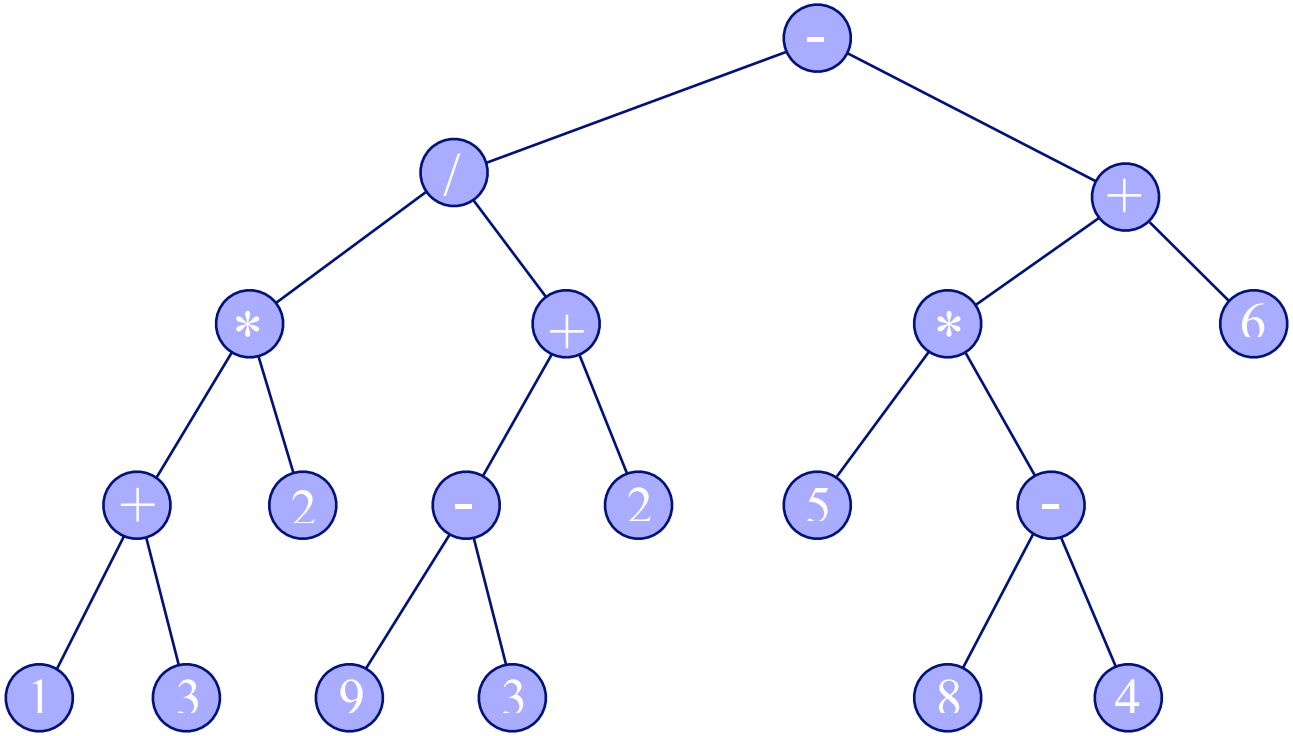
Algoritmizace

Rekurzivní datové struktury

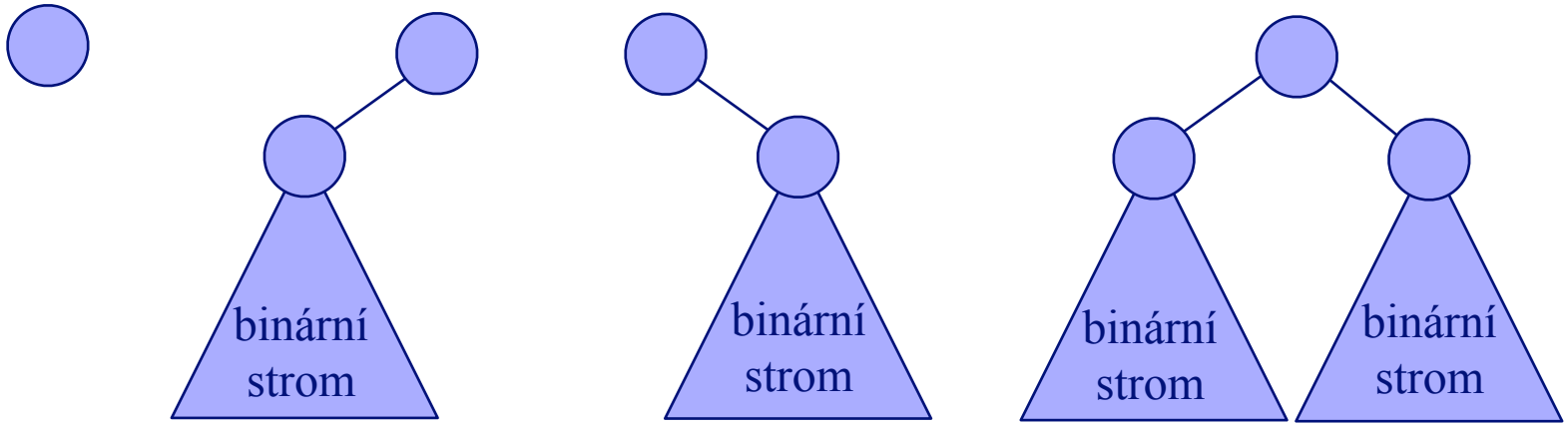


Binární stromy

Binární strom

- uspořádaný kořenový strom
- každý vrchol má nejvýše dvě děti

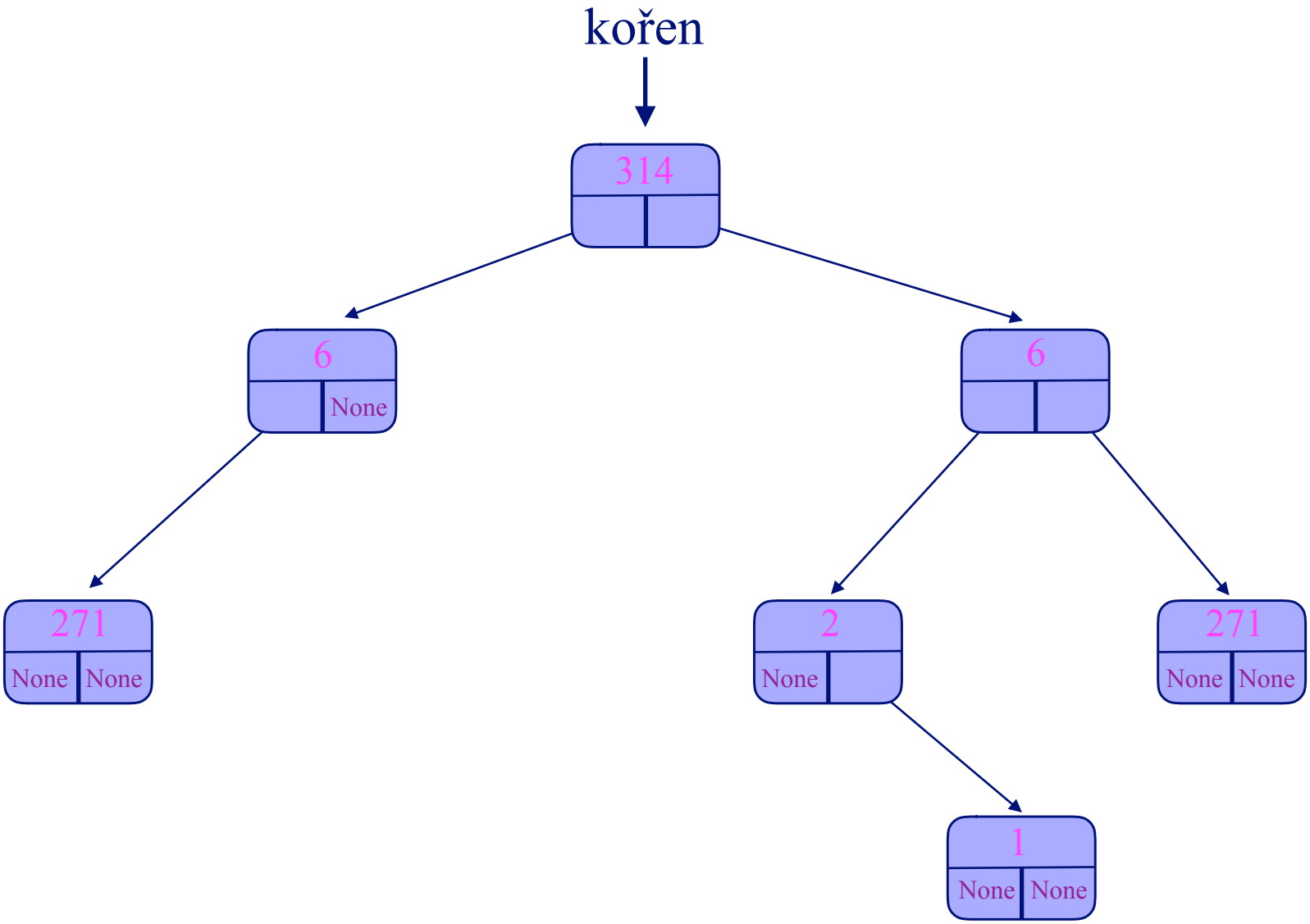
Rekurzivní definice



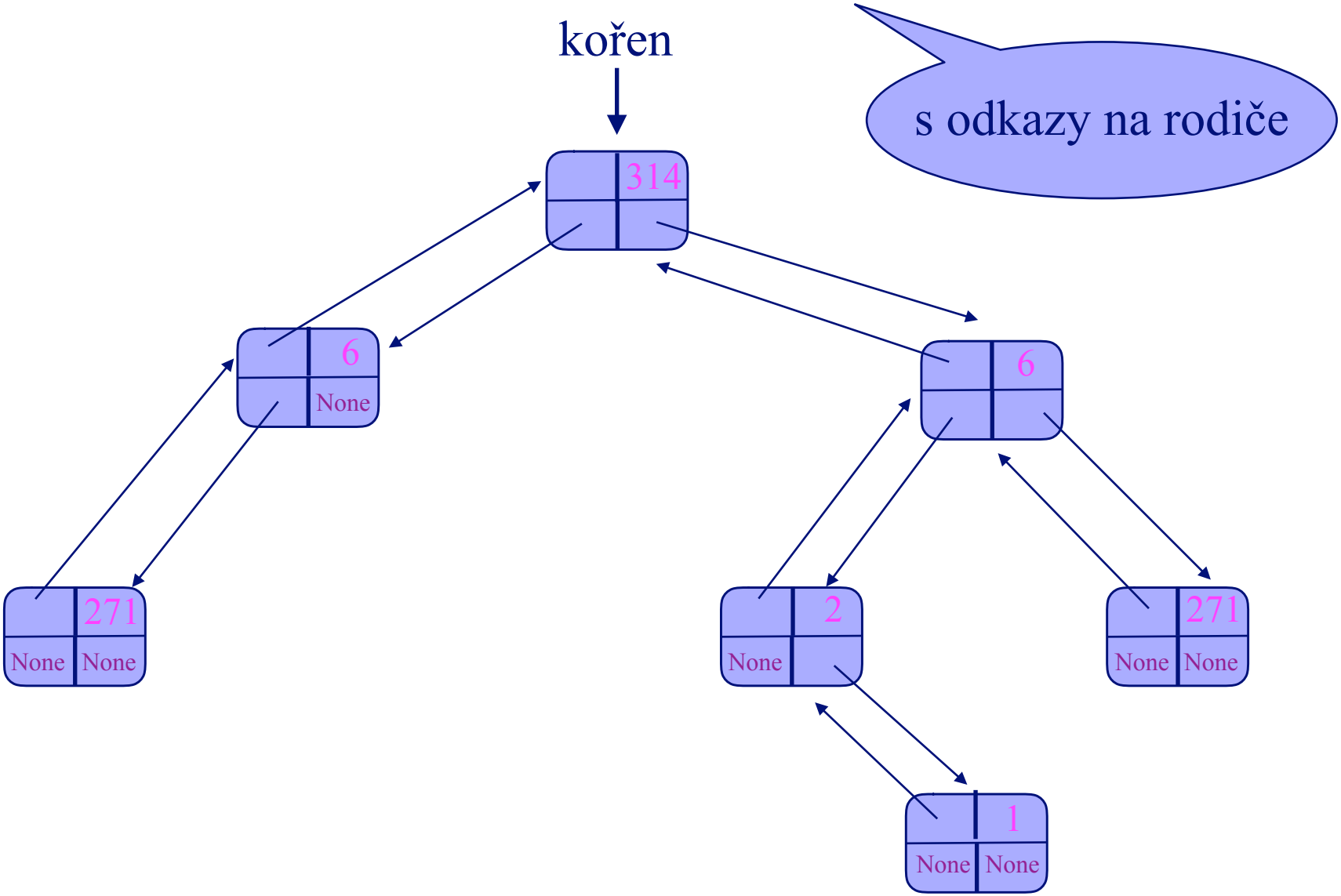
Reprezentace binárního stromu

```
class VrcholBinStromu:
  """třída pro reprezentaci vrcholu
     binárního stromu"""
  def __init__(this,
               x = None,
               levy = None,
               pravy = None):
    this.info  = x      # data
    this.levy  = levy   # levé dítě
    this.pravy = pravy  # pravé dítě
```

Reprezentace binárního stromu



Alternativní reprezentace



Průchod binárním stromem

Navštívíme postupně každý vrchol stromu

- můžeme v něm provést zadanou akci

Preorder

- zpracuj kořen stromu
- rekurzivně projdi levý podstrom
- rekurzivně projdi pravý podstrom

Inorder

- rekurzivně projdi levý podstrom
- zpracuj kořen stromu
- rekurzivně projdi pravý podstrom

Postorder

- rekurzivně projdi levý podstrom
- rekurzivně projdi pravý podstrom
- zpracuj kořen stromu

Preorder – implementace

```
def preorder(koren):
    """vypíše vrcholy stromu se zadaným
       kořenem v pořadí preorder"""
    if kořen != None:
      print(koren.info)  # zpracuj kořen
      preorder(koren.levy)
      preorder(koren.pravy)
```

Časová složitost

- každou hranou projedene v každém směru 1x
- čas O(n), n = počet vrcholů = počet hran + 1

Inorder – implementace

```
def inorder(koren):
    """vypíše vrcholy stromu se zadaným
       kořenem v pořadí inorder"""
    if kořen != None:
      inorder(koren.levy)
      print(koren.info)  # zpracuj kořen
      inorder(koren.pravy)
```

Čas $O(n)$

Postorder – implementace

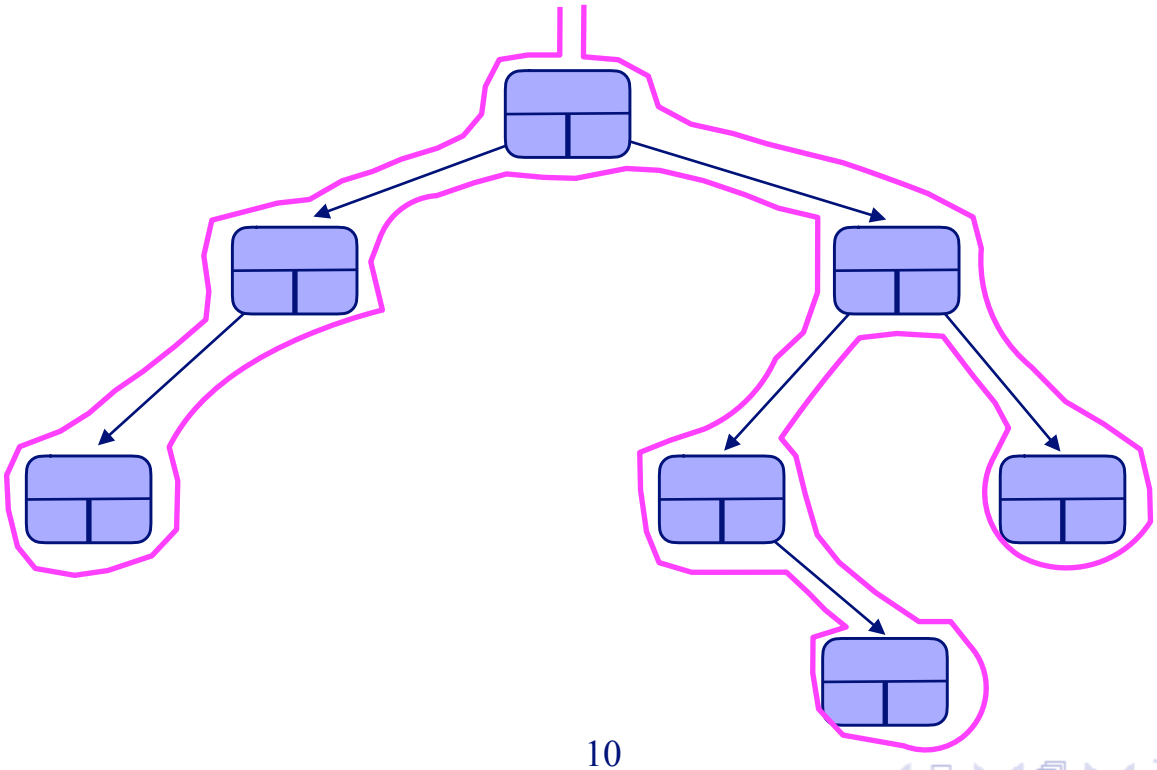
```
def postorder(koren):
    """vypíše vrcholy stromu se zadaným
       kořenem v pořadí postorder"""
    if kořen != None:
      postorder(koren.levy)
      postorder(koren.pravy)
      print(koren.info)  # zpracuj kořen
```

Čas $O(n)$

Průchod do hloubky

Průchod do hloubky (depth-first search, DFS)

- navštiv kořen
- rekurzivně projdi podstromy
- preorder, inorder a postorder – speciální případy



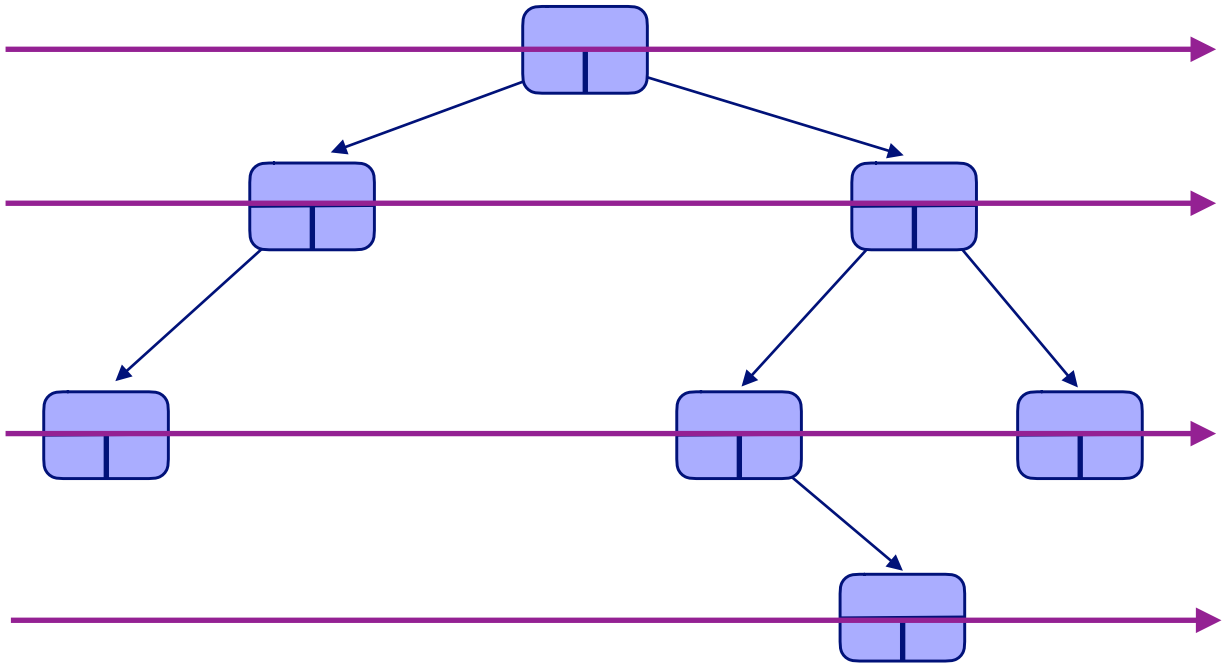
Průchod do hloubky bez rekurze

```
def dfs(koren):
    z = Zasobnik()
    z.push(koren)
    while not z.empty():
        vrchol = z.pop()
        if vrchol != None:
          print(vrchol.info)
          z.push(vrchol.pravy)
          z.push(vrchol.levy)
```

Průchod do šířky

Průchod do šířky (breadth-first search, BFS)

- místo zásobníku použijeme frontu
- vrcholy navštěvuje po hladinách
- “algoritmus vlny”



Průchod do šířky – implementace

```
def bfs(koren):
    f = Fronta()
    f.enqueue(koren)
    while not f.empty():
        vrchol = f.dequeue()
        if vrchol != None:
            print(vrchol.info)
            f.enqueue(vrchol.levy)
            f.enqueue(vrchol.pravy)
```

✎ Problémy: Binární stromy

① V jazyce Python navrhněte funkci, která obdrží binární strom a spočítá jeho

- výšku
- průměrnou výšku

Přitom výška (průměrná výška) stromu je definována jeko maximání (průměrná) délka cesty z kořene do listu.

② Navrhněte efektivní algoritmus, který zjistí, zdali je zadaný binární strom symetrický dle svislé osy, procházející jeho kořenem.

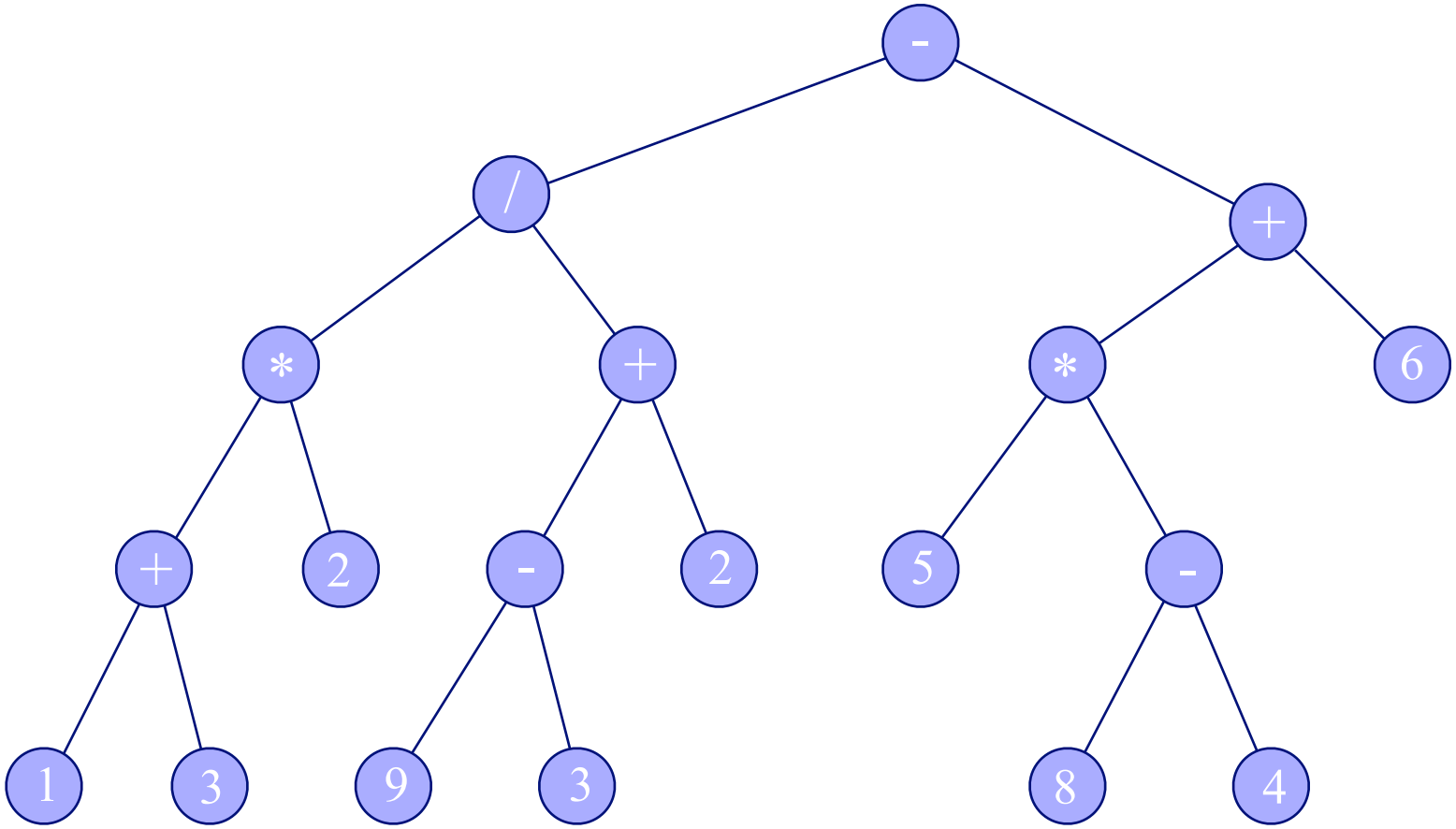
Aritmetické výrazy

Každý aritmetický výraz lze reprezentovat

binárním stromem

- vnitřní vrcholy – operátory
- listy – operandy
- závorky explicitně ve stromě nejsou
- implicitně jsou reprezentovány podstromy

Příklad: strom arit. výrazu



(((1+3)*2)/((9-3)+2))-((5*(8-4))+6)

Strom arit. výrazu – konstrukce

Vstup: aritmetický výraz v infixové notaci,
plně uzávorkovaný

① Rekurzivně, shora dolů

if výraz je tvořen operandem `o` **:**

```
    return VrcholBinStromu(o)
```

else:

vyhledej operátor `op`, který by byl v kořeni stromu

- plně uzávorkovaný výraz: operátor mimo závorky
- rekurzivně sestroj stromy `l`, `p` pro podvýrazy vlevo a vpravo
- `return VrcholBinStromu(op,l,p)`

důkaz: strom rekurze

Čas $\Theta(n^2)$

- n – délka výrazu (počet operátorů, operandů a závorek)
- na vyhledání operátoru v kořeni je třeba čas $\Theta(n)$

Strom rekurze

Vrcholy

- podúlohy (rekurzivně) volané během (rekurzivního) algoritmu

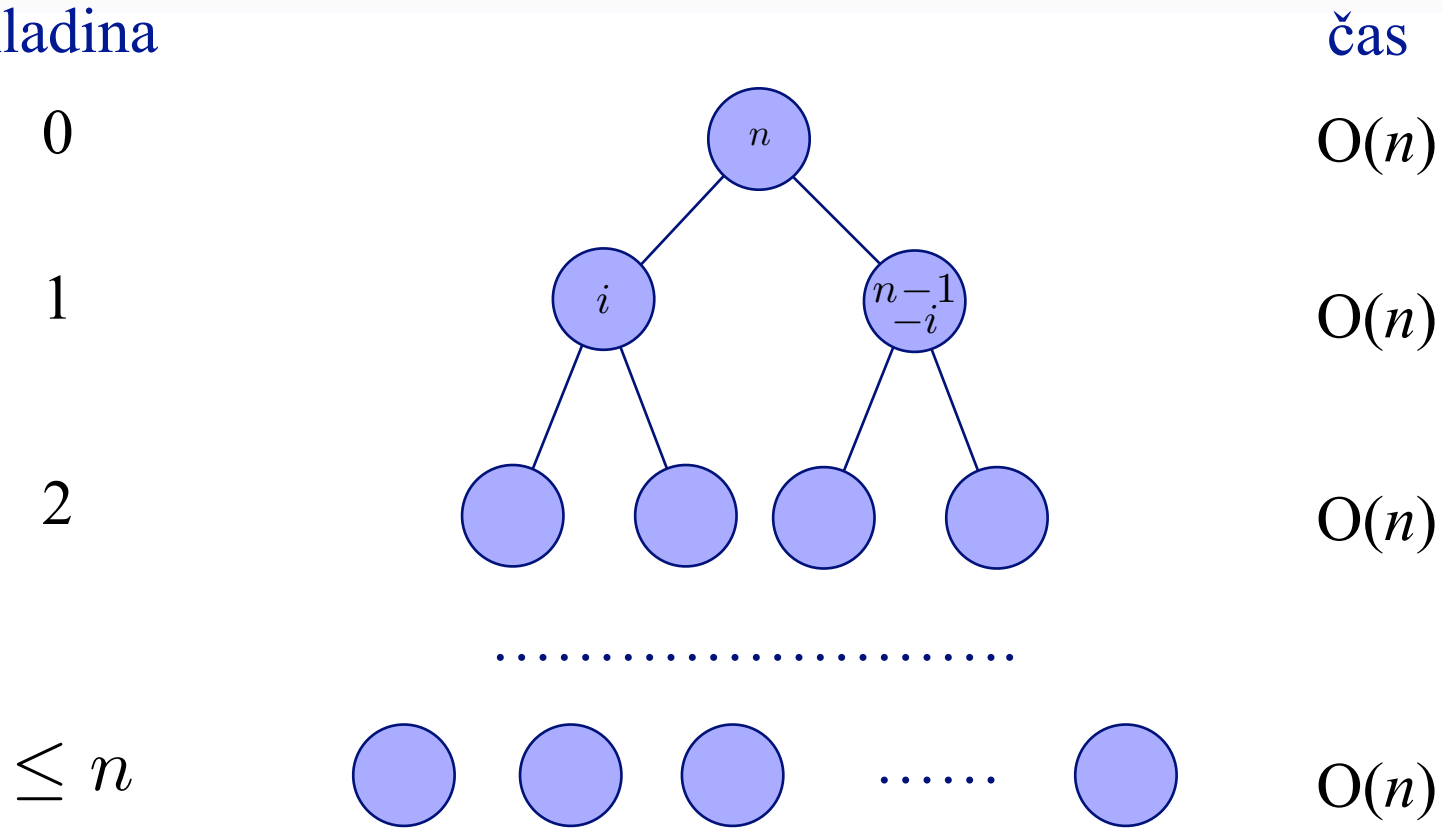
Kořen

- původní úloha

Děti každého rodiče

- představují podúlohy, (rekurzivně) volané v rodičovské úloze

Strom rekurze



Časová složitost

- $T(n) = n \cdot O(n) = O(n^2)$

✎ **Problém**:

Platí $T(n) = \Omega(n^2)$?

Strom arit. výrazu – konstrukce

② Zdola nahoru, využijeme zásobník `z`

Další prvek `p` na vstupu je

- operand : `v = VrcholBinStromu(p), z.push(v)`
- operátor : `z.push(p)`
- `)` : `pravý,op,levý = z.pop(), z.pop(),z.pop()`
- `v = VrcholBinStromu(op,levý,pravý)`
- `z.push(v)`

Na konci obsahuje `z` kořen stromu výrazu

Čas $\Theta(n)$

- n – délka výrazu (počet operátorů, operandů a závorek)

✎ **Problém**: Co když výraz není plně uzávorkovaný?

Aritmetické výrazy – notace

Notace

- infixová: operátor mezi operandy, (1+2)*3
- postfixová: operátor za operandy, 1 2 + 3 *
- prefixová: operátor před operandy, * + 1 2 3

Souvisí s průchodem stromem aritmetického výrazu

- průchod binárním stromem
- metodami inorder, postorder a preorder
- zpracování vrcholu = výpis hodnoty v něm uložené

Aritmetické výrazy – notace

Průchod stromem aritmetického výrazu

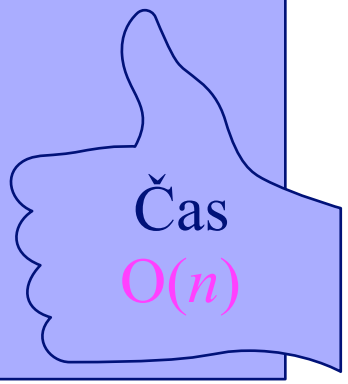
- preorder ⇒ výraz v prefixové notaci
- postorder ⇒ výraz v postfixové notaci
- inorder ⇒ výraz v infixové notaci, ale bez závorek !
  - » lze snadno napravit

Výpis výrazu v infixové notaci

- průchod stromem výrazu metodou inorder
- před rekurzivním voláním na levý / pravý podstrom
- stačí vytisknout ' (' , po návratu ') '

Aritmetické výrazy – vyhodnocení

```
def vyhodnot(koren):
    """vyhodnocení aritmetického výrazu
       reprezentovaného stromem s korenem"""
    if koren.levy = None:
        return koren.info # operand v listu
    else:
        l = vyhodnot(koren.levy)
        p = vyhodnot(koren.pravy)
        if koren.info = '+':
            return l + p
        elif koren.info = '-':
            return l - p
        elif koren.info = '*':
            return l * p
        elif koren.info = '/':
            return l / p
```



Vyhodnocení výrazu – postfix

Vyhodnocení aritmetického výrazu v postfixové notaci

- jeden průchod zleva doprava
- zásobník `z`

Na vstupu je

- operand `o`: `z`.push(`o`)
- operátor `op`:
  - » `pravý, levý = z.pop(), z.pop()`
  - » aplikuj `op` na `levý` a `pravý` operand
  - » výsledek ulož do zásobníku

Po zpracování celého výrazu

- `z` obsahuje výslednou hodnotu výrazu

Čas $O(n)$

- n – délka výrazu

konstrukce stromu z postfix. výrazu: analogicky !

Vyhodnocení výrazu – prefix

Vyhodnocení aritmetického výrazu v prefixové notaci

- popíšeme 3 metody

① Zprava doleva, zásobník

- podobně jako postfix

Rozdíl

- na vrcholu zásobníku je vždy `levý` operand
- pod ním je `pravý`
- tj. `levý, pravý = z.pop(), z.pop()`

Čas $O(n)$

- n – délka výrazu

Vyhodnocení výrazu – prefix

② Zleva doprava, zásobník `z`

Na vstupu je operand nebo operátor `o`

- `z.push(o)`

konstrukce stromu z prefix. výrazu: analogicky !

`while` na vrcholu zásobníku jsou dvě čísla **:**

- `pravý, levý, op = z.pop(), z.pop(), z.pop()`
- aplikací `op` na `levý` a `pravý` operand získáme `vysledek`
- `z.push(vysledek)`

Po zpracování celého výrazu (a celého zásobníku)

- `z` obsahuje výslednou hodnotu výrazu

Čas $O(n)$

- n – délka výrazu

Vyhodnocení výrazu – prefix

③ Zleva doprava, rekurzivně

konstrukce stromu z prefix. výrazu: analogicky !

Na vstupu je číslo `c`

- **`return`** `c`

Na vstupu je operátor `op`

- rekurzivním voláním na dosud nezpracovanou část vstupu obdržíme `levý` operand
- dalším rekurzivním voláním na dosud nezpracovanou část vstupu obdržíme `pravý` operand
- aplikací `op` na `levý` a `pravý` operand získáme `vysledek`
- **`return`** `vysledek`

Čas $O(n)$

- n – délka výrazu

Vyhodnocení výrazu – infix

① Rekurzivně

Imitace konstrukce / vyhodnocení stromu arit. výrazu

`if` výraz je tvořen operandem `o` **`: return`** `o`

`else:` vyhledej operátor `op`, který by byl v kořeni stromu

- plně uzávorkovaný výraz: operátor mimo závorky
- neúplně u.v.: operátor nejnižší priority mimo závorky
  co nejvíce vpravo
- rekurzivně vyhodnoť podvýrazy vlevo a vpravo
- na takto získané hodnoty operandů aplikuj `op`
- **`return`** `výsledek`

Čas $\Theta(n^2)$

- na vyhledání operátoru v kořeni je třeba čas $\Theta(n)$

Vyhodnocení výrazu – infix

② Převodem do postfixové notace

Převod výrazu z infixové do postfixové notace

- v lineárním čase, viz dále

Vyhodnocení výrazu v postfixové notaci

Lze realizovat souběžně

- vytvářený postfixový zápis se průběžně vyhodnocuje

Čas $O(n)$

Převod z infixové do postfixové notace

Zásobník `z` operátorů a závorek

- pořadí operandů je v obou notacích stejné
- operátory je třeba při převodu "zdržet" v zásobníku

Další prvek `p` na vstupu je

- operand : `print(p)`
- `(` : `z.push(p)`
- `)` : `z.pop(op), while op != '(': print(op)`
- operátor : nesmí "předběhnout" operátory ≥ priority
  - » `z.pop(op), print(op)` dokud nenarazíme na dno, na `(`, nebo na operátor < priority než `p`
  - » `z.push(p)`

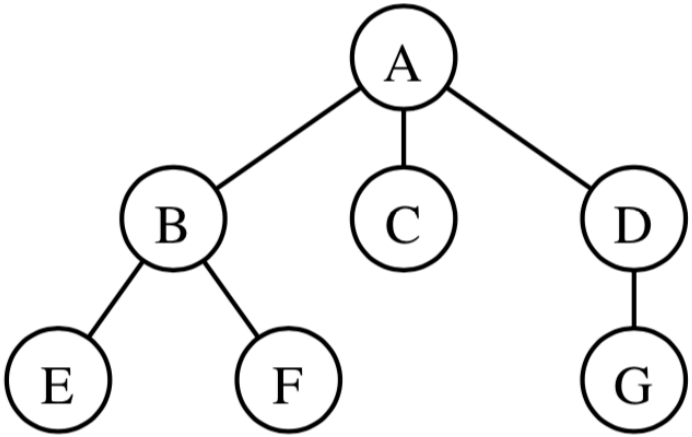
Na konci odešleme operátory ze zásobníku na výstup

Obecné (k-ární) stromy

k-ární strom

- uspořádaný kořenový strom
- každý vrchol má nejvýše k dětí

☀ **Příklad**: ternární strom



Reprezentace k-árního stromu

① Pole odkazů na děti

- každý vrchol obsahuje pole délky k
- vhodné, pokud k známe a je malé

② Spojový seznam dětí

- každý vrchol obsahuje odkaz
  - » na nejlevější dítě
  - » na sourozence (bezprostředně vpravo)
  - » pokud neexistují, odkaz nabývá hodnoty `None`
- fakticky jde tedy o reprezentaci binárním stromem

Reprezentace obecného stromu

```
class VrcholStromu:
  """třída pro reprezentaci vrcholu
     obecného stromu"""
  def __init__(this,
               x = None,
               dite = None,
               sou = None):
    this.info  = x     # data
    this.dite  = dite  # nejlev. dítě
    this.sou   = sou   # pravý
                       # sourozenec
```

Příklad: obecný strom