Amortizovaná složitost. Prioritní fronty, haldy (binární, d-regulární, binomiální, Fibonacciho), operace nad nimi a jejich složitost

1. Asymptotické odhady

Asymptotická složitost je deklarována na základě nejhorší (nejlepší) možné instance běhu algoritmu, což ale není vždy vypovídající, protože i nejhorší sekvence případů může mít výrazně lepší průběh, než by asymptotická složitost napovídala. Operace s vysokou složitostí **změní datovou strukturu** tak, že takto špatný případ nenastane po nějakou delší dobu - tím se složitá operace **amortizuje**.

horní asymptotický odhad (`f` je shora asymptoticky ohraničená funkcí `g` až na konstantu):
$f(n) \in O(g(n))$

$$(\exists c > 0)(\exists n_0)(\forall n > n_0) : c * g(n) \geq f(n)$$

dolní asymptotický odhad (`f` je zespoda asymptoticky ohraničená funkcí `g` až na konstantu):
$f(n) \in \Omega(g(n))$

$$(\exists c > 0)(\exists n_0)(\forall n > n_0) : c * g(n) \leq f(n)$$

optimální asymptotický odhad (`f` je asymptoticky ohraničená funkcí `g` z obou stran až na konstantu):
$f(n) \in \boldsymbol{\Omega}(g(n))$

$$(\exists c_1, c_2 > 0)(\exists n_0)(\forall n > n_0) : c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

2. Amortizovaná složitost

= průměrný čas na vykonání operace v sekvenci operací v nejhorším případě
- nevyužívá pravděpodobnost => průměrný čas na operaci skutečně zaručený

Příklad: vkládání prvků do ArrayListu
- list zdvojnásobuje svou velikost pokaždé, když dojde k jeho naplnění
- vkládání prvků (bez realokace) vyžaduje čas O(1), pro N prvků je to O(N)
- v nejhorším případě vkládání operace potřebuje čas až O(N)
- vložení N prvků (včetně realokace) je tedy potřeba v nejhorším případě O(N) + O(N) = O(N)
- amortizovaný čas na jedno vložení prvku je pak O(N)/N= O(1)

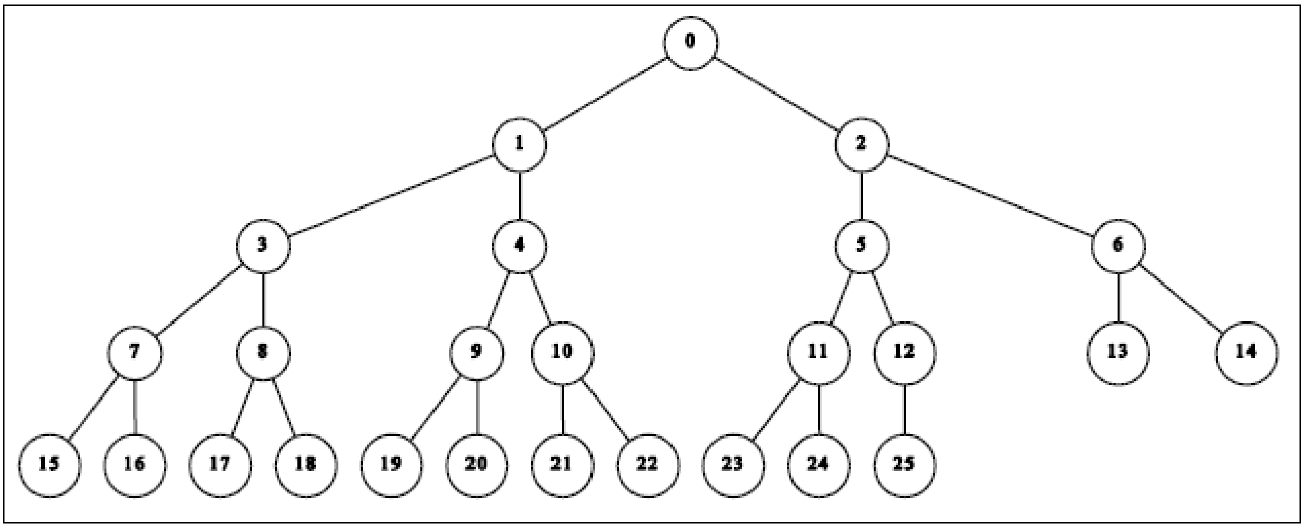
3. Prioritní fronty

- **abstraktní datový typ** (ADT)
- každý **element** má přiřazenu svou prioritu
- první jsou z fronty vybírány elementy s **nejnižší**/**nejvyšší prioritou**
- PF nemusí být implementována pouze haldou

- PF musí implementovat alespoň tyto **operace**:
  - `void push(Element e)    // vloží element s prioritou`
  - `Element pull()          // vybere z fronty element s nejnižší/nejvyšší prioritou`

4. Binární halda

- je to implementace **ADT** prioritní fronty
- stromová struktura s vlastností: pokud A je potomek B, potom B<=A
- **operace**:
  - `insert(x)         // vloží prvek x do haldy`
  - `accessMin()       // vrátí nejmenší prvek haldy`
  - `deleteMin()       // odstraní z haldy nejmenší prvek (obvykle kořen)`
  - `decreaseKey(x,d)  // zmenší hodnotu prvku x o d`
  - `merge(H1, H2)     // sloučí haldy H1a H2 do jedné, kterou vrátí`
  - `delete(x)         // odstraní prvek x z haldy`

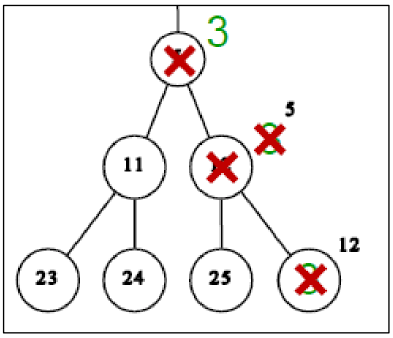


Obrázek 1 - Binární halda je stromová struktura

insert(x)

- složitost `O(log(n))`

```
1. přidáme prvek x na konec haldy;
2. while( parent(x)>x) {
       prohodíme prvek x s prvkem parent(x);
3. }
```



accessMin()

- vrátí hodnotu kořene stromu
- složitost **O(1)**

deleteMin()

- složitost `O(log(n))`

```
1. vrátí element x, který je kořenem stromu;
```

```
2. na místo x vloží nejpravější prvek y ze spodního patra (pozor, ten nemusí být
   maximální!)
3. while(y > nejmenší z jeho dětí){
      prohoď y a jeho nejmenšího potomka // y probublává dolů stromem
4. }
```

decreaseKey(x, d)

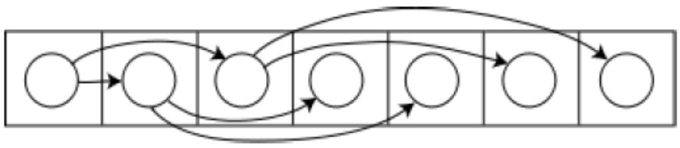
- zmenšíme hodnotu prvku x o d, prvek x necháme probublat stromem nahoru

delete (x)

- složitost `O(log(n))`
- podobné jako `deleteMin()`

Reprezentace binární haldy v paměti

1. **stromovou dynamickou datovou strukturou** s ukazateli v obou směrech
2. **polem** (kořen má index 1, potomci mají index 2k a 2k+1)



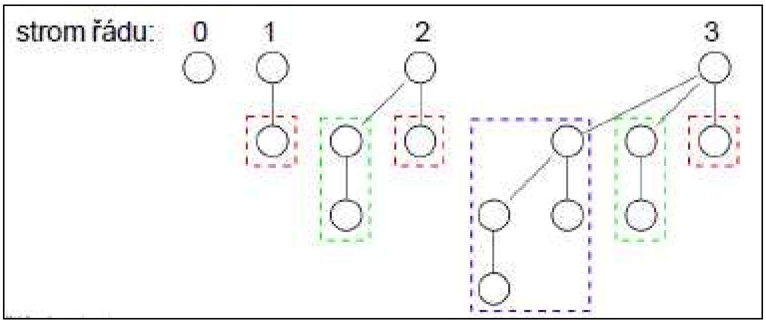
Obrázek 2 - Bin. halda jako pole

5. d-regulární halda

- d udává **stupeň štěpení stromu** haldy
- pro d=2 je d-regulární halda právě binární halda
- operace a jejich složitost nad d-regulární haldou jsou analogické jako v případě binární haldy
- přesná složitost se liší základem **logaritmu** (základ je d)

6. Binomiální halda

- mn. stromů řádu i=1,…, log(n), každý řád je zastoupen max. jedním stromem
- pro binomiální strom řádu i platí:
  - každý vrchol je menší nebo roven všem svým potomkům
  - má 2^i vrcholů
  - má hloubku i
  - jeho kořen má i synů
  - strom řádu i vznikne ze dvou stromů řádu i -1



Obrázek 3 - Řády stromu

-**implementace**: pole ukazatelů na kořeny stromů řádu i, plus zvláštní ukazatel na min prvek (kořen jednoho ze stromů)

merge(h_1, h_2)

- **spojení dvou stromů**
- ke stromu, jehož kořen je menší, se jako další syn připojí strom s větším kořenem

insert(x)

- složitost O(log(n)), amortizovaná složitost je konstantní
- prvek tvoří strom řádu 0, strom se sloučí s jiným stromem řádu 0 (pokud existuje) a vznikne jeden strom řádu 1, ten se dále sloučí s dalším existujícím stromem řádu 1....

accessMin()

- složitost O(1)
- vrátí prvek reprezentovaný kořenem stromu, na nějž ukazuje MIN ukazatel

deleteMin()

- složitost O(log(n))
- vezmou se všechny podstromy, které vznikly odebráním kořene a ty se postupně mergují s ostatními stromy

decreaseKey()

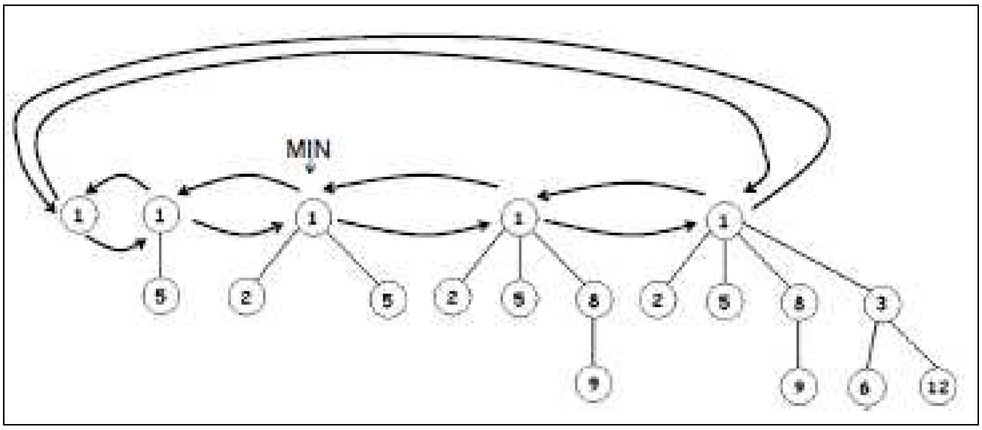
- složitost O(log(n))
- funguje jako u binární haldy, poté se aktualizuje min. ukazatel

7. Fibonacciho halda

- velmi podobná binomiální haldě, ale některé mají amortizovanou složitost
- operace insert, accessMin a merge probíhají v O(1)
- vnitřní struktura je flexibilnější
- podstromy 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 => **jednoduchá implementace**
- **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

- každý vrchol má nejvýše log(n) synů
- velikost stromu řádu k je nejméně F_k +2, kde F_k je k-té Fibonacciho číslo
- kořen každého stromu řádu k má právě k potomků
- stromy haldy propojeny dvojitým kruhovým spojovým seznamem



Obrázek 4 - Kruhový seznam

8. Zdroje

[1] Genyk-Berezovskyj, Marko: Přednáška 1 z PAL.
https://cw.felk.cvut.cz/lib/exe/fetch.php/courses/a4m33pal/pal01.pdf

[2] Genyk-Berezovskyj, Marko: Přednáška 5 z PAL.
https://cw.felk.cvut.cz/lib/exe/fetch.php/courses/a4m33pal/pal05.pdf

[3] Wikipedia: Priority Queue.
http://en.wikipedia.org/wiki/Priority_queue

[4] Mička, Pavel: Amortizovaná složitost.
http://www.algoritmy.net/article/3024/Amortizovana-slozitost