



## Лекция 4. Деревья

Алгоритмы и  
структуры данных



Глушенков Д.А.

# План лекции 6 «Деревья»



1. Определения, примеры деревьев
2. Представление в памяти
3. Обходы дерева в глубину, в ширину
4. Двоичные деревья поиска
5. АВЛ-деревья
6. Красно-черные деревья
7. В-деревья
8. АТД «Ассоциативный массив»



# Определения деревьев



**Определение 1.** Дерево (свободное) – непустая коллекция вершин и ребер, удовлетворяющих определяющему свойству дерева.

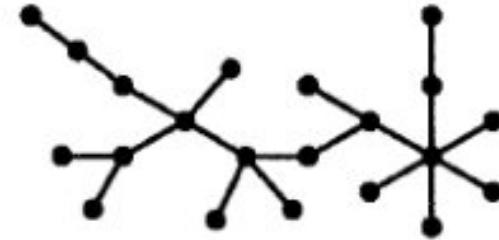
**Вершина (узел)** – простой объект, который может содержать некоторую информацию.

**Ребро** – связь между двумя вершинами.

**Путь в дереве** – список отдельных вершин, в котором следующие друг за другом вершины соединяются ребрами дерева.

**Определяющее свойство дерева** – существование только одного пути, соединяющего любые два узла.

**Определение 2 (равносильно первому).** Дерево (свободное) – неориентированный связный граф без циклов.



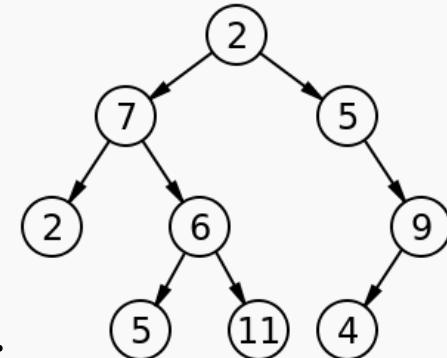
# Определения деревьев



## Определение 3. Дерево с корнем —

дерево, в котором один узел выделен и назначен «корнем» дерева.

Существует только один путь между корнем и каждым из других узлов дерева.



## Определение 4. Высота (глубина) дерева с корнем —

количество вершин в самом длинном пути от корня.

Обычно дерево с корнем рисуют с корнем, расположенным сверху. Узел у располагается под узлом х (а х располагается над у), если х располагается на пути от у к корню.

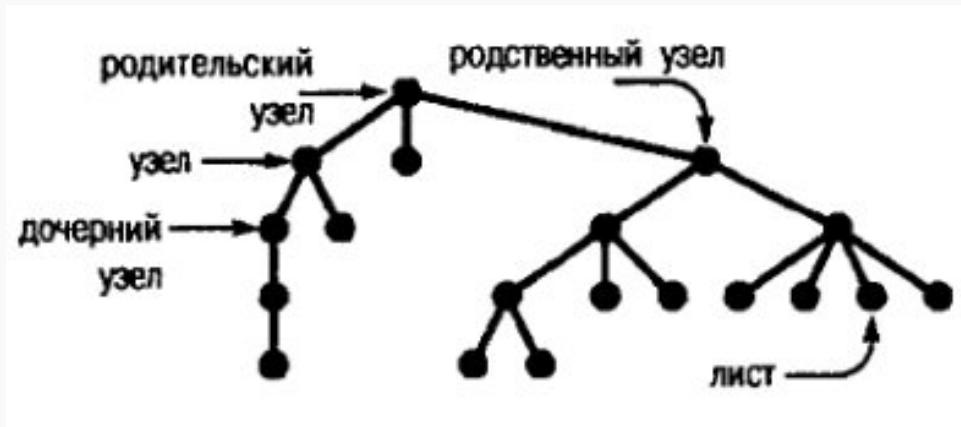
# Определения деревьев



**Определение 5.** Каждый узел (за исключением корня) имеет только один узел, расположенный над ним. Такой узел называется **родительским**.

Узлы, расположенные непосредственно под данным узлом, называются его **дочерними** узлами.

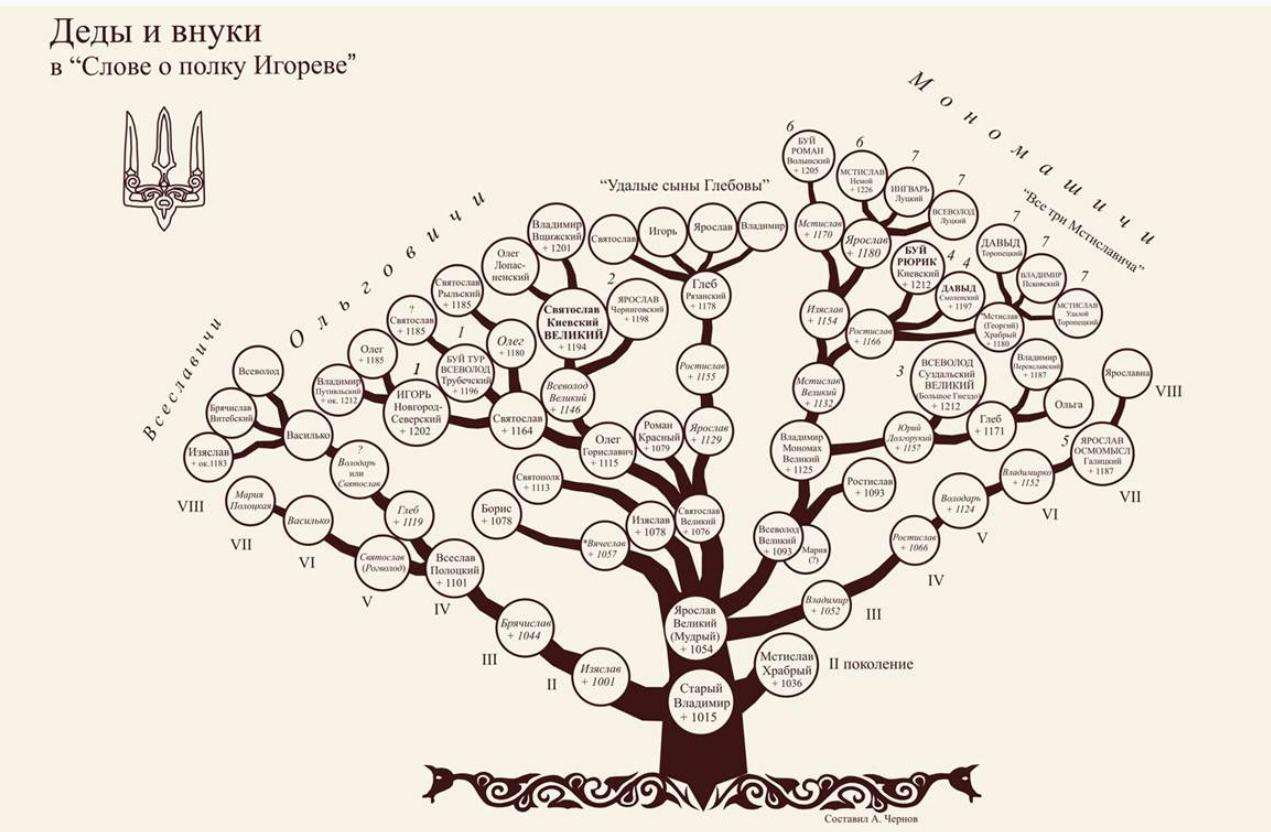
Узлы, не имеющие дочерних узлов называются **листьями**.



# Примеры деревьев



## Генеалогическое дерево



# Примеры деревьев



## XML

```
<?xml version="1.0"?>
- <job>
  - <production>
    <ApprovalType>WebCenter</ApprovalType>
    <Substrate>carton 150 gr</Substrate>
    <SheetSize>220-140</SheetSize>
    <press>SuperFlat2</press>
    <finishing>standard</finishing>
    <urgency>normal</urgency>
  </production>
  - <customer>
    <name>FruitCo</name>
    <number>2712</number>
    <currency>USD</currency>
  </customer>
</job>
```

# Примеры деревьев



## Орг. структура компании



# Примеры деревьев



## Файловая система

The screenshot shows a Windows-style file explorer interface. On the left, a tree view displays the directory structure: E:\GAME\GAMES\ARCON. The ARCON folder is selected. Inside ARCON, there are subfolders: LINE, TETRIS\_P, GOMOKU, and another ARCON folder. Below these are MEHANICA, ARH\_SEM, and ARH\_EC. On the right, a detailed file list is shown for the selected ARCON folder. The columns are labeled: Имя (Name), Размер (Size), and Изменен (Last Modified). The files listed are: .., dos4gw.exe, ke.exe, ke\_main.dig, ke\_end.dig, ke\_tit.dig, ke\_go.dig, ke\_tit.gif, ke\_menu.dig, ke\_score.dig, ke\_font.bob, ke\_brick.bob, ke\_infos.dig, ke\_fill.bob, and ke\_menu.bob.

Имя	Размер	Изменен
..	<UP--DIR	05/13/98 04:5
dos4gw.exe	254556	05/31/94 02:0
ke.exe	210031	07/21/94 09:3
ke_main.dig	119396	01/06/80 01:5
ke_end.dig	94536	01/06/80 01:5
ke_tit.dig	85304	01/06/80 01:5
ke_go.dig	83496	01/06/80 01:5
ke_tit.gif	71668	01/06/80 01:5
ke_menu.dig	65840	01/06/80 01:4
ke_score.dig	65708	01/06/80 01:5
ke_font.bob	62100	01/06/80 01:5
ke_brick.bob	59464	01/06/80 01:5
ke_infos.dig	45864	01/06/80 01:5
ke_fill.bob	42080	01/06/80 01:5
ke_menu.bob	41284	01/06/80 01:5

# Число вершин и ребер



**Утверждение 1.** Любое дерево (с корнем) содержит листовую вершину.

**Доказательство.** Самая глубокая вершина является листовой.

**Утверждение 2.** Дерево, состоящее из  $N$  вершин, содержит  $N - 1$  ребро.

**Доказательство.** По индукции.

База индукции.  $N = 1$ . Одна вершина, ноль ребер.

Шаг индукции. Пусть дерево состоит из  $N + 1$  вершины.

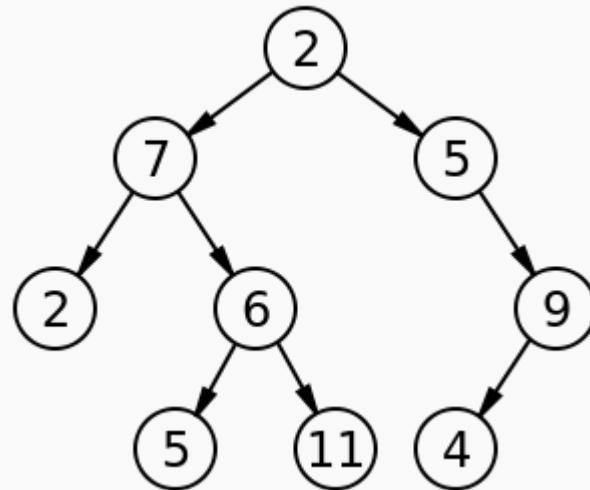
Найдем листовую вершину. Эта вершина содержит ровно 1 ребро. Дерево без этой вершины содержит  $N$  вершин, а по предположению индукции  $N - 1$  ребро. Следовательно, исходное дерево содержит  $N$  ребер, ч.т.д.

# Виды деревьев



**Определение 6а. Двоичное (бинарное) дерево —**  
это дерево, в котором степени вершин не превосходят 3.

**Определение 6б. Двоичное (бинарное) дерево с корнем —** это дерево, в котором каждая вершина имеет не более двух дочерних вершин.

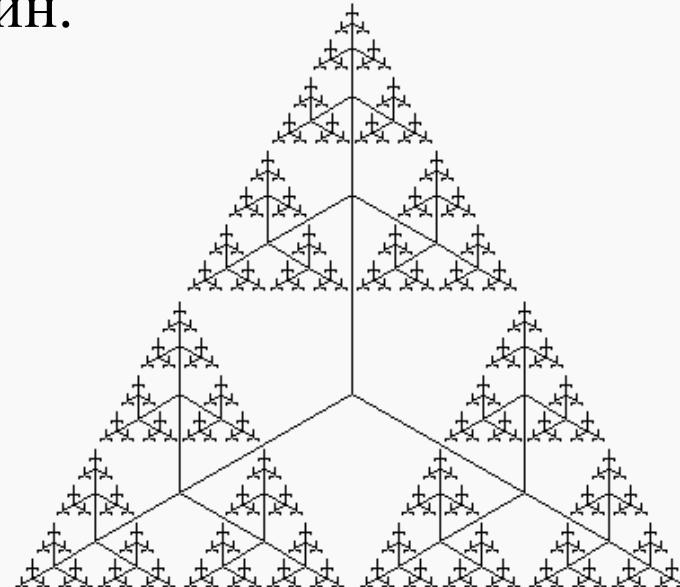


# Виды деревьев



**Определение 7а.** **N-арное дерево** — это дерево, в котором степени вершин не превосходят  $N + 1$ .

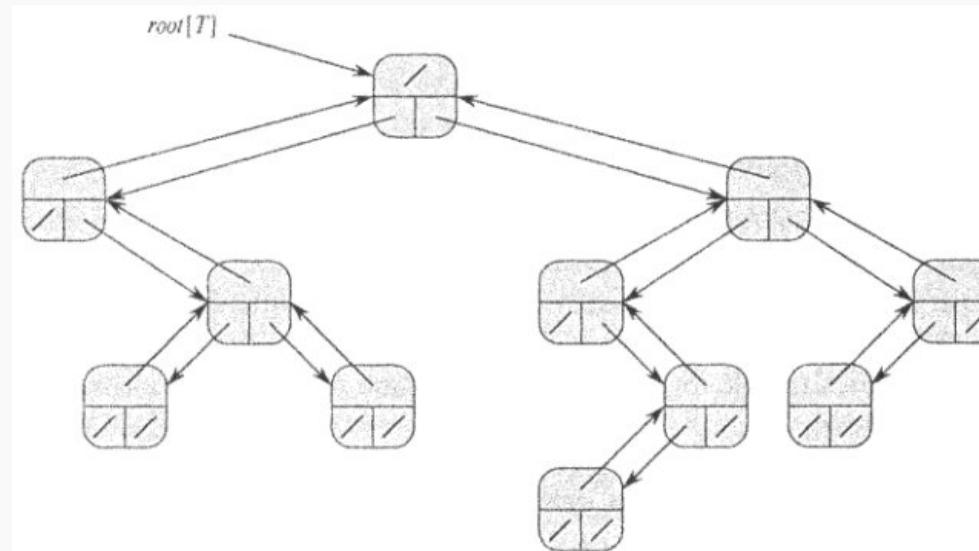
**Определение 7б.** **N-арное дерево с корнем** — это дерево, в котором каждая вершина имеет не более  $N$  дочерних вершин.



## Определение 8. СД «Двоичное дерево» —

представление двоичного дерева с корнем.

Узел – структура, содержащая данные и указатели на левый и правый дочерний узел. Также может содержать указатель на родительский узел.

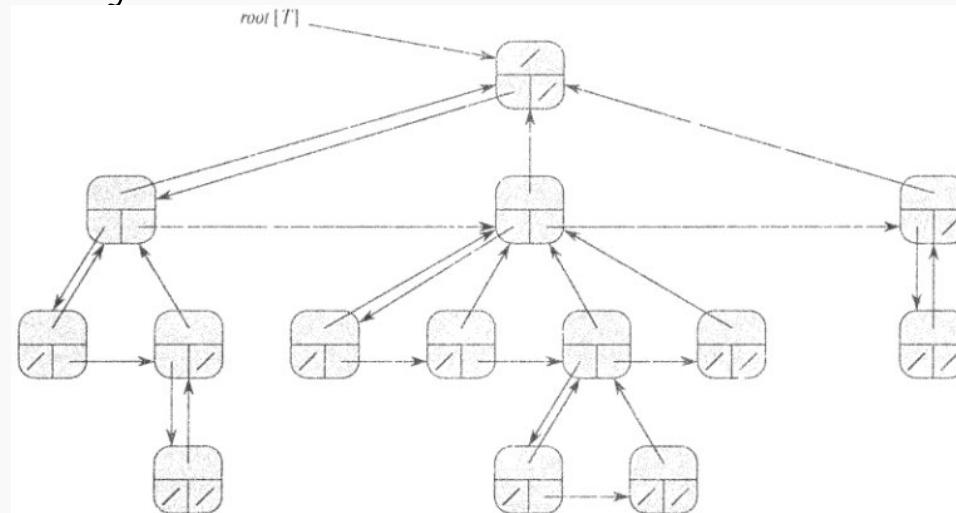


# Структуры данных

## **Определение 9.** СД «N-арное дерево» —

представление  $N$ -арного дерева с корнем.

Узел – структура, содержащая данные, указатель на следующий родственный узел и указатель на первый дочерний узел. Также может содержать указатель на родительский узел.





# Структуры данных



```
// Узел двоичного дерева с данными типа int.  
struct CBinaryNode {  
    int Data;  
    CBinaryNode* Left; // NULL, если нет.  
    CBinaryNode* Right; // NULL, если нет.  
    CBinaryNode* Parent; // NULL, если корень.  
};  
  
// Узел дерева с произвольным ветвлением.  
struct CTREENode {  
    int Data;  
    CTREENode* Next; // NULL, если нет следующих.  
    CTREENode* First; // NULL, если нет дочерних.  
    CTREENode* Parent; // NULL, если корень.  
};
```

# Обход дерева в глубину



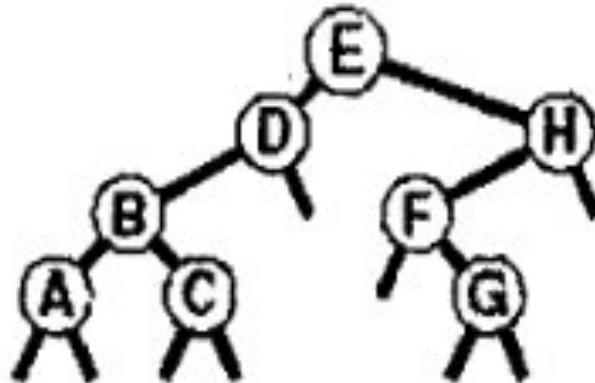
**Определение 10.** Пошаговый перебор элементов дерева по связям между узлами-предками и узлами-потомками называется **обходом дерева**.

**Определение 11. Обходом двоичного дерева в глубину (DFS)** называется процедура, выполняющая в некотором заданном порядке следующие действия с поддеревом:

- \* просмотр (обработка) узла-корня поддерева,
- \* рекурсивный обход левого поддерева,
- \* рекурсивный обход правого поддерева.

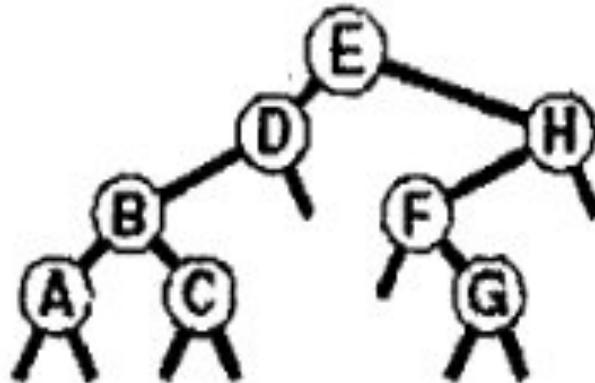
DFS – Depth First Search.

# Обход дерева в глубину



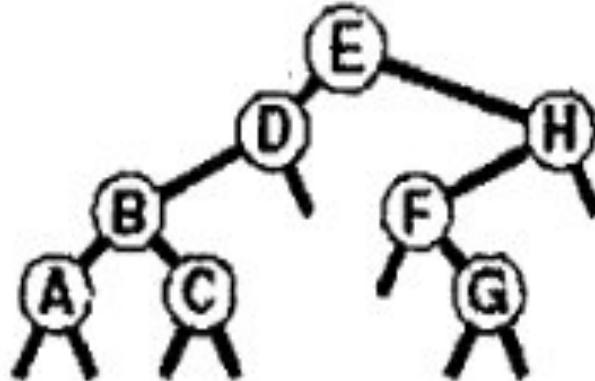
- **Прямой обход (сверху вниз, pre-order).** Вначале обрабатывается узел, затем посещается левое и правые поддеревья.  
Порядок обработки узлов дерева на рисунке:  
E, D, B, A, C, H, F, G.

# Обход дерева в глубину



- **Обратный обход (снизу вверх, post-order).** Вначале посещаются левое и правое поддеревья, а затем обрабатывается узел.  
Порядок обработки узлов дерева на рисунке:  
A, C, B, D, G, F, H, E.

# Обход дерева в глубину



- **Поперечный обход (слева направо, in-order).** Вначале посещается левое поддерево, затем узел и правое поддерево.  
Порядок обработки узлов дерева на рисунке:  
A, B, C, D, E, F, G, H.



# Обход дерева в глубину



```
// Обратный обход в глубину.  
void TraverseDFS( CBinaryNode* node )  
{  
    if( node == 0 )  
        return;  
    TraverseDFS( node->Left );  
    TraverseDFS( node->Right );  
    visit( node );  
};
```

# Обход дерева в глубину



**Задача.** Вычислить количество вершин в дереве.

**Решение.** Обойти дерево в глубину в обратном порядке. После обработки левого и правого поддеревьев вычисляется число вершин в текущем поддереве.

**Реализация:**

```
// Возвращает количество элементов в поддереве.  
int Count( CBinaryNode* node )  
{  
    if( node == 0 )  
        return 0;  
    return Count( node->Left ) + Count( node->Right ) + 1;  
};
```

# Обход дерева в глубину



- **Обход в глубину** не начинает обработку других поддеревьев, пока полностью не обработает текущее поддерево.

Для прохода по слоям в прямом или обратном порядке требуется другой алгоритм.

# Обход дерева в ширину



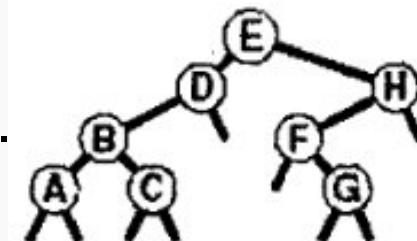
**Определение 12. Обход двоичного дерева в ширину (BFS)** — обход вершин дерева по уровням (слоям), начиная от корня.  
BFS – Breadth First Search.

Используется очередь, в которой хранятся вершины, требующие просмотра.

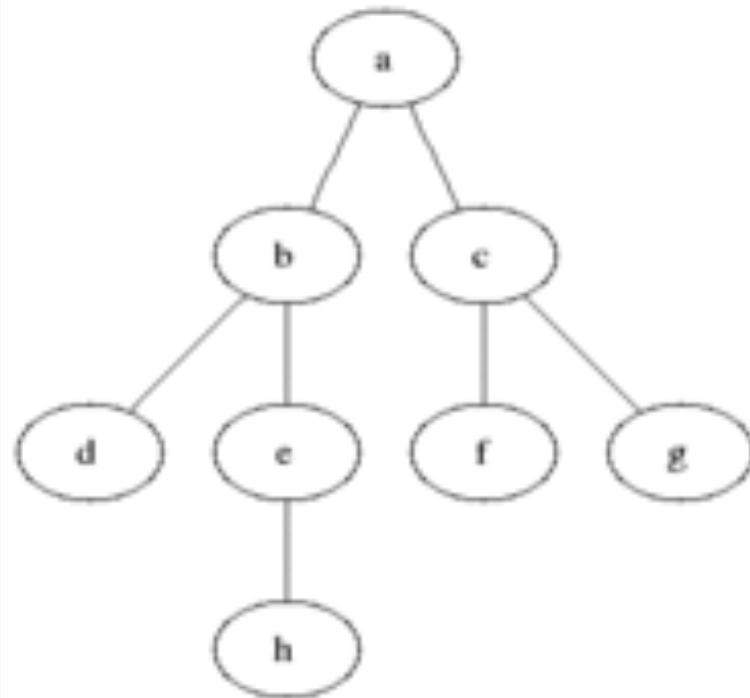
За одну итерацию алгоритма:

- \* если очередь не пуста, извлекается вершина из очереди,
- \* посещается (обрабатывается) извлеченная вершина.
- \* в очередь помещаются все дочерние.

Порядок обработки узлов дерева на рис.  
E, D, H, B, F, A, C, G.



# Обход дерева в ширину





# Обход дерева в ширину



```
// Обход в ширину.  
void TraverseBFS( CBinaryNode* root )  
{  
    if( root == nullptr ) {  
        return;  
    }  
    queue<CBinaryNode*> q;  
    q.put( root );  
    while( !q.empty() ) {  
        CBinaryNode* node = q.pop();  
        visit( node );  
        if( node->Left != nullptr )  
            q.push( node->Left );  
        if( node->Right != nullptr )  
            q.push( node->Right );  
    }  
};
```

# Разница между BFS и DFS



BFS (в ширину)	DFS (в глубину)
<ul style="list-style-type: none"><li>Сложность по памяти – <math>O(w)</math>, где <math>w</math> – максимальная ширина дерева (максимальная ширина на глубине <math>h = 2^h</math>). Экономичнее в несбалансированных деревьях.</li><li>Эффективнее, если при обходе нас интересуют элементы, находящиеся близко к корню.</li></ul>	<ul style="list-style-type: none"><li>Сложность по памяти – <math>O(h)</math>, где <math>h</math> – максимальная глубина дерева (максимальная глубина в дереве из <math>n</math> элементов = <math>n</math>). Экономичнее в сбалансированных деревьях.</li><li>Эффективнее, если при обходе нас интересуют элементы, находящиеся близко к листьям.</li></ul>

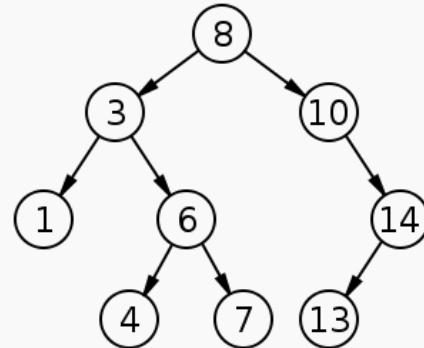
# Двоичные деревья поиска



**Определение 13.** Двоичное дерево поиска (binary search tree, BST) – это двоичное дерево, с каждым узлом которого связан ключ, и выполняется следующее дополнительное условие:

- Ключ в любом узле X больше или равен ключам во всех узлах левого поддерева X и меньше или равен ключам во всех узлах правого поддерева X.

**Пример:**



# Двоичные деревья поиска



Операции с двоичным деревом поиска:

1. Поиск по ключу.
2. Поиск минимального, максимального ключей.
3. Вставка.
4. Удаление.
5. Обход дерева в порядке возрастания ключей.

# Двоичные деревья поиска



## Поиск по ключу.

Дано: указатель на корень дерева X и ключ K.

Задача: проверить, есть ли узел с ключом K в дереве, и если да, то вернуть указатель на этот узел.

Алгоритм: Если дерево пусто, сообщить, что узел не найден, и остановиться.

Иначе сравнить K со значением ключа корневого узла X.

- Если  $K == X$ , выдать ссылку на этот узел и остановиться.
- Если  $K > X$ , рекурсивно искать ключ K в правом поддереве X.
- Если  $K < X$ , рекурсивно искать ключ K в левом поддереве X.

Время работы:  $O(h)$ , где h – глубина дерева.



# Двоичные деревья поиска



```
// Поиск. Возвращает узел с заданным ключом. NULL, если
узла
// с таким ключом нет.
CNode* Find( CNode* node, int value )
{
    if( node == NULL )
        return NULL;
    if( node->Data == value )
        return node;
    if( node->Data > value )
        return Find( node->Left, value );
    else
        return Find( node->Right, value );
};
```

# Двоичные деревья поиска



## Поиск минимального ключа.

Дано: указатель на корень непустого дерева X.

Задача: найти узел с минимальным значением ключа.

Алгоритм: Переходить в левый дочерний узел, пока такой существует.

Время работы:  $O(h)$ , где  $h$  – глубина дерева.



# Двоичные деревья поиска



```
// Поиск узла с минимальным ключом.  
CNode* FindMinimum( CNode* node )  
{  
    assert( node != NULL );  
    while( node->Left != NULL )  
        node = node->Left;  
    return node;  
};
```

# Двоичные деревья поиска



## Добавление узла.

Дано: указатель на корень дерева X и ключ K.

Задача: вставить узел с ключом K в дерево (возможно появление дубликатов).

Алгоритм: Если дерево пусто, заменить его на дерево с одним корневым узлом и остановиться.

Иначе сравнить K с ключом корневого узла X.

- Если  $K < X$ , рекурсивно добавить K в левое поддерево X.
- Иначе рекурсивно добавить K в правое поддерево X.

Время работы:  $O(h)$ , где  $h$  – глубина дерева.



# Двоичные деревья поиска



```
// Вставка. Не указываем parent.
void Insert( CNode*& node, int value )
{
    if( node == NULL ) {
        node = new CNode( value );
        return;
    }
    if( node->Data > value )
        Insert( node->Left, value );
    else
        Insert( node->Right, value );
};
```

# Двоичные деревья поиска



## Удаление узла.

Дано: указатель на корень дерева  $X$  и ключ  $K$ .

Задача: удалить из дерева узел с ключом  $K$  (если такой есть).

Алгоритм: Если дерево пусто, остановиться.

Иначе сравнить  $K$  с ключом корневого узла  $X$ .

- Если  $K < X$ , рекурсивно удалить  $K$  из левого поддерева  $T$ .
- Если  $K > X$ , рекурсивно удалить  $K$  из правого поддерева  $T$ .
- Если  $K == X$ , то необходимо рассмотреть три случая:
  1. Обоих дочерних нет. Удаляем узел  $X$ , обнуляем ссылку.
  2. Одного дочернего нет. Переносим дочерний узел в  $X$ , удаляем узел.
  3. Оба дочерних узла есть.

# Двоичные деревья поиска

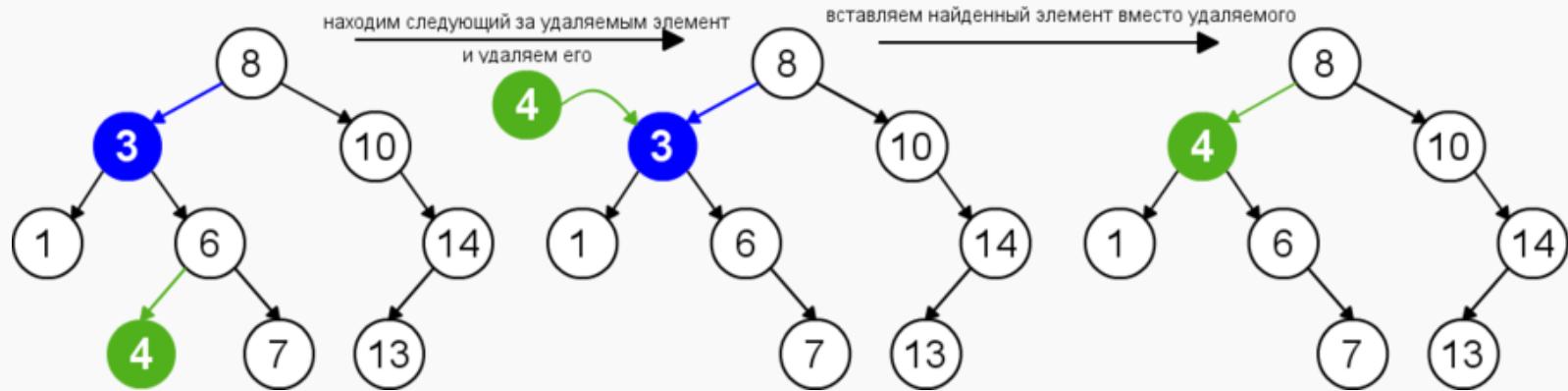


**Удаление узла. Случай 3.** Есть оба дочерних узла. Заменяем ключ удаляемого узла на ключ минимального узла из правого поддерева, удаляя последний.

Пусть удаляемый узел – X, а Y – его правый дочерний.

- Если у узла Y отсутствует левое поддерево, то копируем из Y в X ключ и указатель на правый узел. Удаляем Y.
- Иначе найдем минимальный узел Z в поддереве Y. Копируем ключ из Z, удаляем Z. При удалении Z копируем указатель на правый дочерний узел Z в левый дочерний узел родителя Z.

Время работы удаления:  $O(h)$ , где  $h$  – глубина дерева.





# Двоичные деревья поиска



```
// Удаление. Возвращает false, если нет узла с заданным ключом.  
bool Delete( CNode*& node, int value )  
{  
    if( node == 0 )  
        return false;  
    if( node->Data == value ) { // Нашли, удаляем.  
        DeleteNode( node );  
        return true;  
    }  
    return Delete( node->Data > value ?  
                  node->Left : node->Right, value );  
};
```



# Двоичные деревья поиска



```
// Удаление узла.
void DeleteNode( CNode*& node )
{
    if( node->Left == 0 ) { // Если нет левого поддерева.
        CNode* right = node->Right; // Подставляем правое, может быть 0.
        delete node;
        node = right;
    } else if( node->Right == 0 ) { // Если нет правого поддерева.
        CNode* left = node->Left; // Подставляем левое.
        delete node;
        node = left;
    } else { // Есть оба поддерева.
        // Ищем минимальный элемент в правом поддереве и его родителя.
        CNode* minParent = node;
        CNode* min = node->Right;
        while( min->Left != 0 ) {
            minParent = min;
            min = min->Left;
        }
        // Переносим значение.
        node->Data = min->Data;
        // Удаляем min, подставляя на его место min->Right.
        (minParent->Left == min ? minParent->Left : minParent->Right)
            = min->Right;
        delete min;
    }
}
```

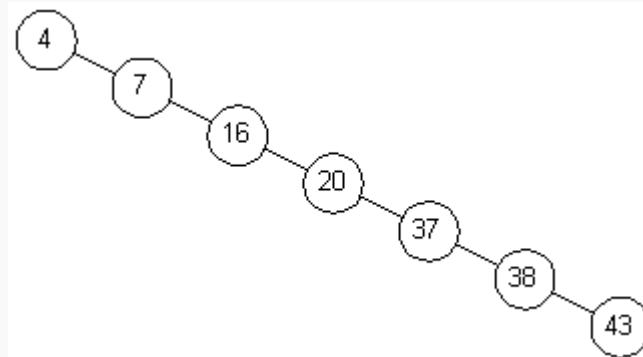
# Балансировка



Все перечисленные операции с деревом поиска выполняются за  $O(h)$ , где  $h$  – глубина дерева.

Глубина дерева может достигать  $n$ .

Последовательное добавление возрастающих элементов вырождает дерево в цепочку:



**Необходима балансировка.**

Определение. АВЛ-дерево — сбалансированное двоичное дерево поиска. Для каждой его вершины высоты её двух поддеревьев различаются не более чем на 1.

Изобретено Адельсон-Вельским Г.М. и Ландисом Е.М. в 1962г.

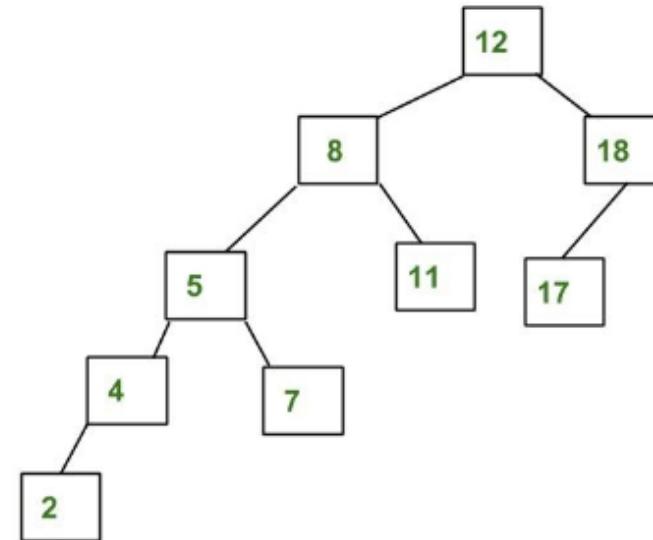
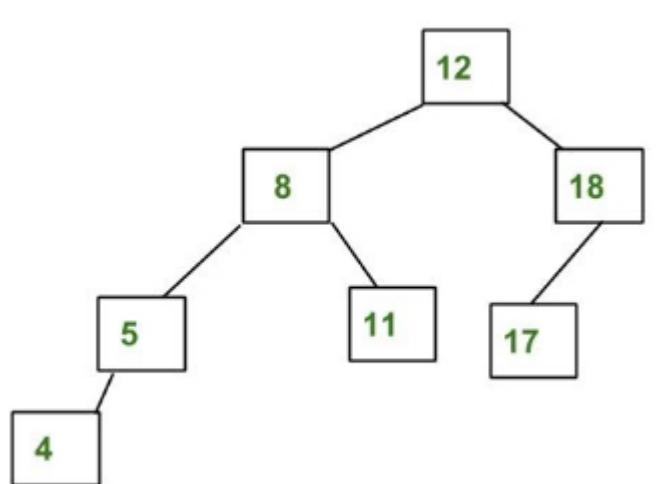
<https://habrahabr.ru/post/150732/> - здесь все хорошо написано, кроме удаления.

# AVL-дерево



AVL дерево

Не AVL дерево



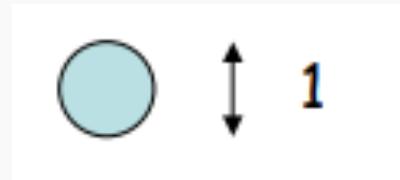
# АВЛ-дерево



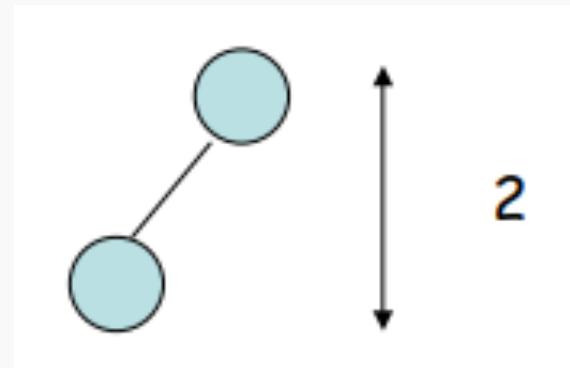
**Теорема.** Высота АВЛ-дерева  $h = O(\log N)$ .

Рассмотрим  $n(h)$ : минимальное число узлов в АВЛ-дереве высоты  $h$ .

$$n(1) = 1$$



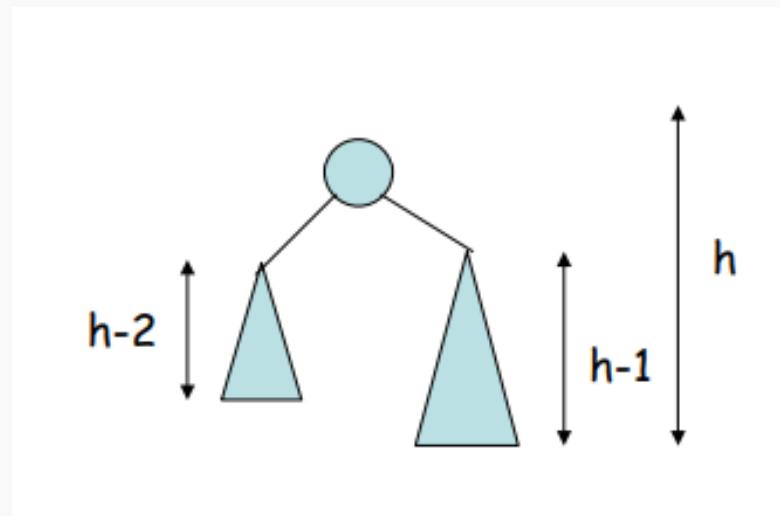
$$n(2) = 2$$



# АВЛ-дерево



Для  $n \geq 3$ , АВЛ-дерево с минимальным числом узлов высоты  $h$  состоит из корня, АВЛ-поддерева высоты  $h-1$  и АВЛ-поддерева высоты  $h-2$ .



Таким образом,  $n(h) = 1 + n(h-1) + n(h-2)$

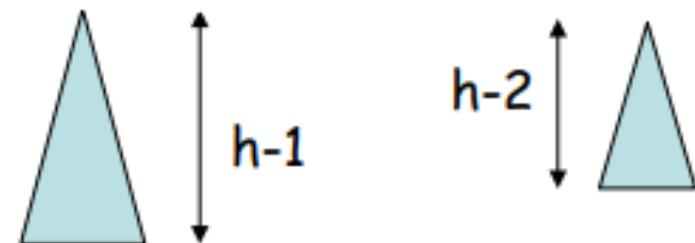
# АВЛ-дерево



$$n(h) = 1 + n(h-1) + n(h-2)$$

Так как  $n(h-1) > n(h-2)$ , то

$$n(h) > 1 + n(h-2) + n(h-2)$$



Таким образом,  $n(h) > 2n(h-2)$

Заметим, что

$$n(h-2) > 2n(h-4)$$

$$n(h-4) > 2n(h-6)$$

...

То есть  $n(h) > 2^i n(h-2i)$

# АВЛ-дерево



$$n(h) > 2^i n(h-2i)$$

При  $i = h/2 - 1$  получается  $n(h) > 2^{h/2-1} * n(1) = 2^{h/2-1}$

Прологарифмируем обе части неравенства:

$$\log(n(h)) > h/2 - 1$$

$$h < 2\log(n(h)) + 2$$

То есть, высота АВЛ-дерева  $h = O(\log N)$

Специальные балансирующие операции, восстанавливающие основное свойство «высоты двух поддеревьев различаются не более чем на 1» – вращения.

- Малое правое вращение,
- Малое левое вращение,
- Большое правое вращение.
- Большое левое вращение,

# AVL-дерево



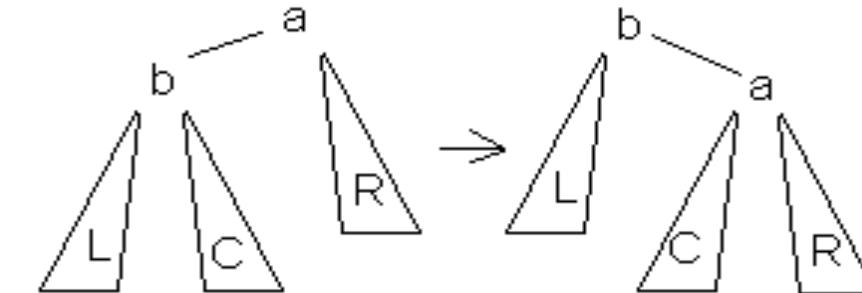
## ▪ Малое правое вращение

Используется, когда:

$$\text{высота}(L) = \text{высота}(R) + 2 \text{ и } \text{высота}(C) \leq \text{высота}(L).$$

После операции:

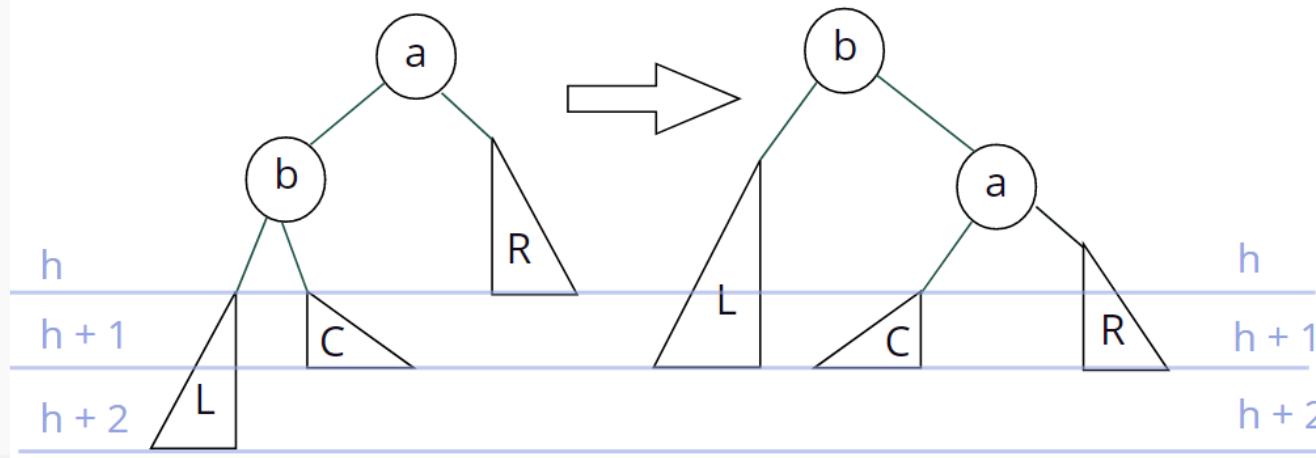
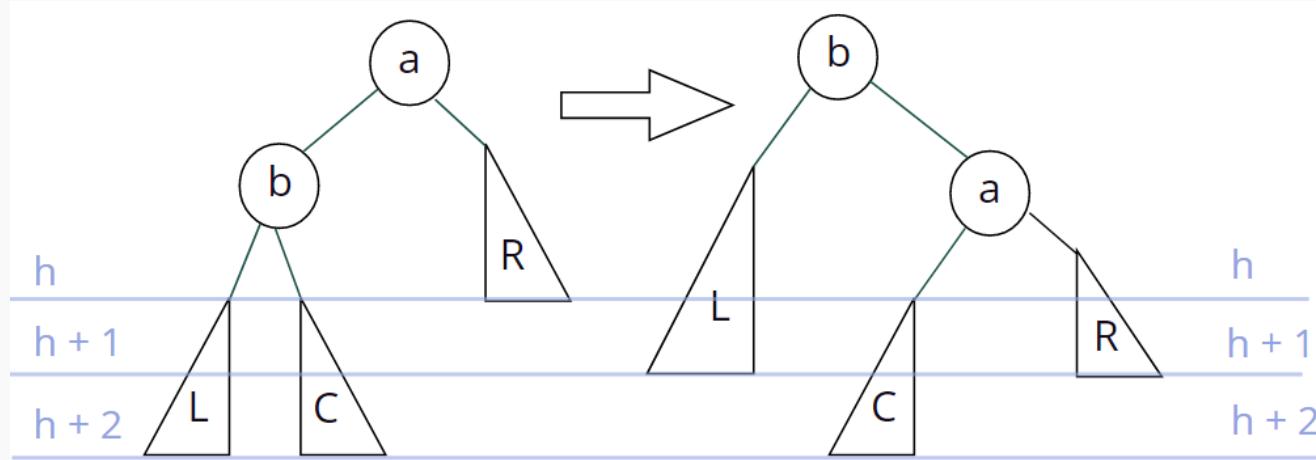
высота дерева останется прежней, если  $\text{высота}(C) = \text{высота}(L)$ ,  
высота дерева уменьшится на 1, если  $\text{высота}(C) < \text{высота}(L)$ .



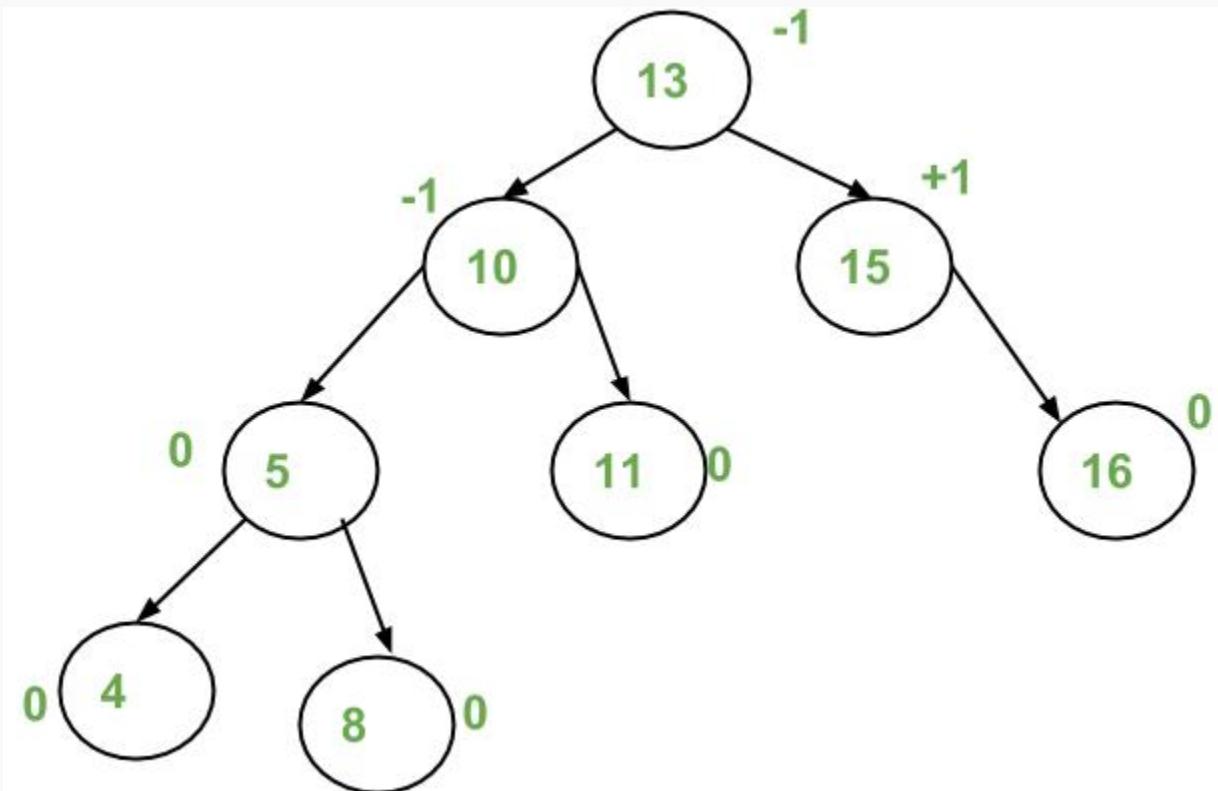
# АВЛ-дерево



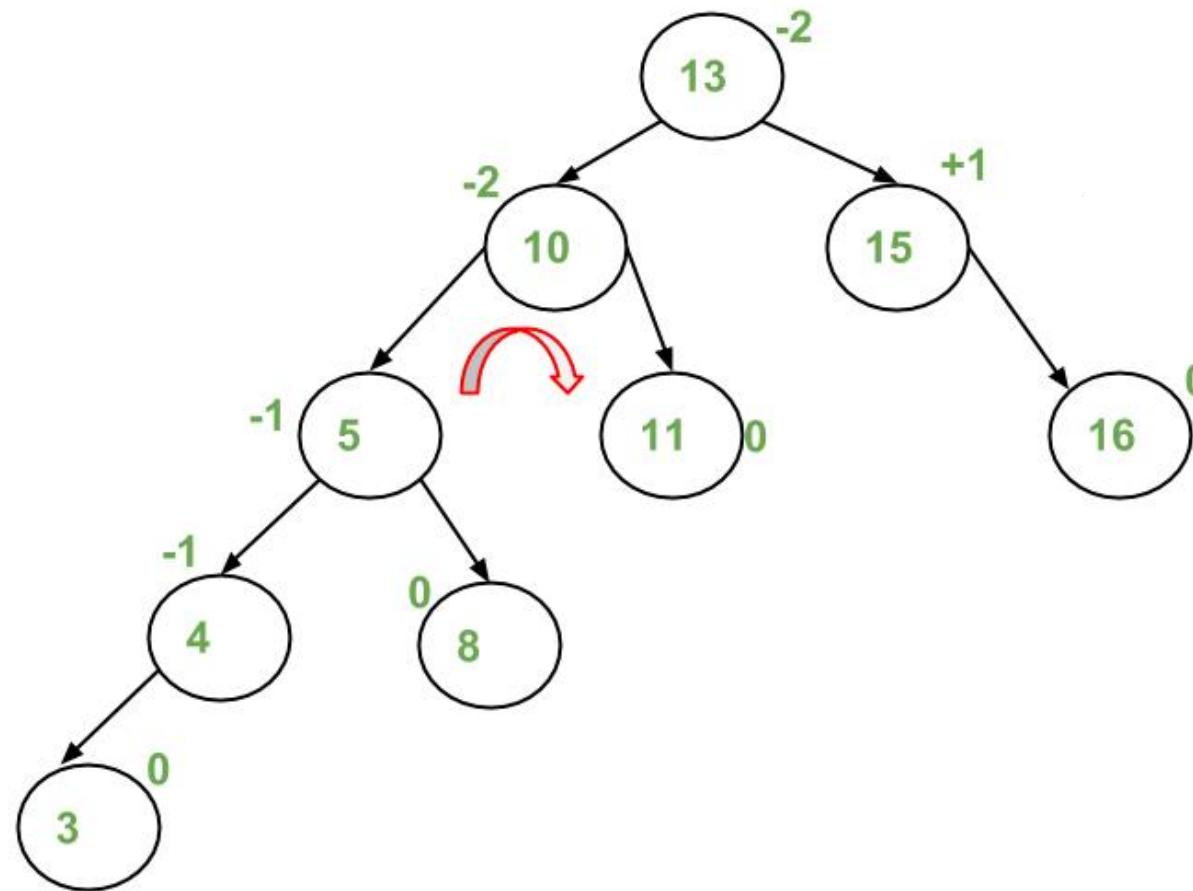
Как малое правое вращение устраниет дисбаланс высот



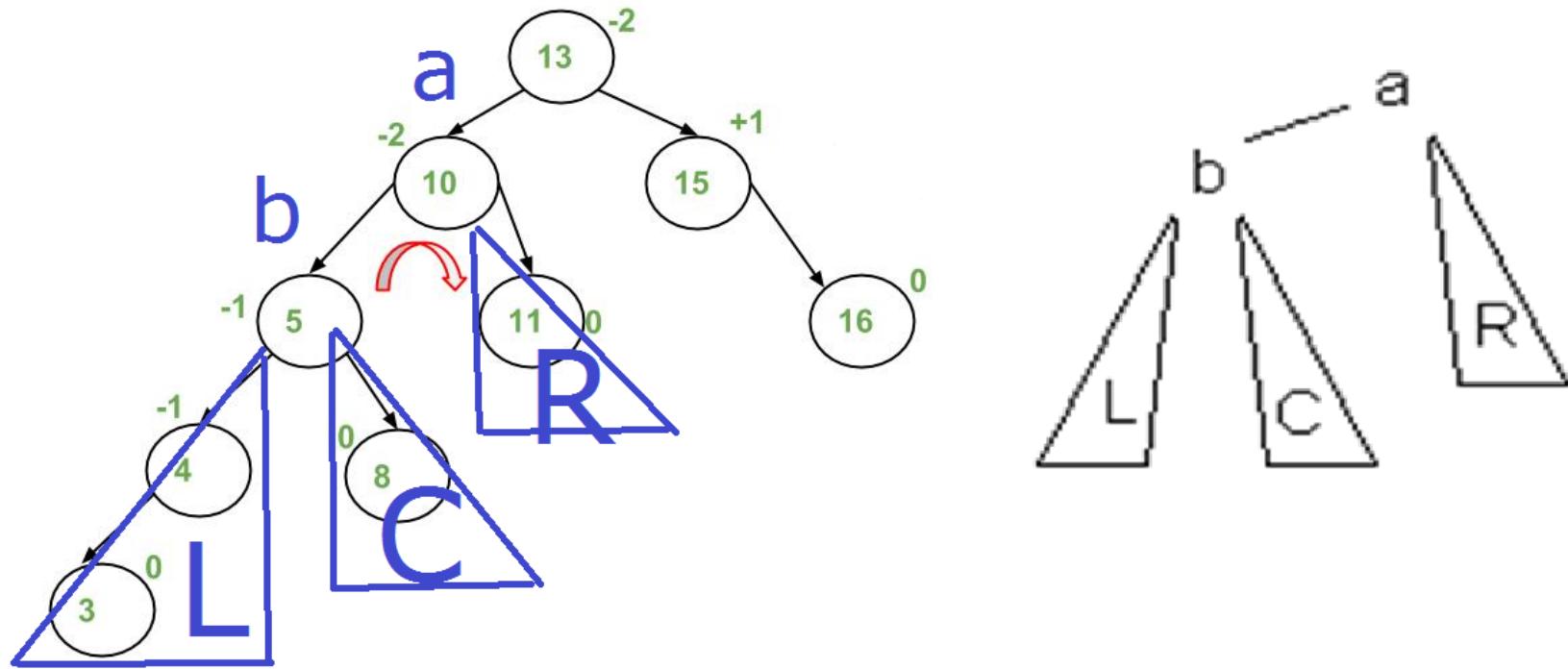
Добавим элемент 3



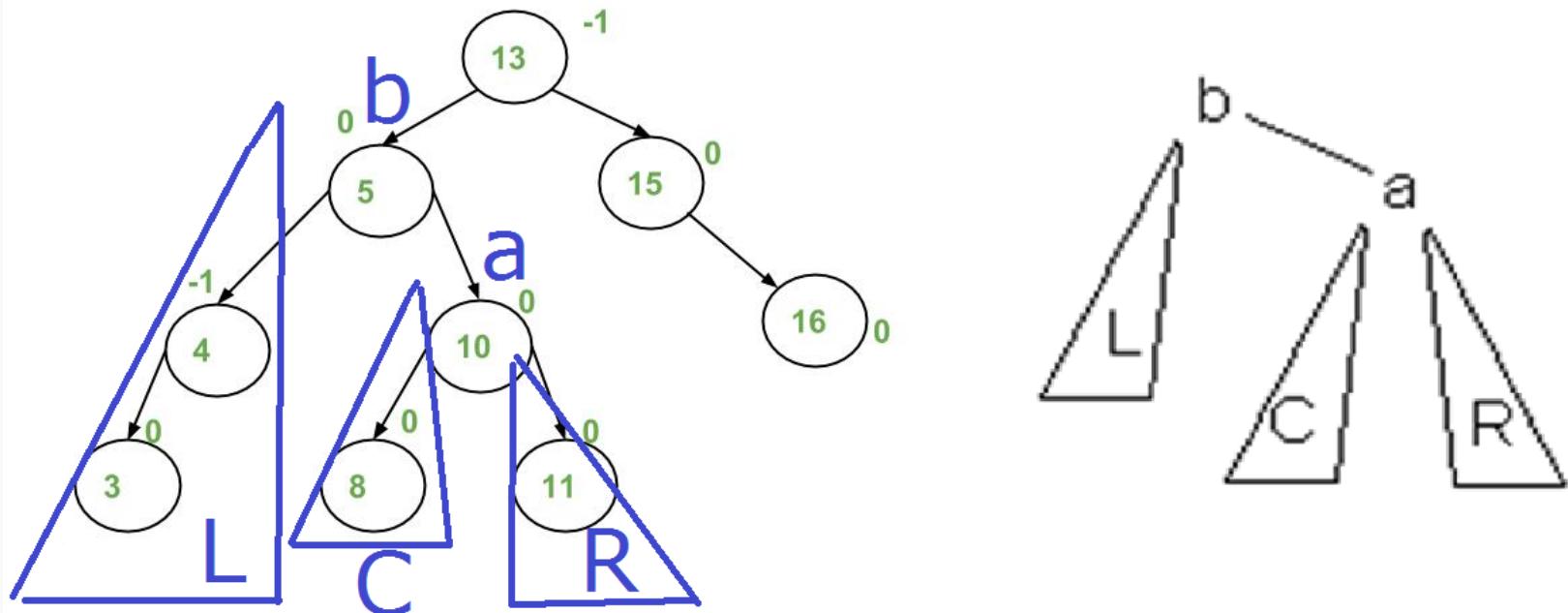
Больше не AVL дерево. Нужно балансировать.



## Делаем малый правый поворот



## Опять стало AVL деревом



# АВЛ-дерево



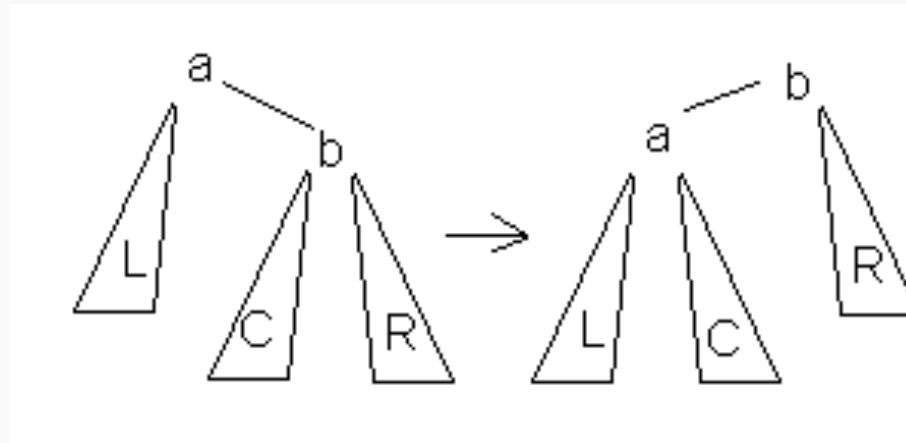
- **Малое левое вращение**

Используется, когда:

высота(R) = высота(L) + 2 и высота(C) ≤ высота(R).

После операции:

высота дерева останется прежней, если высота(C) = высота(R),  
высота дерева уменьшится на 1, если высота(C) < высота(R).



# АВЛ-дерево



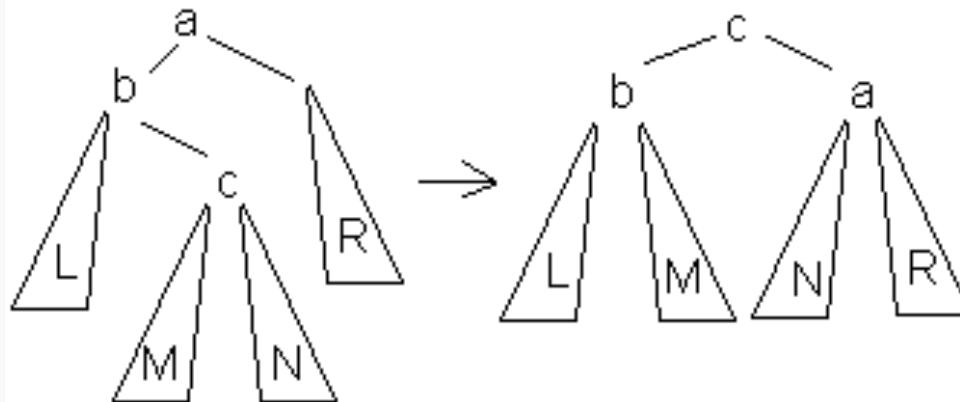
- Большое правое вращение**

Используется, когда:

высота(L) = высота(R) + 1 и высота(C) = высота(R) + 2.

После операции:

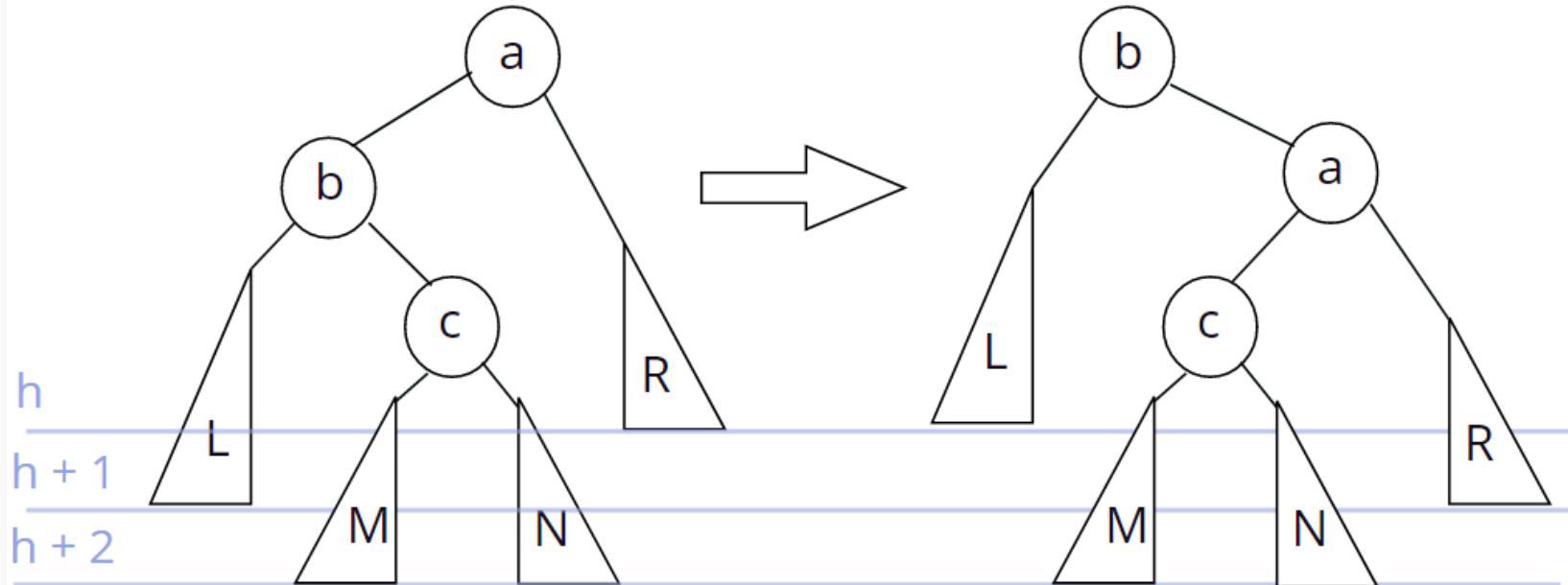
высота дерева уменьшается на 1.



# AVL-дерево



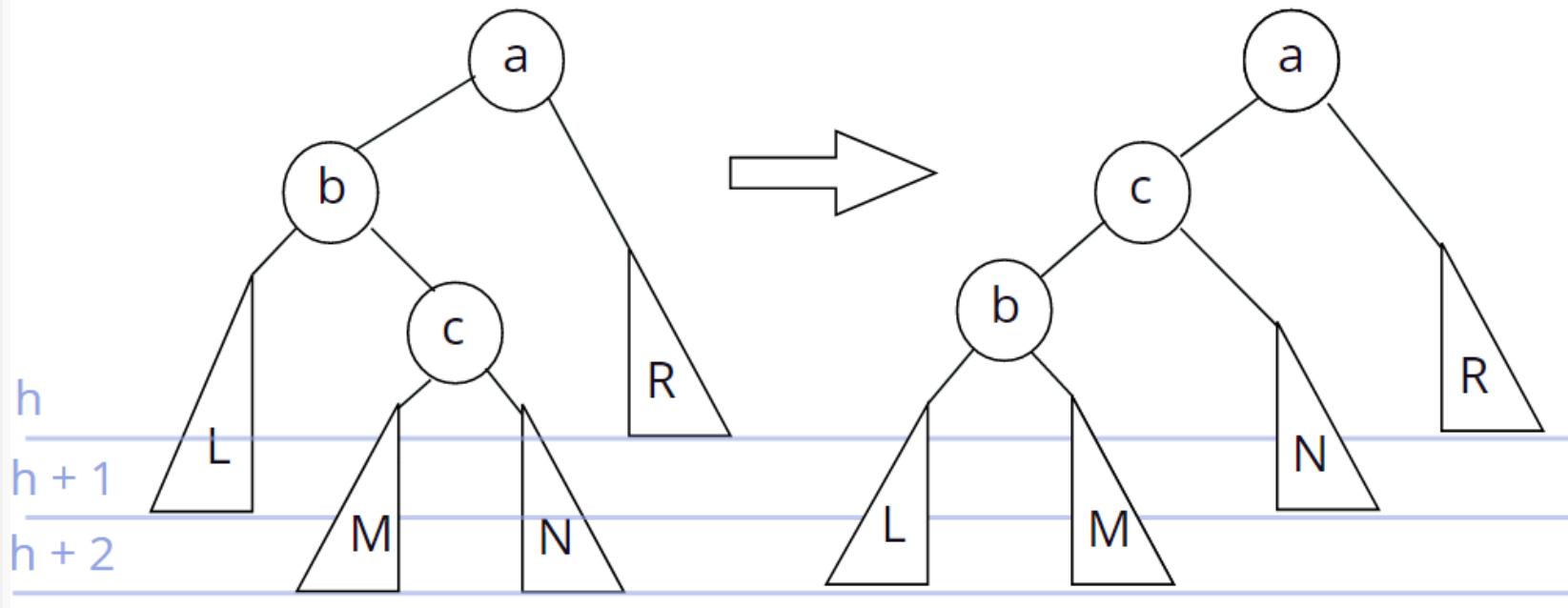
**Большое правое вращение. Почему малое правое вращение не поможет.**



Разницу высот устранить не смогли! Высота( $L$ ) – высота( $c$ ) = 2

## Большое правое вращение.

Шаг 1. Малое левое вращение в b

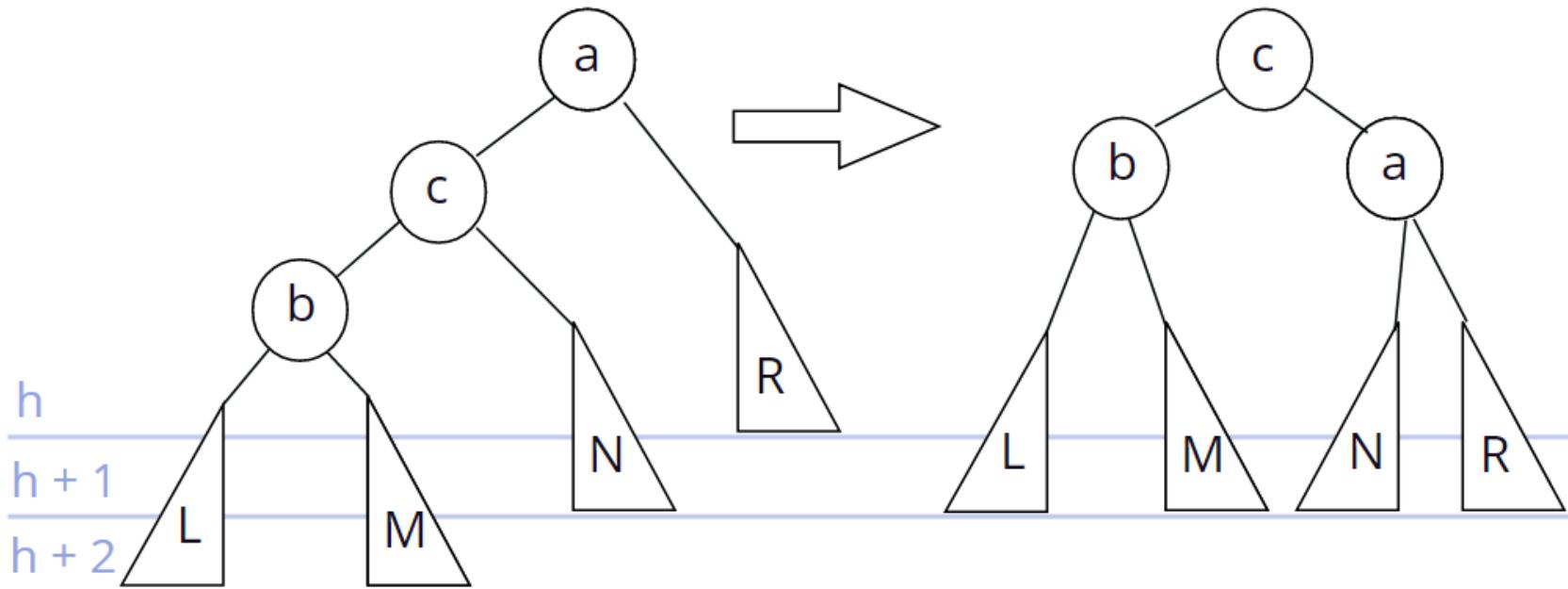


Также возможны ситуации:

Высота(**M**) =  $h + 1$ , Высота(**N**) =  $h + 2$   
Высота(**M**) =  $h + 2$ , Высота(**N**) =  $h + 1$

## Большое правое вращение.

Шаг 2. Малое правое вращение в а

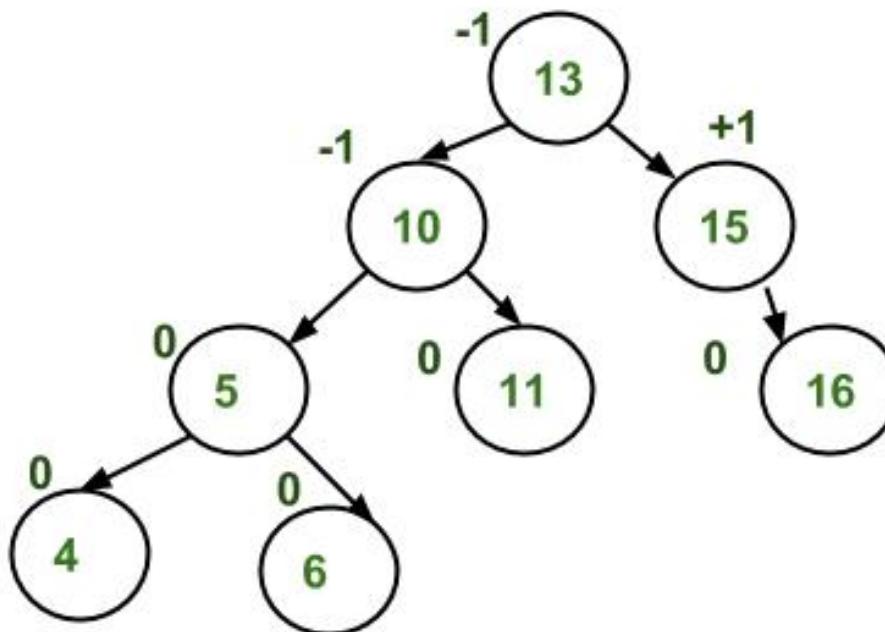


Если в изначальном дереве высота(M) была  $h + 1$ , то теперь она  $h$   
Если в изначальном дереве высота(N) была  $h + 1$ , то теперь она  $h$   
В любом случае, мы исправили дисбаланс и уменьшили высоту на 1

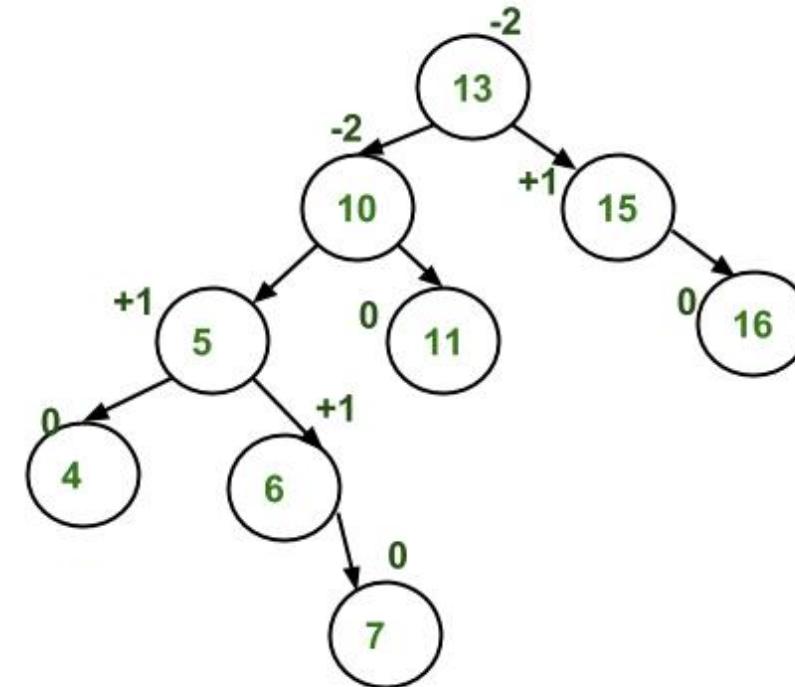
# АВЛ-дерево



Добавим элемент 7



Нарушили баланс. Больше не AVL дерево.

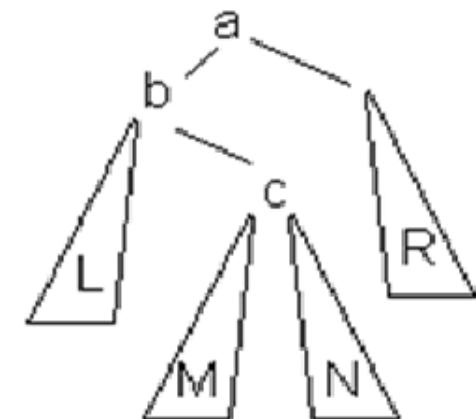
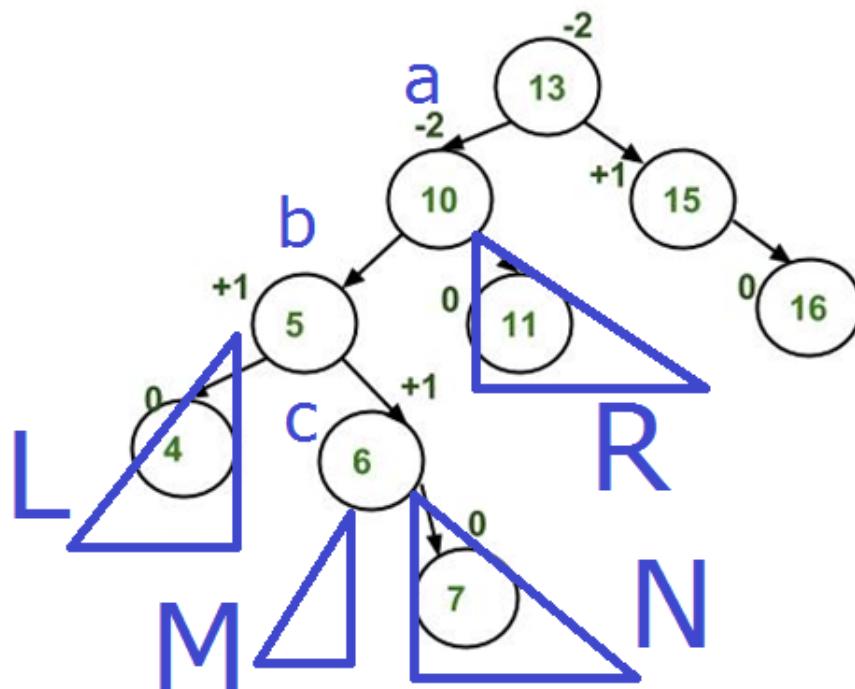


# AVL-дерево

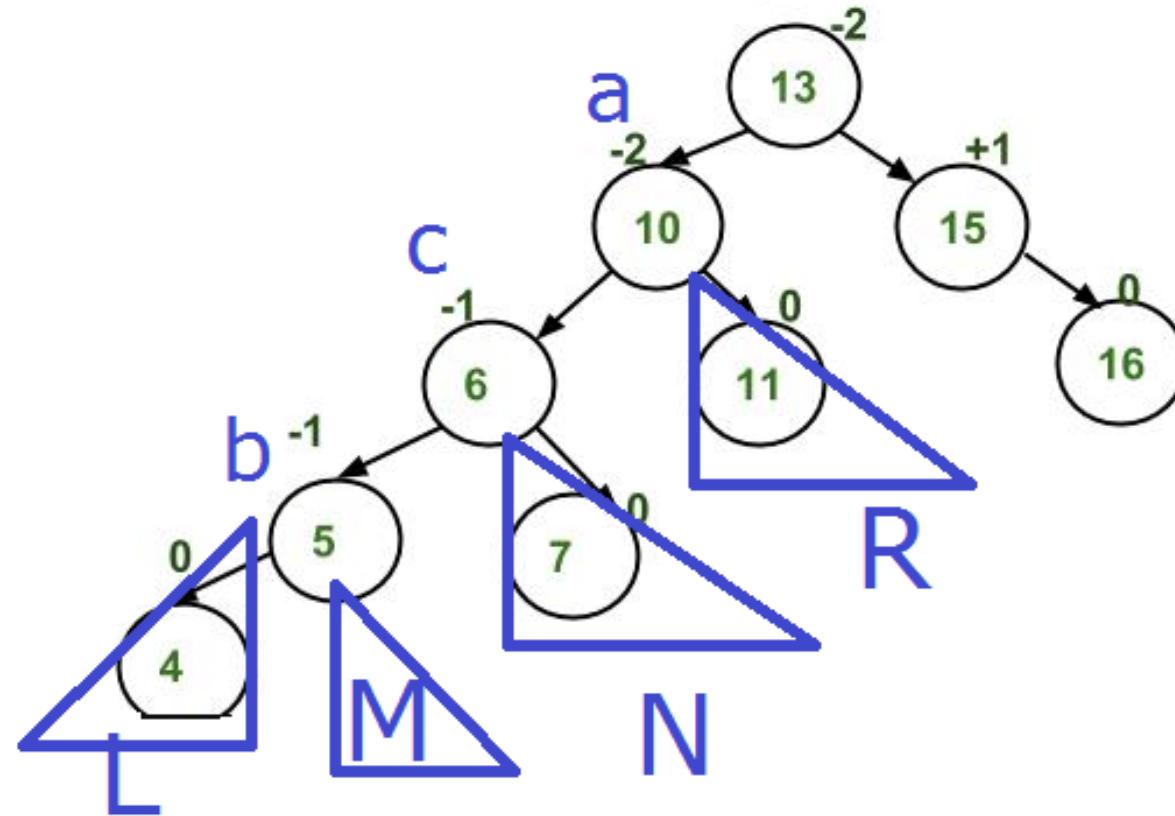


**Необходимо большое правое вращение:**

1. Малое левое вращение в b.
2. Малое правое вращение в a.



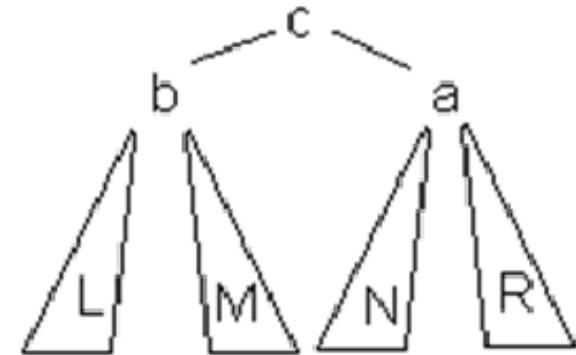
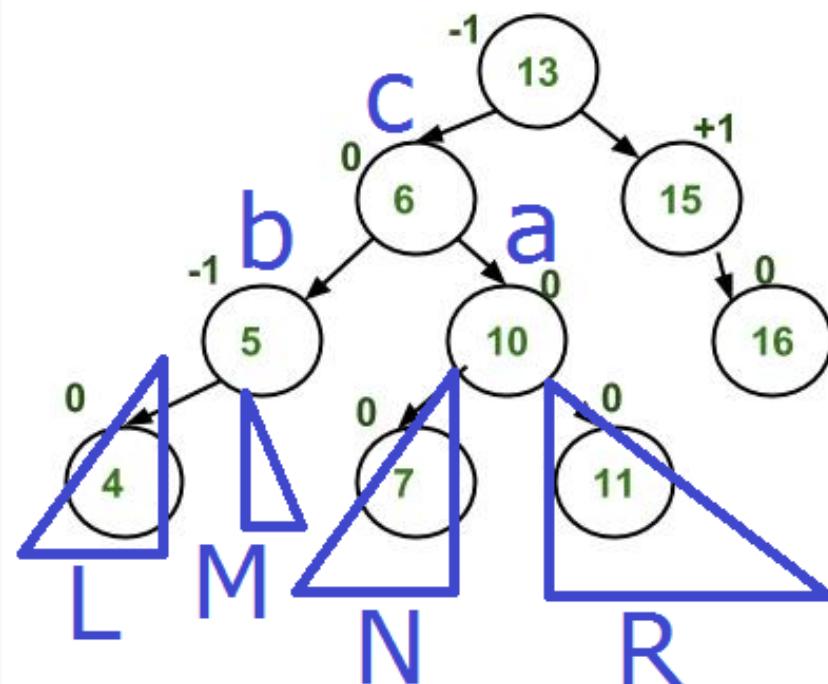
Сделали малое левое вращение



# AVL-дерево



Сделали малое правое вращение. Снова AVL дерево.



# AVL-дерево



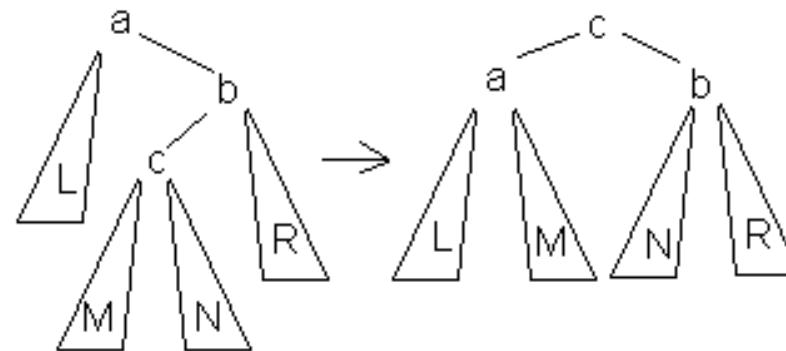
- Большое левое вращение**

Используется, когда:

$$\text{высота}(R) = \text{высота}(L) + 1 \text{ и } \text{высота}(C) = \text{высота}(L) + 2.$$

После операции:

высота дерева уменьшается на 1.



## Вставка элемента

1. Проходим по пути поиска, пока не убедимся, что ключа в дереве нет.
2. Включаем новую вершину как в стандартной операции вставки в дерево поиска.
3. "Отступаем" назад от добавленной вершины к корню. Проверяем в каждой вершине сбалансированность. Если разность высот поддеревьев равна 2 – выполняем нужное вращение.

Время работы =  $O(\log n)$ .

## Удаление элемента

1. Ищем вершину D, которую требуется удалить.
2. Проверяем, сколько поддеревьев в D:
  - Если D – лист или D имеет одно поддерево, то удаляем D.
  - Если D имеет два под дерева, то ищем вершину M, следующую по значению после D. Как в стандартном алгоритме удаления из дерева поиска. Переносим значение из M в D. Удаляем M.
3. "Отступаем" назад от удаленной вершины к корню.  
Проверяем в каждой вершине сбалансированность. Если разность высот поддеревьев равна 2 – выполняем нужное вращение.

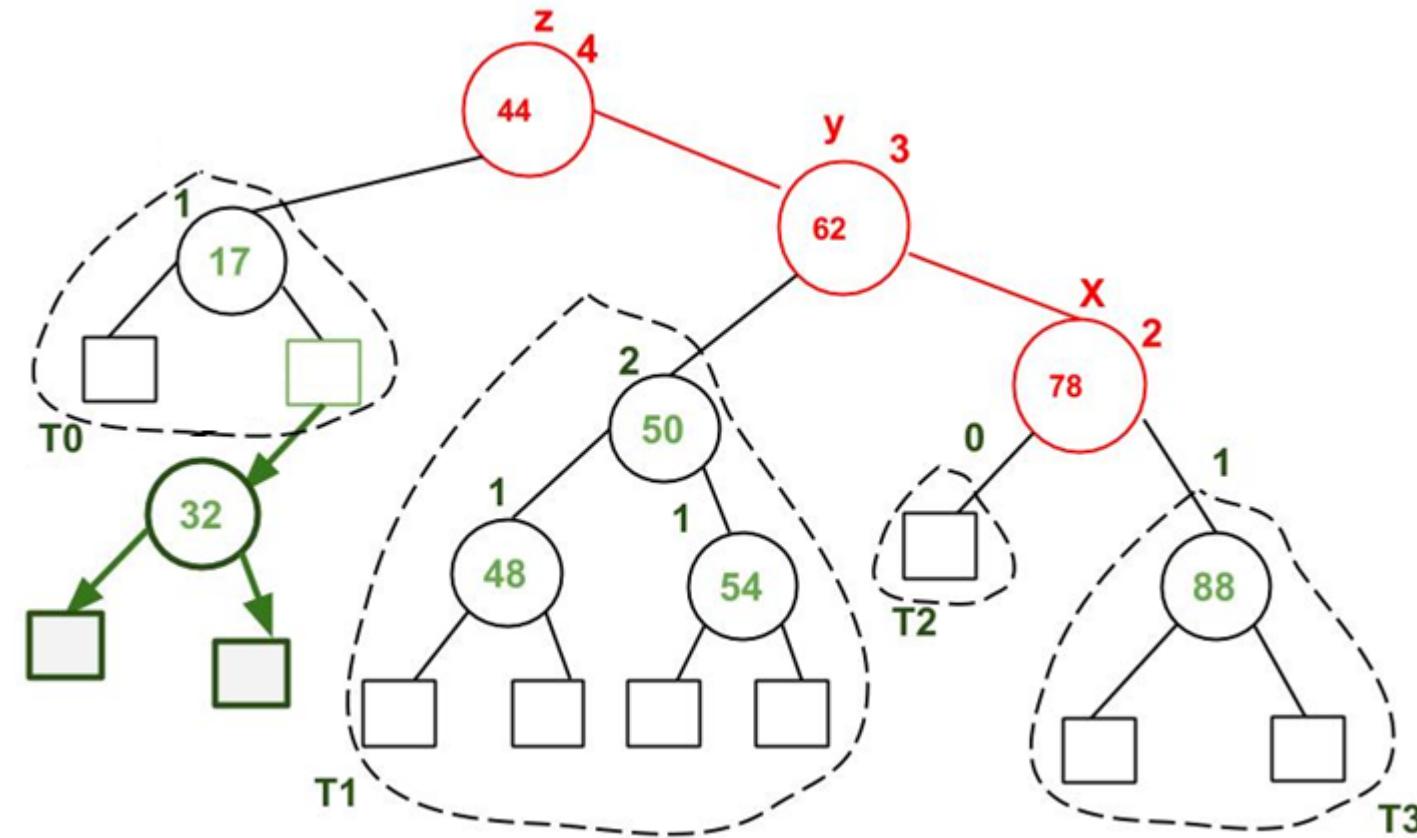
Время работы =  $O(\log n)$ .

# АВЛ-дерево



Удаляем элемент 32.

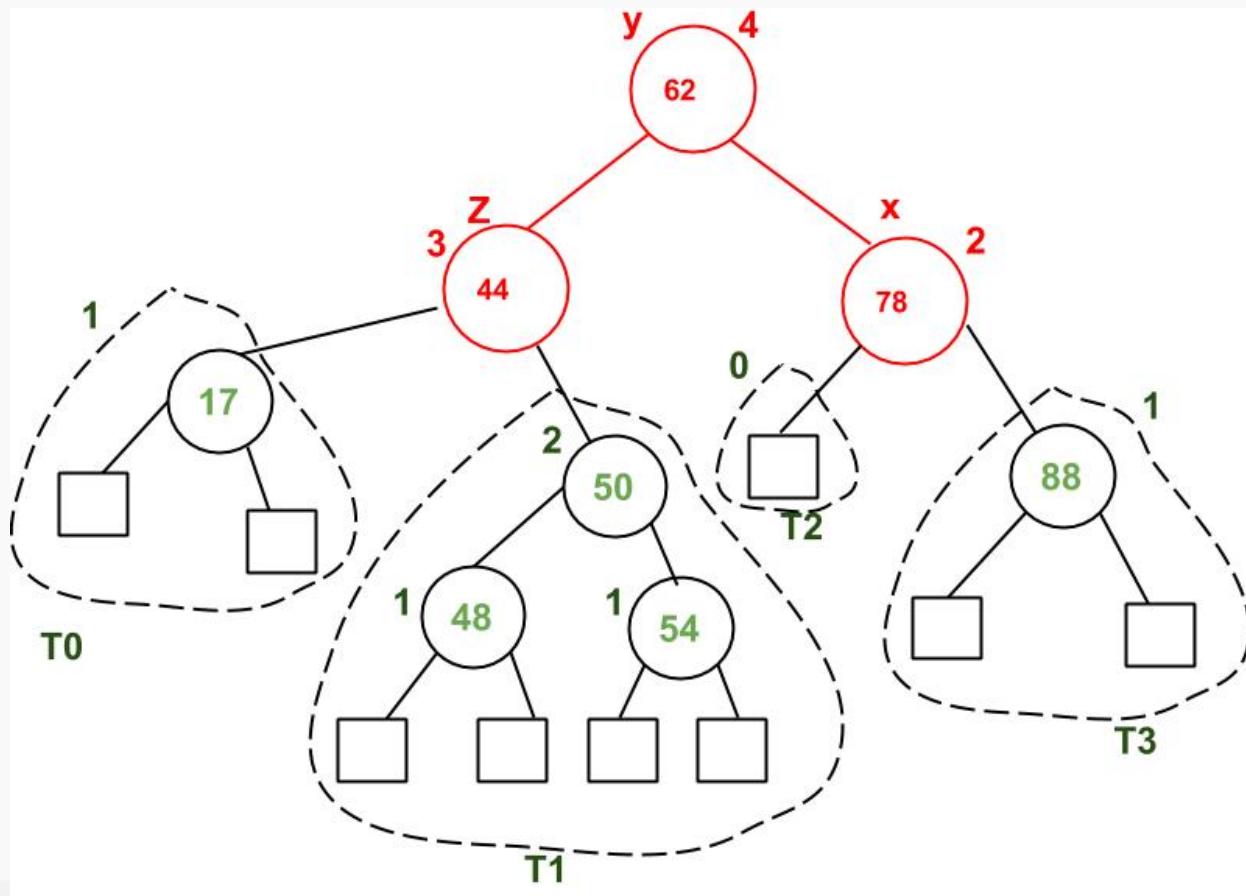
Высота T0 была 2, теперь 1. Нужно малое левое вращение в z.



# AVL-дерево



Восстановили сбалансированность



## Расход памяти и время работы.

	В среднем случае	В худшем случае
Расход памяти	$O(n)$	$O(n)$
Поиск	$O(\log(n))$	$O(\log(n))$
Вставка	$O(\log(n))$	$O(\log(n))$
Удаление	$O(\log(n))$	$O(\log(n))$

# Красно-чёрные деревья

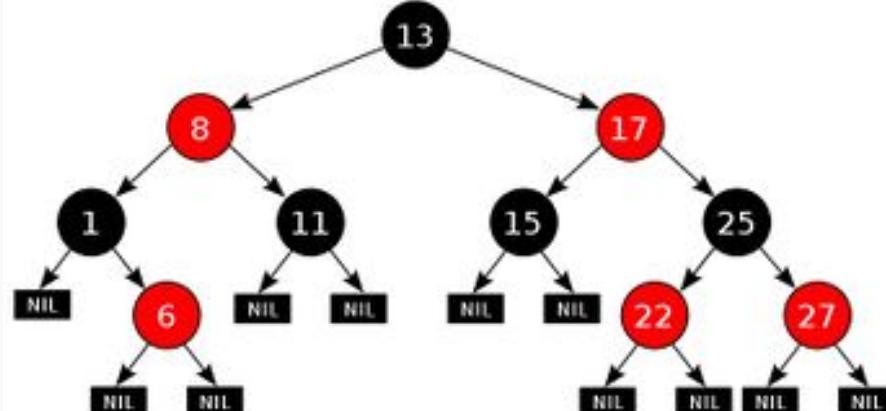


Красно-чёрное дерево – двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" и "чёрный".

Все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными.

Для экономии памяти фиктивные листья делают одним общим фиктивным листом

Изобретатель –  
Рудольф Байер (1972г).



# Красно-черные деревья



Определение 2. Красно-чёрное дерево – двоичное дерево поиска, у которого каждому узлу сопоставлен дополнительный атрибут – цвет и для которого выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла  $x$  до листьев содержат одинаковое количество чёрных узлов

# Красно-черные деревья

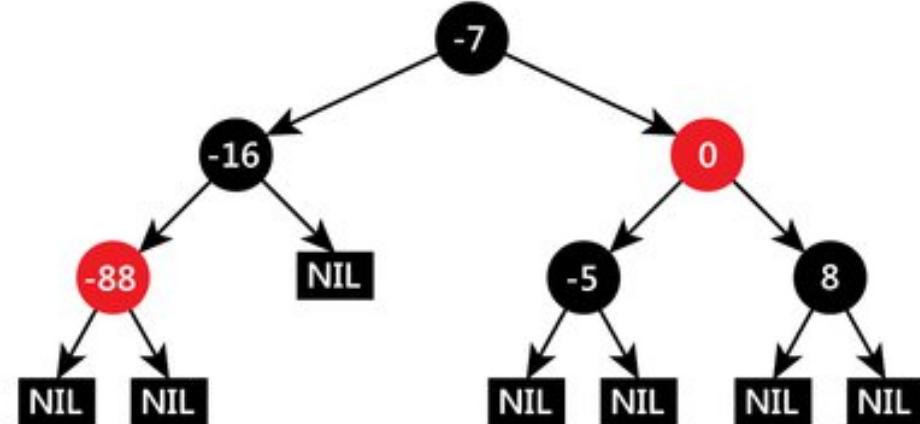


Определение 3. Черная высота вершины  $x$  – число черных вершин на пути из  $x$  в лист, не учитывая саму вершину  $x$ .

Пример.

Для вершин «-16» и «-88»  
черная высота = 1.

Для вершин «-7» и «0»  
черная высота = 2.



# Красно-черные деревья



**В красно-черном дереве с корнем в узле  $x$  содержится по крайней мере  $2^{bh(x)} - 1$  внутренних вершин.**

Докажем по индукции по  $h(x)$ :

- $h(x) = 0$   
Значит,  $x$  – лист (NIL). Тогда дерево с корнем  $x$  содержит  $2^{bh(x)} - 1 = 2^0 - 1 = 0$  внутренних вершин.
- $h(x) > 0$ .  
 $X$  – внутренняя вершина, у нее 2 потомка. У каждого потомка черная высота либо  $bh(x)$ , либо  $bh(x) - 1$ , в зависимости от цвета потомка (красный и черный, соответственно).

Так как  $h(\text{потомок } x) < h(x)$ , то к нему применимо предположение индукции: у каждого потомка по крайней мере  $2^{bh(x)-1} - 1$  внутренняя вершина. Тогда в дереве с корнем в  $x$  их по крайней мере  $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ . Переход доказан. Следовательно, утверждение верно и для всего дерева.

# Красно-чёрные деревья



## Теорема:

Красно-чёрное дерево с  $N$  ключами имеет высоту  $h = O(\log N)$ .

## Доказательство:

▷

Рассмотрим красно-чёрное дерево с высотой  $h$ . Так как у красной вершины чёрные дети, чёрных вершин не меньше, чем  $h/2$ .

По доказанной лемме, для количества внутренних вершин в дереве  $N$  выполняется неравенство:

$$N \geq 2^{h/2} - 1$$

Прологарифмировав неравенство, имеем:

$$\log(N + 1) \geq h/2$$

$$2 \log(N + 1) \geq h$$

$$h \leq 2 \log(N + 1)$$

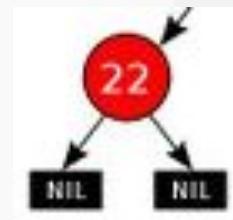
△

# Красно-черные деревья. Вставка



## Вставка элемента.

- Каждый элемент вставляется вместо листа.
- Для выбора места вставки идём от корня в нужную сторону, как в наивном методе построения дерева поиска. До тех пор, пока не остановимся в листе (в фиктивной вершине).
- Вставляем вместо листа новый элемент красного цвета с двумя листами-потомками.
- Теперь восстанавливаем свойства красно-черного дерева.



# Красно-черные деревья. Вставка



## Вставка элемента. Что мы можем сломать?

Поскольку добавленный узел автоматически окрашивается в красный цвет, то нарушить можем только эти свойства красно-чёрного дерева:

- Корень дерева — чёрный
- У красного узла родительский узел — чёрный

# Красно-черные деревья. Вставка



Ситуации после вставки:

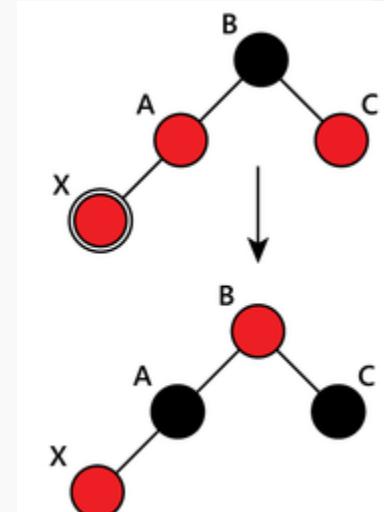
- Если отец нового элемента черный, то ничего делать не надо.
- Если отец нового элемента красный, то возможны 3 случая (без учета симметрии):
  1. Отец красный, дядя красный.
  1. Отец красный, дядя черный, новый элемент левый потомок.
  1. Отец красный, дядя черный, новый элемент правый потомок.

# Красно-чёрные деревья. Вставка



## Случай 1. Отец и дядя красные.

- Перекрашиваем «отца» и «дядю» в чёрный цвет, а «деда» - в красный.
- Поскольку «дед» может нарушать свойство дерева (вдруг его отец красный), придется рекурсивно восстанавливать свойства дерева, двигаясь к предкам.
- Если мы таким образом дойдём до корня, то в нём в любом случае ставим чёрный цвет.



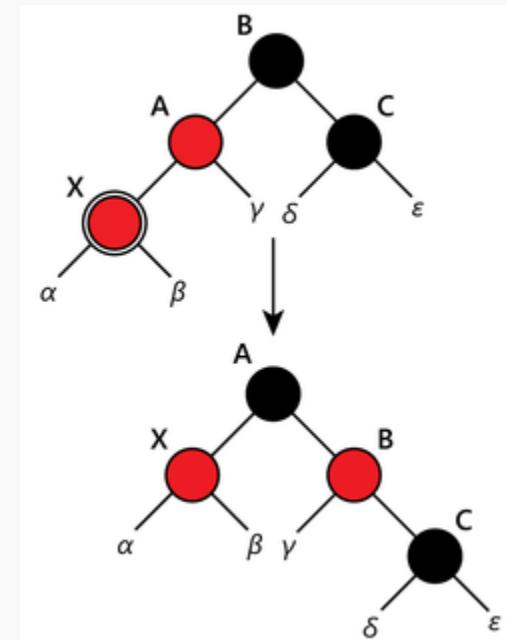
# Красно-черные деревья. Вставка



Случай 2. «Дядя» черный и правый, новый элемент — левый потомок.

Просто выполнить перекрашивание отца в черный цвет нельзя, чтобы не нарушить постоянство чёрной высоты дерева по ветви с отцом.

- Выполняем правый поворот B.
- Перекрашиваем A и B.
- Останавливаемся — больше ничего делать не требуется.



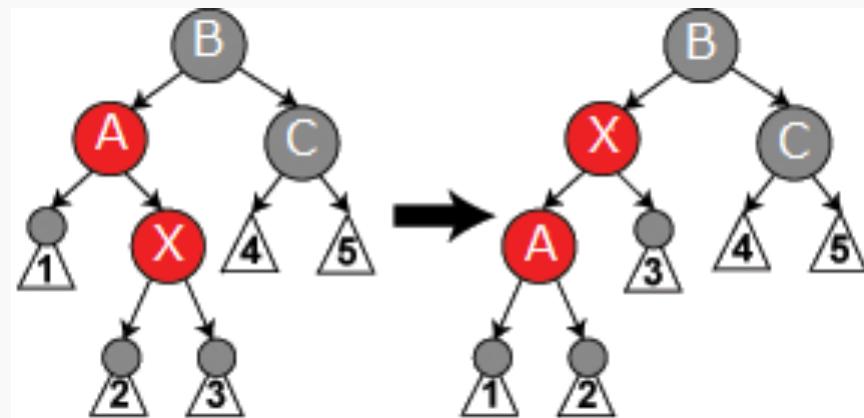
# Красно-черные деревья. Вставка



Случай 3. «Дядя» черный и правый, новый элемент — правый потомок.

Добавленный узел  $X$  — правый потомок отца  $A$ . Выполним левое вращение в  $A$ , тем самым сделав  $A$  левым потомком  $X$ .

Случай сводится к предыдущему.



# Красно-черные деревья. Вставка



Всего один шаг, если:

- Родитель черный, тогда вообще ничего делать не нужно
- Случай 1, когда дед — корень. Перекрасим его в черный и завершим алгоритм
- Случай 2
- Случай 3

Длинная цепочка действий возможна только при многократном повторении случая 1.

Сложность вставки  $O(\log N)$ .

# Красно-черные деревья. Удаление



## Как удалять:

1. Если нет детей, изменяем указатель на неё у родителя на фиктивный лист.
2. Если только один ребёнок, делаем у родителя ссылку на ребенка вместо этой вершины.
3. Если два ребёнка, находим вершину со следующим значением ключа. У нее нет левого ребёнка. Удаляем уже эту вершину способом, описанным в первом или во втором пункте, скопировав её ключ в изначальную вершину.

**Таким образом, удаление всегда выполняется для вершины, имеющей не более одной дочерней.**

# Красно-чёрные деревья. Удаление



## Удаление элемента. Что мы можем сломать?

- Корень дерева — чёрный
- У красного узла родительский узел — чёрный
- Все простые пути из любого узла  $x$  до листьев содержат одинаковое количество чёрных узлов

# Красно-черные деревья. Удаление



## A. Удаление красной вершины.

При удалении красной вершины свойства дерева не нарушаются.

### Если потомок единственный

Красная вершина, не может иметь единственного потомка. Если бы потомок существовал, то он был бы черным и нарушилось бы свойство постоянства черной глубины для потомка и его соседней фиктивной вершины.

### Если потомков нет

Действия:

- Удалить красную вершину (заменить на лист)
- Конец



# Красно-черные деревья. Удаление

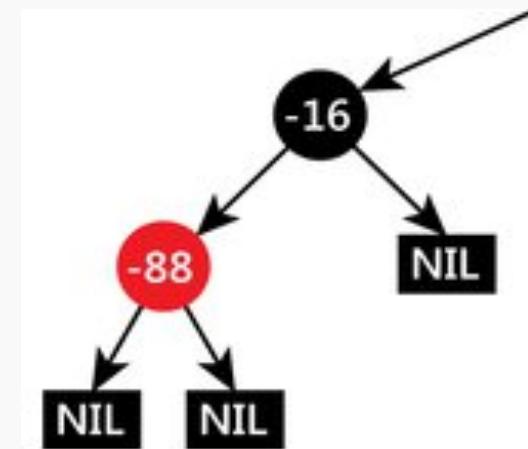


## Б. Удаление черной вершины с единственным потомком.

Единственным потомком черной вершины может быть только красная вершина. Иначе нарушилось бы свойство постоянства черной глубины для потомка и его соседней фиктивной вершины.

### Действия:

- В черную вершину заносим данные красной.
- Удаляем красную (заменяем на лист)
- Конец



# Красно-черные деревья. Удаление



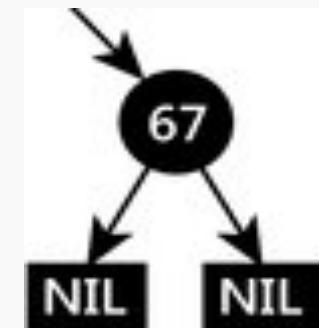
## В. Удаление черной вершины без потомков. Это самый сложный случай.

Действия:

- Удалим черную вершину (заменим на лист).
- Лист на месте удаленной вершины обозначим «х».

Путь в «х» имеет меньшее количество черных вершин (черную глубину), чем в другие вершины. Будем помнить об этом и называть «х» дважды черным.

Теперь с помощью перекрашиваний и вращений будем пытаться восстановить свойства красно-черного дерева.



# Красно-черные деревья. Удаление

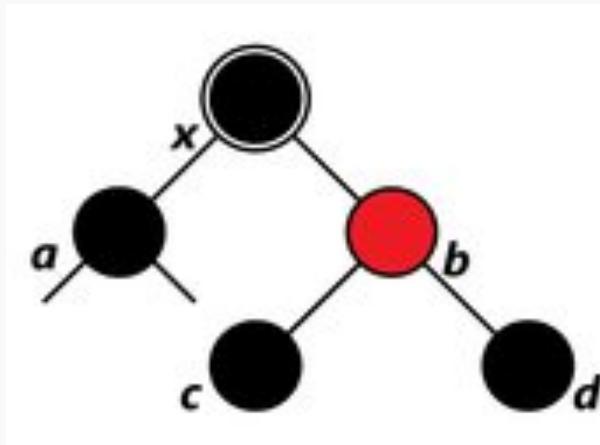


Восстановление свойств. Случай 0.

Если дважды черная вершина  $x$  – корень.

- Оставим корень просто черным (один раз черным)
- Конец

Так черная глубина всего дерева уменьшится на 1.



# Красно-черные деревья. Удаление

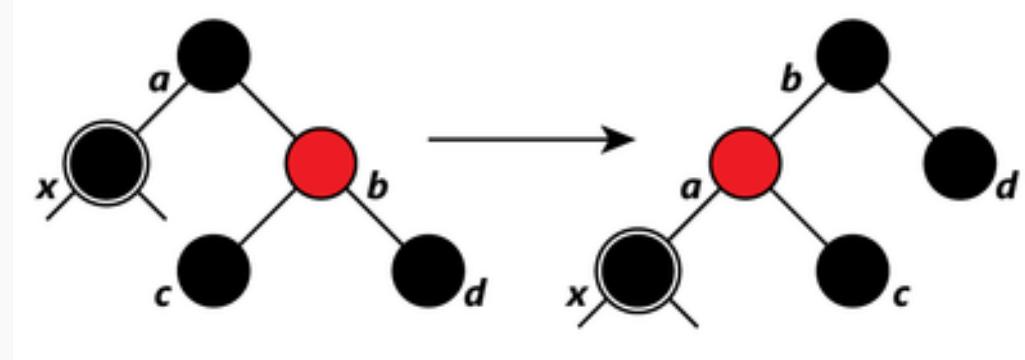


Восстановление свойств. Случай 1.

Если брат ***b*** дважды черной вершины ***x*** – красный.

- Делаем вращение вокруг ребра между отцом ***a*** и братом ***b***, тогда брат становится родителем отца.
- Красим нового деда ***b*** в чёрный, а отца ***a*** – в красный цвет.

Теперь брат ***x*** – черный, переходим в случай 2, 3 или 4 в зависимости от цветов детей узла ***c***.

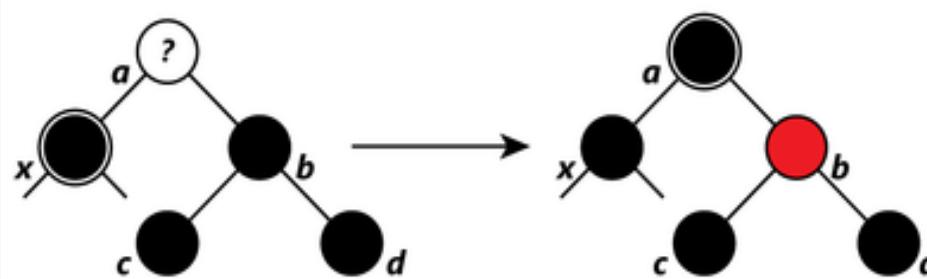


# Красно-черные деревья. Удаление



## Восстановление свойств. Случай 2.

Если брат ***b*** вершины ***x*** – черный, и оба дочерних узла брата ***c*** и ***d*** – черные. ***c*** и ***d*** могут быть листьями. Красим брата ***b*** в красный цвет.



Какого цвета был отец ***a***?

КРАСНЫЙ	ЧЕРНЫЙ
<ul style="list-style-type: none"><li>Красим отца <b><i>a</i></b> в черный цвет, так черная глубина <b><i>a</i></b> восстановится</li><li>Конец</li></ul>	<ul style="list-style-type: none"><li>Считаем отца <b><i>a</i></b> дважды черным, рассматриваем его как <b><i>x</i></b></li><li>Теперь может быть любой случай</li></ul>

# Красно-черные деревья. Удаление



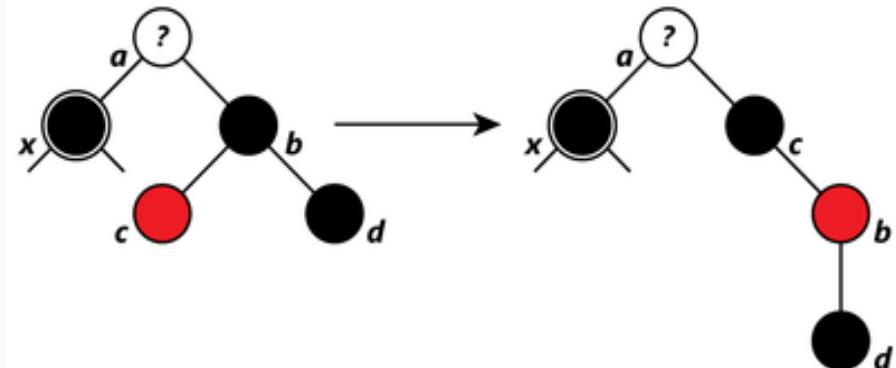
Восстановление свойств. Случай 3.

Если брат ***b*** дважды черной вершины ***x*** – черный, левый ребенок брата ***c*** – красный, а правый ***d*** – черный.

- Делаем правое вращение ***c – b***
- Красим ***b*** в красный цвет
- Красим ***c*** в черный цвет

Так у брата правый ребенок станет красным.

После случая 3 всегда случай 4, затем конец.



# Красно-черные деревья. Удаление

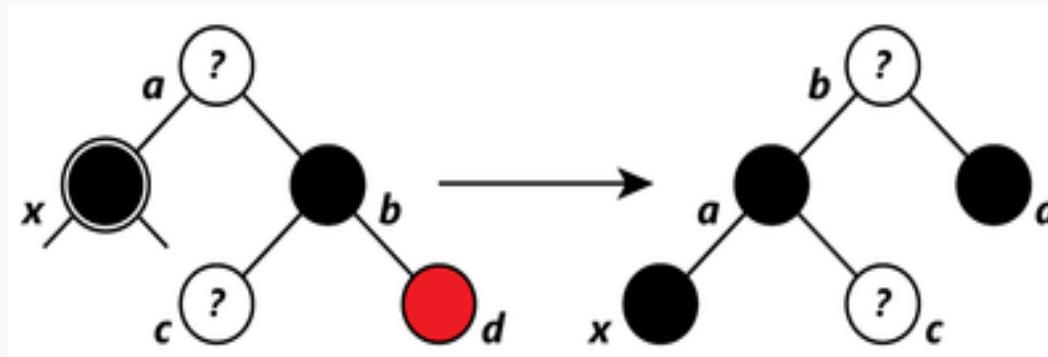


Восстановление свойств. Случай 4.

Если брат  $b$  дважды черной вершины  $x$  – черный, правый ребенок брата  $d$  – красный.

- Делаем левое вращение  $b - a$
- Красим  $b$  в цвет, который был у  $a$
- Красим  $a$  в черный цвет

Так черная глубина  $x$  увеличится на 1, то есть восстановится.  
Конец.



# Красно-черные деревья. Удаление



## Восстановление свойств.

Последовательность обработки случаев:

- Случай 1 → Случай 2 → Конец (отец узла x гарантированно красный после Случая 1)
- Случай 1 → Случай 3
- Случай 1 → Случай 4
- Случай 3 → Случай 4
- Случай 4 → Конец
- Случай 2 → Любой случай (если отец узла x черный)

Длинная цепочка возможна только при переходах  
Случай 2 → Случай 2.

# Красно-черные деревья. Восстановление свойств.



Вставка	Удаление
<ul style="list-style-type: none"><li>■ Случай выбирается по цвету «дяди».</li><li>■ Максимум 2 вращения</li></ul>	<ul style="list-style-type: none"><li>■ Случай выбирается по цвету «брата» и его потомков.</li><li>■ Максимум 3 вращения</li></ul>

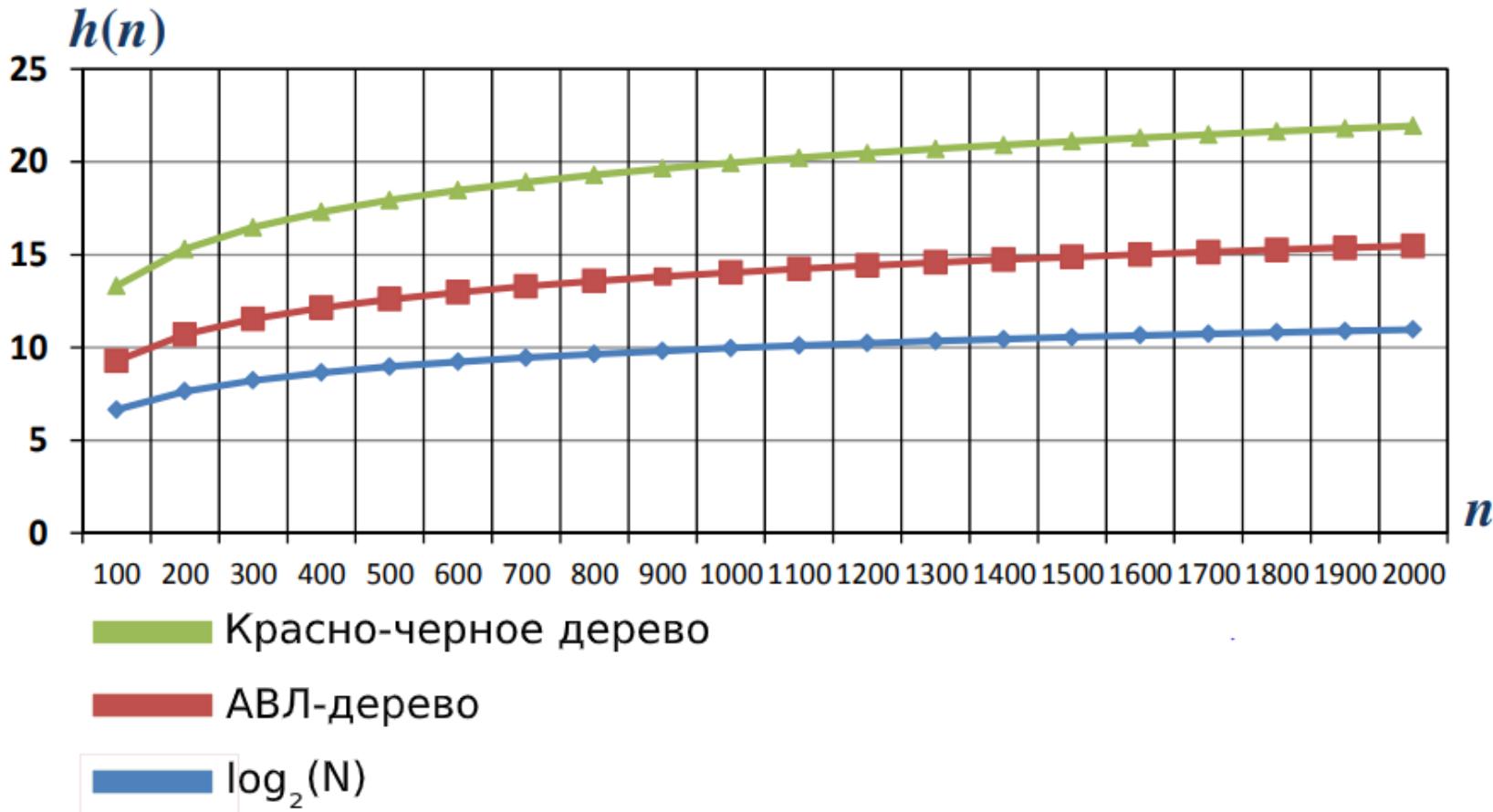
# Красно-черные деревья



Расход памяти и время работы.

	В среднем случае	В худшем случае
Расход памяти	$O(n)$	$O(n)$
Поиск	$O(\log n)$	$O(\log n)$
Вставка	$O(\log n)$	$O(\log n)$
Удаление	$O(\log n)$	$O(\log n)$

# Оценка сверху высот АВЛ и красно-черных деревьев



# Сравнение АВЛ и красно-черных деревьев



- AVL деревья более строго сбалансированы, быстрее выполняют поиск элемента. Максимальная высота  $\sim 1.44 * \log_2 n$ .
- Красно-черные деревья быстрее выполняют вставку и удаление элемента.
- AVL деревья хранят в узлах значение баланса или высоту узла, нужно тратить память на целочисленную переменную.
- Красно-черные деревья хранят в узлах цвет (всего 2 возможных значения).
- Красно-черные деревья получили более широкое распространение: стандартные контейнеры в C++ STL, Java, ядро Linux (например, планировщик) и пр.

# В-дерево



- Сбалансированное дерево поиска, обобщает понятие двоичного дерева поиска.
- Узлы В-деревьев могут иметь тысячи потомков
- Оптимизирует работу с диском, минимизируя число операций чтения/записи.
- Многие БД хранят данные в В-деревьях

Придумано Р. Бэйером (англ. R. Bayer) и Э. МакКрейтом (англ. E. McCreight) в 1970 году.

# В-дерево. Определение.



В-дерево – дерево с корнем, обладающее следующими свойствами:

1) Каждый узел X содержит:

- X.n – число ключей в узле
- Сами ключи в отсортированном порядке:  
 $X.key_1 \leq X.key_2 \leq \dots \leq X.key_n$
- X.leaf – булево значение “узел является листом”

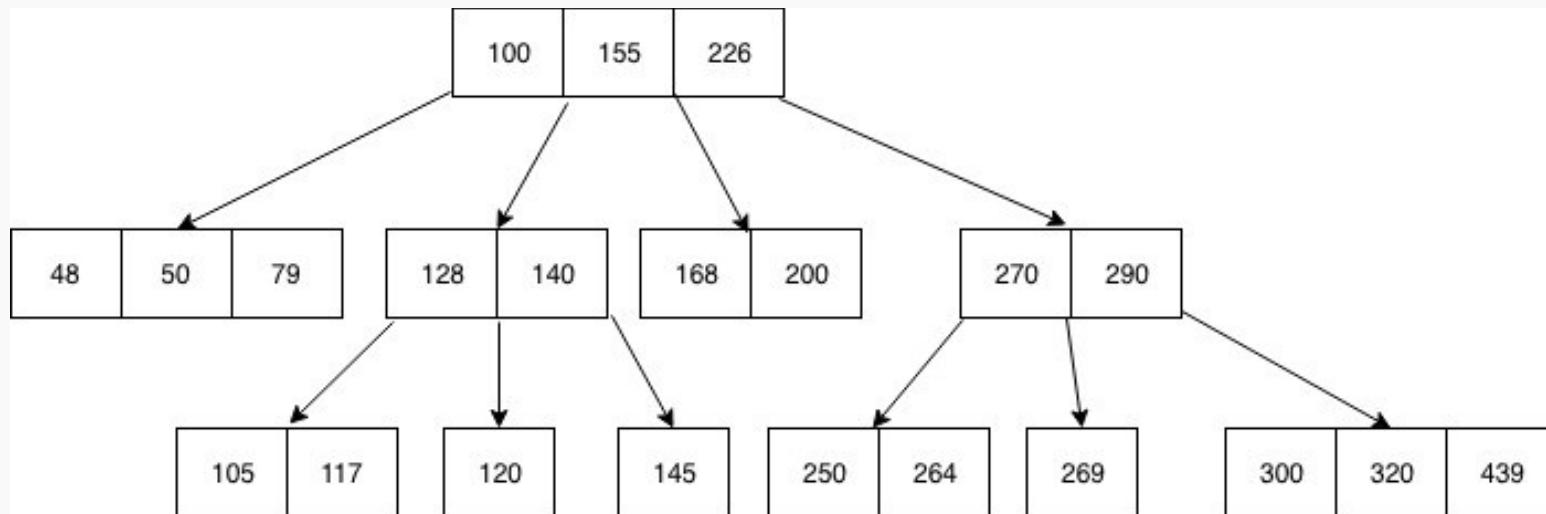
2) Каждый внутренний узел также содержит:

- X.n + 1 указателей x.c<sub>1</sub>, x.c<sub>2</sub>, ..., x.c<sub>n+1</sub> на потомков.

# В-дерево. Определение.



- 3) Ключи  $X.key_i$  внутри узла определяют, в какое поддерево переходим при поиске/вставке/удалении.
- 4) Пусть  $\{k_i\}$  – все ключи из под дерева, на которое указывает  $X.c_i$ , тогда  $\{k_1\} \leq X.key_1 \leq \{k_2\} \leq X.key_2 \leq \dots \leq X.key_{x.n} \leq \{k_{x.n+1}\}$ .



# В-дерево. Определение.



- 4) Все листья В-дерева находятся на одной глубине
- 5) Минимальная степень В-дерева – целое число  $t \geq 2$ , определяет минимум и максимум числа ключей в узле.
  - Все внутренние узлы, помимо корня, должны хранить не менее  $t - 1$  ключей (следовательно, иметь не менее  $t$  потомков). В непустом дереве в корне хранится по крайней мере 1 ключ.
  - Все узлы хранят максимум  $2t - 1$  ключ (следовательно, имеют не более  $2t$  потомков). Если в узле  $2t - 1$  ключ, то его называют заполненным.
  - Чем больше  $t$ , тем меньше высота дерева.

# В-дерево. Высота дерева.



## Теорема:

Если  $N \geq 1$ , то для любого В-дерева, содержащего  $N$  ключей и минимальной степенью  $t \geq 2$ , верно утверждение

$$h \leq \log_t\left(\frac{n+1}{2}\right).$$

## Доказательство:

Корень В-дерева содержит по крайней мере 1 ключ, прочие узлы – по крайней мере  $t - 1$  ключ. Следовательно, дерево Т с высотой  $h$  имеет по крайней мере 2 узла на глубине 1,  $2t$  узлов на глубине 2,  $2t^2$  узлов на глубине 3, ...,  $2t^{h-1}$  узлов на глубине  $h$ .

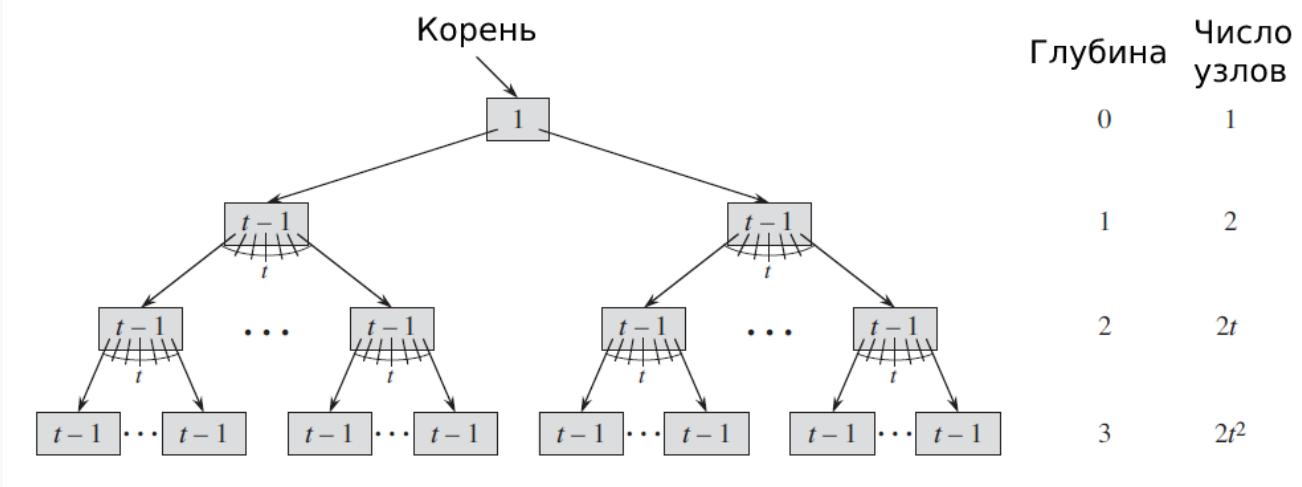
(Продолжение на следующем слайде)

# В-дерево. Высота дерева.



**Доказательство:** (продолжение)

Иллюстрация для  $h = 3$ :



Таким образом, имеет место неравенство:  $n \geq 1 + (t - 1) \sum_{i=1}^h 2t^{i-1} = 1 + 2(t - 1) \left( \frac{t^h - 1}{t - 1} \right) = 2t^h - 1$

Следовательно,  $t^h \leq \left( \frac{n+1}{2} \right)$ . Возьмем  $\log_t$  от обеих частей неравенства:  $h \leq \log_t \left( \frac{n+1}{2} \right)$ . Что и требовалось доказать.

# В-дерево. Устройство жесткого диска.



- Данные хранятся на одной или несколько пластинах
- Пластины вращаются с постоянной скоростью вокруг шпинделя
- Считывающая головка перемещается к шпинделю/от шпинделя

# В-дерево. Сопоставление RAM и HDD.



- Стоимость 1Гб оперативной памяти в ~100 раз выше, чем стоимость 1Гб на жестком диске.
- Как правило, емкость установленных в компьютере жестких дисков как минимум на 2 порядка превышает объем доступной оперативной памяти.
- Доступ к данным в RAM занимает ~50 нс, в HDD ~8-11 мс (в 160,000 - 220,000 раз медленнее). Все из-за механики – нужно прокрутить дисковые пластины и переместитьчитывающую головку.

# В-дерево. Особенности работы HDD.



- Минимальная единица чтения/записи – страница.
- Все страницы равного размера (обычно от 2 до 16 килобайт). Все биты в странице хранятся на диске единым неделимым блоком.
- Выгодно сразу читать/писать несколько страниц: основные затраты времени приходятся на механическое позиционирование пластин и считающей головки на начало первой страницы. Само чтение/запись происходят быстро – там одна электроника, без механики (без учета постоянного вращения пластины).

# В-дерево. Подстраиваемся под HDD.

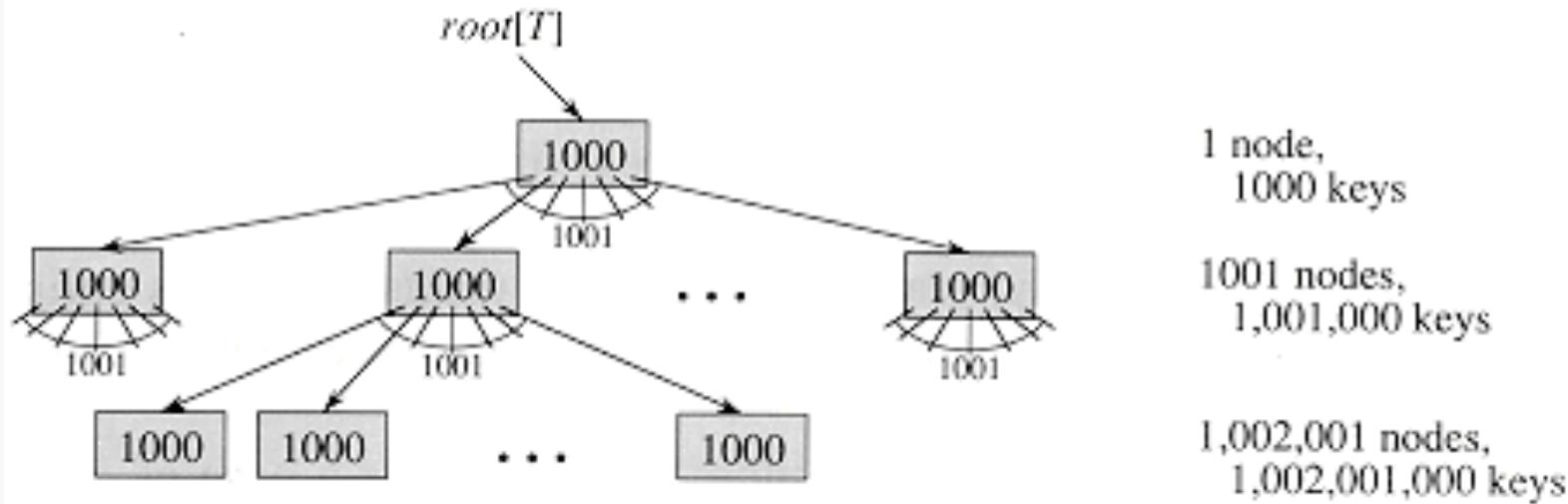


- Предполагается, что хранимые данные не помещаются в оперативную память.
- В-дерево держит в оперативной памяти ограниченное константой число страниц. По мере надобности страницы подгружаются с диска в память, а при модификации записываются на диск.
- Для оптимизации чтения/записи размер узла В-дерева подбирается под размер страницы жесткого диска. Это накладывает ограничения на число хранимых в узле ключей и, соответственно, число потомков.

# В-дерево. Подстраиваемся под HDD.



- В больших В-деревьях фактор ветвления обычно от 50 до 2000 (зависит от размера ключа). Чем он больше, тем меньше высота дерева, а значит меньше доступов к диску.
- Пример. Фактор ветвления 1001, высота 2. Помещается более миллиарда ключей. Если корень всегда в RAM, любой ключ достижим не более чем за 2 доступа к диску.



# В-дерево. Операции с В-деревьями.



- Нет гарантий, что нужный нам узел X находится в RAM или не будет вскоре вытеснен из нее.
- Введем операцию DISK-READ(X), зачитывающую узел X с диска в RAM. Если X уже в ней, то DISK-READ не будет обращаться к диску.
- Операция DISK-WRITE(X) записывает на диск узел X. Необходимо ее вызывать после любой модификации узла X.
- Никогда не будем вытеснять корень дерева из RAM, поэтому для него DISK-READ не понадобится. Но при модификации корня все равно нужен DISK-WRITE.

# В-дерево. Создание дерева.



B-TREE-CREATE создает пустой корневой узел.

```
B-TREE-CREATE (T)
    X = ALLOCATE-NODE ()
    X.leaf = TRUE
    X.n = 0
    DISK-WRITE (X)
    T.root = X
```

Вспомогательная операция ALLOCATE-NODE выделяет страницу для нового узла за  $O(1)$ .

B-TREE-CREATE требует  $O(1)$  дисковых операций и  $O(1)$  процессорного времени.

# В-дерево. Поиск ключа.



B-TREE-SEARCH( $X, k$ ) – поиск ключа  $k$  в В-дереве с корнем  $X$ . Если  $k$  в дереве, то вернет  $(Y, i)$  (где  $Y$  – узел,  $i$  – индекс ключа в  $Y$ , такой что  $Y.key_i == k$ ), иначе NIL.

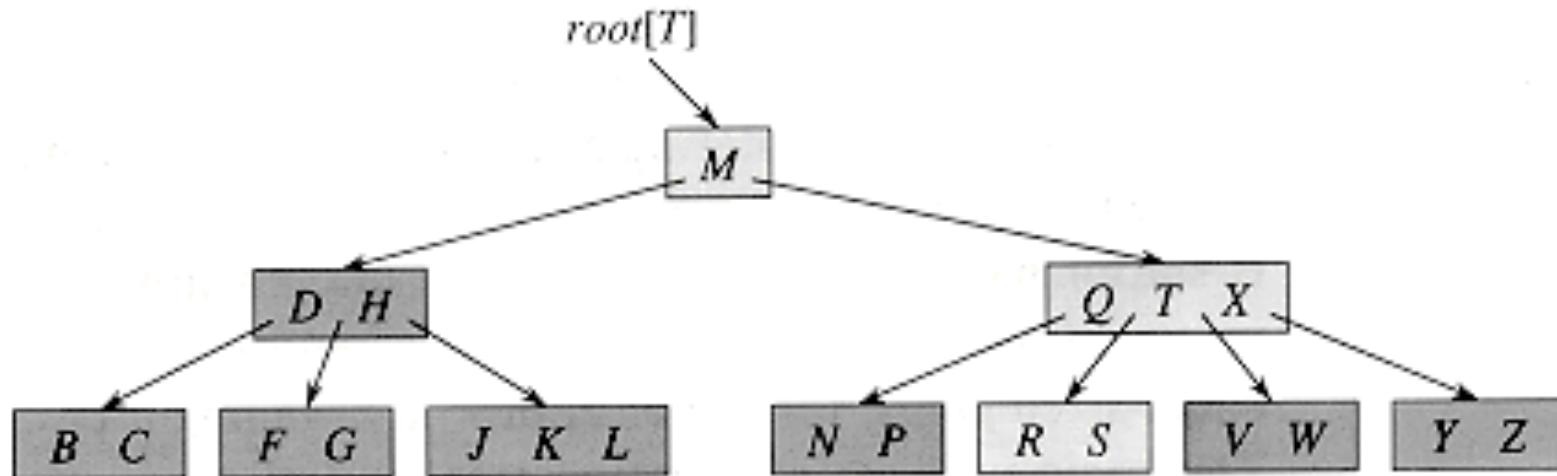
```
B-TREE-SEARCH (X, k)
    i = 1
    while i ≤ X.n and k > X.keyi
        i = i + 1
    if i ≤ X.n and k == x.keyi
        return (X, i)
    elseif X.leaf
        return NIL
    else
        DISK-READ (X.ci)
        return B-TREE-SEARCH (X.ci, k)
```

# В-дерево. Поиск ключа.



Сложность поиска:  $O(t * \log_t N)$ .

Пример: В-дерево построено на согласных буквах английского алфавита, ищем в нем ключ R. Светлым подсвечены узлы, по которым пройдем при поиске ключа.



# В-дерево. Вставка ключа.



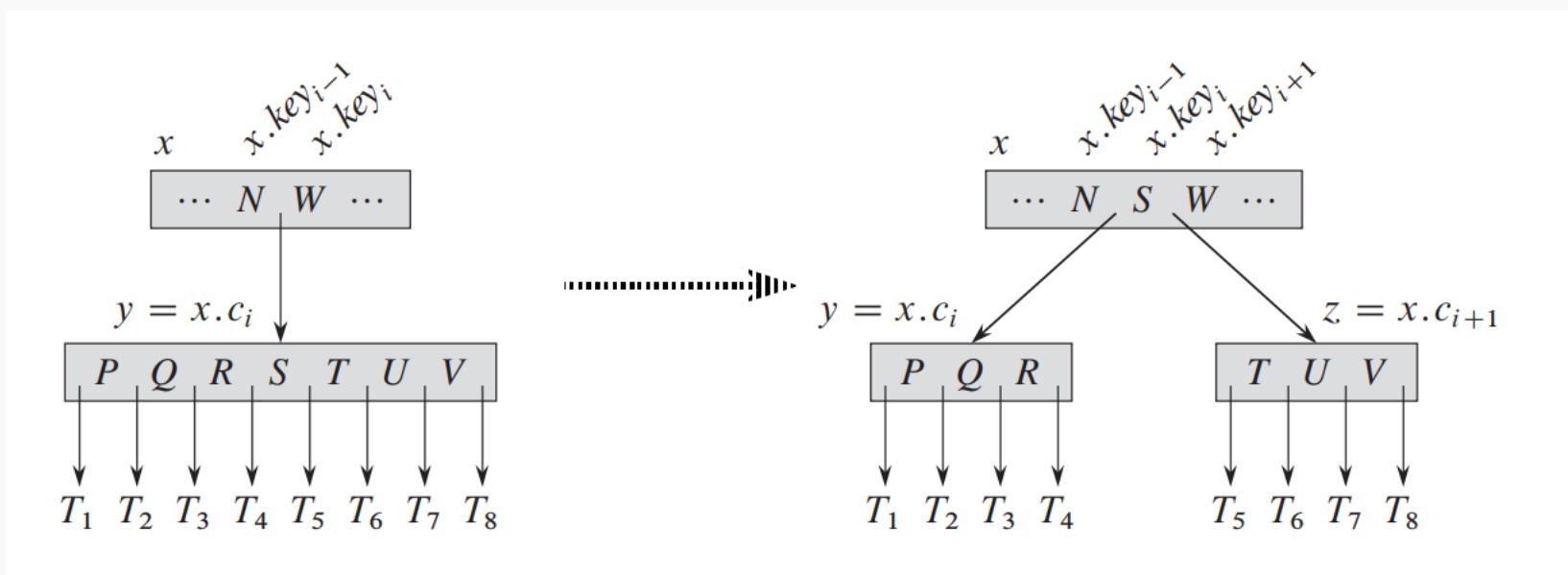
- Вставка сложнее, чем в бинарном дереве поиска: мы не можем создать новый лист в дереве – результат перестанет быть В-деревом.
- Будем добавлять ключ в какой-то существующий лист.
- Нельзя добавлять ключ в полный узел (хранищий  $2t - 1$  ключей).
- Чтобы бороться с переполнением узлов, введем операцию B-TREE-SPLIT-CHILD, разбивающую полный узел  $Y$  по медианному ключу  $Y.key_t$  на 2 новых узла, в каждом по  $t - 1$  ключей. Ключ  $Y.key_t$  перенесем в родителя  $Y$ . Новые деревья подцепим к нему же,  $Y.key_t$  станет для них разделителем.

# В-дерево. Вставка ключа. B-TREE-SPLIT-CHILD.



Пример применения B-TREE-SPLIT-CHILD: В-дерево, минимальная степень  $t = 4$ . Узел  $y$  содержит  $2t - 1 = 7$  ключей, он полный.

Разбиваем по медиане ( $S$ ) на 2 новых узла,  $S$  добавляем к ключам родителя ( $x$ ), новые деревья цепляем по указателям  $x.c_i$  и  $x.c_{i+1}$ . У родителя +1 к ключам и +1 к потомкам.



# В-дерево. Вставка ключа. B-TREE-SPLIT-CHILD.



- Внутри B-TREE-SPLIT-CHILD один из ключей разбиваемого узла поднимается к родителю. Но что если родитель тоже уже полный? Придется рекурсивно вызывать B-TREE-SPLIT-CHILD. Так можно до корня дойти.
- Дополнительная оптимизация: мы не будем ждать, пока потребуется добавлять ключ в полный узел. В процессе спуска по дереву для всех полных узлов заранее вызываем B-TREE-SPLIT-CHILD (включая сам лист).
- Благодаря этому при вызове B-TREE-SPLIT-CHILD будем уверены, что у родителя есть место для нового ключа.

# В-дерево. Вставка ключа. В-TREE-SPLIT-CHILD.



В-TREE-SPLIT-CHILD( $X, i$ ) – для неполного внутреннего узла  $X$  вызвать разбиение полного потомка по указателю  $X.c_i$ .

В-TREE-SPLIT-CHILD( $X, i$ )

```
Z = ALLOCATE-NODE() // новый узел-потомок X
Z.leaf = Y.leaf // если Y – лист, то Z тоже
Z.n = t - 1 // половина ключей Y уходит в Z
for j = 1 to t - 1 // переносим их
    Z.keyj = Y.keyj+t
if not Y.leaf // если у Y есть потомки
    for j = 1 to t // половину потомков Y
        Z.cj = Y.cj+t // переносим в Z
Y.n = t - 1 // поправим счетчик ключей в Y
for j = X.n + 1 downto i + 1
    X.cj+1 = X.cj // сдвиг указателей на 1 вправо
X.ci+1 = z // чтобы добавить указатель на Z
// продолжение на следующем слайде
```

# В-дерево. Вставка ключа. В-TREE-SPLIT-CHILD.



B-TREE-SPLIT-CHILD( $X, i$ ) – для неполного внутреннего узла  $X$  вызвать разбиение полного потомка по указателю  $X.c_i$ .

B-TREE-SPLIT-CHILD( $X, i$ )

```
... // начало на предыдущем слайде
for j = X.n downto i // сдвиг ключей в X вправо
    X.keyj+1 = X.keyj // на 1 для вставки медианы
X.keyi = Y.keyt // бывшего узла Y (до разбиения)
X.n = X.n + 1 // из-за этого +1 к ключам в X
DISK-WRITE(Y)
DISK-WRITE(Z)
DISK-WRITE(X)
```

Сложность разбиения  $O(t)$ .

# В-дерево. Вставка ключа. В-TREE-SPLIT-CHILD.



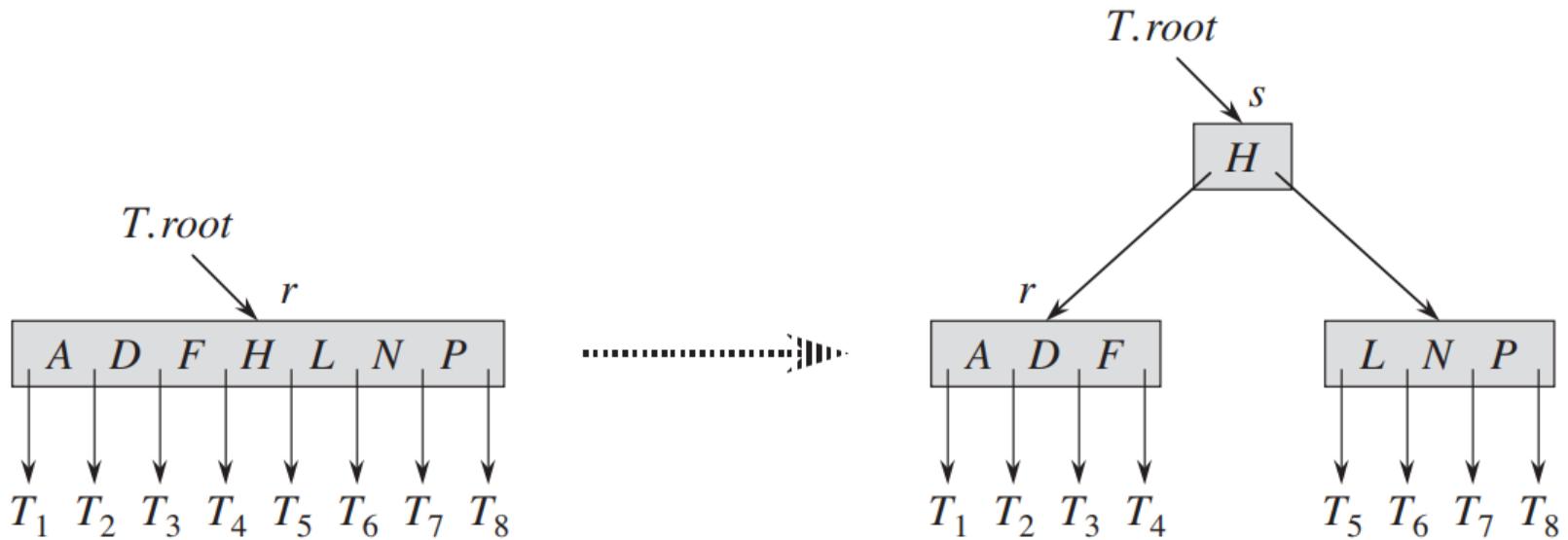
Как разбить полный корневой узел:

- Создаем новый пустой корневой узел и цепляем к нему потомком старый заполненный корневой узел.
- Вызываем В-TREE-SPLIT-CHILD на новом корневом узле. Старый корень распадется на 2 узла, они подцепятся к новому корню, бывшая медиана старого корня переместится в новый корень.
- В результате высота дерева увеличится на 1.
- Разбиение узлов – единственный способ увеличить высоту В-дерева.

# В-дерево. Вставка ключа. B-TREE-SPLIT-CHILD.



Пример вызова B-TREE-SPLIT-CHILD для полностью заполненного корня ( $t = 4$ ):



В отличие от бинарного дерева поиска, В-дерево растет вверх, а не вниз.

# В-дерево. Вставка ключа.



B-TREE-INSERT( $T, k$ ) добавляет ключ  $k$  в В-дерево  $T$ .  
Фактически вставка производится в B-TREE-INSERT-NONFULL.

```
B-TREE-INSERT(T, k)
    r = T.root
    if r.n == 2t - 1 // если корень заполнен
        s = ALLOCATE-NODE() // создаем новый узел
        T.root = s // делаем его корнем
        s.leaf = FALSE // у него будут потомки
        s.n = 0
        s.c1 = r // цепляем к нему старый корень
        B-TREE-SPLIT-CHILD(s, 1) // старый корень
        B-TREE-INSERT-NONFULL(s, k)
    else
        B-TREE-INSERT-NONFULL(r, k)
```

# В-дерево. Вставка ключа.



B-TREE-INSERT-NONFULL( $X$ ,  $k$ ) добавляет ключ  $k$  в неполный (обязательно!) узел  $X$ .

```
B-TREE-INSERT-NONFULL ( $X$ ,  $k$ )
     $i = X.n$ 
    if  $X.\text{leaf}$  // если  $X$  – лист, в этот узел запишем
    ключ  $k$ 
        // ищем позицию вставки справа налево, сдвигая
    обойденные ключи на 1 вправо
        while  $i \geq 1$  and  $k < X.\text{key}_i$ 
             $X.\text{key}_{i+1} = X.\text{key}_i$ 
             $i = i - 1$ 
        // добавляем ключ на освободившееся место
         $X.\text{key}_{i+1} = k$ 
         $X.n = X.n + 1$  // +1 к хранимым ключам
        DISK-WRITE ( $X$ )
    // конец if, продолжение на следующем слайде
    ...
```

# В-дерево. Вставка ключа.



B-TREE-INSERT-NONFULL( $X, k$ ) добавляет ключ  $k$  в неполный (обязательно!) узел  $X$ .

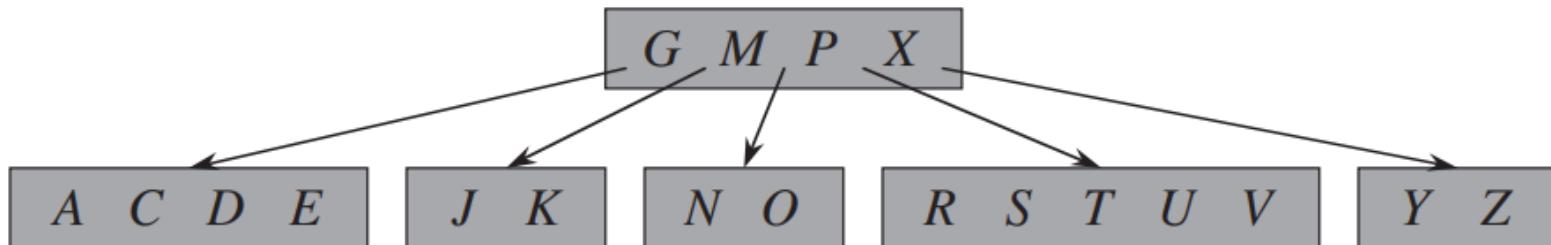
```
B-TREE-INSERT-NONFULL (X, k)
else // если же X – внутренний узел
    // надо найти потомка, куда будем писать
    while i ≥ 1 and k < X.keyi
        i = i - 1
    i = i + 1
    DISK-READ (X.ci)
    // если потомок полон, сначала разобъем его
    if X.ci.n == 2t - 1
        B-TREE-SPLIT-CHILD (X, i)
        // в какого из новых потомков пойдем?
        if k > X.keyi
            i = i + 1
    B-TREE-INSERT-NONFULL (X.ci, k)
```

# В-дерево. Вставка ключа. Примеры ситуаций.

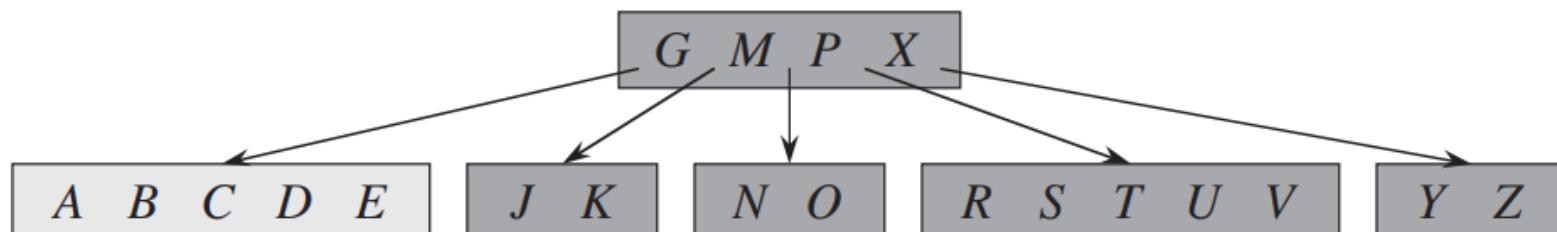


Примеры ситуаций при вставке.

Исходное В-дерево,  $t = 3$ . Узел может содержать от 2-х до 5 ключей (корень должен содержать хотя бы 1).



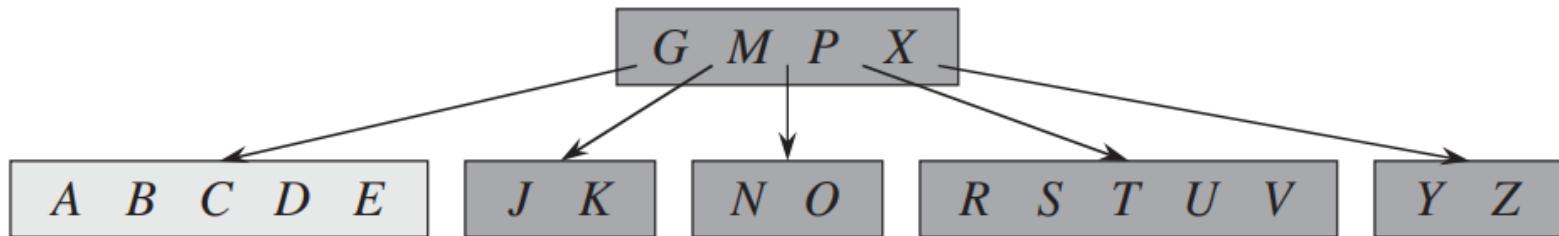
Добавляем ключ B, он просто записывается в лист.



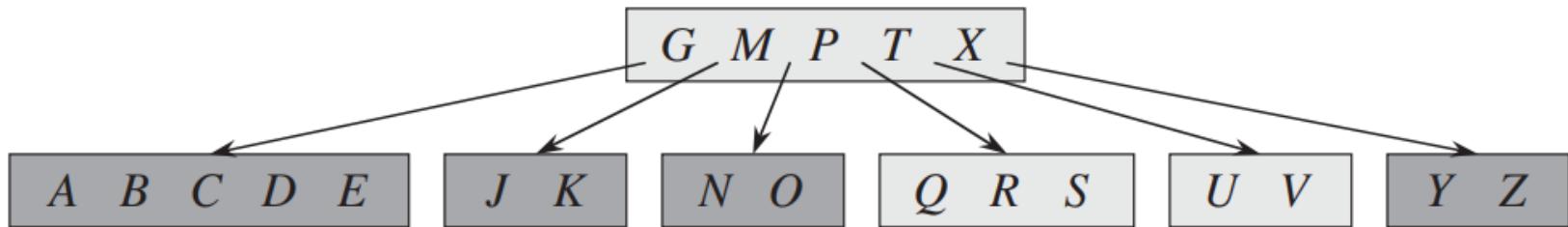
# В-дерево. Вставка ключа. Примеры ситуаций.



Примеры ситуаций при вставке.



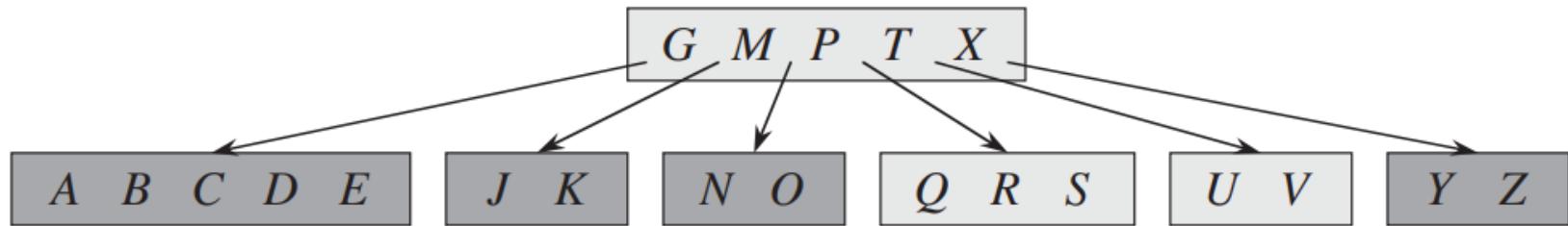
Добавляем ключ Q. Он должен попасть в лист RSTUV, но тот полон. Разбиваем его на RS и UV, Т переносим в родительский узел, Q добавляем в левого потомка (RS).



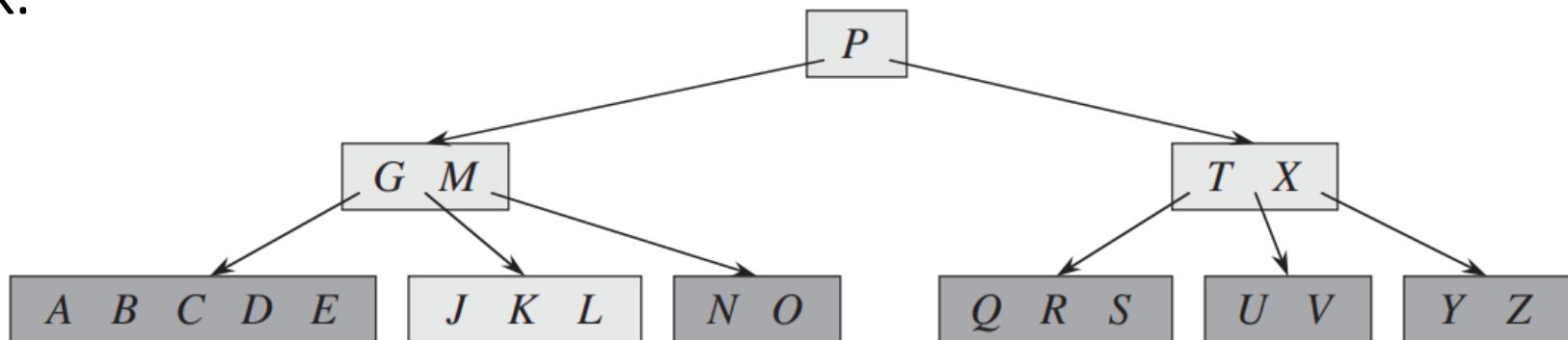
# В-дерево. Вставка ключа. Примеры ситуаций.



Примеры ситуаций при вставке.



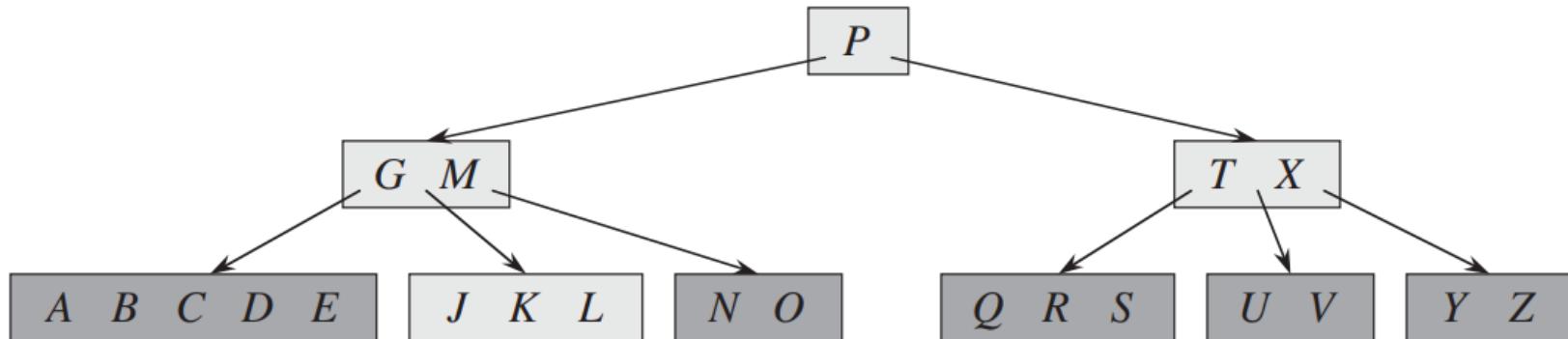
Добавляем ключ L. По пути обнаруживаем, что корень полон, надо разбивать. Получаем новый корень Р с потомками GM и TX. Высота дерева увеличилась на 1. L будет записан в лист JK.



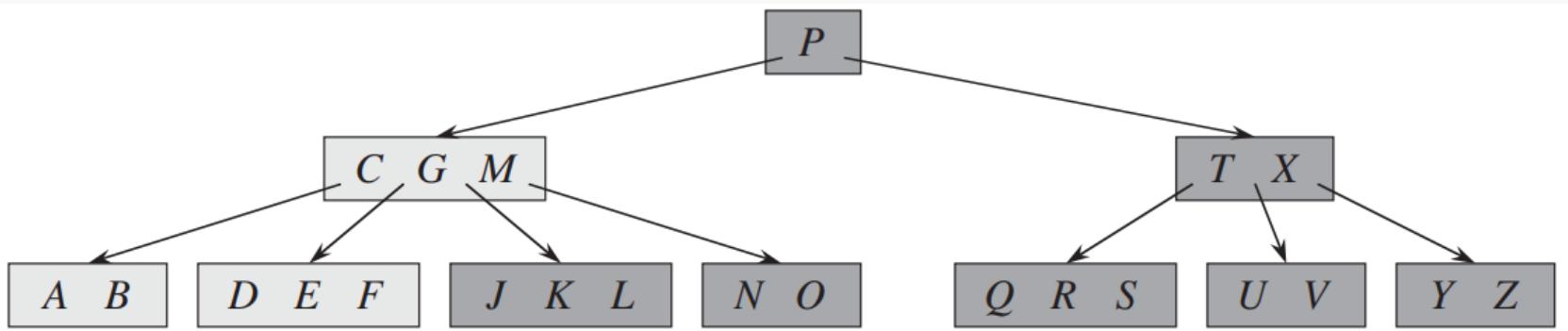
# В-дерево. Вставка ключа. Примеры ситуаций.



Примеры ситуаций при вставке.



Добавляем ключ F. Лист ABCDE полон, разбиваем на AB и DE.  
Ключ С переносим к родителю. F попадет в узел DE.



# В-дерево. Вставка ключа. Сложность.



В рамках вызова B-TREE-INSERT-NONFULL производится  $O(1)$  вызовов DISK-READ и DISK-WRITE, значит вызов B-TREE-INSERT выполняет  $O(h) = O(\log_t N)$  дисковых операций.

Общая сложность алгоритма вставки ключа  $O(t * h) = O(t * \log_t N)$ .

# В-дерево. Удаление ключа.



Удаление сложнее вставки – удалять можно из произвольного узла, не только листа.

При вставке мы могли переполнить узел – превысить лимит на  $2t - 1$  ключ.

При удалении может произойти обратное – в узле станет менее  $t - 1$  ключей.

При проходе по дереву в поиске удаляемого ключа будем принимать меры, чтобы у потомка, в который переходим, было по крайней мере  $t$  ключей (на 1 больше минимума).

# В-дерево. Удаление ключа.



Удаление сложнее вставки – удалять можно из произвольного узла, не только листа.

При вставке мы могли переполнить узел – превысить лимит на  $2t - 1$  ключ.

При удалении может произойти обратное – в узле станет менее  $t - 1$  ключей.

При проходе по дереву в поиске удаляемого ключа будем принимать меры, чтобы у потомка, в который переходим, было по крайней мере  $t$  ключей (на 1 больше минимума).

# B-дерево. Удаление ключа.



B-TREE-DELETE( $X, k$ ) удаляет ключ  $k$  из поддерева с корнем в узле  $X$ .

Псевдокод слишком громоздкий, чтобы привести его на слайдах, поэтому просто рассмотрим возможные случаи. С псевдокодом можно ознакомиться по ссылке:

<https://sites.math.rutgers.edu/~ajl213/CLRS/Ch18.pdf>  
(страницы 9 и 10).

# В-дерево. Удаление ключа. Случай.



1. Ключ  $k$  нашелся в узле  $X$ ,  $X$  – лист.  
Удаляем  $k$  из  $X$ .
2. Ключ  $k$  нашелся в узле  $X$ ,  $X$  – внутренний узел.
  - a) Если потомок  $Y$ , предшествующий  $k$  в узле  $X$ , содержит не менее  $t$  ключей.  
Для ключа  $k$  находим в узле  $Y$  предшественника --  $k'$ .  
Рекурсивно удаляем  $k'$  из  $Y$ , а в  $X$  вместо  $k$  ставим  $k'$ .
  - b) Если в потомке  $Y$  только  $t - 1$  ключ, то пытаемся произвести симметричные действия с потомком  $Z$ , следующим за  $k$  в узле  $X$ . Если в  $Z$  хотя бы  $t$  ключей, тогда для ключа  $k$  находим в узле  $Z$  последующий ключ --  $k'$ . Рекурсивно удаляем  $k'$  из  $Z$ , а в  $X$  вместо  $k$  ставим  $k'$ .

# B-дерево. Удаление ключа. Случай.



2. Ключ  $k$  нашелся в узле  $X$ ,  $X$  – внутренний узел (продолжение).
  - c) Если оба потомка  $Y$  и  $Z$  хранят всего по  $t - 1$  ключей каждый.  
Сливаем ключ  $k$  и весь узел  $Z$  в  $Y$ . Из  $X$  уходят ключ  $k$  и указатель на  $Z$ . Узел  $Y$  теперь содержит  $2t - 1$  ключ. Освобождаем  $Z$ , рекурсивно удаляем  $k$  из  $Y$ .
3. Ключ  $k$  не нашелся во внутреннем узле  $X$ . Найдем корень  $X.c_i$  поддерева, в котором продолжим поиски  $k$ . Если в  $X.c_i$  только  $t - 1$  ключ, выполним шаг 3a или 3b, чтобы в  $X.c_i$  стало  $t$  ключей. Делаем  $B\text{-TREE-DELETE}(X.c_i, k)$ .

# B-дерево. Удаление ключа. Случай.



3. Ключ  $k$  не нашелся во внутреннем узле  $X$  (продолжение).

- a) Если в  $X.c_i$  только  $t - 1$  ключ, но у ближайшего соседа справа или слева по крайней мере  $t$  ключей.

Возьмем из  $X$  ключ «между  $X.c_i$  и соседом  $X.c_{i+1}$ » и спустим его в  $X.c_i$ , поднимем ключ из соседа  $X.c_{i+1}$  в  $X$ , на место перенесенного. Также перенесем соответствующий указатель на потомка из соседа  $X.c_i$  в  $X.c_{i+1}$ . Не забудем поправить указатель на потомка в  $X.c_i$ .

# B-дерево. Удаление ключа. Случай.



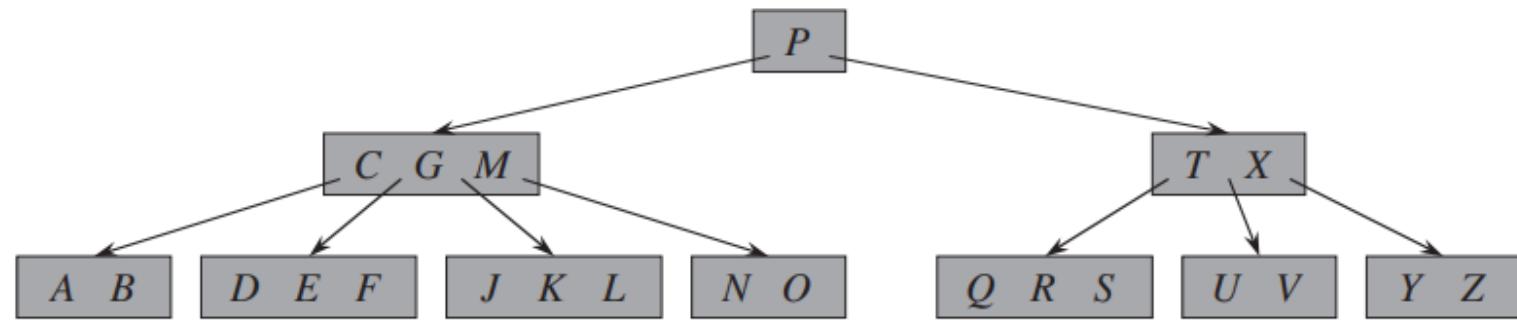
3. Ключ  $k$  не нашелся во внутреннем узле  $X$  (продолжение).
  - b) Если в  $X.c_i$  только  $t - 1$  ключ, и у обоих ближайших соседей справа и слева по  $t - 1$  ключу.  
Возьмем одного из соседей, ключ из  $X$ , разделяющий соседа и  $X.c_i$ , и сольем их в новый узел. В  $X$  теперь на 1 ключ и на 1 ссылку меньше.

# В-дерево. Удаление ключа. Примеры ситуаций.

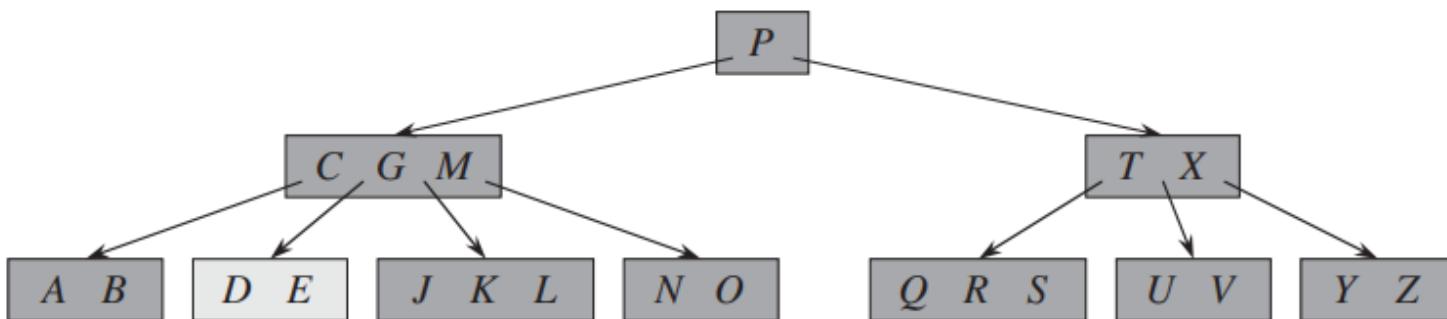


Примеры ситуаций при удалении.

Исходное В-дерево,  $t = 3$ . Узел может содержать от 2-х до 5 ключей (корень должен содержать хотя бы 1).



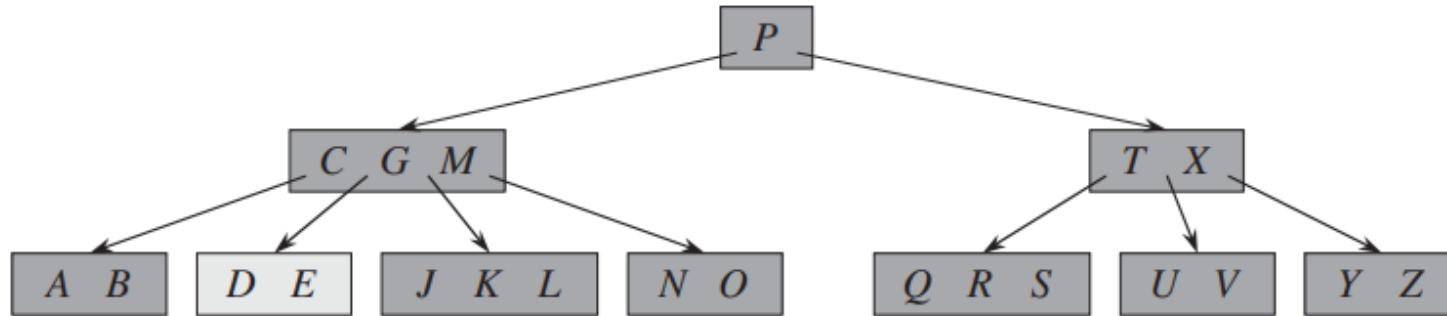
Удаляем ключ F. Тут простое удаление из листа. Случай 1.



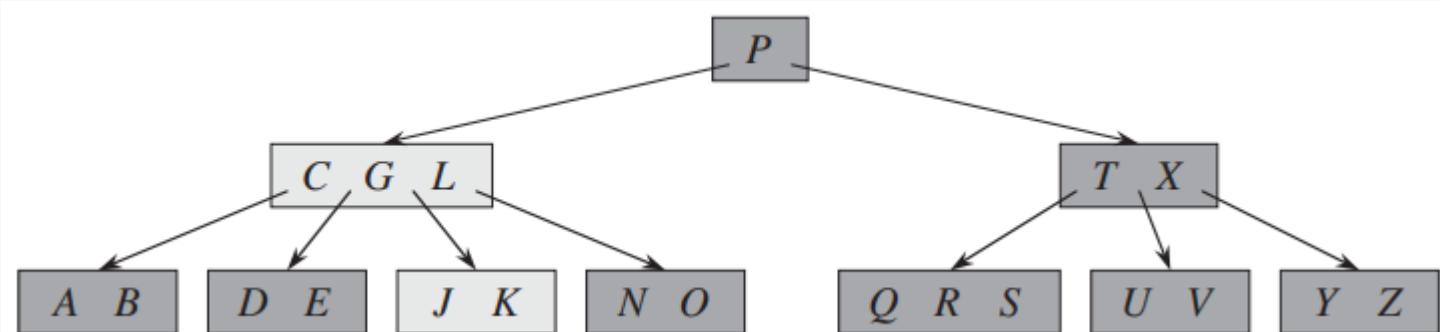
# В-дерево. Удаление ключа. Примеры ситуаций.



Примеры ситуаций при удалении.



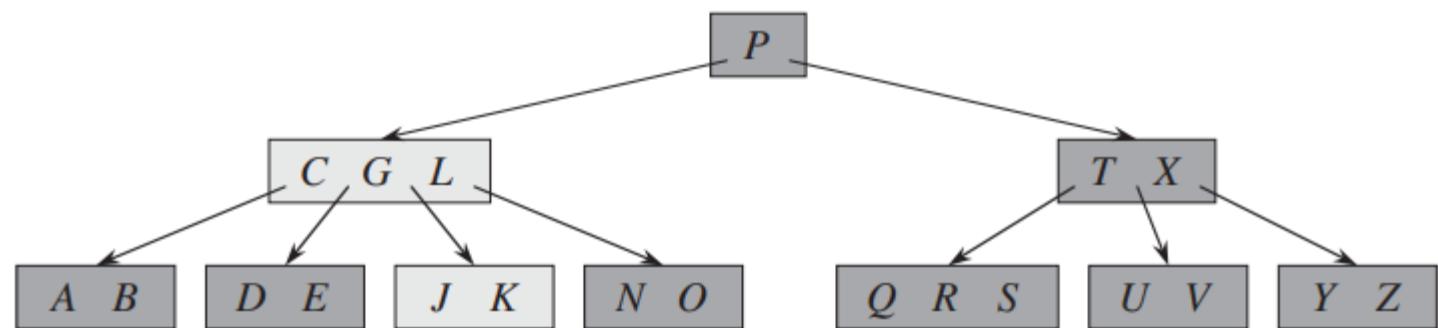
Удаляем ключ М. Случай 2а. Вместо М подставим предшественника из левого потомка – L.



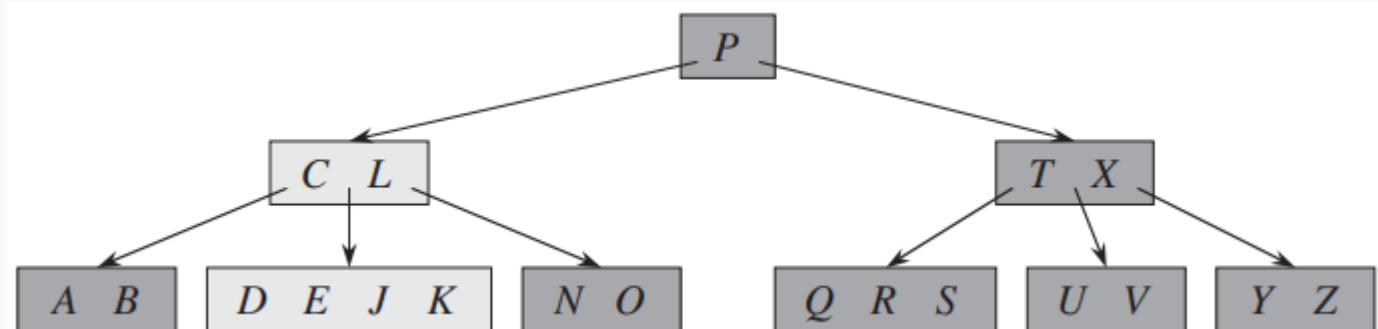
# В-дерево. Удаление ключа. Примеры ситуаций.



Примеры ситуаций при удалении.



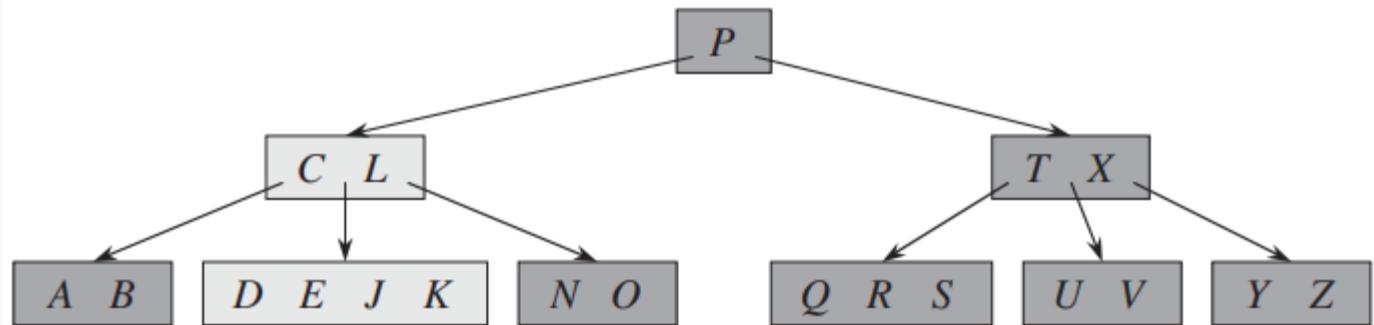
Удаляем ключ G. Случай 2с. Объединянем ключ G и узлы DE и JK в один узел DEGJK. Из CGL убрали 1 ключ и 1 указатель. Рекурсивно удаляем G из нового узла (1-й случай).



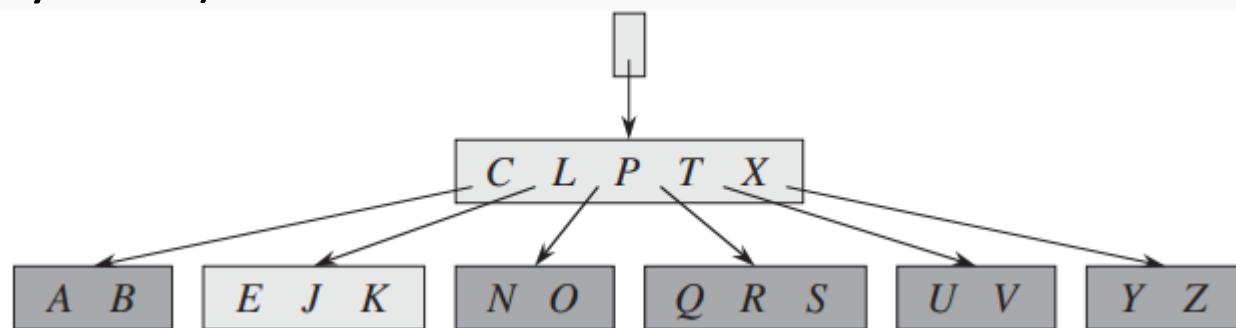
# В-дерево. Удаление ключа. Примеры ситуаций.



Примеры ситуаций при удалении.



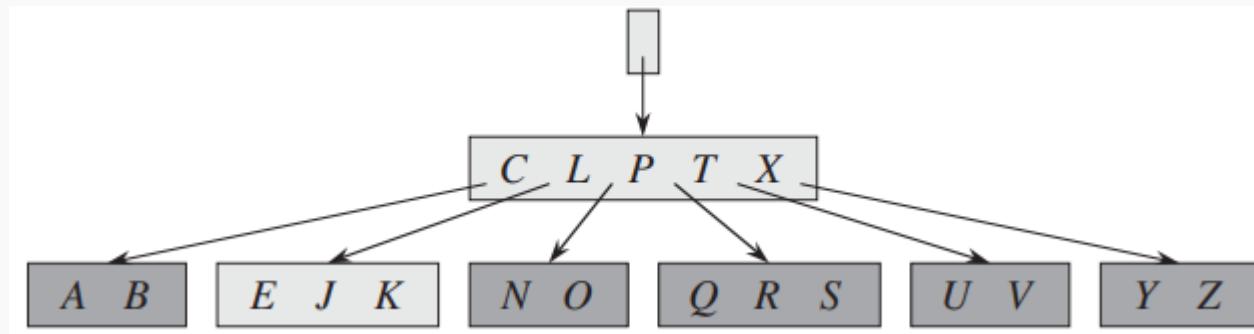
Удаляем ключ D. Случай 3b. Мы не можем просто спуститься в CL – он имеет минимальное допустимое количество ключей. Сливаем CL, Р и TX в новый узел. Затем удаляем D из листа (случай 1).



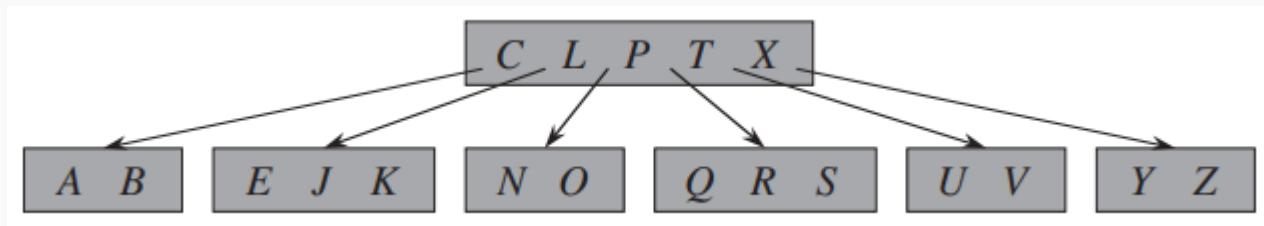
# В-дерево. Удаление ключа. Примеры ситуаций.



Примеры ситуаций при удалении.



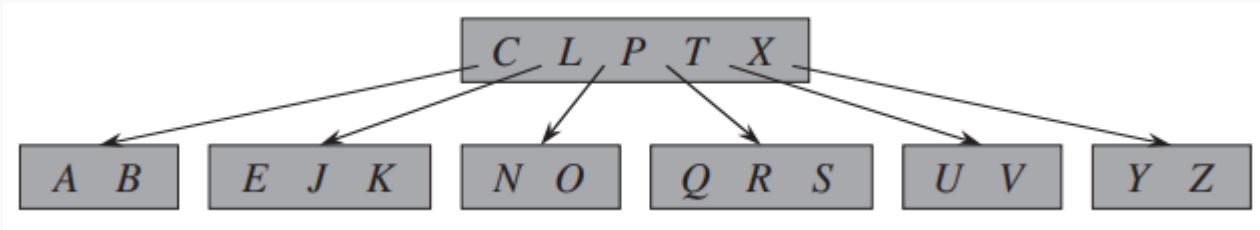
Завершение случая с прошлого слайда. Удаляем пустой корень, теперь корнем стал узел CLPTX. Высота дерева уменьшилась на 1.



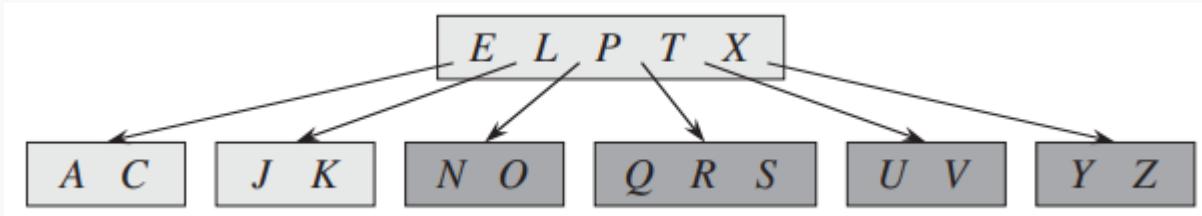
# В-дерево. Удаление ключа. Примеры ситуаций.



Примеры ситуаций при удалении.



Удаляем В. Случай За. С переносим на место В, Е переносим на место С.



# В-дерево. Удаление ключа. Сложность.



В рамках вызовов B-TREE-DELETE производится  $O(h)$  вызовов DISK-READ и DISK-WRITE, значит происходит  $O(h) = O(\log_t N)$  дисковых операций.

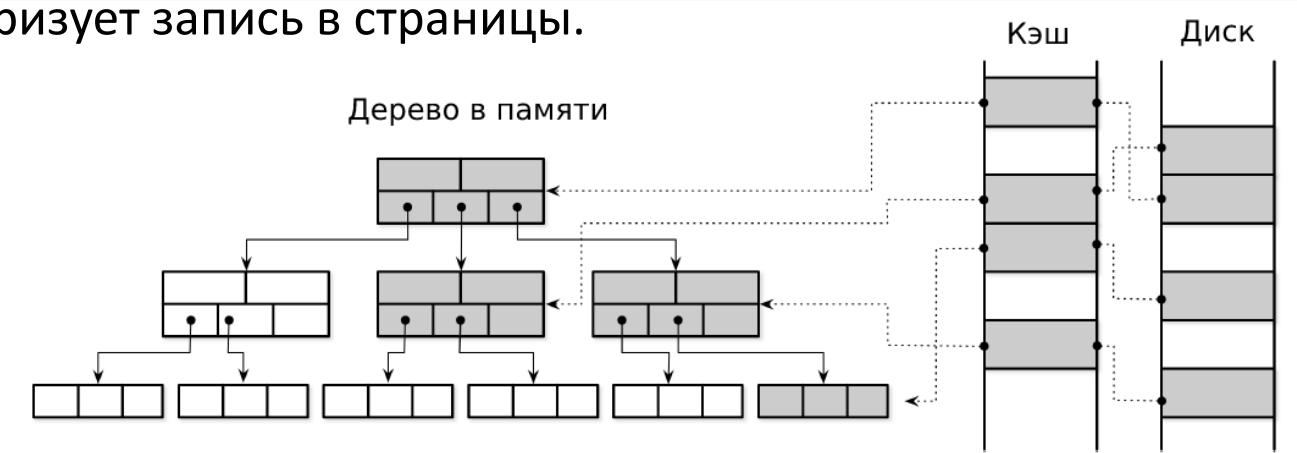
Общая сложность алгоритма удаления ключа  $O(t * h) = O(t * \log_t N)$ .

# В-дерево. Дисковый кэш.



## Назначение дискового кэша:

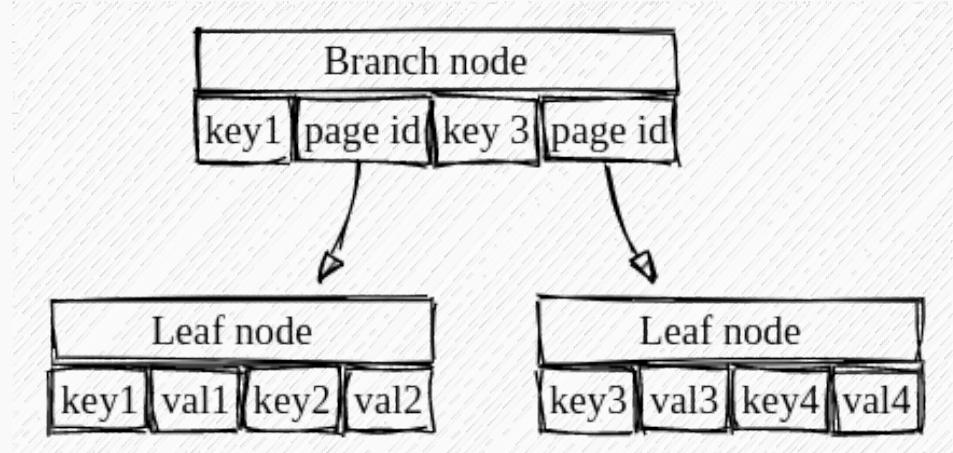
- Хранит содержимое страницы (узла дерева) в RAM.
- Если запрошенной страницы нет в RAM, и там для нее есть место, загружает ее содержимое с диска. Если места в RAM нет, вытесняет одну из загруженных страниц, при этом записывает ее актуальную версию на диск.
- Если запрошенная страница уже загружена в RAM, возвращает закэшированную версию.
- Буферизует запись в страницы.



# B<sup>+</sup>-дерево.



B<sup>+</sup>-дерево – разновидность B-дерева, в котором ассоциированные с ключами значения хранятся только в листьях. Внутренние узлы хранят лишь копии ключей-разделителей и ссылки на потомков. Обычно B<sup>+</sup>-деревья также хранят в листьях ссылки на соседние листья.



# **B<sup>+</sup>-дерево. Сравнение с обычными B-деревьями.**



**Преимущества:**

- Во внутренних узлах не храним значения, поэтому в них помещается больше ключей. Это позволяет увеличить коэффициент ветвления дерева, уменьшить его высоту.
- Если листья связаны ссылками на соседей, удобно делать обход диапазонов ключей.

**Недостатки:**

- Внутренние узлы не могут содержать значений, поэтому всегда придется спускаться до листьев. В обычном B-дереве путь до значения может быть короче.

# B<sup>+</sup>-дерево. Где используется.



- В файловых системах. Например, NTFS, APFS, ext4 (использует extent trees -- модифицированные B<sup>+</sup>-деревья).
- В реляционных БД. Например, Microsoft SQL Server, Oracle (с 8-й версии), SQLite.
- No-SQL БД. Например, CouchDB, MongoDB (при использовании подсистемы хранения WiredTiger), Tokyo Cabinet.

**Определение 4.** Ассоциативный массив — абстрактный тип данных, позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:

- INSERT(ключ, значение).
- FIND(ключ). Возвращает значение, если есть пара с заданным ключом.
- REMOVE(ключ).

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

## Расширение ассоциативного массива.

Обязательные три операции часто дополняются другими. Наиболее популярные расширения включают следующие операции:

- CLEAR — удалить все записи.
- EACH — «пробежаться» по всем хранимым парам
- MIN — найти пару с минимальным значением ключа
- MAX — найти пару с максимальным значением ключа

В последних двух случаях необходимо, чтобы на ключах была определена операция сравнения.

# АТД «Ассоциативный массив»



Структура данных	Поиск	
	в среднем	в худшем случае
Хэш-таблица	$O(1)$	$O(n)$
Сбалансированное дерево поиска	$O(\log n)$	$O(\log n)$
Несбалансированное дерево поиска	$O(\log n)$	$O(n)$
Неотсортированный массив	$O(n)$	$O(n)$
Отсортированный массив	$O(\log n)$	$O(\log n)$

# АТД «Ассоциативный массив»



Структура данных	Вставка	
	в среднем	в худшем случае
Хэш-таблица	O(1)	O( $n$ )
Сбалансированное дерево поиска	O(log $n$ )	O(log $n$ )
Несбалансированное дерево поиска	O(log $n$ )	O( $n$ )
Неотсортированный массив	O(1)	O( $n$ )
Отсортированный массив	O( $n$ )	O( $n$ )

# АТД «Ассоциативный массив»



Структура данных	Удаление	
	в среднем	в худшем случае
Хэш-таблица	$O(1)$	$O(n)$
Сбалансированное дерево поиска	$O(\log n)$	$O(\log n)$
Несбалансированное дерево поиска	$O(\log n)$	$O(n)$
Неотсортированный массив	$O(n)$	$O(n)$
Отсортированный массив	$O(n)$	$O(n)$

# АТД «Ассоциативный массив»



Структура данных	Ключи упорядочены
Хэш-таблица	Нет
Сбалансированное дерево поиска	Да
Несбалансированное дерево поиска	Да
Неотсортированный массив	Нет
Отсортированный массив	Да

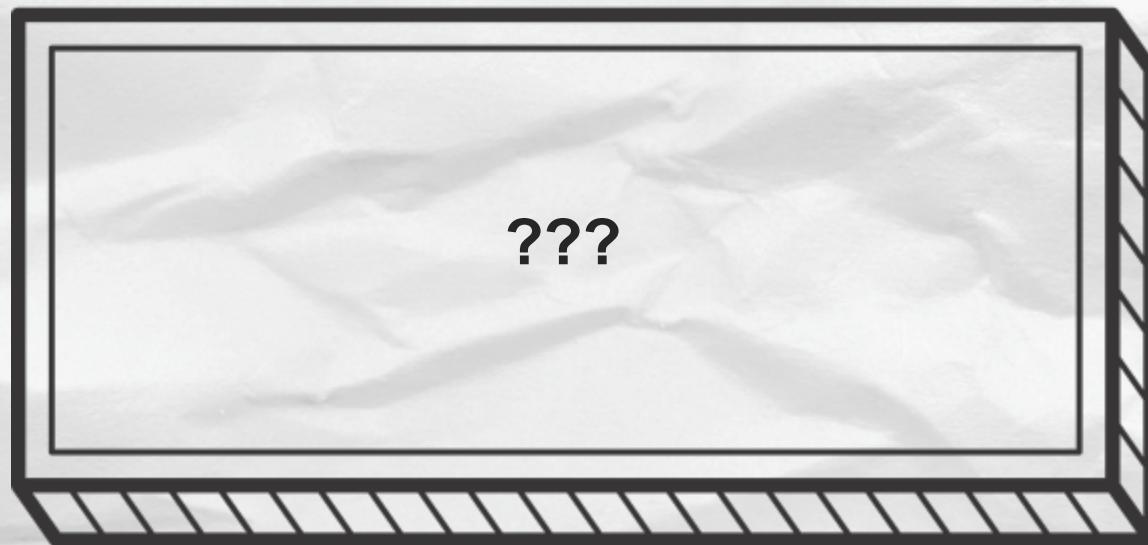
## Ассоциативные массивы в STL:

- std::map и std::multimap реализованы на красно-черных деревьях
- std::unordered\_map и std::unordered\_multimap на хэш-таблицах

# Визуализаторы AVL, красно-черных и В-деревьев



- <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>
- <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>
- <https://www.cs.usfca.edu/~galles/visualization/BTree.html>



**Спасибо за  
внимание!**