# 1. Vyhledávací stromy

V algoritmech potřebujeme zacházet s různými druhy dat – posloupnostmi, množinami, grafy, ... Často máme k dispozici více způsobů, jak tato data uložit do paměti počítače. Mohou se lišit spotřebou paměti, ale také rychlostí různých operací s daty. Volíme tedy podle toho, jaké operace využívá konkrétní algoritmus.

Otázky tohoto druhu se přitom opakují. Proto je zkoumáme obecně, což vede ke studiu datových struktur. V této kapitole se budeme věnovat jedné z nejznámějších datových struktur, totiž vyhledávacím stromům.

## 1.1. Datové struktury

Nejprve si rozmysleme, co od datové struktury očekáváme. Z pohledu programu má struktura jasné rozhraní: reprezentuje nějaký druh dat a umí s ním provádět určité operace. Uvnitř datové struktury pak volíme konkrétní uložení dat v paměti a algoritmy pro provádění jednotlivých operací. Z toho pak plyne prostorová složitost struktury a časová složitost operací.

**Posloupnosti**

Jednoduchým příkladem je *posloupnost* celých čísel. Její rozhraní může vypadat takto:

| | |
|---|---|
| INDEX($i$) | vrátí $i$-tý prvek posloupnosti |
| SET($i, x$) | nastaví hodnotu $i$-tého prvku na $x$ |
| FIND($x$) | najde první prvek s hodnotou $x$ |
| INSERTAT($i, x$) | na $i$-tou pozici vloží $x$, následující prvky se posunou |
| DELETEAT($i$) | smaže $i$-tý prvek, následující prvky se posunou |

Pokud posloupnost uložíme do *pole,* operace INDEX a SET budou pracovat v konstantním čase, zatímco FIND, INSERTAT a DELETEAT v lineárním, neboť v nejhorším případě musíme projít celé pole.

Hledání můžeme zrychlit *uspořádáním* (setříděním) pole. Pak může FIND binárně vyhledávat v logaritmickém čase, ovšem vkládání i mazání zůstanou lineární.

Použijeme-li *spojový seznam,* všechny operace budou lineární, neboť v seznamu neumíme efektivně najít $i$-tý prvek. Pokud prvky seznamu propojíme obousměrně a upravíme INSERTAT a DELETEAT, aby místo polohy prvku dostaly ukazatel na už nalezený prvek, klesne jejich složitost na konstantní.

Není samozřejmě potřeba omezovat se na posloupnosti celých čísel. Můžeme si zvolit nějaké jiné *universum* – množinu, z níž budou pocházet prvky. Ve zbytku kapitoly budeme předpokládat, že prvky zvoleného universa lze v konstantním čase přiřazovat a porovnávat na rovnost a „je menší než“. Fix!

**Fix:** Co když čas není konstantní? Nějaký příklad na řetězce ve stromech? Cvičení?

### Fronty obyčejné a prioritní

*Fronta* si také pamatuje posloupnost prvků, ale dovede pouze přidávat nové prvky na konec posloupnosti a staré odebírat ze začátku. Pokud ji implementujeme jako seznam, obojí zvládne v konstantním čase.

Zajímavější je „fronta s předbíháním“, obvykle se jí říká *prioritní fronta.* Každý prvek má přiřazenu *prioritu* a na řadu vždy přijde prvek s nejvyšší prioritou. Operace vypadají následovně:

| | |
|---|---|
| ENQUEUE$(x, p)$ | přidá do fronty prvek $x$ s prioritou $p$ |
| DEQUEUE | nalezne prvek s nejvyšší prioritou a odebere ho (pokud je takových prvků víc, vybere libovolný z nich) |

Prioritní frontu lze reprezentovat polem nebo seznamem, ale nalezení maxima z priorit bude pomalé – v $n$-prvkové frontě $\Theta(n)$. Raději použijeme *haldu*, s níž dosáhneme časové složitosti $\mathcal{O}(\log n)$ pro obě operace. Ref

### Množiny a slovníky

*Množina* obsahuje konečný počet prvků universa a nabízí následujicí operace:

| | |
|---|---|
| MEMBER$(x)$ | zjistí, zda $x$ leží v množině (někdy též FIND$(x)$) |
| INSERT$(x)$ | vloží $x$ do množiny (pokud tam už bylo, nestane se nic) |
| DELETE$(x)$ | odebere $x$ z množiny (pokud tam nebylo, nestane se nic) |

Zobecněním množiny je *slovník.* Ten si pamatuje konečnou množinu *klíčů* a každému z nich přiřazuje *hodnotu* (to může být prvek nějakého dalšího universa, nebo třeba ukazatel na jinou datovou strukturu). Slovník je tedy konečná množina dvojic (*klíč*, *hodnota*), v níž se neopakují klíče. Typické slovníkové operace jsou tyto:

| | |
|---|---|
| GET$(x)$ | zjisti, jaká hodnota je přiřazena klíči $x$ (pokud nějaká) |
| SET$(x, y)$ | přiřadí klíči $x$ hodnotu $y$; pokud už nějaká dvojice s klíčem $x$ existovala, tak ji nahradí |
| DELETE$(x)$ | smaže dvojici s klíčem $x$ (pokud existovala) |

Někdy nás také zajímá vzájemné pořadí prvků – tehdy definujeme *uspořádanou množinu,* která má navíc tyto operace:

| | |
|---|---|
| MIN$(x)$ | vrátí nejmenší hodnotu v množině |
| MAX$(x)$ | vrátí největší hodnotu v množině |
| PRED$(x)$ | vrátí největší prvek menší než $x$, nebo řekne, že takový není |
| SUCC$(x)$ | vrátí nejmenší prvek větší než $x$, nebo řekne, že takový není |

Obdobně můžeme zavést uspořádané slovníky.

Množiny a slovníky můžeme reprezentovat pomocí polí a seznamů, ale jak ukazuje následující tabulka, vždy je část operací pomalá. V této kapitole proto vybudujeme vyhledávací stromy, které budou mnohem efektivnější. Abychom věděli, na

co se těšit, prozradíme už teď složitosti jednotlivých operací:

| | Insert | Delete | Member | Min | Pred |
|---|---|---|---|---|---|
| pole | $\Theta(1)^*$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| uspořádané pole | $\Theta(n)$ | $\Theta(n)$ | $\Theta(\log n)$ | $\Theta(1)$ | $\Theta(\log n)$ |
| spojový seznam | $\Theta(1)^*$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| uspořádaný seznam | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ | $\Theta(n)$ |
| vyhledávací strom | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ |

Operace Max a Succ jsou stejně rychlé jako Min a Pred. Složitosti označené hvězdičkou platí jen tehdy, slíbíme-li, že se prvek v množině dosud nenachází; v opačném případě je potřeba provést ještě Member. U polí předpokládáme, že dopředu známe horní odhad velikosti množiny. **Fix!**

## 1.2. Binární vyhledávací stromy

Jak jsme viděli, uspořádané pole umí rychle vyhledávat, ale veškeré změny trvají dlouho. Pokusíme se proto od pole přejít k obecnější struktuře, která bude „pružnější“.

Zavzpomínejme, jak se hledá v uspořádaném poli. Zvolíme prvek uprostřed pole, porovnáme ho s hledaným a podle výsledku porovnání se zaměříme buďto na levý nebo na pravý interval. Tam opakujeme stejný algoritmus. Jelikož velikosti intervalů klesají exponenciálně, zastavíme se po $\mathcal{O}(\log n)$ krocích.

Možné průběhy vyhledávání můžeme popsat stromem. Kořen stromu odpovídá prvnímu porovnání: obsahuje prostřední prvek pole a má dva syny – *levého* a *pravého*. Ti odpovídají dvěma možným výsledkům porovnání: pokud je hledaná hodnota menší, jdeme doleva; pokud větší, tak doprava. Následující vrchol nám řekne, jaké další porovnání máme provést, a tak dále až do doby, kdy buďto nastane rovnost (takže jsme našli), nebo se pokusíme přejít do neexistujícího vrcholu (takže hledaná hodnota v poli není). **Fix!**

Pro úspěšné vyhledávání přitom nepotřebujeme, abychom z každého intervalu vybrali právě ten prostřední prvek. Pokud bychom volili jinak, dostaneme odlišný strom. Pomocí něj také půjde hledat, jen možná pomaleji – to je vidět na následujícím obrázku.

Co všechno musí strom splňovat, aby se podle něj dalo hledat, přetavíme do nasledujících definic.

**Definice:** Strom $T$ nazveme *binární*, pokud je zakořeněný a každý vrchol $v \in V(T)$ má nejvýše dva syny, u nichž rozlišujeme, který je levý a který pravý.

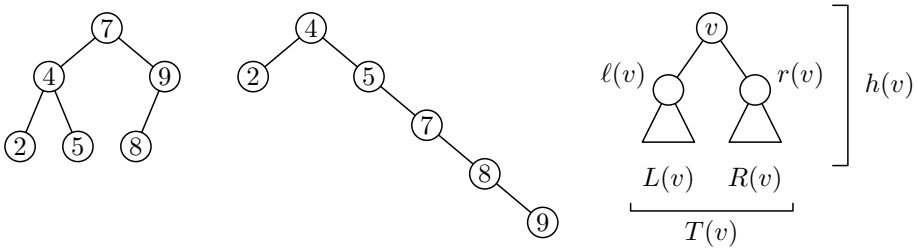
**Definice:** Pro vrchol $v$ binárního stromu $T$ značíme:

- $\ell(v)$ a $r(v)$ – levý a pravý syn vrcholu $v$

---

**Fix:** Odkaz na amortizované natahování pole.

**Fix:** Zmínit rozhodovací stromy z kapitoly o třídění.




Obr. 1.1: Dva binární vyhledávací stromy a jejich značení

- $L(v)$ a $R(v)$ – levý a pravý podstrom vrcholu $v$
- $T(v)$ – podstrom obsahující vrchol $v$ a všechny jeho potomky
- $h(v)$ – hloubka stromu $T(v)$, čili maximum z délek cest z $v$ do listů

Pokud vrchol nemá levého syna, položíme $\ell(v) = \emptyset$ a podobně pro $r(v)$ a $p(v)$. Pak se hodí dodefinovat, že $T(\emptyset)$ je prázdný strom a $h(\emptyset) = -1$.

**Definice:** *Binární vyhledávací strom* (BVS) je binární strom, jehož každému vrcholu $v$ přiřadíme unikátní *klíč* $k(v)$ z universa. Přitom musí pro každý vrchol $v$ platit:

- Kdykoliv $u \in L(v)$, pak $k(u) < k(v)$.
- Kdykoliv $u \in R(v)$, pak $k(u) > k(v)$.

### Operace s BVS

Pomocí vyhledavacích stromů můžeme přirozeně reprezentovat množiny: klíče uložené ve vrcholech budou odpovídat prvkům množiny. A kdybychom místo množiny chtěli slovník, přidáme do vrcholu hodnotu přiřazenou danému klíči.

Nyní ukážeme, jak provádět jednotlivé množinové operace. Jelikož stromy jsou definované rekurzivně, je přirozené zacházet s nimi rekurzivními funkcemi. Dobře je to vidět na následující funkci, která vypíše všechny prvky množiny. Definice BVS nám dokonce zaručuje, že budou vypsány v pořadí od nejmenšího k největšímu.

**Procedura** BvsShow
*Vstup:* Kořen BVS $v$

1. Pokud $v = \emptyset$, jedná se o prázdný strom a hned skončíme.
2. Zavoláme BvsShow($\ell(v)$).
3. Vypíšeme klíč uložený ve vrcholu $v$.
4. Zavoláme BvsShow($r(v)$).

Funkce BvsFind slouží k nalezení vrcholu s daným klíčem $x$. Prochází stromem od kořene a každý vrchol $v$ porovná s $x$. Pokud je $x < k(v)$, pak se podle definice nemůže $x$ nacházet v pravém podstromu, takže pokračujeme doleva. Je-li naopak $x > k(v)$, nic nepokazíme krokem doprava. V konečném čase proto $x$ buďto najdeme, nebo vyloučíme všechny možnosti, kde by se mohlo nacházet.

BvsFind formulujeme jako rekurzivní funkci, kterou vždy voláme na kořen nějakého podstromu a vrátí nám nový kořen.

**Procedura** BvsFind
*Vstup:* Kořen BVS $v$, hledaný klíč $x$

1. Pokud $v = \emptyset$, vrátíme $\emptyset$.
2. Pokud $x = k(v)$, vrátíme $v$.
3. Pokud $x < k(v)$, vrátíme BvsFind$(\ell(v), x)$.
4. Pokud $x > k(v)$, vrátíme BvsFind$(r(v), x)$.

*Výstup:* Vrchol s klíčem $x$, anebo $\emptyset$.

Minimum z prvků množiny spočteme snadno: půjdeme stále doleva, dokud to jde. Klíče menší než ten aktuální se totiž mohou nacházet pouze v levém podstromu.

**Procedura** BvsMin
*Vstup:* Kořen BVS $v$

1. Pokud $\ell(v) = \emptyset$, vrátíme vrchol $v$.
2. Jinak vrátíme BvsMin$(\ell(v))$.

*Výstup:* Vrchol obsahující nejmenší klíč

Vkládání nového prvku funguje velmi podobně jako vyhledávání s tím rozdílem, že v okamžiku, kdy by vyhledávací algoritmus měl přejít do neexistujícího vrcholu, tento nový vrchol vytvoříme a vložíme do něj vkládaný prvek. Rozmyslíme si, že toto je jediné místo, kde podle definice nový prvek smí ležet.

**Procedura** BvsInsert
*Vstup:* Kořen BVS $v$, vkládaný klíč $x$

1. Pokud $v = \emptyset$, vytvoříme nový vrchol $v$ s klíčem $x$.
2. Pokud $x < k(v)$, položíme $\ell(v) \leftarrow$ BvsInsert$(\ell(v), x)$.
3. Pokud $x > k(v)$, položíme $r(v) \leftarrow$ BvsInsert$(r(v), x)$.
4. Pokud $x = k(v)$, klíč $x$ se ve stromu již nachází, není třeba nic měnit.

*Výstup:* Nový kořen $v$

Při mazání může nastat několik různých případů. Nechť $v$ je vrchol, který chceme smazat. Je-li $v$ list, můžeme tento list prostě odstranit, čímž vlastně provedeme operaci opačnou k BvsInsertu. Má-li $v$ právě jednoho syna, postačí $v$ nahradit tímto synem.

Ošemetný je případ se dvěma syny. Tehdy totiž nemůžeme $v$ jen tak smazat, jelikož by tyto syny nebylo kam připojit. Proto nalezneme následníka vrcholu $v$, což je nejlevější vrchol v pravém podstromu. Ten má nejvýše jednoho syna, takže ho smažeme místo $v$ a jeho hodnotu přesuneme do $v$.

**Procedura** BvsDelete
*Vstup:* Kořen BVS $v$, mazaný klíč $x$

1. Pokud $v = \emptyset$, vrátíme $\emptyset$. *(Klíč $x$ ve stromu nebyl.)*
2. Pokud $x < k(v)$, položíme $\ell(v) \leftarrow$ BvsDelete$(\ell(v), x)$.

3. Pokud $x > k(v)$, položíme $r(v) \leftarrow \textsc{BvsDelete}(r(v), x)$.
4. Pokud $x = k(v)$: *(Chystáme se smazat vrchol $v$.)*
5. Pokud $\ell(v) = r(v) = \emptyset$, vrátíme $\emptyset$. *(Byl to list.)*
6. Pokud $\ell(v) = \emptyset$, vrátíme $r(v)$. *(Existoval jen pravý syn.)*
7. Pokud $r(v) = \emptyset$, vrátíme $\ell(v)$. *(Existoval jen levý syn.)*
8. $s \leftarrow \textsc{BvsMin}(r(v))$ *(Máme oba syny: nalezneme následníka $s$.)*
9. $k(v) \leftarrow k(s)$.
10. $r(v) \leftarrow \textsc{BvsDelete}(r(v), s)$
11. Vrátíme $v$.

*Výstup:* Nový kořen $v$

**Vyváženost stromů**

Zamysleme se nad složitostí stromových operací pro strom na $n$ vrcholech.

BvsShow projde všechny vrcholy a v každém stráví konstantní čas, takže běží v čase $\Theta(n)$.

Ostatní operace projdou po nějaké cestě od kořene směrem k listům, a to buďto jednou, nebo (v nejsložitějším případě BvsDelete) dvakrát. Jejich časová složitost proto bude $\Theta(\textit{hloubka stromu})$.

Hloubka přitom závisí na tom, jak moc je strom „košatý“. V příznivém případě vyjde sympatických $\mathcal{O}(\log n)$, ovšem dalšími operacemi může může strom degenerovat. Například začneme-li s prázdným stromem a postupně vložíme klíče $1, \ldots, n$ v tomto pořadí, vznikne „liána“ hloubky $\Theta(n)$.

Budeme se proto snažit stromy *vyvažovat*, aby jejich hloubka příliš nerostla. Zkusme se opět držet paralely s binárním vyhledáváním.

**Definice:** Binární vyhledávací strom nazveme *dokonale vyvážený*, pokud pro každý jeho vrchol $v$ platí

$$\big||L(v)| - |P(v)|\big| \leq 1.$$

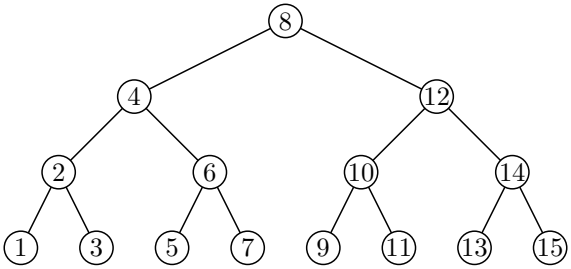
Jinými slovy velikost levého a pravého podstromu se smí lišit nejvýše o 1 vrchol.

**Pozorování:** Dokonale vyvážený strom má hloubku $\lfloor \log_2 n \rfloor$, jelikož na kterékoliv cestě z kořene do listu velikost podstromů s každým krokem klesá alespoň dvakrát.

Dokonale vyvážený strom tedy zaručuje rychlé vyhledávání. Navíc pokud všechny prvky množiny známe předem, můžeme si takový strom snadno pořídit: z uspořádané posloupnosti ho vytvoříme v lineárním čase (viz cvičení 1.2.3). Tím bohužel dobré zprávy končí: ukážeme, že po vložení nebo smazání vrcholu nelze dokonalou vyváženost obnovit rychle.

**Věta:** Pro každou implementaci operací Insert a Delete v dokonale vyváženém stromu platí, že buď Insert nebo Delete trvá $\Omega(n)$.

*Důkaz:* Nejprve si představíme, jak bude vypadat dokonale vyvážený BVS s klíči $1, \ldots, n$, kde $n = 2^k - 1$. Sledujme obrázek 1.2. Tvar stromu je určen jednoznačně: Kořenem musí být prostřední z klíčů (jinak by se levý a pravý podstrom kořene lišily o více než 1 vrchol). Levý a pravý podstrom proto mají právě $(n-1)/2 =$




Obr. 1.2: Dokonale vyvážený BVS

$2^{k-1}-1$ vrcholů, takže jejich kořeny jsou opět určeny jednoznačně a tak dále. Navíc si všimneme, že všechna lichá čísla jsou umístěna v listech stromu.

Nyní na tomto stromu provedeme následující posloupnost operací:

$$\textsc{Insert}(n+1),\ \textsc{Delete}(1),\ \textsc{Insert}(n+2),\ \textsc{Delete}(2),\ \ldots$$

Po provedení $i$-té dvojice operací bude strom obsahovat hodnoty $i+1,\ldots,i+n$. Podle toho, zda je $i$ sudé nebo liché, se budou v listech nacházet buď všechna sudá, nebo všechna lichá čísla. Pokaždé se proto všem vrcholům změní, zda jsou listy, na což je potřeba upravit $\Omega(n)$ ukazatelů. Tedy aspoň jedna z operací Insert a Delete trvá $\Omega(n)$. □

**Cvičení**

1. Rekurze je pro operace s BVS přirozená, ale v některých programovacích jazycích je pomalejší než obyčejný cyklus. Navrhněte, jak operace s BVS naprogramovat nerekurzivně.
2. Místo vrcholu se dvěma syny jsme mazali jeho následníka. Samozřejmě bychom si místo toho mohli vybrat předchůdce. Jak by se algoritmus změnil?
3. Navrhněte algoritmus, který ze setříděného pole vyrobí v lineárním čase dokonale vyvážený BVS.
4. Navrhněte algoritmus, který v lineárním čase zadaný BVS dokonale vyváží.
5.** Vyřešte předchozí cvičení tak, aby vám kromě zadaného stromu stačilo konstantní množství paměti. Pokud nevíte, jak na to, zkuste to nejprve s logaritmickou pamětí.
6. Navrhněte algoritmus, který dostane dva BVS $T_1$ a $T_2$ sloučí jejich obsah do jediného BVS. Algoritmus by měl pracovat v čase $\mathcal{O}(|T_1|+|T_2|)$.
7. Navrhněte operaci BvsSplit, která dostane BVS $T$ a hodnotu $s$, a rozdělí strom na dva BVS $T_1$ a $T_2$ takové, že hodnoty v $T_1$ jsou menší než $s$ a hodnoty v $T_2$ jsou větší než $s$.
8. Naše tvrzení o náročnosti operací Insert a Delete v dokonale vyváženém stromu lze ještě zesílit. Dokažte, že lineární musí být složitost *obou* operací.

## 1.3. Hloubkové vyvážení: AVL stromy

Zjistili jsme, že dokonale vyvážené stromy nelze efektivně udržovat. Důvodem je, že jejich definice velmi striktně omezuje tvar stromu, takže i vložení jediného klíče může vynutit přebudování celého stromu. Zavedeme proto o trochu slabší podmínku.

**Definice:** Binární vyhledávací strom nazveme *hloubkově vyvážený*, pokud pro každý jeho vrchol $v$ platí

$$\big|h(\ell(v)) - h(r(v))\big| \leq 1.$$

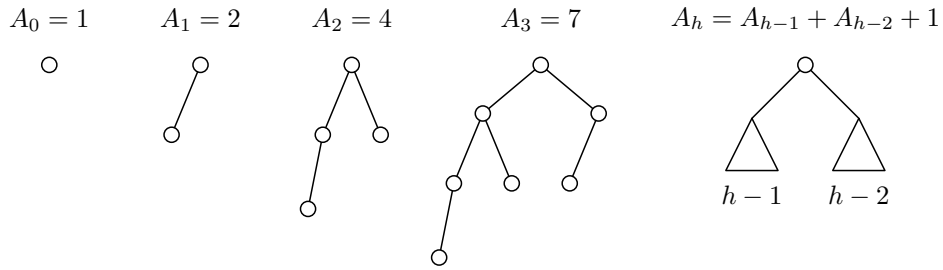
Jinými slovy hloubka levého a pravého podstromu se vždy musí lišit nejvýše o jedna.

Stromům s hloubkovým vyvážením se říká *AVL stromy,* neboť je vymysleli v roce 1962 ruští matematikové G. M. Adělson-Veľskij a E. M. Landis. Nyní dokážeme, že AVL stromy mají logaritmickou hloubku.

**Tvrzení:** AVL strom na $n$ vrcholech má hloubku $\Theta(\log n)$.

*Důkaz:* Nejprve pro každé $h \geq 0$ stanovíme $A_h$, což bude minimální možný počet vrcholů v AVL stromu hloubky $h$, a dokážeme, že tento počet roste s hloubkou exponenciálně.

Pro malá $h$ stačí rozebrat možné případy podle obrázku 1.4.




Obr. 1.4: Minimální AVL stromy pro hloubky 0 až 3 a obecný případ

Pro větší $h$ uvažujme, jak může minimální AVL strom o $h$ hladinách vypadat. Jeho kořen musí mít dva podstromy, jeden z nich hloubky $h-1$ a druhý hloubky $h-2$ (kdyby měl také $h-1$, měl by zbytečně mnoho vrcholů). Oba tyto podstromy musí být minimální AVL stromy dané hloubky. Musí tedy platit $A_h = A_{h-1} + A_{h-2} + 1$.

To je rekurence podobná té z Fibonacciho posloupnosti. Vskutku: platí $A_h = F_{h+3} - 1$, kde $F_k$ je $k$-té Fibonacciho číslo. Z toho bychom mohli získat explicitní vzorec pro $A_h$, ale pro dukaz našeho tvrzení postačí jednodušší asymptotický odhad.

Dokážeme indukcí, že $A_h \geq 2^{h/2}$. Jistě je $A_0 = 4 \geq 2^{0/2} = 1$ a $A_1 = 2 \geq 2^{1/2} \doteq 1.414$. Indukční krok pak vypadá následovně:

$$A_h = 1 + A_{h-1} + A_{h-2} > 2^{\frac{h-1}{2}} + 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}} \cdot (2^{-\frac{1}{2}} + 2^{-1}) \geq 2^{\frac{h}{2}} \cdot 1.2 > 2^{\frac{h}{2}}.$$

Tím jsme dokázali, že $A_h \geq c^h$ pro $c = \sqrt{2}$. Proto AVL strom o $n$ vrcholech může mít nejvýše $\log_c n$ hladin – kdyby jich měl více, obsahoval by více než $c^{\log_c n} = n$ vrcholů.

Zbývá dokázat, že logaritmická hloubka je také nutná. K tomu dojdeme podobně: nahlédneme, že největší možný AVL strom hloubky $h$ je úplný binární strom s $2^{h+1} - 1$ vrcholy. Tudíž minimální možná hloubka AVL stromu je $\Omega(\log n)$. □

**Vyvažování rotacemi**

Jak budou vypadat operace na AVL stromech? FIND bude totožný. Operace INSERT a DELETE začnou stejně jako u obecného BVS, ale poté ještě ověří, zda strom zůstal hloubkově vyvažený, a případně zasáhnou, aby se vyváženost obnovila.

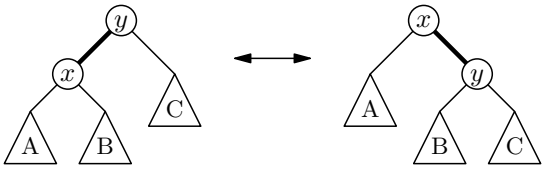
Abychom poznali, kdy je zásah potřeba, budeme v každém vrcholu $v$ udržovat číslo $\delta(v) = h(r(v)) - h(\ell(v))$. To je takzvané *znaménko* vrcholu, které v korektním AVL stromu může nabývat jen těchto hodnot:

- $\delta(v) = 1$ (pravý podstrom je hlubší) – takový vrchol značíme $\oplus$,
- $\delta(v) = -1$ (levý podstrom hlubší) – značíme $\ominus$,
- $\delta(v) = 0$ (oba podstromy stejně hluboké) – značíme $\odot$.

Jakmile narazíme na jiné $\delta(v)$, strom opravíme provedením jedné nebo více rotací.

*Rotace* je operace, která „otočí“ hranu mezi dvěma vrcholy a přepojí jejich podstromy tak, aby byli i nadále synové vzhledem k otcům správně uspořádáni. To lze provést jediným způsobem, který najdete na obrázku 1.5.

Často také potkáme *dvojitou rotaci* z obrázku 1.6. Tu lze složit ze dvou jednoduchých rotací, ale bývá přehlednější uvažovat o ní vcelku jako o „překořenění“ celé konfigurace za vrchol $y$.

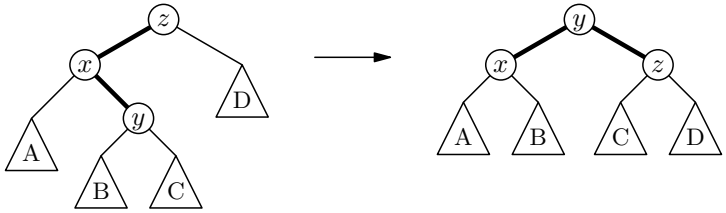



Obr. 1.5: Jednoduchá rotace

**Vkládání do stromu**

Nový prvek vložíme jako list se znaménkem $\odot$. Tím se z prázdného podstromu hloubky $-1$ stal jednovrcholový podstrom hloubky 0, takže může být potřeba přepočítat znaménka na cestě ke kořeni.

Proto se budeme vracet do kořene a propagovat do vyšších pater informaci o tom, že se podstrom prohloubil. (To můžeme elegantně provést během návratu z rekurze v proceduře BVSINSERT.)




Obr. 1.6: Dvojitá rotace

Ukážeme, jak bude vypadat jeden krok. Nechť do nějakého vrcholu $x$ přišla z jeho syna informace o prohloubení podstromu. Bez újmy na obecnosti se jednalo o levého syna – v opačném případě provedeme vše zrcadlově a prohodíme roli znamének $\oplus$ a $\ominus$. Rozlišíme několik případů.

*Případ 1:* Vrchol $x$ měl znaménko $\oplus$.

- Hloubka levého podstromu se právě vyrovnala s hloubkou pravého, čili znaménko $x$ se změní na $\odot$.
- Hloubka podstromu $T(x)$ se nezměnila, takže propagování informace ukončíme.

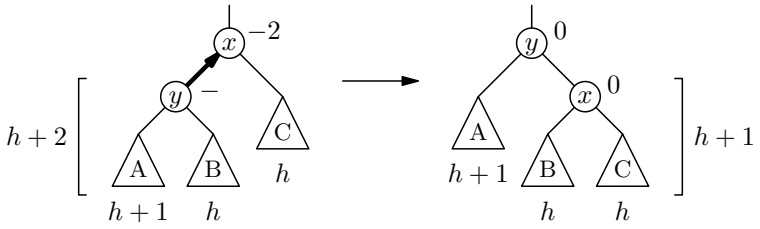
*Případ 2:* Vrchol $x$ měl znaménko $\odot$.

- Znaménko $x$ se změní na $\ominus$.
- Hloubka podstromu $T(x)$ vzrostla, takže v propagování musíme pokračovat.

*Případ 3:* Vrchol $x$ měl znaménko $\ominus$, tedy teď získá $\delta(v) = -2$. To definice AVL stromu nedovoluje, takže musíme strom vyvážit. Označme $y$ vrchol, z nějž přišla informace o prohloubení, čili levého syna vrcholu $x$. Rozebereme případy podle jeho znaménka.

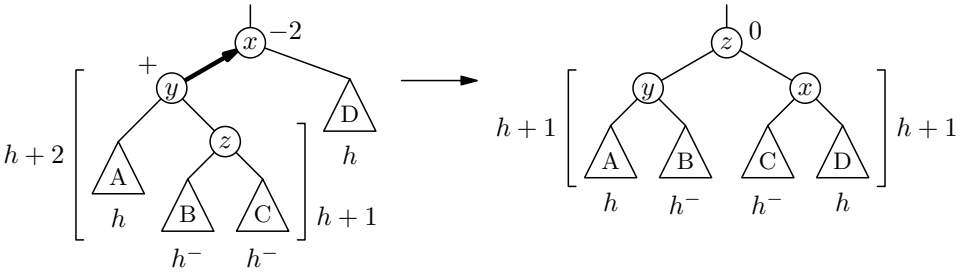
*Případ 3a:* Vrchol $y$ má znaménko $\ominus$. Situaci sledujme na obrázku.

- Označíme-li $h$ hloubku podstromu $C$, podstrom $T(y)$ má hloubku $h+2$, takže podstrom $A$ má hloubku $h+1$ a podstrom $B$ hloubku $h$.
- Provedeme rotaci hrany $xy$.
- Tím získá vrchol $x$ znaménko $\odot$, podstrom $T(x)$ hloubku $h+1$, vrchol $y$ znaménko $\odot$ a podstrom $T(y)$ hloubku $h+2$.
- Jelikož před započetím operace INSERT měl podstrom $T(x)$ hloubku $h+2$, z pohledu vyšších pater se nic nezměnilo. Propagování tedy opět zastavíme.




*Případ 3b:* Vrchol $y$ má znaménko $\oplus$. Sledujme opět obrázek.

- Označíme $z$ pravého syna vrcholu $y$ (uvědomte si, že musí existovat).
- Označíme jednotlivé podstromy tak jako na obrázku a spočítáme jejich hloubky. Referenční hloubku $h$ zvolíme podle podstromu $D$. Hloubky $h^-$ znamenají „buď $h$ nebo $h-1$“.
- Provedeme dvojitou rotaci, která celou konfiguraci překoření za vrchol $z$.
- Přepočítáme hloubky a znaménka. Vrchol $x$ bude mít znaménko buď $\ominus$ nebo $\odot$, vrchol $y$ buď $\odot$ nebo $\oplus$, každopádně oba podstromy $T(x)$ a $T(y)$ získají hloubku $h+1$. Proto vrchol $z$ získá znaménko $\odot$.
- Před započetním InsertTu činila hloubka celé konfigurace $h+2$, nyní je také $h+2$, takže propagování zastavíme.




*Případ 3c:* Vrchol $y$ má znaménko $\odot$.

Tento případ je ze všech nejjednodušší – nemůže totiž nikdy nastat. Z vrcholu se znaménkem $\odot$ se informace o prohloubení v žádném z předchozích případů nešíří.

**Mazání ze stromu**

Budeme postupovat obdobně jako u InsertTu: vrchol smažeme podle původního algoritmu BvsDelete a po cestě zpět do kořene propagujeme informaci o snížení hloubky podstromu. Připomeňme, že pokaždé mažeme list nebo vrchol s jediným synem, takže stačí propagovat od místa smazaného vrcholu nahoru.

Opět popíšeme jeden krok propagování. Nechť do vrcholu $x$ přišla informace o snížení hloubky podstromu, bez újmy na obecnosti z levého syna. Rozlišíme následující případy.

*Případ 1:* Vrchol $x$ má znaménko $\ominus$.

- Hloubka levého podstromu se právě vyrovnala s hloubkou pravého, znaménko $x$ se mění na $\odot$.
- Hloubka podstromu $T(x)$ se snížila, takže pokračujeme v propagování.

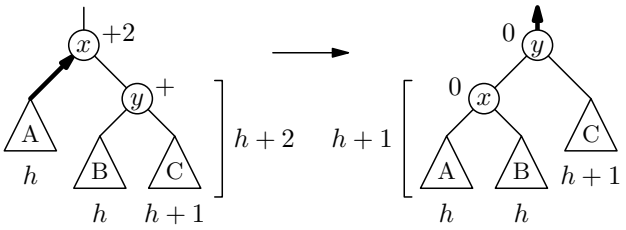
*Případ 2:* Vrchol $x$ má znaménko $\odot$.

- Znaménko $x$ se změní na $\oplus$.
- Hloubka podstromu $T(x)$ se nezměnila, takže propagování ukončíme.

*Případ 3:* Vrchol $x$ má znaménko $\oplus$. Tehdy se jeho znaménko změní na $+2$ a musíme vyvažovat. Rozebereme tři případy podle znaménka pravého syna $y$ vrcholu $x$. (Všimněte si, že na rozdíl od vyvažování po INSERTu to musí být opačný syn než ten, ze kterého přišla informace o změně hloubky.)

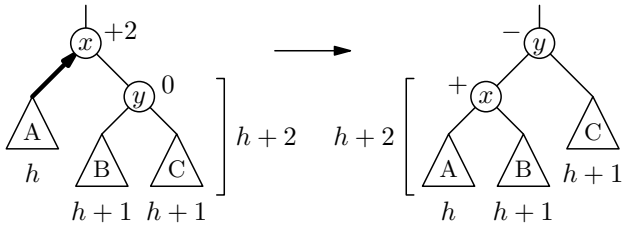
*Případ 3a:* Vrchol $y$ má také známénko $\oplus$.

- Označíme-li $h$ hloubku podstromu $A$, bude mít $T(y)$ hloubku $h+2$, takže $C$ hloubku $h+1$ a $B$ hloubku $h$.
- Provedeme rotaci hrany $xy$.
- Tím vrchol $x$ získá znaménko $\odot$, podstrom $T(x)$ hloubku $h+1$, takže vrchol $y$ dostane také znaménko $\odot$.
- Před započetím DELETE měl podstrom $T(x)$ hloubku $h+3$, nyní má $T(y)$ hloubku $h+2$, takže z pohledu vyšších hladin došlo ke snížení hloubky. Proto změnu propagujeme dál.



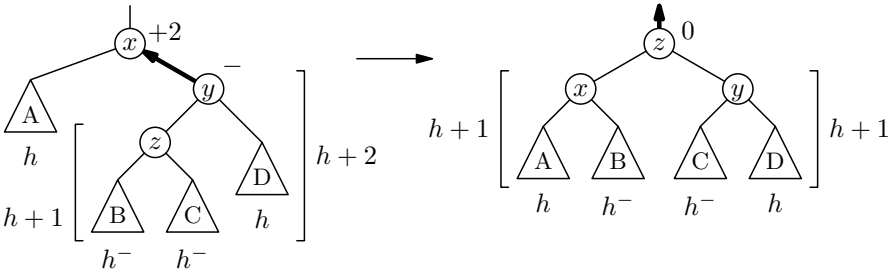

*Případ 3b:* Vrchol $y$ má znaménko $\odot$.

- Nechť $h$ je hloubka podstromu $A$. Pak $T(y)$ má hloubku $h+2$ a $B$ i $C$ hloubku $h+1$.
- Provedeme rotaci hrany $xy$.
- Vrchol $x$ získává znaménko $\oplus$, podstrom $T(x)$ hloubku $h+2$, takže vrchol $y$ obdrží znaménko $\ominus$.
- Hloubka podstromu $T(x)$ před začátkem DELETE činila $h+3$, nyní má podstrom $T(y)$ hloubku také $h+3$, pročež propagování ukončíme.




*Případ 3c:* Vrchol $y$ má znaménko $\ominus$.

- Označíme $z$ levého syna vrcholu $y$.
- Označíme podstromy podle obrázku a spočítáme jejich hloubky. Referenční hloubku $h$ zvolíme opět podle $A$. Hloubky $h^-$ znamenají „buď $h$ nebo $h-1$“.
- Provedeme dvojitou rotaci, která celou konfiguraci překoření za vrchol $z$.
- Přepočítáme hloubky a znaménka. Vrchol $y$ bude mít znaménko buď $\odot$ nebo $\ominus$, $x$ buď $\odot$ nebo $\oplus$. Podstromy $T(y)$ a $T(x)$ budou každopádně hluboké $h+1$. Proto vrchol $z$ obdrží znaménko $\odot$.
- Původní hloubka podstromu $T(x)$ před začátkem DELETE činila $h+3$, nyní hloubka $T(z)$ činí $h+2$, takže propagujeme dál.




**Složitost operací**

Dokázali jsme, že hloubka AVL stromu je vždy $\mathcal{O}(\log n)$. Původní implementace operací BvsFind, BvsInsert a BvsDelete tedy pracují v logaritmickém čase. Po BvsInsert a BvsDelete ještě musí následovat vyvážení, které se ovšem vždy vrací po cestě do kořene a v každém vrcholu provede $\mathcal{O}(1)$ operací, takže celkově také trvá $\mathcal{O}(\log n)$.

**Cvičení**

1. Dokažte, že pro minimální velikost $A_k$ AVL stromu hloubky $k$ platí vztah $A_k = F_{k+3} - 1$ (kde $F_n$ je $n$-té Fibonacciho číslo). Z toho odvoďte přesný vzorec pro minimální a maximální možnou hloubku AVL stromu na $n$ vrcholech.
2. Při vyvažování po INSERTu jsme se nemuseli zabývat případem 3c proto, že z $\odot$ se informace o prohloubení nikdy nešíří. Nemůžeme stejným způsobem dokázat, že případ 3b také nikdy nenastane? (Pozor, chyták!)

3. Upravte AVL stromy tak, aby dokázaly pro libovolné $k$ najít $k$-tý nejmenší prvek. Pokud doplníte nějaké další informace do vrcholů stromu, nezapomeňte, že je musíte udržovat i při vyvažování.

## 1.4. Více klíčů ve vrcholech: (a,b)-stromy

Nyní prozkoumáme obecnější variantu vyhledávacích stromů, která připouští proměnlivý počet klíčů ve vrcholech. Tím si sice trochu zkomplikujeme úvahy o struktuře stromů, ale za odměnu získáme přímočařejší vyvažovací algoritmy bez složitého rozboru případů.

**Definice:** *Obecný vyhledávací strom* je zakořeněný strom s určeným pořadím synů každého vrcholu. Vrcholy dělíme na vnitřní a vnější, přičemž platí:

*Vnitřní (interní) vrcholy* obsahují klíče, v každém obecně jiný počet. Pokud vrchol obsahuje klíče $x_1 < \ldots < x_k$, pak má $k+1$ synů, které označíme $s_0, \ldots, s_k$. Klíče slouží jako oddělovače hodnot v podstromech, čili platí:

$$T(s_0) < x_1 < T(s_1) < x_2 < \ldots < x_{k-1} < T(s_{k-1}) < x_k < T(s_k),$$

kde $T(s_i)$ značí množinu všech klíčů z daného podstromu. Často se hodí dodefinovat $x_0 = -\infty$ a $x_{k+1} = +\infty$, aby nerovnost $x_i < T(s_i) < x_{i+1}$ platila i pro krajní syny.

*Vnější (externí) vrcholy* neobsahují žádná data a nemají žádné potomky. Jsou to tedy listy stromu. Na obrázku je značíme jako malé čtverečky, v programu je můžeme reprezentovat nulovými ukazateli (`NULL` v jazyku C, `nil` v Pascalu).

Podobně jako BVS, i obecné vyhledávací stromy mohou degenerovat. Přidáme proto další podmínky pro zajištění vyváženosti.

**Definice:** *(a,b)-strom* pro parametry $a \geq 2$, $b \geq 2a-1$ je obecný vyhledávací strom, pro který navíc platí:

1. Kořen má 2 až $b$ synů, ostatní vnitřní vrcholy $a$ až $b$ synů.
2. Všechny vnější vrcholy jsou ve stejné hloubce.

Požadavky na $a$ a $b$ mohou vypadat tajemně, ale jsou snadno splnitelné a později vyplyne, proč jsme je potřebovali. Chcete-li konkrétní příklad, představujte si ten nejmenší možný: $(2,3)$-strom. Vše ovšem budeme odvozovat obecně. Přitom budeme předpokládat, že $a$ a $b$ jsou konstanty, které se mohou „schovat do $\mathcal{O}$“. Později prozkoumáme, jaký vliv má volba těchto parametrů na vlastnosti struktury. Nyní začneme odhadem hloubky.

**Lemma:** $(a,b)$-strom s $n$ klíči má hloubku $\Theta(\log n)$.

*Důkaz:* Počítejme minimální počet klíčů $m_h$ ve stromu hloubky $h \geq 1$. Vrcholy budou obsahovat minimální povolené počty klíčů a rozdělíme je podle hloubky do hladin: na 0-té hladině je kořen se dvěma klíči, na $h$-té leží listy bez klíčů, na mezilehlých hladinách ostatní vnitřní vrcholy s $a-1$ klíči. Na $i$-té hladině bude tedy ležet $2 \cdot a^{i-1}$

vrcholů. Sečtením přes hladiny získáme:

$$m_h = 1 + (a-1) \cdot \sum_{i=1}^{h-1} 2 \cdot a^{i-1} = 1 + 2 \cdot (a-1) \cdot \sum_{j=0}^{h-2} a^j.$$

Sečteme-li geometrickou řadu v poslední sumě, dostaneme:

$$m_h = 1 + 2 \cdot (a-1) \cdot \frac{a^{h-1} - 1}{a-1} = 1 + 2 \cdot (a^{h-1} - 1) = 2a^{h-1} - 1.$$

Víme tedy, že minimální počet klíčů roste s hloubkou exponenciálně. Proto maximální hloubka musí s počtem klíčů růst nejvýše logaritmicky. (Srovnejte s výpočtem maximální hloubky AVL stromů.)

Podobně spočítáme, že maximální počet klíčů $M_h$ roste také exponenciálně, takže minimální možná hloubka je také logaritmická. Tentokrát uvážíme strom, jehož všechny vnitřní vrcholy včetně kořene obsahují nejvyšší povolený počet $b-1$ klíčů:

$$M_h = (b-1) \cdot \sum_{i=0}^{h-1} b^i = (b-1) \cdot \frac{b^h - 1}{b-1} = b^h - 1.$$

□

**Hledání klíče**

Hledání klíče v $(a,b)$-stromu probíhá podobně jako v BVS: začneme v kořeni a v každém vnitřním vrcholu se porovnáváním s jeho klíči rozhodneme, do kterého podstromu se vydat. Přitom buď narazíme na hledaný klíč, nebo dojdeme až do listu a tam skončíme s nepořízenou.

**Vkládání do stromu**

Při vkládání nejprve zkusíme nový klíč vyhledat. Pokud ve stromu ještě není přítomen, skončíme v nějakém listu. Nabízí se změnit list na vnitřní vrchol přidáním jednoho klíče a dvou listů jako synů. Tím bychom ovšem porušili axiom o stejné hloubce listů.

Raději se proto zaměříme na otce nalezeného listu a vložíme klíč do něj. To nás donutí přidat jednoho syna, ale jelikož ostatní synové jsou listy, i tento může být list. Pokud jsme přidáním klíče vrchol nepřeplnili (má nadále nejvýš $b-1$ klíčů), jsme hotovi.

Pakliže jsme vrchol přeplnili, rozdělíme jeho klíče mezi dva nové vrcholy, přibližně napůl. K nadřazenému vrcholu ovšem musíme místo jednoho syna připojit dva nové, takže v nadřazeném vrcholu musí přibýt klíč. Proto přeplněný vrchol raději rozdělíme na tři části: prostřední klíč, který budeme vkládat o patro výš, a levou a pravou část, z nichž se stanou nové vrcholy. Obr

Tím jsme vložení klíče do aktuálního vrcholu převedli na tutéž operaci o patro výš. Tam může opět dojít k přeplnění a následnému štěpení vrcholu a tak dále, možná

až do kořene. Pokud rozštěpíme kořen, vytvoříme nový kořen s jediným klíčem a dvěma syny (zde se hodí, že jsme kořeni dovolili mít méně než $a$ synů) a celý strom se o hladinu prohloubí.

Naše ukázková implementace má podobu rekurzivní funkce ABInsert2$(v, x)$, která dostane za úkol vložit do podstromu s kořenem $v$ klíč $x$. Jako výsledek vrátí trojici $(p, x', q)$, pokud došlo k štěpení vrcholu $v$ na vrcholy $p$ a $q$ oddělené klíčem $x'$, anebo $\emptyset$, pokud $v$ zůstalo kořenem podstromu. Hlavní procedura ABInsert navíc ošetřuje případ štěpení kořene.

**Procedura** ABInsert

*Vstup:* Kořen stromu $r$, vkládaný klíč $x$

1. $t \leftarrow$ ABInsert2$(r, x)$
2. Pokud $t$ má tvar trojice $(p, x', q)$:
3. $\quad r \leftarrow$ nový kořen s klíčem $x'$ a syny $p$ a $q$

*Výstup:* Nový kořen $r$

**Procedura** ABInsert2$(v, x)$

*Vstup:* Kořen podstromu $v$, vkládaný klíč $x$

1. Pokud $v$ je list, skončíme a vrátíme trojici $(\ell_1, x, \ell_2)$, kde $\ell_1$ a $\ell_2$ jsou nově vytvořené listy.
2. Označíme $x_1, \ldots, x_k$ klíče ve vrcholu $v$ a $s_0, \ldots, s_k$ jeho syny.
3. Pokud $x = x_i$ pro nějaké $i$, skončíme a vrátíme $\emptyset$.
4. Najdeme $i$ tak, aby platilo $x_i < x < x_{i+1}$ ($x_0 = -\infty$, $x_{k+1} = +\infty$).
5. $t \leftarrow$ ABInsert2$(s_i, x)$
6. Pokud $t = \emptyset$, skončíme a také vrátíme $\emptyset$.
7. Označíme $(p, x', q)$ složky trojice $t$.
8. Mezi klíče $x_i$ a $x_{i+1}$ vložíme klíč $x'$.
9. Syna $s_i$ nahradíme dvojicí synů $p$ a $q$.
10. Pokud počet synů nepřekročil $b$, skončíme a vrátíme $\emptyset$.
11. $m \leftarrow \lfloor (b-1)/2 \rfloor$ *(Došlo k štěpení, volíme prostřední klíč.)*
12. Vytvoříme nový vrchol $v_1$ s klíči $x_1, \ldots, x_{m-1}$ a syny $s_0, \ldots, s_{m-1}$.
13. Vytvoříme nový vrchol $v_2$ s klíči $x_{m+1}, \ldots, x_b$ a syny $s_m, \ldots, s_{b+1}$.
14. Vrátíme trojici $(v_1, x_m, v_2)$.

Zbývá dokázat, že vrcholy vznikné štěpením mají dostatečný počet synů. Vrchol $v$ jsme rozštěpili v okamžiku, kdy dosáhl právě $b+1$ synů, a tedy obsahoval $b$ klíčů. Jeden klíč posíláme o patro výš, takže novým vrcholům $v_1$ a $v_2$ přidělíme po řadě $\lfloor (b-1)/2 \rfloor$ a $\lceil (b-1)/2 \rceil$ klíčů. Kdyby některý z nich byl „podměrečný“, muselo by platit $(b-1)/2 < a-1$, a tedy $b-1 < 2a-2$, čili $b < 2a-1$. Ejhle, podmínka na $b$ v definici $(a,b)$-stromu byla zvolena přesně tak, aby této situaci zabránila.

### Mazání ze stromu

Chceme-li ze stromu smazat nějaký klíč, nejprve ho vyhledáme. Pokud se nachází na předposlední hladině (té, pod níž jsou už pouze listy), můžeme ho smazat přímo, jen musíme ošetřit případné podtečení vrcholu.

Klíče ležící na vyšších hladinách nemůžeme mazat jen tak, neboť smazáním klíče přicházíme i o místo pro připojení podstromu. To je situace podobná mazání vrcholu se dvěma syny v binárním stromu a vyřešíme ji také podobně. Mazaný klíč nahradíme jeho následníkem. To je nejlevější vrchol v pravém podstromu, který tudíž leží na předposlední hladině a může být smazán přímo.

Zbývá tedy vyřešit, co se má stát v případě, že vrchol $v$ s $a$ syny přijde o klíč, takže už je „pod míru“. Tehdy budeme postupovat opačně než při vkládání – pokusíme se vrchol sloučit s některým z jeho bratrů. To je ovšem možné provést pouze tehdy, když bratr také obsahuje málo klíčů; pokud jich naopak obsahuje hodně, nějaký klíč si od něj můžeme půjčit.

Nyní popíšeme, jak to přesně provést. Bez újmy na obecnosti předpokládejme, že vrchol $v$ má pravého bratra $b$ odděleného nějakým klíčem $o$ v otci. Pokud by existoval pouze levý bratr, vybereme toho a následující postup provedeme zrcadlově převrácený.

Pokud má bratr pouze $a$ synů, sloučíme vrcholy $v$ a $b$ do jediného vrcholu a přidáme do něj ještě klíč $o$ z otce. Tím vznikne vrchol s $(a-2)+(a-1)+1 = 2a-2$ klíči, což není větší než $b-1$. Problém jsme tedy převedli na mazání klíče z otce, což je tentýž problém o hladinu výš. Obr

Má-li naopak bratr více jak $a$ synů, odpojíme od něj jeho nejlevějšího syna $\ell$ a nejmenší klíč $m$. Poté klíč $m$ přesuneme do otce a klíč $o$ odtamtud přesuneme do $v$, kde se stane nejpravějším klíčem, za který přepojíme syna $\ell$. Poté mají $v$ i $b$ povolené počty synů a můžeme skončit. (Všimněte si, že tato operace je podobná rotaci hrany v binárním stromu.)

Nyní tento postup zapíšeme jako rekurzivní proceduru ABDelete2. Ta dostane kořen podstromu a klíč, který má smazat. Jako výsledek vrátí vrátí podstrom s tímtéž kořenem, ovšem možná podměrečným. Hlavní procedura ABDelete navíc ošetřuje případ, kdy z kořene zmizí všechny klíče, takže je potřeba kořen smazat a tím snížit celý strom o hladinu.

**Procedura** ABDelete
*Vstup:* Kořen stromu $r$ a mazaný klíč $x$

1. Zavoláme ABDelete2$(r, x)$.
2. Pokud $r$ má jediného syna $s$:
3. Zrušíme vrchol $r$.
4. $r \leftarrow s$

*Výstup:* Nový kořen $r$

**Procedura** ABDelete2
*Vstup:* Kořen podstromu $v$ a mazaný klíč $x$

1. Označíme $x_1, \ldots, x_k$ klíče ve vrcholu $v$ a $s_0, \ldots, s_k$ jeho syny.
2. Pokud $x = x_i$ pro nějaké $i$: *(Našli jsme)*
3. Pokud $s_i$ je list: *(Jsme na předposlední hladině.)*
4. Odstraníme z $v$ klíč $x_i$ a list $s_i$.

5. Skončíme.
6. Jinak: *(Jsme výš, musíme nahrazovat.)*
7. $m \leftarrow$ minimum podstromu s kořenem $s_i$
8. $x_i \leftarrow m$
9. Zavoláme ABDELETE2$(s_i, m)$.
10. Jinak: *(Mažeme z podstromu.)*
11. Najdeme $i$ takové, aby $x_i < x < x_{i+1}$ ($x_0 = -\infty$, $x_{k+1} = +\infty$).
12. Pokud $s_i$ je list, skončíme. *(Klíč ve stromu není.)*
13. Zavoláme ABDELETE2$(s_i, x)$.
14. *(Vrátili jsme se z $s_i$ a kontrolujeme, zda tento syn není pod míru.)*
15. Pokud $s_i$ má alespoň $a$ synů, skončíme.
16. Je-li $i < k$: *(Existuje pravý bratr $s_{i+1}$.)*
17. Pokud má $s_{i+1}$ alespoň $a + 1$ synů: *(Půjčíme si klíč.)*
18. Odpojíme z $s_{i+1}$ nejmenší klíč $m$ a nejlevějšího syna $\ell$.
19. K vrcholu $s_i$ připojíme jako poslední klíč $x_i$ a jako nejpravějšího syna $\ell$.
20. $x_i \leftarrow m$
21. Jinak: *(Slučujeme syny.)*
22. Vytvoříme nový vrchol $s$, který bude obsahovat všechny klíče a syny z vrcholů $s_i$ a $s_{i+1}$ a mezi nimi klíč $x_i$.
23. Z vrcholu $v$ odstraníme klíč $x_i$ a syny $s_i$ a $s_{i+1}$. Tyto syny zrušíme a na jejich místo připojíme syna $s$.
24. Jinak provedeme kroky 17 až 23 zrcadlově pro levého bratra $s_{i-1}$ místo $s_{i+1}$.

**Časová složitost**

Pro rozbor časové složitosti předpokládáme, že parametry $a$ a $b$ jsou konstanty. Hledání, vkládání i mazání proto tráví na každé hladině stromu čas $\mathcal{O}(1)$ a jelikož můžeme počet hladin odhadnout jako $\Theta(\log n)$, celková časová složitost všech tří základních operací činí $\Theta(\log n)$.

Vraťme se nyní k volbě parametrů $a$, $b$. Především je známo, že se nevyplácí volit $b$ výrazně větší než je dolní mez $2a - 1$ (detaily viz cvičení 1.4.1). Proto se obvykle používají $(a, 2a-1)$-stromy, případně $(a, 2a)$-stromy. Rozdíl mezi $b = 2a-1$ a $b = 2a$ se zdá být zcela nepodstatný, ale jak je vidět v cvičeních 1.4.5 a 1.4.6, o jedničku větší manévrovací prostor má zásadní vliv na amortizovanou složitost.

Pokud chceme datovou strukturu udržovat v klasické paměti, vyplácí se volit $a$ co nejnižší. Vhodné parametry jsou například $(2, 3)$ nebo $(2, 4)$.

Ukládáme-li data na disk, nabízí se využít toho, že je rozdělen na bloky. Přečíst celý blok je přitom zhruba stejně rychlé jako přečíst jediný byte, zatímco skok na jiný blok trvá dlouho. Proto nastavíme $a$ tak, aby jeden vrchol stromu zabíral celý blok. Například pro disk s 4 KB bloky, 32-bitové klíče a 32-bitové ukazatele zvolíme $(256, 511)$-strom. Strom pak bude opravdu mělký: čtyři hladiny postačí pro uložení

více než 33 milionů klíčů. Navíc na poslední hladině jsou pouze listy, takže při každém hledání přečteme pouhé tři bloky.

V dnešních počitačích často mezi procesorem a hlavní pamětí leží *cache* (rychlá vyrovnávací paměť), která má také blokovou strukturu s typickou velikostí bloku 64 B. Často se proto i u stromů v hlavní paměti volit trochu větší vrcholy, aby odpovídaly blokům cache. Pro 32-bitové klíče a 32-bitové ukazatele tedy použijeme $(4, 7)$-strom. Jen si musíme dávat pozor na správné zarovnání adres vrcholů na násobky 64 B.

### Další varianty

Ve světě se lze setkat i s jinými definicemi $(a, b)$-stromů, než je ta naše. Často se například dělá to, že data jsou uložena pouze ve vrcholech na druhé nejnižší hladině, zatímco ostatní hladiny obsahují pouze pomocné klíče, typicky minima z podstromů. Tím si trochu zjednodušíme operace (viz cvičení 1.4.3), ale zaplatíme za to vyšší redundancí dat. Může to nicméně být šikovné, pokud potřebujeme implementovat slovník, který klíčům přiřazuje rozměrná data.

V teorii databází a souborových systémů se často hovoří o *B-stromech.* Pod tímto názvem se skrývají různé datové struktury, většinou $(a, 2a - 1)$-stromy nebo $(a, 2a)$-stromy, nezřídka v úpravě dle předchozího odstavce.

### Cvičení

1. Odhalte, jak závisí složitost operací s $(a, b)$-stromy na parametrech $a$ a $b$. Z toho odvoďte, že se nikdy nevyplatí volit $b$ výrazně větší než $2a$.

2.* Naprogramujte $(a, b)$-stromy a změřte, jak jsou na vašem počítači rychlé pro různé volby $a$ a $b$. Projevuje se vliv cache tak, jak jsme naznačili?

3. Rozmyslete, jak provádět operace INSERT a DELETE na variantě $(a, b)$-stromů, která ukládá užitečná data jen do nejnižších vnitřních vrcholů. Analyzujte časovou složitost a srovnejte s naší verzí struktury.

4. Ukažte, že pokud budeme do prázdného stromu postupně vkládat klíče $1, \ldots, n$, provedeme celkem $\Theta(n)$ operací. K tomu potřebujeme pamatovat si, ve kterém vrcholu skončil předchozí vložený klíč, abychom nemuseli pokaždé hledat znovu od kořene.

5. Někdy se hodí minimalizovat vedle časové složitosti také počet *strukturálních změn* stromu během operace. Tak se říká změnám klíčů a ukazatelů uložených ve vrcholech. Ukažte, že pokud v původně prázdném $(2, 3)$-stromu provedeme $n$ operací INSERT, každá z nich provede amortizovaně konstantní počet strukturálních změn. Zobecněte pro libovolné $(a, b)$-stromy.

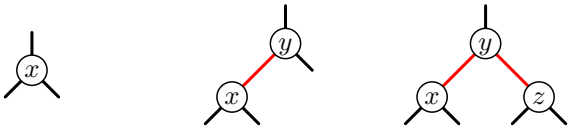
6.* Podobně jako v předchozím cvičení budeme počítat strukturální změny, tentokrát pro $(2, 4)$-strom a libovolnou kombinaci operací INSERT a DELETE. Ukažte, že nadále jedna operace provede amortizovaně $\mathcal{O}(1)$ změn. Zobecněte na $(a, 2a)$-stromy a ukažte, že v $(a, 2a - 1)$-stromech nic takového neplatí.

## 1.5. Červeno-černé stromy

Nyní se od obecných $(a, b)$-stromů vrátíme zpět ke stromům binárním. Ukážeme, jak překládat $(2, 4)$-stromy na binární stromy, čímž získáme další variantu BVS s logaritmickou hloubkou a jednoduchým vyvažováním. Říká se jí *červeno-černé stromy* (red-black trees, RB stromy). My si je předvedeme v trochu neobvyklé, ale příjemnější variantě navržené v roce 2008 R. Sedgewickem pod názvem left-leaning red-black trees (LLRB stromy).

Překlad bude fungovat tak, že každý vrchol $(2, 4)$-stromu nahradíme konfigurací jednoho nebo více binárních vrcholů. Aby bylo možné rekonstruovat původní $(2, 4)$-strom, zavedeme barvy hran: *červené hrany* budou spojovat vrcholy tvořící jednu konfiguraci, *černé hrany* povedou mezi konfiguracemi, čili to budou hrany původního $(2, 4)$-stromu. Barvu hrany si můžeme budeme pamatovat například v jejím spodním vrcholu.

Strom přeložíme podle následujícího obrázku. Vrcholům $(2, 4)$-stromu budeme podle počtu synů říkat 2-vrcholy, 3-vrcholy a 4-vrcholy. 2-vrchol zůstane sám sebou. 3-vrchol nahradíme dvěma binárními vrcholy, přičemž červená hrana musí vždy vést doleva (to je ono LL v názvu LLRB stromů, obecné RB stromy nic takového nepožadují, což situaci později dost zkomplikuje). 4-vrchol nahradíme „třešničkou“ ze tří binárních vrcholů.




Pokud podle těchto pravidel transformujeme i definici $(2, 4)$-stromu, vznikne následující definice LLRB stromu.

**Definice:** *LLRB strom* je binární vyhledávací strom s vnějšími vrcholy, jehož hrany jsou obarveny červeně a černě. Přitom platí následující axiomy:

1. Neexistují dvě červené hrany bezprostředně nad sebou.
2. Jestliže z vrcholu vede dolů jediná červená hrana, pak vede doleva.
3. Listy jsou vždy obarveny černě. (To se hodí, jelikož listy jsou pouze virtuálni, takže do nich neumíme barvu hrany uložit.)
4. Na každé cestě z kořene do listu je stejný počet černých hran.

Prvním dvěma axiomům budeme říkat červené, zbylým dvěma černé. **Fix!**

**Pozorování:** Z axiomů plyne, že každá konfigurace pospojovaná červenými hranami vypadá jedním z uvedených způsobů. Proto je každý LLRB strom překladem nějakého $(2, 4)$-stromu podle našich pravidel.

**Důsledek:** Hloubka LLRB stromu s $n$ klíči je $\Theta(\log n)$.

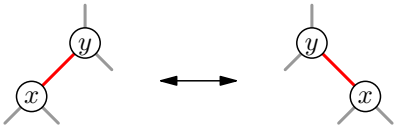
*Důkaz:* Hloubka $(2, 4)$-stromu s $n$ klíči činí $\Theta(\log n)$, překlad na LLRB strom počet hladin nesníží a nejvýše zdvojnásobí. □

---

**Fix:** Obrázek s příkladem překladu.

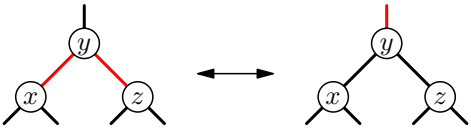
### Vyvažovací operace

Operace s LLRB stromy se skládají ze dvou základních úprav. Tou první je opět rotace, ale používáme ji pouze pro červené hrany:




*Rotace červené hrany* zachovává nejen správné uspořádání klíčů ve vrcholech, ale i černé axiomy. Platnost červených axiomů záleží na barvách okolních hran, takže rotaci budeme muset používat opatrně. (Rotování černých hran se vyhýbáme, protože by navíc porušovalo i axiom 4.)

Dále budeme používat ještě *přebarvení 4-vrcholu.* Dvojici červených hran tvořících 4-vrchol přebarvíme na černou, a naopak černou hranu vedoucí z 4-vrcholu nahoru přebarvíme na červenou:

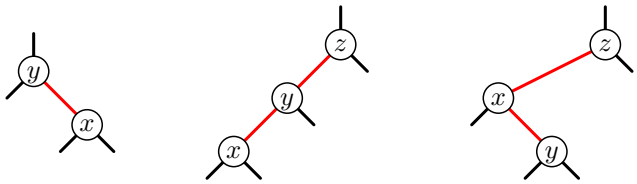



Tato úprava odpovídá rozštěpení 4-vrcholu na dva 2-vrcholy, přičemž prostřední klíč $y$ do nadřazeného $k$-vrcholu. Černé axiomy zůstanou zachovány, ale může dojít k porušení červených axiomů o patro výše.

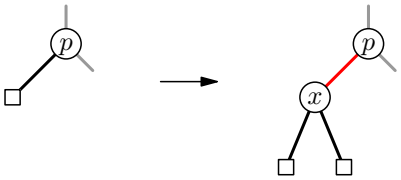
### Vkládání štěpením shora dolů

Nyní popíšeme, jak se do LLRB stromu vkládá. Půjdeme na to asi takto: místo pro nový vrchol budeme hledat obvyklým zpusobem, ale kdykoliv cestou potkáme 4-vrchol, rovnou ho rozštěpíme přebarvením. Až dorazíme do listu, připojíme místo něj nový vnitřní vrchol a hranu, po které jsme přišli, obarvíme červeně. Tím se nový klíč připojí k nadřazenému 2-vrcholu nebo 3-vrcholu. To zachovává černé axiomy, ale průběžně jsme porušovali ty červené, takže se budeme vracet zpět do kořene a rotacemi je opravovat.

Nyní podrobněji. Během hledání sestupujeme z kořene dolů a udržujeme invariant, že aktuální vrchol není 4-vrchol. Jakmile na nějaký 4-vrchol narazíme, přebarvíme ho. Tím se rozštěpí na dva 2-vrcholy a prostřední klíč se stane součástí nadřazeného $k$-vrcholu. Víme ovšem, že to nebyl 4-vrchol, takže se z něj nyní stane 3-vrchol nebo 4-vrchol. Jen možná bude nekorektně zakódovaný: 3-vrchol ve tvaru pravé odbočky nebo 4-vrchol se dvěma červenými hranami nad sebou:



Nakonec nás hledání nového klíče dovede do listu, což je místo, kam bychom klíč chtěli vložit. Nad námi leží 2-vrchol nebo 3-vrchol. List změníme na vnitřní vrchol s novým klíčem, pod něj pověsíme dva nové listy připojené černými hranami, hranu do otce přebarvíme na červenou:




Co se stane? Nový klíč leží na jediném místě, kde ležet může. Černé axiomy jsme neporušili, červené jsme opět mohli porušit vytvořením nekorektního 3-vrcholu nebo 4-vrcholu o patro výše.

Nyní se začneme vracet zpět do kořene a přitom opravovat všechna porušení červených axiomů tak, aby černé axiomy zůstaly zachovány.

Kdykoliv pod aktuálním vrcholem leží levá černá hrana a pravá červená, tak tuto hranu zrotujeme. Tím z nekorektního 3-vrcholu uděláme korektní a nekorektního 4-vrcholu uděláme takový nekorektní, jehož obě hrany jsou levé.

Poté otestujeme, zda pod aktuálním vrcholem leží levá červená hrana do syna, který má také levou červenou hranu. Pokud ano, objevili jsme zbývající případ nekorektního 4-vrcholu, který rotací jeho horní červené hrany převedeme na korektní.

Až dojdeme do kořene, struktura opět splňuje všechny axiomy LLRB stromů.

Následuje implementace v pseudokódu. Externí vrcholy ukládáme jako konstantu $\emptyset$, barvu hran si pamatujeme v jejich spodním vrcholu.

**Procedura** LlrbInsert
*Vstup:* Kořen stromu $v$, vkládaný klíč $x$

1. Pokud $v = \emptyset$, skončíme a vrátíme nově vytvořený červený vrchol $v$ s klíčem $x$.
2. Pokud $x = k(v)$, skončíme (klíč $x$ se ve stromu již nachází).
3. Jsou-li $\ell(v)$ i $r(v)$ červené, přebarvíme $\ell(v)$, $r(v)$ i $v$.
4. Pokud $x < k(v)$, položíme $\ell(v) \leftarrow$ LlrbInsert$(\ell(v), x)$.
5. Pokud $x > k(v)$, položíme $r(v) \leftarrow$ LlrbInsert$(r(v), x)$.
6. Je-li $\ell(v)$ černý a $r(v)$ červený, rotujeme hranu $(v, r(v))$.
7. Je-li $\ell(v)$ červený a $\ell(\ell(v))$ také červený, rotujeme hranu $(v, \ell(v))$.

*Výstup:* Nový kořen $v$

### Vkládání štěpením zdola nahoru

FIXME

### Mazání

FIXME

### Časová složitost

FIXME

### Cvičení

1. Spočítejte přesně, jaká může být minimální a maximální hloubka LLRB stromu s $n$ klíči.