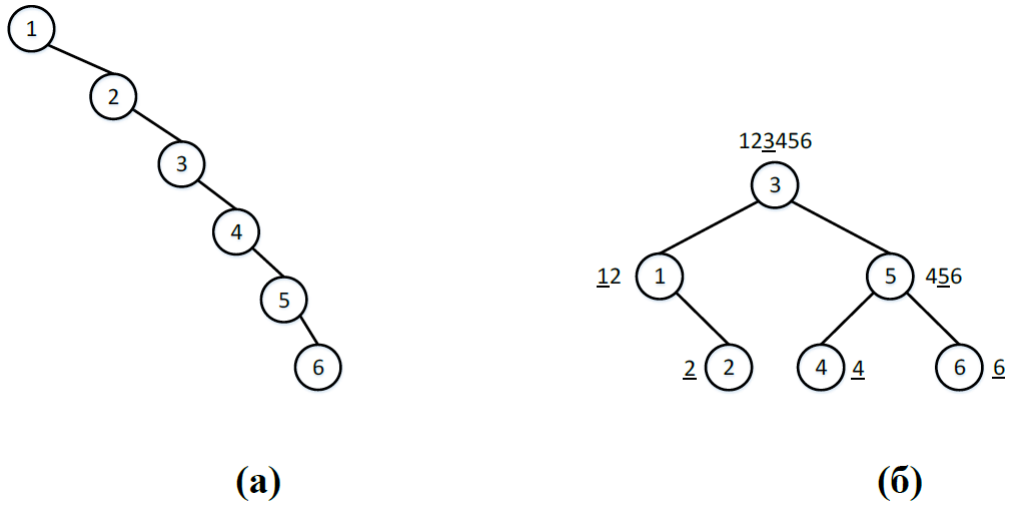
ЛЕКЦИЯ №4

БАЛАНСИРОВКА ДЕРЕВА

БАЛАНСИРОВКА ДЕРЕВА

Время выполнения базовых операций в дереве поиска линейно зависит от его высоты. Но из одного и того же набора ключей можно построить разные деревья поиска, как показано на рисунке:



(а) (б)

БАЛАНСИРОВКА ДЕРЕВА

В приведенном примере оба дерева построены из одного и того же набора ключей, но высота первого дерева больше, поэтому время выполнения операций над ним будет больше. Например, при поиске узла с ключом 6 в случае (а) требуется просмотреть все шесть узлов, а в случае (б) только три узла: 3->5->6. Второе дерево *лучше сбалансировано*, чем первое.

В этом случае хорошо видно преимущество двоичного дерева поиска над обычным двоичным деревом. Если бы дерево на рисунке (б) не было деревом поиска, то при поиске узла с ключом 6 надо было бы просмотреть от 4 до 6 узлов, в зависимости от порядка обхода дерева. А в дереве поиска при поиске узла движение происходит только вниз, поэтому количество шагов ограничено высотой дерева.

Чем больше узлов в дереве, тем более очевидно преимущество дерева поиска над обычным двоичным деревом при условии, что дерево поиска хорошо сбалансировано.

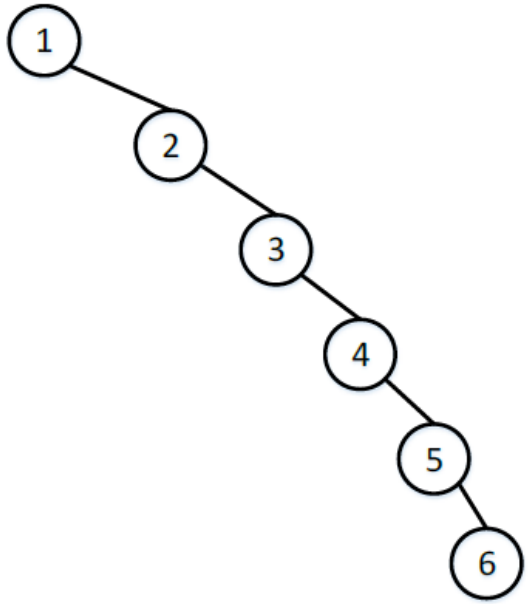
Как построить дерево поиска минимальной высоты? Если набор ключей известен заранее, то его надо упорядочить. Корнем подде-рева становится узел с ключом, значение которого – медиана этого набора. Для упорядоченного набора, содержащего нечетное коли-чество ключей, – это ключ, находящийся ровно в середине набора. Для набора 1,2,3,4,5,6 из примера на первом рисунке, который содержит четное количество ключей, в качестве медианы может быть выбран ключ как со значением 3, так и со значением 4. Для определенности выберем 3. Узел с этим ключом будет корнем дерева. Далее, ключи, меньшие 3, попадут в левое поддерево, а ключи, большие 3, попадут в правое поддерево.

Для построения левого и правого поддеревьев проделываем ту же процедуру на соответствующих наборах ключей. И так до тех пор, пока все ключи не будут включены в дерево. На рисунке (б) для каждого узла указан набор ключей, для которых строится дерево с корнем в этом узле, и выделена медиана этого набора, которая и попадает в корень.

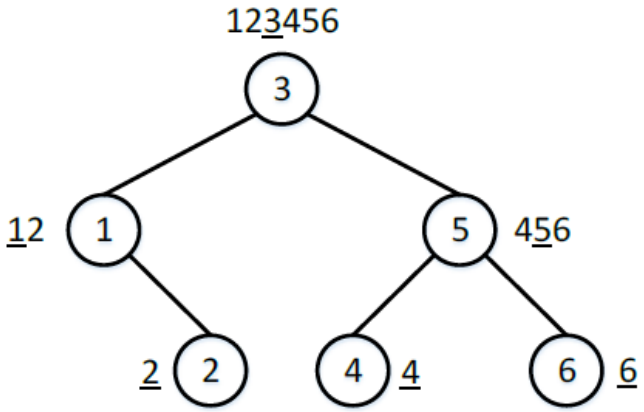
Дерево, изображенное на рисунке (б) является идеально сбалансированным, то есть, для каждого его узла количество узлов в левом и правом поддеревьях различается не более, чем на 1.

Такое дерево можно построить для любого набора ключей. Достаточно заметить, что при выборе медианы на каждом шагу при построении дерева набор ключей разбивается на две части, и количество ключей в них может различаться не более, чем на 1.

Еще раз приведем рисунок описываемого дерева и его сравнение с другим возможным вариантом:



(а)



(б)

БАЛАНСИРОВКА ДЕРЕВА

В этом случае хорошо видно преимущество двоичного дерева поиска над обычным двоичным деревом. Если бы дерево на рисунке (б) не было деревом поиска, то при поиске узла с ключом 6 надо было бы просмотреть от 4 до 6 узлов, в зависимости от порядка обхода дерева. А в дереве поиска при поиске узла движение происходит только вниз, поэтому количество шагов ограничено высотой дерева.

Чем больше узлов в дереве, тем более очевидно преимущество дерева поиска над обычным двоичным деревом при условии, что дерево поиска хорошо сбалансировано.

Как построить дерево поиска минимальной высоты? Если набор ключей известен заранее, то его надо упорядочить. Корнем поддерева становится узел с ключом, значение которого – медиана этого набора. Для упорядоченного набора, содержащего нечетное количество ключей, – это ключ, находящийся ровно в середине набора.

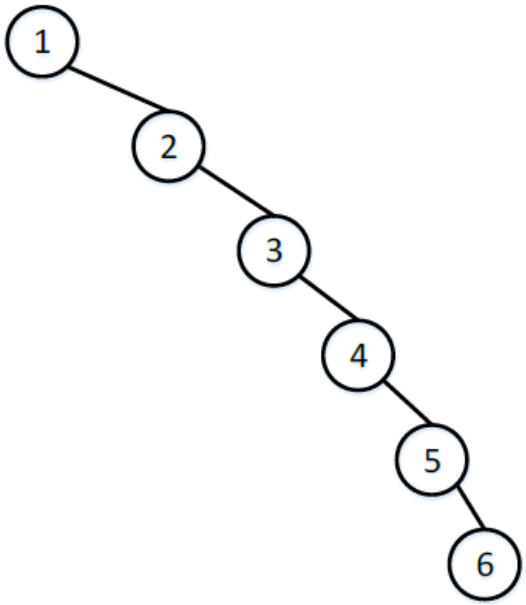
Для набора 1,2,3,4,5,6, который содержит четное количество ключей, в качестве медианы может быть выбран ключ как со значением 3, так и со значением 4.

Для определенности выберем 3. Узел с этим ключом будет корнем дерева.

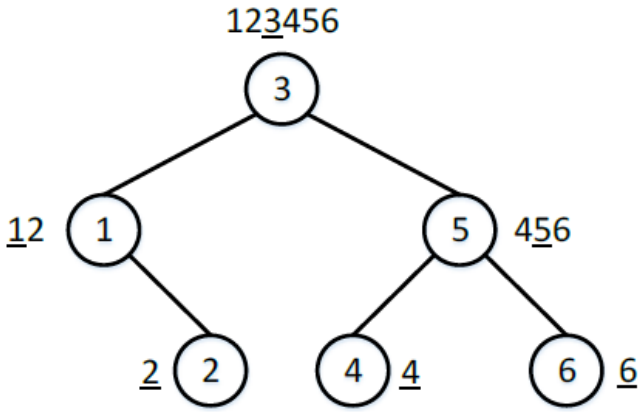
Далее, ключи, меньшие 3, попадут в левое поддерево, а ключи, большие 3, попадут в правое поддерево. Для построения левого и правого поддеревьев проделываем ту же процедуру на соответствующих наборах ключей. И так до тех пор, пока все ключи не будут включены в дерево.

В нашем дереве для каждого узла указан набор ключей, для которых строится дерево с корнем в этом узле, и выделена медиана этого набора, которая и попадает в корень.

Дерево на правом рисунке является идеально сбалансированным, то есть, для каждого его узла количество узлов в левом и правом поддеревьях различается не более, чем на 1.



(а)



(б)

Такое дерево можно построить для любого набора ключей. Достаточно заметить, что при выборе медианы на каждом шагу при построении дерева набор ключей разбивается на две части, и количество ключей в них может различаться не более, чем на 1.

Полный набор ключей не всегда известен заранее. Если ключи поступают по очереди, то построение дерева поиска будет зависеть от порядка их поступления. Если, например, ключи будут поступать в порядке 1,2,3,4,5,6, то получится дерево, изображенное на левом рисунке.

Высота такого дерева максимальна для этого набора ключей, следовательно, и время выполнения операций над ним также будет максимальным. Поэтому при добавлении очередного узла, возможно, дерево понадобится перестраивать, чтобы уменьшить его высоту, сохраняя тот же набор узлов. Идеальную балансировку поддерживать сложно. Если при добавлении очередного узла количество узлов в левом и правом поддеревьях какого-либо узла дерева станет различаться более, чем на 1, то дерево не будет являться идеально сбалансированным, и его надо будет перестраивать, чтобы восстановить свойства идеально сбалансированного дерева поиска. Поэтому обычно требования к сбалансированности дерева менее строгие.

АВЛ-ДЕРЕВЬЯ

В настоящем курсе лекций мы постараемся рассмотреть только три основных вида сбалансированных деревьев поиска: АВЛ-деревья, красно-черные деревья и самоперестраивающиеся деревья (splay-деревья). Для АВЛ-деревьев сбалансированность определяется разностью высот правого и левого поддеревьев любого узла. Если эта разность по модулю не превышает 1, то дерево считается сбалансированным. Данное условие проверяется после каждого добавления или удаления узла, и определен минимальный набор операций перестройки дерева, который приводит к восстановлению свойства сбалансированности, если оно оказалось нарушено.

КРАСНО-ЧЕРНЫЕ-ДЕРЕВЬЯ

В красно-черных деревьях каждый узел имеет дополнительное свойство – цвет, красный или черный.

На дерево наложены ограничения по расположению и количеству узлов в зависимости от цвета, и определен набор операций перестройки дерева в случае нарушения этих ограничений после добавления или удаления узла. Если АВЛ-деревья накладывают достаточно строгие условия на сбалансированность, и при добавлении узлов дерево достаточно часто приходится перестраивать, то в красно-черном дереве у каждого узла высоты левого и правого поддеревьев могут отличаться не более, чем в два раза.

САМОПЕРЕСТРАИВАЮЩИЕСЯ-ДЕРЕВЬЯ

И, наконец, третий вид рассматриваемых сбалансированных деревьев поиска – самоперестраивающиеся деревья. В отличие от двух предыдущих видов, у этих деревьев нет никаких ограничений на расположение узлов, а сбалансированность в среднем достигается за счет того, что каждый раз перед выполнением операции над узлом этот узел перемещается в корень дерева. Но есть вероятность того, что дерево может оказаться не сбалансированным, как, например, на нашем левом рисунке.

АВЛ-ДЕРЕВЬЯ

АВЛ-деревом называется такое дерево поиска, в котором для любого его узла высоты левого и правого поддеревьев отличаются не более, чем на 1. Эта структура данных разработана советскими учеными Адельсон-Вельским Георгием Максимовичем и Ландисом Евгением Михайловичем в 1962 году. Аббревиатура АВЛ со-ответствует первым буквам фамилий этих ученых.

Первоначально АВЛ-деревья были придуманы для организации перебора в шахматных программах. Советская шахматная программа «Каисса» стала первым официальным чемпионом мира в 1974 году.

АВЛ-ДЕРЕВЬЯ

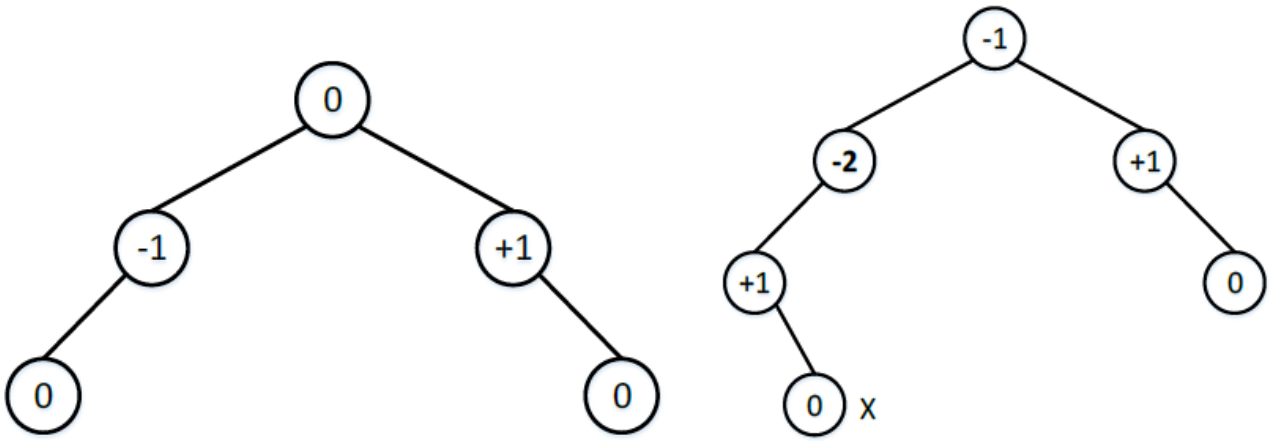
В каждом узле АВЛ-дерева, помимо ключа, данных и указателей на левое и правое поддеревья (левого и правого сыновей), хранится показатель баланса – разность высот правого и левого поддеревьев. В некоторых реализациях этот показатель может вычисляться отдельно в процессе обработки дерева тогда, когда это необходимо.

АВЛ-ДЕРЕВЬЯ

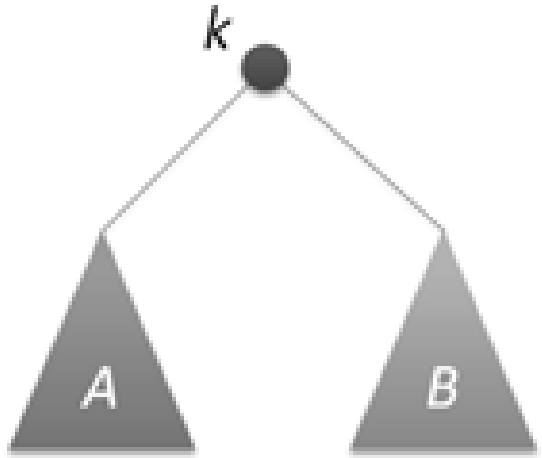
```
struct AVLNode
{
key_type key;
struct AVLNode *left;
struct AVLNode *right;
struct AVLNode *parent;
int balance; // показа-тель баланса
};
```

АВЛ-ДЕРЕВЬЯ

На левом рис. приведен пример АВЛ-дерева. Таким образом, получается, что в АВЛ-дереве показатель баланса balance для каждого узла, включая корень, по модулю не превосходит 1. нижеприведен пример дерева (правый рис.), которое не является АВЛ-деревом, поскольку в одном из узлов баланс нарушен, т.е. |balance|>1.



АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются.



$A < k < B$

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от log2(n + 1) до 1.44 log2(n + 2) − 0.328. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем гарантированную логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве.

Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается не равной нулю.

СТРУКТУРА УЗЛОВ

Будем представлять узлы АВЛ-дерева следующей структурой:

```
struct node // структура для представления узлов дерева
{
	int key;
	unsigned char height;
	node* left;
	node* right;
	node(int k) { key = k; left = right = 0; height = 1; }
};
```

СТРУКТУРА УЗЛОВ

Поле key хранит ключ узла, поле height — высоту поддерева с корнем в данном узле, поля left и right — указатели на левое и правое поддеревья. Простой конструктор создает новый узел (высоты 1) с заданным ключом k.

Традиционно, узлы АВЛ-дерева хранят не высоту, а разницу высот правого и левого поддеревьев (так называемый balance factor), которая может принимать только три значения -1, 0 и 1.

ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ

Определим три вспомогательные функции, связанные с высотой.

Первая является оберткой для поля height, она может работать и с нулевыми указателями (с пустыми деревьями):

```
unsigned char height(node* p)
{
    return p?p->height:0;
}
```

ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ

Вторая вычисляет balance factor заданного узла (и работает только с ненулевыми указателями):

```
int bfactor(node* p)
{
        return height(p->right)-height(p->left);
}
```

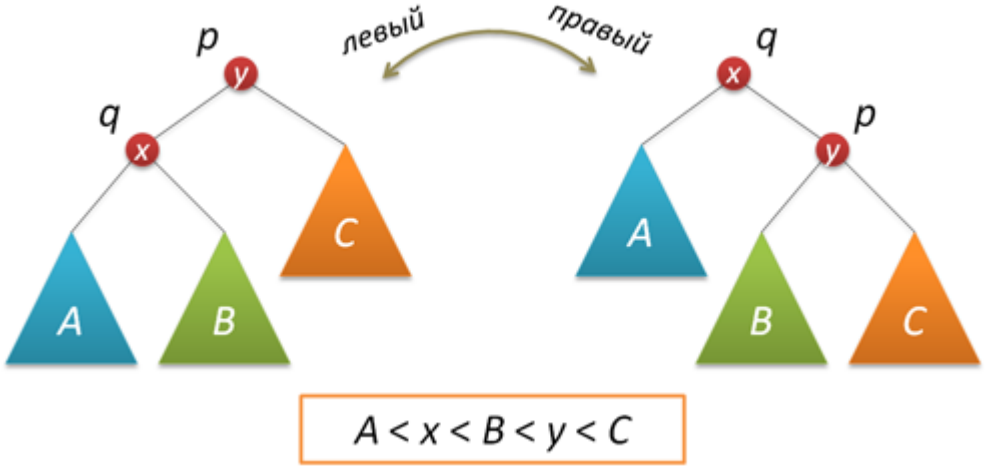
ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ

Третья функция восстанавливает корректное значение поля height заданного узла (при условии, что значения этого поля в правом и левом дочерних узлах являются корректными):

```
void fixheight(node* p)
{
        unsigned char hl = height(p->left);
        unsigned char hr = height(p->right);
        p->height = (hl>hr?hl:hr)+1;
}
```

БАЛАНСИРОВКА УЗЛОВ

В процессе добавления или удаления узлов в АВЛ-дереве возможно возникновение ситуации, когда balance factor некоторых узлов оказывается равными 2 или -2, т.е. возникает расбалансировка поддерева. Для выправления ситуации применяются хорошо нам известные повороты вокруг тех или иных узлов дерева. Напомню, что простой поворот вправо (влево) производит следующую трансформацию дерева:



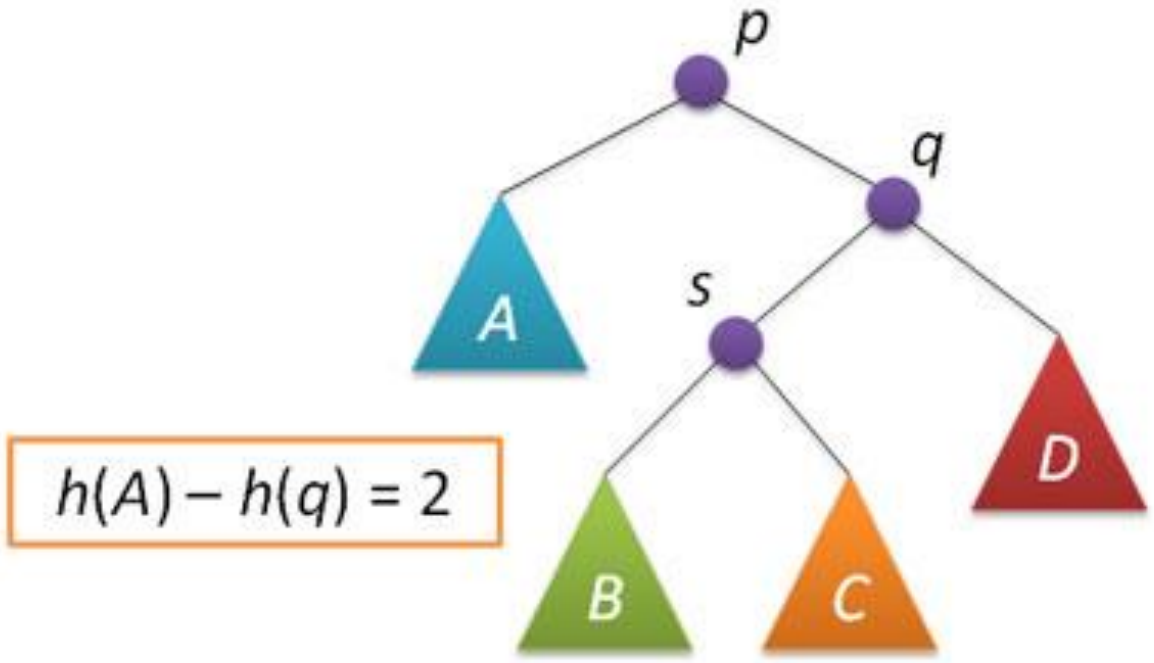
Код, реализующий правый поворот, выглядит следующим образом (как обычно, каждая функция, изменяющая дерево, возвращает новый корень полученного дерева):

```
node* rotateright(node* p) // правый поворот вокруг p
{
	node* q = p->left;
	p->left = q->right;
	q->right = p;
	fixheight(p);
	fixheight(q);
	return q;
}
```

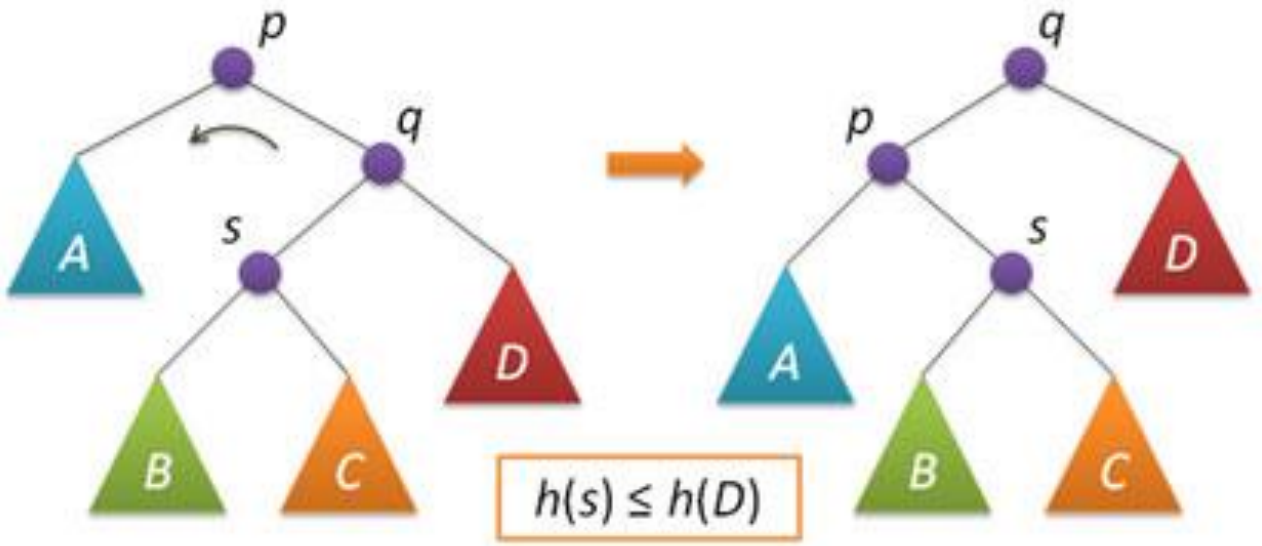
Левый поворот является симметричной копией правого:

```
node* rotateleft(node* q) // левый поворот вокруг q
{
        node* p = q->right;
        q->right = p->left;
        p->left = q;
        fixheight(q);
        fixheight(p);
        return p;
}
```

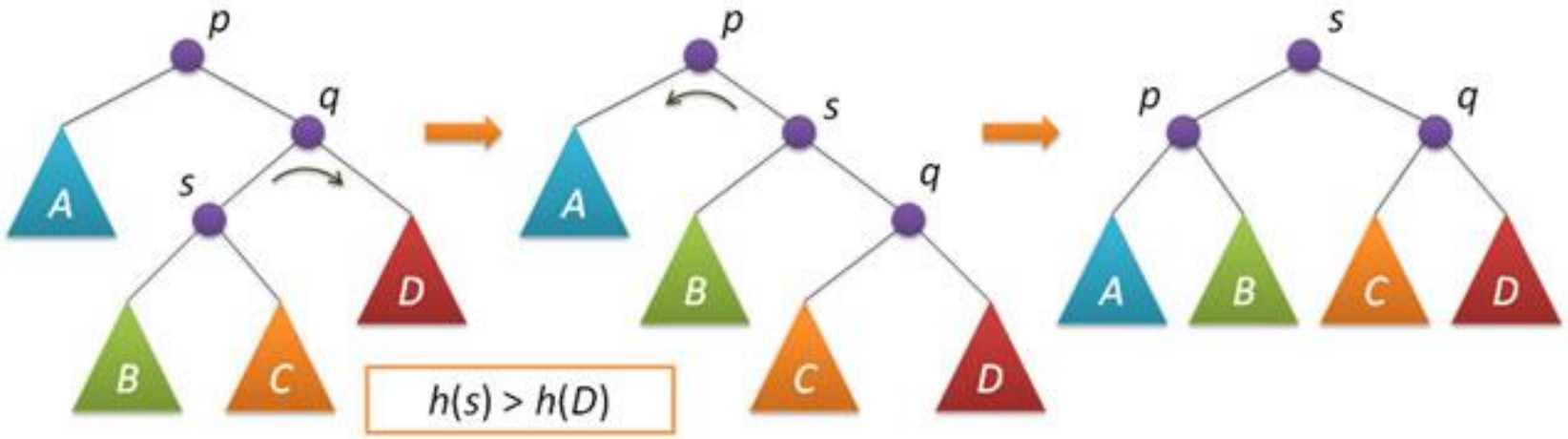
Рассмотрим теперь ситуацию дисбаланса, когда высота правого поддерева узла p на 2 больше высоты левого поддерева (обратный случай является симметричным и реализуется аналогично). Пусть q — правый дочерний узел узла p, а s — левый дочерний узел узла q.



Анализ возможных случаев в рамках данной ситуации показывает, что для исправления расбалансировки в узле p достаточно выполнить либо простой поворот влево вокруг p, либо так называемый большой поворот влево вокруг того же p. Простой поворот выполняется при условии, что высота левого поддерева узла q больше высоты его правого поддерева: h(s)≤h(D).



Большой поворот применяется при условии h(s)>h(D) и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p.



Код, выполняющий балансировку, сводится к проверке условий и выполнению поворотов:

```
node* balance(node* p) // балансировка узла p
{
    fixheight(p);
    if( bfactor(p)==2 )
    {
        if( bfactor(p->right) < 0 )
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
```

Продолжение кода:

```
if( bfactor(p)==-2 )
        {
                if( bfactor(p->left) > 0  )
                        p->left = rotateleft(p->left);
                return rotateright(p);
        }
        return p; // балансировка не нужна
}
```

Описанные функции поворотов и балансировки также не содержат ни циклов, ни рекурсии, а значит выполняются за постоянное время, не зависящее от размера АВЛ-дерева.

ВСТАВКА КЛЮЧЕЙ

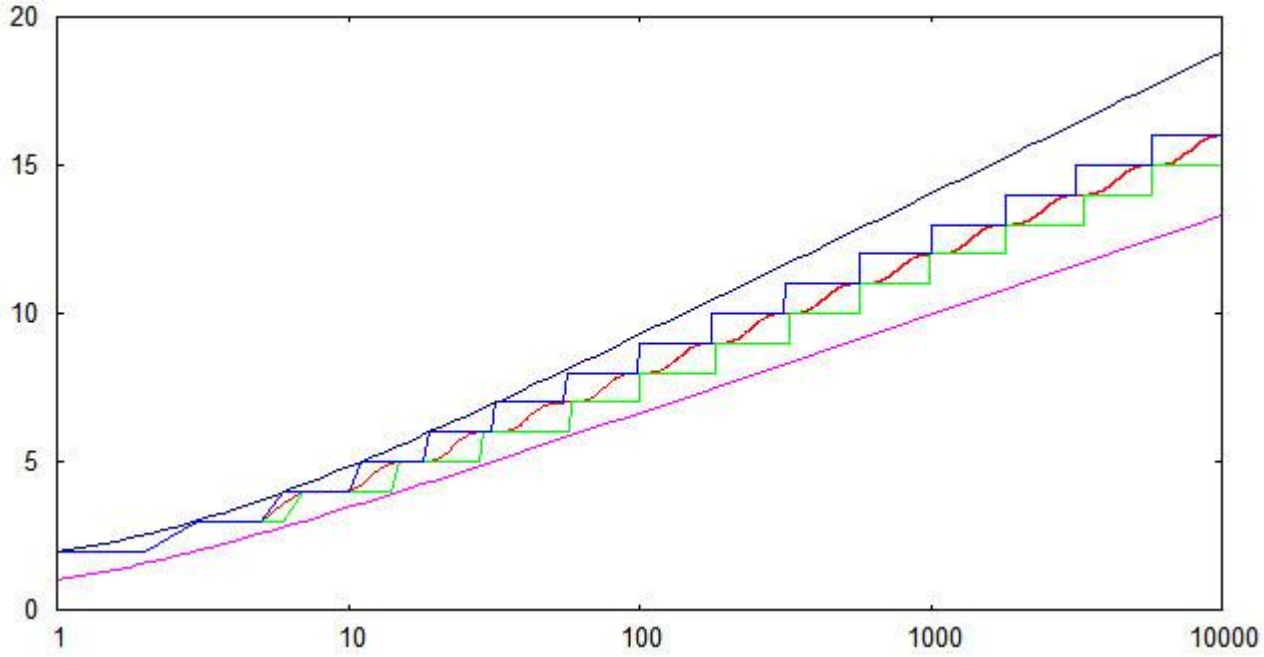
Вставка нового ключа в АВЛ-дерево выполняется, по большому счету, так же, как это делается в простых деревьях поиска: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего узла. Строго доказывается, что возникающий при такой вставке дисбаланс в любом узле по пути движения не превышает двух, а значит применение вышеописанной функции балансировки является корректным.

Код, выполняющий вставку ключей:

```
node* insert(node* p, int k) // вставка ключа k в дерево с корнем p
{
    if( !p ) return new node(k);
    if( k<p->key )
        p->left = insert(p->left,k);
    else
        p->right = insert(p->right,k);
    return balance(p);
}
```

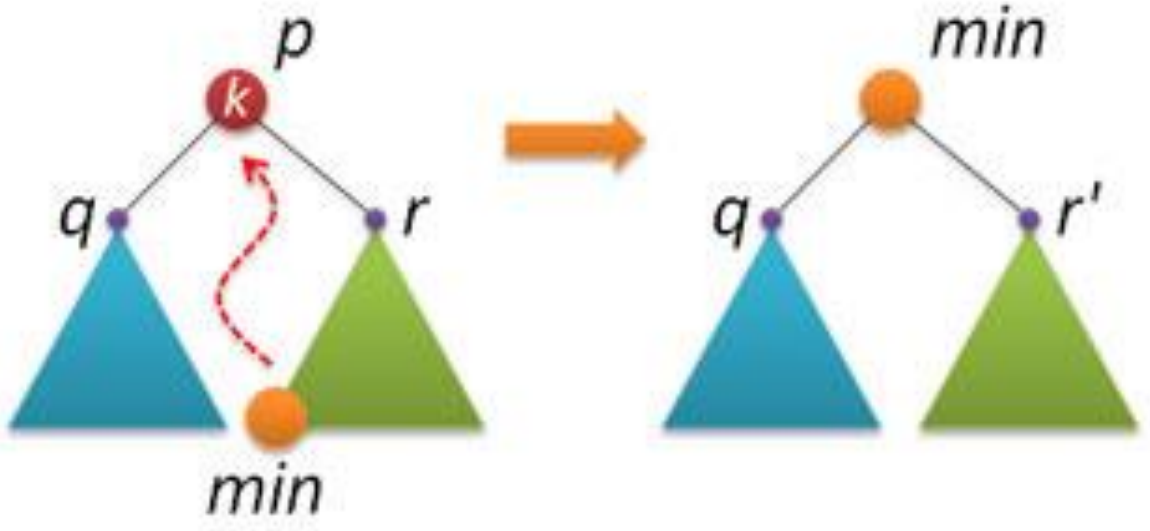
Чтобы проверить соответствие реализованного алгоритма вставки теоретическим оценкам для высоты АВЛ-деревьев, был проведен несложный вычислительный эксперимент. Генерировался массив из случайно расположенных чисел от 1 до 10000, далее эти числа последовательно вставлялись в изначально пустое АВЛ-дерево и измерялась высота дерева после каждой вставки. Полученные результаты были усреднены по 1000 расчетам. На следующем графике показана зависимость от n средней высоты (красная линия); минимальной высоты (зеленая линия); максимальной высоты (синяя линия). Кроме того, показаны верхняя и нижняя теоретические оценки.

Видно, что для случайных последовательностей ключей экспериментально найденные высоты попадают в теоретические границы даже с небольшим запасом. Нижняя граница достижима (по крайней мере в некоторых точках), если исходная последовательность ключей является упорядоченной по возрастанию.



УДАЛЕНИЕ КЛЮЧЕЙ

Процедура удаления узлов из АВЛ-дерева, к сожалению, не очень элегантная. Идея следующая: находим узел p с заданным ключом k, в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min.



При реализации возникает несколько нюансов. Прежде всего, если у найденный узел p не имеет правого поддерева, то по свойству АВЛ-дерева слева у этого узла может быть только один единственный дочерний узел (дерево высоты 1), либо узел p вообще лист. В обоих этих случаях надо просто удалить узел p и вернуть в качестве результата указатель на левый дочерний узел узла p.

Пусть теперь правое поддерево у p есть. Находим минимальный ключ в этом поддереве. По свойству двоичного дерева поиска этот ключ находится в конце левой ветки, начиная от корня дерева.

Применяем рекурсивную функцию:

```
node* findmin(node* p) // поиск узла с минимальным ключом в дереве p
{
        return p->left?findmin(p->left):p;
}
```

Еще одна служебная функция у нас будет заниматься удалением минимального элемента из заданного дерева. Опять же, по свойству АВЛ-дерева у минимального элемента справа либо подвешен единственный узел, либо там пусто. В обоих случаях надо просто вернуть указатель на правый узел и по пути назад (при возвращении из рекурсии) выполнить балансировку. Сам минимальный узел не удаляется, т.к. он еще будет нужен.

```
node* removemin(node* p) // удаление узла с минимальным ключом из дерева p
{
    if( p->left==0 )
        return p->right;
    p->left = removemin(p->left);
    return balance(p);
}
```

Теперь все готово для реализации удаления ключа из АВЛ-дерева. Сначала находим нужный узел, выполняя те же действия, что и при вставке ключа:

```
node* remove(node* p, int k) // удаление ключа k из дерева p
{
        if( !p ) return 0;
        if( k < p->key )
                p->left = remove(p->left,k);
        else if( k > p->key )
                p->right = remove(p->right,k);
```

Как только ключ k найден, переходим к плану Б: запоминаем корни q и r левого и правого поддеревьев узла p; удаляем узел p; если правое поддерево пустое, то возвращаем указатель на левое поддерево; если правое поддерево не пустое, то находим там минимальный элемент min, потом его извлекаем оттуда, слева к min подвешиваем q, справа — то, что получилось из r, возвращаем min после его балансировки.

```
else //  k == p->key
{
        node* q = p->left;
        node* r = p->right;
        delete p;
        if( !r ) return q;
        node* min = findmin(r);
        min->right = removemin(r);
        min->left = q;
        return balance(min);
}
```

При выходе из рекурсии не забываем выполнить балансировку:

```
        return balance(p);
}
```

Поиск минимального узла и его извлечение, в принципе, можно реализовать в одной функции, при этом придется решать проблему с возвращением из функции пары указателей. Зато можно сэкономить на одном проходе по правому поддереву.

Очевидно, что операции вставки и удаления (а также более простая операция поиска) выполняются за время пропорциональное высоте дерева, т.к. в процессе выполнения этих операций производится спуск из корня к заданному узлу, и на каждом уровне выполняется некоторое фиксированное число действий. В силу того, что АВЛ-дерево является сбалансированным, его высота зависит логарифмически от числа узлов. Таким образом, время выполнения всех трех базовых операций гарантированно логарифмически зависит от числа узлов дерева.