IAL – 6. přednáška

Vyhledávací tabulky II.

22. a 23. října 2024

Obsah přednášky

- Binární vyhledávání
  - Dijkstrova metoda
  - binární vyhledávací stromy
    - BVS se zarážkou
    - AVL stromy
- Stromy s více klíči ve vrcholech
  - (a,b)-stromy
  - RB stromy (LLRB)

Binární vyhledávání

- Lze provést nad seřazenou množinou klíčů ve struktuře s náhodným přístupem (v poli).
- Připomíná metodu půlení intervalu pro hledání jediného kořene funkce v daném intervalu
- Výhoda: časová složitost vyhledávání je v nejhorším případě logaritmická: $\boldsymbol{log_2(n)}$

- *K zamyšlení:* Pro $n \in \{10, 100, \ldots, 1.000.000\}$ porovnejte zaokrouhlené hodnoty pro nejhorší případ binárního vyhledávání a pro nejhorší případ sekvenčního vyhledávání.

Binární vyhledávání – algoritmus

- Binární vyhledávání lze použít pro seřazenou množinu klíčů, tzn. musí platit:

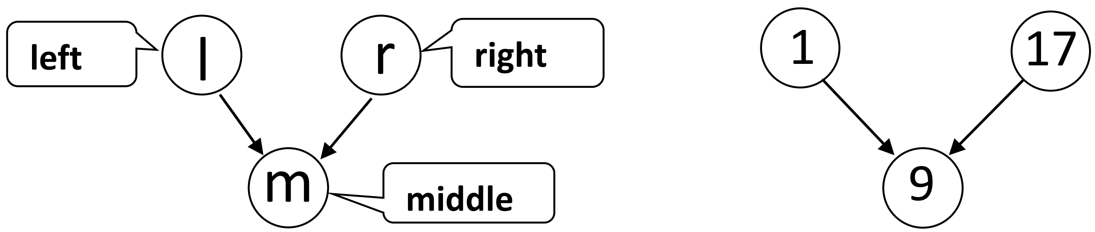
```
t.array[0].key < t.array[1].key < … < t.array[t.n-1].key
```

Binární vyhledávání – algoritmus

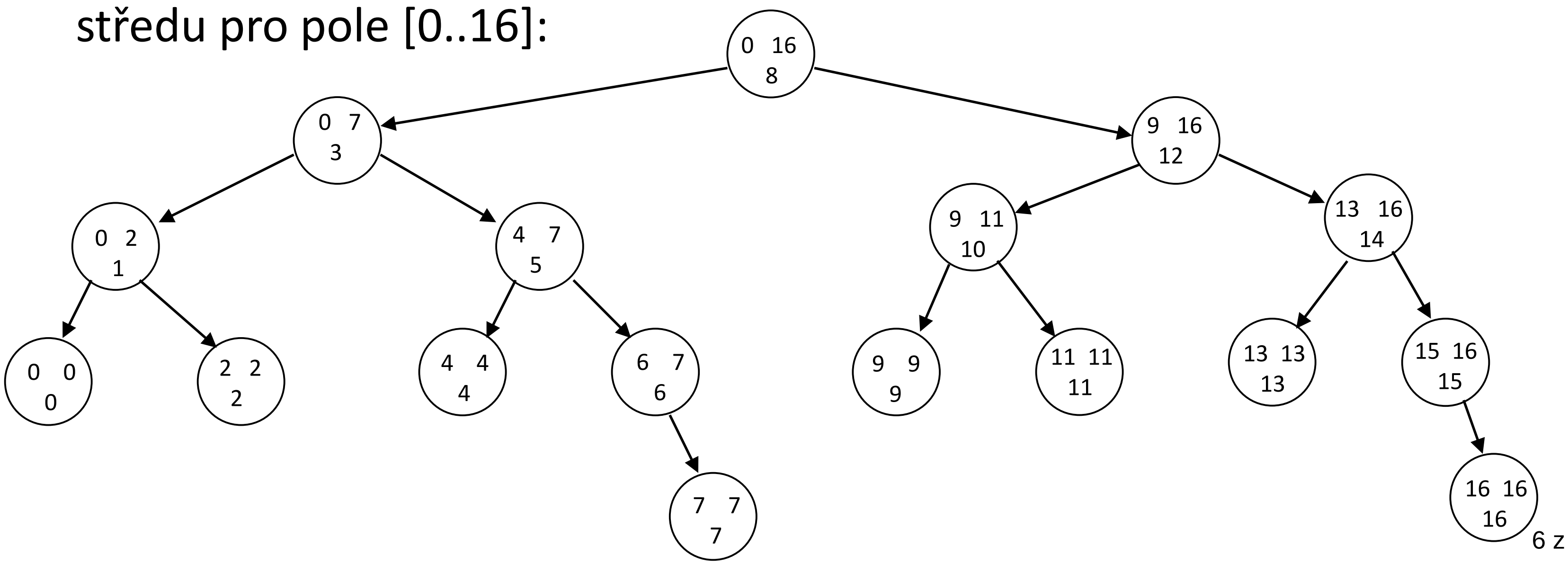
```
…
left ← 0              // levý index
right ← t.n-1         // pravý index
do:
  middle ← (left+right) div 2
  if k < t.array[middle].key:
    right ← middle-1     // hledaná položka je vlevo
  else:
    left ← middle+1      // hledaná položka je vpravo
while (k ≠ t.array[middle].key) and (left ≤ right)
return (k = t.array[middle].key)
```

Binární vyhledávání

- Mechanismus výpočtu středu je ⇨(left + right) div 2



- Rozhodovací strom binárního vyhledávání popisuje proces vývoje výpočtu středu pro pole [0..16]:



Binární vyhledávání – Dijkstrova varianta

- E.W. Dijkstra – významný teoretik programování druhé poloviny minulého století.
- Vychází z předpokladu, že v poli může být více položek se shodným klíčem.
  - Nenastává ve vyhledávací tabulce.
  - Použití pro účely řazení metodou Binary-insert sort.
- Je-li v seřazeném poli více klíčů se stejnou hodnotou, polohu kterého z nich má vrátit mechanismus Search?
  - Obvykle některý z krajních.
  - Nejčastěji poslední ze stejných.

Dijkstrova varianta – algoritmus

- Dijkstrova varianta umožňuje existenci více prvků se shodným klíčem, pro hledání nejpravějšího musí platit:

```
t.array[0].key ≤ t.array[1].key ≤ ...
... ≤ t.array[t.n-2].key < t.array[t.n-1].key
```

- a pro vyhledávaný klíč musí platit:

```
(k < t.array[t.n-1].key)
```

Dijkstrova varianta – algoritmus

```
…
left ← 0
right ← t.n-1
while right ≠ (left+1):
  middle ←(left+right) div 2
  if t.array[middle].key ≤ k:
    left ← middle
  else:
    right ← middle
return ((k = t.array[left].key), left)
```

Dijkstrova varianta – příklad

- *Příklad:*

  V poli: 1,1,1,1,1,1,1,1,1,**1**,2 najde algoritmus klíč K=1 na pozici 9.

  V poli: 1,2,3,4,5,5,6,6,**6**,8,9,13 najde algoritmus klíč K=6 na pozici 8 (počítáno od 0).

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
hodnota	1	2	3	4	5	5	6	6	6	8	9	13

middle = 5 (5<6): left = 5

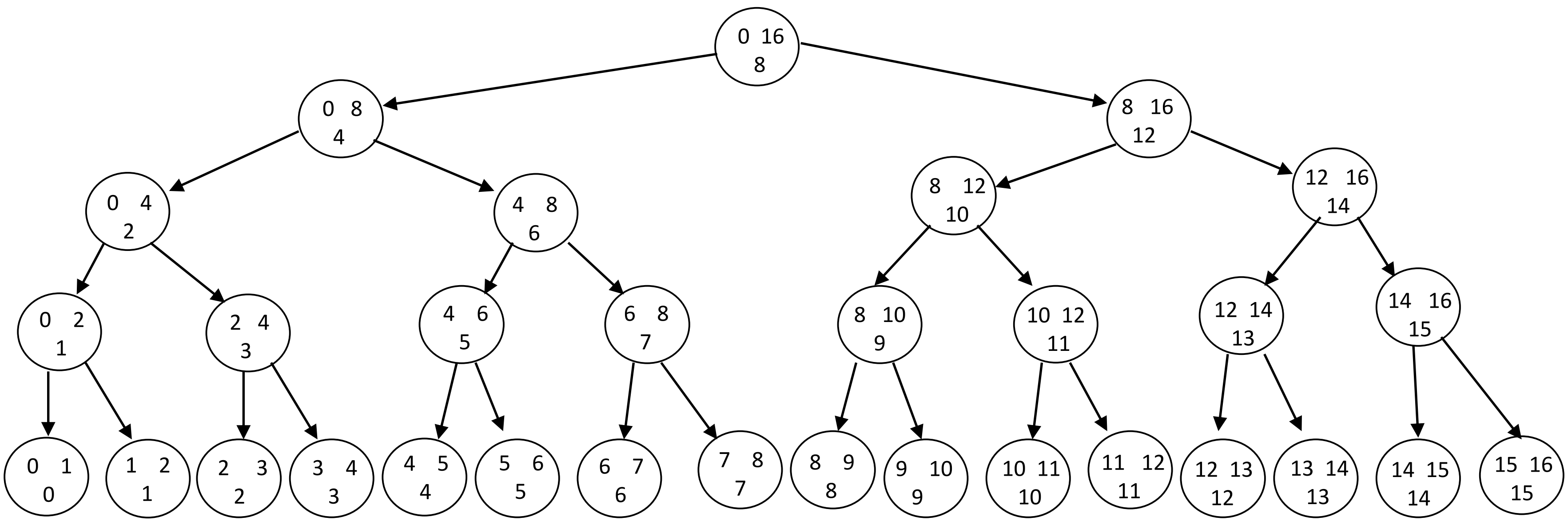
middle = 8 (6≤6): left = 8

middle = 9 (8 ≰ 6): right = 9

na pozici left je hledaný klíč

Dijkstrova varianta

- Rozhodovací strom Dijkstrovy varianty pro pole [0..16] má tvar:



- Dijkstrova varianta končí vždy za stejnou dobu, určenou hodnotou dvojkového logaritmu počtu prvků.

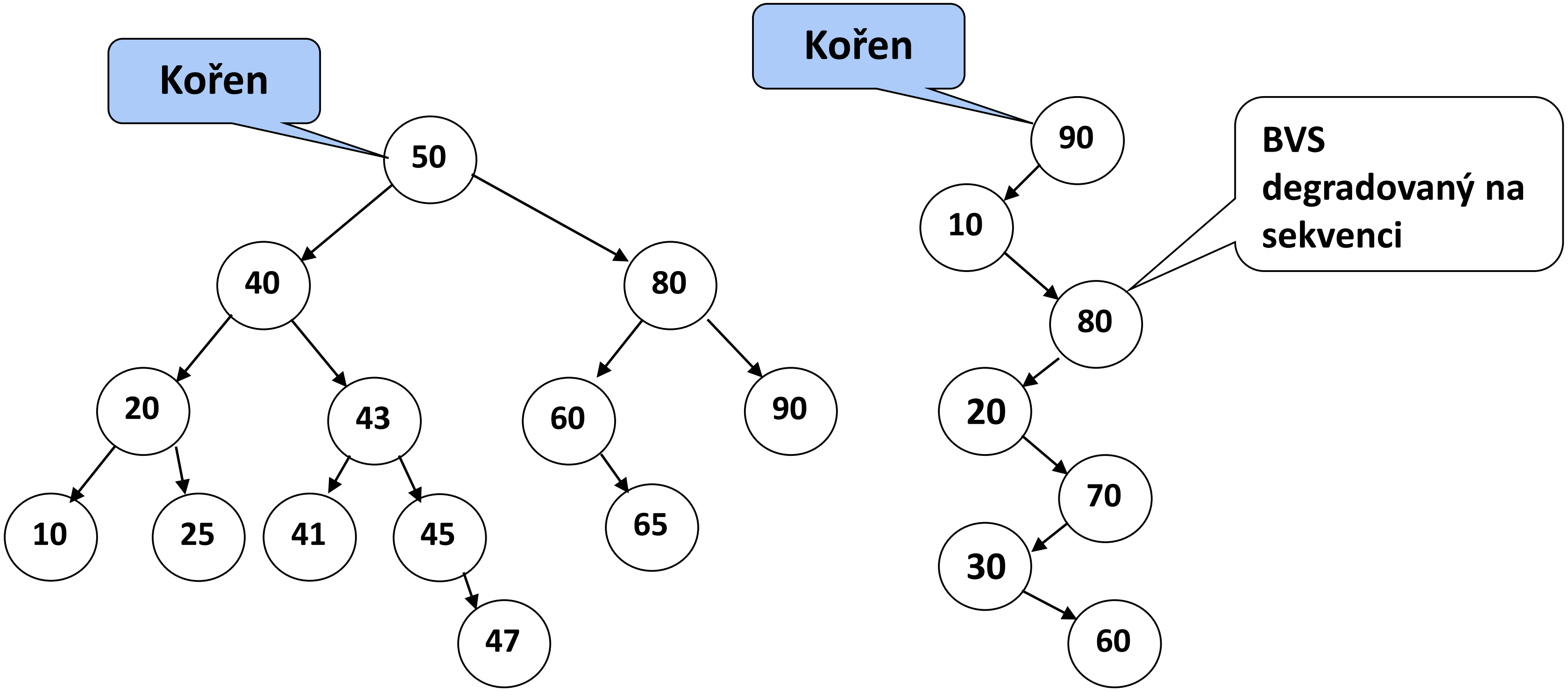
Binární vyhledávání – hodnocení

- Vyhledávání (operace Search) má logaritmickou časovou složitost: ***$log_2(n)$***
- Operace Insert a Delete mají stejný charakter jako u sekvenčního vyhledávání v seřazeném poli – vyžadují posuny segmentů pole!
- Je výhodné pro statické tabulky, kde se nemění počet prvků a není nutný potenciálně časově náročný posun segmentu pole.

Binární vyhledávací strom

- Uspořádaný strom: kořenový strom, pro jehož každý uzel platí, že n-tice jeho synů je uspořádaná.
- Binární vyhledávací strom:
  Binární uspořádaný strom, pro jehož každý uzel platí:
  - levý podstrom tohoto uzlu je buď prázdný nebo obsahuje uzly, jejichž hodnota je menší než hodnota tohoto uzlu a
  - pravý podstrom tohoto uzlu je buď prázdný nebo obsahuje uzly, jejichž hodnota je větší než hodnota tohoto uzlu.

Binární vyhledávací strom – příklady



- Průchod InOrder BVS stromem dává seřazenou posloupnost:
  - levý strom: 10, 20, 25, 40, 41, 43, 45, 47, 50, 60, 65, 80, 90
  - pravý strom: 10, 20, 30, 60, 70, 80, 90

Vyhledávání v binárním stromu

- Vyhledávání v BVS je podobné binárnímu vyhledávání v seřazeném poli:
  - Je-li vyhledávaný **klíč roven kořeni**, vyhledávání končí úspěšným vyhledáním.
  - **Je-li klíč menší**, pokračuje vyhledávání v levém podstromu, **je-li větší**, pokračuje v pravém podstromu.
  - Vyhledávání končí **neúspěšně**, pokud je prohledávaný (**pod**)**strom prázdný**.

BVS – implementace

- Datové typy používané pro BVS jsou podobné jako pro dvojsměrný seznam nebo binární strom:

```
typedef struct tnode
{
    TKey key;
    TData data;
    struct tnode *lPtr;
    struct tnode *rPtr;
}TNode;
```

BVS – Search (rekurzivní verze)

```
bool function Search (TNode *rootPtr, TKey k)
  if rootPtr = NULL:
    return (false)                          // nenašli jsme
  else:
    if rootPtr->key = k:
      return (true)                   // našli jsme
    else:
      if k < rootPtr->key:    // hledáme v levém podstromu
        return (Search(rootPtr->lPtr,k))
      else:                   // hledáme v pravém podstromu
        return (Search(rootPtr->rPtr,k))
```

BVS – Search (vracející ukazatel)

```
TNode* function SearchTree (TNode *rootPtr, TKey k)
  if rootPtr = NULL:
    return (NULL)                             // nenašli jsme
  else:
    if rootPtr->key ≠ k:
      if k < rootPtr->key:                    // hledáme vlevo
        return (SearchTree(rootPtr->lPtr, k))
      else:                                   // hledáme vpravo
        return (SearchTree(rootPtr->rPtr, k))
    else:                           // našli jsme a vracíme uzel
      return (rootPtr)
```

BVS – Search (nerekurzivní verze)

```
bool function Search (TNode *rootPtr, TKey k)
  search ← false
  finish ← rootPtr = NULL
  while not finish:
    if rootPtr->key = k:              // našli jsme
      finish ← true
      search ← true
    else:
      if k < rootPtr->key:
        rootPtr ← rootPtr->lPtr       // hledáme vlevo
      else:
        rootPtr ← rootPtr->rPtr       // hledáme vpravo
      if rootPtr = NULL:
        finish ← true
  return (search)
```

K procvičení

- Vytvořte následující nerekurzivní varianty zápisu funkce pro vyhledávání v BVS:
  a) funkce vrátí booleovskou hodnotu a prostřednictvím parametru `where` ukazatel na nalezený uzel (pro `Search = false` je `where` nedefinováno)
  b) funkce vrátí ukazatel `where` na nalezený uzel a **`NULL`** v případě neúspěšného vyhledávání

BVS – Insert

- Aplikuje *aktualizační sémantiku*:
  - pokud uzel s daným klíčem existuje, přepíše stará data novými.
  - Když uzel s daným klíčem neexistuje, vloží nový uzel jako terminální tak, aby byla dodržena pravidla BVS.

BVS – Insert

- Pro zkrácení zápisu použijeme pomocnou funkci, která vytvoří uzel:

```
TNode* CreateNode (TKey k, TData d)
{
   TNode *newPtr = (TNode *) malloc(sizeof(TNode));
            // zkontrolovat úspěšnost operace malloc
   newPtr->key = k;
   newPtr->data = d;
   newPtr->lPtr = NULL;
   newPtr->rPtr = NULL;
   return newPtr;
}
```

BVS – Insert (rekurzivní zápis)

```
TNode* function Insert (TNode *rootPtr, TKey k, TData d)
  if rootPtr = NULL:           // vytvoření nového uzlu
    return CreateNode(k,d)
  else:
    if k < rootPtr->key:                     // jdeme vlevo
      rootPtr->lPtr ← Insert(rootPtr->lPtr,k,d)
    else:
      if rootPtr->key < k:      // jdeme vpravo
        rootPtr->rPtr ← Insert(rootPtr->rPtr,k,d)
      else                    // přepíšeme stará data novými
        rootPtr->data ← d
    return rootPtr
```

BVS – Insert (nerekurzivní verze)

- Použijeme pomocnou funkci **`SearchIns`**, která se pokusí najít prvek s daným klíčem.
- Vrací následující hodnoty:
  - Booleovská hodnota, která udává, zda byl prvek s daným klíčem nalezen.
  - Ukazatel na uzel, ve kterém hledání skončilo:
    - Úspěšně – bude přepsána hodnota v tomto uzlu.
    - Neúspěšně – uzel, ke kterému bude nový prvek připojen.

BVS – Insert (nerekurzivní verze)

```
(bool, TNode*) function SearchIns (TNode *rootPtr, TKey k)
// Vyhledání za účelem nerekurzivního vkládání
  found ← false
  if rootPtr = NULL:
    where ← NULL
  else:
    do:
      where ← rootPtr          // uchování hodnoty where
      if k < rootPtr->key:                // posun doleva
        rootPtr ← rootPtr->lPtr
      else:
        if rootPtr->key < k:           // posun doprava
          rootPtr ← rootPtr->rPtr
        else:
          found ← true                   // našel
    while not found and (rootPtr ≠ NULL)
  return (found, where)
```

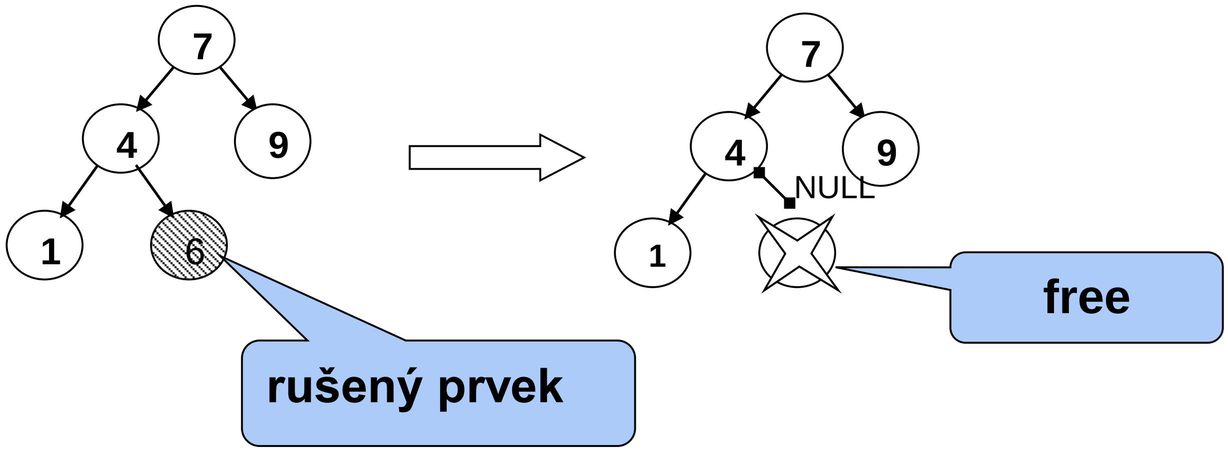
BVS – Insert (nerekurzivní verze)

```
TNode* function Insert (TNode *rootPtr, TKey k, TData d)
  found, where ← SearchIns(rootPtr,k)
  if found:
    where->data ← d          // přepsání starých dat novými
  else:
    newPtr ← CreateNode(k,d)
    if where = NULL:         // nový kořen do prázdného stromu
      rootPtr ← newPtr
    else:                    // nový se připojí jako list …
      if k < where->key:                         // … vlevo
        where->lPtr ← newPtr
      else:                                      // … vpravo
        where->rPtr ← newPtr
  return rootPtr
```

BVS Rušení uzlu – operace Delete

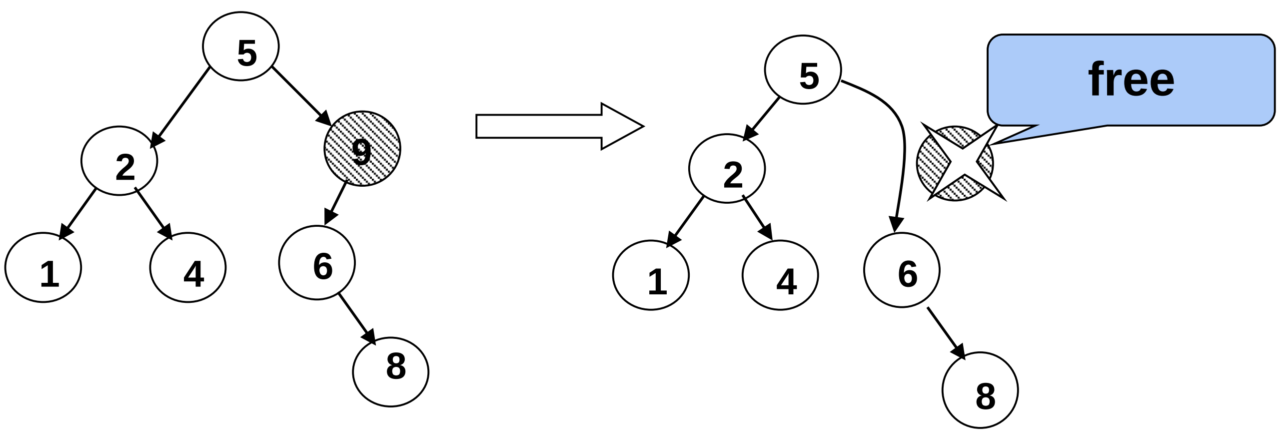
- Při rušení uzlu je třeba rozlišit, o jaký uzel se jedná:
  - Terminální uzel – rušení je snadné.
  - Uzel s jedním synem – také snadné.
  - Uzel se dvěma syny – komplikovanější.

- Rušení terminálního uzlu:



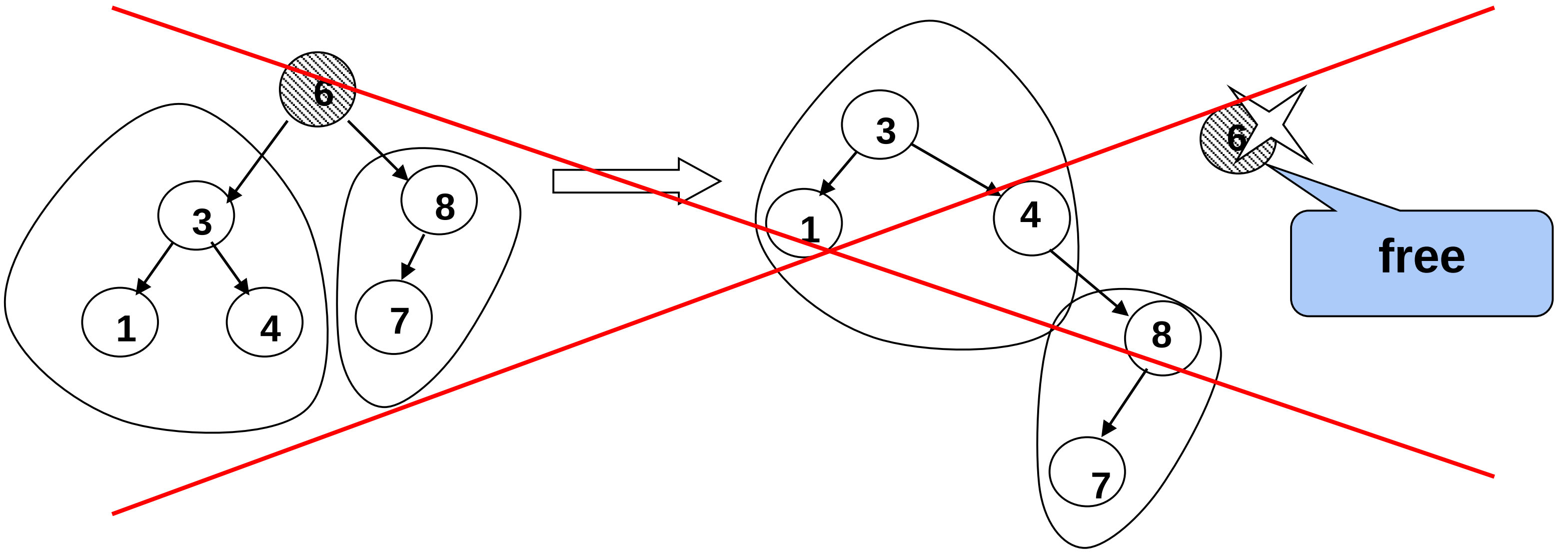
BVS Rušení uzlu – operace Delete

- Rušení uzlu, který má pouze jednoho syna:



BVS Rušení uzlu – operace Delete

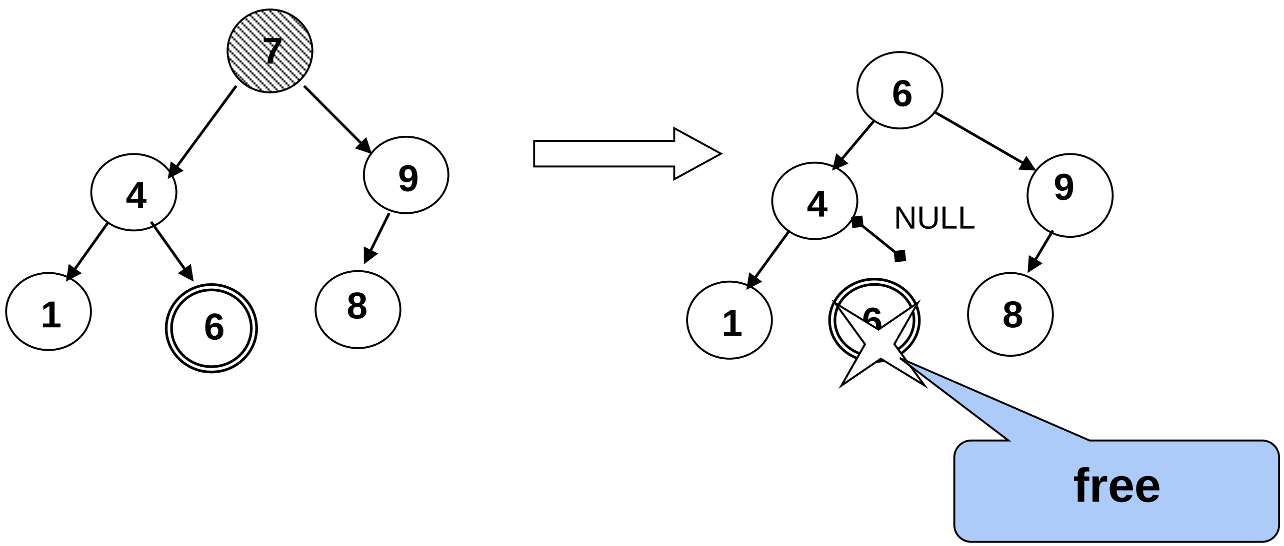
- Rušení uzlu se dvěma syny – kam připojit oba syny?
- Špatné řešení:
  - levý podstrom rušeného uzlu připojíme na nejlevější uzel pravého podstromu,
  - nebo pravý podstrom rušeného uzlu připojíme na nejpravější uzel levého podstromu (viz obrázek níže).
  - zvyšuje výšku stromu, a tím prodlužuje maximální dobu vyhledávání.

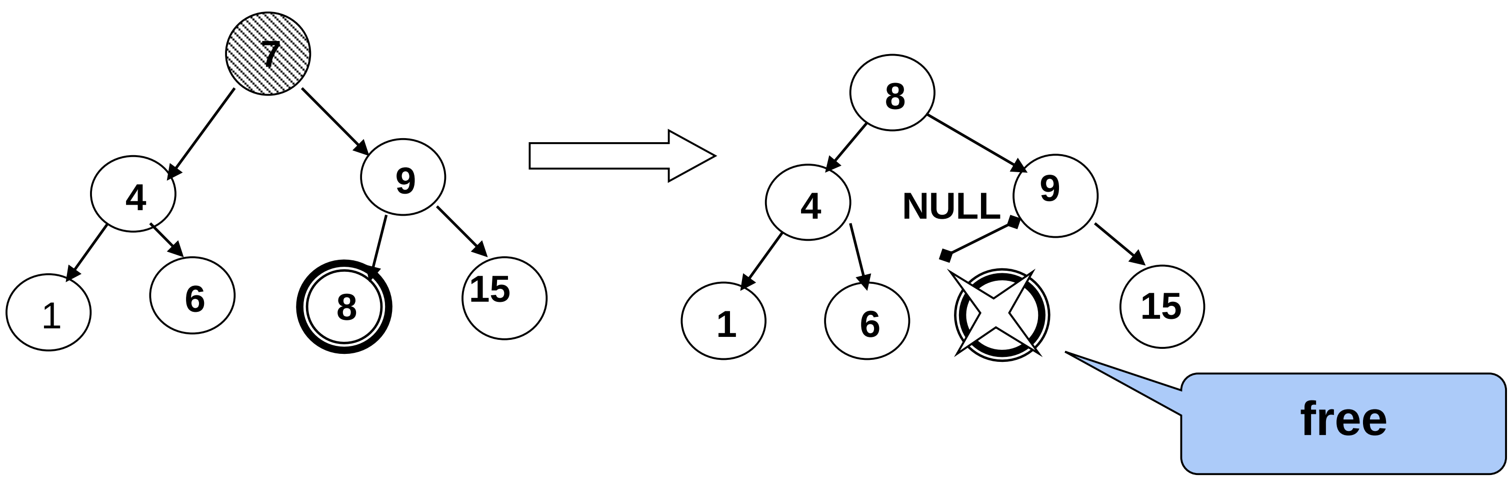


BVS Rušení uzlu – operace Delete

- Rušení uzlu se dvěma syny – kam připojit oba syny?
- **Správné řešení:**
  - uzel nezrušíme fyzicky, ale přepíšeme hodnotou takového uzlu, který lze zrušit snadno, a při přepisu nedojde k porušení uspořádání BVS
  - Vhodný uzel:
    - nejpravější uzel levého podstromu rušeného uzlu (maximum v levém podstromu) nebo
    - nejlevější uzel pravého podstromu rušeného uzlu (minimum v pravém podstromu).

BVS – rušení uzlu se dvěma syny





BVS – Delete (rekurzivní verze)

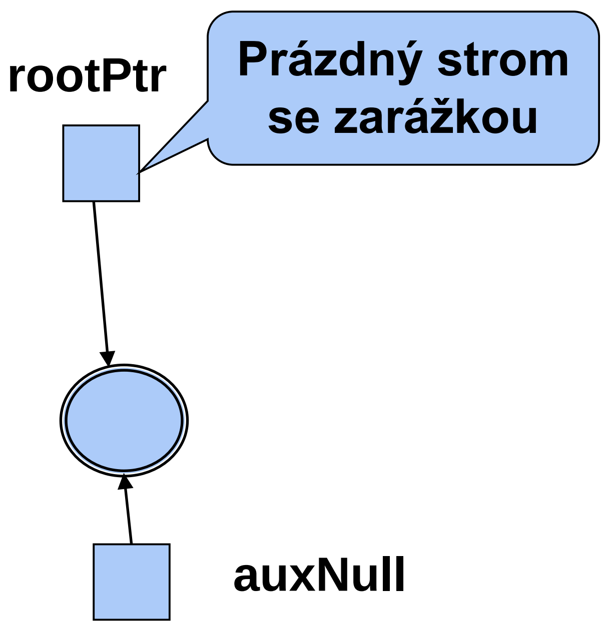
- Použijeme pomocnou funkci **BVSMin**, která nalezne uzel, jehož hodnotou lze přepsat rušený uzel – varianta vracející nejlevější uzel v daném podstromu:

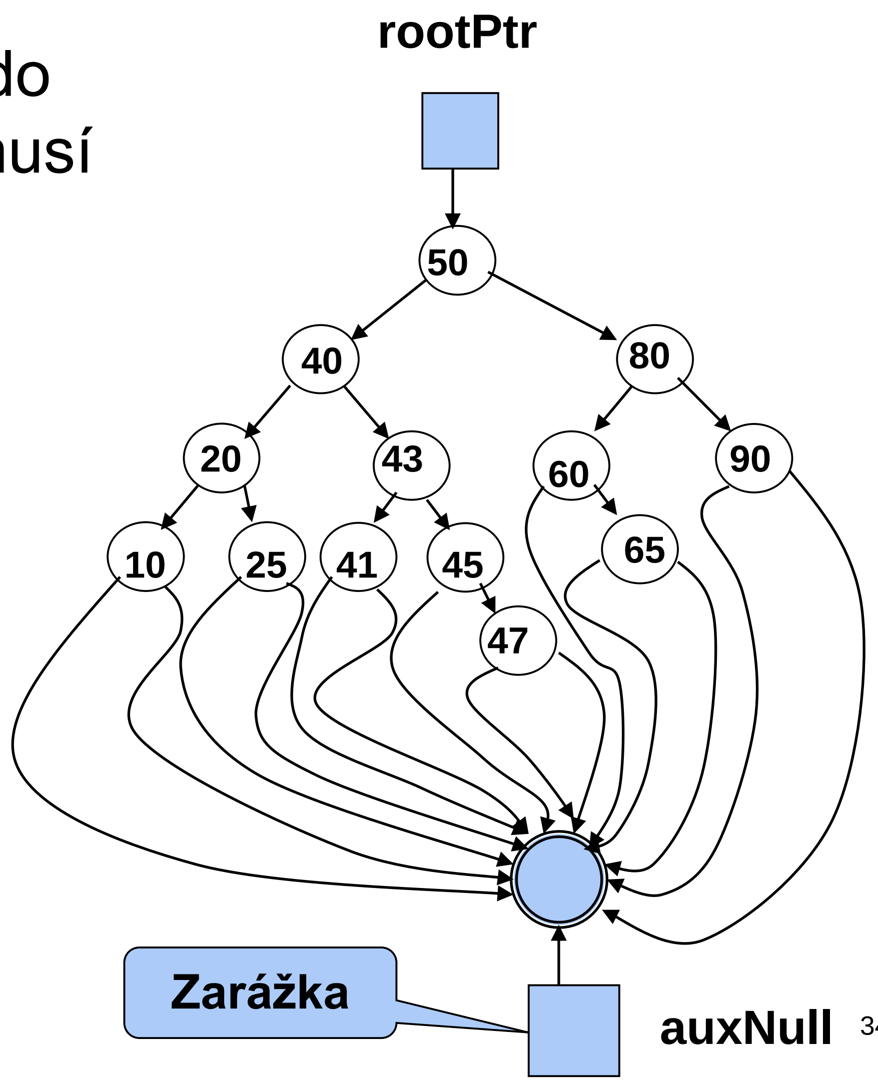
```
TNode* function BVSMin (TNode *rootPtr)
// funkce vrátí ukazatel na nejlevější uzel v daném
// neprázdném(!) stromu
  if rootPtr->lPtr = NULL:     // další levý už neexistuje
    return rootPtr
  else:                        // pokračujeme vlevo
    return BVSMin(rootPtr->lPtr)
```

```
TNode* function BVSDelete (TNode *rootPtr, int k)
 if rootPtr = NULL:                          // prázdný (pod)strom
   return NULL
 else:
   if k < rootPtr->key:             // rušený klíč je v levém podstromu
     rootPtr->lPtr ← BVSDelete(rootPtr->lPtr,k)
     return rootPtr
   else:
     if rootPtr->key < k:           // rušený klíč je v pravém podstromu
       rootPtr->rPtr ← BVSDelete(rootPtr->rPtr,k)
       return rootPtr
     else:                          // nalezen uzel s daným klíčem
       if (rootPtr->lPtr = NULL) and (rootPtr->rPtr = NULL):
           free(rootPtr)                 // rušený nemá žádného syna
           return NULL
       else:
         if (rootPtr->lPtr ≠ NULL) and (rootPtr->rPtr ≠ NULL):
                                        // rušený má oba podstromy
           TNode *min ← BVSMin(rootPtr->rPtr) // najdi minimum
           rootPtr->key ←  min->key                // nahraď
           rootPtr->data ← min->data
           rootPtr->rPtr ← BVSDelete(rootPtr->rPtr,min->key)
           return rootPtr
         else:                          // rušený má pouze jeden podstrom
           if rootPtr->lPtr = NULL:  // rušený nemá levého syna
             TNode *onlyChild ← rootPtr->rPtr
           else:                           // rušený nemá pravého syna
             TNode *onlyChild ← rootPtr->lPtr
           free(rootPtr)
           return onlyChild
```

BVS se zarážkou

- Před vyhledáváním se do zarážky vloží klíč a nemusí se kontrolovat konec.





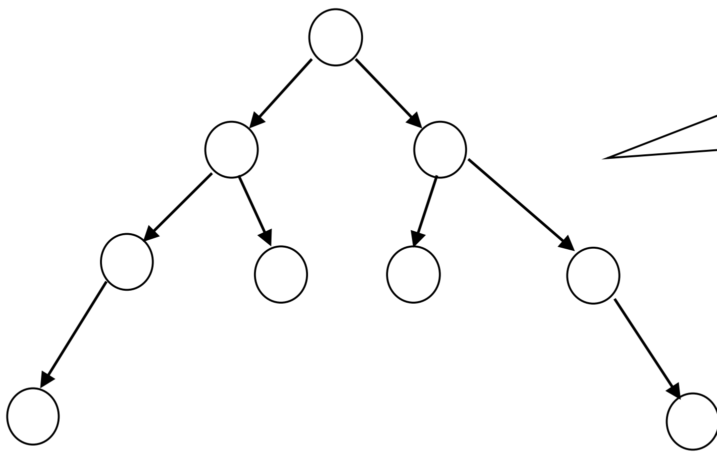
K procvičení:

- Implementujte operaci **Search** v BVS se zarážkou.
- Je dán nevyvážený BVS a je zadán (maximální) počet jeho uzlů. Vytvořte jeho váhově vyváženou verzi. Řešení (včetně definice typů) zapište ve formě rekurzivní i nerekurzivní funkce.
  Nápověda: Do pomocného pole vložíte všechny hodnoty nevyváženého stromu a z pole pak vytvoříte nový, váhově vyvážený strom.
  Obtížnější varianta: Počet uzlů stromu zadán není, je třeba jej nejdříve spočítat.

AVL stromy

- Výškově vyvážený strom – AVL podle ruských matematiků Adělson-Velski a Landis.
- Je maximálně o 45 % vyšší než váhově vyvážený strom.
- Výškově vyvážený binární vyhledávací strom je strom, pro jehož každý uzel platí, že výška jeho dvou podstromů je stejná nebo se liší o 1.
- Znovuustavení vyváženosti binárních stromů po operacích Insert nebo Delete:
  - U váhově vyvážených stromů obtížné.
  - U výškově vyvážených stromů lze provést rekonfiguraci uzlů (rotací) v okolí tzv. kritického uzlu.

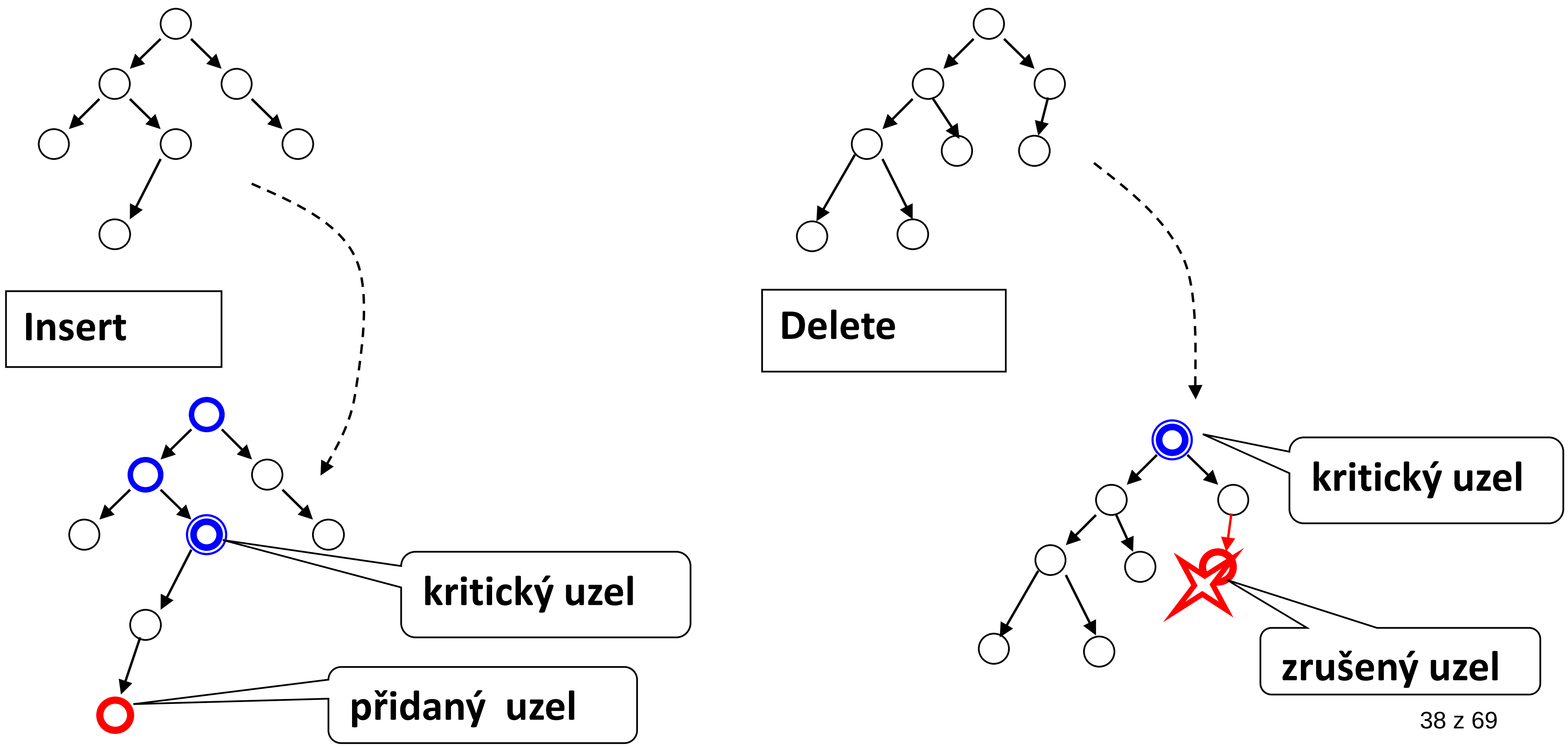
Výškově vyvážený strom



Příklad výškově vyváženého stromu

AVL stromy

- Kritický uzel – nejvzdálenější uzel od kořene, v němž je v důsledku vkládání nebo rušení porušená rovnováha.
- Příklady porušení rovnováhy operacemi Insert a Delete:



AVL stromy

- Každému uzlu přiřadíme váhu takto:
  - 0: zcela vyvážený uzel
  - -1: výška levého podstromu je o jedna větší
  - 1: výška pravého podstromu je o jedna větší
- Pokud v rámci operace Insert nebo Delete dojde ke změně váhy na hodnotu -2/2, je potřeba situaci napravit.
- Mohou nastat 4 různé situace, které se napravují různými způsoby:
  - LL: kritický uzel je příliš těžký vlevo a jeho levý syn je těžký vlevo
  - LR: kritický uzel je příliš těžký vlevo a jeho levý syn je těžký vpravo
  - RR: kritický uzel je příliš těžký vpravo a jeho pravý syn je těžký vpravo
  - RL: kritický uzel je příliš těžký vpravo a jeho pravý syn je těžký vlevo

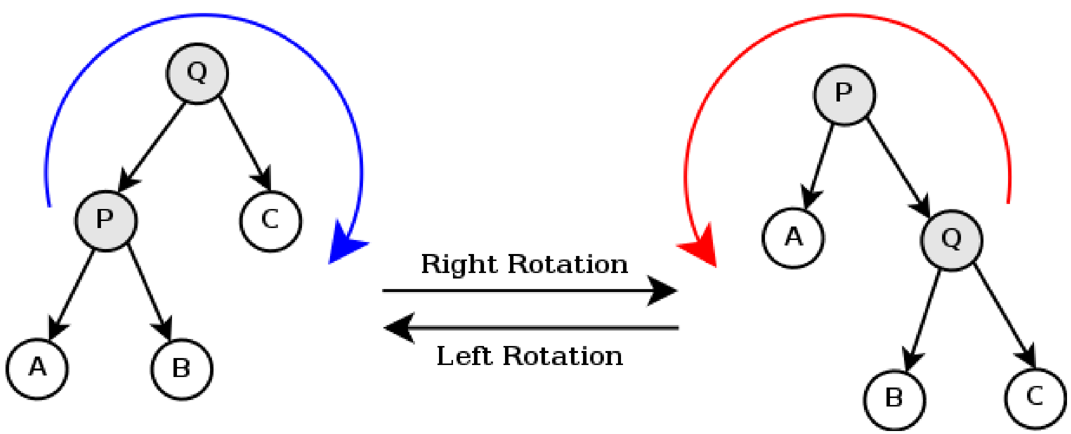
AVL stromy

- Po operaci Delete mohou nastat 2 další situace:
  - kritický uzel je příliš těžký vlevo a jeho levý syn je vyvážený – obdoba situace LL, řeší se stejným způsobem.
  - Kritický uzel je příliš těžký vpravo a jeho pravý syn je vyvážený – obdoba situace RR, řeší se stejným způsobem.

- *Pozn.:* Při operaci Insert nás zajímá váha toho syna kritického uzlu, v jehož podstromu došlo k vložení uzlu. Při operaci Delete nás zajímá váha opačného syna než toho, v jehož podstromu došlo k odstranění uzlu.

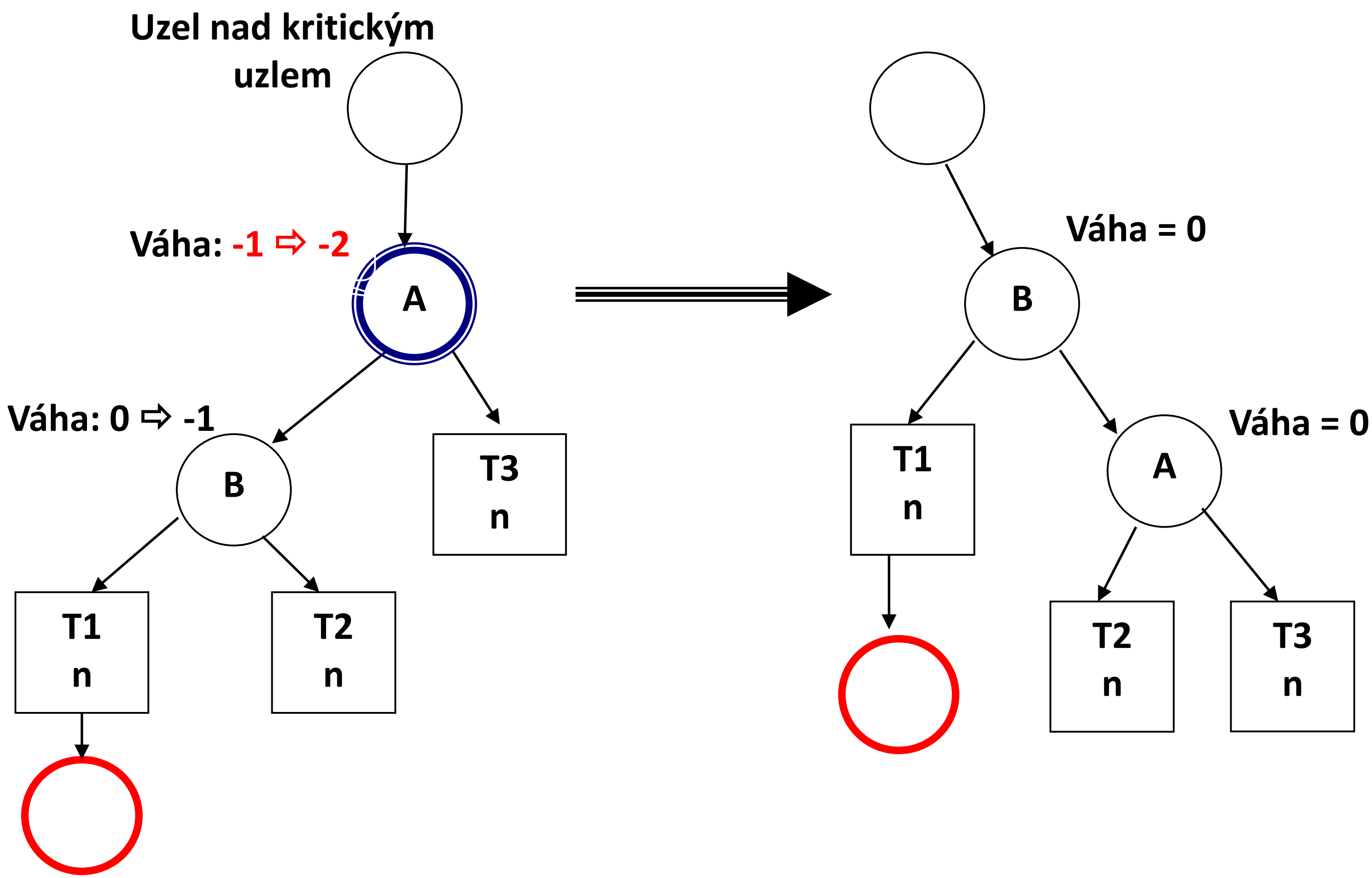
AVL stromy

- Situaci LL opravíme pravou rotací
- Situaci LR opravíme dvojitou rotací – levá rotace následovaná pravou rotací
- Situaci RR opravíme levou rotací
- Situaci RL opravíme dvojitou rotací – pravá rotace následovaná levou rotací



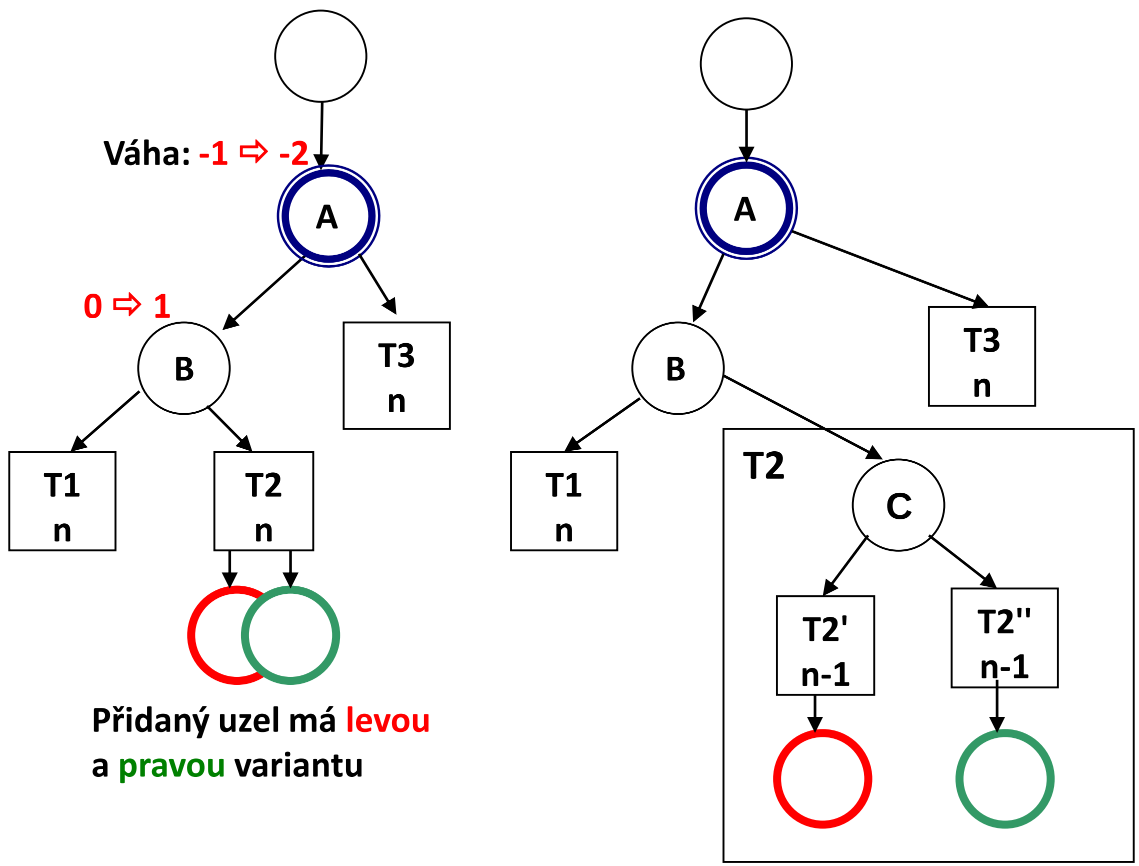
Situace LL po operaci Insert

- Opravíme jednoduchou rotací vpravo

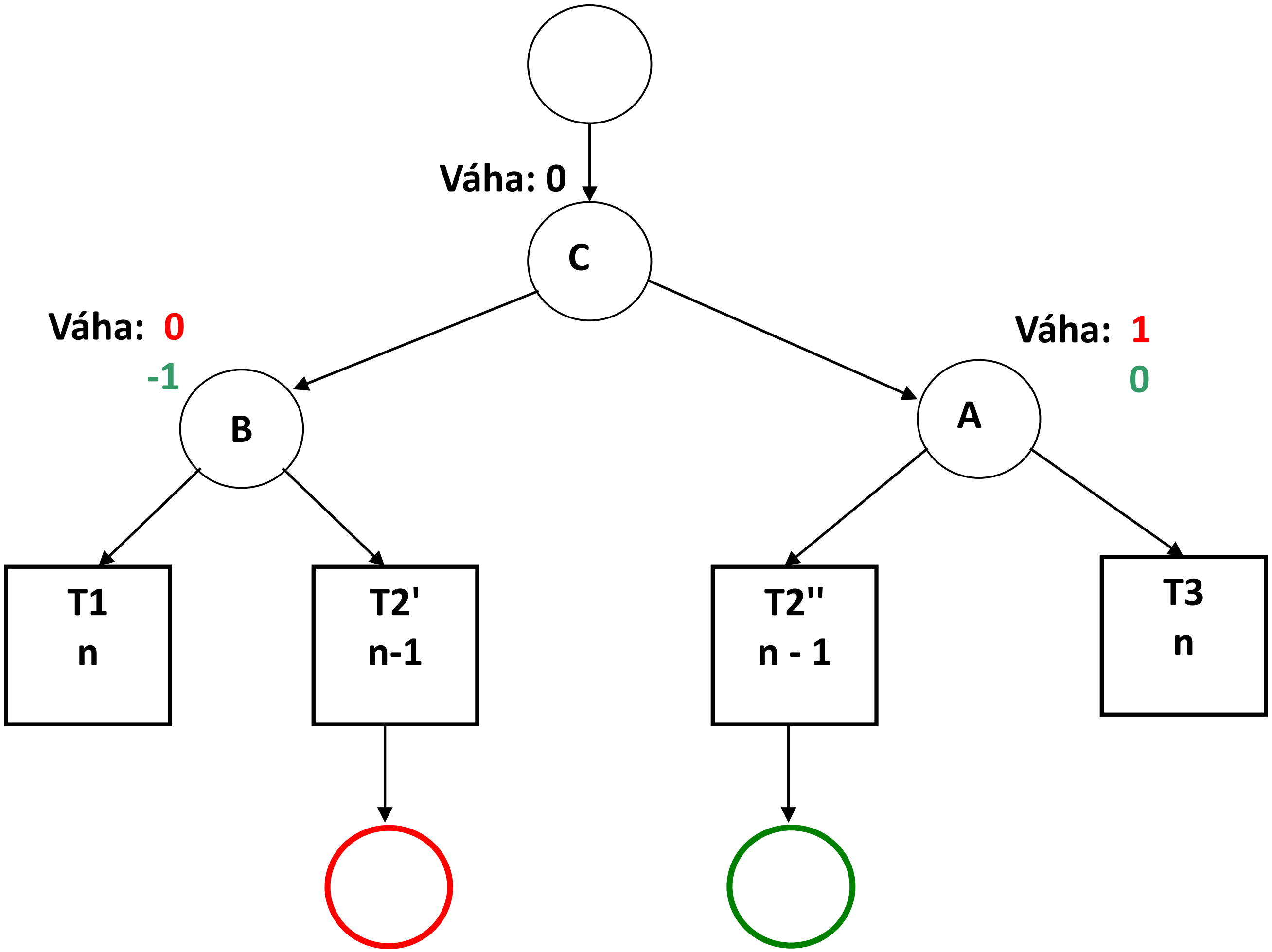


Situace LR po operaci Insert

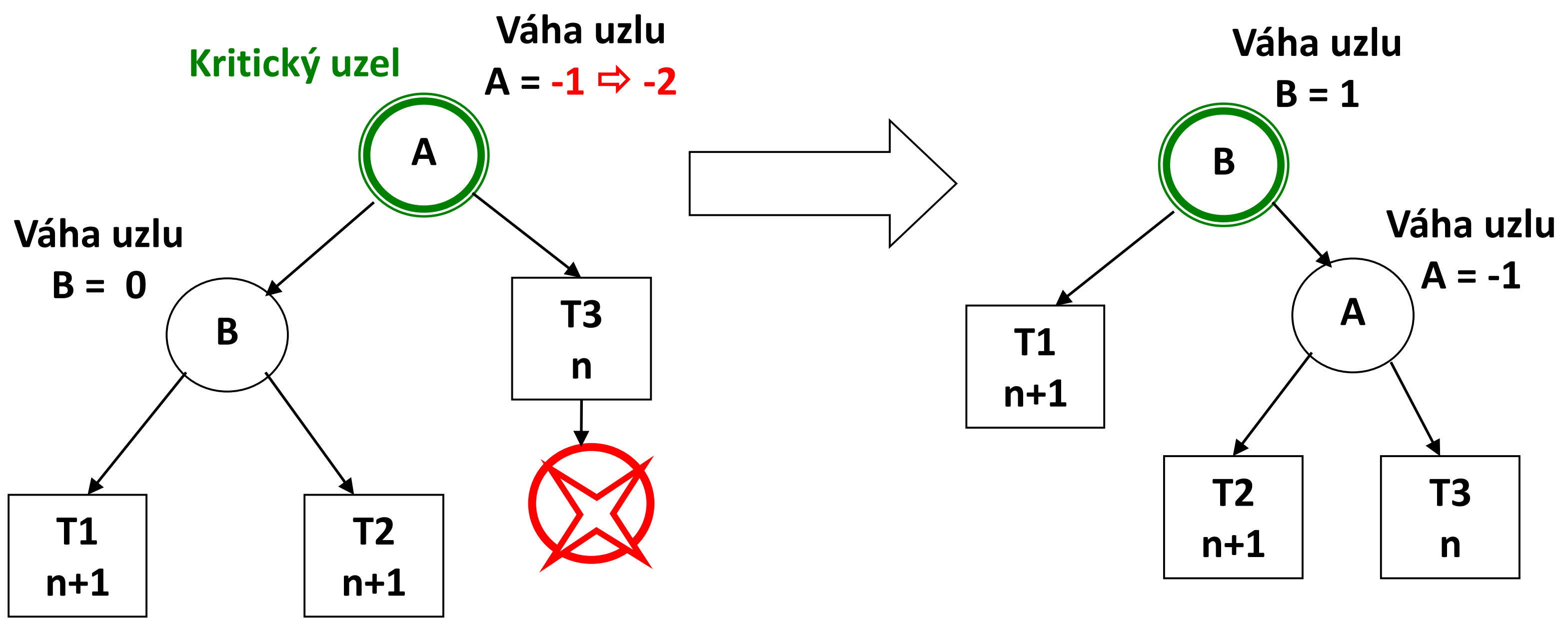
- Opravíme dvojitou rotací – vlevo a vpravo
- Konfiguraci lze překreslit do tvaru uvedeného vpravo.



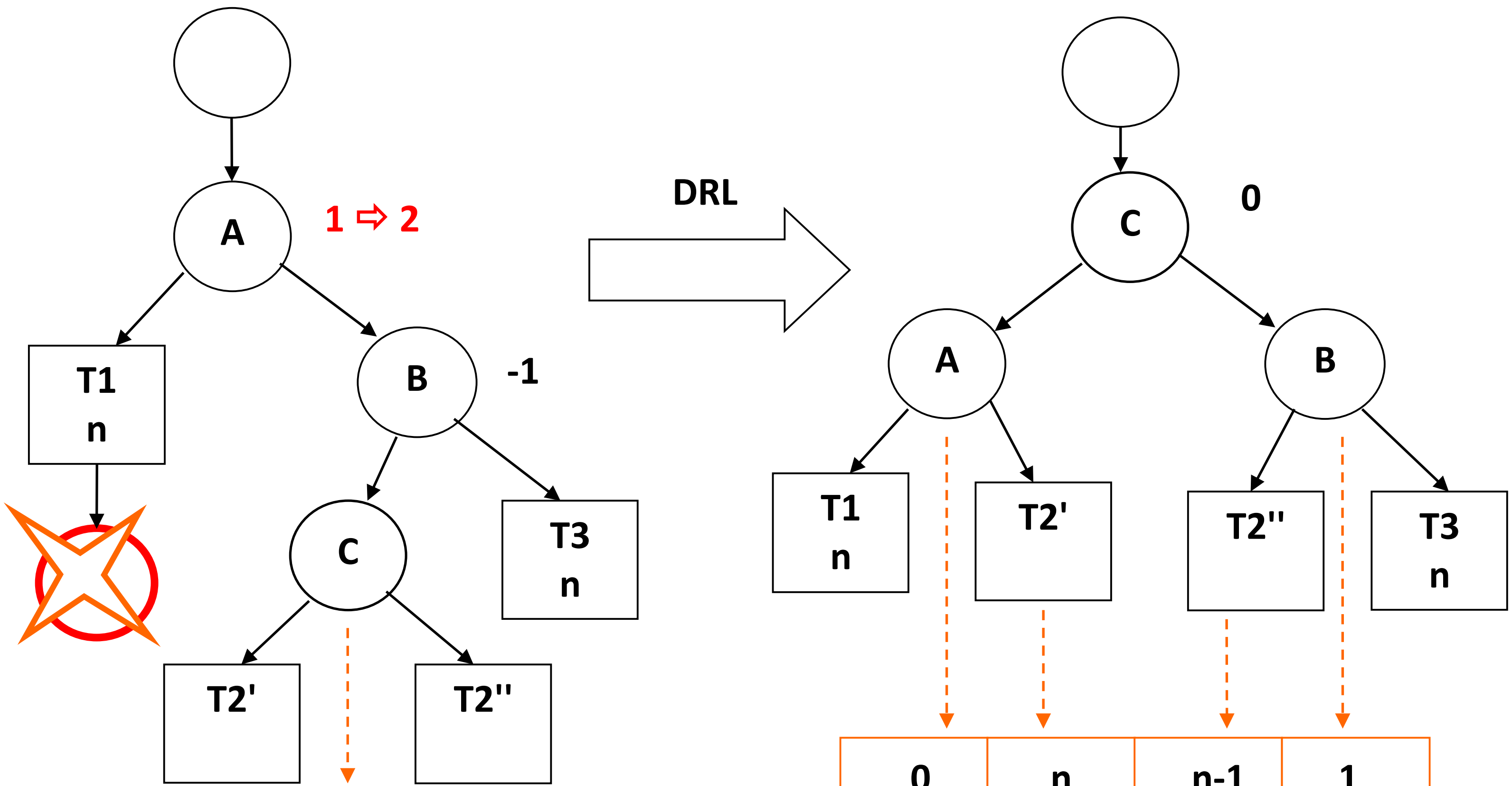
Situace LR po dvojité rotaci



Situace LL po operaci Delete



Situace RL po operaci Delete



n	-1	n-1
n-1	1	n
n	0	n

0	n	n-1	1
-1	n-1	n	0
0	n	n	0

AVL stromy – implementace

- Po operaci Insert/Delete je potřeba šířit informaci o změně výšky některého podstromu (pokud k ní dochází) směrem ke kořeni.
- Pokud je v některém uzlu porušena výšková vyváženost (kritický uzel):
  - Vyhodnotíme situaci.
  - Provedeme příslušnou akci k nápravě.
  - V případě potřeby (pokud dojde ke změně) šíříme dále informaci o změně výšky celého podstromu (včetně kritické uzlu).
  - *Pozn.:* Snazší ale méně efektivní řešení: žádnou informaci o změně výšky nešíříme, ale kontrolujeme výšku obou podstromů ve všech uzlech směrem ke kořeni.

AVL stromy – implementace

- Šíření informace a případnou nápravu snadno zajistíme při využití rekurze:
  - Při hledání místa pro vložení nového uzlu procházíme stromem směrem dolů.
  - Po vložení se ukončují jednotlivá volání funkce Insert – než se ukončí můžeme (při návratu z rekurze):
    - Správně nastavit novou váhu daného uzlu
    - Zkontrolovat výškovou vyváženost
    - Případně provést potřebnou rotaci.

AVL stromy – implementace

```
TNode* function Insert (TNode *rootPtr, TKey k, TData d)
  if rootPtr = NULL:                        // vytvoření nového uzlu
    return CreateNode(k,d)
  else:
    if k < rootPtr->key:                           // jdeme vlevo
      rootPtr->lPtr ← Insert(rootPtr->lPtr,k,d)
    else:
      if rootPtr->key < k:                         // jdeme vpravo
        rootPtr->rPtr ← Insert(rootPtr->rPtr,k,d)
      else:                    // přepíšeme stará data novými
        rootPtr->data ← d
    return rootPtr
```

Zde provedeme to, co je třeba provést před návratem z rekurze

Pravá rotace – implementace

```
TAVLNode * rightRotate (TAVLNode *y)
{ // TAVLNode obsahuje proměnou height typu int
    TAVLNode *x = y->lPtr;
    y->lPtr = x->rPtr;
    x->rPtr = y;
    // oprava výšek prohozených uzlů
    y->height = max(heightN(y->lPtr), heightN(y->rPtr))+1;
    x->height = max(heightN(x->lPtr), heightN(x->rPtr))+1;
    return x;
}
```

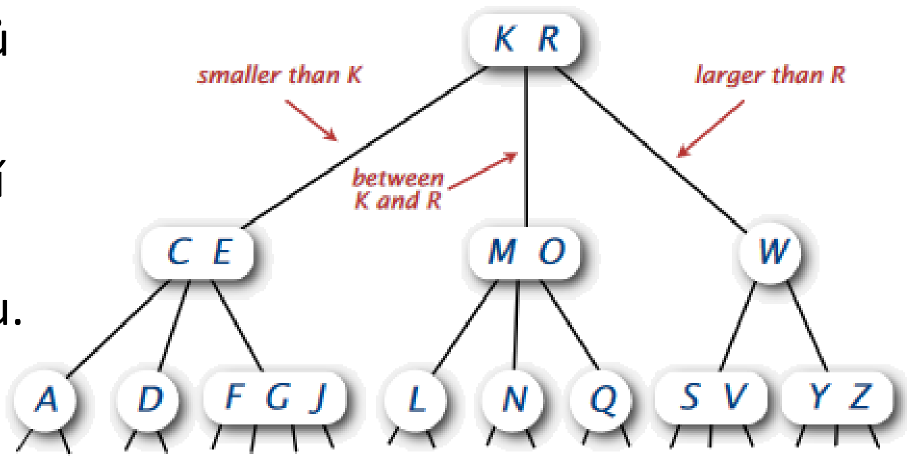
Pozn.: Funkce `heightN()` zde vrací hodnotu `height` uloženou v daném uzlu, pokud tento uzel existuje. Pokud neexistuje vrací hodnotu 0.

Vyhledávací strom

- *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.
- *Vnější (externí) vrcholy* neobsahují žádná data a nemají žádné potomky. Jsou to tedy listy stromu.

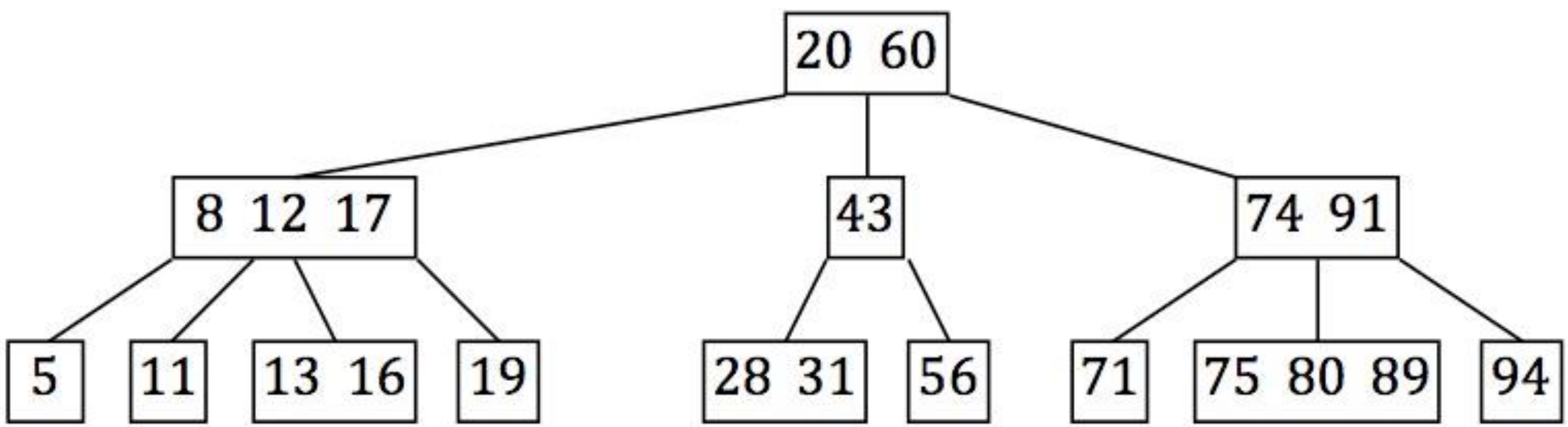


(a,b)-stromy

(a,b)-strom pro parametry a ≥ 2, b ≥ 2a-1 je obecný vyhledávací strom, pro který navíc platí:

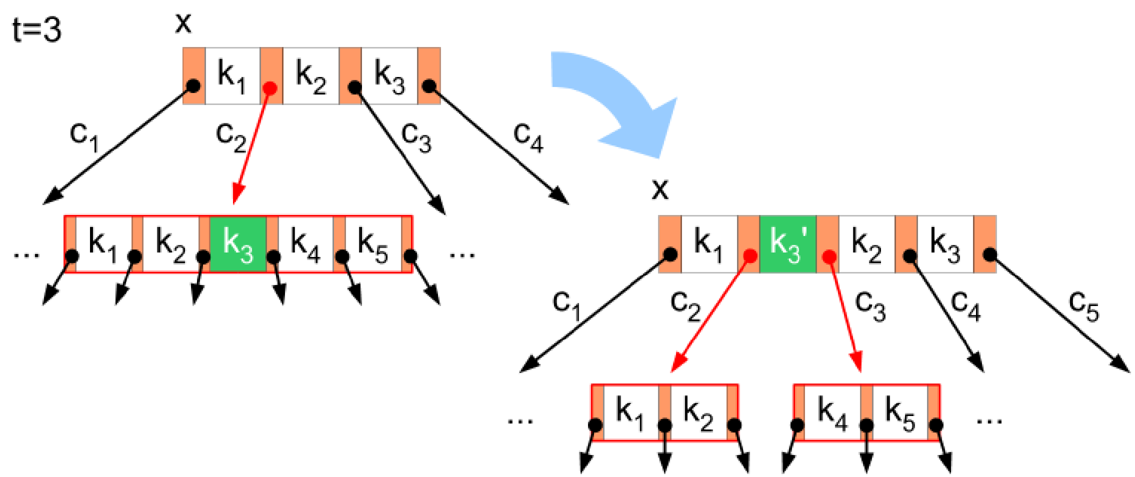
1. Kořen má 2 až b synů, ostatní vnitřní vrcholy a až b synů.
2. Všechny vnější vrcholy jsou ve stejné hloubce.

(a,b)-strom s *n* klíči má hloubku Θ(log n).



Vkládání do (a,b)-stromu

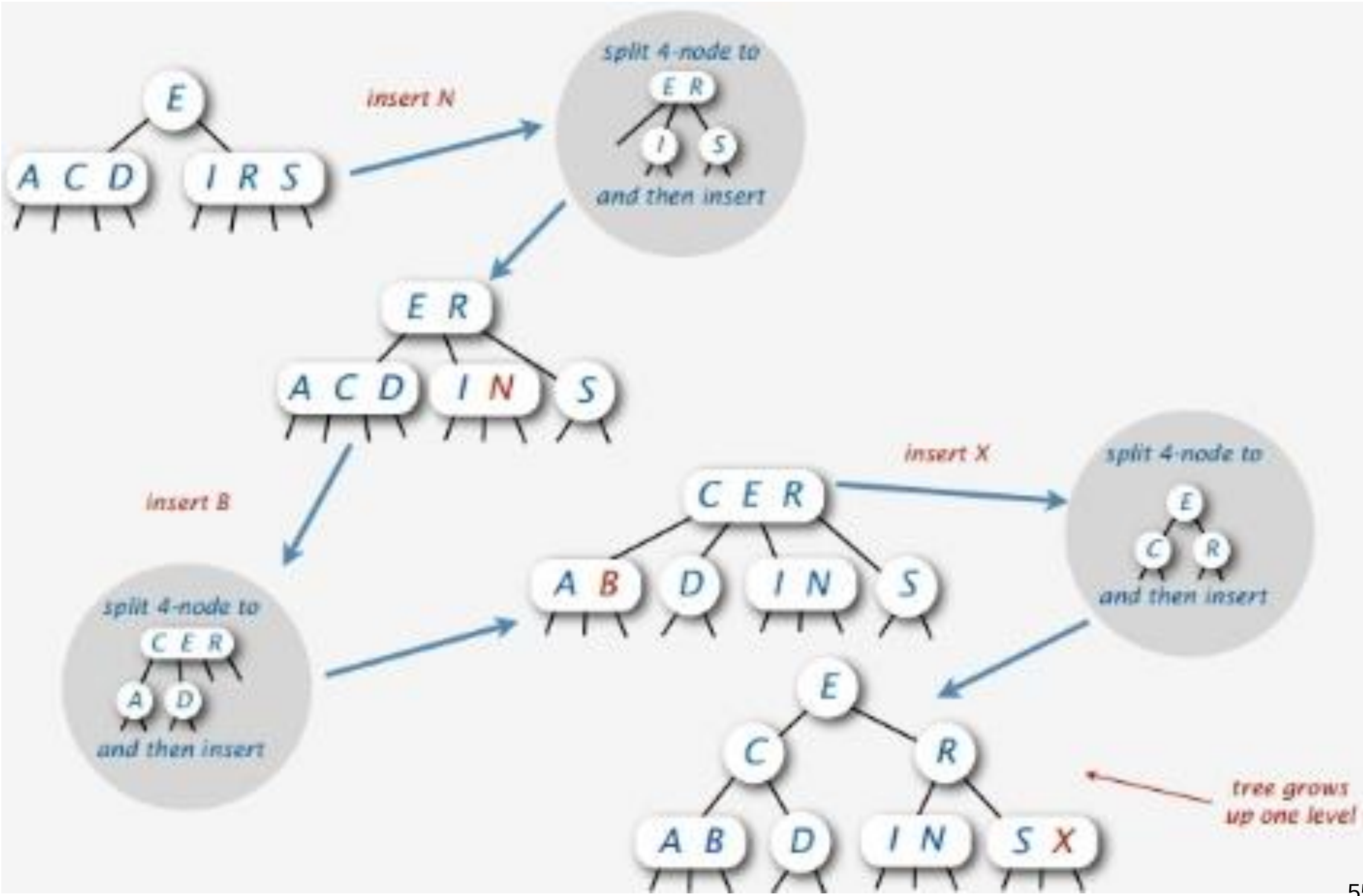
- Nevkládáme nový list (porušení pravidla o stejné hloubce vnějších uzlů)
- Jde-li vložit další klíč do příslušného uzlu na nejnižší hladině, aniž by došlo k přeplnění uzlu – vložíme.
- Pokud by mělo dojít k přeplnění uzlu, uzel rozštěpíme, prostřední klíč vložíme do nadřazeného uzlu (abychom mohli připojit 1 syna navíc) a zbývající klíče přiřadíme do nových vrcholů.
- Přidáním klíče do nadřazeného vrcholu posuneme problém štěpení uzlu o úroveň výš.
- Bude-li potřeba rozštěpit kořen, vytvoříme nový kořen s jediným klíčem a celý strom se o hladinu prohloubí.



Vkládání do (a,b)-stromu

- Varianta: zcela naplněné uzly jsou štěpeny už cestou dolů stromem, při vyhledávání místa, kam má být nový prvek vložen.

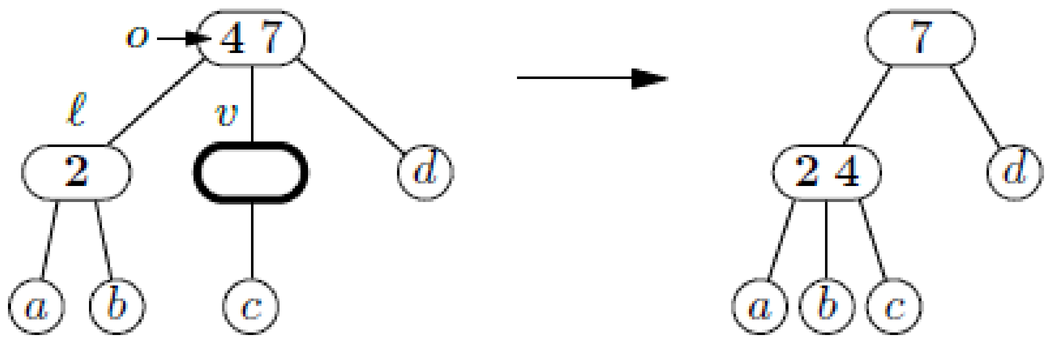
Vkládání do (2,4)-stromu

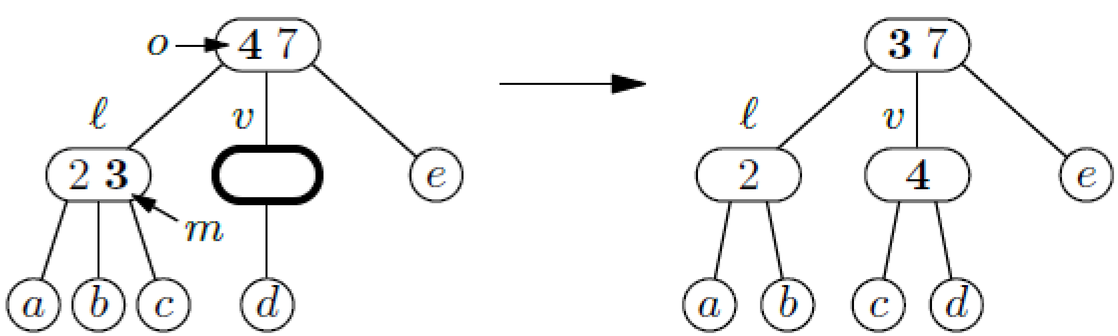


Mazání v (a,b)-stromu

- Klíč na nejnižší hladině lze smazat přímo, ale nesmí vzniknout uzel s nedostatečným počtem synů.
- Klíče na vyšších hladinách nelze smazat přímo, nahradíme jejich hodnotu např. nejnižším klíčem z nejlevějšího vrcholu pravého podstromu a ten potom smažeme.
- Řešení nedostatečného počtu synů s využitím bratra:
  - Má-li bratr (lze vybrat levého i pravého) pouze *a* synů, sloučíme podměrečný uzel s bratrem a doplníme uzel klíčem z otce (možný problém nedostatku synů se přesune na otce).
  - Má-li bratr více než *a* synů, odpojíme od něj nejpravějšího syna *c* a největší klíč *m*. Klíč *m* přesuneme do otce, z otce příslušný klíč přesuneme do podměrečného uzlu a před něj připojíme syna *c*.
- Pokud zmizí z kořene všechny klíče, je kořen smazán, čímž se sníží výška stromu.

Mazání ve (2,3)-stromu





Shrnutí (a,b)-stromy

- Časová složitost: $\Theta(\log n)$ (délka všech cest od kořene k listům je stejná)
- Doporučení: nevolit b výrazně větší než je dolní mez (2a-1)
- Obvykle se používají (a, 2a-1) nebo (a,2a)-stromy, časté parametry: (2,3) nebo (2,4)
- Varianta: data jsou uložena pouze na nejnižší hladině, vnitřní hladiny obsahují pouze pomocné klíče (často minima z podstromů)
- B-stromy: varianta optimalizovaná pro práci s velkými bloky dat.

LLRB stromy

- *Left-leaning red-black trees*
- Varianta červeno-černých stromů (RB stromů)
- *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.
- LLRB strom – překlad (2,4) stromu na BVS s logaritmickou hloubkou a možností vyvažování

Překlad (2,4)-stromu na LLRB

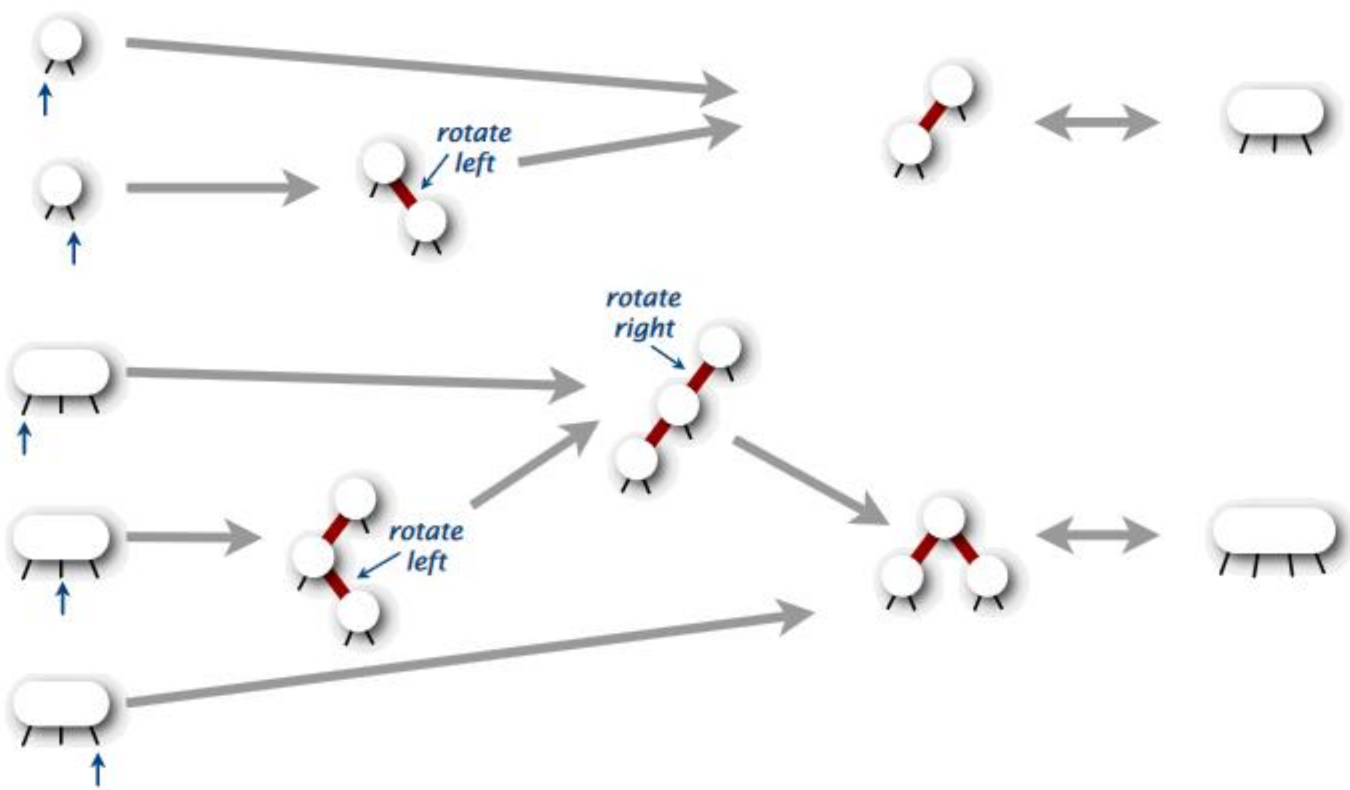
- Každý vrchol (2,4)-stromu nahradíme konfigurací jednoho nebo více binárních vrcholů
- Pro zachování korespondence mezi stromy zavedeme 2 barvy hran:
  - Červené hrany – spojují vrcholy tvořící 1 konfiguraci
  - Černé hrany – hrany mezi konfiguracemi (hrany původního stromu)
- Barvu hrany si můžeme pamatovat v jejím spodním vrcholu
- Vrcholy označujeme dle počtu synů jako 2-vrchol, 3-vrchol, 4-vrchol

- Transformace 3-vrcholu – nahradíme 2 vrcholy a červená hrana musí vždy vést doleva

Překlad (2,4)-stromu na LLRB

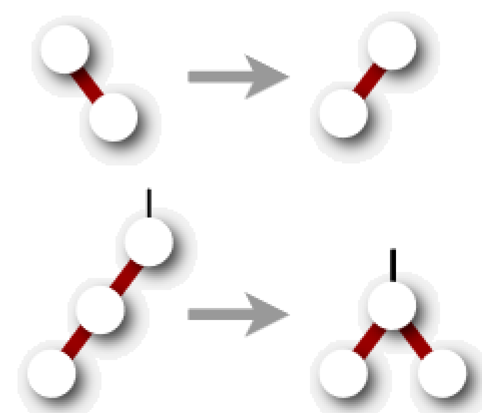
Vkládání v LLRB

- Vyváženost stromu je udržována rotacemi, a to jen **červených** hran.
- Nový uzel vkládáme na nejnižší hladinu, připojujeme ke stromu pomocí červené hrany a v případě potřeby (červená hrana vedoucí doprava nebo 2 červené hrany nad sebou) rotujeme.



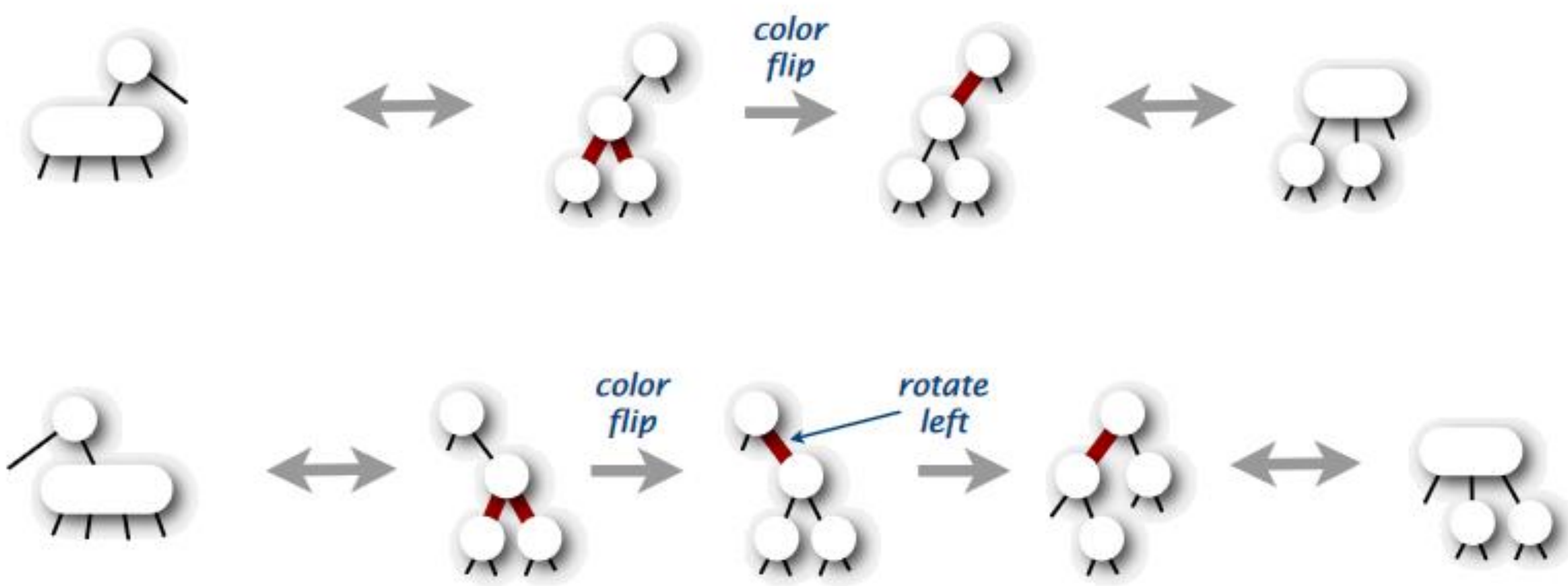
Vkládání v LLRB

- Při cestě stromem dolů štěpíme zcela zaplněné uzly (4-vrcholy)
- Štěpení je realizováno pomocí přebarvení – tím se uzel rozštěpí a prostřední klíč se stane součástí nadřazeného vrcholu (víme jistě, že se tam vleze, protože všechny 4-vrcholy rovnou štěpíme).
- Na nejnižší úrovni vložíme uzel.
- Štěpení může zanechat ve stromu špatné konfigurace červených hran (červená hrana vedoucí doprava, nebo 2 červené hrany nad sebou) – opravujeme pomocí rotací při cestě stromem zpět ke kořeni (jednoduché při využití rekurze).



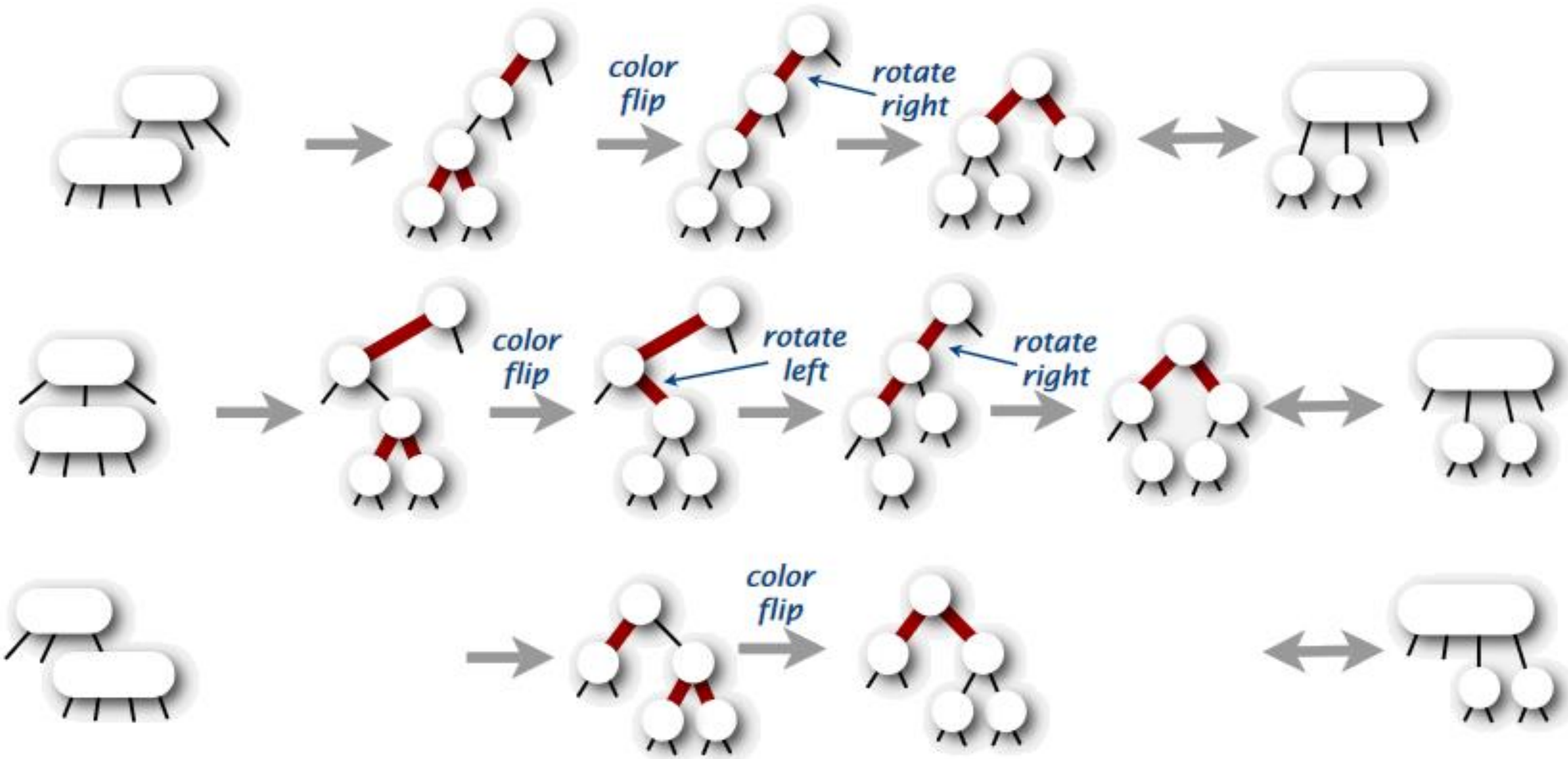
Vkládání v LLRB

- Jak přebarvení realizuje štěpení uzlu?



Vkládání v LLRB

- Jak přebarvení realizuje štěpení uzlu?

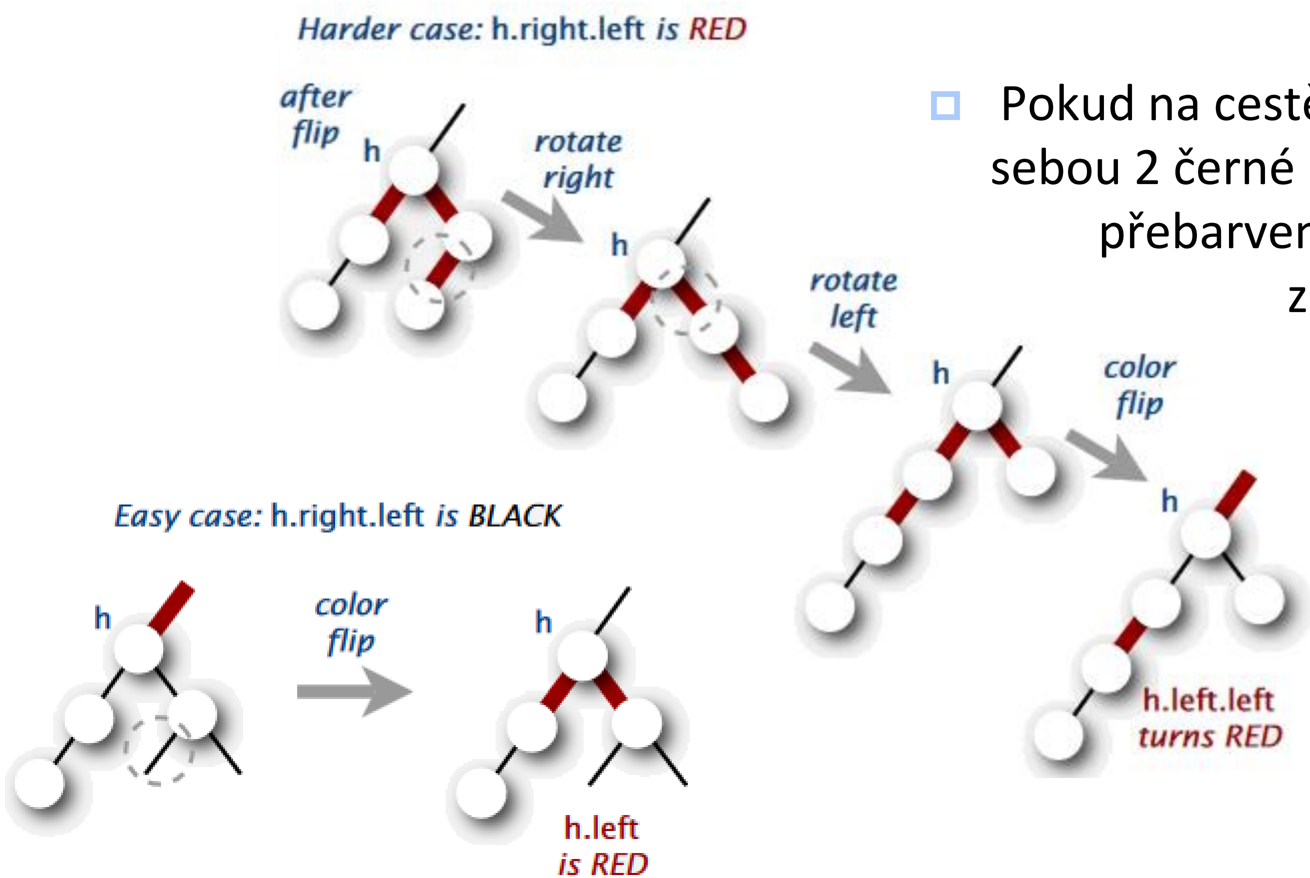


Mazání v LLRB

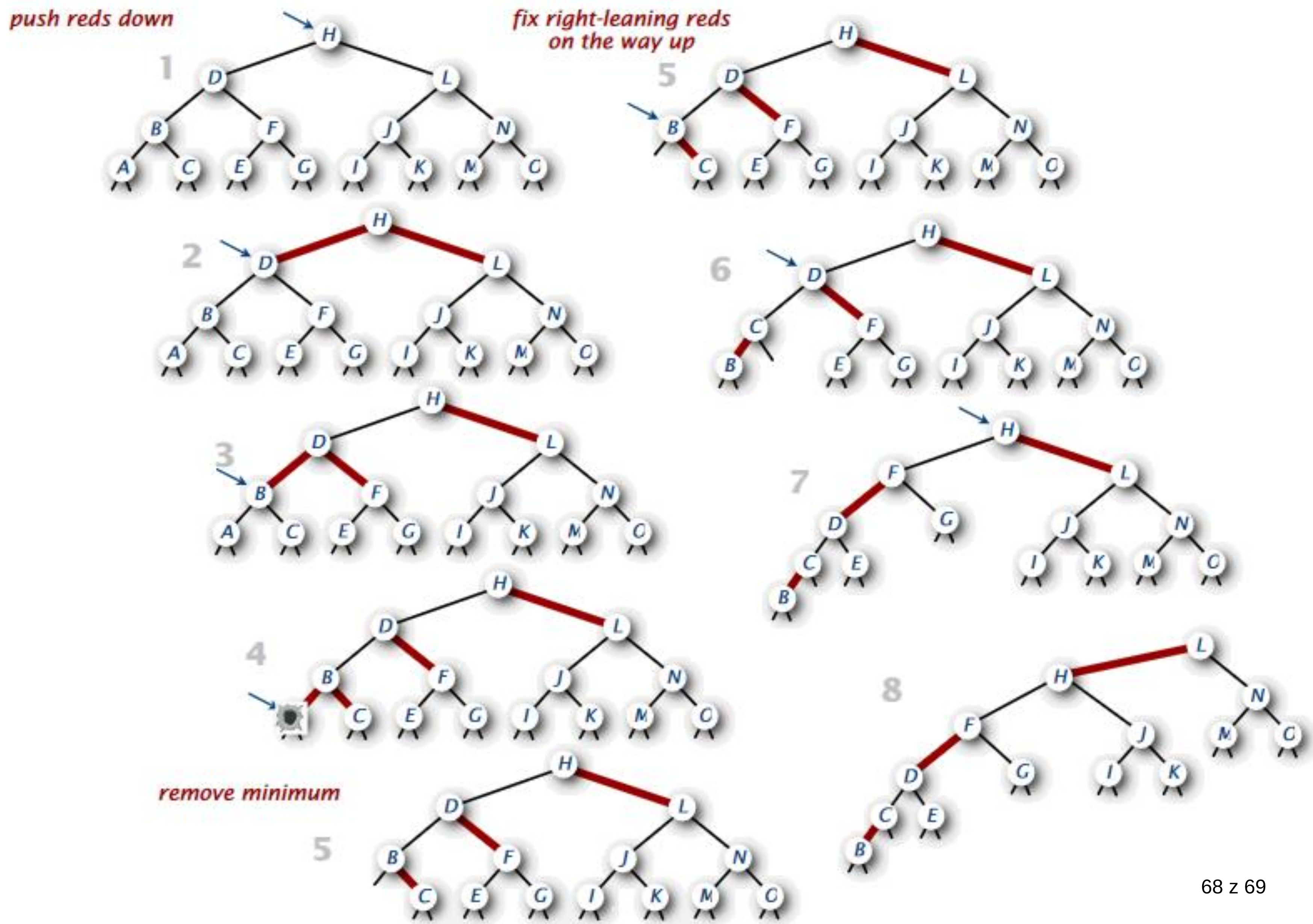
- Mazání vnitřních uzlů se opět řeší náhradou hodnoty z vhodného uzlu na nejnižší hladině a smazáním tohoto uzlu – tedy mažeme buďto minimum z pravého podstromu nebo maximum z levého podstromu.
- Mazání minima: pokud do uzlu na nejnižší hladině vede červená hrana, lze smazat přímo (odpovídá to mazání klíče z 3-vrcholu).
- Problém: pokud v okolí vrcholu není žádná červená hrana (mazání klíče z 2-vrcholu – uzel by si musel půjčit klíč od souseda nebo se s ním spojit).
- Řešení: cestou stromem dolů provádíme úpravy tak, aby aktuální uzel nebyl 2-vrchol.
- Pomocí úprav mohou vzniknout nekorektní 3-vrcholy nebo 4-vrcholy – ty jsou upraveny při návratu z rekurze (cestou stromem zpět ke kořeni).

Mazání v LLRB

- Aby aktuální uzel nebyl 2-vrchol, dodržujeme cestou dolů tento invariant: Stojíme-li ve vrcholu *v*, pak vede červená hrana buďto shora do *v* nebo z *v* do jeho levého syna. Výjimku dovolujeme pro kořen.



- Pokud na cestě doleva jsou pod sebou 2 černé hrany, použijeme přebarvení a pomocí rotací zkorigujeme okolí.


push reds down
fix right-leaning reds
on the way up
remove minimum
1
2
3
4
5
5
6
7
8