Programování 2

8. cvičení, 16-4-2024

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Minulý týden jsme neměli cvičení**, byl jsem bohužel nemocný. Nakonec jsem vám nechal jenom dokončit domácí úkoly a pokračovat budeme až dnes.
3. **Domácí úkoly**: Měli jste na 3 úlohy o spojovaných seznamech 2 týdny. Zvládli jste to dobře, přišlo mnoho dobrých řešení.
4. **Zápočtový program**: *Do konce dubna poprosím o návrhy na zápočtové programy*. S několika z vás jsme se už domluvili. Prosím neodkládejte to.

Dnešní program:

- Kvíz
- Rekurze
- Binární stromy: základy

Na zahřátí

```
import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
```

```
Namespaces are one honking great idea -- let's do more of those!
```

Navštivte https://testdriven.io/blog/clean-code-python/ pro množství rad o tom, jak psát v Pythonu dobrý kód.

Co dělá tento kód

```
(lambda : 50)()
```

`None`

V Pythonu je `None` unikátní objekt typu `NoneType`, který signalizuje

- hodnotu `null` (pro pointry, které nikam neukazují) nebo
- chybějící hodnotu (např. pro návratovou hodnotu funkce, která nic nevrací).

Konvence je nepoužívat pro testování na `None` relační operátory, ale operátor `is` (`None` je unikátní objekt) nebo konverzi objektu na `bool` - cokoli co je `None` se konvertuje na `FALSE`.

```
if x == None:        # Špatně (ale funguje)
if x != None

if x is None:        # Správně
if x is not None:

if x:                # Správně
if not x:
```

Poslední způsob psaní je nejúspornější a nejčitelnější a nejspíš se u něj vyhnete potřebě závorkování.

Mini-tutoriál: konstanty

Konstanty pot\řebujeme, když máme v kódu fixní číselné hodnoty nebo řetězce.

Je dobrou praxí definovat konstantní čísla, řetězce nebo jiné objekty na jediném místě,

- abychom je mohli v případě potřeby lehce změnit
- aby byl kód srozumitelnější.

V Pythonu máme některé konstanty v modulu `math`, například `math.pi`, `math.e` a pod. Pro metody strojového učení nebo fyzikální simulace často potřebujeme rozsáhlé seznamy konstant.

1. Tradiční způsob

Podle příručky Pythonského stylu *Python Style Guide* konstanty označujeme identifikátory z velkých písmen a umísťujeme je na začátek modulu:

```
N_AVOGADRO = 6.022e23
K_GRAVITY = 6.673e-11
H_PLANCK = 6.626e-34

def atoms_from_moles(moles: float) -> float:
    return moles * N_AVOGADRO
...
```

- Velká písmena zabezpečují lehkou identifikaci objektu jako konstanty a tedy i jako určitou ochrana před náhodným přepsáním.
- Hodnotu konstanty je ale možné přepsat a toto riziko je větší díky tomu, že konstanty umísřujeme do globálního prostoru jmen. Například můžeme omylem předefinovat konstantu konstantou stejného jména, ale v jiných jednotkách, z jiného modulu.
- Umístění na začátku modulu sice umožňuje lehce identifikovat konstantní hodnoty v kódu, ale může se nám tam nahromadit spousta čisel různého významu. Někdy je lepší definovat konstanty tam, kde je používáme.

2. Třídy konstant

Pokud spravujeme různé soubory konstant, můžeme použít jiný přístup - definovat konstanty jako atributy tříd, které nelze měnit.

Metoda 1: Dataclass

Dataclass (modul `dataclasses`) představují efektivní způsob tvorby tříd, jejichž primárním účelem je uchovávat data. Pro účely této části si jenom řekneme, že dataclass můžeme vytvořit jako zmrzlý - frozen - tedy jako konstantní třídu, jejíž atributy nelze měnit.

```
# Const class:
# Container for numbers protected against change
# Variant 1: frozen dataclass

from dataclasses import dataclass


@dataclass(frozen=True)
class ConstClass:
    PI: float = 3.1416
    E: float = 2.7183


ConstClass.EULER_GAMMA = 0.57722 # Can add attributes to class
ConstClass.E = 2.71828 # Can modify class attributes

const = ConstClass()   # Instances are frozen:
                       # no adding or modifying
print(const.PI, const.E)
```

```
const.PI = 42 # dataclasses.FrozenInstanceError:
              # cannot assign to field 'PI'

-----
3.1416 2.71828
Traceback (most recent call last):
...
dataclasses.FrozenInstanceError: cannot assign to field 'PI'
```

- Konstanty jsou atributy třídy `Const` a není nutné, aby měly identifikátory z velkých písmen. To může být graficky méně rušivé.
- Analyzátor kódu i interpret brání změně atributů instancí zmrzlé třídy, i když samotnou třídu modifikovat lze.

Metoda 2 `__setattr__`

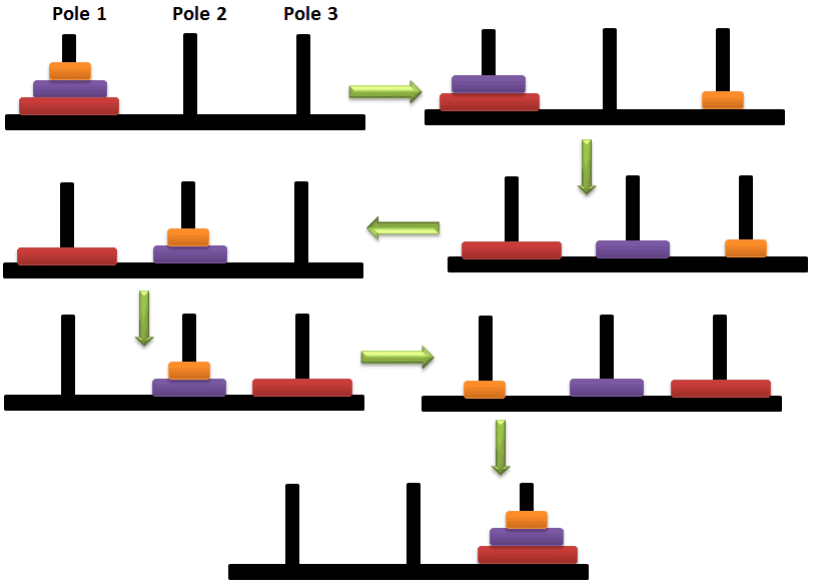
Můžeme také použít běžnou třídu, ale upravit metodu, kterou se do třídy přidávají atributy - tedy metodu `__setattr__`:

```
class Const:
    def __setattr__(self, name, value):
        if hasattr(self, name):
            raise TypeError('cannot mutate constant')
        super().__setattr__(name, value)

chem = Const()    # potřebujeme instanci
chem.avogadro = 6.022e23

physics = Const()
physics.G = 6.673e-11
physics.h = 6.626e-34

chem.avogadro = 7 # TypeError: cannot mutate constant


Traceback (most recent call last):
...
TypeError: cannot mutate constant
```

Rekurze

O rekurzi jste dost mluvili na přednášce a teoretickém cvičení, takže pojďme radši něco naprogramovat.

Hanojské věže



Máme 3 kolíčky a sadu kroužků různých velikostí.

Kroužky jsou na začátku na jediném kolíčku uspořádané podle velikosti, největší vespod.

Úloha je přesunout kroužky na jiný kolíček tak, že v každém okamžiku budou kolečka na všech kolících uspořádaná podle velikosti - tedy nesmíme větší kolečko uložit na menší.

Kde tady nalézt rekurzi? Použijeme princip podobný matematické indukci:

- Úlohu umíme vyřešit pro 1 kroužek.
- Pokud bychom znali řešení pro n-1 kroužků, uměli bychom úlohu vyřešit pro n kroužků? Uměli:
  - Přenést n-1 kroužků na vedlejší sloupek, můžeme si pomoct cílovým sloupkem.
  - Přenést spodní kroužek na cílový sloupek
  - Přenést n-1 kroužků z vedlejšího sloupku na cílový sloupek, můžeme si pomoct výchozím sloupkem.

(Kód v `code/Ex8/hanoi.py`)

```
def move(n: int, start: str, end: str, via: str) -> None:
    if n == 1:
        print(f"Moved {start} to {end}")
    else:
        move(n-1, start, via, end)
        move(1, start, end, via)
        move(n-1, via, end, start)
    return


if __name__ == '__main__':
    move(5, "A", "B", "C")
```

Všechny možné rozklady přirozeného čísla

1 -> (1)

2 -> (2), (1,1)

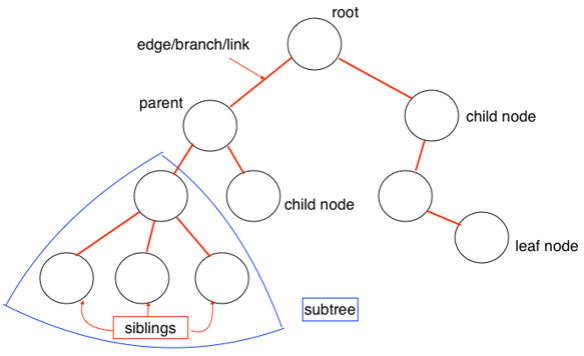
3 -> (3), (2, 1), (1, 1, 1)

4 -> (4), (3, 1), (2, 2), (2, 1, 1), (1, 1, 1, 1)

Podobný styl rekurze: indukce

- Máme řesení pro několik malých čísel 1, 2, ...
- Z řešení pro n umíme zkonstruovat rešení pro n+1.
- Musíme dát pozor, abychom opakovaně negenerovali stejné rozklady. Je lepší to udělat pomocí organizace výpočtu než to kontrolovat dodatečně pomocí `set` a p.

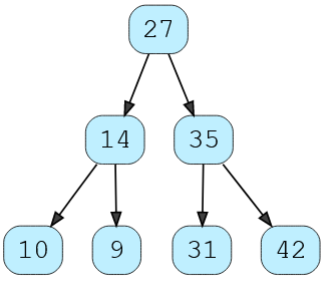
Stromy



Obecný strom: jeden kořen, mnoho větví, neomezené větvení.

Binární stromy

Každý uzel má nejvíc dvě větve:



Počítání uzlů:

- Kořen je úroveň 0, pak na úrovni K máme maximálně 2^K uzlů
- plný graf s hloubkou D bude mít $2^D - 1$ uzlů. Hloubka je počet úrovní grafu.
- Graf s N vrcholy má nejméně $log_2(N + 1)$ úrovní.
- Binární strom s L listy má nejméně $log_2(L) + 1$ úrovní.

Pro binární stromy nám bude stačit implementovat uzel, nepotřebujeme samostatnou třídu binárního stromu.

```
class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

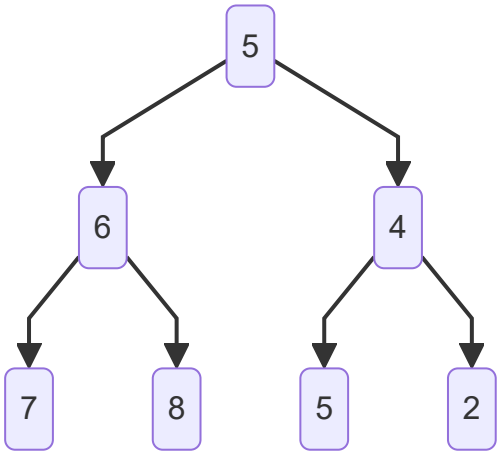
Je to dizajnová volba: u lineárních seznamů vlastnil objekt seznamu ukazatel na první uzel seznamu, ale slouží také jako kontejner na metody seznamu. Ty však můžeme implementovat také jako metody uzlu.,

Takovýto objekt nám stačí na vytvoření stromu.

Jak s takovýmto objektem vytvářet binární stromy a pracovat s nimi?

Vytváření stromů je lehké díky tomu, že v konstruktoru můžeme zadat dceřinné uzly:

```
    tree = Node(
        5,
        Node(
            6,
            Node(7),
            Node(8)
        ),
        Node(
            4,
            Node(5),
            Node(2)
        )
    )
```

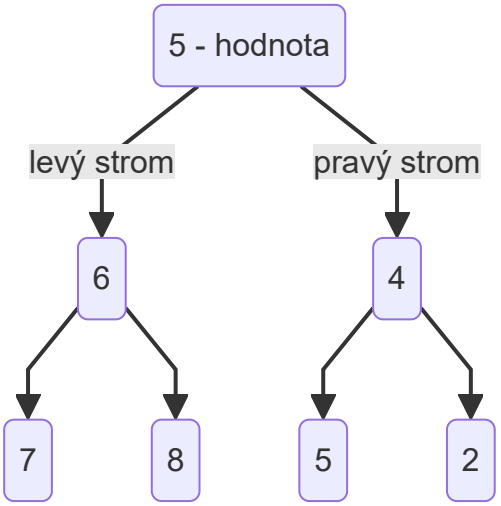


Takže umíme vytvořit strom, ale také potřebujeme vypsat hodnoty ze stromu nebo dokonce strom zobrazit.

Rekurze

Mnoho věcí umíme lehce definovat, pokud si uvědomíme rekurzivní podstatu binárního stromu:

U každého uzlu máme hodnotu, levý strom a pravý strom:



Takže například lehce spočteme hodnoty v seznamu:

```
    def count(self):
        count = 1                       # kořen
        if self.left is not None:
            count += self.left.count()  # levý strom
        if self.right is not None:
            count += self.right.count()  # pravý strom
        return count

7
```

Tady jsme jenom počítali hodnoty, takže výsledek nezávisel od toho, jak jsme stromem procházeli.

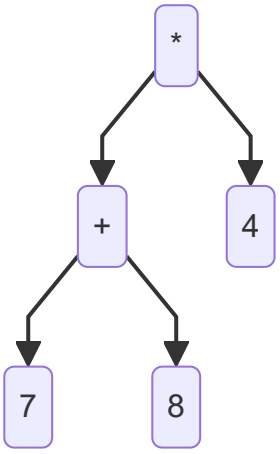
Stejně můžeme chtít vypsat hodnoty ve všech uzlech stromu. V takovém případě ale musíme definovat, jak budeme stromem procházet:

```
    def to_list_preorder(self):
        flat_list = []
        flat_list.append(self.data)
        if self.left is not None:
            flat_list.extend(self.left.to_list_preorder())
        if self.right is not None:
            flat_list.extend(self.right.to_list_preorder())
        return flat_list

    def to_list_inorder(self):
        flat_list = []
        if self.left is not None:
            flat_list.extend(self.left.to_list_inorder())
        flat_list.append(self.data)
        if self.right is not None:
            flat_list.extend(self.right.to_list_inorder())
        return flat_list
```

```
    def to_list_postorder(self):
        flat_list = []
        if self.left is not None:
            flat_list.extend(self.left.to_list_postorder())
        if self.right is not None:
            flat_list.extend(self.right.to_list_postorder())
        flat_list.append(self.data)
        return flat_list

[5, 6, 7, 8, 4, 5, 2]
[6, 7, 8, 5, 4, 5, 2]
[6, 7, 8, 4, 5, 2, 5]
```

Pořadí	**Použití**
Pre-order	Výpis od kořene k listům, kopírování, výrazy s prefixovou notací
In-order	Ve vyhledávacích stromech dává inorder hodnoty v uzlech v neklesajícím pořadí.
Post-order	Vymazání stromu

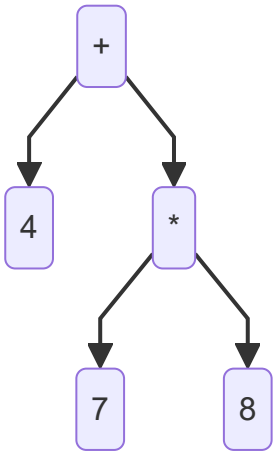
Aritmetické výrazy:



Výraz ve tvaru binárního stromu je jednoznačný a nepotřebuje závorky. Podle toho, jak výraz ze stromu přečteme, dostáváme různé typy notace:

- in-order --> infixová notace (běžná notace, potřebuje závorky) (7+8) x 4
- Pre-order --> prefixová notace (Polská logika, nepotřebuje závorky) * 4 +7 8
- Post-order --> postfixová notace (RPL, nepotřebuje závorky) 7 8 + 4 *

Pro binární operátory je binární graf jednoznačným zápisem výrazu a nepotřebuje závorky. Pro výraz 7 + 8 * 4 máme úplně jiný strom než pro (7 + 8) * 4:



Úkol Jak vypočíst hodnotu takovéhoto stromu?

Uměli bychom strom nějak zobrazit? Můžeme třeba zkusit posouvat jednotlivé úrovně stromu a použít in-order průchod stromem:

```
    def to_string(self, level = 0):
        strings = []
        if self.left is not None:
            strings.append(self.left.to_string(level + 1))
        strings.append(' ' * 4 * level + '-> ' + str(self.data))
        if self.right is not None:
            strings.append(self.right.to_string(level + 1))
        return "\n".join(strings)

    def __str__(self):
        return self.to_string()

        -> 7
    -> 6
        -> 8
-> 5
        -> 5
    -> 4
        -> 2
```

Výsledek sice neoslní, ale jakž-takž vyhoví.

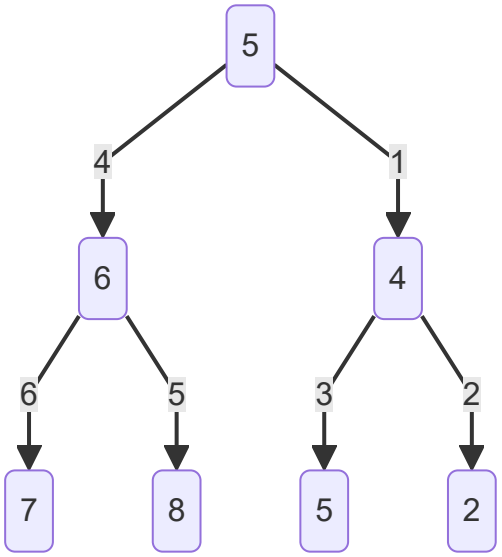
`to_string` musí být oddělená od `__str__`, protože potřebujeme jinou signaturu.

Nerekurzivní průchod stromem

Rekurze je sice elegantní způsob, jak implementovat metody procházení binárními stromy, ale víme, že bychom brzo narazili na meze hloubky rekurze. Proto je zajímavé zkusit implementovat nerekurzivní verze těchto metod.

1. Použít zásobník: FIFO pro prohledávání do hloubky (depth-first):
    - Jako zásobník by nám stačil obyčejný seznam (list), tady používáme collections.deque
    - cestou tiskneme stav zásobníku, abychom viděli, co se děje

```
from collections import deque
...

    def to_list_depth_first(self):
        stack = deque()
        df_list = []
        stack.append(self)
        print(stack)
        while len(stack)>0:
            node = stack.pop()
            df_list.append(node.data)
            if node.left:
                stack.append(node.left)
            if node.right:
                stack.append(node.right)
            print(stack)
        return df_list

deque([5])
deque([6, 4])
deque([6, 5, 2])
deque([6, 5])
deque([6])
deque([7, 8])
deque([7])
deque([])
[5, 4, 2, 5, 6, 8, 7]
```

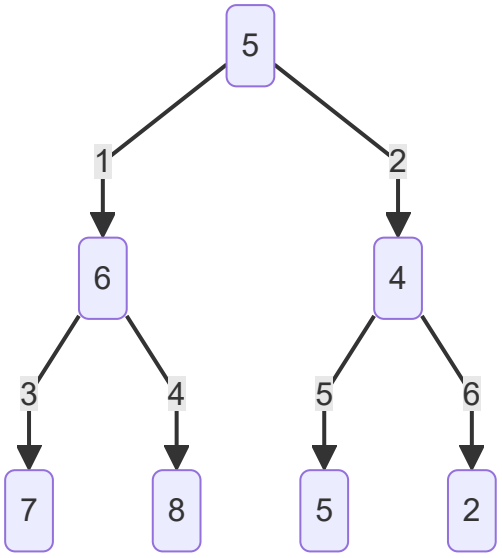


Pořadí dáme do pořádku přehozením levé a pravé větve.

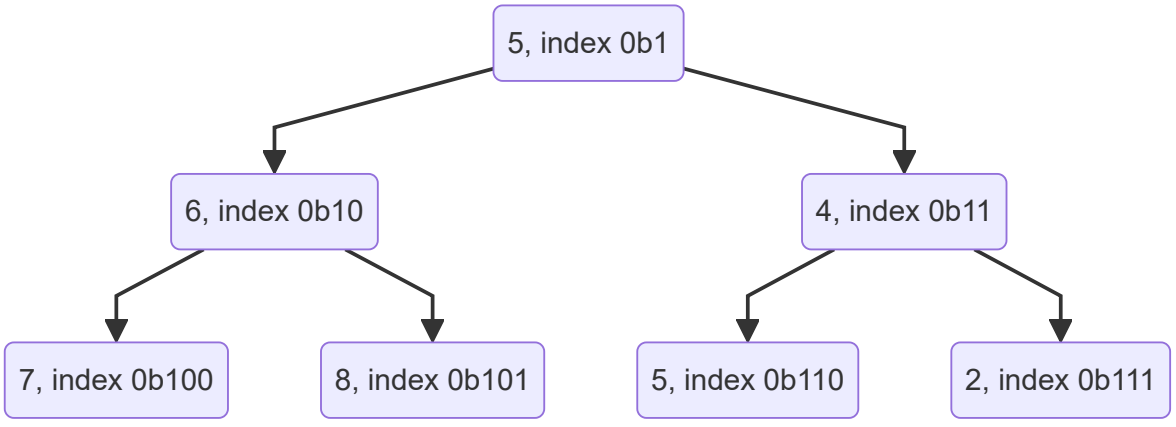
2. Použít frontu LIFO pro prohledávání do šířky (breadth-first)

```
    def to_list_breadth_first(self):
        queue = deque()
        bf_list = []
        queue.append(self)
        print(queue)
        while len(queue)>0:
            node = queue.popleft()
```

```
            bf_list.append(node.data)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
            print(queue)
        return bf_list

deque([5])
deque([6, 4])
deque([4, 7, 8])
deque([7, 8, 5, 2])
deque([8, 5, 2])
deque([5, 2])
deque([2])
deque([])
[5, 6, 4, 7, 8, 5, 2]
```



Tato poslední metoda je zvlášť důležitá, protože umožňuje jednoduché mapování binárního stromu do pole (už jsme viděli u hromady, že je praktické nepoužít nultý prvek pole).



- Potomci uzlu na indexu k jsou 2k a 2k+1
- Předek uzlu na indexu k je k // 2
- Uzel k je levý potomek svého předka, pokud k % 2 == 0, jinak je to pravý potomek.

Úkol Zkuste popřemýšlet, jak byste ze seznamu hodnot, který poskytuje metoda `to_list_breadth_first` zrekonstruovali původní strom.

(Kód v `code/Ex9/list_tree.py`)

```
...
def tree_from_list(values: list[int]) -> Node:
    values = [0] + values
    queue = deque()
    index = 1
    tree = Node(values[index])
    index += 1
    queue.append(tree)
    while index < len(values):
        print(queue)
        node = queue.popleft()
        node.left = Node(values[index])
        print(index)
        index += 1
        queue.append(node.left)
        if index == len(values):
            node.right = None
            break
        node.right = Node(values[index])
        print(index)
        index += 1
        queue.append(node.right)
    return tree


def main() -> None:
    tree = Node(
        5,
        Node(
            6,
            Node(7),
            Node(8)
        ),
        Node(
            4,
            Node(5),
            Node(2)
        )
    )

    print(tree.to_string())
    values = tree.to_list_breadth_first()
    tree2 = construct_from_list(values)
    print(tree2.to_string())


if __name__ == '__main__':
    main()
```

Tato metoda funguje jenom pro úplné binární stromy, tedy v případě, že chybějí jenom několik listů na levé straně poslední vrstvy. Jinak bychom museli dodat do seznamu doplňující informaci - buď "uzávorkování" potomků každého uzlu, anebo u každého uzlu uvést počet potomků.

Úkol Napište metodu `Node.copy(self)`, která vrátí kopii stávajícího stromu.

Úkol Napište metodu `Node.delete(self)`, která vymaže strom s kořenem v `Node`.

Nerekurzivní inorder a postorder průchody binárním stromem

Metoda pro nerekurzivní průchod stromem s využitím zásobníku, kterou jsme ukazovali výše, je pre-order metodou, protože vypisuje hodnotu uzlu před hodnotami uzlů v podstromech.

```
    def to_list_depth_first(self):
        stack = deque()
        df_list = []
        stack.append(self)
        print(stack)
        while len(stack)>0:
            node = stack.pop()
            df_list.append(node.data)
            if node.right:
                stack.append(node.right)
            if node.left:
                stack.append(node.left)
            print(stack)
        return df_list
```

Je logické se ptát, zdali můžeme implementovat i nerekurzivní inorder a postorder průchody.

Můžeme, i když implementace je mírně odlišná.

Nerekurzivní in-order průchod binárním stromem:

Uložíme do zásobníku nejdříve celý levý podstrom, pak hodnotu, a pak pravý podstrom.

```
    def to_list_df_inorder(self):
        stack = deque()
        df_list = []
        current = self
        while True:
            if current is not None:
                stack.append(current)
                current = current.left
            elif stack:
                current = stack.pop()
                df_list.append(current.data)
                current = current.right
            else:
                break
        return df_list
```

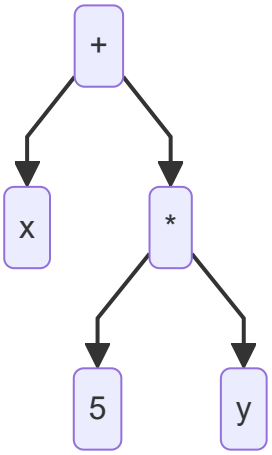
Nerekurzivní post-order průchod binárním stromem

Toto je komplikovanější případ, potřebujeme dva zásobníky, přičemž do druhého si za pomoci prvního ukládáme uzly ve správném pořadí.

```
    def to_list_df_postorder(self):
        s1 = deque()
        s2 = deque()
        df_list = []
        s1.append(self)
        while s1:
            node = s1.pop()
            s2.append(node)
            if node.left:
                s1.append(node.left)
            if node.right:
                s1.append(node.right)
        while s2:
            node = s2.pop()
            df_list.append(node.data)
        return df_list
```

(V našem případě vypadá poslední cyklus poněkud směšně, protože výstupní seznam je prostě stack s2 v obráceném pořadí; z pedagogických důvodů je ale vhodnější odlišit výstupní seznam od s2.)

Operace s výrazy ve tvaru stromů




```
class Expression:
    ...


class Constant(Expression):
    def __init__(self, value):
        self.value = value
```

```
    def __str__(self):
        return str(self.value)

    def eval(self, env):
        return self.value

    def derivative(self, by):
        return Constant(0)


class Variable(Expression):
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def eval(self, env):
        return env[self.name]

    def derivative(self, by):
        if by == self.name:
            return Constant(1)
        else:
            return Constant(0)


class Plus(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return f"({self.left} + {self.right})"

    def eval(self, env):
        return self.left.eval(env) + self.right.eval(env)

    def derivative(self, by):
        return Plus(
            self.left.derivative(by),
            self.right.derivative(by)
        )


class Times(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return f"({self.left} * {self.right})"

    def eval(self, env):
        return self.left.eval(env) * self.right.eval(env)
```

```
    def derivative(self, by):
        return Plus(
            Times(
                self.left.derivative(by),
                self.right
            ),
            Times(
                self.left,
                self.right.derivative(by)
            )
        )


def main():
    vyraz = Plus(
        Variable("x"),
        Times(
            Constant(5),
            Variable("y")
        )
    )
    print(vyraz)
    print(vyraz.eval({"x": 2, "y": 4}))
    print(vyraz.derivative(by="x"))
    print(vyraz.derivative(by="y"))


if __name__ == '__main__':
    main()

-----------
(x + (5 * y))
(1 + ((0 * y) + (5 * 0)))
(0 + ((0 * y) + (5 * 1)))
```

Sice to funguje, ale dostáváme strom, ve kterém je spousta hlušiny:

- přičítání nuly a násobení nulou
- násobení jedničkou

Můžeme si vytvořit čistící proceduru, která stromy rekurzivně vyčistí, a opět postupujeme tak, že určité uzly či struktury ve stromu rekurzivně nahrazujeme jinými uzly či strukturami.

```
class Expression:
    ...


class Constant(Expression):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return str(self.value)
```

```
    def eval(self, env):
        return self.value

    def derivative(self, by):
        return Constant(0)

    def prune(self):
        return self

# Testování konstanty, zdali je či není 0 nebo 1 !!

def is_zero_constant(x):
    return isinstance(x, Constant) and x.value == 0


def is_unit_constant(x):
    return isinstance(x, Constant) and x.value == 1


class Variable(Expression):
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def eval(self, env):
        return env[self.name]

    def derivative(self, by):
        if by == self.name:
            return Constant(1)
        else:
            return Constant(0)

    def prune(self):
        return self


class Plus(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return f"({self.left} + {self.right})"

    def eval(self, env):
        return self.left.eval(env) + self.right.eval(env)

    def derivative(self, by):
        return Plus(
            self.left.derivative(by),
            self.right.derivative(by)
        )
```

```
    def prune(self):
        self.left = self.left.prune()
        self.right = self.right.prune()
        if is_zero_constant(self.left):
            if is_zero_constant(self.right):
                return Constant(0)
            else:
                return self.right
        if is_zero_constant(self.right):
            return self.left
        return self


class Times(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return f"({self.left} * {self.right})"

    def eval(self, env):
        return self.left.eval(env) * self.right.eval(env)

    def derivative(self, by):
        return Plus(
            Times(
                self.left.derivative(by),
                self.right
            ),
            Times(
                self.left,
                self.right.derivative(by)
            )
        )

    def prune(self):
        self.left = self.left.prune()
        self.right = self.right.prune()
        if is_zero_constant(self.left) | is_zero_constant(self.right):
            return Constant(0)
        if is_unit_constant(self.left):
            if is_unit_constant(self.right):
                return Constant(1)
            else:
                return self.right
        if is_unit_constant(self.right):
            return self.left
        return self


def main():
    vyraz = Plus(
        Variable("x"),
        Times(
            Constant(5),
```

```
            Variable("y")
        )
    )
    print(vyraz)
    print(vyraz.derivative(by="x"))
    print(vyraz.derivative(by="x").prune())
    print(vyraz.derivative(by="y"))
    print(vyraz.derivative(by="y").prune())


if __name__ == '__main__':
    main()
-----------
(x + (5 * y))
(1 + ((0 * y) + (5 * 0)))
1
(0 + ((0 * y) + (5 * 1)))
5
```

- Všimněte si post-order procházení stromu při prořezáváni.
- Metodu `prune` definujeme také pro konstanty a proměnné, i když s nimi nedělá nic. Ulehčuje to rekurzivní volání metody.
- Musíme být pozorní při testování, zda je daný uzel/výraz nulová nebo jedničková konstanta. Nestačí operátor rovnosti, musíme nejdřív zjistit, zda se jedná o konstantu a pak otestovat její hodnotu. V principu bychom mohli dvě testovací funkce proměnit v metody třídy `Expression`.

Domácí úkol

Implementujte konstrukci, která ze stromu, kódujícího polynomiální funkci, vytvoří strom, kódující její primitivní funkci (podle některé proměnné).