Vyvážené stromy

Cíl: zajistit výšku stromu O(log *N*)
$\rightarrow$ v případě BVS časová složitost všech operací O(log *N*)

Dokonale vyvážený binární strom
pro každý uzel platí:
počet uzlů v jeho levém a pravém podstromu se liší nejvýše o 1

- nejlepší možné vyvážení, výška stromu s *N* uzly je $\lceil \log N \rceil$
- lze snadno postavit z předem známé množiny hodnot
- je obtížné udržovat strom dokonale vyvážený při přidávání a odebírání hodnot

$\rightarrow$ proto se v praxi používají jiné (slabší) definice vyváženosti, strom nebude tak dokonale vyvážený, ale půjde snadněji udržovat

Postavení dokonale vyváženého binárního stromu s *N* vrcholy

```
def postav(n):
    """
      postavení dokonale vyváženého
      binárního stromu s "n" vrcholy
    """
    if n == 0:
        return None
    p = Vrchol()
    p.levy = postav((n-1)//2)
    p.pravy = postav(n-1 - (n-1)//2)
    return p
```

Funkce vrací ukazatel na kořen sestrojeného stromu.
Hodnoty „info“ ve vrcholech stromu zatím nejsou definovány.

Postavení dokonale vyváženého binárního vyhledávacího stromu
s danými *N* hodnotami ve vrcholech stromu

1. varianta řešení:
- ukládané hodnoty uspořádat vzestupně
- postavit dokonale vyvážený binární strom s *N* vrcholy pomocí předchozí funkce „postav“ („info“ hodnoty vrcholů zatím nejsou definovány)
- projít sestrojený strom metodou inorder a přitom do vrcholů stromu postupně zapisovat hodnoty v pořadí od nejmenší po největší

2. varianta řešení:
- ukládané hodnoty uspořádat vzestupně (seznam ***a***)
- při konstrukci stromu rovnou vkládat do info-položek vrcholů hodnoty
- parametry funkce „strom“ určují rozsah indexů v seznamu ***a***,
tzn. udávají, které hodnoty ze seznamu ***a*** patří do příslušného podstromu
- funkce bude volána „strom(0, N-1)“, vrací ukazatel na kořen sestrojeného stromu

```
def strom(a, x, y):
    """
      postavení dokonale vyváženého binárního
      stromu s hodnotami z uspořádaného seznamu "a"
      v úseku od indexu "x" po index "y" včetně
    """
    if x > y:
        return None
    p = Vrchol(a[(x+y)//2])
    p.levy = strom(a, x, (x+y)//2 - 1)
    p.pravy = strom(a, (x+y)//2 + 1, y)
    return p
```

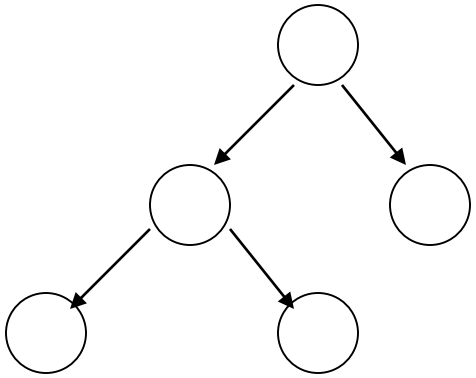
Výškově vyvážený binární strom

AVL – strom (G. M. Adeľson-Velskij, E. M. Landis, 1962)
pro každý uzel platí:
výška jeho levého a pravého podstromu se liší nejvýše o 1

- slabší požadavek, ale stačí: AVL-strom je maximálně o 45% vyšší než dokonale vyvážený strom se stejným počtem uzlů

- každý dokonale vyvážený strom je AVL-stromem
- AVL-strom nemusí být dokonale vyvážený

Příklad:



Realizace:

V každém uzlu *p* je navíc uložena položka „*balance*“, jejíž hodnota -1, 0 nebo 1 určuje, jak se liší výška levého a pravého podstromu tohoto uzlu:

balance(*p*) = výška(*p.levy*) - výška(*p.pravy*)

Pomocí této technické položky lze do AVL-stromu hodnoty snadno přidávat a z něj odebírat – s časovou složitostí O(log *N*).

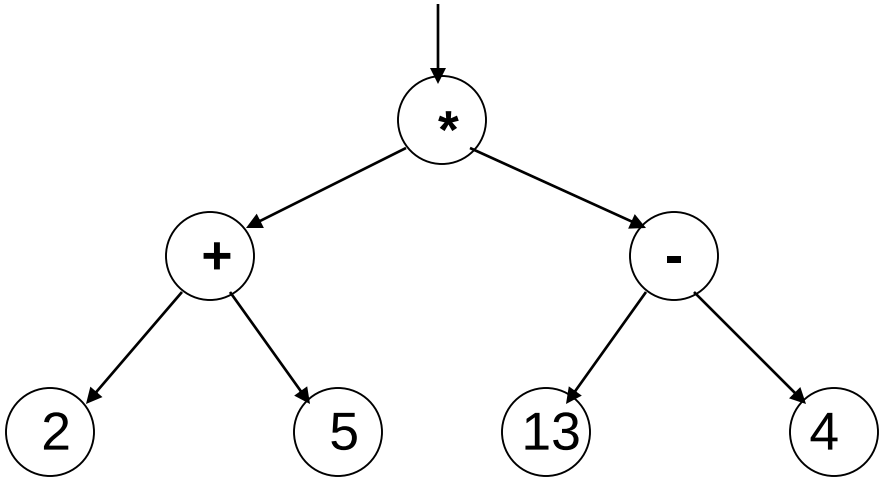
Výška AVL-stromu s N uzly:

- minimálně $\log_2(N)$ – úplný binární strom s N uzly
- nepřesáhne $1{,}45 \log_2(N)$ – důkaz pomocí Fibonacciho čísel

Reprezentace aritmetického výrazu

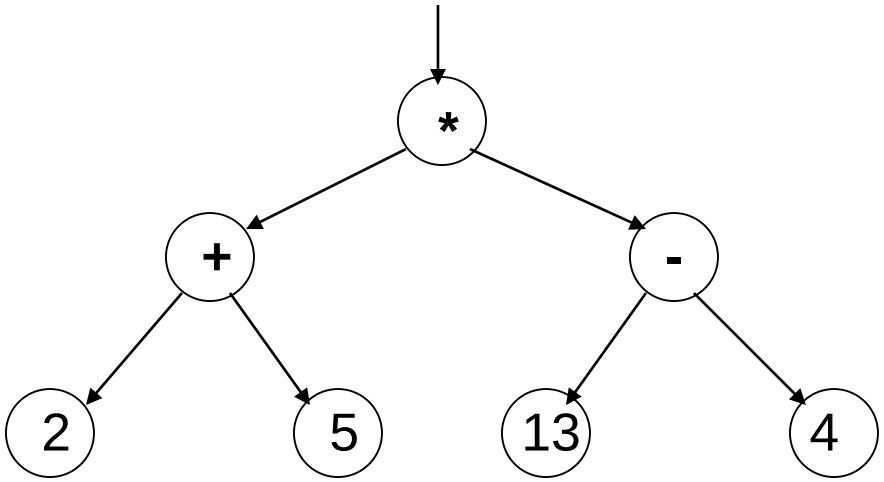
- binární strom reprezentující aritmetický výraz

(2 + 5) * (13 - 4)



- listy stromu obsahují operandy (čísla)
- vnitřní uzly obsahují operátory (znaménka)
- závorky ve stromě nejsou,
  pořadí vyhodnocení je určeno strukturou stromu

```
class Vrchol:
    """vrchol binárního stromu"""

    def __init__(self, x = None):
        self.info = x               # uložená hodnota
        self.levy = None            # levý syn
        self.pravy = None           # pravý syn
```




```
v = Vrchol('*')
v.levy = Vrchol('+')
v.levy.levy = Vrchol(2)
v.levy.pravy = Vrchol(5)
v.pravy = Vrchol('-')
v.pravy.levy = Vrchol(13)
v.pravy.pravy = Vrchol(4)

print(v.vyraz())
```

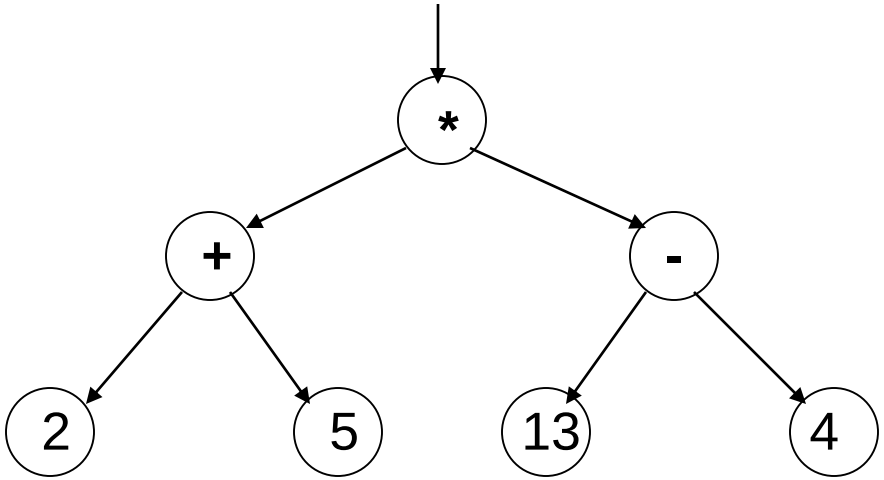
Vyhodnocení aritmetického výrazu reprezentovaného binárním stromem – rekurzívně (metoda Rozděl a panuj):

```
def vyraz(self):
    """vyhodnocení aritmetického výrazu reprezentovaného
       stromem s kořenem v tomto vrcholu
    """
    if self.levy == None:      # list
        return self.info
    elif self.info == '+':
        return self.levy.vyraz() + self.pravy.vyraz()
    elif self.info == '-':
        return self.levy.vyraz() - self.pravy.vyraz()
    elif self.info == '*':
        return self.levy.vyraz() * self.pravy.vyraz()
    elif self.info == '/':
        return self.levy.vyraz() / self.pravy.vyraz()
```

Notace aritmetického výrazu

- průchod binárním stromem reprezentujícím aritmetický výraz
- v navštívených uzlech vypisujeme uloženou hodnotu

(2 + 5) * (13 - 4)



průchod preorder → PREFIX	* + 2 5 - 13 4
průchod inorder → INFIX (bez závorek!)	2 + 5 * 13 - 4
průchod postorder → POSTFIX	2 5 + 13 4 - *

```
class Vrchol:
    """vrchol binárního stromu"""

    def __init__(self, x = None):
        self.info = x               # uložená hodnota
        self.levy = None            # levý syn
        self.pravy = None           # pravý syn

    def preorder(self):
        """průchod stromem s kořenem v tomto vrcholu
           metodou preorder, vypisuje hodnoty všech
           vrcholů
        """
        print(self.info)
        if self.levy != None:
            self.levy.preorder()
        if self.pravy != None:
            self.pravy.preorder()
```

```
def inorder(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou inorder, vypisuje hodnoty všech
       vrcholů"""
    if self.levy != None:
        self.levy.inorder()
    print(self.info)
    if self.pravy != None:
        self.pravy.inorder()

def postorder(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou postorder, vypisuje hodnoty všech
       vrcholů"""
    if self.levy != None:
        self.levy.postorder()
    if self.pravy != None:
        self.pravy.postorder()
    print(self.info)
```

průchod preorder → PREFIX	* + 2 5 - 13 4
průchod inorder → INFIX (bez závorek!)	2 + 5 * 13 - 4
průchod postorder → POSTFIX	2 5 + 13 4 - *

- vždy stejné pořadí operandů – listy stromu procházíme ve všech případech zleva doprava (2 5 13 4)
- v prefixovém zápisu operátor bezprostředně předchází své dva argumenty (tzn. čísla nebo podvýrazy), v postfixovém je následuje
- v prefixovém a postfixovém zápisu výrazu nejsou závorky, pořadí vyhodnocování je plně určenou strukturou výrazu
- inorder průchod stromem vytvořil chybný infixový zápis bez závorek, z něhož není zřejmé správné pořadí vyhodnocování výrazu

Terminologická poznámka:
prefix = polská notace (Polish notation) – Łukasiewicz
postfix = reverzní polská notace (reverse Polish notataion, RPN)

Získání správného infixového zápisu výrazu:

```
def infix(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou inorder, vypisuje hodnoty všech
       vrcholů
    """
    if self.levy == None:      # je to list
        print(self.info, end='')
    else:                      # není to list
        print('(', end='')
        self.levy.infix()
        print(self.info, end='')
        self.pravy.infix()
        print(')', end='')
```

Vyhodnocení výrazu v postfixové notaci

- snadné, využití např. dříve u kalkulaček, v překladačích
- jeden průchod zápisem výrazu zleva doprava
- používá zásobník na ukládání číselných hodnot

Postup zpracování postfixového zápisu:
číslo $\rightarrow$ vložit do zásobníku
znaménko $\rightarrow$ vyzvednout ze zásobníku horní dvě čísla
provést s nimi operaci určenou znaménkem
výsledek operace vložit do zásobníku
konec $\rightarrow$ na zásobníku je jediné číslo = hodnota výrazu

- pozor na pořadí operandů u nekomutativních operátorů
  (na vrcholu zásobníku je pravý operand, pod ním levý)
- časová složitost O(*N*), kde *N* je délka výrazu

```
class Stack:
    def __init__(self):
        ...
    def push(self, value):
        ...
    def pop(self):
        ...
    def count(self):
        ...

OPERATORS = {
    "+": (lambda a, b: a + b),
    "-": (lambda a, b: a - b),
    "*": (lambda a, b: a * b),
    "/": (lambda a, b: a // b)
}
```

```
def evaluate_postfix(expression):
    """
    vyhodnocení aritemetického výrazu v postfixu
    ve výrazu vše odděleno mezerami
    """
    parts = expression.split()
    stack = Stack()
    for part in parts:
        if part in OPERATORS.keys():
            arg1 = stack.pop()
            arg2 = stack.pop()
            result = OPERATORS[part](arg2, arg1)
            stack.push(result)
        else:
            stack.push(int(part))
    result = stack.pop()
    assert stack.count() == 0
    return result
```

Vyhodnocení výrazu v prefixové notaci

1. možnost:

- průchod výrazem **odzadu**, postup jako u postfixu
- pouze se změní pořadí operandů při zpracování znaménka:
  při vyzvednutí ze zásobníku je na vrcholu zásobníku levý operand, pod ním je pravý
- časová složitost O(*N*), kde *N* je délka výrazu

2. možnost:
- jeden průchod zápisem výrazu zleva doprava
- **zásobník** na ukládání znamének a číselných hodnot

Postup zpracování prefixového zápisu odpředu:
- znaménko nebo číslo → vložit do zásobníku
- když se tím na vrcholu zásobníku sejdou dvě čísla → vyzvednout je ze zásobníku, dále vyzvednout znaménko uložené pod nimi, provést s čísly operaci určenou znaménkem a výsledek operace vložit do zásobníku (což může opětovně vyvolat tentýž proces vyhodnocení)
- konec → na zásobníku je jediné číslo = hodnota výrazu

- pozor na pořadí operandů u nekomutativních operátorů (na vrcholu zásobníku je pravý operand, pod ním levý)
- časová složitost O(*N*), kde *N* je délka výrazu

3. možnost: **rekurze**

- rekurzivní funkce na vyčíslení prefixového zápisu od zadaného indexu
- globálně udržujeme pozici indexu

- když je prvním znakem výrazu číslice, výrazem je jen jedno číslo
  $\rightarrow$ funkce vrátí jeho hodnotu (a posune index za něj)
- když je prvním znakem znaménko Z, funkce posune index za něj, potom provede dvě rekurzivní volání sebe sama a s výsledky těchto volání vykoná operaci určenou znaménkem Z

- celkem se provede jeden průchod zápisem výrazu zleva doprava
- časová složitost O(*N*), kde *N* je délka výrazu

Převod infix → postfix

máme zadán aritmetický výraz v běžné infixové notaci,
chceme ho převést do postfixové notace

- provede se jeden průchod zápisem výrazu zleva doprava,
  tedy časová složitost O(*N*)
- používá zásobník na ukládání znamének
- v postfixovém zápisu jsou čísla ve stejném pořadí jako v infixovém,
  znaménka je proto třeba pozdržet na zásobníku, aby se dostala
  na správné místo až za svoje argumenty

Postup zpracování infixového zápisu:

číslo → zapsat přímo na výstup

levá závorka → vložit do zásobníku

pravá závorka → tuto závorku zrušit,
ze zásobníku postupně přenést na výstup všechna znaménka až k nejbližší uložené levé závorce,
pak tuto levou závorku ze zásobníku zrušit

znaménko → vložit do zásobníku,
předtím ale ze zásobníku postupně přenést na výstup všechna znaménka vyšší nebo stejné priority, nejvýše však k první uložené levé závorce

konec → ze zásobníku přenést na výstup všechna uložená znaménka

Vyhodnocení výrazu v infixové notaci

spojení dvou předchozích algoritmů:

- převod výrazu z infixu do postfixu v čase O(*N*)
- vyhodnocení postfixové notace v čase O(*N*)

→ celková časová i paměťová složitost **O(*N*)**

obě fáze výpočtu se mohu provádět

- buď postupně (s uložením vytvořené postfixové notace výrazu)
- nebo souběžně (tzn. vznikající postfixová notace se neukládá, ale rovnou se průběžně vyhodnocuje)
  → algoritmus používá dva zásobníky – jeden na znaménka a druhý na čísla

Postavení aritmetického binárního stromu ze zápisu výrazu

postfixová nebo prefixová notace
- algoritmus podobný jako při vyhodnocování výrazu,
do zásobníku se vždy ukládá odkaz na nově vytvořený uzel,
místo provádění operací se uzly s operandy zapojují pod uzel
s operátorem jako jeho synové

infixová notace
- nejprve výraz převedeme do postfixové notace,
z té pak postavíme aritmetický strom

Obecný strom

1. známe maximální stupeň větvení *M*

- podobná reprezentace jako u binárního stromu
- v každém uzlu je připraveno *M* odkazů na syny,
  z nich několik prvních je využito, ostatní mají hodnotu **`None`**
- v listu mají všechny odkazy na syny hodnotu **`None`**
- použitelné, pokud *M* předem známe a je dostatečně malé

2. obecné řešení

- v každém uzlu je uložen seznam odkazů na syny potřebné délky
- v listu je tento seznam prázdný
- *viz program na následující straně*

```
class Vrchol:
    """vrchol obecného stromu"""

    def __init__(self, x = None):
        self.info = x            # uložená hodnota
        self.synove = []         # seznam synů

    def pruchod(self):
        """
        průchod stromem s kořenem v tomto vrcholu
        metodou preorder, vypisuje hodnoty všech
        vrcholů
        """
        print(self.info)
        for x in self.synove:
            x.pruchod()
```

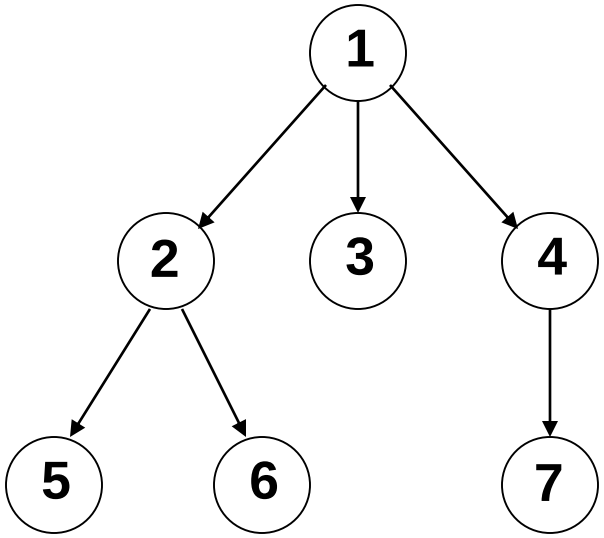
3. kanonická reprezentace

- reprezentace obecného stromu binárním stromem

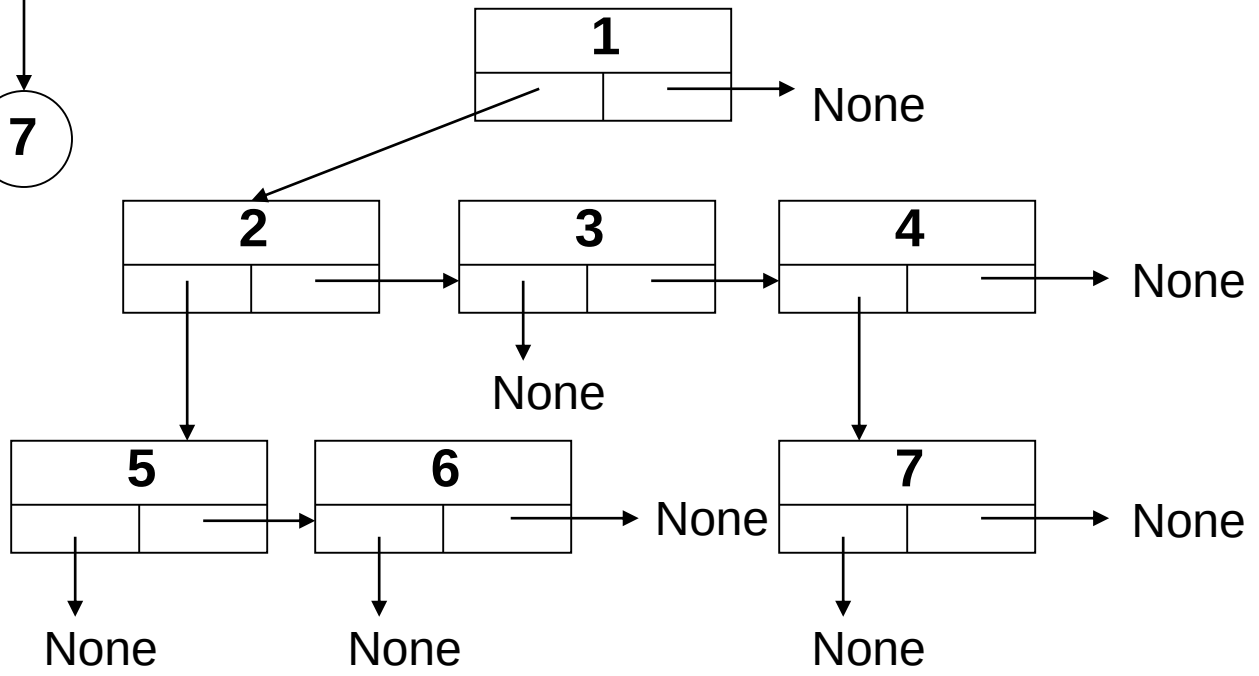
```
class Vrchol:
    """vrchol stromu"""

    def __init__(self, x = None):
        self.info = x               # uložená hodnota
        self.syn = None             # nejstarší syn
        self.bratr = None           # mladší bratr
```

- každý uzel ukazuje jen na svého nejstaršího syna (položka „syn“)
- všichni synové téhož uzlu jsou navzájem propojeni pomocí odkazů „bratr“
- v listu má položka „syn“ hodnotu **None**

příklad stromu:



uložení v datové struktuře:



Příklad použití obecného stromu: *písmenkový strom (trie)*

- datová struktura vhodná k uložení množiny slov a jejich rychlému hledání
- kořen = prázdné slovo, sestup po hladinách stromu podle písmen slova
  (v uzlech stromu je např. 26 odkazů na syny pode písmen ‘a’ až ‘z’)
- uzel s koncem slova je označen např. jeho číselným kódem,
  nebo překladem do cizího jazyka,
  příp. pouze příznakem typu **bool**, že zde končí slovo
- operace: hledání, přidání, odebrání slova – složitost O(délka slova)

- alternativní řešení téhož problému: použít hešování *(bude později)*
- v Pythonu (a některých jiných programovacích jazycích):
  datová struktura slovník, dictionary (**dict**)

Metody ukládání a vyhledávání dat – shrnutí

data = záznamy s klíčem, podle kterého vyhledáváme

- pole: vyhledávání O(*N*), vkládání O(1)
- uspořádané pole: binární vyhledávání O(log *N*), vkládání O(*N*)
- lineární spojový seznam: vyhledávání O(*N*), vkládání O(1)
  jako v poli
- uspořádaný LSS: vyhledávání O(*N*), vkládání O(N)
  průchod seznamem lze předčasně ukončit

- binární vyhledávací strom: všechny operace v průměru O(log *N*),
  v nejhorším případě O(*N*)
- vyvážený binární vyhledávací strom: všechny operace O(log *N*)
- vícecestný vyhledávací strom