Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

Тема 9. Бинарное дерево поиска. Реализация на основе ООП

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

апрель, 2025

9.1 Определения

Вершина — это базовая сущность для описания графа.

Граф — это пара (V, E), состоящая из непустого множества вершин V и множества пар вершин E.

Пара вершин из множества E может быть упорядоченной и неупорядоченной. **Ребро** — это неупорядоченная пара вершин.

Граф называется *неориентированным*, если множество E состоит только из ребер.

Граф называется *связным*, если для любых двух вершин из множества V существует маршрут из последовательности ребер графа, соединяющий эти вершины.

Цепь -- это маршрут, состоящий из различных ребер. **Цикл** -- это циклическая цепь.

Граф называется *ациклическим*, если он не содержит циклов.

Дерево — это неориентированный связный ациклический граф.

Корневое дерево — это дерево с одной выделенной вершиной, которая называется *корнем* дерева.

Бинарное дерево — это корневое дерево, вершины которого, исключая корень, содержатся в двух непересекающихся множествах: *левое поддерево* и *правое поддерево*. При этом каждое поддерево либо является пустым множеством, либо бинарным деревом.

Любая вершина называется *родительской вершиной* для своих поддеревьев.

Каждая родительская вершина бинарного дерева может иметь не более двух *вершин-потомков*, левую и правую.

Вершина, которая не имеет потомков, называется *листом* дерева.

Бинарное дерево поиска (binary search tree) — это бинарное дерево, каждая вершина которого имеет значения одного типа, для которых определены операции сравнения (например, значениями вершин являются объекты типа `int` или `float`). При этом значения вершин непустого левого поддерева меньше ($<$), чем значение соответствующей родительской вершины, а значения вершин непустого правого поддерева больше либо равны ($\geq$) значению соответствующей родительской вершины.

Бинарное дерево поиска — это упорядоченная динамическая структура данных, которая позволяет быстро производить поиск элементов, а также быстро удалять и добавлять элементы.

9.2 Представление бинарного дерева поиска

Анализируя структуру бинарного дерева поиска, нетрудно видеть, что его построение можно свести к умению строить **элемент дерева** (вершина) – сущность, имеющую значение и связанную с левым и правым поддеревом, если они существуют.

Тогда уместно рекурсивное определение: **бинарное дерево** – это структура данных, которая либо пуста, либо содержит элемент дерева (корневую вершину), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями.

В силу этого, представление бинарного дерева сводится к представлению элементов дерева и установлению/описанию связей между элементами дерева.

Элемент дерева будет представлен экземпляром класса `BinaryNode`.

Бинарное дерево поиска будет представлено экземпляром класса `BinaryTree`.

Пустое бинарное дерево поиска будет представлено экземпляром класса `EmptyNode`.

Экземпляр класса `BinaryTree` будет содержать ссылку на корень дерева. Корень дерева будет представлен экземпляром класса `EmptyNode`, если дерево пустое, или экземпляром класса `BinaryNode`, если дерево не пусто.

Для связывания элемента дерева со своими поддеревьями экземпляр класса `BinaryNode` будет содержать ссылки на корни левого и правого поддеревьев. Корень поддерева представляется экземпляром класса `EmptyNode`, если поддерево пустое, или экземпляром класса `BinaryNode`, если поддерево не пусто.

В частности, лист дерева будет представлен экземпляром класса `BinaryNode`, правое и левое поддеревья которого являются экземплярами класса `EmptyNode`.

9.3 Проектирование классов

Бинарное дерево поиска будем описывать с помощью трех классов: `BinaryTree`, `BinaryNode`, `EmptyNode`.

Класс `BinaryTree` является основным. Он определяет шаблон для создания объекта, представляющего бинарное дерево поиска.

Класс `BinaryNode` является вспомогательным. Он определяет шаблон для создания объекта, представляющего элемент (вершину) бинарного дерева поиска.

Класс `EmptyNode` является вспомогательным. Он определяет шаблон для создания объекта, представляющего "пустой" элемент (вершину) бинарного дерева поиска без значения и без поддеревьев. Экземпляры класса `EmptyNode` будут использоваться для представления пустого дерева и пустых поддеревьев.

Классы `BinaryTree`, `BinaryNode`, `EmptyNode` наследуются только от `object`.

Экземпляр класса `BinaryTree` содержит атрибут `root`, который представляет корень дерева. Атрибут `root` хранит ссылку на экземпляр класса `EmptyNode`, если дерево пустое, или экземпляр класса `BinaryNode`, если дерево не пусто.

При таком связывании классов говорят о реализации концепции ООП **композиция**. Экземпляр класса `BinaryTree` называют *объектом-контейнером* для экземпляра класса `EmptyNode` или `BinaryNode`.

Экземпляр класса `BinaryNode` содержит атрибуты `left` и `right`, которые хранят ссылки на экземпляры классов `BinaryNode` или `EmptyNode`. Экземпляр класса `BinaryNode` является *объектом-контейнером* для двух экземпляров класса `BinaryNode` или `EmptyNode`, которые ссылаются на корневые вершины левого и правого поддеревьев.

Проектирование класса BinaryTree

Класс `BinaryTree` определяет шаблон для создания объекта, представляющего бинарное дерево поиска.

Определение класса `BinaryTree` содержит ТРИ метода: `__init__`, `__repr__` и `insert`.

Метод инициализации экземпляра класса `__init__` вызывается без аргументов и инициализирует единственный атрибут экземпляра класса `root` экземпляром класса `EmptyNode`:

```
def __init__(self):
    self.root = EmptyNode()
```

Назначение атрибута `root` -- это хранение ссылки на корень дерева.

Метод строкового представления экземпляра класса `__repr__` возвращает строковое представление для корня дерева `root` :

```
def __repr__(self):
    return repr(self.root)
```

Метод `insert(value)` является методом вставки в дерево НОВОГО элемента. Новый элемент всегда является листом со значением `value` .

Метод вставки элемента в дерево реализуется через метод вставки элемента в корневую вершину дерева `root` . При этом атрибут `root` переопределяется ссылкой на новый объект для представления корня

```
def insert(self, value):
    self.root = self.root.insert(value)
```

Определим класс `BinaryTree` :

In [1]:
```
class BinaryTree:
    def __init__(self):
        self.root = EmptyNode()

    def __repr__(self):
        return repr(self.root)

    def insert(self, value):
        self.root = self.root.insert(value)
```

Создать экземпляр класса `BinaryTree` пока нельзя, так как не определен класс `EmptyNode`

In [2]:
```
tree = BinaryTree()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 tree = BinaryTree()

Cell In[1], line 3, in BinaryTree.__init__(self)
      2 def __init__(self):
----> 3     self.root = EmptyNode()

NameError: name 'EmptyNode' is not defined
```

Проектирование класса BinaryNode

Класс `BinaryNode` определяет шаблон для создания объекта, представляющего элемент бинарного дерева поиска.

Определение класса `BinaryNode` содержит ТРИ метода: `__init__` , `__repr__` и `insert` .

Метод инициализации экземпляра класса вызывается с тремя аргументами `left` , `value` и `right` для инициализации одноименных атрибутов экземпляра класса `left` , `value` и `right` соответствующими значениями

```
def __init__(self, left, value, right):
        self.left = left
        self.value = value
        self.right = right
```

Назначение атрибута `left` -- это хранение ссылки на корень левого поддерева.
Назначение атрибута `right` -- это хранение ссылки на корень правого поддерева.
Назначение атрибута `value` -- это хранение значения вершины.

Метод строкового представления экземпляра класса `__repr__` возвращает кортеж, состоящий из строкового представления корня левого поддерева `left` , из строкового представления значения `value` вершины и из строкового представления корня правого поддерева `right`

```
def __repr__(self):
    return f'({self.left}, {self.value}, {self.right})'
```

Метод `insert(value)` является методом вставки в текущую вершину НОВОЙ вершины со значением `value` . Метод `insert` содержит основной алгоритм построения бинарного дерева поиска и будет определен позже

```
def insert(self, value):
    ...
```

Определим класс `BinaryNode` :

In [3]:
```
class BinaryNode:
    def __init__(self, left, value, right):
        self.left = left
        self.value = value
        self.right = right

    def __repr__(self):
        return f'({self.left}, {self.value}, {self.right})'

    def insert(self, value):
        ...
```

Создадим экземпляр класса `BinaryNode` :

In [4]:
```
node = BinaryNode(None, 5, None)
```

Сделаем вывод строкового представления для экземпляра класса `BinaryNode` :

In [5]:
```
print(node)
```

```
(None, 5, None)
```

Проектирование класса EmptyNode

Класс `EmptyNode` определяет шаблон для создания объекта, представляющего "пустой" элемент (вершину) бинарного дерева поиска без значения и без поддеревьев. Экземпляры класса `EmptyNode` будут использоваться для представления пустого дерева и пустых поддеревьев.

Класс `EmptyNode` определяется ДВУМЯ методами: `__repr__` и `insert`.

Метод инициализации экземпляра класса отсутствует, так как экземпляры класса `EmptyNode` не хранят никаких данных.

Метод строкового представления экземпляра класса `__repr__` возвращает один символ `*`

```
def __repr__(self):
    return '*'
```

Метод `insert(value)` является методом вставки в пустую вершину НОВОЙ вершины со значением `value`. Метод `insert` возвращает новый экземпляр класса `BinaryNode` со значением `value`. Левое и правое поддерево определяются ссылками на пустую вершину

```
def insert(self, value):
    return BinaryNode(self, value, self)
```

При вставке НОВОГО элемента в бинарное дерево поиска ссылка на экземпляр класса `EmptyNode` будет заменяться на ссылку на новый экземпляр класса `BinaryNode`.

Экземпляр класса `EmptyNode` создается только один раз при создании экземпляра класса `BinaryTree`. Каждое пустое поддерево для экземпляров класса `BinaryNode` будет ссылаться на ЕДИНСТВЕННЫЙ экземпляр класса `EmptyNode`.

Определим класс `EmptyNode`:

In [6]:
```
class EmptyNode:
    def __repr__(self):
        return '*'

    def insert(self, value):
        return BinaryNode(self, value, self)
```

Создадим экземпляр класса `EmptyNode`:

In [7]:
```
empty_node = EmptyNode()
```

Сделаем вывод строкового представления для экземпляра класса `EmptyNode`:

In [8]:
```
print(empty_node)
```

*

Создадим экземпляр класса `BinaryTree`:

In [9]:
```
tree = BinaryTree()
```

Бинарное дерево поиска пусто, корень представлен экземпляром класса `EmptyNode`. Пустое дерево выводится с помощью символа `*`

In [10]:
```
print(tree)
```

*

Вставим в дерево новый элемент со значением `10`:

In [11]:
```
tree.insert(10)
```

Сделаем вывод дерева:

In [12]:
```
print(tree)
```

```
(*, 10, *)
```

Дальнейшая вставка элементов в дерево невозможна, так как не определен метод `insert` класса `BinaryNode`.

9.4 Построение бинарного дерева поиска

Создадим экземпляр класса `BinaryTree`. Изначально дерево пусто, корень представлен экземпляром класса `EmptyNode`. Дерево выводится выводится с помощью символа `*`

In [40]:
```
tree = BinaryTree()
tree
```

Out[40]: *

Пусть задан список числовых значений `source_data`, на основании которого будет строиться бинарное дерево поиска

In [35]:
```
source_data = [5,1,10,3,4]
```

Элементы списка `source_data` будут добавляться в бинарное дерево поиска последовательно: сначала добавим значение `5`, потом значение `1` и т.д.

Сначала в дерево добавляется значение `source_data[0]` с помощью метода `insert` экземпляра класса `tree`. Метод `insert` класса `BinaryTree` вызывает метод `insert` для объекта, на который указывает атрибут `self.root` для корня. На текущем шаге корень представлен экземпляром класса `EmptyNode`. В результате создается экземпляр класса `BinaryNode`, который *замещает* пустую вершину в корне дерева.

In [41]:
```
tree.insert(source_data[0])
tree
```

Out[41]:
```
(*, 5, *)
```

Экземпляр класса BinaryTree не содержит поддеревьев, поэтому выводится в виде `(*,5,*)`.

Далее в дерево добавляется значение `source_data[1]` с помощью метода `insert` экземпляра класса `tree`. Метод `insert` класса `BinaryTree` вызывает метод `insert` для корня. На текущем шаге корень не пустой, он представлен экземпляром класса `BinaryNode`.

Алгоритм метода `insert(self, value)` класса `BinaryNode` следующий:

- добавляемое значение `value` сравнивается со значением корня `self.value`;
- если `value<self.value`, то вызывается метод `insert(value)` для левого поддерева;
- если `value>=self.value`, то вызывается метод `insert(value)` для правого поддерева.

In [42]:
```
tree.insert(source_data[1])
tree
```

Out[42]:
```
((*, 1, *), 5, *)
```

Так как `source_data[1]<self.value` (`1<5`), то в результате создается экземпляр класса `BinaryNode`, который *замещает* пустую вершину для левого поддерева корневой вершины.

Далее в дерево добавляется значение `source_data[2]` с помощью метода `insert` экземпляра класса `tree`. Метод `insert` класса `BinaryTree` вызывает метод `insert(value)` для корневой вершины.

In [43]:
```
tree.insert(source_data[2])
tree
```

Out[43]:
```
((*, 1, *), 5, (*, 10, *))
```

Так как `source_data[2]>self.value` (`10>5`), то в результате создается экземпляр класса `BinaryNode`, который *замещает* пустую вершину для правого поддерева корневой вершины.

Далее в дерево добавляется значение `source_data[3]`

In [44]:
```
tree.insert(source_data[3])
tree
```

Out[44]:
```
((*, 1, (*, 3, *)), 5, (*, 10, *))
```

Так как `source_data[3]<self.value` для корневой вершины (`3<5`), то значение добавляется в левое поддерево корневой вершины. Вершина левого поддерева сравнивается с добавляемым значением. Так как `source_data[3]>self.value` для корневой вершины левого поддерева (`3>1`), то в результате создается экземпляр класса `BinaryNode` , который *замещает* пустую вершину для правого поддерева корня левого поддерева вызовом метода `insert` .

Добавим в бинарное дерево поиска `tree` следующее значение `source_data[4]`

In [45]:
```
tree.insert(source_data[4])
tree
```

Out[45]:
```
((*, 1, (*, 3, (*, 4, *))), 5, (*, 10, *))
```

Добавим в созданное дерево числовые значения списка `source_date` еще раз

In [26]:
```
for i in source_data:
    tree.insert(i)
    print(tree)
```

```
((*, 1, (*, 3, (*, 4, *))), 5, ((*, 5, *), 10, *))
((*, 1, ((*, 1, *), 3, (*, 4, *))), 5, ((*, 5, *), 10, *))
((*, 1, ((*, 1, *), 3, (*, 4, *))), 5, ((*, 5, *), 10, (*, 10, *)))
((*, 1, ((*, 1, *), 3, ((*, 3, *), 4, *))), 5, ((*, 5, *), 10, (*, 10, *)))
((*, 1, ((*, 1, *), 3, ((*, 3, *), 4, (*, 4, *)))), 5, ((*, 5, *), 10, (*, 10,
*)))
```

9.5 Перегрузка операции принадлежности in

Спроектируем и реализуем для нашего бинарного дерева операцию проверки принадлежности `in` .

Операция `in` , вызванная для экземпляра `tree` класса `BinaryTree` в виде

`value in tree`
будет возвращать `True` , если значение `value` найдено в дереве, и `False` , если значение `value` не найдено в дереве.

Встроенная операция принадлежности `in` вызывает метод `__contains__` , поэтому этот метод необходимо определить для ТРЕХ классов `BinaryTree` , `EmptyNode` , `BinaryNode` .

Метод `__contains__` класса `BinaryTree` возвращает результат вызова операции `in` для корневой вершины

```
def __contains__(self, value):
    return value in self.root
```

Метод `__contains__` класса `EmptyNode` возвращает `False`

```
def __contains__(self, value):
    return False
```

Алгоритм метода `__contains__(self, value)` класса `BinaryNode` очень похож на алгоритма метода `insert`:

- проверяемое значение `value` сравнивается со значением корневой вершины `self.value`;
- если `value==self.value`, то возвращается `True`;
- если `value<self.value`, то возвращается результат вызова операции `in` для корня левого поддерева;
- если `value>self.value`, возвращается результат вызова операции `in` для корня правого поддерева.

9.6 Перегрузка встроенной функции len

Спроектируем и реализуем для нашего бинарного дерева встроенную функцию `len`.

Встроенная функция `len`, вызванная для экземпляра класса `BinaryTree`, будет возвращать количество элементов бинарного дерева поиска. Функция `len` вызывает метод `__len__`, поэтому этот метод необходимо определить для трех классов `BinaryTree`, `EmptyNode`, `BinaryNode`.

Метод `__len__` класса `BinaryTree` вызывает функцию `len` для корневой вершины

```
def __len__(self):
    return len(self.root)
```

Метод `__len__` класса `EmptyNode` возвращает 0

```
def __len__(self):
    return 0
```

Метод `__len__` класса `BinaryNode` возвращает значение атрибута `numberOfNodes` класса `BinaryNode`, который используется как счетчик созданных экземпляров класса `BinaryNode`

```
def __len__(self):
    return self.numberOfNodes
```

Атрибут `numberOfNodes` является атрибутом класса `BinaryNode`, а не атрибутом экземпляра класса `BinaryNode`. При создании класса атрибут `numberOfNodes` инициализируется значением 0. При создании экземпляра класса значение атрибута `numberOfNodes` увеличивается на 1. При создании дерева атрибут `numberOfNodes` "сбрасывается" в значение 0.

9.7 Обход бинарного дерева поиска

Обход графа — это маршрут, состоящий из ВСЕХ ребер графа.

Большинство алгоритмов работы с деревьями основаны на обходах деревьев (**tree traversing**).

Основные обходы бинарного дерева поиска

Прямой обход (префиксный, левый, preorder, `CLR`) выполняется по правилу: корень `C` -> *прямой обход* левого поддерева `L` -> *прямой обход* правого поддерева `R`.

Центрированный обход (инфиксный, симметричный, inorder, `LCR`) выполняется по правилу: *центрированный обход* левого поддерева `L` -> корень `C` -> *центрированный обход* правого поддерева `R`.

Обратный обход (постфиксный, правый, postorder, `LRC`) осуществляется по правилу: *обратный обход* левого поддерева (`L`) -> *обратный обход* правого поддерева `R` -> корень `C`.

Центрированный обход бинарного дерева поиска

Центрированный обход бинарного дерева поиска реализует **сортировку** по возрастанию числовой последовательности, записанной в виде дерева.

Эффективность сортировки числовой последовательности с помощью центрированного обхода бинарного дерева поиска равна

$$O(n \log(n)) + O(n),$$

где $O(n \log(n))$ операций необходимо для вставки n элементов в бинарное дерево поиска, $O(n)$ операций необходимо для обхода дерева, состоящего из n вершин.

Реализацию центрированного обхода осуществим с помощью метода `lcr` для трех классов `BinaryTree`, `EmptyNode`, `BinaryNode`.

Метод `lcr` класса `BinaryTree` вызывает метод `lcr` для корневой вершины

```
def lcr(self):
    return self.root.lcr()
```

Метод `lcr` класса `EmptyNode` возвращает пустой список

```
def lcr(self):
    return []
```

Метод `lcr` класса `BinaryNode` возвращает список значений вершин, соответствующих центрированному обходу дерева с корнем `self.value` для экземпляра класса `BinaryNode`. Результирующий список формируется на основе вызова метода `lcr` для левого поддерева `self.left` и правого поддерева `self.right`, осуществляя конкатенацию списков по правилу центрированного обхода

```
def lcr(self):
    ...
    result = self.left.lcr() + ...
    ...
    return result
```

Для тестирования реализации метода `lcr` для центрированного обхода дерева создадим список из 10^3 случайных чисел, равномерно распределенных на отрезке $[0, 100]$. Для этого воспользуемся функцией `uniform` из модуля `random`

In [28]:
```
import random as rnd
source_data = list(rnd.uniform(0,100) for _ in range(10**3))
```

Создадим бинарное дерево поиска на основе списка `source_data`

In [29]:
```
tree =  BinaryTree()
for i in source_data:
    tree.insert(i)
```

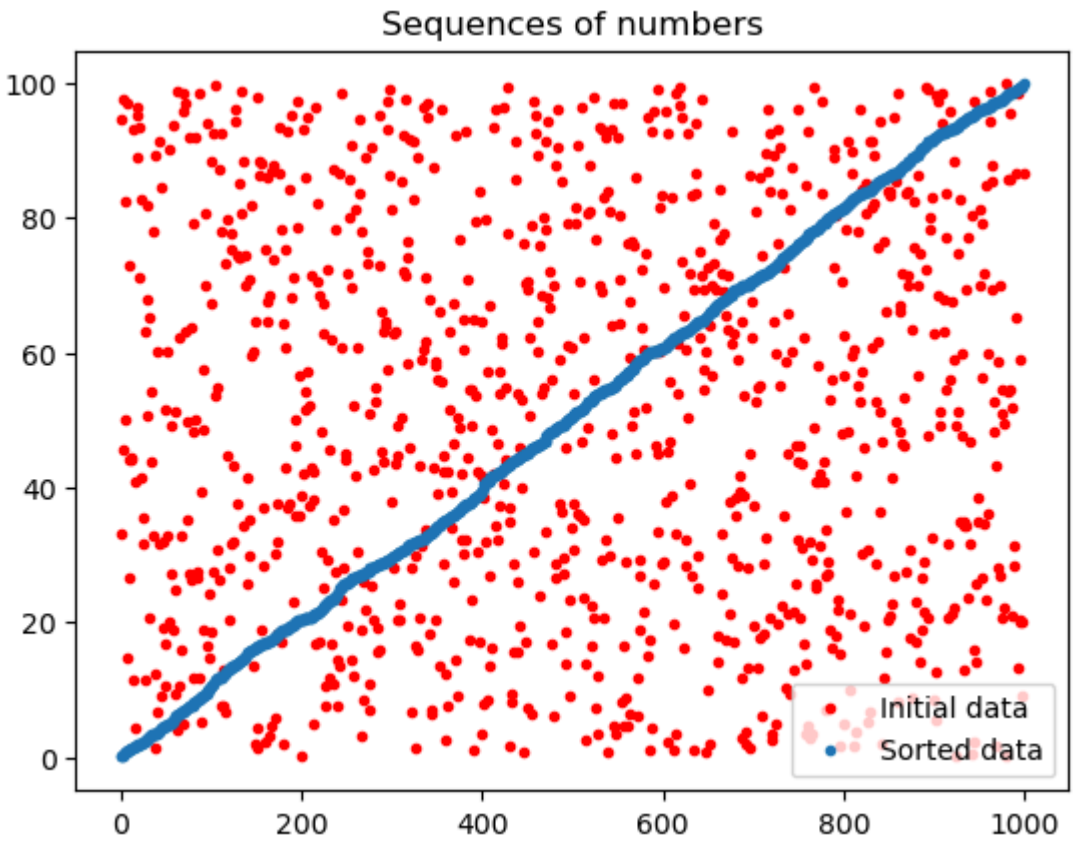
Вызовем метод `lcr` для экземпляра `tree`

In [30]:
```
sorted_data = tree.lcr()
```

Построим в одной графической области значения исходного списка и списка, полученного в результате центрированного обхода

In [32]:
```
import matplotlib.pyplot as plt
plt.plot(source_data,'r.', label='Initial data')
plt.plot(sorted_data,' .',label='Sorted data')
plt.title('Sequences of numbers')
plt.legend()
```

Out[32]:
```
<matplotlib.legend.Legend at 0x27013fede50>
```



Поиск минимального и максимального значений

Нахождение минимального и максимального значений в бинарном дереве поиска основано на *центрированном обходе дерева*.

Для поиска минимального значения нужно "двигаться" от корня дерева по левым поддеревьям. Для поиска максимального значения нужно "двигаться" от корня дерева по правым поддеревьям.

Реализацию нахождения минимального значения осуществим с помощью метода `min` для трех классов `BinaryTree`, `EmptyNode`, `BinaryNode`.

Метод `min` класса `BinaryTree` вызывает метод `min` для корневой вершины

```
def min(self):
    return self.root.min()
```

Метод `min` класса `EmptyNode` возвращает `None`

```
def min(self):
    return None
```

Метод `min` класса `BinaryNode` анализирует только левое поддерево для экземпляра класса `BinaryNode`

```
def min(self):
   if isinstance(self.left, EmptyNode):
   ...
```

Метод `min` класса `BinaryNode` возвращает либо `self.value`, если левое поддерево является пустой вершиной, либо вызывает метод `min` для корня левого поддерева.