ALGO2 / Stromy I

Petr Osička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Strom jako graf

Graf je dán dvojicí (V, H), kde V je konečná množina *vrcholů* a H je množina *hran*.

U neorientovaného grafuje je hrana neuspořádaná dvojice vrcholů. Formálně je to tedy dvouprvková množina vrcholů, přesto budeme značit (u, v).

Cesta z s do t: posloupnost vzájemně různých vrcholů $s = v_1, v_2, \ldots, v_n = t$, taková, že pro $1 \leqslant i < n$ je (v_i, v_{i+1}) hrana (řekneme, že na cestě leží). Délka cesty je počet hran, které na ní leží.

V grafu existuje *kružnice* pokud existují různé vrcholy s, t tak, že z s do t vede cesta, a v grafu je hrana (t, s), která na této cestě neleží. (Cesta doplněná o zmíněnou hranu je potom ona kružnice)

Graf je *souvislý*, pokud mezi libovolnou dvojicí různých vrcholů existuje cesta.

Strom je graf, který je souvislý a neobsahuje kružnici.

Věta

Pro neorientovaný graf $G = (V, H)$ *jsou následující ekvivalentní*

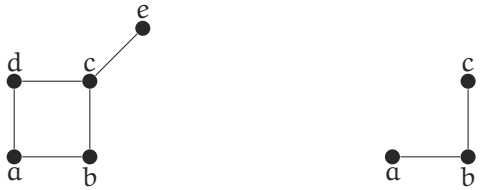
a. G *je strom.*
b. *Mezi každými dvěma různými vrcholy* G *existuje právě jedna cesta.*
c. G *neobsahuje kružnici a* $|V| = |E| + 1$.
d. G *je souvislý a* $|V| = |E| + 1$.
e. G *je souvislý, vynecháním jakékoliv hrany vznikne nesouvislý graf.*
f. G *neobsahuje kružnice, ale přidáním jakékoliv hrany vznikne graf s kružnicí.*

Podgrafy

Podgrafem grafu $G = (V, H)$ indukovaném množinou vrcholů V' je graf $G' = (V', H')$, kde

$$H' = \{(x, y) \in H \mid x, y \in V'\}$$

Př.



Kořenový strom

- Jeden vrchol je označen jako *kořen*.
- Získáme směr: uvažujeme cesty z kořene do ostatních vrcholů, směr dolů (do hloubky) je ve směru těchto cest.
- Vrchol x je *následníkem* vrcholu $y \neq x$, pokud y leží na cestě z kořene do vrcholu x. O vrcholu y říkáme, že je *předchůdcem* (či *předkem*) vrcholu x.
- Vrchol x je *potomkem* vrcholu y, pokud je x sousedem y a současně i jeho následníkem. O vrcholu y říkáme, že je *rodičem* vrcholu x.
- Vrcholy x a y jsou *sourozenci*, pokud mají stejného rodiče.
- Vrcholy x je *list*, pokud nemá žádného potomka.

Odteď pojmem strom myslíme kořenový strom.

Pro vrchol x stromu T definujeme *podstrom s kořenem* x jako podgraf stromu T indukovaný množinou vrcholů $\{x\} \cup \{y \mid y \text{ je následník } x\}$.

Jako *podstrom vrcholu* x ve stromu T označíme libovolný podstrom, jehož kořenem je potomek vrcholu x.

Hloubka nekořenového vrcholu x ve stromu T je délka cesty z kořene do x. (Hloubka kořene je tak 0). *Výška* stromu T je maximum z hloubek vrcholů v T.

Věta

Pro libovolný vrchol x ve stromu platí:

1. *Počet předchůdců x je roven jeho hloubce.*
2. *Různé podstromy x obsahují disjuktní množiny vrcholů.*
3. *Výška podstromu s kořenem x je rovna maximu výšek podstromů vrcholu x, ke kterému přičteme jedna.*

Kořenový strom je m-ární, pokud každý jeho vrchol má nejvýše m potomků. 2-ární strom nazýváme binární.

Věta

V binárním stromu výšky h obsahujícím n vrcholů platí

$$h + 1 \leqslant n \leqslant 2^{h+1} - 1, \tag{1}$$

$$\lfloor \ln(n) \rfloor \leqslant h \leqslant n - 1. \tag{2}$$

Strom v pointer machine

`struct` pro vrchol. Potomci přímo v položkách, v poli, nebo v seznamu. Strom si pamatujeme pomocí kořene.

Potomci v poli

```
struct Node
  id: Key
  children : [m]Node
  n: Int
```

Potomci v seznamu

```
struct Node
  id : Key
  child : Node
  sibling : Node
```

Binární vyhledávací stromy

Binární strom s klíči ve vrcholech, ve kterém korektně funguje následující algoritmus vyhledávání

Struktura pro vrchol

```
struct Node
    id : Key — klíč
    left : Node — levý potomek (nil, pokud neexistuje)
    right : Node — pravý potomek (nil, pokud neexistuje)
    parent : Node — rodič (nil, pokud neexistuje)
```

`tree-search-rec`
⟵ `k: Key` — *hledaný klíč*
⟵ `x: Node` — *kořen prohledávaného podstromu*
⟶ `Node` — *vrchol obsahující nalezený klíč nebo nil*

1
$$\text{return} \begin{cases} \texttt{x} & \text{pokud } (\texttt{x == nil || x.id == k}) \\ \texttt{tree-search-rec(k, x.left)} & \text{pokud } (\texttt{x.id > k}) \\ \texttt{tree-search-rec(k, x.right)} & \text{pokud } (\texttt{x.id < k}) \end{cases}$$

Korektnost znamená: pro libovolný klíč `k` v případě, že se vrchol s klíčem `k` ve stromu nachází, vrátí procedura právě tento vrchol, a v opačném případě vrátí `nil`.

Ekvivalentní podmínkou je *podmínka uspořádání:* Pro každou dvojici různých vrcholů `x, y` máme, že

- pokud je vrchol `y` v levém podstromu `x`, pak `y.id < x.id`,
- pokud je `y` v pravém podstromu `x`, pak `y.id > x.id`.

(*Doporučuji čtenáři si zdůvodnit, proč jsou podmínky ekvivalentní*)

Iterativní verze vyhledávání

```
tree-search
⟵ k: Key  – hledaný klíč
⟵ x: Node  – kořen prohledávaného podstromu
⟶ Node  – vrchol obsahující nalezený klíč
```

```
1  y ← x
2  while (y ≠ nil)
3     if (k = y.id) then break
4     y ← { y.left   pokud (k < y.id)
          { y.right  pokud (k > y.id)
5  return y
```

Složitost vyhledávání

- Po každé iteraci (v každém novém rekurzivním zavolání) se nacházíme ve vrcholu, který má o jedna větší hloubku
- V nejhorším případě projdeme cestu od kořene do listu. Provedeme maximálně $h + 1$ iterací cyklu (nebo rekurzivních zavolání), kde h je výška stromu. V každé iteraci cyklu (nebo rekurzivním zavolání), provedeme $\Theta(1)$ porovnání, celkově je složitost $\Theta(h)$.
- V nejlepším případě je hledaný klíč nalezen v kořeni. Složitost $\Theta(1)$.
- Nejhorší a nejlepší rozeznáváme i vzhledem ke tvaru stromu (ne pouze vzhledem k vyhledávanému klíči ve fixním stromě).

strom / klíč	nejlepší	nejhorší
nejlepší	$\Theta(1)$	$\Theta(\lg n)$
nejhorší	$\Theta(1)$	$\Theta(n)$

- Analogické rozlišení případů uvidíme i u dalších operací, jejichž složitost záleží na výšce stromu.

Průchody vyhledávacím stromem

```
in-order-walk
⟵ x: Node
```

```
1 if (x == nil) then return
2 in-order-walk(x.left)
3 navštiv x
4 in-order-walk(x.right)
```

Věta

Procedura `in-order-walk` *navštíví vrcholy ve vzestupném pořadí (podle hodnot klíčů). (Angl. symmetric order.) Složitost procedury je lineární.*

Analogicky `post-order-walk`, `pre-order-walk`, jména podle pořadí rekurzivních volání a navštívení vrcholu.

Limit rychlosti konstrukce vyhledávacího stromu

Věta

Pokud máme n různých klíčů a chceme vytvořit binární vyhledávací strom obsahující právě tyto klíče, pak při konstrukci stromu použijeme operaci porovnání klíčů $\Omega(n \lg n)$ krát.

Důkaz: Sporem.
Předp. že to jde v čase $o(n \lg n)$. Potom

```
1  vytvoř ze vstupních klíčů binární vyhledávací strom T
2  projdi T pomocí in-order-walk (při návštěvě vrcholy postupně vkládej do pole).
```

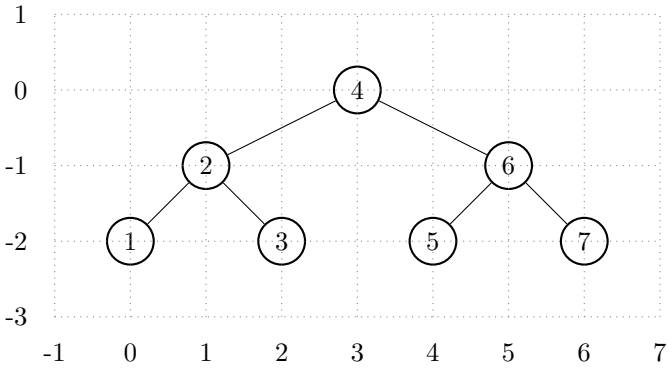
je třídicí algoritmus provádějící $o(n \lg n)$ porovnání. To je ovšem spor, porovnání potřebujeme nejméně $\Omega(n \lg n)$ (to je tvrzení známé z prvního semestru). □

Jednoduché kreslení vyhledávacího stromu

Přidáme do struktury pro vrchol položky `x`, `y` pro souřadnice. Jedním průchodem spočítáme hodnoty souřadnic. Dalším průchodem potom můžeme strom nakreslit.

`node-coordinate`
⟵ `r: Node` — *kořen stromu (předp.* `r` $\neq$ `nil`*)*
⟵ `y: Int` — *vertikální souřadnice vrcholu*
⟵ `x: Int` — *horizontální souřadnice dosud nejpravějšího vrcholu*
⟶ `Int` — *horizontální souřadnice nejpravějšího vrcholu v podstromu s kořenem* `r`

```
1  r.y ← y
2  ret ← x
3  if (r.left ≠ nil) then ret ← node-coordinates(r.left, y-1, x)
4  ret ← ret + 1
5  r.x ← ret
6  if (r.right ≠ nil) then ret ← node-coordinates(r.right, y-1, ret)
6  return ret
```


1
0
-1
-2
-3
-1
0
1
2
3
4
5
6
7
4
2
6
1
3
5
7

Minimum, maximum

```
tree-min
⟵ r: Node — kořen stromu (předp. r ≠ nil)
⟶ Node — vrchol s nejmenším klíčem
```

```
1  y ← r
2  while (y.left ≠ nil) do y ← y.left
3  return y
```

- `y ← y.left` je korektní, protože všechny vrcholy ve stromu `y.right` mají větší klíče než `y`. Tj. vrchol s minimálním klíčem nemůže být v `y.right`.
- Odtud máme, že vrchol s minimálním klíčem má levého potomka `nil`.
- Složitost operace je lineární vzhledem k výšce stromu.
- Maximum je analogická operace.

Pořádkový následník

Pořádkový následník vrcholu x je vrchol, který má v množině

$$\{\texttt{z} \mid \texttt{z.id} > \texttt{x.id}\}$$

nejmenší klíč. Pokud je tato množina prázdná, pořádkový následník neexistuje.

Věta (Kde je pořádkový následník?)

a. *Pokud má* x *neprázdný pravý podstrom, je jeho pořádkový následník minimálním prvkem tohoto podstromu.*

b. *Pokud má* x *prázdný pravý podstrom a jeho pořádkový následník* y *existuje, pak je nejhlubším předkem vrcholu* x *takovým, že* `y.left` *je buď také předkem* x *nebo přímo je přímo* x.

Důkaz.

Předpokládejme, že existuje vrchol y, který je pořádkovým následníkem x. Pokud se y nenachází v pravém podstromu x, pak existuje předek p vrcholu x tak, že p je přímo y, nebo je y v pravém podstromu p. (Z podmínky uspořádání plyne, že x je v levém podstromu p.) Přitom musí platit první možnost: pokud by y byl v pravém podstromu p, pak bychom měli `x.id < p.id < y.id`, a y by nebyl pořádkovým následníkem x.

(a) Pokud je pravý podstrom x neprázdný, jsou vrcholy pravého podstromu x v levém podstromu y. Pro každý vrchol w v pravém podstromu x tak máme `x.id < w.id < y.id`. To je spor s tím, že y je pořádkový následník x.

(b)Předpokládejme, že existují dva různí předci r a s vrcholu x tak, že `r.left` i `s.left` jsou předci x nebo (jeden z nich je) přímo x. Pokud je vrchol r je ve větší hloubce než s, pak je r v levém podstromu vrcholu s, a tedy `x.id < r.id < s.id` a vrchol s není pořádkovým následníkem x. Odtud plyne maximální hloubka. □

`tree-successor`
⟵ `r: Node` — *vrchol, jehož pořádkového následníka hledáme*
⟶ `Node` — *pořádkový následník nebo* `nil`

```
1  if (r.right ≠ nil) then return tree-min(r.right)
2  x ← r
3  while (x.parent ≠ nil)
4     y ← x.parent
5     if (x = y.left) then return y
6     x ← y
7  return nil
```

Složitost je lineární vzhledem k výšce stromu, protože strom projdeme nejhůře od `r` do listu, nebo od `r` do kořene. V obou případech je délka cesty omezena výškou stromu.

Duální pojem je pořádkový předchůdce, operace jeho nalezení a důkaz její správnosti jsou analogické.

Operace měnící strom

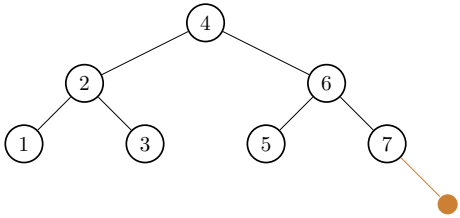
Operace může změnit kořen stromu.
Buď vracíme kořen stromu jako návratovou hodnotu, nebo vytvoříme analogii zarážky.

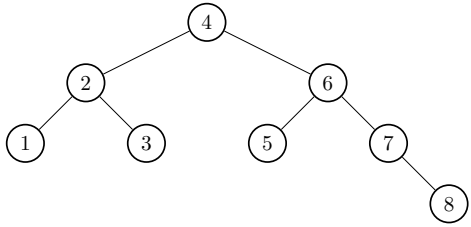
```
struct tree
   root:node — kořen stromu
```

Vkládání vrcholu

- Kam vkládaný vrchol umístit? Po vložení musí platit podmínka uspořádání.
- Vrchol umístíme na místo, na kterém (neúspěchem) skončí vyhledávání jeho klíče.
- Po přidání jej algoritmus vyhledávání úspěšně najde.

Příklad: přidání vrcholu u s klíčem 8.





Nastavování potomků

```
set-left-child
 ⟵ r: Node — rodič
 ⟵ c: Node — nový levý potomek
```

```
r.left ← c
if (c ≠ nil) then c.parent ← r
```

Analogicky `set-right-child`

```
tree-insert
⟵ t: Tree — strom, do kterého vkládáme
⟵ added: Node — vkládaný vrchol
```

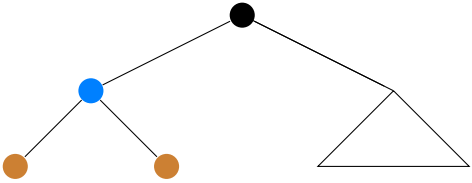
```
1  if (t.root == nil)
2     t.root ← added
3  else
4     y ← poslední navštívený vrchol při neúspěšném hledání klíče added.id
5     if (added.id < y.id)
6        set-left-child(y, added)
6     else
8        set-right-child(y, added)
```

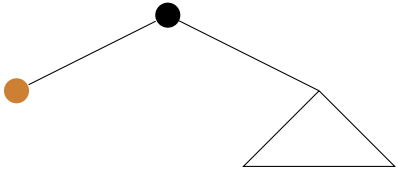
Složitost a korektnost: stejné jako u vyhledávání klíče.

Odebírání vrcholu

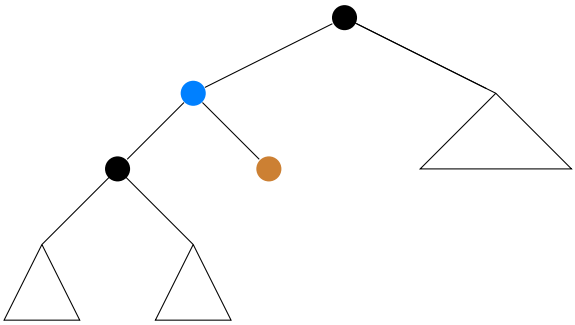
Tři případy, podle toho, kolik má odebíraný vrchol potomků. (odebíraný vrchol je modrý)

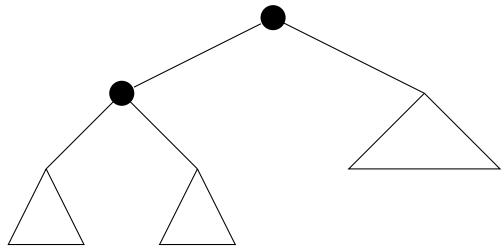
Případ 1: nemá žádné potomky



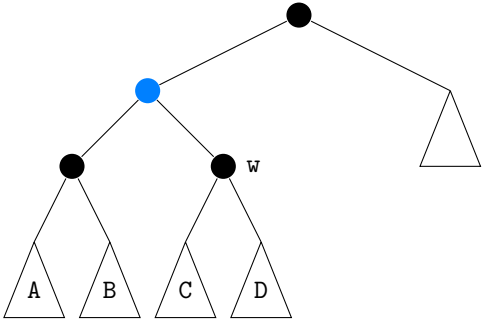


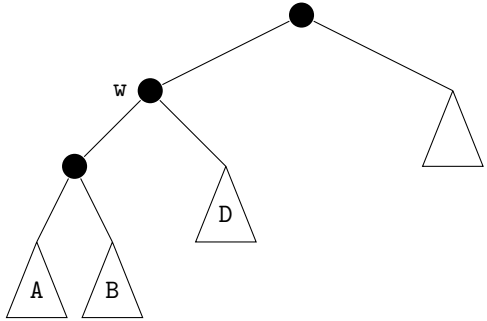
Případ 2: má jednoho potomka





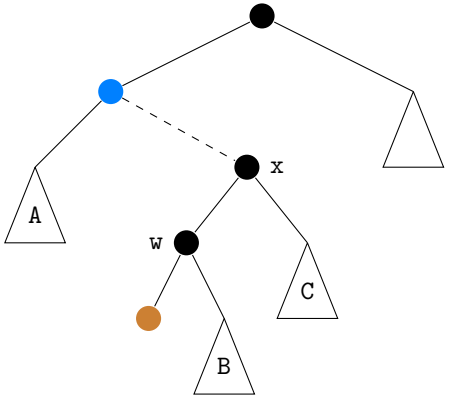
Případ 3: má dva potomky

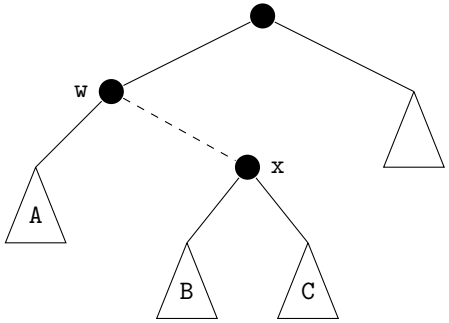




Nefunguje, pokud není strom c prázdný !!!

Potřebujeme mazaný vrchol nahradit vrcholem `w`, který je z jeho pravého podstromu. Vrchol `w` musí mít levý podstrom prázdný. Navíc musí být z pravého podstromu nejmenší, jinak pokazíme podmínku uspořádání.





Pomocné procedury

Nahrazení podstromu jiným podstromem

```
tree-swap
⟵ t: Tree
⟵ u: Node — kořen nahrazovaného podstromu
⟵ v: Node — kořen podstromu, kterým nahrazujeme
```

```
1  if (t.root = u) then t.root ← v
2  else
3     y ← u.parent
4     if (u = y.left) then set-left-child(y, v)
5     else set-right-child(y, v)
```

Pozor: procedura nezajišťuje dodržení podmínky uspořádání, tu musíme zajistit při jejím použití.

Nahrazení vrcholu jiným vrcholem

```
node-swap
⟵ t: Tree
⟵ u: Node — nahrazovaný vrchol
⟵ v: Node — vrchol, kterým nahrazujeme
```

```
1  set-left-child(v, u.left)
2  set-right-child(v, u.right)
3  if (t.root == u) then t.root ← v
4  else
     y ← u.parent
5     if (u == y.left) then set-left-child(y, v)
6     else set-right-child(y, v)
```

Pozor: procedura nezajišťuje dodržení podmínky uspořádání, tu musíme zajistit při jejím použití

Procedura odebírání

```
tree-delete
⟵ t: Tree
⟵ z: Node — odebíraný vrchol, předp. že je ve stromu
```

```
1  if (z.left == nil)
2     tree-swap(t, z, z.right)
3     return
5  if (z.right == nil)
6     tree-swap(t, z, z.left)
7     return
8  y ← tree-min(z.right)
9  tree-delete(t, y)
10 node-swap(t, z, y)
```

Korektnost odebírání vrcholu

- řádky 1 až 7 řeší případy, kdy z nemá 2 potomky. Neporušíme podmínku uspořádání, protože nahrazujeme z jedním z jeho podstromů, jejichž klíče jsou ve stejném vztahu ke klíči vrcholu `z.parent` jako klíč samotného z.
- na řádku 8 víme, že z má dva potomky, a proto je y pořádkovým následníkem z.
- y je minimem v `z.right` a proto nemá levého potomka. Rekurzivní zavolání `tree-delete` na řádku 9 tak skončí na řádcích 1 až 7.
- nahrazením vrcholu z vrcholem y neporušíme podmínku uspořádání: y má větší klíč než vsechny vrcholy v podstromu s kořenem `z.left` a menší klíč než všechny vrcholy v podstromu s kořenem `z.right` (mimo sebe samotného). Současně byl y před řádkem 4 v podstromu s kořenem z, a proto je uspořádání `y.id` a `z.parent.id` stejné, jako uspořádání `z.id` a `z.parent.id`.

Složitost odebírání vrcholu

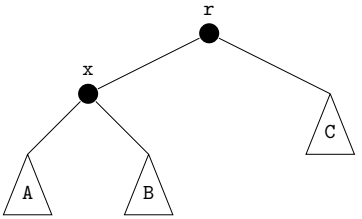
- `node-swap` a `tree-swap` pracují v konstantním čase.
- `tree-min` pracuje v nehorším případě (mažeme kořen, minimální prvek v pravém podstromu je v maximální hloubce) v čase lineárním vhledem k výšce stromu.
- Celkem je tedy složitost odebírání lineární vzhledem k výšce stromu.

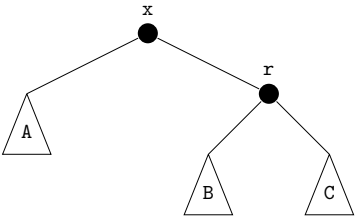
Rotace

- lokální operace prohození rodiče s potomkem
- musíme si dát pozor na správné přepojení podstromů, abychom zachovali podmínku uspořádání
- existují dvě symetrické rotace.
  - *levá*: prohazujeme rodiče s pravým potomkem
  - *pravá*: prohazujeme rodiče s levým potomkem
- procedury napíšeme tak, aby vraceli nový kořen uvažovaného podstromu. Při použití nesmíme zapomenout jej správně do uvažovaného stromu zapojit (tj. nový kořen nastavit jako správného potomka vrcholu, který byl rodičem kořene před rotací).
- Složitost obou rotací je konstantní.

`rotate-R`
⟵ `r: Node` — *kořen rotovaného stromu*

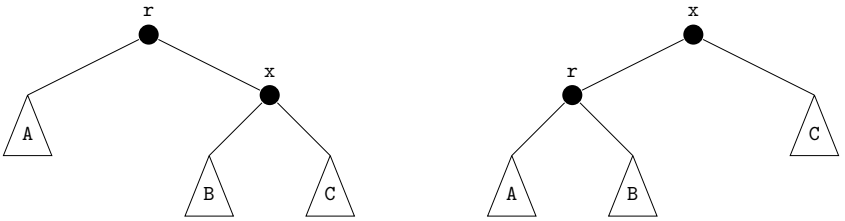
```
1  x ← r.left
2  B ← x.right
3  set-left-child(r, B)
4  set-right-child(x, r)
5  return x
```





```
rotate-L
⟵ r: Node — kořen rotovaného stromu
```

```
1  x ← r.right
2  B ← x.left
3  set-right-child(r, B)
4  set-left-child(x, r)
5  return x
```



Pamatování si velikosti stromu

- Do struktury `Node` přidáme položku `count: Int` pro počet vrcholů v podstromu, jehož kořenem je daný vrchol. Pro vrchol `x` dvěma potomky tak bude platit

  $$\texttt{x.count} = \texttt{x.left.count} + \texttt{x.right.count} + 1$$

  Pokud je `x.left` nebo `x.right` rovno `nil`, nahradíme v rovnosti příslušnou `count` položku nulou.
- položka `count` je užitečná pro operace, které potřebují znát velikosti podstromů. Ty bychom mohli počítat na místě. Rychlejší je ovšem tuto položku mít rovnou ve struktuře pro vrchol.
- Musíme upravit ovšem upravit operace měnící strukturu stromu tak, aby tuto položku udržovali konzistentní.

Úpravy procedur

- `tree-insert`: vloženému vrcholu nastavíme `count` na jedna. Všem vrcholům navštíveným při hledání místa pro vkládání zvedneme položku `count` o jedna. Složitost zůstává $O(h)$.
- `tree-delete`: před provedením `tree-swap` zmenšíme všem vrcholům na cestě od rodiče nahrazovaného vrcholu do kořene položku `count` o jedna. Navíc v proceduře `node-swap` zkopírujeme z nahrazovaného vrcholu do vrcholu, kterým nahrazujeme, hodnotu položky `count`. Oproti původní verzi potřebujeme jeden průchod do kořene navíc, ovšem složitost zůstává $O(h)$.
- `rotate-R`: Po provedení rotace upravíme počty následovně (značením odkazujeme do procedury na předchozích slajdech)

$$\texttt{transfer} \leftarrow \begin{cases} \texttt{0} & \text{pokud } \texttt{B} = \texttt{nil} \\ \texttt{B.count} & \text{jinak} \end{cases}$$

```
r.count ← r.count + transfer - x.count
x.count ← x.count - transfer + r.count
```

Složitost zůstává konstantní. Levá rotace je se upraví analogicky.

Pořádkové statistiky

k-tá pořádková statistika je vrchol, který obsahuje k-tý nejmenší klíč.

Naivně: `in-order-walk`, který *včas* stopneme.

Lepší nápad: z podmínky uspořádání máme pro strom s kořenem x

- pokud `x.left.count > k - 1`, pak k-tá pořádková statistika ve stromu s kořenem x je k-tou pořádkovou statistikou ve stromu s kořenem `x.left`.
- pokud `x.left.count == k - 1`, pak je k-tou pořádkovou statistikou vrchol x.
- pokud `x.left.count < k - 1`, pak k-tá pořádková statistika ve stromu s kořenem x je `(k - 1 - x.left.count)`-tou pořádkovou statistikou ve stromu s kořenem `x.right`.

```
select
⟵ r: Node — kořen stromu, ve kterém hledáme
⟵ k: Int — parametr statistiky
```

```
1  t ← { 0              pokud r.left == nil
       { r.left.count   jinak

2  return { select(r.left, k)           pokud t > k - 1
          { select(r.right, k - t - 1)  pokud t < k - 1
          { r                           jinak
```

Složitost $O(h)$.

Pořadí vrcholu

K vrcholu x nalezneme k tak, že x je k-tá pořádková statistika.

```
rank
⟵ r: Node — kořen stromu
⟵ x: Node — vrchol, pro nějž hledáme pořadí
⟶ Int
```

```
1  t ← 1 + { 0              pokud x.left = nil
           { x.left.count   jinak
2  y ← x
3  while (y ≠ r)
4     if (y = y.parent.right)
5        t ← t + 1 + { 0                      pokud y.parent.left = nil
                     { y.parent.left.count    jinak
6     y ← y.parent
7  vrať t
```

Kořenové verze operací

- významný vrchol je po provedení operace v kořeni
- `insert` - vkládaný vrchol
  `select`, `search` - hledaný vrchol
- algoritmická myšlenka: potřebný vrchol dostaneme do kořene pomocí posloupnosti rotací. Asymtotická složitost operací je potom často stejná jako u nekořenových operací (tj. lineárně závisí na výšce stromu).

```
root-insert
⟵ r: Node — kořen stromu, do kterého vkládáme
⟵ added: Node — vkládaný vrchol
⟶ Node — kořen stromu po vložení
```

```
1  if (r == nil) then return added
2  if (r.id > added.id)
3     set-left-child(r, root-insert(r.left, added))
4     return rotate-R(r)
5  else
6     set-right-child(r, root-insert(r.right, added))
7     return rotate-L(r)
```

Složitost. V nejhorším případě procházíme strom od kořene do listu. Složitost rotací a pomocných procedur je konstantní. Celkově je tedy složitost kořenového vkládání $O(h)$, kde h je výška stromu.

```
partition
⟵ r: Node — kořen stromu, ve kterém hledáme
⟵ k: Int — parametr statistiky
⟶ Node — kořen stromu po hledání
```

$$t \leftarrow \begin{cases} 0 & \text{pokud } \texttt{r.left} = \texttt{nil} \\ \texttt{r.left.count} & \text{jinak} \end{cases}$$

```
if (t = k - 1) then return r
if (t > k - 1)
   set-left-child(r, partition(r.left, k))
   return rotate-R(r)
else
   set-right-child(r, partition(r.right, k))
   return rotate-L(r)
```

Složitost je lineární vzhledem k výšce stromu.

Manuální vyvažování stromu

Několik metod

- Pomocí `in-order-walk` uložíme vrcholy do pole uspořádány podle klíčů. Potom pomocí `insert` vytvoříme strom minimální výšky.
- Využijeme `partition` a s jeho pomocí umístíme medián do kořene. Potom rekurzivně provedeme totéž pro levý a pravý podstrom.

Složitost

Složitost první metody je $\Theta(n)$ (rekurence $T(n) = 2T(n/2) + \Theta(\lg n)$, nebo dokonce $T(n) = 2T(n/2) + \Theta(1)$, pokud konstrujeme strom přímo a ne insertem). U druhé metody je složitost v nejhorším případě $O(n \lg n)$, pokud ji aplikujeme na strom, který je v podstatě seznamem (rekurence $T(n) = 2T(n/2) + \Theta(n)$). Při aplikaci na strom výšky $\Theta(\lg n)$ máme složitost $\Theta(n)$ (rekurence $T(n) = 2T(n/2) + \Theta(\lg n)$).

`balance`
⟵ `r: Node` — *kořen vyvažovaného stromu*
⟶ `Node` — *kořen vyváženého stromu*

```
1  if (r == nil or r.count ⩽ 2) then return r
2  ret ← partition(r, floor(r.count / 2) + 1)
3  set-left-child(ret, balance(ret.left)
4  set-right-child(ret, balance(ret.right)
5  return ret
```