**ФХарківський національний економічний університет**

**імені Семена Кузнеця**

**ЗВІТ**

**З ВИКОНАННЯ ІНДИВІДУАЛЬНОГО ЗАВДАННЯ № 1**

**за дисципліною: *“*Розробка та аналіз алгоритмів**”

**На тему: «Створення бінарного дерева з наданої множини вузлів та виведення послідовності вершин, які утворюються під час обходу цього дерева в інфіксному порядку»**

**Варіант: 5**

**Виконав: студент факультету Інформаційних технологій**

**1 курсу, спец. Кібербезпека,**

**групи 6.04.125.010.21.2**

**Бойко Вадим Віталійович**

**Перевірив:**

**Солодовник Ганна Валеріївна**

**ХНЕУ ім. С. Кузнеця**

**2022**

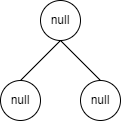
**Код з імплементацією структури даних бінарне дерево   
  
Я вирішив написати клас на мові програмування Python**

*class* Tree:  
 *def \_\_init\_\_*(self, *value* = *None*, *left* = *None*, *right* = *None*):  
 self.value = *value  
 if left* != *None*:  
 self.left = Tree(*left*)  
 *else*:  
 self.left = *left  
  
 if right* != *None*:  
 self.right = Tree(*right*)  
 *else*:  
 self.right = *right  
  
  
 def* add\_value(self, *value*):  
 *if* self.value == *None*:  
 self.value = *value  
 else*:  
 *if value* < self.value:  
 *if* (self.left == *None*):  
 self.left = Tree(*value*)  
 *else*:  
 self.left.add\_value(*value*)  
 *else*:  
 *if* (self.right == *None*):  
 self.right = Tree(*value*)  
 *else*:  
 self.right.add\_value(*value*)  
  
  
 *def* add\_value\_prefix(self, *main*, *left*, *right*):  
 *if left* == '^':  
 left = *None  
 if right* == '^':  
 right = *None  
 if* self.value == *None*:  
 self.value = *main* self.add\_leaf(*left*, *right*)  
 *return* tree = self.find\_tree\_by\_value\_type\_prefix(*main*)  
 tree.add\_leaf(*left*, *right*)  
  
  
 *def* find\_tree\_by\_value\_type\_prefix(self, *value*):  
 *if* self.value == *value*:  
 *return* self  
 *if* self.left == *None and* self.right == *None*:  
 *return  
 if* self.left != *None*:  
 l\_t = self.left.find\_tree\_by\_value\_type\_prefix(*value*)  
 *if* type(l\_t) == Tree:  
 *return* l\_t  
 *if* self.right != *None*:  
 r\_t = self.right.find\_tree\_by\_value\_type\_prefix(*value*)  
 *if* type(r\_t) == Tree:  
 *return* r\_t  
  
  
 *def* get\_value(self):  
 *return* self.value  
  
  
 *def* get\_left\_leaf(self):  
 *return* self.left  
  
  
 *def* add\_leaf(self, *left* = *None*, *right* = *None*):  
 *if left* != *None*:  
 self.left = Tree(*left*)  
 *if right* != *None*:  
 self.right = Tree(*right*)  
  
  
 *def* get\_right\_leaf(self):  
 *return* self.right  
  
  
 *def* get\_tree(self):  
 *return* self  
  
  
 *def* set\_value(self, *value* = *None*):  
 self.value = *value  
  
  
 def* prefix\_loop(self):  
 *if* self.value != *None*:  
 print(self.value)  
 *if* self.left == *None and* self.right == *None*:  
 *return  
 if* self.left != *None*:  
 self.left.prefix\_loop()  
 *if* self.right != *None*:  
 self.right.prefix\_loop()  
  
  
 *def* postfix\_loop(self):  
 *if* self.left != *None*:  
 self.left.postfix\_loop()  
 *if* self.right != *None*:  
 self.right.postfix\_loop()  
 print(self.value)  
  
  
 *def* infix\_loop(self):  
 *if* self.left == *None or* self.right == *None*:  
 print(self.value)  
 *if* self.left != *None*:  
 self.left.infix\_loop()  
 *if* self.right != *None*:  
 self.right.infix\_loop()  
 *if* self.left != *None and* self.right != *None*:  
 print(self.value)

Наприклад для побудови дерева ми знаємо корінь дерева та знаємо які гілки в нас будуть та сігнатура буде така (корінь, ліва гілка, права гілка)  
  
якщо значення гілки буде дорівнювати ‘^’, тоді гілка буде пуста   
  
наприклад в нас є такий масив вхідних даних

[  
 ('A', 'B', 'C'),  
 ('B', 'D', 'E'),  
 ('C', 'F', 'G'),  
 ('D', 'H', 'I'),  
 ('E', 'J', '^'),  
 ('F', 'K', 'L'),  
 ('G', 'M', '^'),  
 ('H', '^', 'N'),  
 ('M', 'P', '^'),  
]

Спочатку нам потрібно ініціалізувати дерево, ми отримаємо так дерево

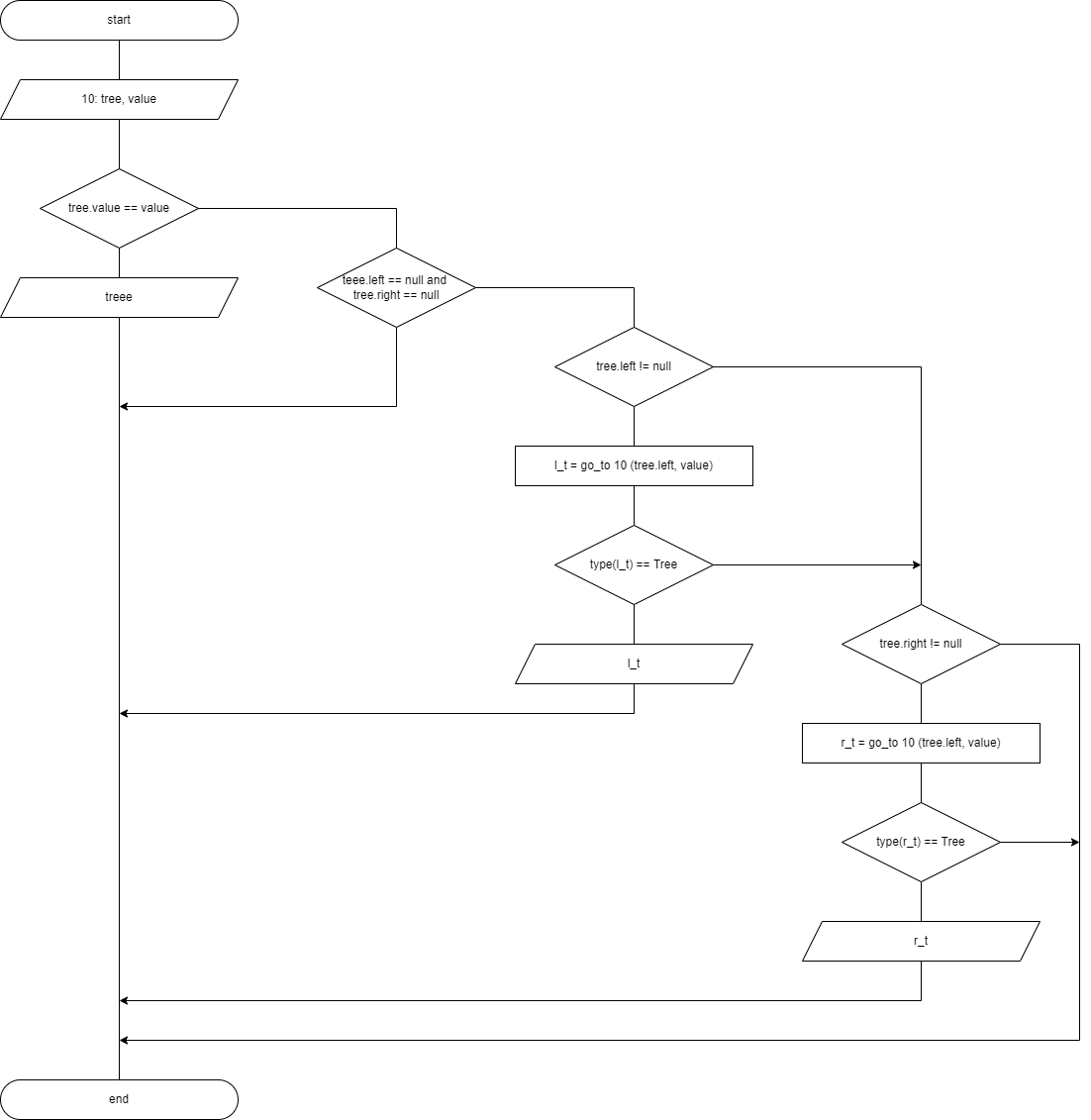


Тепер завдяки циклу ми проходимо по масиву та наповнюємо дерево

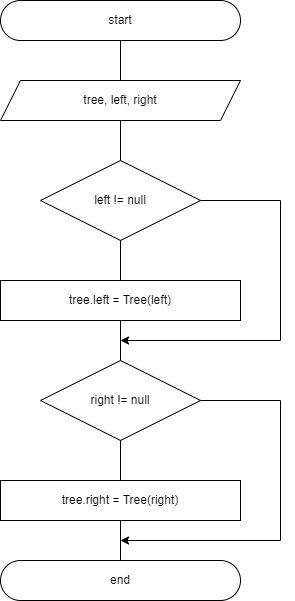
Для наповнення дерева ми будемо використовувати метод класу: «**add\_value\_prefix**» яка в свою чергу всередині викликає: «**find\_tree\_by\_value\_type\_prefix**» для пошуку під-дерева

«**add\_leaf**» для додавання значень до гілокдерева

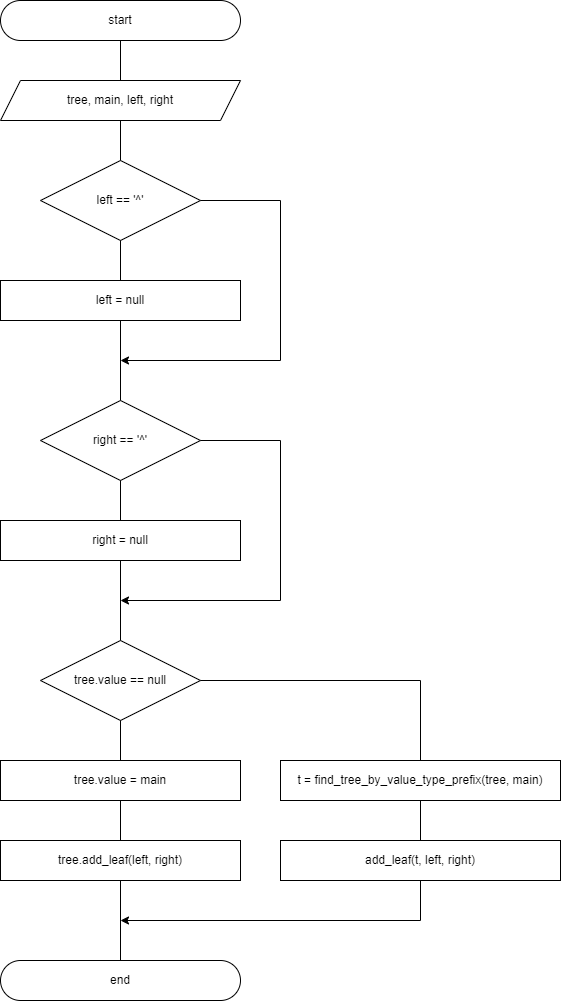
Алгоритм методу «**find\_tree\_by\_value\_type\_prefix**» в результаті повертає дерево, яке ми шукали по за корнем або нічого



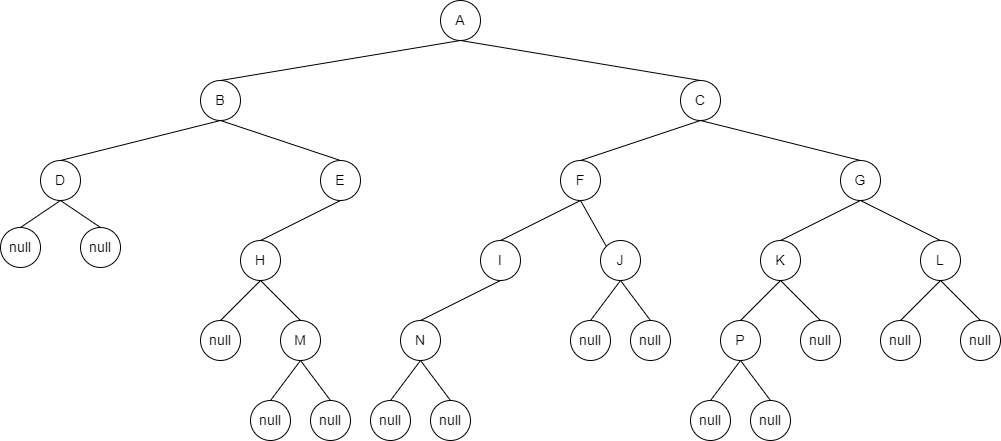
Алгоритм методу «**add\_leaf**», який створює нові гілки в дереві



Алгоритм «**add\_value\_prefix**»



В результаті ми отримаємо таке дерево



Бінарні дерева пошуку набагато ефективніші в операціях пошуку, аніж лінійні структури, в яких витрати часу на пошук пропорційні [*O(n)*](https://uk.wikipedia.org/wiki/%D0%9D%D0%BE%D1%82%D0%B0%D1%86%D1%96%D1%8F_%D0%9B%D0%B0%D0%BD%D0%B4%D0%B0%D1%83), де n — розмір [масиву](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D0%B8%D0%B2_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)) даних, тоді як в повному бінарному дереві цей час пропорційний в середньому *O(log2n)* або *O(h)*, де h — висота дерева (хоча гарантувати, що h не перевищує log2n можна лише для [збалансованих дерев](https://uk.wikipedia.org/wiki/%D0%97%D0%B1%D0%B0%D0%BB%D0%B0%D0%BD%D1%81%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), які є ефективнішими в алгоритмах пошуку, аніж прості бінарні дерева пошуку)

Обчислювальна складність у записі великого О

|  |  |  |
| --- | --- | --- |
| Тип | Середня | Найгірша |
| Пошук | O(log n) | O(n) |
| Вставлення | O(log n) | O(n) |
| Видалення | O(log n) | O(n) |

Використання структури даних

Бінарні дерева пошуку: Бінарні дерева використовуються для побудови бінарного дерева пошуку, яке використовується у багатьох пошукових програмах, таких як набори та карти у багатьох мовних бібліотеках.