



Binárne vyhľadávacie stromy a algoritmy na vyvažovanie

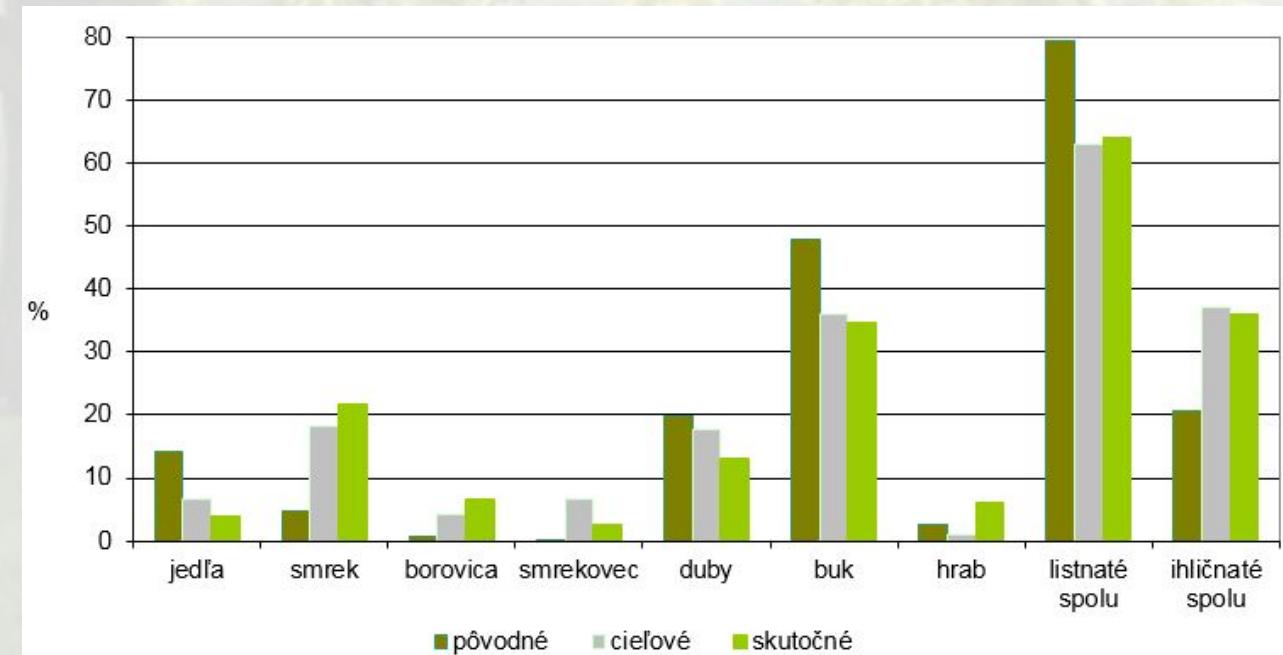
Adam Valach

FIIT STU

2021/22

Opakovanie – čo sú to stromy

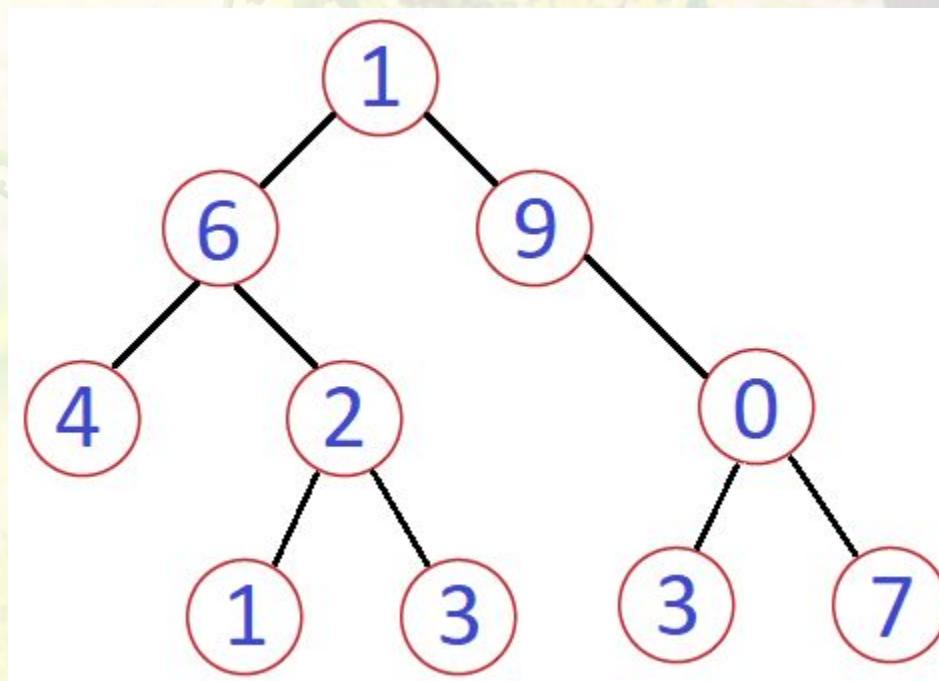
- Rastová forma cievnatých rastlín
- Radíme medzi dreviny
 - Disponujú drevnatou stonkou
- Fotoautotrofné rastliny
 - Ich výživa prebieha vďaka fotosyntéze
- Ihličnany netvoria pravé kvety



Obr. 1: Porovnanie skutočného zastúpenia vybraných drevín v lesoch SR (2020) s pôvodným a cieľovým (výhľadovým)

Opakovanie – čo sú to binárne stromy

- "Stromová štruktúra"
- Orientovaný graf s jedným "koreňovým/root" vrcholom/uzlom
- Každý vrchol má max. 2 "potomkov/child uzly"



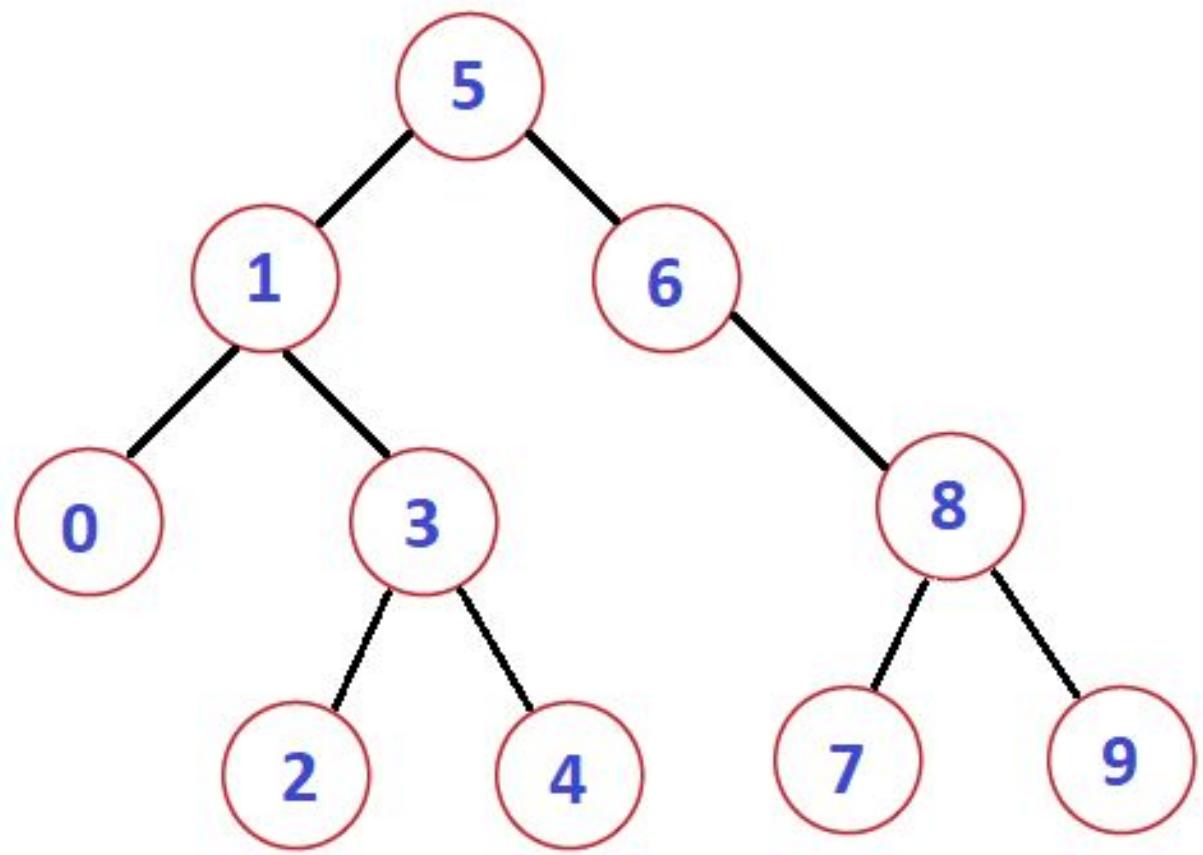
Obr. 2: Binárny stromek

Opakovanie – čo sú to binárne vyhľadávacie



- Binárne stromy, v ktorých sú prvky usporiadané podľa nejakého pravidla pre efektívnejšie vykonávanie operácií nad danou dátovou štruktúrou
 - Ľavý podstrom uzla obsahuje prvky menšie ako hodnota v ňom uložená
 - Pravý podstrom uzla obsahuje prvky väčšie ako hodnota v ňom uložená
 - Tieto pravidlá platia pre každý podstrom

Príklad binárneho vyhľadávacieho stromu #1



Obr. 3: Binárny stromek 🌳

Príklad binárneho vyhľadávacieho stromu #2

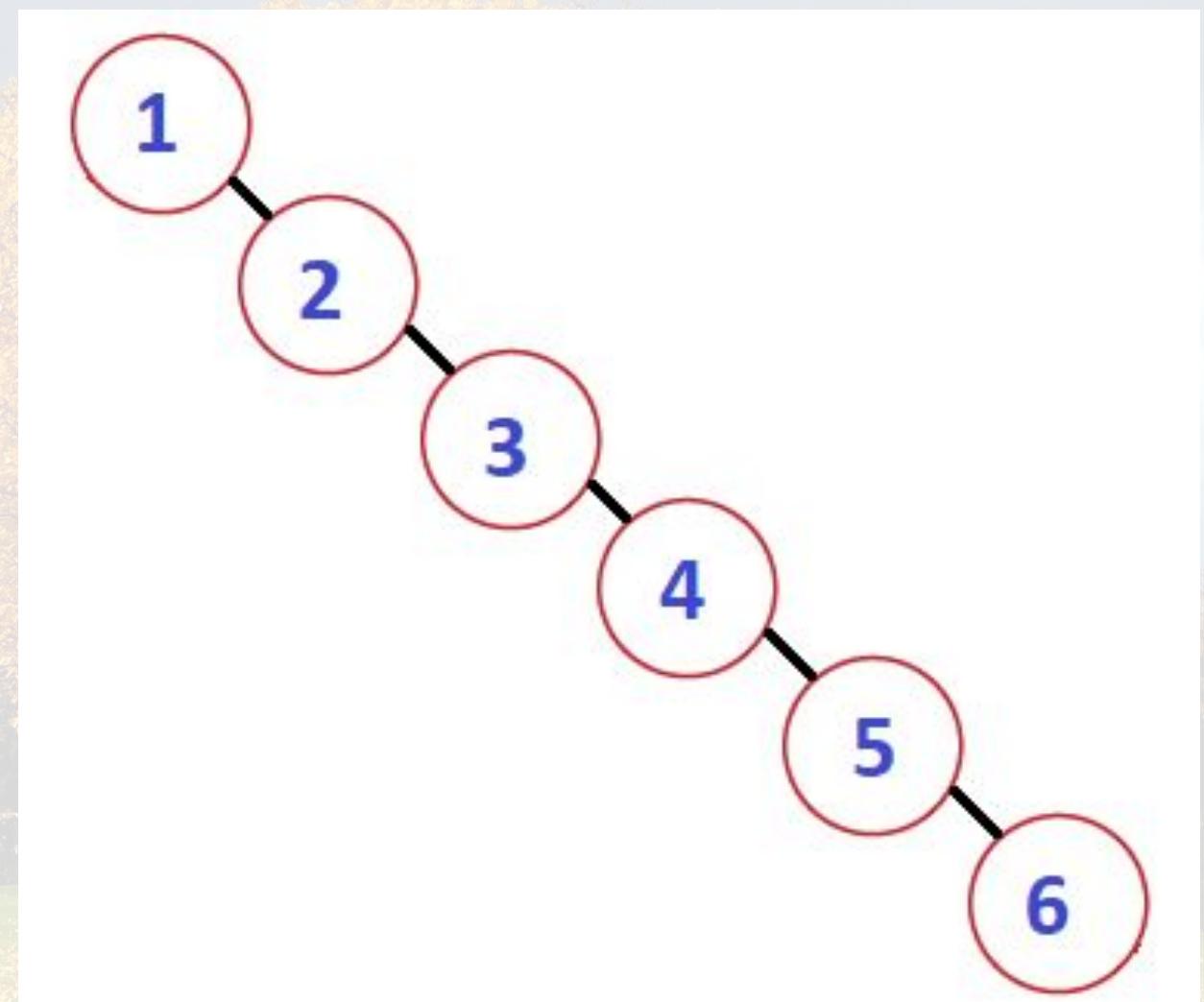


Obr. 4: Vyhľadávací strom hehe

Príklad binárneho vyhľadávacieho stromu #3

```
int cisla[] = {1, 2, 3, 4, 5, 6};
```

Obr. 5: Pole s rovnakými prvками

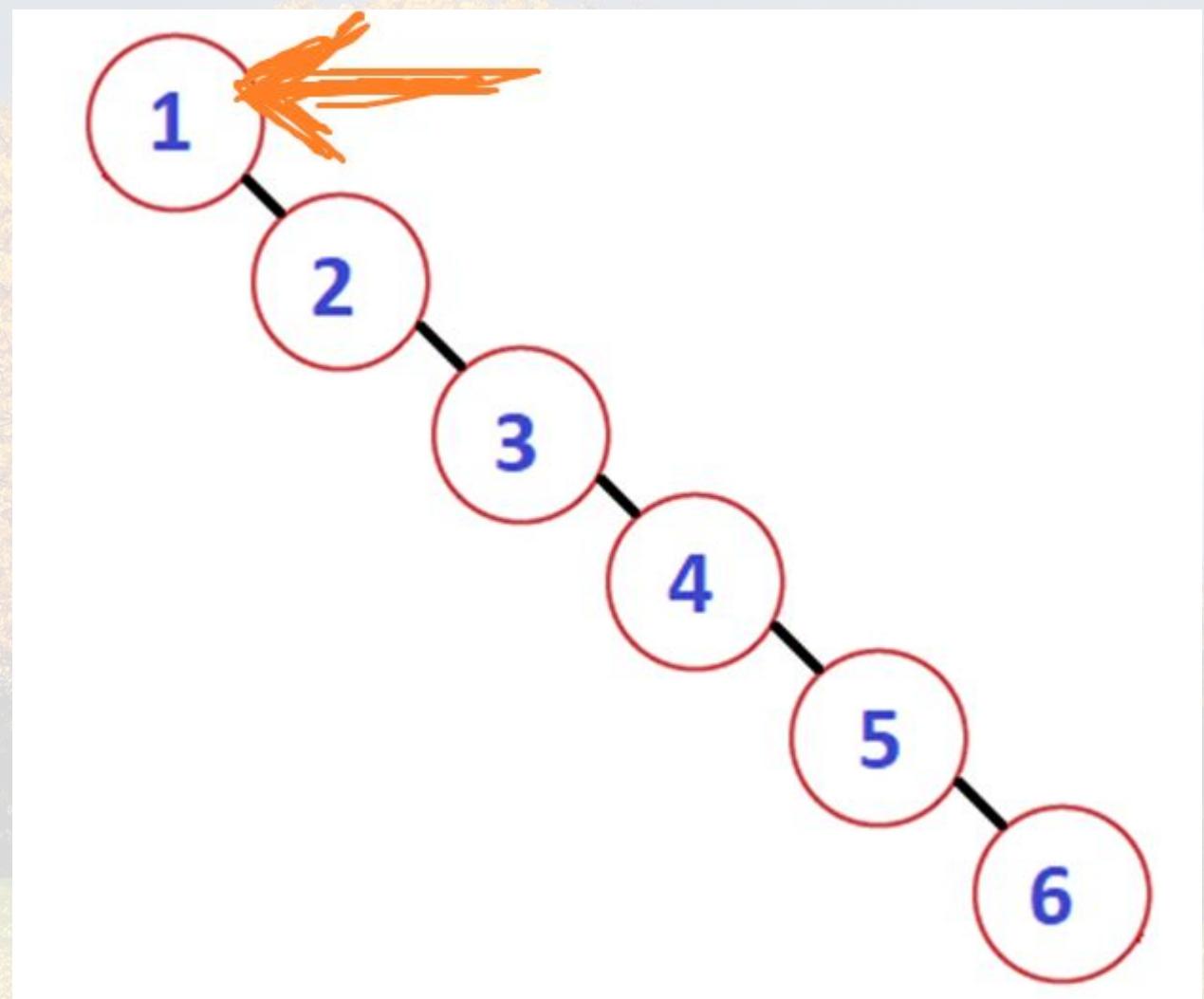


Obr. 6: Binárny bambus

Príklad - Nájdi najväčšie číslo

```
int cisla[] = {1, 2, 3, 4, 5, 6};
```

Obr. 5: Pole s rovnakými prvками

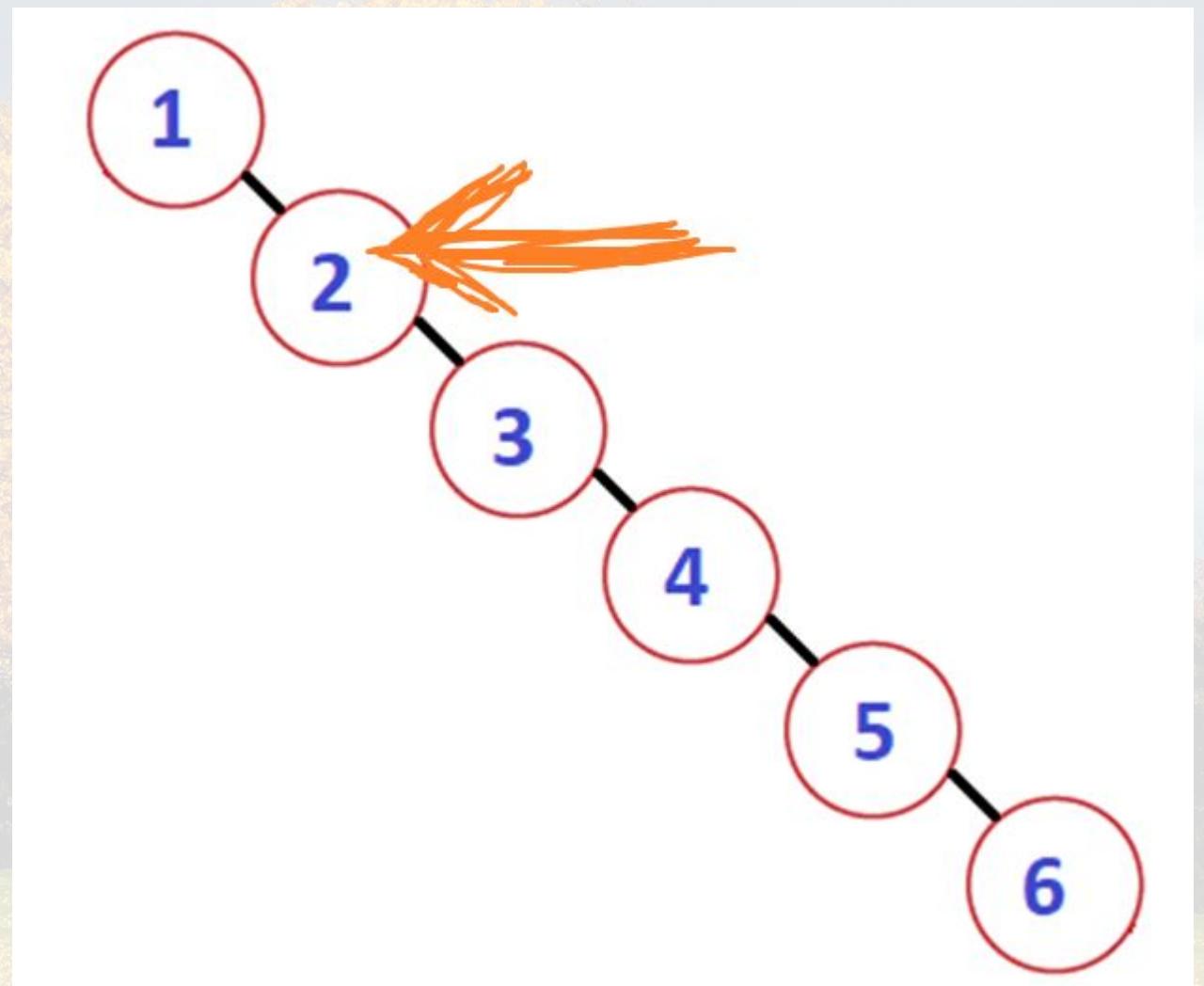


Obr. 6: Binárny bambus

Príklad - Nájdi najväčšie číslo

```
int cisla[] = {1, 2, 3, 4, 5, 6};
```

Obr. 5: Pole s rovnakými prvками

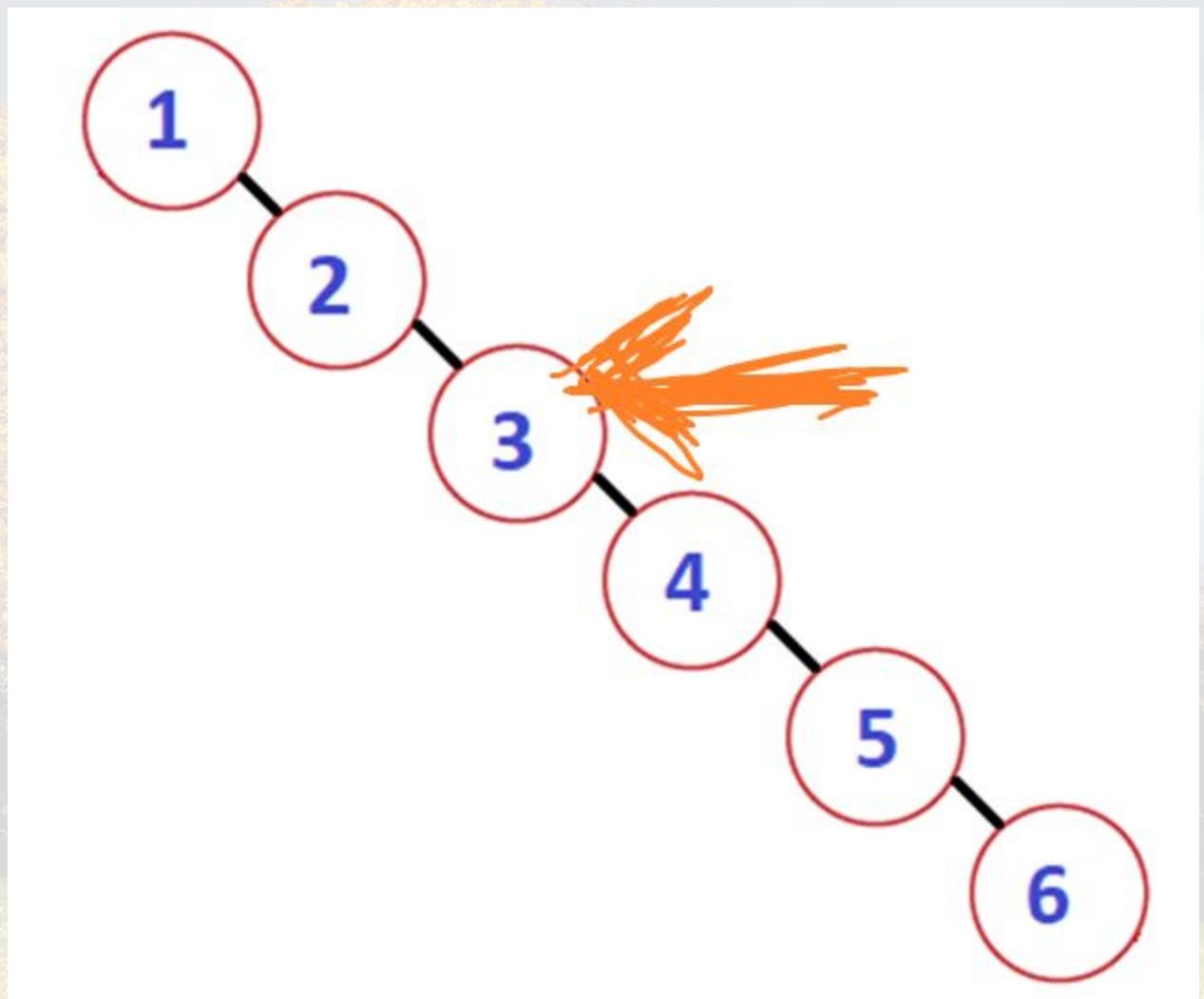


Obr. 6: Binárny bambus

Príklad - Nájdi najväčšie číslo

```
int cisla[] = {1, 2, 3, 4, 5, 6};
```

Obr. 5: Pole s rovnakými prvками

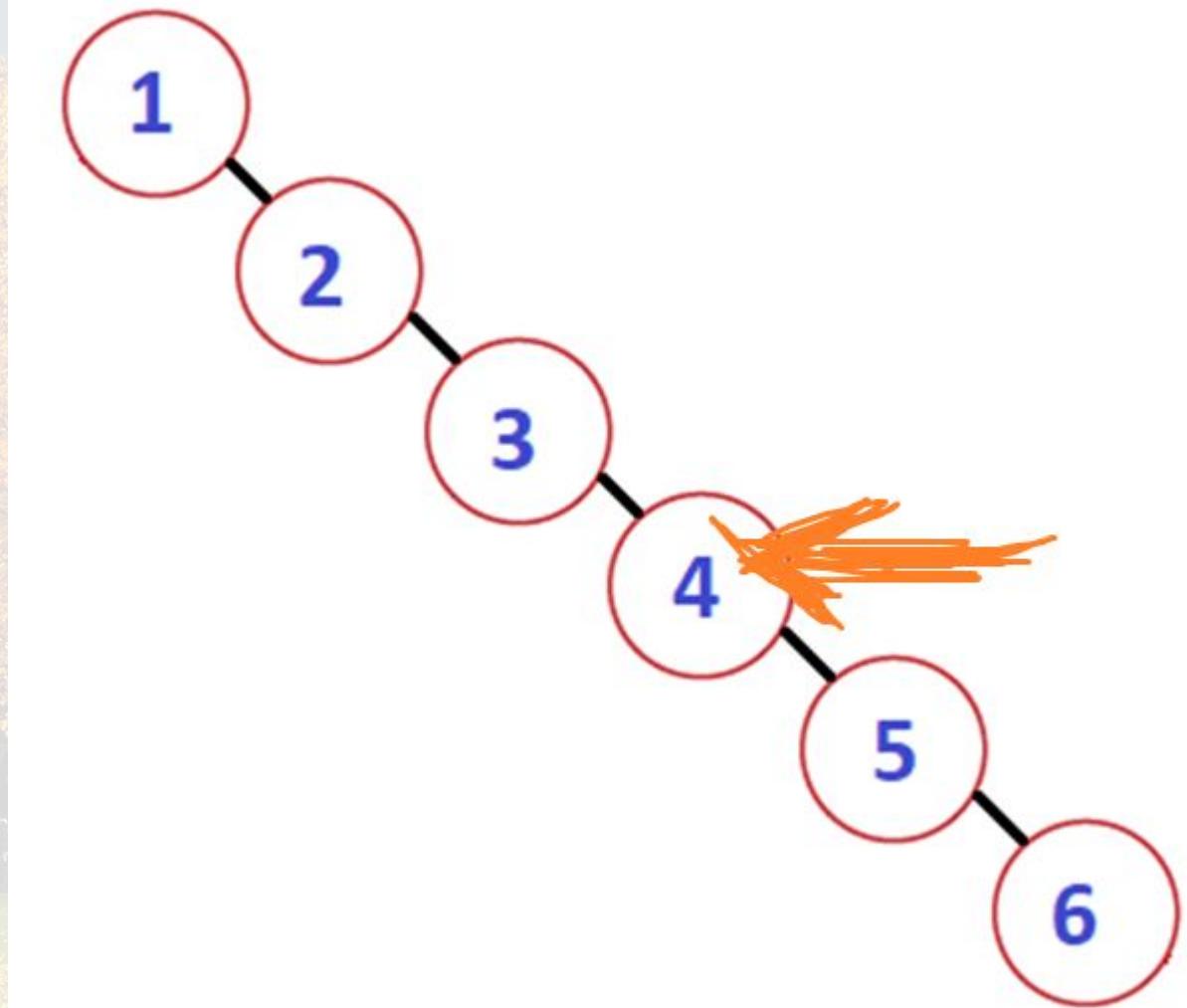


Obr. 6: Binárny bambus

Príklad - Nájdi najväčšie číslo

```
int cisla[] = {1, 2, 3, 4, 5, 6};
```

Obr. 5: Pole s rovnakými prvками

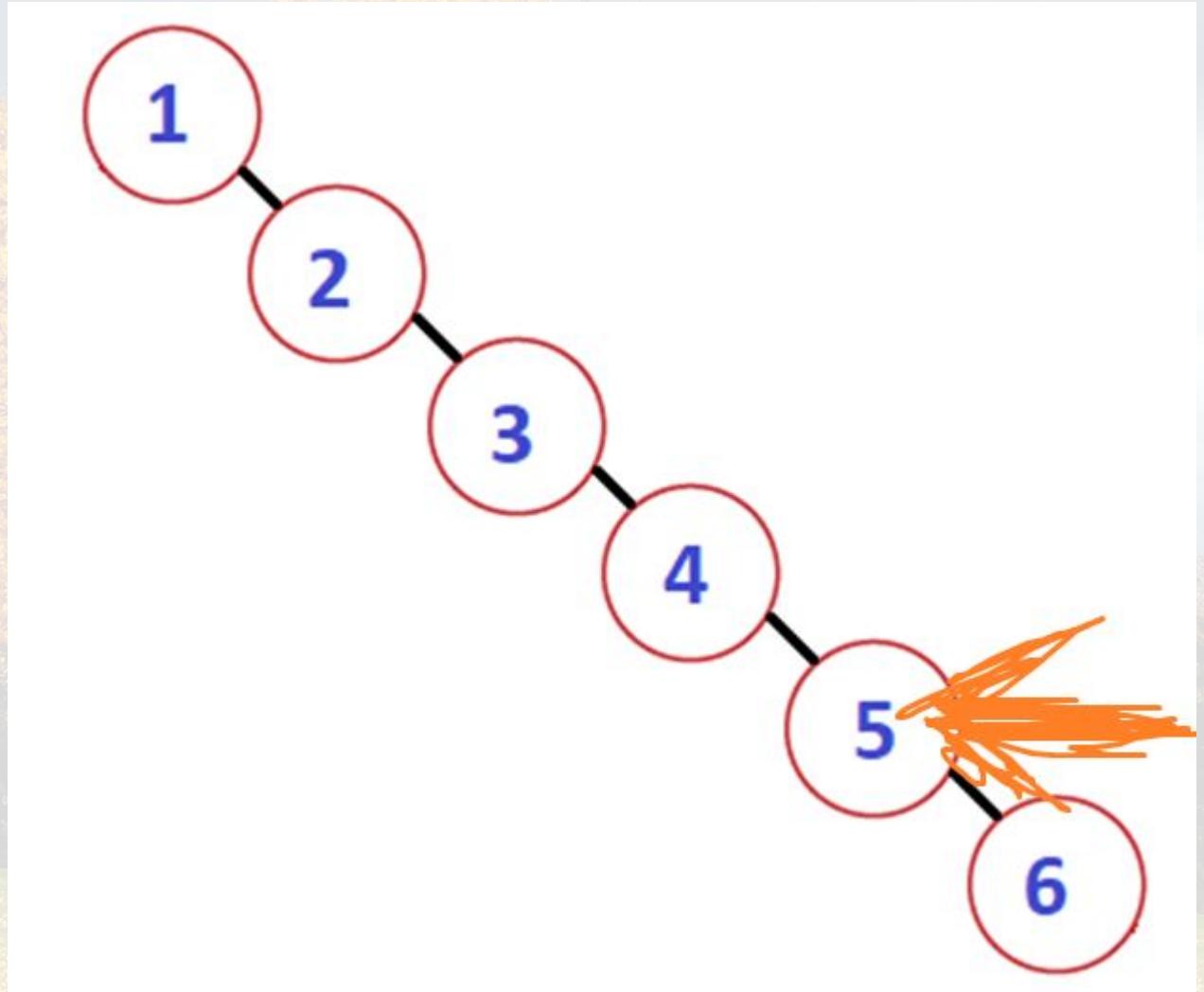


Obr. 6: Binárny bambus

Príklad - Nájdi najväčšie číslo

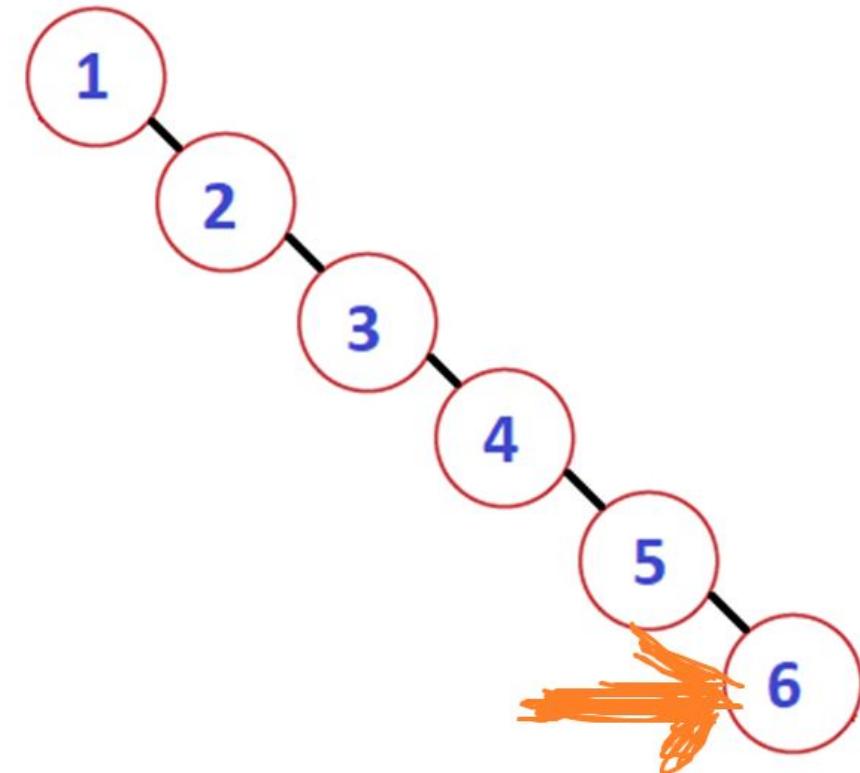
```
int cisla[] = {1, 2, 3, 4, 5, 6};
```

Obr. 5: Pole s rovnakými prvками



Obr. 6: Binárny bambus

Príklad - Nájdi najväčšie číslo



Obr. 6: Binárny bambus

```
int cisla[] = {1, 2, 3, 4, 5, 6};
```

Obr. 5: Pole s rovnakými prvkami



Obr. 7: Citoslovce radosti z dôvodu nájdenia najv. č.

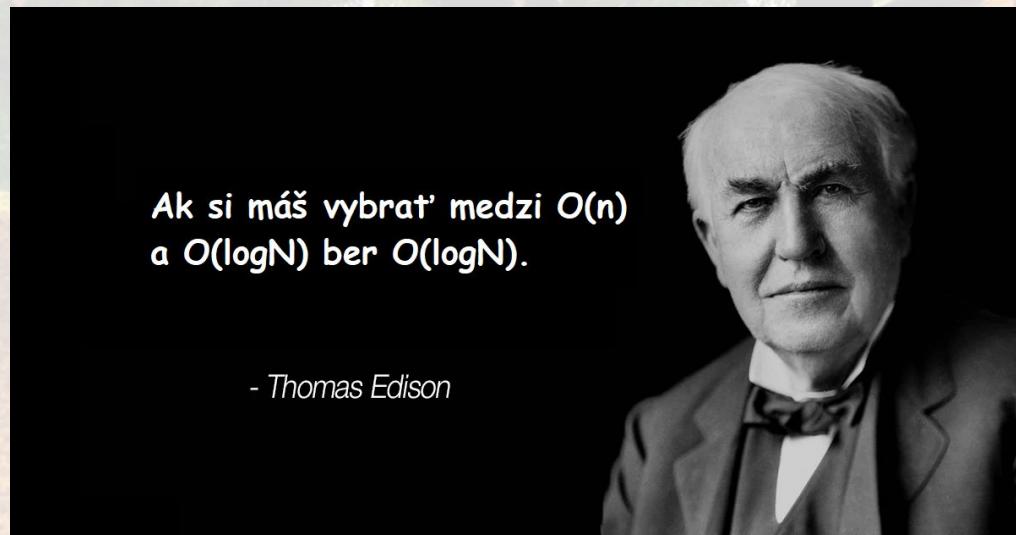
Kde je problém s predchádzajúcim príkladom?

 $O(n)$

Obr. 8b: Lineárna časová zložitosť

Kde je problém s predchádzajúcim príkladom?

- Ide to spraviť aj efektívnejšie
- Počet krokov na vykonanie bol lineárne závislý od počtu prvkov
- V spomenutom krajinom prípade mal binárny vyhľadávací strom tiež zložitosť $O(n)$



Obr. 9: Citátik

N	$\log N$
10	4
1000	10
1 000 000	20
2 000 000 000	32

Obr. 10: Porovnanie $O(n)$ a $O(\log N)$ z prednášky

Riešenie – Práca s samo-vyvažovacími

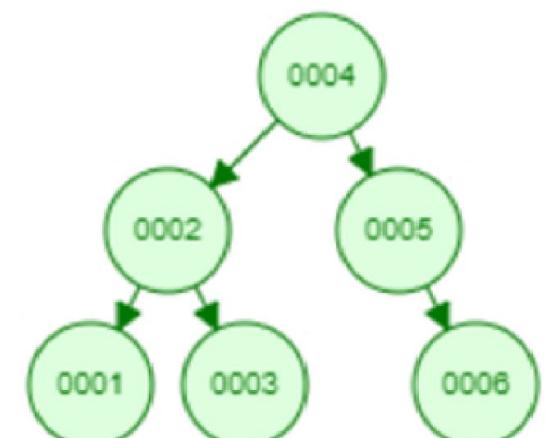
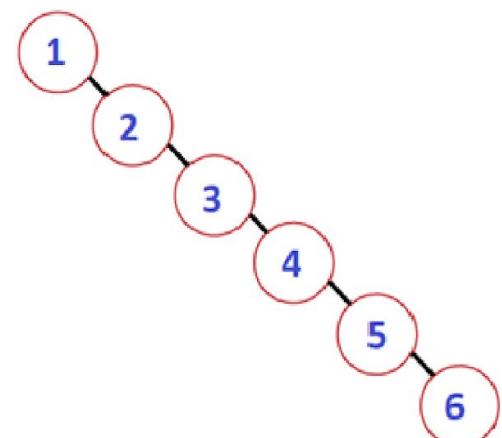
- Existuje ich za kýbel, napr:
 - 2–3 tree
 - 2-3-4 tree
 - AA tree
 - AVL tree
 - B-tree
 - Red–black tree
 - Scapegoat tree
 - Splay tree
 - Tango tree
 - Treap
 - Weight-balanced tree



Obr. 11: Zdôraznenie existencie AVL stromov

AVL strom

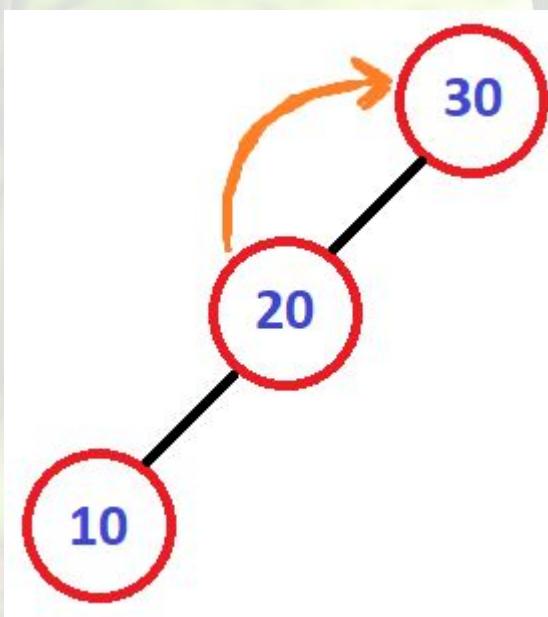
- Samovyvažovací strom (cool stuff)
- Rozdiel medzi hĺbkami ľavého a pravého podstromu nemôže byť väčší ako 1
 - V prípade, že sa po pridaní/vymazaní stane nevyváženým, vyváži sa pomocou rotácií
 - Nemôže teda nastať bambus situation



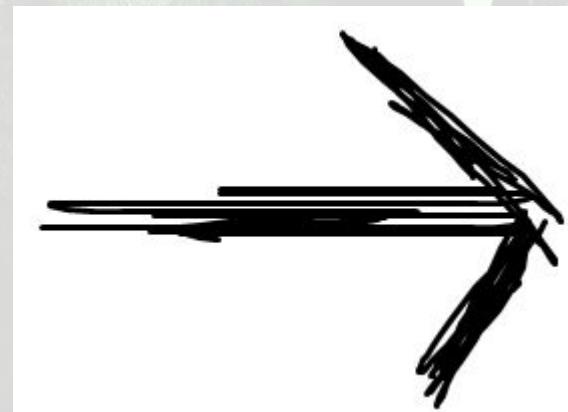
Obr. 12: Drake approves vyvážený strom

Príklad samovyvažovania #1

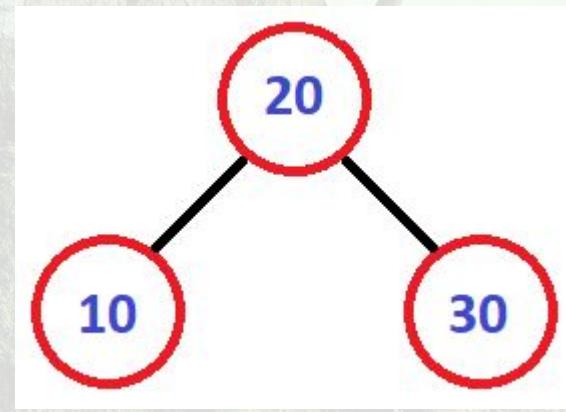
- Insert: 30, 20, 10



Obr. 13a: Nevyvážený “strom”



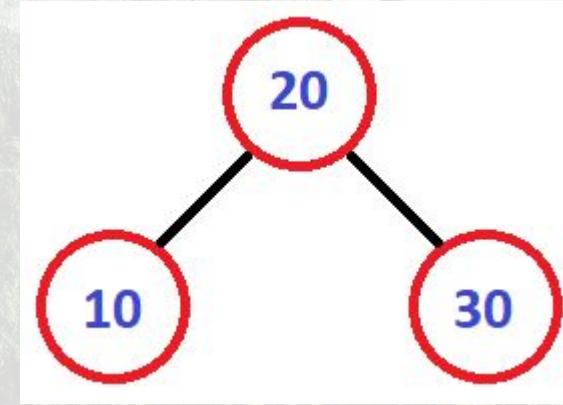
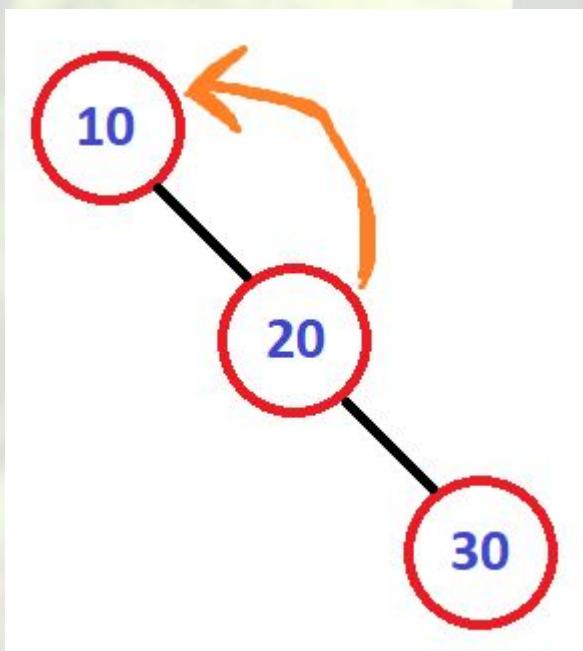
Obr. 13b: Čierna šípka otočená doprava



Obr. 13c: Vyvážený “strom”

Príklad samovyvažovania #2

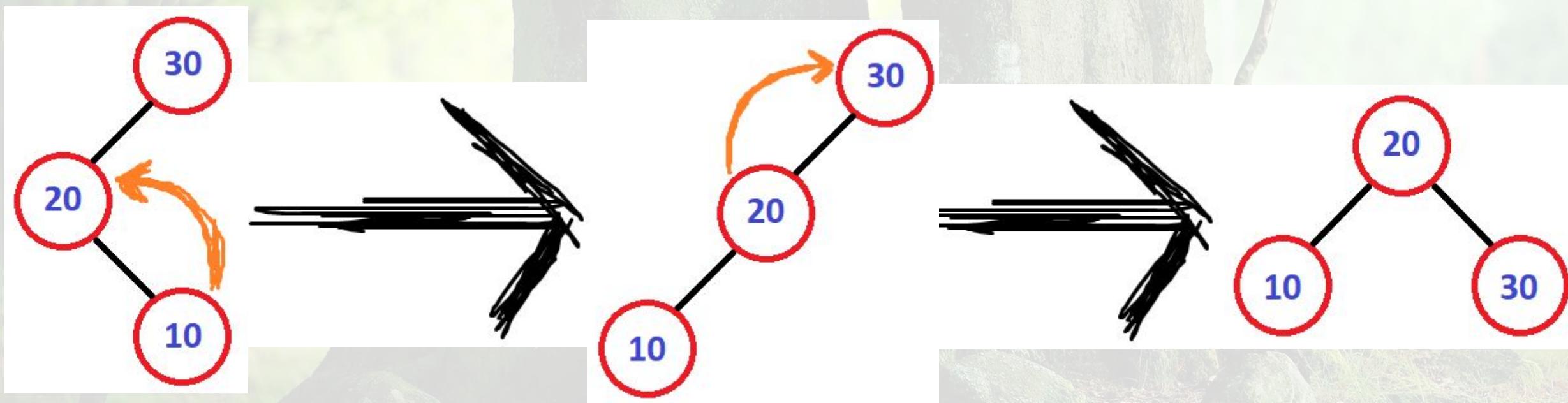
- Insert: 10, 20, 30



Obr. 14: Ukážka #2

Príklad samovyvažovania #3

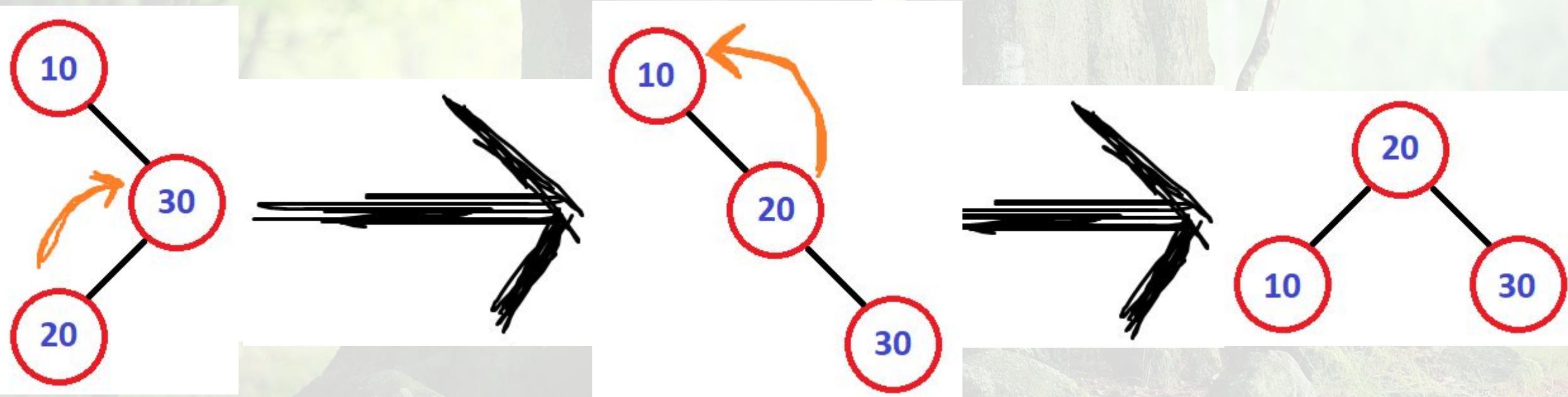
- Insert: 30, 10, 20



Obr. 15: Ukážka #3

Príklad samovyvažovania #4

- Insert: 10, 30, 20



Obr. 16: Ukážka #4

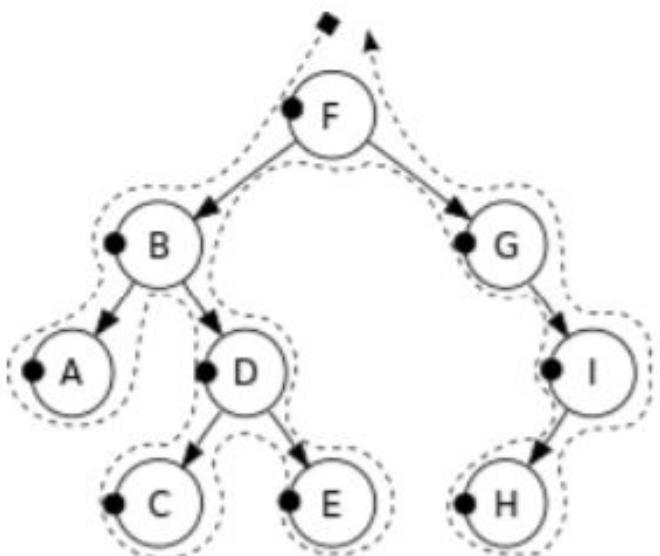
Prechádzanie stromov (Tree traversals)

- 3 spôsoby:
 - PRE-ORDER
 - Inorder
 - Postorder

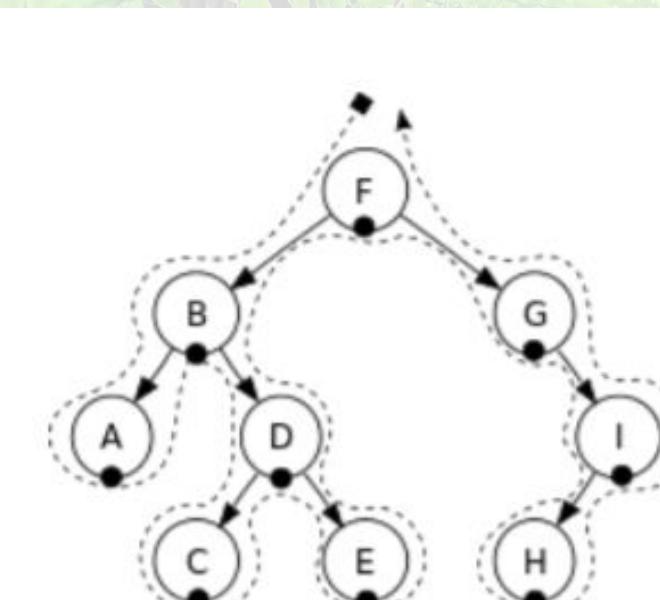


Obr. 17: Zdôraznenie existencie 3 spôsobov prechádzania

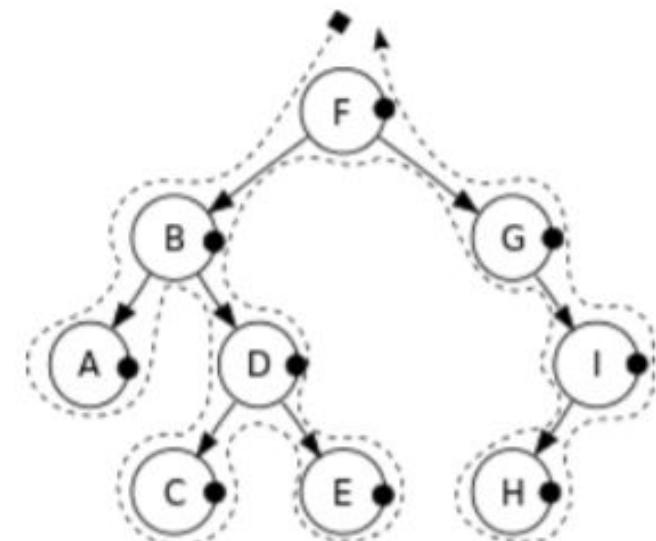
Ukážka prechádzania stromov



Obr. 18: Preorder



Obr. 19: Inorder

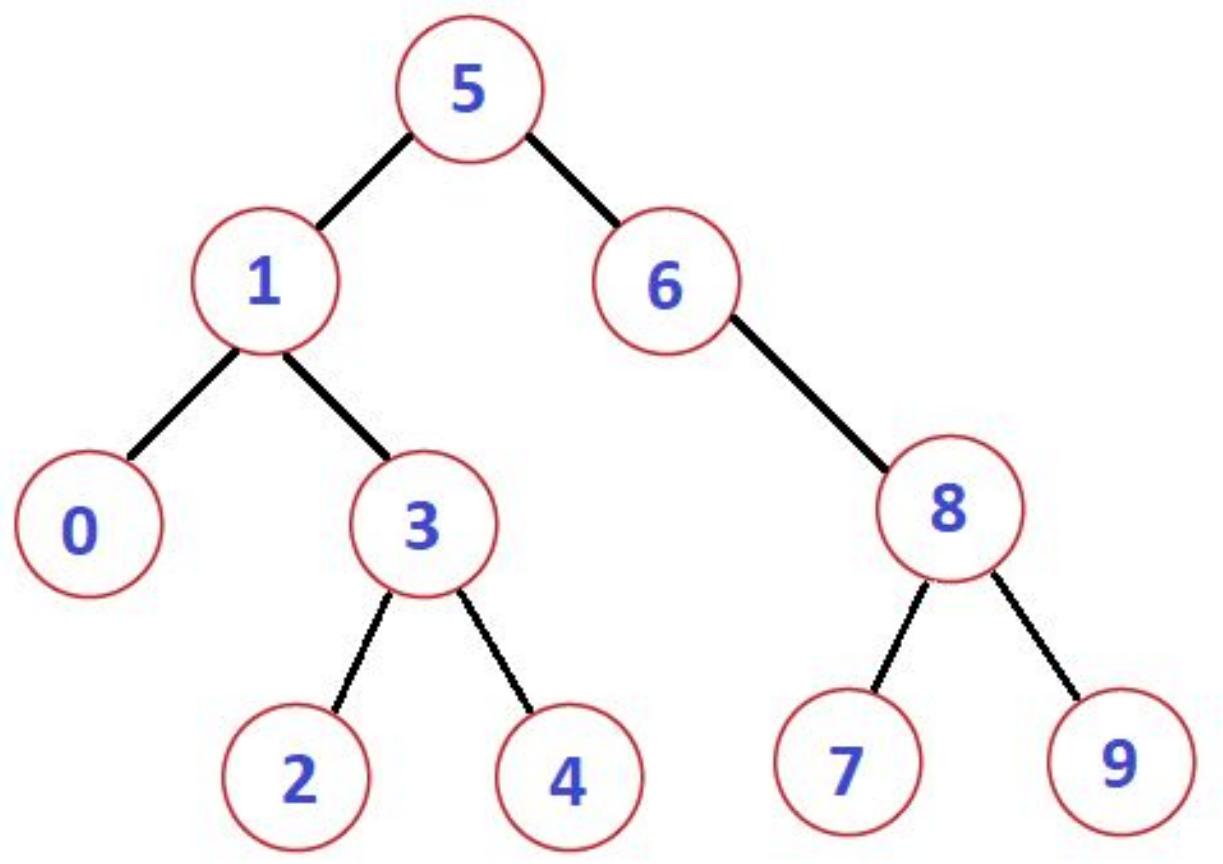


Obr. 20: Postorder

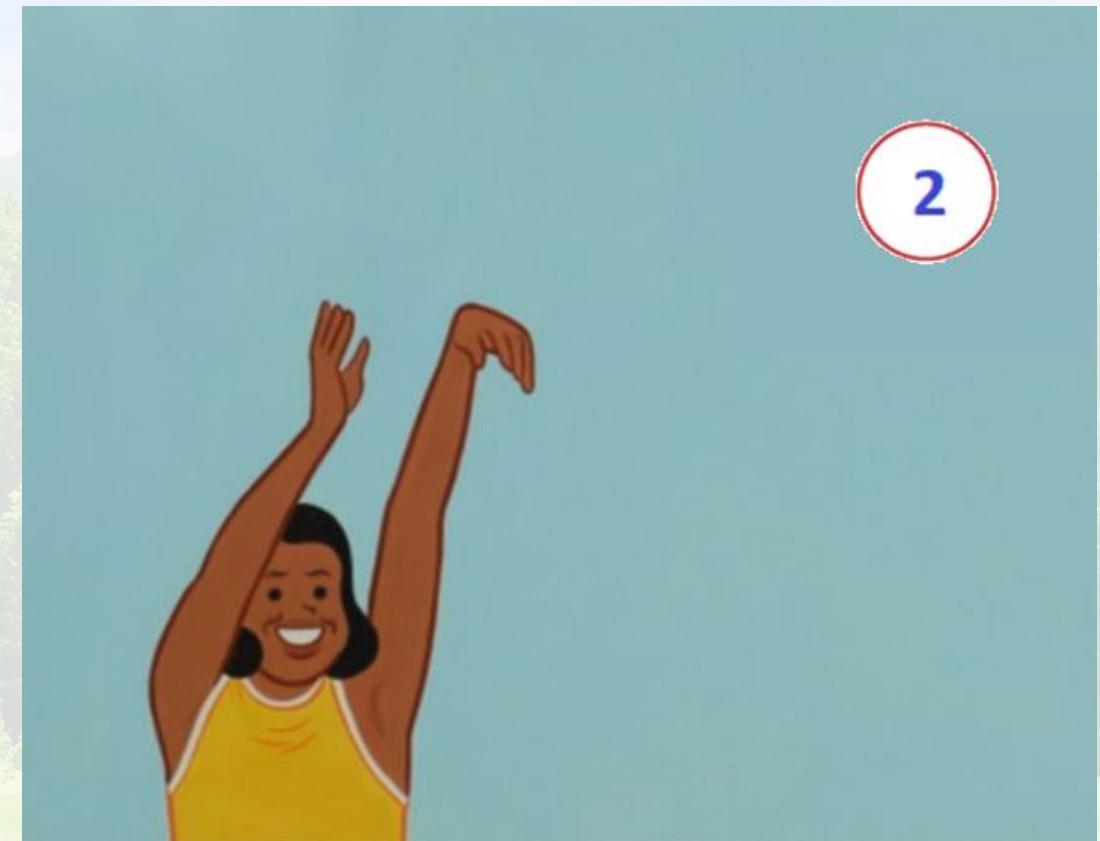
DELETE v binárnom vyhľadávacom strome

- Ak je vymazávaný uzol „list“
 - Just yeet it out
- Ak má vymazávaný uzol jedného potomka
 - Prekopírovať potomka na miesto vymazávaného prvku
 - Vymazat' prekopírovaného potomka
- Uzol má dvoch potomkov
 - Nájst' inorder nasledovníka daného uzla
 - Prekopírovať obsah nájdeného nasledovníka na miesto vymazávaného prvku
 - Zmazať obsah prekopírovaného potomka

Príklad #1: DELETE 2

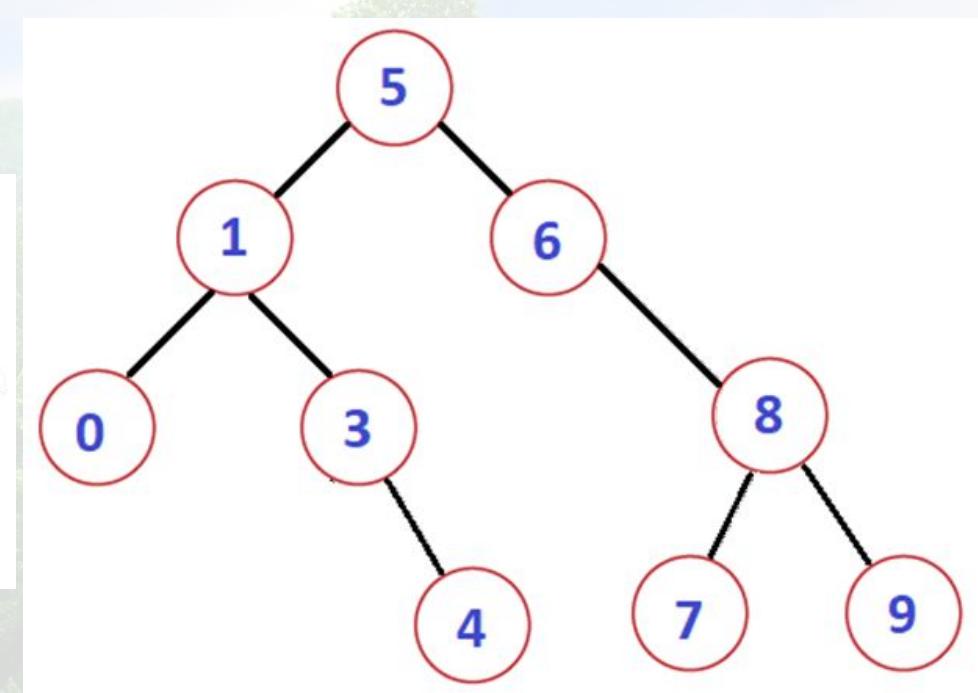
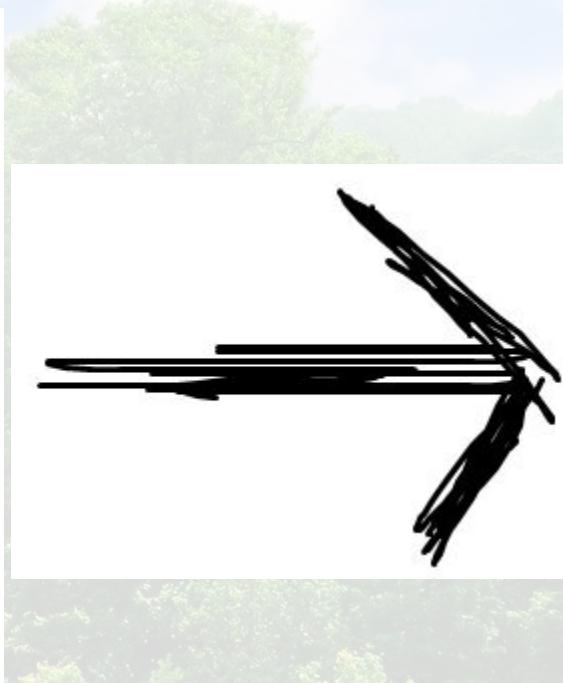
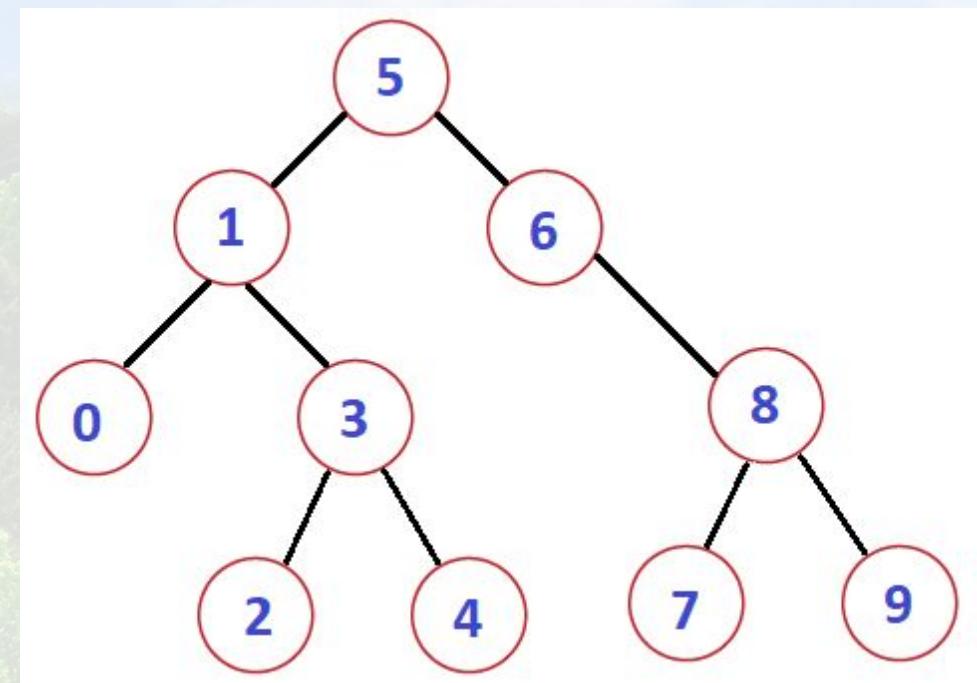


Obr. 21: Príklad binárneho vyhľadávacieho stromu



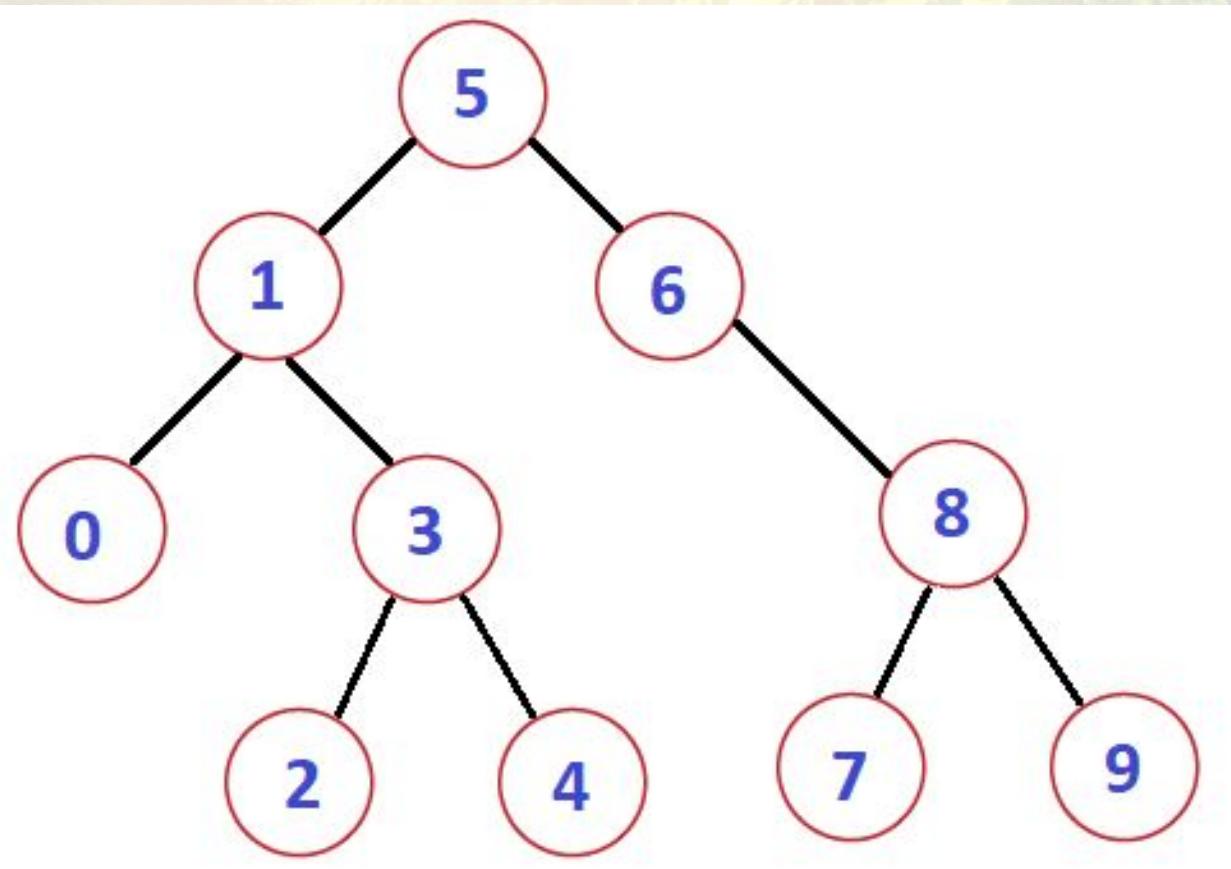
Obr. 22: Yeet node 2

Príklad #1: DELETE 2

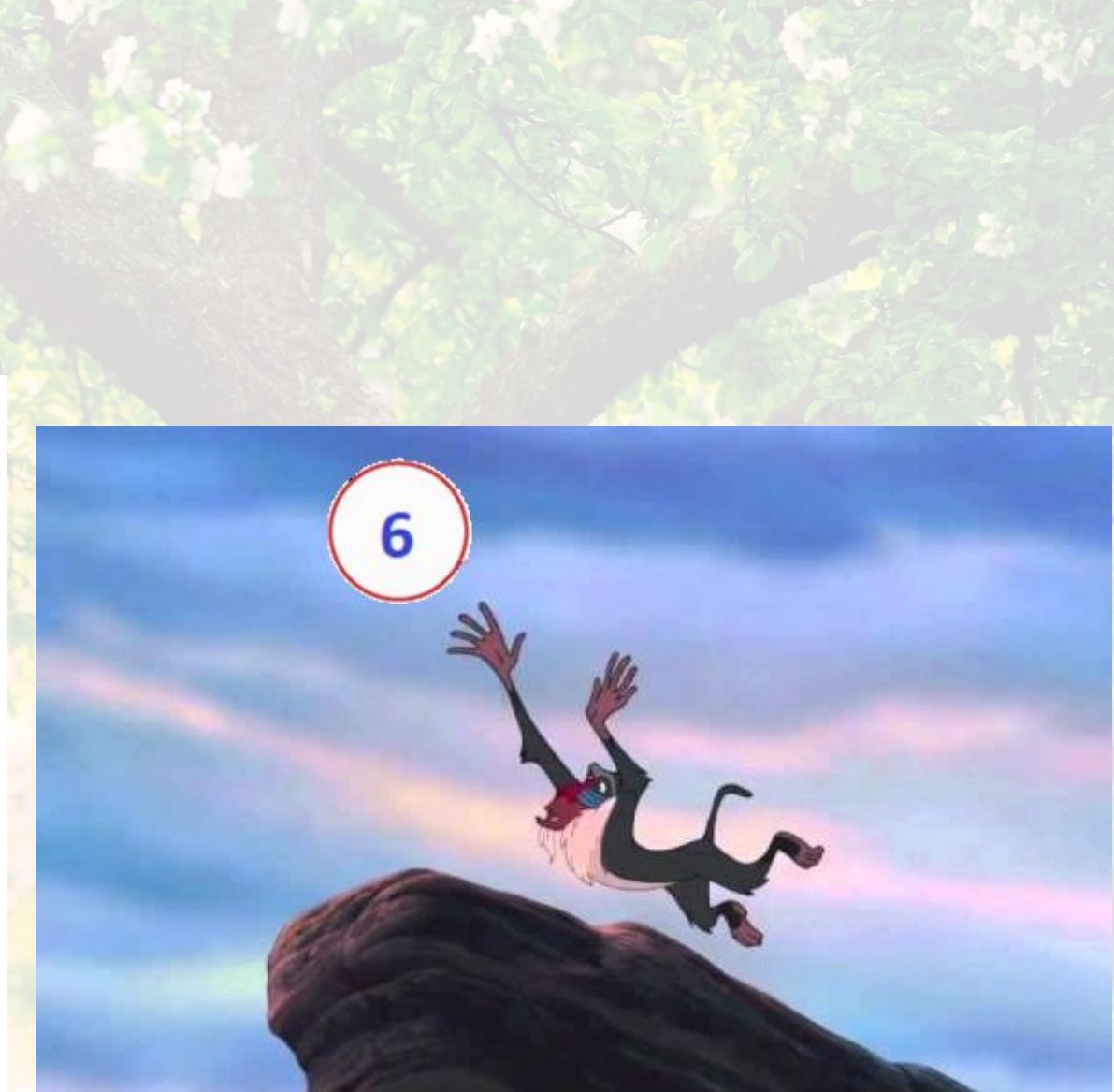


Obr. 23: Iba sme odstránili list

Príklad #2: DELETE 6

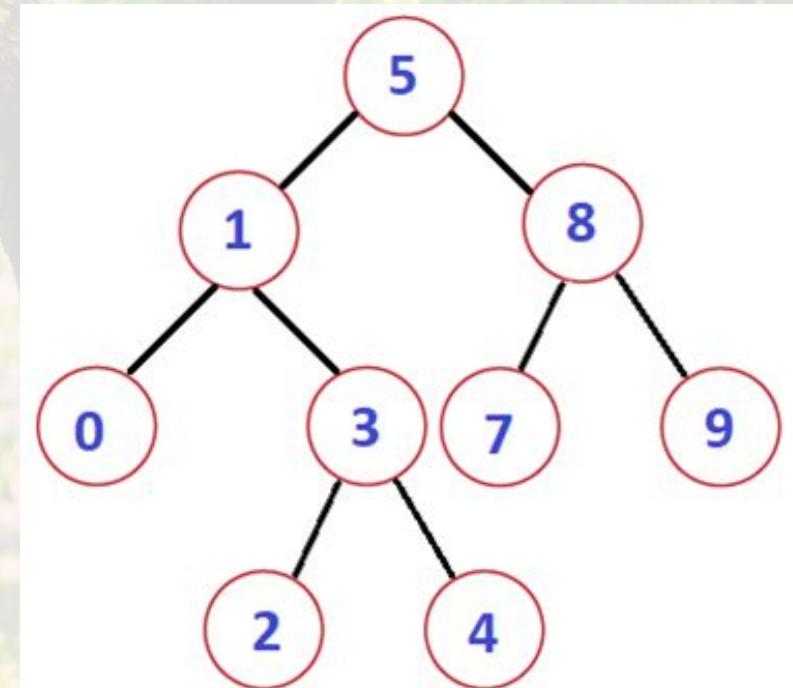
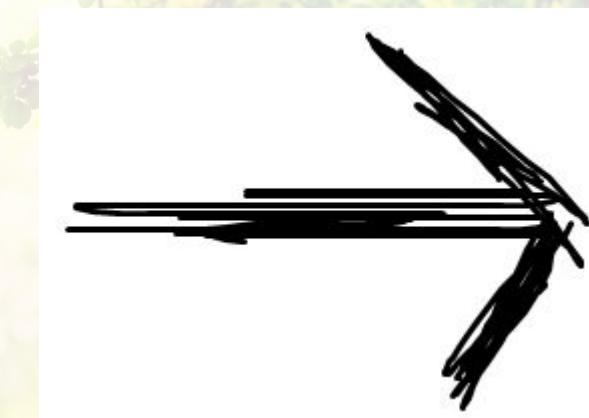
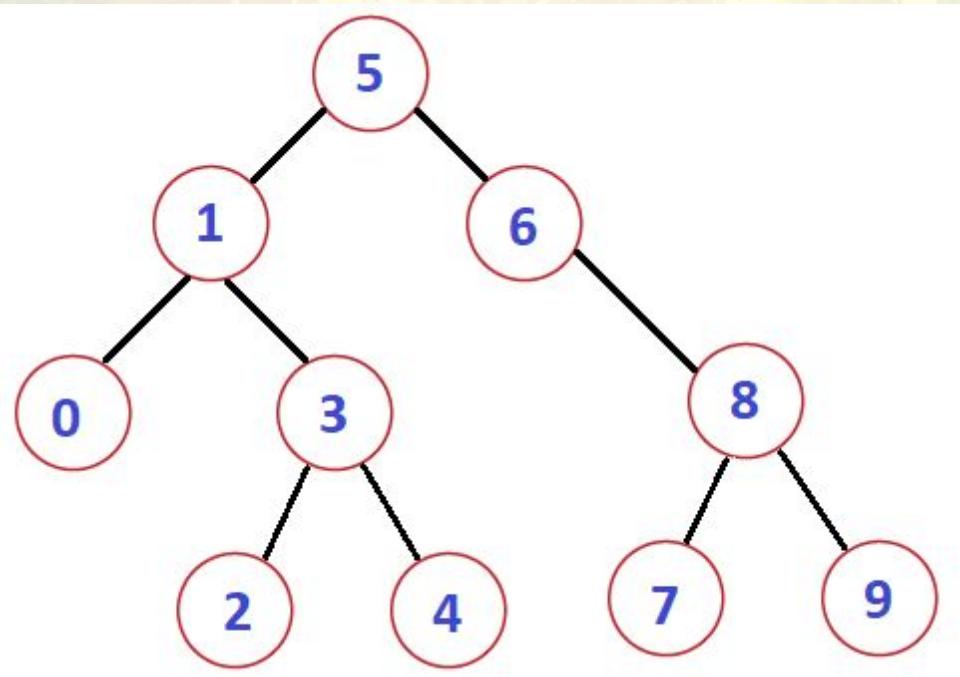


Obr. 24: Príklad binárneho vyhľadávacieho stromu 🌳



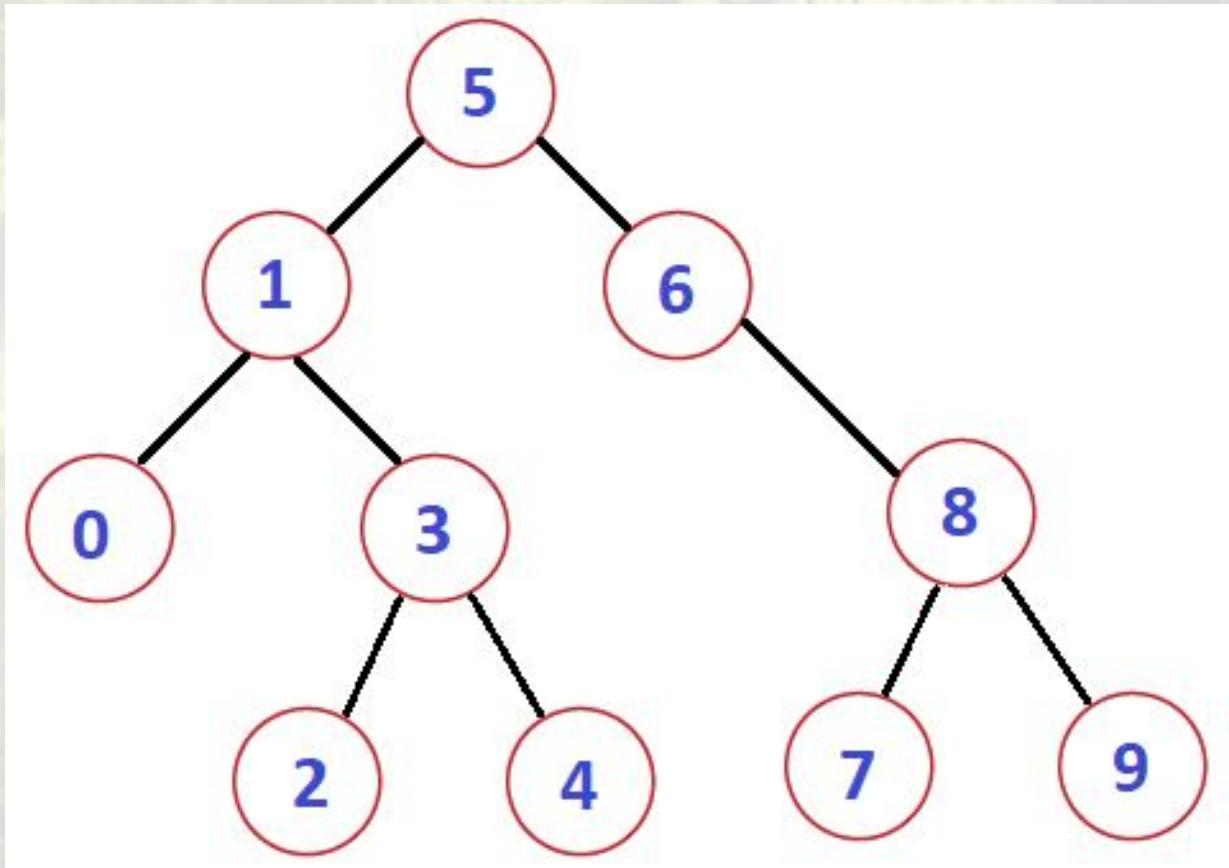
Obr. 25: Node 6 is yote out

Príklad #2: DELETE 6



Obr. 26: Prekopírovali sme obsah 8 do 6 a vymazali

Príklad #3: DELETE 1

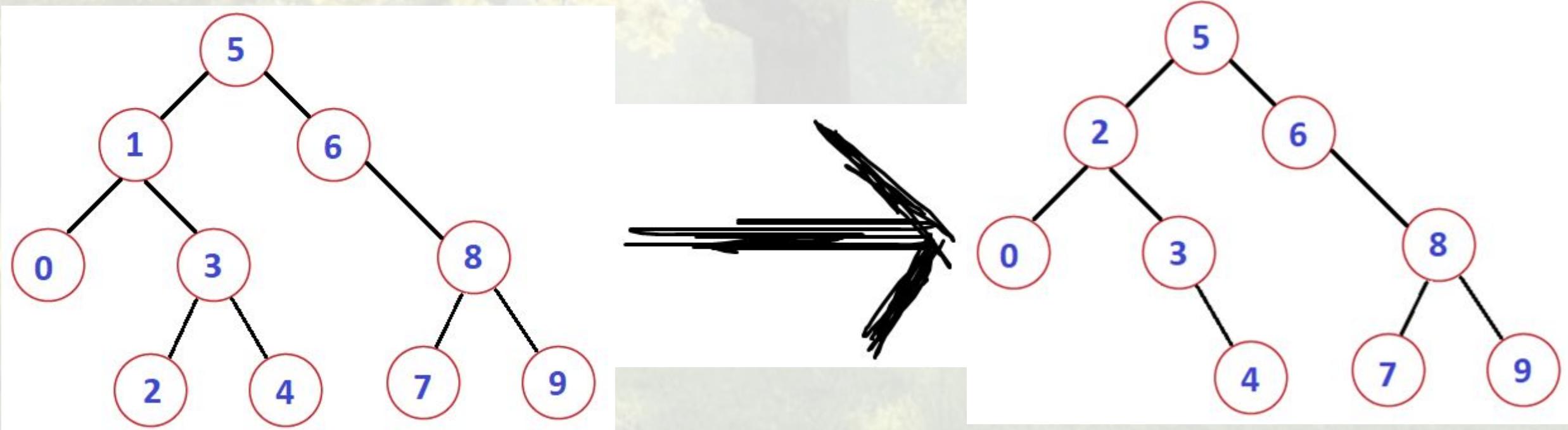


Obr. 27: Príklad binárneho vyhľadávacieho stromu 🌳



Obr. 28: 👏 node 1

Príklad #3: DELETE 1



Obr. 29: Našli sme inorder nasledovníka a nahradili ním obsah 1 a vymazali

V prípade, že po vymazaní uzla dôjde pri samovyvažovacích  k „nerovnováhe“ je potrebné ich ešte vyvážiť



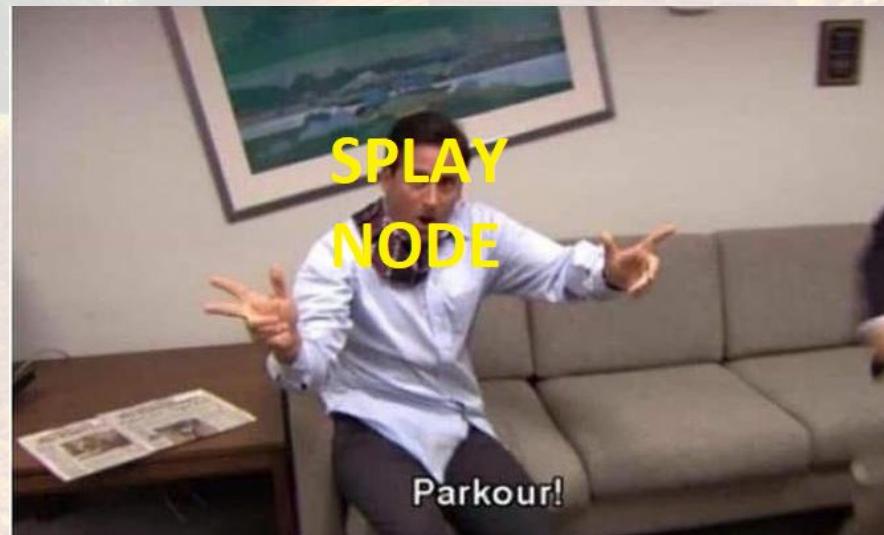
Obr. 30: Stromos



Obr. 31: Zdôraznenie existencie Splay stromov

Splay tree

- Samovyvažovací strom
 - Neprodukuje však kyslík
- Oproti klasickým operáciám ako SEARCH, INSERT a DELETE disponuje operáciou SPLAY
 - Presunie vyhľadávaný/vložený prvok do koreňa stromu za pomocí rotácií



Obr. 32: Splayed node parkour goes brrrrrr

Rozlišujeme 3 základné situácie pri vykonávaní operácie S

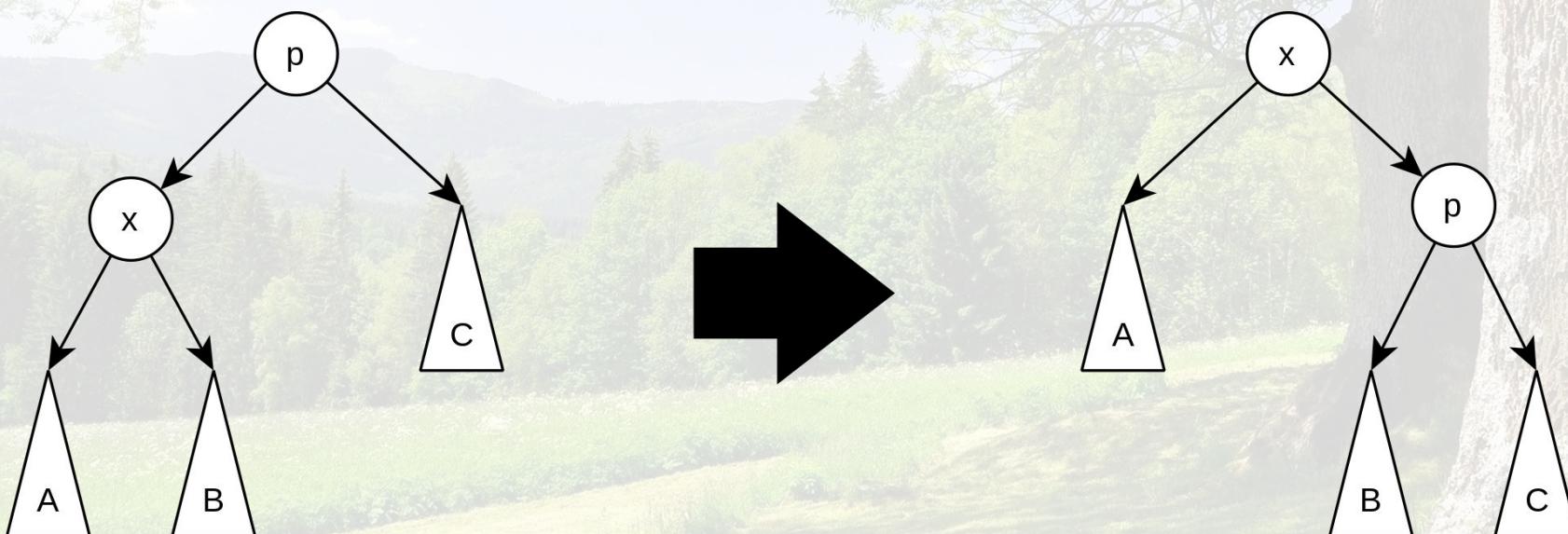
- **Rodič** uzla x je koreň a uzol x ľavý potomok
- **Rodič** uzla x nie je koreň a je **tiež** ako uzol x ľavým potomkom
- **Rodič** uzla x nie je koreň a je ľavým potomkom **na rozdiel** od uzla x , ktorý je pravým potomkom

Toto platí aj symetricky, teda odvodené situácie sú:

- **Rodič** uzla x je koreň a uzol x pravý potomok
- **Rodič** uzla x nie je koreň a je **tiež** ako uzol x pravým potomkom
- **Rodič** uzla x nie je koreň a je pravým potomkom **na rozdiel** od uzla x , ktorý je ľavým potomkom

Rodič uzla x je koreň a uzol x ľavý potomok

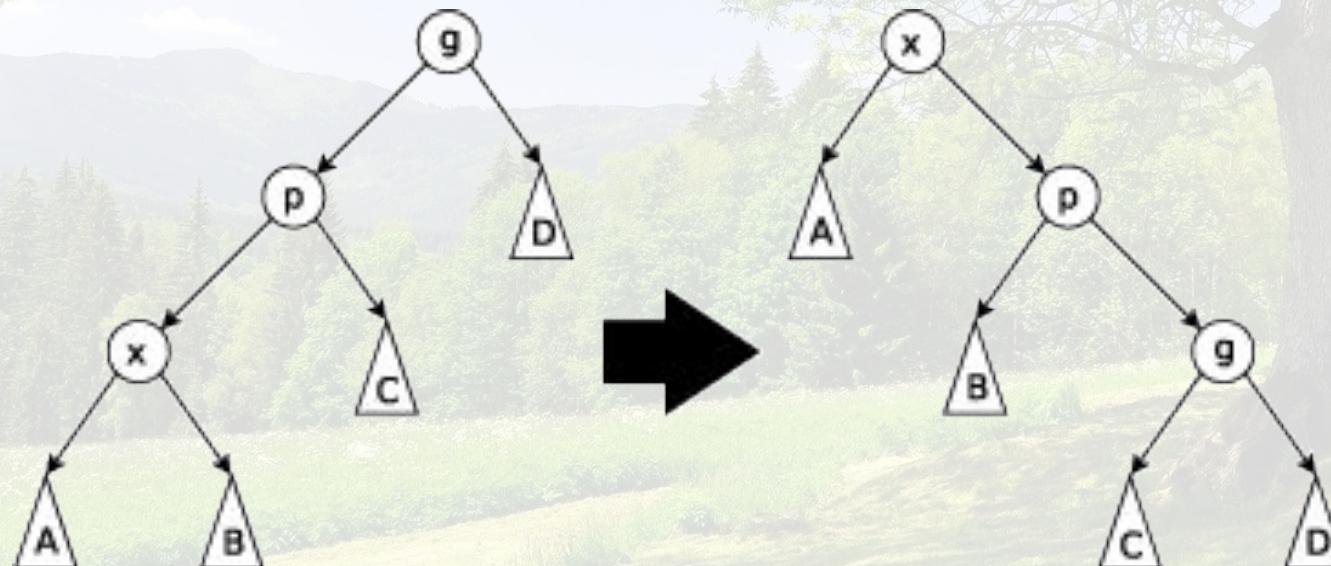
- Najjednoduchšia situácia
 - Vykoná sa iba jedna rotácia doprava („zig“ rotation)
 - V opačnom prípade (x je pravý potomok) – p doľava („zag“ rotation)



Obr. 33: Ukážka situácie #1

Rodič uzla x nie je koreň a je tiež ako uzol x ľavým potomkom

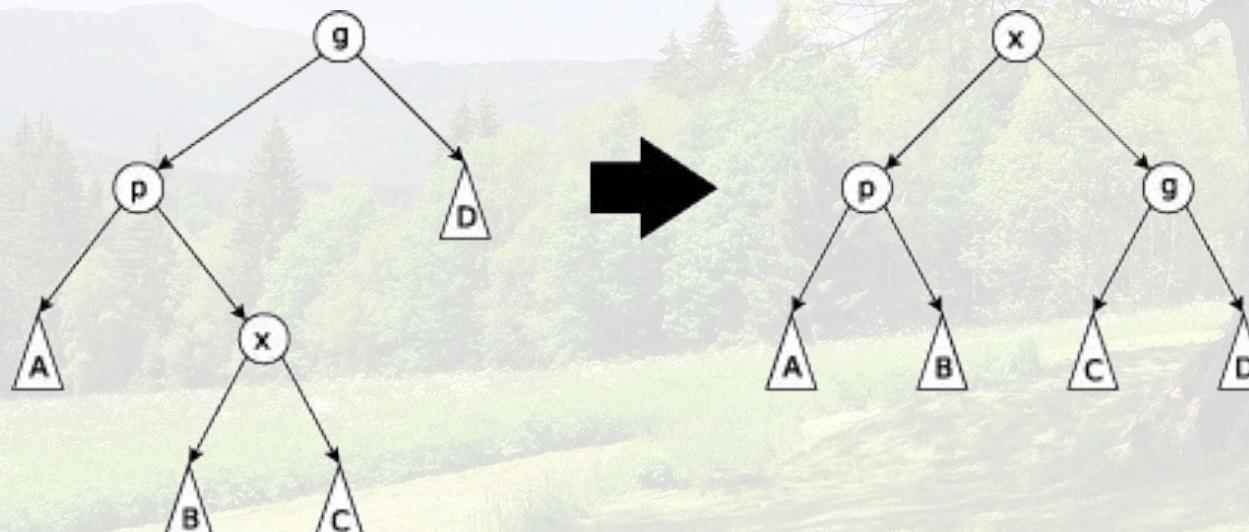
- Vykonajú sa 2 rotácie doprava („zig-zig“ rotation)
- V opačnom prípade (Rodič uzla x nie je koreň a je tiež ako uzol x **pravým** potomkom) sa vykonajú 2 rotácie doľava („zag-zag“ rotation)



Obr. 34: Ukážka situácie #2

Rodič uzla x nie je koreň a je ľavým potomkom na rozdiel od uzla x , ktorý je pravým potomkom

- Vykonajú sa 2 rotácie: doľava potom doprava („zig-zag“ rotation)
- V opačnom prípade (Rodič uzla x nie je koreň a je pravým potomkom na rozdiel od uzla x , ktorý je ľavým potomkom) sa vykoná najprv rotácia doprava a potom doľava („zag-zig“ rotation))

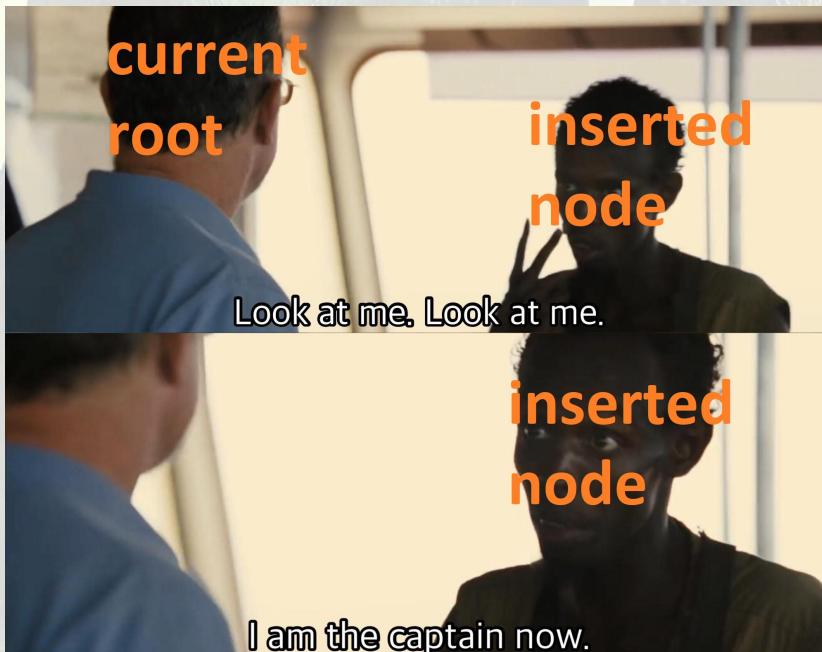


Obr. 35: Ukážka situácie #3

INSERT do Splay



- 2 kroky:
 - Vložíme prvok rovnako ako pri klasickom BVS
 - Vykonáme operáciu Splay(pridaný prvok)
 - Teda sa za pomoci rotácií presunie do koreňa stromu



Obr. 36: Vkladaný uzol je teraz koreň

DELETE zo Splay



- 2 kroky:
 - Vymažeme prvok rovnako ako pri klasickom BVS
 - Vykonáme operáciu Splay(rodič zmazaného prvku)
 - Rodič zmazaného prvku sa presunie do koreňa stromu



Obr. 37: Parent uzol je teraz koreň