Algoritmy 2 - B-stromy

Petr Osička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Motivace

- vrcholy stromu jsou uloženy na pomalém médiu (např. na disku)
- načítání paměti z pomalého média (do rychlého např. operační paměti) se děje po blocích; analogicky se děje zápis
- chceme minimalizovat počet bloků, které je potřeba načíst, nebo do kterých je potřeba zapsat
- přirozeně přijdeme na to, že je výhodné umístit do vrcholů více klíčů (tak, aby se velikost vrcholu zespodu co nejvíce přiblížila velikosti bloku).
- Současně ovšem chceme zachovat logaritmickou výšku stromu.

Definice

Strom parametrizujeme přirozeným číslem $t > 2$.

B strom je definován následujícími podmínkami

1. *Počet klíčů ve vrcholech*
   V každém vrcholu stromu je maximálně $2t - 1$ klíčů a minimálně $t - 1$ klíčů. Výjimkou je kořen, kde může být méně než $t - 1$ klíčů.
2. *Počet potomků*
   Pokud vrchol obsahuje n klíčů, má 0 (pak je to list) nebo $n + 1$ potomků.
3. *Hloubka listů*
   Všechny listy jsou ve stejné hloubce.
4. *Podmínka uspořádání*
   Klíče jsou ve vrcholu uspořádány vzestupně. Označíme-li klíče $k_0 < k_1 < \dots k_{n-1}$ a podstromy $\alpha_0, \alpha_1, \dots, \alpha_n$, pak
   - pro $0 \leqslant i < n - 1$ jsou klíče v podstromu α_i menší než k_i,
   - pro $0 < i \leqslant n$ jsou klíče v podstromu α_i větší než k_{i-1}.

Věta (O výšce B-stromu)

B strom s parametrem t obsahující $n \geqslant 1$ klíčů má výšku nejvýše $\log_t \frac{n+1}{2}$.

Důkaz. Kořen obsahuje alespoň jeden klíč, ostatní vrcholy obsahují alespoň $t-1$ klíčů. Strom tak má aspoň 2 vrcholy v hloubce 1, aspoň 2t vrcholů v hloubce 2, a obecně aspoň $2t^{i-1}$ vrcholů v hloubce i. Pro n tedy platí

$$
\begin{aligned}
n &\geqslant 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\
&= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\
&= 2t^h - 1
\end{aligned}
$$

Odtud dostaneme $t^h \leqslant (n+1)/2$ a obě strany logaritmujeme.

Struktury pro vrchol a strom

```
struct Node(t)
    id: [2t-1]Key — pole klíčů
    children: [2t]Node — pole potomků
    parent: Node
    n: Int
    leaf: Bool

struct Tree
    root: Node — kořen stromu
```

Kořen má položku `parent` nastavenu na `nil`.

V prázdném B-stromu není položka `root` rovna `nil`, ale je to `Node`, jehož položka `n` je rovna 0.

Vyhledávání

- Zobecnění vyhledávání ve vyhledávacím stromě. Rozhodování, do kterého podstromu se vydáme je nyní založeno na porovnávání s polem klíčů.

```
search
⟵ x: Node
⟵ k: Key
⟶ Node — vrchol, ve kterém se nachází klíč nebo nil
⟶ Int — index, na kterém se klíč nachází v polozce id
```

```
1  i ← 0
2  while (i < x.n) and (k > x.id[i])
3     i ← i + 1
4  if (i < x.n) and (k == x.id[i])
5     return x, i
6  if x.leaf
7     return nil, i
8  return search(x.children[i], k)
```

- cyklus na řádku 2 skončí pokud `i = x.n` nebo `k` $\leqslant$ `x.keys[i]`.
- Otestujeme, který z těchto důvodů to byl. Pokud to byl ten druhý a navíc `k` $=$ `x.keys[i]`, nalezli jsme požadovaný klíč.
- Na řádku 6 ošetříme situaci, kdy hledaný klíč není ve vrcholu `x`.
- Korektnost plyne z podmínky uspořádání.
- Složitost je $O(t \log_t n)$

Vložení klíče

Najdeme vrchol, do kterého vkládaný klíč patří. Následující proceduru dostaneme jednoduchou úpravou `search`.

```
find-insertion-vertex
 ⟵ r: Node
 ⟵ k: Key
 ⟶ Node — vrchol, do kterého vložíme, nutně list
```

Předpokládejme, že výsledkem volání `find-insertion-vertex` je `x`.

Pokud bychom se mohli spolehnout, že `x.n < 2t-1`, pak by stačilo provést následující.

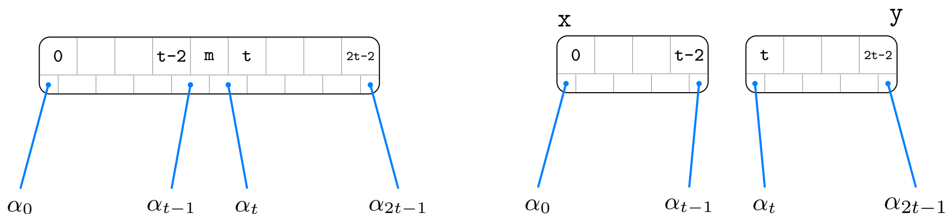
```
1 i ← 0
2 while (i < x.n) and (k > x.id[i]) do i ← i + 1
3 od indexu i posuneme obsah pole x.id o jedno políčko doprava
4 x.id[i] ← k
5 x.n ← x.n + 1
```

Vkládáme do listu a nemusíme se starat o potomky. Posun na řádku 3 můžeme bezpečně provést, protože pole `id` není obsazené a na posledním políčku není klíč.

Pokud ovšem `x.n == 2t-1`, vložením klíče do `x` bychom v tomto vrcholu měli `2t` klíčů a porušili bychom podmínku I z definice B-stromu.

Vrchol `x` má `2t-1` klíčů. Klíč `x.id[t-1]` je mediánem mezi klíči v poli `x.id`, označíme si jej `m`.

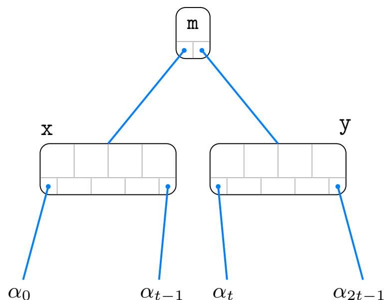
Vytvoříme nový vrchol `y`, do kterého z vrcholu `x` přesuneme klíče na indexech `t` až `2t - 2` a potomky na indexech `t` až `2*t-1`.



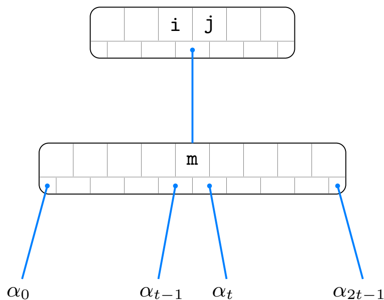
Nyní přidáme klíč k do správného vrcholu. Pokud `k < m` přidáme jej do `x`, jinak do `y`.

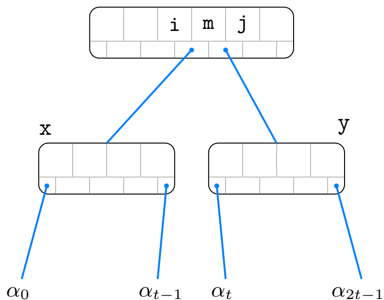
Klíč `m` umístíme do rodiče vrcholu `x`.

Speciální případ je, pokud byl `x` před rozdělením kořen. To vytvoříme nový kořen a vložíme do něj `m` jako jediný klíč. Vrcholy `x` a `y` po rozdělení pak nastavíme jako potomky nového kořene.



Pokud `x` nebyl před rozdělením kořen, vložíme `m` do vrcholu `x.parent`, přičemž vrcholy `x` a `y` jsou potomci *okolo* klíče `m`.





Pokud je ovšem `x.parent` před vložením `m` zaplněný (tj. `x.parent.n == 2t-1`), musíme jej nejdříve právě popsanou metodou rozdělit a až poté vložit `m` (a potomky `x`, `y`) do správného z rozdělených vrcholů.

Při rozdělování `x.parent` dostaneme nový medián, který pak nutno zařadit do stromu. To uděláme právě popsaným způsobem.

Můžeme provést celou kaskádu dělení vrcholů, při které rozdělíme zaplněné vrcholy na cestě do kořene. Může být nutné vytvořit i nový kořen. Toto je jediný způsob, jakým lze zvýšit výšku B-stromu.

Klíčová procedura je pro vložení klíče a *příslušných potomků* do vrcholu.

```
insert-with-subtrees
 ⟵ tr: Tree
 ⟵ x: Node — vrchol, do kterého vkládáme
 ⟵ k: Key — vkládaný klíč
 ⟵ left: Node — podstrom nalevo od klíče
 ⟵ right: Node — podstrom napravo od klíče
```

Tuto proceduru můžeme použít použít i pro `insert` (s argumenty `tr` a k).

```
1 target ← find-insertion-vertex(tr.root, k)
2 insert-with-subtrees(tr, target, k, nil, nil)
```

Procedura insert-with-subtrees

Půlení kořene

```
1  if x == nil
2     z ← Node()
3     z.id[0] ← k
4     z.leaf ← false
5     z.n ← 1
6     z.parent ← nil
7     set-child(z, left, 0)
8     set-child(z, right,1)
9     tr.root ← z
10    return
```

x *je zaplněný*

```
11 if x.n == 2*t-1
12    p ← x.parent
13    rozděl vrchol x a tím získej klíč m a vrcholy l a r.
14    if k < m
15       target ← l
16    else
17       target ← r
18    insert-with-subtrees(tr, target, k, left, right)
19    insert-with-subtrees(tr, p, m, l, r)
20    return
```

x *není zaplněný*

```
21 if x.n < 2*t-1
22    i ← 0
21    while (i < x.n) and (k > x.id[i]) do i ← i + 1
23    od indexu i posuneme obsah polí x.id a x.children o jedno políčko doprava
24    x.id[i] ← k
25    set-child(x, left, i)
26    set-child(x, right, i+1)
27    x.n ← x.n + 1
```

Procedura `set-child(x, y, i)` nastaví potomka na indexu `i` vrcholu `x` na vrchol `y`.

Složitost vložení

- Procedura `find-insertion-node` má stejnou složitost jako procedura vyhledávání. Půlení vrcholu provádíme v každé úrovni stromu nejvýše jednou, přitom jedno půlení má složitost lineárně závislou na t. Proto je složitost přidávání $O(t \cdot \log_t n)$, kde n je počet klíčů ve stromu.

Vložení prvku jedním průchodem

Vrcholy rozdělujeme už ve fázi vyhledávání (při průchodu do listu) a zabráníme tak kaskádě dělení v opačném směru (tím ušetříme opakované načítání vrcholů z pomalejší paměti).

Zajistíme, že pokud vkládáme do vrcholu x, tak x není zaplněný.

- *zaplněný kořen*
  Pokud je kořen zaplněný, rozdělíme jej a vytvoříme nový kořen. Proceduru přidávání spustíme v tomto novém koření.
- *nezaplněný vrchol*
  Předp. že aktuálně přidáváme do nezaplněného vrcholu x. Pokud je x list, pak můžeme klíč přidat bez nutnosti dělení.

  Pokud x není list a máme pokračovat do potomka y, nejdříve zkontrolujeme, jestli je y zaplněný. Pokud ano, rozdělíme jej. Toto rozdělení nemůže vést k potřebě rozdělit x, protože x není zaplněný.

Poznámky

- Složitost je $O(t \cdot \log_t n)$, strom projdeme od kořene do listu, přitom může být v každé úrovni potřeba dělit vrchol. Dělení vrcholu má složitost lineárně závislou na t.
- Při jednofázovém přidávání vlastně nepotřebujeme pointery na rodiče vrcholů. (A šlo by je tedy ze struktury pro vrchol smazat.)

Mazání klíče

Smazání klíče `k` ze stromu `tree` má 2 fáze. V první fázi klíč skutečně smažeme.

1. Pomocí `search(tree,k)` najdeme vrchol `x`, ve kterém se `k` nachází a index `i`, na kterém se `k` nachází v poli `x.keys`.
2. `x` *je list*
   V poli `x.keys` posuneme prvky s indexem větším než `i` o jedno políčko doleva. Položku `x.n` dekrementujeme o 1.
3. `x` *není list*
   Najdeme v podstromu `x.children[i+1]` minimální klíč `m`. Toto je pořádkový následník klíče `k` ve stromu. Tímto klíčem nahradíme ve vrcholu `x` klíč na indexu `i`.

   Poté rekurzivně smažeme klíč `m` z listu, ve kterém se vyskytuje.

V obou případech ubude klíč v listu. Může se stát, že po smazání má list `t-2` klíčů a došlo k porušení podmínky 1 z definice B-stromu.

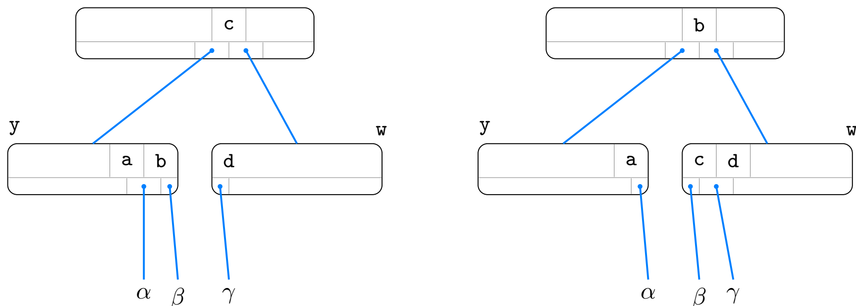
Ve druhé fázi algoritmu mazání upravujeme nekorektní počty klíčů ve vrcholech. Aktuálně upravovaný vrchol označíme `w`. Na začátku fáze je `w` list, ze kterého jsme odebrali klíč v první fázi.

1. *Můžeme skončit?*
   Pokud je `w` kořen, nebo (`w.n` $\geqslant$ `t-1`) algoritmus končí.

2 *Převod klíče ze sourozence*

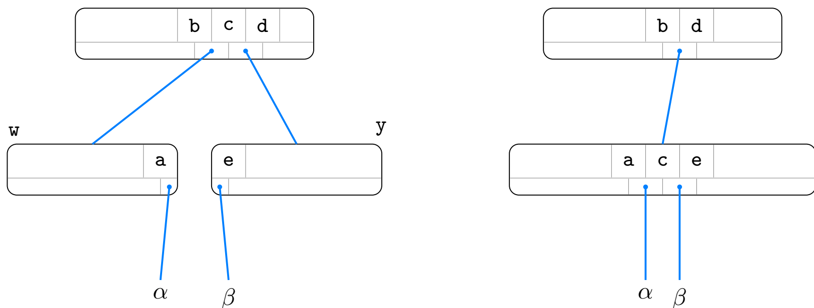
Lze provést, pokud `w` má sourozence `y`, jehož index v rodičově poli `children` se liší o 1 (takovému sourozenci budeme říkat soused), takového, že `(y.n > t-1)`.

Přesuneme do `w` jeden klíč z vrcholu `w.parent`, a ten zase nahradíme klíčem z vrcholu `y`. Mezi `y` a `w` musíme také přesunout jednoho potomka (tj. provedeme analogii rotace). Vrcholu `y` ubude jeden klíč. Existují dvě symetrické varianty této úpravy (podle toho, jestli je `y` napravo nebo nalevo od `w`), níže je zobrazena jedna z nich.



3. *Sloučení se sourozencem*

Nelze provést předchozí bod, protože sousedi vrcholu `w` obsahují `t-1` klíčů. Vybereme si jednoho ze sousedů, označíme ho `y`, a spojíme `w` a `y` do nového vrcholu `z`. K tomu musíme z rodiče vrcholu `w` odebrat klíč, který leží mezi `w` a `y` a vložit jej do nového vrcholu.



Pokud je `w.parent` kořen, ze kterého jsme odebrali poslední klíč, je novým kořenem `z`. Jinak nastavíme `w` $\leftarrow$ `z` a pokračujeme krokem 1.

Složitost

První průchod je vyhledávání klíče (a případná operace vyhledání minima). Ten je v čase $O(t \log_t n)$. Samotné smazání klíče z vrcholu je v čase $O(t)$.

V druhé fázi procházíme strom směrem ke kořeni, v každém level děláme $O(t)$ práce. Celkem tedy $O(t \log_t n)$.

Složitost mazání je tak $O(t \log_t n)$.

Mazání jedním průchodem

Aktuální vrchol je x a mažeme klíč k.

Budeme předpokládat, že x obsahuje alespoň t klíčů. (*Po odebrání jednoho klíče jich bude mít pořád dostatek*).

1. Pokud je x list obsahující k, klíč smažeme standardním způsobem.
2. Pokud x není list a obsahuje klíč k
   a. *Levý potomek* y *klíče* k *má alespoň* t *klíčů.*
   Najdeme pořádkového předchůdce l klíče k a rekurzivně jej smažeme z vrcholu y. (Lze provést najednou jedním průchodem! Nevíme, dopředu, který klíč mažeme, ale víme, kde ve stromu je. Je to maximum v podstromu s kořenem y). Potom ve vrcholu x nahradíme k klíčem l.
   b. *Pravý potomek* z *klíče* k *má alespoň* t *klíčů.*
   Analogicky předchozímu bodu, pouze hledáme pořádkového následníka.
   c. *Oba potomci (*y *a* z*) mají* t - 1 *klíčů.*
   Spojíme y a z do jednoho vrcholu w, tak jako ve dvouprůchodovém mazání. Tím z vrcholu x ztratíme klíč k, který se přesune do w (ale x má pořád dostatek klíčů). Rekurzivně mažeme klíč k z vrcholu w.

3. x není list a neobsahuje klíč k. Klíč k se nachází v podstromu, jehož kořenem je potomek y vrcholu x.
   a. y *má* t-1 *klíčů a má souseda* z *s alespoň* t *klíči.*
      Rotací převedeme jeden klíč z vrcholu z, tak jako při dvouprůchodovém mazání.
   b. y *má* t-1 *klíčů a nemá souseda s alespoň* t *klíči.*
      Sloučíme y s jedním sousedem tak jako při dvouprůchodovém mazání (a ubereme tím jeden klíč z x). Výsledný vrchol označíme y.

   Rekurzivně mažeme klíč k z vrcholu y.

Pokud se v kroku 2 c) nebo 3 b) stane, že (v případě, že x je kořen obsahující jeden klíč) vytvoříme kořen bez klíčů, stane se v tomto kroku nově vytvořený vrchol novým kořenem stromu.

Vezmeme-li v úvahu předchozí odstavec, je předpoklad, že x má dostatečný počet klíčů v průběhu mazání vždy zajištěn.