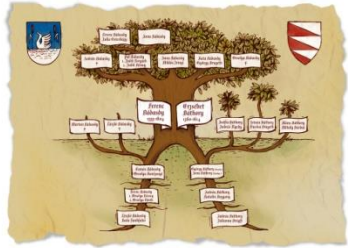




4. prednáška (6.3.2023)



Stromy

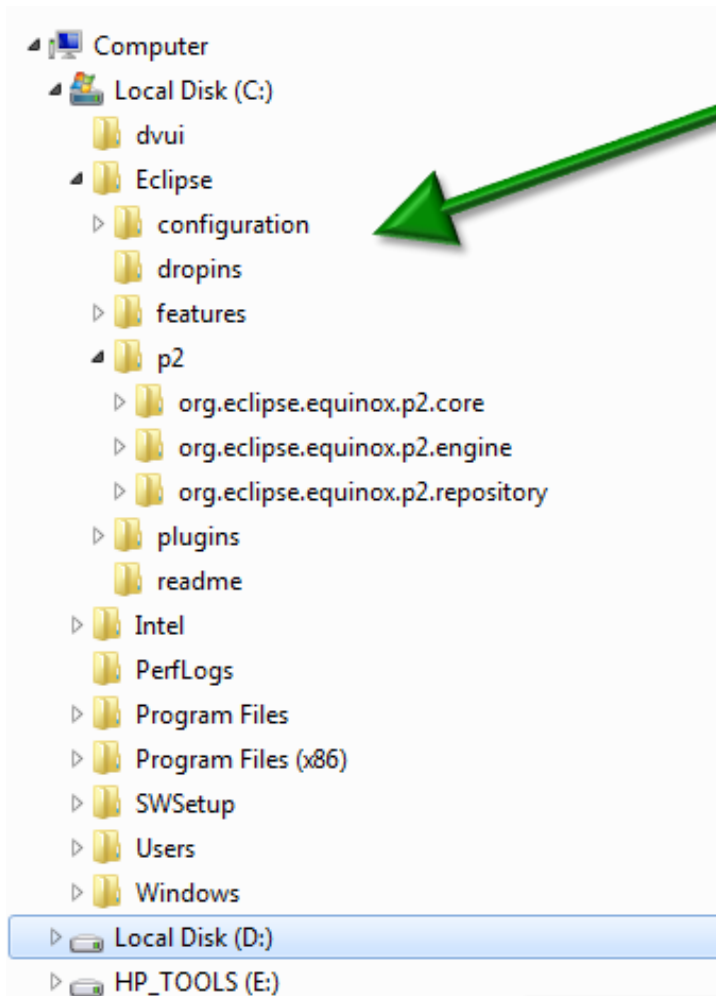
alebo

rodostromy a
tajomstvá TreeSet-u

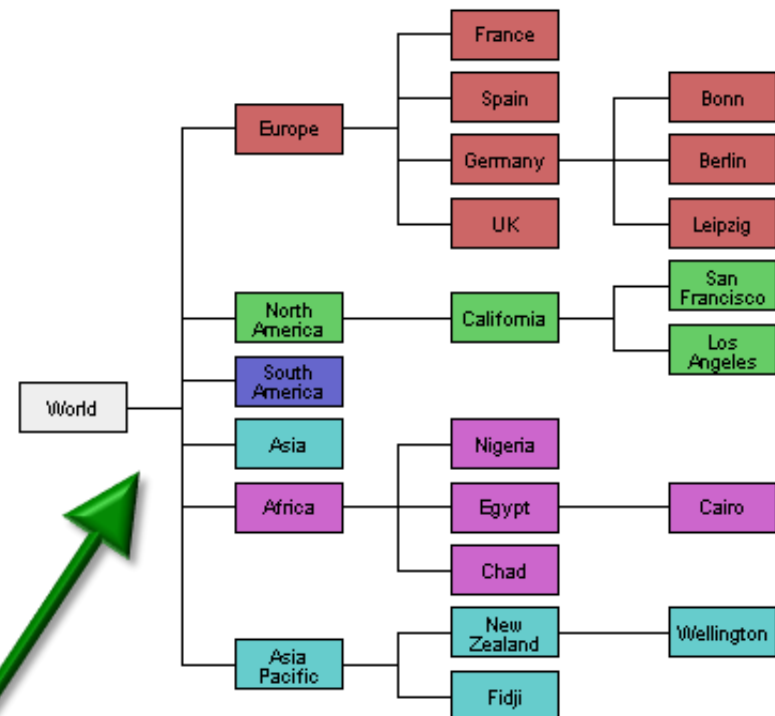




Hierarchie okolo nás



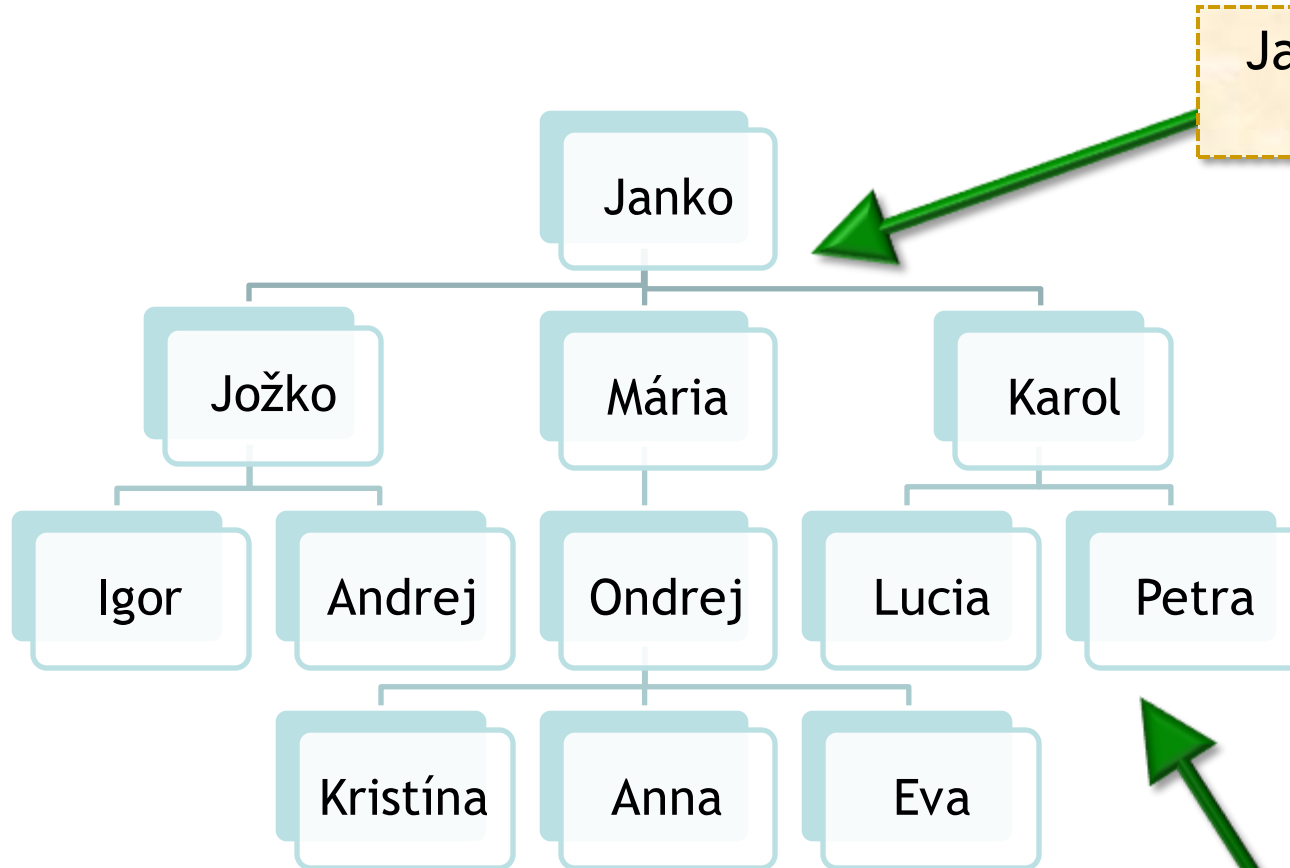
Stromové zobrazenie
adresárovej štruktúry



Hierarchický diagram



Strom potomkov



Janko má deti Jožka, Máriu a Karola.



Karol má dcéry Luciu a Petru.



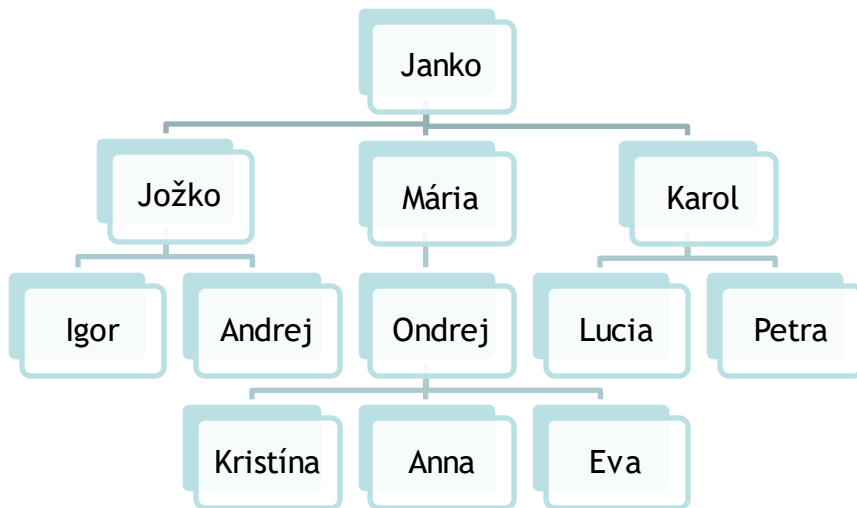
Strom potomkov v Java

```
import java.util.List;
```

```
public class Osoba {
    private String meno;
    private List<Osoba> deti;
}
```

Meno, resp. ďalšie údaje o osobe.

Zoznam referencií na osoby, ktoré sú **deťmi** tejto osoby.



Rekurzívna údajová štruktúra.
Osoba je definovaná zoznamom osôb.



Strom potomkov v Java

```
import java.util.*;

public class Osoba {
    private String meno;
    private List<Osoba> deti = new ArrayList<Osoba>();

    public Osoba(String meno) {
        this.meno = meno;
    }

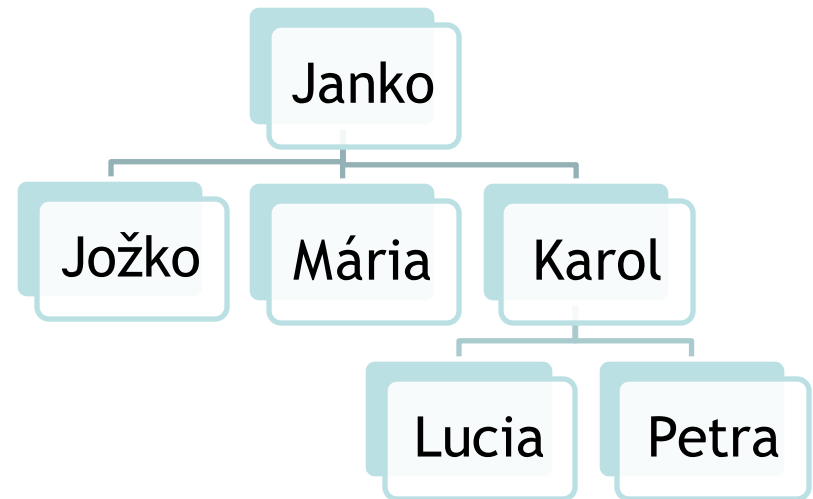
    public void pridajDieta(Osoba dieta) {
        deti.add(dieta);
    }
}
```

V takto navrhnutej údajovej štruktúre osoba pozná len svoje deti, nepozná svojho rodiča.



Strom potomkov v Java

```
public static void main(String[] args) {  
    Osoba janko = new Osoba("Janko");  
    Osoba jozko = new Osoba("Jozko");  
    Osoba maria = new Osoba("Maria");  
    Osoba karol = new Osoba("Karol");  
    Osoba lucia = new Osoba("Lucia");  
    Osoba petra = new Osoba("Petra");  
    janko.pridajDieta(jozko);  
    janko.pridajDieta(maria);  
    janko.pridajDieta(karol);  
    karol.pridajDieta(lucia);  
    karol.pridajDieta(petra);  
}
```





Genealogické otázky

- Osoby sa môžeme:
 - spýtať na celkový počet potomkov,
 - spýtať na počet generácií potomkov,
 - spýtať na zoznam všetkých potomkov,
 - spýtať na zoznam potomkov nejakej generácie,
 - ...





Poččet potomkov

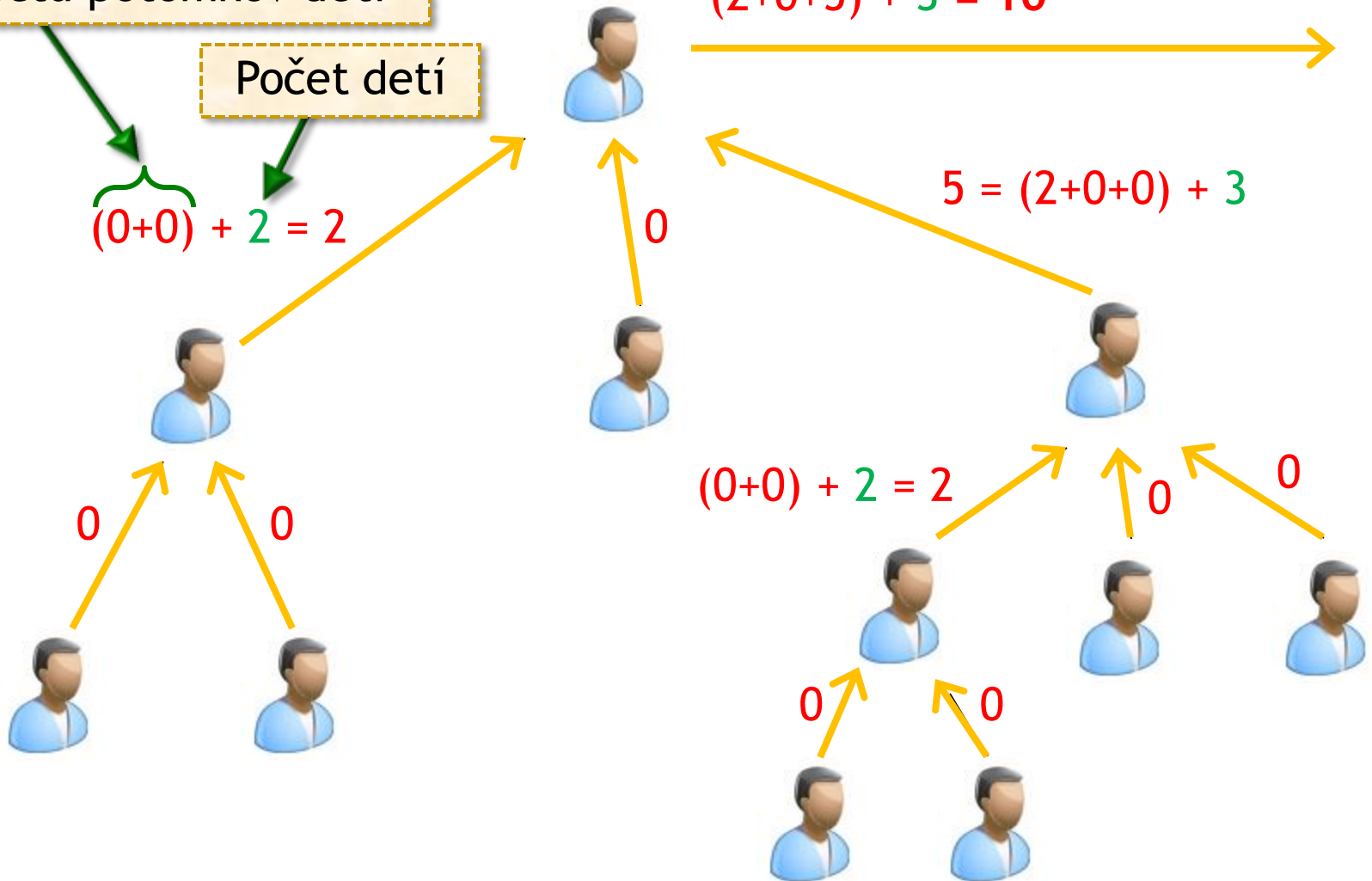
- **Potomkovia** sú deti osoby, ich deti, ich detí deti, ich detí detí deti, ich detí detí detí deti, ...
- **Obmedzenie štruktúry:** Osoba pozná len svoje deti, nie vnúčatá a ďalších potomkov, ...!
- **Algoritmus:**
 - spýtaj sa každého dieťaťa na počet potomkov
 - počet potomkov osoby je:
počet potomkov detí + počet detí



Počet potomkov

Súčet počtu potomkov detí

Počet detí





Počet potomkov (v Jave)

Spýtame sa detí na počty potomkov a sčítame ich odpovede.

```
public int pocetPotomkov() {  
    int pocetPotomkovDeti = 0;  
    for (Osoba dieta: deti)  
        pocetPotomkovDeti += dieta.pocetPotomkov();  
  
    return pocetPotomkovDeti + deti.size();  
}
```

Metódu `pocetPotomkov` voláme nad iným objektom. Je to rekurzia?

Výsledná odpoveď je **súčet** počtu potomkov detí a **počet detí**



Výpis rodostromu osoby

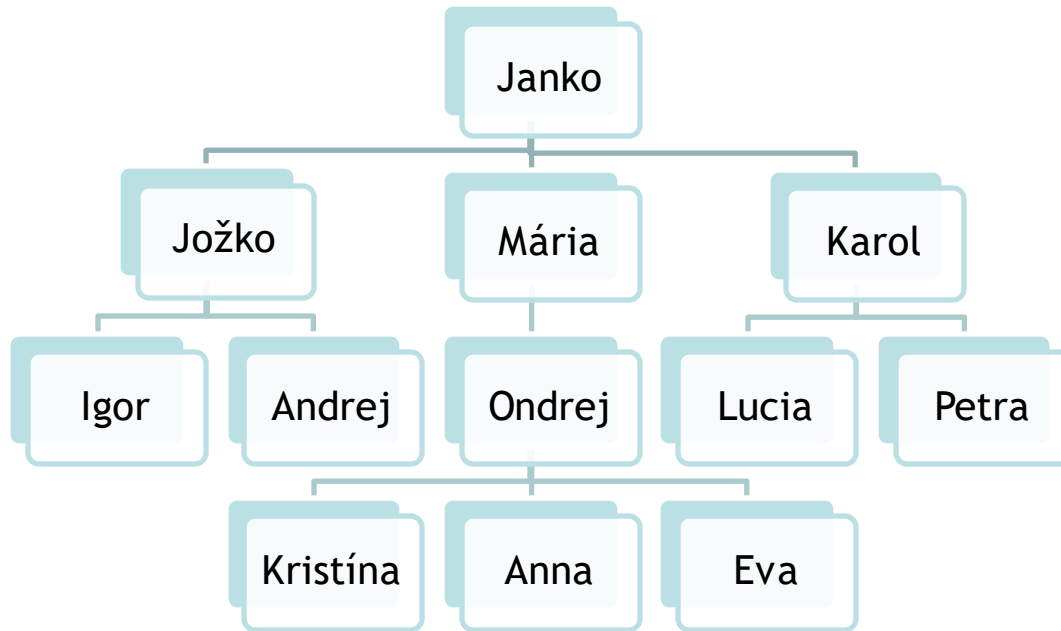
● Algoritmus:

- vypíšeme meno osoby
- požiadame deti, aby vypísali svoje rodostromy

```
public void vypisRodostrom() {  
    System.out.println(meno);  
  
    for (Osoba dieta: deti)  
        dieta.vypisRodostrom();  
}
```



Výpis rodostromu osoby



janko.vypisRodostrom();

Janko

Jožko

Igor

Andrej

Mária

Ondrej

Kristína

Anna

Eva

Karol

Lucia

Petra

```

public void vypisRodostrom() {
    System.out.println(meno);

    for (Osoba dieta: deti)
        dieta.vypisRodostrom();
}
  
```



Základná schéma

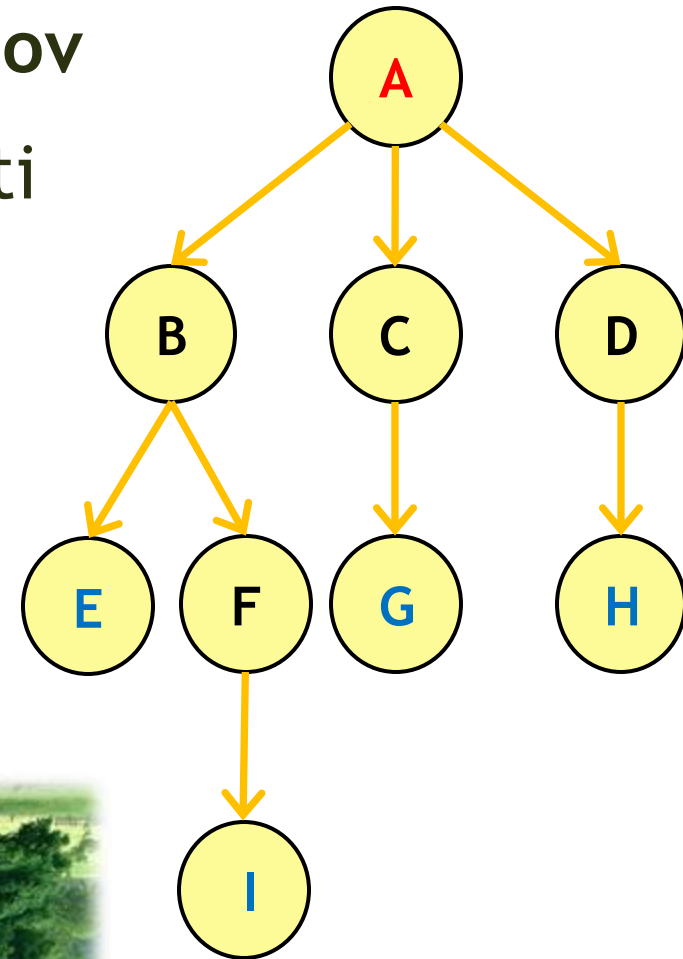
- Základná schéma vykonania *akcie* objektu **Osoba**:
 - **priprav sa** na vykonanie *akcie* u detí (ak treba)
 - **požiadaj deti** (postupne po jednom) o vykonanie *akcie*
 - **na základe výsledkov** akcií **detí** dokonči svoju *akciu* (ak treba)

```
public int pocetPotomkov() {  
    int pocetPotomkovDeti = 0;  
    for (Osoba dieta: deti)  
        pocetPotomkovDeti += dieta.pocetPotomkov();  
  
    return pocetPotomkovDeti + deti.size();  
}
```



Stromová terminológia

- Strom sa skladá z vrcholov/uzlov
- Uzol môže, ale nemusí mať deti
- Špeciálny uzol na vrchole hierarchie: **koreň**
- Uzly delíme na:
 - **vnútorné uzly** - uzly, ktoré majú aspoň jedno dieťa
 - **listy** - uzly bez detí





Stromová terminológia

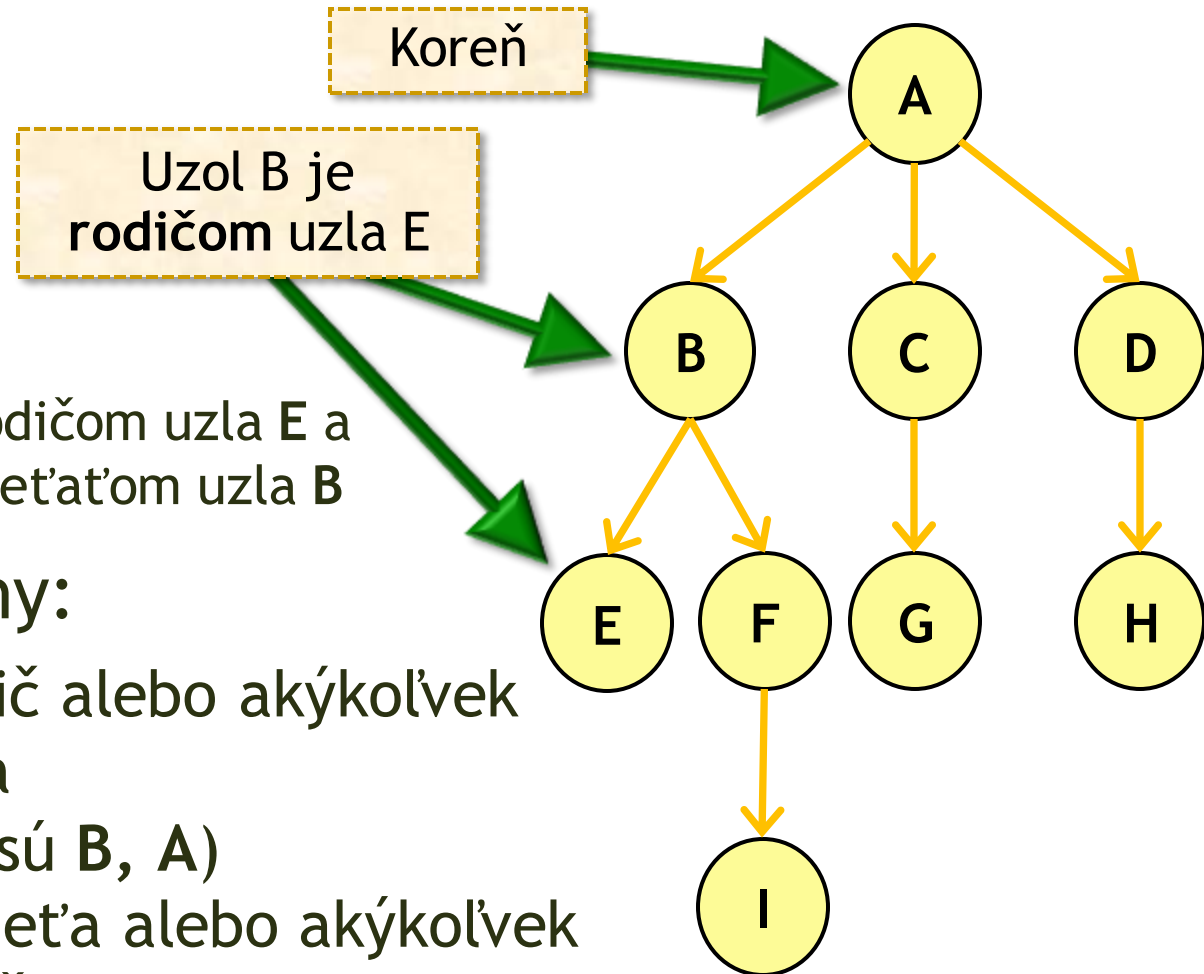
- Priame vzťahy:

- rodič/dieťa

- uzol B je rodičom uzla E a
 - uzol E je dieťaťom uzla B

- Nepriame vzťahy:

- predok** = rodič alebo akýkoľvek predok rodiča
(predkovia E sú B, A)
 - potomok** = dieťa alebo akýkoľvek potomok rodiča
(potomkovia B sú E, F, I)

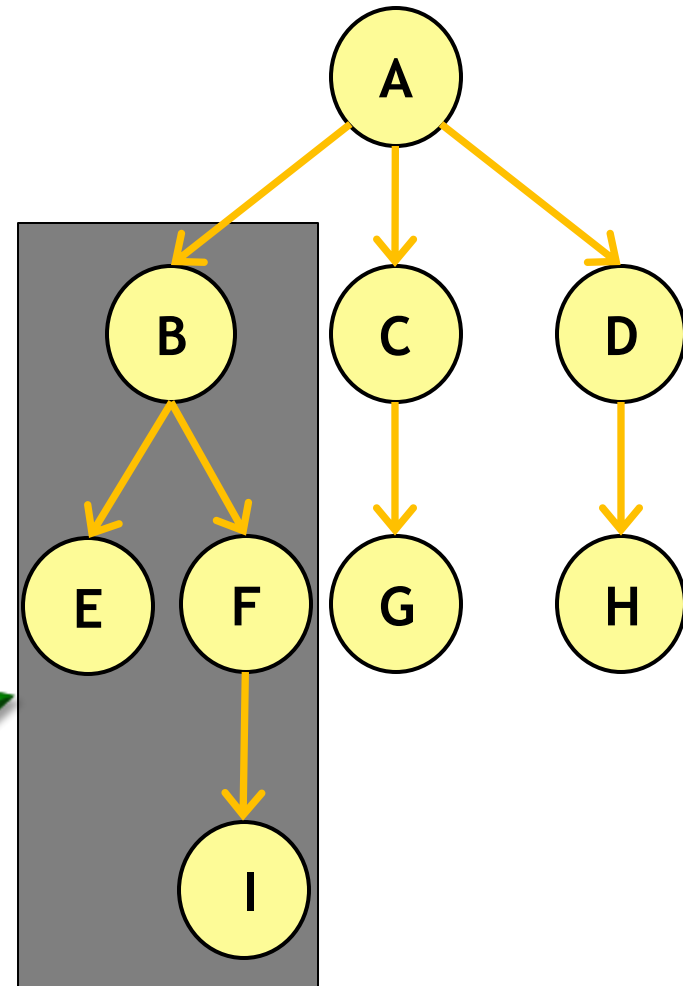




Podstromy

- Ak si vyberieme akýkoľvek uzol stromu, ten spolu so všetkými svojimi potomkami tvorí **podstrom**.
- **Rekurzia**: časťou stromu je strom

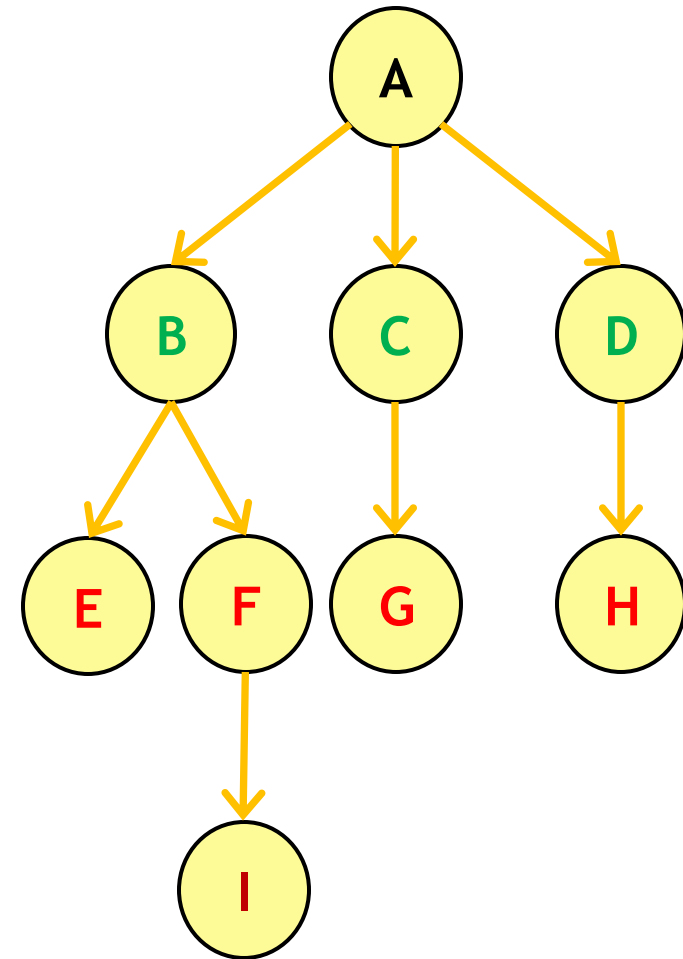
Podstrom s koreňom
v uzle B





Stromová terminológia

- **Výška stromu** - 3
 - počet generácií potomkov koreňa stromu
 - maximálna vzdialenosť nejakého uzla od koreňa stromu
- **Úroveň stromu**
 - uzly stromu v rovnakej vzdialenosti od koreňa
- **Šírka stromu** - 4
 - maximum z počtu uzlov v jednej úrovni





Stromy - sumarizácia

- Strom je **rekurzívna** štruktúra:
 - koreň
 - podstromy zakorenené v deťoch koreňa
- **Rekurzívna údajová štruktúra** = rekurzívne algoritmy
- Z koreňa sa vieme dostať do ľubovoľného uzla stromu (jednosmerná cesta)
- Porovnanie so **spájanými zoznamami**:
 - uzol spájaného zoznamu = 0 alebo jeden nasledovník
 - uzol stromu = 0 alebo veľa nasledovníkov
- Aplikácie: súborový systém, hierarchie, ...



„Politika dvoch detí“



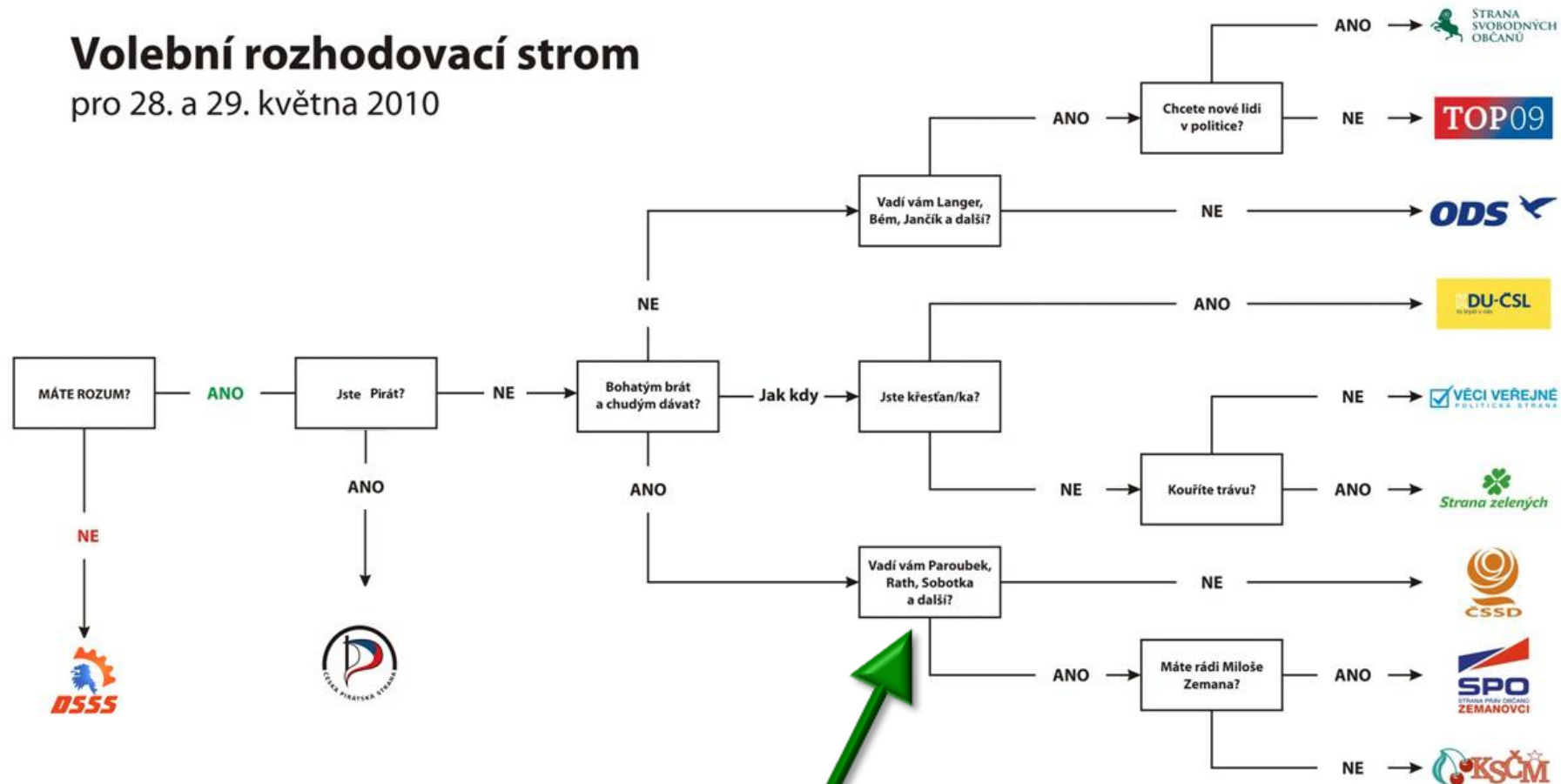
Pre mnohé aplikácie stromov stačí limit dvoch detí pre každý uzol stromu ...



Ked' (skoro) dve deti stačia

Volební rozhodovací strom

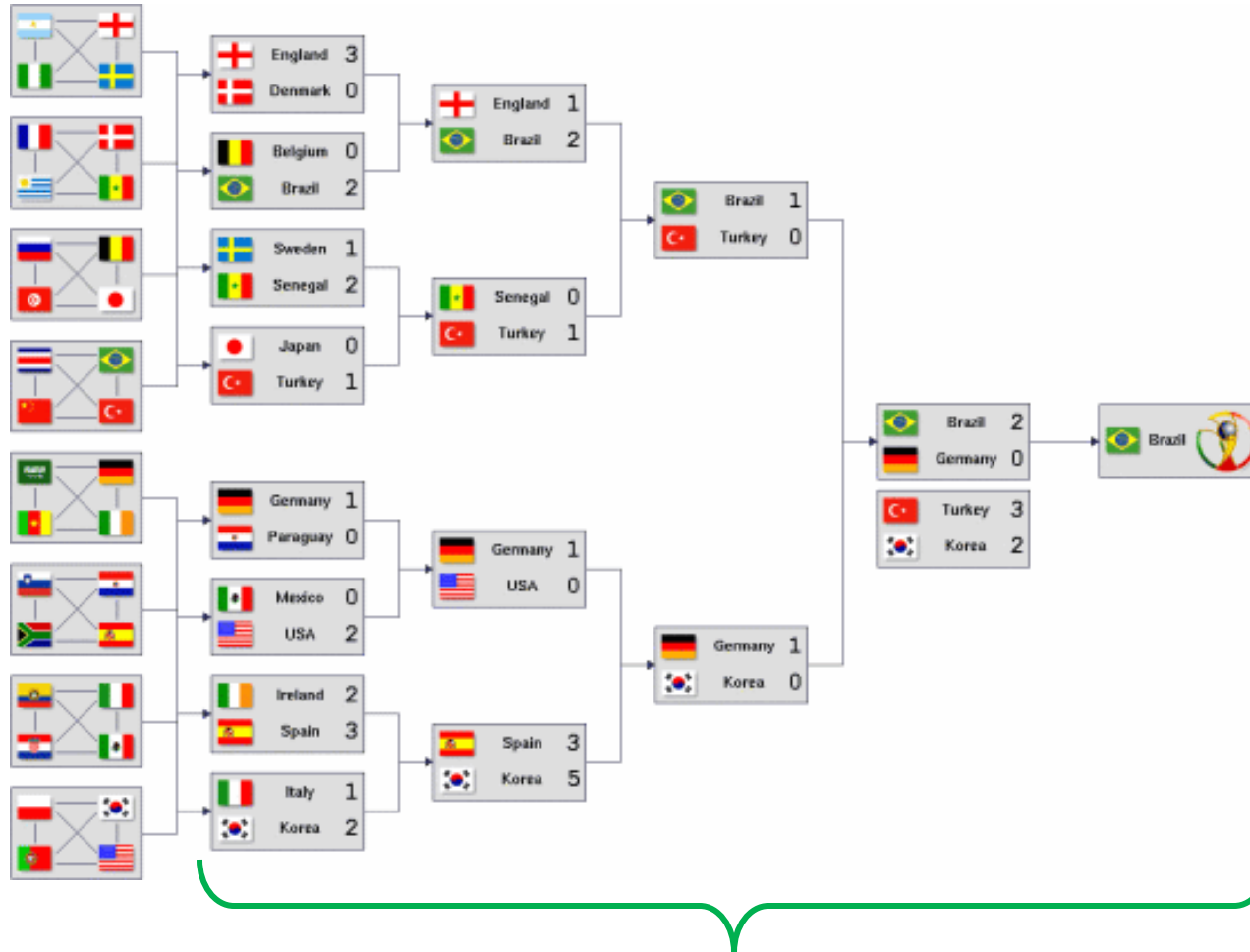
pro 28. a 29. května 2010



Rozhodovací strom:
Áno/Nie pokračovanie



Ked' dve deti stačia



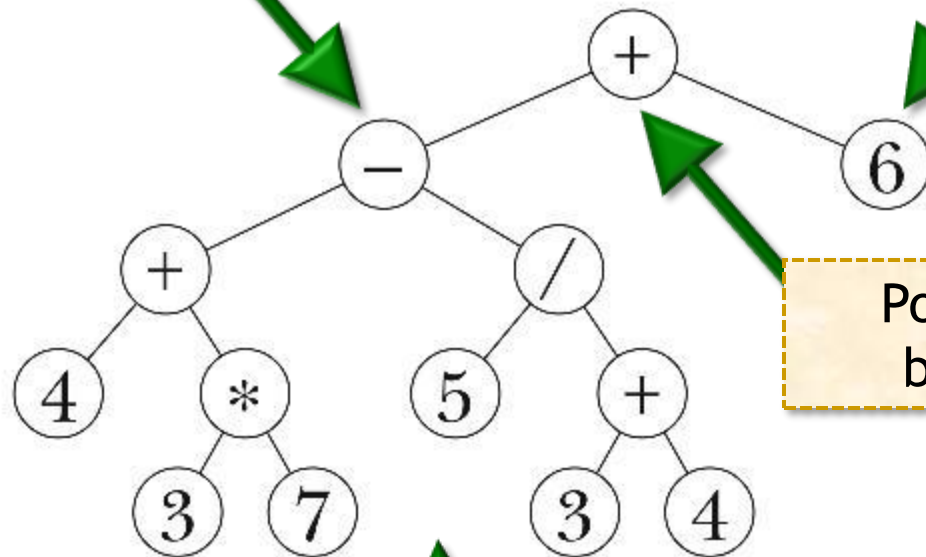
**Strom zápasov v turnaji
(„play-off“ časť)**



Ked' dve deti stačia

Pravý podvýraz

Ľavý podvýraz



Posledná vykonaná
binárna operácia

Aritmetický strom:

$((4+(3*7))-(5/(3+4)))+6$



Binárne stromy

- Binárne stromy sú špeciálnym prípadom stromov, v ktorých má každý uzol **nanajvýš 2 deti**:

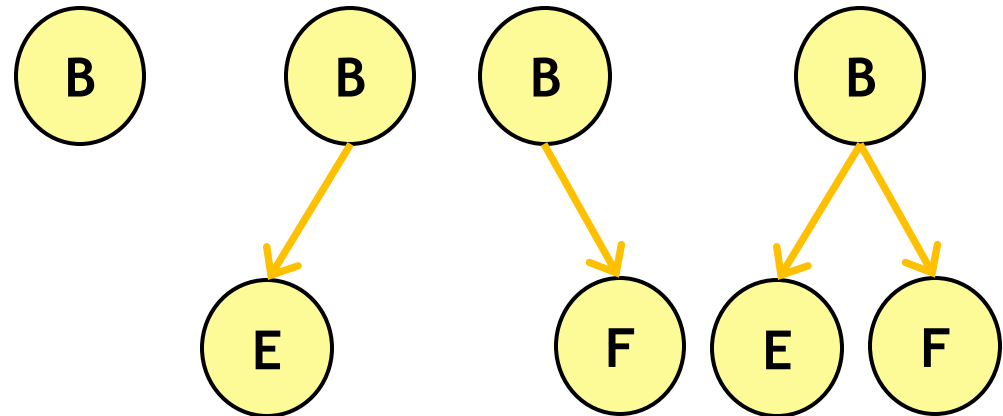
- **ľavý syn**
- **pravý syn**

Vo väčšine aplikácií má zmysel rozlišovať synov (napr. áno/nie)...

- Možné „stavy“ uzla:

- žiadny syn
- len ľavý syn
- len pravý syn
- ľavý aj pravý syn

E - ľavý syn
F - pravý syn





Binárne stromy (v Jave)

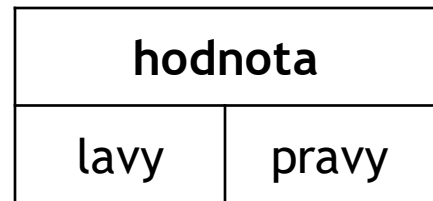
```
public class Uzo1 {
    int hodnota;

    Uzo1 lavy;
    Uzo1 pravy;
}
```

Hodnota uložená v uzle

Referencia na ľavého syna

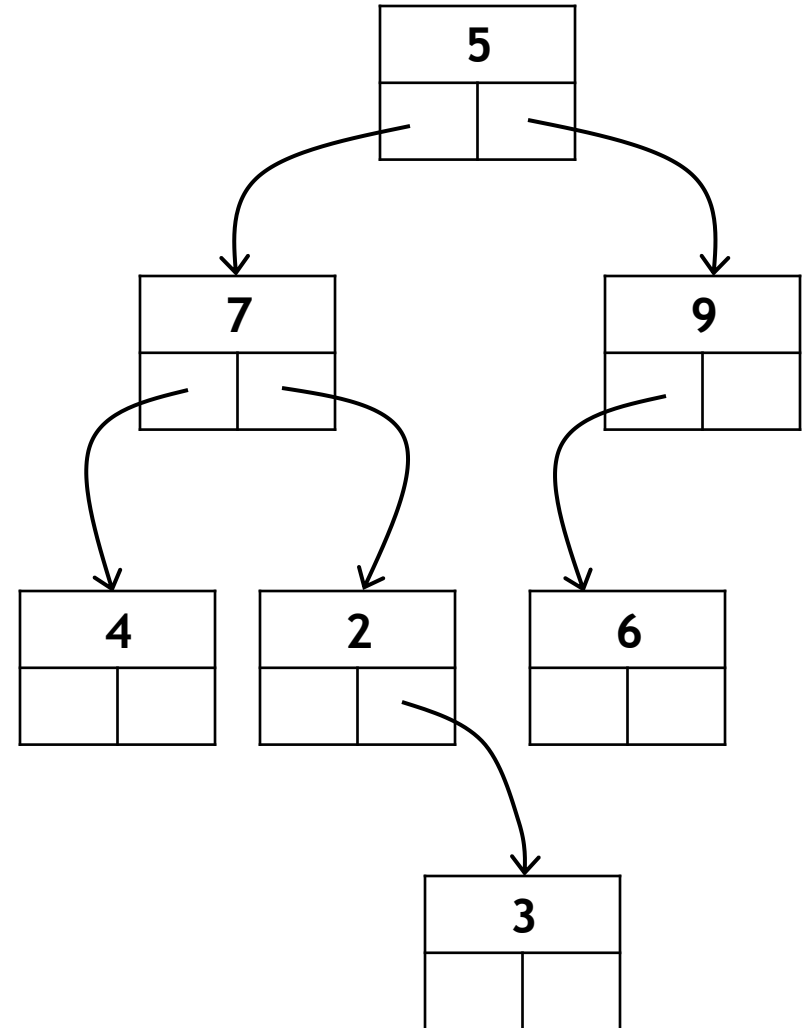
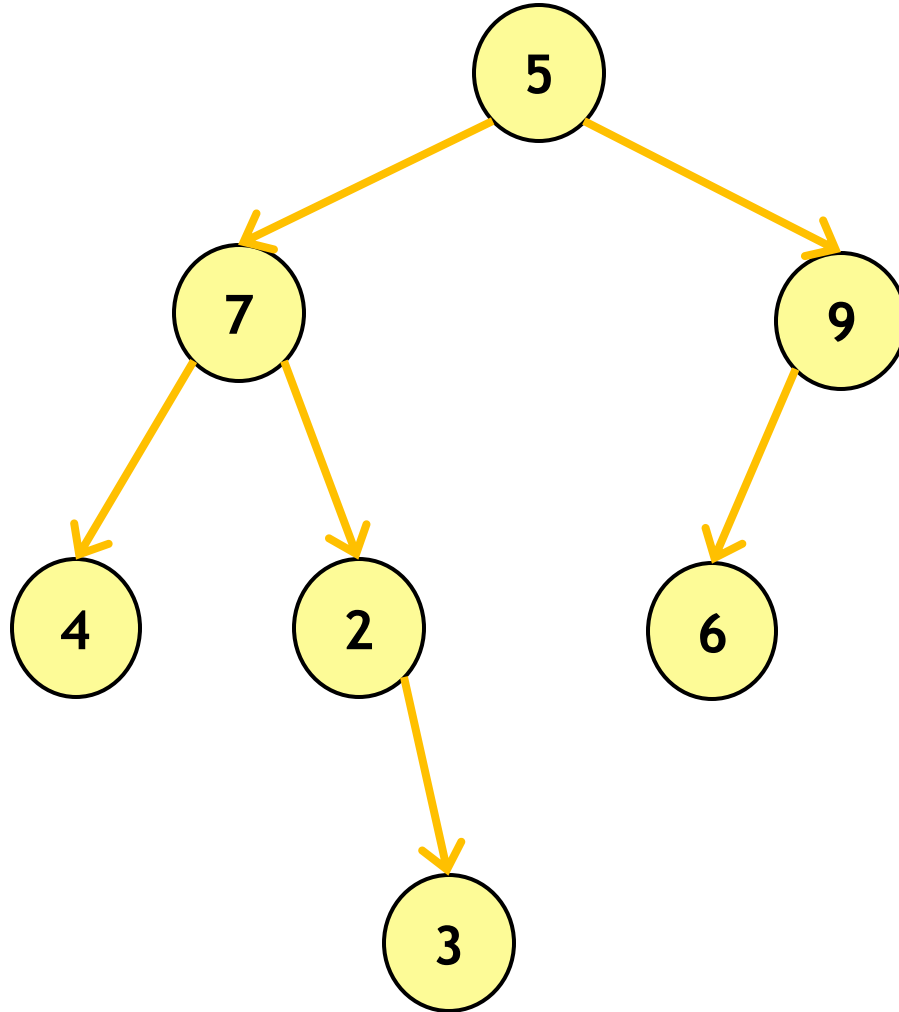
Referencia na pravého syna



Základná stavebná jednotka binárneho stromu



Binárne stromy





Čo so stromami?

- Strom **uchováva údaje** v uzloch (zvyčajne je to viac ako jedna hodnota), našim cieľom je s týmito údajmi niečo spraviť:
 - výpis údajov
 - aritmetický výraz (infixová vs. postfixová notácia), odohrané zápasy, všetky otázky rozhodovacieho stromu
 - zistiť niečo ...
 - zápas, v ktorom sa dalo najviac gólov
 - najčastejší výsledok rozhodovacieho stromu
 - zistiť, či aritmetický výraz obsahuje konkrétnu operáciu alebo číselnú hodnotu



Prechody binárnym stromom

- Na to, aby sme sa dostali k ľubovoľnému vrcholu stromu, stačí **referencia na koreň**
- **Problém:** ako systematicky navštíviť všetky hodnoty (resp. uzly) stromu?
- **Rekurzívna idea:**
 - spracuj hodnotu uloženú v aktuálnom uzle
 - navštív hodnoty v podstrome zakorenenom v ľavom synovi
 - navštív hodnoty v podstrome zakorenenom v pravom synovi



Výpis hodnôt v binárnom strome

```
public void vypis() {  
    System.out.println(hodnota);
```

Spracovanie hodnoty v
aktuálnom uzle

```
    if (lavy != null)  
        lavy.vypis();
```

```
    if (pravy != null)  
        pravy.vypis();
```

```
}
```

Spracovanie hodnôt
uložených najprv v
ľavom a potom v
pravom podstrome.

Pozor: Pred prístupom k
synovi musíme otestovať
jeho existenciu!



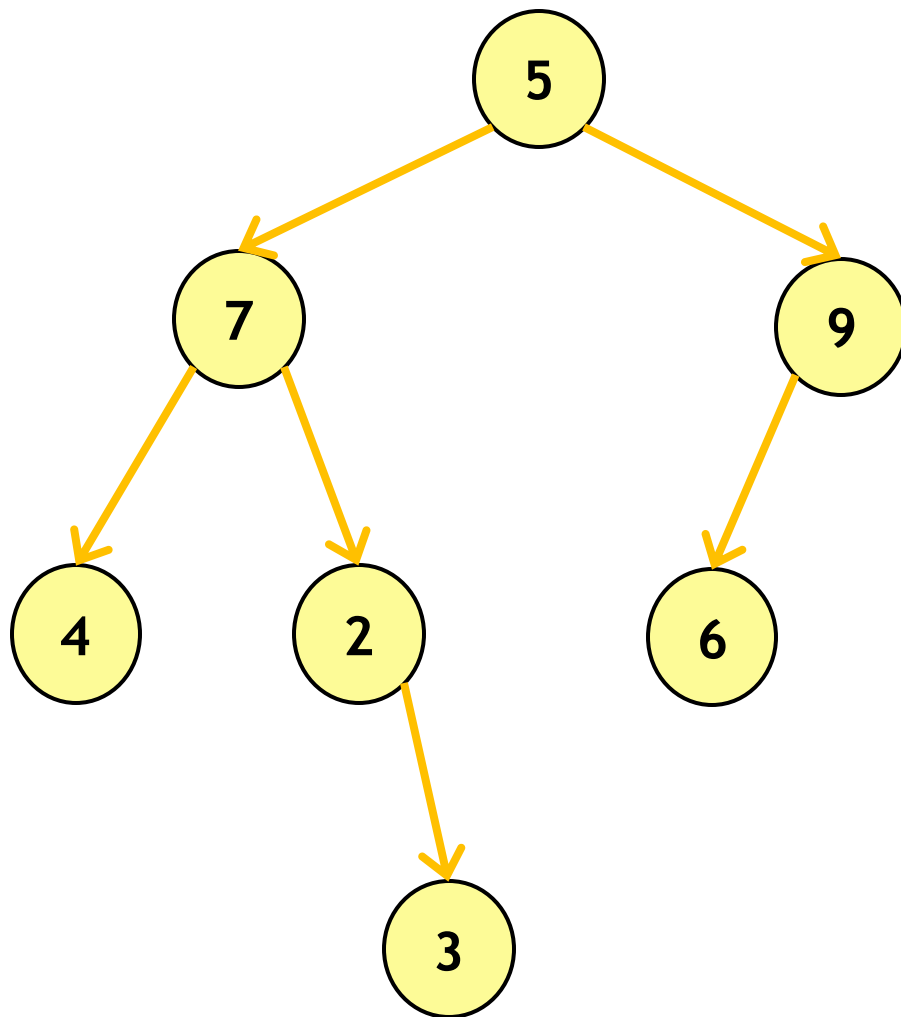
Prechody binárnym stromom

- Podľa toho, kedy spracujeme hodnotu v uzle, rozlišujeme niekoľko prechodov:
 - **Preorder:**
 - **uzol**, ľavý podstrom, pravý podstrom
 - **Inorder:**
 - ľavý podstrom, **uzol**, pravý podstrom
 - **Postorder:**
 - ľavý podstrom, pravý podstrom, **uzol**





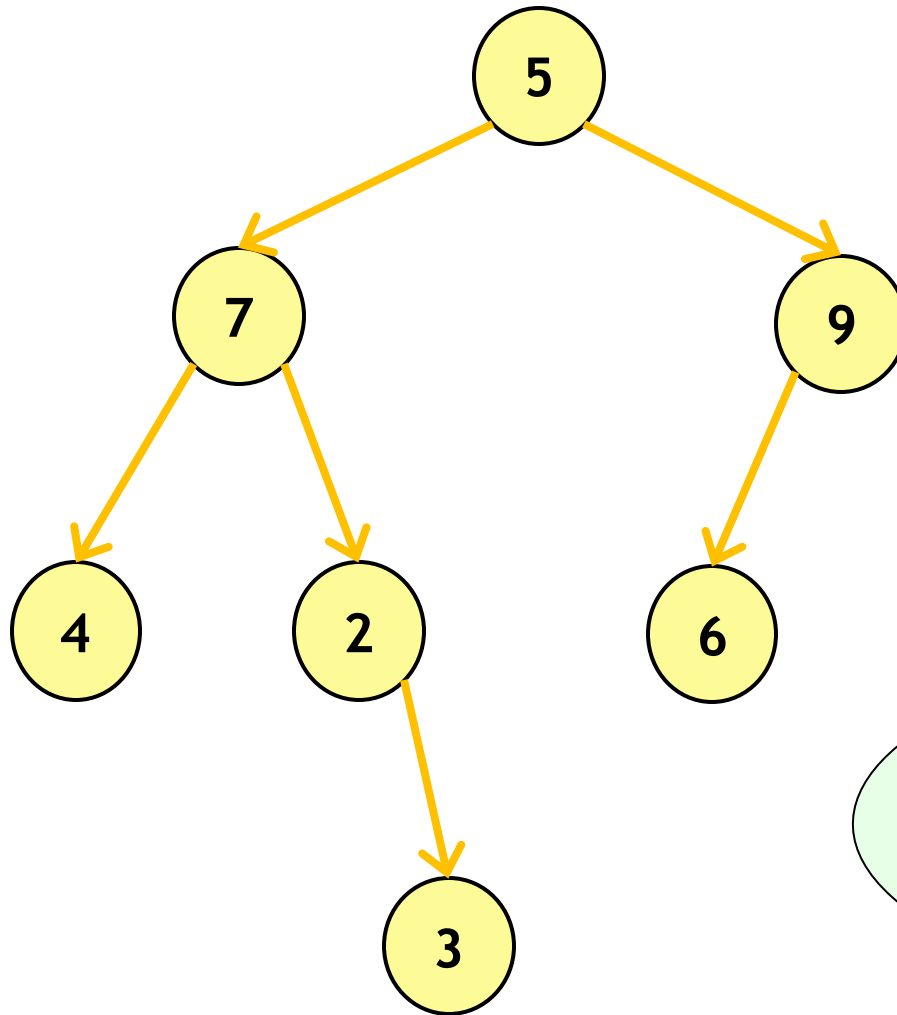
Príklady prechodov



| Preorder | Inorder | Postorder |
|----------|---------|-----------|
| 5 | 4 | 4 |
| 7 | 7 | 3 |
| 4 | 2 | 2 |
| 2 | 3 | 7 |
| 3 | 5 | 6 |
| 9 | 6 | 9 |
| 6 | 9 | 5 |



Príklady prechodov



Preorder:

5, 7, 4, 2, 3, 9, 6

Inorder:

4, 7, 2, 3, 5, 6, 9

Postorder:

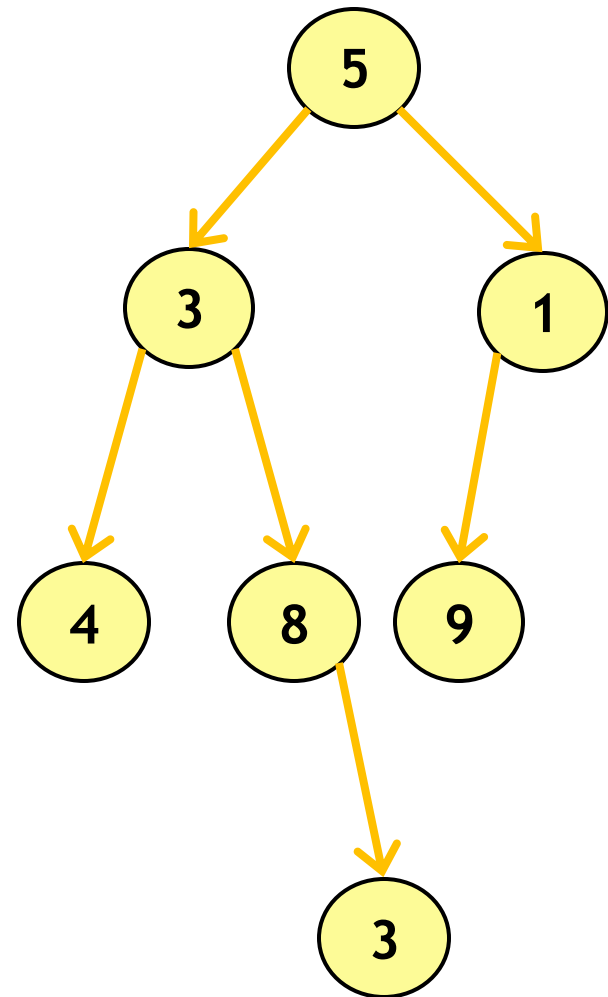
4, 3, 2, 7, 6, 9, 5

Ako z dvoch postupností
prechodu stromom
zrekonštruovať binárny
strom?



Výpočet maxima v strome

- **Idea** (na začiatku sa pýtame koreňa):
 - ak má uzol ľavého syna, požiadame ho o maximum v ňom zakorenenom podstrome
 - ak má uzol pravého syna, požiadame ho o maximum v ňom zakorenenom podstrome
 - odpoveďou je maximum z uloženej hodnoty a maxim od synov





Výpočet maxima v strome

```
public int maximum() {  
    int vysledok = hodnota;  
  
    if (lavy != null)  
        vysledok = Math.max(vysledok, lavy.maximum());  
  
    if (pravy != null)  
        vysledok = Math.max(vysledok, pravy.maximum());  
  
    return vysledok;  
}
```

- Časová zložitost': Každý uzol navštívime len raz, v každom uzle strávime operáciami čas $O(1)$. Celkový čas je $O(n)$ v strome s n uzlami.



Hľadanie hodnoty v strome

```

public boolean obsahuje(int hľadany) {
    if (hodnota == hľadany)
        return true;

    if (lavy != null)
        if (lavy.obsahuje(hľadany))
            return true;

    if (pravy != null)
        if (pravy.obsahuje(hľadany))
            return true;

    return false;
}

```

Ak je ľavý syn, skúsime nájsť hodnotu v ľavom podstrome.

Ak je pravý syn, skúsime nájsť hodnotu v pravom podstrome.

- Časová zložitosť: $O(n)$, kde n je počet uzlov stromu



„Mashup nápad“

- Čo už vieme:
 - **spájaný zoznam** - dobrá vec na uchovávanie dynamicky sa meniaceho zoznamu hodnôt
 - **binárne vyhľadávanie** - superrýchla ($O(\log n)$) stratégia na nájdenie prvku v usporiadanom zozname
- usporiadaný spájaný zoznam + binárne vyhľadávanie?
 - **nefunguje** - len na to, aby sme sa dostali na zadaný index (napr. do stredu), potrebujeme čas $O(n)$

Nedá sa s tým niečo spraviť?



Čo je binárne vyhľadávanie?

Hľadaná hodnota: **H**

Pozri na hodnotu **X**
v strede zoznamu

ak **H** < **X**

ak **X** < **H**



hľadaj **H** „vľavo“

hľadaj **H** „vpravo“

Referencia na „ľavú“
časť usporiadaného
zoznamu

Referencia na „pravú“
časť usporiadaného
zoznamu



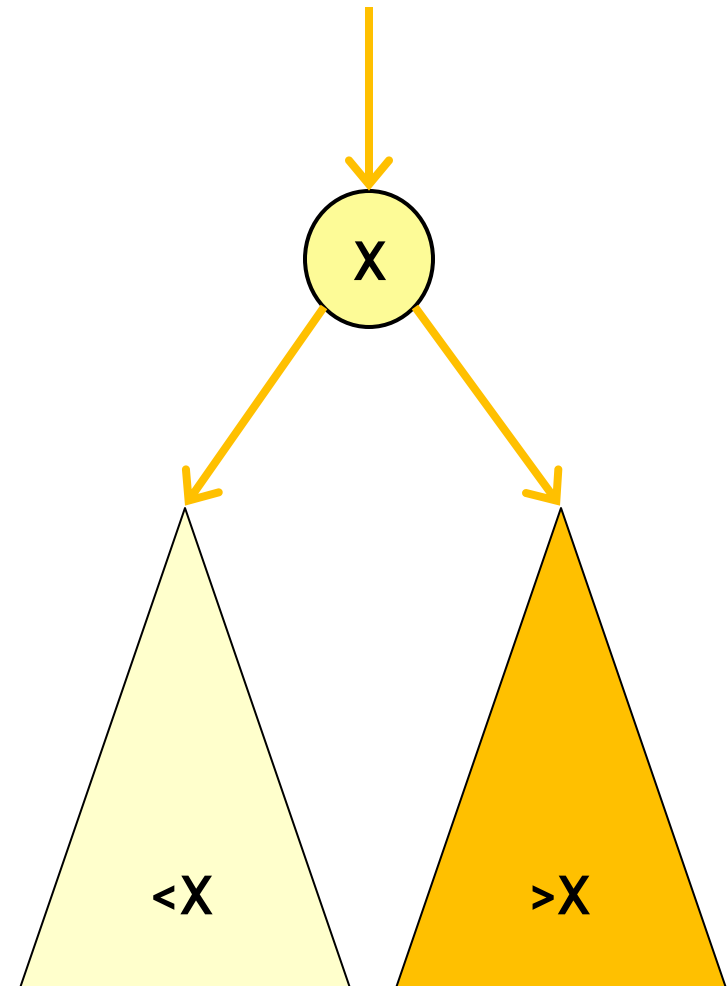
Binárne vyhľadávacie stromy (BVS)

- **Binárne vyhľadávacie stromy** slúžia na uloženie dynamicky sa meniacej **množiny hodnôt** typu, ktorého prvky môžeme navzájom porovnávať:
 - trieda `TreeSet<E>` uchováva prvky množiny interne uložené v modifikovanej verzii BVS
 - `TreeSet<E>` vyžaduje, aby `E` implementovalo `Comparable` alebo `Comparator<E>` ako porovnávač
- **Idea:** chytrým umiestnením hodnôt do uzlov binárneho stromu vieme dosiahnuť rýchlu realizáciu operácií *add*, *remove* a *contains*.



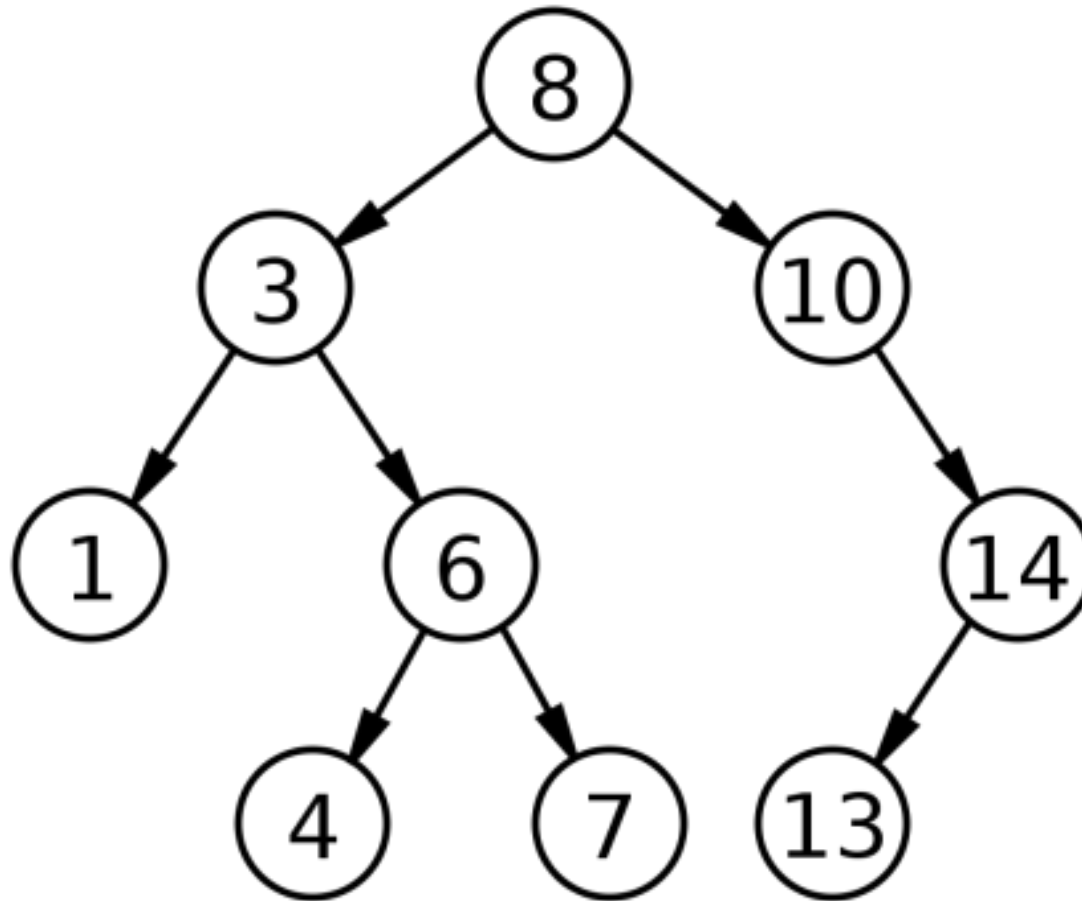
BVS - vlastnosti

- Hodnoty sú uložené v uzloch
 - každá hodnota len raz (lebo množina...)
- Pre **každý** uzol BVS platí:
 - každá hodnota uložená v **ľavom podstrome** uzla je **menšia** ako hodnota v tomto uzle
 - každá hodnota uložená v **pravom podstrome** uzla je **väčšia** ako hodnota v tomto uzle





Príklad BVS





Využitie vlastností BVS

- Pri hľadaní hodnoty nemusíme hľadať hodnotu v oboch podstromoch, stačí sa pozrieť do jedného.
- Algoritmus **hľadania hodnoty H** pre uzol s uloženou hodnotou V :
 - ak $H = V$, potom H je v BVS
 - ak $H < V$, tak v hľadaní treba pokračovať v ľavom podstrome (ak existuje, v opačnom prípade H nie je v BVS)
 - ak $H > V$, tak v hľadaní treba pokračovať v pravom podstrome (ak existuje, v opačnom prípade H nie je v BVS)



Hľadanie hodnoty v BVS

```
public boolean obsahujeVBVS(int hladany) {  
    if (hodnota == hladany)  
        return true;  
  
    if (hladany < hodnota)  
        return (lavy != null) && lavy. obsahujeVBVS(hladany);  
  
    if (hodnota < hladany)  
        return (pravy != null) && pravy. obsahujeVBVS(hladany);  
  
    return false;  
}
```



Nerekurzívne hľadanie

```
public static boolean obsahujeVBVS(Uzol koren, int hladany){
    Uzol aktualny = koren;

    while (aktualny != null) {
        if (aktualny.hodnota == hladany)
            return true;

        if (hladany < aktualny.hodnota) {
            aktualny = aktualny.lavy;
        } else if (aktualny.hodnota < hladany) {
            aktualny = aktualny.pravy;
        }
    }

    return false;
}
```

Ako ďalší uzol na navštívenie vyberieme to dieťa, kam nás pošle porovnanie **hľadanej hodnoty** a **hodnoty v aktuálnom uzle**.



Maximum v BVS

- Vďaka vlastnostiam BVS je algoritmus na nájdenie maximálnej hodnoty v BVS:
 - kým sa dá, presuň sa do pravého syna aktuálneho uzla
 - uzol, v ktorom hľadanie skončilo (nemá pravého syna), je maximum (prečo?)

```
public static int maximum(Uzol koren) {  
    Uzol aktualny = koren;  
    while (aktualny.pravy != null)  
        aktualny = aktualny.pravy;  
  
    return aktualny.hodnota;  
}
```

Kým sa dá, ideme do
pravého syna.



Pridanie hodnoty do BVS

● Problém:

- hodnotu nepridávame do BVS, ak tam už je
- uzol s vkladanou hodnotu nemôžeme „zavesiť“ hocikde, pretože môžeme narušiť vlastnosti BVS

● Algoritmus:

- tvárime sa, že hodnotu nechceme do BVS pridať, ale chceme zistiť, či sa tam nachádza
 - ak ju nájdeme, nie je čo riešiť ...
 - ak ju nenájdeme, algoritmus skončí tak, že v hľadaní budeme presmerovaní na **neexistujúci** uzol - toto je **správne miesto**, kam máme uzol s vkladanou hodnotou „zavesiť“ (prečo?)



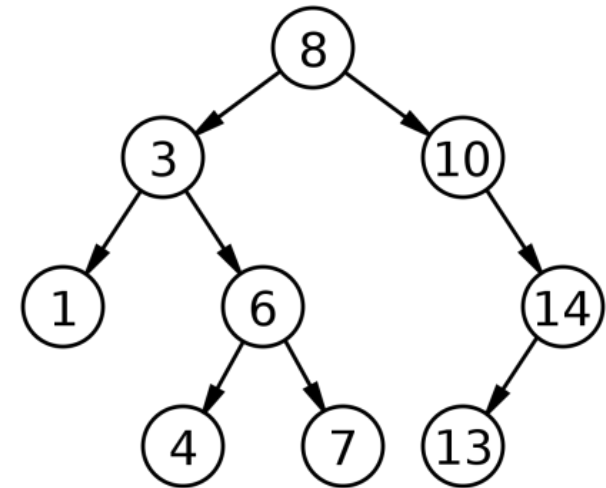
Odstránenie hodnoty z BVS

● Problém:

- uzol s odstraňovanou hodnotou treba najprv nájsť
- odstránenie uzla s hodnotou v strome môže narušiť vlastnosti BVS

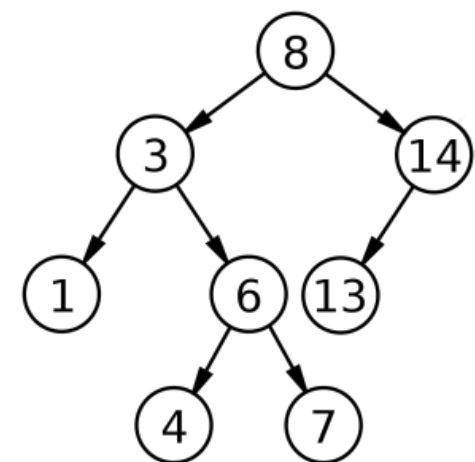
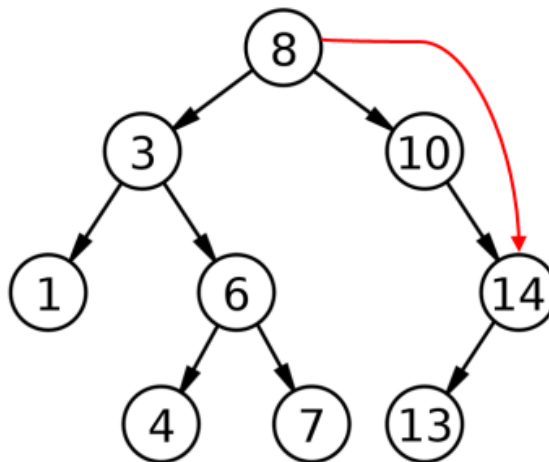
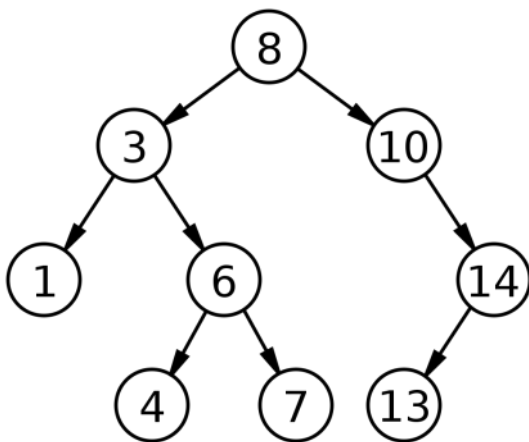
● 3 situácie:

- odstraňovaný uzol je listom
- odstraňovaný uzol má 1 dieťa
- odstraňovaný uzol má 2 deti



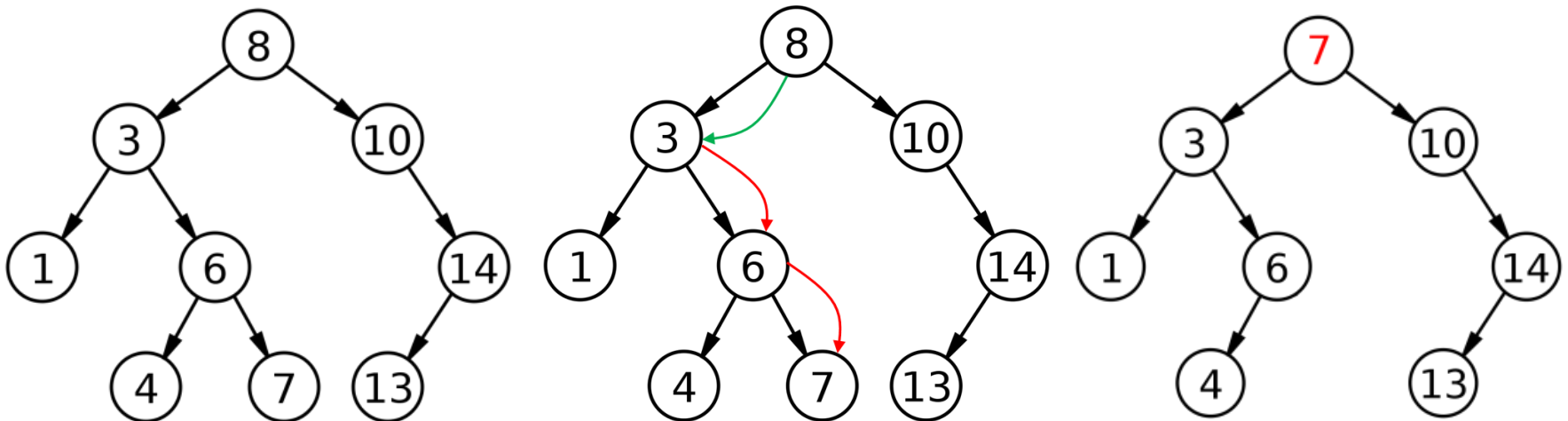
Odstránenie hodnoty z BVS

- Odstraňovaný uzol **je listom**:
 - uzol jednoducho odstránime (nie je čo pokaziť)
- Odstraňovaný uzol (10) **má 1 dieťa**:
 - odstraňovaný uzol (u jeho rodiča) nahradíme dieťaťom odstraňovaného uzla



Odstránenie hodnoty z BVS

- Odstraňovaný uzol (8) **má 2 deti**:
 - namiesto odstraňovaného uzla odstránime iný uzol
 - nájdeme **maximum v ľavom podstrome** odstraňovaného uzla (raz vľavo, potom stále vpravo kým sa dá)
 - odstránime uzol s nájdeným **maximom** a hodnotu nájdeného maxima uložíme do pôvodne odstraňovaného uzla



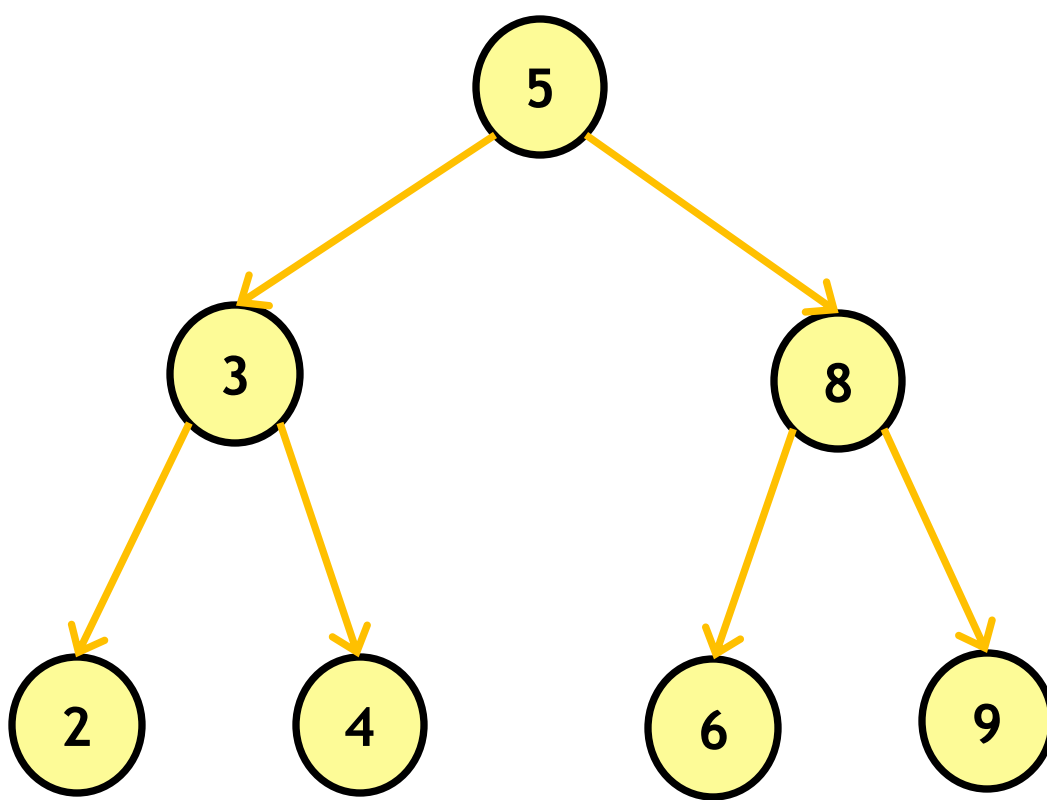


Časová zložitost' operácií

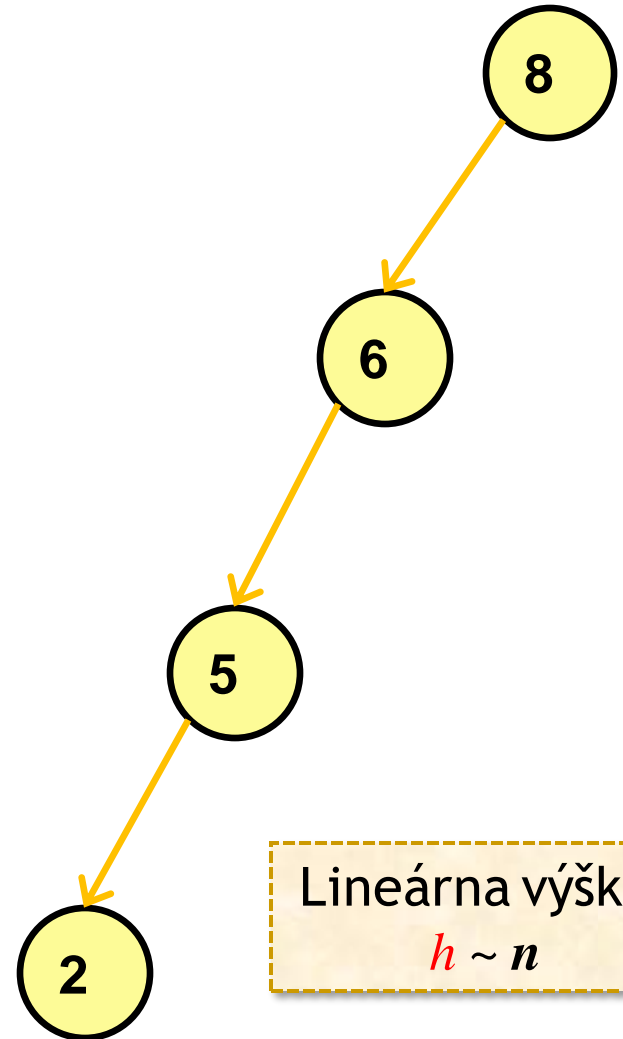
- Pri každej operácii (contains, add, remove) sa schádza od koreňa k jednému z uzlov stromu:
 - v každom uzle čas $O(1)$, navštívených je najviac $h+1$ uzlov, kde h je **výška stromu**
 - dôsledok: časová zložitost' každej z operácií je $O(h)$, kde h je výška stromu
- Ako závisí výška stromu h od počtu v ňom uložených hodnôt n ? Určite $h \leq n$.
 - Dôsledok: BVS na implementáciu množiny nemôže byť horší ako pole, kde každá z týchto operácií potrebuje $O(n)$



BVS s rôznymi výškami



Logaritmická výška:
 $h \sim \log(n)$



Lineárna výška:
 $h \sim n$



Časová zložitosť BVS

- Fakty o BVS s výškou h :
 - každá operácia v BVS trvá **v najhoršom prípade $O(h)$**
 - výška stromu závisí od postupnosti realizovaných operácií:
 - $\text{add}(1), \text{add}(2), \text{add}(3), \text{add}(4), \dots, \text{add}(n)$ vytvorí **degenerovaný BVS** s výškou $n-1$
 - pre výšku binárneho stromu s n uzlami platí:
 - **$\log(n) \leq h < n$**
- Ako dosiahnuť logaritmickú výšku?

Viac na cvičeniach



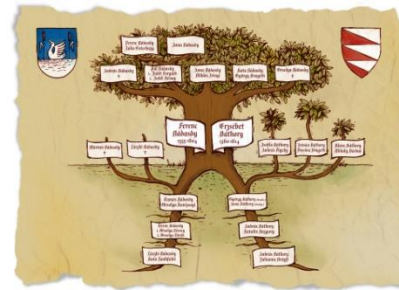
Samovyvažovacie stromy

- BVS je dobrá štruktúra, ak je strom **vyvážený**
- **Samovyvažovacie BVS**
 - **chytré algoritmy** zabezpečujúce, že ak sa naruší vyváženosť, tak sa sériou niekoľkých operácií (rotácií) strom opäť vyváži
 - časová zložitosť **opravy** narušenia **vyváženosti** je $O(h)$
 - t.j. rovnaká ako zložitosť modifikujúcej operácie
 - **AVL stromy, RB-stromy** (červeno-čierne), ...
- Trieda **java.util.TreeSet** interne ukladá hodnoty v (samovyvažovacom) **RB-strome**



Aplikácie stromov

- Strom potomkov
- Uloženie hierarchií
- Aritmetické stromy
- Binárne vyhľadávacie stromy a ich modifikácie/zovšeobecnenia
 - Samovyvažovanie stromy
 - B-stromy (na indexovanie v databázach)



ak nie sú otázky...

Ďakujem za pozornosť!

