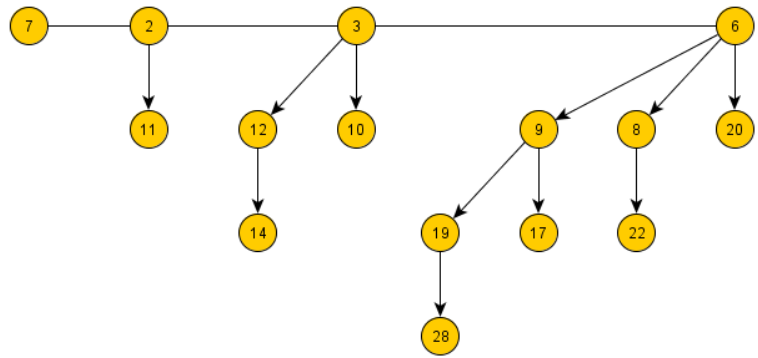


## Binomiální halda

= Binomial heap

- je halda složená z posloupnosti binomiálních stromů
- halda NENÍ strom, ale skládá se ze stromů
- platí  $h$  vlastnost – zajišťuje se při slučování
- stromy jsou uspořádané
- každý stupeň je zastoupen pouze jednou
- binomiální halda má výhodu v rychlejších operacích sloučení hald  $O(\log n)$  a vložení prvku – amortizovaně  $O(1)$
- halda není optimální pro vyhledávání
- binom. halda má odkaz na strom s nejnižším stupněm, FH má odkaz na strom, obsahující MAX

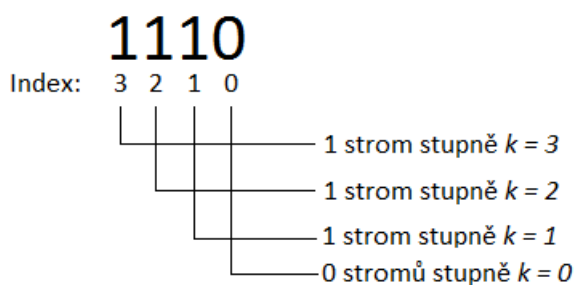


## Binomiální strom

- je rekurzivně definován
- binomiální strom stupně  $n$  vznikne sloučením dvou stromů stupně  $n - 1$  tak, že kořen jednoho se stane nejlevějším synem druhého
- při sloučení porovnáme prioritu kořenů obou stromů a následně ten s nižší prioritou zavěšíme pod kořen s vyšší prioritou.
- binomiální strom stupně 0 obsahuje právě jeden uzel
- stupeň stromu = počet synů, ne hloubka stromu
- platí  $h$  vlastnost = všichni potomci jsou menší než daný uzel, tzn. každý prvek každého stromu má vyšší prioritu, než kterýkoliv z jeho potomků, halda se proto chová jako prioritní fronta

Binomiální halda obsahuje maximálně jeden strom každého řádu.

- Binomiální strom stupně  $n$  obsahuje  $2^n$  uzlů.
- Binomiální strom stupně  $n$  má hloubku  $n$ .
- Binomiální halda stupně  $n$  má právě jeden strom stupně  $n$ .
- Kořen binomiálního stromu má  $n$  potomků.
- Kořen binomiálního stromu obsahuje jako své potomky všechny stromy nižších řádů.
- V hloubce  $i$  je ( $n$  nad  $i$ ) uzlů
- Při slučování stromů  $k$  a  $l$  -  $k = l$ , sloučí se do stromu stupně  $k + 1$   
-  $k \neq l$ , lze sloučit do haldy



stupeň stromu	počet stromů (binárně) = potence haldy				počet vnitřních uzlů
	3	2	1	0	
0				1	0
1			1		1
0 a 1			1	1	2
2		1			3
0 a 2		1		1	4
1 a 2		1	1		5
0 a 1 a 2		1	1	1	6
3	1				7

## Základní operace binomiální haldy

Operace jsou popsány pro min-heap – nižší prvek má vyšší prioritu. Max-heap (tj. vyšší prvek má vyšší prioritu) funguje analogicky.

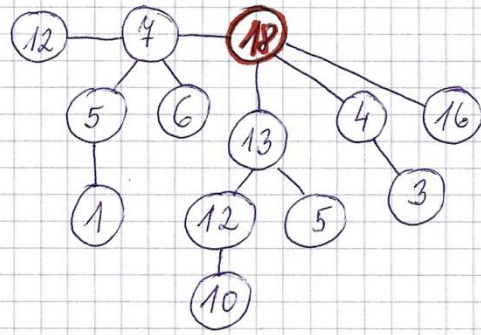
### Vložení

- vloží prvek do haldy
- vytvoří pro prvek binomiální strom řádu 0
- a provede sloučení tohoto stromu s haldou
- pokud vkládáme prvek s nižší prioritou, než je priorita současného nejnižšího prvku haldy, tak tento prvek označíme za minimální

- asymptotická složitost operace je  $O(\log n)$ , amortizovaná složitost  $O(1)$

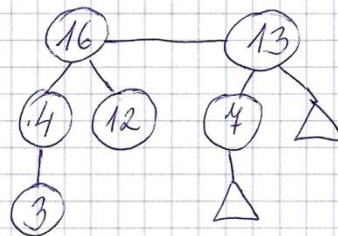
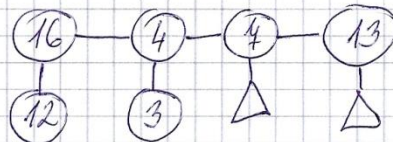
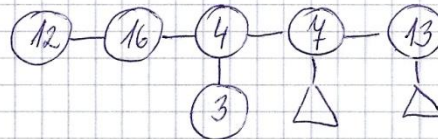
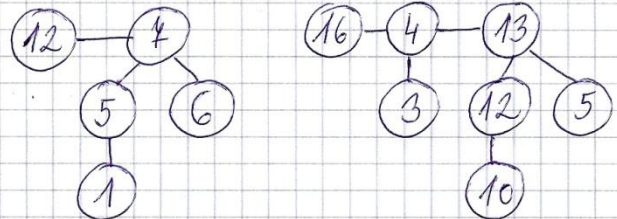
### Mazání

- mazané číslo zvětším na hodnotu > kořen
- přesunu ho do kořene
- smažu prvek
- syny přesunu do kořenů, čímž vytvořím novou haldu
- sloučím haldy zpět do jedné
- složitost je  $O(\log n)$ , protože při nalezení prvku musí projít všechny stromy



Chci smazat MAX (MIN) prvek – tj. č. 18

- hledám MAX (MIN) prvek – vždy v kořenech
- složitost  $O(\log n)$ , protože musíme projít všechny stromy
- prvek smažu a vytvořím novou haldu tím, že po odstranění kořene přesunu všechny syny do seznamu, tj. udělám z nich kořeny nových stromů
- následně stromy spojm
- spojuji je tak dlouho, až je to binom. halda



### Merge

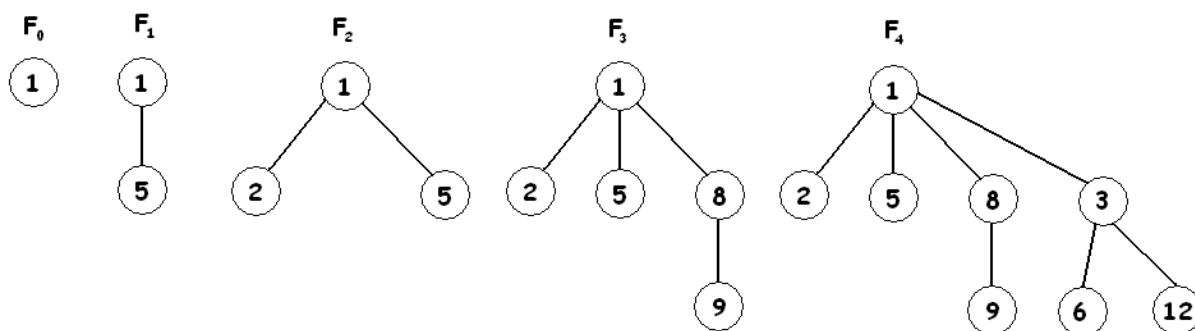
- funkce, která provede sloučení haldy B do haldy A
- tato operace odpovídá sčítání dvou binárních čísel
- algoritmus prochází obě haldy od nejnižšího řádu
- pokud halda B obsahuje strom řádu  $n$  a halda A nikoliv, pak jej pouze překopíruje
- pokud obě haldy obsahují stromy stejného řádu, tak je sloučí do přenosového stromu (obdoba přenosového bitu u sčítání), který zohlední při následném spojování stromů řádu  $n + 1$
- nakonec porovná minimální prvky obou slučovaných hald a za minimální prvek výsledné haldy označí nižší z nich
- asymptotická složitost operace je  $O(\log n)$

### Fibonacciho halda

zdroj: [http://cs.wikipedia.org/wiki/Fibonacciho\\_halda](http://cs.wikipedia.org/wiki/Fibonacciho_halda)

- principiálně vychází z binomiální haldy, je jí velmi podobná
- kořeny stromů jsou cyklicky zřetězené
- FH má odkaz na strom obsahující MIN (nebo MAX), binom. halda má na strom s nejnižším stupněm, MIN (MAX) je v kořeni
- můžou se opakovat stromy stejného stupně
- operace vložení, hledání minima, snížení hodnoty klíče a spojování stromů probíhají v konstantním čase, amortizovaně  $O(1)$
- operace mazání a mazání minimálního prvku pracuje s časovou složitostí  $O(\log n)$ , přičemž k výraznému zrychlení výpočtů oproti binomiální haldě dochází zejména v momentě, kdy některá z větví stromu neobsahuje žádná data.

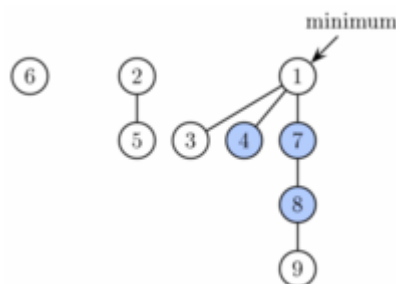
- mezi nejčastější aplikace Fibonacciho haldy patří Jarníkův a Dijkstrův algoritmus, které slouží k vyhledávání minimální kostry grafu a k určení nejkratší cesty v grafu.
- označení Fibonacciho haldy vychází z Fibonacciho čísel, která mají souvislost s počtem vrcholů v každém stromě.



Počty vrcholů stromů  $F_0, F_1, \dots$  tvoří Fibonacciho posloupnost

### Struktura Fibonacciho haldy

Na obr. - příklad Fibonacciho haldy, která je tvořena třemi stromy stupňů 0, 1 a 3. Tři prvky jsou zvýrazněny (modrou barvou). Potence haldy je 9.



Fibonacciho haldu tvoří

- skupina stromů vyhovující lokální podmínce na uspořádání haldy
- podmínka vyžaduje, aby pro každý uzel stromu platilo, že prvek, který reprezentuje, je menší než prvek reprezentovaný jeho potomky
- z této podmínky vyplývá, že MIN prvkem je vždy kořen jednoho ze stromů
- vnitřní struktura Fibonacciho haldy je v porovnání s binomiální haldou daleko více flexibilní.
- jednotlivé stromy nemají pevně daný tvar a v extrémním případě může každý prvek haldy tvořit izolovaný strom nebo naopak všechny prvky mohou být součástí jediného stromu hloubky  $N$ .
- tato flexibilní struktura umožňuje velmi jednoduchou implementaci operací s haldou
- operace, které nejsou potřebné, odkládáme a vykonáváme je až v okamžiku, kdy je to nevyhnutelné, například spojení nebo vložení nového prvku se jednoduše provede spojením kořenových seznamů (s konstantní náročností) a jednotlivé stromy spojíme až při operaci snížení hodnoty klíče.
- obecně lze říci, že stupeň uzlu (zde je stupeň myšlen jako počet synů prvku) je velmi nízký: každý uzel má stupeň nejvýše  $\log N$  a velikost podstromu vycházejícího z kořenového uzlu stupně  $K$  je nejméně  $F_{K+2}$ , kde  $F_K$  je  $K$ -té Fibonacciho číslo
- toho je dosaženo díky pravidlu, které dovoluje oříznout nejvýše jednoho syna od každého nekořenového prvku
- pokud je odříznut druhý syn, uzel musí být odříznut od svého otce a stává se kořenem nového stromu
- počet stromů je snižován při operaci odstranění prvku s nejnižší hodnotou klíče, kdy dochází ke spojování stromů.

Nejhorší případ stromu stupně  $d$ , tj. nejmenší a tedy i nejméně košatý strom, jaký je povolen, se konstruuje složením nejmenších stromů stupně  $d-1$  a  $d-2$ .

Počet vrcholů v nejmenším stromu stupně  $d$  je  $F_d = F_{d-1} + F_{d-2}$ .

Formulka je stejná jako při výpočtu Fibonacciho čísel, proto se této haldě říká Fibonacciho haldy.

Fibonacciho haldy realizuje operace

- MIN
- INSERT
- DECREASE KEY
- INCREASE KEY

v amortizovaném čase  $O(1)$

a operace

- DELETE MIN
- DELETE

v amortizovaném čase  $O(\log n)$ .

Implementace této haldy je podstatně složitější a konstanty před časovými složitostmi jednotlivých operací jsou poměrně vysoké. Na druhou stranu použitím Fibonaccioho haldy můžeme dosáhnout podstatně lepších asymptotických časových složitostí.

### Vkládání

- vloží se nový prvek (uzel) a víc není nutno řešit
- pokud je nový prvek MIN, přepojí se na něj "pointer haldy"

### Mazání

- z kořene - kořen smažeme  
- jeho syny vložíme do seznamu, tzn. vytvoří kořenové prvky nových stromů  
- provedeme „konsolidaci“ = spojíme 2 stromy stejného stupně (u binom. H se děje při vkládání)  
- nakonec se najde nové MIN, na které směřuje ukazatel haldy
- z uzlu - uzel s podstromem vložíme do seznamu, tím se z něj stane kořen  
- pak smažeme jako kořen  
- výjimka – „trhání ručiček“ u rodiče – syna posuneme do seznamu, až když má pryč obě ručičky

### Implementace binomální haldy

\* Binomialni halda

\* Radi prvky dle priority (mensi == vyssi priorita)

```
public class BinomialHeap<ENTITY extends Comparable> {  
    private Node<ENTITY>[] nodes;  
    private Node<ENTITY> min;
```

\* Konstruktor

\* @param capacity kapacita haldy

```
public BinomialHeap(int capacity) {  
    min = null;  
    nodes = new Node[((int) (Math.log(capacity) / Math.log(2))) + 2];  
}
```

\* Vlozi prvek do haldy, pokud je mensi nez aktualni minimalni, tak jej ulozi jako nejmensi prvek

\* @param e

```
public void insert(ENTITY e) {  
    Node<ENTITY> n = new Node<ENTITY>(e);  
    if (nodes[0] != null) merge(n, nodes[0]);  
    else nodes[0] = n;  
  
    if (min == null) min = n;  
    else if (min.value.compareTo(n.value) == 1) min = n;  
}
```

\* Smaze a vrati entitu s nejvyssi prioritou

\* @return entita s nejvyssi prioritou, @null, pokud je halda prazdna

```
public ENTITY returnTop() {  
    if(min == null) return null;  
    ENTITY tmp = min.value;  
    nodes[min.order] = null; //strom vyjmeme  
    for (Node n : min.children) { //a z potomku udelame nove stromu  
        n.parent = null;  
        if(nodes[n.order] != null) merge(n, nodes[n.order]);  
        else nodes[n.order] = n;  
    }  
    min.children.clear();  
    ENTITY minVal = null;
```

```

Node minNode = null;
for(int i = 0; i < nodes.length; i++){
    if(nodes[i] != null){
        if(minVal == null){
            minVal = nodes[i].value;
            minNode = nodes[i];
        }
        else if(minVal.compareTo(nodes[i].value) == 1){
            minVal = nodes[i].value;
            minNode = nodes[i];
        }
    }
}
min = minNode;
return tmp;
}

```

**\* Slouci haldy, pokud mergovana halda obsahuje mensi prvek, nez halda, na ktere je volana operace, pak je tento prvek minimalni i ve sloucene halde**  
**\* @param heap**

```

public void mergeHeaps(BinomialHeap<ENTITY> heap) {
    Node<ENTITY> carry = null;
    for (int i = 0; i < heap.nodes.length; i++) {
        if(nodes[i] == null){
            if(heap.nodes[i] == null){
                nodes[i] = carry;
                carry = null;
            }else{
                if(carry == null){
                    nodes[i] = heap.nodes[i];
                }else{
                    carry = mergeTrees(heap.nodes[i], carry);
                }
            }
        }else{
            if(heap.nodes[i] == null){
                if(carry != null){
                    carry = mergeTrees(nodes[i], carry);
                    nodes[i] = null;
                }
            }else{
                if(carry == null){
                    carry = mergeTrees(nodes[i], heap.nodes[i]);
                    nodes[i] = null;
                }else{
                    carry = mergeTrees(carry, heap.nodes[i]);
                }
            }
        }
    }
    if(carry != null) throw new RuntimeException("Preteceni haldy (halda je preplnena)");

    if (this.min == null) this.min = heap.min;
    else if(this.min != null && heap.min != null &&
        heap.min.value.compareTo(min.value) == -1) min = heap.min;
}

```

**@Override**

```
public String toString() {
    StringBuilder builder = new StringBuilder();
    for(int i = 0; i < nodes.length; i++){
        if(nodes[i] == null) builder.append(i + ": null\n");
        else builder.append(i + ":\n" + nodes[i].toString() + "\n");
    }
    return builder.toString();
}
```

**\* Provede sloucení binomialních stromu (včetně přetečení do vyšších radu)**

**\* @param a**

**\* @param b**

```
private void merge(Node a, Node b) {
    if (a.order != b.order)
        throw new IllegalArgumentException("Stromy nejsou stejného radu");
    int tmpOrder = a.order;
    nodes[tmpOrder] = null;
    Node newRoot = null;
    newRoot = mergeTrees(a, b);
    if (nodes[tmpOrder + 1] == null) nodes[tmpOrder + 1] = newRoot;
    else merge(newRoot, nodes[tmpOrder + 1]);
}
```

**\* Sloučí binomialní stromy v jeden (zavěsí strom s nižší prioritou pod strom s vyšší prioritou)**

**\* @param a strom a**

**\* @param b strom b**

**\* @return**

```
private Node mergeTrees(Node a, Node b) {
    Node newRoot = null;
    if (a.value.compareTo(b.value) < 0) {
        b.parent = a;
        a.children.add(b);
        a.order++;
        newRoot = a;
    } else {
        a.parent = b;
        b.children.add(a);
        b.order++;
        newRoot = b;}
    return newRoot;
}
```

**\* Reprezentuje uzel binomialní haldy**

**\* @param <ENTITY>**

```
private class Node<ENTITY extends Comparable> {
    Node<ENTITY> parent;
    ENTITY value;
    List<Node> children;
    int order; //rad binomialni haldy
}
```

```
public Node(ENTITY value) {
    this.value = value;
    children = new ArrayList<Node>();
    order = 0;}
}
```