1. Vyhledávací stromy

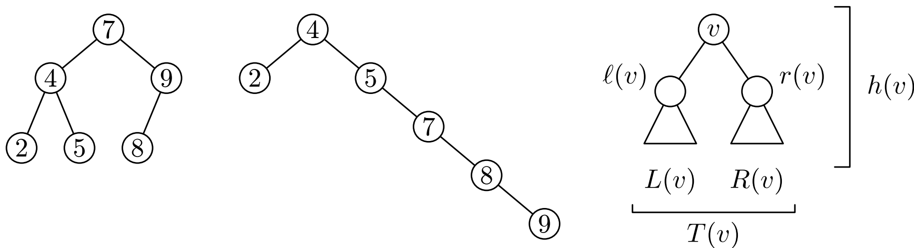
V kapitole **??** jsme se vydali po stopě datových struktur pro efektivní reprezentaci *množin* a *slovníků*. To nás nyní dovede k různým variantám vyhledávacích stromů. Začneme těmi binárními, ale později zjistíme, že se může hodit uvažovat o stromech obecněji.

1.1. Binární vyhledávací stromy

Množinu můžeme uložit do uspořádaného pole. V něm už umíme v logaritmickém čase vyhledávat, ovšem jakákoliv změna je pomalá. Pokusíme se proto od pole přejít k obecnější struktuře, která bude „pružnější“. Inspirujeme se popisem binárního vyhledávání pomocí rozhodovacích stromů. Ty už se nám hodily v oddílu **??**: dokázali jsme pomocí nich, že binární vyhledávání je optimální.

Připomeňme si, jak takový rozhodovací strom sestrojit. Kořen popisuje první porovnání: leží v něm 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).

Pro úspěšné vyhledávání přitom nepotřebujeme, abychom při konstrukci stromu pokaždé vybírali prostřední prvek intervalu. Pokud bychom volili jinak, dostaneme odlišný strom. Pomocí něj také půjde hledat, jen možná pomaleji – to je vidět na následujícím obrázku.



Obr. 1.1: Dva binární vyhledávací stromy a jejich značení

Co všechno musí strom splňovat, aby se podle něj dalo hledat, přetavíme do následujících definic. Nejprve připomeneme definici binárního stromu z oddílu **??**:

Definice: Strom nazveme *binární*, pokud je zakořeněný a každý vrchol má nejvýše dva syny, u nichž rozlišujeme, který je levý a který pravý.

Značení: Pro vrchol v binárního stromu T značíme:

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

Pokud vrchol nemá levého syna, položíme $\ell(v) = r(v) = \emptyset$. 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 univerza. Přitom musí pro každý vrchol v platit:

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

Jinak řečeno, vrchol v odděluje klíče v levém a pravém podstromu.

Základní operace

Pomocí vyhledávacích 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.

Procedura BvsShow (uspořádaný výpis BVS)

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($r(v)$).

Pokaždé tedy projdeme levý podstrom, pak kořen, a nakonec pravý podstrom. To nám dává tak zvané *symetrické pořadí* vrcholů (někdy také *in-order*). Jejich klíče přitom vypisujeme od nejmenšího k největšímu.

Hledání 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 zamíříme doleva. Je-li naopak $x > k(v)$, nic nepokazíme krokem doprava. Časem tedy x najdeme, anebo vyloučíme všechny možnosti, kde by se mohlo nacházet.

Algoritmus formulujeme jako rekurzivní funkci, kterou vždy voláme na kořen nějakého podstromu a vrátí nám nalezený vrchol.

Procedura BvsFind (hledání v BVS)

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 je kam. Klíče menší než ten aktuální se totiž mohou nacházet pouze v levém podstromu.

Procedura BvsMin (minimum v BVS)

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, pouze v okamžiku, kdy by vyhledávací algoritmus měl přejít do neexistujícího vrcholu, připojíme tam nový list. Rozmyslíme si, že to je jediné místo, kde podle definice nový prvek smí ležet. Operaci opět popíšeme rekurzivně. Funkce dostane na vstupu kořen (pod)stromu a vrátí nový kořen.

Procedura BvsInsert (vkládání do BVS)

Vstup: Kořen BVS v, vkládaný klíč x

1. Pokud $v = \emptyset$, vytvoříme nový vrchol v s klíčem x a skončíme.
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 a není třeba nic měnit.

Výstup: Nový kořen v

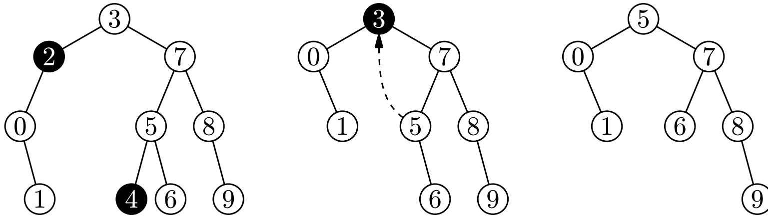
Při *mazání* může nastat několik různých případů (viz obrázek 1.2). Nechť v je vrchol, který chceme smazat. Je-li v list, můžeme tento list prostě odstranit, čímž provedeme operaci opačnou k BvsInsertu. Má-li v právě jednoho syna, postačí v „vystřihnout“, tedy nahradit synem.

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

Procedura BvsDelete (mazání z BVS)

Vstup: Kořen BVS v, mazaný klíč x

1. Pokud $v = \emptyset$, vrátíme $\emptyset$. $\triangleleft$ *klíč x ve stromu nebyl*
2. Pokud $x < k(v)$, položíme $\ell(v) \leftarrow$ BvsDelete($\ell(v), x$).
3. Pokud $x > k(v)$, položíme $r(v) \leftarrow$ BvsDelete($r(v), x$).
4. Pokud $x = k(v)$: $\triangleleft$ *chystáme se smazat vrchol v*
5. Pokud $\ell(v) = r(v) = \emptyset$, vrátíme $\emptyset$. $\triangleleft$ *byl to list*
6. Pokud $\ell(v) = \emptyset$, vrátíme $r(v)$. $\triangleleft$ *existoval jen pravý syn*



Obr. 1.2: Různé situace při mazání.
Nejprve mažeme vrcholy 2 a 4, poté 3.

7. Pokud $r(v) = \emptyset$, vrátíme $\ell(v)$. ◁ *existoval jen levý syn*
8. $s \leftarrow$ BvsMin$(r(v))$ ◁ *máme oba syny: nahradíme následníka s*
9. $k(v) \leftarrow k(s)$
10. $r(v) \leftarrow$ 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 tam a jednou zpět, nebo (v nejsložitějším případě BvsDelete) oběma směry dvakrát. Jejich časová složitost v nejhorším případě proto bude $\Theta(\textit{hloubka stromu})$.

Hloubka ovšem závisí na tom, jak moc je strom „košatý“. V příznivém případě vyjde sympatických $\mathcal{O}(\log n)$, jenže dalšími operacemi 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 počet vrcholů levého a pravého podstromu se smí lišit nejvýše o 1.

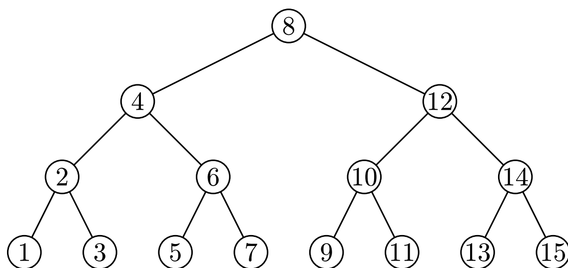
Pozorování: Dokonale vyvážený strom má hloubku $\lfloor \log_2 n \rfloor$, jelikož na kterékoliv cestě z kořene do listu klesá velikost podstromů s každým krokem alespoň dvakrát.

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 (cvičení 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 udržujících strom dokonale vyvážený platí, že 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 1.3. 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 $(n-1)/2 = 2^{k-1} - 1$ vrcholů, takže jejich kořeny jsou opět určeny jednoznačně a tak dále. Navíc si všimneme, že všechna lichá čísla jsou umístěna v listech stromu.



Obr. 1.3: Dokonale vyvážený BVS

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.* Ukázali jsme, že dokonale vyvážený strom o $2^k - 1$ vrcholech je *úplný* – všech k hladin obsahuje nejvyšší možný počet vrcholů. Dokažte, že ostatní dokonale vyvážené stromy mají podobnou strukturu, jen na poslední hladině mohou některé vrcholy chybět.

7. Zatím jsme předpokládali, že klíče je možné porovnávat v konstantním čase. Jak to dopadne, pokud klíče budou třeba řetězce o ℓ znacích? Stanovte složitost vyhledávání a srovnejte ji s hledáním v písmenkovém stromu z oddílu **??**.

8. Navrhněte algoritmus, který dostane dva BVS T_1, T_2 a sloučí jejich obsah do jediného BVS. Algoritmus by měl pracovat v čase $\mathcal{O}(|T_1| + |T_2|)$.

9. 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.

10. Naše tvrzení o náročnosti operací Insert a Delete v dokonale vyváženém stromu lze ještě zesílit. Dokažte, že lineární musí být složitost *obou* operací.

11. Ukažte, jak zjistit *následníka* daného vrcholu, tedy vrchol s nejbližší větší hodnotou.

12. Dokažte, že projdeme-li celý strom opakovaným hledáním následníka, strávíme tím čas $\Theta(n)$.

13.* *Úsporné stromy:* Obvyklá reprezentace BVS v paměti potřebuje v každém vrcholu 3 ukazatele: na levého syna, na pravého syna a na otce. Ukažte, jak si vystačit se dvěma ukazateli. Původní 3 ukazatele by z těch vašich mělo jít spočítat v konstantním čase.

1.2. 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

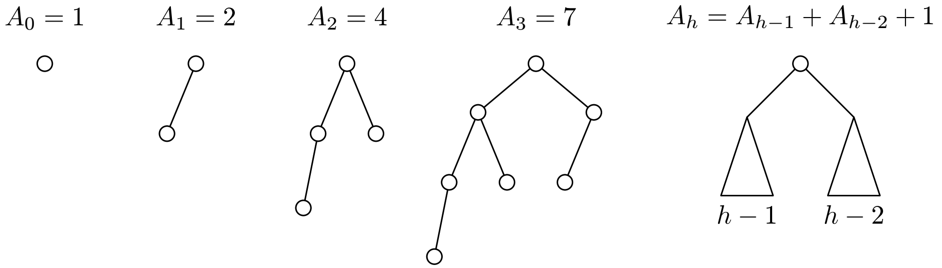
$$\Big|h(\ell(v)) - h(r(v))\Big| \leq 1.$$

Jinými slovy, hloubka levého a pravého podstromu se vždy liší nejvýše o jedna.

Stromům s hloubkovým vyvážením se říká *AVL stromy,* neboť je vymysleli v roce 1962 ruští matematici Georgij Maximovič Aděľson-Veľskij a Jevgenij Michailovič 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.



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

Pro malá h stačí rozebrat možné případy podle obrázku 1.4.

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 $A_h = A_{h-1} + A_{h-2} + 1$.

Tato rekurence připomíná Fibonacciho posloupnost z oddílu **??**. 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 důkaz našeho tvrzení postačí jednodušší asymptotický odhad.

Dokážeme indukcí, že $A_h \geq 2^{h/2}$. Jistě je $A_0 = 1 \geq 2^{0/2} = 1$ 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 vyvážený, 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:

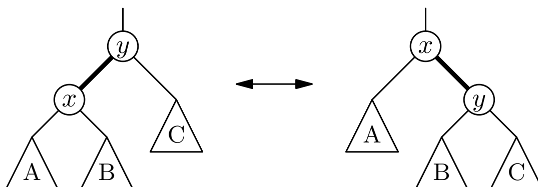
- $\delta(v) = 1$ (pravý podstrom je hlubší) – takový vrchol značíme +,

- $\delta(v) = -1$ (levý podstrom hlubší) – značíme $-$,
- $\delta(v) = 0$ (oba podstromy stejně hluboké) – značíme 0.

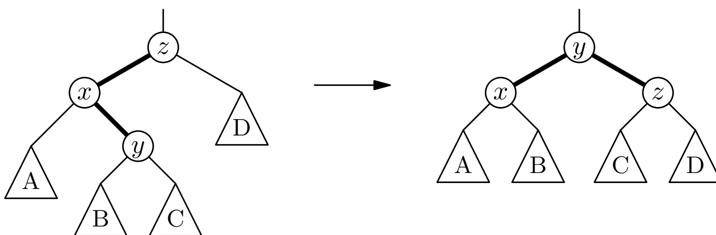
Jakmile narazíme na jiné $\delta(v)$, strom opravíme provedením jedné nebo více rotací.

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



Obr. 1.6: Dvojitá rotace

Vkládání do stromu

Nový prvek vložíme jako list se znaménkem 0. 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.)

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 $+$ a $-$. Rozlišíme několik případů.

Případ 1: Vrchol x měl znaménko $+$.

- Hloubka levého podstromu se právě vyrovnala s hloubkou pravého, čili znaménko x se změní na 0.
- Hloubka podstromu $T(x)$ se nezměnila, takže propagování informace ukončíme.

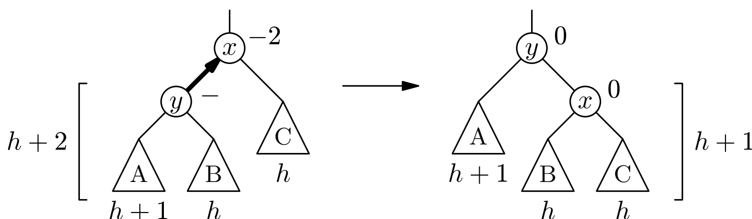
Případ 2: Vrchol x měl znaménko 0.

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

Případ 3: Vrchol x měl znaménko $-$, 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 $-$. Situaci sledujme na obrázku.

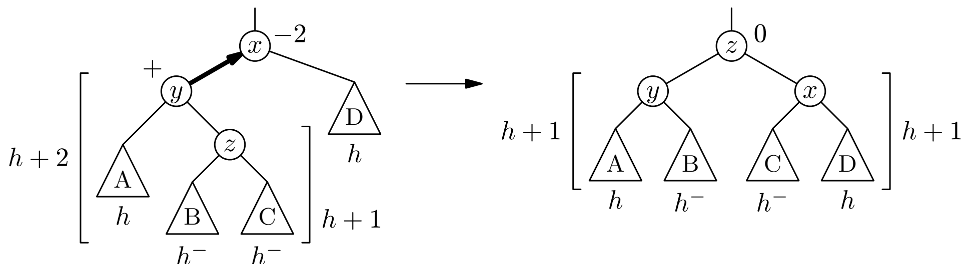
- Označíme-li h hloubku podstromu C, podstrom $T(y)$ má hloubku $h+2$, takže podstrom A má hloubku $h+1$ a podstrom B hloubku h.
- Provedeme rotaci hrany xy.
- Tím získá vrchol x znaménko 0, podstrom $T(x)$ hloubku $h + 1$, vrchol y znaménko 0 a podstrom $T(y)$ hloubku $h + 2$.
- Jelikož před započetím operace Insert měl podstrom $T(x)$ hloubku $h + 2$, z pohledu vyšších pater se nic nezměnilo. Propagování tedy zastavíme.



Případ 3b: Vrchol y má znaménko $+$. 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 $-$ nebo 0, vrchol y buď 0 nebo $+$, každopádně oba podstromy $T(x)$ a $T(y)$ získají hloubku $h+1$. Proto vrchol z získá znaménko 0.

- Před započtením INSERTu činila hloubka celé konfigurace $h+2$, nyní je také $h+2$, takže propagování zastavíme.



Případ 3c: Vrchol y má znaménko 0.

Tento případ je ze všech nejjednodušší – nemůže totiž nikdy nastat. Z vrcholu se znaménkem 0 se informace o prohloubení v žádném z předchozích případů nešíří.

Mazání ze stromu

Budeme postupovat obdobně jako u INSERTu: 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 $-$.

- Hloubka levého podstromu se právě vyrovnala s hloubkou pravého, znaménko x se mění na 0.
- Hloubka podstromu $T(x)$ se snížila, takže pokračujeme v propagování.

Případ 2: Vrchol x má znaménko 0.

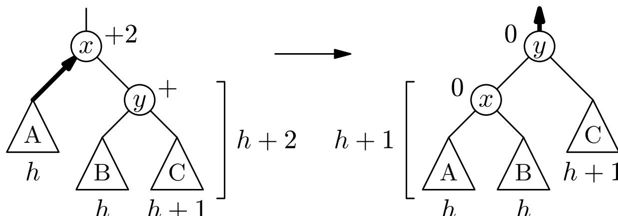
- Znaménko x se změní na $+$.
- Hloubka podstromu $T(x)$ se nezměnila, takže propagování ukončíme.

Případ 3: Vrchol x má znaménko $+$. 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 znaménko $+$.

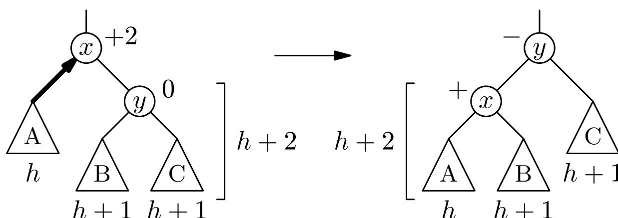
- Označíme-li h hloubku podstromu A, bude mít $T(y)$ hloubku $h+2$, takže C hloubku $h+1$ a B hloubku h.
- Provedeme rotaci hrany xy.

- Tím vrchol x získá znaménko 0, podstrom $T(x)$ hloubku $h+1$, takže vrchol y dostane také znaménko 0.
- 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.



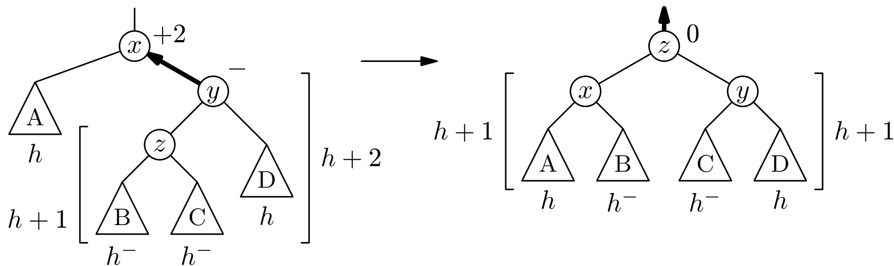
Případ 3b: Vrchol y má znaménko 0.

- 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 $+$, podstrom $T(x)$ hloubku $h+2$, takže vrchol y obdrží znaménko $-$.
- 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 $-$.

- 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 x bude mít znaménko buď 0 nebo $-$, y buď 0 nebo $+$. Podstromy $T(y)$ a $T(x)$ budou každopádně hluboké $h+1$. Proto vrchol z obdrží znaménko 0.
- 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 $\Theta(\log n)$. 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 $\Theta(1)$ operací, takže celkově také trvá $\Theta(\log n)$.

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. Může se hodit vztah z cvičení **??**.
2. Při vyvažování po Insertu jsme se nemuseli zabývat případem 3c proto, že z 0 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.
4. Mějme AVL strom použitý jako slovník: v každém vrcholu sídlí klíč a nějaká celočíselná hodnota. Upravte strom, aby uměl rychle zjistit největší hodnotu přiřazenou nějakému klíči z intervalu $[a, b]$.

5.* Pokračujme v předchozím cvičení: Také chceme, aby strom uměl ve všech vrcholech s klíči v zadaném intervalu $[a, b]$ rychle zvýšit hodnoty o δ. Může se hodit princip líného vyhodnocování z oddílu **??**.

6. AVL stromy potřebují pamatovat si v každém vrcholu znaménko. To může nabývat třech možných hodnot, takže na jeho uložení jsou potřeba dva bity. Ukažte, jak si vystačit s jediným bitem na vrchol.

1.3. 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 libovolný nenulový počet klíčů. Pokud ve vrcholu leží 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.

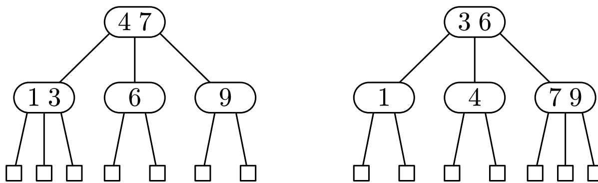
Vnější (externí) vrcholy neobsahují žádná data a nemají žádné potomky. Jsou to tedy listy stromu. Na obrázku je značíme jako malé čtverečky, v programu je můžeme reprezentovat nulovými ukazateli (`NULL` v jazyku C, `nil` v Pascalu).

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.



Obr. 1.7: Dva $(2,3)$-stromy pro tutéž množinu klíčů

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

Důkaz: Půjdeme na to podobně jako u AVL stromů. Uvažujme, jak vypadá strom hloubky $h \geq 1$ s nejmenším možným počtem klíčů. Všechny jeho vrcholy musí mít minimální povolený počet synů (jinak by strom bylo ještě možné zmenšit). Vrcholy rozdělíme do hladin podle hloubky: na 0-té hladině je kořen se dvěma syny a jedním klíčem, úplně dole na h-té hladině leží vnější vrcholy bez klíčů. Na mezilehlých hladinách jsou všechny ostatní vnitřní vrcholy s a syny a $a-1$ klíči.

Na i-té hladině pro $0 < i < h$ bude tedy ležet $2 \cdot a^{i-1}$ vrcholů a v nich celkem $2 \cdot a^{i-1} \cdot (a-1)$ klíčů. Sečtením přes hladiny získáme minimální možný počet klíčů m_h:

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

Poslední sumu sečteme jako geometrickou řadu a dostaneme:

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

Vidíme 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.

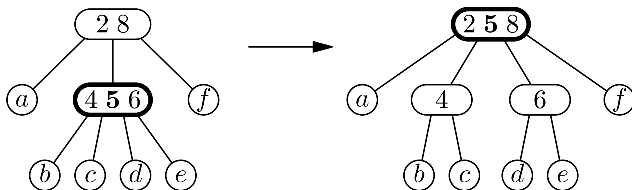
Vkládání do stromu

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 mu syna, ale jelikož ostatní synové jsou listy, tento může být též 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.

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,



Obr. 1.8: Štěpení přeplněného vrcholu při vkládání do $(2, 3)$-stromu

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 (vkládání do (a, b)-stromu)

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 (ℓ_1, x, ℓ_2), kde ℓ_1 a ℓ_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 zrušíme a nahradíme dvojicí synů p a q.
10. Pokud počet synů nepřekročil b, skončíme a vrátíme $\emptyset$.
11. $m \leftarrow \lfloor (b-1)/2 \rfloor$ $\quad$ ◁ *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 $s_m, \ldots, s_b$.

14. Vrátíme trojici (v_1, x_m, v_2).

Zbývá dokázat, že vrcholy vzniklé 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

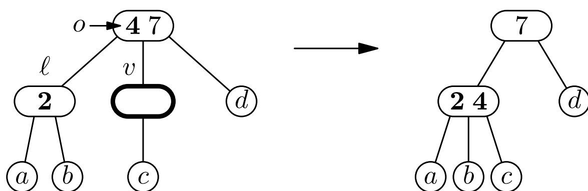
Chceme-li ze stromu smazat nějaký klíč, nejprve ho vyhledáme. Pokud se nachází na předposlední hladině (té, pod níž jsou už pouze listy), můžeme ho smazat přímo, jen musíme ošetřit případné podtečení vrcholu.

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 levého bratra ℓ odděleného nějakým klíčem o v otci. Pokud by existoval pouze pravý bratr, vybereme toho a následující postup provedeme zrcadlově převráceně.

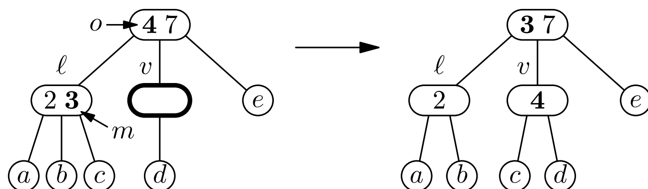
Pokud má bratr pouze a synů, sloučíme vrcholy v a ℓ do jediného vrcholu a přidáme do něj ještě klíč o z otce. Tím vznikne vrchol s $(a-2)+(a-1)+1 = 2a-2$ klíči, což není větší než $b-1$. Problém jsme tedy převedli na mazání klíče z otce, což je tentýž problém o hladinu výš.



Obr. 1.9: Sloučení vrcholů při mazání z $(2,3)$-stromu

Má-li naopak bratr více než a synů, odpojíme od něj jeho nejpravějšího syna c a největší klíč m. Poté klíč m přesuneme do otce a klíč o odtamtud přesuneme do v,

kde se stane nejmenším klíčem, před který přepojíme syna c. Poté mají v i ℓ povolené počty synů a můžeme skončit. (Všimněte si, že tato operace je podobná rotaci hrany v binárním stromu.)



Obr. 1.10: Doplnění vrcholu ve $(2, 3)$-stromu půjčkou od souseda

Nyní tento postup zapíšeme jako rekurzivní proceduru AbDelete2. Ta dostane kořen podstromu a klíč, který má smazat. Jako výsledek 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 (mazání z (a, b)-stromu)
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 $i \geq 1$: ◁ *existuje levý bratr s_{i-1}*
17. Pokud má s_{i-1} alespoň $a+1$ synů: ◁ *půjčíme si klíč*
18. Odpojíme z s_{i-1} největší klíč m a nejpravějšího syna c.
19. K vrcholu s_i připojíme jako první klíč x_i a jako nejlevějšího syna c.
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-1} a s_i a mezi nimi klíč x_i.
23. Z vrcholu v odstraníme klíč x_i a syny s_{i-1} a s_i. Tyto syny zrušíme a na jejich místo připojíme syna s.
24. Jinak provedeme kroky 17 až 23 zrcadlově pro pravého 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 $\Theta(1)$ 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 3). Proto se obvykle používají $(a, 2a-1)$-stromy, případně $(a, 2a)$-stromy. Volby $b = 2a-1$ a $b = 2a$ vedou na stejnou složitost operací v nejhorším případě, ale jak uvidíme ve cvičeních **??** a **??**, vedou na úplně jiné dlouhodobé chování struktury.

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 počítačích 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 vyplatí 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 5), 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. Dokažte, že procházíme-li obecný vyhledávací strom v symetrickém pořadí vrcholů, pravidelně se střídají vnitřní vrcholy s vnějšími. To znamená, že obsahují-li vnitřní vrcholy klíče $x_1, \ldots, x_n$, pak vnější vrcholy odpovídají intervalům $(-\infty, x_1)$, (x_1, x_2), (x_2, x_3), …, $(x_n, +\infty)$.

2.* Využijte předchozí cvičení k sestrojení obecnější varianty intervalových stromů z oddílu **??**. Hranice intervalů jsou tentokrát libovolná reálná čísla. Na počátku si strom pamatuje interval $(-\infty, +\infty)$, který pak umí v libovolném bodě podrozdělovat. Mimo to podporuje změny hodnot, intervalové dotazy a případně intervalové změny, stejně jako klasický intervalový strom.

3. 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 $2a$.

4.* 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?

5. Rozmyslete si, 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.

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

7. Navrhněte operaci Join(X, Y), která dostane dva (a, b)-stromy X a Y a sloučí je do jednoho. Může se přitom spolehnout na to, že všechny klíče z X jsou menší než všechny z Y. Zkuste dosáhnout složitosti $\mathcal{O}(\log|X| + \log|Y|)$.

8.* Navrhněte operaci Split(T, x), která zadaný (a, b)-strom T rozdělí na dva stromy. V jednom budou klíče menší než x, v druhém ty větší. Pokuste se o logaritmickou časovou složitost.

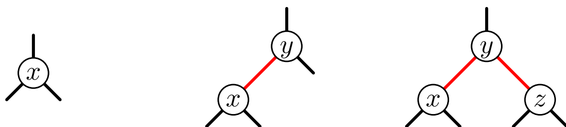
9. Nevýhodou (a, b)-stromů je, že plýtvají pamětí – může se stát, že vrcholy jsou zaplněné jen z poloviny. Navrhněte úpravu, která zaručí zaplnění z alespoň $2/3$.

1.4.* 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 poměrně 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 Robertem 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, rozlišíme dvě 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 pamatovat například v jejím spodním vrcholu.

Strom přeložíme podle následujícího obrázku. Vrcholům $(2,4)$-stromu budeme v závislosti na 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 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:

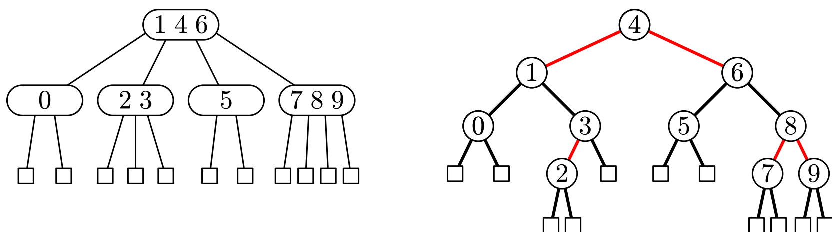
1. Neexistují dvě červené hrany bezprostředně nad sebou.
2. Jestliže z vrcholu vede dolů jediná červená hrana, pak vede doleva.
3. Hrany do listů jsou vždy obarveny černě. (To se hodí, jelikož listy jsou pouze virtuální, takže do nich neumíme barvu hrany uložit.)
4. Na všech cestách z kořene do listu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

Prvním dvěma axiomům budeme říkat červené, zbylým dvěma černé.

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.

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

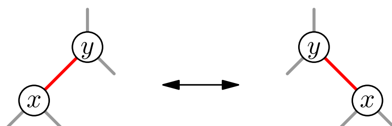
Důkaz: Hloubka $(2,4)$-stromu s n klíči činí $\Theta(\log n)$, překlad na LLRB strom počet hladin nesníží a nejvýše zdvojnásobí. □



Obr. 1.11: Překlad $(2, 4)$-stromu na LLRB strom

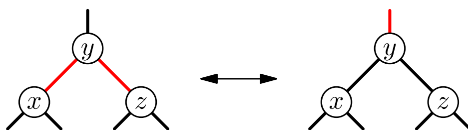
Vyvažovací operace

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 hrozilo porušení axiomu 4.)

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



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

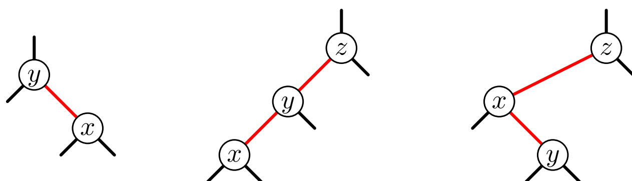
Dodejme ještě, že přebarvení jde použít i v kořeni. Můžeme si představovat, že do kořene vede shora nějaká virtuální hrana, již můžeme bez porušení axiomů libovolně přebarvovat.

Vkládání štěpením shora dolů

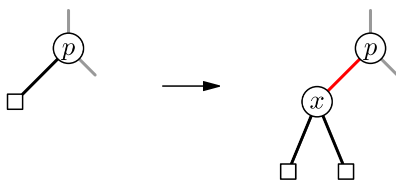
Nyní popíšeme, jak se do LLRB stromu vkládá. Půjdeme na to asi takto: místo pro nový vrchol budeme hledat obvyklým způsobem, ale kdykoliv cestou potkáme 4-vrchol, rovnou ho rozštěpíme přebarvením. Až dorazíme do listu, připojíme místo něj nový vnitřní vrchol a hranu, po které jsme přišli, obarvíme červeně. 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 z 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 červenou hranu zrotujeme. Tím z nekorektního 3-vrcholu uděláme korektní a z 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(v, x) (vkládání do LLRB stromu)

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 $(v, r(v))$ a do v uložíme původní $\ell(v)$.
7. Je-li $\ell(v)$ červený a $\ell(\ell(v))$ také červený, rotujeme hranu $(v, \ell(v))$ a do v uložíme původní $\ell(v)$.

Výstup: Nový kořen v

Vkládání štěpením zdola nahoru

Implementace vyšla překvapivě jednoduchá, ale to největší překvapení nás teprve čeká: Pokud v proceduře LlrbInsert přesuneme krok 3 za krok 7, dostaneme implementaci $(2, 3)$-stromů.

Vskutku: pokud se před vkládáním prvku ve stromu nenacházel žádný 4-vrchol, nepotřebujeme štěpení 4-vrcholů cestou dolů. Nový list tedy přidáme k 2-vrcholu nebo 3-vrcholu. Pokud dočasně vznikne 4-vrchol, rozštěpíme ho cestou zpět do kořene. Tím mohou vznikat další 4-vrcholy, ale průběžně se jich zbavujeme.

Tím jsme získali kód velice podobný proceduře BvsInsert pro nevyvažované stromy, pouze si musíme dávat pozor, aby nově vzniklé vrcholy dostávaly červenou barvu a abychom před každým návratem z rekurze zavolali následující opravnou proceduru:

Procedura LlrbFixup(v)

Vstup: Kořen podstromu v

1. Je-li $\ell(v)$ černý a $r(v)$ červený, rotujeme hranu $(v, r(v))$ a do v uložíme původní $r(v)$.
2. Je-li $\ell(v)$ červený a $\ell(\ell(v))$ také červený, rotujeme hranu $(v, \ell(v))$ a do v uložíme původní $\ell(v)$.
3. Jsou-li $\ell(v)$ i $r(v)$ červené, přebarvíme $\ell(v)$, $r(v)$ i v.

Výstup: Nový kořen podstromu v

Mazání minima

Mazání bývá o trochu složitější než vkládání a LLRB stromy nejsou výjimkou. Proto si zjednodušíme práci, jak to jen půjde.

Především využijeme toho, že se při vkládání umíme vyhnout 4-vrcholům, takže budeme předpokládat, že strom žádné neobsahuje. To speciálně znamená, že se nikde nevyskytuje pravá červená hrana.

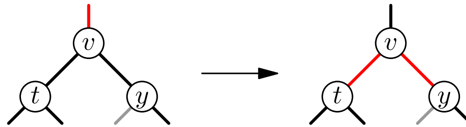
Také nám situaci zjednoduší, že se voláním LlrbFixup při návratu z rekurze umíme zbavovat případných nekorektních 3-vrcholů a jakýchkoliv (potenciálně i nekorektních) 4-vrcholů. Proto nevadí, když během mazání nějaké vyrobíme.

Než přikročíme k obecnému mazání, rozmyslíme si, jak smazat minimum. Najdeme ho tak, že z kořene půjdeme stále doleva, až narazíme na vrchol v, jehož levý syn je vnější. Všimněte si, že pravý syn musí být také vnější. Pokud by do v vedla shora červená hrana, mohli bychom v smazat a nahradit vnějším vrcholem. To odpovídá situací, kdy mažeme klíč z 3-vrcholu.

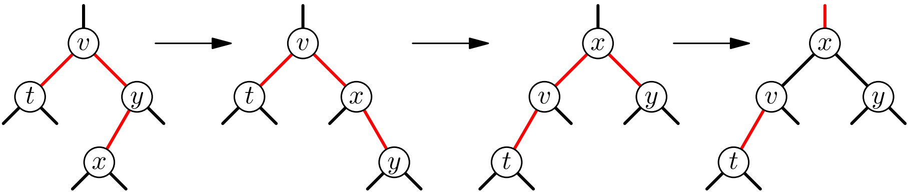
Horší je, jsou-li všechny hrany okolo v černé. Ve $(2, 3)$-stromu jsme tedy potkali 2-vrchol, takže ho potřebujeme sloučit se sousedem, případně si od souseda půjčit klíč. Jak už se nám osvědčilo v první verzi vkládání, budeme to provádět preventivně při průchodu shora dolů, takže až opravdu dojde na mazání, žádný problém nenastane. Cestou proto budeme dodržovat:

Invariant L: Stojíme-li ve vrcholu v, pak vede červená hrana buďto shora do v, nebo z v do jeho levého syna. Výjimku dovolujeme pro kořen.

Jelikož z v pokaždé odcházíme doleva, jediný problém nastane, vede-li z v dolů levá černá hrana a pod ní je další taková. Co víme o hranách v okolí? Shora do v vede díky invariantu červená. Všechny pravé hrany jsou, jak už víme, černé. Situaci se pokusíme napravit přebarvením všech hran okolo v:



Invariant opět platí, ale pokud měl pravý syn levou červenou hranu, vyrobili jsme nekorektní 5-vrchol, navíc v místech, kudy se později nebudeme vracet. Poradíme si podle následujícího obrázku: rotací hrany yx, rotací hrany vx a nakonec přebarvením v okolí x.



Celou funkci pro nápravu invariantu můžeme napsat takto (opět předpokládáme barvy uložené ve spodních vrcholech hran):

Procedura MoveRedLeft(v)

Vstup: Kořen podstromu v

1. Přebarvíme v, $\ell(v)$ a $r(v)$.
2. Pokud je $\ell(r(v))$ červený:
3. Rotujeme hranu $(r(v), \ell(r(v)))$.

4. $\quad x \leftarrow r(v)$
5. $\quad$ Rotujeme hranu (v, x).
6. $\quad$ Přebarvíme x, $\ell(x)$ a $r(x)$.
7. $\quad v \leftarrow x$

Výstup: Nový kořen podstromu v

Jakmile umíme dodržet invariant, je už mazání minima snadné:

Procedura LLRBDELETEMIN(v) (mazání minima z LLRB stromu)
Vstup: Kořen stromu v

1. Pokud $\ell(v) = \emptyset$, položíme $v \leftarrow \emptyset$ a skončíme.
2. Pokud $\ell(v)$ i $\ell(\ell(v))$ jsou černé:
3. $\quad v \leftarrow$ MOVEREDLEFT(v)
4. $\ell(v) \leftarrow$ LLRBDELETEMIN($\ell(v)$)
5. $v \leftarrow$ LLRBFIXUP(v)

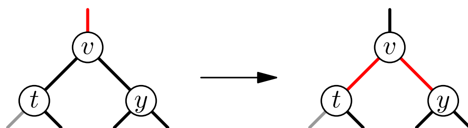
Výstup: Nový kořen v

Mazání maxima

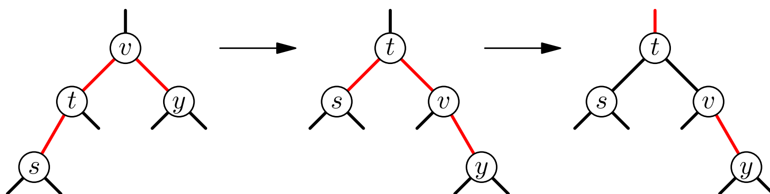
Nyní se naučíme mazat maximum. U obyčejných vyhledávacích stromů je to zrcadlová úloha k mazání minima, ne však u LLRB stromů, jejichž axiomy nejsou symetrické. Bude se každopádně hodit dodržovat stranově převrácenou obdobu předchozího invariantu:

Invariant R: Stojíme-li ve vrcholu v, pak vede červená hrana buďto shora do v, nebo z v do jeho pravého syna. Výjimku dovolujeme pro kořen.

Na cestě z kořene k maximu půjdeme stále doprava. Do pravého syna červená hrana sama od sebe nevede, ale pokud nějaká povede doleva, zrotujeme ji a tím invariant obnovíme. Problematická situace nastane, vedou-li z v dolů černé hrany a navíc z pravého syna vede doleva další černá hrana. Tehdy se inspirujeme mazáním minima a přebarvíme hrany v okolí v:



Pokud z t vede doleva černá hrana, je vše v pořádku. V opačném případě jsme nalevo vytvořili nekorektní 4-vrchol, který musíme opravit. Pomůže nám rotace hrany vt a přebarvení v okolí t:



Tato úvaha nás dovede k následující funkci pro opravu invariantu, na níž založíme celé mazání maxima.

Procedura MoveRedRight(v)
Vstup: Kořen podstromu v

1. Přebarvíme v, $\ell(v)$ a $r(v)$.
2. Pokud je $\ell(\ell(v))$ červený:
3. $\quad t \leftarrow \ell(v)$
4. $\quad$ Rotujeme hranu (v, t).
5. $\quad$ Přebarvíme t, $\ell(t)$ a $r(t)$.
6. $\quad v \leftarrow t$

Výstup: Nový kořen podstromu v

Procedura LlrbDeleteMax(v) (mazání maxima z LLRB stromu)
Vstup: Kořen stromu v

1. Pokud $\ell(v)$ je červený, rotujeme hranu $(v, \ell(v))$.
2. Pokud $r(v) = \emptyset$, položíme $v \leftarrow \emptyset$ a skončíme.
3. Pokud $r(v)$ i $\ell(r(v))$ jsou černé:
4. $\quad v \leftarrow$ MoveRedRight(v)
5. $r(v) \leftarrow$ LlrbDeleteMax($r(v)$)
6. $v \leftarrow$ LlrbFixup(v)

Výstup: Nový kořen v

Mazání obecně

Pro mazání obecného prvku nyní stačí vhodně zkombinovat myšlenky z mazání minima a maxima. Opět půjdeme shora dolů a budeme se vyhýbat tomu, abychom skončili ve 2-vrcholu. Pomůže nám k tomu tato kombinace invariantů **L** a **R**:

Invariant D: Stojíme-li ve vrcholu v, pak vede červená hrana buďto shora do v, nebo do syna, kterým se chystáme pokračovat. Výjimku dovolujeme pro kořen.

Pokud při procházení shora dolů chceme pokračovat po levé hraně, použijeme trik z mazání minima a pokud by pod námi byly dvě levé černé hrany, napravíme situaci pomocí MoveRedLeft. Naopak chceme-li odejít pravou hranou, chováme se jako při mazání maxima a v případě problémů povoláme na pomoc MoveRedRight.

Po čase najdeme vrchol, který chceme smazat. Má-li pouze vnější syny, můžeme ho přímo nahradit vnějším vrcholem. Jinak použijeme obvyklý obrat: vrchol nahradíme minimem z pravého podstromu, čímž problém převedeme na mazání minima, a to už umíme.

Procedura LlrbDelete(v, x) (mazání z LLRB stromu)
Vstup: Kořen stromu v, mazaný klíč x

1. Pokud $v = \emptyset$, vrátíme se. $\triangleleft$ *klíč x ve stromu nebyl*
2. Pokud $k(v) < x$: $\triangleleft$ *pokračujeme doleva jako při mazání minima*

3. Pokud $\ell(v)$ i $\ell(\ell(v))$ existují a jsou černé:
4. $v \leftarrow$ MoveRedLeft(v)
5. $\ell(v) \leftarrow$ LlrbDelete$(\ell(v), x)$
6. Jinak: ◁ *buďto hotovo, nebo doprava jako při mazání maxima*
7. Pokud $\ell(v)$ je červený, rotujeme hranu $(v, \ell(v))$.
8. Pokud $k(v) = x$ a $r(v) = \emptyset$:
9. $v \leftarrow \emptyset$ a skončíme.
10. Pokud $r(v)$ i $\ell(r(v))$ existují a jsou černé:
11. $v \leftarrow$ MoveRedRight(v)
12. Pokud $k(v) = x$:
13. Prohodíme $k(v)$ s minimem pravého podstromu $R(v)$.
14. $r(v) \leftarrow$ LlrbDeleteMin$(r(v))$
15. Jinak:
16. $r(v) \leftarrow$ LlrbDelete$(r(v), x)$
17. $v \leftarrow$ LlrbFixup(v)

Výstup: Nový kořen v

Časová složitost

Ukázali jsme tedy, jak pomocí binárních stromů kódovat $(2,4)$-stromy, nebo dokonce $(2,3)$-stromy. Časová složitost operací Find, Insert i Delete je zjevně lineární s hloubkou stromu a o té jsme již dokázali, že je $\Theta(\log n)$.

Dodejme na závěr, že existují i jiné varianty červeno-černých stromů, které jsou založeny na podobném překladu (a,b)-stromů na binární stromy. Některé z nich například zaručují, že při každé operaci nastane pouze $\mathcal{O}(1)$ rotací. Je to ovšem vykoupeno podstatně složitějším rozborem případů. Časová složitost samozřejmě zůstává logaritmická, protože je potřeba prvek nalézt a přebarvovat hrany.

Cvičení

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

2.* Navrhněte, jak z LLRB stromu mazat, aniž bychom museli při průchodu shora dolů rotovat. Všechny úpravy struktury provádějte až při návratu z rekurze podobně, jako se nám to podařilo při vkládání.

3. LLRB stromy jsou asymptoticky stejně rychlé jako AVL stromy. Zamyslete se nad jejich rozdíly při praktickém použití.

1.5. Další cvičení

1. Uspořádejme všechny permutace na množině $\{1, \ldots, n\}$ lexikograficky. Vymyslete algoritmus, který pro dané k sestrojí v pořadí k-tou permutaci v čase $\mathcal{O}(n \log n)$. Navrhněte též převod permutace na její pořadové číslo.

2. Vymyslete jiné uspořádání všech permutací, v němž půjde mezi permutací a jejím pořadovým číslem převádět v lineárním čase.
3. Dokažte, že budeme-li reprezentovat množiny binárními vyhledávacími stromy, nelze sjednocení provést rychleji než lineárně v nejhorším případě. Platí to dokonce i tehdy, máme-li na vstupu zaručený dokonale vyvážený strom a výstup může být jakkoliv nevyvážený.
4. *Okénkový medián:* Na vstupu postupně přicházejí čísla. Kdykoliv přijde další, vypište medián z posledních k čísel. Dosáhněte časové složitosti $\mathcal{O}(\log k)$ na operaci.
5. Dokažte, že u předchozího cvičení je čas $\Theta(\log k)$ nejlepší možný, pokud umíme čísla pouze porovnávat.
6. Sestrojte datovou strukturu pro uložení seznamu tak, abychom uměli rychle najít k-tý prvek a přesunout ho na začátek.