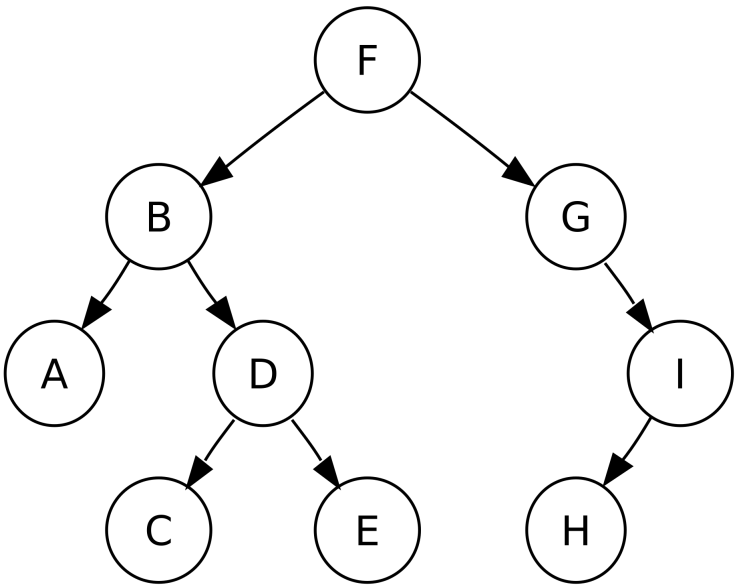
Algorytmy i struktury danych (studia stacjonarne)

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

Ćwiczenie 6 (Drzewo) 19.11.2025

Zadanie 5.1. Drzewa są abstrakcyjną strukturą danych służącą do przechowywania danych w sposób hierarchiczny. Drzewa składają się z węzłów, które mają rodzica i dzieci. Szczególnym przypadkiem węzła są: korzeń, który nie posiada rodzica oraz liście, które nie posiadają dzieci.

Przykład drzewa może stanowić strona internetowa zakodowana przy użyciu języka HTML.



Przygotować implementację drzewa wg następujących wymagań:

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

```
from typing import Any, List, Union, Callable

class TreeNode:

    value: Any
    children: List["TreeNode"]
```

W klasie należy umieścić następujące metody:

(a) funkcja `__str__` zwraca *value*

(b) `is_leaf() -> bool`, która sprawdzi czy węzeł jest liściem

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

(d) `for_each_deep_first_rec(visit: Callable[["TreeNode"], None]) -> None`, która wykona operację przechodzenia po węzłach metodą **deep first** według następujących instrukcji:

- odwiedź bieżący wierzchołek i wykonaj na nim funkcję `visit` (przyjętą w parametrze)
- dla wszystkich dzieci bieżącego węzła wykonaj metodę `for_each_deep_first_rec()`

(e) `for_each_breadth_first(visit: Callable[["TreeNode"], None]) -> None`, która wykona operację przechodzenia po węzłach metodą **breadth seach/level order** według następujących instrukcji:

- odwiedź bieżący wierzchołek i wykonaj na nim funkcję `visit` (przyjętą w parametrze)
- wszystkie dzieci bieżącego węzła dodaj do pustej kolejki FIFO (uzyj wbudowany typ **list**)
- dopóki kolejka nie jest pusta, dla każdego pierwszego elementu w kolejce (element):
  - odwiedź element
  - dodaj do kolejki wszystkie węzły, których rodzicem jest element

(f) * `for_each_deep_first(visit: Callable[["TreeNode"], None]) -> None`, która wykona operację przechodzenia po węzłach metodą **deep first** nierekurencyjnie. Wskazowka: podobnie do breadth, tylko użyj LIFO.

(g) `search(value: Any) -> Union["TreeNode", None]`, która sprawdzi czy bieżący węzeł lub jego dzieci zawierają wartość podaną w parametrze, przy użyciu dowolnej metody przechodzenia po węzłach

(h) rekurencyjna metoda `node_to_nested_dict(self))-> dict` przekształca kolejność węzły na zagnieżdżone słownik:

- utwórz słownik, gdzie pod kluczem self.value leży pusty słownik
- dla każdego dziecka dodaj do pustego słownika pod kluczem wartości tego dziecka słowniki, stworzone dla odpowiedniego dziecka

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

```
class Tree:
    root: TreeNode
```

(a) metoda `add(value: Any, parent_value: Any) -> None` doda nowe dziecko z wartościąprzekazaną w parametrze value, którego rodzicem będzie węzeł z wartiscią, przekazanej w parametrze `parent_value`

(b) metoda `for_each_deep_first_rec(visit: Callable[["TreeNode"], None]) -> None` wykona operację przechodzenia po węzłach metodą **deep first**, rozpoczynając od korzenia

(c) metoda `for_each_breadth_first(self, visit: Callable[["TreeNode"], None]) -> None` wykona operację przechodzenia po węzłach metodą **breadth seach/level order** , rozpoczynając od korzenia

(d)

(e) metoda `tree_to_nested_dict(self))-> dict` przekształca drzewo na zagnieżdżone słownik zaczynając od korzenia

(f) metoda `tree_to_dict(self)-> dict` przekształca drzewo na słownik `value : values of children`. Podpowiedz: użyj `for_each_breadth_first`

(g) metoda `dict_to_tree(self, d: dict)-> None` przekształca słownik na drzewo. Uwaga: zerowy klucz słownika jest korzeniem

Testy:

(a) Węzły:

```
node0 = TreeNode("F")
assert node0.value == "F"
assert node0.children == []
assert str(node0) == "F"
assert node0.is_leaf() == True
assert node0.node_to_nested_dict() == {"F": {}}

node1 = TreeNode("B", [TreeNode("A"),TreeNode("D")])
node2 = TreeNode("G")
assert node2.children == []
assert node1.children[0].value == "A"
node0.add(node1)
node0.add(node2)
assert node0.children == [node1, node2]
assert node2.children == []
l_deep_rec = []
node0.for_each_deep_first_rec(l_deep_rec.append)
assert [x.value for x in l_deep_rec] == ["F", "B", "A", "D", "G"]

#l_deep = []
#node0.for_each_deep_first(l_deep.append)
#assert [x.value for x in l_deep] == ["F", "B", "A", "D", "G"]

l_breadth = []
node0.for_each_breadth_first(l_breadth.append)
assert [x.value for x in l_breadth] == ["F", "B", "G", "A", "D"]
assert node0.search("G") == node2
assert node1.search("G") is None

assert(node0.node_to_nested_dict()) == {"F": {"B": {"A": {},
"D": {}}, "G": {}}}
```

(b) Drzewo:

```
tree_ = Tree()
```

```
assert tree_.root is None
tree_.root = node0

tree_.add("C","D")
tree_.add("E","D")
tree_.add("I","G")
tree_.add("H","I")

l_deep_tree = []
tree_.for_each_deep_first_rec(l_deep_tree.append)
assert [x.value for x in l_deep_tree] == ["F", "B", "A", "D", "C",
 "E", "G", "I", "H"]

l_breadth_tree = []
tree_.for_each_breadth_first(l_breadth_tree.append)
assert [x.value for x in l_breadth_tree] == ["F", "B", "G", "A", "D",
 "I", "C", "E", "H"]
d = tree_.tree_to_dict()
assert d == {"F": ["B", "G"], "B": ["A", "D"],
"G": ["I"], "A": [], "D": ["C", "E"], "I": ["H"],
"C": [], "E": [], "H": []}

d1 = tree_.tree_to_nested_dict()
assert d1 == {"F": {"B": {"A": {}, "D": {"C": {}, "E": {}}},
 "G": {"I": {"H": {}}}}}

tree_1 = Tree()
assert tree_1.root == None
tree_1.dict_to_tree(d)
l_deep_tree_1 = []
tree_1.for_each_deep_first_rec(l_deep_tree_1.append)
assert [x.value for x in l_deep_tree_1] == ["F", "B", "A", "D",
"C", "E", "G", "I", "H"]
```