Практическая работа №6.

Тема: «Структура данных «дерево».

Цель работы: изучить СД «дерево», научиться их программно реализовывать и использовать.

Двоичное дерево поиска (англ. binary search tree, BST) – это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева – левое и правое – являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше либо равны, нежели значение ключа данных самого узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

Определим класс бинарного дерева поиска, он инициализируется тремя значениями: корень, левый потомок и правый потомок (Листинг 1). Так как данная реализация рекурсивна, каждый его узел является поддеревом главного дерева.

Листинг 1. Инициализация класса бинарного дерева поиска.

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

Реализуем публичный метод для вставки элемента в бинарное дерево (Листинг 2). Диаграмма деятельностей для данного метода представлена на рисунке 1.

Листинг 2. Публичный метод вставки элемента.

```
def insert(self, data):
        if data <= self.root and self.left:
            self.left.insert(data)
        elif data <= self.root:
            self.left = MyBinarySearchTree(data)
        elif data > self.root and self.right:
            self.right.insert(data)
        else:
            self.right = MyBinarySearchTree(data)
```

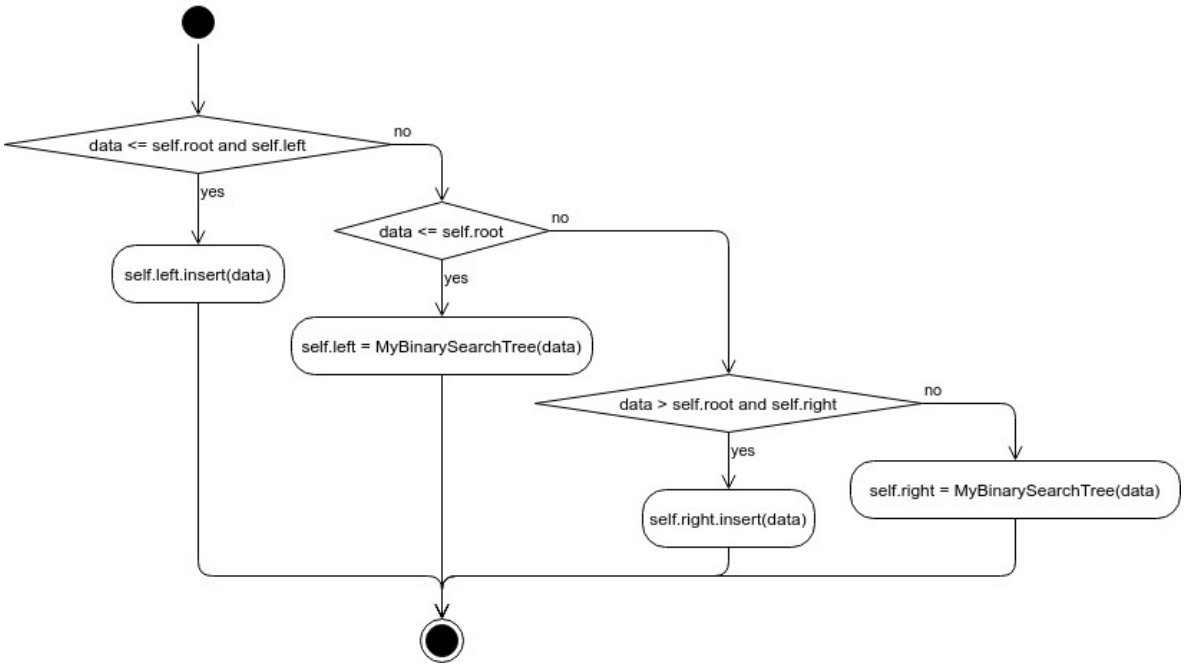

Рисунок 1. Диаграмма деятельностей метода для вставки элемента в бинарное дерево.

Реализуем публичный метод для поиска элемента в бинарном дереве (Листинг 3). Диаграмма деятельностей для данного метода представлена на рисунке 2.

Листинг 3. Публичный метод поиска элемента.

```
def search(self, value):
        if value < self.root and self.left:
            return self.left.search(value)
        elif value > self.root and self.right:
            return self.right.search(value)
        return value == self.root
```

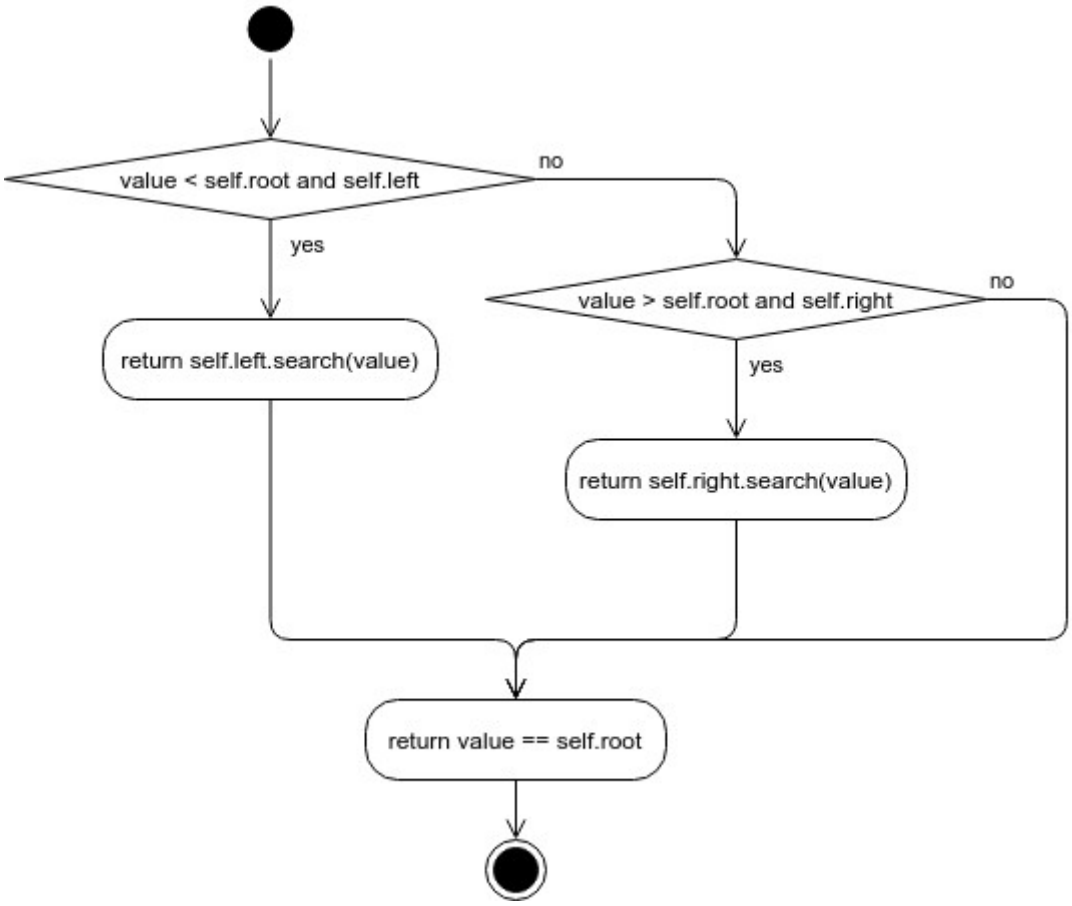

Рисунок 2. Диаграмма деятельностей метода для поиска элемента в бинарном дереве.

Приватный метод для поиска поддерева (Листинг 4) отличается от публичного метода поиска тем, что в случае равенства искомого значения и корня поддерева он возвращает это поддерево. Этот метод необходим для реализации подсчета элементов в поддереве.

Листинг 4. Приватный метод поиска поддерева.

```
def __subtree_search(self, value):
        if value < self.root and self.left:
            return self.left.__subtree_search(value)
        elif value > self.root and self.right:
            return self.right.__subtree_search(value)
        if value == self.root:
            return self
```

Для реализации публичного метода удаления элемента необходимо реализовать ещё два приватных метода: clear_node для очистки узла и find_minimum_value для поиска минимального элемента в поддереве (Листинг 5).

Листинг 5. Приватные методы, необходимые для метода удаления элемента.

```
    def __clear_node(self):
            self.root = None
            self.left = None
            self.right = None

    def __find_minimum_value(self):
            if self.left:
                return self.left.__find_minimum_value()
            else:
                return self.root
```

Реализуем публичный метод для удаления элемента в бинарном дереве (Листинг 6). Его логика работы следующая. Если удаляемый узел не имеет потомков, он просто удаляется из дерева. Если удаляемый узел имеет одного потомка (левого или правого), узел удаляется, а на его место ставится единственный потомок. Если удаляемый узел имеет двух потомков, необходимо найти минимальное значение потомков в поддереве, и поставить это значение на место удаляемого узла.

Листинг 6. Публичный метод удаления элемента.

```
    def delete_node(self, value, parent):
            if value < self.root and self.left:
                return self.left.delete_node(value, self)
            elif value < self.root:
                return False
            elif value > self.root and self.right:
                return self.right.delete_node(value, self)
            elif value > self.root:
                return False
            else:
                if self.left is None and self.right is None and
self == parent.left:
                    parent.left = None
                    self.__clear_node()
```

```
                elif self.left is None and self.right is None
and self == parent.right:
                    parent.right = None
                    self.__clear_node()
                elif self.left and self.right is None and self
== parent.left:
                    parent.left = self.left
                    self.__clear_node()
                elif self.left and self.right is None and self
== parent.right:
                    parent.right = self.left
                    self.__clear_node()
                elif self.left is None and self.right and self
== parent.left:
                    parent.left = self.right
                    self.__clear_node()
                elif self.left is None and self.right and self
== parent.right:
                    parent.right = self.right
                    self.__clear_node()
                else:
                    self.root                                =
self.right.__find_minimum_value()
                    self.right.delete_node(self.root, self)
                return True
```

Для вывода на экран элементов дерева реализуем методы обходов дерева в глубину (Листинг 7). Реализован префиксный (прямой) обход pre_order, инфиксный (симметричный) обход in_order и постфиксный (обратный) обход post_order.

Листинг 7. Публичные методы для обхода дерева в глубину.

```
def pre_order(self):
        print(self.root, end=" ")
        if self.left:
            self.left.pre_order()
```

```
            if self.right:
                self.right.pre_order()

    def in_order(self):
            if self.left:
                self.left.in_order()
            print(self.root, end=" ")
            if self.right:
                self.right.in_order()

    def post_order(self):
            if self.left:
                self.left.post_order()
            if self.right:
                self.right.post_order()
            print(self.root, end=" ")
```

Для решения задачи вывода упорядоченной по возрастанию последовательности элементов дерева, меньших определенного значения ключа, реализуем приватный метод инфиксного обхода дерева, который заполняет динамический массив элементами дерева (Листинг 8). Выбран именно инфиксный обход, потому что он возвращает уже отсортированную по возрастанию последовательность элементов. Диаграмма деятельностей для данного метода представлена на рисунке 3.

Листинг 8. Приватный метод инфиксного обхода дерева.

```
def __in_order(self, array):
        if self.left:
            self.left.__in_order(array)
        array.append(self.root)
        if self.right:
            self.right.__in_order(array)
```

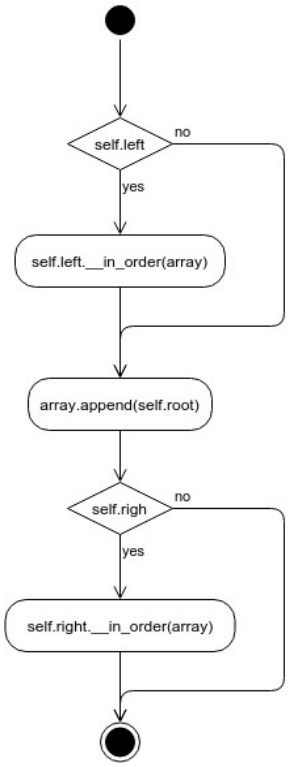

Рисунок 3. Диаграмма деятельностей приватного метода для инфиксного обхода дерева.

Реализуем публичный метод, для вывода упорядоченной последовательности элементов дерева, меньших заданного значения ключа (Листинг 9). Диаграмма деятельностей для данного метода представлена на рисунке 4.

Листинг 9. Метод вывод последовательности элементов дерева.

```
def sequence_less_key(self, key):
        keys = []
        out = []
        self.__in_order(keys)
        for i in range(len(keys)):
            if keys[i] < key:
                out.append(keys[i])
            else:
                break
        print(out)
```

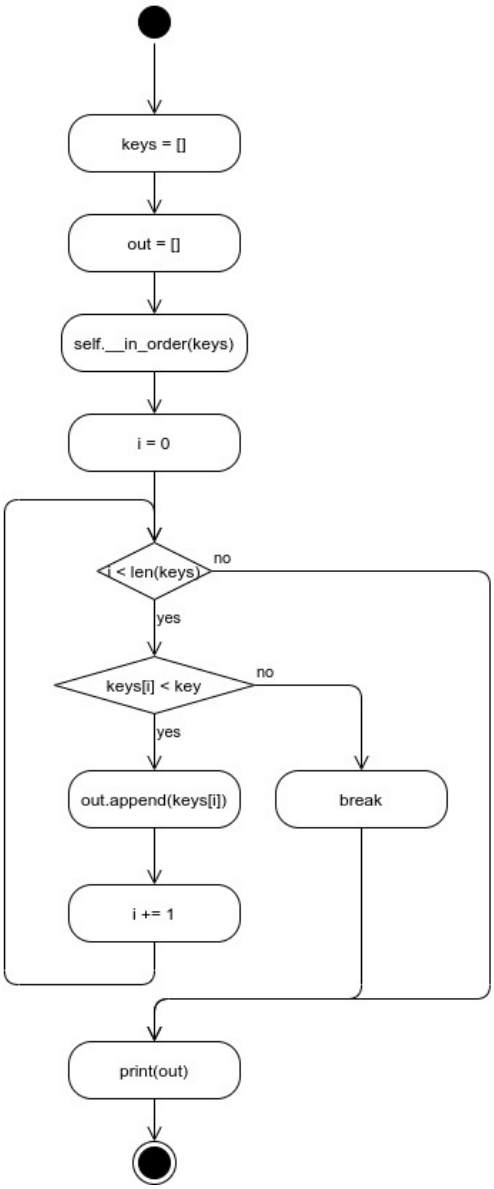

Рисунок 4. Диаграмма деятельностей метода для вывода упорядоченной последовательности элементов дерева.

Для решения задачи вывода вершин, для которых левое и правое поддерево имеют равное количество вершин, реализуем приватный метод подсчета количества вершин в поддереве (Листинг 10). Диаграмма деятельностей для данного метода представлена на рисунке 5. Этот метод позволяет сравнить количество вершин в левом и правом поддеревьях некоторого дерева. Возвращает истину, если количества вершин идентичны.

Листинг 10. Приватный метод подсчета вершин в поддереве.

```
def __subtrees_sizes(self, root):
        left = []
        right = []
        found = self.__subtree_search(root)
```

```
        if found.left:
            found.left.__in_order(left)
        elif found.right:
            found.right.__in_order(right)
        return len(left) == len(right)
```

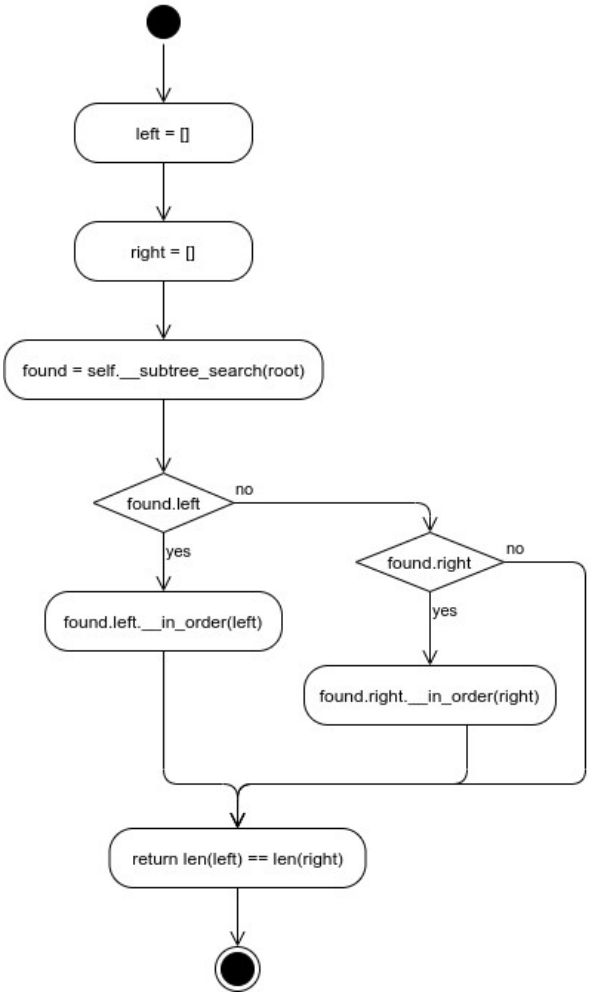

Рисунок 5. Диаграмма деятельностей для приватного метода подсчета элементов в поддереве.

Реализуем публичный метод вывода вершин, для которых левое и правое дерево имеют равно количество вершин (Листинг 11). Диаграмма деятельностей для данного метода представлена на рисунке 6.

Листинг 11. Публичный метод вывода вершин.

```
def sub_trees_compare(self):
        keys = []
        nodes = []
        self.__in_order(keys)
        for i in range(len(keys)):
```

```
            root = keys[i]
            compare = self.__subtrees_sizes(root)
            if compare:
                nodes.append(root)
    print(nodes)
```

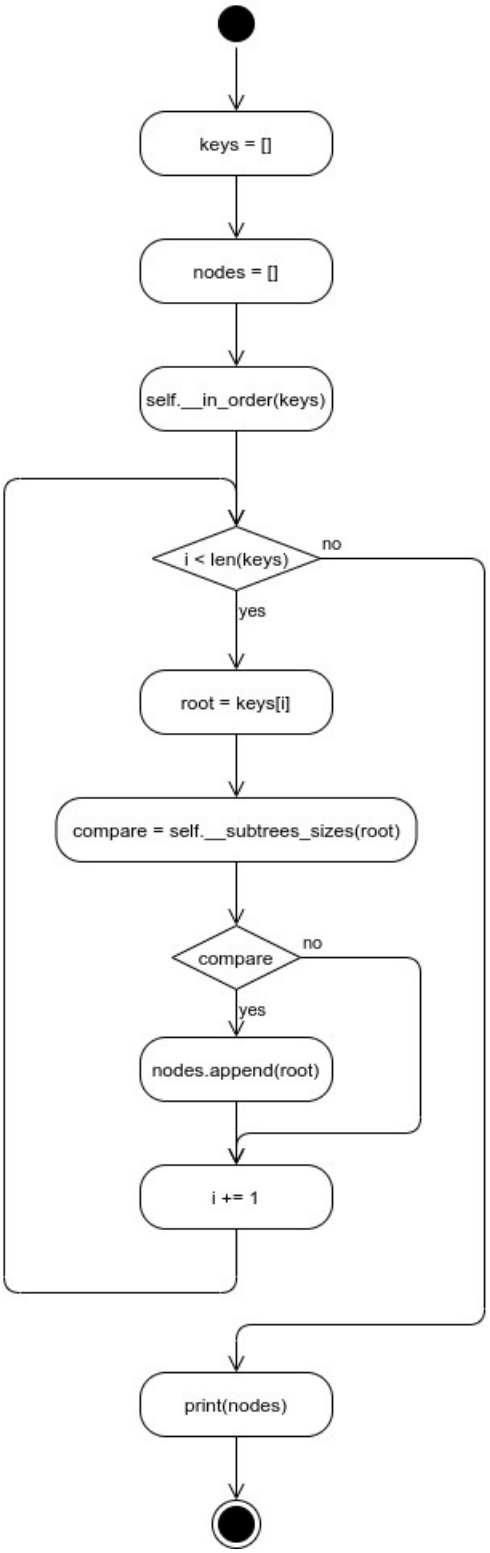

Рисунок 6. Диаграмма деятельностей для публичного метода вывода вершин, для которых левое и правое поддерево имеют равное количество вершин

Исходный код программы для выполнения задания представлен на листинге 12.

Листинг 12. Исходный код программы.

```
from Binary_search_tree.binary_search_tree import MyBinarySearchTree
import random as rd


tree = MyBinarySearchTree(rd.randint(5, 50))

for i in range(20):
    tree.insert(rd.randint(5, 50))

print('Pre-order traversal:', end=' ')
tree.pre_order()
print('\nIn-order traversal:', end=' ')
tree.in_order()
print('\nPost-order traversal:', end=' ')
tree.post_order()
key = rd.randint(5, 50)
print('\nSequence less', key, ':', end=' ')
tree.sequence_less_key(key)
print('Tree\'s nodes with same count in subtrees\' nodes:', end=' ')
tree.sub_trees_compare()
```

Вывод: в ходе выполнения данной практической работы была изучена структура данных «дерево», её программная реализация и использование.