Binární vyhledávací stromy II

DSA - Přednáška 10

Josef Kolář

O čem bude řeč?

- randomizované BVS
- vyvažování BVS
  - AVL
  - červeno-černé BVS
- B-stromy

Randomizovaný BVS

Náhodný BVS vznikne tehdy, když se vytváří z náhodné posloupnosti hodnot klíčů. V tom případě může být kořenem libovolný uzel stromu, totéž platí pro podstromy na všech úrovních. Posloupnosti klíčů ale **nejsou náhodné.**

- náhodnosti lze dosáhnout bez jakéhokoliv předpokladu o výchozí posloupnosti
- při vkládání nového uzlu do BVS o N uzlech uložíme uzel do kořene s pravděpodobností 1 / (N+1)
- pokud nebyl umístěn do kořene, pokračujeme podle hodnoty klíče vložením do levého nebo pravého podstromu, opět s příslušnou pravděpodobností

Randomizovaný BVS - insert

```
void insert ( Elem x )
  { head = insertRec (head, x); }
private Node insertRec ( Node h, Elem x )
  { if (h == null) return new Node(x);
    if (Math.random()*h.cnt < 1.0) return insertRoot(h, x);
    if (x.key < h.item.key)
         h.left = insertRec(h.left, x);
    else h.right = insertRec(h.right, x);
    h.cnt++;
    return h;
  }
```

- cena vytvoření randomizovaného BVS s N uzly je průměrně 2N ln N srovnání
- operace hledání v randomizovaném BVS potřebuje průměrně 2 . ln N srovnání
- pravděpodobnost, že cena vytvoření randomizovaného BVS je α-krát větší než průměrná, je menší než $e^{-\alpha}$.

Randomizovaný BVS - join

- operaci join koncipujeme stejně jako dříve s tím, že místo libovolného určení uzlu v kořeni zajistíme, aby všechny uzly měly stejnou pravděpodobnost se stát kořenem
- předpokládáme úpravy hodnot složky cnt v operacích rotace

```
void join ( Node b )
  { int N = head.cnt;
    if (Math.random()*(N+b.cnt) < 1.0*N)
         head = joinRec(head, b);
    else head = joinRec(b, head);
  }
private Node joinRec ( Node a, Node b )
  { if (b == null) return a;
    if (a == null) return b;
    b = insertRoot(b, a.item);
    b.left = joinRec(a.left, b.left);
    b.right = joinRec(a.right, b.right);
    updateCnt; return b;
  }
```

Původní verze:

```
void join (Node b)
  { head = joinRec(head, b); }
```

Randomizovaný BVS - remove

- operaci remove koncipujeme stejně jako dříve s tím, že operaci joinLR nahradíme následujícím kódem
- joinLR provádí náhodné rozhodnutí, zda vypouštěný uzel nahradí jeho předchůdcem nebo následníkem

Původní verze:

```
private Node joinLR (Node a, Node b)
  { if ( b == null ) return a;
    b = partRec(b, 0);
    b.left = a;
    return b;
  }
```

```
private Node joinLR ( Node a, Node b )
  { int N = a.cnt + b.cnt;
    if (a == null) return b;
    if (b == null) return a;
    if ( Math.random()*N < 1.0*a.cnt )
         { a.right = joinLR(a.right, b); return a; }
    else { b.left  = joinLR(a, b.left); return b; }
  }
```

Vytvoření BVS pomocí libovolné posloupnosti randomizovaných operací insert, remove a join je ekvivalentní vytvoření standardního BVS z náhodné permutace klíčů.

Vyvažování BVS

Kdy má BVS opravdu zaručenou výšku (hloubku) log N
nebo aspoň O(log N) ??

- perfektní BVS – (ú-)plný binární strom, listy v hloubce *h (a h-1)*
  - plný BVS má přesně $2^{h+1} – 1$ uzlů – my potřebujeme libovolné N
  - daný BVS pro libovolný počet uzlů projdeme a perfektně vyvážíme – **cena?**

```
private Node balanceRec ( Node h )
  {
    if ((h == null) || (h.cnt == 1)) return h;
    h = partRec(h, h.cnt/2);
    h.left  = balanceRec(h.left);
    h.right = balanceRec(h.right);
    updateCnt(h.left); updateCnt(h.right);
    updateCnt(h);
    return h;
  }
```

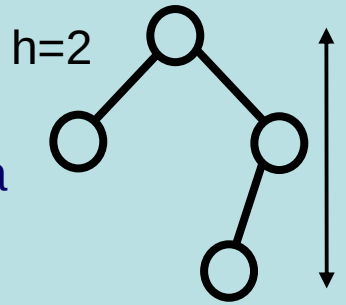
Vyvažování BVS

Místo perfektního vyvážení dovolíme malé rozdíly mezi levým a pravým podstromem nebo ve stupních uzlů a průběžně je udržujeme, kritérium může být

- **výška podstromů - AVL strom**
- **tzv. černá výška – červeno-černý strom**
- **výška + počtu potomků - 1-2 strom**
- **váha podstromů (počty uzlů) - váhově vyvážený strom**
- **2 – 3 – 4 strom**
- **atd.**

AVL stromy

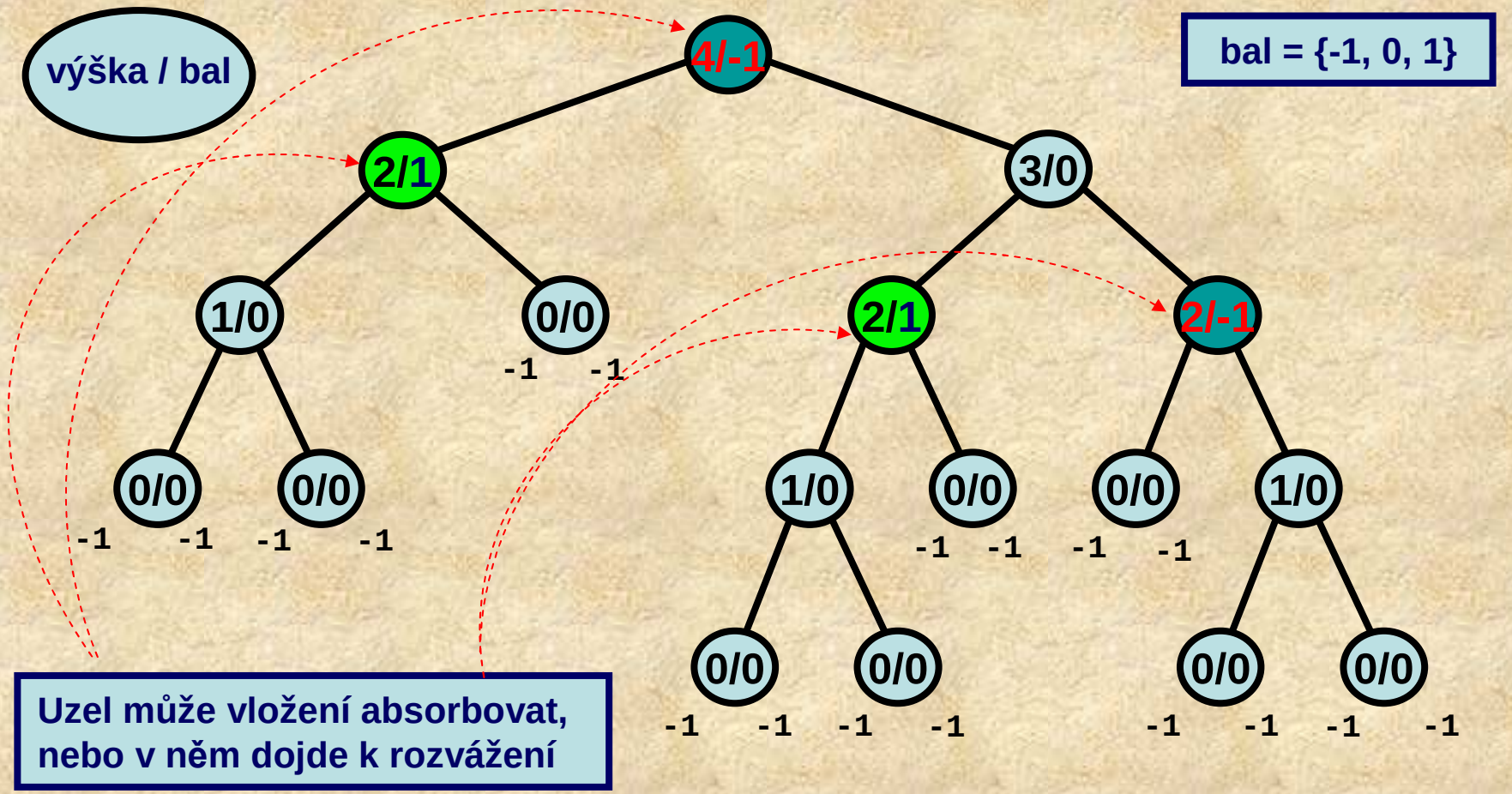
- AVL strom je **výškově vyvážený** BVS
- autoři Adelson-Velskij a Landis, 1962
- definice výšky pro AVL stromy:
  - prázdný strom má výšku -1
  - neprázdný strom výška = hloubka nejhlubšího potomka
- výškové vyvážení znamená, že rozdíl výšek potomků **`bal`** = {-1, 0, 1}



Předpokládáme pro jednoduchost, že každý uzel obsahuje složku height udávající výšku odpovídajícího AVL podstromu.

AVL strom - výšky a rozvážení

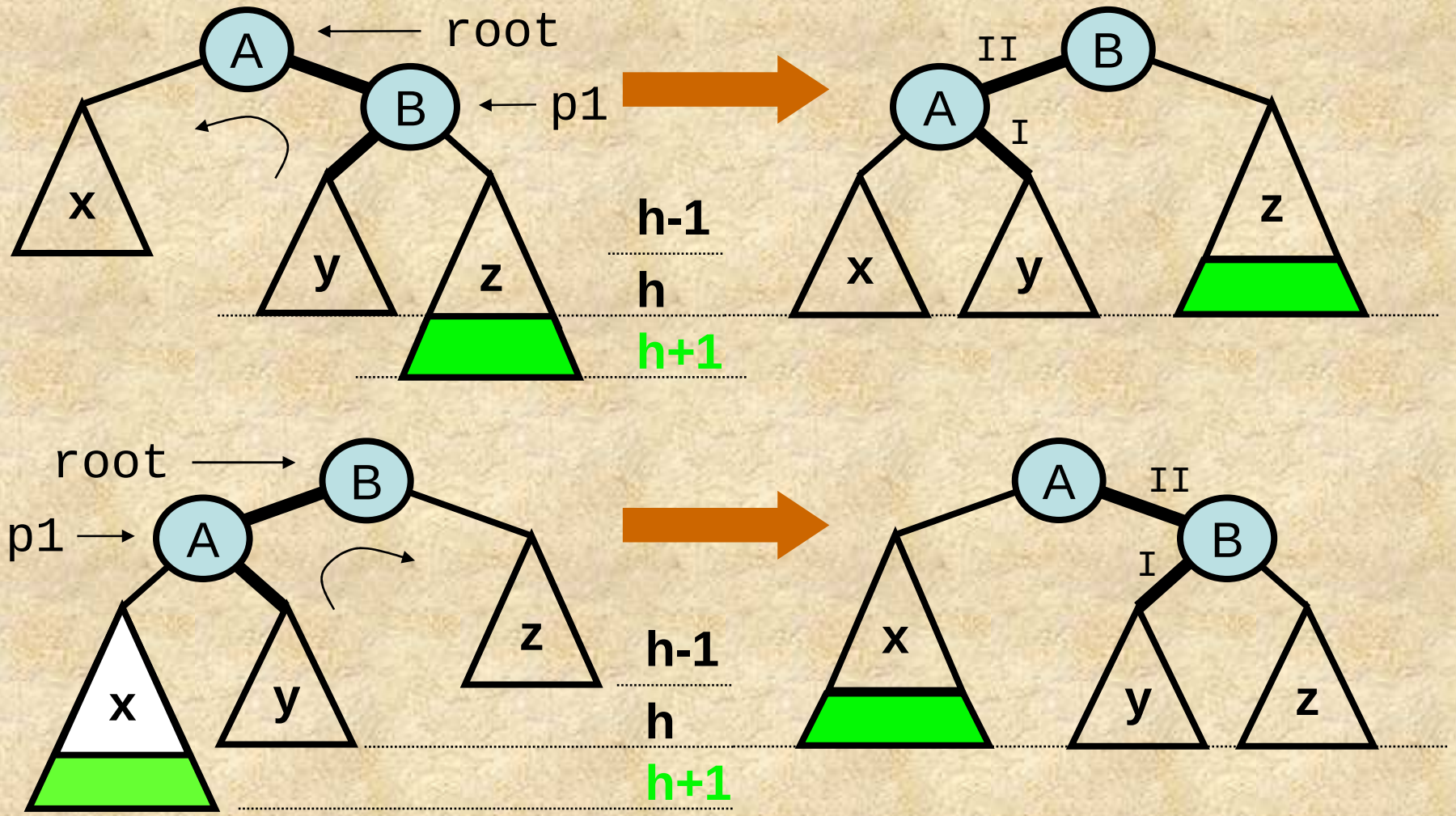
bal(x) = výška(x.left) – výška(x.right)



Změna výšek po rotaci

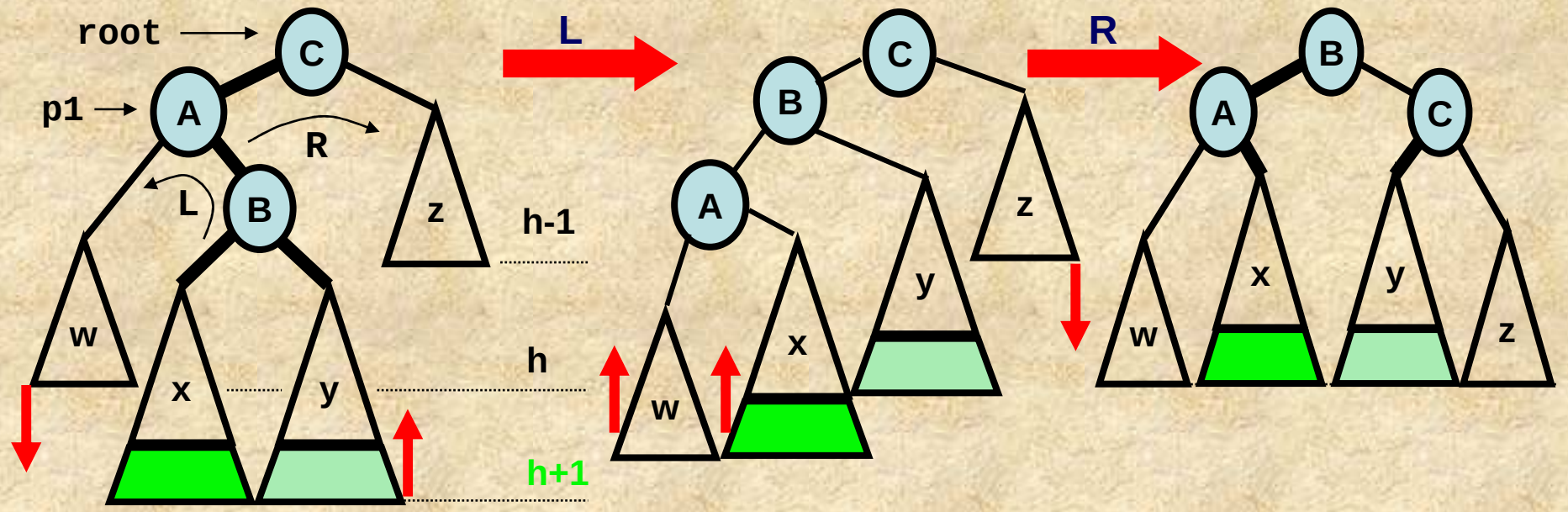

```
int height (Node h) { return (h == null) ? -1 : h.height; }

Node rotateRight (Node h)
  { Node p = h.left; h.left = p.right; p.right = h;
    h.height = max(height(h.left), height(h.right)) + 1;
    p.height = max(height(p.left), h.height) + 1;
    return p;
  }
Node rotateLeft (Node h)
  { Node p = h.right; h.right = p.left; p.left = h;
    h.height = max(height(h.left), height(h.right)) + 1;
    p.height = max(height(p.right), h.height) + 1;
    return p
  }
```

Vliv levé/pravé rotace na výšku



Java-like pseudo code

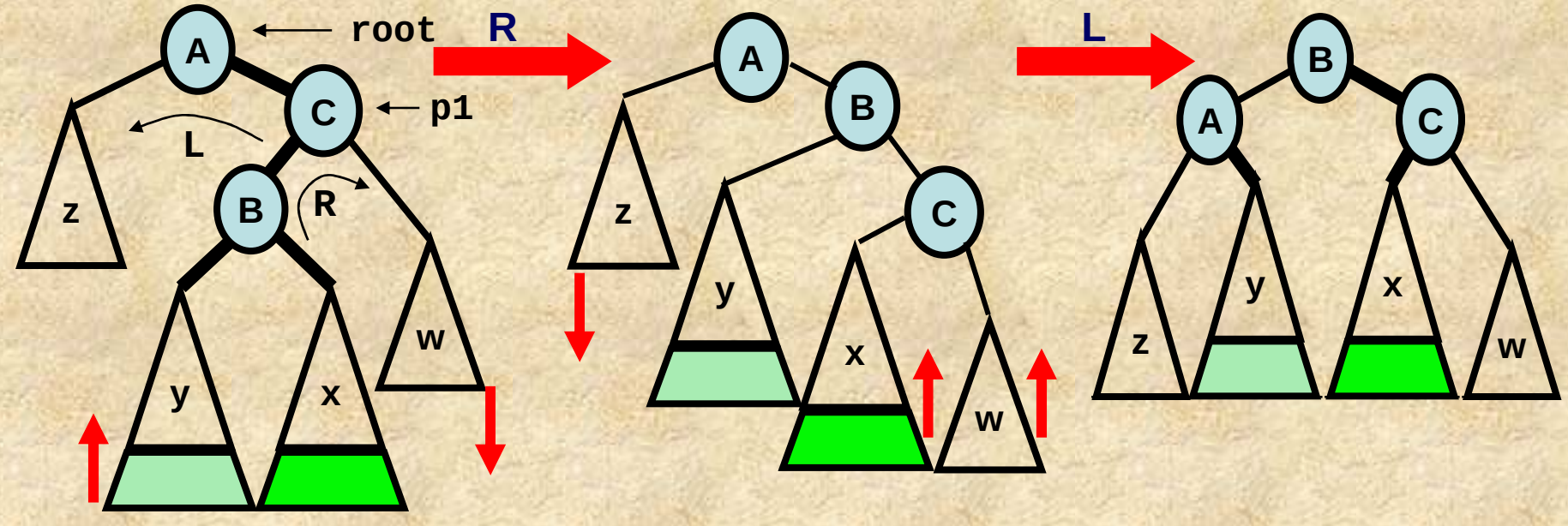
Dvojitá LR a RL rotace




```
Node rotateLR( Node root )
  { Node p1 = root.left;
    root.left = rotateLeft(p1);
    return rotateRight(root);
}
```

```
Node rotateRL( Node root )
 { Node p1 = root.right;
    root.right = rotateRight(p1);
    return rotateLeft(root);
 }
```

Dvojitá LR a RL rotace

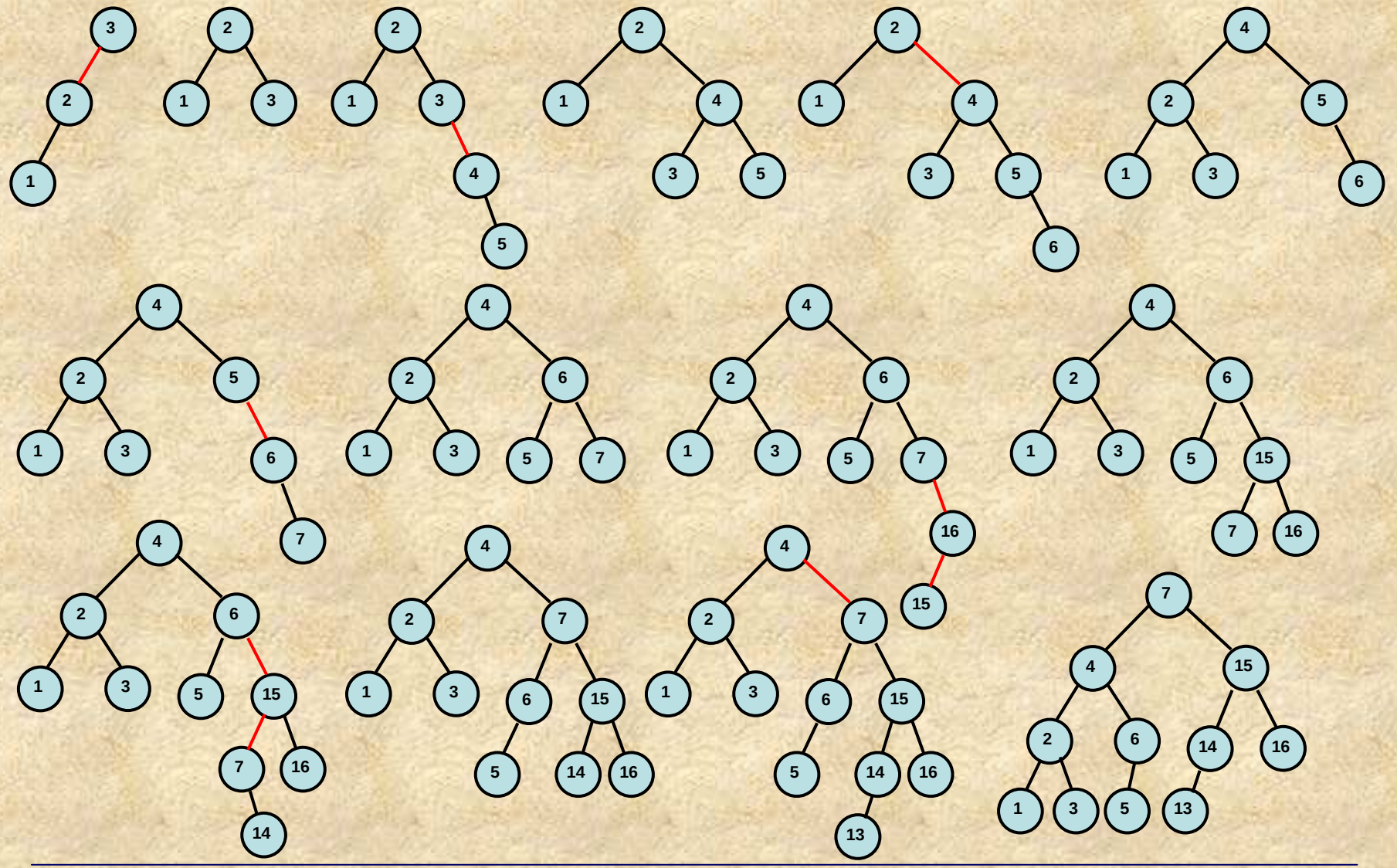



```
Node rotateLR( Node root )
  { Node p1 = root.left;
    root.left = rotateLeft(p1);
    return rotateRight(root);
  }
```

```
Node rotateRL( Node root )
  { Node p1 = root.right;
    root.right = rotateRight(p1);
    return rotateLeft(root);
  }
```

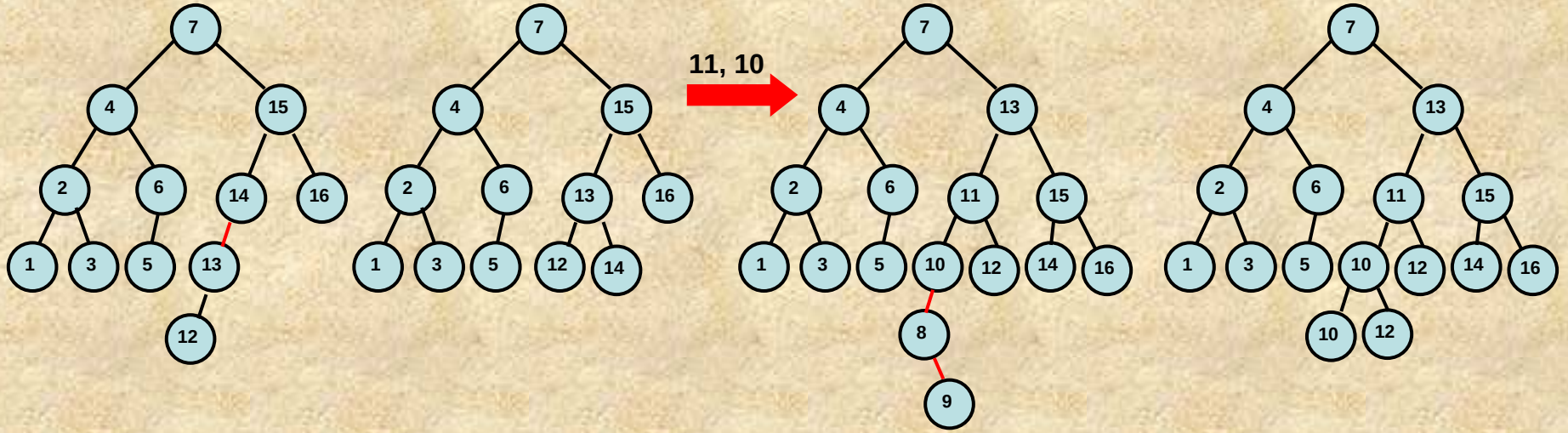
Příklad vytvoření AVL stromu

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



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 ⇒ rotace vpravo**
2. **vložení do pravého podstromu levého potomka uzlu x ⇒ dvojitá LR rotace**
3. **vložení do levého podstromu pravého potomka uzlu x ⇒ dvojitá RL rotace**
4. **vložení do pravého podstromu pravého potomka uzlu x ⇒ rotace vlevo**

AVL insert (s vyvažováním)

```
Node avlInsert( Node h, Elem x )
  { if ( h == null ) h = new Node(x);
    else if ( x.key < h.item.key )
      { h.left = avlInsert(h.left, x);
        if ( height(h.left) - height(h.right) == 2 )
          if( x.key < h.left.item.key )    // 1. insert byl vlevo-vlevo
                h = rotateRight(h);
            else h = rotateLR(h);          // 2. insert byl vlevo-vpravo
      }
    else if( x.key > h.item.key )
      { h.right = avlInsert(h.right,x);
        if ( height(h.right) - height(h.left) == 2 )
          if ( x.key > h.right.item.key )  // 4. insert byl vpravo-vpravo
                h = rotateLeft(h);
            else h = rotateRL(h);          // 3. insert byl vpravo-vlevo
      }
    else h.item = x;                       // existing key – copy value
    h.height = max(height(h.left), height(h.right)) + 1;
    return h;
  }
```

AVL - výška stromu

Pro AVL strom T s *N* uzly platí

- výška h(T) je maximálně o 45% větší než výška optimálního stromu
- $\log_2(n+1) \leq h(T) \leq 1.4404 \log_2(n+2) - 0.328$

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**

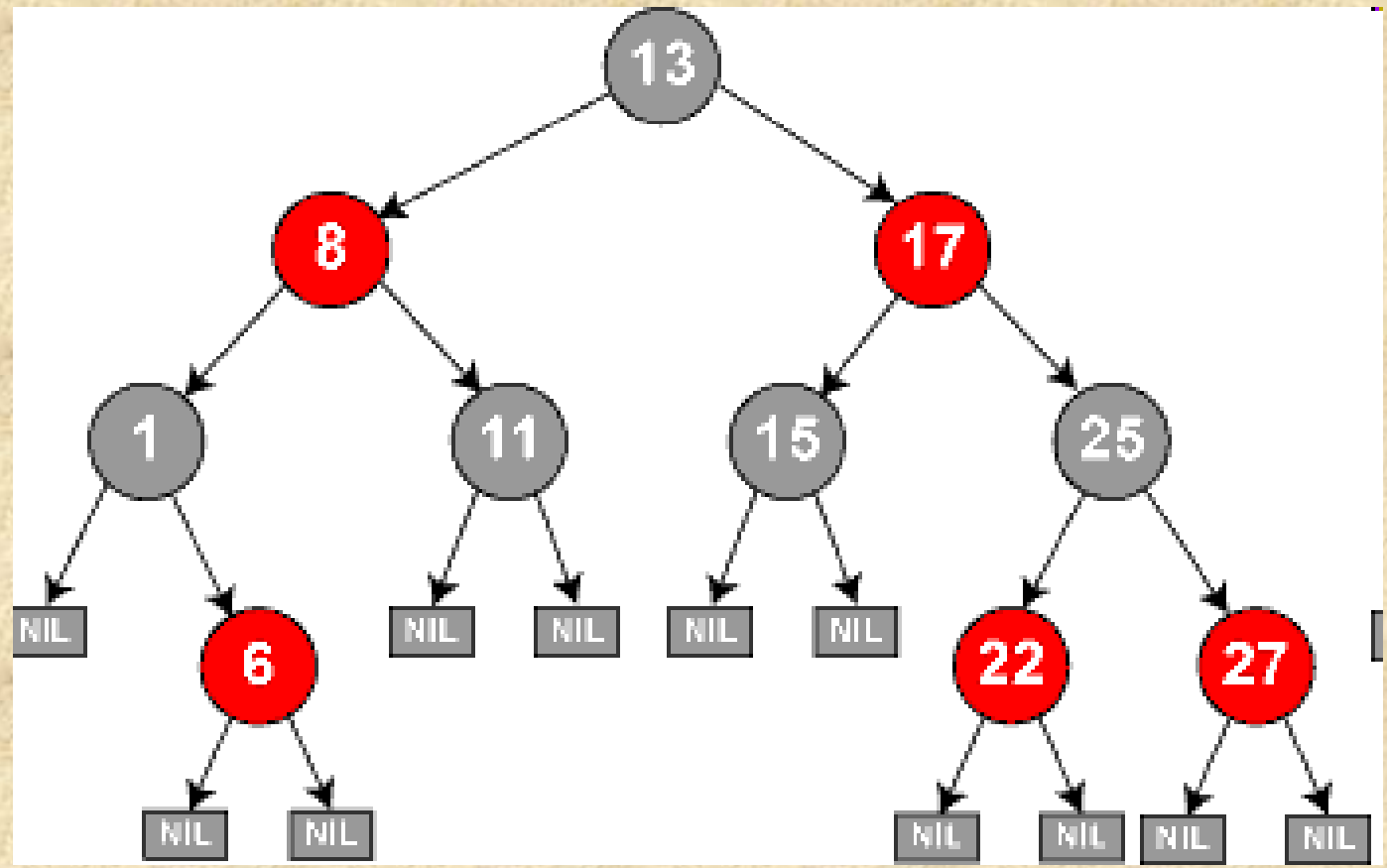
Definice RB stromu - RB strom je BVS s následujícími vlastnostmi:

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. **pro každý uzel platí, že všechny cesty vedoucí z tohoto uzlu do listů obsahují stejný počet černých uzlů**

Předpokládané složky uzlu v RB stromu:

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

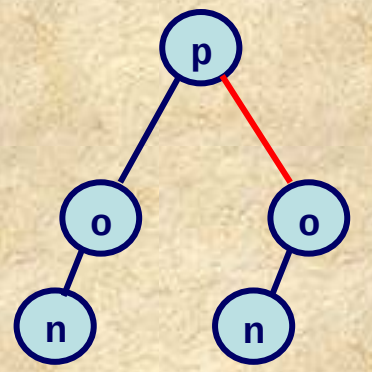
Příklad RB stromu



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

Rotace v RB stromech

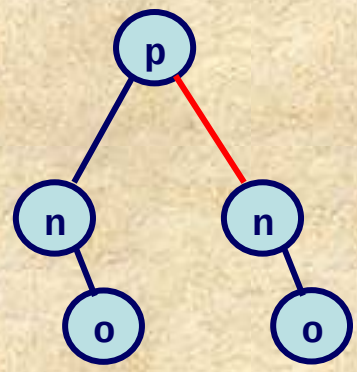
Existence odkazu na rodiče trochu komplikuje implementaci rotací.
Je třeba změnit odkaz v rodiči na uzel vytažený rotací nahoru.



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 h )
  { Node p = h.left;
    replaceNode(h, p);
    h.left = p.right;
    if ( p.right != NIL )
      p.right.parent = h;
    p.right = h;
    h.parent = p;
  }
```

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

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

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

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 sporozence 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 (jako první vložený uzel) byl přebarven na černo
- druhý test checkCase2 zkontroluje, zda je rodič vloženého uzlu černý

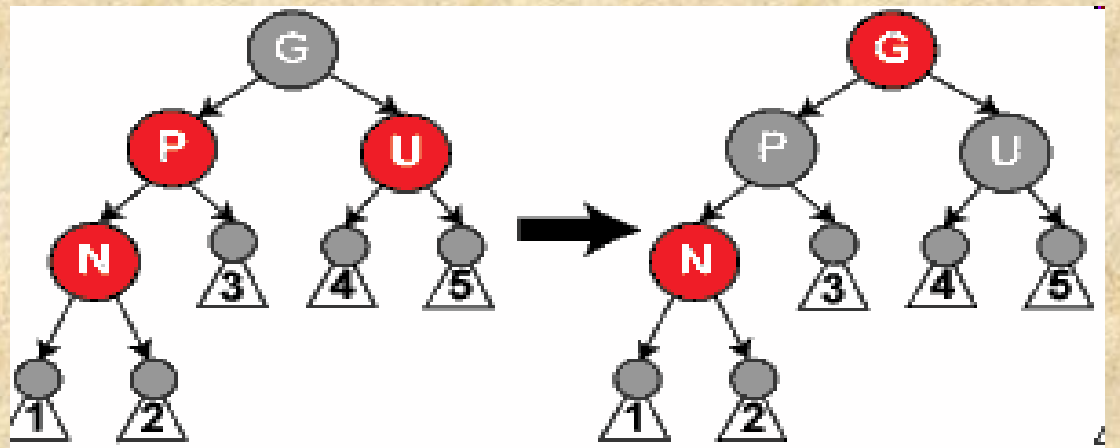
```
private void checkCase1 ( Node n )
  {
    if (n.parent == NIL) n.color = BLACK; // the root was tested
    else checkCase2(n);
  }

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

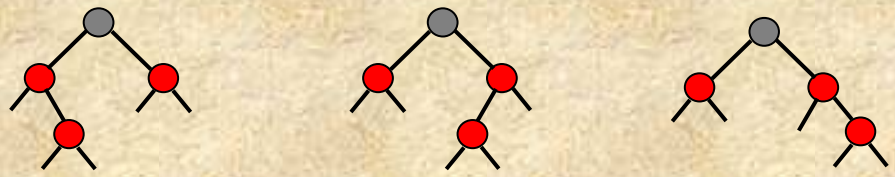
RB stromy – insert (4)

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 prověřujeme prarodiče
- jinak pokračujeme dalším testem

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



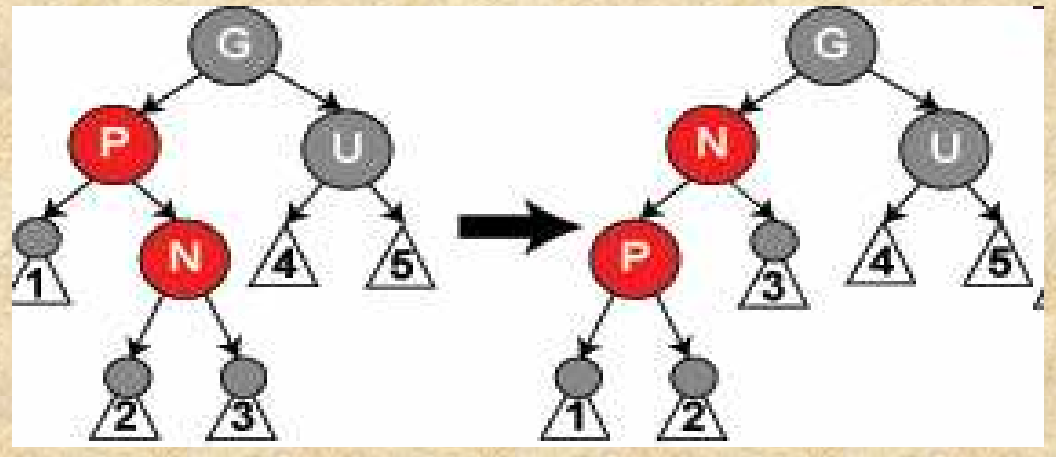


```
void checkCase3 ( Node n )
  { if ( n.uncle().color == RED ) {
      n.parent.color = BLACK; n.uncle().color = BLACK;
      n.grandparent().color = RED;
      checkCase1(n.grandparent());
    } else checkCase4(n);
  }
```

RB stromy – insert (5)

Víme, že strýc je černý.

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

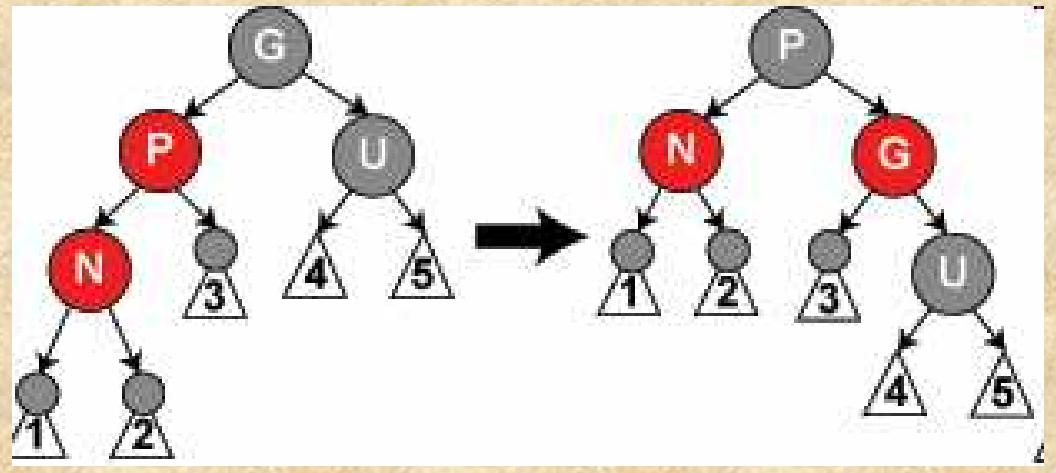



```
void checkCase4 ( Node n )
  { if (n == n.parent.right && n.parent == n.grandparent().left)
      { rotateLeft(n.parent); n = n.left; }
    else if (n == n.parent.left && n.parent == n.grandparent().right)
      { rotateRight(n.parent); n = n.right; }
  insertCase5(n);
  }
```

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)

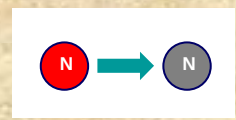
Shrnutí postupu při testování (a úpravě) okolí uzlu n:

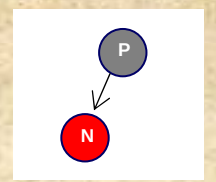
- jedná-li se o kořen, obarvíme jej na černo a je hotovo
- jinak, je-li rodič uzlu n černý, je hotovo

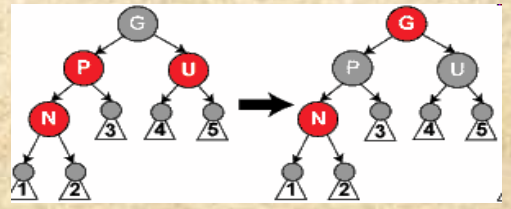
- jinak (víme, že uzel n i jeho rodič mají červenou barvu), je-li také jeho strýc červený, přebarvíme rodiče i strýce na černo, prarodiče na červeno a prověřujeme prarodiče

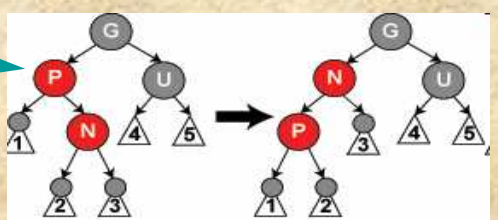
- jinak, je-li n pravým synem svého rodiče, který je levým synem jeho prarodiče ⇒ provedeme rotaci vlevo kolem rodiče a pokračujeme testováním spodního uzlu
- jinak, je-li n levým synem svého rodiče, který je pravým synem jeho prarodiče ⇒ provedeme rotaci vpravo kolem rodiče a pokračujeme testováním spodního uzlu

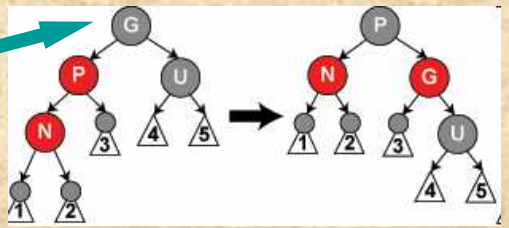
- jinak, je-li n levým synem svého rodiče, který je levým synem jeho prarodiče ⇒ provedeme rotaci vpravo kolem prarodiče
- jinak, n je pravým synem svého rodiče, který je pravým synem jeho prarodiče ⇒ provedeme rotaci vlevo kolem rodiče











Prameny

- **Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, MIT Press, 1990**
- **Sedgewick, R.: Algorithms in Java (Parts 1 – 4: Fundamentals, Data Structures, Sorting, Searching).Third edition, Addison Wesley / Pearson Education, Boston, 2003**
- http://en.literateprograms.org/Red-black_tree_%28Java%29