Лекция 4
Деревья поиска
Алгоритмы и структуры данных

Глушенков Д.А.

План лекции

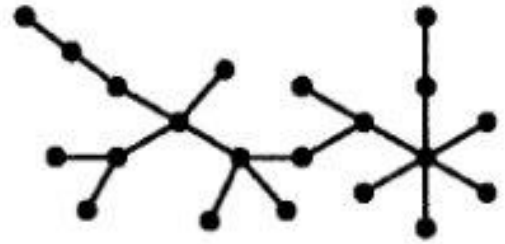
1. Определения, примеры деревьев
2. Представление в памяти
3. Обходы дерева в глубину, в ширину
4. Двоичные деревья поиска
5. АВЛ-деревья
6. Красно-черные деревья
7. B-деревья
8. АТД «Ассоциативный массив»

Определения деревьев

Определение 1.

Дерево – 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 (– неориентированный связный граф без циклов.

Определения деревьев

Определение 3.

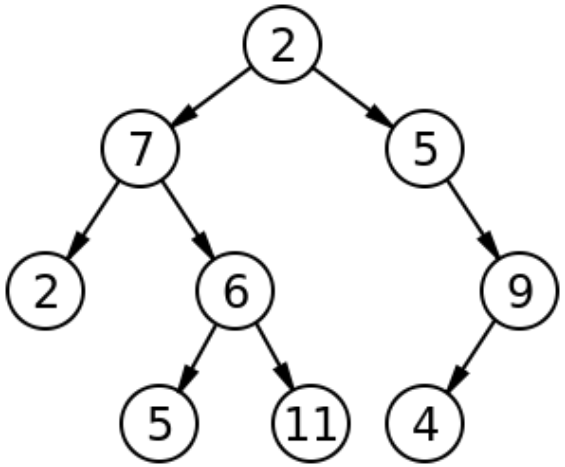
Дерево с корнем — дерево, в котором один узел выделен и назначен «корнем» дерева.

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 y располагается под узлом x (а x располагается над y), если x располагается на пути от y к корню.



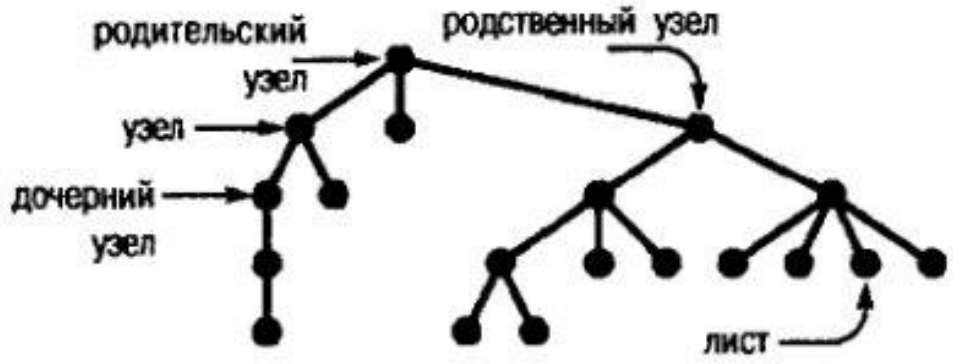
Определения деревьев

Определение 5.

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

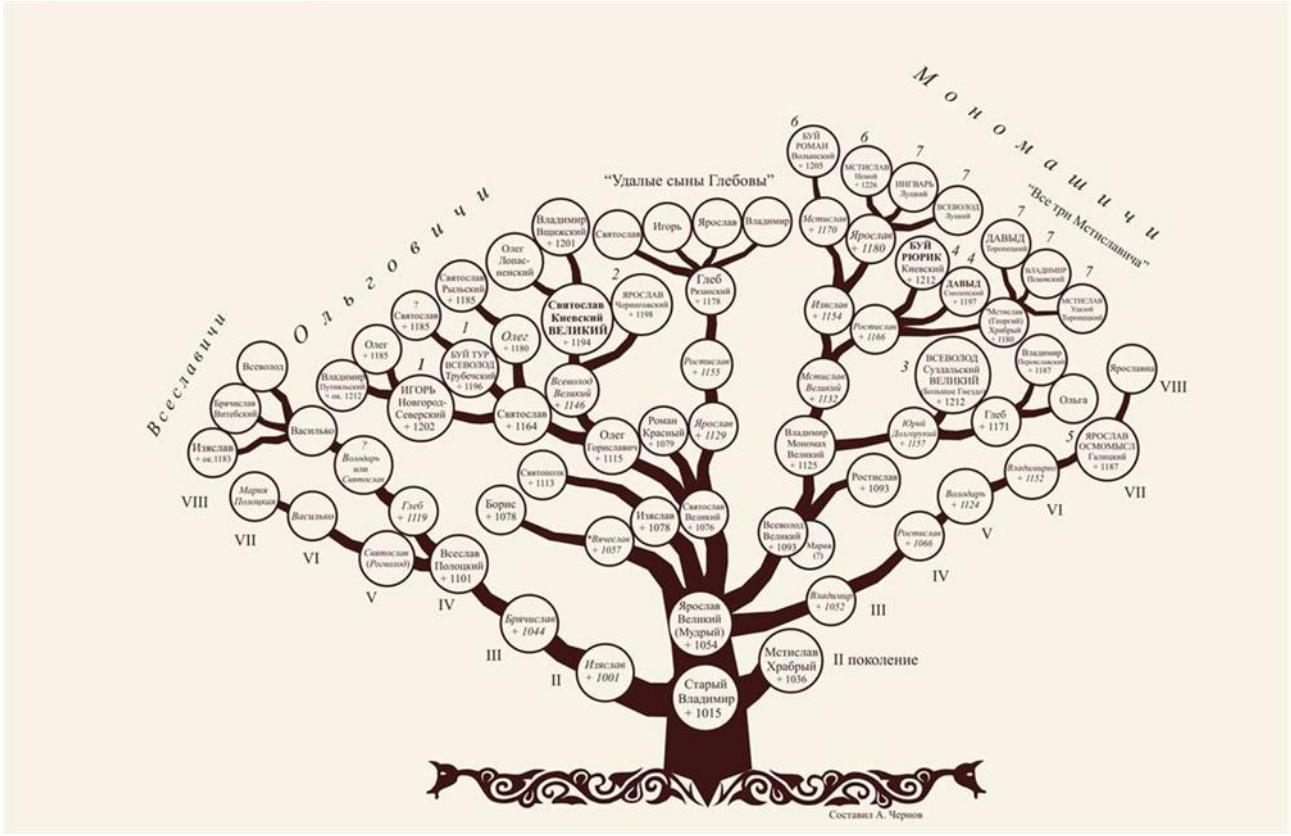
Узлы, расположенные непосредственно под данным узлом, называются его **дочерними** узлами.

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



Примеры деревьев

Генеалогическое дерево



Примеры деревьев

XML

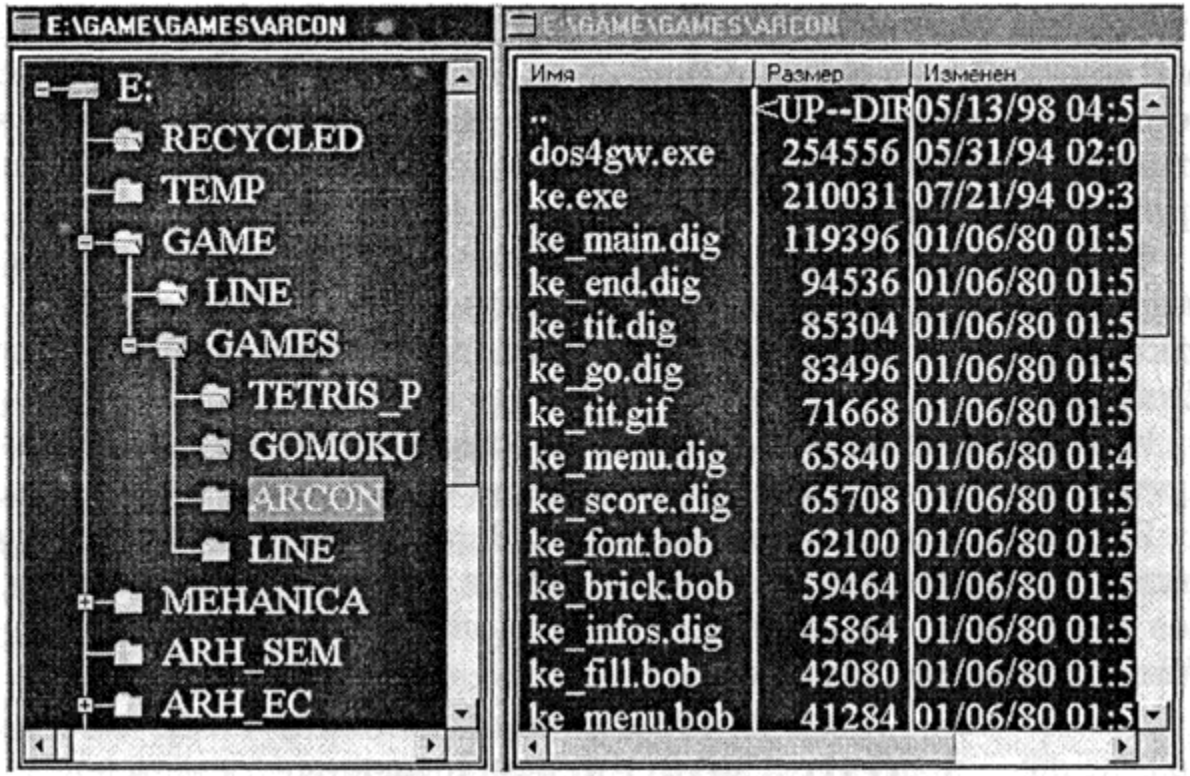
```
<?xml version="1.0"?>
- <job>
  - <production>
      <ApprovalType>WebCenter</ApprovalType>
      <Substrate>carton 150 gr</Substrate>
      <SheetSize>220-140</SheetSize>
      <press>SuperFlat2</press>
      <finishing>standard</finishing>
      <urgency>normal</urgency>
    </production>
  - 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

Примеры деревьев

Организационная структура компании

Примеры деревьев

Иерархическая файловая система



Число вершин и ребер

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

Доказательство.
Самая глубокая вершина является листовой.

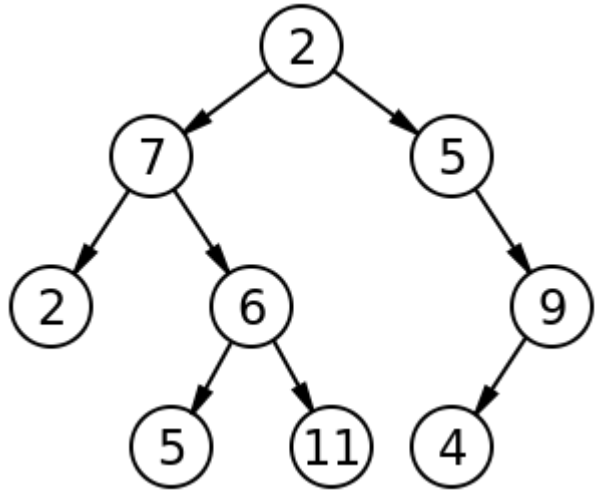
Утверждение 2. Дерево, состоящее из N вершин, содержит N – 1 ребро.

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

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

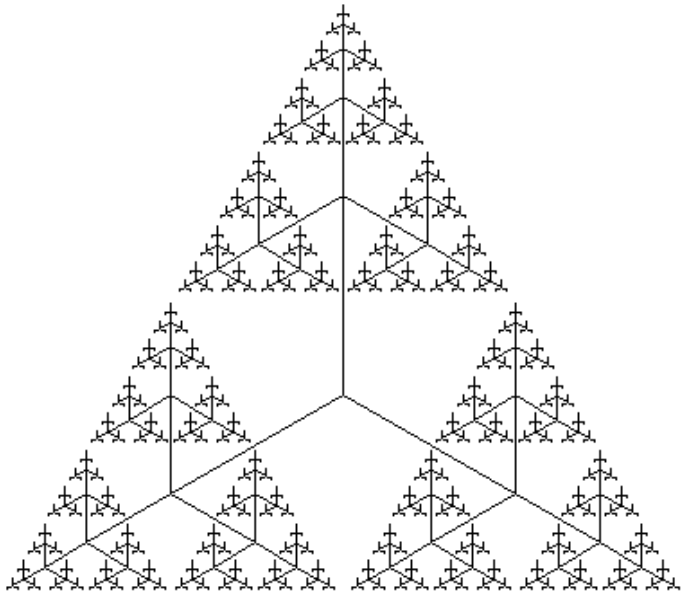
Двоичное (бинарное) дерево с корнем — это дерево, в котором каждая вершина имеет не более двух дочерних вершин.



Виды деревьев

N-арное дерево — это дерево, в котором степени вершин не превосходят N + 1.

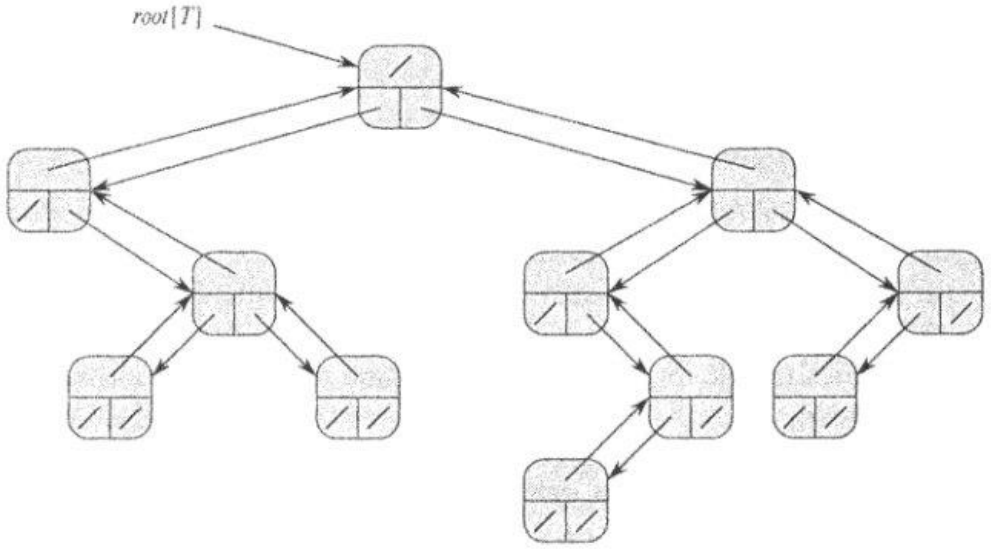
N-арное дерево с корнем — это дерево, в котором каждая вершина имеет не более N дочерних вершин.



Структуры данных

СД «Двоичное дерево» — представление двоичного дерева с корнем.

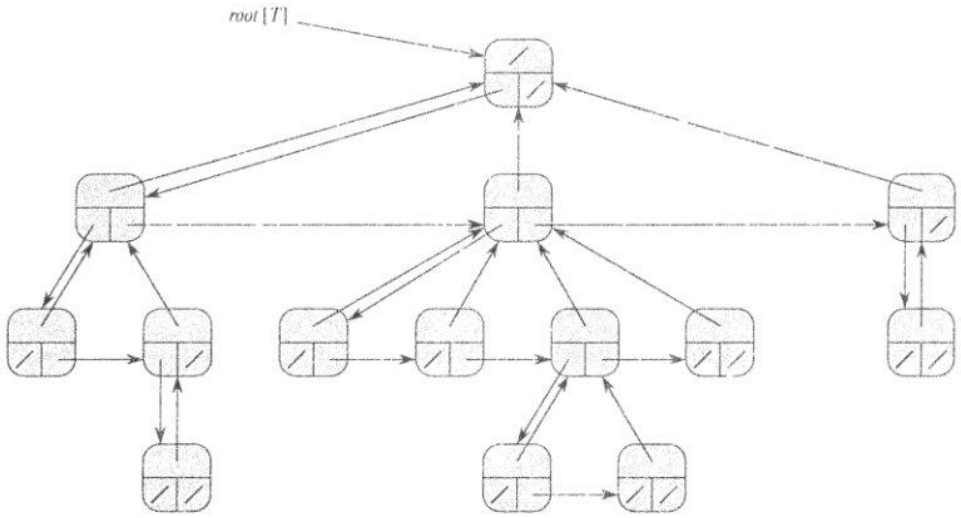
Узел – структура, содержащая данные и указатели на левый и правый дочерний узел. Также может содержать указатель на родительский узел.



Структуры данных

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

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

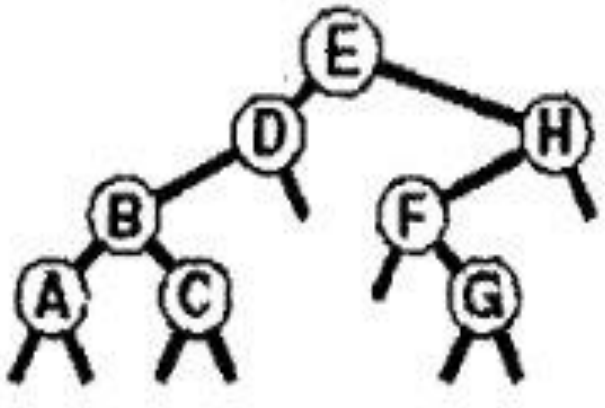
Обходом двоичного дерева в глубину (DFS) называется процедура, выполняющая в некотором заданном порядке следующие действия с поддеревом:

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

DFS – Depth First Search.

Обход дерева в глубину

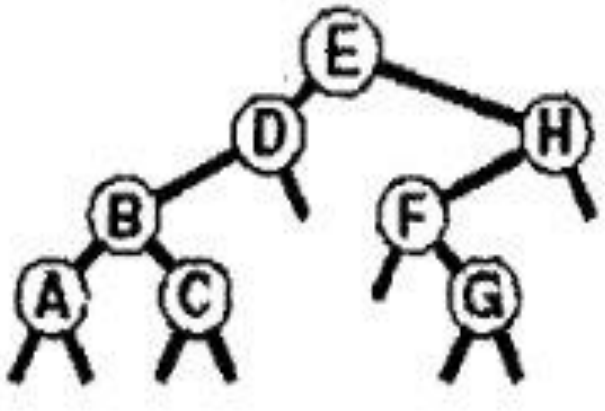
Прямой обход (сверху вниз, pre-order). Вначале обрабатывается узел, затем посещается левое и правые поддеревья.



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

Обход дерева в глубину

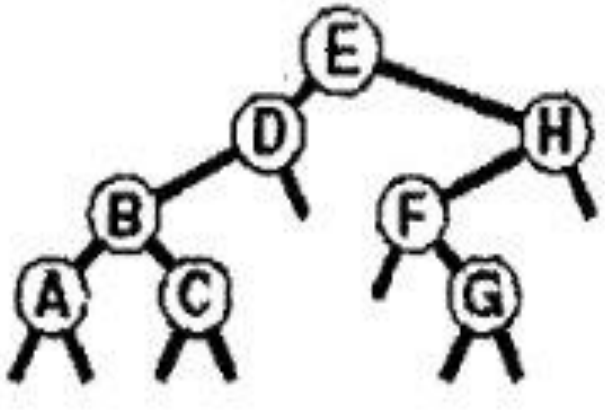
Обратный обход (снизу вверх, post-order). Вначале посещаются левое и правое поддеревья, а затем обрабатывается узел.



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;
};
```

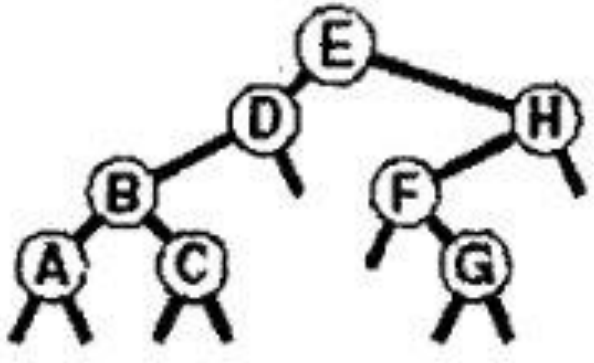
Обход дерева в глубину

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

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



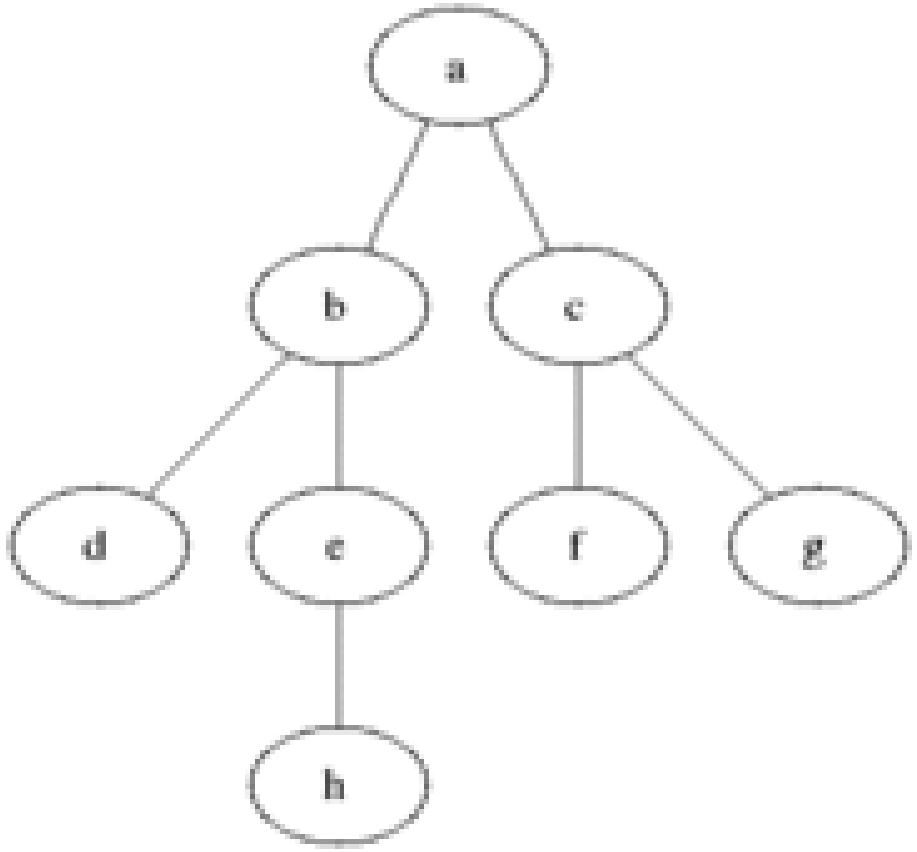
Используется очередь, в которой хранятся вершины, требующие просмотра. За одну итерацию алгоритма:

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

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

Обход дерева в ширину

Черные вершины посещены.
Серые вершины еще не посещены, но уже в очереди (посетили родительскую вершину).
Белые вершины еще не рассмотрены.



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 );
    }
};
```

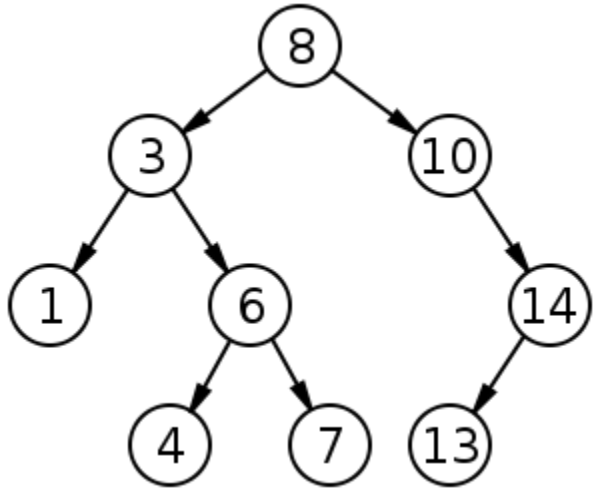
Разница между BFS и DFS

BFS (в ширину)	DFS (в глубину)
Сложность по памяти – $O(w)$, где w – максимальная ширина дерева (максимальная ширина на глубине $h = 2^h$). Экономичнее в несбалансированных деревьях.	Сложность по памяти – $O(h)$, где h – максимальная глубина дерева (максимальная глубина в дереве из n элементов = n). Экономичнее в сбалансированных деревьях.
Эффективнее, если при обходе нас интересуют элементы, находящиеся близко к корню.	Эффективнее, если при обходе нас интересуют элементы, находящиеся близко к листьям.

Двоичные деревья поиска

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.



Двоичные деревья поиска

Операции с двоичным деревом поиска:

- Поиск по ключу.
- Поиск минимального, максимального ключей.
- Вставка.
- Удаление.
- 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.

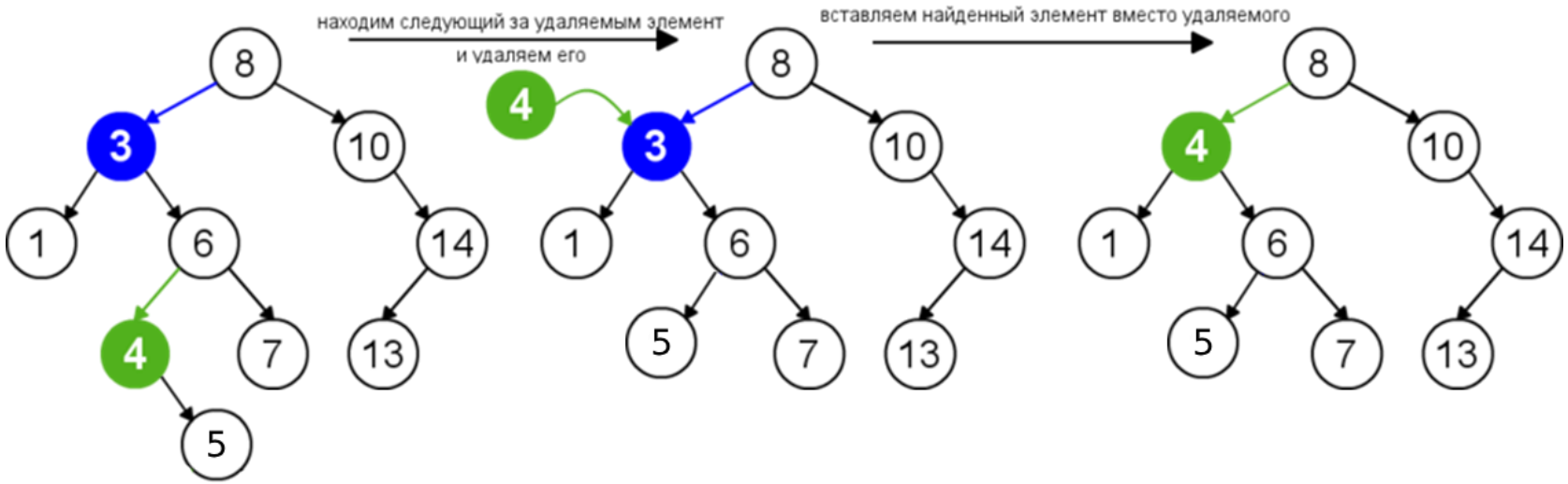
Двоичные деревья поиска

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

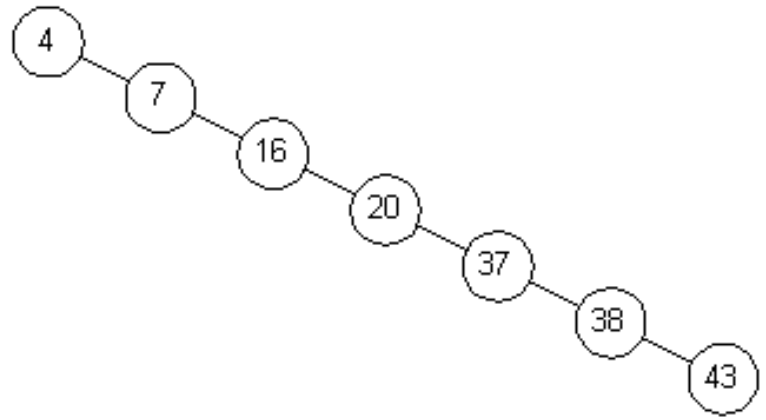
```
// Удаление узла.
void DeleteNode( CNode*& node )
{
    if( node->Left == 0 ) { // Если нет левого поддерева.
        CNode* right = node->Right; // Подставляем правое, может быть 0.
        delete node;
        node = right;
    } else if( node->Right == 0 ) { // Если нет правого поддерева.
        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.

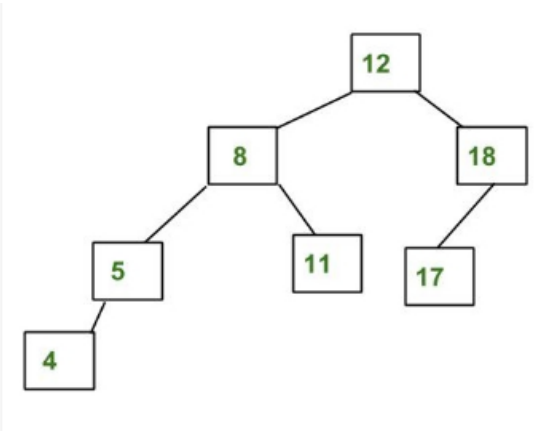


АВЛ-дерево

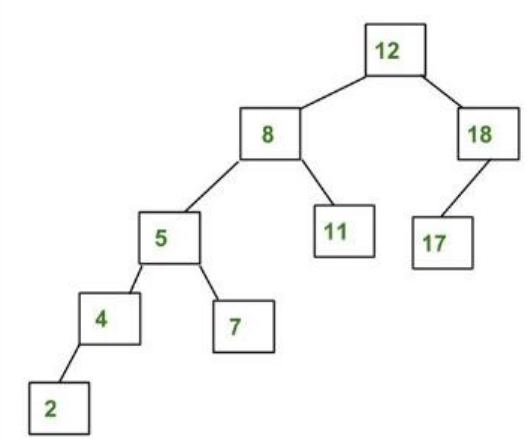
АВЛ-дерево — сбалансированное двоичное дерево поиска. Для каждой его вершины высоты поддеревьев различаются не более чем на 1.

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

AVL дерево



Не AVL дерево

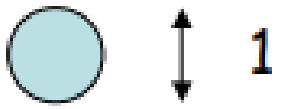


АВЛ-дерево. Оценка высоты.

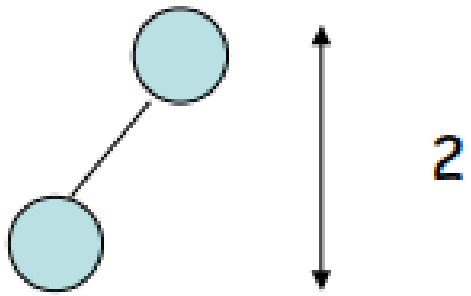
Теорема. Высота АВЛ-дерева $h = O(logN)$.

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

$n(1) = 1$

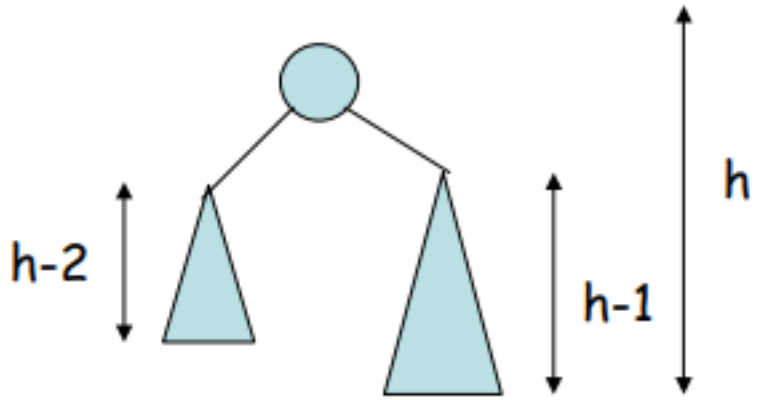


$n(2) = 2$



АВЛ-дерево. Оценка высоты.

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. Оценка высоты.

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

Так как $n(h-1) > n(h-2)$, то

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

Таким образом, $n(h) > 2n(h-2)$

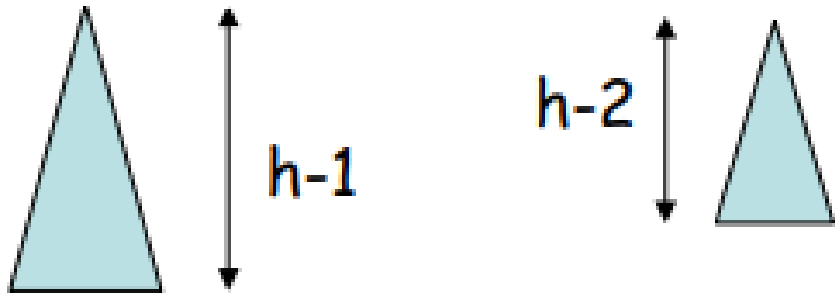
Заметим, что

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

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

…

То есть $n(h) > 2^i n(h-2i)$



АВЛ-дерево. Оценка высоты.

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

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

Прологарифмируем обе части неравенства:

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

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

То есть, высота АВЛ-дерева $h = O(logN)$

АВЛ-дерево. Восстановление свойств.

Специальные балансирующие операции:

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

АВЛ-дерево. Восстановление свойств.

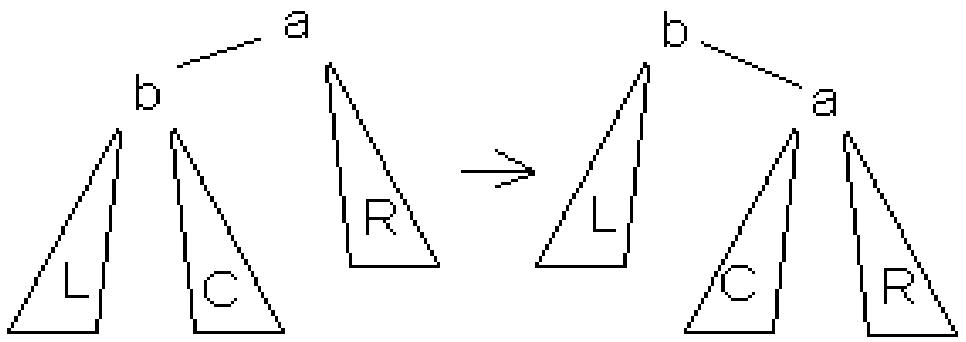
Малое правое вращение

Используется, когда:
$\text{высота}(L) = \text{высота}(R) + 2$,
причем $\text{высота}(C) \leq \text{высота}(L)$.

После операции:

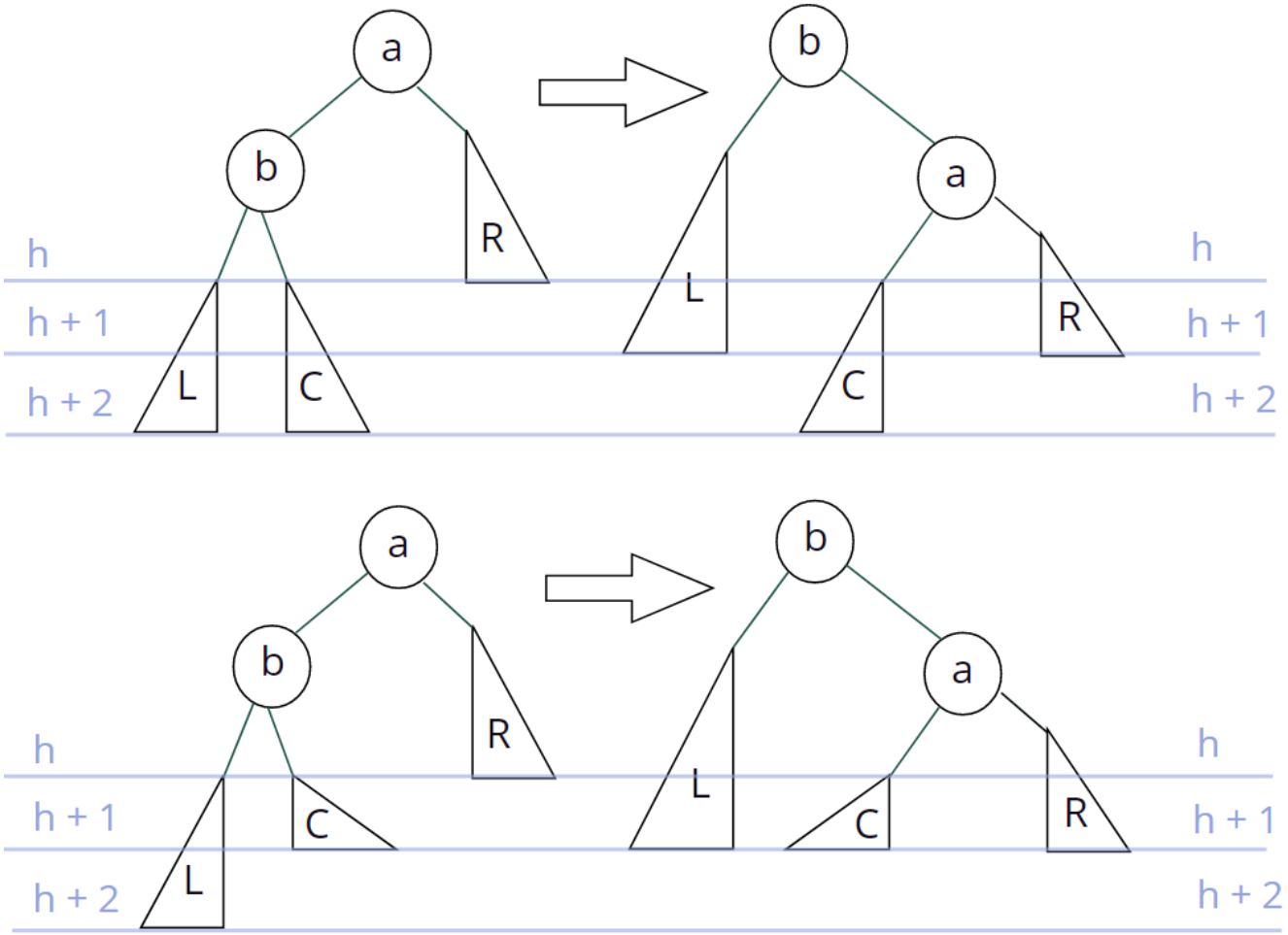
высота дерева останется прежней,
если $\text{высота}(C) = \text{высота}(L)$,

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



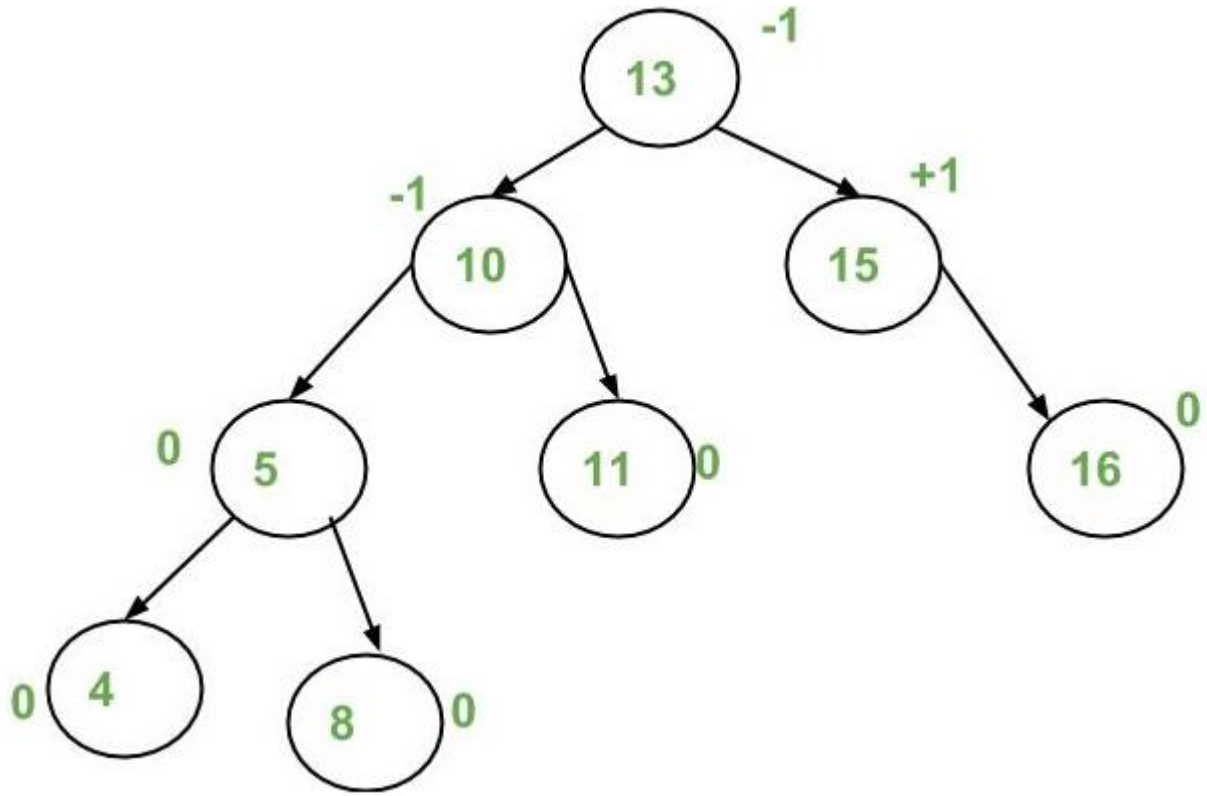
АВЛ-дерево. Малое правое вращение.

Каким образом малое правое вращение устраняет дисбаланс высот?



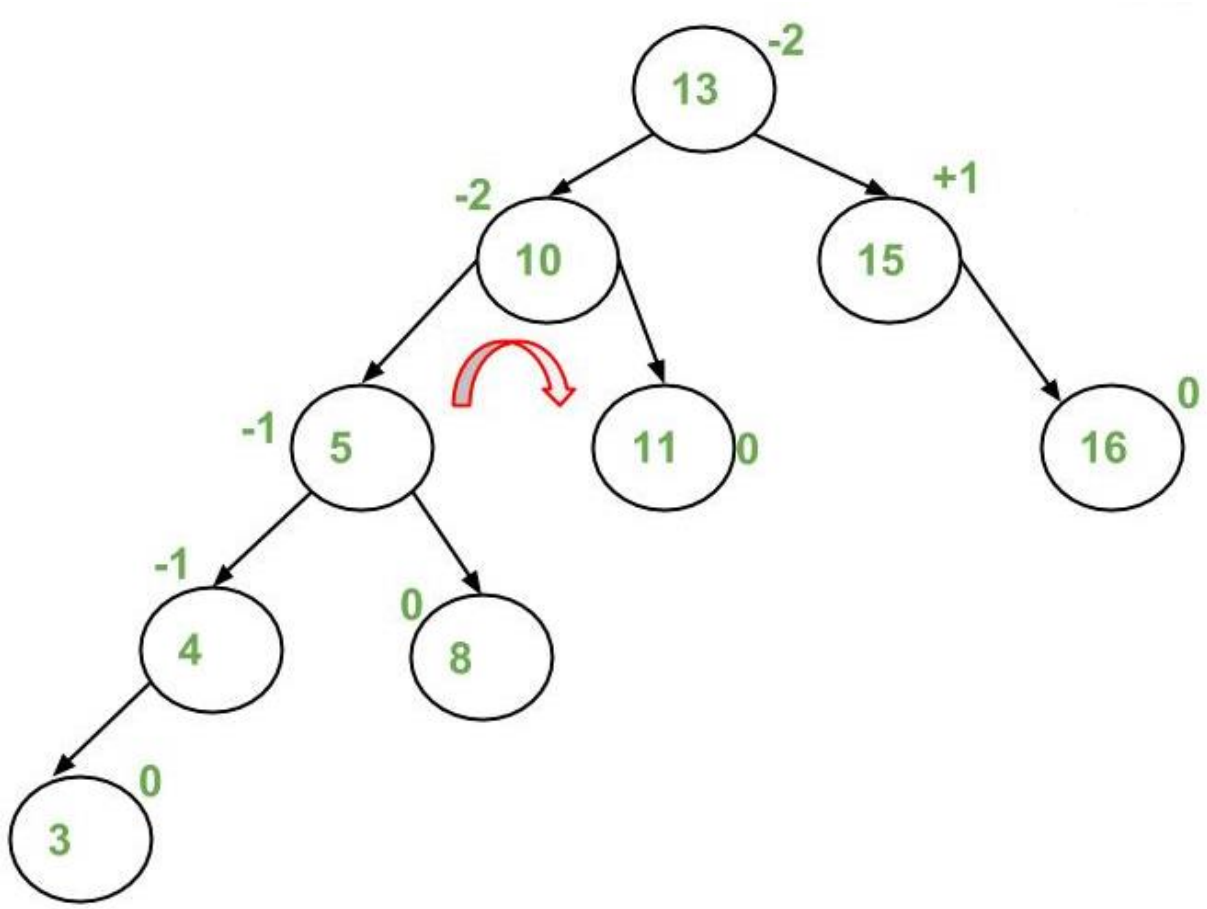
АВЛ-дерево. Малое правое вращение.

Добавим элемент 3.



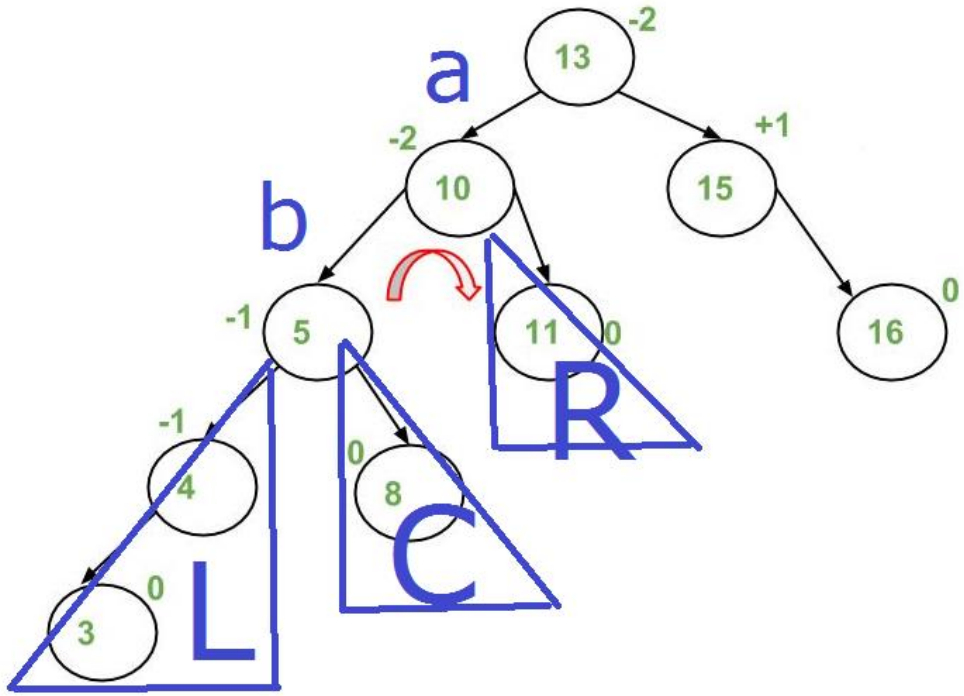
АВЛ-дерево. Малое правое вращение.

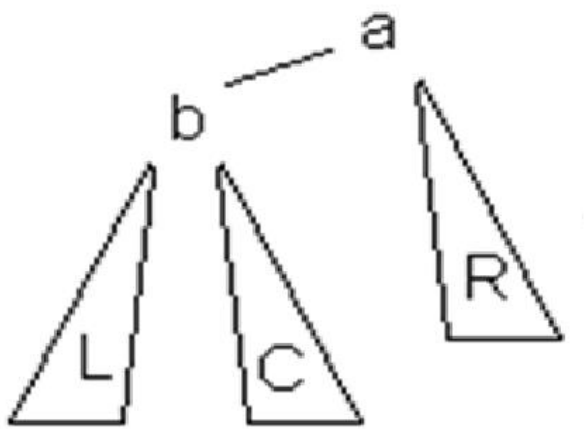
Больше не АВЛ-дерево,
нужно балансировать.



АВЛ-дерево. Малое правое вращение.

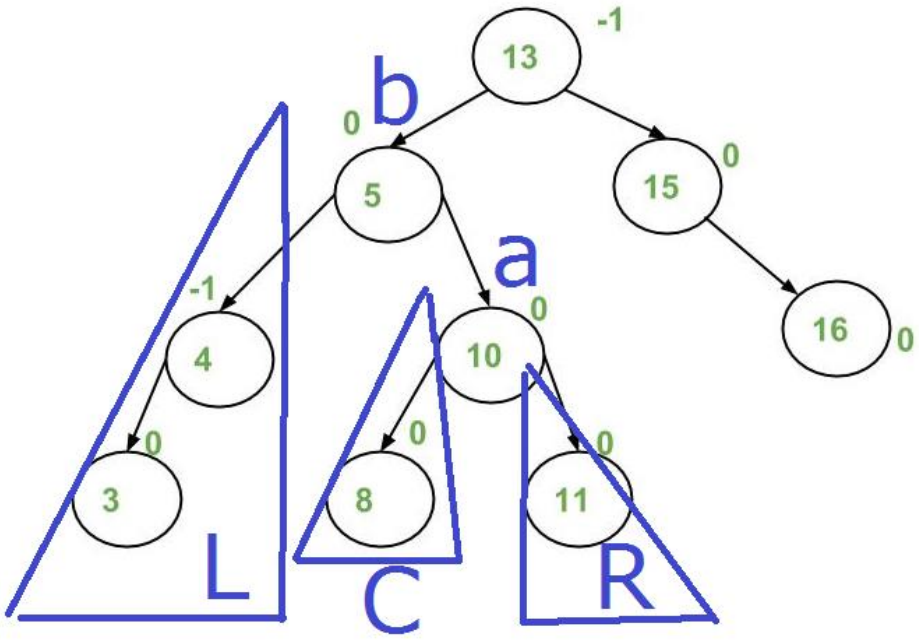
Делаем малый правый поворот.

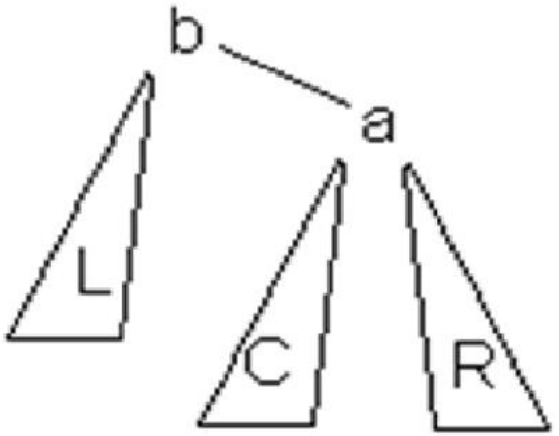




АВЛ-дерево. Малое правое вращение.

Опять стало АВЛ-деревом.

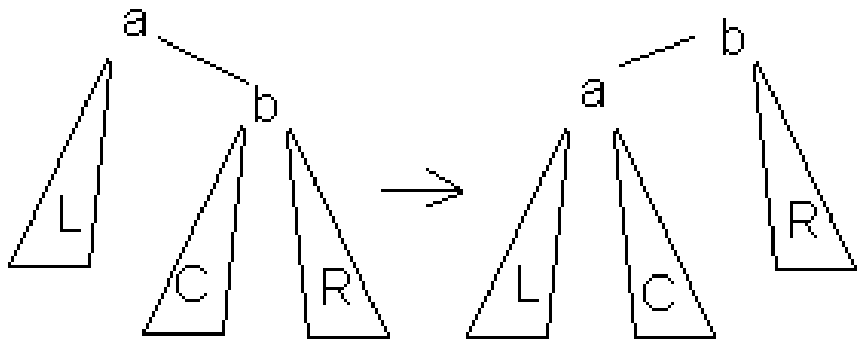




АВЛ-дерево. Восстановление свойств.

Малое левое вращение

Используется, когда:
$\text{высота}(R) = \text{высота}(L) + 2$,
причем $\text{высота}(C) \leq \text{высота}(R)$.



После операции:

высота дерева останется прежней,
если $\text{высота}(C) = \text{высота}(R)$,

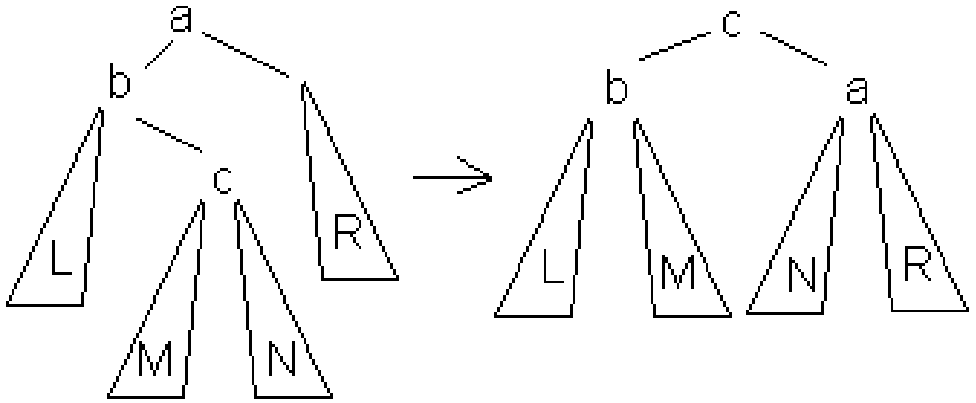
высота дерева уменьшится на 1,
если $\text{высота}(C) < \text{высота}(R)$.

АВЛ-дерево. Восстановление свойств.

Большое правое вращение

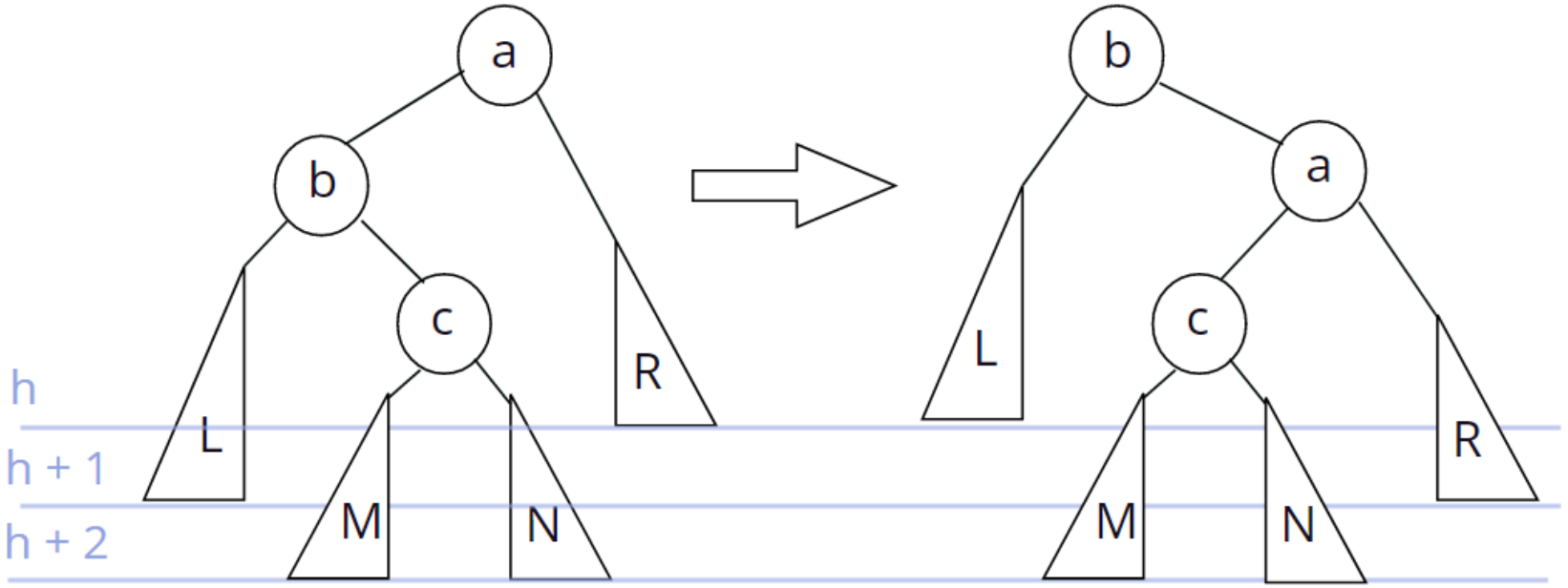
Используется, когда:
$\text{высота}(L) = \text{высота}(R) + 1$, причем
$\text{высота}(C) = \text{высота}(R) + 2$.

После операции:
высота дерева уменьшается на 1.



АВЛ-дерево. Большое правое вращение.

Почему малое правое вращение не поможет?



Разницу высот устранить не смогли! $\text{высота}(L) - \text{высота}(c) = 2$.

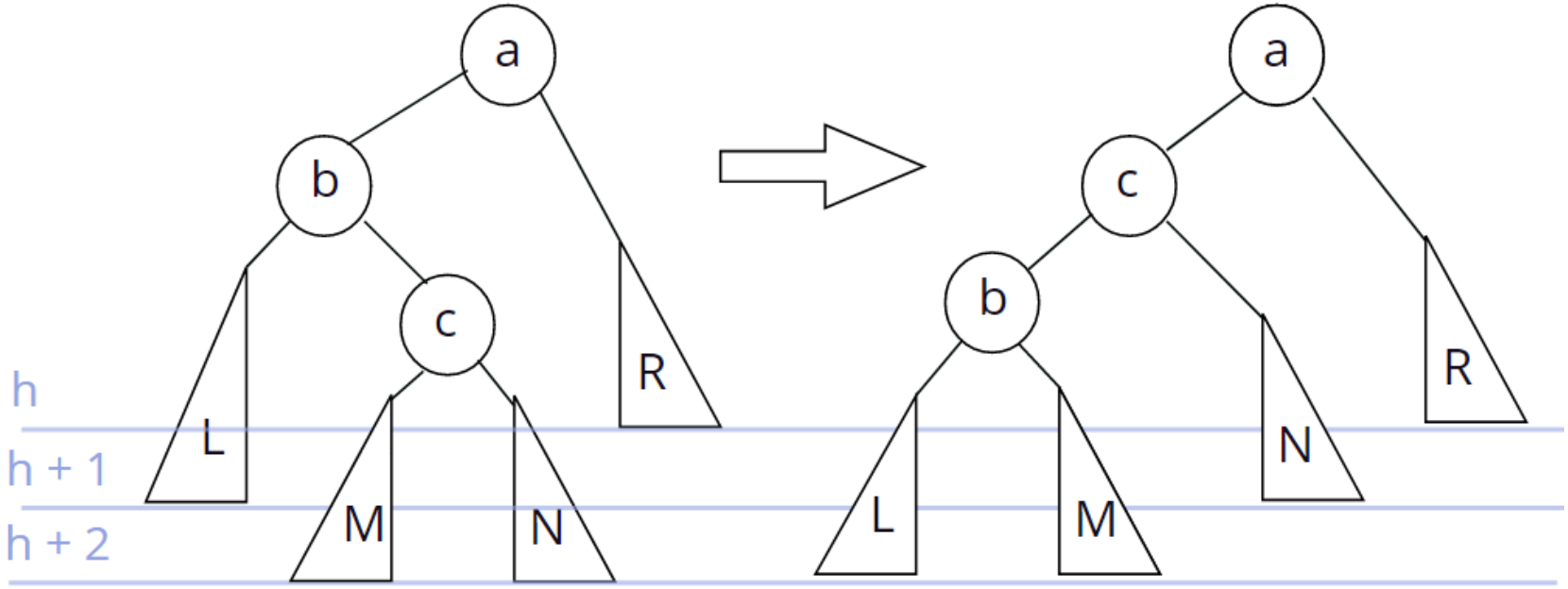
АВЛ-дерево. Большое правое вращение.

Шаг 1. Малое левое вращение в b.

Также возможно:

$\text{Высота}(M) = h + 1, \text{Высота}(N) = h + 2$

$\text{Высота}(M) = h + 2, \text{Высота}(N) = h + 1$



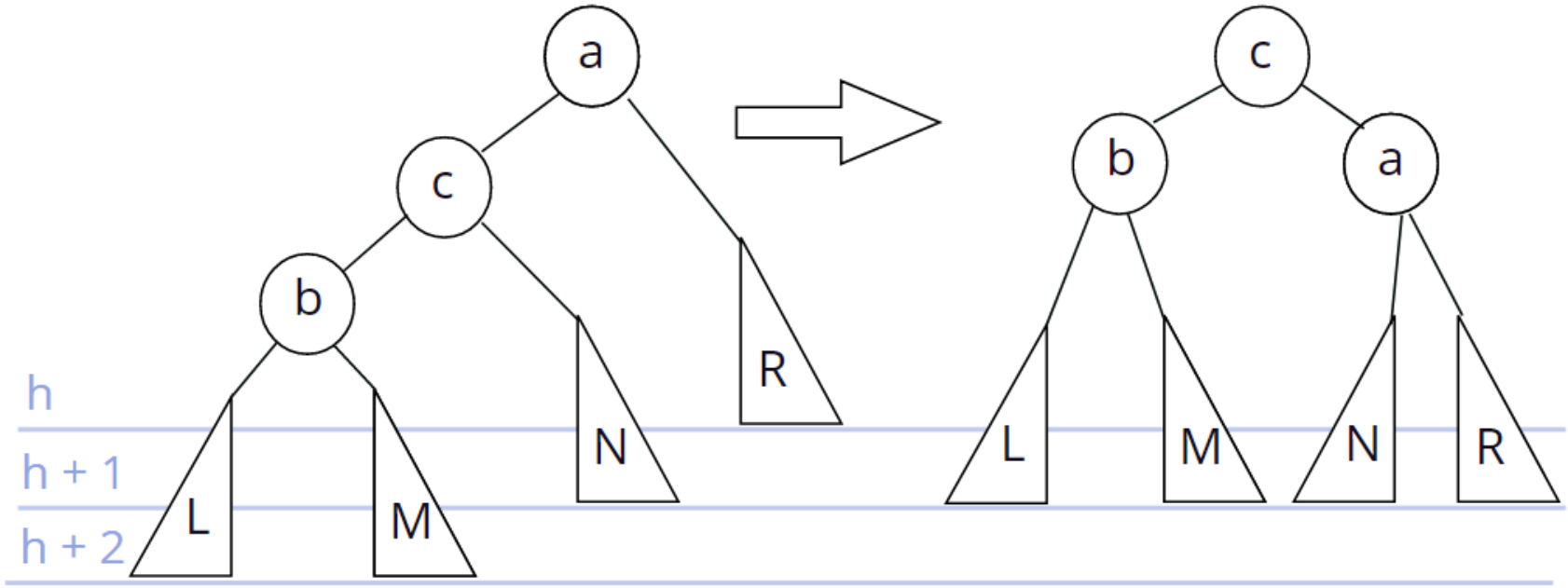
АВЛ-дерево. Большое правое вращение.

Шаг 2. Малое правое вращение в a.

Если в изначальном дереве $\text{высота}(M)$ была $h + 1$, то теперь она h

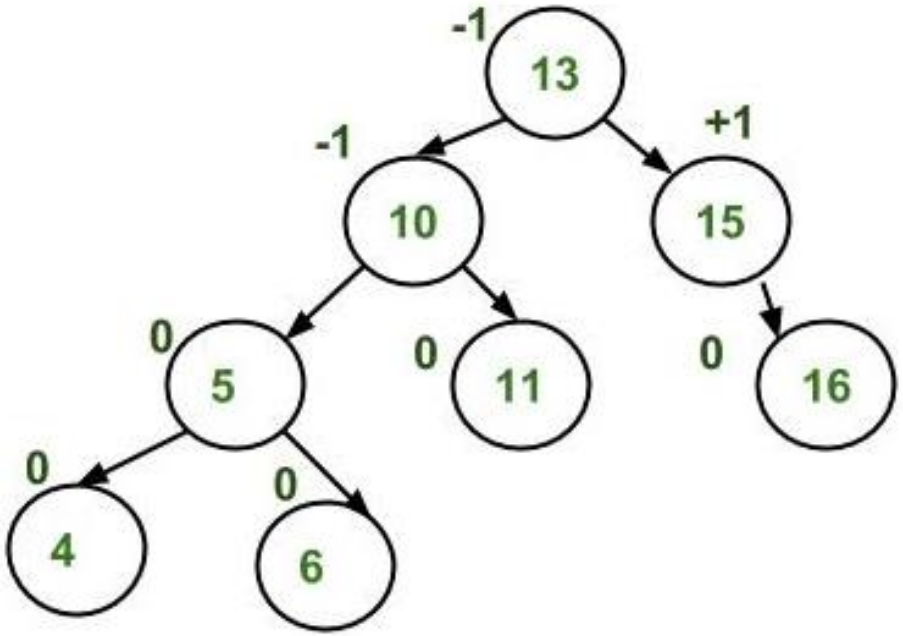
Если в изначальном дереве $\text{высота}(N)$ была $h + 1$, то теперь она h

В любом случае, мы исправили дисбаланс и уменьшили высоту на 1.



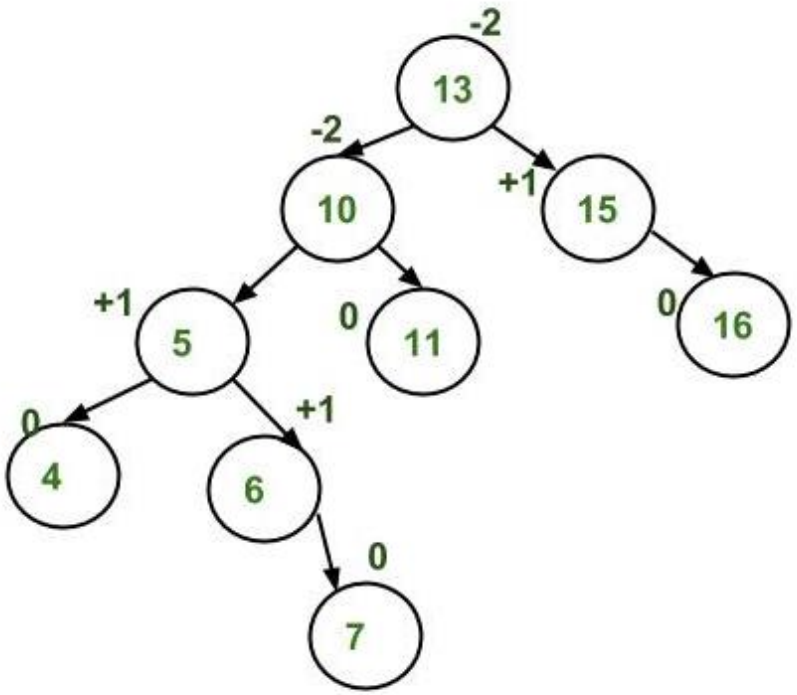
АВЛ-дерево. Большое правое вращение.

Добавим элемент 7.



АВЛ-дерево. Большое правое вращение.

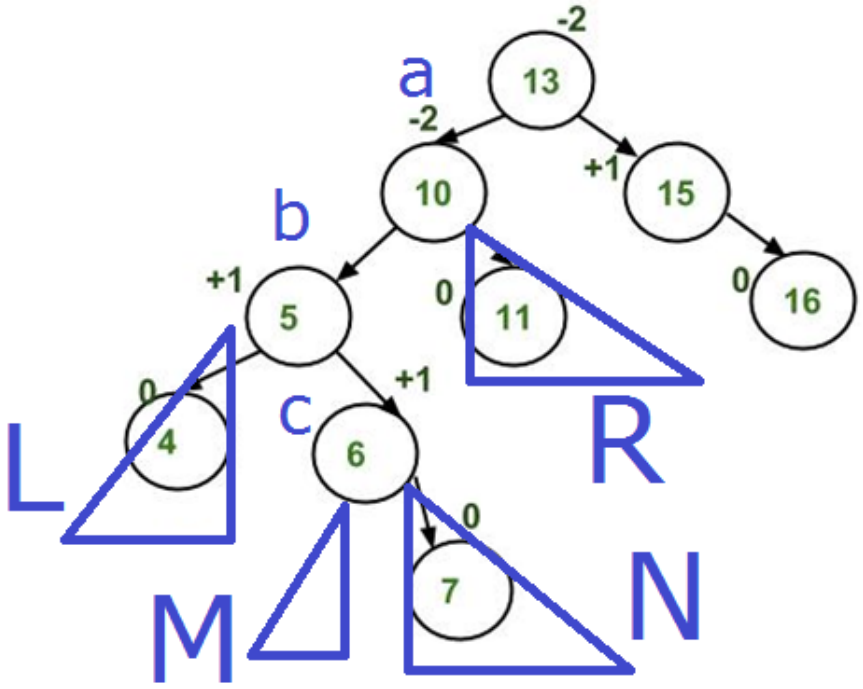
Нарушили баланс, больше не АВЛ-дерево.

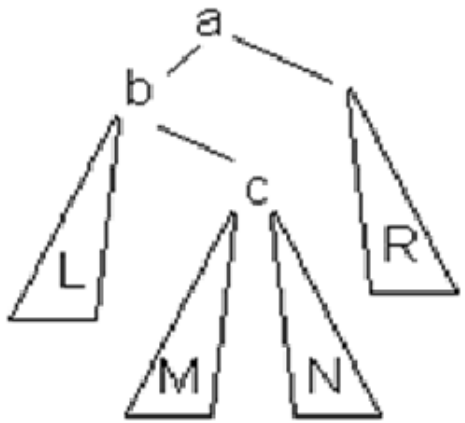


АВЛ-дерево. Большое правое вращение.

Необходимо большое правое вращение:

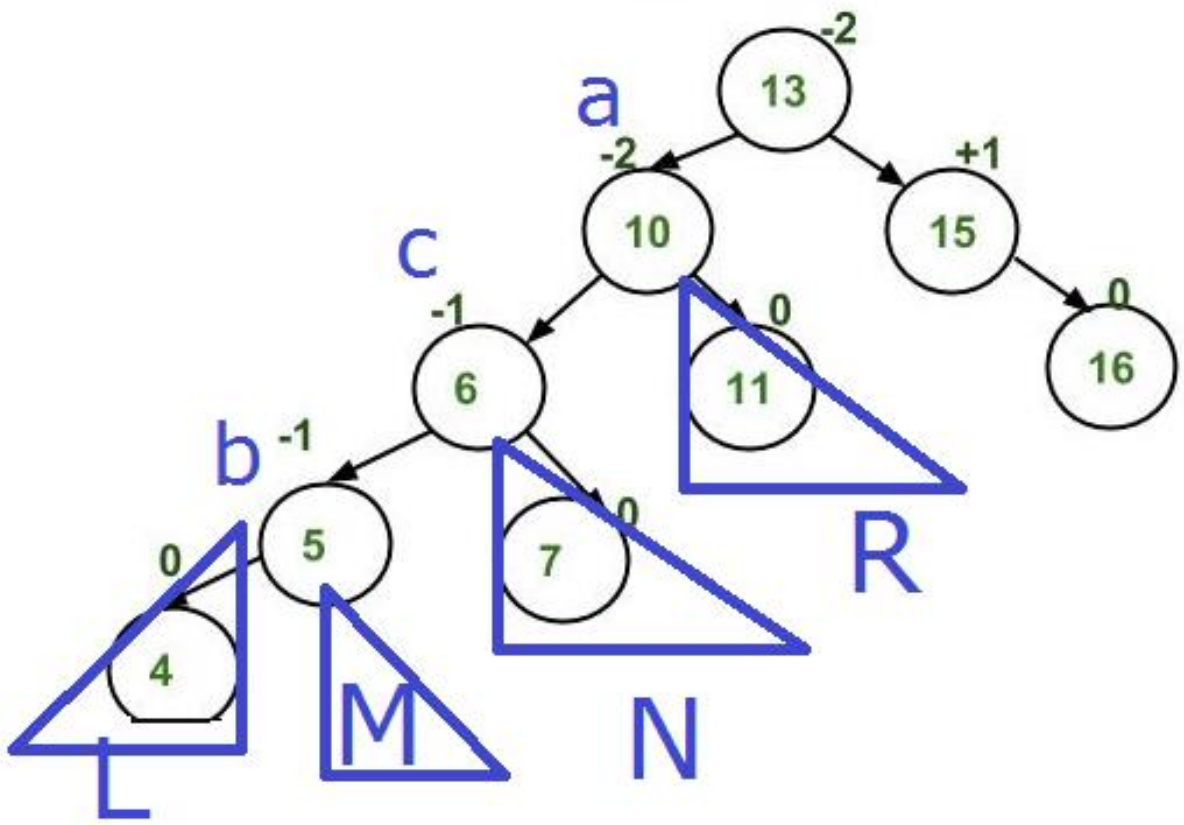
1. Малое левое вращение в b.
2. Малое правое вращение в a.





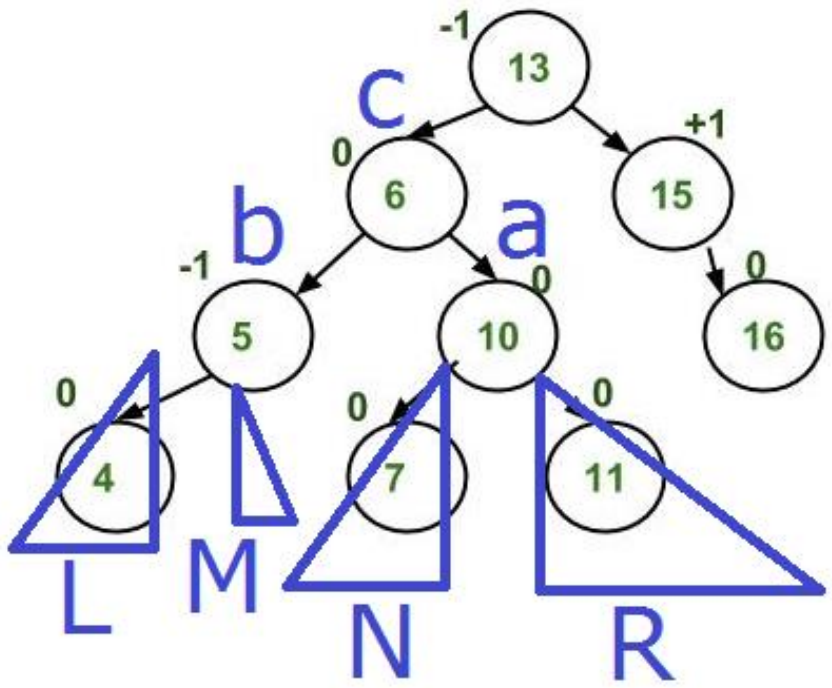
АВЛ-дерево. Большое правое вращение.

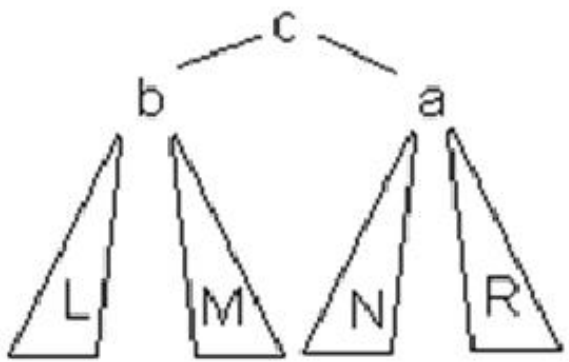
Сделали малое левое вращение:



АВЛ-дерево. Большое правое вращение.

Сделали малое правое вращение. Восстановили свойства АВЛ-дерева:



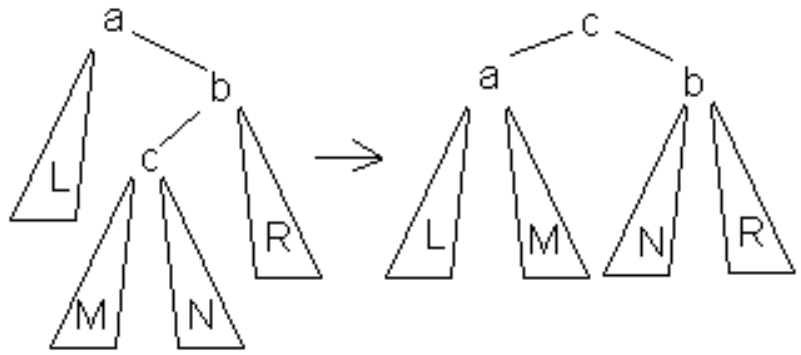


АВЛ-дерево. Восстановление свойств.

Большое левое вращение

Используется, когда:
$\text{высота}(R) = \text{высота}(L) + 1$, причем
$\text{высота}(C) = \text{высота}(L) + 2$.

После операции:
высота дерева уменьшается на 1.



АВЛ-дерево

Вставка элемента

1. Проходим по пути поиска, пока не убедимся, что ключа в дереве нет.
2. Добавляем 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 $= O(\log(n))$.

АВЛ-дерево

Удаление элемента

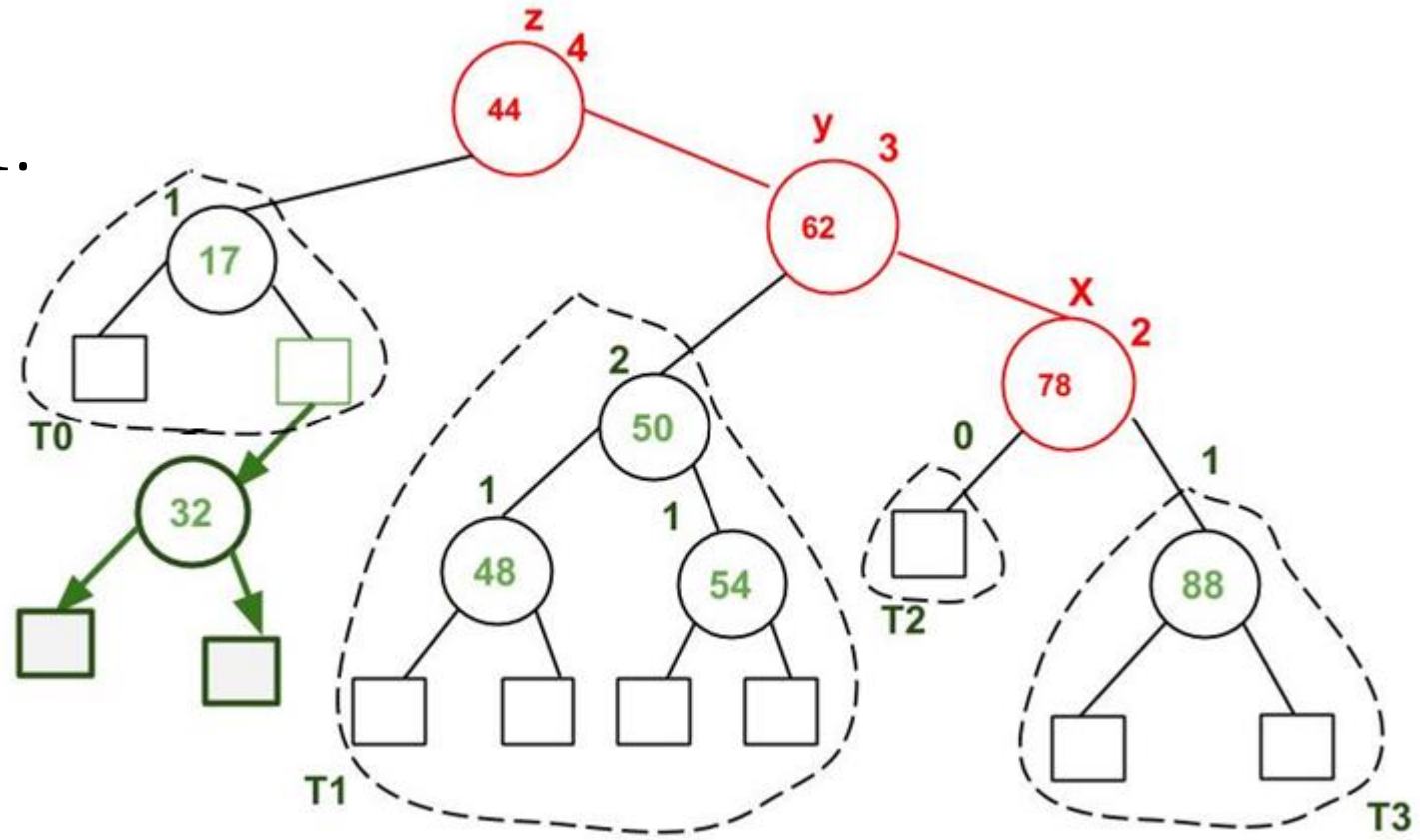
1. Ищем вершину D, которую требуется удалить.
2. Проверяем, сколько поддеревьев в D:
   - Если D – лист или D имеет одно поддерево, то удаляем D.
   - Если D имеет два поддерева, то ищем вершину M, следующую по значению после D. Как в стандартном алгоритме удаления из дерева поиска. Переносим значение 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. Удаление элемента.

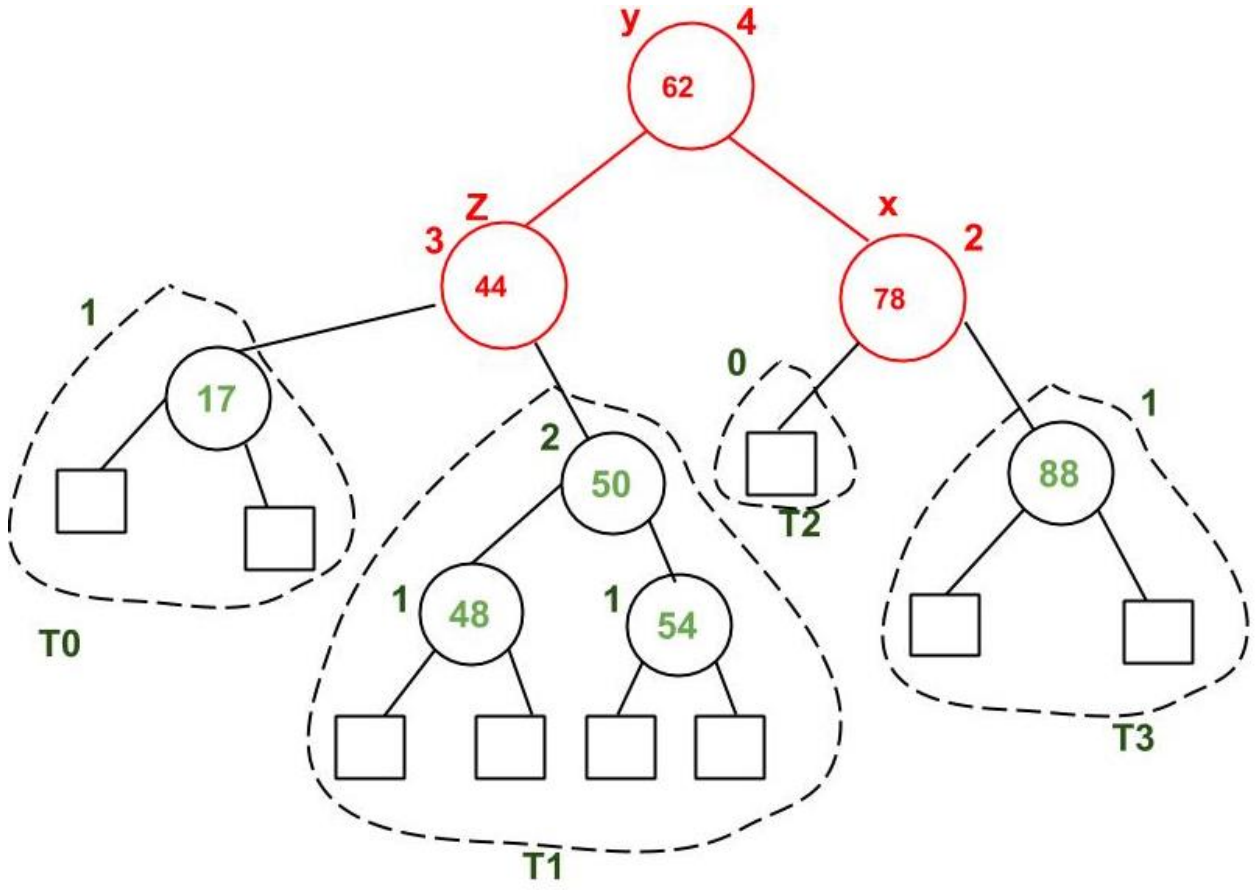
Удаляем элемент 32.

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



АВЛ-дерево. Удаление элемента.

Восстановили сбалансированность.



АВЛ-дерево

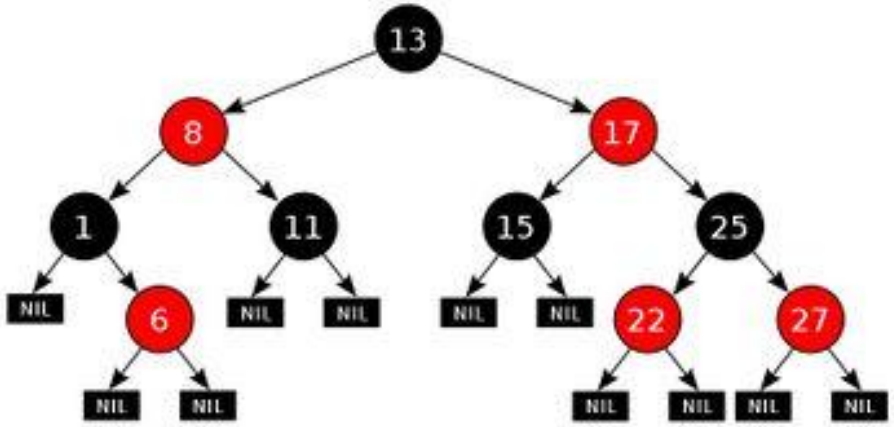
Расход памяти и время работы.

	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. Определение.

Красно-черное 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 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

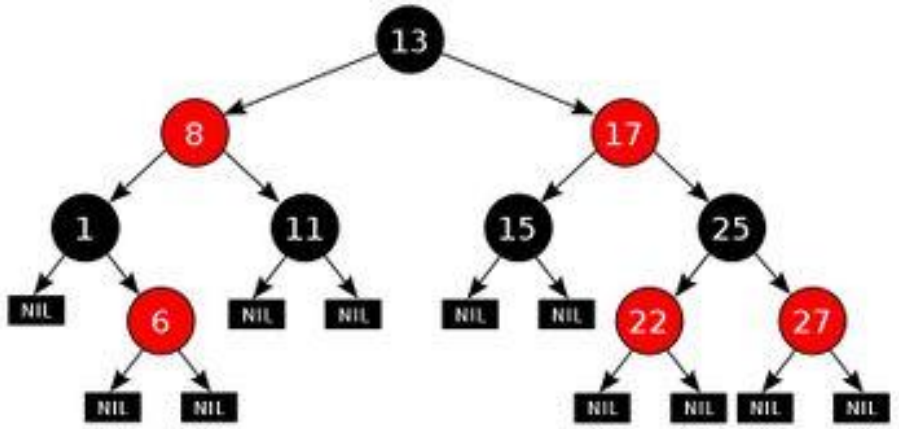


Придумал Рудольф Байер (1972г).

Красно-чёрные деревья. Фиктивные листья.

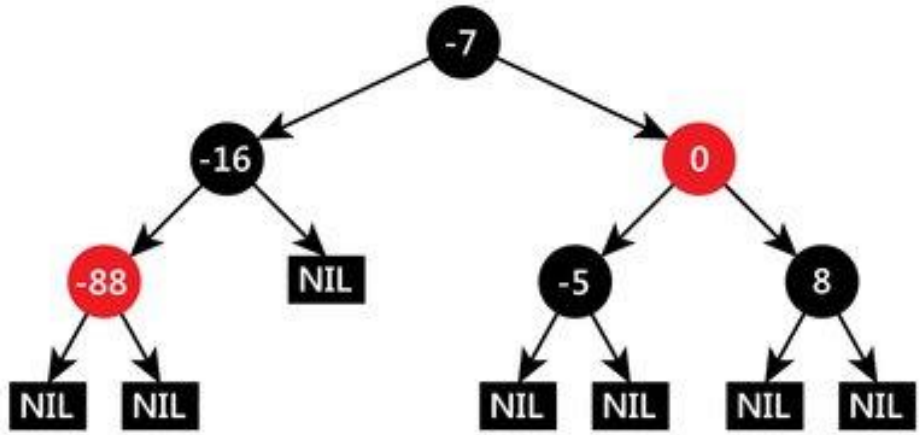
Все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными.

Для экономии памяти фиктивные листья делают одним общим фиктивным листом.



Красно-чёрные деревья. Чёрная высота.

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-чёрные деревья

<u>Лемма</u>. В красно-чёрном 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.
- $h(x) > 0$. Значит, x – внутренняя вершина, у неё 2 потомка. У каждого потомка чёрная высота либо $bh(x)$, либо $bh(x) - 1$, в зависимости от цвета потомка (красный и чёрный, 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 $h = O(logN)$.

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:

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

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

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

ч.т.д.

Красно-чёрные 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 потомками (фиктивными вершинами).
- Теперь восстанавливаем свойства красно-чёрного дерева.

Красно-чёрные 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-чёрные деревья. Вставка.

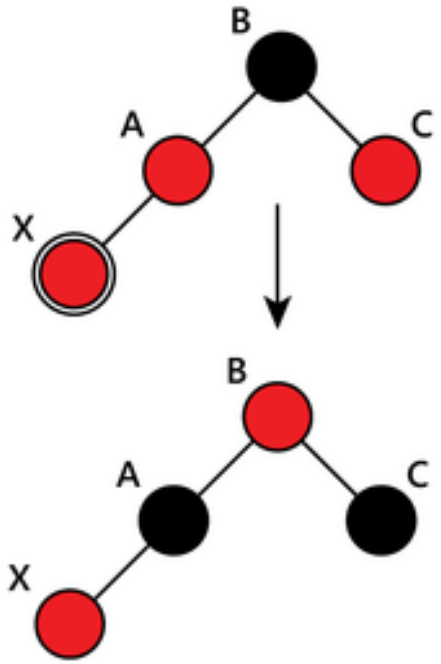
Ситуации после вставки:

- Если отец нового элемента чёрный, то ничего делать не надо.
- Если отец нового элемента красный, то возможны 3 случая (без учета симметрии):
  1. Отец красный, дядя красный.
  2. Отец красный, дядя чёрный, новый элемент левый потомок.
  3. Отец красный, дядя чёрный, 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 таки образом дойдём до корня, то в нём в любом случае ставим чёрный цвет.

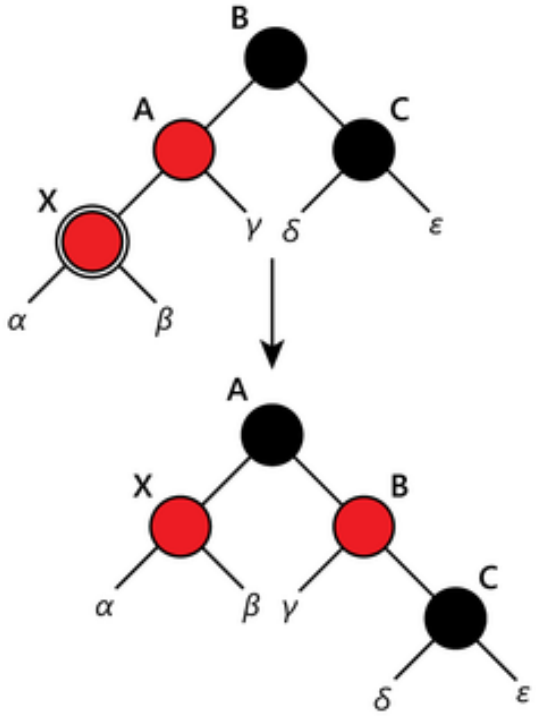


Красно-чёрные деревья. Вставка.

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

Просто выполнить перекрашивание отца в чёрный 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.

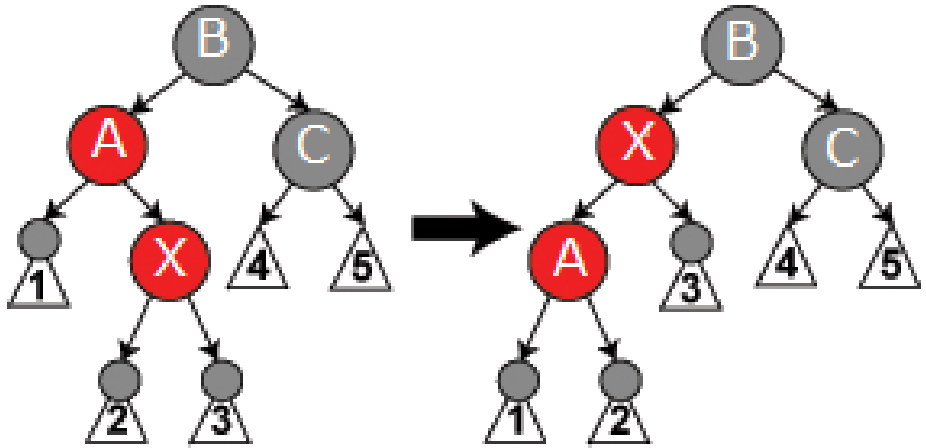


Красно-чёрные деревья. Вставка.

Случай 3. «Дядя» чёрный 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-чёрные деревья. Вставка.

Всего один шаг, если:

- Родитель черный, тогда вообще ничего делать не нужно
- Случай 1, когда отец деда черный или дед — корень.
- Случай 2
- Случай 3

Длинная цепочка действий возможна только при многократном повторении случая 1.

Сложность вставки $O(logN)$.

Красно-чёрные деревья. Удаление.

Как производится удаление:

1. Если у удаляемой вершины нет детей, у родителя перенаправляем указатель на фиктивный лист.
2. Если только один потомок, у родителя перенаправляем указатель на этого потомка.
3. Если потомка два, ищем в поддеревьях следующую или предыдущую вершину. Вместо исходной, удаляем именно эту вершину способом из п.1 или п.2, предварительно 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-чёрные деревья. Удаление.

А. 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 чёрным и нарушилось бы свойство постоянства чёрной глубины для потомка и его соседней фиктивной вершины.

Если потомков нет
Действия:

- Удалить красную вершину (заменить на лист)
- Конец

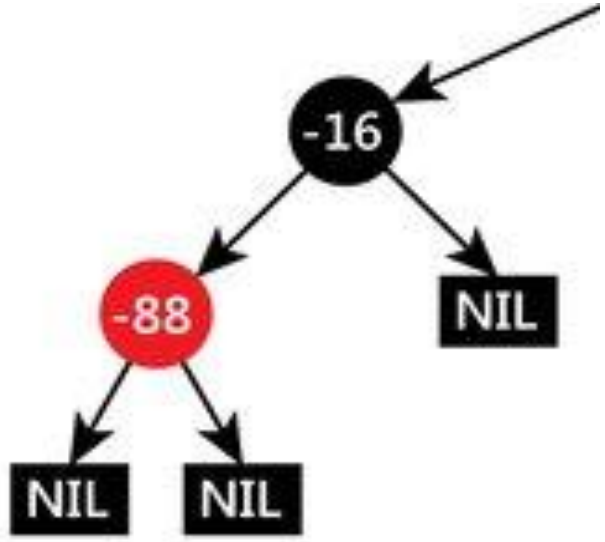
Красно-чёрные деревья. Удаление.

Б. Удаление черной вершины с единственным потомком.

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

<u>Действия:</u>

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



Красно-чёрные деревья. Удаление.

B. Удаление чёрной вершины без потомков. Это самый сложный случай.

Действия:

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

Путь в «x» имеет меньшее количество чёрных вершин (чёрную глубину), чем в другие вершины. Будем помнить об этом и называть «x» дважды чёрным.

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

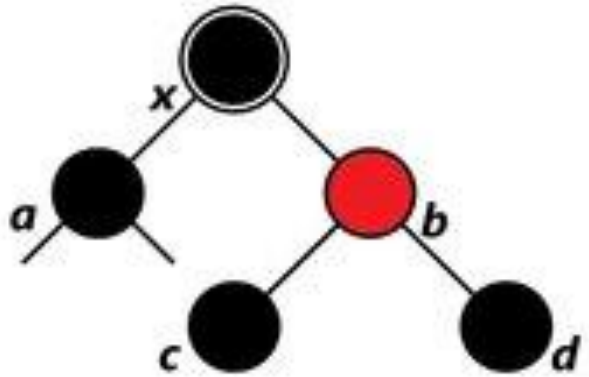
Красно-чёрные деревья. Удаление.

Восстановление свойств. Случай 0.

Если дважды чёрная вершина x – корень.

- Оставим корень просто чёрным (единожды чёрным)
- Конец

Так чёрная глубина всего дерева уменьшится на 1.



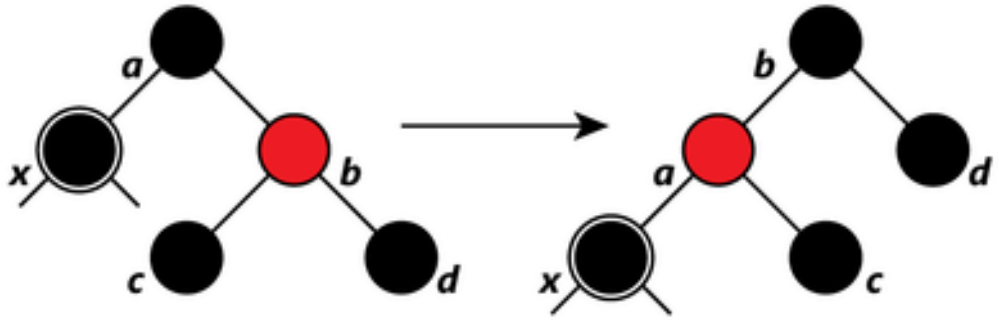
Красно-чёрные деревья. Удаление.

Восстановление свойств. Случай 1.

Если у дважды чёрной вершины $\boldsymbol{x}$ красный брат ($\boldsymbol{b}$).

- Делаем малый левый поворот в $\boldsymbol{a}$. Вершина $\boldsymbol{b}$ становится дедом $\boldsymbol{x}$.
- Красим $\boldsymbol{b}$ в чёрный, $\boldsymbol{a}$ – в красный цвет.

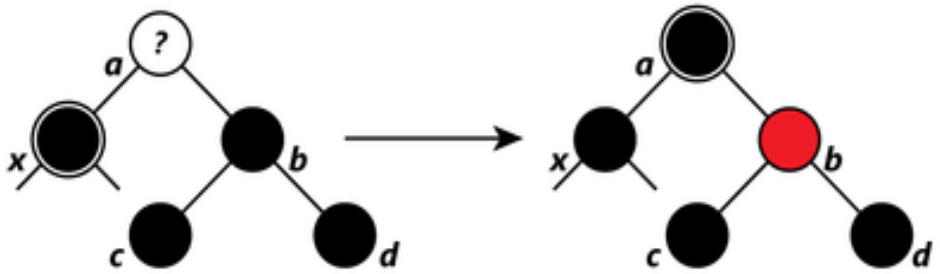
Теперь у $\boldsymbol{x}$ чёрный брат, переходим в случай 2, 3 или 4 в зависимости от цветов детей узла c.



Красно-чёрные деревья. Удаление.

Восстановление свойств. Случай 2.

Если у вершины x чёрный брат b, у которого оба дочерних узла c и d чёрные (c и d могут быть листьями). Красим b в красный цвет.

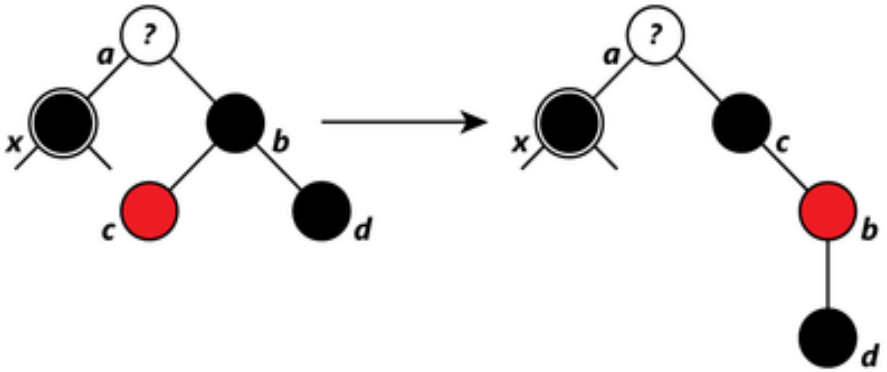


Какого цвета был отец (a)?	
КРАСНЫЙ	ЧЕРНЫЙ
• Красим a в чёрный цвет, так чёрная глубина a восстановится • Конец	• Считаем a дважды чёрной, в ней продолжим восстановление свойств. • Теперь может быть любой случай

Красно-чёрные деревья. Удаление.

Восстановление свойств. Случай 3.

Если у дважды чёрной вершины x чёрный брат b, левый ребенок брата c – красный, а правый d – чёрный.



- Делаем малое правое вращение в b
- Красим b в красный цвет
- Красим c в чёрный цвет

Так у брата правый ребенок станет красным.
После случая 3 всегда случай 4, затем конец.

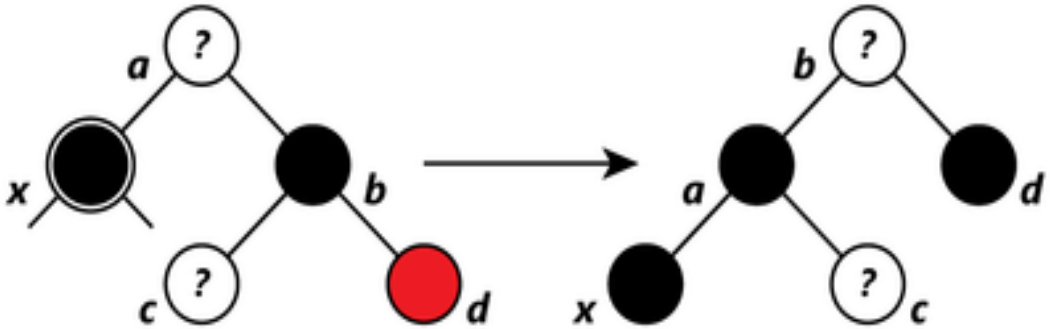
Красно-чёрные деревья. Удаление.

Восстановление свойств. Случай 4.

Если у дважды черной вершины x черный брат b, а правый ребенок брата – красный (d).

- Делаем малое левое вращение в 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-чёрные 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 чёрный)

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

Красно-чёрные деревья.

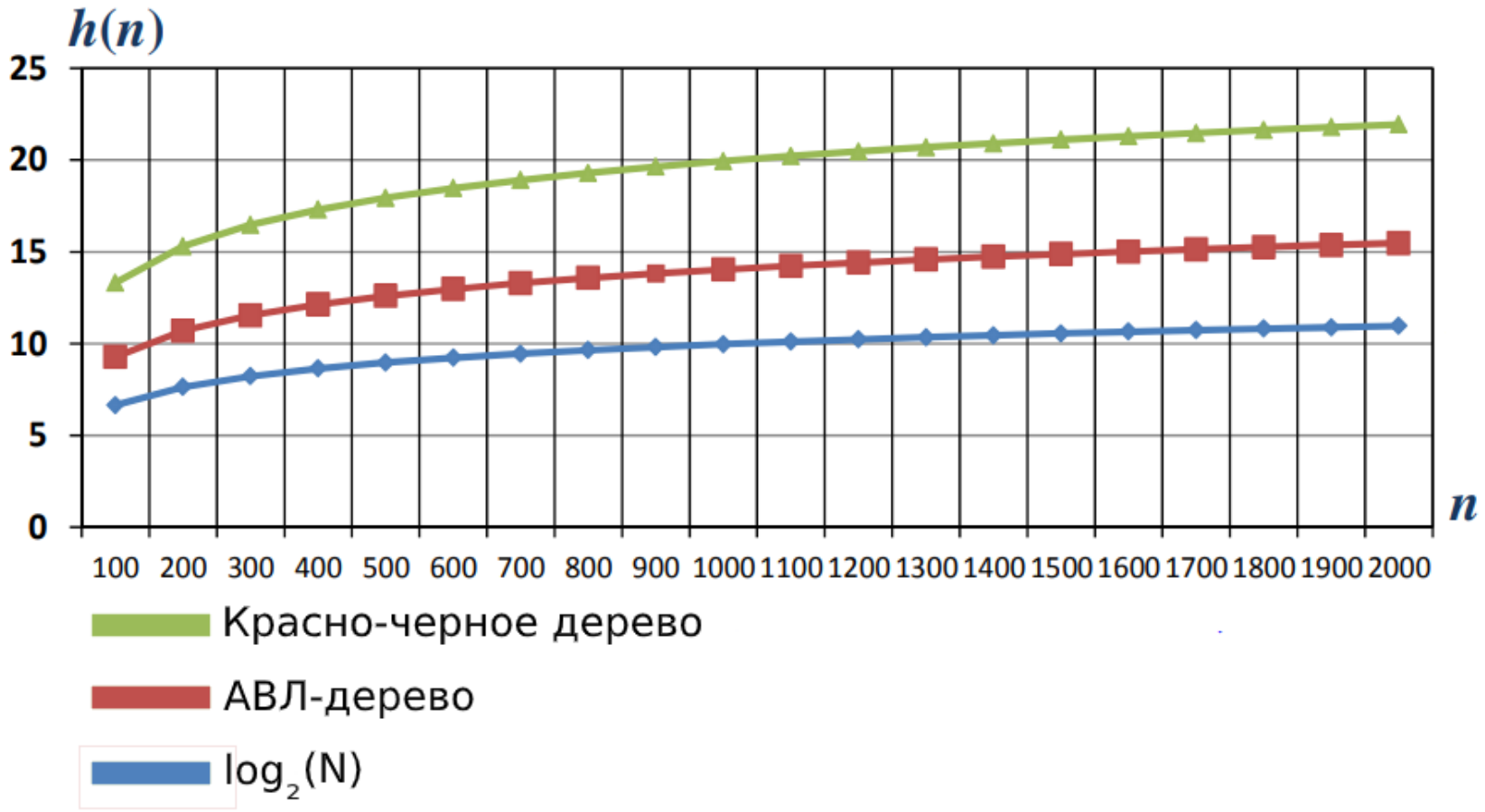
Вставка	Удаление
▪ Случай выбирается по цвету «дяди». ▪ Максимум 2 вращения	▪ Случай выбирается по цвету «брата» и его потомков. ▪ Максимум 3 вращения

Красно-чёрные 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 высот деревьев



Сравнение АВЛ и красно-чёрных деревьев

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

B-деревья

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

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

B-деревья. Определение.

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

1) Каждый узел X содержит:
   - $X.n$ – число ключей в узле
   - Сами ключи в отсортированном порядке:

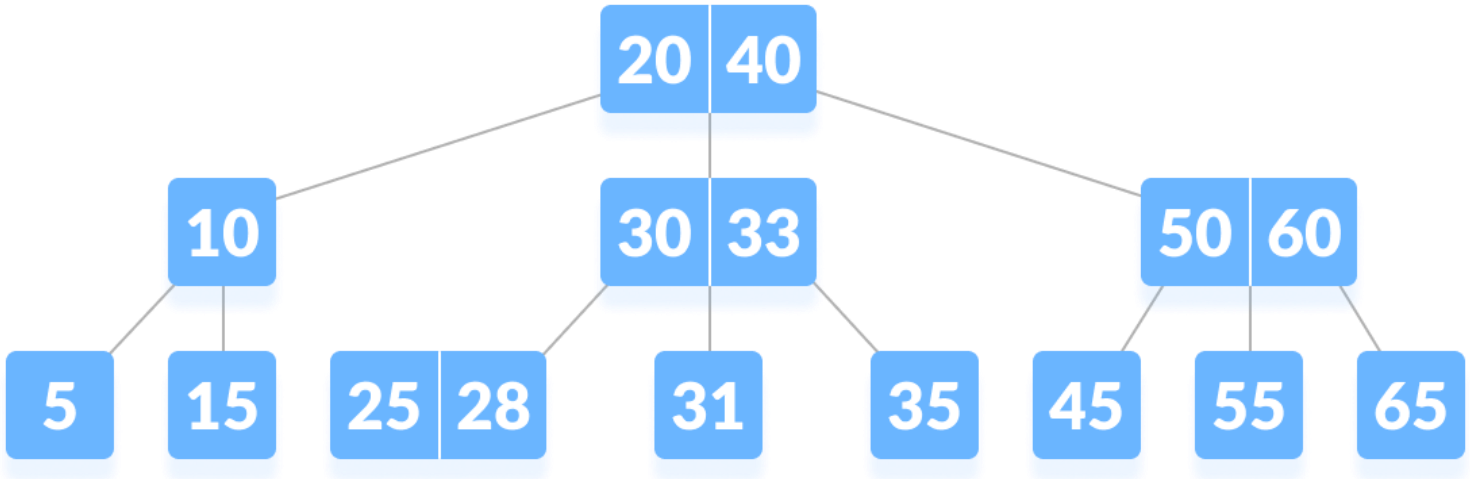
     $X.key_1 \leq X.key_2 \leq \ldots \leq X.key_n$
   - $X.leaf$ – булево значение “узел является листом”

2) Каждый внутренний узел также содержит:
   - $X.n + 1$ указателей $x.c_1, x.c_2, \ldots, x.c_{n+1}$ на потомков.

B-деревья. Определение.

3) Ключи $X.key_i$ внутри узла определяют, в какое поддерево переходим при поиске/вставке/удалении.

4) Пусть $\{k_i\}$ – все ключи из поддерева, на которое указывает $X.c_i$, тогда $\{k_1\} \le X.key_1 \le \{k_2\} \le X.key_2 \le \ldots \le X.key_{X.n} \le \{k_{X.n+1}\}$.



B-деревья. Определение.

5) Все листья B-дерева находятся на одной глубине

6) Минимальная степень B-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.

B-деревья. Высота дерева.

Теорема:
Если $N \geq 1$, то для любого B-дерева, содержащего N ключей и минимальной степенью $t \geq 2$, верно утверждение $h \leq \log_t\left(\frac{n+1}{2}\right)$.

Доказательство:
Корень B-дерева содержит по крайней мере 1 ключ, прочие узлы – по крайней мере $t - 1$ ключ. Следовательно, дерево T 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)

B-деревья. Высота дерева.

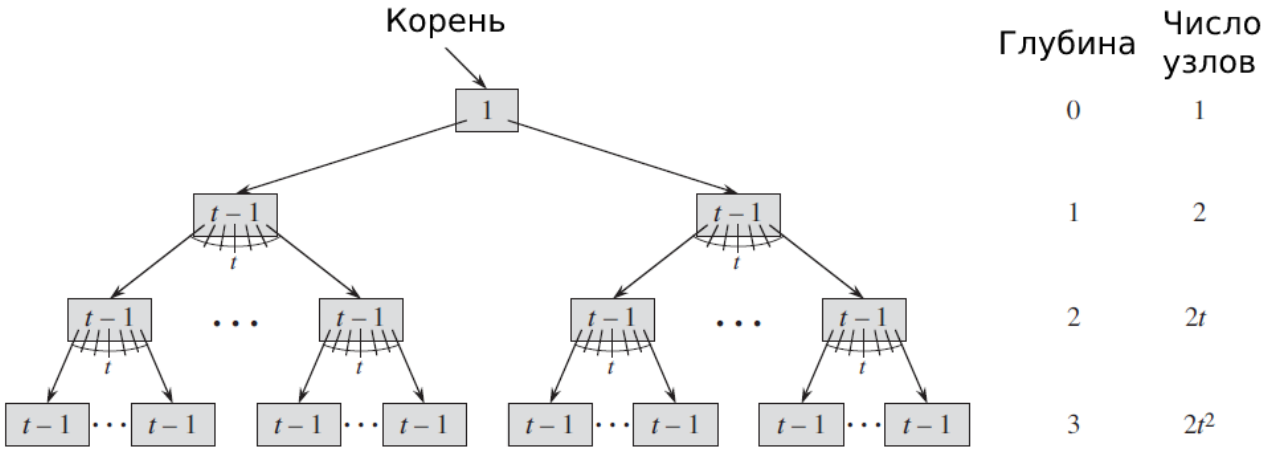
Доказательство: (продолжение)

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

Что и требовалось доказать.



B-деревья. Устройство жесткого диска.

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

B-деревья. 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 $\sim 8 - 11$ мс (в $160{,}000 - 220{,}000$ раз медленнее). Все из-за механики – нужно прокрутить дисковые пластины и переместить считывающую головку.

B-деревья. 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 траты времени приходятся на механическое позиционирование пластин и считывающей головки на начало первой страницы. Само чтение/запись происходят быстро – там одна электроника, без механики (без учета постоянного вращения пластины).

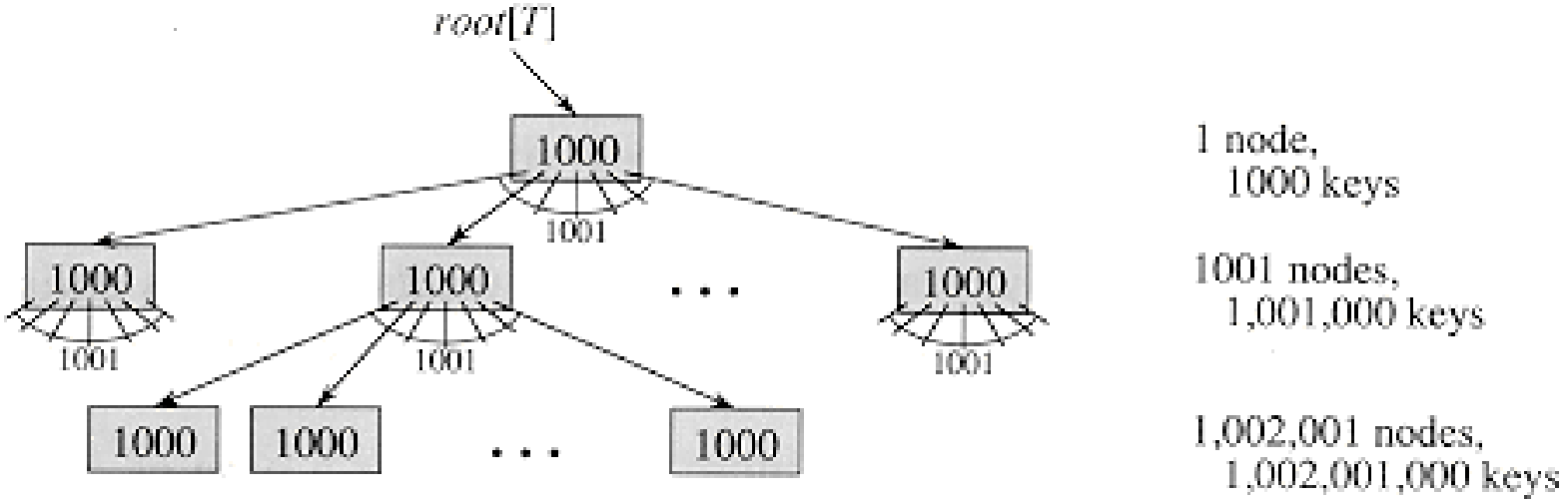
B-деревья. Подстраиваемся под HDD.

- Предполагается, что хранимые данные не помещаются в оперативную память.
- B-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 B-дерева подбирается под размер страницы жесткого диска. Это накладывает ограничения на число хранимых в узле ключей и, соответственно, число потомков.

B-деревья. Подстраиваемся под HDD.

- В больших B-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.



B-деревья. Операции с B-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.

B-деревья. Создание B-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.

B-деревья. Поиск ключа.

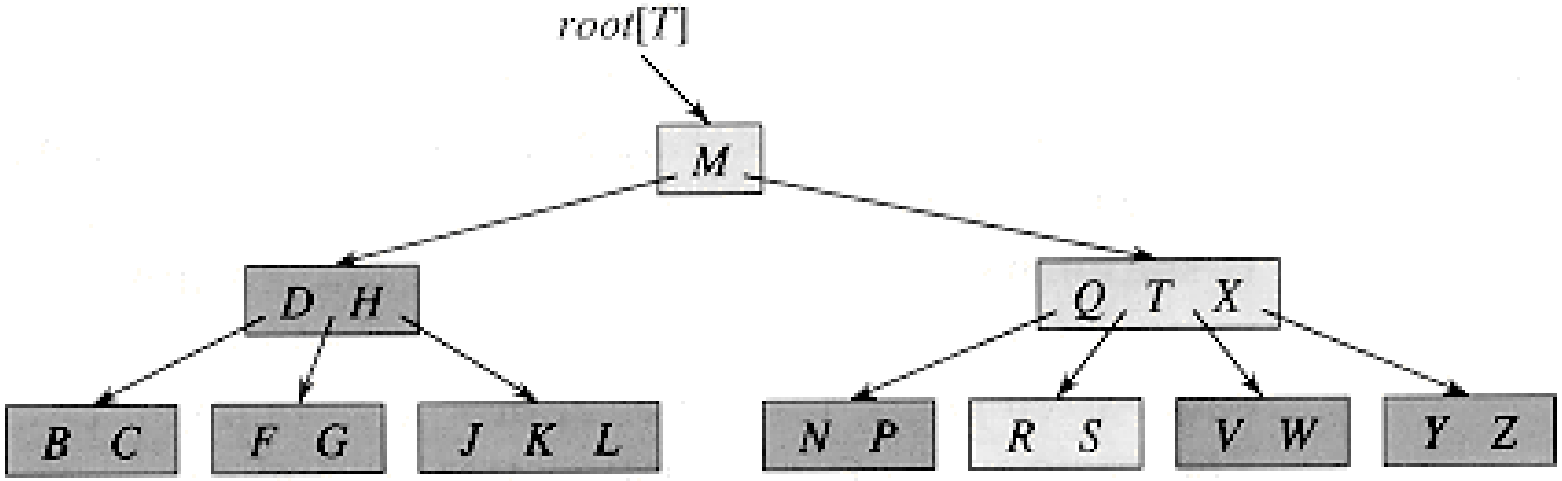
B-TREE-SEARCH(X, k) – поиск ключа k в B-дереве с корнем X. Если k в дереве, то вернет (Y, i) (где Y – узел, i – индекс ключа в Y, такой что $Y.key_i == k$), иначе NIL.

```
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)
```

B-деревья. Поиск ключа.

Сложность поиска: $O(t\ log_t N)$.

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

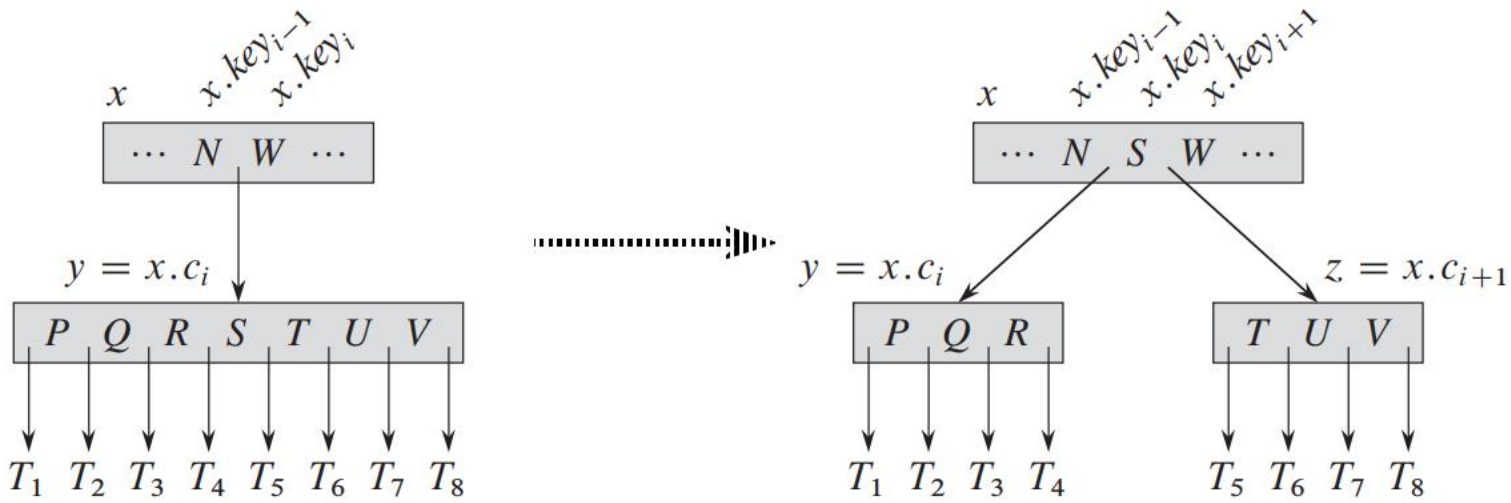


B-деревья. Вставка ключа.

- Вставка сложнее, чем в бинарном дереве поиска: мы не можем создать новый лист в дереве – результат перестанет быть B-деревом.
- Будем добавлять ключ в какой-то существующий лист.
- Нельзя добавлять ключ в полный узел (хранящий $2t - 1$ 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.

B-деревья. Вставка ключа. B-TREE-SPLIT-CHILD.

Пример применения B-TREE-SPLIT-CHILD: B-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.



B-деревья. 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 заблаговременно 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.

B-деревья. Вставка ключа. B-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)
  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.key_j = Y.key_j+t
  if not Y.leaf // если у Y есть потомки
      for j = 1 to t // половину потомков Y
          Z.c_j = Y.c_j+t // переносим в Z
   Y.n = t – 1 // поправим счетчик ключей в Y
  for j = X.n + 1 downto i + 1
      X.c_j+1 = x.c_j // сдвиг указателей на 1 вправо
  x.c_i+1 = z // чтобы добавить указатель на Z
  // продолжение на следующем слайде
```

B-деревья. Вставка ключа. B-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.key_{j+1} = X.key_j // на 1 для вставки медианы
  X.key_i = Y.key_t // 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 $O(t)$.

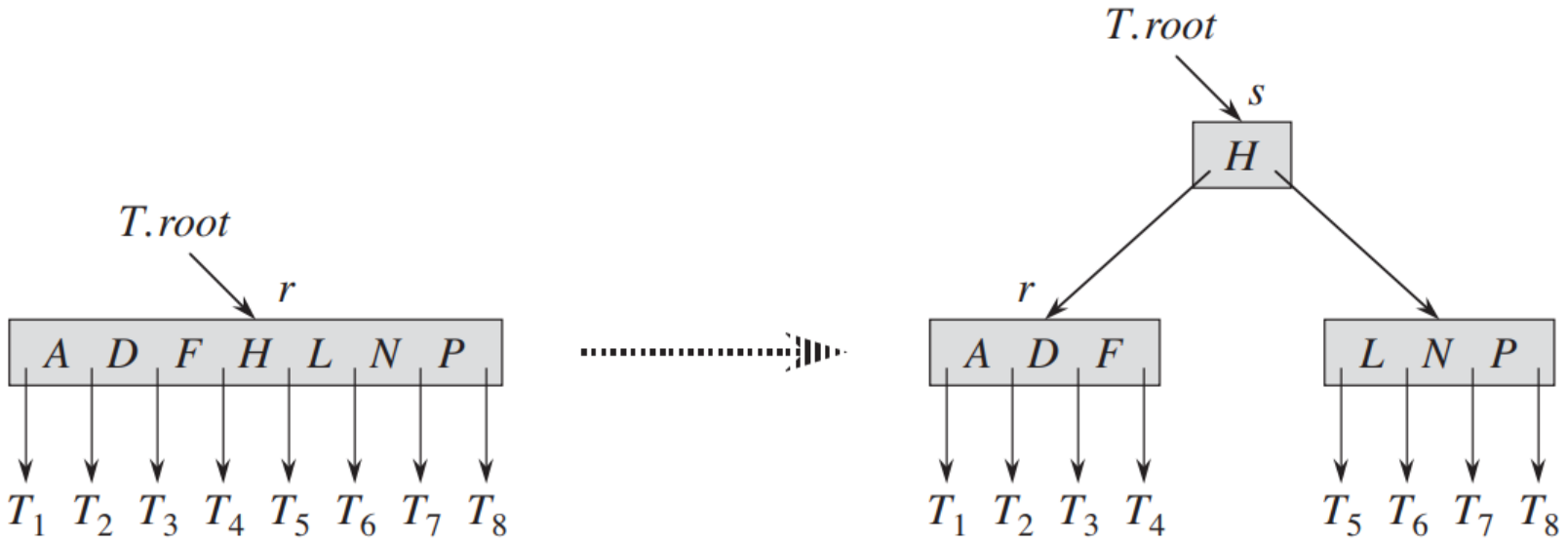
B-деревья. Вставка ключа. B-TREE-SPLIT-CHILD.

Как разбить полный корневой узел:

- Создаем новый пустой корневой узел и цепляем к нему потомком старый заполненный корневой узел.
- Вызываем B-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 B-дерева.

B-деревья. Вставка ключа. B-TREE-SPLIT-CHILD.

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



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

B-деревья. Вставка ключа.

B-TREE-INSERT(T, k) добавляет ключ k в B-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)
```

B-деревья. Вставка ключа.

B-TREE-INSERT-NONFULL(X, k) добавляет ключ k в <u>неполный</u> (обязательно!) узел X.

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

B-деревья. Вставка ключа.

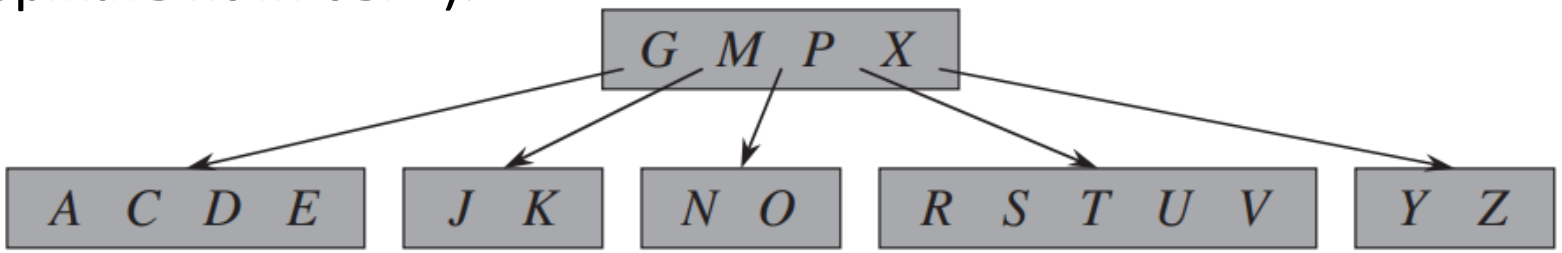
B-TREE-INSERT-NONFULL(X, k) добавляет ключ k в неполный (обязательно!) узел X.

```
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 разобьем 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)
```

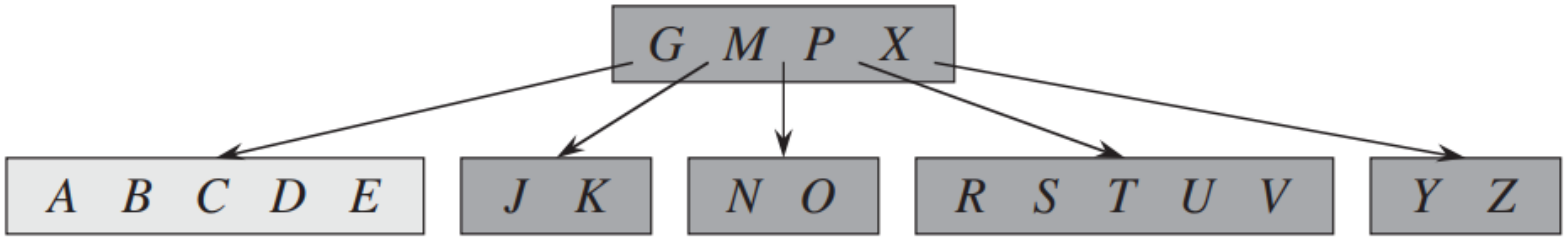
B-деревья. Вставка ключа. Примеры ситуаций.

Примеры ситуаций при вставке.

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

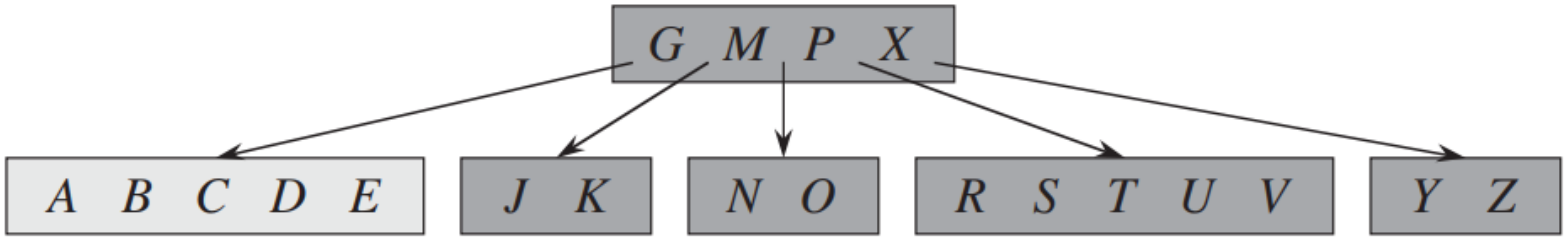


Добавляем ключ B, он просто записывается в лист.

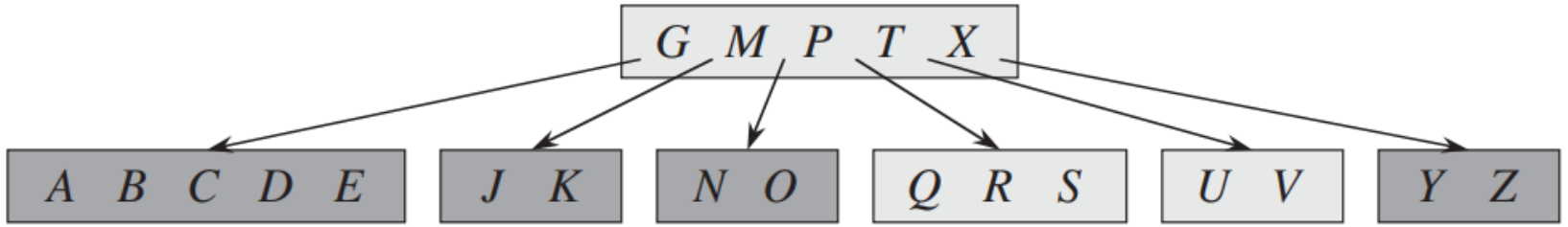


B-деревья. Вставка ключа. Примеры ситуаций.

Примеры ситуаций при вставке.

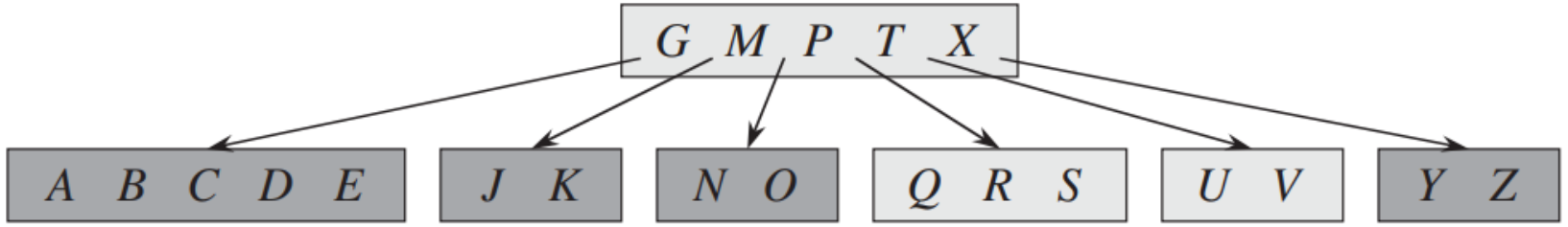


Добавляем ключ Q. Он должен попасть в лист RSTUV, но тот полон. Разбиваем его на RS и UV, T переносим в родительский узел, Q добавляем в левого потомка (RS).

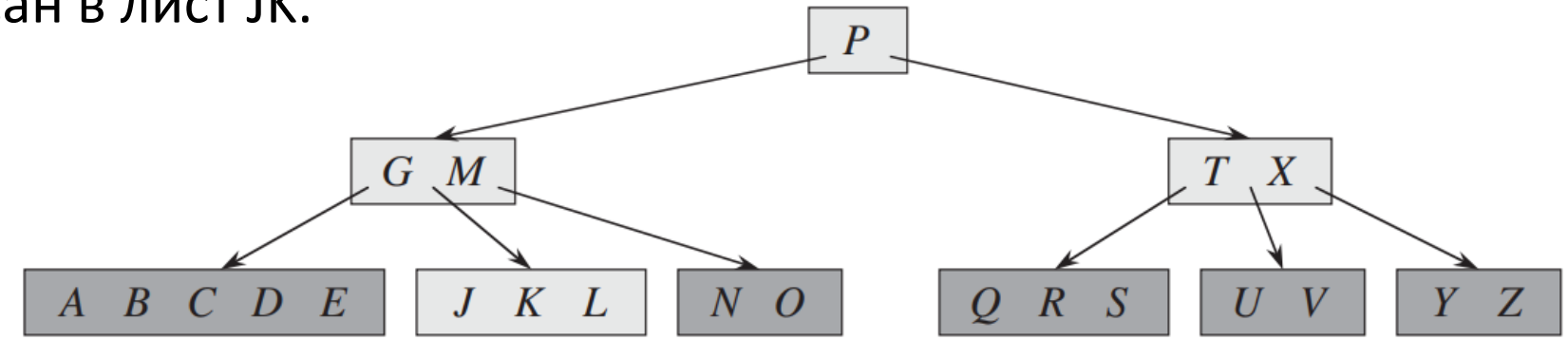


B-деревья. Вставка ключа. Примеры ситуаций.

Примеры ситуаций при вставке.

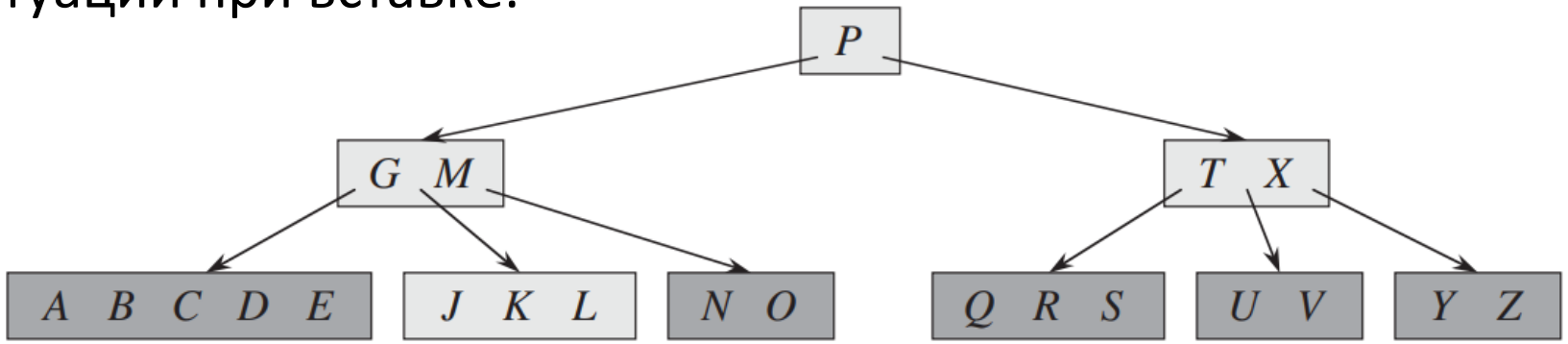


Добавляем ключ L. По пути обнаруживаем, что корень полон, надо разбивать. Получаем новый корень P с потомками GM и TX. Высота дерева увеличилась на 1. L будет записан в лист JK.

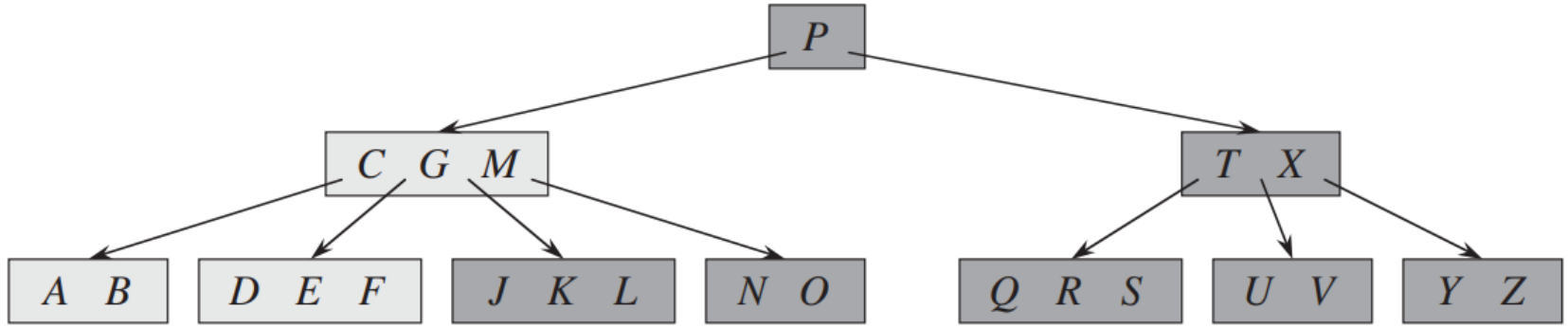


B-деревья. Вставка ключа. Примеры ситуаций.

Примеры ситуаций при вставке.



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



B-деревья. 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 $O(t * h) = O(t * log_t N)$.

B-деревья. 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-деревья. 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).

B-деревья. Удаление ключа. Случаи.

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-деревья. Удаление ключа. Случаи.

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 $X.ci$ поддерева, в котором продолжим поиски k. Если в $X.ci$ только $t - 1$ ключ, выполним шаг 3a или 3b, чтобы в $X.ci$ стало t ключей. Делаем B-TREE-DELETE($X.ci, k$).

B-деревья. Удаление ключа. Случаи.

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

   а) Если в $X.ci$ только $t - 1$ ключ, но у ближайшего соседа справа или слева по крайней мере t ключей.
   Возьмем из X ключ «между $X.ci$ и соседом $X.ci$» и спустим его в $X.ci$, поднимем ключ из соседа $X.ci$ в X, на место перенесенного. Также перенесем соответствующий указатель на потомка из соседа $X.ci$ в $X.ci$. Не забудем поправить указатель на потомка в $X.ci$.

B-деревья. Удаление ключа. Случаи.

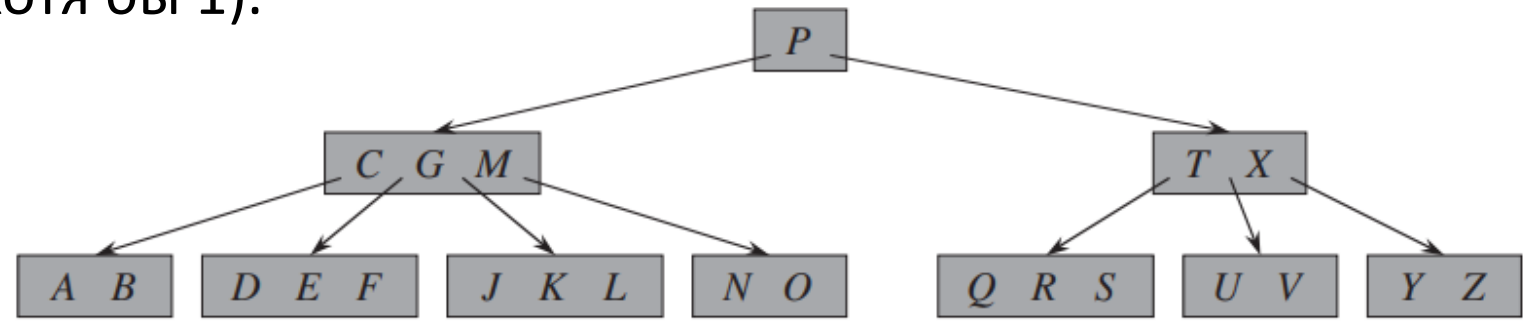
3. Ключ k не нашелся во внутреннем узле X (продолжение).

   b) Если и в $X.ci$ 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 $X.ci$, и сольем их в новый узел. В X теперь на 1 ключ и на 1 ссылку меньше.

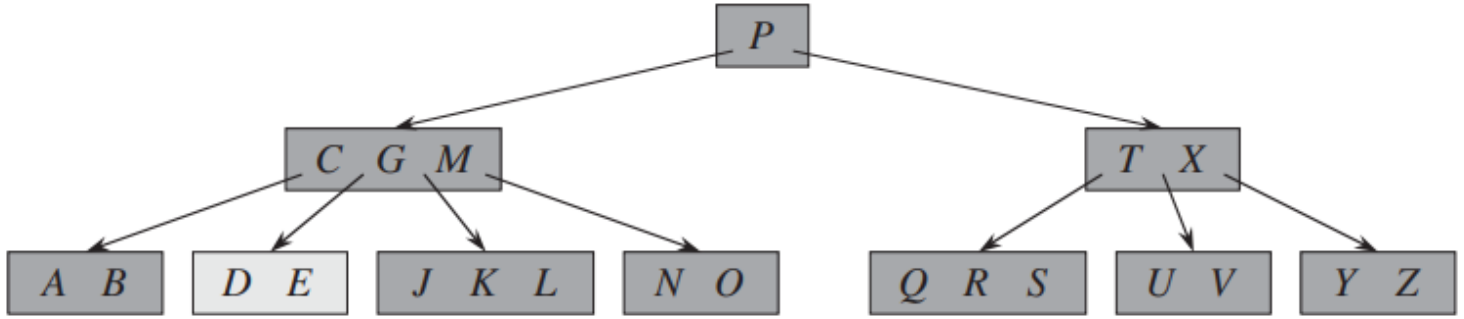
B-деревья. Удаление ключа. Примеры ситуаций.

Примеры ситуаций при удалении.

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

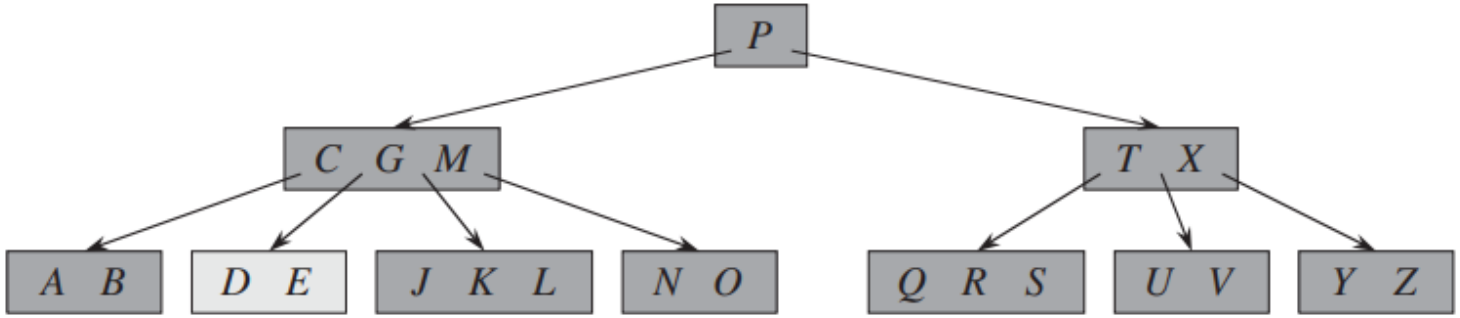


Удаляем ключ F. Тут простое удаление из листа. Случай 1.

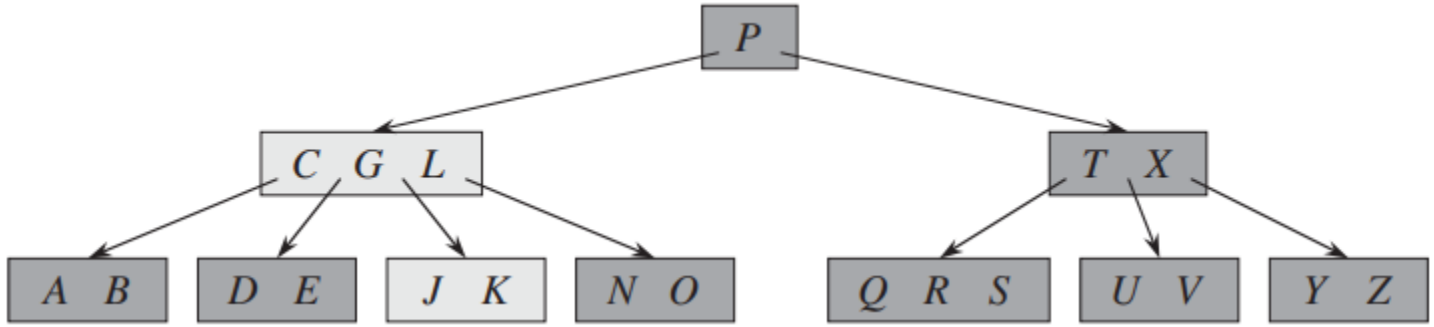


B-деревья. Удаление ключа. Примеры ситуаций.

Примеры ситуаций при удалении.

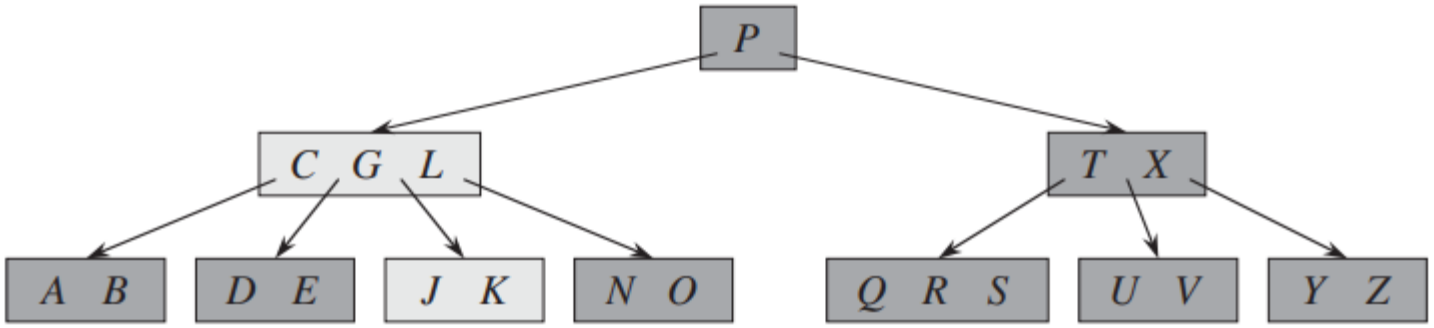


Удаляем ключ M. Случай 2a. Вместо M подставим предшественника из левого потомка – L.

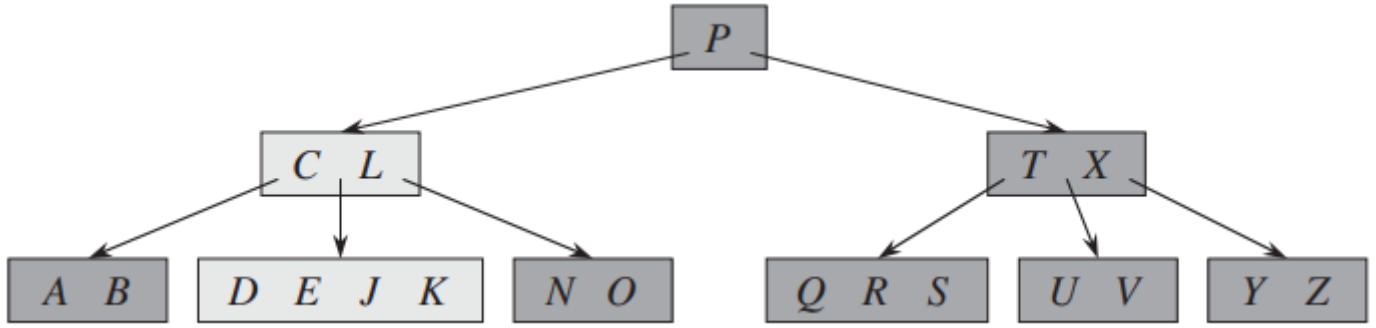


B-деревья. Удаление ключа. Примеры ситуаций.

Примеры ситуаций при удалении.

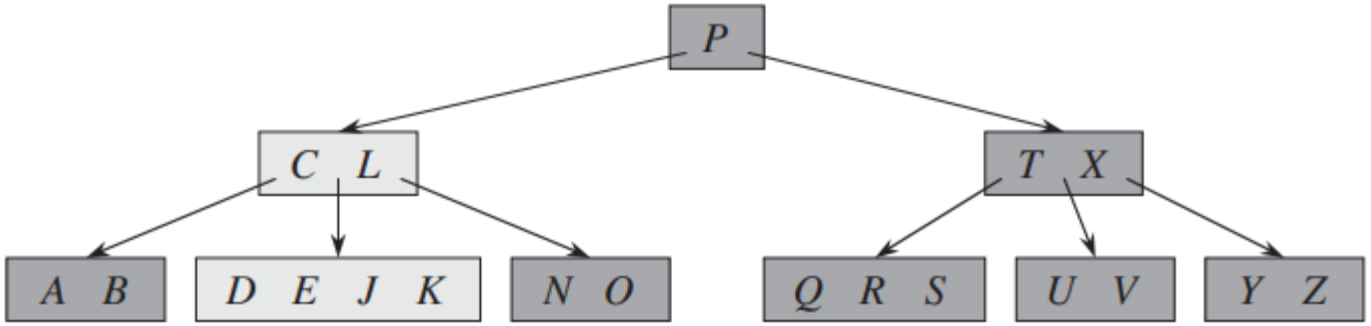


Удаляем ключ G. Случай 2c. Объединяем ключ G и узлы DE и JK в один узел DEGJK. Из CGL убрали 1 ключ и 1 указатель. Рекурсивно удаляем G из нового узла (1-й случай).

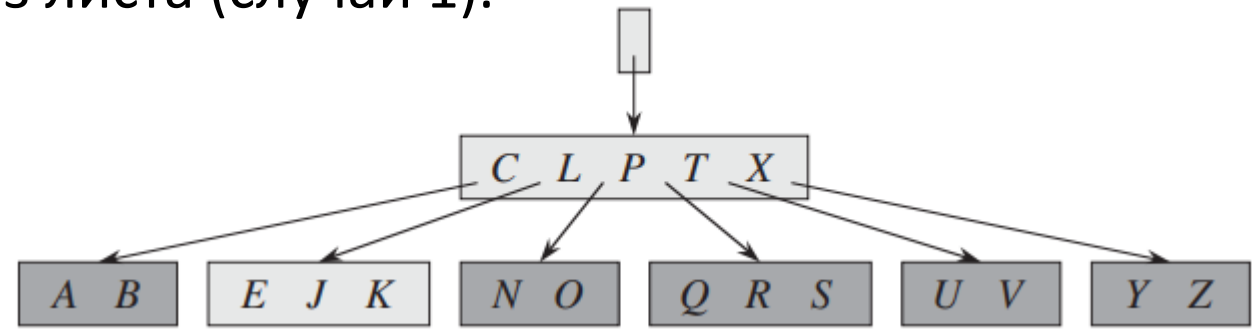


B-деревья. Удаление ключа. Примеры ситуаций.

Примеры ситуаций при удалении.

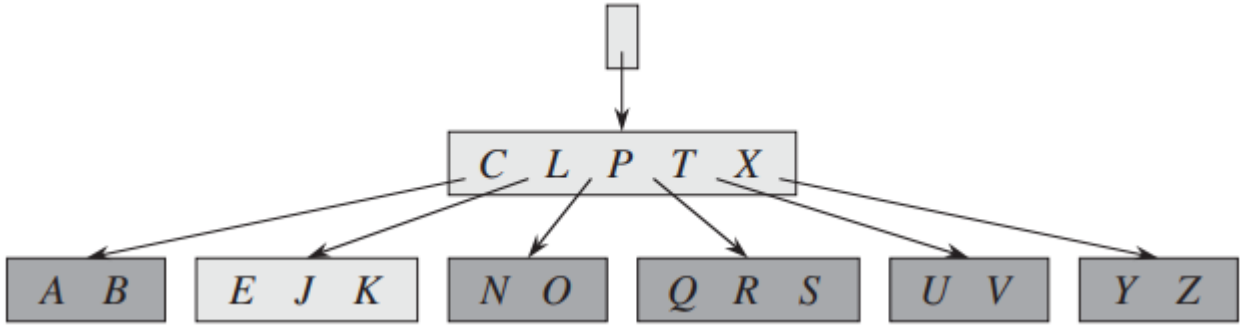


Удаляем ключ D. Случай 3b. Мы не можем просто спуститься в CL – он имеет минимальное допустимое количество ключей. Сливаем CL, P и TX в новый узел. Затем удаляем D из листа (случай 1).

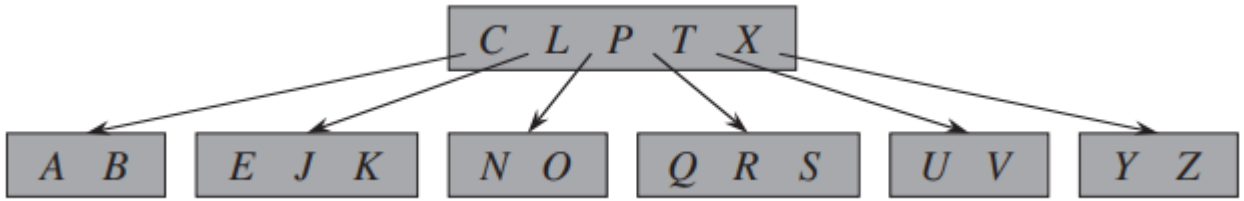


B-деревья. Удаление ключа. Примеры ситуаций.

Примеры ситуаций при удалении.

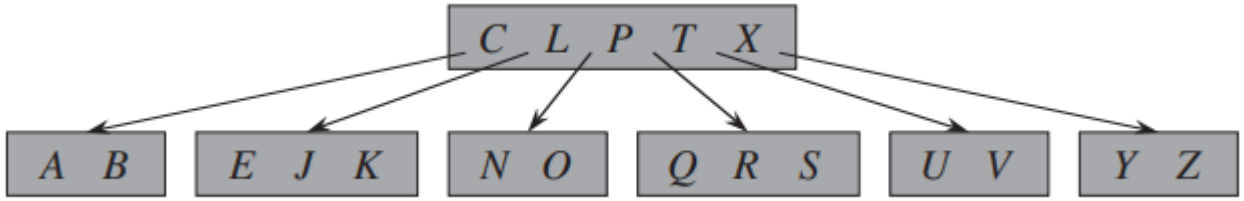


Завершение случая с прошлого слайда. Удаляем пустой корень, теперь корнем стал узел CLPTX. Высота дерева уменьшилась на 1.

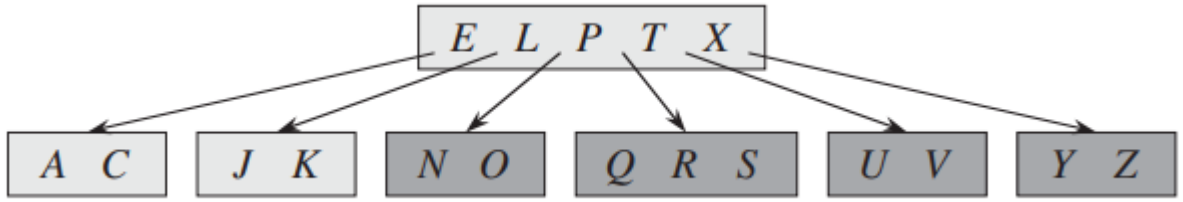


B-деревья. Удаление ключа. Примеры ситуаций.

Примеры ситуаций при удалении.



Удаляем B. Случай 3a. C переносим на место B, E переносим на место C.



B-деревья. Удаление ключа. Сложность.

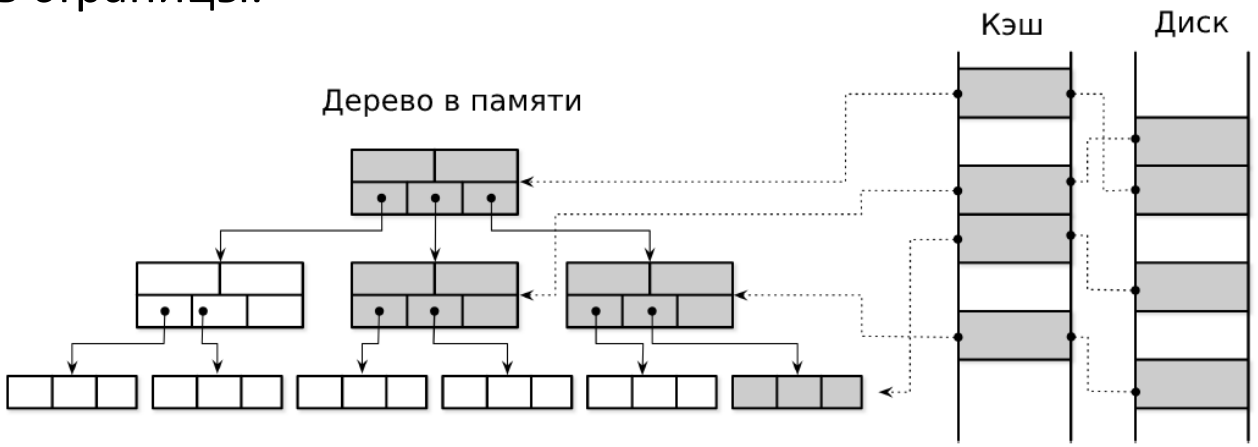
В рамках вызовов B-TREE-DELETE производится $O(h)$ вызовов DISK-READ и DISK-WRITE, значит происходит $O(h) = O(log_t N)$ дисковых операций.

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

B-деревья. Дисковый кэш.

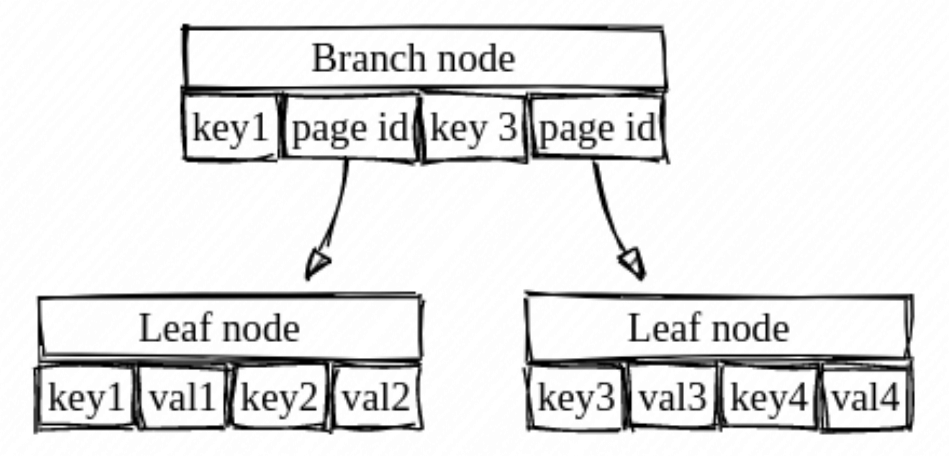
Назначение дискового кэша:

- 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^+-деревья

B^+-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^+-деревья также хранят в листьях ссылки на соседние листья.



B^+-деревья. 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^+-деревья. Где используются.

- В файловых системах. Например, NTFS, APFS, ext4 (использует extent trees -- модифицированные B^+-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.

АТД «Ассоциативный массив»

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.

АТД «Ассоциативный массив»

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

АТД «Ассоциативный массив»

Ассоциативные массивы в STL:

- std::map и std::multimap реализованы на красно-чёрных деревьях
- std::unordered_map и std::unordered_multimap на хеш-таблицах

Визуализаторы 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!

Дмитрий Глушенков