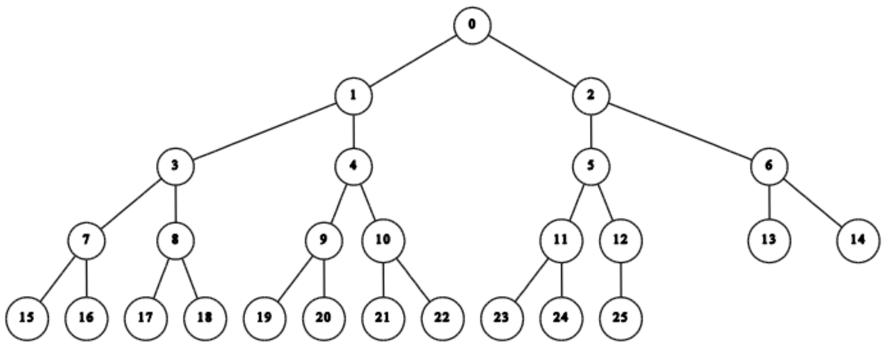
[4] Heaps: Binary, d-ary, binomial, Fibonacci, heaps comparison

- **halda** (**heap**) = datová struktura (většinou stromová), která splňuje vlastnost haldy (heap property): **Je-li B potomkem uzlu A, pak key(B) >= key(A).**
- jedna z nejefektivnějších implementací abstraktní datové struktury – prioritní fronty

- časté operace:
    - **Insert**(*x*) – přidá do haldy nový klíč *x*
    - **AccessMin** – najde a vrátí minimální prvek haldy
    - **DeleteMin** – odstraní minimální prvek (ten je většinou v kořeni)
    - **DecreaseKey(*x*,*d*)** – zmenší prvek *x* v haldě o *d*
    - **Merge(H_1,H_2)** – sloučí dvě haldy do jedné (bude obsahovat všechny prvky z obou)
    - **Delete(*x*)** – odstraní z haldy klíč *x*

Binární halda

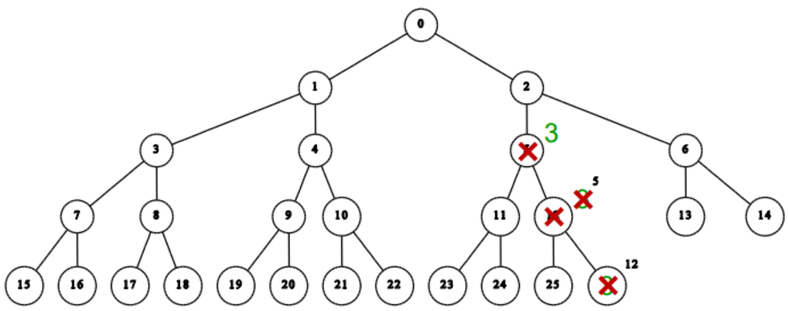
- **binární halda** = binární strom se dvěma dalšími omezeními:
    - je to kompletní binární strom, kromě poslední úrovně
        - tzn., že všechny úrovně (ne nutně ta nejhlubší) jsou zcela plné
        - není-li poslední úroveň plná, uzly jsou plněny zleva doprava
    - každý uzel je menší či roven svému potomku



- **Insert(x):**

```
while ( key(parent(x)†) > key(x) ) {
    Swap a location of the node x with the node parent(x);
}
```

†parent(*x*) returns the parent of a node *x*. It returns *x* in the case where *x* has no parent.



- **AccessMin** – vrátí kořen stromu binárního stromu haldy

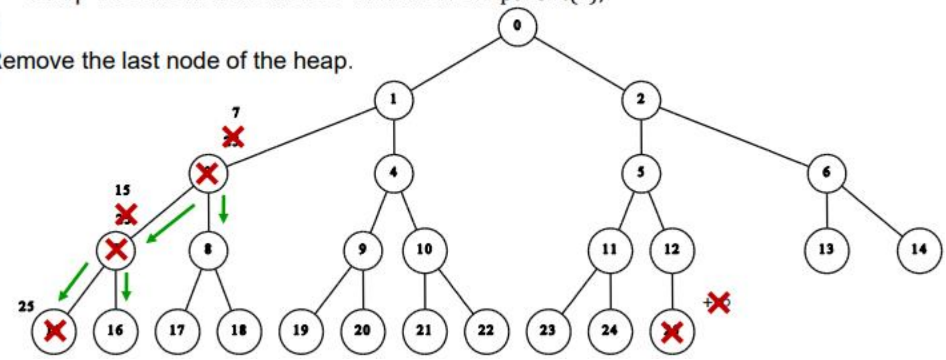
- **DeleteMin**:

```
&x = a location of the root of the heap;
key(x) = +∞;
&y = a location of the last node of the heap;
do {
    Swap a location of the node x with a location of the node y;
    &x = &y;
    for each z ∈ descendants(x) do
        if ( key(y) > key(z) ) then &y = &z;
} while ( &x ≠ &y );
Remove the last node of the heap.
```

- **DecreaseKey(*x*,*d*) –** nejprve sníží klíč x o hodnotu d, pak použije algoritmus podobný insertu

- **Delete(&x):**

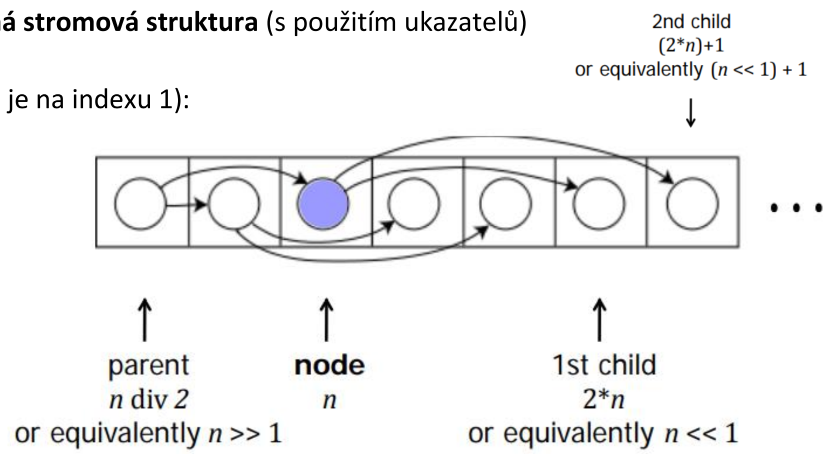
```
key(x) = +∞; &y = a location of the last node of the heap;
do {
    Swap a location of the node x with a location of the node y;
    &x = &y;
    for each z ∈ descendants(x) do
        if ( key(y) > key(z) ) then &y = &z;
} while (&x ≠ &y );
while ( key(parent(x)) > key(x) ) {
    Swap a location of the node x with the node parent(x);
}
Remove the last node of the heap.
```



Reprezentace binární haldy

- **obousměrná stromová struktura** (s použitím ukazatelů)

- **pole** (kořen je na indexu 1):



BuildHeap

```
BuildHeap ( array A )
for i = ⌊length(A)/2⌋ downto 1 do {
        Heapify(A,i);
}
```

```
Heapify ( array A, index i )
min = i;
do {
        left = 2·i;
        right = 2·i + 1;
        if (left ≤ length(A)) and (A[left] < A[min]) then min = left;
        if (right ≤ length(A)) and (A[right] < A[min]) then min = right;
        if min = i then break;
        swap A[i] ↔ A[min];
        i = min;
} while true;
```

Časová složitost

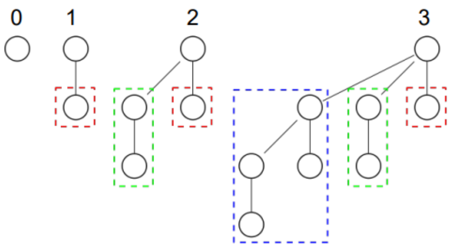
- **Insert** = O(log(n))
- **Delete** = O(log(n))
- **AccessMin** = O(1)
- **DeleteMin** = O(log(n))
- **DecreaseKey** = O(log(n))
- **BuildHeap** = O(n) = $\sum_{h=0}^{\lfloor\log(n)\rfloor}(\text{number of nodes at heigh } h)\cdot O(h) \le \sum_{h=0}^{\lfloor\log(n)\rfloor}\left\lceil\frac{n}{2^{h+1}}\right\rceil\cdot O(h) \le O(n\cdot\sum_{h=0}^{\infty}\frac{h}{2^h})$
- **Merge** = O(n) tvorbou nové haldy

D-regulární halda (d-ary heap)

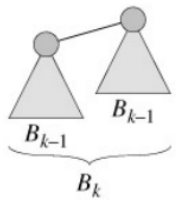
- zobecnění binární haldy, kde uzly mají *d* potomků (namísto 2)
- operace jsou analogické, asymptotická složitost též
- přesná složitost se liší, neboť má logaritmus základ *d*
- pro efektivnější implementaci je vhodné zvolit *d* jako mocněnec 2
- bývá mnohem rychlejší než binární halda v případě velikosti přesahující cache paměti

Binomiální halda

- kolekce binomiálních stromů (BT) stupně i = 0,…,⌊log(n)⌋
- pro každý stupeň může být buď pouze jeden či žádný BT
- každý BT v haldě splňuje vlastnost haldy
- **BT** je definován rekurzivně
    - BT stupně 0 je uzel
    - BT stupně *k* má kořen, jehož potomci jsou kořeny BT stupňů *k-1,k-2,…2,1,0*



- **pro binomiální strom Bk stupně k platí:**
  - splňuje vlastnost haldy
  - výška stromu je k
  - jeho kořen má k potomků
  - má 2k uzlů
  - v hloubce i = 0,…,k je přesně $\binom{k}{i}$ uzlů



- **alternativní definice binárního stromu:**
  - BT stupně k se skládá ze dvou BT stupně k-1, které jsou spojeny
  - kořen jednoho, který je větší než ten druhý, je jeho potomek, který je nejvíce nalevo

Reprezentace binomiální haldy

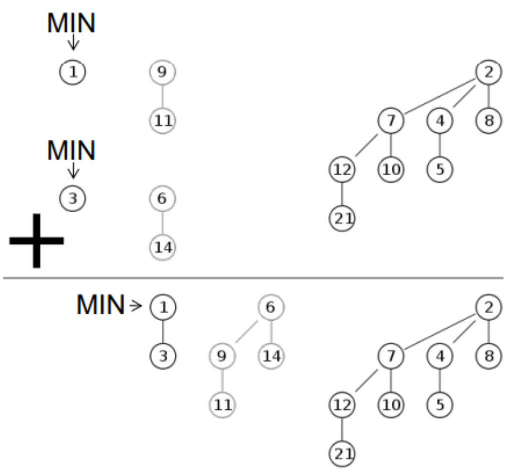
- žádná operace nepotřebuje náhodný přístup ke kořenům BT, kořeny mohou být uloženy ve spojovém seznamu seřazeném dle stupně stromu (vzestupně), popř. klasicky v poli
- binomiální halda je tvořena binomiálními stromy a dodatečným ukazatelem na BT s minimálním uzlem celé haldy (MIN pointer)
  - dle vlastnosti haldy je MIN vždy kořenem
  - MIN musí být při každé operaci (kromě AccessMin) aktualizován
    - to lze provést v O(log(n))

Operace binální haldy

- **Insert(x):**
  - vytvoří novou haldu s jediným prvkem (jeden strom stupně 0)
  - spojí ji s původní haldou

- **AccessMin** – vrátí kořen BT dle MIN pointeru

- **Merge(H_1,H_2):**
  - každý BT v haldě odpovídá bitu v bitové reprezentaci své velikosti
    →analogie mezi slučovámím dvou hald a binárním součtem jejich velikostí (zprava)
  - každý přenos součtu odpovídá slučování dvou binárních stromů
    - BT mohou být sloučeny triviálně (díky své struktuře)
    - kořen je nejmenší prvek → porovnání dvou klíčů vrátí nejmenší klíč a stane se novým kořenem
    - druhý strom se pak stane podstromem sloučeného stromu
    - na konci aktualizujeme MIN pointer



- **DeleteMin**:

```
1. procedure DeleteMin(binomial_heap H)
2. tree_with_minimum = H.MIN;
3. for each tree ∈ tree_with_minimum.subTrees do {
4.     tmp.addTree(tree);
5. }
6. H.removeTree(tree_with_minimum )†;
7. H = Merge(H, tmp);
```

† Technically, this operation removes only the root of tree_with_minimum. All children subtrees of the root are used in tmp heap which is merged at line 7.

- **DecreaseKey**:
    - analogická operace u binární haldy
    - po snížení hodnoty prvku se může stát menším než jeho rodič (x heap property)
        - je pak nutno jej prohodit s rodičem, prarodičem atd., dokud není dodržena
    - každý BT má výšku nejvýše log(n), operace tedy zabere O(log(n)) času

- **Delete(x):**
    - sníží hodnotu prvku x na -∞ (čímž se stane nejmenším prvkem haldy)
    - odstraní jej pomocí operace DeleteMin

Časová složitost

- **Merge** = O(log(n))
- **Insert** = O(log(n)), amortizovaná složitost je O(1)
- **AccessMin** = O(1)
- **DeleteMin** = O(log(n))
- **DecreaseKey** = O(log(n))
- **Delete** = O(log(n))

Amortizovaná složitost

- čas potřebný k vykonání sekvence operací je zprůměrován přes všechny vykonané operace
- lze použít k ukázce, že průměrná cena operace je nízká, i když jedna operace může být drahá
- od average-case se liší tím, že se nebere v úvahu pravděpodobnost – amortizovaná složitost garantuje průměrný výkon každé operace v nejhorším případě
-
- **příklad** – složitost INSERT operace u dynamického pole:
    - DP je pole, které se při naplnění zdvojnásobí
    - INSERT bez změny velikosti stojí O(1), pro N elementů O(N)
    - je-li pole plné, je třeba realokace, což zabere v nejhorším případě O(N)
    - pro vložení N elementů s realokací potřebujeme nejhůře O(N/2) + O(N/4) + … + $(O/2^{\lfloor \log(N) \rfloor})$ + O(N) = O(N) + O(N) = O(N)

$$\sum_{i=0}^{\lfloor \log N \rfloor} \left\lfloor \frac{N}{2^i} \right\rfloor < N \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2N$$

    - amortizovaná složitost INSERT operace je pak O(N)/N = O(1)

Fibonacciho halda

- vychází z binomiální haldy
- má volnější strukturu, což umožňuje snížit hranice asymptotické složitosti
- podporuje stejné operace, ale není-li potřeba mazání prvku, jejich amort. sl. je O(1)
- Delete a DeleteMin mají amortizovanou složitost O(log(n))
- není vhodné pro real-time systémy (některé operace totiž mohou mít i lineární složitost)
- kvůli konstatním faktorům a implementační náročnosti se ale používají méně než klasické binární či d-regulární haldy
- Fibonacciho halda je také kolekce stromů splňujících heap property
    - na rozdíl od binomální haldy, zde jsou stromy sice kořenové, ale nesetříděné
    - NBT U_0 se skládá z jednoho uzlu, NBT U_k se skládá ze dvou binomiálních stromů U_{k-1} tak, že kořen jednoho je kterýmkoliv potomkem kořenu druhého

- Fibonacciho halda má také flexibilnější strukturu – stromy nemají předurčený tvar a v extrémním případě může mít halda každý prvek v samostatném stromě
- flexibilita umožňuje tzv. „lazy“ operace – odložení práce pro pozdější operace

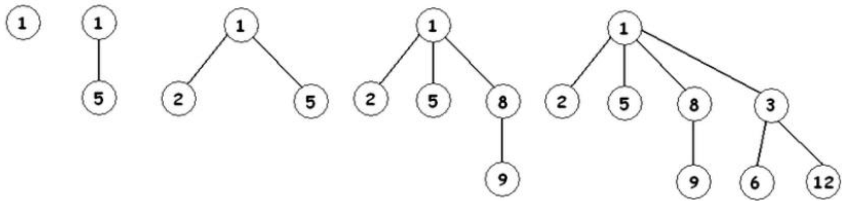
Fibonacciho stromy

- každý uzel má stupeň (tj. počet potomků) nejvýše O(log(n)
- velikost podstromu s kořenem v uzlu stupně *k* je nejméně F_{k+2}, kde F_k je k-té Fibonacciho číslo

$$F_n = \begin{cases} 0, & \text{for } n = 0; \\ 1, & \text{for } n = 1; \\ F_{n-2} + F_{n-1} & \text{otherwise.} \end{cases} \quad \Leftrightarrow \quad F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}} \text{ where } \varphi = \frac{1+\sqrt{5}}{2} \approx 1.618;$$

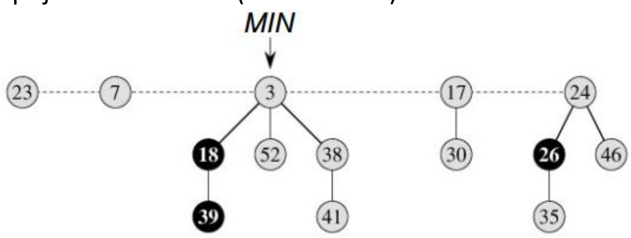
- toho je dosaženo dodržováním dvou **pravidel Fibonacciho stromů**:
    - lze oříznout nejvýše jednoho potomka každého nekořenového uzlu
    - je-li oříznut druhý potomek, uzel samotný je třeba oříznout od svého rodiče a stane se kořenem nového stromu

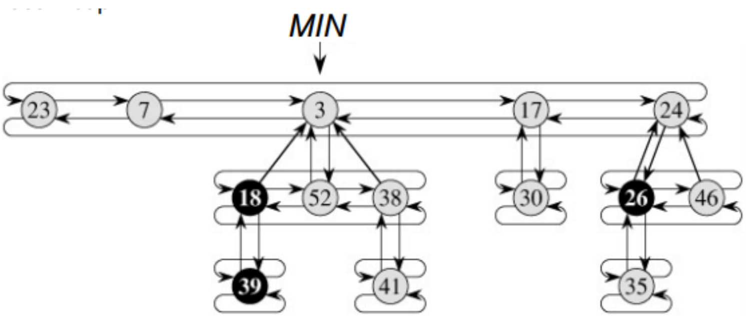
- počet stromů se snižuje operací DeleteMin, kde se slučují do sebe



Reprezentace Fibonacciho haldy

- stromy jsou kořenové, ale **nesetříděné**
- každý uzel *x* má ukazatel na svého rodiče a ukazatel na svého potomka
- potomci *x* jsou spojeny v **obousměrném kruhovém spojovém seznamu**
- kořeny všech stromů Fibonacciho haldy jsou také spojeny v obousměrném kruhovém spojovém seznamu (tzv. **root list**)





- **N** = počet prvků v haldě
- **MIN** = ukazatel na minimální prvek haldy (kořen z root listu)
- **key(x)** = hodnota klíče prvku x
- **descendants(x)** = všechny potomci x
- **parent(x)** = rodič prvku x (případně x, pokud nemá rodiče)
- **mark(x)** = boolean hodnota indikující, zda-li prvek x ztratil potomka poté, co se stal potomkem jiného prvku
    - nové prvky jsou neoznačené, stejně tak prvky, které se stanou potomky

Operace Fibonacciho haldy

- **Merge(H_1,H_2)** – spojí oba spojové seznamy do jednoho a pak aktualizuje MIN; **O(1)**
- **AccessMin** – vrátí kořen Fibonacciho stromu z MIN pointeru; **O(1)**
- **Insert(x):**
    - vytvoří novou haldu obsahující pouze uzel *x* (jeden strom stupně 0)
    - mark(*x*) = false
    - sloučí jej s původní haldou
    - časová složitost **O(1)**

DeleteMin

```
z = MIN;
if z ≠ null then {
    for each x ∈ descendants(z) do
        add x to the root list of the heap;
    remove z from the root list of the heap;
    if N = 1 then
        MIN = null
    else {
        MIN = any pointer to a root from the root list of the heap;
        Consolidate;
    }
    N--;
}
```

time complexity: **O(*N*)**
amortized: **O(log(*N*))**

// Ukázku operace DeleteMin přikládám na konec dokumentu.

Consolidate

```
for i = 0 to max. possible degree of a tree in Fibonacci heap of size N  do A[i] = null;
for each w ∈ all trees in the root list of the heap do {
  x = w; d = a degree of the tree w;
  while A[d] ≠ null do {
    y = A[d];
    if key(x) > key(y) then swap x and y;
    remove y from the root list of the Heap;
    make y a child of x, incrementing the degree of x;
    mark(y) = false; A[d] = null; d++;
  }
  A[d] = x;
}
MIN = null;
for i = 0 to max. degree of a tree in the array A do
  if  A[i] ≠ null then {
    add A[i] to the root list of the heap;
    If  (MIN = null) or (key(A[i]) < key(MIN)) then MIN = A[i];
  }
```

time complexity: **O(*N*)**
amortized: **O(log *N*)**

DecreaseKey (*x*,*d*)

```
key(x) = key(x) – d;
y = parent(x);
if  (x ≠ y) and (key(x) < key(y)) then {
    Cut(x,y);
    Cascading-Cut(y);
}
If  key(x) < key(MIN) then MIN = x;
```

time complexity: **O(log *N*)**
amortized: **O(1)**

Cut(*x*,*y*)

```
remove x from the child list of y,
decrementing the degree of y;
add x to the root list of the heap;
mark(x) = false;
```

time complexity: **O(1)**

Delete(*x*)

```
DecreaseKey(x,∞)
DeleteMin;
```

time complexity: **O(*N*)**
amortized: **O(log *N*)**

Cascading-Cut (*y*)

```
z = parent(y);
if  (y ≠ z) then
    if mark(y) = false then mark(y) = true
    else  {
        Cut(y,z);
        Cascading-Cut(z);
    }
```

time complexity: **O(log *N*)**
amortized: **O(1)**

Porovnání složitostí hald

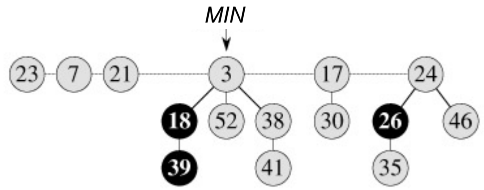
	binary heap	*d* -ary heap	binomial heap	Fibonacci heap
AccessMin	Θ(1)	Θ(1)	Θ(1)	Θ(1)
DeleteMin	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n) amortized: O(log(n))
Insert	Θ(log(n))	Θ(log(n))	O(log(n)) amortized: O(1)	Θ(1)
Delete	Θ(log(n))	Θ(log(n))	O(log(n))	O(n) amortized: O(log(n))
Merge	Θ(n)	Θ(n)	O(log(n))	Θ(1)
DecreaseKey	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n)) amortized: O(1)

Zdroj: https://cw.fel.cvut.cz/b181/_media/courses/b4m33pal/2016pal04.pdf

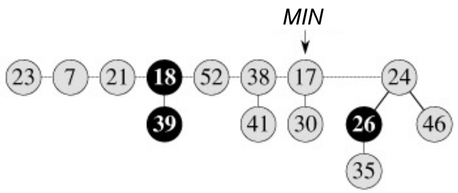
// Následuje ukázka operace DeleteMin.

Fibonacci Heap – DeleteMin Example

1. Consider the following Fibonacci heap.

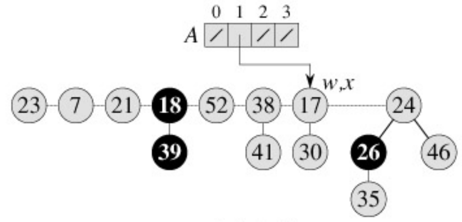


2. The situation after the minimum node *z* is removed from the root list and its children are added to the root list.

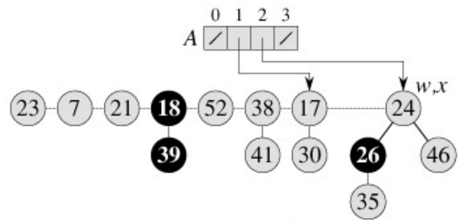


Fibonacci Heap – DeleteMin Example

3. The array *A* and the trees after each of the first three iterations of the *for each* loop of lines 2-12 of the procedure Consolidate. The root list is processed by starting at the node pointed to by *MIN* and following *right* pointers. Each part shows the values of *w* and *x* at the end of an iteration.

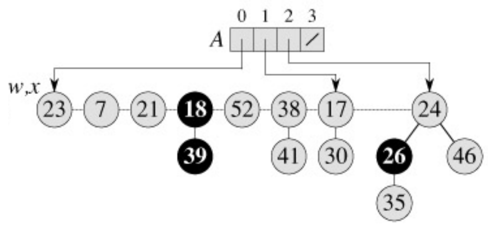


4.

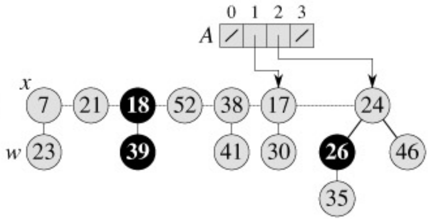


Fibonacci Heap – DeleteMin Example

5.

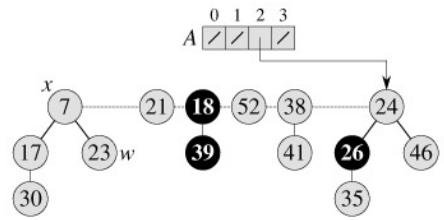


6. The Figure shows the situation after the first time through the *while* loop. The node with key 23 has been linked to the node with key 7, which is now pointed to by *x*.

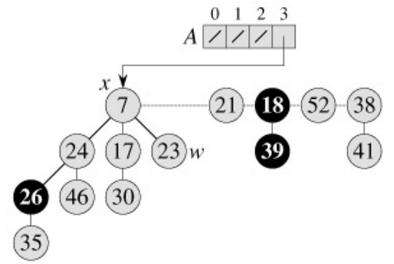


Fibonacci Heap – DeleteMin Example

7. The node with key 17 has been linked to the node with key 7, which is still pointed to by *x*.

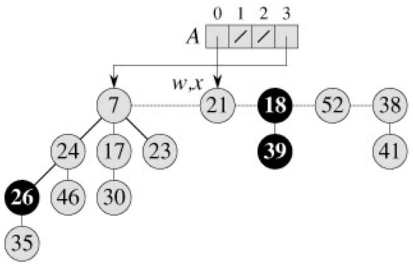


8. The node with key 24 has been linked to the node with key 7. Since no node was previously pointed to by *A*[3], at the end of the *for each* loop iteration, *A*[3] is set to point to the root of the resulting tree.

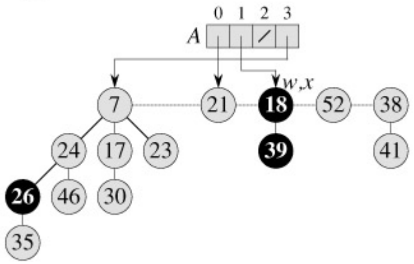


Fibonacci Heap – DeleteMin Example

9. The situations after each of the next four iterations of the *for each* loop.

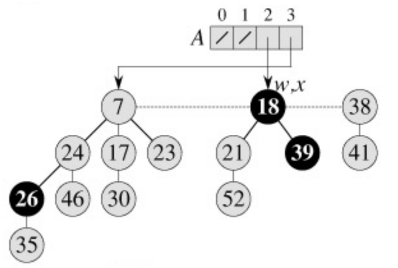


10.

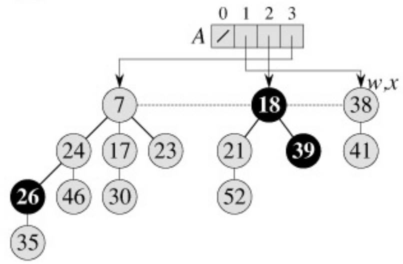


Fibonacci Heap – DeleteMin Example

11.



12.



Fibonacci Heap – DeleteMin Example

13. The Fibonacci heap after reconstruction of the root list from the array *A* and determination of the new *MIN* pointer.