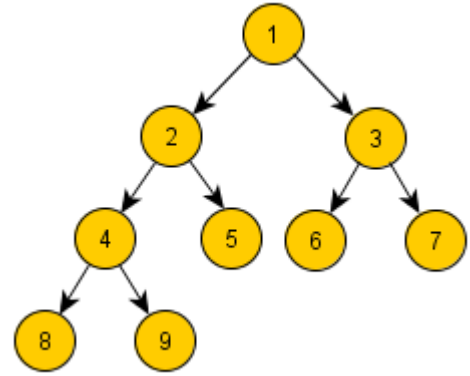


Stromy = Trees

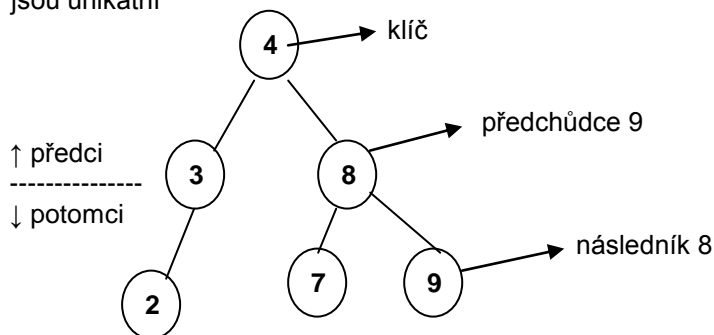
Binární pravidelný orientovaný strom

- hierarchická struktura, kde každý otec (rodič) má 0 až mnoho dětí a každé dítě právě jednoho otce (rodiče)
- uzel, který je praotcem všech ostatních uzlů nazveme kořenem
- **uzel** = má potomky, alespoň jednoho
- **stupeň uzlu** = počet přímých následovníků vnitřního uzlu
- **stupeň stromu** = maximální stupeň mezi všemi uzly stromu
- **list** = uzel, který nemá žádné potomky, je koncový
- **předek** = určen podle pořadí ve stromu
- **předchůdce** = určen podle hodnoty, MIN nemá předchůdce
- být stromem je rekurzivní vlastnost - každý podstrom stromu S je také stromem
- strom je velmi populární pro svoji jednoduchost a použitelnost
- halda = jeden typ binárního stromu, kdy rodič je větší než syn



Klíče

- jsou unikátní



Vlastnosti stromu

- **N-arita** = kolik smí mít každý uzel maximálně potomků, z tohoto hlediska patří mezi neoblíbenější binární stromy (každý uzel má 0, 1 nebo 2 potomky).
- **Hloubka** = délka cesty od kořene k uzlu, tj. počet hran od kořene k listu, hloubkou rozumíme maximální hloubku libovolného uzlu (kořen je v hloubce 0, potomci v hloubce 1, vnuci v hloubce 2...).
- **Délka cesty** = je rovna počtu hran, které cesta obsahuje, tedy "**počet uzlů posloupnosti - 1**".
- **Cesta** k nějakému uzlu = je definována jako posloupnost všech uzlů od kořene k uzlu.
- **Výška stromu** = je rovna hodnotě maximální hloubky uzlu, označuje se též jako hloubka stromu.
- **Šířka stromu** = počet uzlů na stejné úrovni. Na "téže úrovni" jsou prvky se stejnou hloubkou.
- **Nejmenší výška** = strom má nejmenší výšku, právě tehdy, když na všech úrovních (s možnou výjimkou té poslední) má tato struktura plný počet uzlů. Úroveň všech listů je stejná nebo se liší max. o 1.
- **Pravidelnost** = N-ární strom je pravidelný, pokud má každý uzel 0 nebo N potomků.
- **Vyváženost** = hloubka stromu se liší max. o 1. N-ární strom je vyvážený, pokud pro všechny listy platí, že nejsou o nic více hlouběji, než kterýkoliv jiný list. Definice „o nic více hlouběji“ se liší v závislosti na konkrétní implementaci.
- **Úplná pravidelnost** = úplným N-árním pravidelným stromem hloubky **k** je strom, jehož každý uzel má 0 nebo N potomků a všechny uzly jsou ve hloubce **k**, tzn. strom, který má všechna patř plná, hloubka je logaritmická

Vyhledávání

- procházím strom a buď najdu nebo ne
- složitost lineární $O(n)$
- při hledání má přednost cesty od kořene dolů, pak prohledávám od listů ke kořeni

Vkládání

- hledám pozici = uzel, kam mohu vložit nový prvek do listu
- hledám MIN prvek – hledám od kořene stále vlevo

- hledám MAX prvek – hledám od kořene stále vpravo
- složitost $O(\log n)$

Mazání

- nalezení hodnoty k mazání

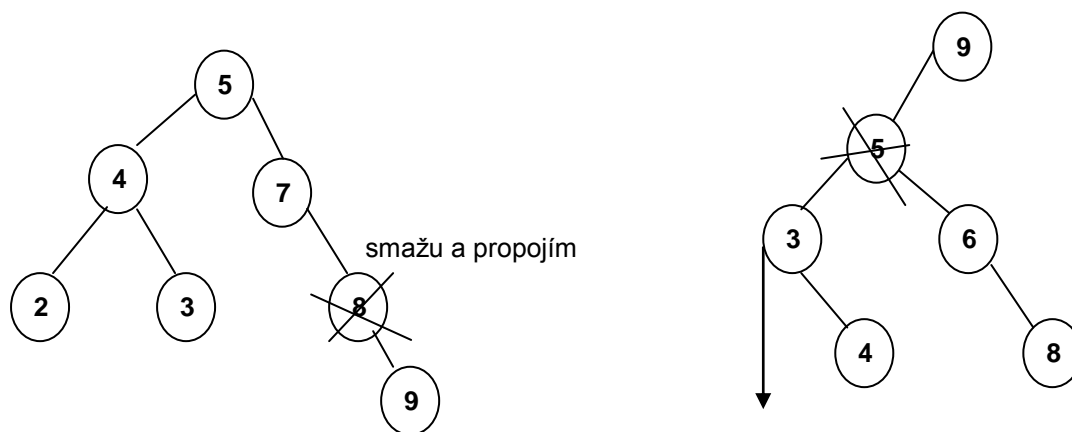
```
parent.left != null
parent.left = parent.value
```

- **z listu** – jednoduše smažeme – složitost $O(1)$
- **uzel s jedním synem** - syna přepojíme na rodiče mazaného uzlu, tzn. na předka - složitost $O(1)$

POZOR – vždy nutno odkazovat na rodiče a nakonec nezapomenout uvolnit paměť

```
p.parent. - left      P - left
           - right    - right
```

- **uzel se 2 syny** - nejprve najdu předchůdce nebo následníka - tím přepíšu mazaný prvek – překopíruju ho - smažu kopírovaný prvek z původní pozice - složitost $O(n)$
- **složitost** lineární $O(n)$, protože $O(n) + O(1) + O(1) | O(1) | O(n) = O(n)$ (nalezení prvku + zjištění kolik má synů + mazání dle druhu)

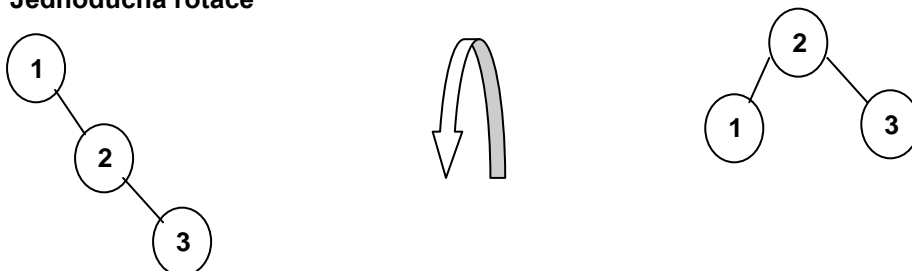


najdu předchůdce (= 4) nebo následníka (= 6) a prohodím

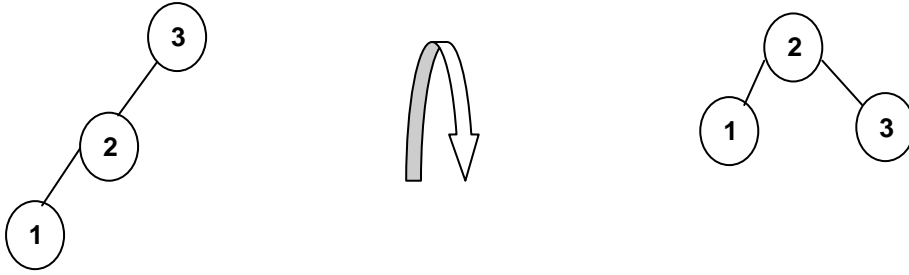
Rotace

- používá se k vyvážení stromu
- jde o převedení nevyváženého stromu na vyvážený
- vyvážení je nutné pro optimalizaci procházení
- složitost operací závisí na hloubce stromu – vyvážený strom $O(\log(n))$, nevyvážený až $O(n)$
- jednoduchá = L nebo P
- dvojitá = L-P nebo P - L

Jednoduchá rotace

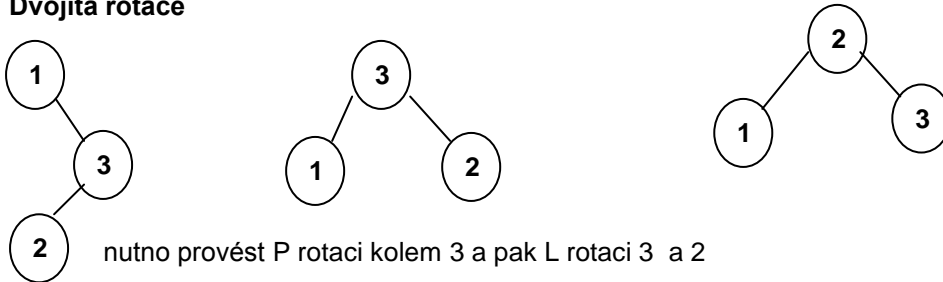


strom je nevyvážený → je nutno provést L rotaci



strom je nevyvážený → je nutno provést P rotaci

Dvojitá rotace



* N-arni strom

```
public class Tree {
```

```
  * Kolekce potomku
```

```
  private ArrayList<Tree> children;
```

```
  * Hodnota uchovavana v uzlu
```

```
  private Object value;
```

```
  * Prida podstrom jako potomka tohoto korene
```

```
  * @param t strom k pridani
```

```
  * @param index poradi potomka
```

```
  public void addChild(Tree t, int index){
```

```
    children.add(index, t);
```

```
  }
```

```
  * Odstrani potomka na danem indexu
```

```
  * @param index index potomka k odstraneni
```

```
  public void removeChild(int index){
```

```
    children.remove(index);
```

```
  }
```

```
  * Vrati potomka na danem indexu
```

```
  * @param index poradi potomka
```

```
  * @return potomek na zadanem indexu
```

```
  public Tree get(int index){
```

```
    return children.get(index);
```

```
  }
```

```
  * @return the value
```

```
  public Object getValue() {
```

```
    return value;
```

```
  }
```

```
  * @param value the value to set
```

```
  public void setValue(Object value) {
```

```
    this.value = value;
```

Binární vyhledávací stromy = BVS

- poziční strom = není nám jedno, zda syn je L nebo P
 - v L se ukládají prvky < než rodič
 - v P se ukládají prvky > než rodič
 - MIN prvek zcela vlevo, MAX prvek zcela vpravo – nemusí být listem!!!
 - každý uzel má max 2 listy, resp. 0 – 2 **SYNY**
 - pokud je nevyvážený, nutno vyvážit = rotace
 - složitost operací – vyvážený strom $O(\log n)$, nevyvážený až $O(n)$
 - nevýhoda – struktura BVS může být velmi rozvolněná – může vést k degeneraci stromu v „hada“
-
- strom hloubky 0 má 1 uzel (= kořen)
 - úplný BVS hloubky h má $2^{h+1} - 1$ VŠECH uzlů
 - počet vnitřních uzlů (tj. bez kořene a listů) má $\lfloor n/2 \rfloor$ - n = počet všech uzlů

```
//Základní třída
package{
    public class Uzel{
        public var levý:Uzel; // uzel vlevo
        public var pravý:Uzel; // uzel vpravo
        public var hodnota:int = int.MAX_VALUE; // hodnota v uzlu
    }
}

//Konstruktor - jeho parametr je číslo, které sem ukládáme. Pokud nebude
//uveden, hodnota v uzlu je int.MAX_VALUE.
public function Uzel(c:int = int.MAX_VALUE):void{
    if(c == int.MAX_VALUE) return; // bez parametru -> vrátíme se
    hodnota = c; // jinak přiřadíme parametr do hodnoty
}

//Vypsání čísel ve struktuře - seřazeně - nejdříve vypíšeme levý podstrom,
//pak vypíšeme hodnotu v tomto uzlu tolikrát, kolikrát tu je, potom pravý
//podstrom.
public function toString():String
{
    var out:String = "";

    if(levý != null) out += levý.toString();
    out += hodnota + ", ";
    if(pravý != null) out += pravý.toString();

    return out;
}

Vyhledávání


- postup od kořene k listu
- složitost  $O(n)$  obecného BVS
- dojde do listu, odkud nemůžu pokračovat = prvek v BVS není



//Zjištění, zda číslo už je ve struktuře - podíváme se, zda je v daném
//uzlu, pokud je menší než hodnota v uzlu, hledáme ho v levém podstromu,
//jinak v pravém.
public function Obsahuje(i:int):Boolean{
    if(i == hodnota) return true;
    if(i < hodnota){
        if(levý != null) return levý.Obsahuje(i);
        else return false;
    } else {
        if(pravý != null) return pravý.Obsahuje(i);
        else return false;
    }
}
}
```

Vkládání

- složitost $O(n)$ – obsahuje nalezení místa vložení
- pozici hledáme od kořene
- v místě, kde nelze jít dál vložíme nový uzel
- každý prvek lze vložit pouze jednou = nutno ošetřit

//Přidání čísla do struktury - pokud daná hodnota už je v uzlu, neděláme nic. Pokud přidáváme menší číslo než hodnota v uzlu, přidáme ho do levého podstromu, jinak do pravého.

```
public function Přidej(c:int){
    if(hodnota == int.MAX_VALUE){
        hodnota = c; return;} // konstruktor byl bez parametru
    if(c == hodnota){return;} // c už tu je
    if(c > hodnota){
        if (pravý != null) pravý.Přidej(c);
        else pravý = new Uzel(c);
    } else {
        if (levý != null) levý.Přidej(c);
        else levý = new Uzel(c);
    }
}
```

Mazání

- viz standard TREE
- složitost $O(n)$

//v C++

```
void DeleteNode(struct node * & node) {
    if (node->left == NULL) {
        struct node *temp = node;
        node = node->right;
        delete temp;
    } else if (node->right == NULL) {
        struct node *temp = node;
        node = node->left;
        delete temp;
    } else { // Uzel má dva potomky, hledej předchůdce
                (nejpravější potomek levého podstromu).
        struct node **temp = &node->left;
        // získej levého potomka mazaného uzlu

        // najdi nejpravějšího potomka podstromu levého uzlu mazaného uzlu
        (předchůdce)
        while ((*temp)->right != NULL) {
            temp = &(*temp)->right;
        }

        // zkopíruj hodnotu předchůdce do mazaného uzlu
        node->value = (*temp)->value;

        // nyní smaž předchůdce - jeho hodnota byla přesunuta do původně
        mazaného uzlu
        DeleteNode(*temp);
    }
}
```

Procházení stromem do hloubky

Procházení začíná v kořeni stromu a postupuje se vždy na potomky daného vrcholu. Procházení končí, když v žádné větvi (tj. v žádném podstromu) již není následník.

Podle pořadí, ve kterém se prochází uzly uspořádaného stromu, se rozlišují tři základní metody:

- PREORDER** - proved' akci
- projdi levý podstrom
- projdi pravý podstrom

- INORDER**
- projdi levý podstrom
 - proved' akci
 - projdi pravý podstrom

Při použití metody INORDER se prochází uzly v uspořádaném stromě podle jejich přirozeného pořadí.

- POSTORDER**
- projdi levý podstrom
 - projdi pravý podstrom
 - proved' akci

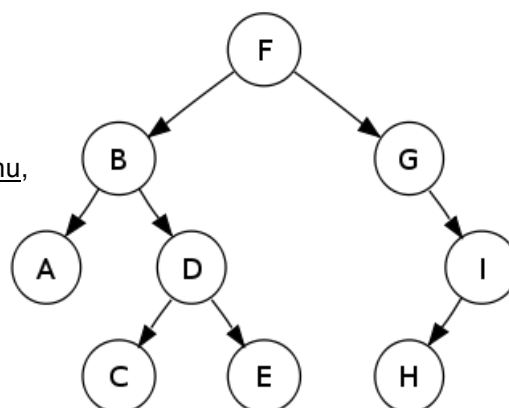
Procházení stromem do hloubky lze řešit pomocí:

- Rekurze – funkce volá sama sebe
- Iterace – provádění stejné operace znovu a znovu

Výsledky způsobů procházení v binárním vyhledávacím stromu,

N = navštívený uzel, **L** = levý, **R** = pravý

- **Preorder** (NLR): F, B, A, D, C, E, G, I, H
- **Inorder** (LNR): A, B, C, D, E, F, G, H, I
- **Postorder** (LRN): A, C, E, D, B, H, I, G, F
- Procházení do šířky (po vrstvách) **Level-order**: F, B, G, A, D, I, C, E, H



Rekurzivně v C

```

//datová struktura stromu
typedef struct node
{
    int val;
    struct node *left, *right;
}*tree;

//Inorder Traversal
void inorder(tree t){
    if(t == NULL)
        return;
    inorder(t->left);
    printf("%d ", t->val);
    inorder(t->right);
}

//Preorder Traversal
void preorder(tree t){
    if(t == NULL)
        return;
    printf("%d ", t->val);
    preorder(t->left);
    preorder(t->right);
}

//Postorder Traversal
void postorder(tree t)
{
    if(t == NULL)
        return;
    postorder(t->left);
    postorder(t->right);
    printf("%d ", t->val);
}
  
```

Iterativně v Javě – preorder a inorder

```

public void iterativePreorder(Node root) {
    Stack nodes = new Stack();
    nodes.push(root);

    Node currentNode;

    while (!nodes.isEmpty()) {
  
```

```

        currentNode = nodes.pop();
        Node right = currentNode.right();
        if (right != null) {
            nodes.push(right);
        }
        Node left = currentNode.left();
        if (left != null) {
            nodes.push(left);
        }
        System.out.println("Node data: "+currentNode.data);
    }
}

public void iterativeInOrder(Node node) {
    //incoming node is root
    Stack<Node> nodes = new Stack<Node>();
    while (!nodes.isEmpty() || null != node) {
        if (null != node) {
            nodes.push(node);
            node = node.left;
        } else {
            node = nodes.pop();
            System.out.println("Node value: " + node.value);
            node = node.right;
        }
    }
}
}

```

Iterativně v C – postorder

```

void iterativePostOrder(Node* root) {
    if (!root) {
        return;
    }
    stack<Node*> nodeStack;

    Node* cur = root;
    while (true) {
        if (cur) {
            if (cur->right) {
                nodeStack.push(cur->right);
            }
            nodeStack.push(cur);
            cur = cur->left;
            continue;
        }

        if (nodeStack.empty()) {
            return;
        }

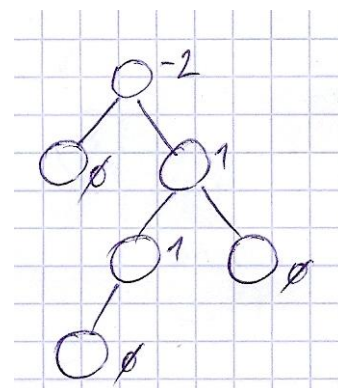
        cur = nodeStack.top();
        nodeStack.pop();

        if (cur->right && !nodeStack.empty() && cur->right ==
nodeStack.top()) {
            nodeStack.pop();
            nodeStack.push(cur);
            cur = cur->right;
        } else {
            std::cout << cur->val << " ";
            cur = NULL;
        }
    }
}
}

```

AVL strom (AVL = zkratka tvůrců)

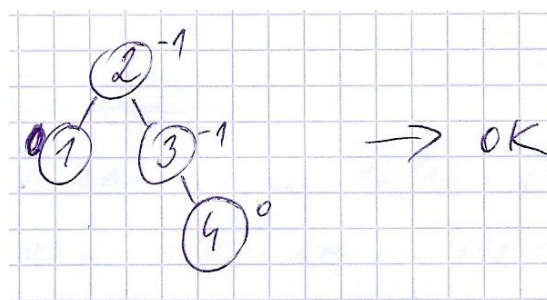
- BVS strom
- podmínka – listy jsou max. ve dvou hladinách
- strom, kde je index rozvážení v každém uzlu v rozmezí -1 – 0 – 1, tzn. délka pravého a levého podstromu se liší max. o 1
- zavádí se **index rozvážení = tzv. balanc**
- výpočet **ind. rozv. = L podstrom – P podstrom**
- úprava vyvážení se provádí rotací
- list (koncový uzel) má balanc 0
- nevýhoda – jsou kaledny příliš velké nároky na strukturu



Není AVL strom – má hlubku -2

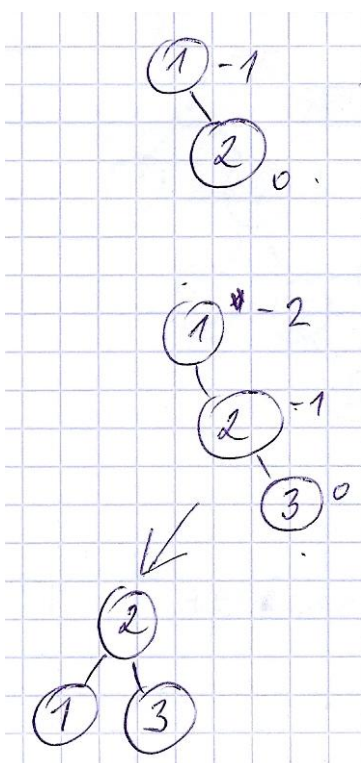
Rotace AVL stromu

- přepočte se balanc každého uzlu – bere se vždy nejdelší cesta list - uzel
- pokud má syn "+", provede se P rotace
- pokud má syn "-" provede se L rotace
- pokud jsou stejná znaménka u dvou uzlů, pak je rotace jednoduchá, pokud je dvojitá
- při implementaci AVL rotace nutno implementovat zásadně s odkazem na otce
- složitost $O(1)$

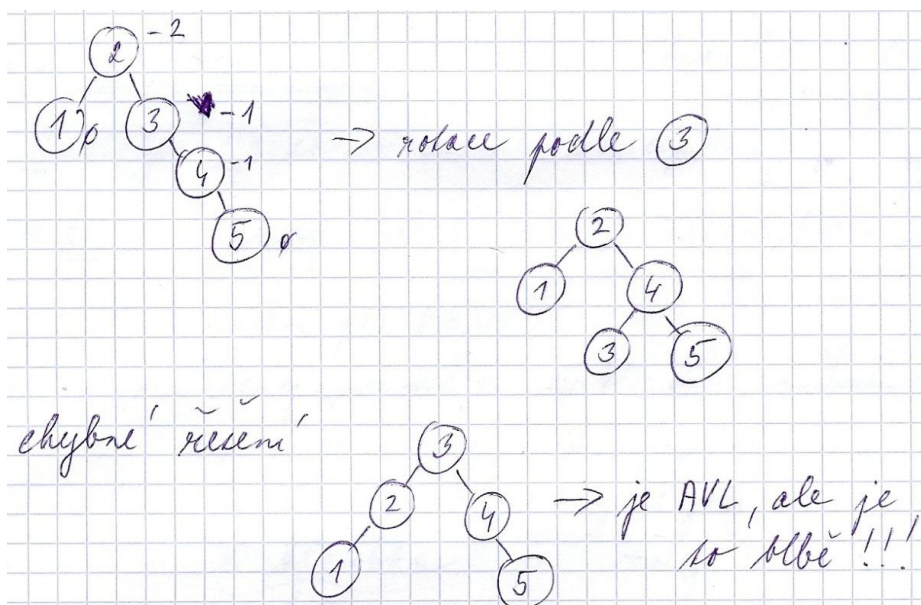


Mazání

- po odstranění uzlu přepočítáme balanc a event. se provede rotace
- složitost $O(\log n)$ = mazání + rotace

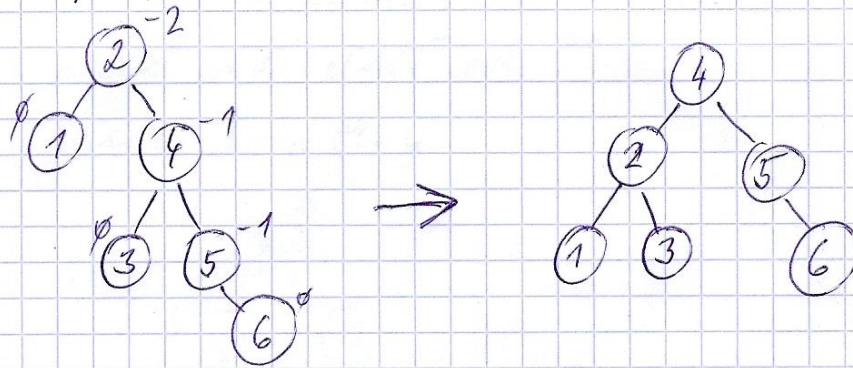


Není AVL strom – syn i rodič mají "-" → provede se jednoduchá L rotace



Rotace se provede vždy, jakmile se zjistí, že nejde o AVL strom, tzn., jakmile se narazí na IR -2.

rotace podstromu



Implementace L rotace AVL stromu

```
rotace (x){
```

```
    y = x.parent;    //pomocná proměnná
```

```
    1. x.parent = x.right; //nejprve je nutno ověřit, zda existuje, tzn. zda není null
```

```
    2. x.right = x.parent.left;
```

```
    3. x.parent.left = x;
```

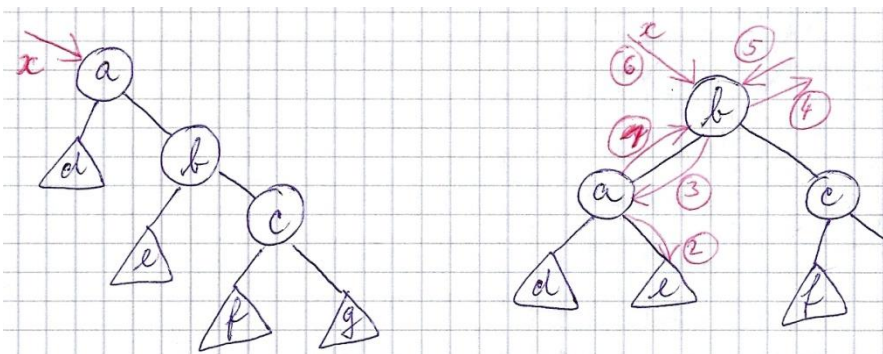
```
    4. x.parent.parent = y;
```

```
    5. if (y.left = x) y.left = x.parent;
```

```
    5. else y.right = x.parent;
```

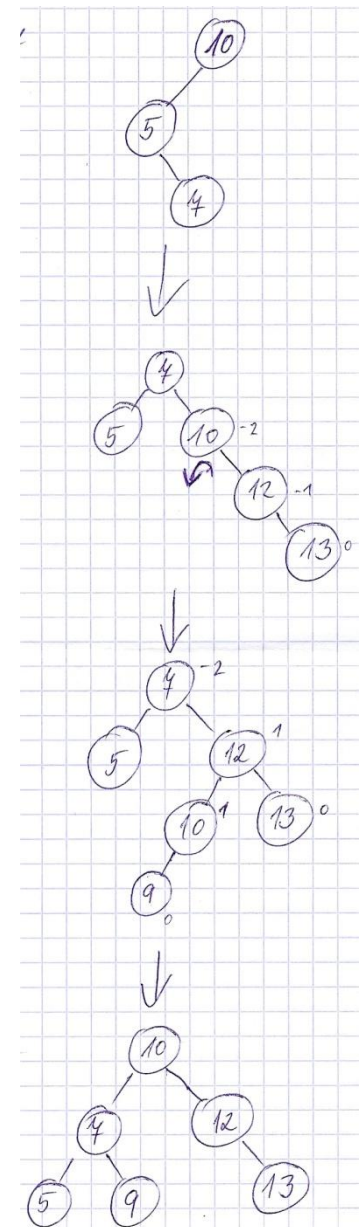
```
    6. x.right.parent = x; //opět nutno otestovat, zda není null
```

```
}
```



Vkládání

- vyhledání pozice $O(\log n)$ + vlastní vložení $O(1) = O(\log n)$
- po vložení každého prvku nutno ověřit, zda je strom AVL, pokud není, pak ihned spravit rotací
- lepší pokud máme stabilní data, tzn. méně vkládání, jinak RBS
- vkládáme 10, 5, 7
- po každém vložení nutno přepočítat uzly
- po vložení 7 není AVL strom, protože 10 má $IR = 2$
- nutno provést L-P rotaci
- vložíme 12, 13
- pak je opět nutno rotovat – 10 má $IR = -2$
- znaménka jsou "-" "-" – provedu P-L rotaci
- nejprve P podle 12
- pak L podle 7



Implementace v C++

```
void LL() {
    p->levy = p->pravy;
    p1->pravy = p;
    p->vyvazenost = 0;
    p = p1;
}

void LR() {
    p2 = p1->pravy;
    p1->pravy = p2->levy;
    p2->levy = p1;
    p->levy = p2->pravy;
    p2->pravy = p;
    if (p2->vyvazenost == -1) {
        p->vyvazenost = 1;
    } else {
        p->vyvazenost = 0;
    }
    if (p2->vyvazenost == 1) {
        p1->vyvazenost = -1;
    } else {
        p1->vyvazenost = 0;
    }
    p = p2;
}

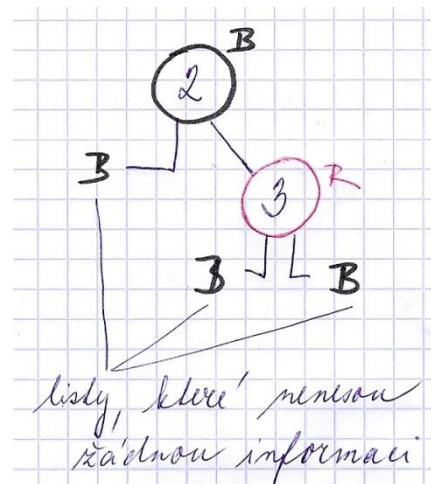
void RR() {
    p->pravy = p1->levy;
    p1->levy = p;
    p->vyvazenost = 0;
    p-> p1;
}
```

RL rotace od Jirky

```
void RL(x) {
    b = x.right.right;
    x.right = x.right.left;
    x.right.parent = x;
    b.left = x.right.right;
    b.left.parent = b;
    x.right.right = b;
    x = x.parent;
    x.parent = x.right;
    x.right = x.right.left;
    x.right.parent = x;
    x.parent.left = x;
    if (x.left == x) y.left = x.parent;
    else y.right = parent;
    x.parent.parent = y;
}
```

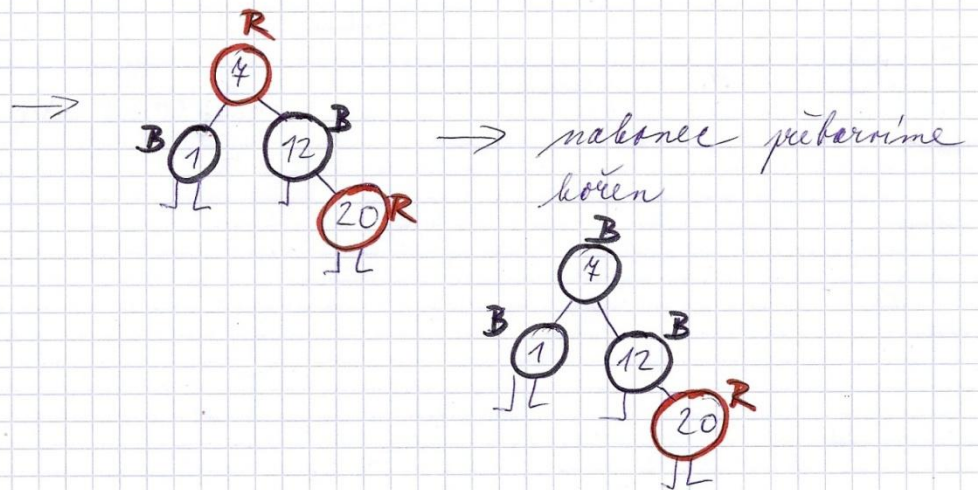
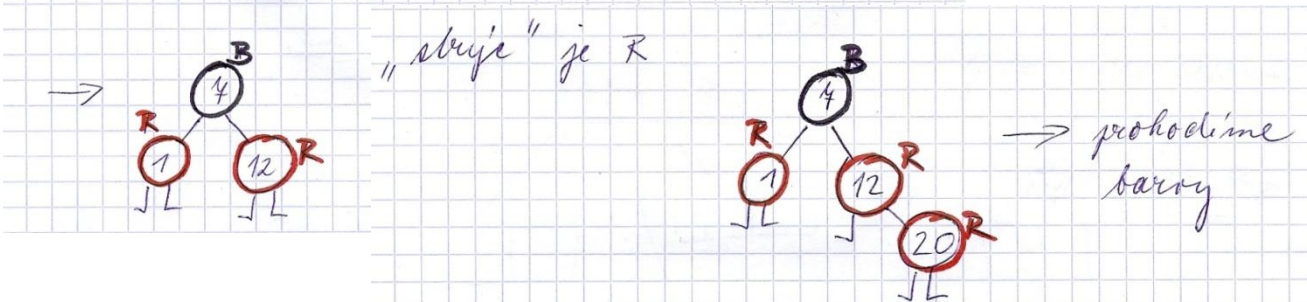
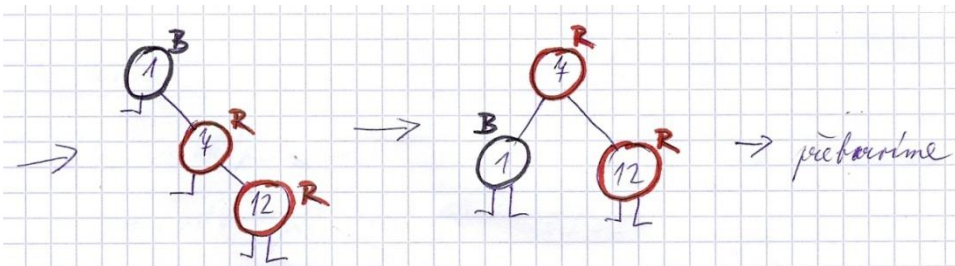
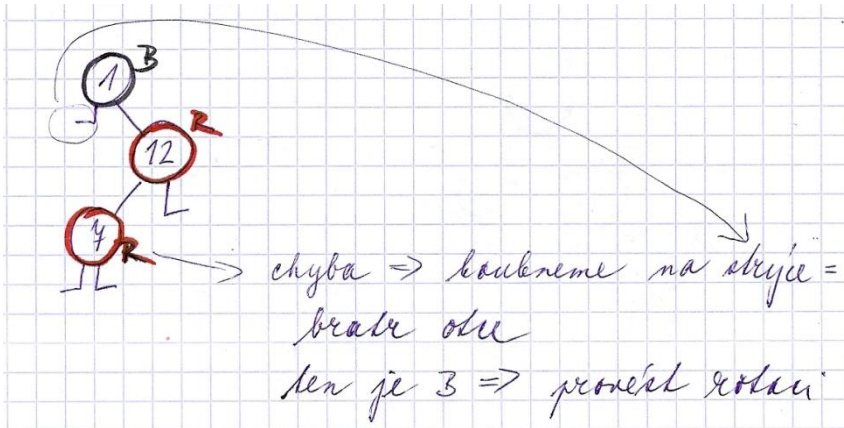
Red – Black stromy (= RBS – autoři konceptu Rudolf a Bayer)

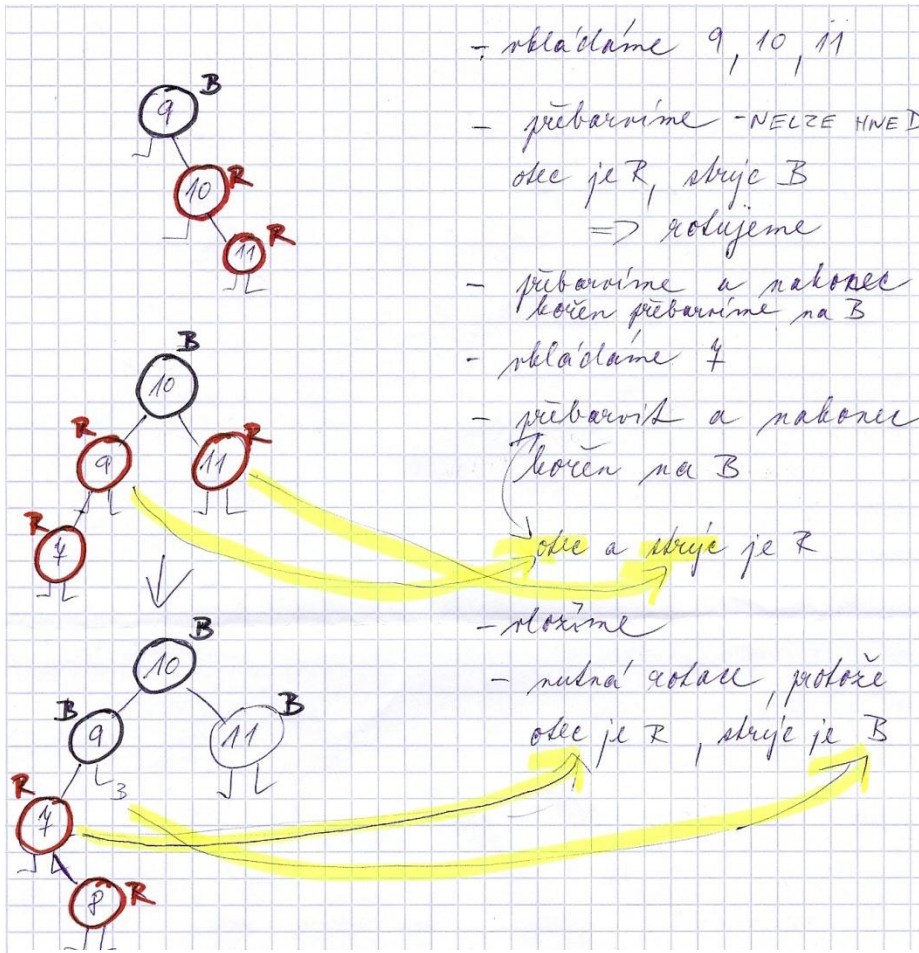
- každý uzel má nějakou barvu = B
- každý kořen je vždy B
- listy nenesou žádnou informaci a jsou vždy B
- R uzel má vždy B syny
- B hloubka (= počet B uzlů a listů z kořene do listu musí být stejný) je vždy B
- použití – tam, kde se často mění objem dat (nestabilní data), protože Insert je jednodušší než u AVL stromu
- nejdelší hloubka je max. dvojnásobkem nejkratší



Vkládání

- stejné jako do Tree
- vložení = přemaže se B list bez informace
- vložení vždy vytvoříme nový list
- nově vložený prvek je vždy R
- pak se přebarví podle podmínek RBS, kořen se přebarvuje vždy nakonec vkládání
- přidáme jako syna B – přebarvení neřešíme
- přidáme jako syna R – nutno přebarvit, nejprve zjistíme barvu strýce (bratr otce)
 - otec B, strýc R – přebarvíme
 - otec R, strýc B nebo oba B B - rotujeme





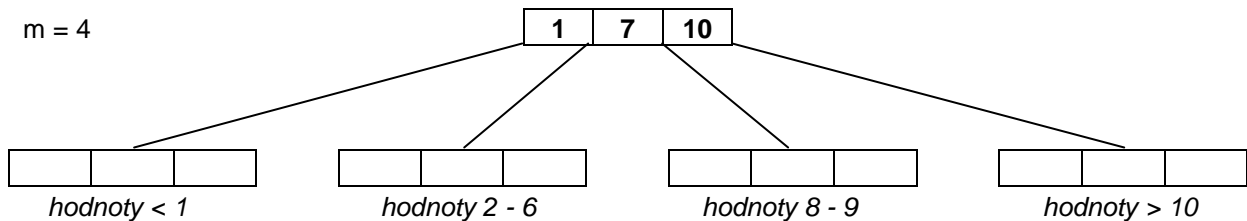
Mazání

- existuje mnoho způsobů
- mažeme R, otec je B – není problém, B hloubka se nemění
- mažeme B, otec je R – lze, ale následníka R je nutno přebarvit
- otec i mazaný syn je B - komplikované

M-ární stromy

- je vyhledávací strom
- má až m synů
- uzel - je listem = nemá syny
- není listem = má alespoň dva syny – NELZE mít 1 syna !!!
- okénka se plní vždy zleva
 q okýnek = $q + 1$ obsazených synů
- γ (= gama) = **index zaplnění** – vyjadřuje aktuální zaplnění "chlíveček"
- γ (= gama) = $m - 1$ = max. počet zaplnění

$m = 4$



Vyhledávání

- podobné jako v BVS
- složitost $O((\log_m n) + O(m))$
- nejprve se prohledá kořen - klíč == x – nalezeno
- klíč > x – nutno prohledat syna předchozího klíče

Vkládání

- najdeme správný list a hodnotu do něj vložíme
-

B stromy

= M-ární vyhledávací stromy, kde platí

- každý uzel je zaplněn alespoň $\lfloor m/2 \rfloor \rightarrow \lfloor m/2 \rfloor + 1$ (tj. z jedné poloviny)
- výjimkou je kořen – může mít jen 1 hodnoty u 2 syny
- všechny listy jsou v jedné úrovni, tj. mají stejnou hloubku
- výjimkou je kořen, který může mít jen 2 syny
- využití – file system, rychlé vyhledávání

Vyhledávání

- stejné jako u m-árních stromů, tj. $O(\log n)$, resp. $O(\log_m n)$

Vkládání

- jsou dvě metody, které určují pořadí štěpení - top-down - bottom-up
- použití metod zpravidla podle toho, zda je m sudé nebo liché
- štěpí se vždy od prostředka – z posloupnosti 3 čísel se vybere tzv. *medián*

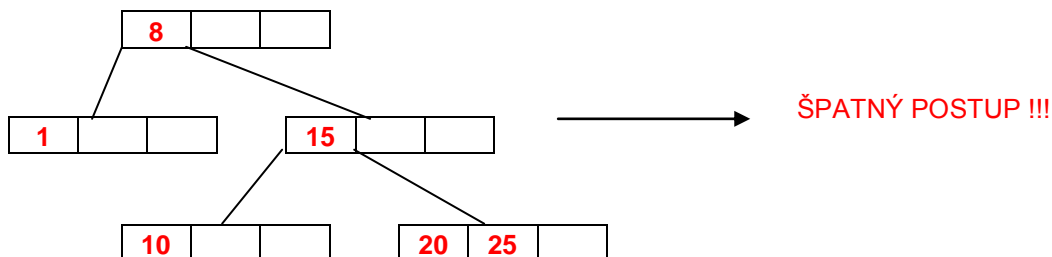
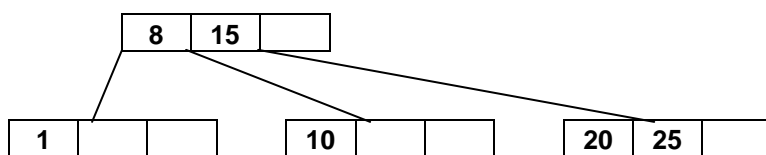
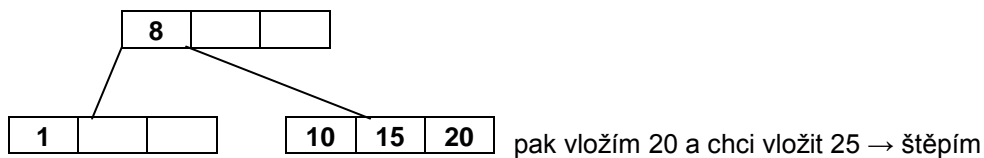
Top down

- štěpí vždy, jak narazí na plný uzel
- m je zpravidla sudé, tzn. m je lichá
- POZOR, pokud je místo v synech, ale zaplníme kořen, pak při vkládání metodou top-up dojde hned ke štěpení!!! To je ale chyba!!!
- štěpí od kořene dolů

$m = 4 \rightarrow$ lichý počet klíčů, protože $\gamma = m - 1$, může mít max 4 syny v jednom patře

1	8	10
---	---	----

 chci vložit 15 \rightarrow rozštěpím kořen



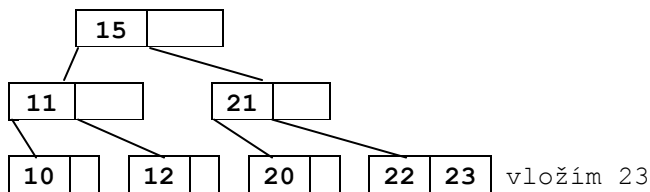
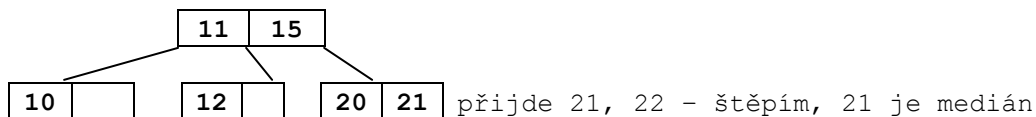
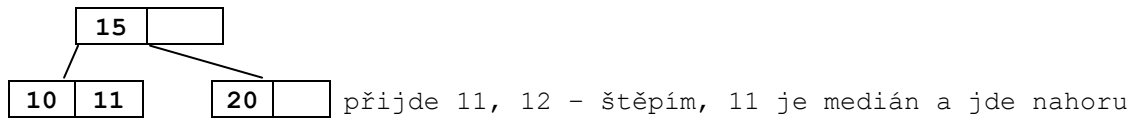
Bottom up

- m je zpravidla liché, tzn. Gama je sudá
- štěpí od kořene nahoru

$m = 3 \rightarrow$ sudý počet klíčů, protože $\gamma = m - 1$, může mít max 3 syny v jednom patře

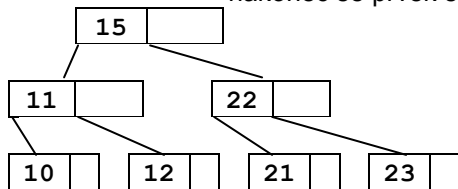
10	20
----	----

 vkládám 15, 15 je medián posloupnosti = vysuneme nahoru

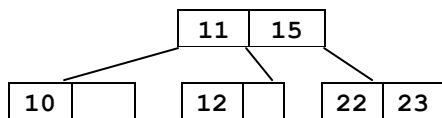


Mazání

- 22 nebo 23 – OK, řeší se v rámci syna
- 20 - půjčím si prvek z L syna, pokud nemá tak z P syna
 - zrotujeme = sloučím syny + kořen
 - nakonec se prvek smaže



- mažu 21



Implementace

N-arní strom

```
public class Tree {
    private ArrayList<Tree> children;           //Kolekce potomku
    private Object value;                     //Hodnota uchovavana v uzlu
}
```

Prida podstrom jako potomka tohoto kořene

```
@param t strom k pridani
@param index poradi potomka
public void addChild(Tree t, int index){
    children.add(index, t);
}
```

Odstrani potomka na danem indexu

```
    @param index index potomka k odstraneni
public void removeChild(int index){
    children.remove(index);
}
```

Vrati potomka na danem indexu

```
    @param index poradi potomka
    @return potomek na zadanem indexu
public Tree get(int index){
    return children.get(index);
}
```

*** Vyhledání v binárním stromě**

```
Find( x - koren, k - hledaná hodnota klíče ){
    while( x != NULL && k != x->klíč ){
        if ( k < x->klíč )
            x = x->levý_syn;
        else
            x = x->pravý_syn;
    }
    return x;}
}
```

@return the value

```
public Object getValue() {
    this.value = value;}
}
```

@param value the value to set

```
public void setValue(Object value) {
    return value;
}
```