Проектирование класса для реализации двоичного дерева поиска

Двоичные деревья поиска (Binary Search Tree - BST) позволяют выполнять все необходимые операции - поиск, вставку и удаление значений за время O(log N), что делает их эффективной структурой данных для хранения упорядоченных фрагментов информации и управления ими.

Эта структура будет особенно полезна, если мы собираемся часто изменять данные (добавлять и удалять), ведь, несмотря на то что упорядоченные массивы не уступают двоичным деревьям при поиске (имеется ввиду алгоритм бинарного поиска), они сильно проигрывают им, когда дело доходит до вставки и удаления значений. Дело в том, что вставка в упорядоченном массиве занимает O(N) времени, потому что помимо поиска мы должны сдвинуть много данных вправо (чтобы освободить место для вставляемого значения) или влево (чтобы занять место удаляемого).

Конечно можно упорядоченную структуру (например, по возрастанию) организовать не как массив, а как список, тем самым сократив время на реализацию удаления или вставки элемента. Однако для связного списка операции поиска элемента происходят за O(N), то есть существенно медленнее, чем у бинарного дерева поиска. А именно операции поиска элемента относятся к наиболее востребованным при управлении данными.

> Следует отметить, что в Python встроенный тип данных list реализован как как массив динамической длины. Не как односвязный или двусвязный список! Это значит, что он представляет собой непрерывный участок памяти, где элементы хранятся последовательно.
>
> На самом деле в списке хранятся только ссылки на объекты, даже если это просто целые числа. Такой подход обеспечивает быстрый доступ к элементам по индексу - O(1). Только вставка или удаление элементов в таком списке может быть неэффективной с асимптотикой O(n), так как требуется сдвиг элементов.

Иной вариант структуры для работы с данными – Хэш-таблицы, которые являются мощным инструментом для хранения и быстрого поиска данных. Но при использовании больших объемов данных использование такой структуры данных может столкнуться с несколько недостатками:

1. Коллизии. При большом объеме данных вероятность коллизий (двух ключей, которые имеют одинаковое хэш-значение) возрастает.

2. Избыточные расходы на память. Хэш-таблицы могут занимать много памяти, особенно если они сильно разрежены. Они резервируют дополнительную память, большую чем необходимо для хранения элементов, чтобы обеспечить достаточное пространство для вставки новых элементов.

3. Расходы на реконфигурацию. При заполнении таблицы срабатывает механизм перерасчёта хэш-значений, для создания новой, более крупной таблицы с затратами на копирование в неё старых данных, что неэффективно при работе с большими объемами данных.

4. Неупорядоченность. Хэш-таблицы не сохраняют порядок элементов. Если требуется доступ к данным в определенном порядке, придется использовать дополнительные структуры данных, что усложняет реализацию.

5. Сложности с изменением данных. Если данные в таблице часто изменяются, это может привести к дополнительным накладным расходам на перерасчет хэш-значений и их повторное распределение в памяти.

Таким образом, во многих случаях такая структура данных как двоичное дерево поиска может быть максимально эффективна, особенно при поиске и обработке данных больших объёмов.

= = =

Пусть требуется написать класс для реализации бинарного дерева поиска. Бинарное дерево состоит из узлов, каждый узел – это объект с тремя полями: хранимое значение (и одновременно ключ) и два поля-потомка (left и right, в левом – ссылка на объект с меньшим значением, справа – с большим) (рис.1). Для хранения отдельного узла создадим дополнительный класс, тогда дерево – это родительский узел (root), в котором хранятся ссылки на двух прямых потомков, в которых, в свою очередь, хранятся ссылки на их потомков и т.д. Если у очередного узла нет потомков, то на их место можно поставить None (пустая ссылка в Python или null в других языках программирования). Таким образом каждый узел

в дереве всегда будет иметь три поля: ключевое (само хранимое значение), и два потомка, один или оба из которых могут содержать None.

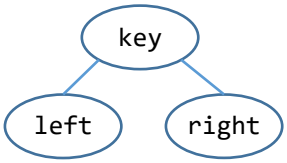

Рисунок 1. Графическое представление узла бинарного дерева.

Поиск в дереве осуществляется так же быстро, как и бинарный поиск в отсортированном массиве, так как после каждого сравнения мы отсекаем сразу половину неподходящих результатов. В итоге поиск во всём дереве будет проходить не более чем за $\log_2(N)$ шагов, при условии, что дерево было сбалансировано.

Пусть у нас будет дерево из трёх элементов: 3, 2, 8, тогда эти значения могут быть размещены так как указано на рисунке 2.

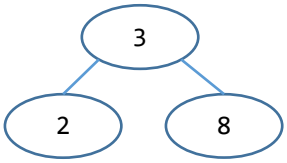

Рисунок 2. Пример сбалансированного дерева из трёх значений.

В этом случае поиск будет проходить не более чем за два шага. Поиск начинается с корневого узла (root). В данном случае это узел 3. Предположим мы ищем элемент 8. Первая проверка покажет, что разыскиваемый элемент больше корневого узла, тогда продолжаем поиск в правом поддереве. Следующая проверка покажет, что искомый элемент равен значению в узле. Таким образом мы нашли узел не более чем за два шага. Если бы пришлось искать 7, то на первом шаге мы также сдвинулись в правое поддерево, а на втором пришлось бы переходить в поддерево левее, чем 8. Напомню, что каждый наш узел – это объект с тремя полями, в которых кроме самого хранимого значения располагаются ссылки на левый узел и на правый (Листинг 1). В данном случае у узла 8 (это значение хранится в поле key), в качестве левого и правого полей хранятся пустые ссылки None. В случае поиска это означает, что значения 7 нет в исследуемом дереве.

Листинг 1. Пример класса Node.

```
class Node:
    def __init__(self, value):
        self.key = value  # вершина - родитель
        self.left = None
        self.right = None
```

Имея в наличии только один класс Node уже можно имитировать работу бинарного дерева поиска. На основе класса Node создадим несколько узлов с целочисленными значениями [3, 4, 5, 6, 8], обозначим вершину дерева (root, значение 5) и добавим к ней остальные узлы, соблюдая правило – меньшие в левое поддерево, большие – в правое (Листинг 2).

Листинг 2. Реализация хранения значений в дереве.

```
class Node:
    def __init__(self, value):
        self.key = value  # вершина - родитель
        self.left = None
        self.right = None

# создаём все узлы
node5 = Node(5)  # root
node8 = Node(8)
node3 = Node(3)
node4 = Node(4)
node6 = Node(6)

# добавляем узлы в дерево
node5.left = node3
node3.right = node4
node5.right = node6
node6.right = node8
```

При такой организации программы уже можно осуществлять поиск в дереве, всегда начиная поиск с корня (root). Прежде чем проверять результативность поиска в дереве, добавим в класс Node метод __repr__ для приведения объекта Node к строке. Это пригодиться при выводе на печать (на экран) найденного значения (или None при его отсутствии). Затем создадим рекурсивную функцию поиска, которая будет переходить к поиску в левом или правом поддереве в зависимости от сравнения разыскиваемого элемента с ключевым (Листинг 3).

Листинг 3. Реализация поиска значений в дереве.

```
class Node:
    def __init__(self, value):
        self.key = value  # parent
        self.left = None
        self.right = None

    def __repr__(self):
        return f"key = {self.key}"

def search(node, value):
    if node:
        if value == node.key:
            return node
        if value < node.key:
            return search(node.left, value)
        else:
            return search(node.right, value)
    else:
        return None

node5 = Node(5)  # root
node8 = Node(8)
node3 = Node(3)
node4 = Node(4)
node6 = Node(6)

node5.left = node3
node3.right = node4
node5.right = node6
node6.right = node8

root = node5  # назначаем корень
# затем ищем элементы в дереве
print(search(root, 7))  # None
print(search(root, 4))  # key = 4
print(search(root, 6))  # key = 6
```

Апробируйте работу программы, убедитесь, что проверка условия if node, работает корректно даже для нулевых значений (даже если узел хранит значение 0, то узел - это же целый непустой объект, поэтому проверка if node должна срабатывать корректно).

Теперь уже можно будет перенести логику работы поиска в класс BST (Binary Search Tree), далее класс Node уже не будем повторять, так как он остаётся без изменений (Листинг 4). В классе BST будет два метода

поиска search и _search. Первый метод нужен, чтобы указать, что поиск всегда начинается с вершины дерева.

Кроме реализации метода поиска в классе BST должен быть реализован метод добавления нового узла. По аналогии сделаем реализацию с двумя методами: insert и _insert, где первый нужен, чтобы обозначить, что рекурсивная процедура добавления должна начинаться всегда с вершины дерева. В случае, когда дерево ещё пустое, то первый же добавляемый элемент устанавливаем на вершину дерева (Листинг 4).

Листинг 4. Реализация методов поиска и добавления элементов.

```
class BST:
    def __init__(self):  # конструктор
        self.root = None

    def search(self, value):
        return self._search(self.root, value)

    def _search(self, node, value):
        if node:
            if value == node.key:
                return node
            if value < node.key:
                return self._search(node.left, value)
            else:
                return self._search(node.right, value)
        else:
            return node

    def insert(self, value):
        if self.root:  # добавляем в НЕ пустое дерево
            self._insert(self.root, value)
        else:
            self.root = Node(value)

    def _insert(self, node, value):
        if value < node.key:  # сравниваем с вершиной
            if node.left:
                self._insert(node.left, value)
            else:
                node.left = Node(value)
        else:
            if node.right:
                self._insert(node.right, value)
            else:
                node.right = Node(value)
```

Метод _insert осуществляет рекурсивный спуск по дереву, начиная от корня, в поисках подходящего месторасположения узла для значения value. Если value < node.key , то продолжаем поиск места вставки в левом

поддереве, а иначе в правом. Прежде чем добавить узел, сначала проверяем потомка (левого или правого) на наличие объекта узла. Если его нет, то там и располагаем новый объект на основе класса Node, если же есть, то продолжаем рекурсивный поиск места для вставки.

Для проверки работоспособности дерева, можно подготовить исходный список элементов и циклом добавить его в дерево (Листинг 5).

```
lst = [5, 3, 8, 6, 2, 4, 9]

tree = BST()  # создаём новый объект дерева
for elm in lst:  # добавляем в него все элементы из списка
    tree.insert(elm)

print(tree.root)  # для контроля - корень

for elm in lst:  # проверка поиска всех элементов
    print(elm, tree.search(elm))

# Вывод на экран для этой программы будет таким:
# key = 5
# 5 key = 5
# 3 key = 3
# 8 key = 8
# 6 key = 6
# 2 key = 2
# 4 key = 4
# 9 key = 9
```

Результаты работы программы демонстрируют работоспособность методов добавления в дерево и поиска в нём элементов. И добавление, и поиск происходят рекурсивно с асимптотикой $\log_2(N)$. На самом деле такой результат получился только потому, что было создано сбалансированное дерево (рис. 3).

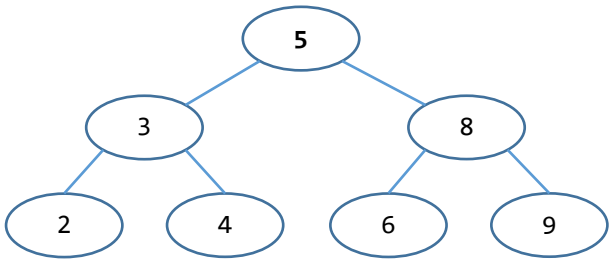

Рисунок 3. Пример сбалансированного дерева из семи значений.

В списке очерёдность значений была подобрана заранее так, чтобы при добавлении получилось именно сбалансированное дерево. Если бы, к примеру, значения в списке располагались изначально по возрастанию [2, 3, 4, 5, 6, 8, 9], то получилось бы вырожденное дерево (рис. 4).

Очевидно, что только для сбалансированного дерева операции добавления и поиска элементов работают за время $log_2(N)$. В вырожденном времени поиск будет занимать линейное время $O(N)$.

Для реализации сбалансированного дерева можно для начала отсортировать исходный список за время $N*log_2(N)$, например, алгоритмом быстрой сортировки или сортировки кучей, а затем использовать метод, который будет брать значения из отсортированного списка в следующем порядке: найти середину оставшегося для добавления списка и добавить в дерево этот элемент, затем рекурсивно продолжить по аналогии в левой части отсортированного списка и в правой.

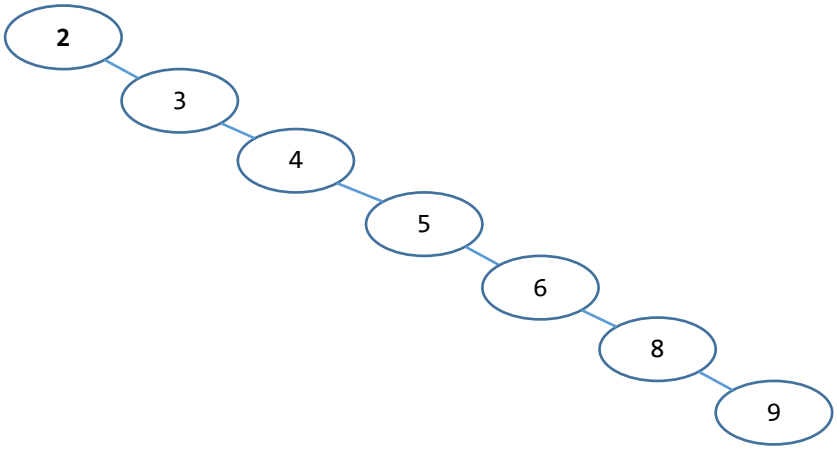

Рисунок 4. Пример вырожденного дерева.

Опыт показывает, что, если элементы, добавляемые в дерево распределены случайным образом (то есть не отсортированы заранее), дают дерево близкое к сбалансированному (в среднем с глубиной поиска в два раза больше, чем у сбалансированного, то есть $2*log_2(N)$). Даже при таком двукратном увеличении глубины скорость операций поиска и добавления всё равно заметно превышает скорость линейного поиска, например, при N=1_000_000, количество шагов линейного поиска – 1_000_000, в сбалансированном дереве – 20, а в дереве, полученном случайным образом - 40. Типичный сценарий использования дерева поиска подразумевает получение данных в случайном порядке, поэтому пока обойдёмся без метода получения сбалансированного дерева.