Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Пермский государственный аграрно-технологический университет
имени академика Д.Н. Прянишникова»

Кафедра информационных технологий и
программной инженерии

# ПРОЕКТИРОВАНИЕ ЭФФЕКТИВНЫХ СТРУКТУР ДАННЫХ

МЕТОДИЧЕСКОЕ ПОСОБИЕ

Беляков А.Ю., Пучкова М.П.

Пермь, 2024

**СОДЕРЖАНИЕ**

# ВВЕДЕНИЕ

На сегодняшний день в информационных системах и технологиях наблюдается тенденция к многократному увеличению объемов хранимых данных. Наращивание объемов данных обусловлено широким распространением информационных технологий в бизнесе и быту, массовым использованием датчиков мониторинга, диагностирования и управления различными объектами.

Методы обработки включают важный этап поиска и изменения данных. Скорость выполнения этих операций определяет эффективность всей системы обработки. Поэтому теоретические исследования и практические разработки в области ускорения поиска необходимых данных для интеллектуального анализа в различных базах данных являются актуальными.

Алгоритмы, которые могут быть реализованы при разработке классов двоичного дерева, потенциально могут обеспечивать достаточную эффективность использования аппаратных ресурсов проектируемой информационной системы.

# 1. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО КЛАССА ДЛЯ РЕАЛИЗАЦИИ ДВОИЧНОГО ДЕРЕВА ПОИСКА

## 1.1. Структура BST

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 необходимо 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 [1]. Дело в том, что вставка в упорядоченном массиве занимает O(N) времени, потому что помимо поиска необходимо сдвинуть много данных вправо (чтобы освободить место для вставляемого значения) или влево (чтобы занять место удаляемого).

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 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.

## 1.2. Алгоритмы работы дерева поиска

Пусть требуется написать класс для реализации бинарного дерева поиска. Бинарное дерево состоит из узлов, каждый узел – это объект минимум 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) (рисунок 1). Для хранения отдельного узла создается 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 пустую ссылку None (если класс разрабатывается на языке Python или null в других языках программирования) [2]. 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.

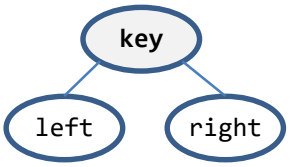


Рисунок 1. Графическое представление узла бинарного дерева

Поиск в дереве осуществляется так же быстро, как и бинарный поиск в отсортированном массиве, так как после каждого сравнения отсекается сразу половина 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 будет дерево из трёх элементов: 3, 2, 8, тогда эти значения могут быть размещены так, как указано на рисунке 2.

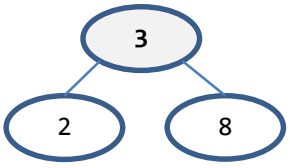


Рисунок 2. Пример сбалансированного дерева из трёх значений

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, что нужно найти элемент 8.

Первая проверка покажет, что разыскиваемый элемент больше корневого узла, тогда необходимо продолжить поиск в правом поддереве. Следующая проверка покажет, что искомый элемент равен значению в узле. Таким образом, узел определен не более чем за два шага.

При поиске числа 7 на первом шаге происходит сдвиг в правое поддерево, а на втором шаге необходим переход в поддерево левее, чем 8. При этом каждый узел – это объект с тремя полями, в которых, кроме самого хранимого значения, располагаются ссылки на левый узел и на правый (рисунок 3).

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

Рисунок 3. Листинг 1. Пример класса Node

В данном случае у узла 8 (значение хранится в поле key), в качестве левого и правого полей хранятся пустые ссылки None (рисунок 4). В случае поиска это означает, что значения 7 нет в исследуемом дереве.

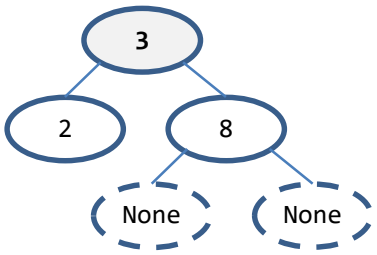


Рисунок 4. «Листочки» бинарного дерева поиска

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 (рисунок 5).

```
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
```

Рисунок 5. Листинг 2. Реализация хранения значений в дереве

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 пригодится 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 (рисунок 6).

При апробации работы программы, можно убедиться, что проверка условия if node работает корректно даже для нулевых значений (даже если узел хранит значение 0, то узел - это целый непустой объект, поэтому

проверка if node должна срабатывать корректно, аналогично проверке – if is not None).

```
    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:  # node != None
            if value == node.key:
                return node
            if value < node.key:
                return search(node.left, value)  # продолжаем в
левом поддереве
            else:
                return search(node.right, value)  # продолжаем в
правом поддереве
        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
```

Рисунок 6. Реализация поиска значений в дереве

В классе BST необходимо использовать два метода поиска search и

_search. Первый метод нужен, чтобы указать, что поиск всегда начинается с вершины дерева. Это освобождает пользователя этого класса от необходимости явно указывать, что поиск следует начинать с корня дерева. В данном случае сам класс будет обеспечивать корректное начало поиска.

**Выводы**

Производительность программных систем зависит не только от используемых алгоритмов, но и от структур данных, с которыми эти алгоритмы работают. Выбор структуры данных, в первую очередь, зависит от конкретных требований к производительности. В различных структурах данных отдельные операции могут значительно различаться и выполняться быстрее, другие медленнее.

Определённые особенности организация хранения данных позволяют повысить эффективность операций, используемых для обработки данных, таких как добавление, удаление и изменений элементов. Например, если требуется часто добавлять/удалять элементы, может быть целесообразнее использовать связный список или хэш-таблицу.

Кроме этого, важно учитывать пространственные характеристики выбранной структуры, дополнительные расходы на хранение элементов и способов доступа к ним. Избыточное использование памяти, в ряде случаев, может является критически значимым фактором. Например, связные списки требуют больше памяти из-за хранения указателей, в то время как массивы могут выделять память неэффективно, если их размер меняется. При этом, массивы имеют лучшие характеристики кэширования из-за своей особенности использования последовательных блоков памяти, что может увеличить производительность по сравнению с более сложными структурами данных.

На практике выбор структуры данных часто зависит от требований конкретной задачи. Например, для приложения, где требуются быстрые операции поиска и добавления, следует использовать хэш-таблицы. Если необходима строгая последовательность и упорядоченный доступ, предпочтительнее использовать деревья.

# 2. АЛГОРИТМЫ С ЭЛЕМЕНТАМИ ДЕРЕВА

## 2.1. Алгоритм добавления элемента в дерево

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, первый же добавляемый элемент устанавливаем на вершину дерева (рисунок 7).

```
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)  # если дерево пока пустое

    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)
```

Рисунок 7. Листинг 4. Реализация методов поиска и добавления элементов

Метод _insert осуществляет рекурсивный спуск по дереву, начиная от корня, в поисках подходящего места для расположения узла со значением 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 (рисунок 8).

```
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
```

Рисунок 8. Листинг 5. Проверка работы методов вставки и поиска элементов

### 2.2. Сбалансированность дерева поиска

Результаты работы программы демонстрируют работоспособность методов добавления узлов 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 (рисунок 9).

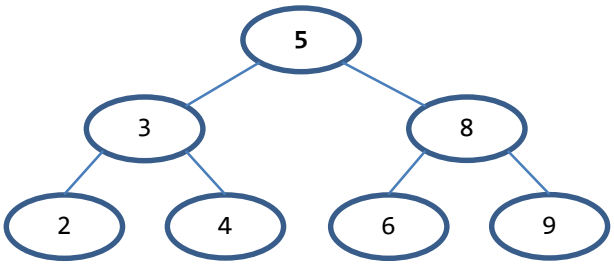


Рисунок 9. Пример сбалансированного дерева из семи значений

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 (рисунок 10).

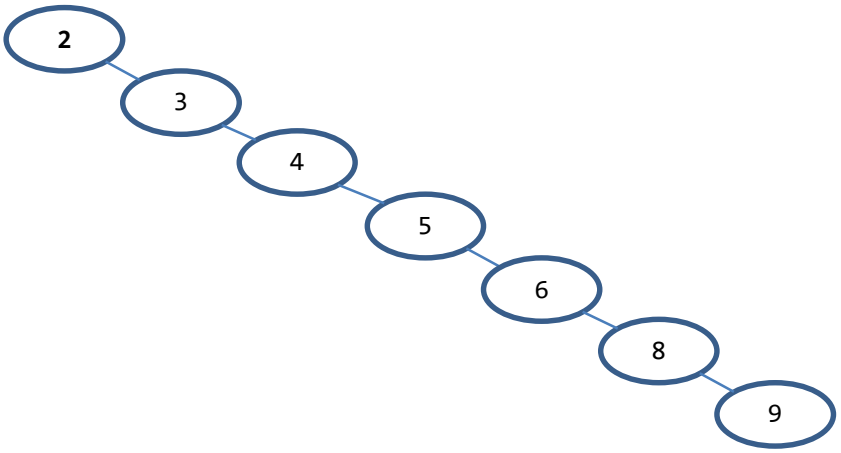


Рисунок 10. Пример вырожденного дерева

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).

Опыт показывает, что, если элементы, добавляемые в дерево распределены случайным образом (то есть, не отсортированы заранее), то это дает 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 метод получения сбалансированного дерева не является обязательным.

Тем не менее, для ряда случаев метод получения сбалансированного дерева может иметь значение, поэтому рассмотрим его алгоритм. Так как мы не будем добавлять его в разрабатываемый класс дерева, то оформим алгоритм в виде функции без синтаксической привязки к классу (просто как пример для изучения).

Для реализации сбалансированного дерева можно использовать простой подход сборки дерева из заранее отсортированной коллекции:

1) Сортировка исходного списка за время $N*\log_2(N)$;

2) Выбор среднего значения из отсортированного списка и добавление его на вершину дерева (root);

3) Рекурсивный выбор срединного элемента в оставшейся части слева и справа (найденный элемент слева становится левым потомком, а справа - правым).

После окончания рекурсивного перебора глубиной $\log_2(N)$ будет получен исходный узел (root) с одинаковым количеством потомков на левой ветви и на правой (рисунок 11).

```
    def get_balance_bst(arr):
        if not arr: return None
        mid = len(arr) // 2  # находим средний по значению элемент
        node = Node(arr[mid])  # ставим его на вершину дерева
        node.left  = get_balance_bst(arr[:mid])    # левое поддерево
        node.right = get_balance_bst(arr[mid+1:])  # правое
поддерево
        return node  # сбалансированное дерево
```

Рисунок 11. Листинг 6. Рекурсивный метод построения сбалансированного дерева

Чтобы наглядно представить результаты размещения узлов в дереве и оценить его сбалансированность, добавим в программу функцию (без

добавления в класс) печати бинарного дерева в терминале, где глубина погружения в дерево будет отмечаться заданным количеством пробелов (рисунок 12).

```
    def print_bst(node, deep=0):
        if node:  # если узел существует
            print_bst(node.right, deep+1)  # печатаем его правое
поддерево
            print(f"{4*' '*deep}{node.value}")  # печатаем родителя
узла
            print_bst(node.left, deep+1)  # печатаем его левое
поддерево
```

Рисунок 12. Листинг 7. Рекурсивный метод печати дерева поиска

При первоначальном запуске функции print_bst назначается глубина погружения в дерево равной нулю. При каждом рекурсивном вызове этот параметр будет увеличиваться на 1, но сам уровень при выводе на экран для наглядности будет отстоять от предыдущего на четыре пробела.

Чтобы проверить работоспособность функций формирования и вывода сбалансированного дерева создадим небольшую программу, которая будет подготавливать отсортированную коллекцию случайных чисел, чтобы создать ситуацию, провоцирующую на создание вырожденного дерева (рисунок 13).

```
    arr = [7, 2, 5, 4, 3, 6, 1]  # исходный несортированный массив
    # отсортируем, чтобы создать худшую
    # ситуацию для формирования дерева
    arr = sorted(arr)  # [1, 2, 3, 4, 5, 6, 7]
    bst = get_balance_bst(arr)
    print_bst(bst)
    # результаты работы программы:
            7
        6
            5
    4
            3
        2
            1
```

Рисунок 13. Листинг 8. Программа проверки вывода сбалансированного дерева

При выводе на печать получим дерево с вершиной 4 и равномерно

нагруженными ветвями. Для удобства вывода на экран дерево наклонено на 90 против часовой стрелки.

### 2.2. Алгоритм удаления элемента из дерева

Приведем краткое описание алгоритма удаления элемента. Сначала на вход подаётся значение удаляемого элемента, по которому, начиная с вершины дерева рекурсивно можно дойти до узла с искомым элементом. После нахождения удаляемого узла требуется принять решение – какое значение ставить на его место после удаления чтобы не нарушить основное правило дерева – все потомки слева меньше родителя, а справа – больше. Возможны следующие варианты удаляемого узла:

1. Узел является листом дерева, то есть у него нет потомков. Тогда вместо удаляемого узла просто ставим пустую ссылку None.
2. У узла существует только один потомок. Тогда вместо удаляемого узла ставим ссылку на этого потомка.
3. У узла два потомка. В этом случае требуется осуществить рекурсивный поиск такого узла (либо в левом поддереве, либо в правом), хранимое значение которого, при постановке вместо удаляемого окажется точно больше всех его потомков слева и точно меньше всех его потомков справа. В нашем случае рассматривается метод с реализацией поиска минимального узла в правом поддереве.

Традиционно для выполнения операции удаления создадим два метода в классе BST: первый (delete) будет начинать с корня дерева, а второй (_delete) – рекурсивно искать удаляемый узел и удалять его, заменяя подходящим объектом (другим узлом или пустой ссылкой None) (рисунок 14).

В начальной части метода _delete проверяется случай, когда у узла не оказалось потомка, тогда на место удаляемого узла возвращаем пустую ссылку None – это очевидный и самый простой случай.

```
    def delete(self, value):
        self.root = self._delete(self.root, value)

    def _delete(self, node, value):
        if node is None:  # если потомка нет
            return None

        if value < node.key:  # если меньше узла
            node.left = self._delete(node.left, value)
        elif value > node.key:  # если больше узла
            node.right = self._delete(node.right, value)
        else:  # нашли удаляемый узел
            # теперь нужно в нём поменять ссылку

            # если потомок только один или их нет
            if node.left is None:  # если левого потомка нет
                return node.right  # то возвращаем правый
            elif node.right is None:  # если правого потомка нет
                return node.left  # то возвращаем левый

            # если оба потомка существуют, то удаляемый узел
            # заменим на минимальный из правого поддерева
            min_node = self._min_node(node.right)  # найти минимальный
справа
            node.key = min_node.key  # записать его на место удаляемого
            node.right = self._delete(node.right, min_node.key)  #
продолжить

        return node
```

Рисунок 14. Листинг 9. Организация удаления узла из дерева поиска

Далее по методу, если узел не пустой, то определяем - разыскиваемая величина меньше или больше значения, хранимого в текущем узле. В зависимости от этого рекурсивно продолжаем поиск удаляемого элемента в левом или правом поддереве.

Если же искомый элемент оказался в текущем узле, то требуется понять, что именно ставить на его место после удаления. Это зависит от его потомков. Если у удаляемого узла существует только один потомок, то его и переставляем на место удаляемого (возвращаем ссылку на этого существующего потомка). Если потомка два (или ни одного), то запускаем метод поиска в правом поддереве минимального элемента (рисунок 15). Это гарантирует сохранность корректности дерева после перемещения минимального из правого поддерева на место удаляемого. Все в левом поддереве будут меньше него, а все в правом – больше.

```
    def _min_node(self, node):
        current = node  # текущий узел - минимальный
        while current.left:  # если потомок слева существует
            current = current.left  # он назначается новым минимальным
        return current  # возвращаем минимальный узел
```

Рисунок 15. Листинг 10. Метод поиска минимального элемента

Рассмотрим пример удаления узла 5 из бинарного дерева, состоящего из семи узлов (рисунок 16).

На первом этапе работает рекурсивный поиск удаляемого элемента self._delete(self.root, value), начиная с корня дерева. В данном случае уже на первом шаге рекурсии находим удаляемый узел - 5. Далее запускаем поиск минимального элемента в правом поддереве self._min_node(node.right), то есть начиная с правого потомка у узла 5 (это узел 8). Далее в методе _min_node в цикле (while current.left) для каждого узла выбираем его меньшего (левого) потомка до тех пор, пока находится не пустой узел.

Найденный последний не пустой узел хранит значение 6, и ссылка на этот узел присваивается в переменную min_node. Далее значение key удаляемого узла заменяется значением найденного минимального node.key = min_node.key.

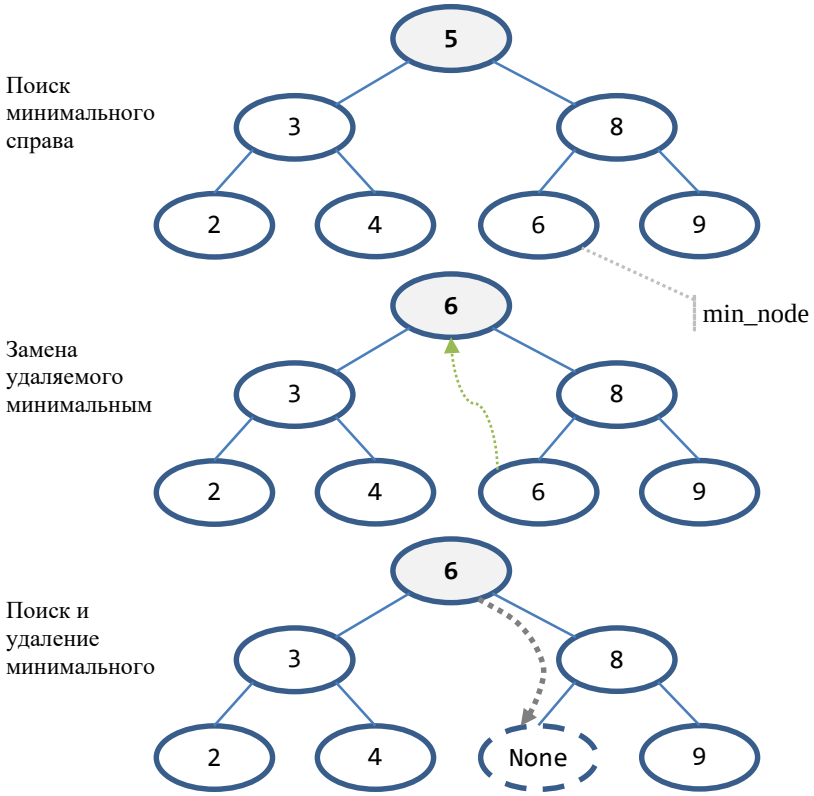


Рисунок 16. Этапы изменения дерева при удалении узла 5

После описанных выше шагов алгоритма значение 6 хранится в двух узлах – на месте удалённого узла и на своём первоначальном месте. Для поиска этого первоначального расположения переставленного узла запускаем рекурсию self._delete(node.right, min_node.key) с параметрами – начать с правого потомка текущего удаляемого узла (это 8) и искать для удаления значение перемещённого узла (это 6).

Запускается рекурсия поиска удаляемого узла аналогичная описанному ранее (только теперь от узла 8 ищем узел 6). Когда найдётся узел со значением 6, то окажется, что у него нет ни одного потомка, и, в этом случае, он заменится на None (рис.6, третий этап). Корректность дерева восстановлена.

Теперь, после добавления всех методов в класс, можно проверить корректность их работы в целом. Добавим в дерево коллекцию [5, 3, 8, 6, 2, 4, 9], удалим корень 5 можно распечатать полученное дерево (рисунок 17).

```
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)

tree.print_tree()
tree.delete(5)
tree.print_tree()
```

Результат вывода исходного дерева:

```
        9
    8
        6
5
        4
    3
        2
```

Рисунок 17. Листинг 11. Метод поиска минимального элемента и результат вывода

Реализация вывода на печать структуры дерева игнорирует несуществующие узлы (None). Если это имеет значение (при увеличении объёма вывода на экран), то можно добавить в метод вывода соответствующую строку (рисунок 18).

```
def _print_tree(self, node, deep=0):
    if node:
        self._print_tree(node.right, deep+1)
        print(f"{4*' '*deep}{node.key}")
        self._print_tree(node.left, deep+1)
    else:
        print(f"{4*' '*deep}None")
```

Рисунок 18. Листинг 12. Вывод полного дерева, включающего пустые узлы

### 2.3 Алгоритм получения упорядоченной коллекции узлов

В некоторых случаях требуется получить из структуры дерева отсортированную коллекцию хранимых в нём элементов. Ввиду того, что элементы хранятся в дереве упорядоченно (иерархия, левые потомки меньше, правые больше), то получить их в определённом порядке не представляется сложной задачей.

Организуем получение упорядоченной коллекции узлов дерева по аналогии с поиском и добавлением в виде двух методов – первый (get_sorted) используется для запуска с корня дерева, а второй (_get_sorted) для рекурсивного перебора структуры данных (рисунок 19).

```
    def get_sorted(self):
        return self._get_sorted(self.root)

    def _get_sorted(self, node):
        srt = []  # сюда будем добавлять элементы в порядке возрастания
        if node:
            srt = self._get_sorted(node.left)  # сначала меньшие
            srt += [node.key]  # срединный элемент - родитель
            srt += self._get_sorted(node.right)  # потом большие
        return srt
```

Рисунок 19. Листинг 13. Организация получения упорядоченной коллекции узлов

На каждом шаге сначала формируется список из меньших элементов из левого поддерева, потом добавляется текущий родительский элемент шага (он срединный), потом добавляются большие элементы из правого поддерева. После выполнения всех рекурсивных вызовов возвращается отсортированный список значений, хранящихся в узлах дерева.

**Выводы**

Бинарное дерево поиска - это структура данных, которая позволяет эффективно хранить и выполнять операции с определённым способом упорядоченными данными. Построение класса для реализации бинарного дерева зависит от накладываемых на структуру данных задач. Как правило, требуется конструктор класса для инициализации объекта бинарного дерева, метод добавления элементов и метод поиска отдельного элемента по заданной характеристике.

Кроме того, для решения практических задач могут потребоваться методы удаления элемента, подсчёта количества элементов, обхода элементов в заданном порядке.

Все значимые методы бинарного дерева могут быть построены с помощью рекурсивного подхода. При этом среднее время реализации метода поиска имеет асимптотику сложности O(log n), что особенно полезно при работе с большими объемами данных.

Операции вставки и удаления также выполняются за O(log n) в сбалансированных деревьях, что делает бинарное дерево эффективным для динамически изменяющихся наборов данных. Однако, при значительном количестве изменений в данных требуются дополнительные подходы для поддержания бинарного дерева в сбалансированном состоянии.

Вследствие эффективности большинства операций работы с данными бинарное дерево поиска используют в системах управления базами данных для индексации полей таблиц, чтобы ускорять выполнение запросов к данным. Ввиду универсальности подходов работы с данными бинарные деревья поиска могут применяться в игровых движках для различных алгоритмов и в рамках решения задач искусственного интеллекта.

Тем не менее, важно помнить, что бинарные деревья без операций балансировки со временем вырождаются, что приводит к ухудшению времени выполнения операций работы с данными вплоть до линейного - O(n). Поэтому часто используются сбалансированные версии деревьев, такие как

AVL-деревья или красно-черные деревья, которые обеспечивают более предсказуемую производительность.

# ЗАКЛЮЧЕНИЕ

Таким образом, бинарное дерево поиска обеспечивает эффективные операции поиска, добавления и удаления элементов, что делает его полезным для реализации различных структур и баз данных, и алгоритмов, которые требуют быстрой сортировки и поиска. В отличие от других типовых структур данных (массив, список, множество) все необходимые операции выполняются с хорошей асимптотикой $N*\log_2(N)$.

Для ряда задач может потребоваться хранение элементов в виде сбалансированного дерева для обеспечения устойчивости производительности в худших случаях, что помогает избежать деградации до линейного времени обработки данных.

При значительном количестве операций добавления и удаления элементов в бинарном дереве поиска может произойти потеря сбалансированности дерева, что приведет к ухудшению производительности операций поиска, добавления и удаления. В худшем случае форма дерева может приблизиться к линейной, что делает время выполнения этих операций линейным, вместо ожидаемого логарифмического.

Чтобы избежать подобных проблем, могут потребоваться дополнительные усилия по восстановлению сбалансированности. Существует несколько структур данных, которые автоматически поддерживают сбалансированность при добавлении и удалении элементов (AVL-деревья, Красно-чёрные деревья, Splay-деревья).

Если используется обычное бинарное дерево поиска без самоподдерживающих механизмов, может потребоваться реализация дополнительных процедур для его балансировки или выбор одной из вышеупомянутых структур данных для обеспечения более эффективной работы в условиях динамических изменений данных.

## ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ИСПОЛНЕНИЯ

**1. Выберите** для себя язык программирования, например, Go, Java, C#, Python, JavaScript.

**2. Спроектируйте** класс, реализующий бинарное дерево (Binary Search Tree - BST). При необходимости пользуйтесь помощью LLM.

Требования к **функционалу** – в классе должны быть реализованы методы из следующего списка (не обязательно все, но **большинство** из перечисленных – на ваше усмотрение):

- вставки элемента;
- поиска элемента;
- получения упорядоченной коллекции узлов дерева;
- удаления элемента;
- проверки дерева на пустоту;
- поиска минимального элемента дерева;
- поиска максимального элемента дерева;
- получения высоты дерева;
- подсчёта количества узлов;
- очистки дерева (удалить все узлы).

По вашему усмотрению можно расширить указанный функционал.

**3. Реализуйте** в классе структурированный **вывод дерева на печать** (в консоль) примерно в таком формате (особенности настройки вывода можете придумать сами):

```
# Root: 5
#     L--- 3
#         L--- 1
#         R--- 4
#     R--- 8
#         L--- 6
#         R--- 9
```

**В качестве результата** работы должны быть представлены:

- модуль с классом бинарного дерева;
- программа, которая использует класс бинарного дерева и демонстрирует разработанный вами функционал класса;
- скриншоты вывода на экран (в консоль) результатов различных запросов, например, получения упорядоченной коллекции узлов дерева, структурированный вывод дерева на печать, вывод максимального элемента, вывод высоты дерева и другие (на ваш выбор);
- выводы (примерно 2-3 абзаца текста) о сложности реализации исследуемой структуры данных и сравнение с другими структурами (коллекциями) языков программирования.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Венгроу Джей. Прикладные структуры данных и алгоритмы. Прокачиваем навыки. Изд-во: Питер, Серия: Библиотека программиста. - 2023. С. 512.

2. Лотт Стивен Ф., Филлипс Дасти Объектно-ориентированный Python. Изд-во: Прогресс книга, Серия: Библиотека программиста. - 2024. С. 704.

3. Меджедович Джейла, Тахирович Эмин. Алгоритмы и структуры для массивных наборов данных. Изд-во: ДМК Пресс. - 2024. С. 340.