AVL Дерево

Толстых Тимур, 11-401

Введение

Определение: АВЛ-дерево (англ. AVL-Tree) — сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

История: АВЛ-деревья названы по первым буквам фамилий их изобретателей, Г. М. Адельсона-Вельского и Е. М. Ландиса, которые впервые предложили использовать АВЛ-деревья в 1962 году.

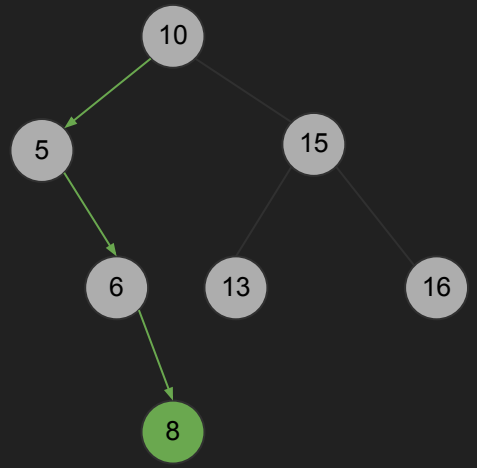
Сложность операций

	Average	Worst
Insertion	O(log(n))	O(log(n))
Deletion	O(log(n))	O(log(n))
Searching	O(log(n))	O(log(n))
Traversal	O(n)	O(n)

*Такая асимптотика достигается за счёт самобалансировки дерева и постоянном поддержании высоты h = O(log(n))

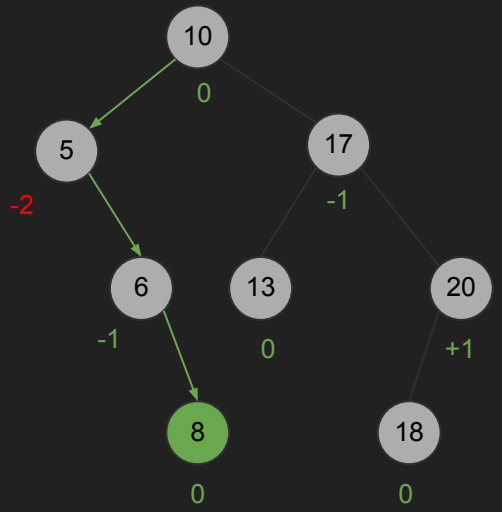
Insertion

Идея: спускаемся вниз по дереву и ищем место для новой ноды на основе свойства упорядоченности.



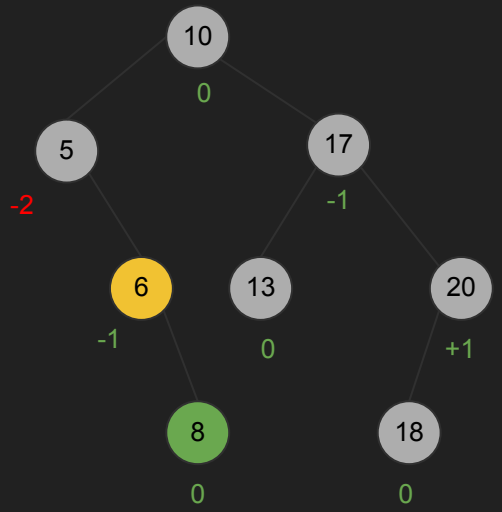
Insertion

Идея: отобразим баланс каждой ноды, допустимый баланс должен быть равен: **-1, 0, 1**. При любом другом значении требуется исправить баланс нод.



Insertion

Идея: Случай в этом примере можно назвать RR (right-right) поворот, так как баланс ноды испортился после вставки ноды в правое поддерево правого ребёнка



Insertion

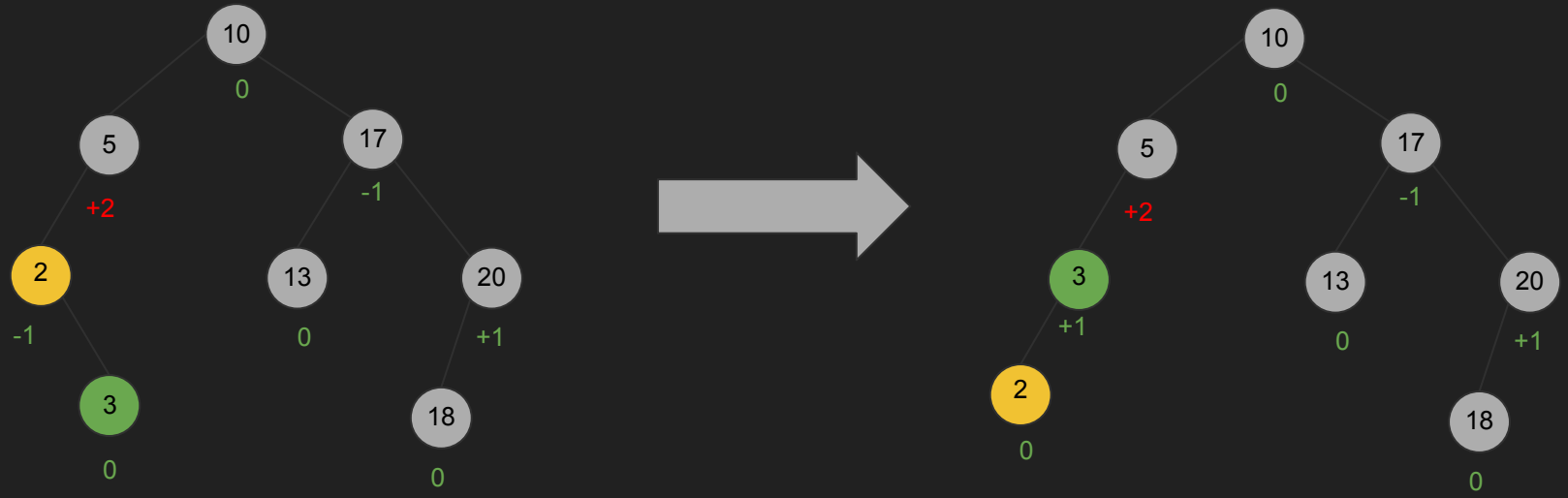
Идея: Чтобы исправить RR случай, достаточно один раз повернуть проблемную ноду против часовой стрелки. Также можно заметить, что после поворота баланс ноды выше может поменяться и испортиться, поэтому важно продолжать процедуру перебалансировки от вставленной ноды до корня дерева

Insertion

Идея: LL случай симметричен, возникает в левом поддереве левого ребенка ноды, которую нужно исправить. Поворачиваем ноду по часовой стрелке.

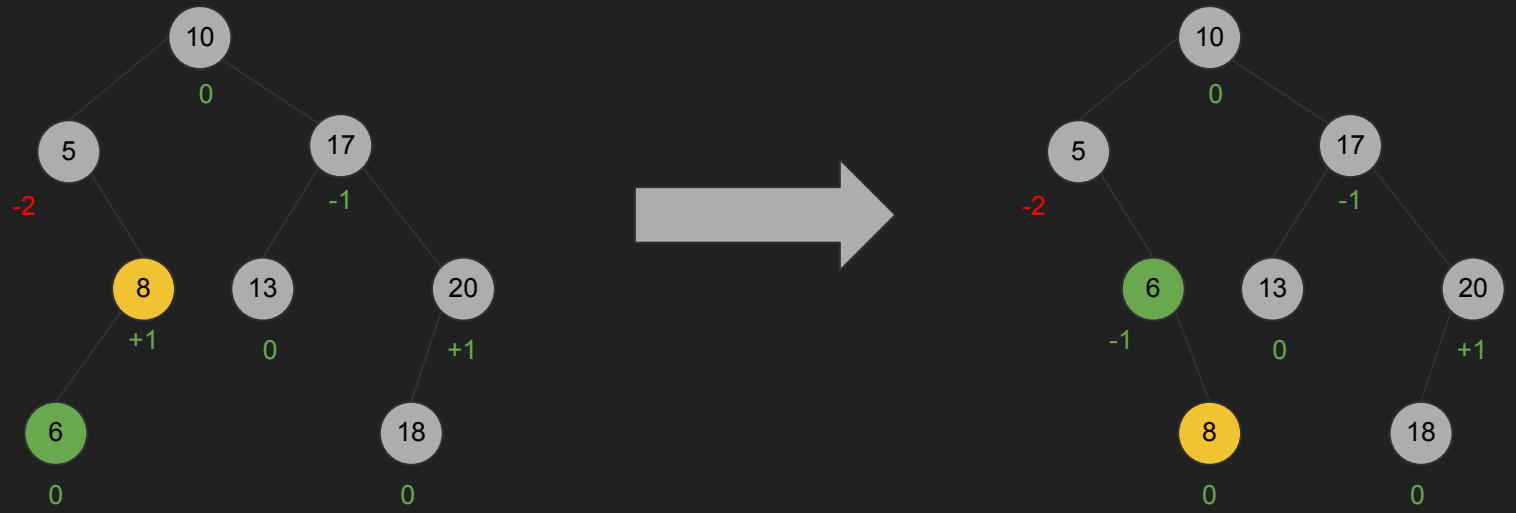
Insertion

Идея: LR случай сводится к LL повороту, возникает в правом поддереве левого ребенка ноды, которую нужно исправить. Для этого поворачиваем левого ребенка против часовой стрелки



Insertion

Идея: RL случай сводится к RR повороту, возникает в левом поддереве правого ребенка ноды, которую нужно исправить. Для этого поворачиваем правого ребенка по часовой стрелке



Insertion

Реализация (рекурсия): на основе сравнения ключа *key* будем выбираем левого или правого потомка, по ним же ведем и рекурсию. При нахождении места для ноды и ее вставке, начинаем подниматься по стеку вызовов вверх, проверять баланс нод и поворачивать их, если необходимо с помощью процедуры *fix_balance*

```
1  template <typename T>
2  Node<T> *AVLTree<T>::insert_recursively(Node<T> *node, const T &key) {
3    if (node == nullptr) {
4      ++size_;
5      return new Node<T>(key);
6    }
7    if (key == node->key_) {
8      return node;
9    }
10   if (key < node->key_) {
11     node->left_ = insert_recursively(node->left_, key);
12     node->left_->parent_ = node;
13   } else if (key > node->key_) {
14     node->right_ = insert_recursively(node->right_, key);
15     node->right_->parent_ = node;
16   }
17   node->recalc_height();
18   return fix_balance(node, key);
19 }
```

Find

Реализация (итеративная): Начинаем идти от корня дерева и сравнивать ключ с ключами в нодах, если наш ключ меньше, то берем левого потомка, иначе правого. Процедура продолжается до тех пор пока мы не найдем нужный ключ, либо найдем пустой лист, что будет значить отсутствие ключа в дереве

```
1  template <typename T> Node<T> *AVLTree<T>::find(const T &key) {
2    if (root_ == nullptr) return nullptr;
3    Node<T> *walk_node = root_;
4    while (walk_node != nullptr && key != walk_node->key_) {
5      walk_node = (key < walk_node->key_)
6        ? walk_node->left_
7        : walk_node->right_;
8    }
9    return walk_node;
10 }
```

Traversal

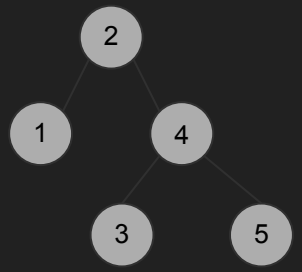
traversal - это метод который обходит все ноды дерева по одному разу.

Есть три вида обхода дерева: **inorder**, **postorder**, **preorder**.

- **inorder:** выводит ноды в порядке возрастания: 1 2 3 4 5.
  Может быть полезен для получения данных в возрастающем порядке

- **preorder:** выводит сначала корень, а потом дочерние узлы: 2 1 4 3 5
  Может использоваться при копировании структуры дерева

- **postorder:** сначала работает с дочерними узлами, а только потом переходит к корню: 1 3 5 4 2
  Может понадобиться при удалении дерева



Traversal

Реализация обходов на основе рекурсии одинаковая, за исключением расположения вызова функции, которая будет осуществлять работу над нодой, в данном случае сохранение ключа в вектор:

inorder:

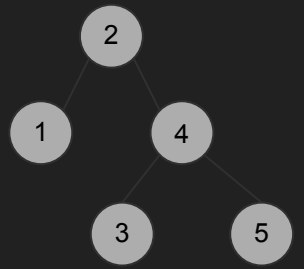
```
2  template <typename T>
3  void AVLTree<T>::in_order_(Node<T> *node, std::vector<T> &vec) const {
4    if (node == nullptr)
5      return;
6    in_order_(node->left_, vec);
7    vec.push_back(node->key_);
8    in_order_(node->right_, vec);
9  }
```

1 2 3 4 5

postorder:

```
2  template <typename T>
3  void AVLTree<T>::post_order_(Node<T> *node, std::vector<T> &vec) const {
4    if (node == nullptr)
5      return;
6    post_order_(node->left_, vec);
7    post_order_(node->right_, vec);
8    vec.push_back(node->key_);
9  }
```

1 3 5 4 2



preorder:

```
2  template <typename T>
3  void AVLTree<T>::pre_order_(Node<T> *node, std::vector<T> &vec) const {
4    if (node == nullptr)
5      return;
6    vec.push_back(node->key_);
7    pre_order_(node->left_, vec);
8    pre_order_(node->right_, vec);
9  }
```

2 1 4 3 5

Deletion

Идея: рассмотрим три ситуации:

1. Если у узла нет потомков, то просто удаляем его и исправляем баланс
2. Если у узла один потомок, то удаляем узел и образуем связь между родительским и дочерним узлом
3. Если есть оба потомка, то находим “следующую” ноду (минимум в правом поддереве), она и должна стать вместо текущего узла, присоединив к себе его левое и правое поддерево

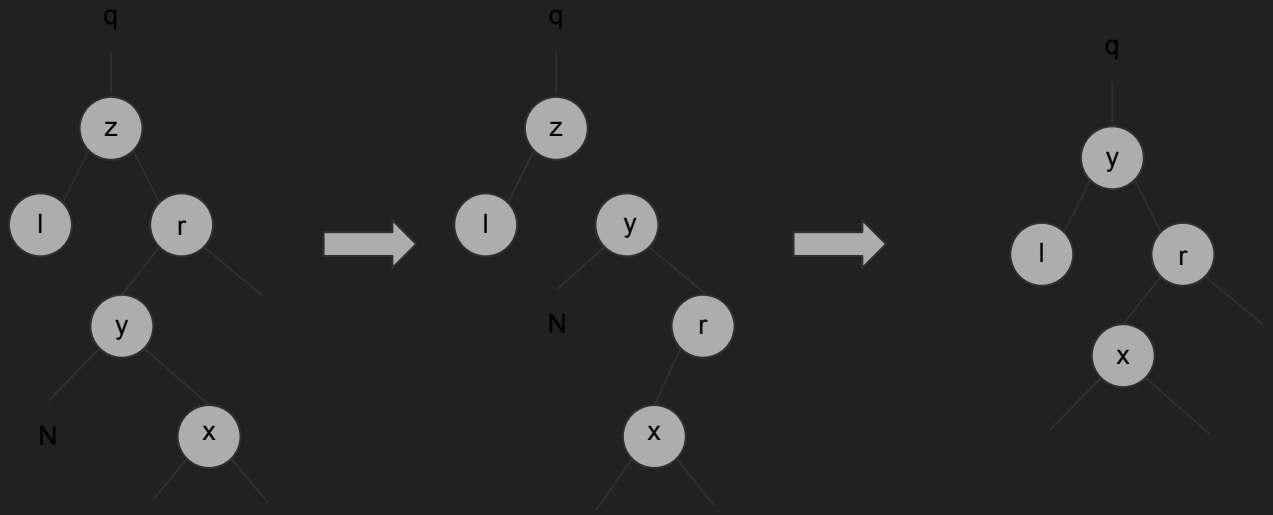
Deletion

Реализация: все три случая можно описать в псевдокоде. Важно отметить, что в третьем случае минимальная нода может и не быть правым ребенком исходной ноды (например, если у правого ребенка есть левое поддерево), такой случай нужно обработать отдельно

```
1   TREE-DELETE(T, z)
2   if z.left == NIL
3     TRANSPLANT(T, z, z.right)
4   elseif z.right == NIL
5     TRANSPLANT(T, z, z.left)
6   else y = TREE-MINIMUM(z.right)
7     if y.p != z
8       TRANSPLANT(T, y, y.right)
9       y.right = z.right
10      y.right.p = y
11    TRANSPLANT(T, z, y)
12    y.left = z.left
13    y.left.p = y
```

Deletion

Пример удаления ноды (3 случай):



Поиск минимума и последующая перебалансировка суммарно дают сложность O(log(n)) для процедуры удаления ноды из дерева