

B-Stromy

Karel Richta a kol.

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

© Karel Richta a kol., 2018

Datové struktury a algoritmy, B6B36DSA
01/2018, Lekce 11

<https://cw.fel.cvut.cz/wiki/courses/b6b36dsa/start>

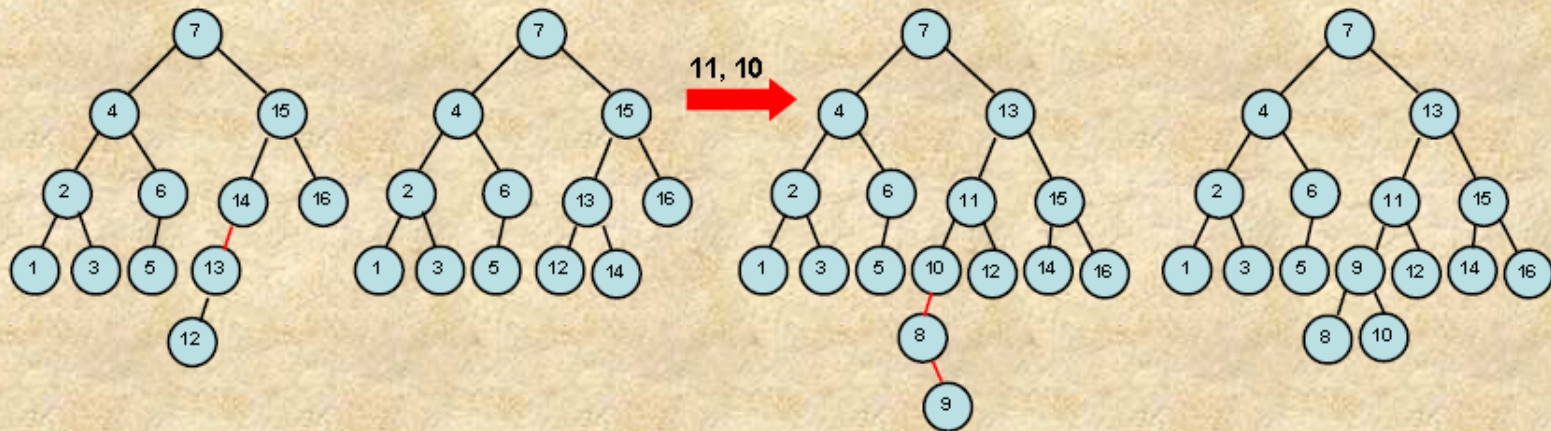


Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Opakování: skončili jsme AVL stromy

Příklad vytvoření AVL stromu

Vkládáme postupně **1, 2, 3, 4, 5, 6, 7, 16, 15, 14, 13, 12, 11, 10, 8, 9**



Shrnutí postupu: Po vložení uzlu najdeme nejbližšího předka x , kde došlo k rozvážení. Příčinou může být jedna z následujících 4 alternativ:

1. vložení do levého podstromu levého potomka uzlu $x \Rightarrow$ **rotace vpravo**
2. vložení do pravého podstromu levého potomka uzlu $x \Rightarrow$ **dvojitá LR rotace**
3. vložení do levého podstromu pravého potomka uzlu $x \Rightarrow$ **dvojitá RL rotace**
4. vložení do pravého podstromu pravého potomka uzlu $x \Rightarrow$ **rotace vlevo**

Obsah

- Červeno-černé stromy (Red-Black trees – RB)
 - Insert
 - Delete
- B-stromy (B-Trees)
 - Motivace
 - Search
 - Insert
 - Delete

Založeno na:

[Cormen, Leiserson, Rivest: Introduction to Algorithms, Chapter 14 and 19, McGraw Hill, 1990]

[Whitney: CS660 Combinatorial Algorithms, San Diego State University, 1996]

[Frederic Maire: An Introduction to Btrees, Queensland University of Technology, 1998]

<http://artax.karlin.mff.cuni.cz/~martin/ds/main.pdf>

Červeno-černé (RB) stromy

(s využitím http://en.literateprograms.org/Red-black_tree_%28Java%29)

RB stromy:

- zaručují $O(\lg N)$ složitost svých operací **v nejhorším případě**
- mají poněkud **náročnější implementaci** svých operací

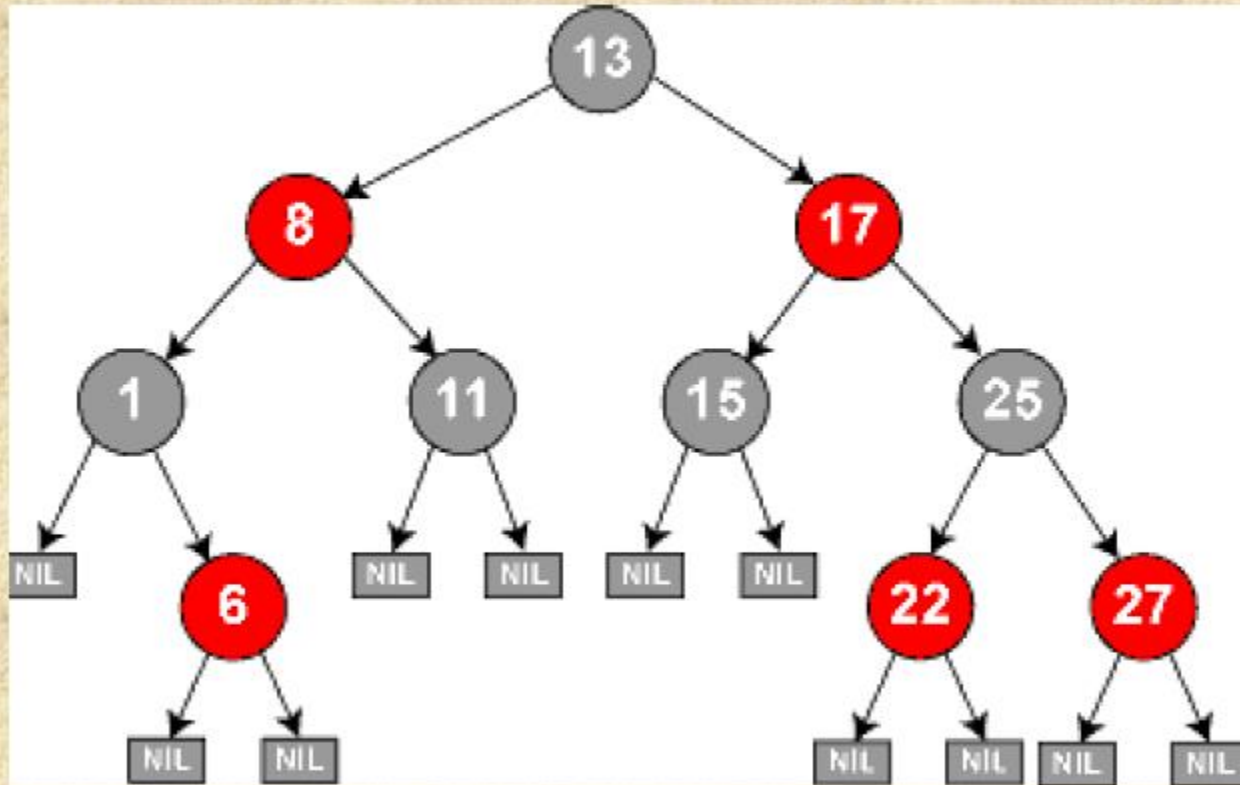
Definice RB stromu - RB strom je BVS s následujícími vlastnostmi:

1. každý uzel je obarven buď **červeně** nebo **černě**
2. kořen stromu je obarven **černě**
3. každý list (vnější uzel NIL) je **černý**
4. **červený uzel má oba své potomky černé**
5. **všechny listy (vnější uzly NIL) mají stejnou tzv. černou hloubku (tj. cesty k nim od kořene obsahují stejný počet černých uzlů)**

Předpokládané složky uzlu v RB stromu:

color, item (key, ...), left, right, parent

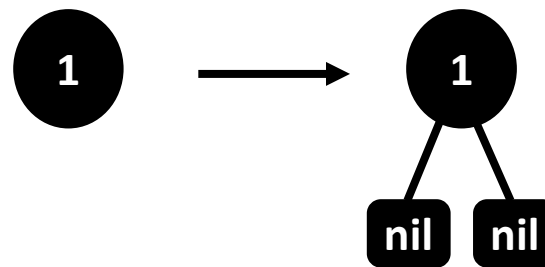
Příklad RB stromu



POZOR: NIL bude reprezentován jako společný "normální" uzel s černou barvou.

Vlastnosti červeno-černých stromů

- Jedná se o přibližně vyvážené BVS – platí:
 $h_{RB} \leq 2x h_{BVS}$ (výška $\leq 2x$ výška perfektně vyváženého stromu)
- Každý uzel obsahuje navíc bit pro barvu {red | black}
- Listy jsou doplněny odkazem na **nil** (neexistující potomek) = ukazatelem na **nil** - listy se stanou vnitřními uzly.

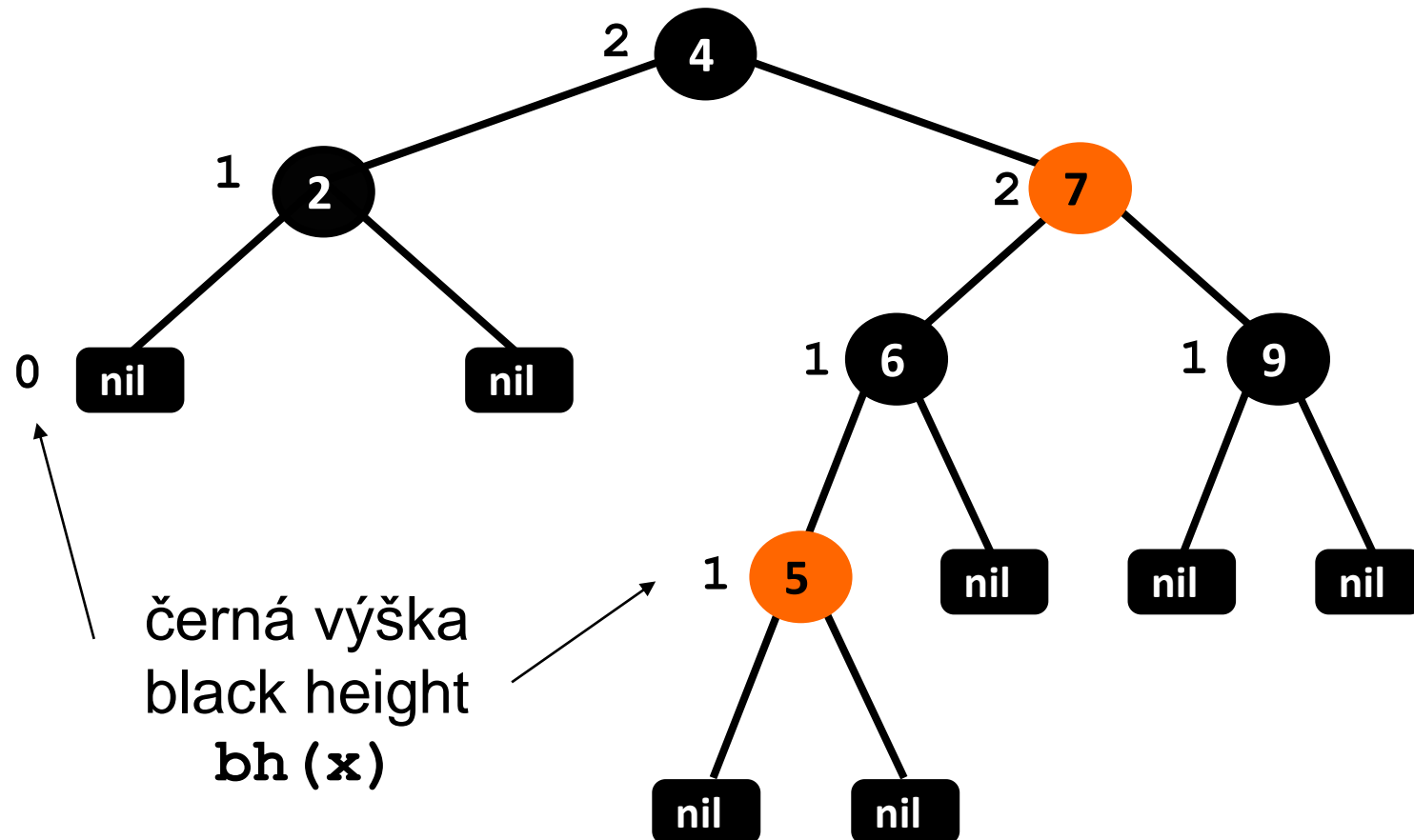


leaf \longrightarrow inner node

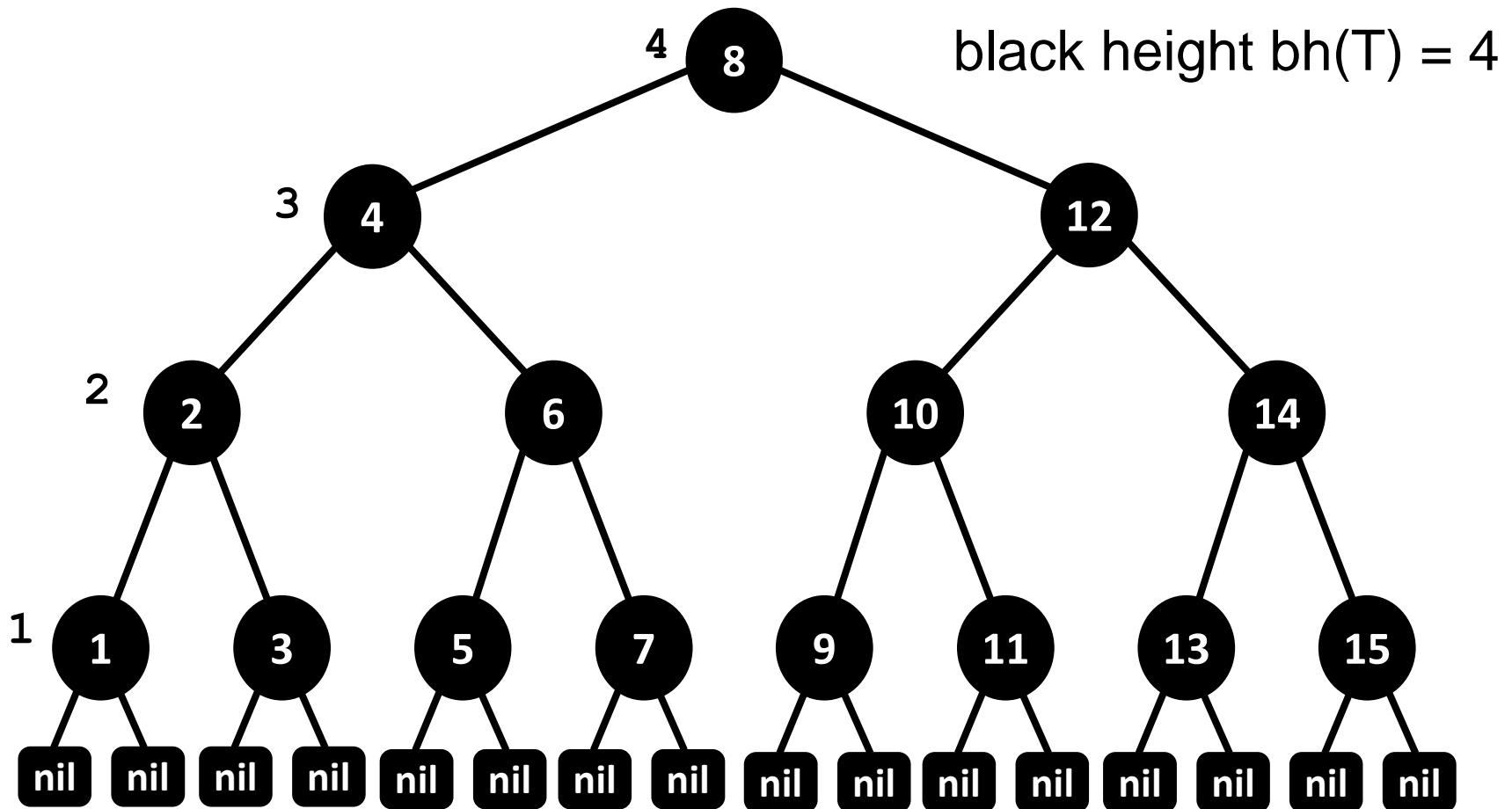
Černá výška v červeno-černých stromech

- Kořen RB-stromu je **černý**.
- Každá větev je zakončena (černým) uzlem **nil**.
- **Černá výška** uzlu **x** (**black-height**) se značí **$bh(x)$** a je definována jako počet černých uzlů na libovolné cestě z **x** do listu (**x** se nepočítá). Pozn.: Všechny cesty jsou stejně dlouhé. Totéž platí pro kořen.
- Černá výška má rozsah od **$h/2$** (kde **h** je výška stromu), jestliže polovina uzlů je červených do **h** , pokud jsou všechny uzly černé.
- **Výška $h(x)$** stromu s kořenem v **x** je maximálně dvakrát větší, než výška optimálně vyváženého stromu.
- **$h \leq 2\lg(n+1)$** **$h \in \Theta(\lg(n))$**
- **$bh(x)$** je v rozsahu od **$\lg(n+1)$** do **$2\lg(n+1)$**

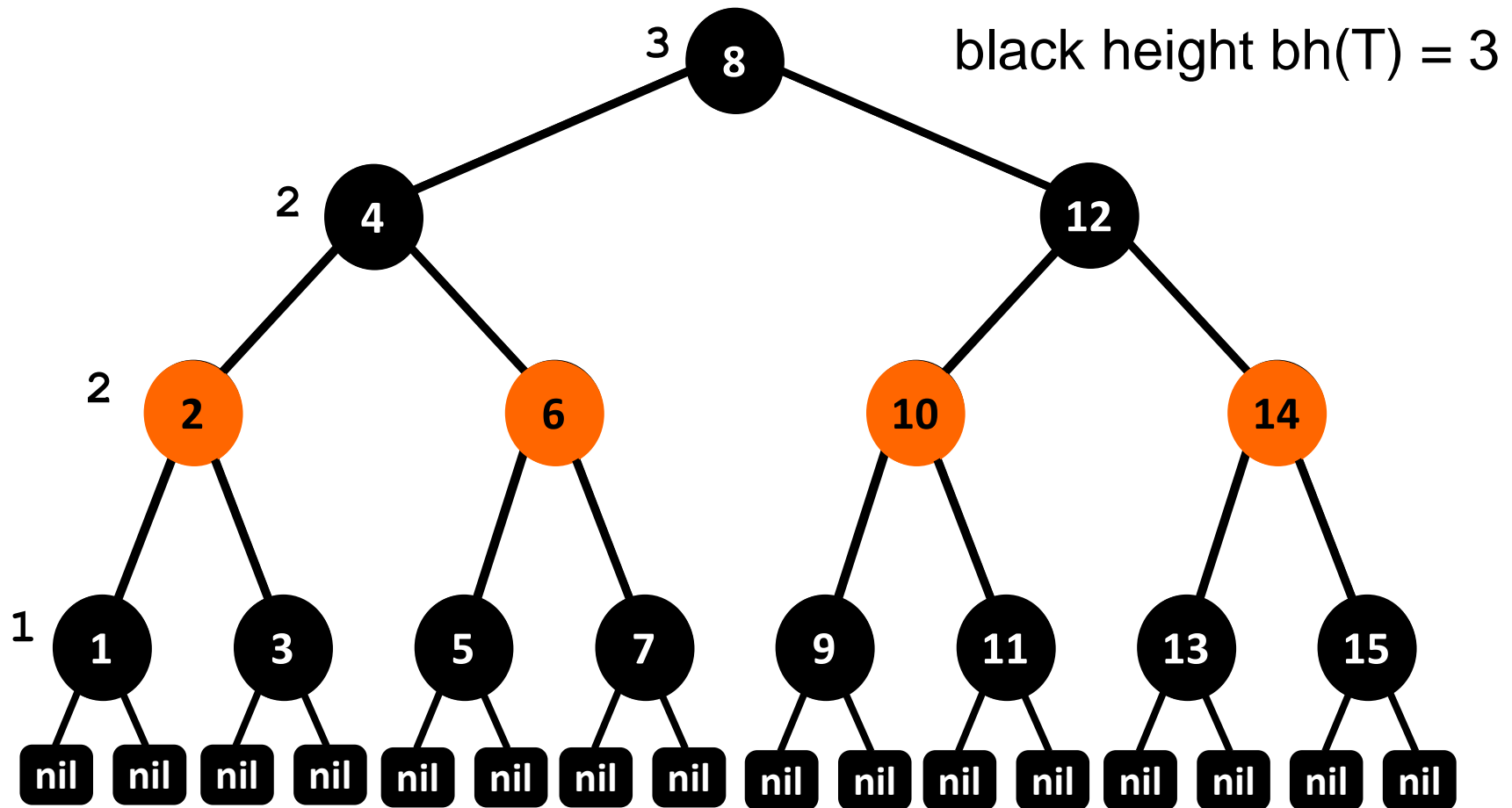
Příklad černé výšky



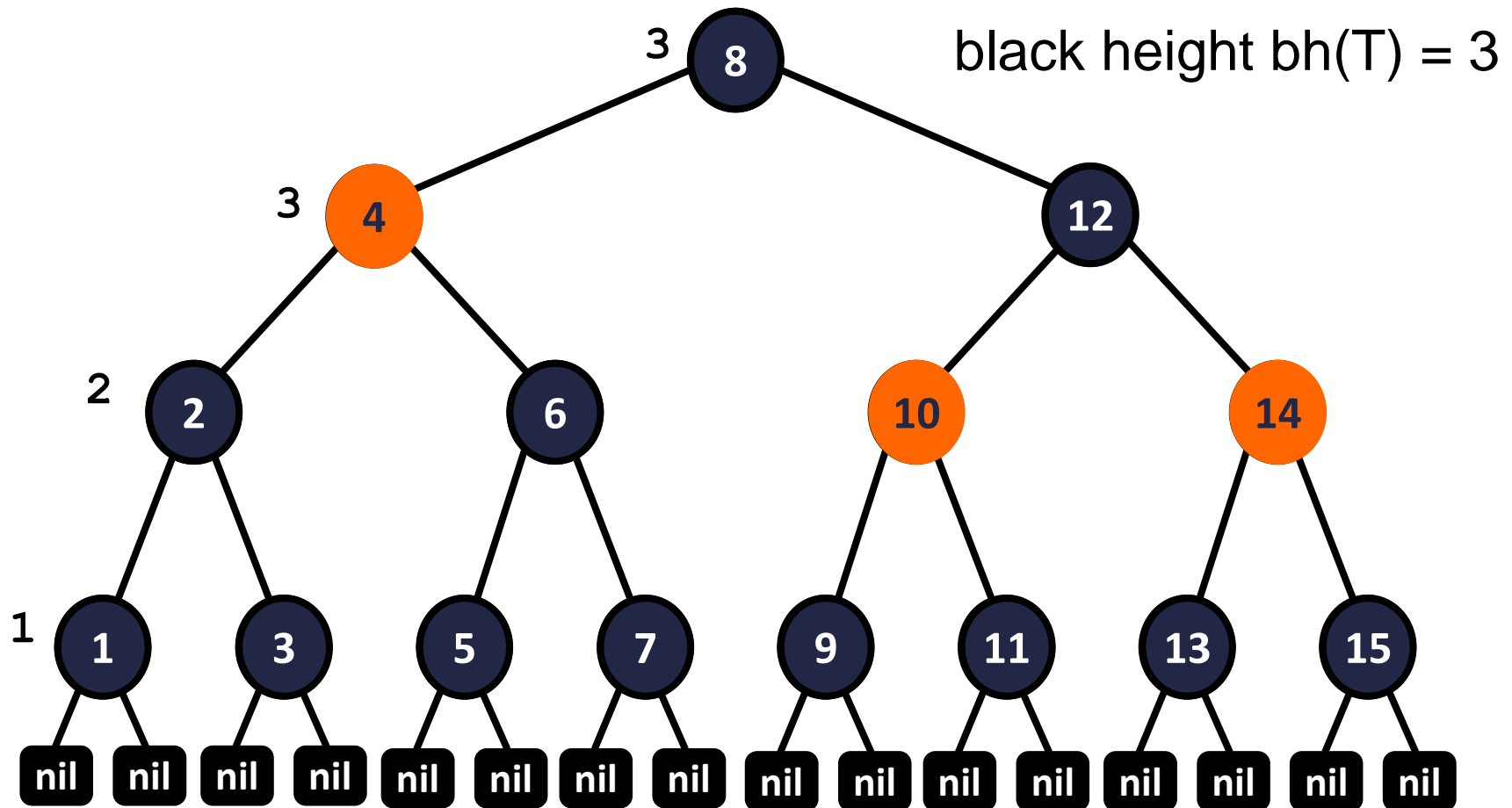
Transformace BVS na RB-strom



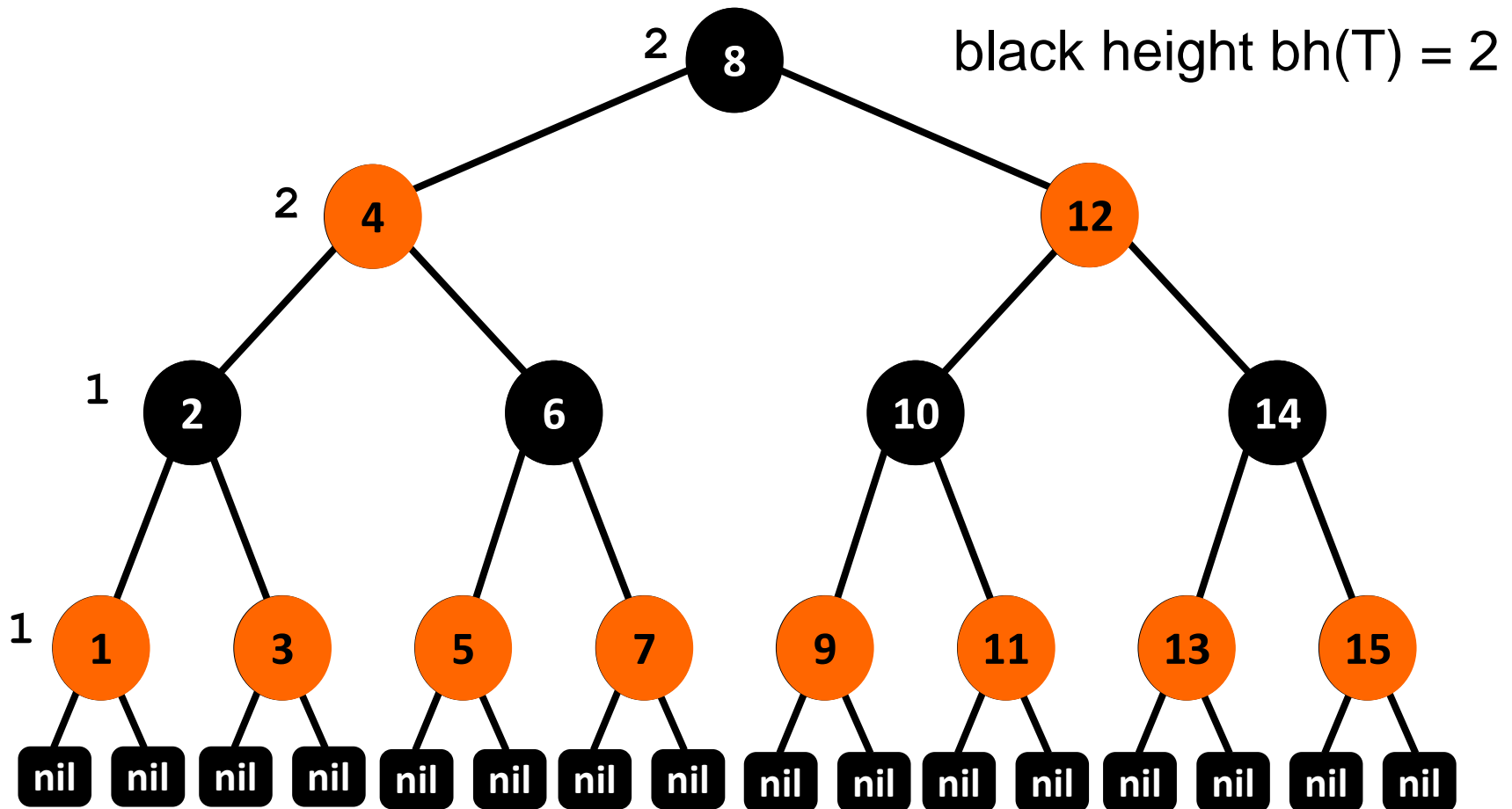
Transformace BVS na RB-strom



Transformace BVS na RB-strom

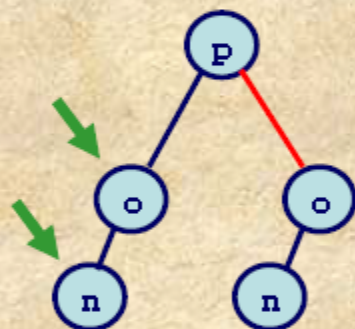


Transformace BVS na RB-strom



Rotace v RB stromech

Existence odkazu na rodiče trochu komplikuje implementaci rotací. Je třeba změnit odkaz v rodiči na uzel vytažený rotací nahoru.



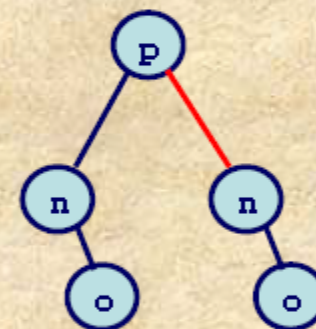
Platí:

$p = o.parent$, $p.left = o$
(nebo $p.right = o$)

Bude platit:

$p = n.parent$, $p.left = n$
(nebo $p.right = n$)

Zvláštní případy: $p=NIL$, $n=NIL$



```
private void replaceNode ( Node o, Node n )
{ Node p = o.parent;
  if ( p == NIL ) head = n;           // o can be the root
  else if ( o == p.left ) p.left = n;
  else p.right = n;
  if ( n != NIL ) n.parent = p;      // n can be external node NIL
```

Rotace v RB stromech

Změny odkazů zařídíme explicitně, operace tedy nebudou vracet hodnotu.

```
void rotateRight ( Node oo )
{
  Node nn = oo.left;
  replaceNode(oo, nn);
  oo.left = nn.right;
  if ( nn.right != NIL )
    nn.right.parent = oo;
  nn.right = oo;
  oo.parent = nn;
}
```

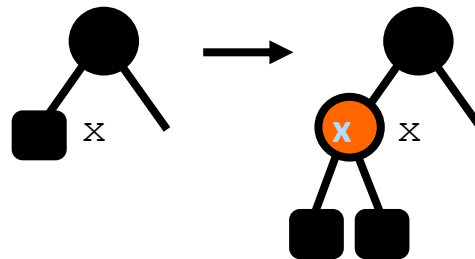
```
void rotateLeft ( Node oo )
{
  Node nn = oo.right;
  replaceNode(oo, nn);
  oo.right = nn.left;
  if ( nn.left != NIL )
    nn.left.parent = oo;
  nn.left = oo;
  oo.parent = nn;
}
```

Operaci `insert` navrhne standardním způsobem s tím, že:

- vložený uzel bude **červený**
- prověříme splnění požadavků RB stromu a provedeme náležité úpravy na cestě vzhůru

Operace INSERT pro RB-stromy

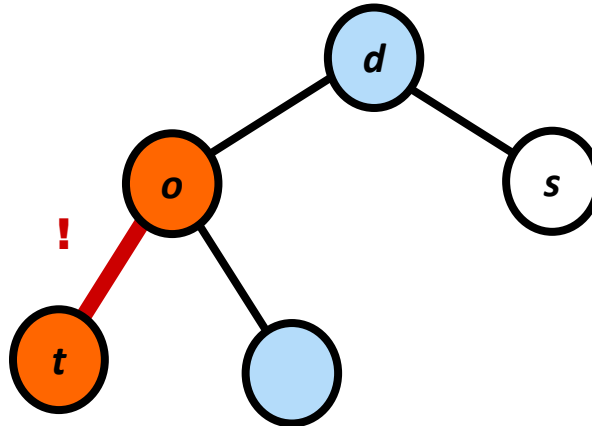
- 1) Nový uzel bude **červený**.
- 2) Vložíme uzel do stromu jako do standardního BVS.
- 3) Jestliže je předek černý, jsme hotovi → strom je **Červeno-Černý**.



- 4) Jestliže nikoliv, mohou nastat 3 následující případy.

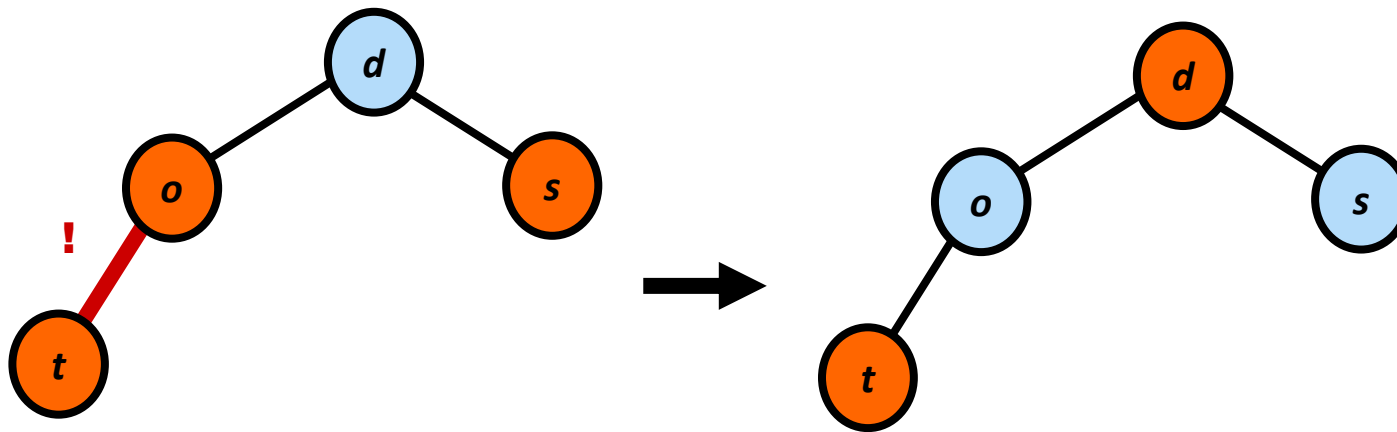
Poruchy při vkládání

- Pokud je vrchol t **červený** a jeho otec je také **červený**, pak řekneme, že vložením t nastala porucha.
- Pokud nastala porucha, pak ji musíme nějak opravit. Situace je na obrázku - nejprve záleží na tom, jakou barvu má s , strýc t :



Poruchy při vkládání (2)

1. s je **červený**. Pak pouze přebarvíme o , d a s podle obrázku. Nyní d může být porucha, ovšem posunutá o 2 hladiny výše.

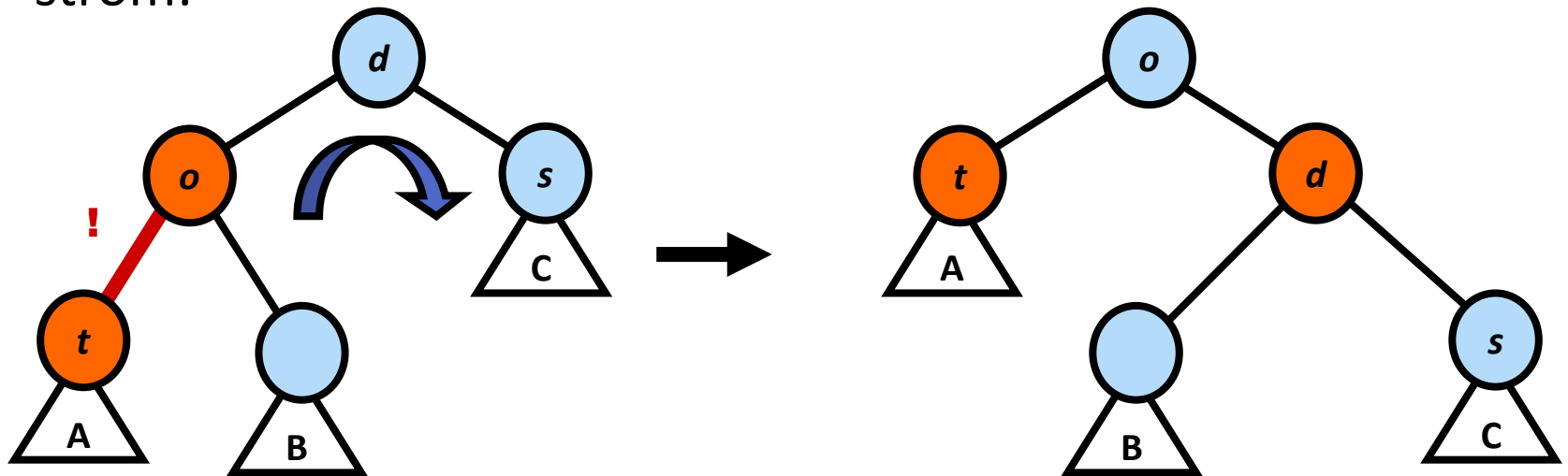


Pro splnění poslední podmínky je ještě nutné přebarvit kořen (d) na černo.

Poruchy při vkládání (3)

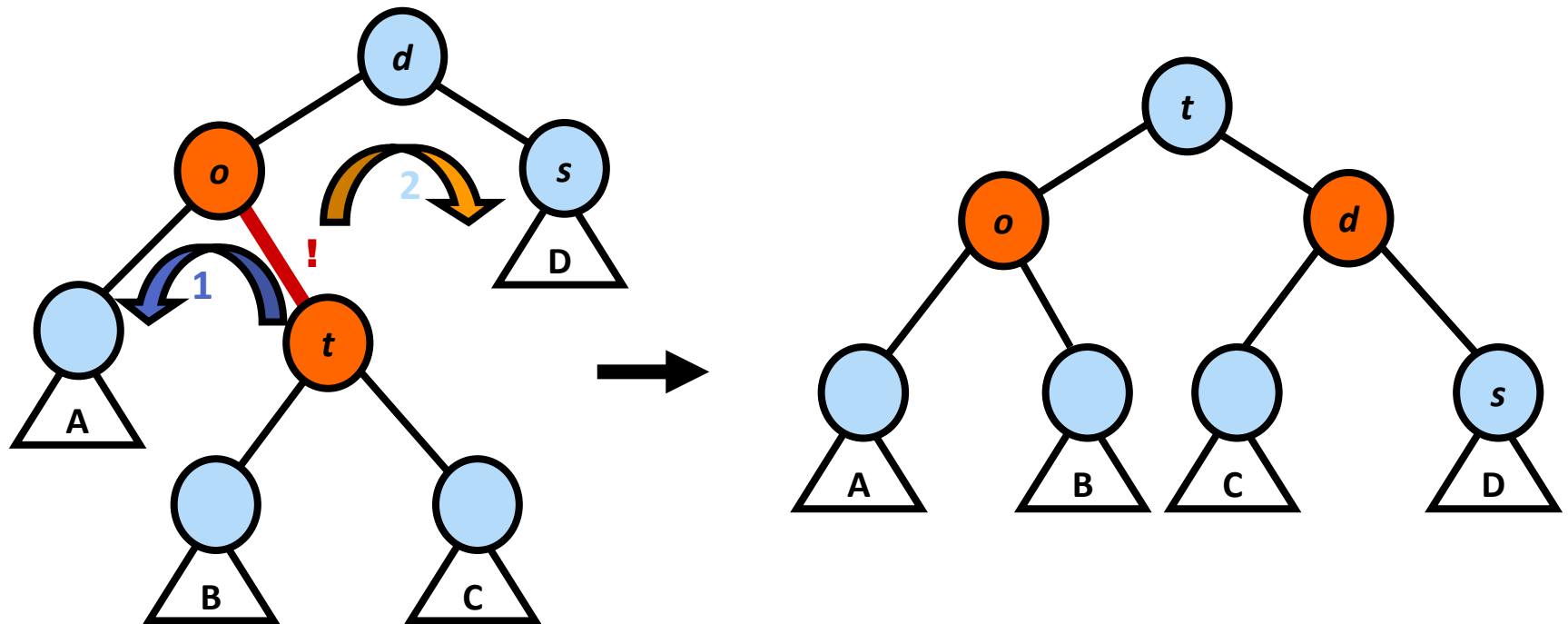
2. s je **černý**. Pak záleží na tom, zda hodnota t leží mezi hodnotami o a d nebo ne. Jinými slovy, zda cesta t - o - d obsahuje zatáčku.

(a) Bez zatáčky: Provedeme rotaci a přebarvíme podle obrázku. Splněny budou podmínky 1, 2 i 3, tedy máme **červeno**-černý strom:



Poruchy při vkládání (4)

(b) Se zatáčkou. Provedeme dvojitou (LR) rotaci a přebarvíme podle obrázku. Splněny budou podmínky 1, 2 i 3, opět máme rovnou **červeno-černý** strom.



RB stromy - insert (1)

```

void insertRB ( Node h, Elem x )
{ Node insertedNode = new Node(x);           // assumed RED, links to NIL
  if (h == NIL) head = insertedNode;         // first node
  else {
    while (true) {                           // start searching the place
      if ( x.key == h.item.key )
        { h.item = x; return;                // same key, change value
        }
      else if ( x.key < h.item.key ) { // left subtree
        if (h.left == NIL)
          { h.left = insertedNode; break; }
        else h = h.left;
      }
      else { // right subtree
        if (h.right == NIL)
          { h.right = insertedNode; break; }
        else h = h.right;
      }
    } // end of the last else
      // end of while loop
    insertedNode.parent = h;
  }
  checkCase1(insertedNode); // check and assure RB tree properties
}

```

RB stromy – insert (2)

Budou se nám hodit (pro přehlednost) následující pomocné funkce:

- `grandParent()` – určí prarodiče uzlu
- `sibling()` – určí sourozence uzlu
- `uncle()` – určí strýce, tj. pravého/levého sourozence rodiče uzlu

```
Node grandParent () // we assume parent != NIL and parent.parent != NIL
{ return parent.parent; }
```

```
Node sibling () // we assume parent != NIL, root has no sibling
{ if ( this == parent.left )
  return parent.right;
  else return parent.left;
}
```

```
Node uncle () // we assume parent != NIL and parent.parent != NIL
{ return parent.sibling(); }
```

RB stromy – insert (3)

- první test `checkCase1` zajistí, aby kořen byl vždy přebarven na černo
- druhý test `checkCase2` zkontroluje, zda je rodič vloženého uzlu černý

```
private void checkCase1 ( Node n )
{
    if (n.parent == NIL) n.color = BLACK; // the root was tested
    else checkCase2(n);
}

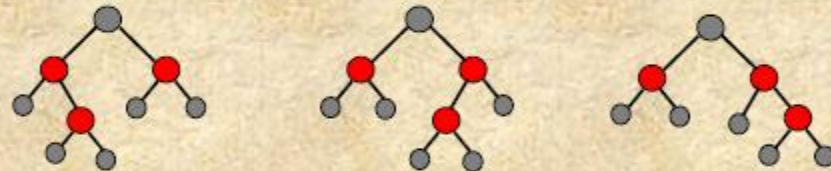
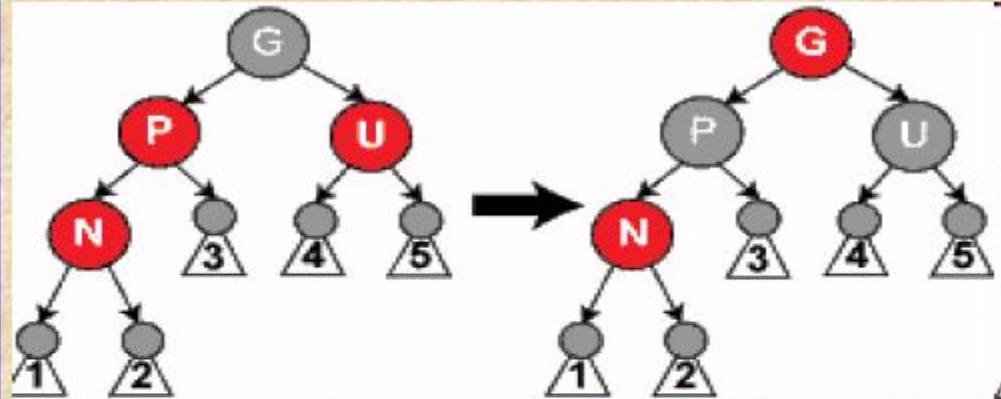
private void checkCase2 ( Node n )
{
    if (n.parent.color == BLACK) return; // tree is still valid
    else checkCase3(n);
}
```


RB stromy – insert (4)

Víme, že uzel n i jeho rodič mají červenou barvu

- je-li také jeho strýc červený, přebarvíme rodiče i strýce na černo, prarodiče na červeno a prověříme prarodiče
- jinak pokračujeme dalším testem

POZOR: strýc může být i nalevo od rodiče, uzel n může být levý nebo pravý potomek

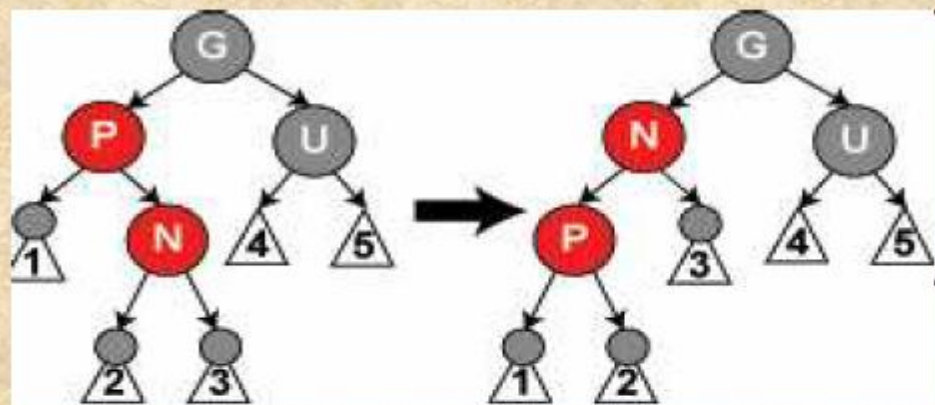


```
void checkCase3 ( Node n )
{ if ( n.uncle().color == RED ) {
  n.parent.color = BLACK; n.uncle().color = BLACK;
  n.grandparent().color = RED;
  checkCase1(n.grandparent());
} else checkCase4(n);
}
```

RB stromy – insert (5)

Víme, že strýc je černý.

- je-li n pravým synem svého rodiče, který je levým synem jeho prarodiče \Rightarrow provedeme **rotaci vlevo kolem rodiče** a pokračujeme testem na spodním uzlu
- je-li n levým synem svého rodiče, který je pravým synem jeho prarodiče \Rightarrow provedeme **rotaci vpravo kolem rodiče** a pokračujeme testem na spodním uzlu

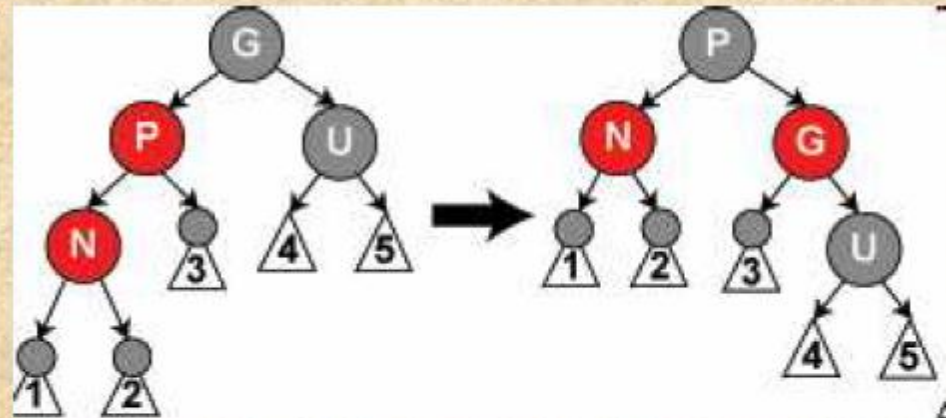


```
void checkCase4 ( Node n )
{ if ( n == n.parent.right && n.parent == n.grandparent().left)
  { rotateLeft(n.parent); n = n.left; }
  else if ( n == n.parent.left && n.parent == n.grandparent().right)
    { rotateRight(n.parent); n = n.right; }
  insertCase5(n);
}
```


RB stromy – insert (6)

Uzel n už je na "vnější" straně

- je-li n levým synem svého rodiče, který je levým synem jeho prarodiče ⇒ provedeme **rotaci vpravo kolem prarodiče**
- je-li n pravým synem svého rodiče, který je pravým synem jeho prarodiče ⇒ provedeme **rotaci vlevo kolem prarodiče**

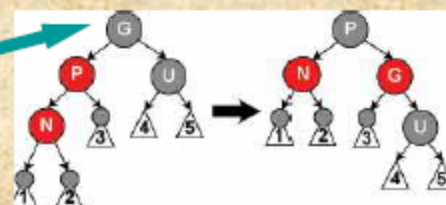
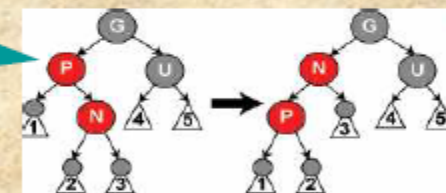
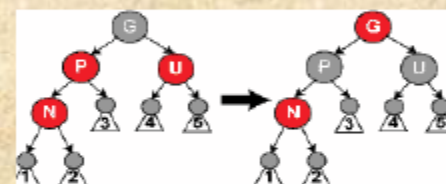


```
void checkCase5 ( Node n )
{
    n.parent.color = BLACK;
    n.grandparent().color = RED;
    if ( n == n.parent.left && n.parent == n.grandparent().left )
        rotateRight( n.grandparent() );
    else rotateLeft( n.grandparent() );
}
```

RB stromy – insert (7)

Shrnutí postupu při testování (a úpravě) okolí uzlu n:

- jedná-li se o kořen, obarvíme jej na černo a **je hotovo**
- jinak, je-li rodič uzlu n černý, **je hotovo**
- jinak (víme, že uzel n i jeho rodič mají červenou barvu), je-li také jeho strýc červený, přebarvíme rodiče i strýce na černo, prarodiče na červeno a **prověřujeme prarodiče**
- jinak, je-li n pravým synem svého rodiče, který je levým synem jeho prarodiče \Rightarrow provedeme **rotaci vlevo kolem rodiče** a pokračujeme testováním spodního uzlu
- jinak, je-li n levým synem svého rodiče, který je pravým synem jeho prarodiče \Rightarrow provedeme **rotaci vpravo kolem rodiče** a pokračujeme testováním spodního uzlu
- jinak, je-li n levým synem svého rodiče, který je levým synem jeho prarodiče \Rightarrow provedeme **rotaci vpravo kolem prarodiče**
- jinak, n je pravým synem svého rodiče, který je pravým synem jeho prarodiče \Rightarrow provedeme **rotaci vlevo kolem rodiče**



Vkládání (Insert) v RB-stromech

- Časová složitost je $O(\log(n))$.
- Vyžaduje maximálně dvě rotace.

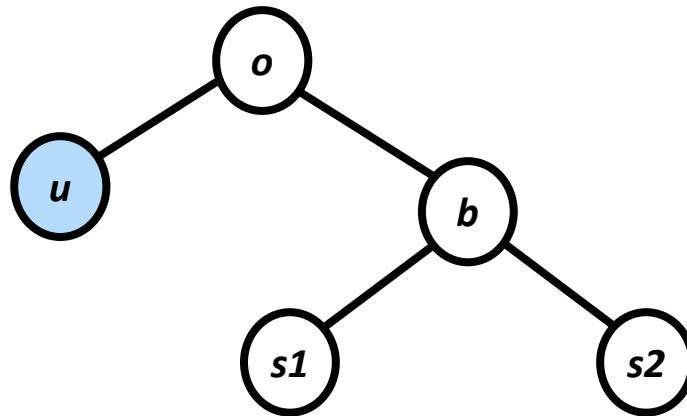
- DEMO:
- <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>
- (Intuitivní, dobré pro porozumění)
- <http://www.youtube.com/watch?v=vDHFF4wjWYU>

Operace DELETE pro RB-stromy

- Zatímco INSERT se příliš nelišil od své obdoby u AVL stromů, operace DELETE u **červeno**-černých stromů je oproti AVL stromům složitější mentálně, ovšem jednodušší časově.
- Situace: odstraňujeme vrchol t a jeho syna, který je list.
- Druhého syna t , u , dáme na místo smazaného t a začerníme ho. Tím máme splněny podmínky 1 a 2. Pokud byl ale t **černý**, chybí nám na cestách procházejících nyní u jeden černý vrchol.

Poruchy při mazání

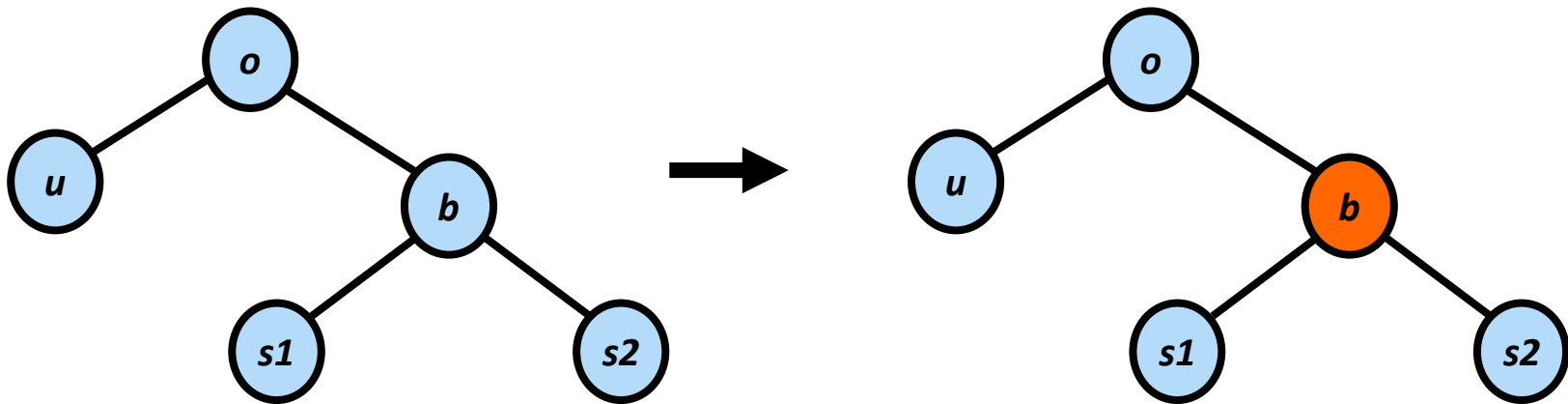
Situace: máme **červeno**-černý strom, u je porucha s otcem o , bratrem b a synovci $s1$, $s2$, viz obrázek.



Oprava záleží na **barvě bratra** (b):

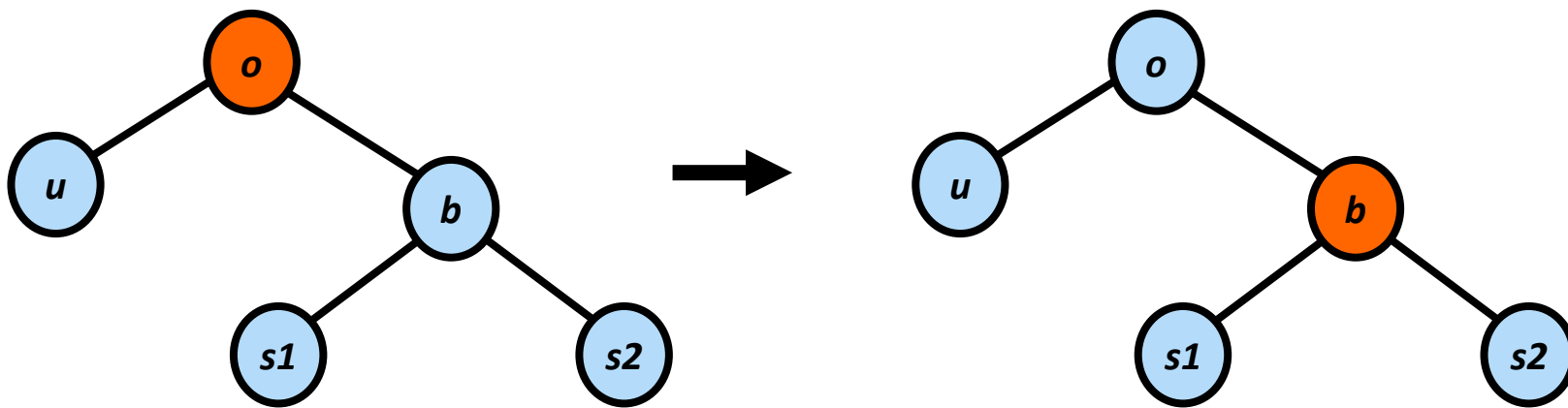
Poruchy při mazání (2)

1. Bratr je **černý**. Rozlišujeme dále 4 případy, z nichž jeden způsobí poruchu o hladinu výše a ostatní skončí s **červeno-černým** stromem.
 - (a) Otec i synovci jsou **černí**. Přebarvíme b na **červeno**, viz obrázek, a tady porucha je o hladinu výše – provedli jsme tedy jakousi částečnou opravu.



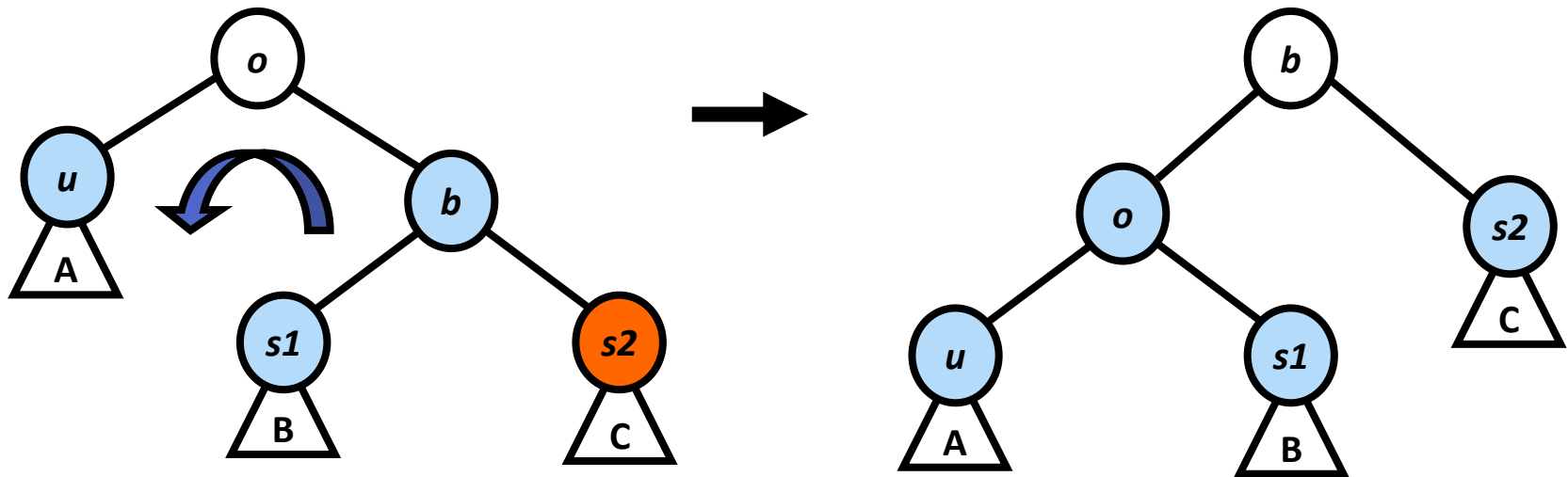
Poruchy při mazání (3)

(b) Otec je **červený**, synovci **černí**. Přebarvíme otce a bratra podle obrázku a dostáváme **červeno-černý** strom.



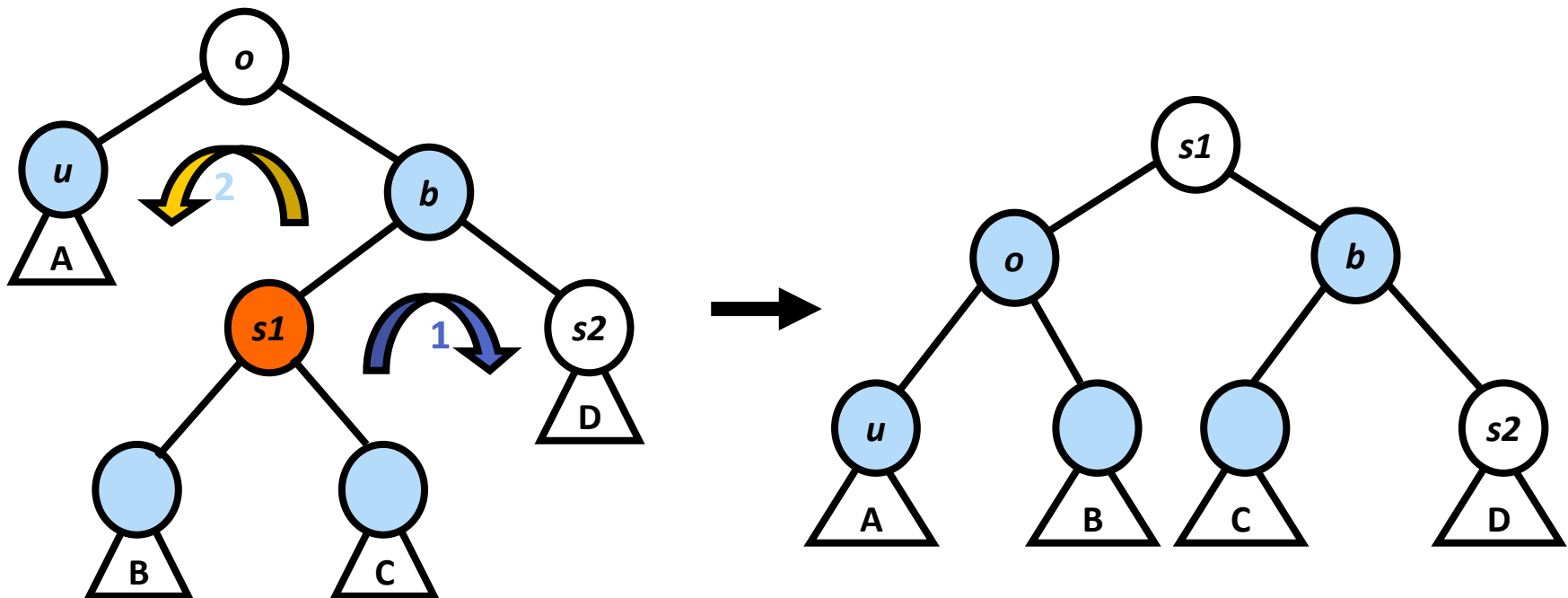
Poruchy při mazání (4)

(c) Synovec $s1$, jehož hodnota leží mezi hodnotami otce a bratra, je **černý**, druhý synovec je **červený**. Přebarvíme a rotujeme podle obrázku, barva otce se nemění (tj., vrchol b bude mít barvu, kterou původně měl vrchol o). Dostáváme **červeno-černý** strom.



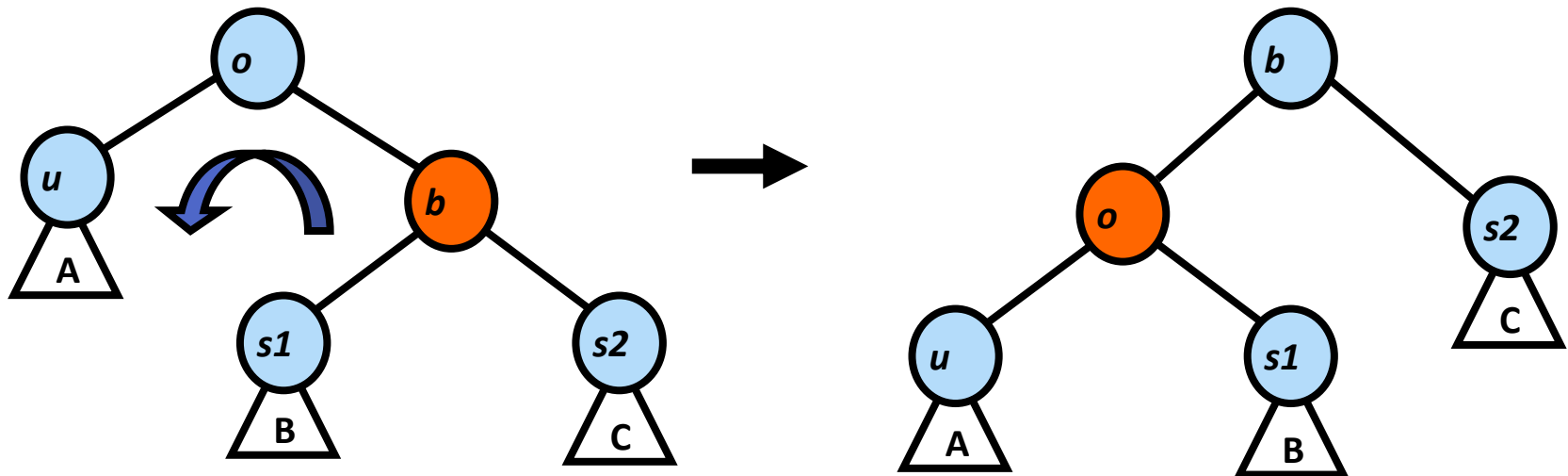
Poruchy při mazání (5)

(d) Synovec $s1$, jehož hodnota leží mezi hodnotami otce a bratra, je **červený**, druhý synovec má libovolnou barvu. Přebarvíme a RL rotací upravíme podle obrázku (tj. vrchol $s1$ bude mít barvu, kterou měl původně vrchol o a barva vrcholu $s2$ se nezmění). Dostáváme **červeno**-černý strom.



Poruchy při mazání (6)

2. Bratr je **červený**. Přebarvíme a rotujeme podle obrázku. Dostáváme **červeno-černý** strom s poruchou, přičemž porucha je o hladinu níže. I když to tak na první pohled nevypadá, máme vyhráno, protože bratr poruchy je **černý** a otec **červený**, tedy příští oprava bude případ 1b, 1c, nebo 1d a skončíme s **červeno-černým** stromem.



Mazání v RB-stromě

- Časová složitost je $O(\log(n))$.
- Jsou zapotřebí maximálně tři rotace.

Důkaz výšky RB-stromu

Věta: RB-strom s n interními uzly má výšku h nejvýše $2\lg(n+1)$.

Důkaz:

- Ukažme, že podstrom s kořenem v x obsahuje nejméně $2^{bh(x)}-1$ interních uzlů. Indukcí dle výšky podstromu s kořenem v x :
 - Pokud je x list, pak $bh(x) = 0$, $2^{bh(x)}-1 = 0$ interních uzlů (uzel **nil**).
 - Předpokládejme, že výška x je h , potomci x mají výšku $h-1$:
 - černá výška potomků x je buď $bh(x)-1$ nebo $bh(x)$ (podle barvy),
 - indukční předpoklad je, že potomci x mají nejméně $(2^{bh(x)}-1)-1$ interních uzlů,
 - pak podstrom s kořenem v x obsahuje nejméně:

$$2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$$
 interních uzlů.
 - Necht' h je výška stromu, který rotujeme v x :
 - $bh(x) \geq h / 2$ (max $\frac{1}{2}$ je červených),
 - pak $n \geq 2h/2 - 1 \Leftrightarrow n + 1 \geq 2h/2 \Leftrightarrow \lg(n+1) \geq h / 2$ a
 - $h \leq 2\lg(n+1)$.

Zhodnocení pro RB-stromy

- Pro binární vyhledávací červeno-černé stromy lze implementovat SEARCH, INSERT a DELETE tak, že vyžadují čas $O(\log n)$ a INSERT používá nejvýše jednu (dvojitou) rotaci a DELETE používá nejvýše dvě rotace nebo rotaci a dvojitou rotaci.
- RB-stromy jsou lepší než AVL stromy, které pro DELETE potřebují až $\log n$ rotací. Oproti váhově vyváženým stromům i proti AVL stromům jsou červeno-černé stromy sice jen konstantně lepší, ale i to je dobré.
- Při použití binárních vyhledávacích stromů ve výpočetní geometrii nese informaci i rozložení prvků ve stromě, a tato informace se musí po provedení rotace nebo dvojitě rotace aktualizovat. To znamená prohledání celého stromu a tedy čas $O(n)$ za každou rotaci a dvojitou rotaci navíc.
- Pro tyto problémy jsou červeno-černé stromy obzvláště vhodné, protože minimalizují počet použitých rotací a dvojitých rotací.

B-stromy (B-trees)

1. Motivace
2. Více-cestné vyhledávací stromy (multiway search trees)
3. Definice B-stromu
4. Search
5. Insert
6. Delete

Motivace pro B-stromy

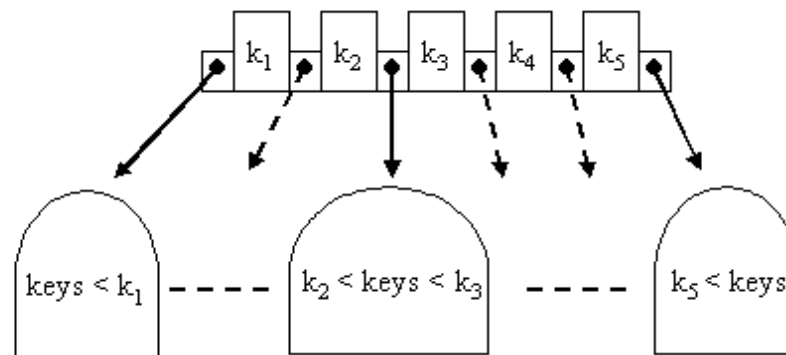
- Velká data se nevejdou do operační paměti -> používáme disky (vnější paměti s přímým přístupem).
- Čas přístupu na disk je určen HW a OS.
- Přístup na disk je MNOHEM pomalejší, v porovnání s přístupem do operační paměti:
 - 1 diskový přístup ~ 13 000 000 instrukcí!!!!
 - Počet diskových operací dominuje v odhadu časové náročnosti.
- Přístup na disk = Disk-Read, Disk-Write
 - Disk je rozdělen do bloků (512, 2048, 4096, 8192 bytes), které se přenášejí jako celek.
 - Proto se může hodit návrh více-cestných vyhledávacích stromů (*multiway search trees*), kde každý uzel stromu „pasuje“ do jednoho bloku na disku.

DISK : 16 ms
Seek 8ms + rotational
delay 7200rpm 8ms

Instruction:
800 MHz 1,25ns

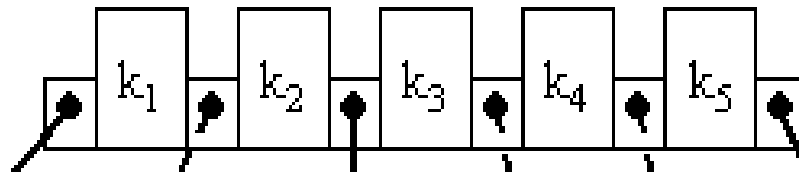
Více-cestné vyhledávací stromy

- Zobecnění binárních vyhledávacích stromů, kde počet následníků byl $m=2$.
- Každý uzel více-cestného stromu má nejvýše m potomků ($m>2$).
- Interní uzly obsahující n klíčů mají $n+1$ následníků, $n < m$ (výjimkou může být kořen stromu).
- Listy nemají následníky.
- Strom je uspořádan.
- Klíče v uzlech vymezují intervaly v podstromech - viz obrázek:



Listy více-cestného vyhledávacího stromu

- Listy více-cestného vyhledávacího stromu nemají žádné následníky a neobsahují žádné ukazatele na podstromy. Obvykle obsahují vlastní hodnoty.
- Platí: $k_1 < k_2 < \dots < k_5$

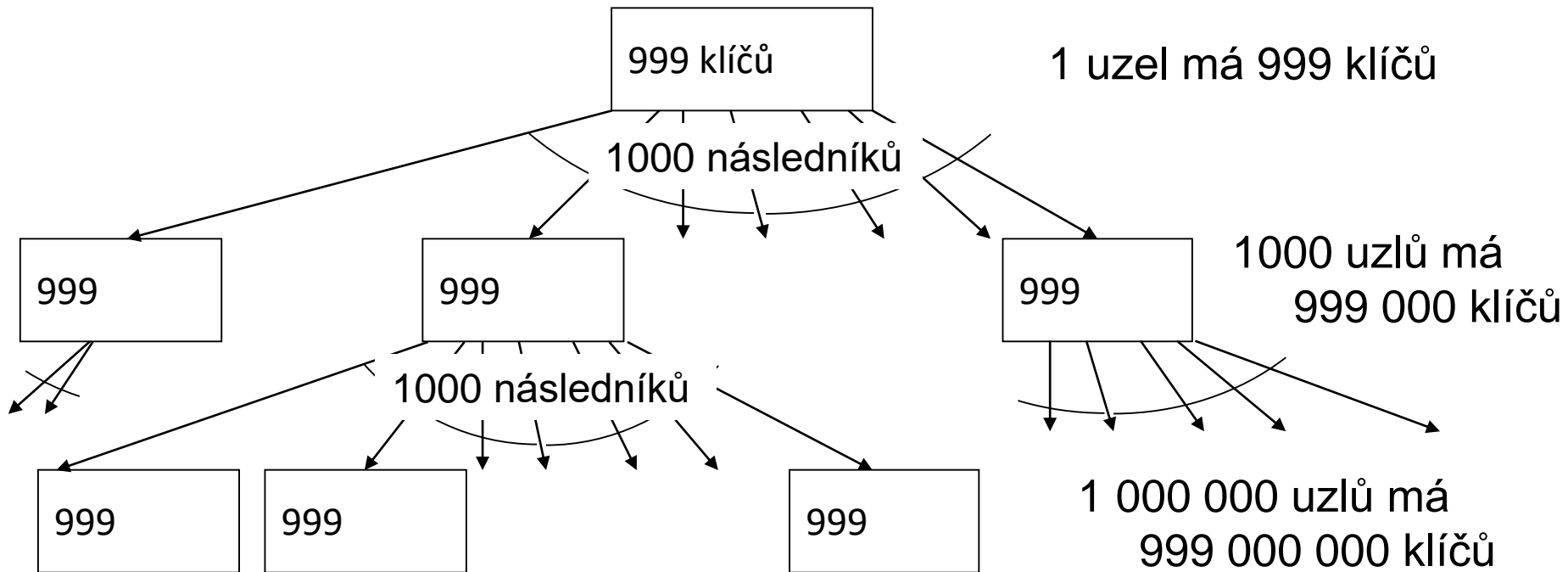


© Frederic Maire, QUT

B-stromy (B-trees)

- **B-strom** je více-cestný vyhledávací strom stupně m , který splňuje další podmínky:
- Všechny listy mají stejnou výšku (B-strom je vyvážený - vybalancovaný).
- Pro všechny interní uzly platí:
 - mají nejméně $m/2$ neprázdných potomků (precizně později) a
 - mají nejvýše m neprázdných potomků.
- Kořen může mít 0 nebo 2 až m potomků:
 - 0 - list
 - m - tzv. **plný uzel (full node)**

Příklad B-stromu



B-strom stupně $m=1000$ výšky 2 obsahuje
 1 001 001 uzlů ($1+1000 + 1\,000\,000$)
 999 999 999 klíčů ~ 1 miliarda klíčů

Obsah uzlů B-stromu

- n ... počet klíčů k_i uložených v uzlu.
- Uzel, kde $n = m-1$ se nazývá **plný uzel (full-node)**.
- k_i ... n klíčů, uložených v neklesajícím pořadí
- $k_1 \leq k_2 \leq \dots \leq k_n$
- *list* ... booleovská hodnota, true pro list, false pro ostatní.
- c_i ... $n+1$ ukazatelů na následníky (nedefinováno pro listy).
- Pro klíče k_i v podstromech platí:
- $keys_1 \leq k_1 \leq keys_2 \leq k_2 \leq \dots \leq k_n \leq keys_{n+1}$

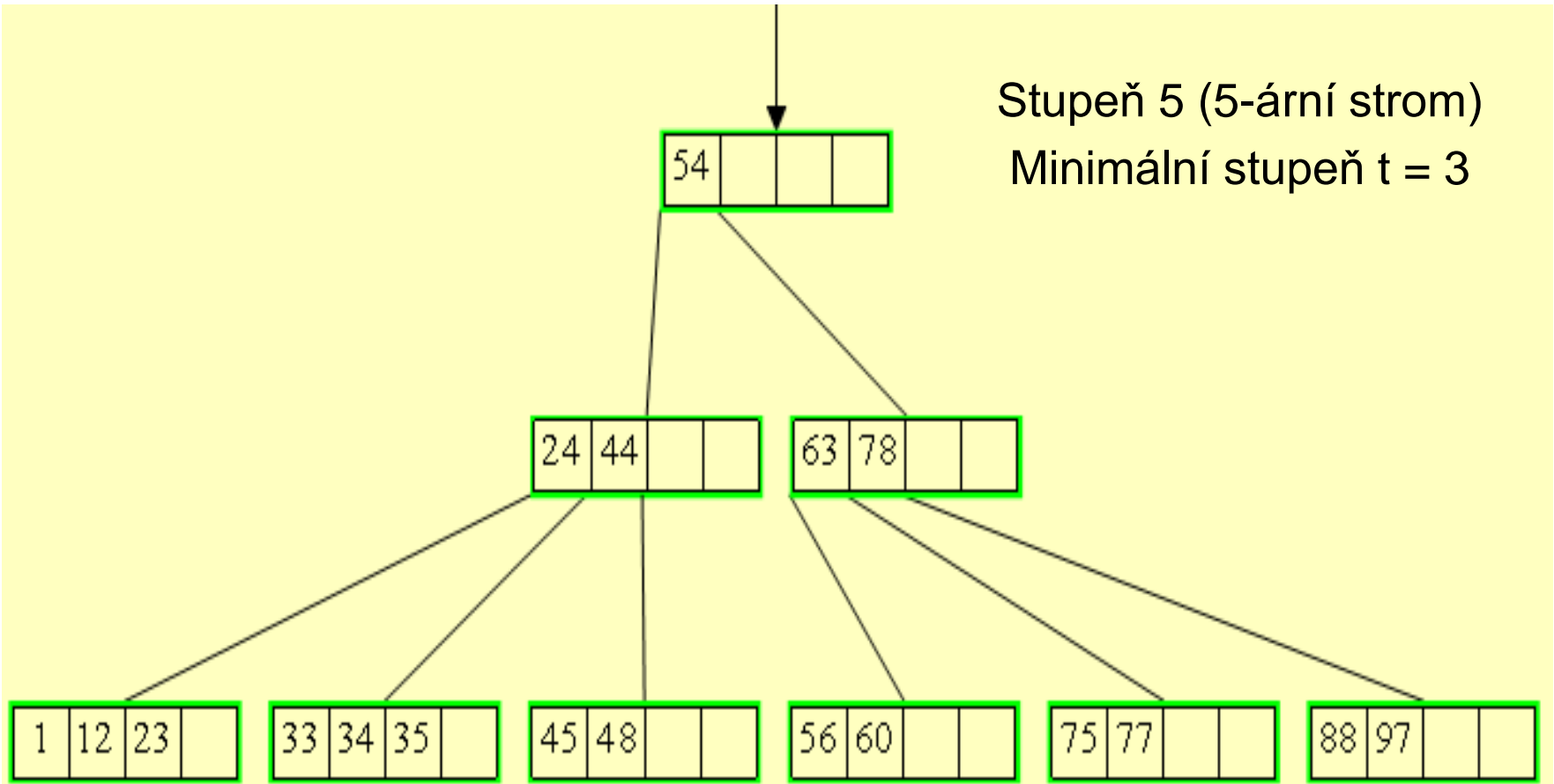
B-stromy (1)

Motivace:

- velmi rozsáhlé datové soubory bývají uloženy na vnější paměti (disku)
- přístup na disk (čtení jedné stránky) je typicky o 4 – 5 řádů pomalejší než přístup do vnitřní paměti
- při hledání v datech potřebujeme minimalizovat počet přístupů
- používáme hierarchickou soustavu indexů v podobě k-árních stromů s vysokým stupněm větvení k (uzel může obsahovat až k-1 klíčů a k odkazů na potomky)
- např. pro k=1001 může strom výšky 2 obsahovat až 1 003 003 000 klíčů (v kořeni až 1000, v potomcích kořene až $1001 \times 1000 = 1\,001\,000$, v listech $1001 \times 1001 \times 1000 = 1\,002\,001\,000$)
- vnitřní uzly stromu obsahují klíče a odkazy na další uzly
- listy obsahují pouze klíče

POZOR: budeme předpokládat, že asociovaná informace je uložena spolu s klíčem (např. formou odkazu na disk apod.)

B-strom



Based on [Cormen] and [Maire]

B-stromy (2)

B-strom zavedeme jako kořenový strom s těmito vlastnostmi:

1. každý jeho uzel x obsahuje tyto složky

- $x.n$ – počet skutečně uložených klíčů v x
- $x.n$ klíčů uspořádaných vzestupně $x.key[1] \leq x.key[2] \leq \dots \leq x.key[x.n]$
- $x.leaf$ – příznak, zda se jedná o list

2. každý vnitřní uzel x obsahuje $x.n+1$ odkazů $x.c[1], x.c[2], \dots, x.c[x.n+1]$ na své potomky, v listech mají tyto odkazy nedefinovanou hodnotu

3. klíče $x.key[i]$ rozdělují rozsahy klíčů uložených v každém podstromu takto: je-li k_i libovolný klíč uložený v podstromu s kořenem $x.c[i]$, pak platí

$$k_1 \leq x.key[1] \leq k_2 \leq x.key[2] \leq k_3 \leq \dots \leq x.key[x.n] \leq k_{x.n+1}$$

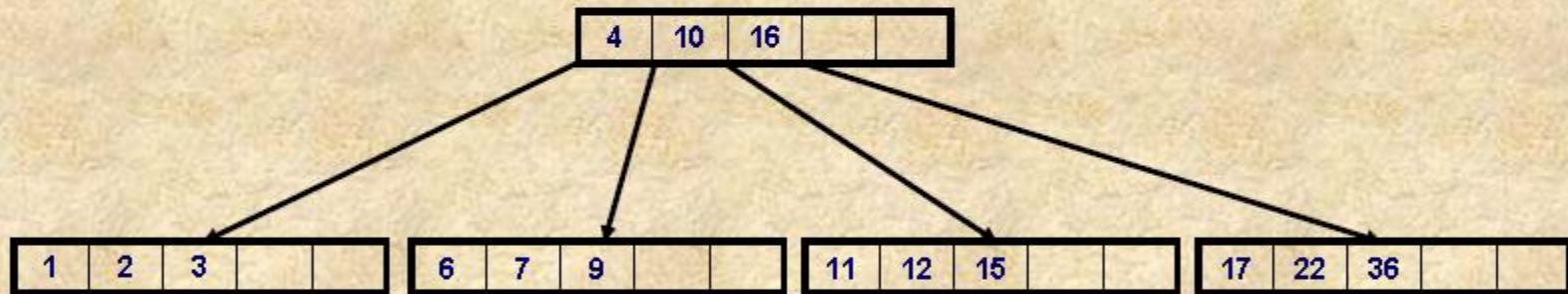
4. všechny listy jsou ve stejné hloubce od kořene, což je výška h daného B-stromu

5. je stanoven minimální a maximální počet klíčů v uzlu pomocí čísla $t \geq 2$ takto:

- kořen (neprázdného stromu) musí obsahovat alespoň jeden klíč, ostatní uzly **alespoň $t-1$ klíčů** (tedy alespoň t potomků)
- každý uzel smí obsahovat **nejvýše $2t-1$ klíčů** (tj. nejvýše $2t$ potomků)

B-stromy (3)

B-strom pro minimální stupeň $t = 3$, $\min t-1 = 2$, $\max 2t-1 = 5$ klíčů



Operace s B-stromy

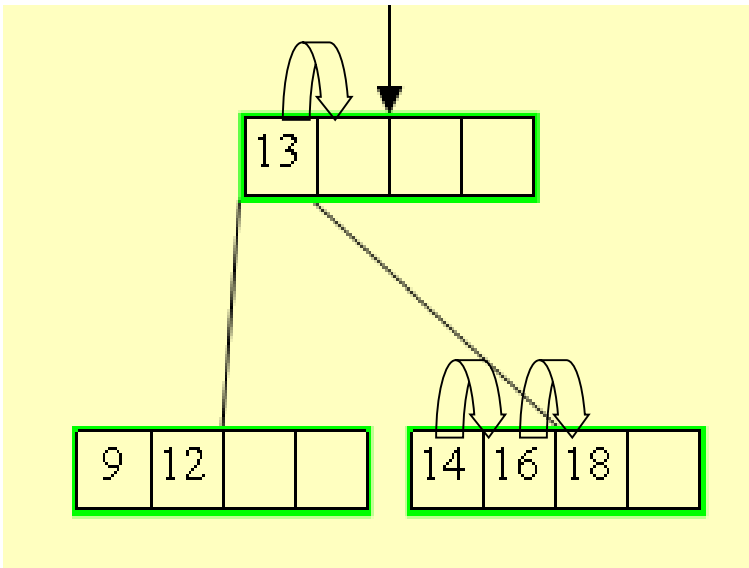
- Search
- Insert
- Delete

Hledání v B-stromech

- Obdobné hledání v BVS.
- V rámci uzlů se hledá sekvenčně, nebo binárně.
- Vstup: ukazatel na kořen stromu a klíč k
- Výstup: uspořádaná dvojice (y, i) , kde y je uzel a i je index v rámci uzlu takový, že $y.k[i] = k$ nebo hodnota NIL, pokud klíč k nebyl nalezen.

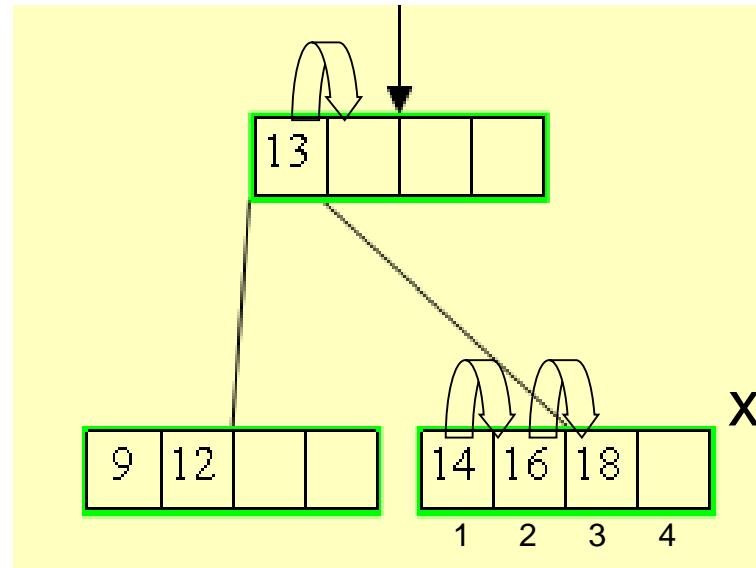
Příklad hledání v B-stromu

- Hledej 17 17



17 not found => return NIL

- Hledej 18 18



18 found => return (x, 3)

B-stromy (4)

Hledání v B-stromu je analogické hledání v BVS:

B-Tree-Search (x, k) (volný pseudokód)

```

1. i = 1;
2. while ((i ≤ x.n) && (k > x.key[i])) i++;
3. if ((i ≤ x.n) && (k = x.key[i])) return (x, i);
4. if (x.leaf) return null;
5. else {
6.     Disk-Read(x.c[i]); return B-Tree-Search(x.c[i], k);
7. }
```

Poznámka – předpokládáme, že jednotlivé uzly/stránky se čtou z disku

Vytvoření prázdného B-stromu:

B-Tree-Create (T)

```

1. x = new BNode();
2. x.leaf = true; x.n = 0; DISK-WRITE(x);
3. T.root = x;
```

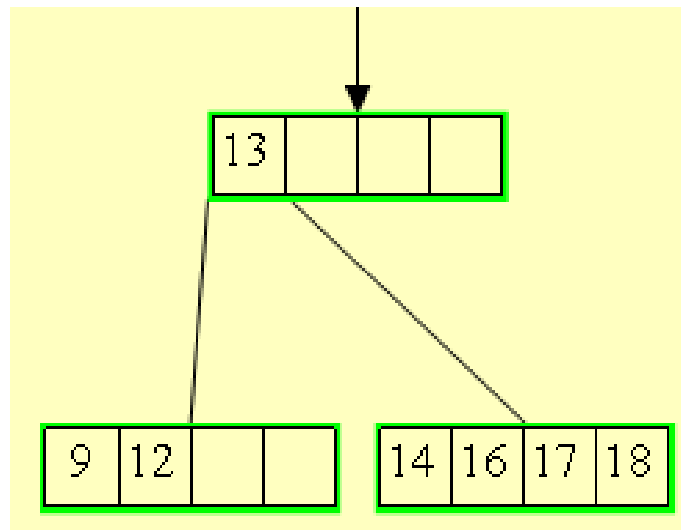
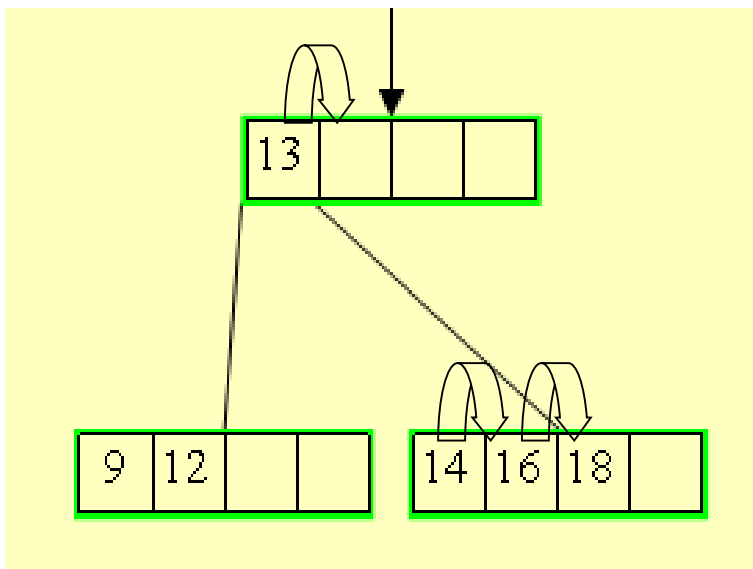
Operační složitost hledání v B-stromu

- Počet čtení z disku:
- $O(h) = O(\log_m n)$
- kde h je výška stromu a m je stupeň stromu a n je počet uzlů stromu.
- Protože počet klíčů je $x.n < m$, celá smyčka má horní odhad $O(m)$ a celková časová složitost je $O(m \log_m n)$.

B-stromy: insert - 1.vícefázová strategie

Insert do uzlu, který není zcela naplněn (**not full node**)

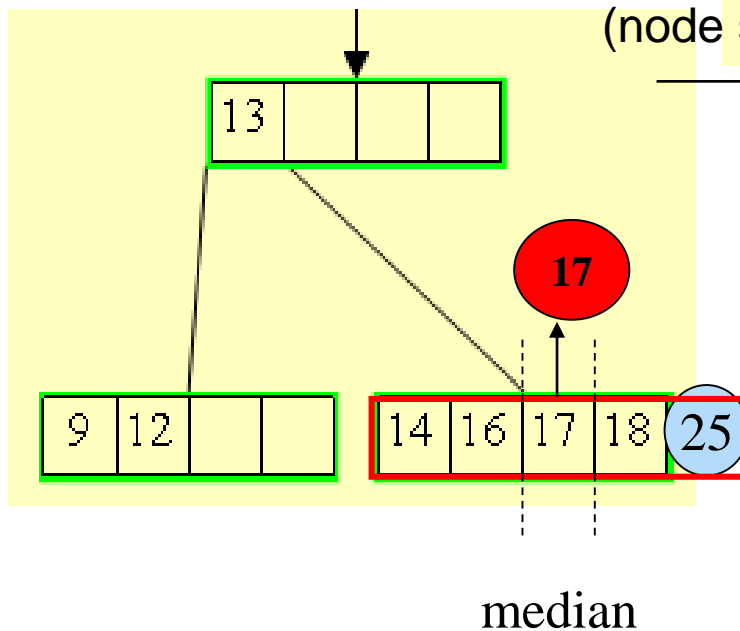
- Insert 17 17



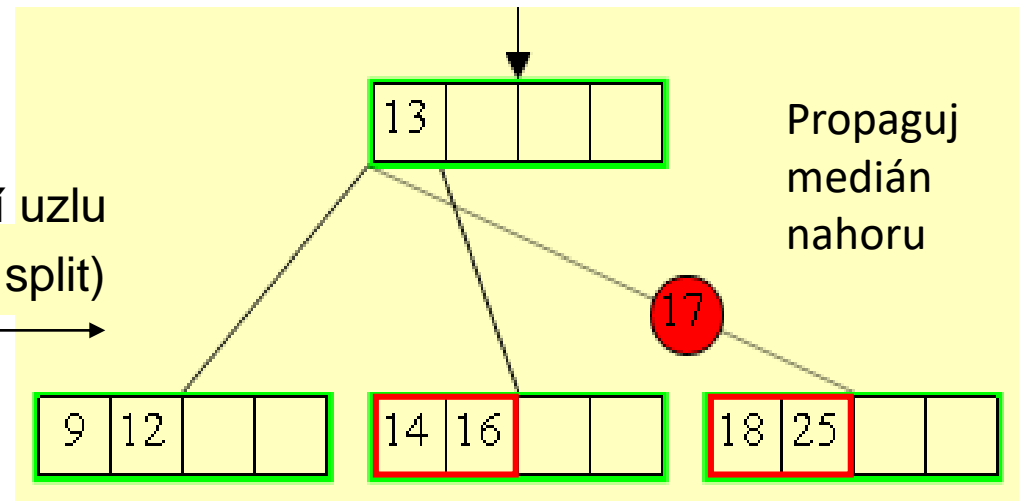
B-stromy: insert - 1.vícefázová strategie

Insert do „plného uzlu“

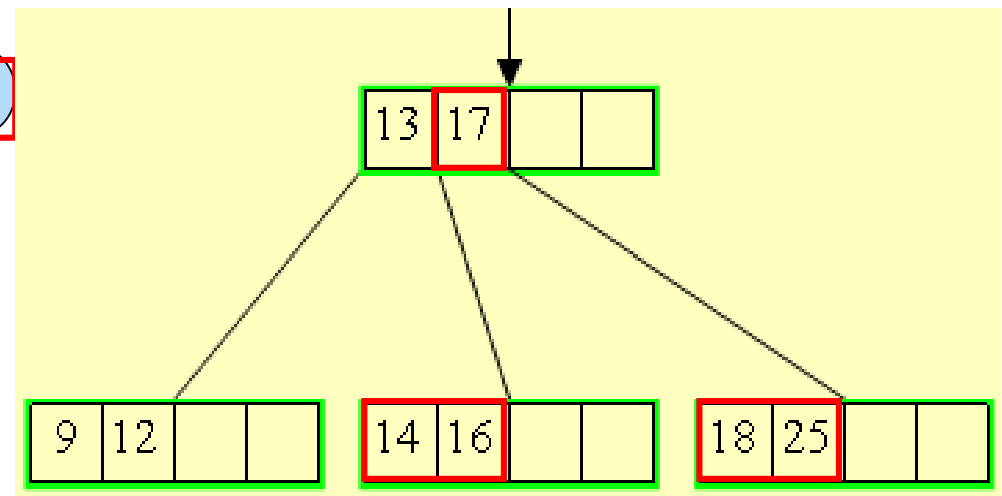
- Insert 25



Dělení uzlu
(node split)



Propaguj
medián
nahoru



1.Vícefázová strategie:

“řeš problém, až když se vyskytne”

B-stromy: insert - 1.vícefázová strategie

Insert (x, T) - pseudokód

- Najdi list pro x
- Pokud uzel není plný, insert x a konec
- **while** (je aktuální uzel plný)
- najdi medián (z klíčů po vložení x)
- rozděl uzel (split node) do dvou uzlů
- propaguj medián směrem nahoru
- aktuální uzel = rodič aktuálního uzlu

Fáze top-down

(node overflow)

Fáze bottom-up

B-stromy: insert - 2.jednofázová strategie

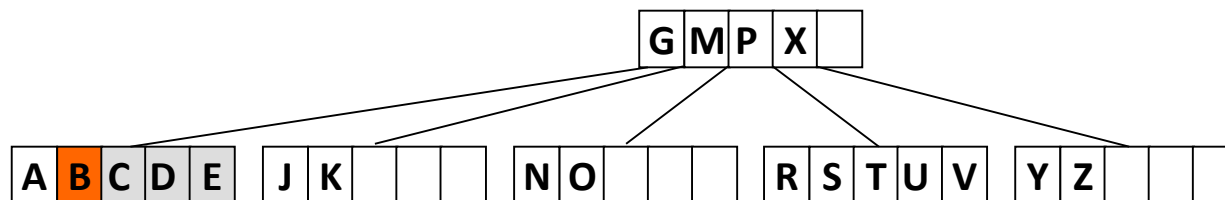
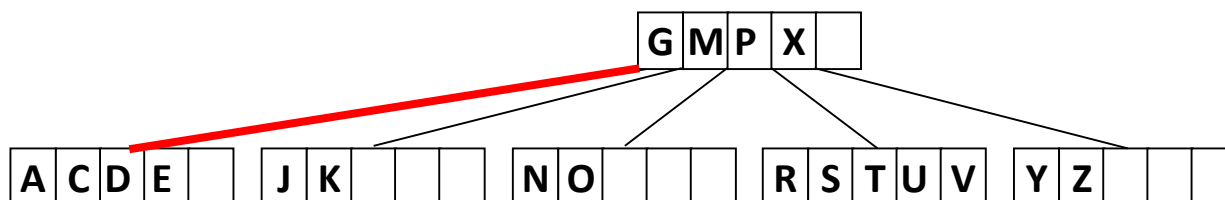
Princip: “ignoruj budoucí problémy”

Pouze fáze top-down

- Rozděl při vstupu naplněný uzel.
- Tím si vytvoříš prostor pro budoucí medián z potomků.
- Fáze „bottom-up“ není zapotřebí.
- Rozdělení (splitting of):
 - kořen => strom naroste o úroveň,
 - vnitřní uzel nebo list => rodič obdrží klíč, který je medián

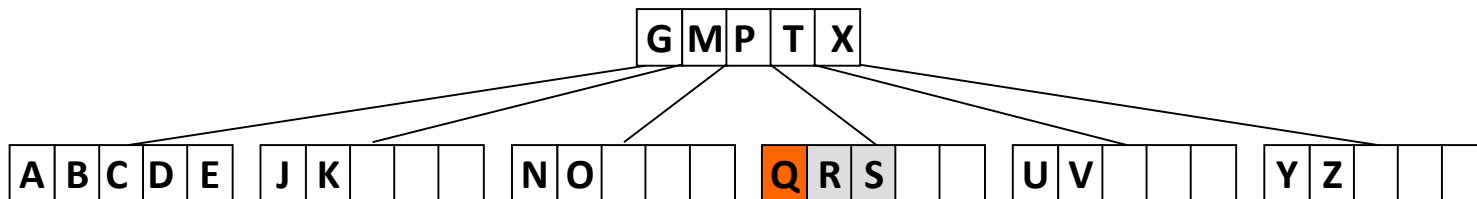
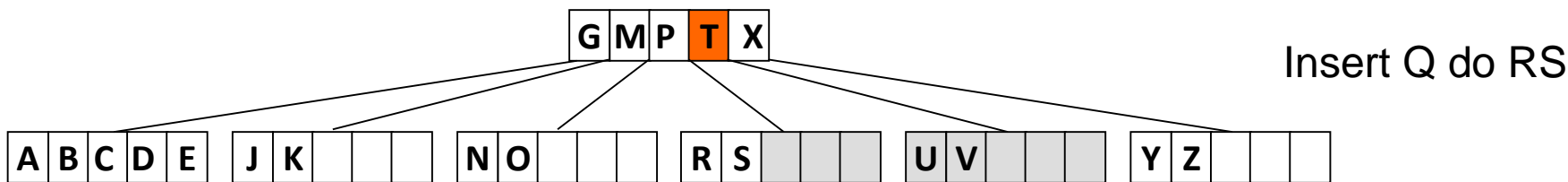
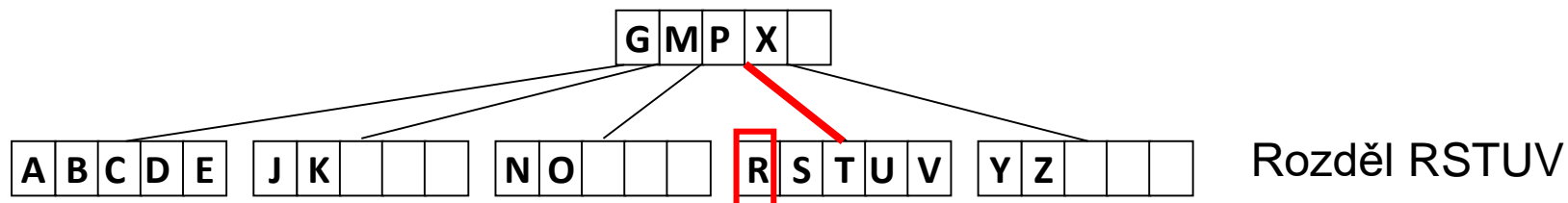
B-stromy: insert - 2.jednofázová strategie

Insert do nenaplněného uzlu (not full)



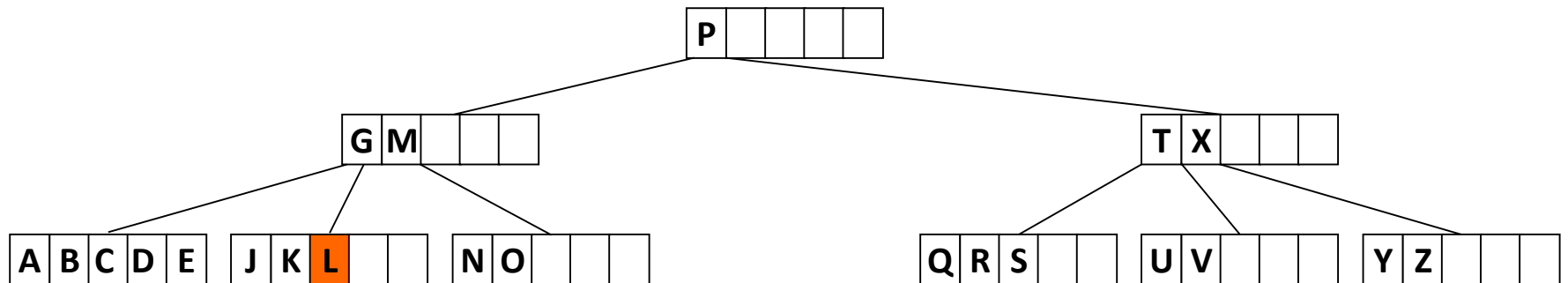
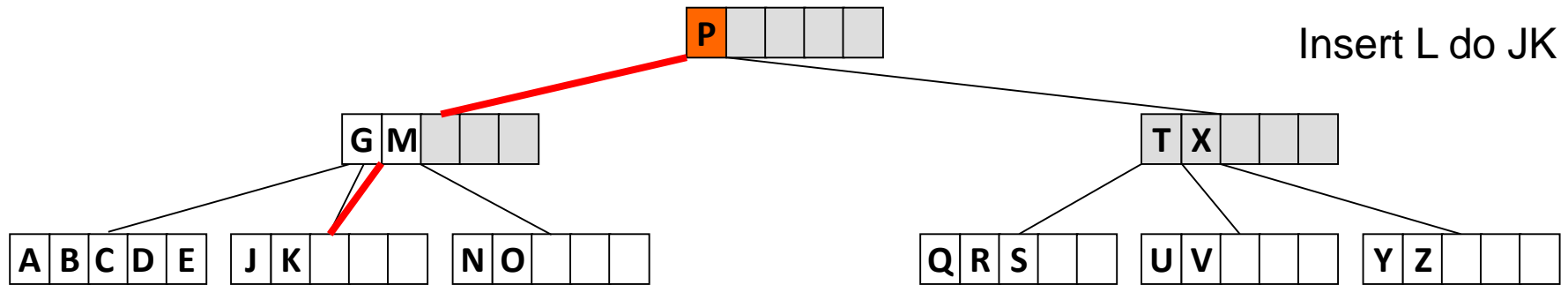
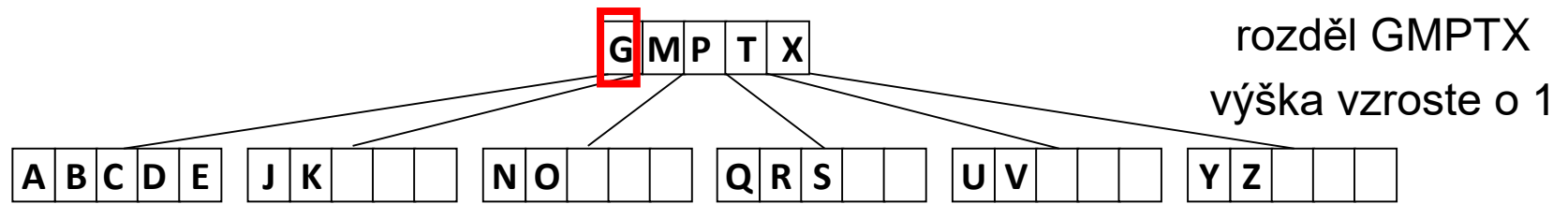
B-stromy: insert - 2.jednofázová strategie

Přeskok „full node“ a vložení do „not full“



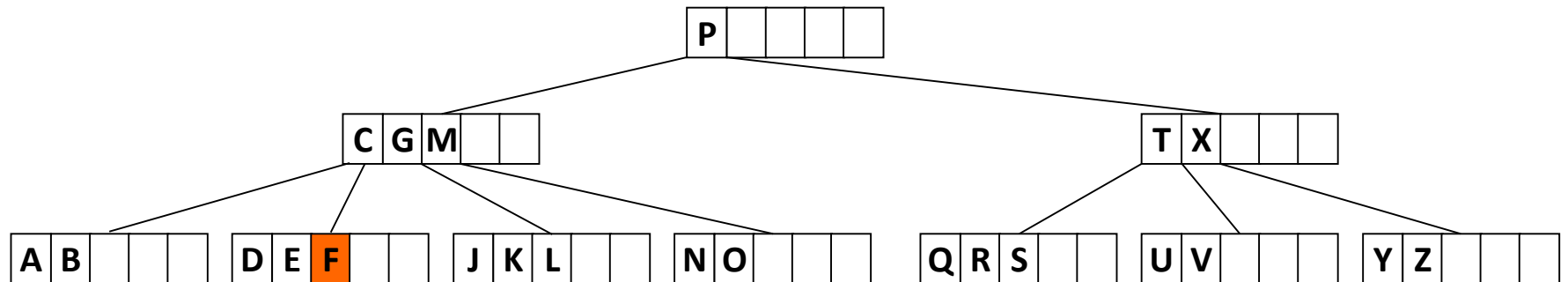
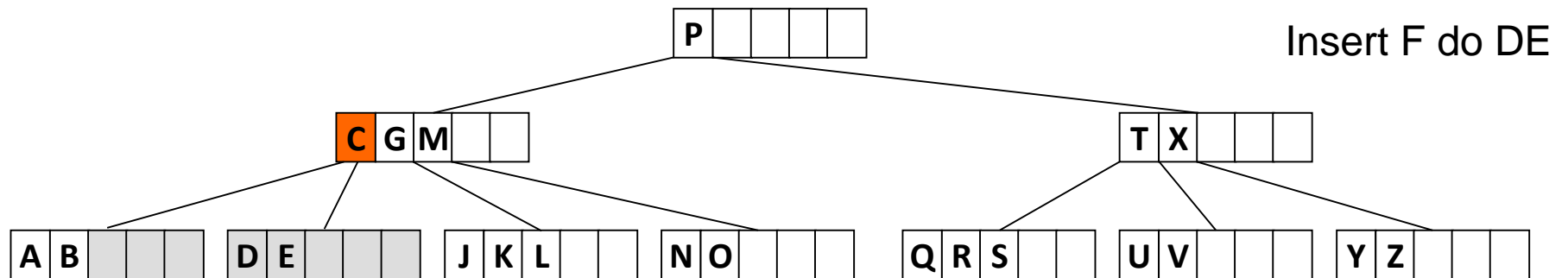
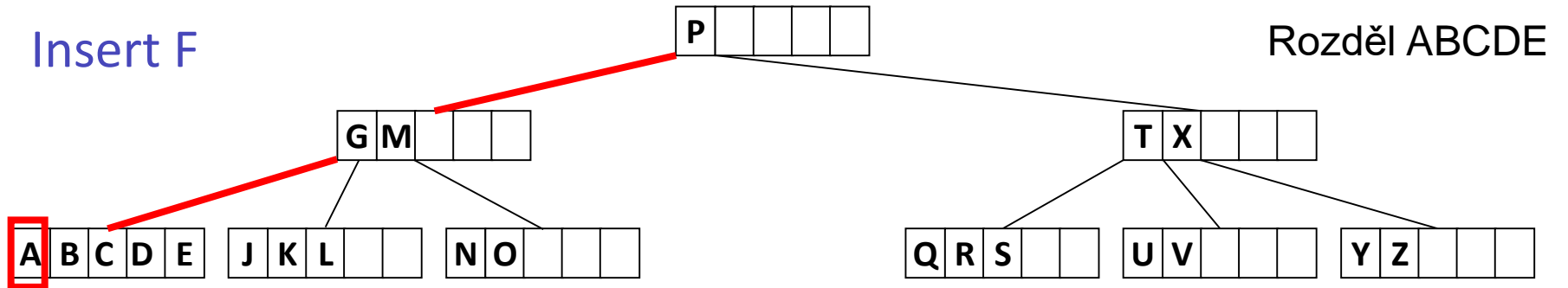
B-stromy: insert - 2.jednofázová strategie

Ignorování „full node“ a vložení do „not full“ uzlu



B-stromy: insert - 2.jednofázová strategie

- Insert F



B-stromy: insert - 2.jednofázová strategie

Insert (x, T) - pseudokód

Top down phase only

- While hledej listy x
- if (je uzel plný)
- najdi medián (mezi klíči naplněného uzlu)
- rozděl (split) uzel na dva
- vlož (insert) medián do rodiče (parent).
- Insert x a zastav.

B-stromy (5)

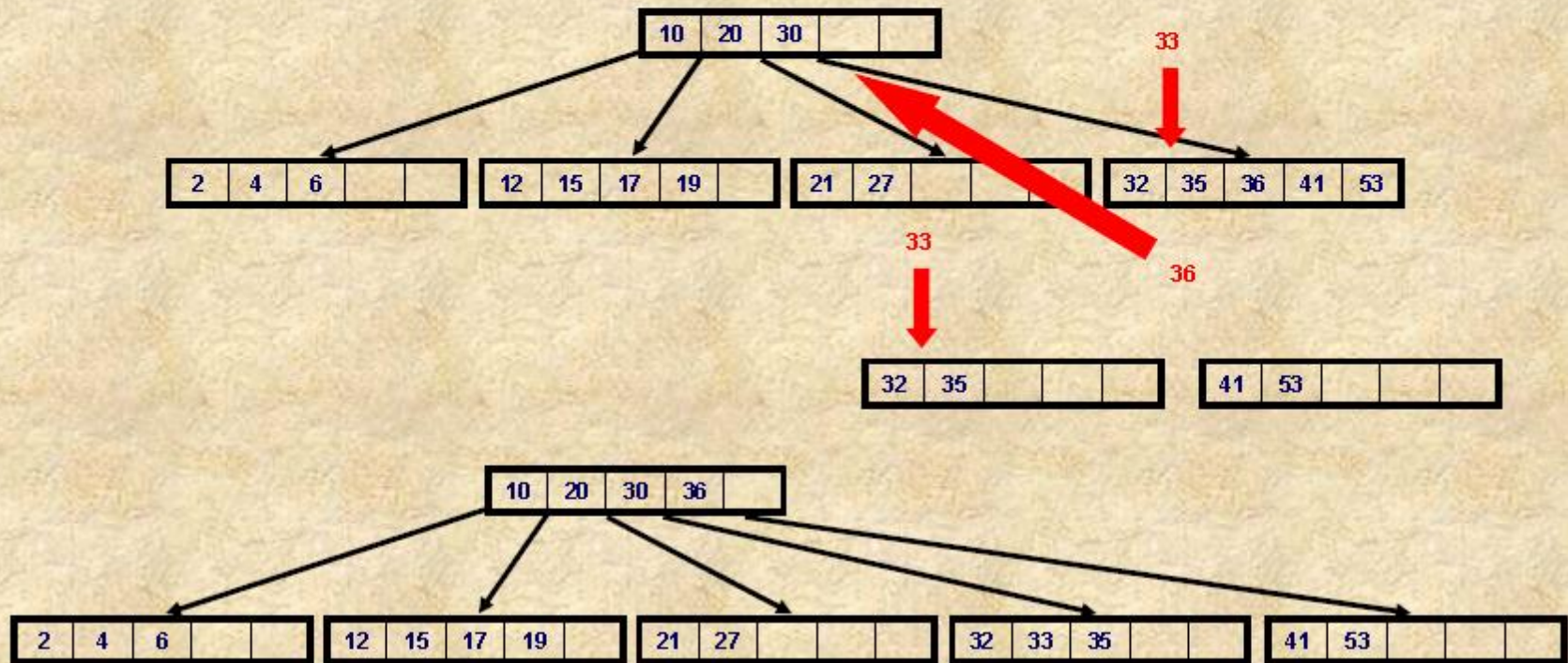
Základní postup vkládání do B-stromu:

- hledáme jako u BVS, kam umístit vkládaný klíč – bude to **vždy v listu**, nemůžeme vytvořit nový list
- pokusíme se vložit klíč do příslušného listu – to nelze provést, pokud je už zcela plný (má $2t-1$ klíčů)
- provedeme operaci **split** (rozdělení) listu podle jeho klíče-mediánu $y.key[t]$ na dva uzly, které mají $t-1$ klíčů každý
- klíč-medián se **posouvá do rodičovského uzlu**, aby určoval rozdělení mezi dvěma vytvořenými uzly
- pokud je ale rodič plný, pak se **musí také rozdělit**, dříve než se do něj nový klíč vloží
- operace **split** tedy může postupovat (potenciálně) **až do kořene** B-stromu
- postup můžeme navrhnout tak, že se **plné uzly (včetně kořene) rozdělují již cestou dolů** ve stromu, takže pro kterýkoliv plný uzel y už máme zaručeno, že jeho rodič plný není

B-stromy (6)

Vložení klíče 33:

příslušný list je plný, takže medián 36 půjde do rodiče, split plného listu na dva listy, klíč 33 jde do prvního listu



Strategie pro úpravy B-stromu (update)

Dvě principiální strategie

1. Více-fázová strategie:
 - “řeš problém, až když se vyskytne”
2. Více-fázová strategie: [Cormen]
 - “neřeš budoucí problémy”
 - Akce:
 - Rozděl plné uzly
 - Sluč uzly, které mají minimální obsah (méně než minimum položek).

Delete pro B-stromy

Delete (x, btree) - principy

Pouze více-fázová strategie

- Najdi (search) hodnotu, která se má zrušit.
- Pokud se jedná o jednoduchý uzel, pak jednoduše zruš uzel, jinak použij algoritmus „zrušEntry“ pro listy stromu a jednoduše uzel zruš (corrections of number of elements later).
- Pokud se jedná o vnitřní uzel, pak:
 - Slouží jako separátor pro dva podstromy.
 - Prohodí se buňky pomocí swap s predecessor(x) nebo successor(x) a
 - Zruší se kořen (leaf).
- Pokud má list více než minimální počet položek:
 - Zruš x z listu a STOP.
- Jinak:
 - redistribuj hodnoty na správné místo (to může posunout problém do rodiče, zastaví se to v kořeni, protože nemá omezené minimum položek).

B-tree delete

Delete (x, btree) - pseudokód

Pouze více-průchodů

```
if(x to be removed is not in a leaf)
    swap it with successor(x)
currentNode = leaf
while(currentNode underflow)
    try to redistribute entries from an immediate
    sibling into currentNode via its parent
    if(impossible) then merge currentNode with a
    sibling and one entry from the parent
currentNode = parent of CurrentNode
```

Maximální výška B-stromu

- $h \leq \log_{\lceil m/2 \rceil} ((N+1)/2)$
- Dává horní hranici přístupů na disk.
- Viz [Maire] nebo [Cormen] podrobnosti

References

- [Cormen] Cormen, Leiserson, Rivest: Introduction to Algorithms, Chapter 14 and 19, McGraw Hill, 1990
- [Whitney]: CS660 Combinatorial Algorithms, San Diego State University, 1996], RedBlack, B-trees
<http://www.eli.sdsu.edu/courses/fall96/cs660/notes/redBlack/redBlack.html#RTFToC5>
- [Wiki] B-tree. (2006, November 24). In Wikipedia, The Free Encyclopedia. Retrieved 20:25, December 12, 2006, from
<http://en.wikipedia.org/w/index.php?title=B-tree&oldid=89805120>
- [Maire] Frederic Maire: An Introduction to Btrees, Queensland University of Technology, 1998]
<http://sky.fit.qut.edu.au/~maire/baobab/lecture/>
- [RB tree] John Franco - [java applet](#)
- <http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html>
- [Jones] Jeremy Jones: B-Tree animation - [java applet](#)
<https://www.cs.tcd.ie/Jeremy.Jones/vivio/trees/B-tree.htm>

The End