

# C2142 Návrh algoritmů pro přírodovědce

## 6. Vyhledávací stromy, hashovací tabulky, trie.

Tomáš Raček

Jaro 2017

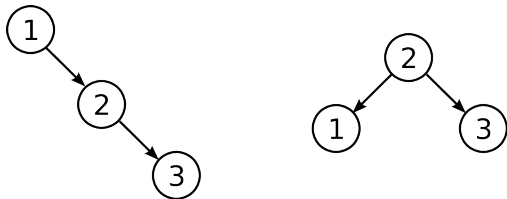
## Nevýhody BST

---

**Opakování.** Binární vyhledávací strom představuje koncepčně jednoduchou formu indexu nad daty.

Jeho hlavní nevýhoda ale může být až **lineární** pro nepříznivou posloupnost operací.

**Myšlenka.** Modifikujme operace přidání a odebrání prvku tak, aby zbytečně nezvyšovaly výšku stromu.



# AVL stromy

---

**AVL strom** je binární vyhledávací strom, který navíc splňuje následující podmínku:

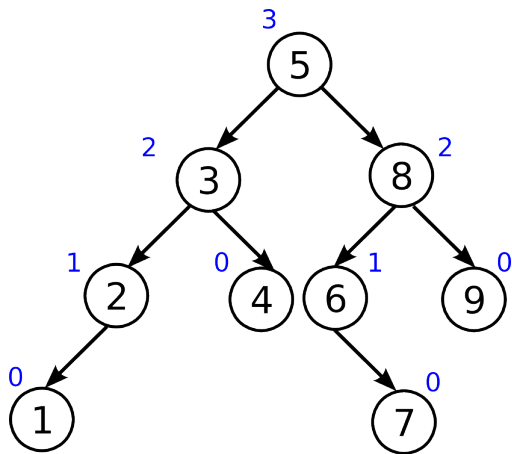
Pro každý uzel AVL stromu platí, že výška jeho levého a pravého podstromu se liší **nejvýše o 1**.

**Poznámka.** Definiční podmínka zaručuje, že výška AVL stromu je nejvýše asi  $1,5 \cdot \log(n)$ , kde  $n$  je počet jeho prvků.

**Implementace.** Každý uzel AVL stromu bude mít u sebe navíc i informaci o své výšce. V případě porušení definiční podmínky při některé z operací bude nutné strom upravit.

# AVL strom – ukázka

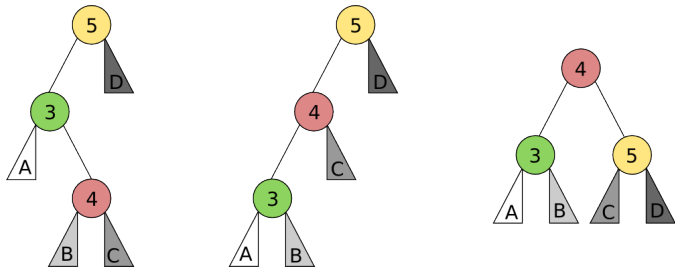
---



# Operace nad AVL stromy

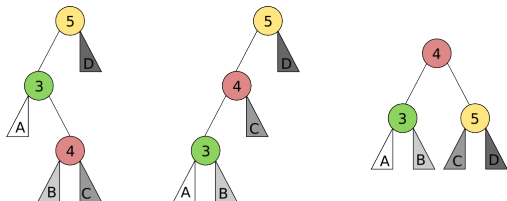
**Vyhledání prvku.** Úplně stejná operace jako v obyčejném BST. Díky zaručené výšce AVL stromu ovšem nyní nejvýše  $O(\log n)$ .

**Přidání/odebrání prvku.** Probíhá obdobně jako u BST. Pokud je narušena vyváženost AVL stromu (porušení definiční podmínky), probíhá vyvažování pomocí tzv. **rotací**.



# Operace nad AVL stromy II

## Rotace



- pouze lokální změna datové struktury
- **konstatní** složitost
- při přidání prvku stačí nejvýše 1
- při odebrání prvku je jejich počet omezen výškou stromu

**Složitost** přidání/odebrání prvku v AVL stromu je  $O(\log n)$ .

# Logaritmická výška I

---

**Shrnutí.** AVL stromy poskytují optimální z pohledu asymptotické složitosti optimální rozšíření BST.

**Poznámka.** Na BST jsou také založeny např. červeno-černé stromy, které mají logaritmickou výšku (i když větší než AVL), nicméně poskytují v praxi rychlejší vyvažování.

**Příklad.** Uvažme data o velikosti  $n = 22\,000\,000$  (přibližný počet sloučenin v databázi PubChem). Vybudujeme nad těmito daty index na bázi binárního stromu.

- výška úplného binárního stromu by byla 25
- v rámci teorie ideální, v praxi může být i to příliš

# Logaritmická výška II

---

**Nápad.** Uvažme vyvážený strom (= s logaritmickou výškou), jehož **arita**  $k$  je větší než 2.

- zvolme např.  $k = 100$
- pak výška tohoto stromu pro stejnou velikost dat bude 4

**Realizace.** Na této myšlence jsou založeny tzv. **B stromy**.

- logaritmická složitost operací
- typicky pro velké objemy dat
- existují různé varianty (B/B+/B\*)

## Využití B stromů

- souborové systémy (NTFS, Ext4, btrfs)
- indexy pro databáze



# Hashovací tabulka

---

**Shrnutí.** Vyvážené vyhledávací stromy poskytují základní operace v logaritmickém čase. Jde dosáhnout i **konstatní** složitosti?

**Nápad.** Uvažme pole o velikosti  $M$  a tzv. **hashovací funkci**, která každé hodnotě klíče přiřadí index  $i$  do tohoto pole, kde se daná položka bude nacházet.

## Hashovací funkce – příklad

- uvažme klíče  $k$  jako přirozená čísla
- pak např.  $i = H(k) = k \bmod M$
- konstatní složitost výpočtu indexu

# Hashovací funkce

---

**Ideální případ.** Uvažme hashovací tabulku o velikosti  $M$  s následujícími vlastnostmi:

- počet vkládaných prvků  $n \leq M$
- hashovací funkce má konstantní složitost
- hashovací funkce je prostá, tj.  $\forall k_1, k_2 : H(k_1) = H(k_2) \rightarrow k_1 = k_2$
- pak složitost přidání, vyhledání a odebrání prvku je **konstantní**

**Kolize.** V praxi ovšem tohoto není často možné dosáhnout, hashovací funkce nebývá prostá  $\rightarrow$  pro dva různé klíče získáme **stejnou hodnotu** hashovací funkce.

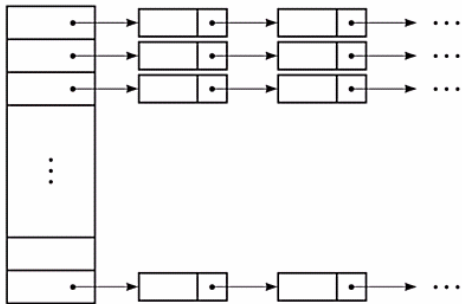
**Příklady řešení pro  $n \leq M$**

- lineární hashování – je-li index  $i$  obsazen, zkusím  $i + 1$  atd
- dvojité hashování – při kolizi volím jinou hashovací funkci

# Řešení kolizí I

---

**Řešení kolizí** v obecném případě spočívá v možnosti vytvoření spojového seznamu pro každý index pole.



# Řešení kolizí II

---

## Vlastnosti

- ideální hashovací funkce rozděluje klíče rovnoměrně
- délka každého seznamu je pak průměrně  $n/M$
- přidání prvku v  $O(1)$
- vyhledání/odebrání prvku ale v  $O(n)$  – stejné jako u spojového seznamu

V praxi je ale často **výrazně lepší** než obyčejný spojový seznam. Srovnajme několik příkladů s ideální hashovací funkcí a  $n = 5000$ :

- $M = 1000$
- $M = 5000$
- $M = 10000$

**Závěr.** Hashovací tabulka je velmi efektivní, pokud známe rozložení prvků ( $\rightarrow$  hashovací funkce) a jejich počet ( $\rightarrow$  velikost tabulky).

# Hashovací tabulka – poznámky

---

**Rozšíření.** V případě, kdy není počet prvků znám dopředu, lze měnit velikost i vlastní tabulky.

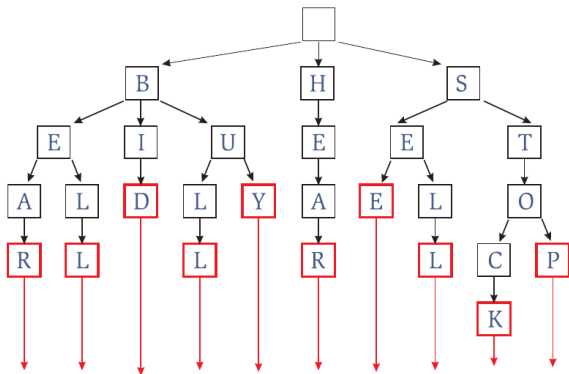
- základem je dynamické pole
- je potřeba sledovat zaplnění tabulky, tedy poměr  $n/M$
- vysoká zaplněnost přináší nižší výkon
- změna velikosti nastává při dosažení zvolené hranice zaplněnosti (např. 2/3 nebo 3/4)

## Použití

- implementuje ADT asociativní pole (slovník), které uchovává dvojice typu (klíč, hodnota)
- v Pythonu typ `dict`, v Javě `HashMap`, v C++ `std::unordered_map`

# Trie

**Trie** (prefixový strom) je stromová datová struktura zejména vhodná pro uložení klíčů, které jsou řetězci.



# Trie – příklad implementace

---

**Příklad.** Uvažme řetězce složené ze znaků anglické abecedy

- každý uzel trie obsahuje pole 26 ukazatelů na další prvky
- v každém uzlu uložíme příznak, který říká, zdali je tento uzel validním klíčem (listy jsou implicitně)

## Složitost operací

- nechť  $h$  je délka nejdelšího řetězce
- pak trie má výšku  $h$
- operace přidání, vyhledání a odebrání prvku mají všechny složitost  $O(h)$
- neúspěšné hledání může skončit v kterékoliv úrovni (srovnejme s BST)