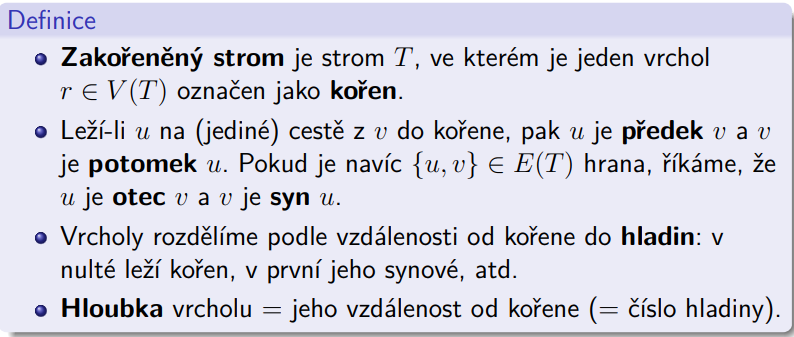
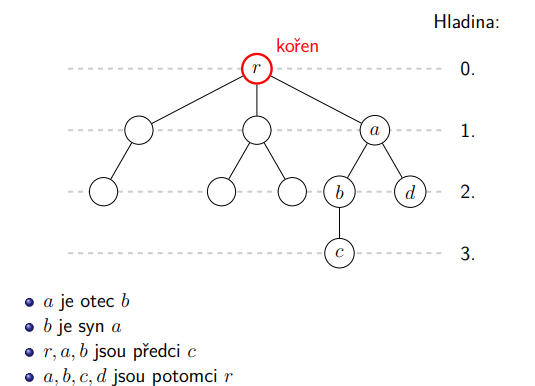
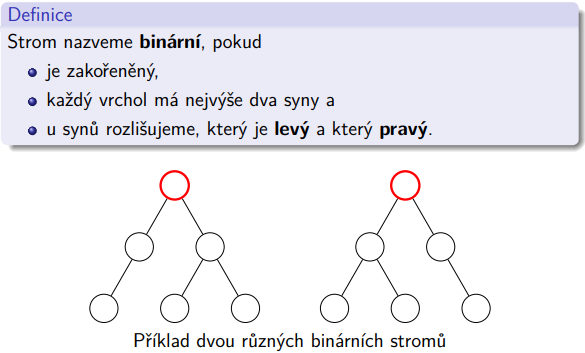
**BI-SPOL-5 Binární haldy, binomiální haldy. Vyhledávací stromy a jejich vyvažování. Tabulky s rozptylováním (hešováním)**

BI-AG1





**Binární strom**



### Binární halda

* binární strom, v každém vrcholu je klíč
* jednotlivé hladiny jsou zaplňovány postupně zleva doprava
* implementace může být s pomocí pole
* uchovává porovnatelné prvky – klíče
* 
* 

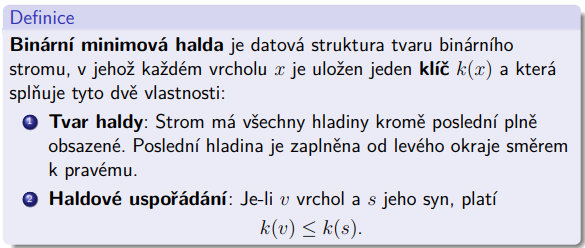
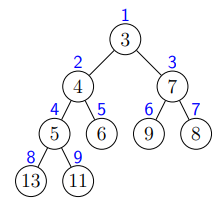
Obsah obrázku text, stůl

Popis byl vytvořen automaticky

- minimová, ale může být i maximová (vrací maxima atd.)

* Tvar haldy: jednotlivé hladiny jsou zaplňovány postupně zleva doprava

#### Binární minimová halda

****

* analogicky i maximová halda
* v paměti můžeme reprezentovat polem
* jednotlivé vrcholy můžeme očíslovat od 1 – potom bude
  + levý syn = 2\*i
  + pravý syn = 2\*i + 1
  + otec = i/2 – dolní celá část
  + i mod 2 udává, jestli je to pravý nebo levý syn

**Nalezení minima** – stačí vzít kořen – O(1)

**Vložení prvku do haldy** – Přidáme prvek na konec nejspodnější hladiny (v případě zaplnění do hladiny nové). Zkontrolujeme vztah parent-child – v případě nekonzistence prohodíme. Do doby, než bude validní strom (BubbleUp – O(*logn*))

**Hledání minima** – jen vrátíme kořen

**Odstranění minima (ExtractMin)** – prohodíme root a poslední list. Odstraníme poslední lid (ve kterém je teď minimum). Teď potřebujeme provést BubbleDown na kořen (prohozený prvek). Prohodíme ho s menším ze dvou synů, aby se zachovaly pravidla. Provádíme tak dlouho, aby platilo haldové uspořádání. (BubbleDown – O(*logn*))

**Vytvoření haldy** – v nejhorším případě z pole postupně bereme prvky(O(n)) a vkládáme (O(logn))– dohromady O(n\*logn)

**Rychlejší vytvoření haldy**

* vycházíme z toho, že pole musíme seřadit jen z půlky. Pro indexy dolní celá část(n/2) … 1 provádíme bubbleDown
* O(n)

Obsah obrázku text

Popis byl vytvořen automaticky

### Binomiální halda

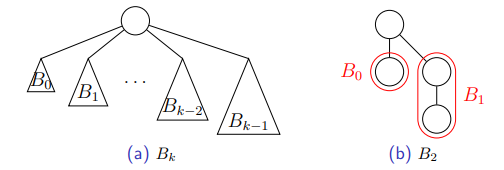
Obsah obrázku stůl

Popis byl vytvořen automaticky

**Binomiální strom**

**Obsah obrázku text

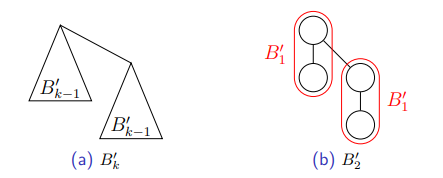
Popis byl vytvořen automaticky**



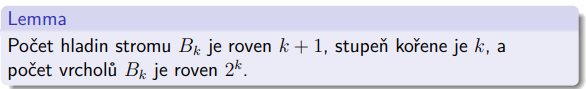
* alternativní definice:

Obsah obrázku text

Popis byl vytvořen automaticky

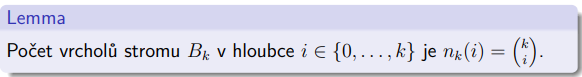


* vlastnosti

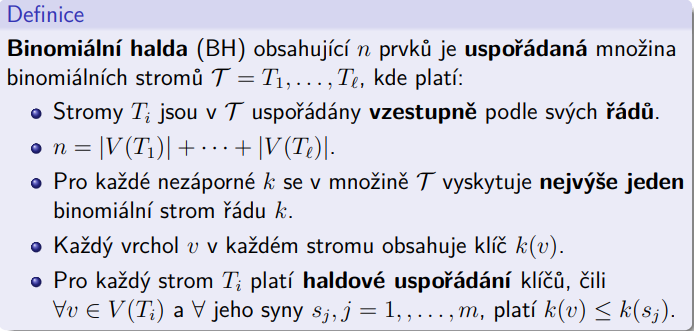
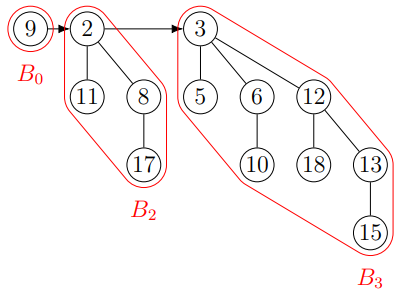


Obsah obrázku text

Popis byl vytvořen automaticky



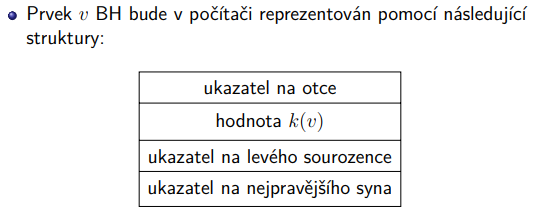
**Binomiální halda**

****

* opět počítáme s minimovou haldou
* binomiální strom Bi se vyskytuje v seznamu T n-prvkové BH právě tehdy, když ve dvojkovém zápisu bk,bk−1,...,b0 čísla n je bi = 1.
* n si tedy převedeme na binární číslo a víme, jaké stromy bude halda obsahovat
* n-prvková BH se skládá z O(log n) binomiálních stromů.

**Reprezentace BH v paměti**

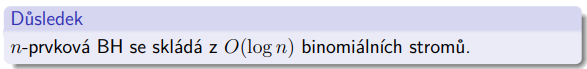
* používá se **spojový seznam**
* seznamy synů jednotlivých vrcholů v binomiálních stromech budeme také udržovat ve spojových seznamech



**Vlastnosti**

Obsah obrázku text

Popis byl vytvořen automaticky



**Nalezení minima (FindMin)**

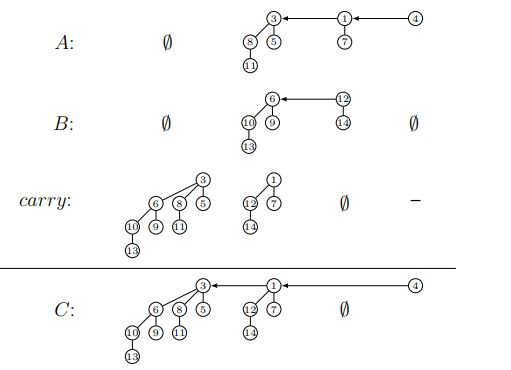
* musí být v kořenech, stačí je projít
* projdeme seznam - O(log n)
* pokud používáme funkci často, vyplatí si udržovat ukazatel na tento globálně nejmenší kořen – O(1)

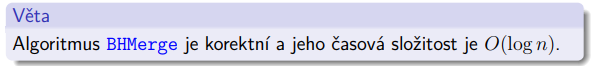
**Sloučení dvou binom. stromů**

* slije dohromady dva binomiální stromy stejného řádu Bi a vytvoří strom Bi+1
* jeden se zavěsí pod druhý jako nejpravějšího syna (podle velikosti kořene – minimový – minimum z těch kořenů bude minimum nového stromu)
* O(1)

**Merge binom. hald**

* Jdeme od nejmenších binom. stromů – vždy sloučíme – slučujeme, dokud to nebude v pořádku





**Přidání nového prvku do BH**

* Vytvoříme BH s jediným prvkem a provedeme merge dvou BH

Obsah obrázku text

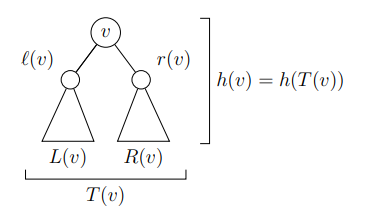
Popis byl vytvořen automaticky

**BHBuild** – n krát po sobě BHInsert – O(n)

**ExtractMin**

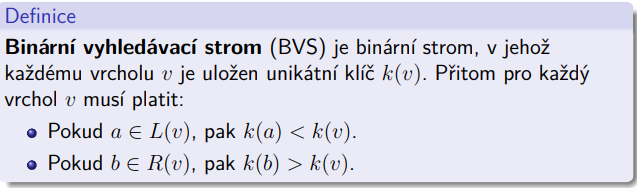
* Najdeme v BH H strom T, jehož kořen je minimum
* Odpojíme tento strom T z BH H
* Odtrhneme z T jeho kořen
* Z jeho synů vytvoříme novou BH
* Zmergujeme do původní BH
* Časová složitost je O(*logn*)

### Vyhledávací stromy a vyvažování

Obsah obrázku text

Popis byl vytvořen automaticky

#### BVS

****

**Operace**

**BVSShow(v)** – vypiš vzestupně uspořádanou posloupnost klíčů všech vrcholů T(v)

* O(|T(v)|)  
  **Obsah obrázku text

  Popis byl vytvořen automaticky**

**BVSMin(v)** – vrať vrchol obsahující minimální klíč v T(v)

* **Obsah obrázku text

  Popis byl vytvořen automaticky**
* **Obsah obrázku text

  Popis byl vytvořen automaticky**

**BVSPred (v, w)** – vrátí předchůdce

Obsah obrázku text

Popis byl vytvořen automaticky

Obsah obrázku text

Popis byl vytvořen automaticky

**BVSFind (v, x)** – vrať vrchol v T(v) s klíčem *x*, pokud takový existuje

Obsah obrázku text

Popis byl vytvořen automaticky

**BVSInsert(v, x)** – vlož prvek do T(v) s klíčem x

**Obsah obrázku text

Popis byl vytvořen automaticky**

**Předpokládáme unikátní klíče**

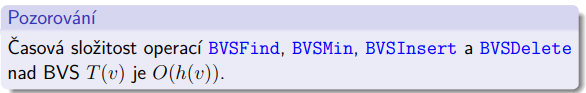
**BVSDelete(v, x)** – Odstraň z T(v) vrchol s klíčem x, pokud takový existuje

* Nejprve nalezneme vrchol s klíčem, který chceme smazat
* Při mazání může nastat několik situací
  + Pokud takový vrchol neexistuje, necháme strom beze změny.
  + Jinak, pokud je nalezený vrchol **listem**, odtrhneme ho od stromu.
  + Má-li nalezený vrchol **jednoho syna**, nahradíme ho tímto synem  
    Obsah obrázku hodinky, nůžky, kompas

    Popis byl vytvořen automaticky
  + Pokud má **dva syny**
    - Nemůžeme odstranit přímo, protože by nebylo kam připojit syny
    - Nahradíme následníkem (minimum v pravém podstromu)

Obsah obrázku text

Popis byl vytvořen automaticky

****

* V nejlepším případě logn

Obsah obrázku text

Popis byl vytvořen automaticky

Obsah obrázku text

Popis byl vytvořen automaticky

Obsah obrázku text

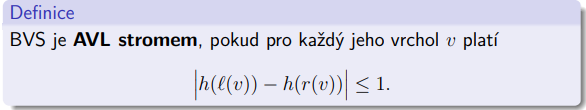
Popis byl vytvořen automaticky

* vložení: na vhodné místo, aby splňoval podmínky
* odstranění: 1 syn = přepojení svého syna na svého otce

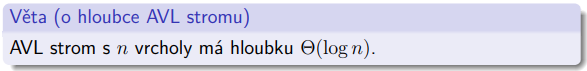
2 synové = nalezení svého předchůdce (nejpravější klíč levého

podstromu)

#### AVL



h – hloubka podstromu



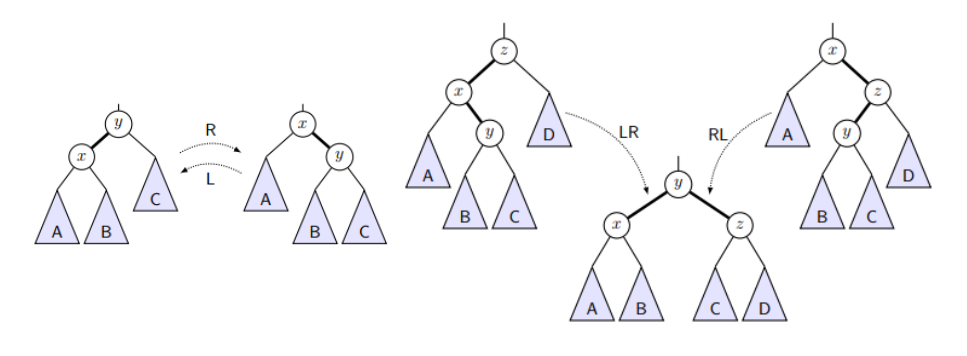
* BVSShow, BVSMin, BVSPred a BVSFind nemění ani tvar ani obsah BVS – fungují stejně pro AVL stromy
* Mění se operace pro vkládání a odstraňování – musíme kontrolovat vyváženost
* v každém vrcholu *v* je číslo *δ(v) = h(r(v)) − h(l(v))*, které nazveme **znaménko vrcholu**

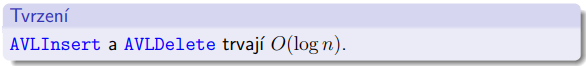
Obsah obrázku text

Popis byl vytvořen automaticky

* Jakmile při kontrole narazíme na jiné *δ(v)* strom opravíme pomocí **jednoduchých** nebo **dvojitých rotací**

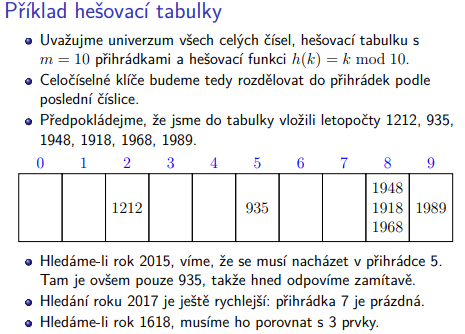
**Rotace AVL**

****



### Tabulky s rozptylováním (hešováním)

* Cílem hešování je skloubit nízké paměťové nároky operací a přitom zachovat konstantní složitost operací, i když pouze v průměrném případě.
* Pro univerzum klíčů U volíme konečnou množinu **přihrádek** P = {0, …, m-1} (**hešovací tabulku**) a vhodnou **hešovací funkci** h: U → P, která každému klíči univerza přidělí jednu přihrádku. Snažíme se, aby bylo co nejméně kolizí.
* Chceme-li uložit množinu prvků s klíči K ⊆ U, rozmístíme její prvky do přihrádek: prvek s klíčem k ∈ K umístíme do přihrádky h(k).
* Budeme-li hledat nějaký prvek s klíčem k ∈ U, víme, že nemůže být jinde než v přihrádce h(k).
* Díky poměru m a |U| (m |U|) se bude stávat, že několik prvků padne do stejné přihrádky. – tomu se říká **kolize**
* Cílem je volit m a h tak, aby se počet kolizí minimalizoval
* Hešovací tabulka
  + Pro rovnoměrné rozložení klíčů v paměti
  + Datová struktura, ze které rychle dostaneme data (database indexing, caching)
  + Hledáme pomocí indexu – index má spojitost s uloženými daty na daném indexu



#### Hash funkce

Obsah obrázku text

Popis byl vytvořen automaticky

**Příklady dobře fungujících hešovacích funkcí**

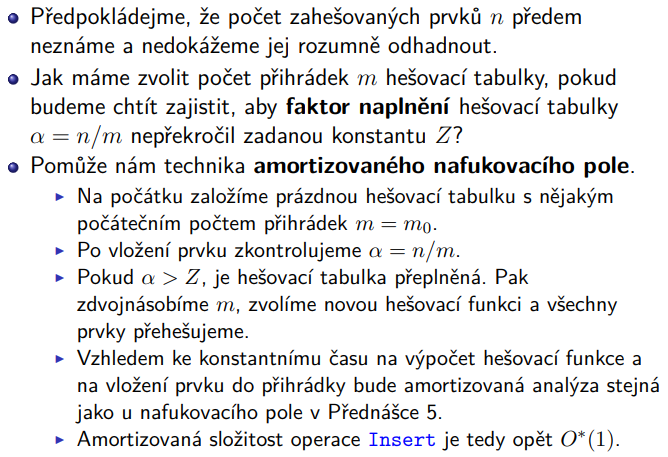
Obsah obrázku text, osoba, snímek obrazovky, dokument

Popis byl vytvořen automaticky

Obsah obrázku text

Popis byl vytvořen automaticky

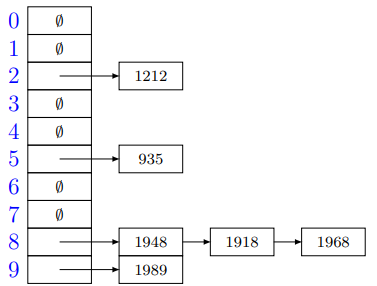
**Nafukovací hešovací tabulka a přehešování**



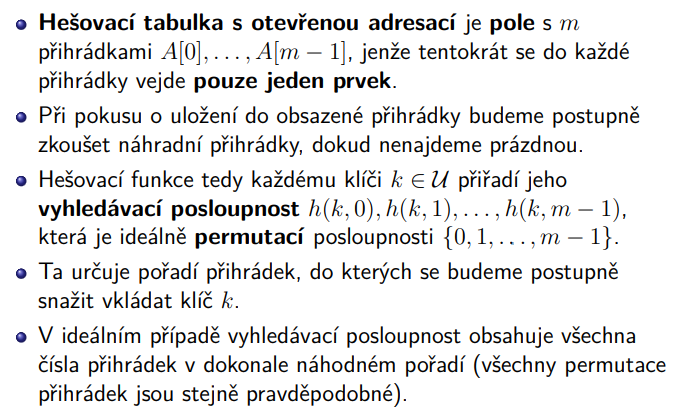
#### Řešení kolizí

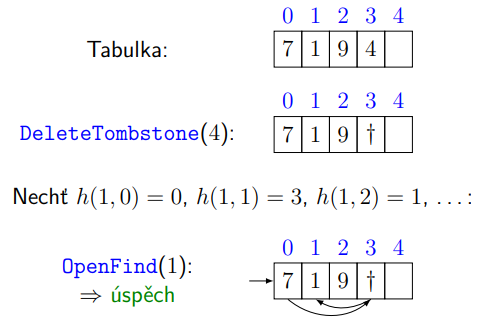
**Hešování s řetízky** (otevřené hešování – otevřené proto, že ukládáme prvky do otevřeného prostoru za hešovací tabulku – do řetízků)

* kolize jsou řešeny pomocí řetízků
* Hešovací tabulka je **pole** *m* přihrádek, které jsou buď **prázdné** nebo obsahují **ukazatel** na **řetízky** ( = spojové seznamy) uložených prvků
* Hledání, vkládání i mazání sestává z výpočtu hešovací funkce a projití řetízku v příslušné přihrádce
* V případě ideální hešovací funkce, budou mít všechny řetízky délku nejvýše horníCeláČást(n/m) prvků
* Zvolíme-li navíc počet přihrádek m = Θ(n), vyjde konstantní délka řetízku, a tím pádem i časová složitost operací



**Hešování s otevřenou adresací** (uzavřené hešování)



* tentokrát se do každé přihrádky vejde pouze jeden prvek
* pro prvek neexistuje jen jedna hash jako v předchozím případě, ale posloupnost hashů (takže když nebude v prvním, může být v druhým atd.)
* při pokusu o uložení do obsazené přihrádky budeme postupně zkoušet náhradní přihrádky, dokud nenajdeme volnou
* v případě mazání nahradíme prvek značkou (*tombstonem*) a v připadě vysokého počtu tombstonů přepočítáme celou tabulku
* možnost dvojitého hashování, kde se počítá s počty neúspěšných pokusů

Obsah obrázku text

Popis byl vytvořen automaticky

**Volba vyhledávácí posloupnosti**

**Obsah obrázku text

Popis byl vytvořen automaticky**

**Obsah obrázku text

Popis byl vytvořen automaticky**