ИКН НИТУ МИСИС
Комбинаторика и теория графов

АВЛ-деревья. Оценка высоты АВЛ-дерева

Исполнитель:
Миронов Е.А. БИВТ-23-8
(https://github.com/Valet-V0ult-de-Fur1e/combinatorics_and_graphs_repo)

Москва 2024 год

Определение

АВЛ-дерево — это самобалансирующееся бинарное дерево поиска, в котором для каждой вершины выполняется условие: разница высот левого и правого поддеревьев не может превышать 1. Это свойство называется балансировкой АВЛ-дерева. АВЛ-дерево было предложено в 1962 году российскими математиками Георгием Адельсоном-Вельским и Евгением Ландисом, в их работе была установлена основа для дальнейшего развития структуры данных с самобалансировкой.

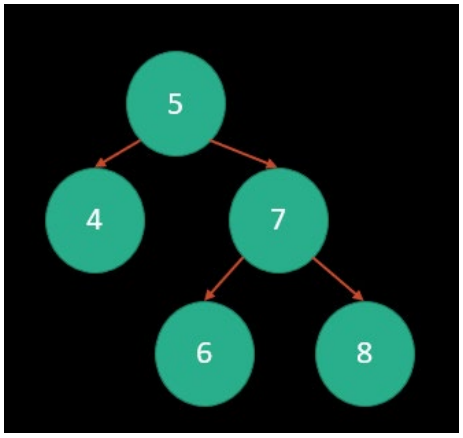

Рис. 1 Пример сбалансированного бинарного дерева

Описание концепции бинарного дерева

Бинарное дерево — это дерево, в котором каждая вершина может иметь не более двух дочерних узлов: левого и правого. В бинарном дереве поиска для каждой вершины выполнено условие: значение в левом поддереве меньше значения вершины, а в правом поддереве — больше. Это свойство бинарного дерева поиска позволяет эффективно искать, вставлять и удалять элементы.

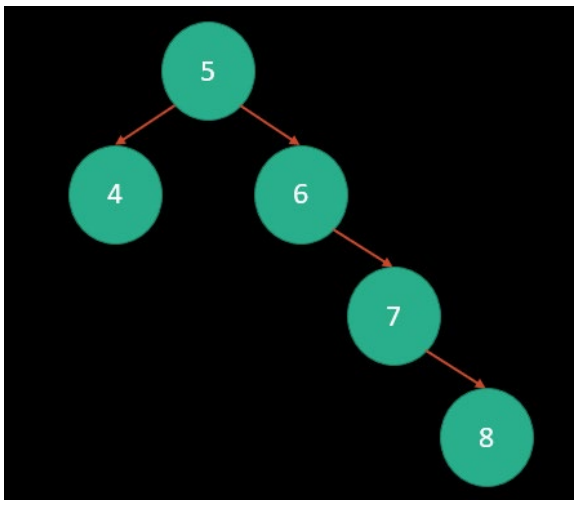

Рис. 2 Пример бинарного дерева

История создания

АВЛ-деревья были предложены в 1962 году Георгием Адельсоном-Вельским и Евгением Ландисом. Их работа основывалась на идее, что для повышения эффективности бинарных деревьев поиска необходимо учитывать баланс поддеревьев. АВЛ-дерево стало первым самобалансирующимся бинарным деревом поиска, что привело к значительному улучшению производительности операций в сравнении с обычными бинарными деревьями поиска.

Теоретическое описание

АВЛ-дерево поддерживает свойство балансировки, заключающееся в том, что разница высот левого и правого поддерева для каждой вершины не может превышать 1. Балансировка дерева обеспечивается с помощью специальных операций, называемых поворотами. Каждое дерево имеет высоту, которая рассчитывается как длина самого длинного пути от корня до листа.

Для обеспечения эффективных операций (вставка, удаление и поиск) необходимо поддерживать сбалансированность дерева. Балансировка поддерживается через повороты, которые могут быть правыми, левыми, право-левыми и лево-правыми.

Реализация поиска вершины

Для поиска вершины в АВЛ-дереве можно использовать рекурсивный или итеративный подход.

Рекурсивный подход:

1. Если дерево пусто, то вершина не найдена. Возвращаем NULL.
2. Иначе сравниваем значение текущего узла с искомым значением.
3. Если значения равны, то мы нашли вершину. Возвращаем текущий узел.
4. Если значение текущего узла меньше искомого значения, то продолжаем поиск в правом поддереве.
5. Если значение текущего узла больше искомого значения, то продолжаем поиск в левом поддереве.
6. Повторяем шаги 2–5 для каждого поддерева, пока не найдём вершину или не дойдём до пустого дерева.

Этот подход использует рекурсию для обхода дерева и сравнения значений узлов. Он может быть эффективным, если дерево не слишком глубокое. Однако при большом количестве уровней рекурсии может возникнуть переполнение стека вызовов.

Итеративный подход:

1. Создаём стек, который будет хранить узлы дерева.
2. Добавляем в стек корень дерева.
3. Пока стек не пуст, выполняем следующие действия:
    a. Извлекаем из стека текущий узел.
    b. Если текущий узел является листом, то проверяем, является ли он вершиной. Если да, то возвращаем текущий узел как найденную вершину.
    c. Иначе сравниваем значение текущего узла с искомым значением.
        i. Если значения равны, то мы нашли вершину. Возвращаем текущий узел.

ii. Если значение текущего узла меньше искомого значения, то добавляем в стек правый дочерний узел текущего узла.
iii. Если значение текущего узла больше искомого значения, то добавляем в стек левый дочерний узел текущего узла.

Итеративный подход позволяет избежать переполнения стека вызовов, но требует больше памяти для хранения стека. Кроме того, этот подход может быть менее эффективным, чем рекурсивный, при поиске вершины в дереве с большим количеством уровней.

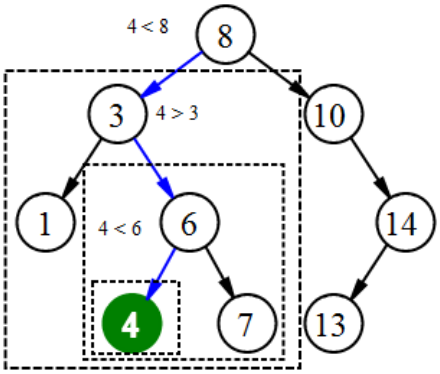

Рис. 3 Пример поиска вершины в АВЛ дереве

```python
def find_value(key:int, node:Node)->bool:
    if key == node.key:
        return True
    if node.height == 1:
        return key == node.key
    return find_value(key, node.left) if node.key > key else find_value(key, node.right)
```

Рис. 4 Код реализации поиска вершины на python 3.9

Реализация вставки вершины

Алгоритм вставки элемента в АВЛ-дерево:

1. Если дерево пусто, то создаём новый узел с данным элементом и возвращаем его как корень дерева.
2. Иначе сравниваем значение текущего узла с вставляемым значением.
3. Если значения равны, то ничего не делаем.
4. Если значение текущего узла меньше вставляемого значения, то продолжаем поиск в правом поддереве.
5. Если значение текущего узла больше вставляемого значения, то продолжаем поиск в левом поддереве.
6. Повторяем шаги 2–5 для каждого поддерева, пока не найдём место для вставки или не дойдём до пустого дерева.
7. Вставляем новый узел в найденное место.
8. Обновляем высоту узлов на пути от места вставки до корня дерева. Это необходимо для поддержания баланса дерева после вставки нового узла.
9. Возвращаем корень обновлённого дерева.

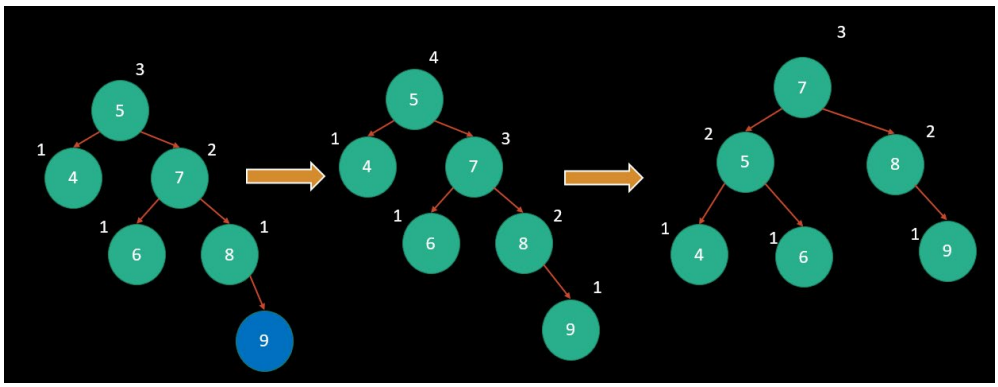

Рис. 5 Пример добавления вершины в АВЛ дереве

```python
def insert(key:int, node:Node=None)->Node:
    if node is None:
        return Node(key)

    if key < node.key:
        node.left = insert(key, node.left)
    elif key > node.key:
        node.right = insert(key, node.right)
    else:
        return node

    node.height = 1 + max(height(node.left), height(node.right))
    balance = get_balance(node)

    if balance > 1 and key < node.left.key:
        return right_rotate(node)
    if balance < -1 and key > node.right.key:
        return left_rotate(node)
    if balance > 1 and key > node.left.key:
        node.left = left_rotate(node.left)
        return right_rotate(node)
    if balance < -1 and key < node.right.key:
        node.right = right_rotate(node.right)
        return left_rotate(node)
    return node
```

Рис. 6 Код реализации добавления вершины на python 3.9

Реализация удаления вершины

Алгоритм удаления элемента из АВЛ-дерева:

1. Если дерево пусто, то ничего не делаем.
2. Иначе сравниваем значение текущего узла с удаляемым значением.
3. Если значения равны, то удаляем текущий узел и возвращаем его как результат операции.
4. Если значение текущего узла меньше удаляемого значения, то продолжаем поиск в правом поддереве.
5. Если значение текущего узла больше удаляемого значения, то продолжаем поиск в левом поддереве.

6. Повторяем шаги 2–5 для каждого поддерева, пока не найдём узел с искомым значением или не дойдём до пустого дерева.
7. Удаляем найденный узел.
8. Обновляем высоту узлов на пути от места удаления до корня дерева. Это необходимо для поддержания баланса дерева после удаления узла.
9. Возвращаем корень обновлённого дерева.

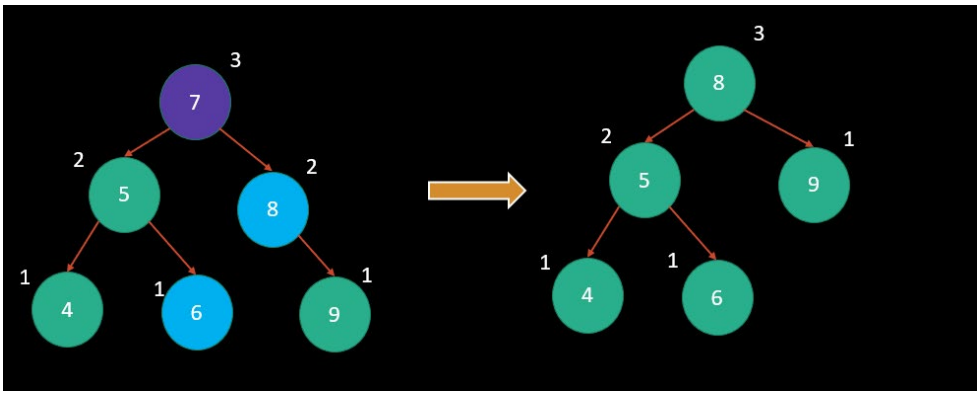

Рис. 7 Пример удаления вершины в АВЛ дереве

```python
def delete(key:int, node:Node=None)->Node:
    if not node:
        return node
    if key < node.key:
        node.left = delete(key, node.left)
    elif key > node.key:
        node.right = delete(key, node.right)
    else:
        if not node.left:
            return node.right
        elif not node.right:
            return node.left
        temp = get_min_value_node(node.right)
        node.key = temp.key
        node.right = delete(temp.key, node.right)

    node.height = 1 + max(height(node.left), height(node.right))
    balance = get_balance(node)

    if balance > 1 and get_balance(node.left) >= 0:
        return right_rotate(node)
    if balance > 1 and get_balance(node.left) < 0:
        node.left = left_rotate(node.left)
        return right_rotate(node)
    if balance < -1 and get_balance(node.right) <= 0:
        return left_rotate(node)
    if balance < -1 and get_balance(node.right) > 0:
        node.right = right_rotate(node.right)
        return left_rotate(node)
    return node

def get_min_value_node(node:Node)->str:
    current = node
    while current.left is not None:
        current = current.left
    return current
```

Рис. 8 Код реализации удаления вершины и поиска наиболее похожей на python 3.9

Асимптотики операций

Асимптотики основных операций для АВЛ-дерева:

- **Поиск**: Операция поиска элемента в АВЛ-дереве имеет сложность O(log n), так как при каждом шаге мы идем по одному из поддеревьев.
- **Вставка**: Сложность вставки также O(log n), так как после добавления элемента необходимо провести балансировку, что требует не более логарифмического времени.
- **Удаление**: Операция удаления элемента имеет сложность O(log n), поскольку после удаления может понадобиться выполнение нескольких поворотов для восстановления баланса дерева.

Доказательство асимптотик

1. **Поиск**: При поиске элемента мы начинаем с корня и переходим по одному из поддеревьев (влево или вправо), что уменьшает количество возможных путей до O(log n), так как дерево всегда остается сбалансированным.
2. **Вставка**: Вставка требует прохождения по дереву до места, где должен быть вставлен новый элемент, что занимает O(log n) времени. После этого необходимо провести не более O(log n) операций для восстановления баланса.
3. **Удаление**: При удалении элемента процесс поиска нужного элемента требует O(log n) времени, после чего может потребоваться выполнение нескольких поворотов, каждый из которых выполняется за O(1) время. Таким образом, общая сложность удаления также O(log n).

Сравнение реальных асимптотик с теоретическими

Теоретические асимптотики для операций в АВЛ-дереве основаны на предположении, что дерево всегда сбалансировано. Однако на практике, в зависимости от конкретной реализации и данных, могут возникать ситуации, когда операции занимают немного больше времени, но в большинстве случаев реальные асимптотики не выходят за пределы O(log n).

Устройство структуры

АВЛ-дерево реализуется с использованием стандартных узлов бинарного дерева, где каждый узел в себе хранит:

- **Ключ**: это значение, которое определяет порядок элементов в дереве (например, для бинарного дерева поиска — это может быть числовое значение).
- **Указатели на левое и правое поддеревья**: они представляют собой указатели на дочерние вершины.
- **Балансирующий коэффициент**: для каждой вершины хранится число, представляющее разницу в высотах левого и правого поддеревьев. Этот коэффициент может быть равен -1, 0 или +1, что указывает на то, насколько сбалансирована вершина. Или можно максимальную высоту поддерева, увеличенную на 1.

```python
class Node:
    def __init__(self, d):
        self.key = d
        self.height = 1
        self.left = None
        self.right = None
```

Рис. 9 Реализация вершины на python 3.9

Балансировка дерева

Балансировкой вершины называется операция, которая в случае разницы весов левого и правого поддеревьев вершины n

$|h(n.l) - h(n.r)| = 2$ изменяет связь «предок-потомок» для восстановления баланса
$|h(n.l) - h(n.r)| \leq 1$

4 вида балансировки:

1. Малый левый поворот
2. Большой левый (Право-левый) поворот
3. Малый правы поворот (отзеркаленный малый левый поворот)
4. Большой правый (Лево-правый) поворот (отзеркаленный большой левый поворот)

Правый поворот (RR)

Правый поворот выполняется, когда левое поддерево узла слишком высокое. В результате поворота корень поддерева становится правым потомком, а левое поддерево этого узла становится новым корнем.

```python
def right_rotate(y:Node)->Node:
    x = y.left
    y.left = x.right
    x.right = y
    y.height = max(height(y.left), height(y.right)) + 1
    x.height = max(height(x.left), height(x.right)) + 1
    return x
```

Рис. 10 Код реализации правого поворота на python 3.9

Левый поворот (LL)

Левый поворот применяется, когда правое поддерево узла слишком высокое. В результате поворота правый потомок узла становится новым корнем, а левое поддерево правого потомка — правым поддеревом узла.

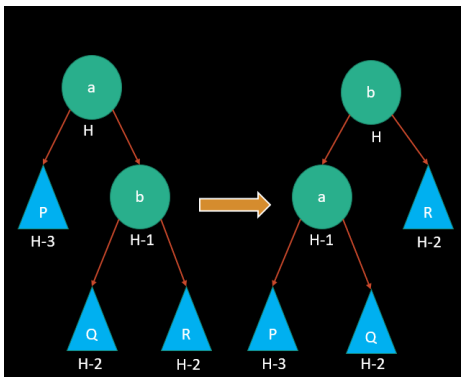

Рис. 11 Пример применения левого поворота

```python
def left_rotate(x:Node)->Node:
    y = x.right
    x.right = y.left
    y.left = x
    x.height = max(height(x.left), height(x.right)) + 1
    y.height = max(height(y.left), height(y.right)) + 1
    return y
```

Рис. 12 Код реализации левого поворота на python 3.9

Право-левый поворот (RL)

Право-левый поворот используется, когда правое поддерево слишком высоко, а в нем, в свою очередь, левое поддерево превышает правое. Для корректировки выполняются два поворота: сначала правый поворот на поддереве правого узла, затем левый поворот на текущем узле.

Лево-правый поворот (LR)

Лево-правый поворот применяется, когда левое поддерево слишком высоко, а в нем правое поддерево превышает левое. Для корректировки выполняются два поворота: сначала левый поворот на поддереве левого узла, затем правый поворот на текущем узле.

```python
def big_right_rotate(x:Node)->Node:
    return right_rotate(left_rotate(x.left))

def big_left_rotate(x:Node)->Node:
    return left_rotate(right_rotate(x.right))
```

Рис. 13 Код реализации Большого правого и большого левого поворотов на python 3.9

Объединение двух АВЛ-деревьев

Процесс объединения двух АВЛ-деревьев обычно состоит из двух этапов:

1. Преобразование одного из деревьев в цепочку с помощью операции удаление (или правого поворота).

2. Слияние двух деревьев путем добавления элементов из одного дерева в другое с помощью последовательных вставок.

Так как операция вставки выполняется за O(log n), объединение двух АВЛ-деревьев может быть выполнено за O(n log n), где n — количество элементов в деревьях.

Слияние двух AVL-деревьев: T1 и T2, где T1 имеет меньше ключей и меньшую высоту.

1. В T1 удаляется самая правая вершина b.
2. В T2 идем от корня в левое поддерево, пока высота поддерева не сравняется с высотой T1.
3. Создаем новое дерево S с корнем b, левым поддеревом T1 и правым поддеревом P.
4. В T2 делаем левое поддерево S и запускаем балансировку.

Временная сложность: O(log(len(T1))) * 2 = O(log(n))

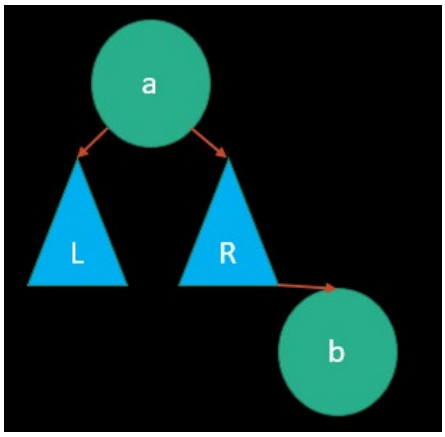

Рис. 14 АВЛ дерево T1

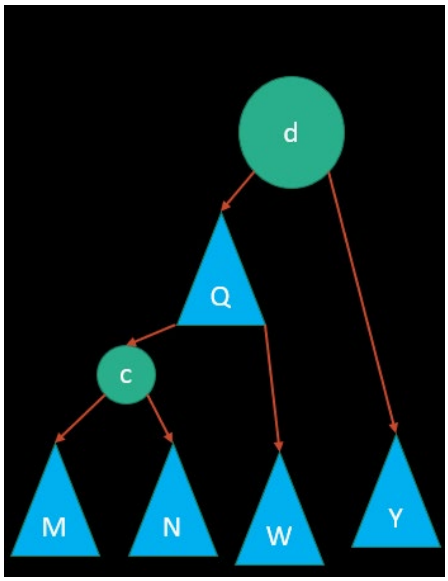

Рис. 15 АВЛ дерево T2

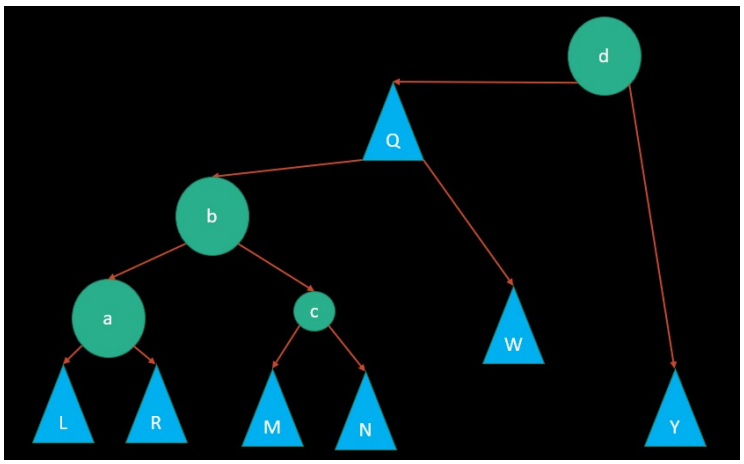

Рис. 16 Результат объединения деревьев T1 и T2

Разъединения АВЛ-дерева на два

Процесс разъединения АВЛ-дерева включает в себя:

1. Поиск элемента, который станет разделяющим элементом.
2. Разбиение дерева на два поддерева.

Так как поиск и разделение требуют O(log n) времени для поиска разделяющего элемента, а затем O(log n) для операции разбиения, общая сложность этого процесса составляет O(log n).

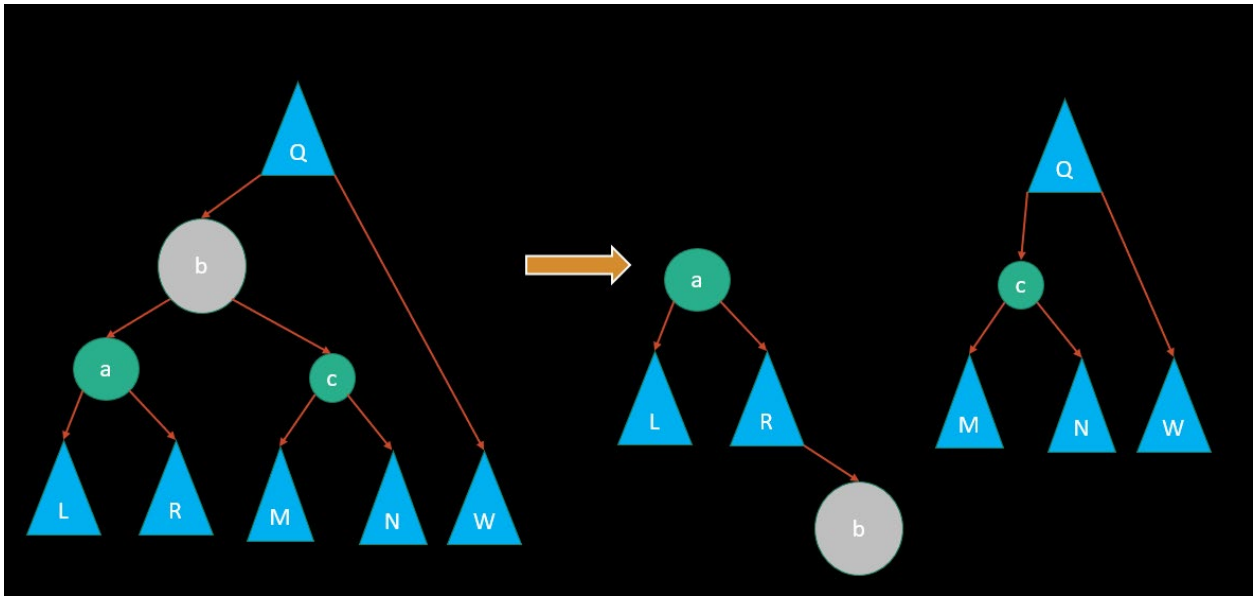

Рис. 17 Пример разделения АВЛ дерева на 2 АВЛ дерева

Примеры применения в реальной жизни

АВЛ-деревья используются в различных областях, где требуется эффективный поиск, вставка и удаление данных:

- **Базы данных**: для индексации данных, когда необходимо поддерживать быстрое выполнение запросов.
- **Файловые системы**: для хранения и быстрого поиска файлов и директорий.
- **Обработчики памяти**: для динамического выделения памяти в операционных системах.
- **Графовые алгоритмы:** АВЛ-деревья могут использоваться для оптимизации поиска кратчайшего пути или поиска в графах с обновляемыми весами рёбер.
- **Реализация контейнеров в стандартных библиотеках**: Например, в C++ STL есть структура данных «`map`», которая может быть реализована с помощью сбалансированных деревьев (красно-черных или АВЛ).
- **Алгоритмы на строках**: АВЛ-деревья могут быть полезны при решении задач, связанных с обработкой строк, таких как задачи поиска подстрок или сжатия данных.

Аналоги и сравнение с ними

Существуют и другие самобалансирующиеся структуры данных, такие как **красно-черные деревья**, **Splay-деревья** и **B-деревья**.

Красно-черные деревья

- **Преимущества**: Более простая реализация, быстрее обрабатывают вставку и удаление (поскольку требуют меньшего числа операций для восстановления баланса).
- **Недостатки**: Чуть более высокая сложность поиска (по сравнению с АВЛ-деревьями), так как балансировка не такая строгая.

B-деревья

- **Преимущества**: Отлично подходят для работы с большими объемами данных, хранящимися на диске. Быстро работают с большими массивами данных.
- **Недостатки**: Для работы с B-деревьями требуется дополнительная инфраструктура, так как они лучше всего применяются в файловых системах.

Таблица сравнения асимптотик аналогов

Операция	АВЛ-дерево	Красно-черное дерево	Splay-дерево	B-дерево
Поиск	O(log n)	O(log n)	O(log n) || O(n)	O(log n)
Вставка	O(log n)	O(log n)	O(log n)	O(log n)
Удаление	O(log n)	O(log n)	O(log n)	O(log n)
Сбалансировка после операции	O(log n)	O(log n)	O(log n)	O(log n)

Ссылка на реализацию

https://github.com/Valet-V0ult-de-Fur1e/combinatorics_and_graphs_repo/tree/main/АВЛ-деревья%20оценка%20высоты%20авл-дерева

Список источников

- https://neerc.ifmo.ru/wiki/index.php?title=АВЛ-дерево
- https://habr.com/ru/articles/150732/
- https://www.geeksforgeeks.org/introduction-to-avl-tree/
- https://blog.skillfactory.ru/glossary/avl-derevo/
- http://shtanyuk.ru/edu/nntu/ads/lections/lec12/lec12.pdf
- https://www.niisi.ru/iont/projects/rfbr/90308/90308_miphi6.php