Algoritmy 2 - vyvážené binární stromy

Petr Osička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Vyvážený strom

- Chceme udržovat výšku stromu s n vrcholy tak, aby složitost operací (závislých lineárně na výšce stromu) zůstala $O(\lg n)$. O stromech, pro které toto platí, mluvíme jako o vyvážených stromech.
- Obvyklý postup:
  1. Definujeme podmínku, která pro konkrétní strom platí nebo neplatí.
  2. Dokážeme, že v množině stromů splňujících naši podmínku platí, že funkce $h : \mathbb{N} \to \mathbb{N}$, kde $h(n)$ je maximální výška stromu s n vrcholy, je seshora omezena $O(\lg n)$.
  3. Upravíme operace modifikující strom, aby podmínku zachovávali (= když podmínka platí před operací, platí i po operaci).
  4. Existuje několik přístupů.

AVL STROMY

Varianta binárního vyhledávacího stromu.

Vyváženost vrcholu x je rozdíl výšky levého podstromu a výšky pravého podstromu vrcholu x. Výšku prázdného podstromu definujeme rovnu -1.

Strom je *přípustný*, pokud je vyváženost každého vrcholu v něm rovna 1, 0, nebo -1.

Věta

Existuje konstanta $c > 0$ tak, že výška každého přípustného stromu s n vrcholy je menší nebo rovna $c \lg n$.

Důkaz. (hlavní idea)

Pokusíme se namalovat přípustný strom, který má výšku $0, 1, 2, \ldots$ tak, aby obsahoval co nejméně vrcholů (u těchto stromů bude funkce popisující výšku stromu v závislosti na počtu vrcholů nejvíce rostoucí).

Všimneme si, že náš strom výšky m má jako levý podstrom náš strom výšky $m - 1$ a jako pravý podstrom náš strom výšky $m - 2$. Proto, pokud počet vrcholů ve stromu výšky m označíme $T(m)$, platí

$$\begin{aligned} T(m) &= T(m-1) + T(m-2) + 1, \\ T(0) &= 1, \\ T(1) &= 2. \end{aligned}$$

Řešením předchozí rekurence je exponenciální funkce (všimněme si, že rekurence je velmi podobná definici Fibonacciho posloupnosti). Funkce vyjadřující výšku v závislosti na počtu vrcholů, které je k T inverzní funkcí, tedy musí být logaritmická funkce.

Úpravy insert a delete

`insert` a `delete` v binárním stromu mohou transformovat přípustný strom na nepřípustný.

- do struktury pro vrchol přidáme položku `bf` pro vyváženost vrcholu (tu budeme udržovat konzistentní),
- provedeme přidání/odebrání jako u obyčejného binárního vyhledávacího stromu,
- procházíme cestu od místa změny do kořene, opravujeme položky `bf`,
- pokud narazíme na nepřípustnou hodnotu položky `bf`, situaci opravíme.

Princip úpravy položky bf

- Položku `x.bf` upravujeme pouze tehdy, pokud se změnila výška některého z podstromů `x.left`, `x.right`.
- Protože vždy přidáváme nebo odebíráme jeden vrchol (a rotace, které si ukážeme níže, mění výšku podstromu max o 1), může se výška podstromu vrcholu x změnit maximálně o 1.

podstrom	výška se zvětšila	výška se zmenšila
`x.left`	$+1$	-1
`x.right`	-1	$+1$

Nepovolené hodnoty položky `bf`

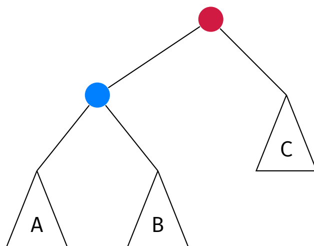
Bude se jednat pouze o hodnoty `2` a `-2`.

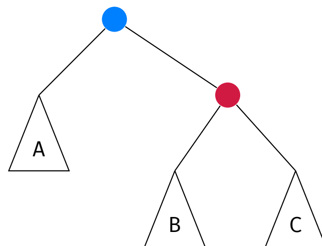
Všechny úpravy stromy totiž změní výšky podstromů nejvýše o 1. To je zřejmé u (neupravených) operací `insert` a `delete`. U dalších úprav to také odvodíme.

Pokud tedy narazíme na vrchol `x` a po úpravě bude mít `x.bf` hodnotu `2` nebo `-2`, pak

1. upravíme podstrom s kořenem `x` na přípustný strom (rotace na dalších slajdech). Upravený podstrom bude mít nový kořen, který připojíme se zbytkem stromu na místě, na kterém byl připojen `x`.
2. úprava z předchozího bodu může změnit výšku upravovaného podstromu, tj. podstrom s kořenem `x` může mít jinou výšku, než přípustný strom, který dostaneme jeho úpravami (výška se změní maximálně o `1`)

JEDNODUCHÉ ROTACE (PRAVÁ A LEVÁ)

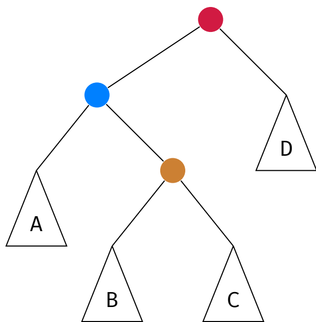


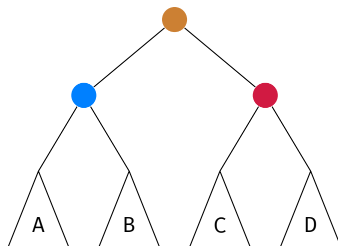


⟶	●	●	●	●	změna výšky
	2	1	0	0	-1
	2	0	1	-1	0

⟵	●	●	●	●	změna výšky
	-2	-1	0	0	-1
	-2	0	1	-1	0

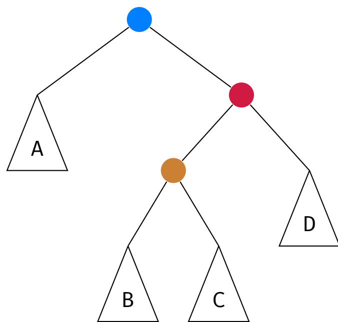
Dvojitá rotace (levo pravá)

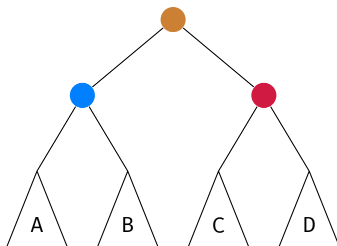




$\longrightarrow$	●	●	●	●	●	●	změna výšky
	2	−1	0	0	0	0	−1
	2	−1	1	0	0	−1	−1
	2	−1	−1	0	1	0	−1

Dvojitá rotace (pravo levá)





$\longrightarrow$	●	●	●	●	●	●	změna výšky
	-2	1	0	0	0	0	-1
	-2	1	-1	0	1	0	-1
	-2	1	1	0	0	-1	-1

Připomenutí struktur

```
struct Node
   id: Key
   left: Node
   right: Node
   parent: Node
   bf : Int — vyváženost
```

```
struct Avl-Tree
   root: Node — kořen stromu
```

Vložení vrcholu

```
avl-insert
⟵ t: Avl-Tree – strom, do kterého přidáváme
⟵ added: Node – přidaný vrchol
```

```
1   vložíme added jako do binárního vyhledávacího stromu
2   nastavíme added.bf ← 0; u ← added.parent; v ← added
3   dokud u ≠ nil
4       u.bf ← u.bf + { 1   pokud v = u.left
                       {-1  pokud v = u.right
5       pokud u.bf = 0, algoritmus končí
6       pokud u.bf = 2 nebo u.bf = -2
7           p ← u.parent
8           provedeme příslušnou rotaci na strom s kořenem u, obdržíme strom s kořenem w
9           pokud p = nil, nastavíme t.root ← w, jinak připojíme w jako příslušného potomka p
10          algoritmus končí
11      nastavíme v ← u, u ← u.parent
```

Komentáře

V cyklu na řádcích 3 až 11 procházíme s pomocí proměnných `u,v` cestu od přidaného vrcholu do kořene, `v` je vždy potomkem `u`.

Na řádku 4 v aktuálním vrcholu upravíme vyváženost vrcholu `u`.

Pokud je (takto aktualizovaná) `u.bf` rovna `0`, algoritmus končí (řádek 5). Pokud je rovna `1` nebo `-1`, pokračujeme na cestě ke kořeni (řádek `11`).

Pokud je `u.bf` rovna `-2` nebo `2`, a přidání porušilo přípustnost stromu, provedeme správnou rotaci a algoritmus končí (řádky 6 až 10)

Složitost: V nejhorším případě procházíme strom od přidaného vrcholu do kořene. Pro každý navštívený vrchol provedeme konstatní počet operací, složitost je tak lineárně závislá na výšce stromu. Vzhledem k větě o výšce přípustného stromu tak máme složitost $\Theta(\lg n)$.

Věta (korektnost úprav položky bf)

Pokud algoritmus provádíme na přípustném stromu, pak:

- (a) Na řádku 3 (i po každé iteraci cyklu na řádcích 3 až 11) je výška podstromu s kořenem v o jedna větší, než byla před přidáním provedeným na řádku 1.
- (b) Po provedení řádku 4 je hodnota u.bf rovna vyváženosti vrcholu u.

Důkaz.

a) Indukcí přes počet provedení cyklu. Výšku podstromu s kořenem v budeme značit $h(v)$.

Před prvním provedením cyklu je v přidaný vrchol. Před přidáním mu odpovídal prázdný podstrom, máme tedy $0 = h(v) = h(\mathtt{nil}) + 1$.

Předpokládáme, že tvrzení platí na řádku 3 na začátku i-té iterace cyklu. Jediný způsob, jak se dostat opět na řádek 3 je provedení řádku 11. Před jeho provedením máme $\mathtt{u.bf} = \pm 1$. To znamená, že před provedením řádku 4 aktuální iterace jsme museli mít $\mathtt{u.bf} = 0$ (vzpomeňme si, že před přidáním byl strom přípustný). Tj. před přidáním platilo $h(u) = h(v) + 1$. Z indukčního předpokladu víme, že $h(v)$ je po přidání o jedna větší než před ním, a tím pádem je po přidání o jedna větší i $h(u)$. Tvrzení potom plyne z přiřazení $v \leftarrow u$ na řádku 11.

b) plyne z a) a řádku 4.

Věta (korektnost ukončení algoritmu řádkem 5)

Pokud algoritmus provádíme na přípustném stromu a ba řádku 5 je hodnota `u.bf` rovna 0, jsou výšky podstromů s kořen u před a po provedení přidání na řádku 1 stejné.

Důkaz.

Z řádku 4 algoritmu, z předchozí věty a z toho, že před přidáním byl strom přípustný plyne, že před provedením řádku 4 byla hodnota `u.bf` rovna `1` nebo `-1`. Přitom na řádku 5 máme `u.bf = 0`.

Podstrom s kořenem v tedy musel mít před přidáním menší výšku než jeho sourozenec.

Zvýšením výšky podstromu s kořenem v o jedna (viz předchozí Tvrzení) se proto výška podstromu s kořenem u nezměnila. □

Věta (korektnost ukončení algoritmu rotací)

Pokud algoritmus provádíme na přípustném stromu, je po provedení řádku 8 algoritmu výška stromu s kořenem `w` stejná jako byla výška stromu s kořenem u před přidáním vrcholu na řádku 1.

Důkaz.

Pokud dojde k rotaci, musí být `u.bf` $= \pm 2$. Před provedením řádku 4 muselo být `u.bf` $= \pm 1$. Výška podstromu s kořenem u je proto o jedna větší po přidání na řádku 1 než před ním. (argument je analogický tomu z předchozího důkazu).

Pokud ukážeme, že použijeme rotaci, které sníží výšku stromu o jedna, bude důkaz hotov. Jediná rotace, která nesnižuje výšku stromu je jednoduchá rotace v situaci, kdy `u.bf` $= \pm 2$ a `v.bf` $= 0$. (Vzpomeňme si, že `u.bf` na řádku 4 měníme proto, že se oproti situaci před přidáním na řádku 1 zvýšila výška podstromu s kořenem v, který tak musí mít větší výšku než jeho sourozenec.) Stav, kdy `v.bf` $= 0$ ovšem nemůže nastat, protože v minulé iteraci cyklu (kdy byl vrchol v vlastně vrcholem u) by algoritmus skončil na řádku 5. □

Věta (Koreknost algoritmu `avl-insert`)

Nechť T je přípustný strom. Potom je strom, který získáme pomocí přidání vrcholu do T algoritmem `avl-insert`, taky přípustný.

Důkaz. Po přidání vrcholu se mohou změnit pouze vyváženosti vrcholů, které jsou na cestě z přidaného vrcholu do kořene, ostatním vrcholům se totiž nezmění výšky podstromů. Algoritmus tuto cestu prochází. Po cestě mění položky `bf` tak, aby odpovídali skutečné hodnotě vyváženosti (viz Věta o korektnosti úprav položky `bf`). Přitom při pohybu směrem ke kořeni mají všechny již zpracované vrcholy vyváženost ± 1. Pokud tedy algoritmus projde celou cestu až do kořene, je strom vyvážený.

Pokud algoritmus skončí předčasně, je výška podstromu, jehož kořenem je aktuálně zpracovaný vrchol (tj. buď u na řádku 5, nebo `w` na řádku 9), stejná, jako výška odpovídajícího podstromu před přidáním vrcholu na řádku 1 (viz předchozí dvě věty). Vyváženosti zbývajících vrcholů na cestě do kořene tedy nemusíme upravovat, protože výšky jejich podstromů se totiž nezměnili. □

Odebrání vrcholu

Potřebujeme výčtový typ = nový typ určený množinou možných hodnot

```
enum Direction {left, right}
```

Upravíme procedury:

- `node-swap`
  Do vrcholu `v` (kterým nahrazujeme), zkopírujeme hodnotu položky `u.bf` (vrcholu `u`, který nahrazujeme).
- `tree-delete`
  vrátí dvojici `(u: Node, from: Direction)`, kde `u` je nejhlubší vrchol, kterému se v důsledku odebrání zmenší některý z podstromů a `from` určuje, kterému z podstromů to je. Označme odebíraný vrchol `z`.
  - `z` *nemá dva potomky:*
    Pokud `z == z.parent.left` funkce vrací `(z.parent, left)`, jinak vrací `(z.parent, right)`.
  - `z` *má dva potomky:*
    Nechť `p` je pořádkový následník `z`. Pokud je `p == z.right`, potom funkce vrací `(p,right)`. Jinak vrací `(p.parent,left)`.

```
avl-delete
 ⟵ t: Aval-Tree
 ⟵ z: Node
```

```
1   u, from ← tree-delete(t,z)
2   dokud u ≠ nil
3       u.bf ← u.bf + { -1   pokud from = left
                        { 1    pokud from = right
4       pokud u.bf = 1 nebo u.bf = -1, algoritmus končí
5       p ← u.parent
6       from ← { left    p ≠ nil a p.left = u
               { right   jinak
7       pokud u.bf = 2 nebo u.bf = -2
8           provedeme příslušnou rotaci na strom s kořenem u, získáme strom s kořenem w
9           pokud p = nil, nastavíme t.root ← w, jinak připojíme w jako příslušného potomka p
10          pokud rotace nesnížila výšku stromu (kořeny u vs w), potom algoritmus končí.
11      u ← p
```

Komentáře

V cyklu na řádcích 2 až 11 procházíme cestu od místa odebrání vrcholu ke kořeni. V proměnné `from` si pamatujeme, jestli jsme k aktuálnímu vrcholu přisli z levého nebo pravého podstromu.

Na řádku 3 upravujeme hodnotu položky `bf` na korektní hodnotu.

V případě, že by hodnota `bf` byla nepřípustná, provedeme rotaci a situaci opravíme.

Algoritmus končí v případě, že dojdeme do kořene, nebo již není nutné opravovat položky `bf` zbývajících vrcholů na cestě do kořene.

Složitost: V nejhorším případě procházíme strom od přidaného vrcholu do kořene. Pro každý navštívený vrchol provedeme konstatní počet operací, složitost je tak lineárně závislá na výšce stromu. Vzhledem k větě o výšce přípustného stromu tak máme složitost $\Theta(\lg n)$.

Věta (korektnost úprav položky bf)

Spustíme-li algoritmus odebrání vrcholu na přípustný strom, potom na řádku 2 algoritmu platí, že výška příslušného podstromu vrcholu u (tj. levého pokud `from` $=$ `left` a pravého pokud `from` $=$ `right`) je o jedna menší než před provedením řádku 1 algoritmu.

Důkaz Indukcí přes počet provedení cyklu na řádcích 2 až 11. Při prvním provádění řádku 2 plyne tvrzení z úprav operace `tree-delete`.
Předpokládejme, že tvrzení platí na začátku i-té iterace cyklu. Na řádek 2 se opakovaně můžeme dostat, pokud `u.bf` $\neq \pm 1$ na řádku 4. Máme tak možnosti

- `u.bf` $= 0$ a `u.bf` $= \pm 1$ před úpravou na řádku 3. Výška podstromu s kořenem u byla před provedením řádku 1 dána výškou toho podstromu u, jehož výšku jsme snížili o 1 (indukční předpoklad a řádky 5,6). Proto se snížila i výška podstromu s kořenem u.
- `u.bf` $= \pm 2$ a `u.bf` $= \pm 1$ před úpravou `u.bf` na řádku 3. Indukční předpoklad tedy můžeme aplikovat na výšku toho podstromu vrcholu u, který měl menší výšku už před provedením řádku 1 (v porovnání s druhým podstromem vrcholu u). Proto může výšku odpovídajícího podstromu (porovnáváme výšky u před rotací a `w` po rotaci) snížit pouze rotace, která snižuje výšku stromu o 1. To ale přesně odpovídá řádku 10.

□

Věta (korektnost předčasného ukončení algoritmu)

Pokud jsme algoritmus spustili na přípustný strom a je-li po úpravě v kroku 3 $\texttt{u.bf} = \pm 1$ nebo jsme na řádku 8 provedli rotaci, která nesnížila výšku rotovaného stromu, je výška podstromu s kořenem u (v případě rotace výška podstromu s kořenem w) stejná, jako byla výška podstromu s kořenem u před provedením řádku 1.

Důkaz

Pokud $\texttt{u.bf} = \pm 1$ po řádku 3, pak $\texttt{u.bf} = 0$ před řádkem 3. Oba podstromy vrcholu u měli před řádkem jedna stejnou výšku. Snížením výšky jednoho z nich (viz předchozí věta), jsme výšku podstromu s kořenem u nezmenšili.

Pokud dojde k rotaci, je analýza situace analogická poslednímu odstavci předchozího důkazu. □

Věta (korektnost algoritmu `avl-delete`)

Nechť T je přípustný strom. Strom, který získáme pomocí algoritmu odebrání vrcholu ze stromu T, je také přípustný

Důkaz

Po odebrání vrcholu se mohou změnit vyváženosti vrcholů, které jsou na cestě z rodiče skutečně odstraněného vrcholu do kořene. Algoritmus tuto cestu prochází a všem vrcholům, které navštíví, korektně upraví položky `u.bf` a v případě jejich nepřípustnosti provede rotaci. Pokud algoritmus skončí předčasně, je to v situaci, kdy se výška aktuálního podstromu oproti situaci před spuštěním algoritmu nezměnila a není tedy nutné upravovat položku `bf` zbývajících vrcholů na cestě do kořene.

RED-BLACK STROMY

Do vrcholů přidáme položku `color`, která může nabývat hodnot `red` a `black`. (Mluvíme pak o *červených* a *černých* vrcholech).

Místo hodnoty `nil` použíme speciální vrchol `NIL`. Všechny listy jsou tak `NIL` vrcholy.

Dále definujeme podmínku vymezující red-black strom.

1. Kořen je černý.
2. `NIL` je černý.
3. Pokud je vrchol červený, oba jeho potomci jsou černí.
4. Pro každý vrchol platí, že všechny cesty z něj do listů obsahují stejný počet černých vrcholů.

Pro vrchol x zavedeme jeho *černou výšku* $\text{BH}(x)$ jako počet černých vrcholů na cestě z x do listů, přičemž samotný x se do toho nepočítá. Speciálně pak zavedeme $\text{BH}(\texttt{NIL}) = 0$.

(*Všimněme si, že v red-black stromu je to korektní definice, všechny cesty do listů obsahují stejný počet černých vrcholů*).

Černá výška stromu je potom černá výška jeho kořene.

Tvrzení

Podstrom s kořenem x má nejméně $2^{\text{BH}(x)} - 1$ vrcholů, které nejsou $\texttt{NIL}$.

Důkaz

Indukcí přes výšku podstromu s kořenem x.

Pokud je výška podstromu rovna 0, obsahuje pouze x, který je $\texttt{NIL}$. Přitom máme $\text{BH}(x) = 0$. Přitom $2^0 - 1 = 0$.

Předpokládejme, že podstrom s kořenem x má výšku $h > 0$ a že pomocné tvrzení platí pro všechny (pod)stromy s menší výškou.

Víme, že x není NIL. Potom levý a pravý podstrom x mají výšku nejvýše $h - 1$. Přitom je jejich černá výška $\text{BH}(x) - 1$. Pokud je totiž potomek x červený, je jeho černá výška $\text{BH}(x)$, je-li černý, je to $\text{BH}(x) - 1$.

Z indukčního předpokladu tak máme, že podstrom s kořenem x má nejméně

$$2 \cdot (2^{\text{BH}(x)-1} - 1) + 1 = 2^{\text{BH}(x)} - 1$$

vrcholů, které nejsou NIL.

Věta (Výška red-black stromu)

Red-black strom s n vrcholy, které nejsou `NIL`, má výšku nejvýše $2\lg(n+1)$.

Důkaz.

Z podmínky 3 v definici red-black stromu (červený vrchol nemá červeného potomka) plyne, že černé vrcholy tvoří nejméně polovinu vrcholů na cestě z kořene stromu do listu. Proto je černá výška kořene nejméně $h/2$, kde h je výška stromu. S použitím předchozího tvrzení tak máme

$$n \geqslant 2^{h/2} - 1.$$

Úpravami dostaneme $h \leqslant 2\lg(n+1)$. □

Připomenutí struktur

`enum Color { red, black }` — *výčtový typ pro barvy*

```
struct Node
   id: Key
   left: Node
   right: Node
   parent: Node
   color: Color — barva
```

```
struct RB-tree
   root: Node — kořen stromu
```

Vkládání

Vrchol vkládáme nabarvený na červeno. Oba jeho potomci jsou `NIL`.

Po jeho vložení (jako do klasického binárního vyhledávacího stromu) mohou být porušeny podmínky 1 (červený kořen) nebo 3 (červený vrchol má červeného potomka).

Porušení podmínek opravíme, začneme od přidávaného vrcholu.

```
rb-insert
 ⟵ t: RB-tree
 ⟵ added: Node
```

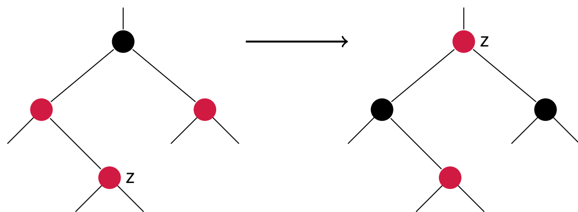
```
1  added.left ← NIL
2  added.right ← NIL
3  added.color ← red
4  tree-insert(t, added)
5  rb-fixup(t, added) — procedura, která opraví strom
```

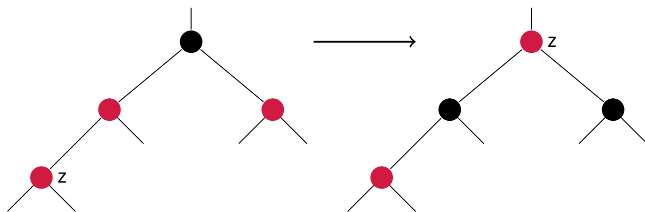
```
rb-fixup
 ⟵ t: RB-tree
 ⟵ z: Node — aktuální vrchol
```

```
1  while (z ≠ t.root)
2     if (z.parent.color = black)
3        return
3     z ← local-fix(t,z) — procedura pro opravy, viz dalsi slajdy
4  t.root.color ← black
```

Procedura `local-fix` detekuje jeden z možných případů. Poté provede úpravy a vrátí vrchol, kde mají úpravy pokračovat. Případy a jejich řešení si ukážeme na obrázcích.

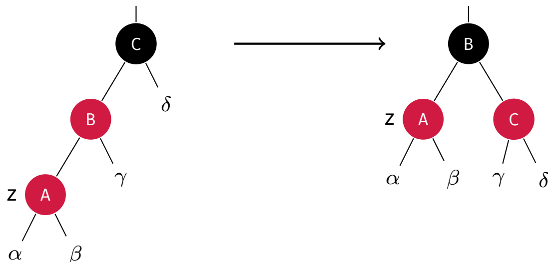
Pokud je strýc vrcholu z červený, stačí přebarvení. Algoritmus pokračuje s novým vrcholem z.





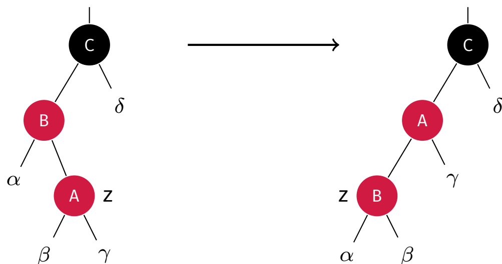
Korektnost: platí podmínka 3 a všem vrchlům v celém podstromu zůstala stejná černá výška, mimo kořen, kterému se černá výška zvětšila o 1. Oba potomci kořene jsou černí, proto podmínka 4 pořád platí.

Pokud je strýc vrcholu z černý (na obrázku je to kořen podstromu δ) a z je levý potomek, provedeme pravou rotaci na vrchol C a prohodíme barvy vrcholů B a C.



Korektnost: Černá výška žádného z podstromů α, β, γ, δ nemění. Černá výška vrcholu B po změně se tak rovná černé výšce vrcholu C před změnou. Černé výšky všech vrcholů ve stromu tak zůstavají stejné. Podmínka 4 není porušena.

Pokud je strýc vrcholu z černý (na obrázku je to kořen podstromu δ) a z je pravý potomek, převedeme levou rotací vrcholu B na předchozí případ.



Pokud je u předchozích úprav vrchol z v pravém podstromu svého dedečka, použijeme symetrické rotace (prohodíme levou a pravou stranu).

Složitost operace `rb-insert` je lineárně závislá na výšce stromu. Po vlastním vložení do stromu, jehož složitost je lineárně závislá na výšce, provedeme buď maximálně 2 rotace, případně se při přebarvování posuneme od přidaného vrcholuu do kořene.

Protože je výška red-black stromu omezena logaritmem, je složitost operace $\Theta(\lg n)$.

Odebírání

Upravíme procedury

- `node-swap`
  Do vrcholu, kterým nahrazujeme zkopírujeme hodnotu položky `color` z vrcholu, který nahrazujeme.
- `tree-delete`
  vrátí dvojici `(u:Node, color:Color)`, kde u je potomek skutečně odebraného vrcholu a `color` je barva skutečně odebraného vrcholu. Označme odebíraný vrchol `z`.
  - *`z` nemá dva potomky*:
    `col` je `z.color` a u je buď potomek `z`, který není `NIL`, nebo `NIL`, pokud `z` takového potomka nemá.
  - *`z` má 2 potomky*
    vrátíme hodnoty získané v předchozím bodu pro pořádkového následníka `z`

```
rb-delete
 ⟵ t: RB-tree
 ⟵ z: Node — odebíraný vrchol
```

```
1  u, col ← tree-delete(t,z)
2  if (col == red)
3     return
4  rb-delete-fixup(t,u)
```

Po odebrání černého vrcholu (v tomto případě je `col` rovno `black`) neplatí podmínka 4 z definice red-black stromu pro předky odebraného vrcholu (například pro rodiče). Tuto situaci opravíme pomocí `rb-delete-fixup`.

Po odebrání červeného vrcholu všechny podmínky z definice red-black stromu platí.

Pomůcka pro pochopení `rb-delete-fixup`

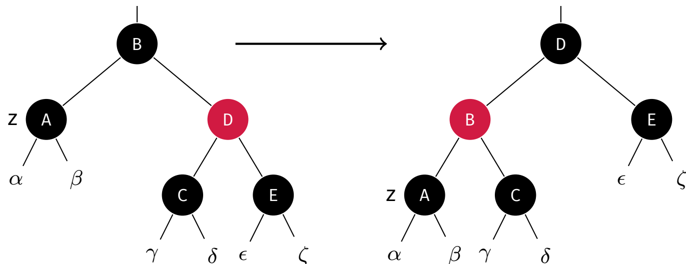
- černou barvu ze smazaného vrcholu si představíme jako token: poukázku na přebarvení na černo.
- tento token budeme přemisťovat mezi vrcholy s cílem se jej zbavit: buď se nám povede jej přemístit do červeného vrcholu, tam jej použijeme (vrchol přebarvíme na černo). Nebo dojdeme do kořene a tam jej zahodíme. Na začátku je token ve vrcholu získaném z procedury `tree-delete`.
- pokud při počítání počtu černých vrcholů na cestách do listů bereme token jako vrchol černé barvy, bude podmínka 4 pořád platit. Z toho plyne korektnost zbavení se tokenu (spolu s tím, že obarvením červeného vrcholu na černo nemůžeme porušit podmínku 3).
- při přemísťování tokenu budeme dělat ve stromu lokální úpravy (rotace a přebarvování vrcholů) tak, aby pro podstrom, jehož kořen vlastní token, platily podmínky 2, 3 a 4.

```
rb-delete-fixup
 ⟵ t: RB-tree
 ⟵ z: Node — aktuální vrchol (vlastnící token)
```

```
1  quit ← false
2  while (z ≠ t.root) and not quit
3     if (z.color = red)
4     z.color ← black
5     return
6     z, quit ← local-delete-fix(t,z) — procedura pro opravy, viz dalsi slajdy
```

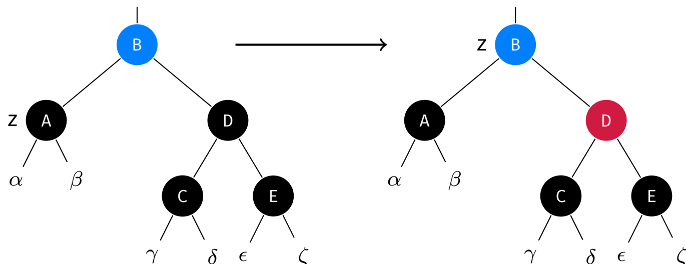
Procedura `local-delete-fix` detekuje jeden z možných případů. Poté provede úpravy a vrátí vrchol, kde mají úpravy pokračovat. Případy a jejich řešení si ukážeme na obrázcích. Pokud je na obrázku po úpravě označen nějaký vrchol `z` je tento vrchol vrácen spolu s hodnotou `false`, jinak je vráceno `true` (první návratová hodnota je libovolný vrchol).

Případy rozlišujeme podle barvy sourozence (a jeho potomků). Je-li sourozenec červený, rotací přejdeme do situace, kdy je sourozenec černý.



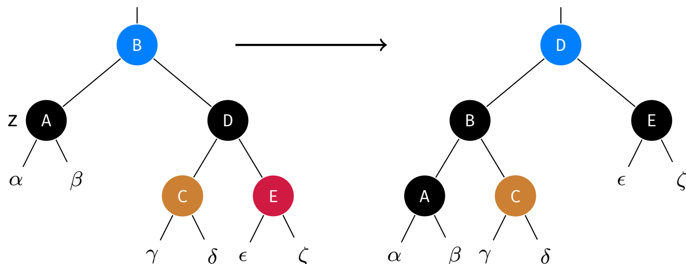
Korektnost: V prvním stromu musí být vrchol `B` černý, protože `D` je červený (podmínka 3). Délky černých cest (počítáme-li token jako černý vrchol) z vrcholu `B` v prvním stromu a vrcholu `D` v druhém stromu jsou totožné.

Sourozenec je černý. Rodič můžeme mít libovolnou barvu (na obrázku je zachycena modře), oba potomci sourozence jsou černí. Vyřešíme přebarvením a posunem tokenu do rodiče.



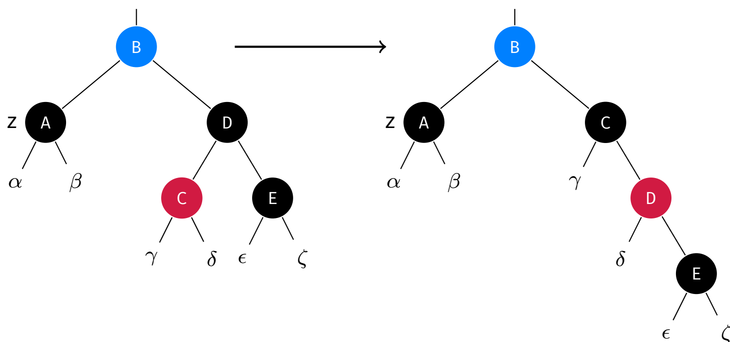
Délky černých cest z vrcholu B v prvním stromu a ve druhém stromu jsou totožné.

Sourozenec je černý, jeho pravý potomek je červený. Rodič a levý potomek sourozence mají libovolné barvy.



Korektnost: Délky černých cest (počítáme-li token) z vrcholu `B` v prvním stromu a vrcholu `D` v druhém stromu jsou totožné.

Sourozenec je černý, jeho pravý potomek černý, levý potomek je červený). Rotací převedeme na případ, kdy je pravý potomek sourozence červený.



Korektnost: Délky černých cest (počítáme-li token) z vrcholu B v prvním stromu a ve druhém stromu jsou totožné.

Složitost: Procedura `local-delete-fix` pracuje v konstantním čase.

Procedura `rb-delete-fixup` buď posloupností nejvýše tří iterací (první, třetí a čtvrtý případ pro `local-delete-fix`) skončí, nebo se aktuální vrchol posune po nejvýše dvou úpravách (první a druhý případ pro `local-delete-fix`) blíže ke kořeni. Celkem je tedy složitost `rb-delete-fixup` lineární vzhledem k výšce stromu.

Díky větě o výšce red-black stromu pak máme, že složitost odebírání vrcholu je $\Theta(\lg n)$.