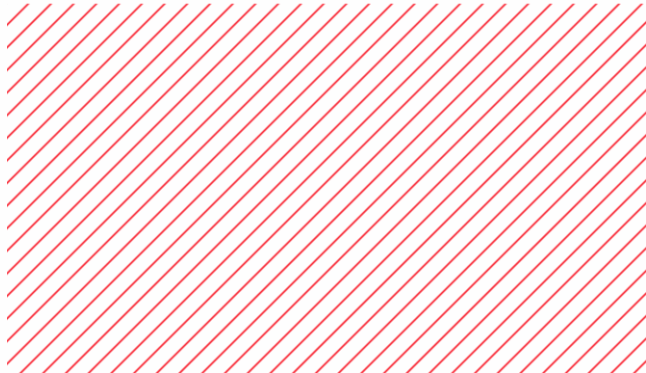
академия больших данных

mail.ru group

Лекция 3. Деревья

Алгоритмы и структуры данных

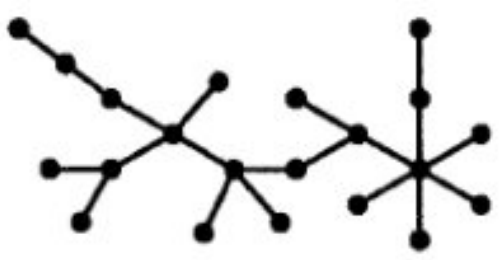
Любимов Яков



План лекции «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. Использование деревьев

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.

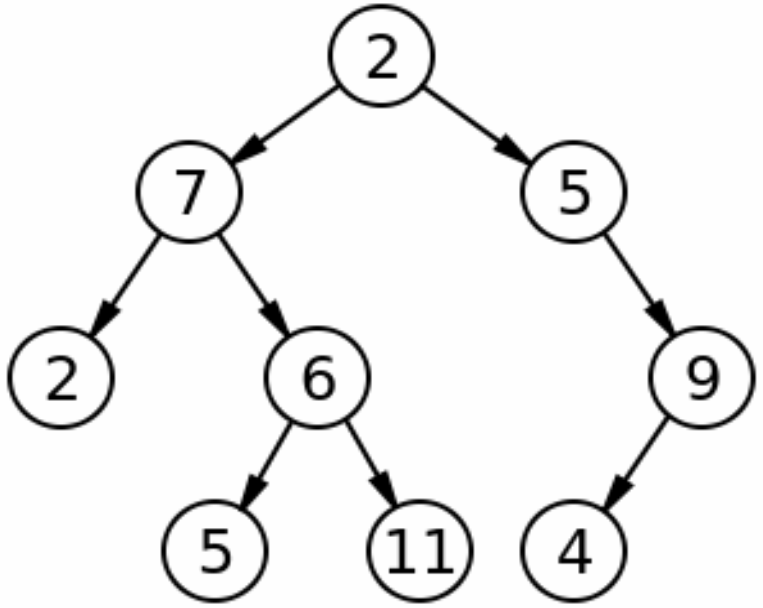
Определения деревьев

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.

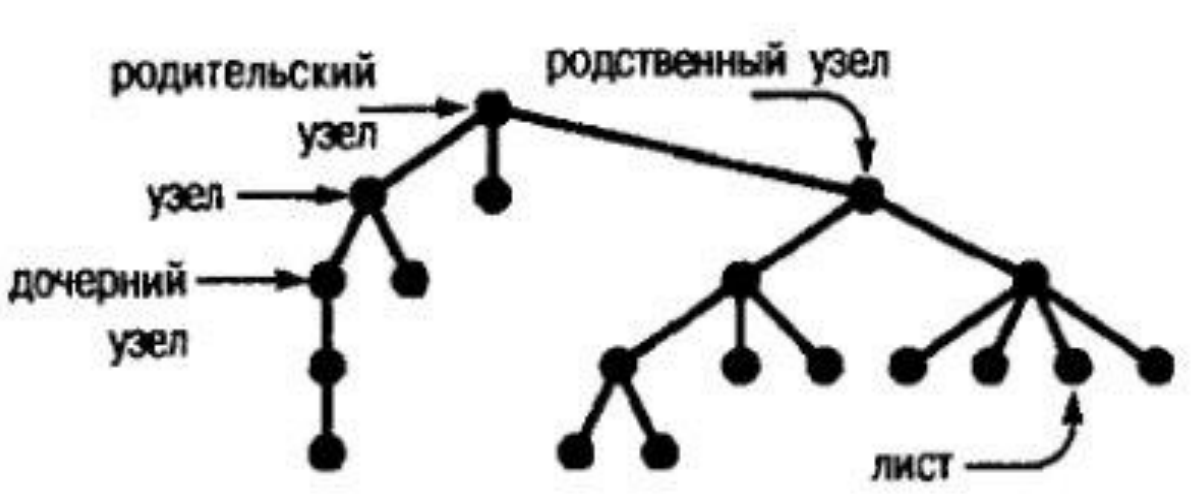
Определение 4. **Высота (глубина)** дерева с корнем — количество вершин в самом длинном пути от корня.

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.

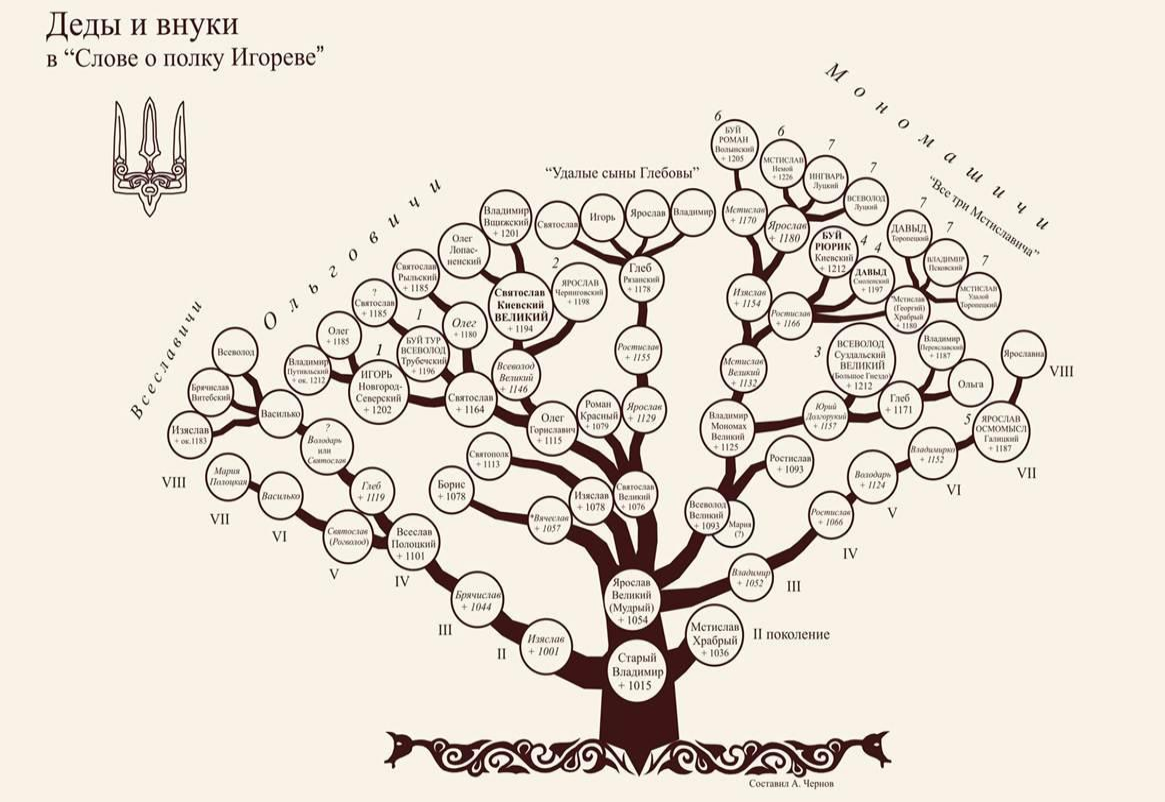


Определения деревьев

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**.



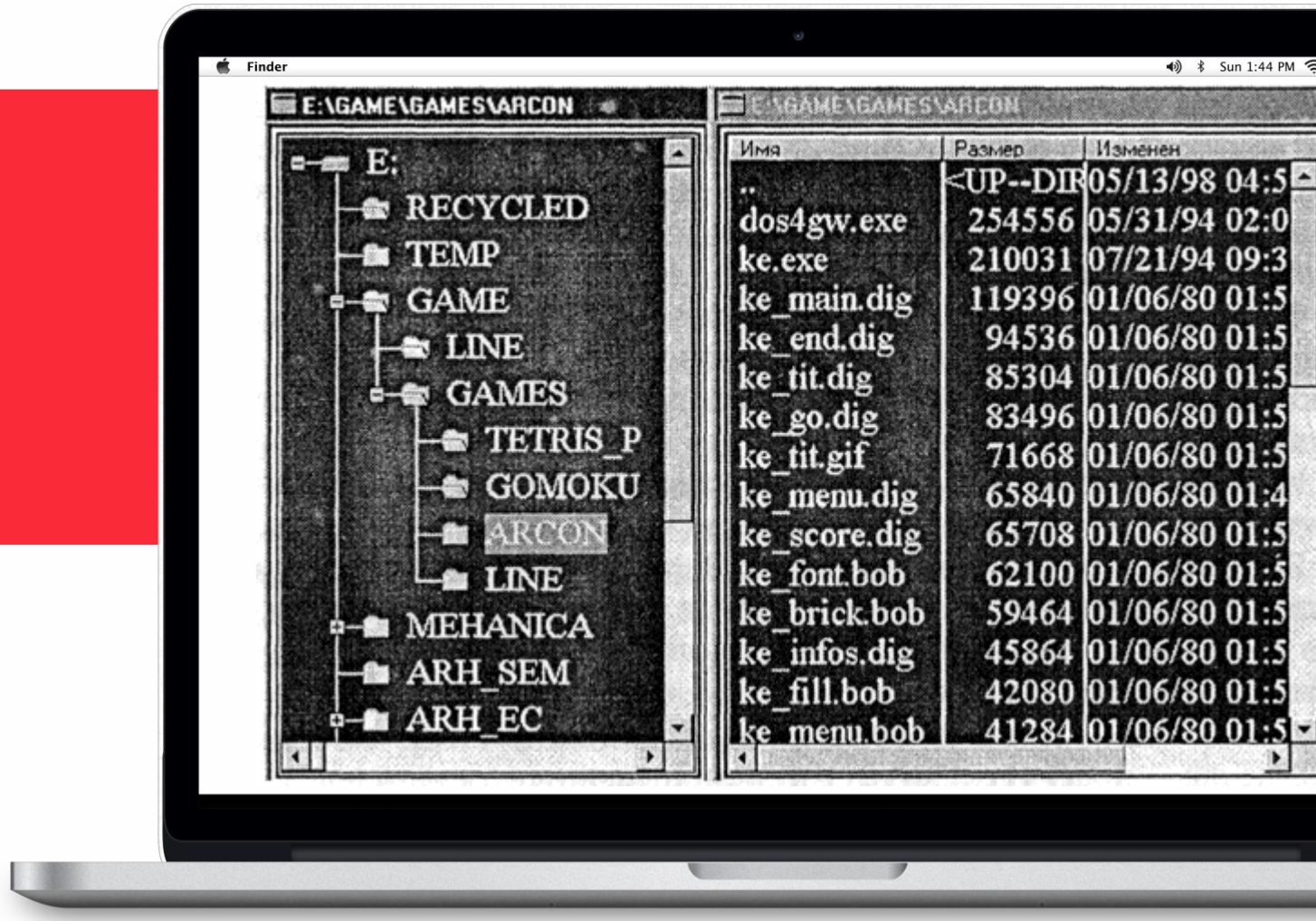
Примеры деревьев



Генеалогическое дерево

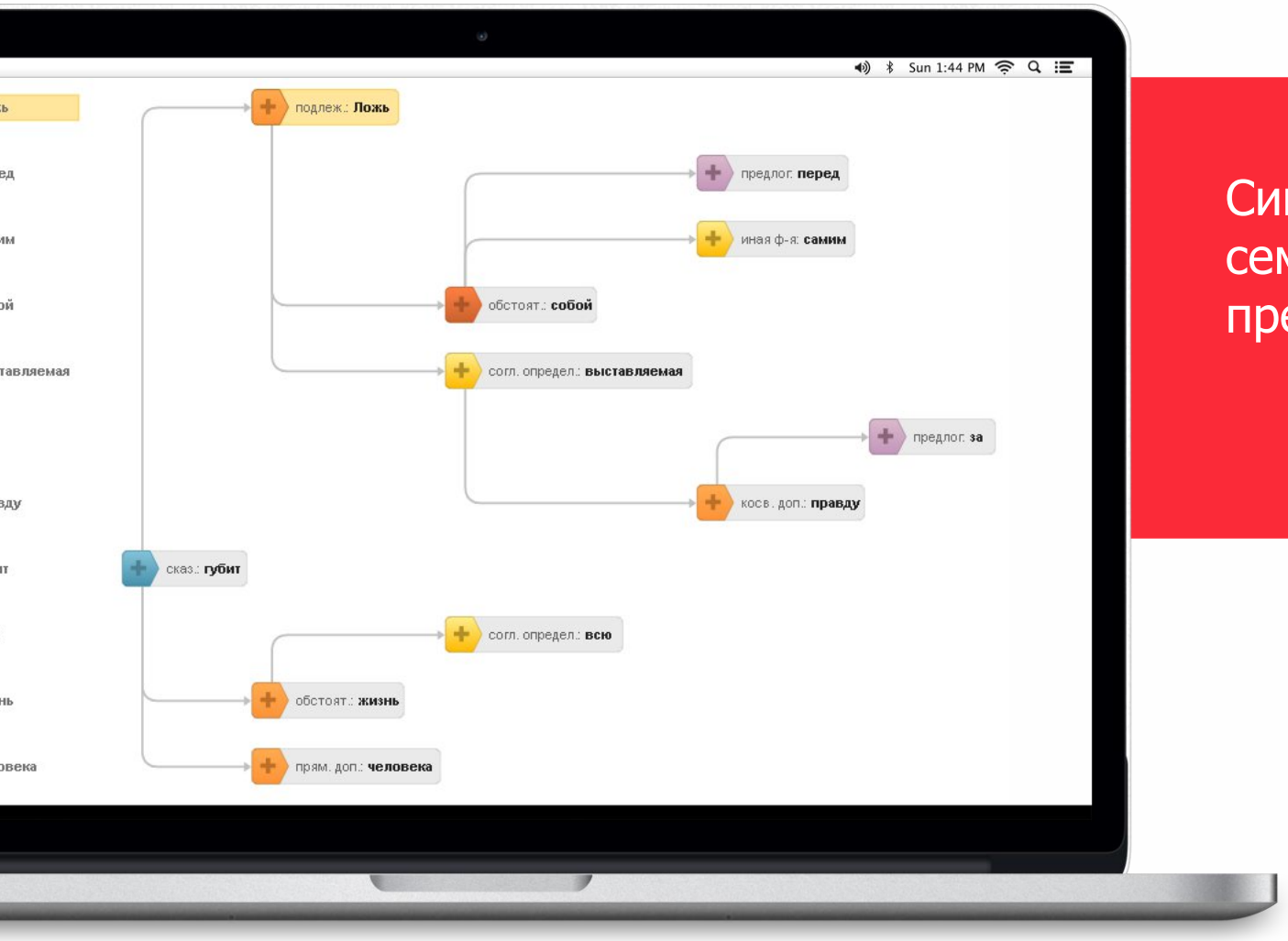
Примеры деревьев

Файловая система



Примеры деревьев

Орг. структура компании



Синтаксический или семантический разбор предложения.

Число вершин и ребер

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

Доказательство.

Самая глубокая вершина является листовой.

Утверждение 2.

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

Доказательство.

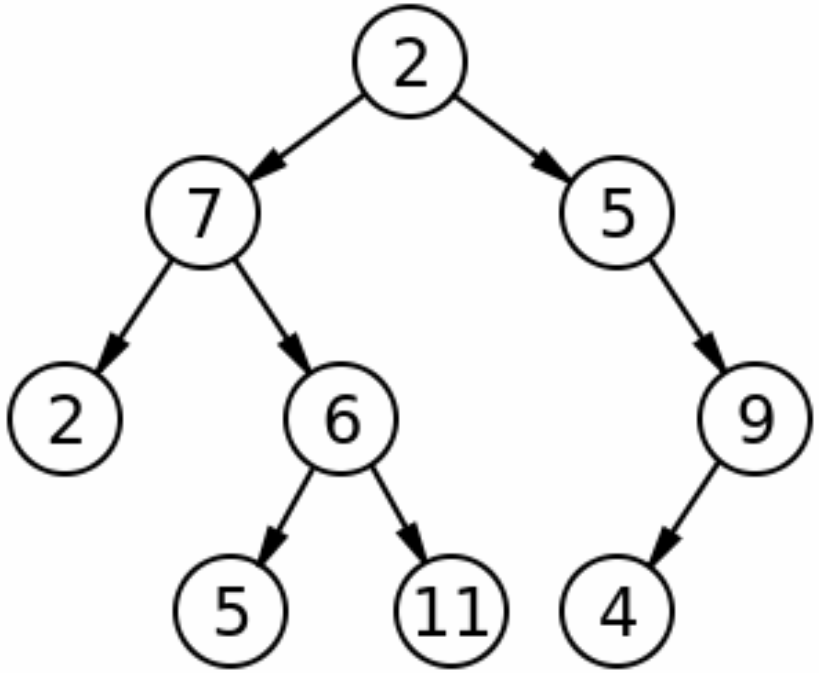
По индукции.
База индукции. N = 1. Одна вершина, ноль ребер.
Шаг индукции. Пусть дерево состоит из N + 1 вершины. Найдем листовую вершину. Эта вершина содержит ровно 1 ребро. Дерево без этой вершины содержит N вершин, а по предположению индукции N – 1 ребро. Следовательно, исходное дерево содержит N ребер, ч.т.д.

Виды деревьев

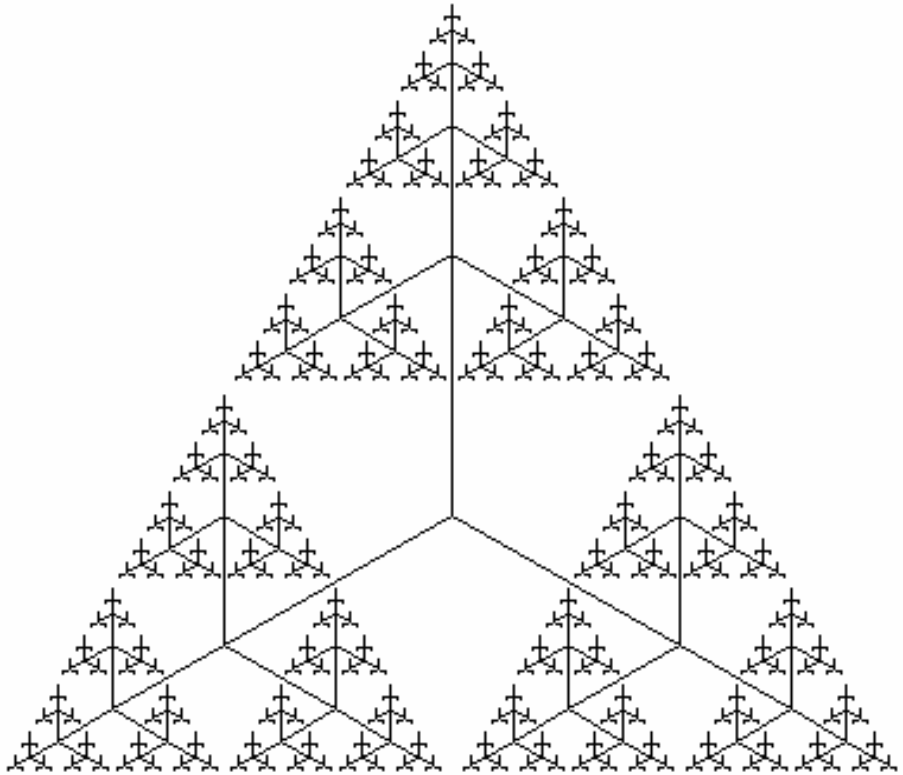
Определение 6а.
Двоичное (бинарное) дерево — это дерево, в котором степени вершин не превосходят 3.

Определение 6б.

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



Виды деревьев



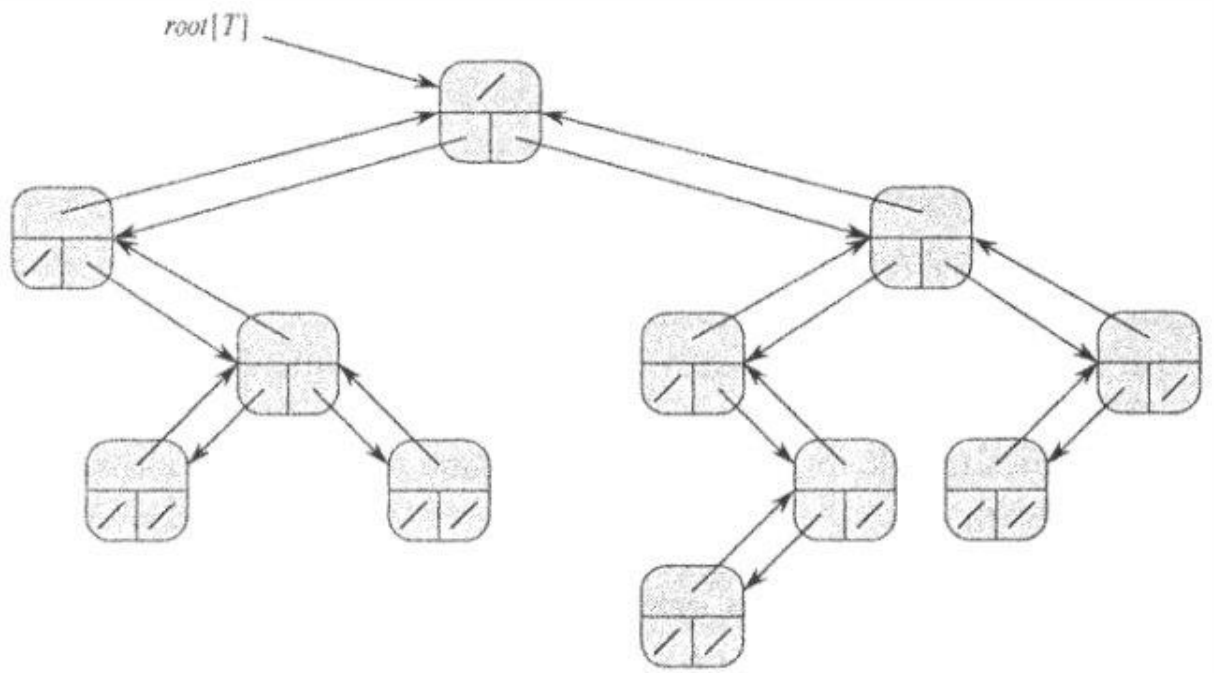
Определение 7а.
N-арное дерево — это дерево, в котором степени вершин не превосходят N + 1.

Определение 7б.

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

Структуры данных

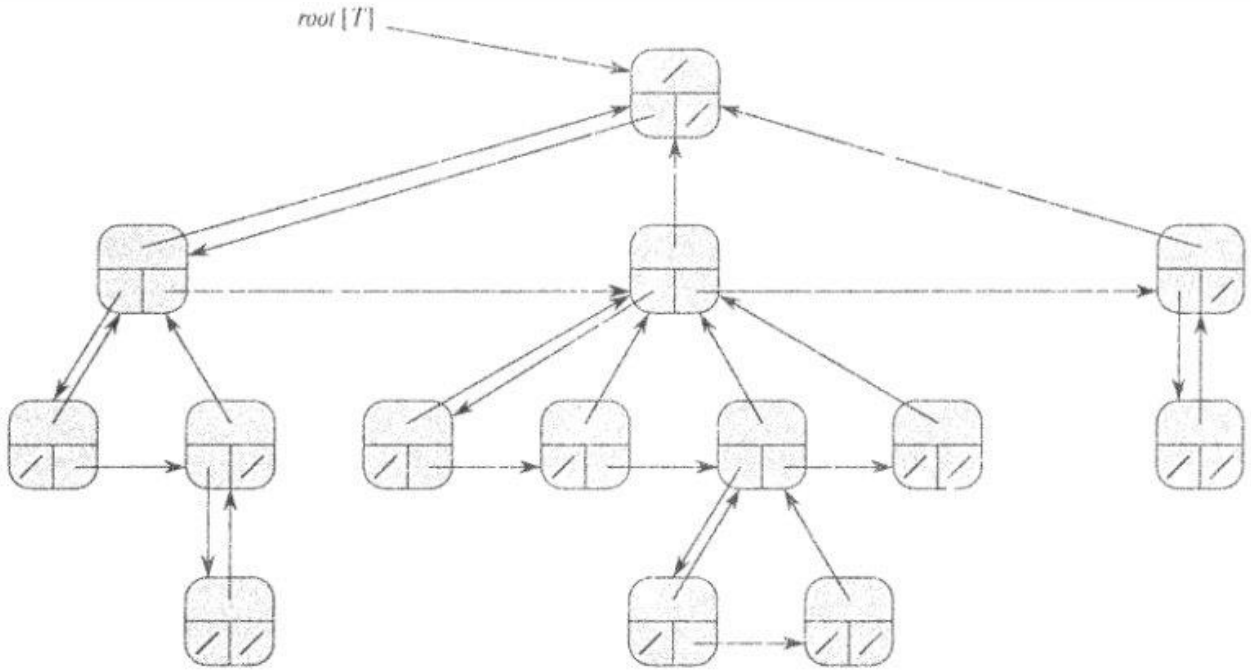
Определение 8. **СД «Двоичное дерево»** — представление двоичного дерева с корнем. Узел – структура, содержащая данные и указатели на левый и правый дочерний узел. Также может содержать указатель на родительский узел.




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

Структуры данных

Определение 9. **СД «N-арное дерево» —** представление N-арного дерева с корнем. Узел – структура, содержащая данные, указатель на следующий родственный узел и указатель на первый дочерний узел. Также может содержать указатель на родительский узел.




```
// Узел дерева с произвольным ветвлением
struct CTreeNode {
    int Data;
    CTreeNode * Next;
    CTreeNode * First;
    CTreeNode * Parent;
};
```

Обход дерева в глубину

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

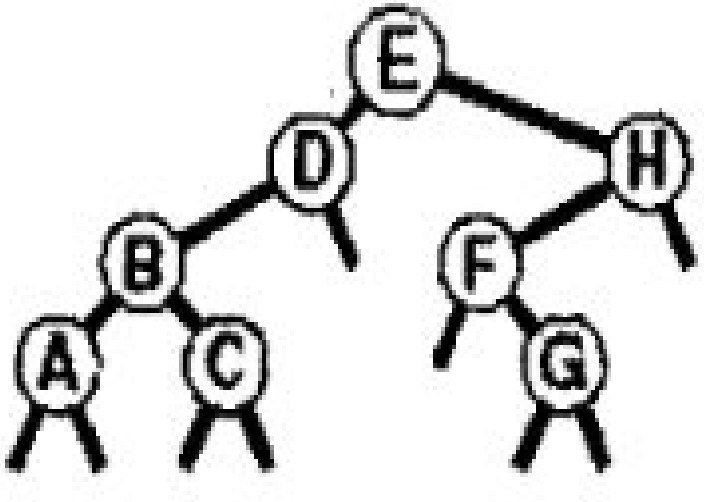
Определение 11.

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.
- **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

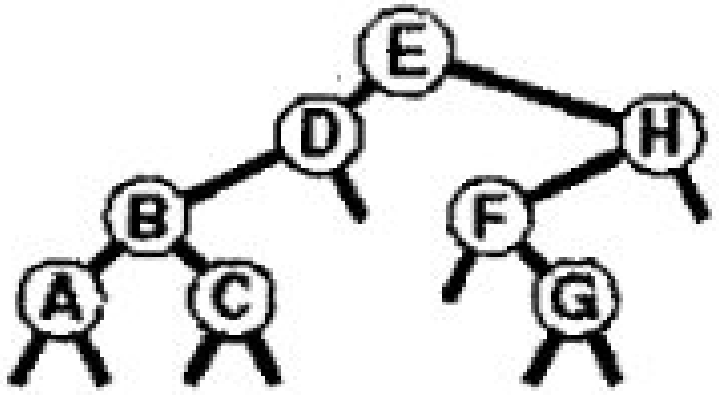
Задача. Вычислить количество вершин в дереве.
Решение. Обойти дерево в глубину в обратном порядке. После обработки левого и правого поддеревьев вычисляется число вершин в текущем поддереве.

Реализация:

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

Обход дерева в ширину

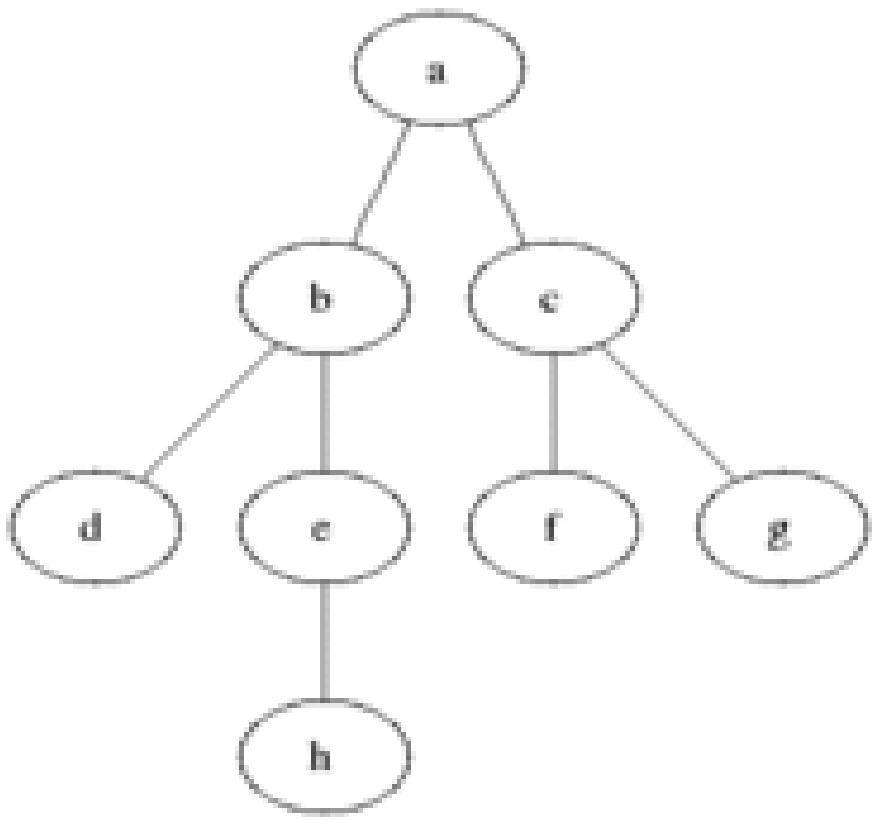
Определение 12.
Обход двоичного дерева в ширину (BFS) — обход вершин дерева по уровням (слоям), начиная от корня.
BFS – Breadth First Search.



Используется очередь, в которой хранятся вершины, требующие просмотра.
За одну итерацию алгоритма:
* если очередь не пуста, извлекается вершина из очереди,
* посещается (обрабатывается) извлеченная вершина,
* в очередь помещаются все дочерние.

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

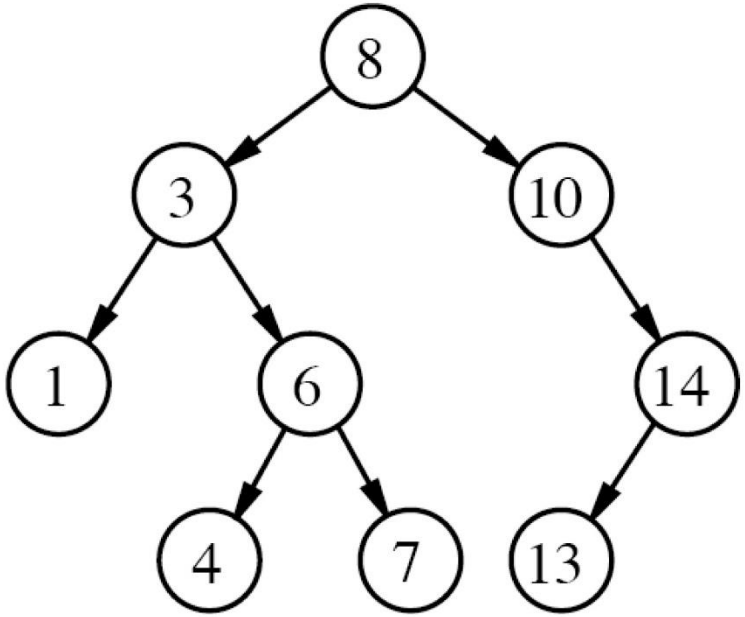
Обход дерева в ширину



Двоичные деревья поиска

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

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



Операции:

1. Поиск по ключу (Find)
2. Поиск минимума, максимума
3. Вставка/Удаление
4. Обход в порядке возрастания/убывания

Алгоритмы

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.

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.

Добавление узла.

Делается аналогично поиску. С той лишь разницей, что спуск идет до конца. При нахождении свободного места, происходит вставка

Алгоримы. Двоичные дервья поиска

Удаление узла.

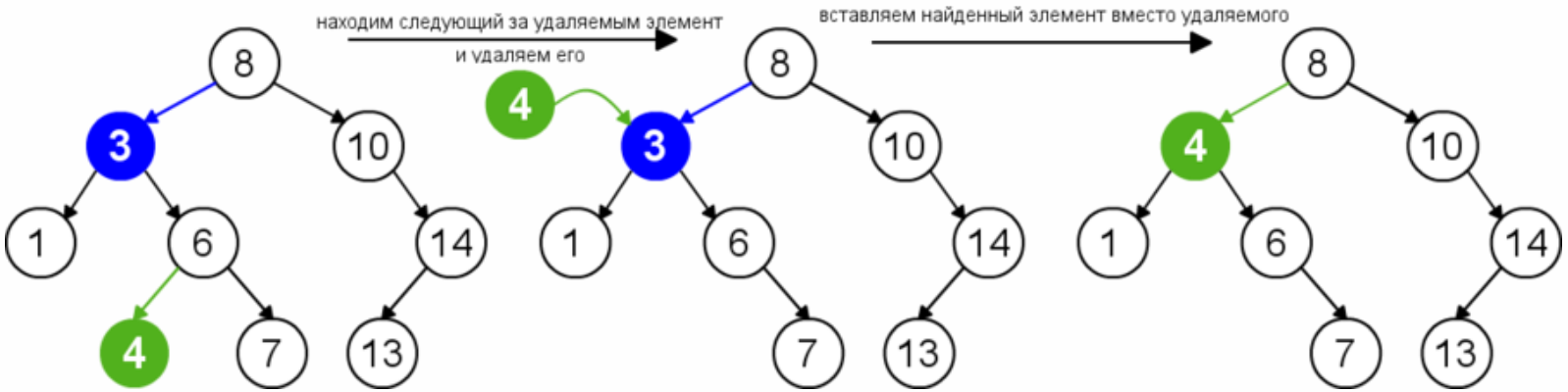
Дано: указатель на корень дерева X и ключ K.

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

Алгоритм: Если дерево пусто, остановиться.

Иначе сравнить K с ключом корневого узла X.

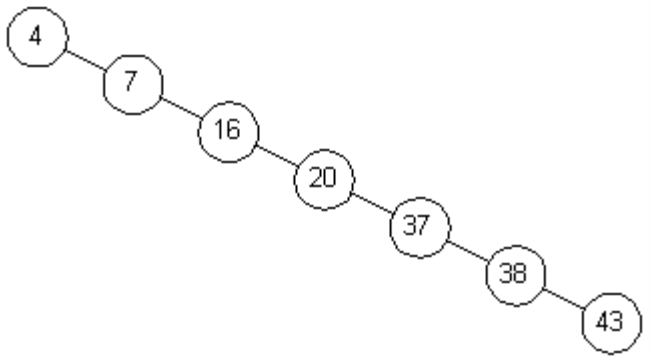
- Если K < X, рекурсивно удалить K из левого поддерева T.
- Если K > X, рекурсивно удалить K из правого поддерева T.
- Если K == X, то необходимо рассмотреть три случая:
  1. Обоих дочерних нет. Удаляем узел X, обнуляем ссылку.
  2. Одного дочернего нет. Переносим дочерний узел в X, удаляем узел.
  3.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



Балансировка

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

Глубина дерева может достигать n.



Необходима балансировка.

Типы балансировок:

Самобалансирующиеся деревья.

Случайная балансировка:

- Декартовы деревья.

Гарантированная балансировка:

- АВЛ-деревья,
- Красно-черные деревья.

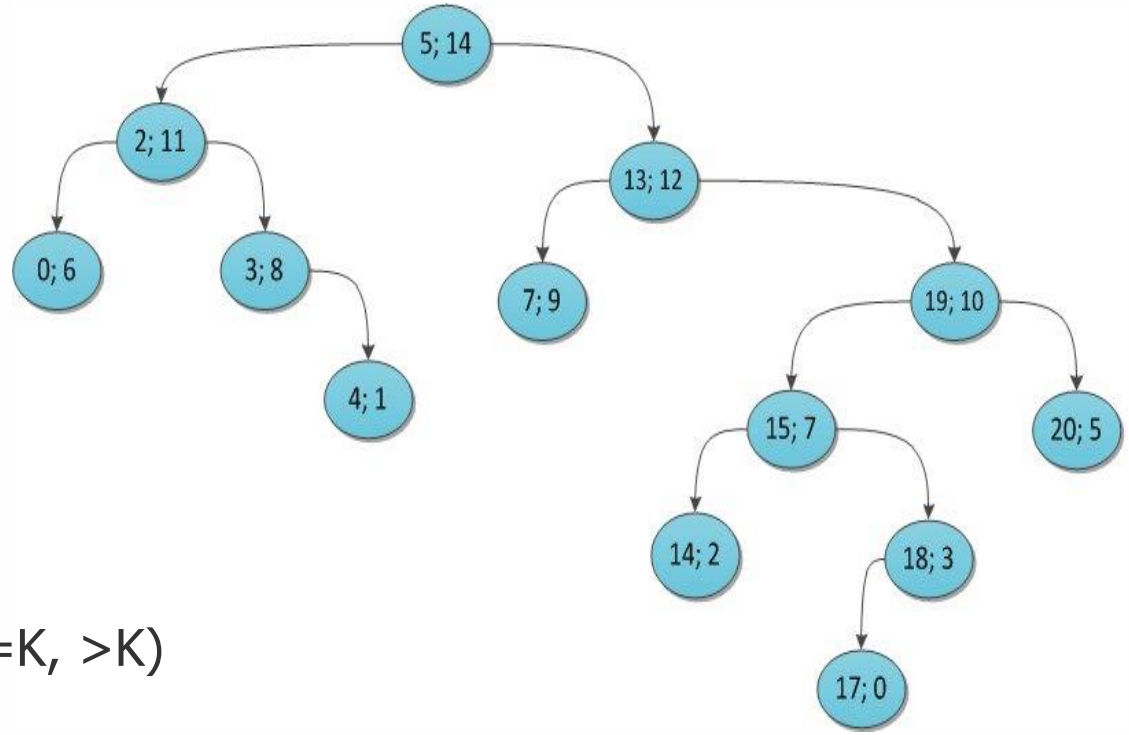
«Амортизированная» балансировка:

- Сплэй-деревья.

Декартовы деревья

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

Декартово дерево – двоичное дерево, в узлах которого хранятся пары ***(x, y)***, где ***x*** – это ключ, а ***y*** – это приоритет. Все ***x*** и все ***y*** являются различными. Если некоторый элемент дерево содержит пару ***(K, P)***, то у всех элементов в левом поддереве ***x < K***, а в правом ***x > K***, приоритеты и в левом, и в правом поддереве меньше ***P (y < P)***



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

Основные операции:

- Split (по ключу K, на выходе 2 дерева (<=K, >K)
- Merge (позволяет слить 2 дерева в одно)

Декартовы деревья

Split:

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

Merge:

```
// Слияние двух декартовых деревьев.
Node* Merge(Node* left, Node* right)
{
  if(!left || !right)
    return left == 0 ? right : left;

  if(left->Priority > right->Priority) {
    left->Right = Merge(left->Right, right);
    return left;
  } else {
    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.

Реализация №1.

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.

АВЛ-дерево

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

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

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

Специальный балансирующие операции:

- Малое левое вращение
- Малое правое вращение
- Большое левое вращение
- Большое правое вращение

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

Идея доказательства. 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 $\phi = (1 + \sqrt{5})/2$ – золотое сечение.

АВЛ-дерево

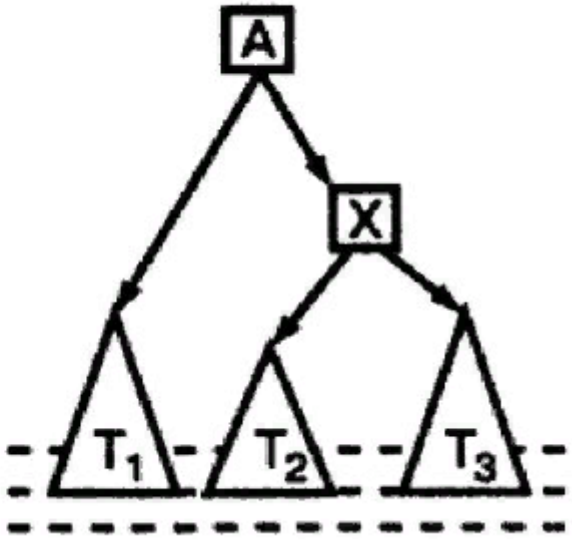
- **Малое правое вращение**

Используется, когда:

- *h(T1) = h(T3) + 2*
- *h(T2) ≤ h(T1).*

После операции:

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



АВЛ-дерево

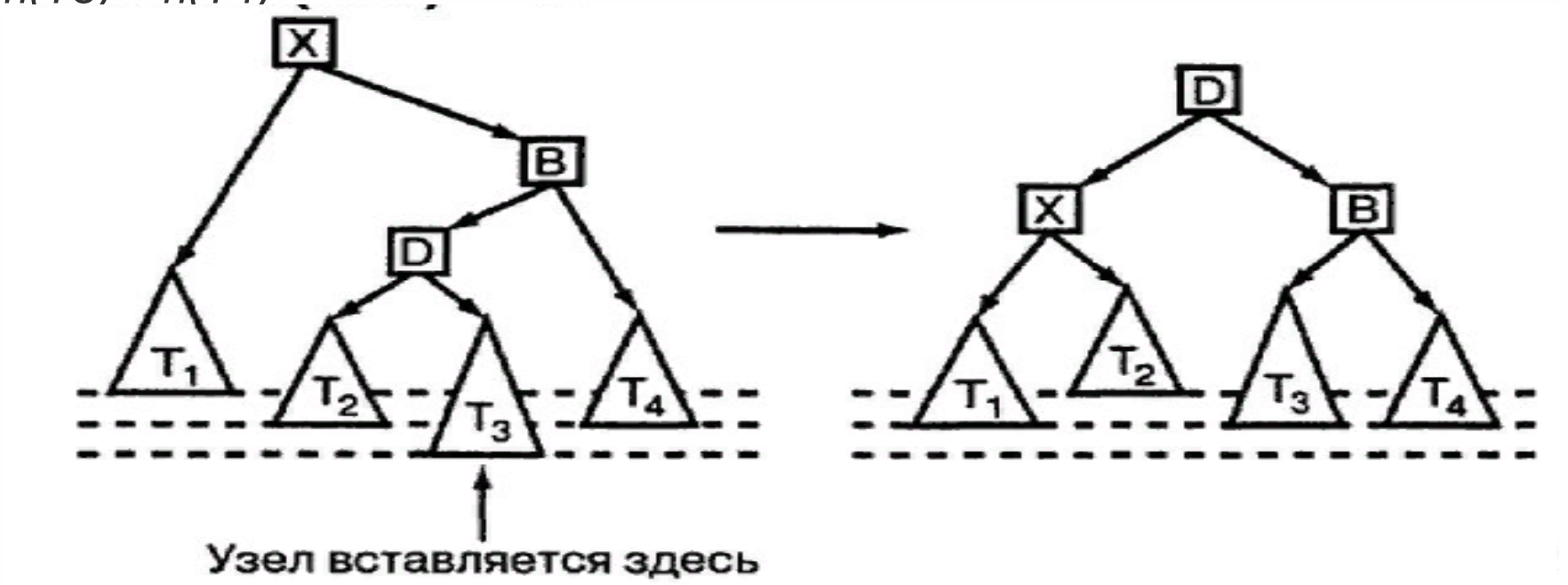
- **Большое левое вращение**

Используется, когда:

- *h(T3) = h(T1) + 2*
- *h(T3) > h(T4)*

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. Включаем новую вершину как в стандартной операции вставки в дерево поиска.
3. "Отступаем" назад от добавленной вершины к корню. Проверяем в каждой вершине сбалансированность. Если разность высот поддеревьев равна 2 – выполняем нужное вращение.

Время работы = $O(\log n)$.

Удаление элемента

1. Ищем вершину D, которую требуется удалить.
2. Проверяем, сколько поддеревьев в D:
   - Если D – лист или D имеет одно поддерево, то удаляем D.
   - Если D имеет два 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 = $O(\log n)$.

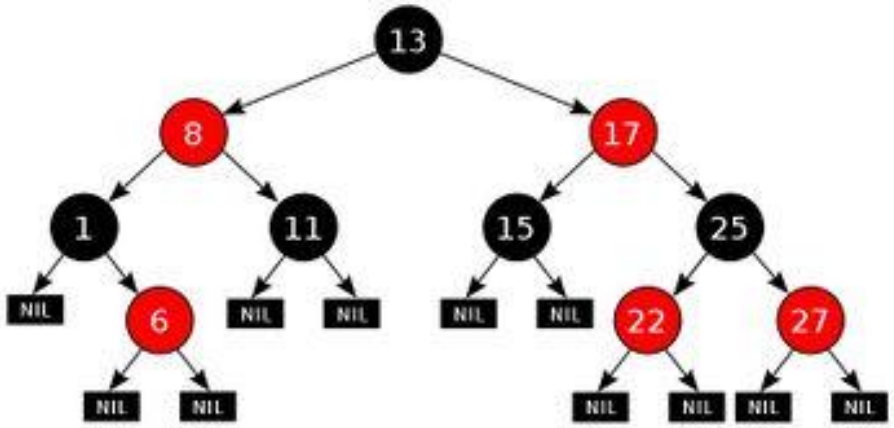
Красно-черные деревья

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).

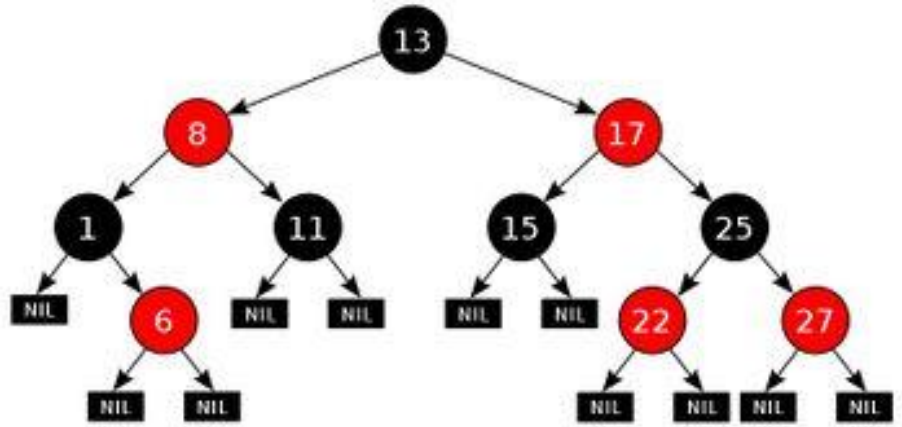


Красно-черные деревья

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

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

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



Красно-черные деревья

Вставка

Вставляем вместо листа новый элемент **красного** цвета

- Если отец нового элемента **черный**, то ничего делать не надо.
- Если отец нового элемента **красный**, смотрим на «дядю»
  - «дядя» этого узла тоже **красный**. Тогда перекрашиваем «отца» и «дядю» в **чёрный** цвет, а «деда» - в **красный**.
  - «дядя» **черный**. Просто выполнить перекрашивание отца в **черный** цвет нельзя, чтобы не нарушить постоянство чёрной высоты дерева по ветви с отцом. Нужны вращения
    - Предположим «дядя» правый сын нашего «дедушки «
    - Если добаленый узел был правым потомком «отца», то делаем большое правое вращение, иначе малое правое
    - Выполняем перекрашивание

Удаление

- При удалении **красной** вершины свойства дерева не нарушаются.
- Удаление **черной** вершины с потомком. Копируем данные из потомка. Удаляем его, он **красный**
- Удаление **черной** вершины без потомков. 5 различных случаев.

Сплей-дерево

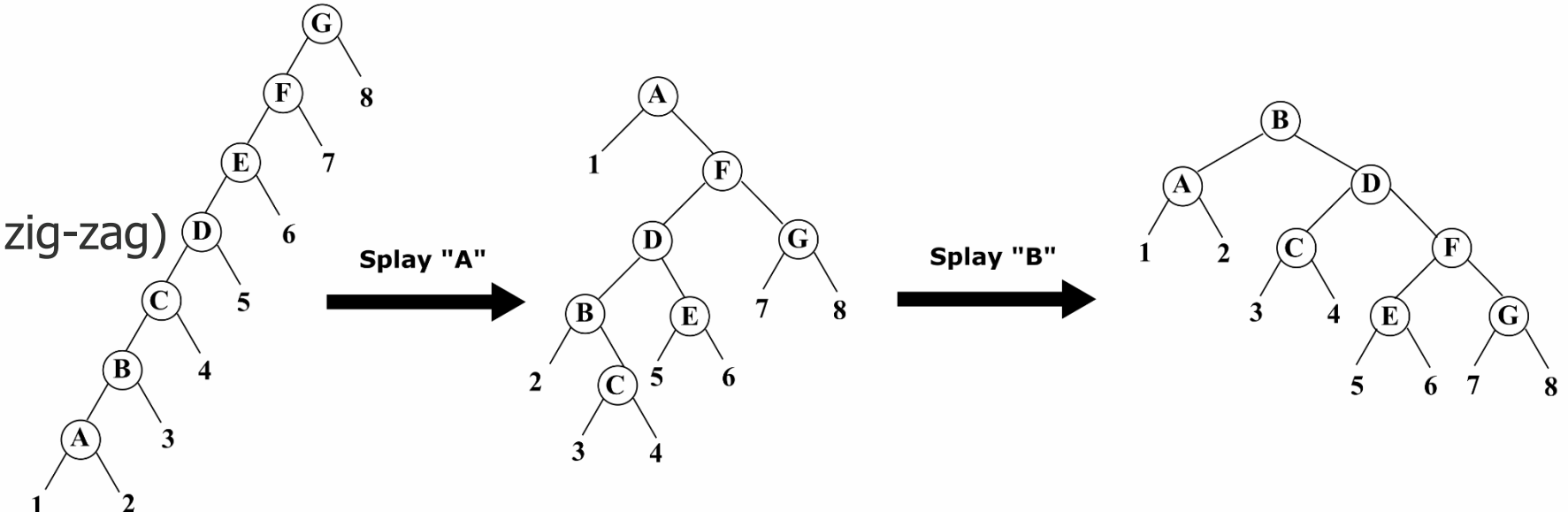
Сплей-дерево — это двоичное дерево поиска. Оно позволяет находить быстрее те данные, которые использовались недавно. Относится к разряду сливаемых деревьев. Сплей-дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году

Эвристика:

Для того, чтобы доступ к недавно найденным данным был быстрее, надо, чтобы эти данные находились ближе к корню.

Операции:

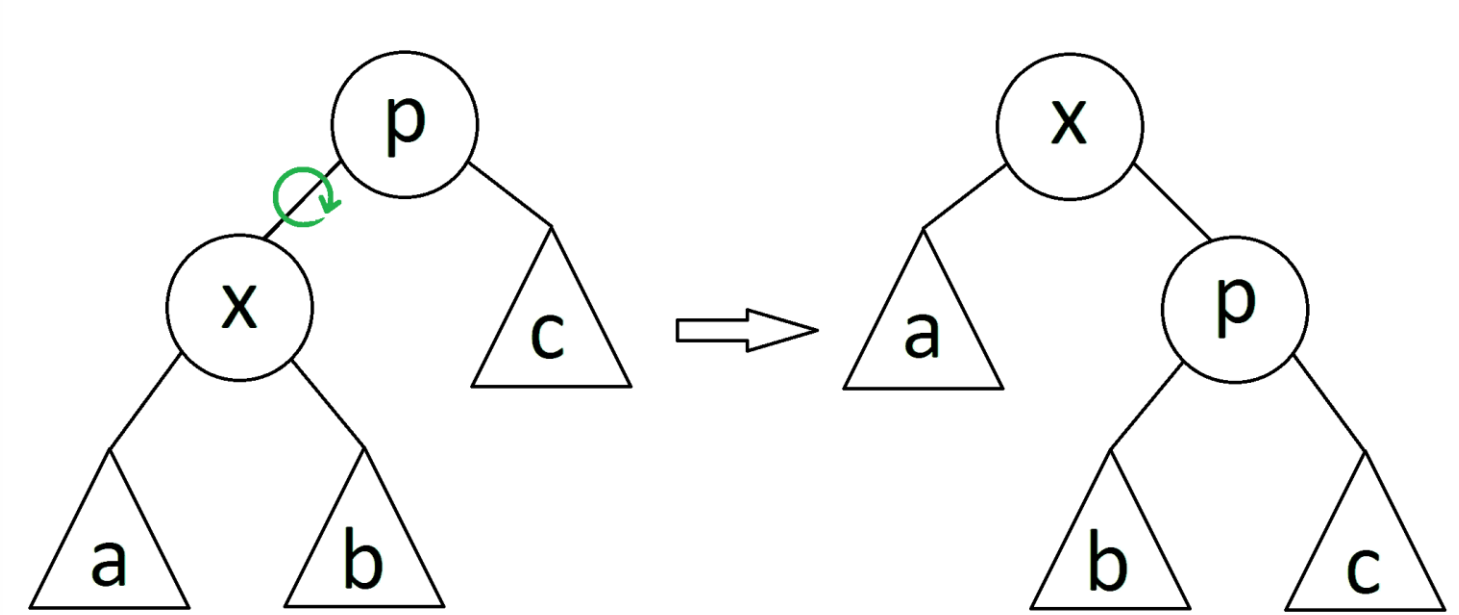
- Splay (zig, zig-zig, zig-zag)
- Split
- Merge



Сплей-дерево

Zig

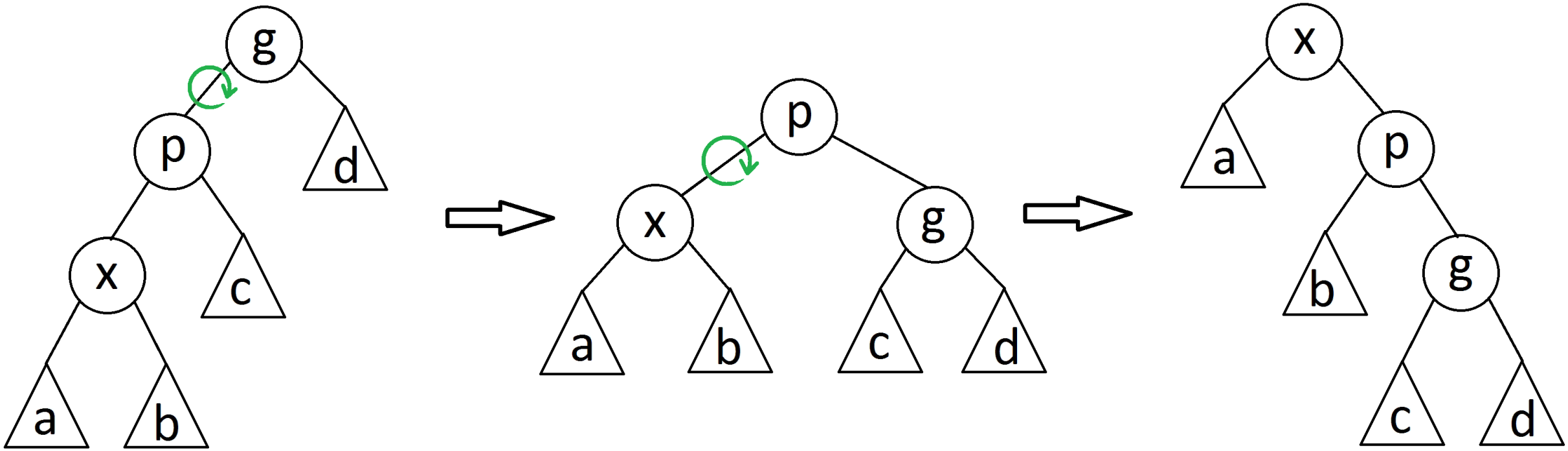
Текущий узел – сын корня. Делаем малый поворот.



Сплей-дерево

ZigZig

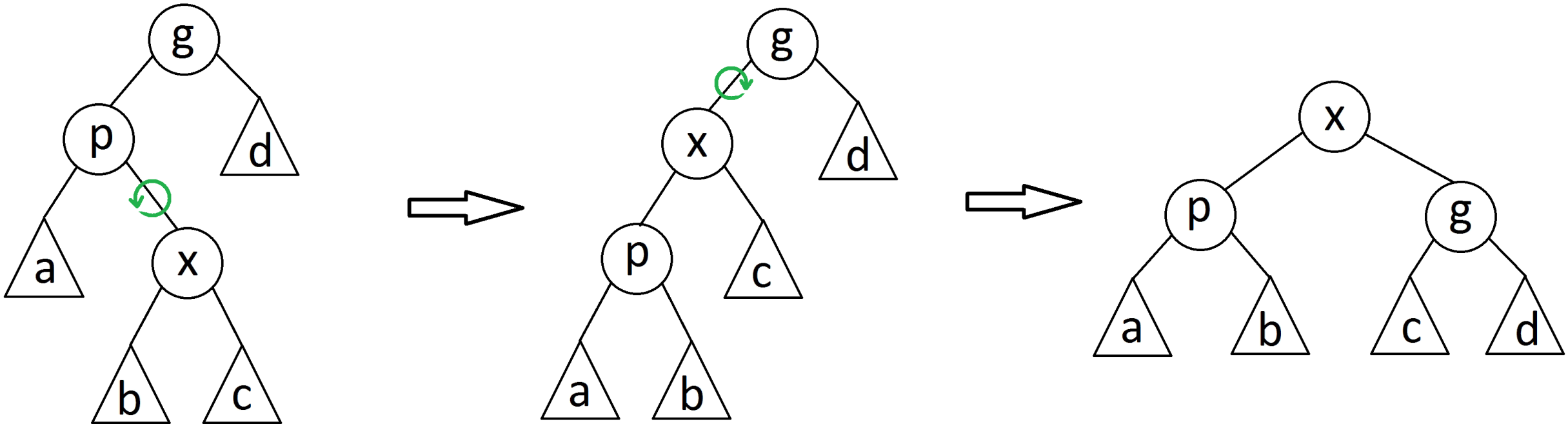
Делается, если узел и родитель – оба правые или левые дети. Делаем два малых поворота.



Сплей-дерево

ZigZag

Узел и предок разные дети своих родителей. Тоже два поворота.



Сплей-дерево. Операции

Вставка: вставляем элемент, как в обычном дерево поиска. После это вызываем **Splay** от вставленного элемента

Поиск: как в обычном дереве поиска. Найденному элементу делаем **Splay**. Возвращаем в ответе корень. Если элемент не найден, все равно делаем **Splay**, тому листу, до которого дошли.

Разделение (Split): Находим элемент, по которому нужно сделать разделение.Вызываем **Splay.** В левом поддереве будут все ключи меньшие, справа – большие.

Слияние (Merge): находим самый большой элемент в дереве, в котором ключи меньше. Делаем для него **Splay**. Очевидно, в результате получим дерево без правого поддерева, присоединяем к нему второе дерево

Удаление: делаем **Splay** от удаляемого элемента. Потом **Merge** его поддеревьев.

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).

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.

???

Спасибо за внимание.

Любимов Яков