Лекция 11. Сбалансированные деревья поиска: AVL, Splay.

1. Введение в деревья поиска

Дерево поиска – это структура данных, предназначенная для эффективного хранения множества элементов (с ключами) и выполнения над ними операций добавления, удаления и поиска.

Основные операции:

- insert(key) – добавить элемент с ключом key.
- search(key) – проверить наличие элемента с ключом key.
- delete(key) – удалить элемент с ключом key.
- find_min(), find_max() – найти минимальный/максимальный ключ.
- lower_bound(key), upper_bound(key) – найти наименьший элемент, больший или равный/строго больший заданного ключа.

Зачем нужны деревья поиска?

Они лежат в основе реализаций типов set и map (множество и словарь) во многих стандартных библиотеках, позволяя выполнять основные операции за время, пропорциональное высоте дерева.

Идея двоичного дерева поиска (BST):

Для любого узла x:

- Все ключи в **левом поддереве** x **меньше** ключа x.
- Все ключи в **правом поддереве** x **больше** ключа x.

Проблема. В худшем случае (например, при добавлении отсортированных данных) дерево может выродиться в связный список с высотой O(n), и все операции будут выполняться за O(n).

Решение. Использовать **сбалансированные деревья поиска**, которые поддерживают высоту дерева порядка O(log n) после любых операций, гарантируя логарифмическое время работы.

2. AVL-дерево

2.1. Определение и инвариант

AVL-дерево – это BST, в котором для каждого узла разница высот его левого и правого поддеревьев (фактор балансировки) не превышает 1 по модулю.

$$|\text{height(left)} - \text{height(right)}| <= 1$$

Этот инвариант гарантирует, что высота дерева всегда равна O(log n).

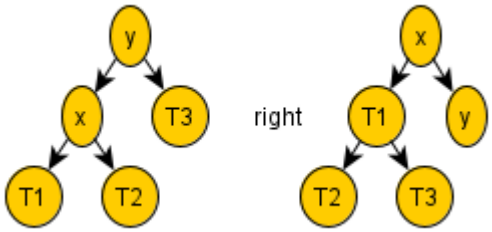
2.2. Балансировка (повороты)

После операций вставки и удаления инвариант может нарушиться. Для его восстановления применяются **повороты**.

Базовые операции поворотов:

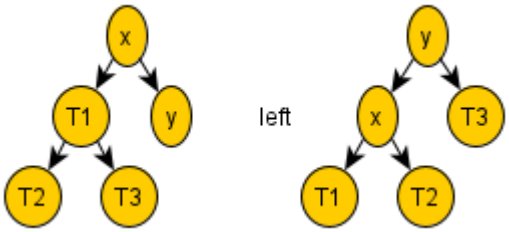
Правый поворот

Применяется, когда левое поддерево слишком высокое.



Левый поворот

Применяется, когда правое поддерево слишком высокое.



2.3. Типы дисбаланса и стратегии балансировки

Выделяют 4 основных случая нарушения баланса:

1. **Left-Left (LL).** Дисбаланс в левом ребенке левого поддерева.

Решение. Один правый поворот вокруг корня.

2. **Right-Right (RR).** Дисбаланс в правом ребенке правого поддерева.

Решение. Один левый поворот вокруг корня.

3. **Left-Right (LR).** Дисбаланс в правом ребенке левого поддерева.

Решение. Сначала левый поворот вокруг левого ребенка, затем правый поворот вокруг корня.

4. **Right-Left (RL).** Дисбаланс в левом ребенке правого поддерева.

Решение. Сначала правый поворот вокруг правого ребенка, затем левый поворот вокруг корня.

2.4. Реализация AVL-дерева

```
from typing import Any, Optional


class AVLNode:
    def __init__(self, key: int, value: Any = None):
        self.key = key  # Ключ узла (по нему происходит сравнение)
        self.value = value  # Значение, хранимое в узле
        self.height = 1  # Высота узла в дереве (изначально 1 для нового
узла)
        self.left: Optional[AVLNode] = None  # Левый ребенок
        self.right: Optional[AVLNode] = None  # Правый ребенок
```

```
    def _get_height(self, node: Optional['AVLNode']) -> int:
        # Вспомогательная функция: возвращает высоту узла, если он
существует, иначе 0
        return node.height if node else 0

    def _get_balance(self) -> int:
        """Вычисляет фактор балансировки узла."""
        # Разница между высотой левого и правого поддеревьев
        # Положительное значение = левое поддерево выше
        # Отрицательное значение = правое поддерево выше
        return self._get_height(self.left) - self._get_height(self.right)

    def _update_height(self) -> None:
        """Обновляет высоту узла."""
        # Высота узла = 1 + максимальная высота из его детей
        self.height = 1 + max(self._get_height(self.left),
self._get_height(self.right))

    def _left_rotate(self) -> 'AVLNode':
        """Левый поворот и возврат нового корня поддерева."""
        # Сохраняем правого ребенка - он станет новым корнем
        new_root = self.right
        # Перемещаем левого ребенка нового корня к текущему корню справа
        self.right = new_root.left
        # Текущий корень становится левым ребенком нового корня
        new_root.left = self

        # Обновляем высоты: сначала старого корня, потом нового
        self._update_height()
        new_root._update_height()
        return new_root  # Возвращаем новый корень поддерева

    def _right_rotate(self) -> 'AVLNode':
        """Правый поворот и возврат нового корня поддерева."""
        # Зеркально противоположно левому повороту
        new_root = self.left
        self.left = new_root.right
        new_root.right = self

        self._update_height()
        new_root._update_height()
        return new_root

    def _balance(self) -> 'AVLNode':
        """Балансирует узел и возвращает новый корень поддерева."""
        # Обновляем высоту текущего узла
        self._update_height()
        balance_factor = self._get_balance()

        # Случай 1: Left-Left - перевесило левое поддерево левого ребенка
        if balance_factor > 1 and self.left._get_balance() >= 0:
            return self._right_rotate()  # Один правый поворот
```

```
        # Случай 2: Right-Right - перевесило правое поддерево правого ребенка
        if balance_factor < -1 and self.right._get_balance() <= 0:
            return self._left_rotate()  # Один левый поворот

        # Случай 3: Left-Right - перевесило правое поддерево левого ребенка
        if balance_factor > 1 and self.left._get_balance() < 0:
            # Сначала левый поворот для левого ребенка, потом правый для
текущего
            self.left = self.left._left_rotate()
            return self._right_rotate()

        # Случай 4: Right-Left - перевесило левое поддерево правого ребенка
        if balance_factor < -1 and self.right._get_balance() > 0:
            # Сначала правый поворот для правого ребенка, потом левый для
текущего
            self.right = self.right._right_rotate()
            return self._left_rotate()

        # Балансировка не требуется - возвращаем текущий узел
        return self


class AVLTree:
    def __init__(self):
        self.root: Optional[AVLNode] = None  # Корень дерева

    def insert(self, key: int, value: Any = None) -> None:
        """Вставляет ключ и значение в дерево."""
        self.root = self._insert(self.root, key, value)

    def _insert(self, node: Optional[AVLNode], key: int, value: Any) ->
AVLNode:
        # Базовый случай: достигли пустого места - создаем новый узел
        if node is None:
            return AVLNode(key, value)

        # Рекурсивно спускаемся в нужное поддерево
        if key < node.key:
            node.left = self._insert(node.left, key, value)  # Идем влево
        elif key > node.key:
            node.right = self._insert(node.right, key, value)  # Идем вправо
        else:
            # Ключ уже существует - обновляем значение
            node.value = value
            return node  # Высоты не изменились, балансировка не нужна

        # После вставки балансируем узел и возвращаем новый корень поддерева
        return node._balance()

    def delete(self, key: int) -> None:
        """Удаляет ключ из дерева."""
        self.root = self._delete(self.root, key)
```

```
    def _delete(self, node: Optional[AVLNode], key: int) ->
Optional[AVLNode]:
        if node is None:
            return None  # Ключ не найден

        # Ищем узел для удаления
        if key < node.key:
            node.left = self._delete(node.left, key)  # Ищем в левом
поддереве
        elif key > node.key:
            node.right = self._delete(node.right, key)  # Ищем в правом
поддереве
        else:
            # Узел для удаления найден
            if node.left is None:
                return node.right  # Нет левого ребенка - возвращаем правого
            elif node.right is None:
                return node.left  # Нет правого ребенка - возвращаем левого
            else:
                # Есть оба ребенка - находим преемника (минимум в правом
поддереве)
                successor = node.right
                while successor.left:
                    successor = successor.left
                # Копируем данные преемника в текущий узел
                node.key = successor.key
                node.value = successor.value
                # Удаляем самого преемника из правого поддерева
                node.right = self._delete(node.right, successor.key)

        # Если узел стал None (например, был листом), возвращаем None
        if node is None:
            return None

        # Балансируем узел после удаления
        return node._balance()

    def search(self, key: int) -> Optional[Any]:
        """Ищет ключ в дереве и возвращает его значение."""
        node = self.root
        while node:
            if key < node.key:
                node = node.left  # Идем влево
            elif key > node.key:
                node = node.right  # Идем вправо
            else:
                return node.value  # Ключ найден
        return None  # Ключ не найден
```

2.5. Сложность AVL-дерева

- **Поиск, вставка, удаление:** O(log n) в худшем случае.
- **Память:** O(n).

3. Splay-дерево

3.1. Идея и эвристика

Splay-дерево – это BST, которое не имеет строгого инварианта сбалансированности, как AVL. Вместо этого после каждой операции (поиск, вставка, удаление) с элементом x этот элемент **поднимается в корень** дерева с помощью специальной операции **splay** («расширение» или «подтягивание»).

Эвристика. Недавно использованные элементы с большей вероятностью будут использоваться снова. Поднимая их в корень, мы ускоряем доступ к ним в будущем.

Амортизированная сложность. Хотя одна операция splay может занять O(n), последовательность из m операций над деревом из n узлов выполняется за O(m log n). **Амортизированная** стоимость одной операции – O(log n).

3.2. Операция Splay

Операция splay(x) выполняется с помощью трех видов шагов (**Zig**, **Zig-Zig**, **Zig-Zag**), применяемых до тех пор, пока x не станет корнем.

• **Zig (или Zag).** Применяется, если родитель x является корнем.

Простой поворот вокруг корня.

• **Zig-Zig.** Применяется, если x и его родитель p являются оба левыми или оба правыми детьми.

Сначала поворот вокруг деда g, затем поворот вокруг родителя p.

• **Zig-Zag.** Применяется, если x – левый ребенок, а p – правый, или наоборот.

Поворот вокруг p, а затем поворот вокруг g.

3.3. Реализация Splay-дерева

```
from typing import Any, Optional


class SplayNode:
    def __init__(self, key: int, value: Any = None):
        self.key = key
        self.value = value
        self.left: Optional['SplayNode'] = None
        self.right: Optional['SplayNode'] = None
        self.parent: Optional['SplayNode'] = None  # Родитель нужен для
операции splay

    def _set_left(self, node: Optional['SplayNode']) -> None:
        # Устанавливает левого ребенка и обновляет ссылку на родителя
        self.left = node
        if node:
            node.parent = self
```

```
    def _set_right(self, node: Optional['SplayNode']) -> None:
        # Устанавливает правого ребенка и обновляет ссылку на родителя
        self.right = node
        if node:
            node.parent = self


class SplayTree:
    def __init__(self):
        self.root: Optional[SplayNode] = None

    def _rotate_left(self, x: SplayNode) -> None:
        """Левый поворот вокруг узла x."""
        p = x.parent  # Родитель x
        y = x.right  # Правый ребенок x (станет новым корнем)
        if y is None:
            return  # Нельзя выполнить поворот

        # Шаг 1: Перевешиваем левого ребенка y к x как правого ребенка
        x._set_right(y.left)

        # Шаг 2: Поднимаем y, делая x его левым ребенком
        y._set_left(x)

        # Шаг 3: Перевешиваем y к родителю x
        if p is None:
            self.root = y  # x был корнем -> y становится корнем
            y.parent = None
        elif p.left == x:
            p._set_left(y)  # x был левым ребенком
        else:
            p._set_right(y)  # x был правым ребенком

    def _rotate_right(self, x: SplayNode) -> None:
        """Правый поворот вокруг узла x."""
        # Зеркально противоположно левому повороту
        p = x.parent
        y = x.left
        if y is None:
            return

        x._set_left(y.right)  # Перевешиваем правого ребенка y к x
        y._set_right(x)  # Поднимаем y, делая x его правым ребенком

        if p is None:
            self.root = y
            y.parent = None
        elif p.left == x:
            p._set_left(y)
        else:
            p._set_right(y)
```

```
    def _splay(self, x: SplayNode) -> None:
        """Поднимает узел x в корень дерева с помощью поворотов."""
        while x.parent is not None:  # Пока x не корень
            p = x.parent
            g = p.parent

            if g is None:
                # Случай Zig: родитель - корень
                if p.left == x:
                    self._rotate_right(p)  # x - левый ребенок
                else:
                    self._rotate_left(p)  # x - правый ребенок
            else:
                if g.left == p and p.left == x:
                    # Случай Zig-Zig: оба ребенка левые
                    self._rotate_right(g)
                    self._rotate_right(p)
                elif g.right == p and p.right == x:
                    # Случай Zig-Zig: оба ребенка правые
                    self._rotate_left(g)
                    self._rotate_left(p)
                elif g.left == p and p.right == x:
                    # Случай Zig-Zag: левый-правый
                    self._rotate_left(p)
                    self._rotate_right(g)
                else:  # g.right == p and p.left == x
                    # Случай Zig-Zag: правый-левый
                    self._rotate_right(p)
                    self._rotate_left(g)

    def search(self, key: int) -> Optional[Any]:
        """Ищет ключ и поднимает его в корень (если найден)."""
        node = self.root
        last_visited = None  # Запоминаем последний посещенный узел

        while node:
            last_visited = node
            if key < node.key:
                node = node.left  # Идем влево
            elif key > node.key:
                node = node.right  # Идем вправо
            else:
                # Ключ найден - поднимаем его в корень
                self._splay(node)
                return node.value

        # Ключ не найден, но поднимаем последний посещенный узел
        # (это улучшает производительность для будущих запросов)
        if last_visited:
            self._splay(last_visited)
        return None
```

```
def insert(self, key: int, value: Any = None) -> None:
    """Вставляет ключ и поднимает его в корень."""
    if self.root is None:
        # Дерево пустое - создаем корень
        self.root = SplayNode(key, value)
        return

    # Спускаемся вниз, чтобы найти место для вставки
    node = self.root
    while node:
        if key < node.key:
            if node.left is None:
                # Нашли место - вставляем новый узел слева
                new_node = SplayNode(key, value)
                node._set_left(new_node)
                self._splay(new_node)  # Поднимаем новый узел в корень
                return
            node = node.left
        elif key > node.key:
            if node.right is None:
                # Нашли место - вставляем новый узел справа
                new_node = SplayNode(key, value)
                node._set_right(new_node)
                self._splay(new_node)  # Поднимаем новый узел в корень
                return
            node = node.right
        else:
            # Ключ уже существует - обновляем значение и поднимаем
            node.value = value
            self._splay(node)
            return

def delete(self, key: int) -> None:
    """Удаляет ключ из дерева."""
    # Сначала находим узел (это автоматически поднимет его в корень)
    if self.search(key) is None:
        return  # Ключа нет в дереве

    # Теперь root - это узел, который мы хотим удалить
    left_tree = self.root.left
    right_tree = self.root.right

    # Отсоединяем детей от удаляемого корня
    if left_tree:
        left_tree.parent = None
    if right_tree:
        right_tree.parent = None

    # Удаляем корень
    self.root = None
```

```
        # Сливаем левое и правое поддеревья
        if left_tree is None:
            self.root = right_tree  # Просто берем правое поддерево
        elif right_tree is None:
            self.root = left_tree  # Просто берем левое поддерево
        else:
            # Оба поддерева существуют
            # Находим максимум в левом поддереве (он не имеет правого
ребенка)
            self.root = left_tree
            max_node = left_tree
            while max_node.right:
                max_node = max_node.right
            # Поднимаем максимум в корень левого поддерева
            self._splay(max_node)
            # Присоединяем правое поддерево к новому корню
            self.root._set_right(right_tree)
```

3.4. Сложность Splay-дерева

• **Амортизированная сложность** операций insert, search, delete: O(log n).

• **Память:** O(n).

4. Сравнение AVL и Splay деревьев

Характеристика	**AVL-дерево**	**Splay-дерево**
Гарантия времени	Строгая O(log n) в худшем случае	**Амортизированная** O(log n)
Балансировка	Строгая, после каждой модификации	Эвристическая, только для используемых элементов
Константы	Меньшие константы из-за строгой балансировки	Большие константы из-за операции splay
Доп. память	O(1) на узел (высота)	O(1) на узел (высота + родитель)
Использование	Хорошо, когда важен худший случай (реальные времени системы)	Хорошо для кэширования, когда данные имеют локальность доступа
Реализация	Относительно сложная (4 случая поворотов)	Сложная (3 случая шагов splay)