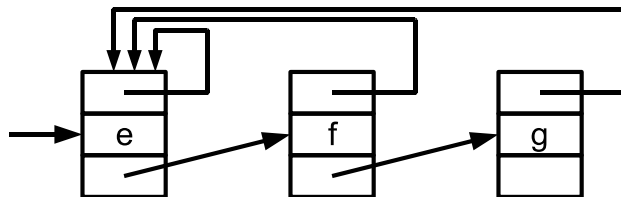
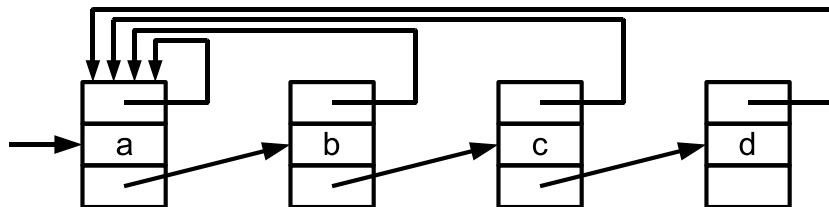
- mamy pewną liczbę elementów pogrupowanych w pewną liczbę zbiorów rozłącznych
- chcemy wiedzieć, do którego zbioru należy element
- chcemy mieć możliwość łączenia dwóch zbiorów

- 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 zmienił)

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

- 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



- operacja MakeSet wymaga tylko utworzenia jednoelementowej listy

MakeSet( $x$ )

- 1:  $L = \mathbf{new}$  ListNode
- 2:  $L.repr = L$
- 3:  $L.next = \mathit{nullptr}$
- 4:  $L.elem = x$
- 5: **return**  $L$

- operacja FindSet zwraca zawartość wskaźnika *repr* węzła

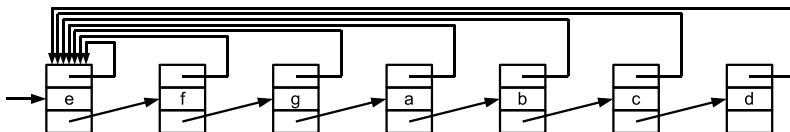
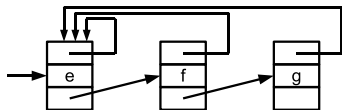
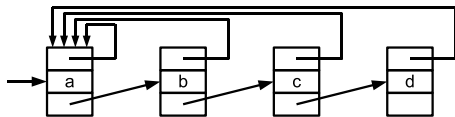
FindSet(*x*)

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

- 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



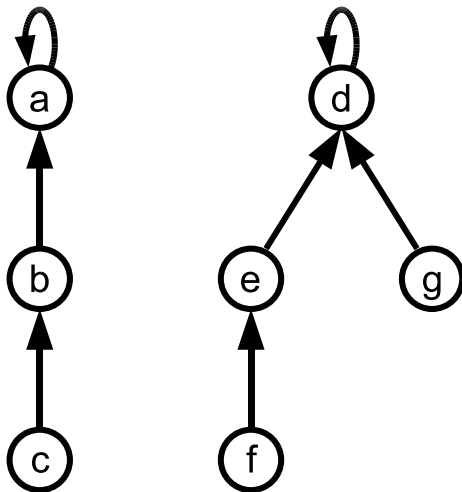
# Reprezentacja listowa



- 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

- 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



- operacja MakeSet wymaga tylko utworzenia jednoelementowego drzewa

MakeSet( $x$ )

- 1:  $T = \mathbf{new}$  TreeNode
- 2:  $T.parent = T$
- 3:  $T.elem = x$
- 4: **return**  $T$

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

FindSet( $x$ )

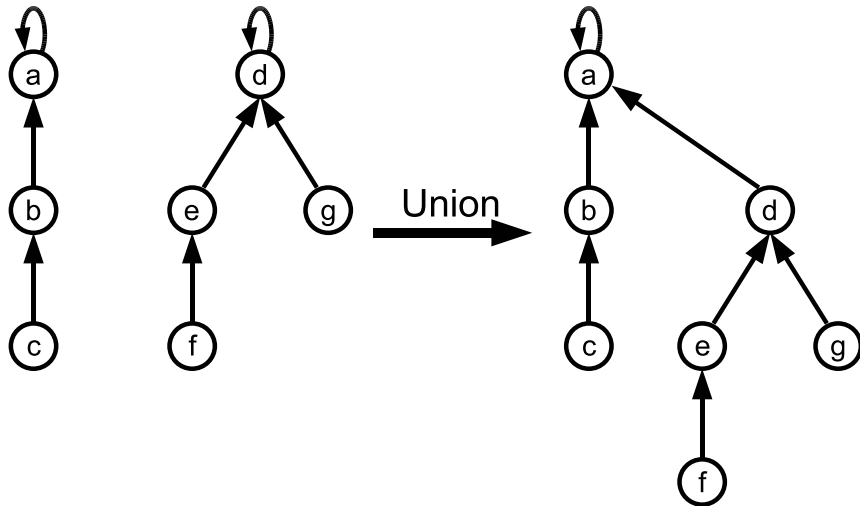
```
1:  $y = x$   
2: while  $y.parent \neq y$  do  
3:      $y = y.parent$   
4: end while  
5: return  $x$ 
```

- 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





- 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

MakeSet( $x$ )

- 1:  $T = \mathbf{new}$  `TreeNode`
- 2:  $T.parent = T$
- 3:  $T.elem = x$
- 4:  $T.rank = 0$
- 5: **return**  $T$

# Łączenie według rangi

Union( $x$ ,  $y$ )

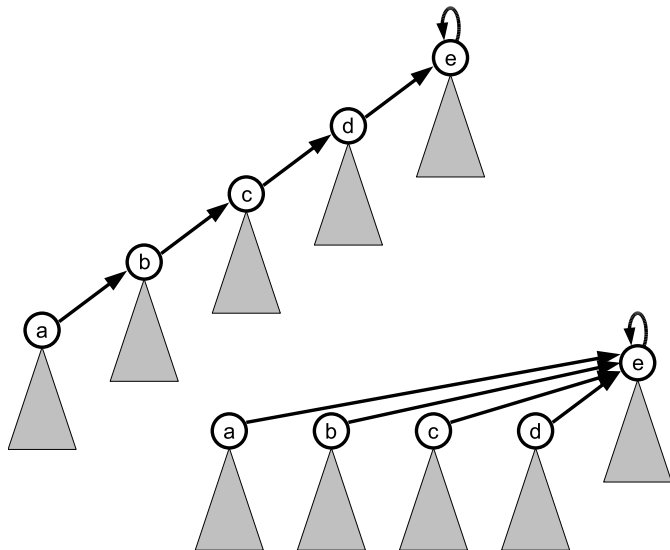
```
1:  $rx = \text{FindSet}(x)$ 
2:  $ry = \text{FindSet}(y)$ 
3: if  $rx.rank > ry.rank$  then
4:      $ry.parent = rx$ 
5: else
6:      $rx.parent = ry$ 
7:     if  $rx.rank = ry.rank$  then  $ry.rank = ry.rank + 1$ 
8: end if
```

- 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 ścieżce, którą przeszliśmy, każdemu z węzłów ustawiając jako nowego rodzica reprezentanta

FindSet( $x$ )

- 1: **if**  $x.parent \neq x$  **then**
- 2:      $x.parent = \text{FindSet}(x.parent)$
- 3: **end if**
- 4: **return**  $x.parent$

# Kompresja ścieżki



- 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



- 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

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