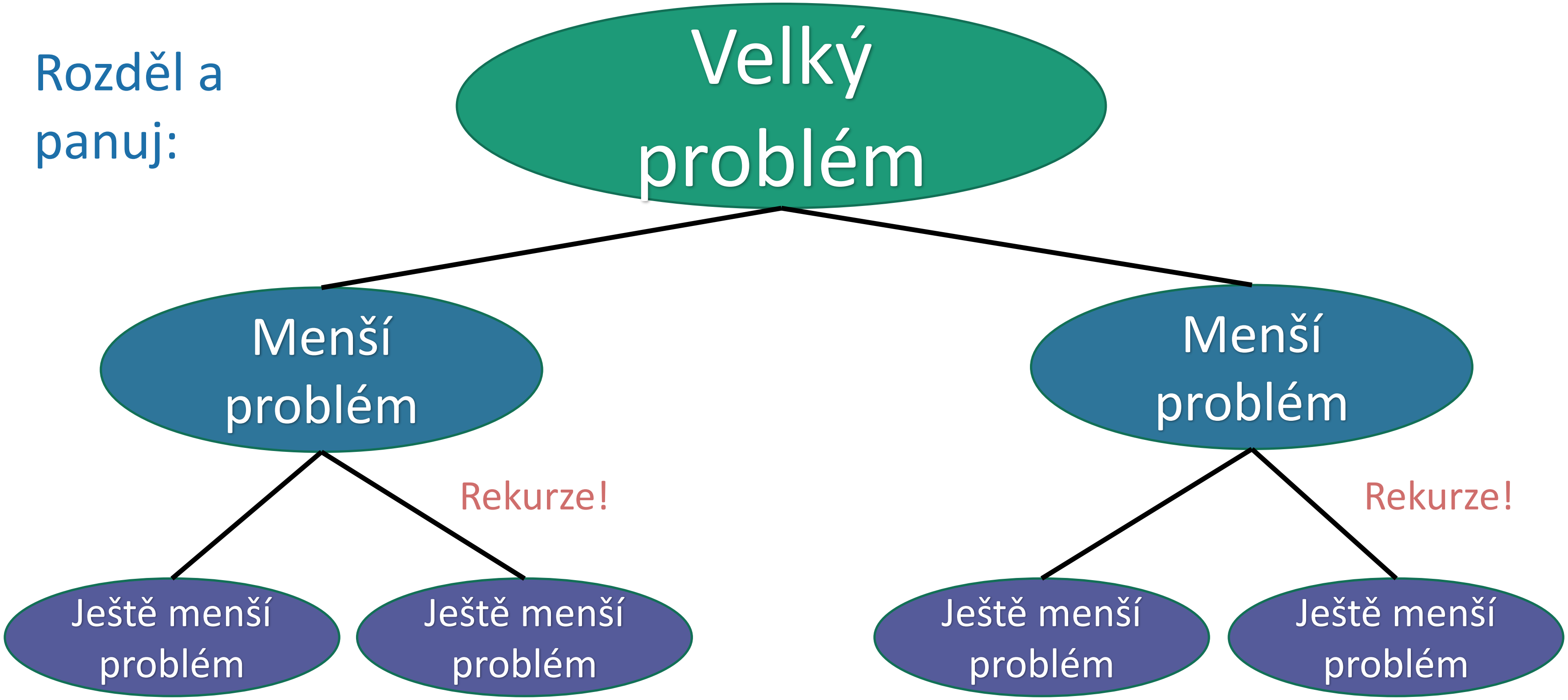
Přednáška

Binární vyhledávací stromy

Připomeňme si:

- Přístup rozděl a panuj.

Rozděl a panuj:



Jak navrhujeme algoritmy rozděl a panuj?

- Dosud jsme probrali některé příklady.
  - Karatsubovo násobění
  - MergeSort
  - QuickSort
- Podívejme se na některé obecné strategie.

Jedna strategie

1. Identifikujeme přirozené dílčí problémy
   - Pole rozdělíme na půl
   - Prvky menší/větší než pivot
2. Představte si, že máte magickou schopnost vyřešit ty přirozené dílčí problémy ... co byste dělali?
   - Vyzkoušíme to se všemi přírozenými dílčími problémy, se kterými můžeme přijít! Cokoli vypadá užitečně.
3. Vypracujeme podrobnosti
   - Zapíšeme pseudokód, atd.

Porovnejte tuto strategie s MergeSort nebo QuickSort!

Jiné postupy

- Malé příklady.
  - Pokud máte nějaký nápad, ale nedokážete zjistit podrobnosti, zkuste to na malém příkladu ručně.
- Podobnost...
  - Čím více algoritmů uvidíte, tím snazší bude vymýšlet nové algoritmy!
- Využívejte své analytické nástroje.
  - Např. pokud dělám divide-and-conquer se dvěmi dílčími problémy velikosti n/2 a chci algoritmus o času O (n logn), vím, že si můžu dovolit dílčí řešení O (n) kombinující mé dílčí problémy.
- Iterace.
  - Aha, tenhle přístup nefunguje! Ale pokud by tento drobný aspekt vylepšil, možná to bude fungovat lépe?
- Každý přistupuje k řešení problémů jinak ... najděte způsob, který vám nejlépe vyhovuje.

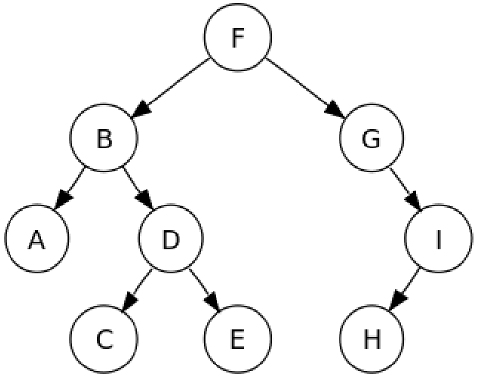
Na návrh algoritmů neexistuje univerzální recept.

- To může být frustrující ….
  - P vs NP: mnohem snazší porozumět důkazu, než s ním přijít!

- Praxe pomáhá!
  - Příklady, které vidíme na přednášce a máte na cvičení, vám mají pomoci procvičit si tuto dovednost.
- Pro zájemce v literatuře je popsáno daleko více algoritmů!
  - Podívejte se do seznamu literatury. Popisy algoritmů jsou dostupné i na internetu.

Dnešní téma

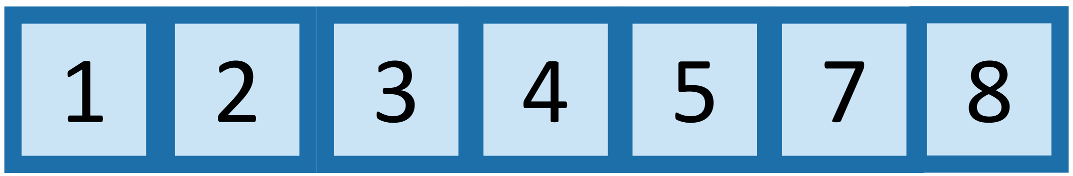
- Binární stromy



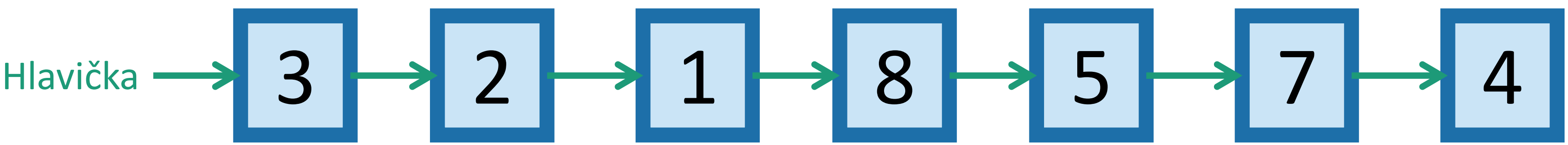
Některé datové struktury

pro uložení objektu jako je $\boxed{5}$ (tedy element s hodnotou)

- (Uspořádané) pole:



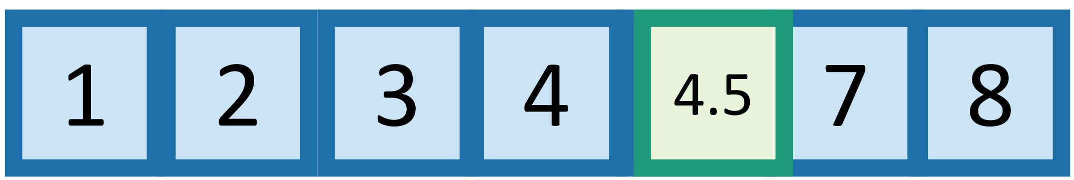
- Spojový seznam:



- Základní operace:
  - VLOŽENÍ, MAZÁNÍ, HLEDÁNÍ

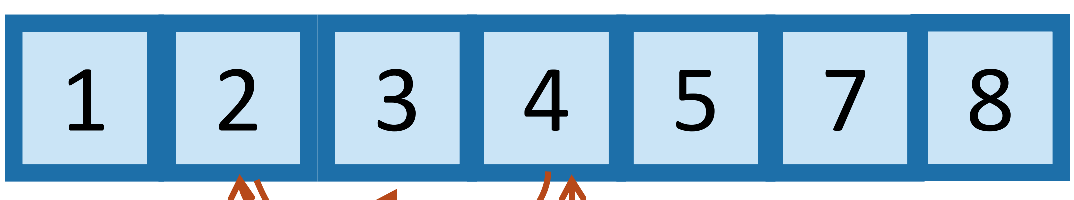
Seřazené pole

- O(n) Vložení/Mazání:
  - Nejprve najdeme příslušný prvek a vložíme další prvky do pole :



Př. vlož 4.5

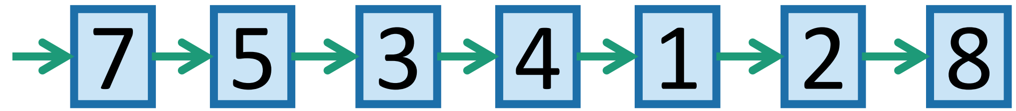
- O(log(n)) Hledání:



Př., binární vyhledávání, hledáme, zda 3 je v poli A.

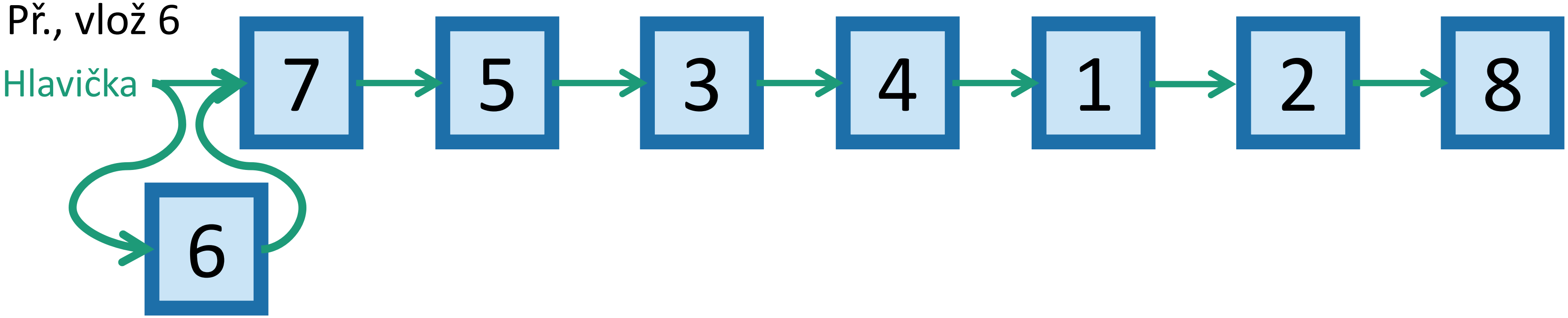
(Ne nutně uspořádaný)

Spojový seznam

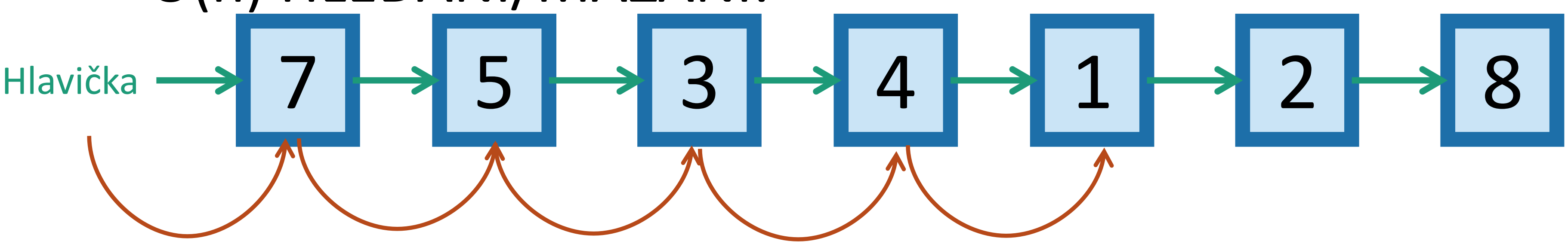


- O(1) VKLÁDÁNÍ:

Př., vlož 6



- O(n) HLEDÁNÍ/MAZÁNÍ:



Např. vyhledáme 1 (a pak jej můžeme smazat manipulací s ukazateli).

Motivace pro binární vyhledávací stromy

Dnes!

	Uspořádané pole	Spojový seznam	(Vyvážený) Binární vyhledávací strom
Hledání	O(log(n))	O(n)	O(log(n))
Mazání	O(n)	O(n)	O(log(n))
Vkládání	O(n)	O(1)	O(log(n))

Předpokládáme, že hodnoty jsou různé

Terminologie binárního stromu

Každý uzel má dvě děti.

Levé dítě (potomek) 3 je 2

Pravé dítě 3 je 4

Rodič 3 je 5

2 je následník 5

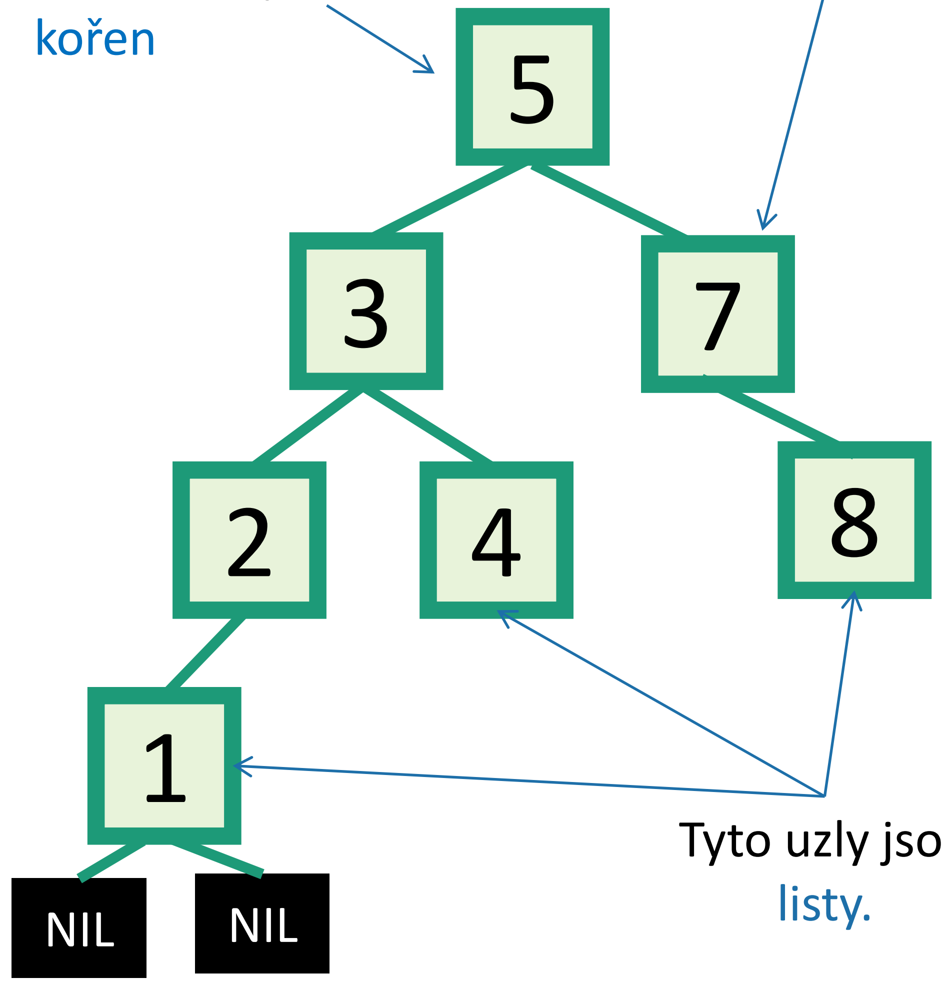
Každý uzel má ukazatel na své levé dítě a pravé dítě.

Obě děti 1 jsou NIL.
Obvykle se nekreslí).

Výška tohoto stromu je 3.
(Maximální počet hran od kořene po list).

Toto je uzel.
Jeho hodnota je (7).

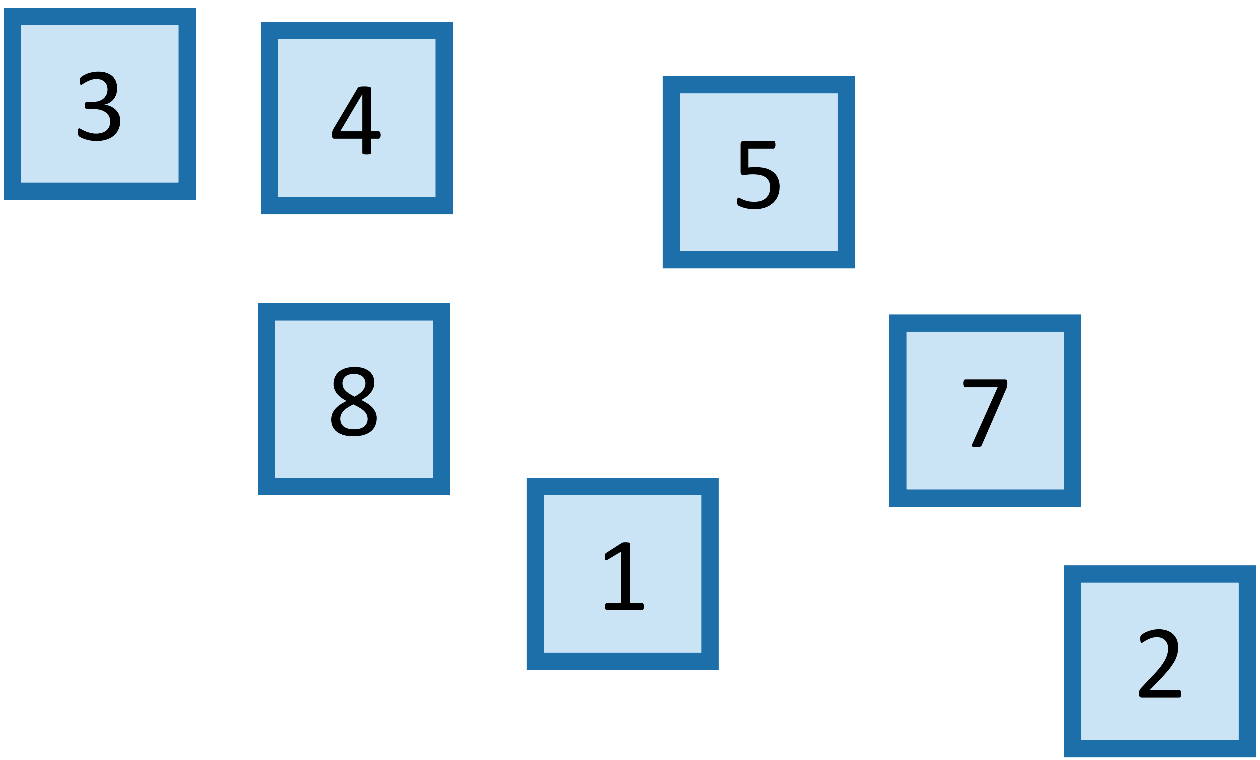
Tento uzel je kořen



Tyto uzly jsou listy.

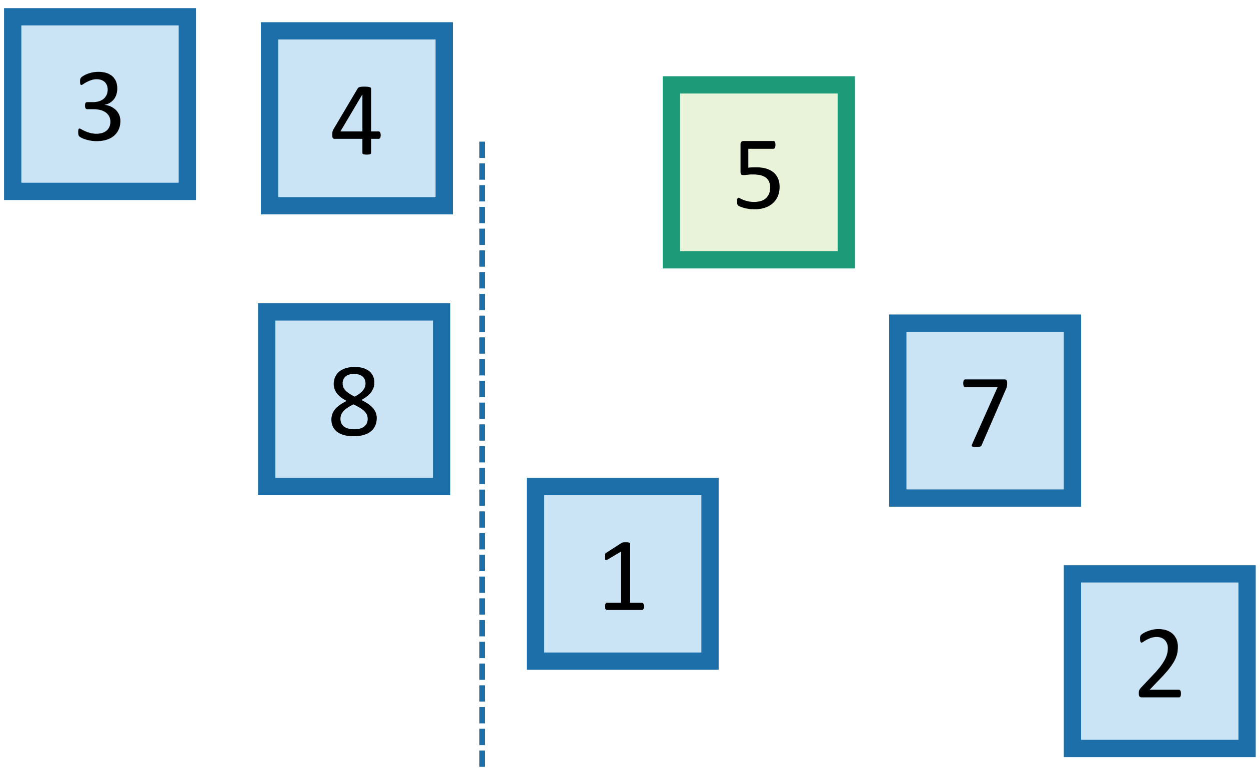
Binární vyhledávací stromy (BST)

- BST je binární strom takový, že:
  - Každý LEVÝ potomek uzlu má klíč menší než tento uzel.
  - Každý PRAVÝ potomek uzlu má klíč větší než tento uzel.
- Příklad sestavení binárního vyhledávacího stromu :



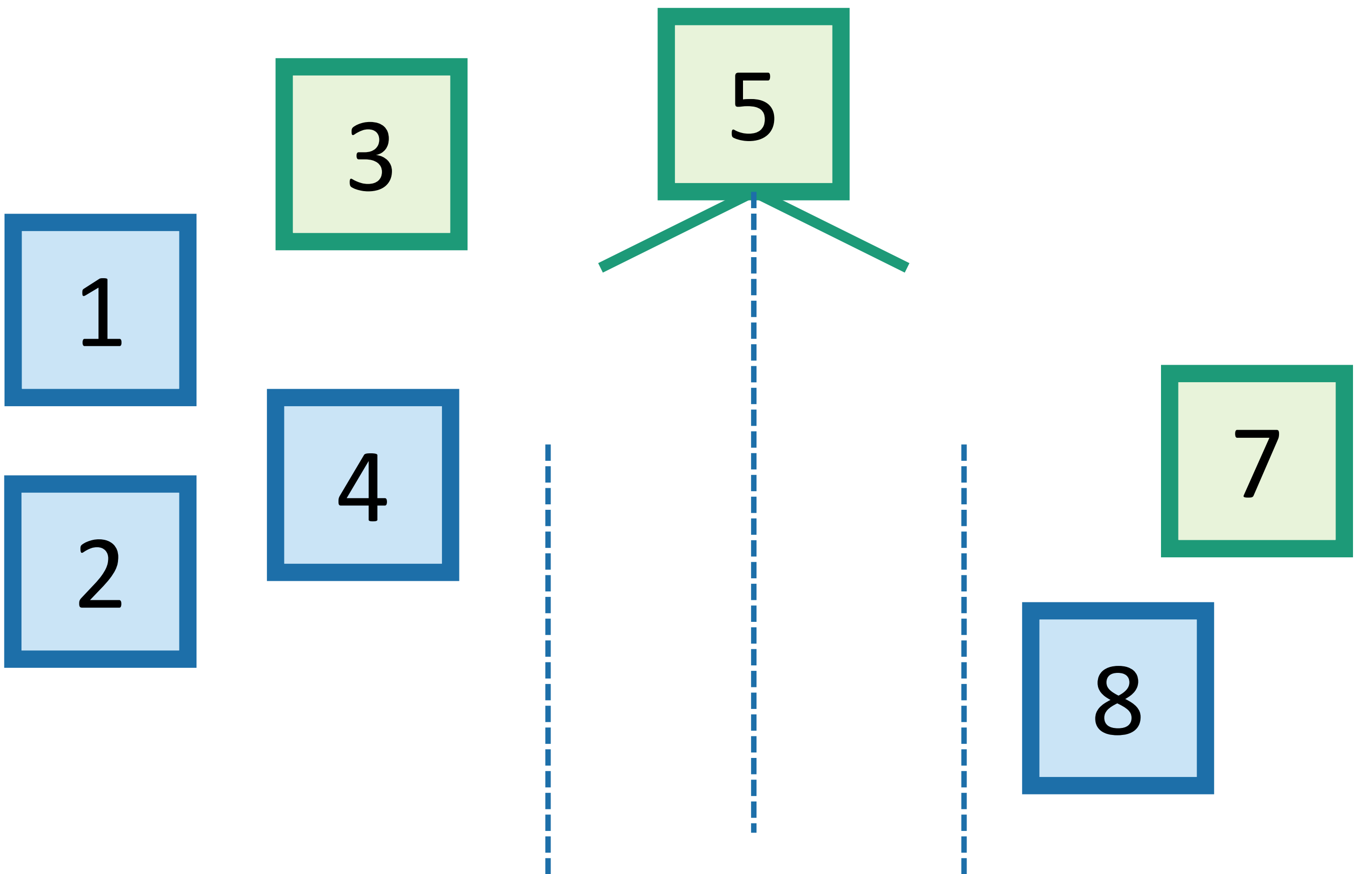
Binární vyhledávací stromy (BST)

- BST je binární strom takový, že:
  - Každý LEVÝ potomek uzlu má klíč menší než tento uzel.
  - Každý PRAVÝ potomek uzlu má klíč větší než tento uzel.
- Příklad sestavení binárního vyhledávacího stromu :



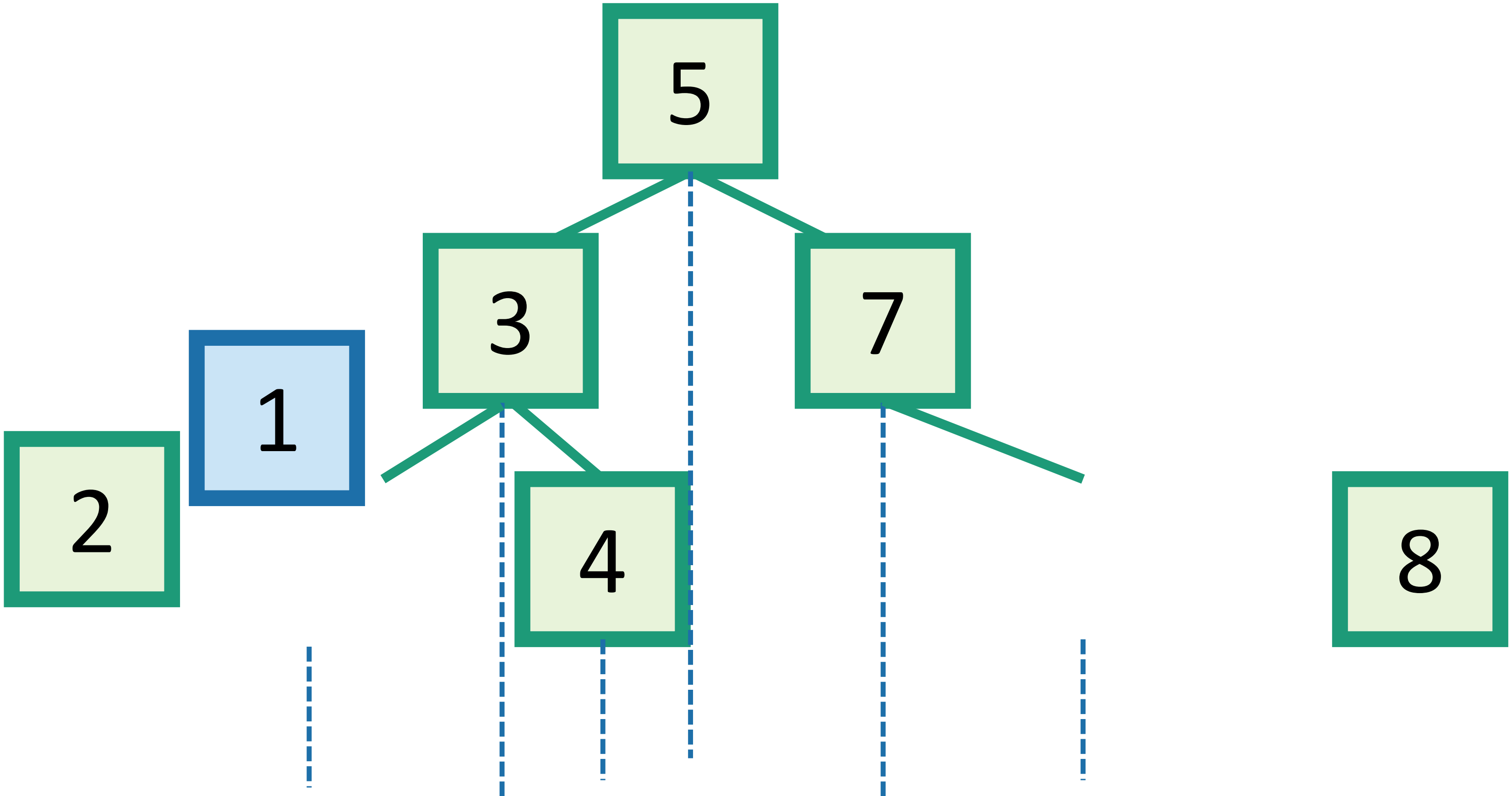
Binární vyhledávací stromy (BST)

- BST je binární strom takový, že:
  - Každý LEVÝ potomek uzlu má klíč menší než tento uzel.
  - Každý PRAVÝ potomek uzlu má klíč větší než tento uzel.
- Příklad sestavení binárního vyhledávacího stromu :



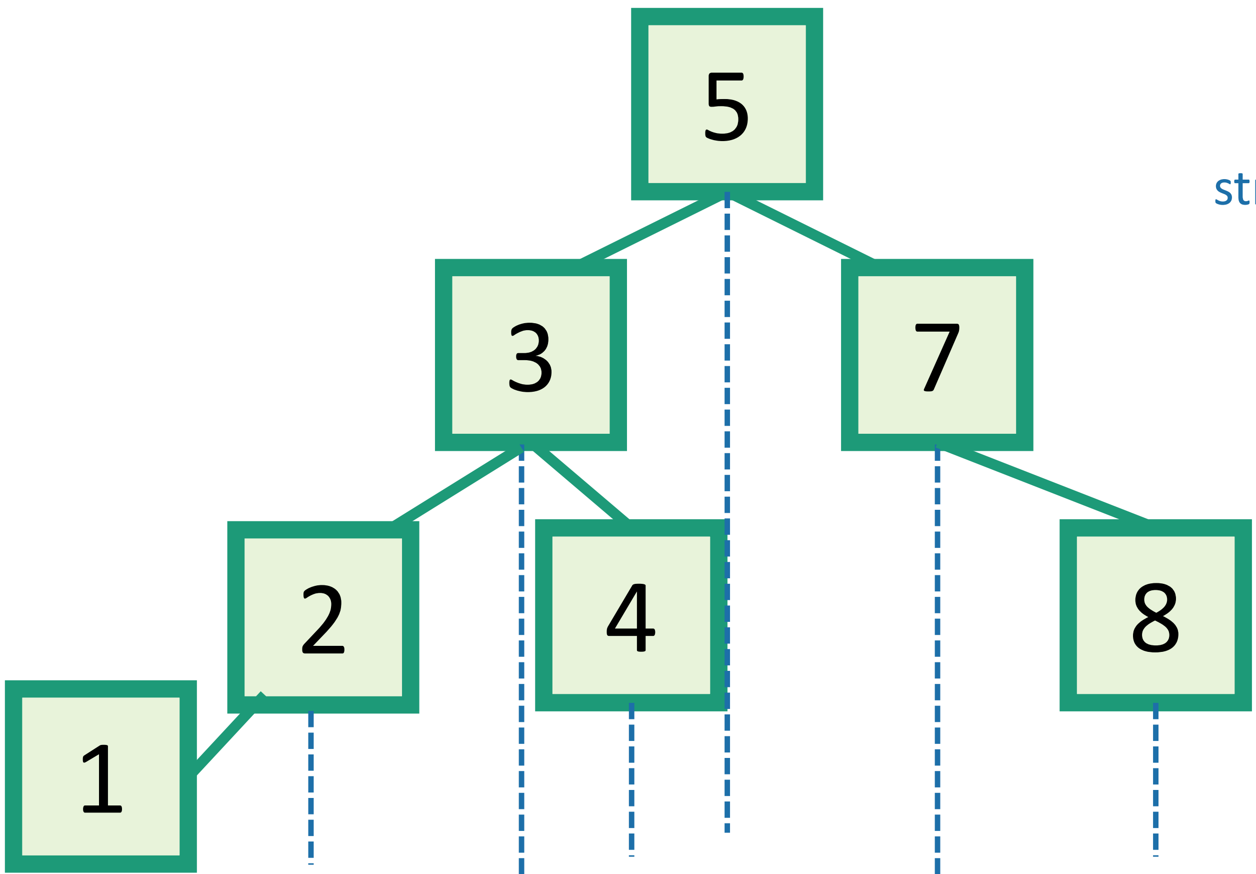
Binární vyhledávací stromy (BST)

- BST je binární strom takový, že:
  - Každý LEVÝ potomek uzlu má klíč menší než tento uzel.
  - Každý PRAVÝ potomek uzlu má klíč větší než tento uzel.
- Příklad sestavení binárního vyhledávacího stromu :



Binární vyhledávací stromy (BST)

- BST je binární strom takový, že:
  - Každý LEVÝ potomek uzlu má klíč menší než tento uzel.
  - Každý PRAVÝ potomek uzlu má klíč větší než tento uzel.
- Příklad sestavení binárního vyhledávacího stromu :

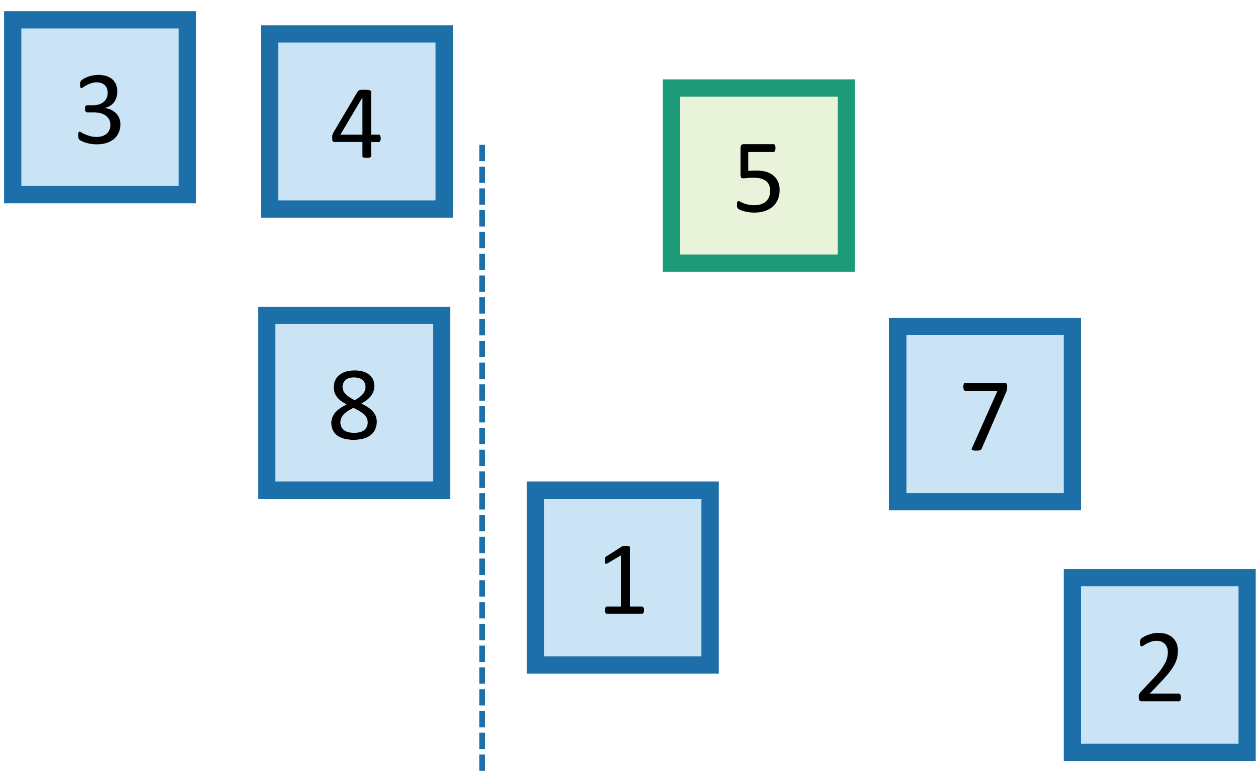


Otázka: Je to jediný binární vyhledávací strom, který bych mohl s těmito hodnotami vytvořit?

Odpověď: Ne. Rozhodl jsem se, které uzly kdy si vybrat. Jakákoli volba by byla v pořádku.

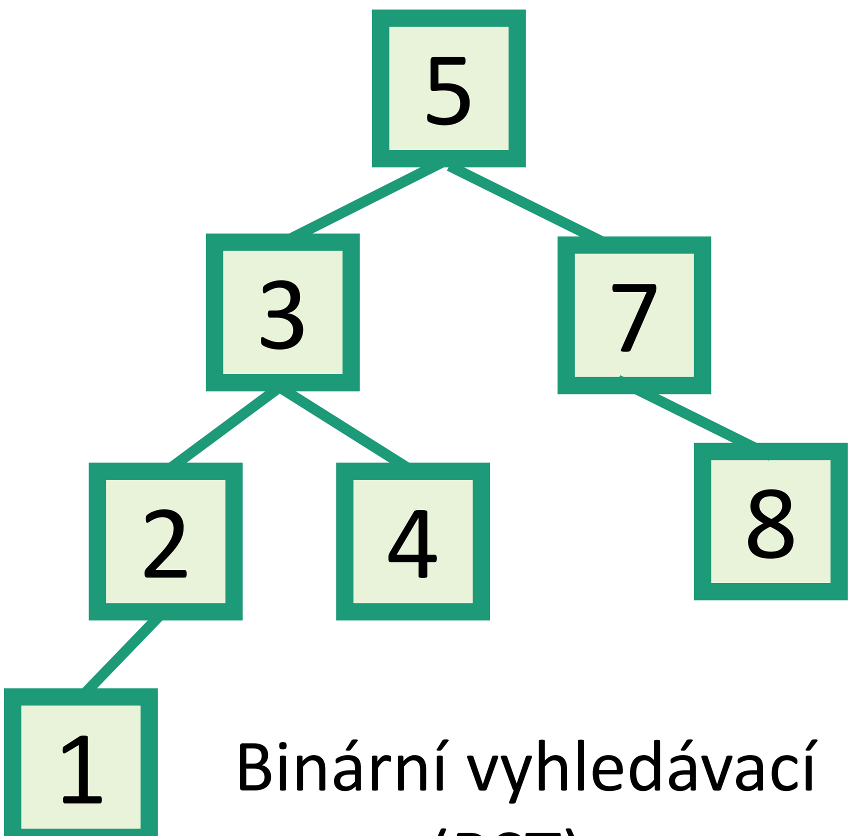
To ale vypadá povědomě

Úplně jako QuickSort

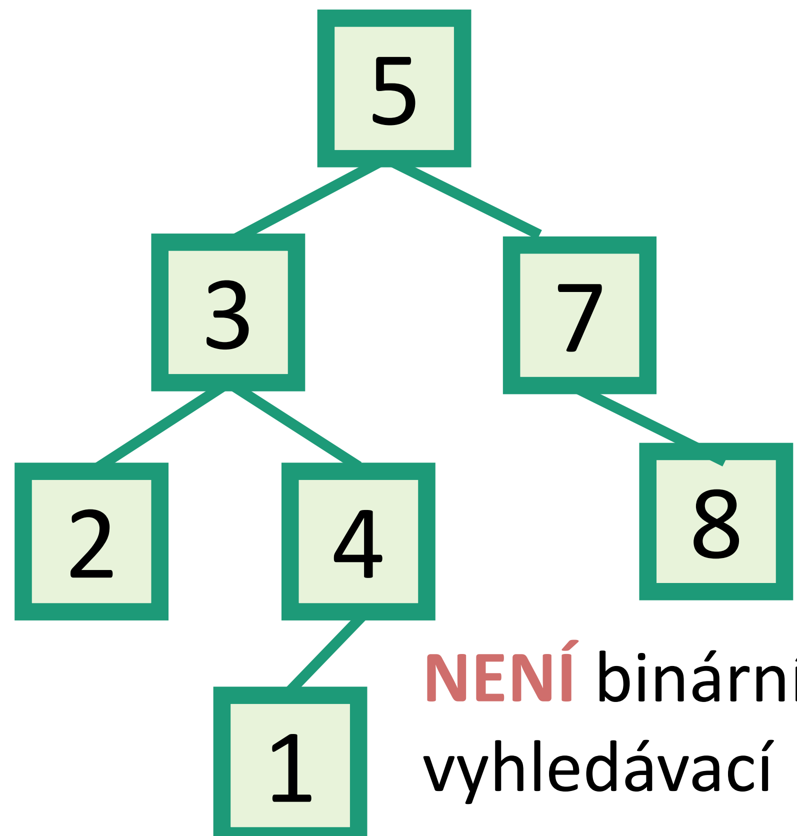


Binární vyhledávací stromy (BST)

- BST je binární strom takový, že:
    - Každý LEVÝ potomek uzlu má klíč menší než tento uzel.
    - Každý PRAVÝ potomek uzlu má klíč větší než tento uzel.



Binární vyhledávací strom (BST)

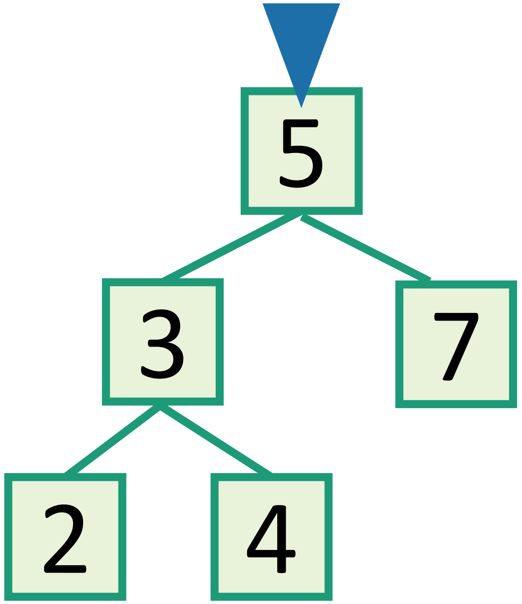


NENÍ binární vyhledávací strom

Procházení BST - In-Order (odpovídá procházení do hloubky)

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)

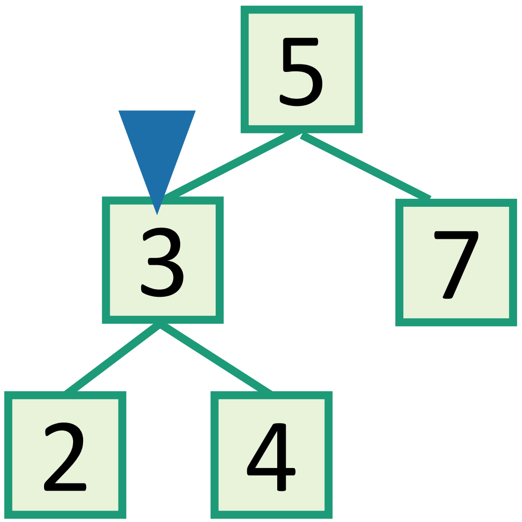


- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)

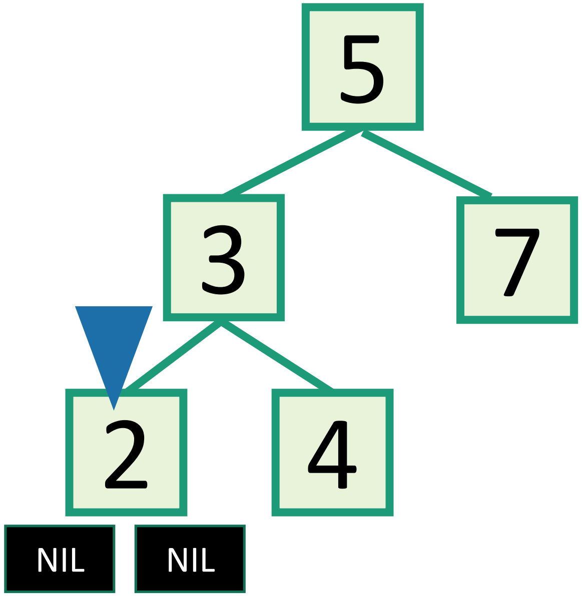


- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
    - if x!= NIL:
        - inOrderTraversal(x.left)
        - print(x.key)
        - inOrderTraversal(x.right)

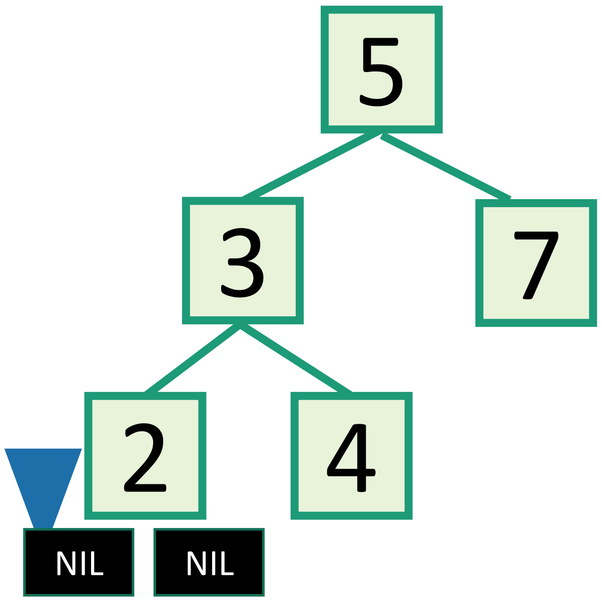


- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
    - if x!= NIL:
        - inOrderTraversal(x.left)
        - print(x.key)
        - inOrderTraversal(x.right)

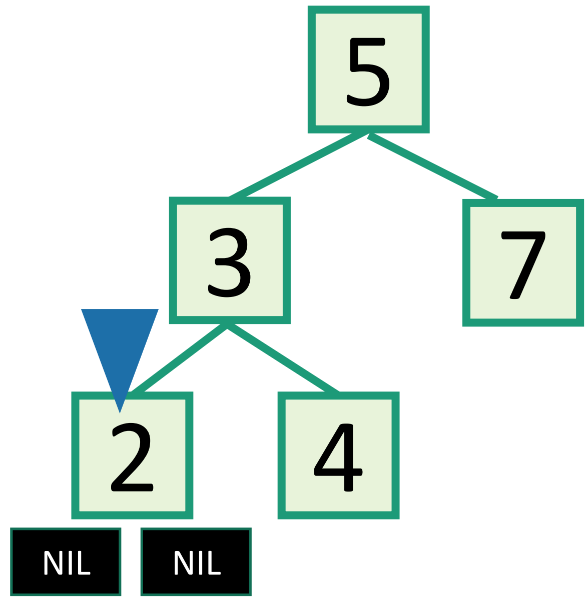


- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)

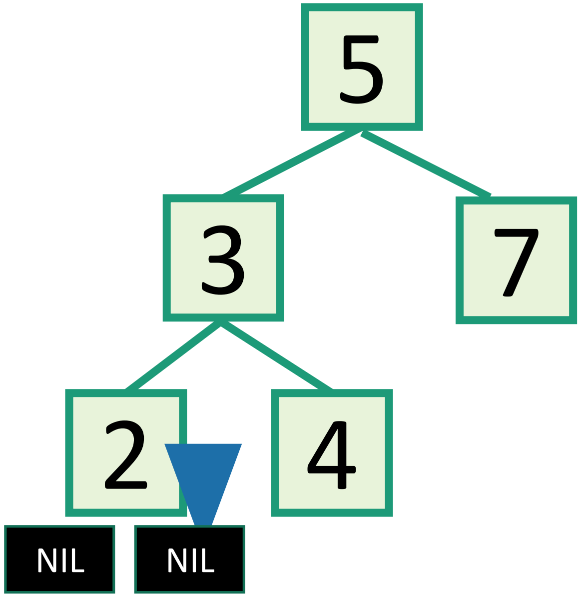


- Schéma procházení IN-ORDER: 2
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
    - if x!= NIL:
        - inOrderTraversal(x.left)
        - print(x.key)
        - inOrderTraversal(x.right)

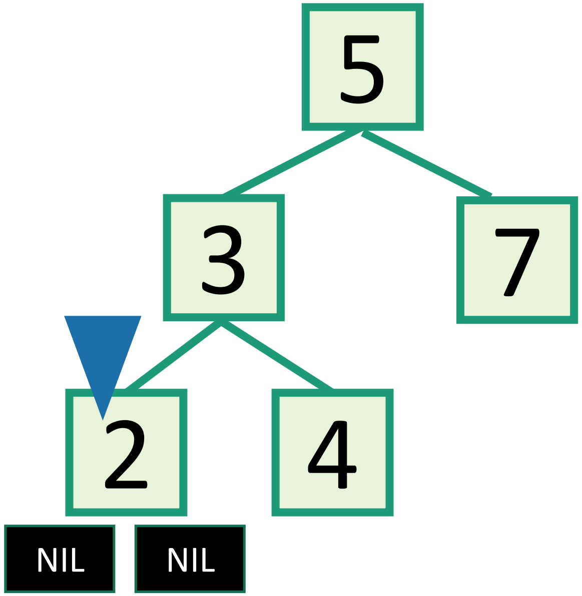


- Schéma procházení IN-ORDER: 2
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



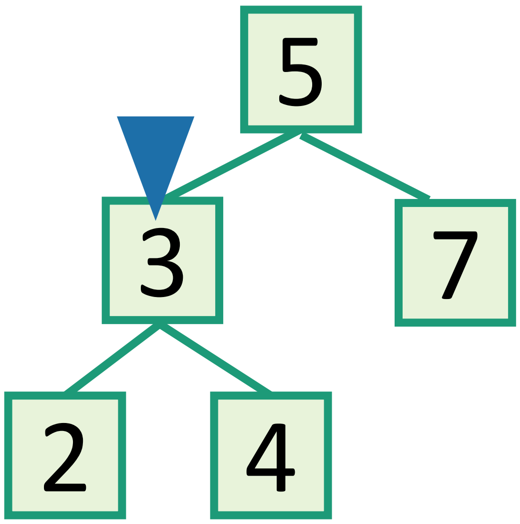
2

- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
    - if x!= NIL:
        - inOrderTraversal(x.left)
        - print(x.key)
        - inOrderTraversal(x.right)



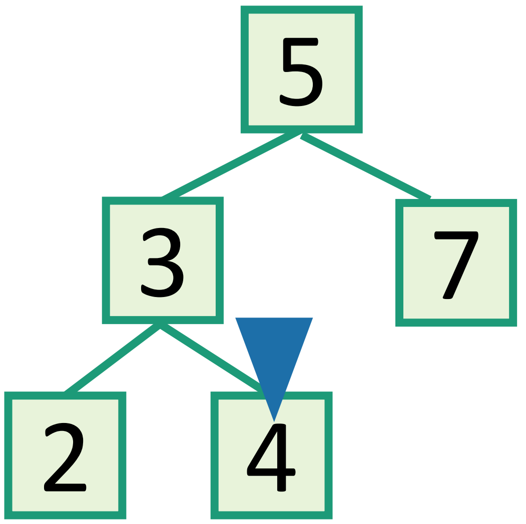
2 3

- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



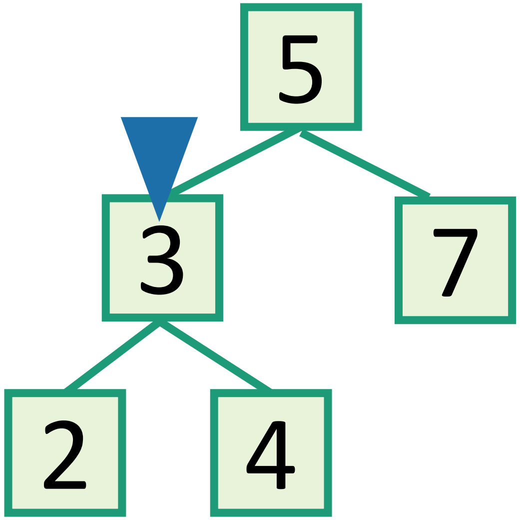
2 3 4

- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



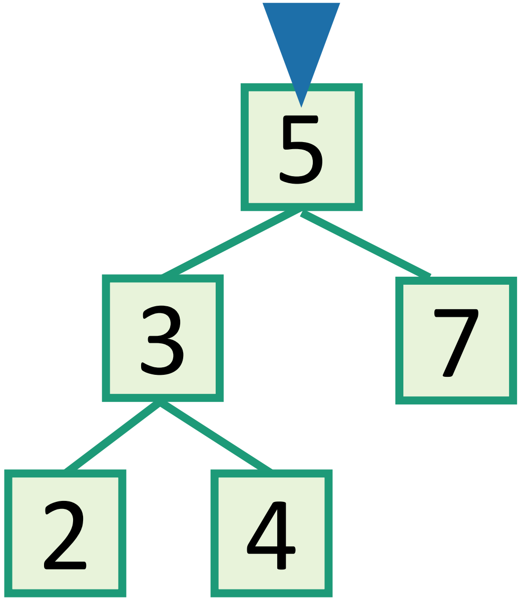
2 3 4

- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



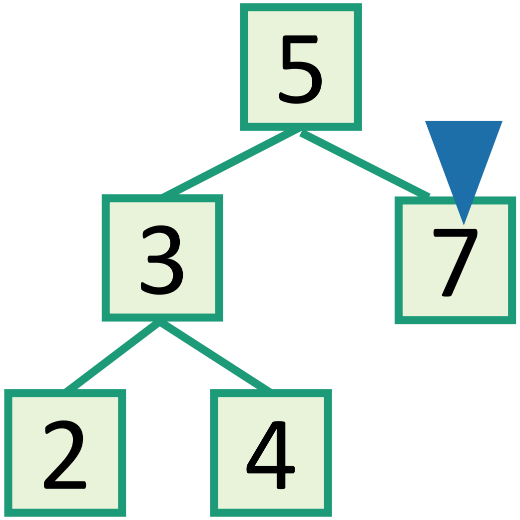
2 3 4 5

- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
    - if x!= NIL:
        - inOrderTraversal(x.left)
        - print(x.key)
        - inOrderTraversal(x.right)



2 3 4 5 7

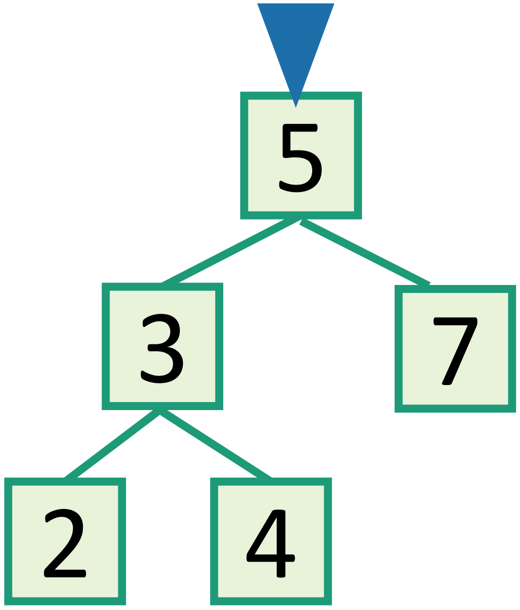
- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST - In-Order

- Výstup všech prvků v seřazeném pořadí!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)

- Doba běhu O(n).



2 3 4 5 7 Seřazeno!

- Schéma procházení IN-ORDER:
- Levý uzel – Kořen – Pravý uzel

Procházení BST (další možnosti)

- Procházení do hloubky – Pre-Order
- Schéma procházení PRE-ORDER:
- Kořen – Levý uzel – Pravý uzel

- Procházení do hloubky – Post-Order
- Schéma procházení POST-ORDER:
- Levý uzel – Pravý uzel – Kořen

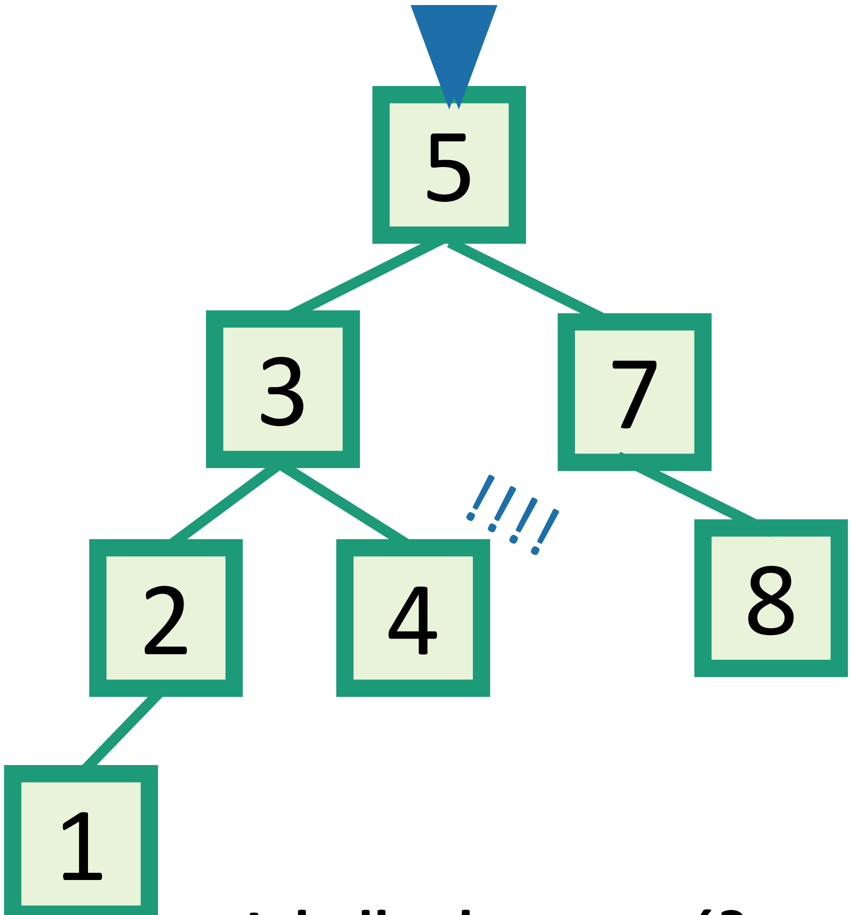
- Procházení do šířky (Level Order)
- Schéma procházení LEVEL-ORDER:
- Po úrovních

Zpět co je naším cílem

Rychlé VYHLEDÁVÁNÍ / VLOŽENÍ / MAZÁNÍ

Jak to uděláme?

HLEDÁNÍ v binárním vyhledávacím stromu
definice příkladem



Příklad: Hledat 4.

Příklad: Hledat 4.5

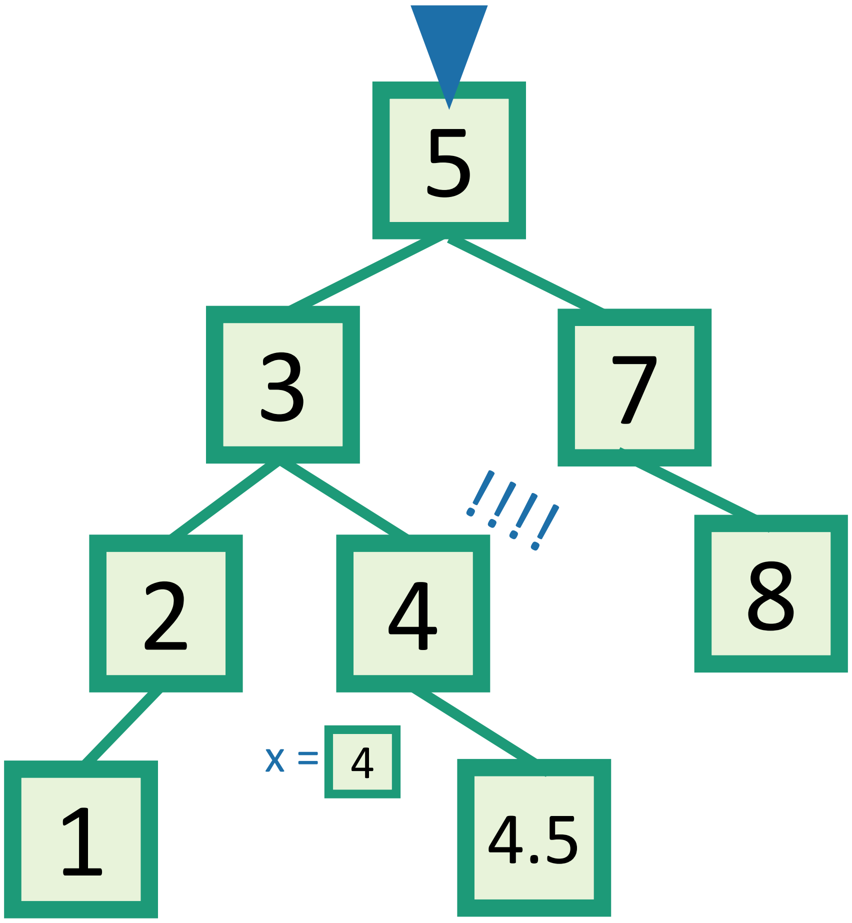
- Pokud hledaný prvek není ani v listu, pak se ve stromu nevyskytuje.

Zkuste napsat pseudokód!

Jak dlouho to trvá?

O(délka nejdelší cesty) = O(výšky)

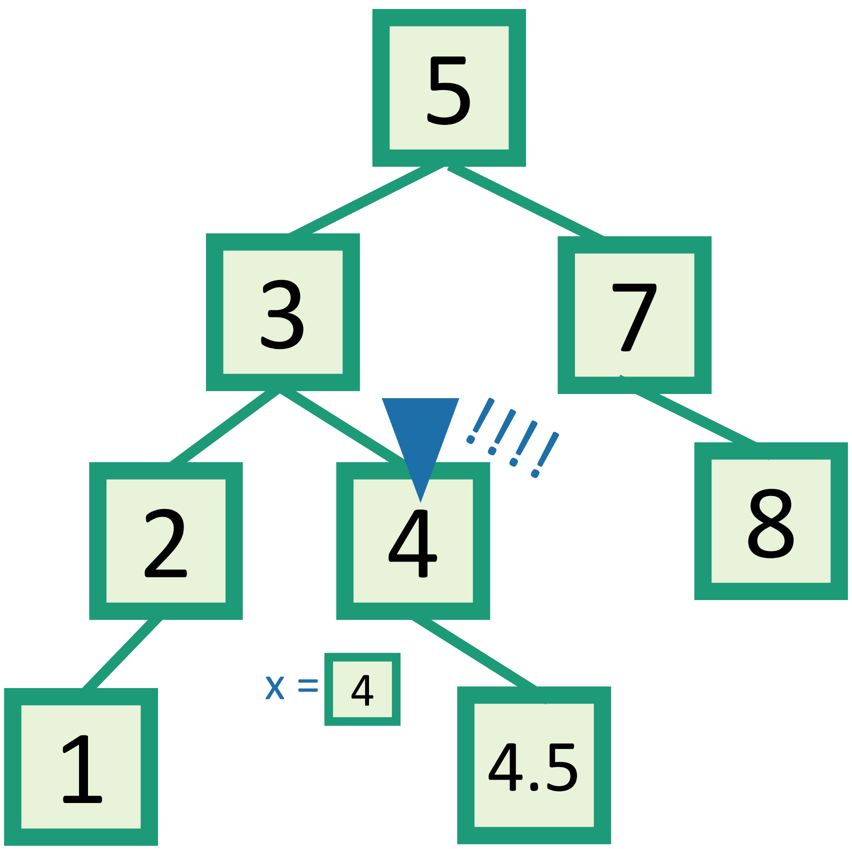
VKLÁDÁNÍ v BST



Příklad: Vložení 4.5

- INSERT(key):
  - x = SEARCH(key)
  - **Vlož** nová uzel s požadovanou hodnotou (klíčem) za pozici za x… (vlevo? vpravo?)

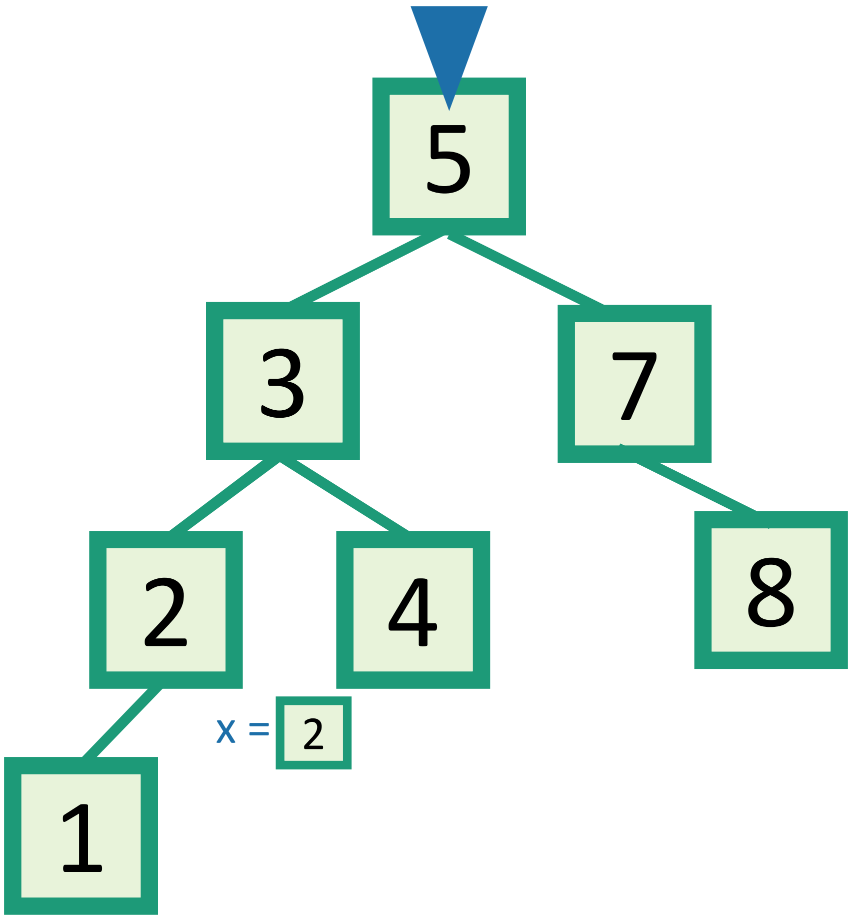
VKLÁDÁNÍ v BST



Příklad: Vložení 4.5

- INSERT(key):
  - x = SEARCH(key)
  - **if** key > x.key:
    - Vytvořte nový uzel se správným klíčem a vložte jej jako pravé dítě uzlu x.
    - **if** key < x.key:
    - Vytvořte nový uzel se správným klíčem a vložte jej jako levé dítě uzlu x.
    - **if** x.key == key:
    - **return**

MAZÁNÍ u Binárního vyhledávacího stromu



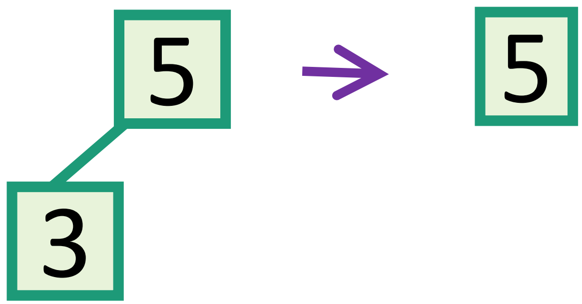
Příklad: Smažeme 2

- DELETE(key):
    - x = SEARCH(key)
    - **if** x.key == key:
        - ....smaž x....

MAZÁNÍ u Binárního vyhledávacího stromu

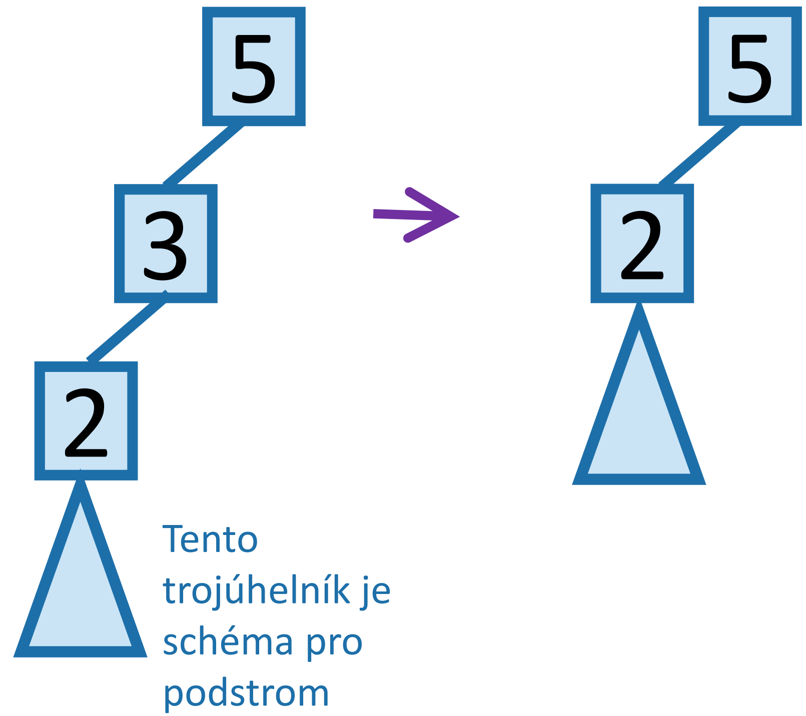
více případů (příklady)

řekněme, že chceme smazat 3



Případ 1: pokud 3 je list, smažeme ho.

Zkuste napsat pseudokód pro tyto případy !

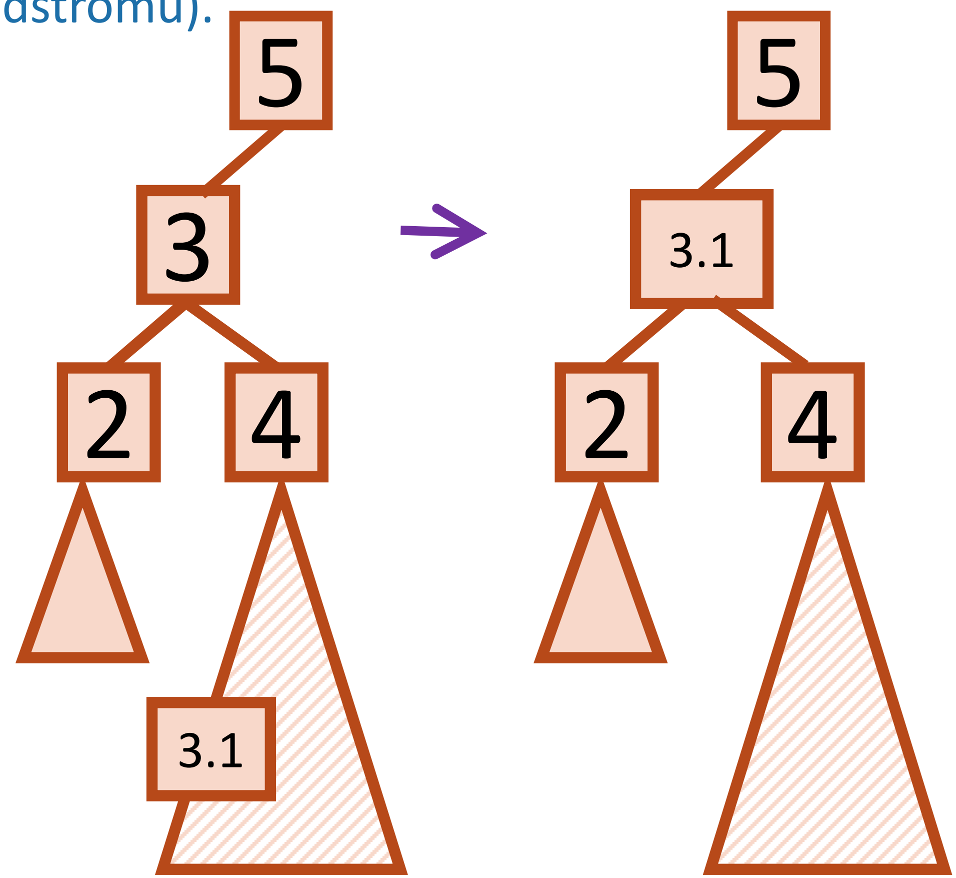


Případ 2: pokud má 3 jediné díte, potom toto dítě posuneme nahoru.

MAZÁNÍ u Binárního vyhledávacího stromu – **případ 3**

Pokud 3 má dvě děti, nahradíme 3 jeho bezprostředním následníkem.

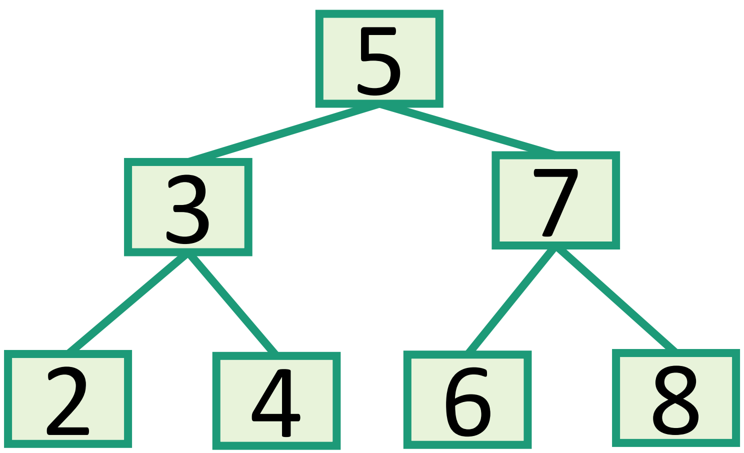
Tedy, prvkem nejvíce nalevo v jeho pravém podstromu (je to nejmenší prvek tohoto podstromu).



- Udržuje to vlastnost BST?
  - Ano.
- Jak najdeme bezprostředního následovníka?
  - HLEDÁME nejlevější prvek v pravém podstromu uzlu 3
- Jak ho odstraníme, když následníka najdeme (musíme totiž ho přemístit na pozici po 3?
  - Pokud má [3.1] 0 nebo 1 dítě, provedeme jeden z předchozích případů.
- Co když má [3.1] dvě děti?
  - Nemá (Proč)

Jak dlouho tyto operace trvají?

- HLEDÁNÍ je nejdelší operace.
  - Všechno ostatní používá HLEDÁNÍ a poté provede nějakou malou operaci O (1).



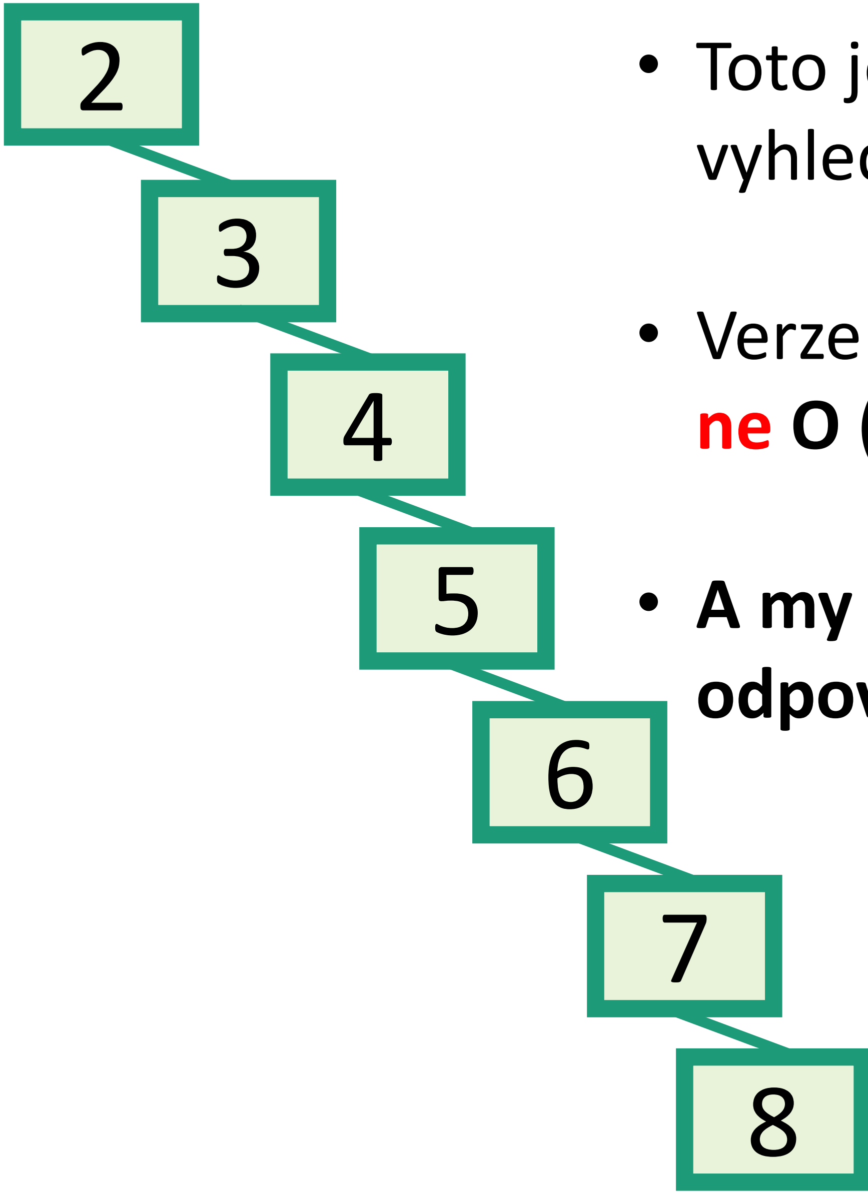
Doba běhu = O(výška stromu)

Stromy mají výšku
(nebo hloubku)
O(log(n)).
Hotovo!

(viz jak jsme dělali MergeSort)

Počítali jsme ale s tím, že strom je vyvážený,
tj. všechny „větve“ mají stejnou délku!

Hledání může také ale trvat dobu O(n).



- Toto je platný binární vyhledávací strom.
- Verze s n uzly má ale hloubku **n**, **ne O (log (n)).**
- **A my víme, že vyhledávání v BST odpovídá O(hloubka stromu)**

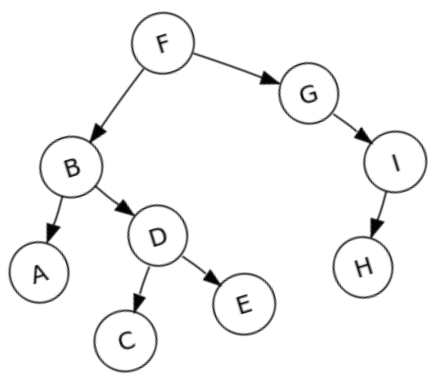
Co dělat?

- Cíl: Rychlé HLEDÁNÍ / VKLÁDÁNÍ / MAZÁNÍ
- Všechny tyto operace vyžadují čas O(výška stromu)
- A výška může být velká!!! ☹

- Nápad 0:
  - Sledovat, jak hluboký se strom stává.
  - Pokud je příliš vysoký, provedeme vše znovu od nuly.
    - Ale nejméně $\Omega(n)$ pokaždé tak často….

- Není to úplně skvělý nápad (mnoho operací navíc = velká doba běhu). Místo toho zavedeme k BST operace vyvažování (tedy budeme mluvit o vyvažovaných BST) …

Vyvažované binární vyhledávací stromy

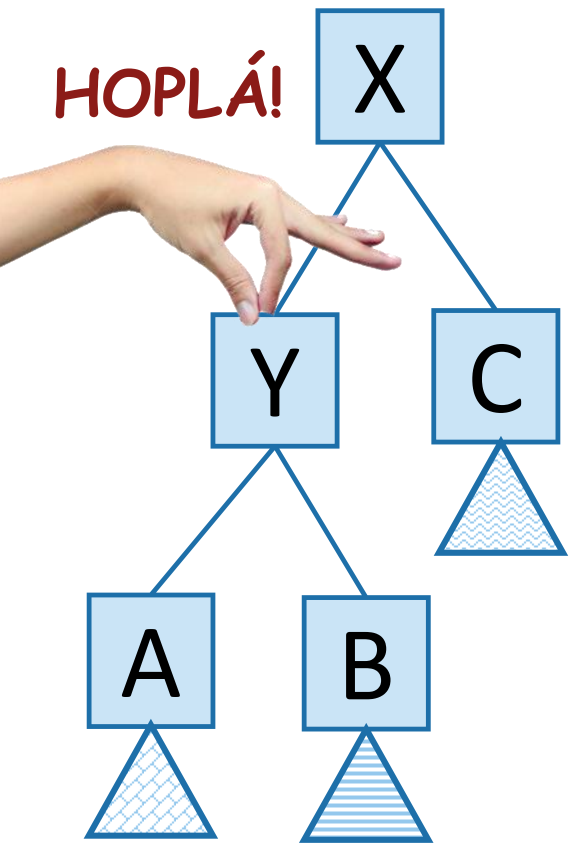


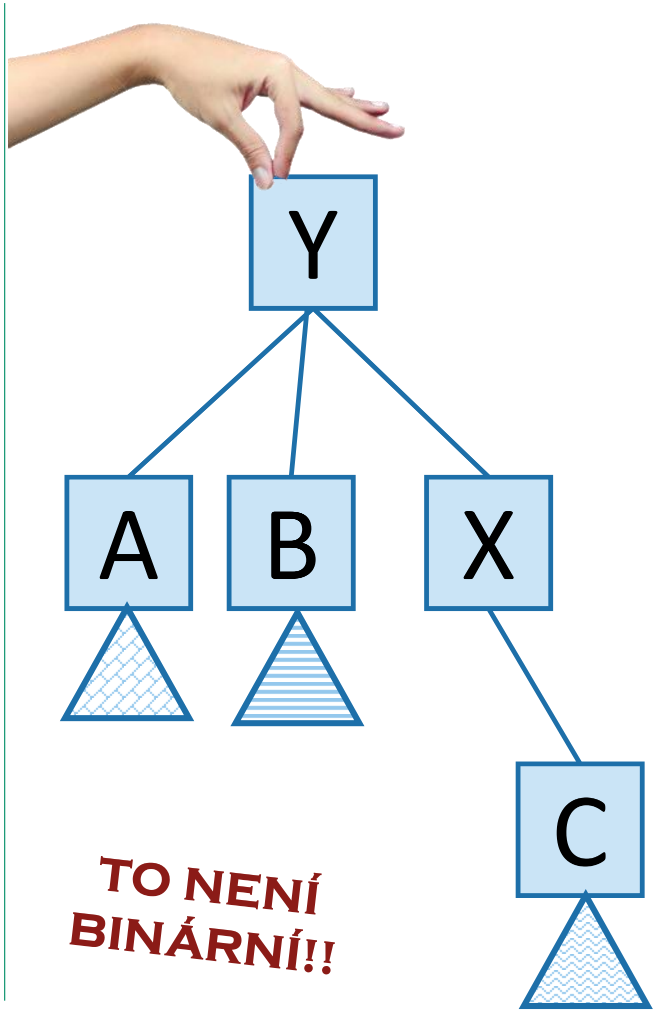
Nápad č. 1: Rotace

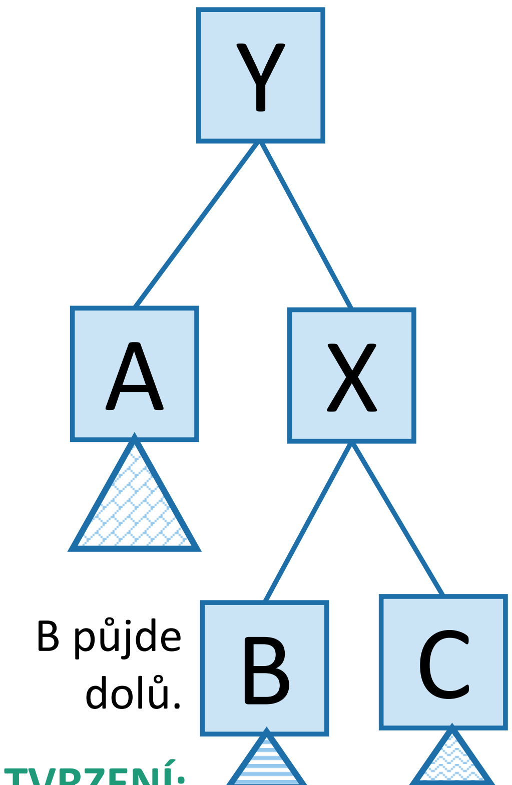
Bez ohledu na to, co máme pod A, B, C, to vyžaduje čas O (1). (Proč?)

- Udržujeme vlastnost binárního vyhledávacího stromu (BST) a přemisťujeme prvky.

Poznámka: A, B, C, X, Y jsou názvy proměnných, nikoli obsah uzlů.

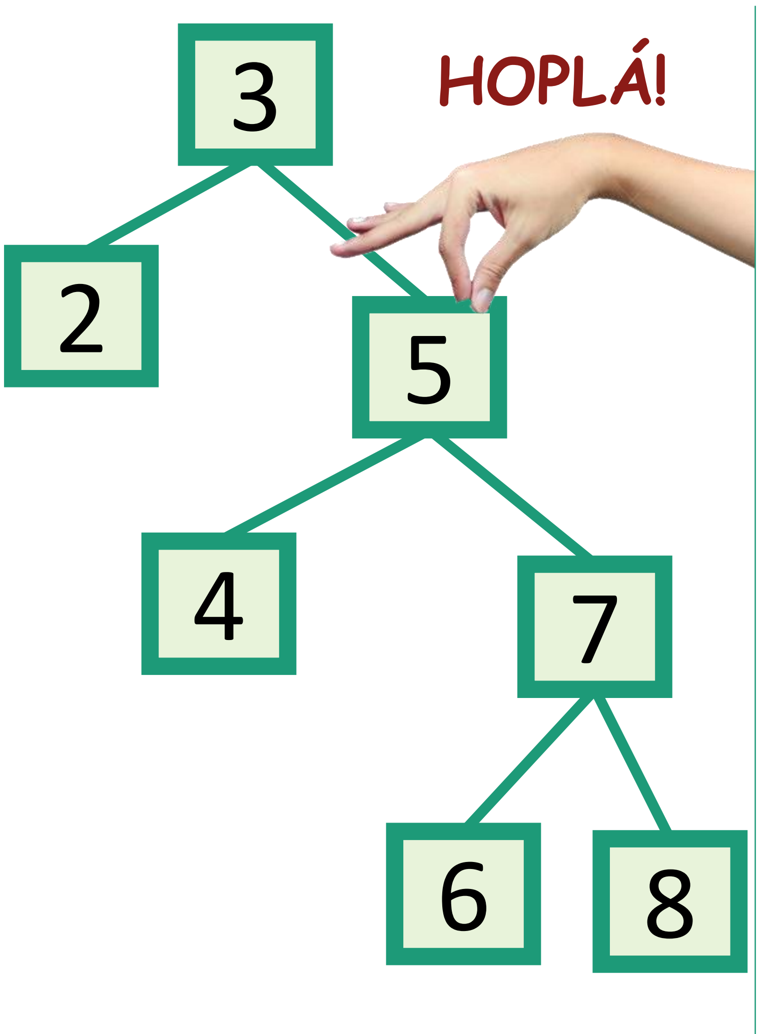


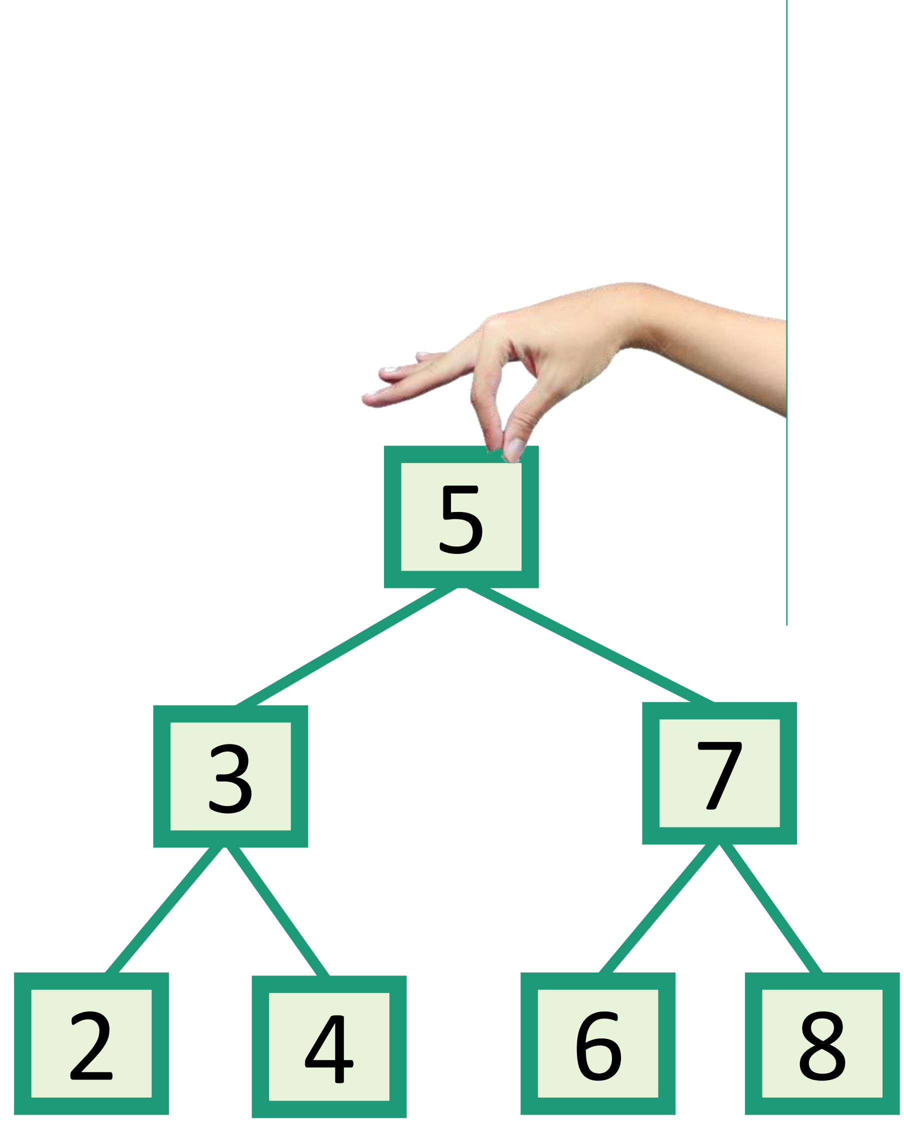




TVRZENÍ: má vlastnost BST.

To se zdá užitečné





Strategie?

- Kdykoli se něco zdá nevyvážené, provádíme rotaci, dokud to není v pořádku.

Nápad č. 2: zavést nějakou vlastnost pro zajištění vyváženosti stromu

- Udržování dokonalého vyvážení je příliš nákladné.
- Místo toho přijdeme s nějakou náhradou pro zajištění vyváženosti (rovnováhy):
  - Pokud strom splňuje [NĚJAKOU VLASTNOST], potom je přibližně vyvážený.
  - Můžeme udržovat [NĚJAKOU VLASTNOST] použitím rotací.

Ve skutečnosti existuje několik způsobů, jak toho dosáhnout, ale my budeme popisovat (příště) red-black stromy …

Příště

- **Vyvažování**