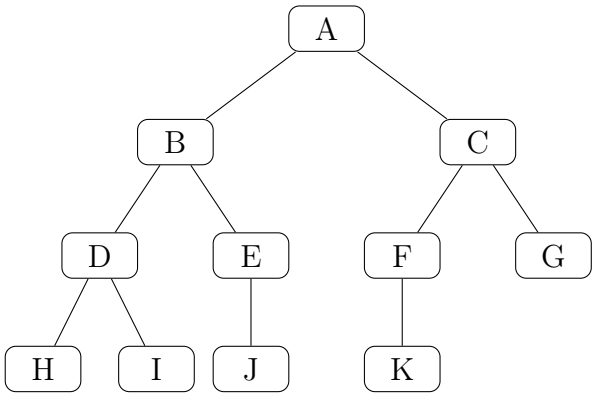
Algorytmy i struktury danych (studia stacjonarne)

Dr Anna Muranova
Semestr zimowy 2025/2026, UWM w Olsztynie

Ćwiczenie 8 (Binarne drzewo poszukiwań) 02.12.2025

Zadanie 6.1. Binarne drzewa – to struktura danych w postaci drzewa, w którym każde węzeł ma co najwyżej dwóch dzieci.



Przygotować implementację drzewa binarnego wg następujących wymagań:

(A) klasa `TreeNode` odpowiedzialna jest za węzeł drzewa, w którym przechowywana jest wartość oraz dzieci.

```
from typing import Any, Callable

class TreeNode:

    value: Any
    left_child: 'TreeNode'
    right_child: 'TreeNode'
```

W klasie należy umieścić następujące metody:

(a) konstruktor `__init(value = None, left_child = None, right_child = None)`

(b) funkcja `__str__` zwraca $str(value)$

(c) funkcja `add_left(child: "TreeNode") -> None`, która doda węzeł przyjęty w argumencie jako lewe dziecko.

(d) funkcja `add_right(child: "TreeNode") -> None`, która doda węzeł przyjęty w argumencie jako prawe dziecko.

(e) metoda rekurencyjna `traverse_in_order(self, visit: Callable[[Any], None])`, która wykona operację poprzecznego przejścia po podrzędnych węzłach: najpierw przechodzimy lewe poddrzewo danego węzła, następnie odwiedzamy węzeł i na koniec przechodzimy prawe poddrzewo.

(f) metoda rekurencyjna `traverse_pre_order(self, visit: Callable[[Any], None])`, która wykona operację wzdłużnego przejścia po podrzędnych węzłach: najpierw odwiedzamy korzeń poddrzewa, a następnie przechodzimy kolejno lewe i prawe poddrzewo.

(g) metoda rekurencyjna `traverse_post_order(self, visit: Callable[[Any], None])`, która wykona operację wstecznego przejścia po podrzędnych węzłach: najpierw przechodzimy lewe poddrzewo, następnie prawe, a dopiero na końcu przetwarzamy węzeł.

(h) metoda rekurencyjna `height(self)`, oblicza wysokość poniżej podanego węzła. Wysokość = 1 + max (wysokości dzieci).

(B) Klasa `BinaryTree` jest odpowiedzialna za przechowywanie całej struktury drzewa, gdzie root wskazuje węzeł będący korzeniem.

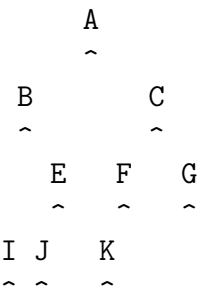
```
class BinaryTree:
    root: TreeNode
```

Zaimplementuj metody

(a) konstruktor `__init(root = None)`

(b) metoda `height(self)`, zwróci wysokość drzewa.

(c) * metoda `show` pokażę drzewo. Nie używaj zewnętrznych bibliotek. Wymysł coś sam. U mnie wyszło tak:



(C) Klasa `BinarySearchTree` pozwala na stworzenie pustego binarnego drzewa i dziedziczy klas `BinaryTree` i na uzupełnienie go kolejnymi wartościami z podanej listy zaczynając od korzenia tak, aby wartość lewego dziecka była zawsze mniej wartości rodzica, a wartość prawego – większa lub równa.

```
class BinarySearchTree(BinaryTree):
    def __init__(self):
        self.root = None
```

Zaimplementuj metody

(a) `def add(self, value):` dodaje nowy węzeł z podanej wartością wedlug zasad uporządkowania. Wskazówka: zrób wewnątrz pomocnicza funkcje rekurencyjną `def ordered_add(node, value):`, która dodaje węzeł zaczynając od wskazanego w następny sposób:

jeżeli value < node.value i nie ma lewego dziecka, to dodaj TreeNode(value) jako lewe dziecko, jeżeli jest lewe dziecko - zrób funkcją na nim. To sam z prawej strony.

(b) Zaimplementuj metodą `to_search_tree(self, elements: iterable)` zapisu listy do drzewa

(c) Zaimplementuj metodą `search_tree(self, value) ->Bool` wyszukiwania, użyj wewnątrz funkcje rekurencyjnej na węzłach.

(d) Która z metod dla teraz węzłów może posortować listą i dlaczego? Zaimplementuj sortowanie list w ten sposób.

Testy:

(a) Węzły:

```
H = TreeNode("H")
I = TreeNode("I")
D = TreeNode("D",H,I)
J = TreeNode("J")
E = TreeNode("E",J)
B = TreeNode("B",D,E)
K = TreeNode("K")
F = TreeNode("F",K)
C = TreeNode("C",F,TreeNode("G"))
A = TreeNode("A",B,C)

in_order = []
A.traverse_in_order(in_order.append)
assert [x.value for x in in_order] == ["H",
          "D", "I", "B", "J", "E", "A", "K", "F", "C", "G"]

pre_order = []
A.traverse_pre_order(pre_order.append)
assert [x.value for x in pre_order] == [ "A","B","D",
               "H","I","E",  "J", "C", "F",  "K",  "G"]

post_order = []
A.traverse_post_order(post_order.append)
assert [x.value for x in post_order] == [ "H","I","D", "J", "E",
                "B","K" , "F",  "G","C","A"]
```

(b) Drzewo binarne

```
my_tree = BinaryTree()
assert my_tree.root == None
my_tree.root = A
my_tree.height() == 3
#my_tree.show()
```

(c) Drzewo wyszukiwań

```
search_tree = BinarySearchTree()
search_tree.root == None
assert search_tree.search(0) == False
search_tree.to_tree([4,3,8,-1,5,1,9,10,2])

#search_tree.show()

pre_order = []
search_tree.root.traverse_pre_order(pre_order.append)
assert [x.value for x in pre_order] == [4, 3, -1, 1, 2, 8, 5, 9, 10]

assert search_tree.search(2) == True
assert search_tree.search(4) == True
assert search_tree.search(5) == True
assert search_tree.search(10) == True
assert search_tree.search(-1) == True
assert search_tree.search(-5) == False
assert search_tree.search(7) == False
assert search_tree.search(100) == False
```