МГТУ
им. Н. Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Лекция 6.
Деревья

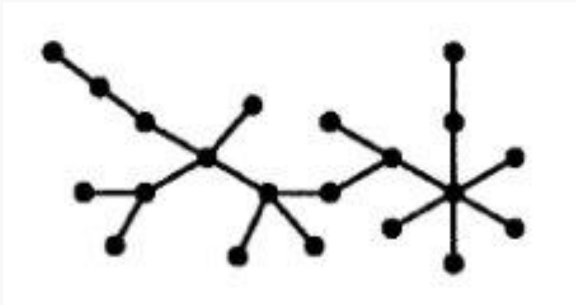
Алгоритмы и
структуры данных

Мацкевич С. Е.

План лекции 6 «Деревья»

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

Определения деревьев



<u>Определение 1.</u> 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.

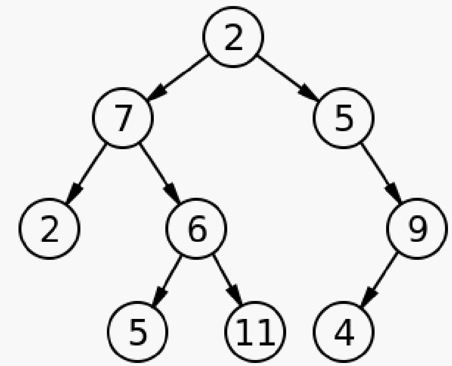
Путь в дереве – список отдельных вершин, в котором следующие друг за другом вершины соединяются ребрами дерева.

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

<u>Определение 2</u> (равносильно первому). Дерево (свободное) – неориентированный связный граф без циклов.

Определения деревьев

<u>Определение 3.</u> Дерево с корнем — дерево, в котором один узел выделен и назначен «корнем» дерева.



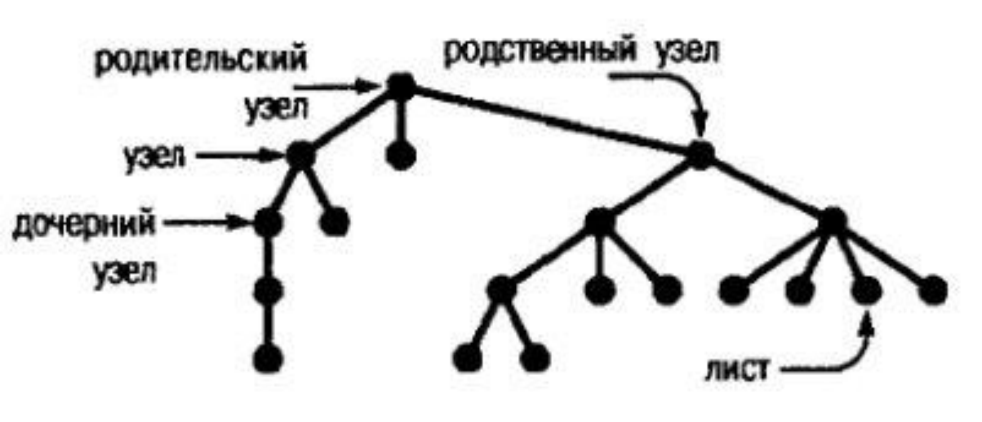
Существует только один путь между корнем и каждым из других узлов дерева.

<u>Определение 4.</u> 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.

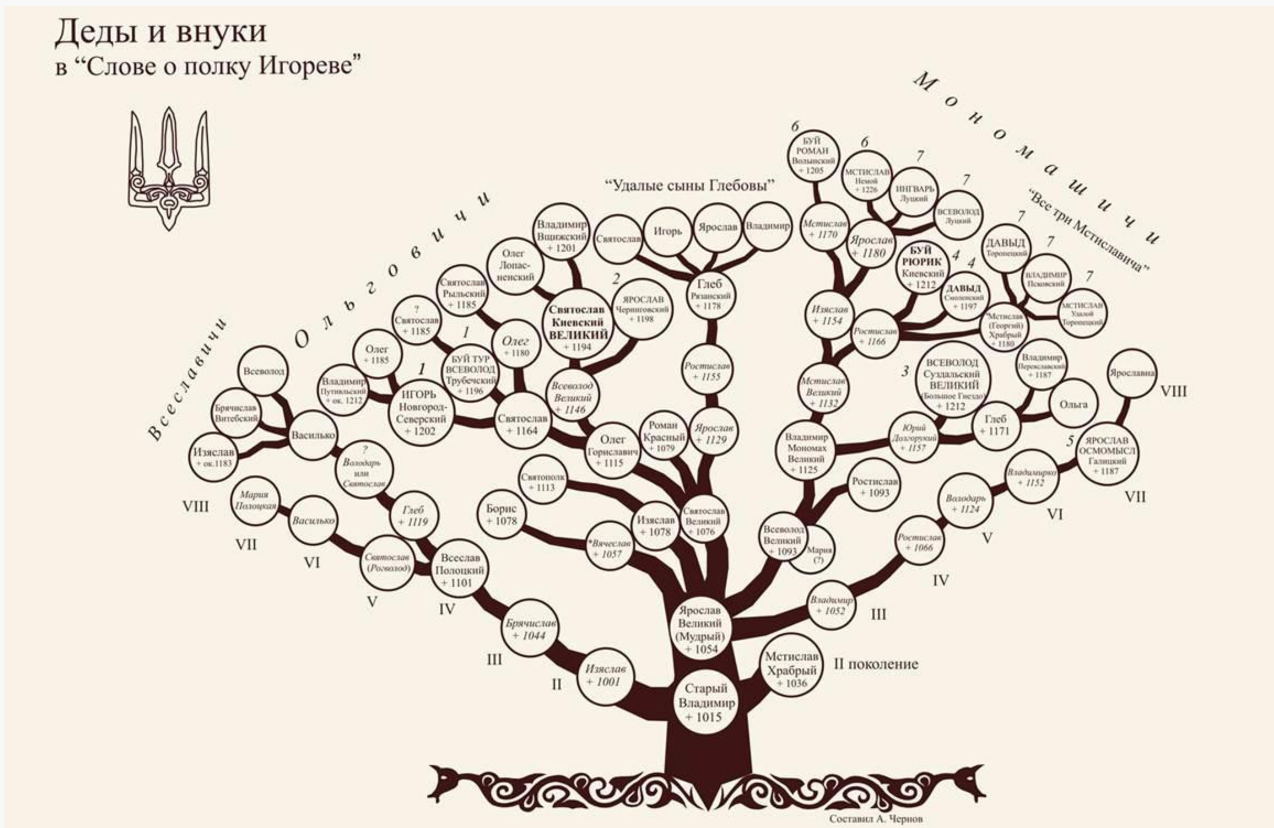
Определения деревьев

<u>**Определение 5.**</u> 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

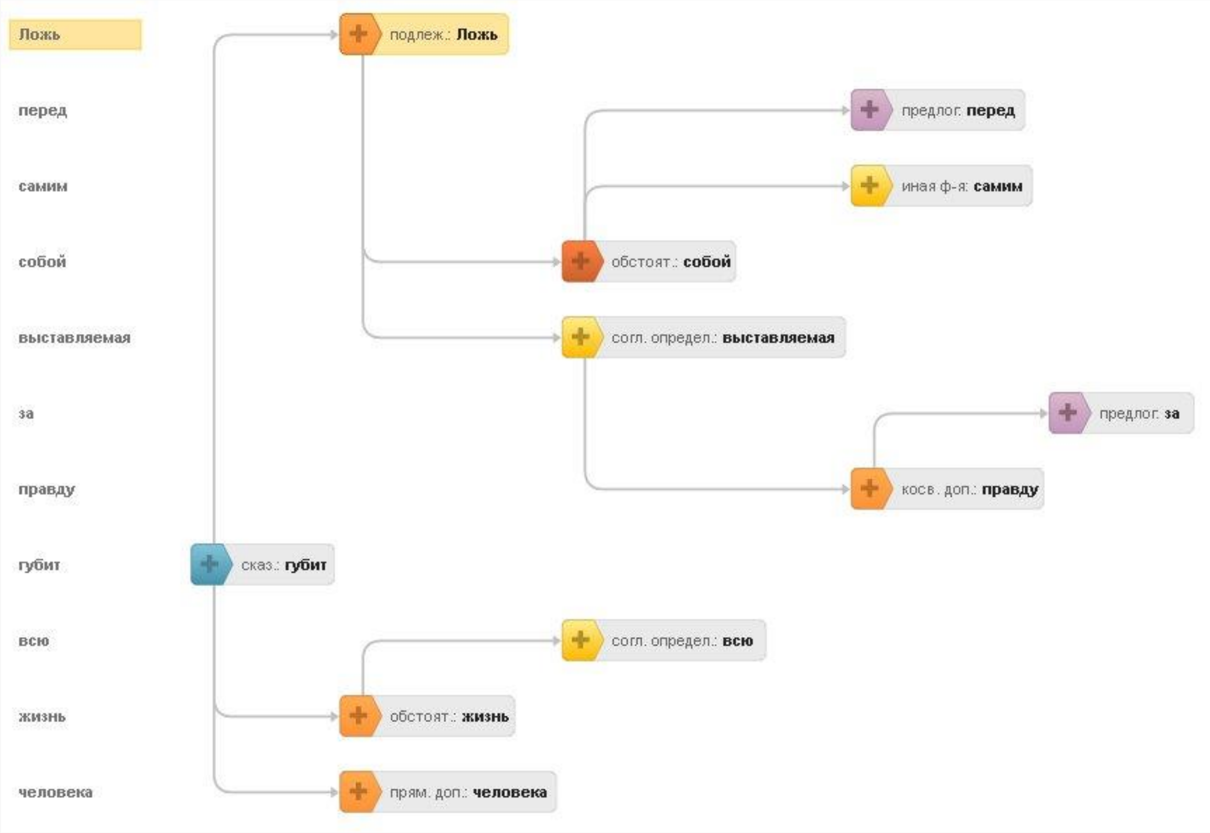
Организация турнира.

Примеры деревьев

Орг. структура компании

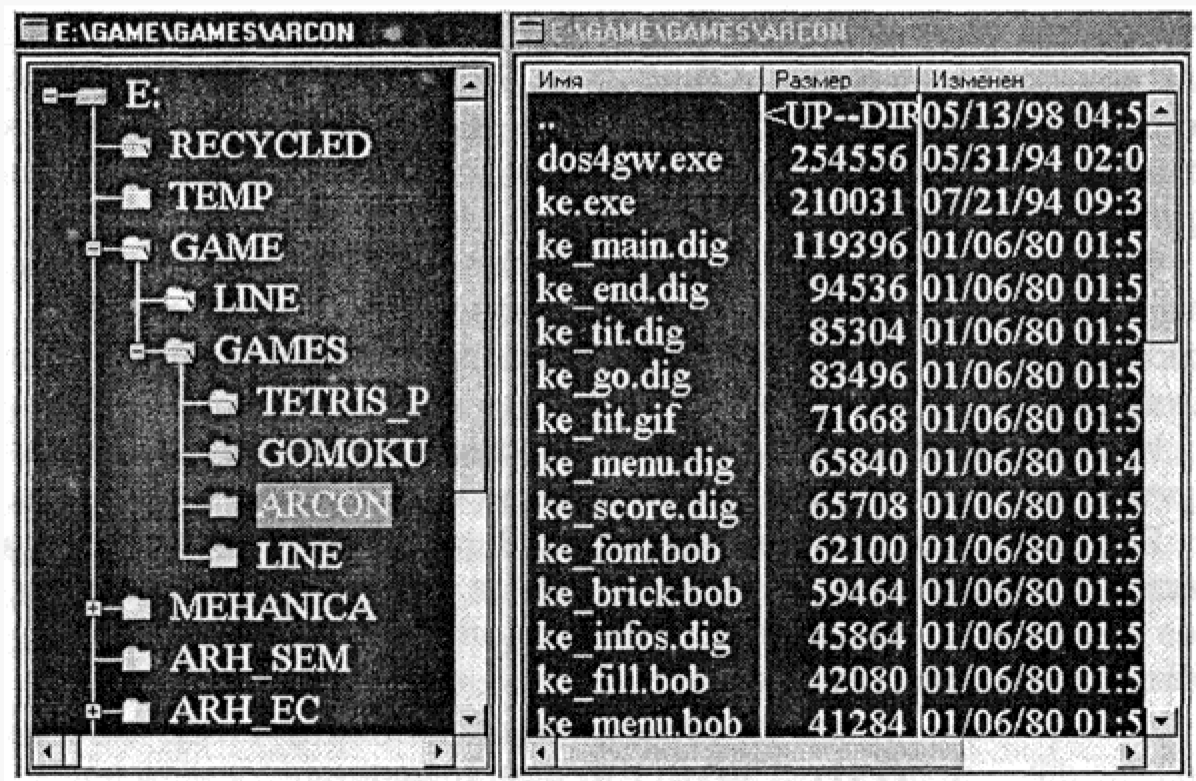
Примеры деревьев

Синтаксический или семантический разбор предложения.



Примеры деревьев

Файловая система



Число вершин и ребер

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

Доказательство. Самая глубокая вершина является листовой.

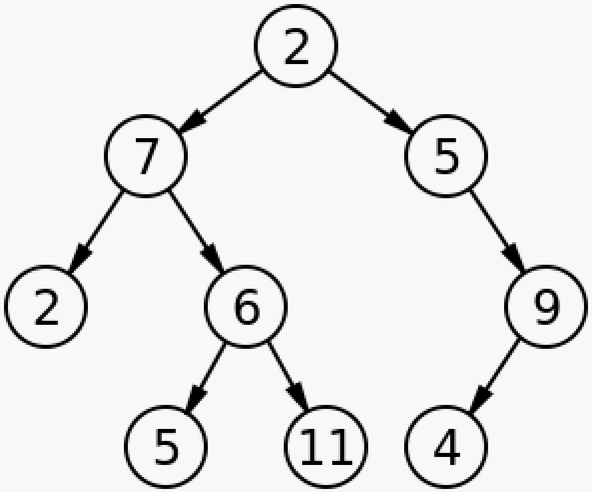
Утверждение 2. Дерево, состоящее из N вершин, содержит N – 1 ребро.

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

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

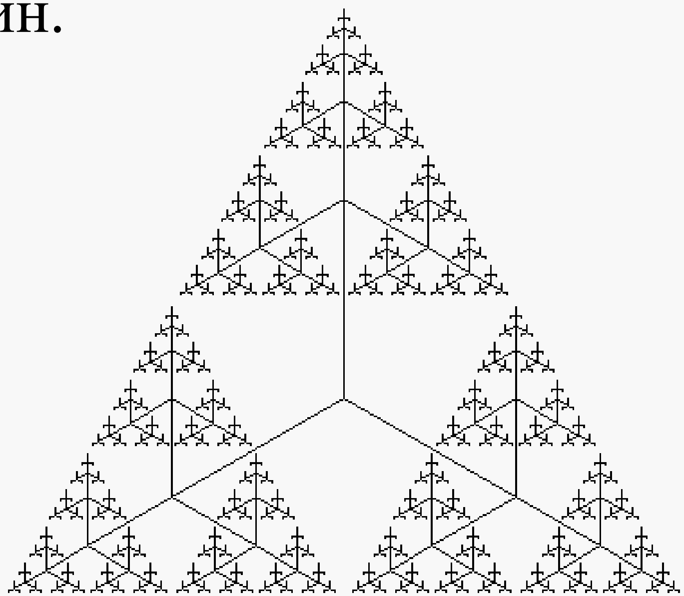
Определение 6б. Двоичное (бинарное) дерево с корнем — это дерево, в котором каждая вершина имеет не более двух дочерних вершин.



Виды деревьев

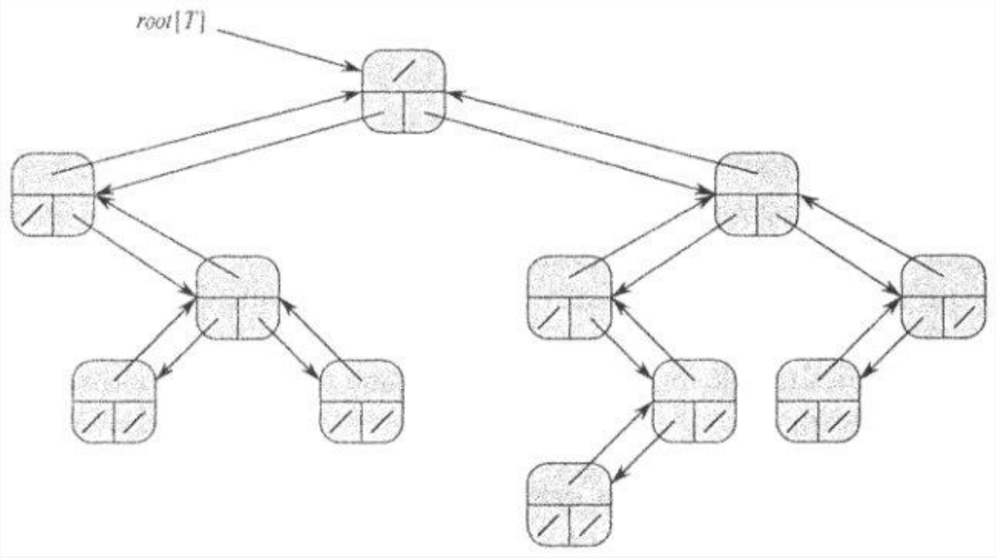
<u>Определение 7а.</u> N-арное дерево — это дерево, в котором степени вершин не превосходят N + 1.

<u>Определение 7б.</u> N-арное дерево с корнем — это дерево, в котором каждая вершина имеет не более N дочерних вершин.



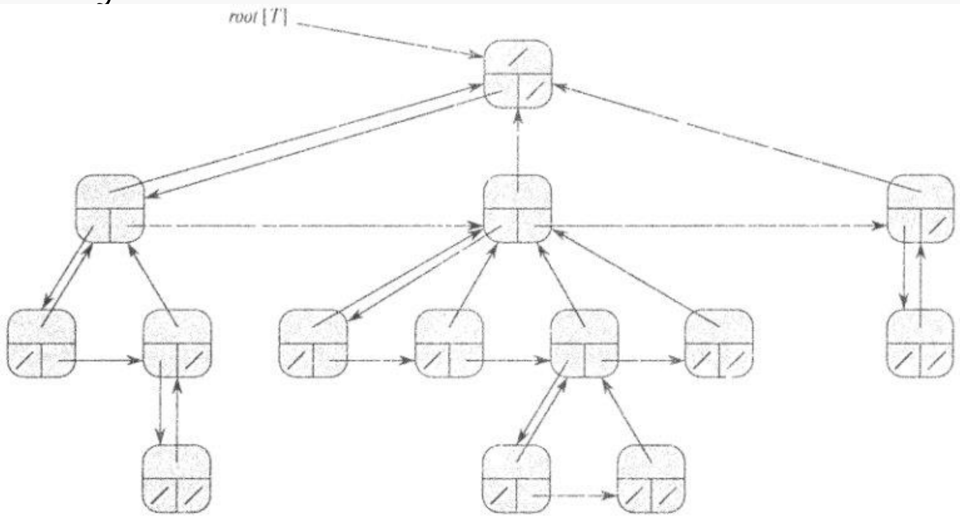
Структуры данных

<u>Определение 8.</u> 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

<u>Определение 9.</u> 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

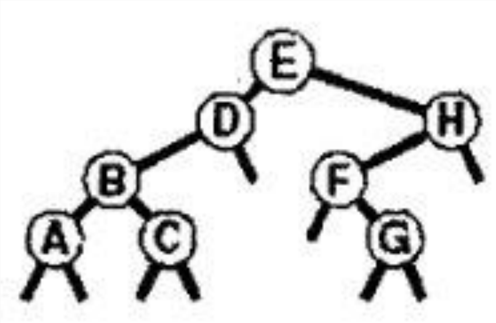
```
// Узел двоичного дерева с данными типа int.
struct CBinaryNode {
    int Data;
    CBinaryNode* Left; // NULL, если нет.
    CBinaryNode* Right; // NULL, если нет.
    CBinaryNode* Parent; // NULL, если корень.
};

// 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**.

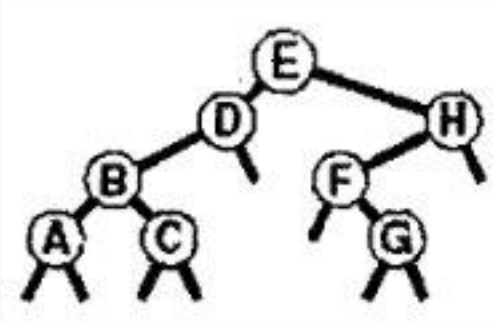
Определение 11. Обходом двоичного дерева в глубину (DFS) называется процедура, выполняющая в некотором заданном порядке следующие действия с поддеревом:
* просмотр (обработка) узла-корня поддерева,
* рекурсивный обход левого поддерева,
* рекурсивный обход правого поддерева.
DFS – Depth First Search.

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.
- **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.

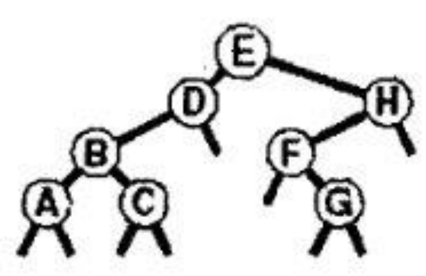
Для прохода по слоям в прямом или обратном порядке требуется другой алгоритм.

Обход дерева в ширину

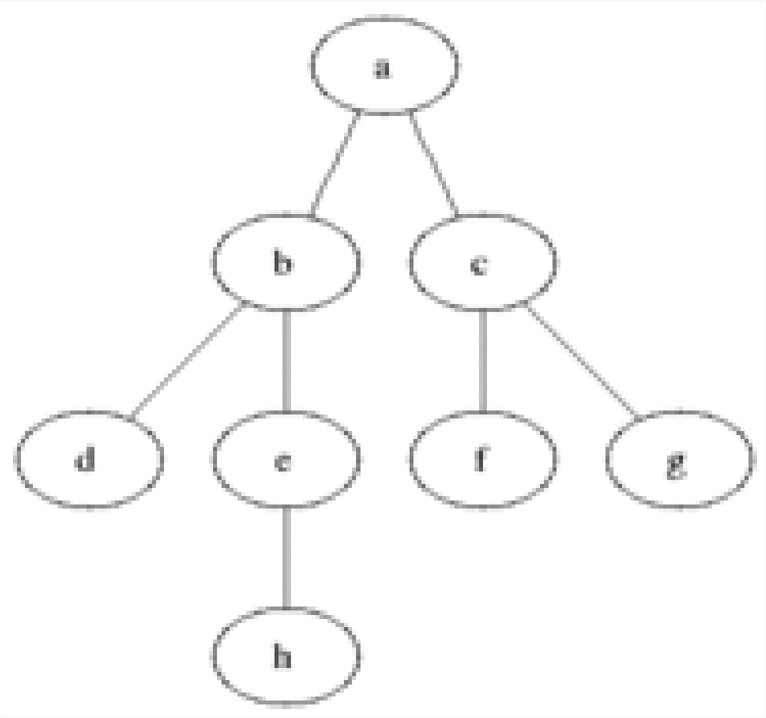
<u>Определение 12.</u> Обход двоичного дерева в ширину (BFS) — обход вершин дерева по уровням (слоям), начиная от корня.
BFS – Breadth First Search.

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

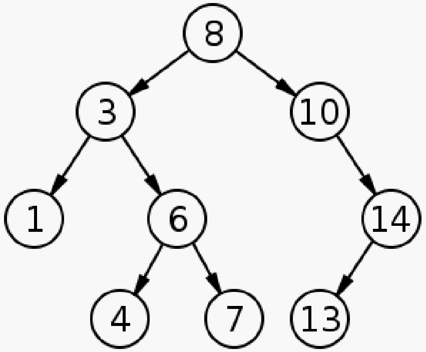
```
// Обход в ширину.
void TraverseBFS( CBinaryNode* root )
{
    queue<CBinaryNode*> q;
    q.put( root );
    while( !q.empty() ) {
        CBinaryNode* node = q.pop();
        visit( node );
        if( node->Left != NULL )
            q.push( node->Left );
        if( node->Right != NULL )
    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

Двоичные деревья поиска

<u>Определение 13.</u> 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.

<u>Пример:</u>



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.

<u>Дано:</u> указатель на корень непустого дерева X.

<u>Задача:</u> найти узел с минимальным значением ключа.

<u>Алгоритм:</u> 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 левого 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.

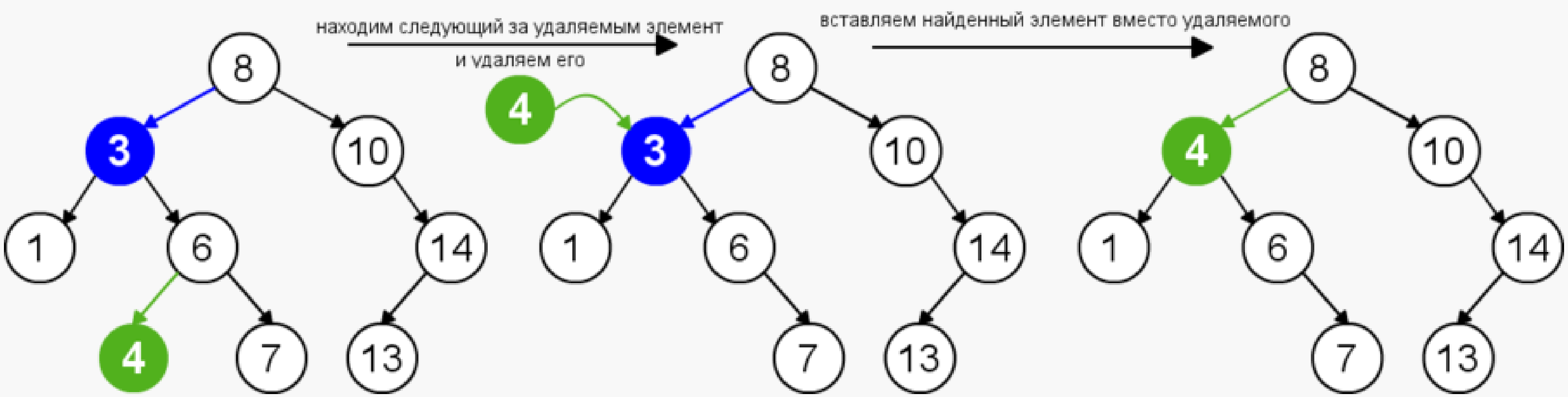
Двоичные деревья поиска

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 левый дочерний узел родителя Z на возможный правый дочерний узел 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 );
};
```

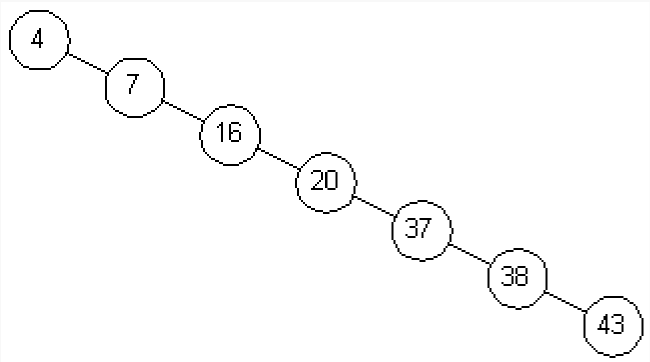
Двоичные деревья поиска

```
// 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.

Балансировка

Самобалансирующиеся деревья.

Случайная балансировка:

- Декартовы деревья.

Гарантированная балансировка:

- АВЛ-деревья,
- Красно-черные деревья.

«Амортизированная» балансировка:

- Сплэй-деревья.

Декартовы деревья

Декартово дерево — это структура данных, объединяющая в себе двоичное дерево поиска и двоичную кучу.

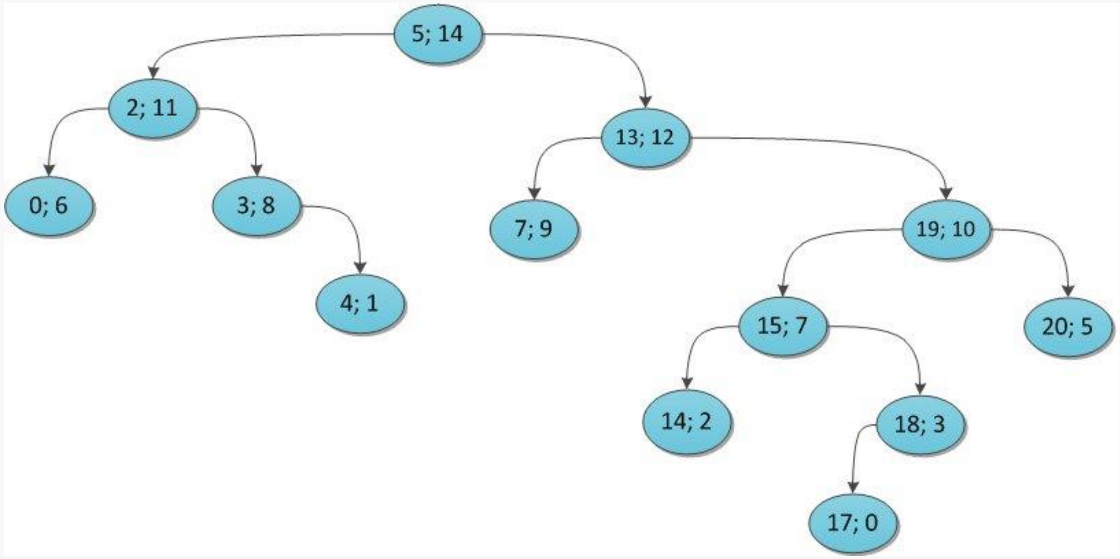
<u>Определение 1.</u> Декартово дерево — двоичное дерево, в узлах которого хранится пары (x, y), где x — это ключ, а y — это приоритет. Все x и все y являются различными. Если некоторый элемент дерева содержит (x_0, y_0), то у всех элементов в левом поддереве $x < x_0$, у всех элементов в правом поддереве $x > x_0$, а также и в левом, и в правом поддереве $y < y_0$.

Таким образом, декартово дерево является двоичным деревом поиска по x и кучей по y.

Декартовы деревья

Другие названия:
* TREAP (TREE + HEAP),
* дуча (дерево + куча),
* дерамида (дерево + пирамида),
* курево (куча + дерево).

Изобретатели — Сидель и Арагон (1989г)



Декартовы деревья

Теорема 1. В декартовом дереве из N узлов, приоритеты которого являются случайными величинами с равномерным распределением, средняя глубина дерева $O(\log n)$.

Без доказательства.

Декартовы деревья

Основные операции:

- Разрезание – SPLIT,
- Слияние – MERGE.

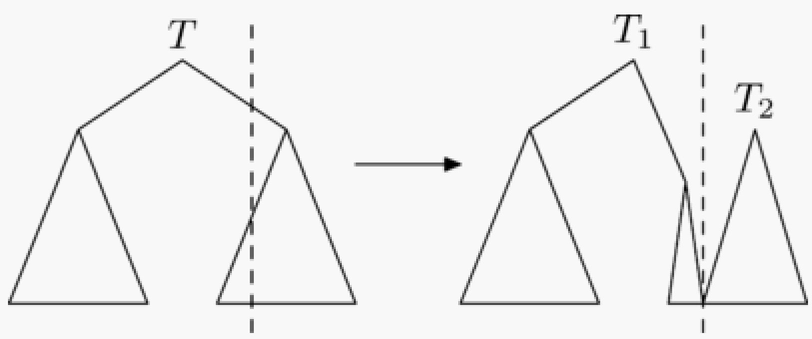
На основе этих двух операций реализуются операции:

- Вставка
- Удаление.

Декартовы деревья

Разрезание — Split

Операция **«разрезать»** позволяет разрезать декартово дерево T по ключу K и получить два других декартовых дерева: T_1 и T_2, причем в T_1 находятся все ключи дерева T, не большие K, а в T_2 – большие K.



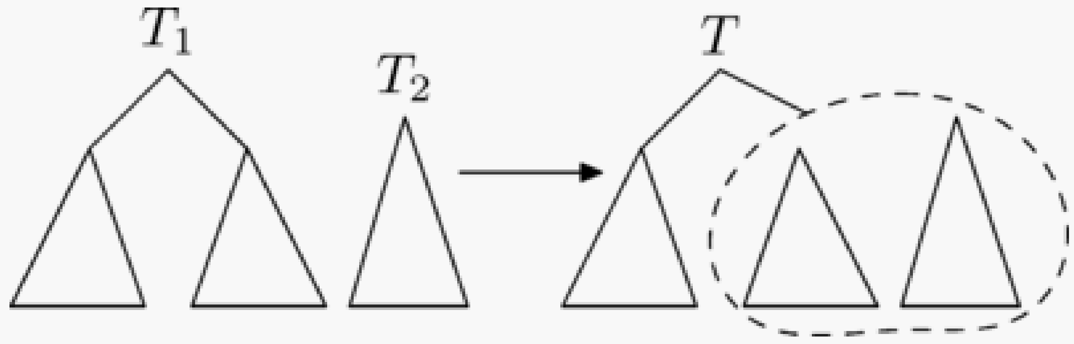
Декартовы деревья

```
// Разрезание декартового дерева по ключу.
void Split( CTreapNode* currentNode, int key, CTreapNode*& left,
    CTreapNode*& right )
{
    if( currentNode == 0 ) {
        left = 0;
        right = 0;
    } else if( currentNode->Key <= key ) {
        Split( currentNode->Right, key, currentNode->Right, right );
        left = currentNode;
    } else {
        Split( currentNode->Left, key, left, currentNode->Left );
        right = currentNode;
    }
}
```

Декартовы деревья

Слияние — MERGE

Операция **«слить»** позволяет слить два декартовых дерева в одно. Причем, все ключи в первом (*левом*) дереве должны быть меньше, чем ключи во втором (*правом*). В результате получается дерево, в котором есть все ключи из первого и второго деревьев.



Декартовы деревья

```
// Слияние двух декартовых деревьев.
CTreapNode* Merge( CTreapNode* left, CTreapNode* right )
{
    if( left == 0 || right == 0 ) {
        return left == 0 ? right : left;
    }
    if( left->Priority > right->Priority ) {
        left->Right = Merge( left->Right, right );
        return left;
    }
    right->Left = Merge( left, right->Left );
    return right;
}
```

Декартовы деревья

Вставка

Добавляется элемент (X, Y), где X – ключ, а Y – приоритет.

Элемент (X, Y) – это декартово дерево из одного элемента. Для того чтобы его добавить в наше декартово дерево T, очевидно, нужно их слить. Но T может содержать ключи как меньше, так и больше ключа X, поэтому сначала нужно разрезать T по ключу X.

<u>Реализация №1.</u>

1. Разобьём наше дерево по ключу X, который мы хотим добавить, на поддеревья T_1 и T_2.
2. Сливаем первое дерево T_1 с новым элементом.
3. Сливаем получившиеся дерево со вторым T_2.

Декартовы деревья

Вставка

Реализация №2.

1. Сначала спускаемся по дереву (как в обычном бинарном дереве поиска по X), но останавливаемся на первом элементе, в котором значение приоритета оказалось меньше Y.
2. Теперь разрезаем поддерево найденного элемента на T_1 и T_2.
3. Полученные T_1 и T_2 записываем в качестве левого и правого сына добавляемого элемента.
4. Полученное дерево ставим на место элемента, найденного в первом пункте.

В первой реализации два раза используется MERGE, а во второй реализации слияние вообще не используется.

Декартовы деревья

Удаление.

Удаляется элемент с ключом X.

Реализация №1.

1. Разобьём дерево по ключу X, который мы хотим удалить, на T_1 и T_2.
2. Теперь отделяем от первого дерева T_1 элемент X, разбивая по ключу $x - \varepsilon$.
3. Сливаем измененное первое дерево T_1 со вторым T_2.

Декартовы деревья

Удаление.

Реализация №2.

1. Спускаемся по дереву (как в обычном двоичном дереве поиска по X), ища удаляемый элемент.
2. Найдя элемент, вызываем слияние его левого и правого сыновей.
3. Результат процедуры ставим на место удаляемого элемента.

В первой реализации два раза используется SPLIT, а во второй реализации разрезание вообще не используется.

Декартовы деревья

Расход памяти и время работы.

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

АВЛ-дерево

<u>**Определение.**</u> **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.

АВЛ-дерево

<u>Теорема.</u> Высота АВЛ-дерева $h = O(\log n)$.

<u>Идея доказательства.</u> В АВЛ-дереве высоты h не меньше F_h узлов, где F_h – число Фибоначчи.
Из формулы Бине следует, что

$$n \geq F_h = \frac{\phi^h - (-\phi)^{-h}}{\phi - (-\phi)^{-1}} \geq C\phi^h,$$

где $\phi = \left(1 + \sqrt{5}\right)/2$ – золотое сечение.

АВЛ-дерево

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

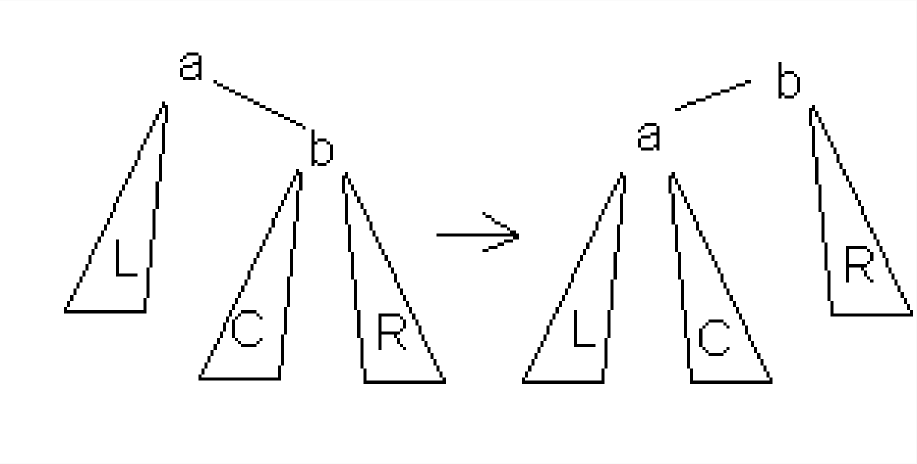
- Малое левое вращение,
- Малое правое вращение,
- Большое левое вращение,
- Большое правое вращение.

АВЛ-дерево

- **Малое левое вращение**

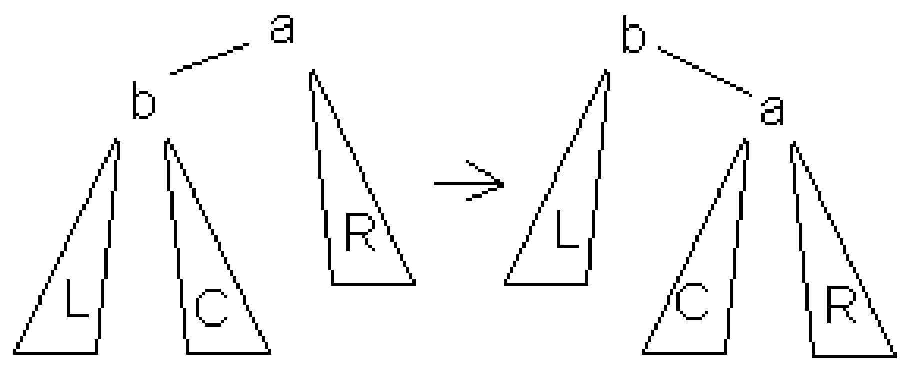
Используется, когда:
высота(R) = высота(L) + 2 и высота(C) ≤ высота(R).

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



АВЛ-дерево

- **Малое правое вращение**

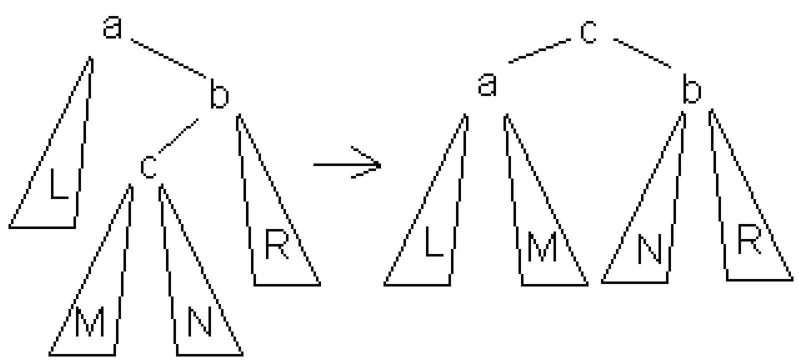


АВЛ-дерево

- **Большое левое вращение**

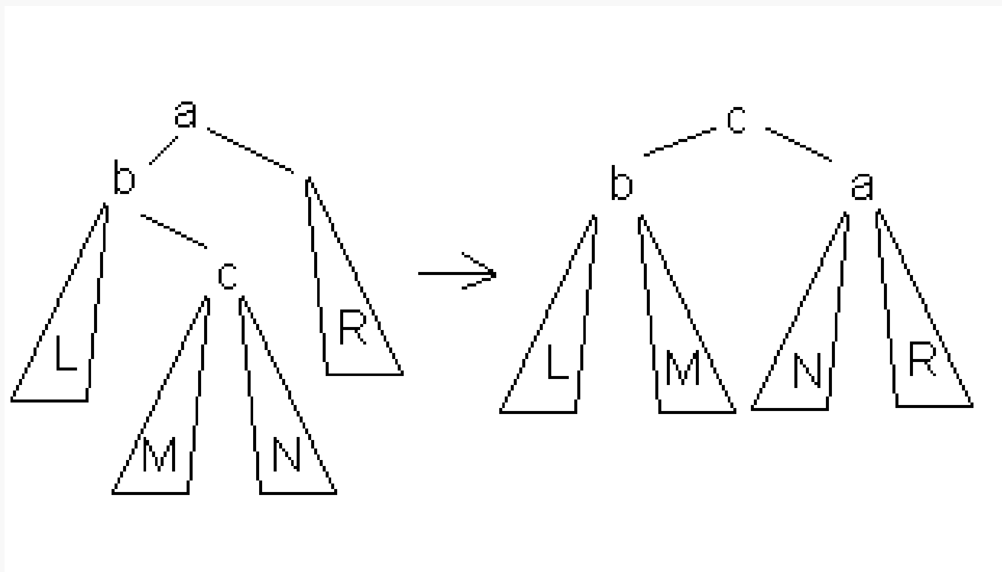
Используется, когда:
высота(R) = высота(L) + 1 и высота(C) = высота(L) + 2.

После операции:
высота дерева уменьшается на 1.



АВЛ-дерево

- **Большое правое вращение**



АВЛ-дерево

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 $= O(\log n)$.

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-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	$O(\log n)$	$O(\log n)$

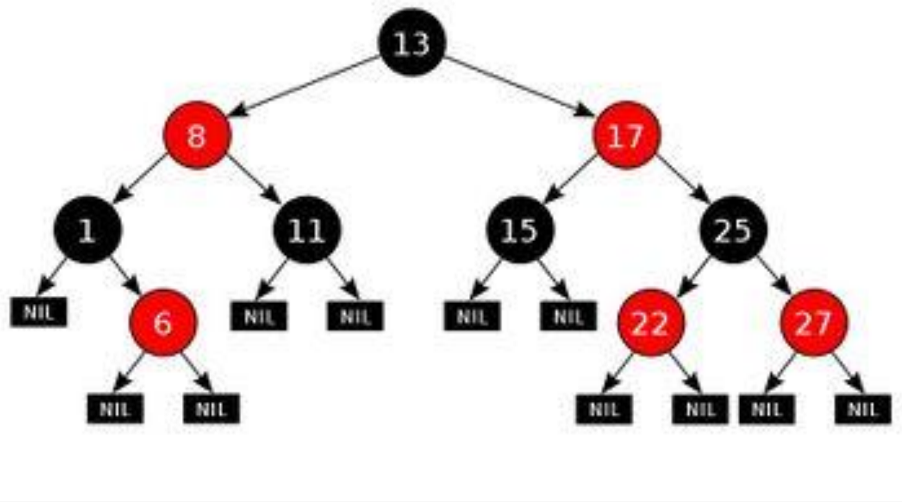
Красно-черные деревья

Красно-черное 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-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 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.

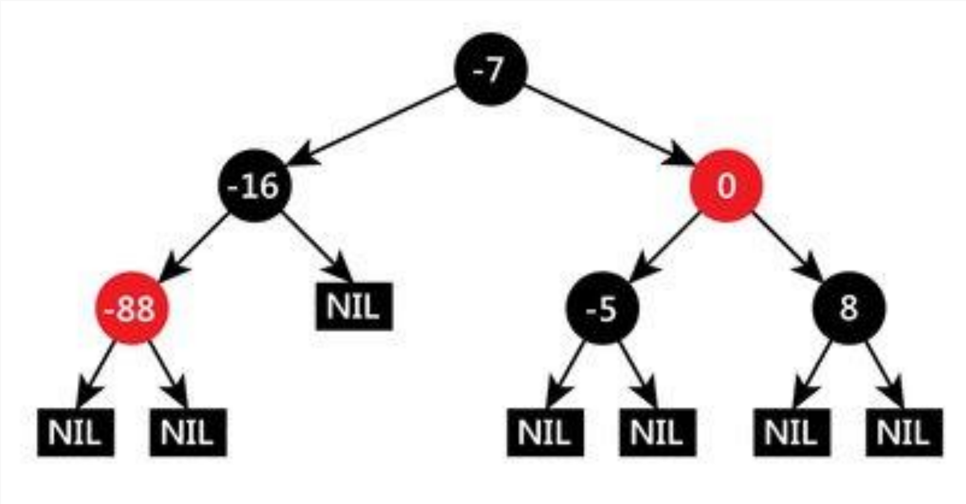
Красно-черные деревья

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

Теорема 2. Красно-черное дерево с N ключами имеет высоту $h = O(\log N)$.

Лемма 1. В красно-черном дереве с черной высотой h_b количество внутренних вершин не менее $2^{h_b+1} - 1$.

Доказательство Леммы 1. По индукции. Предположение индукции: Если черная высота дерева не меньше H, то количество внутренних вершин не менее $2^{H+1} - 1$.
Для листа (фиктивной вершины) лемма верна.
Рассмотрим внутреннюю вершину x.
Пусть ее черная высота $h_b(x) = H + 1$. Если ее потомок p — черный, то черная высота такого потомка $h_b(p) = H$. Если потомок p — красный, то черная высота такого потомка $h_b(p) = H + 1$. В любом случае по предположению индукции в каждом поддереве содержится не менее $2^{H+1} - 1$ вершин.
Тогда во всем дереве не менее $2 \cdot (2^{H+1} - 1) + 1 = 2^{H+2} - 1$, ч.т.д.

Красно-черные деревья

<u>Доказательство Теоремы 2.</u> Если обычная высота равна h, то черная высота дерева будет не меньше $h/2 - 1$.
По лемме 1 количество внутренних вершин N в дереве

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

Прологарифмируем:

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

Итого,

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

т. е.

$$h = O(\log(N)).$$

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.

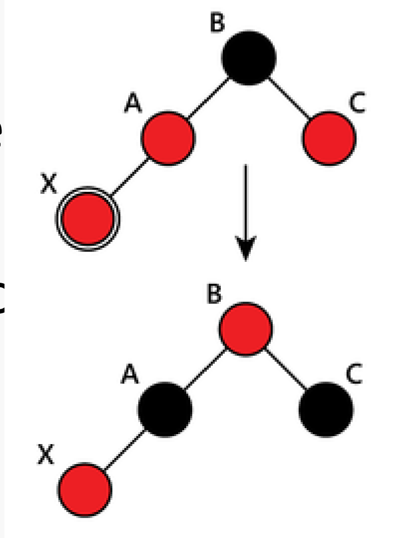
Теперь восстанавливаем свойства красно-черного дерева.
Если отец нового элемента черный, то ничего делать не надо.
Если отец нового элемента красный, то достаточно рассмотреть только два случая:

Красно-черные деревья. Вставка

Случай 1. «Дядя» этого узла тоже красный. Тогда перекрашиваем «отца» и «дядю» в чёрный цвет, а «деда» - в красный.

Теперь «дед» может нарушать свойство дере «Прадед» может быть красного цвета.

Так рекурсивно пытаемся восстановить свойс дерева, двигаясь к предкам.

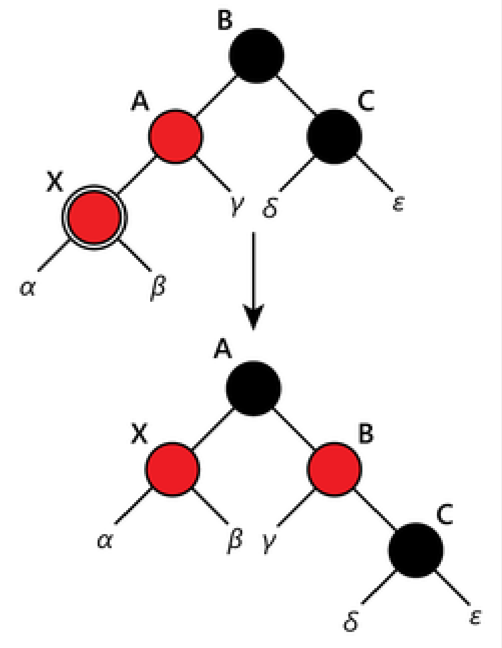


Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет.

Красно-черные деревья. Вставка

<u>Случай 2.</u> «Дядя» черный и правый. Просто выполнить перекрашивание отца в черный цвет нельзя, чтобы не нарушить постоянство чёрной высоты дерева по ветви с отцом.

1) Если добавленный узел X был правым потомком отца A,
то необходимо сначала выполнить левое вращение, которое сделает отца A левым потомком X.
2) Выполняем правый поворот A и B.
Перекрашиваем A и B.
Больше ничего делать не требуется.



Если дядя левый, то порядок действий симме
описанному.

Красно-черные деревья. Вставка

```
// Вставка узла с заданным ключом.
void CRBTree::Insert( double key )
{
    CRBNode* x = root; // Корень может быть nil.
    CRBNode* y = nil; // Узел, под который будем вставлять.
    while( x != nil ) {
        y = x;
        x = key < x->Key ? x->Left : x->Right;
    }
    CRBNode* z = new CRBNode( key );
    z->Parent = y;
    if( y == nil ) // Дерево пусто.
        root = z;
    else if( key < y->Key )
        y->Left = z;
    else
        y->Right = z;
    z->Left = z->Right = nil;
    z->Color = Red;
    InsertFixup( z ); // Восстановление свойств.
}
```

Красно-черные деревья. Вставка

Последовательность обработки.

Случай 1, конец. Если родитель – черный или пришли в корень.

Случай 1 → Случай 1.

Случай 1 → Случай 2, конец.

Случай 2, конец.

Длинная цепочка только:
Случай 1 → Случай 1 → Случай 1 → … → Случай 1 → Случай X, конец.
При этом один переход – переход к отцу X.

Общее время работы вставки – $O(\log N)$.

Красно-черные деревья. Удаление

Удаление вершины.

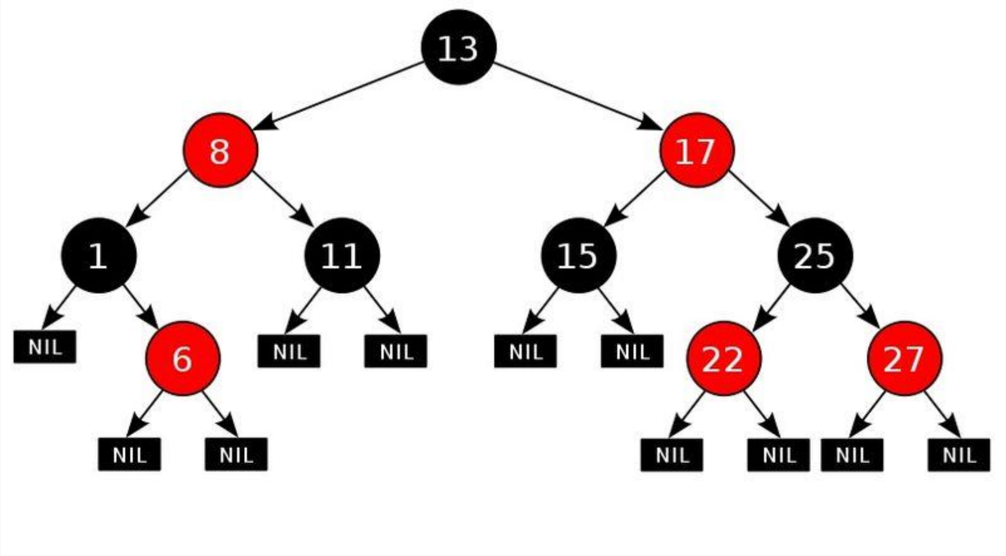
При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на фиктивный лист.
2. Если у неё только один ребёнок, то делаем у родителя ссылку на ребенка вместо этой вершины.

Красно-черные деревья. Удаление

3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка. Удаляем уже эту вершину способом, описанным в первом или во втором пункте, скопировав её ключ в изначальную вершину.

Вершина со следующим значением ключа – самая левая вершина в правом поддереве.
Для вершины 13 – 15.
Для вершины 17 – 22.



Красно-черные деревья. Удаление

Итак, удаляется вершина, имеющая не более одной дочерней.

А. Удаление красной вершины.

При удалении красной вершины свойства дерева не нарушаются. Более того, красная вершина, не может иметь одного потомка. Если бы потомок существовал, то он был бы черным и нарушилось бы свойство постоянства черной глубины для потомка и его соседней фиктивной вершины.

<u>Действие:</u>
* Удалить красную вершину (заменить на лист).

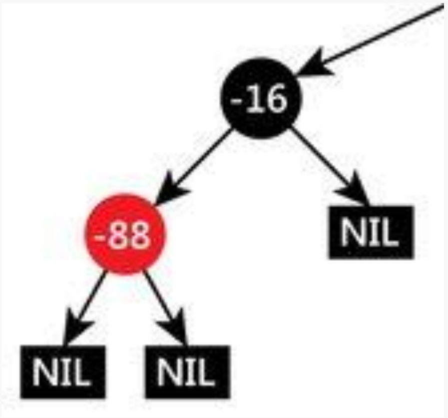
Восстановление свойств потребуется только при удалении чёрной.

Красно-черные деревья. Удаление

Б. Удаление черной вершины с 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).



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 «X».

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

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

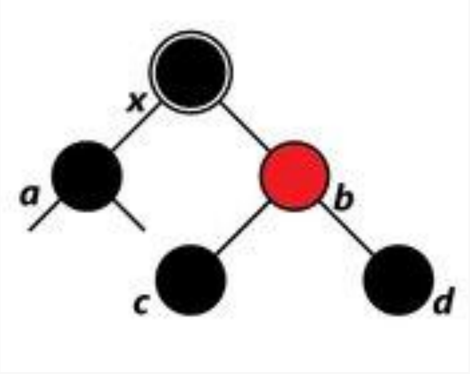
Красно-черные деревья. Удаление

Восстановление свойств. Случай 0.

Если вершина ***x*** – корень.

* Оставим корень просто черным (один раз черным).

Так черная глубина всего дерева уменьшится на 1.



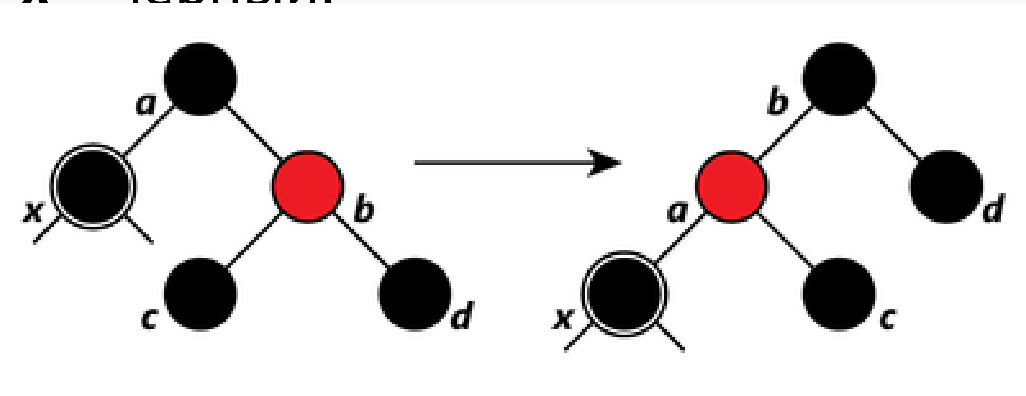
Красно-черные деревья. Удаление

Восстановление свойств. Случай 1.

Если брат ***b*** 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.

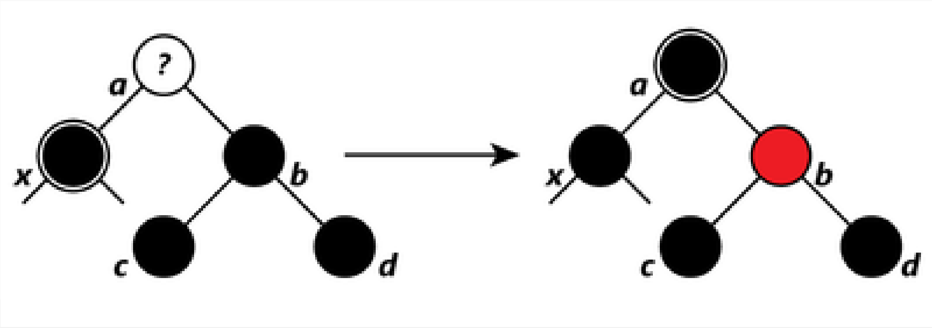


Красно-черные деревья. Удаление

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. Так поддеревья x и b теперь недокрашены в черный.
* Если отец ***a*** был красного цвета, то красим его в черный и завершаем работу. Так черная глубина ***a*** восстановится.
Иначе, считаем отца ***a*** дважды черным, рассматриваем его как x.



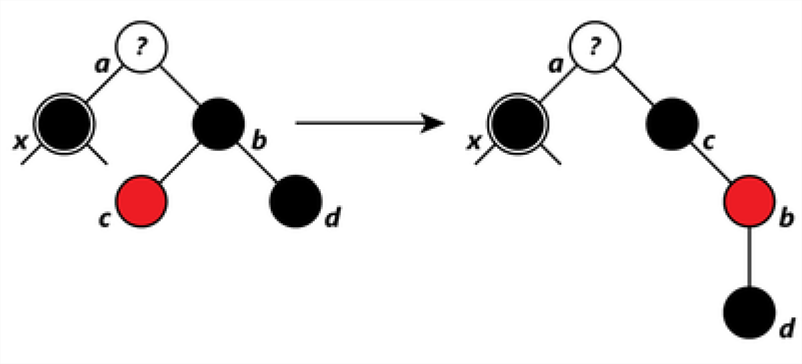
Красно-черные деревья. Удаление

Восстановление свойств. Случай 3.

Если брат ***b*** 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.



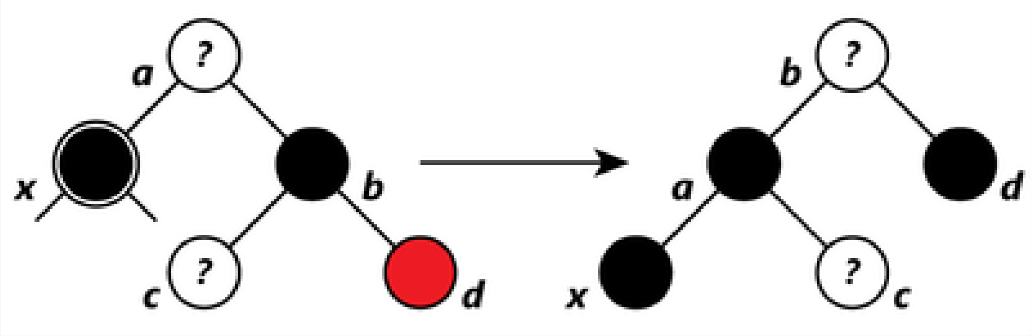
Красно-черные деревья. Удаление

Восстановление свойств. Случай 4.

Если брат ***b*** 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

```
// Удаление вершины.
void CRBTree::Delete( CRBNode* z)
{
    CRBNode* y = nil; // Реально удаляемая вершина.
    if( z->Left == nil || z->Right == nil )
        y = z;
    else
        y = Next( z );
    // Дважды черная.
    CRBNode* x = y->Left != nil ? y->Left : y->Right;
    x->Parent = y->Parent;
    if( y->Parent == nil ) // Удаляется корень.
        root = x;
    else if( y == y->Parent->Left )
        y->Parent->Left = x;
    else
        y->Parent->Right = x;
    if( y != z )
        z->Key = y->Key;
    if( y->Color == Black )
        DeleteFixup( x ); // Восстановление свойств.
    delete y;
}
```

Красно-черные деревья. Удаление

Восстановление свойств.

Последовательность обработки при восстановлении свойств.

- Случай 1 → Случай 2, конец, т. к. отец x – красный.
- Случай 3 → Случай 4, конец.
- Случай 4, конец.
- Случай 2 → любой случай, в том числе Случай 0 и Случай 2.

Длинная цепочка только:
Случай 2 → Случай 2 → Случай 2 → … → Случай 2 → Случай X, конец.
При этом один переход – переход к отцу ***x***.

Общее время работы удаления – $O(\log N)$.

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)

АТД «Ассоциативный массив»

<u>Определение 4.</u> 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»

Реализации ассоциативного массива.

- Массив пар, упорядоченный по ключу. Поиск – бинарный.

Время поиска $O(\log n)$. Время вставки и удаления $O(n)$.

- Сбалансированное дерево поиска.

Время работы операций поиска, вставки и удаления – $O(\log n)$.

STD::MAP реализован на основе красно-черного дерева.

- Хеш-таблицы.

Все операции в среднем – $O(1)$, в худшем – $O(n)$.

МГТУ им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

???

Спасибо за внимание!