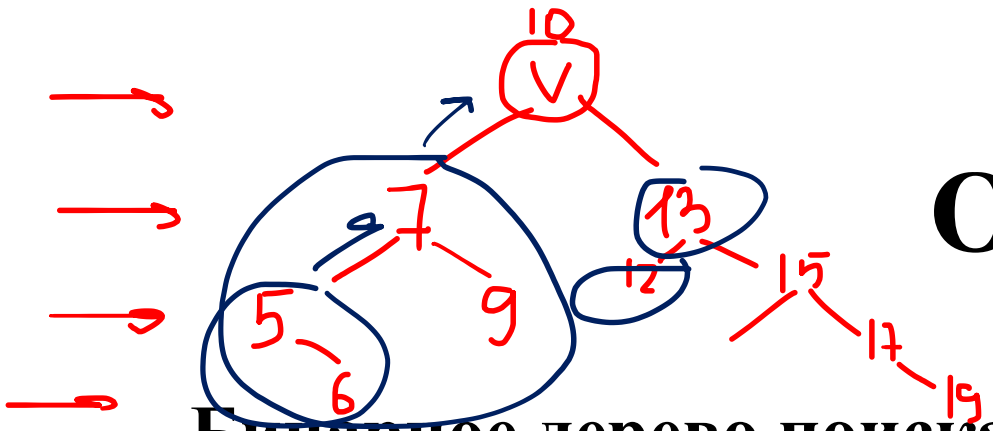
Бинарные деревья поиска. Сбалансированные деревья. Декартово дерево



Определение

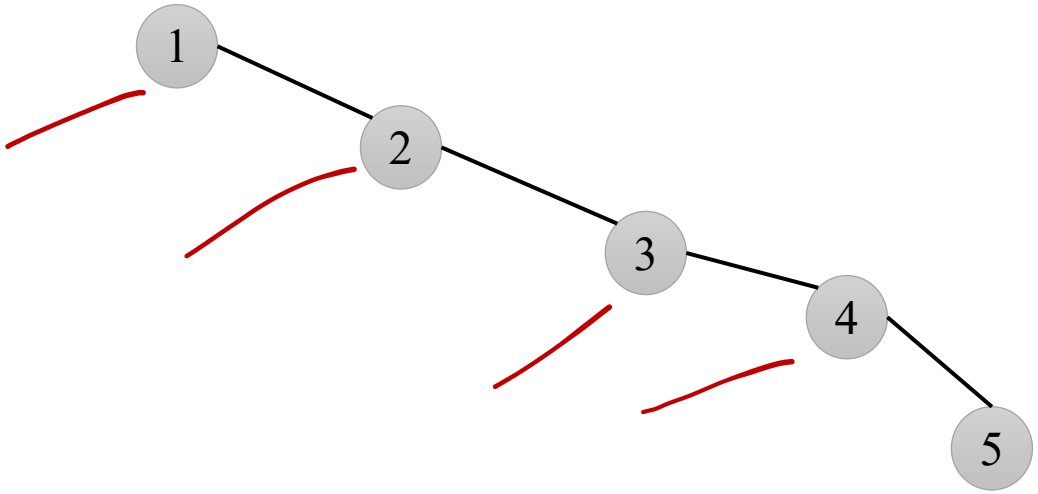
Бинарное дерево поиска – это структура данных, в которой каждый узел имеет не более двух потомков (детей), при этом выполняются следующие дополнительные условия (свойства дерева поиска):

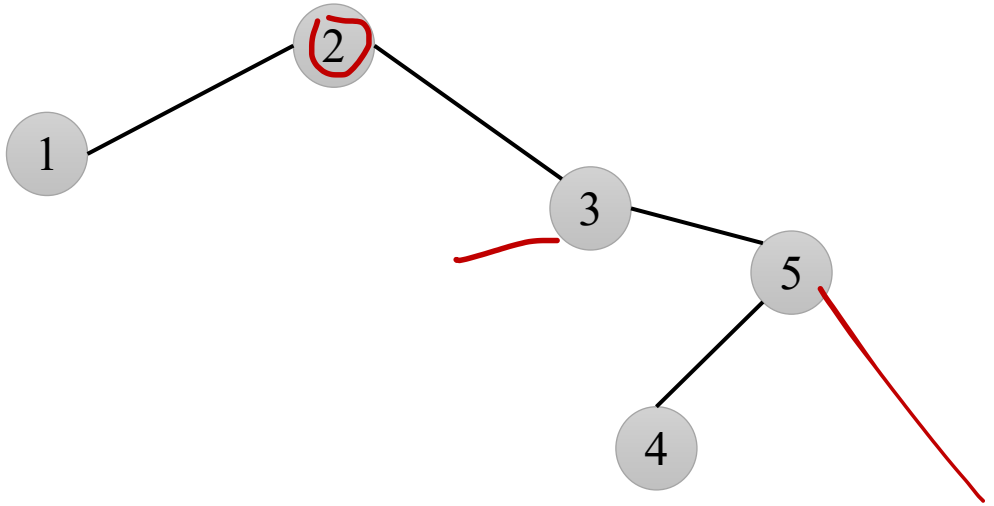
- оба поддерева — левое и правое — являются двоичными деревьями поиска;
- левый сын произвольного узла имеет ключ меньше, чем данный узел;
- правый сын произвольного узла имеет ключ больше, чем данный узел.

Обычно в бинарных деревьях поиска нет элементов с одинаковыми ключами, но если это условие не выполнено, то можно, например, в каждом узле хранить количество элементов с таким ключом.

Соответственно ключами в бинарном дереве могут быть любые элементы, сравнимые на больше/меньше.

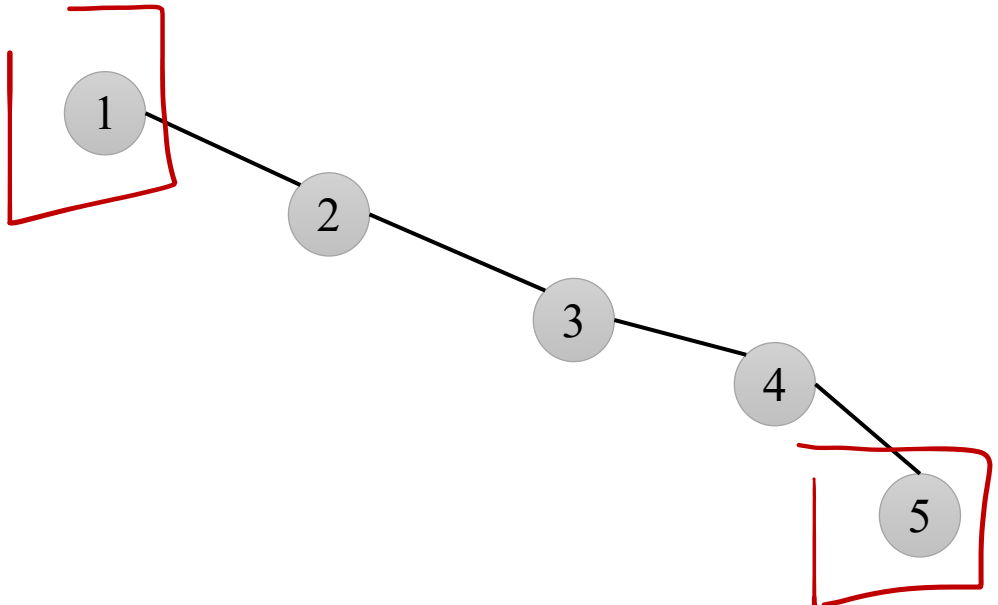
Ниже приведены образцы двух бинарных деревьев поиска. Несмотря на то, что ключи у них идентичны – деревья имеют разный вид, так как **порядок добавления элементов** в дерево отличался (хотя и для разных порядков могут получиться одинаковые деревья). В первом случае был взят отсортированный массив, во втором – те же элементы в произвольном порядке.

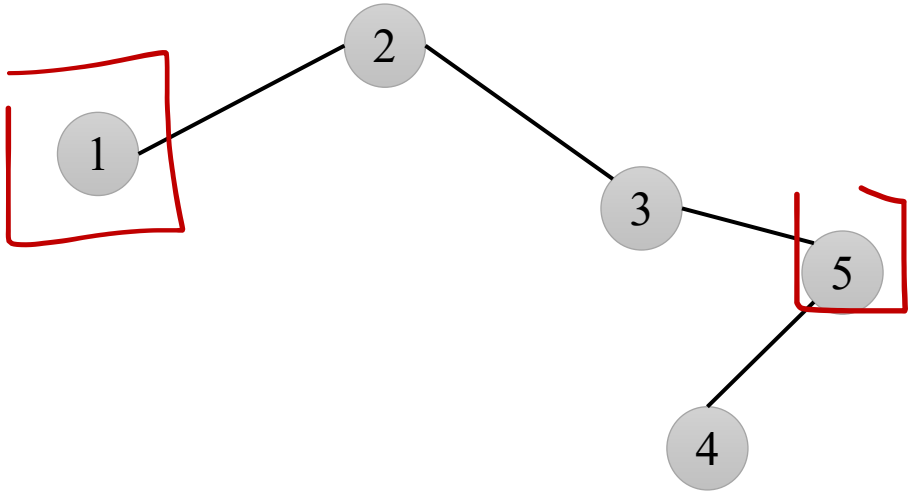




Пусть ***H*** – максимальная глубина дерева. Для случайного порядка добавления ***H*** = *O(log N)*, где N – количество элементов. В среднем бинарное дерево поиска позволяет искать элемент за *O(H)*, как и делать вставку за аналогичное время. При этом для отсортированного массива это будет занимать *O(N)* времени, как видно из первого примера.

В бинарном дереве поиска легко найти минимальный и максимальный элементы: это самый левый и самый правый соответственно. Для их нахождения нужно идти «до упора» влево и вправо, начиная от корня.

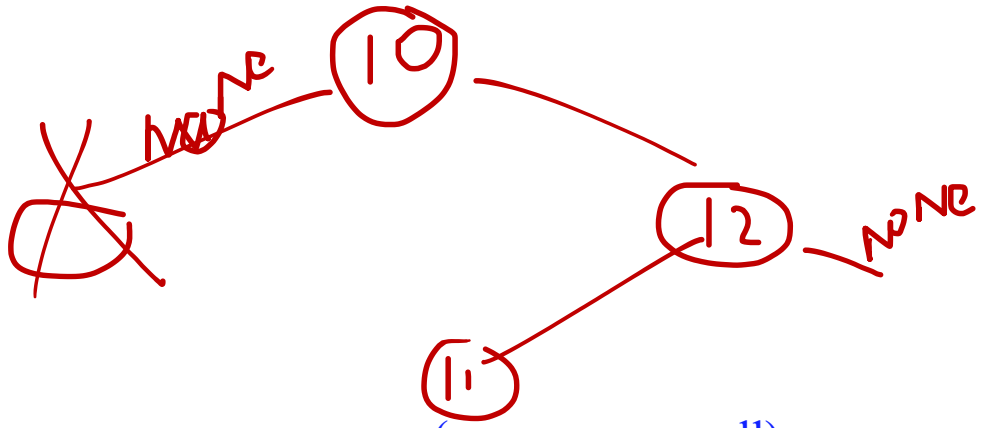




Представление

Для хранения узла дерева необходима структура с полем для ключа, а также с двумя ссылками на левого и правого сыновей.

Например,

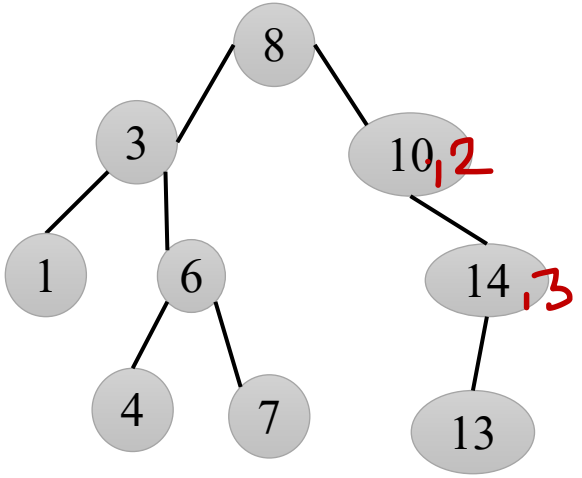



```
struct Node:
    T key
    Node left
    Node right
// иногда имеет смысл сохранять также родителя вершины (у корня это null)
```

Операции над деревом

Пусть у нас уже построено дерево, представленное на рисунке ниже. Базовый интерфейс дерева поиска состоит из операций find, insert, remove, однако мы рассмотрим следующие действия:

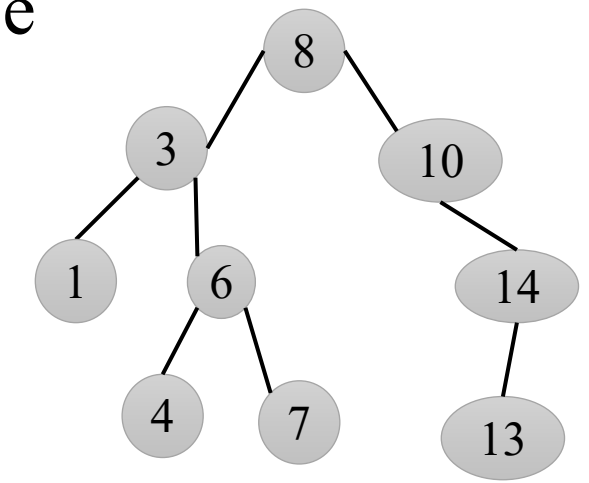
- **обход дерева поиска**
- **поиск элемента (find)**
- **поиск минимума и максимума**
- **вставка (insert)**
- **удаление (remove)**



Обход дерева

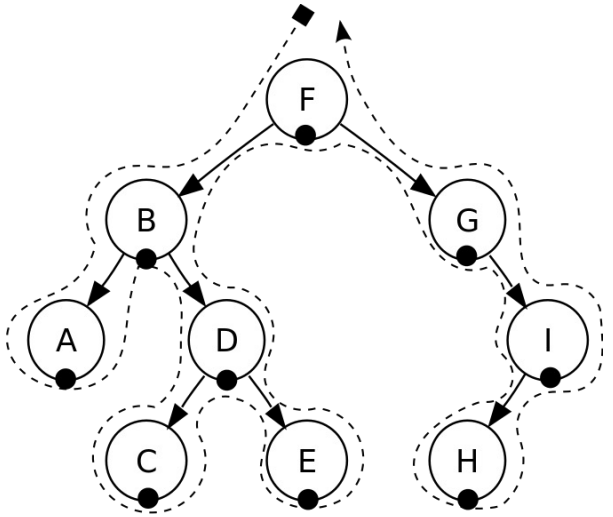
Существуют три различных способа обхода узлов дерева:

- ***inorderTraversal*** – обход узлов в отсортированном порядке
- ***preorderTraversal*** – обход узлов в порядке: вершина, левое поддерево, правое поддерево
- ***postorderTraversal*** – обход узлов в порядке: левое поддерево, правое поддерево, вершина.



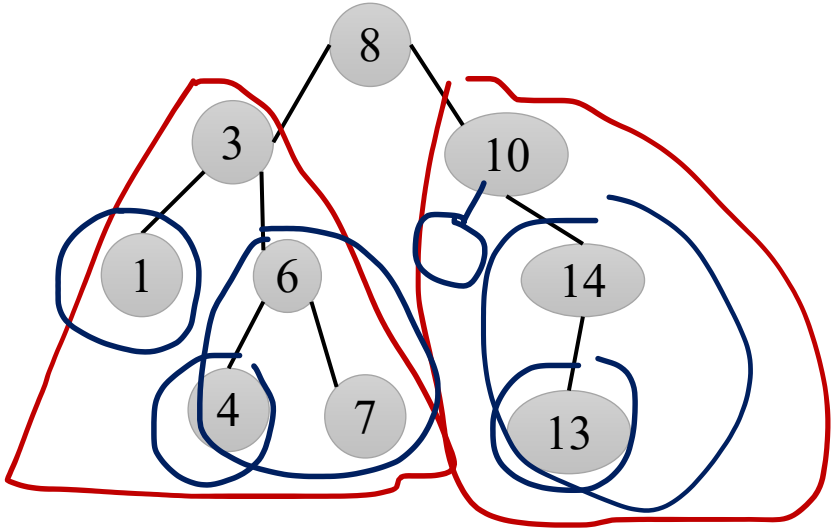
Центрированный (inorder) обход

1. Проверяем, не является ли узел пустым/null
2. Обходим левое поддерево рекурсивно, вызвав функцию обхода.
3. Показываем поле данных корня (или текущего узла).
4. Обходим правое поддерево рекурсивно, вызвав функцию обхода.




```
func inorderTraversal(Node x):
  if x != null
    inorderTraversal(x.left)
    print(x.key)
    inorderTraversal(x.right)
```

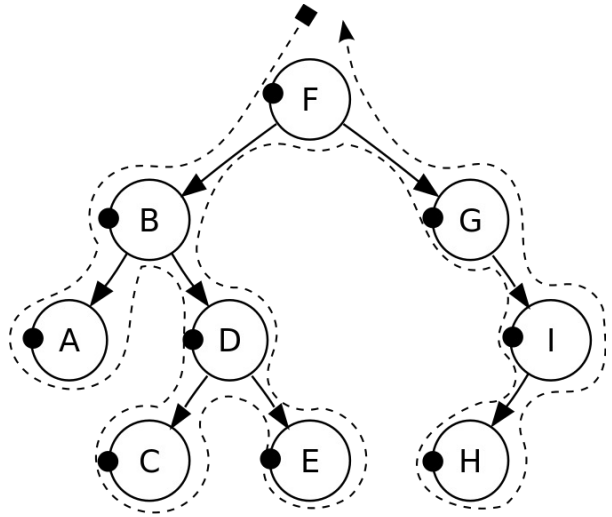
1 3 4 6 7 8 10 13 14



Данный обход в двоичном дереве поиска извлечет данные в отсортированном порядке: **1 3 4 6 7 8 10 13 14**

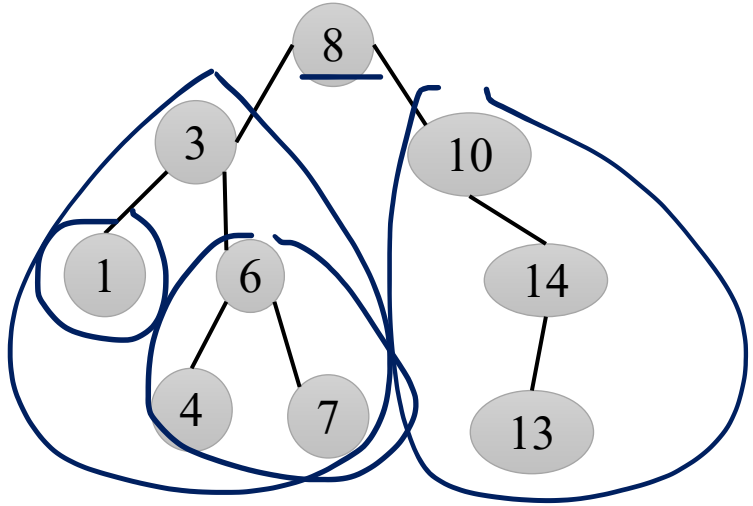
Прямой (preorder) обход

1. Проверяем, не является ли узел пустым/null
2. Показываем поле данных корня (или текущего узла).
3. Обходим левое поддерево рекурсивно, вызвав функцию обхода.
4. Обходим правое поддерево рекурсивно, вызвав функцию обхода.




```
func preorderTraversal(Node x):
    if x != null
        print(x.key)
        preorderTraversal(x.left)
        preorderTraversal(x.right)
```

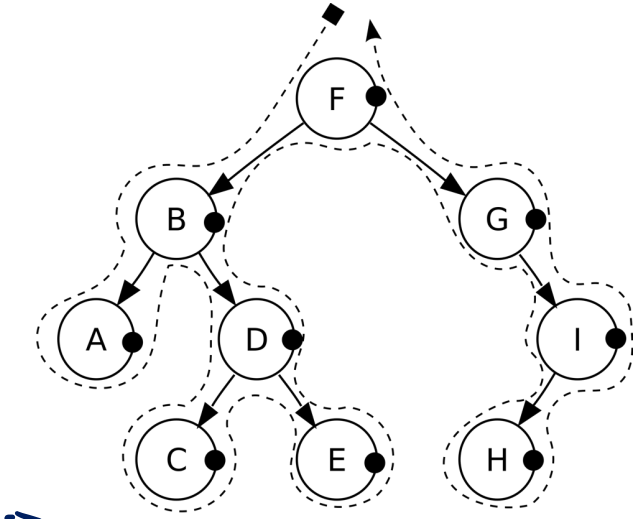
8 3 1 6 4 7 10 14 13



Данный обход в двоичном дереве поиска извлечет данные в следующем порядке: **8 3 1 6 4 7 10 14 13**

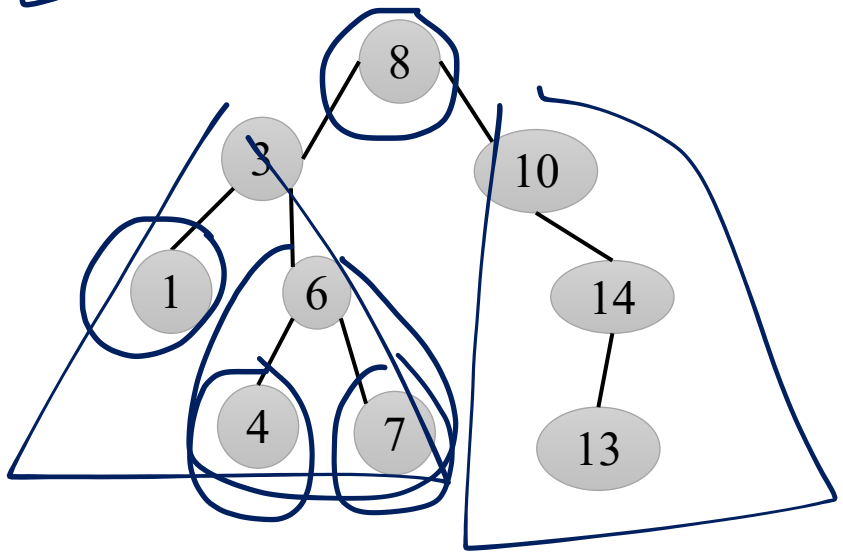
Обратный (postorder) обход

1. Проверяем, не является ли узел пустым/null
2. Обходим левое поддерево рекурсивно, вызвав функцию обхода.
3. Обходим правое поддерево рекурсивно, вызвав функцию обхода.
4. Показываем поле данных корня (или текущего узла).




```
func postorderTraversal(Node x):
    if x != null
        preorderTraversal(x.left)
        preorderTraversal(x.right)
        print(x.key)
```

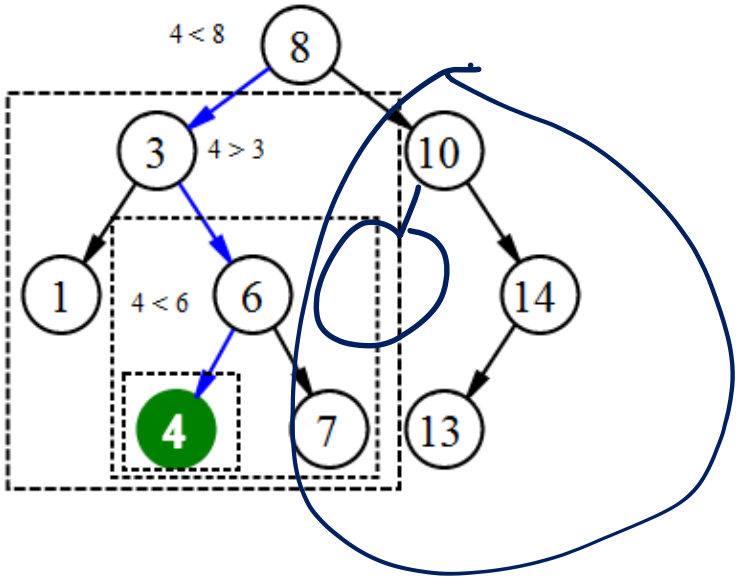
1 4 7 6 3 13 14 10 8



Данный обход в двоичном дереве поиска извлечет данные в следующем порядке: **1 4 7 6 3 13 14 10 8**

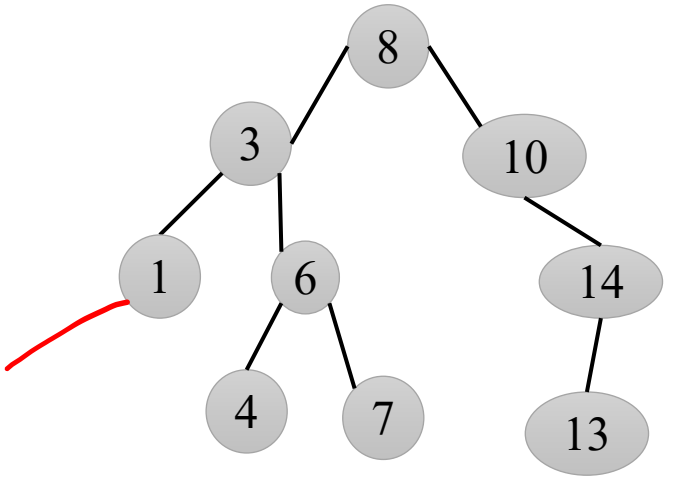
Поиск элемента

Поиск элемента 4



Поиск элемента в дереве осуществляется по ключу. Функция принимает на вход в качестве параметров корень дерева и искомый ключ. Для каждого узла функция сравнивает значение его ключа с искомым ключом. Если ключи одинаковы, то функция возвращает текущий узел, в противном случае функция вызывается рекурсивно для левого или правого поддерева. Узлы, которые посещает функция, образуют нисходящий путь от корня, так что время ее работы $O(H)$.

```
// изначально в качестве x передаем корень
Node search(Node x, T k):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```



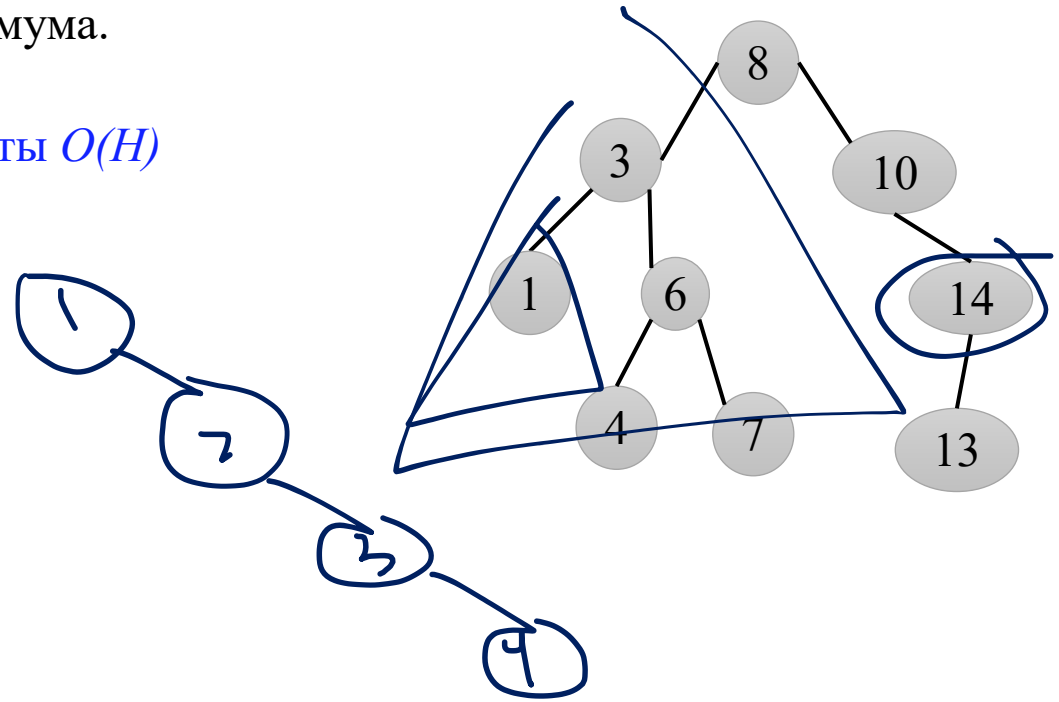
Поиск минимума и максимума

Как уже было сказано ранее, в бинарном дереве поиска минимальный и максимальный элементы – это самый левый и самый правый соответственно. Для их нахождения нужно идти «до упора» влево и вправо, начиная от корня. То есть необходимо следовать указателям ***left***, пока не встретишь ***null*** – для минимума, аналогично для максимума.

Минимум: 1

Максимум: 14

```
// изначально в качестве x передаем корень, время работы O(H)
Node minimum(Node x):
    if x.left == null
        return x
    return minimum(x.left)
Node maximum(Node x):
    if x.right == null
        return x
    return maximum(x.right)
```



Вставка

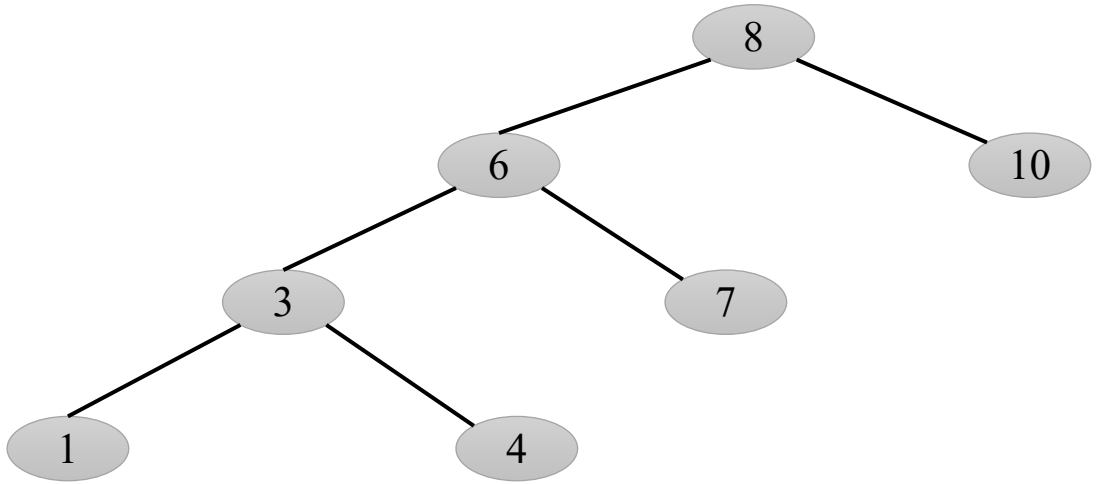
Операция вставки работает аналогично поиску элемента, только при обнаружении у элемента отсутствия ребенка нужно подвесить на него вставляемый элемент. Время работы такого алгоритма будет также *O(H)*.

```
// x – корень поддерева, z – вставляемый ключ
Node insert(Node x, T z):
    if x == null
        return Node(z)                    // подвесим Node с key = z
    else if z < x.key
        x.left = insert(x.left, z)
    else if z > x.key
        x.right = insert(x.right, z)
    return x
```

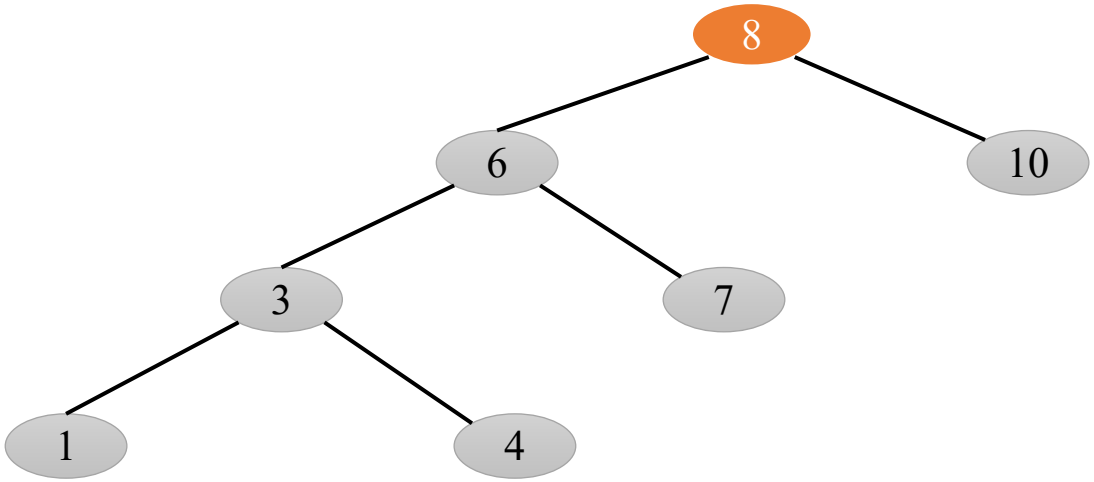
Пример вставки

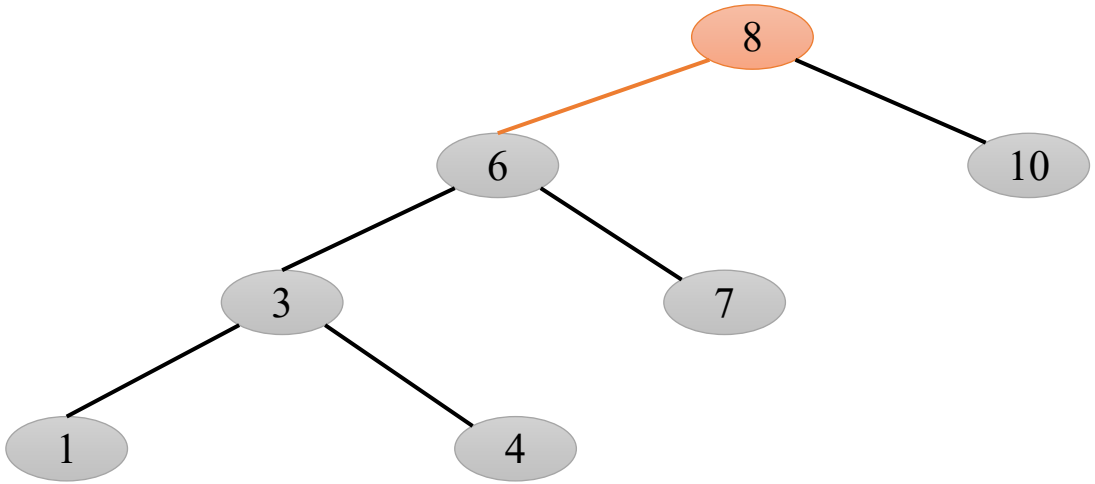
Дано: бинарное дерево поиска

Надо: вставить в него элемент с ключом ***5***

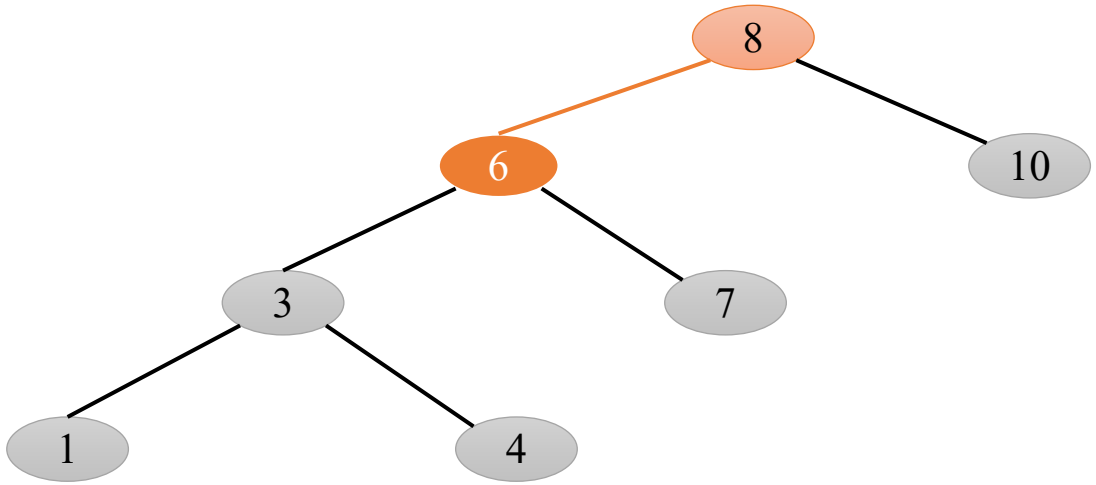


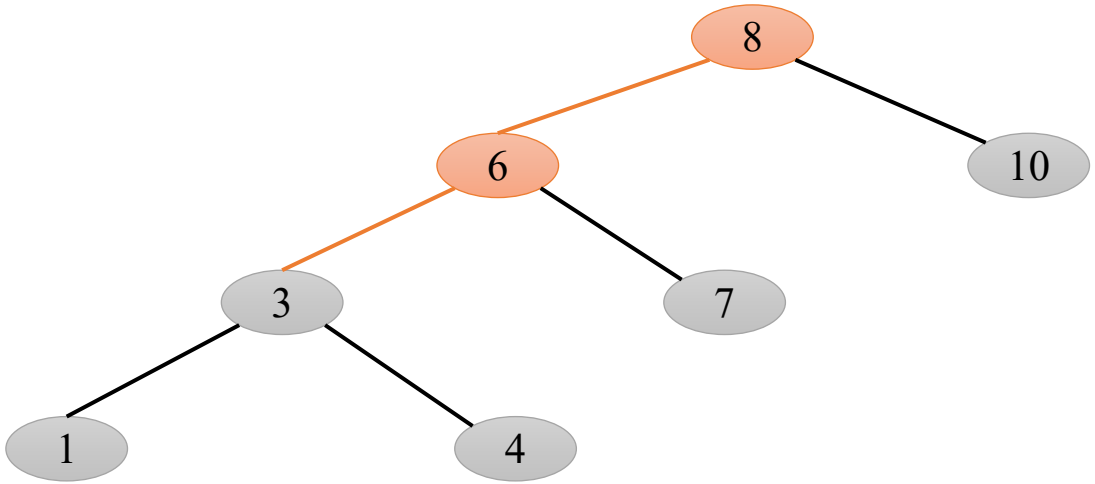
```
// x = 8, T = 5
Node insert(Node x, T z):
    if x == null                        // ложь
        return Node(z)
    else if z < x.key                   // истина, идем в левого ребенка
        x.left = insert(x.left, z)
    else if z > x.key
        x.right = insert(x.right, z)
    return x
```



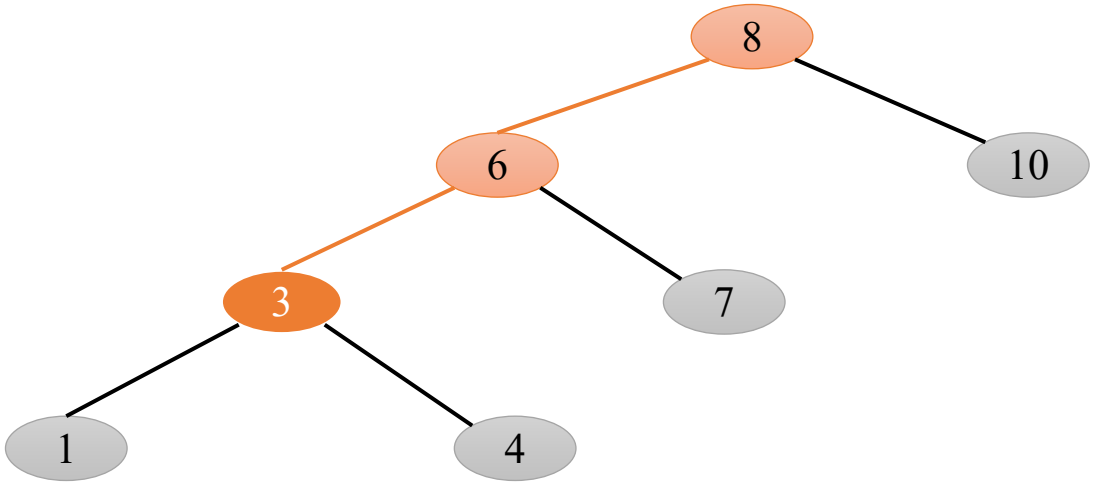


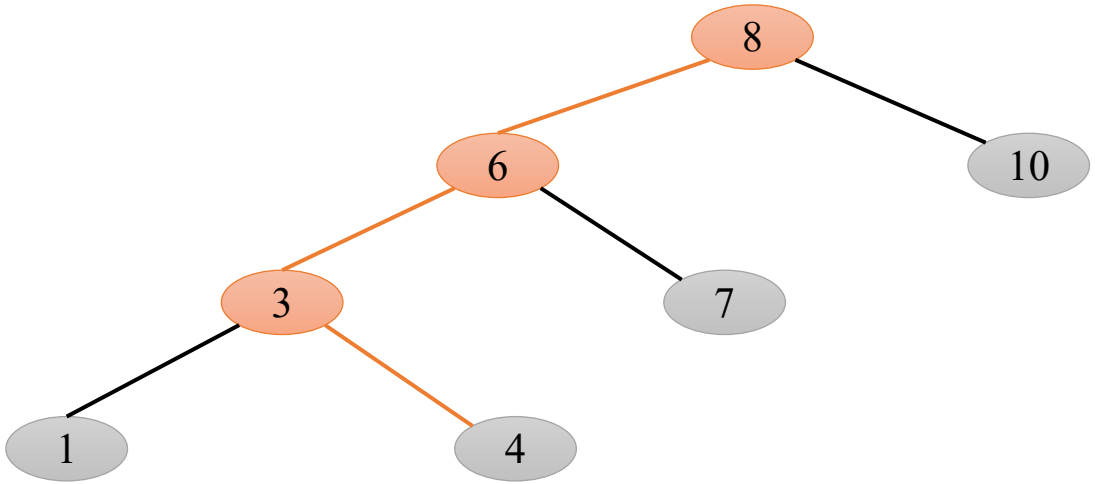
```
// x = 6, T = 5
Node insert(Node x, T z):
    if x == null                        // ложь
        return Node(z)
    else if z < x.key                   // истина, идем в левого ребенка
        x.left = insert(x.left, z)
    else if z > x.key
        x.right = insert(x.right, z)
    return x
```



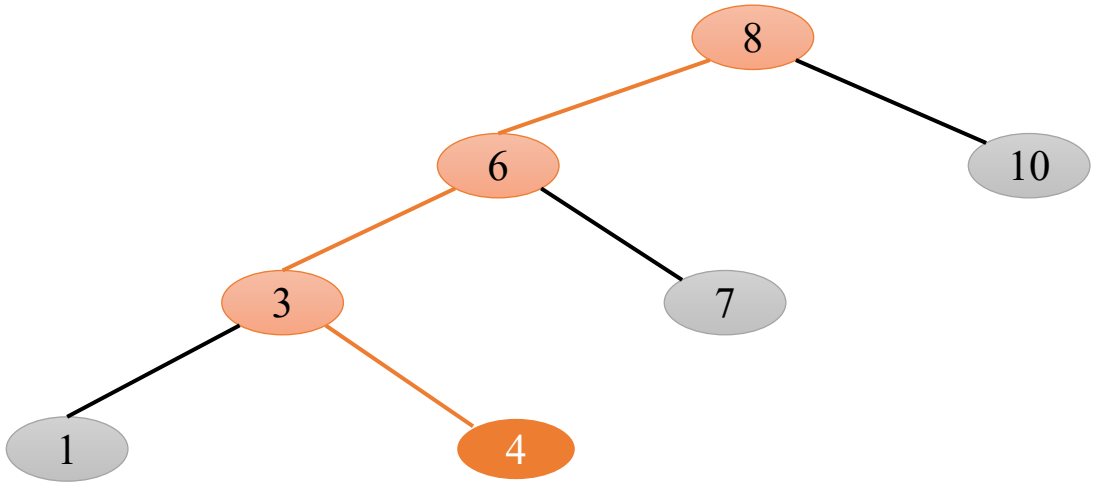


```
// x = 3, T = 5
Node insert(Node x, T z):
    if x == null                    // ложь
        return Node(z)
    else if z < x.key               // ложь
        x.left = insert(x.left, z)
    else if z > x.key               // истина, идем в правого ребенка
        x.right = insert(x.right, z)
    return x
```

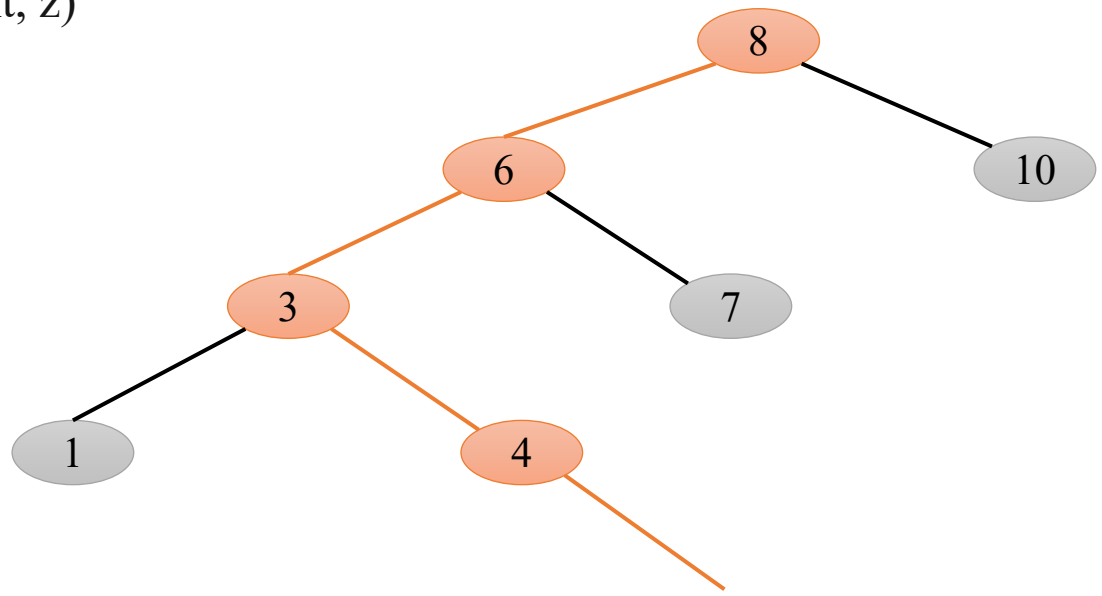




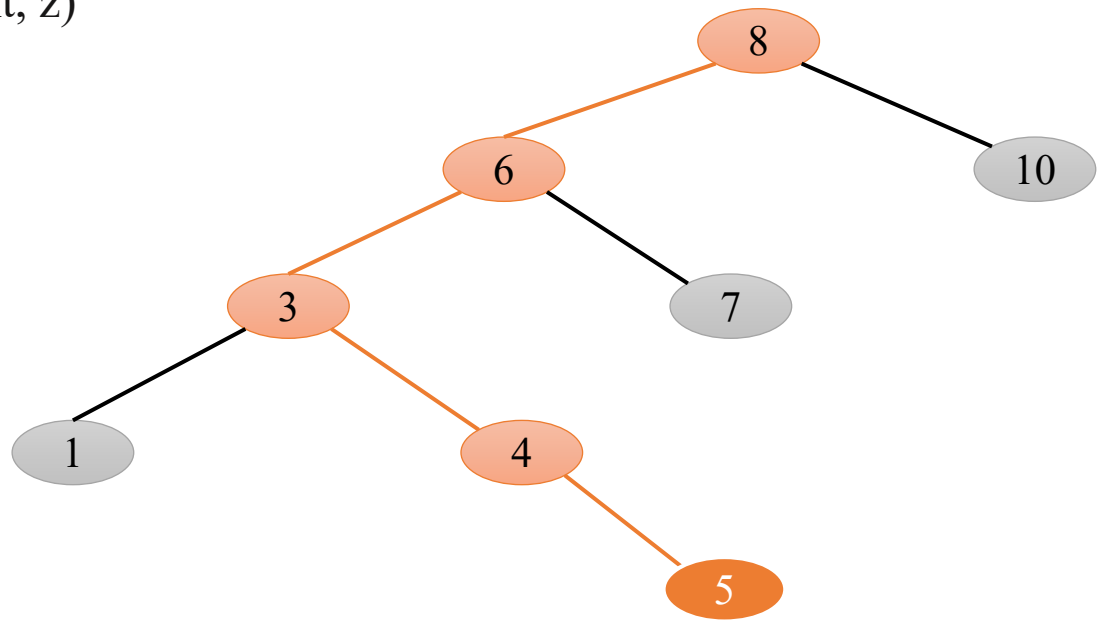
```
// x = 4, T = 5
Node insert(Node x, T z):
    if x == null                        // ложь
        return Node(z)
    else if z < x.key                   // истина, идем в правого ребенка
        x.left = insert(x.left, z)
    else if z > x.key
        x.right = insert(x.right, z)
    return x
```



```
// x = ?, T = 5
Node insert(Node x, T z):
    if x == null                          // истина, необходимо подвесить вершину
        return Node(z)
    else if z < x.key
        x.left = insert(x.left, z)
    else if z > x.key
        x.right = insert(x.right, z)
    return x
```



```
// Подвесили вершину, рекурсия завершает работу
Node insert(Node x, T z):
    if x == null
        return Node(z)
    else if z < x.key
        x.left = insert(x.left, z)
    else if z > x.key
        x.right = insert(x.right, z)
    return x
```

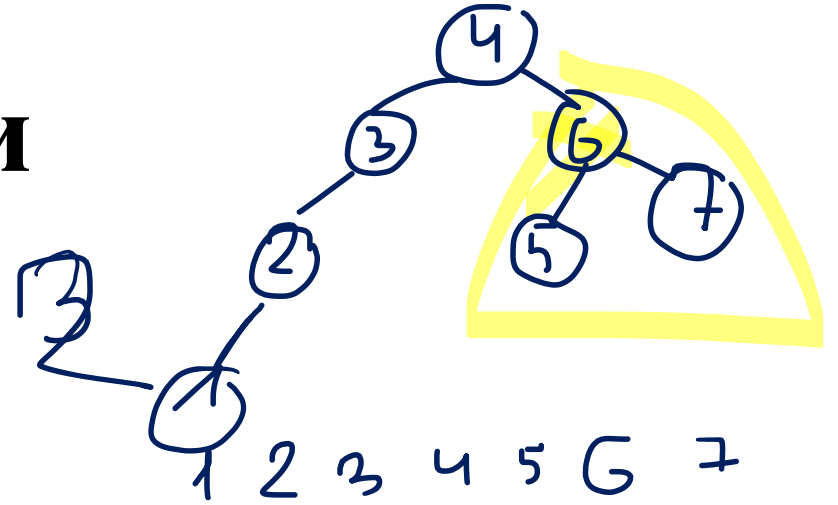


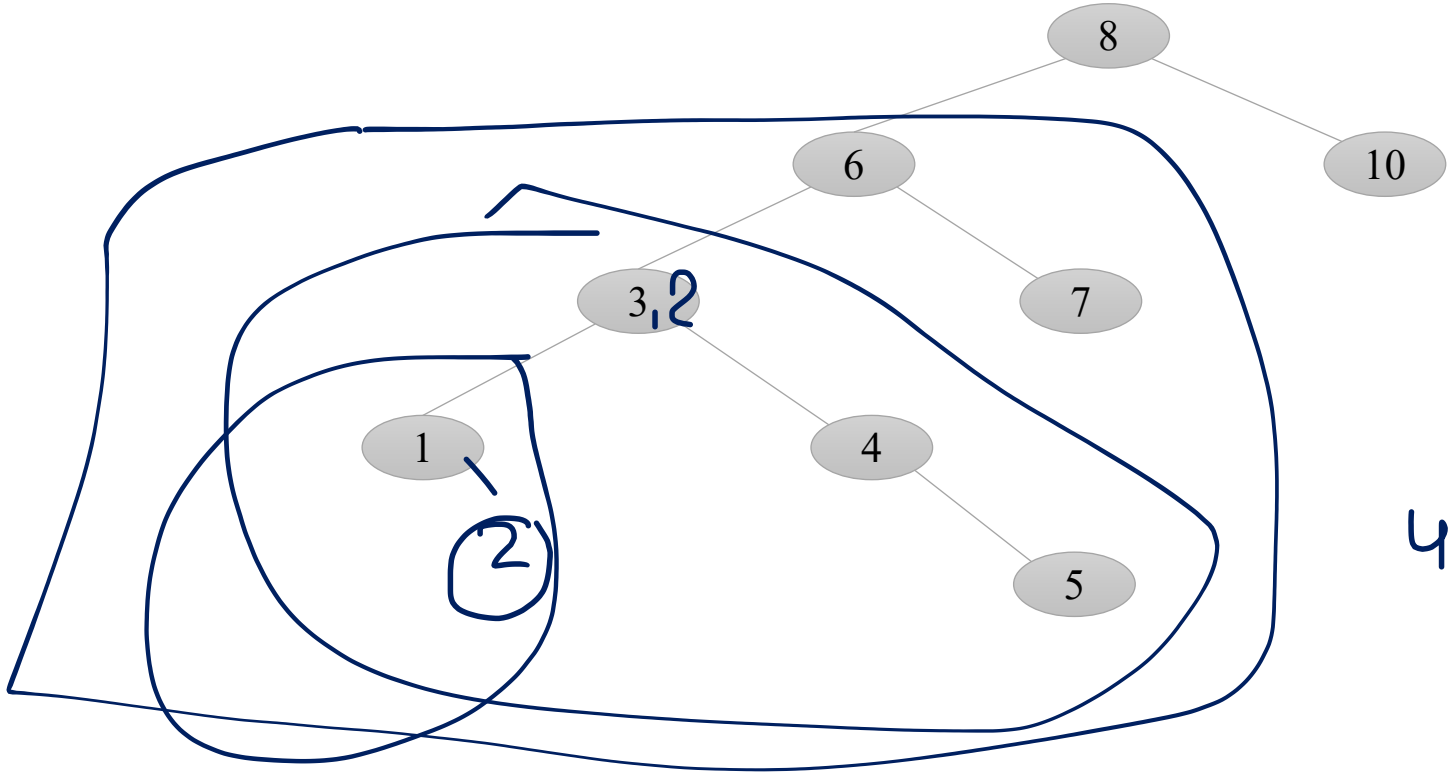
Пример вставки

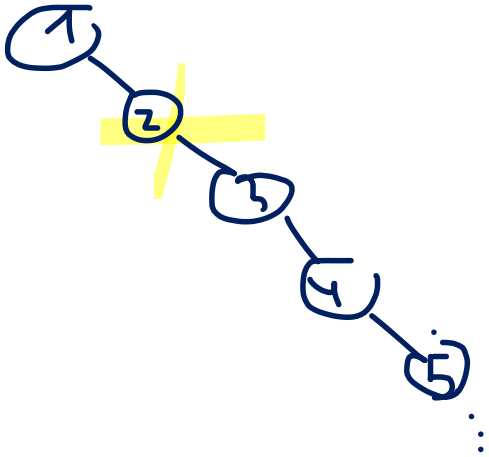
Дано: бинарное дерево поиска

Надо: вставить в него элемент с ключом ***5***

Вывод:





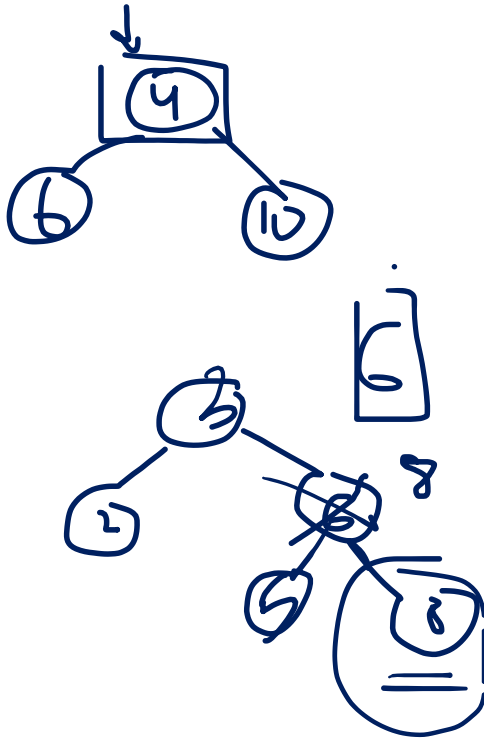


4637521

Удаление

Рассмотрим рекурсивное удаление элемента по ключу. нужно рассмотреть три случая: удаляемый элемент находится в левом или правом поддереве текущего поддерева или удаляемый элемент находится в корне. В двух первых случаях нужно рекурсивно удалить элемент из нужного поддерева. Если удаляемый элемент находится в корне текущего поддерева и имеет два дочерних узла, то нужно заменить его минимальным элементом из правого поддерева и рекурсивно удалить этот минимальный элемент из правого поддерева. Иначе, если удаляемый элемент имеет один дочерний узел, нужно заменить его потомком. Время работы алгоритма – $O(H)$.

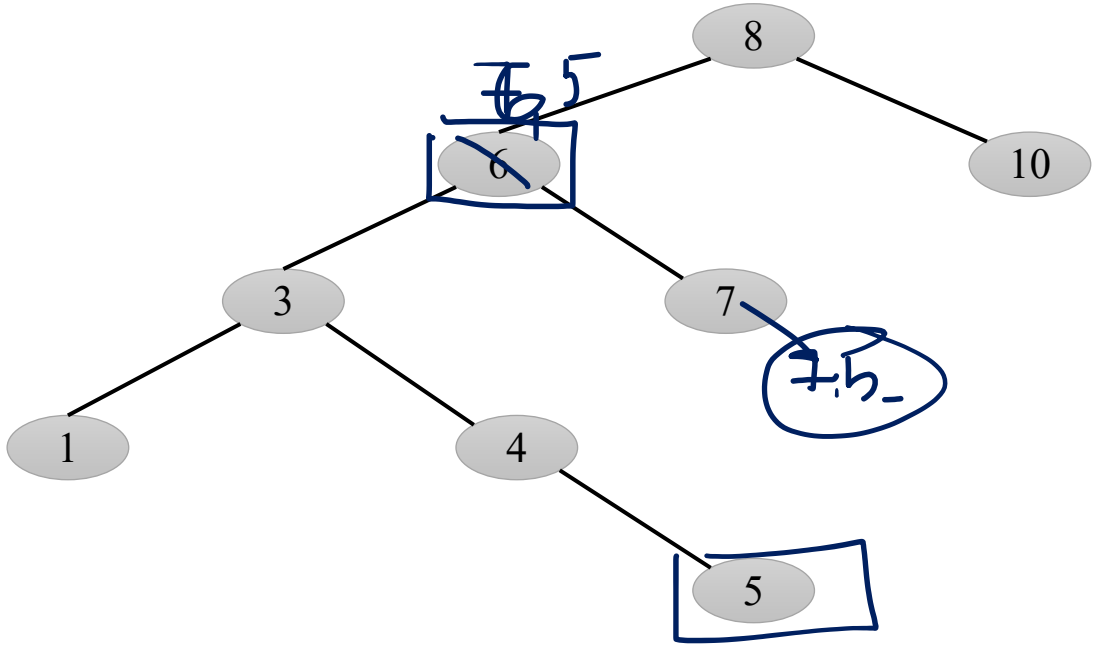
```
// x – корень поддерева, z – удаляемый ключ
Node delete(Node root, T z):
    if root == null
        return root
    if z < root.key
        root.left = delete(root.left, z)
    else if z > root.key
        root.right = delete(root.right, z)
    else if root.left != null and root.right != null
        root.key = min(root.right).key
        root.right = delete(root.right, root.key)
    else
        if root.left != null
            root = root.left
        else if root.right != null
            root = root.right
        else
            root = null
    return root
```



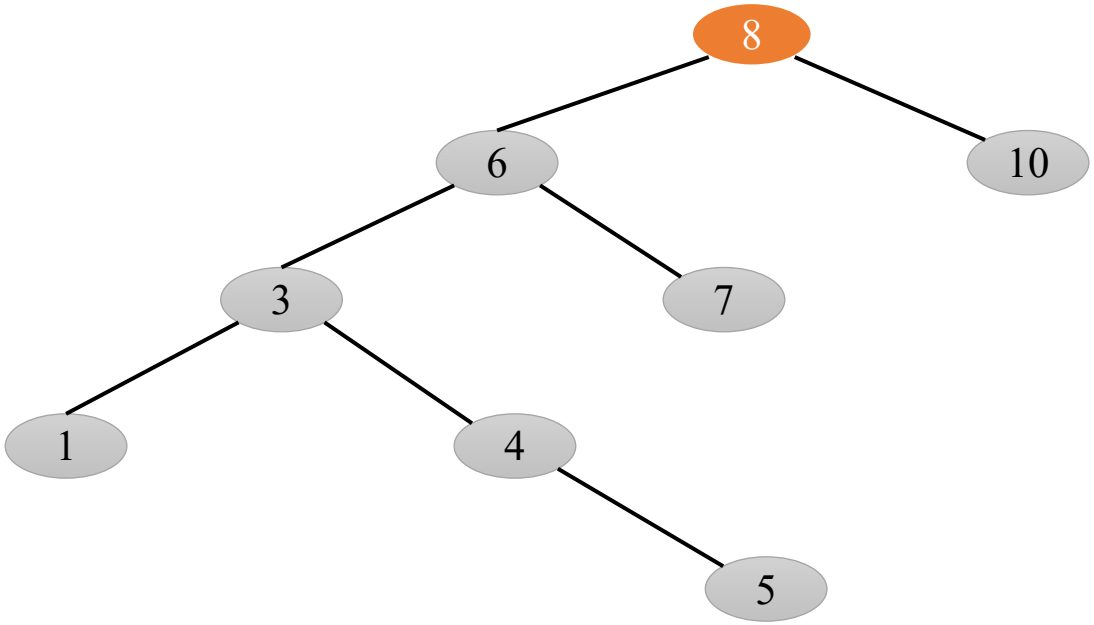
Пример удаления

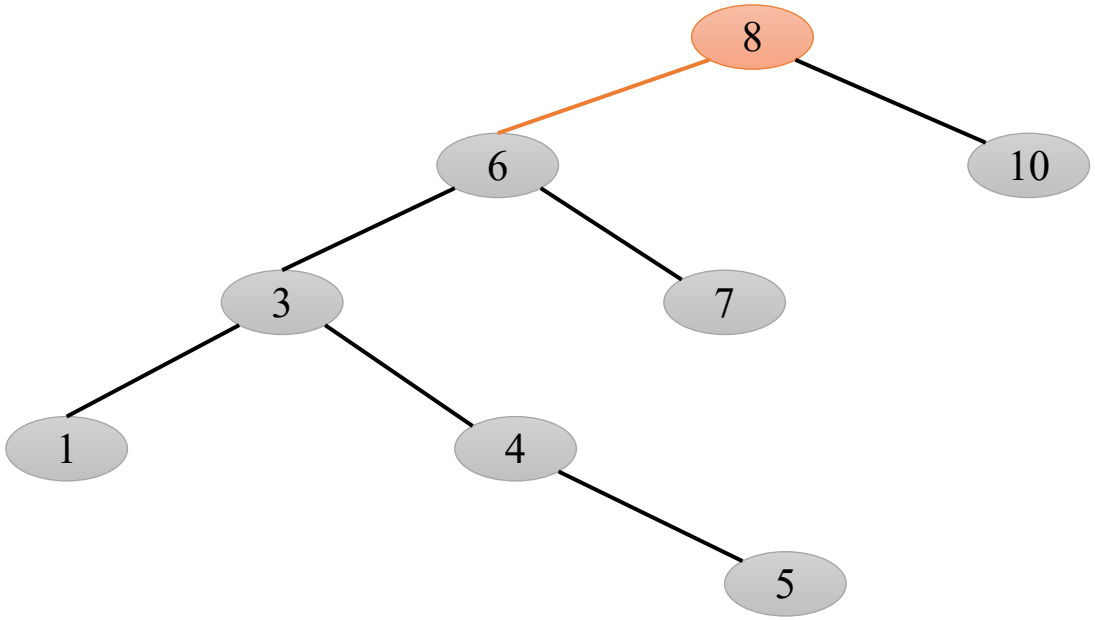
Дано: бинарное дерево поиска

Надо: удалить из него элемент с ключом ***3***

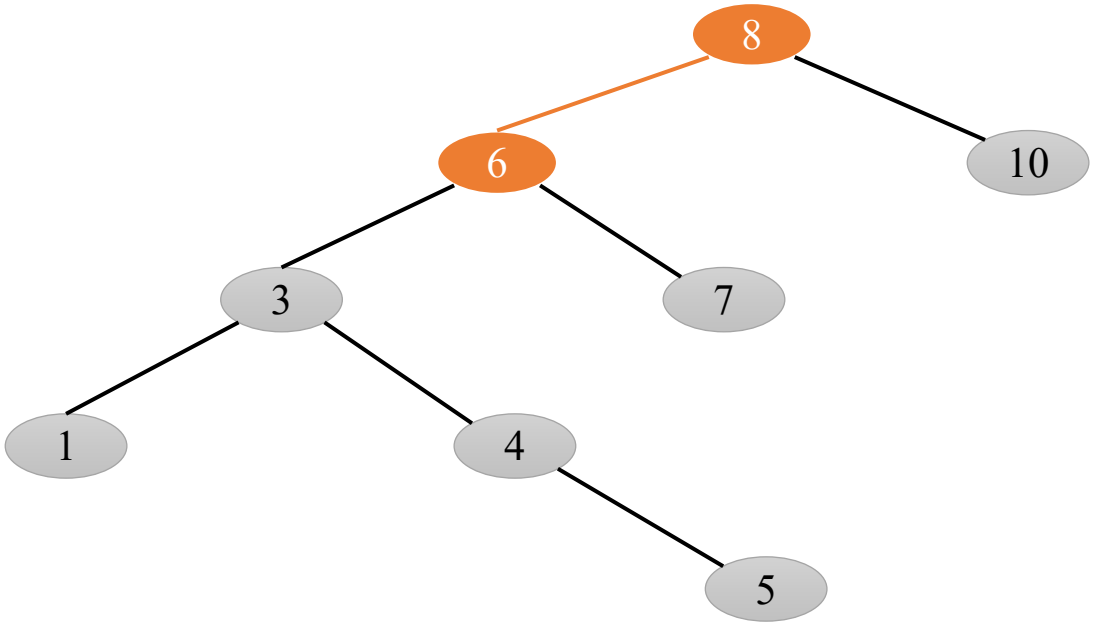


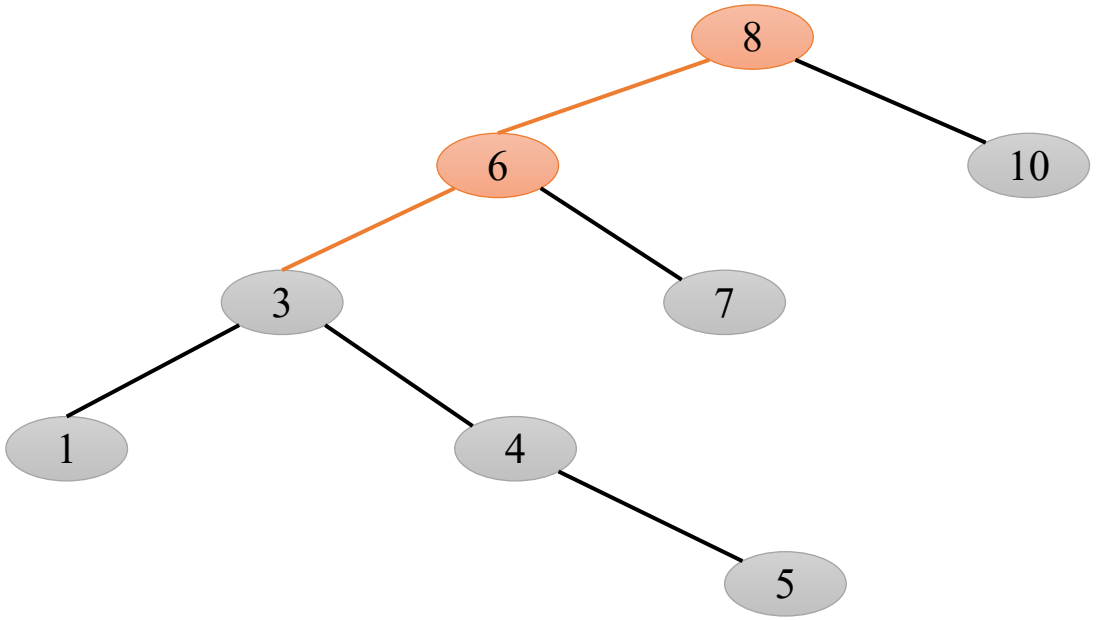
```
// root =  8, z = 3
Node delete(Node root, T z):
    if root == null          // ложь
        return root
    if z < root.key
        root.left = delete(root.left, z)
    else if z > root.key
        root.right = delete(root.right, z)
    else if root.left != null and root.right != null
        root.key = min(root.right).key
        root.right = delete(root.right, root.key)
    else
        if root.left != null
            root = root.left
        else if root.right != null
            root = root.right
                    else
            root = null
    return root
```



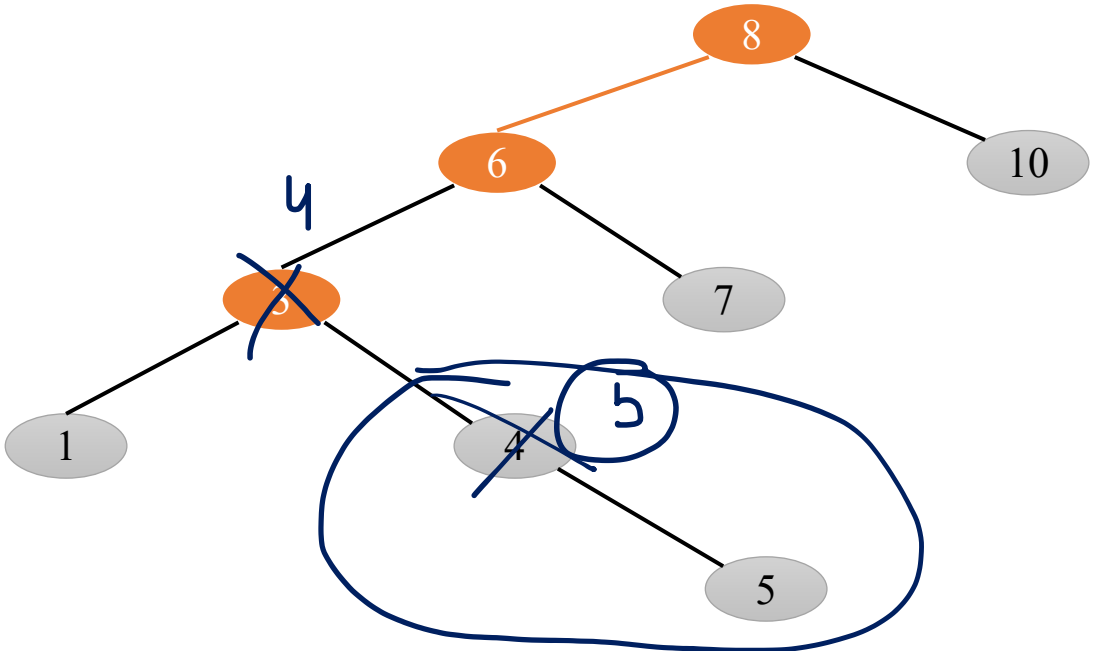


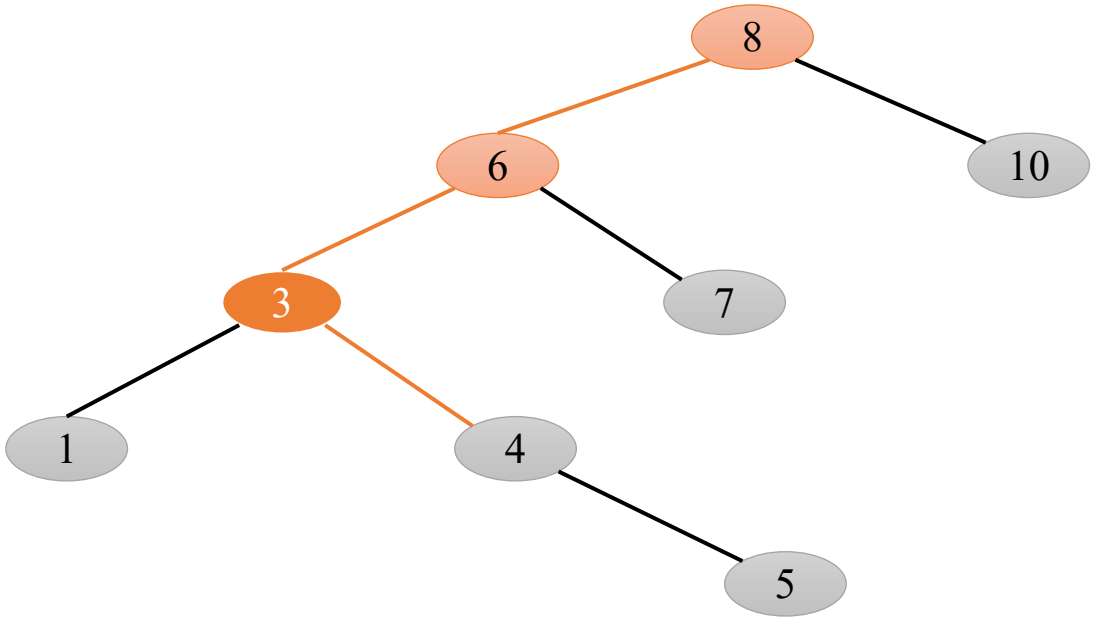
```
// root =  6, z = 3
Node delete(Node root, T z):
    if root == null          // ложь
        return root
    if z < root.key
        root.left = delete(root.left, z)
    else if z > root.key
        root.right = delete(root.right, z)
    else if root.left != null and root.right != null
        root.key = min(root.right).key
        root.right = delete(root.right, root.key)
    else
        if root.left != null
            root = root.left
        else if root.right != null
            root = root.right
                    else
            root = null
    return root
```



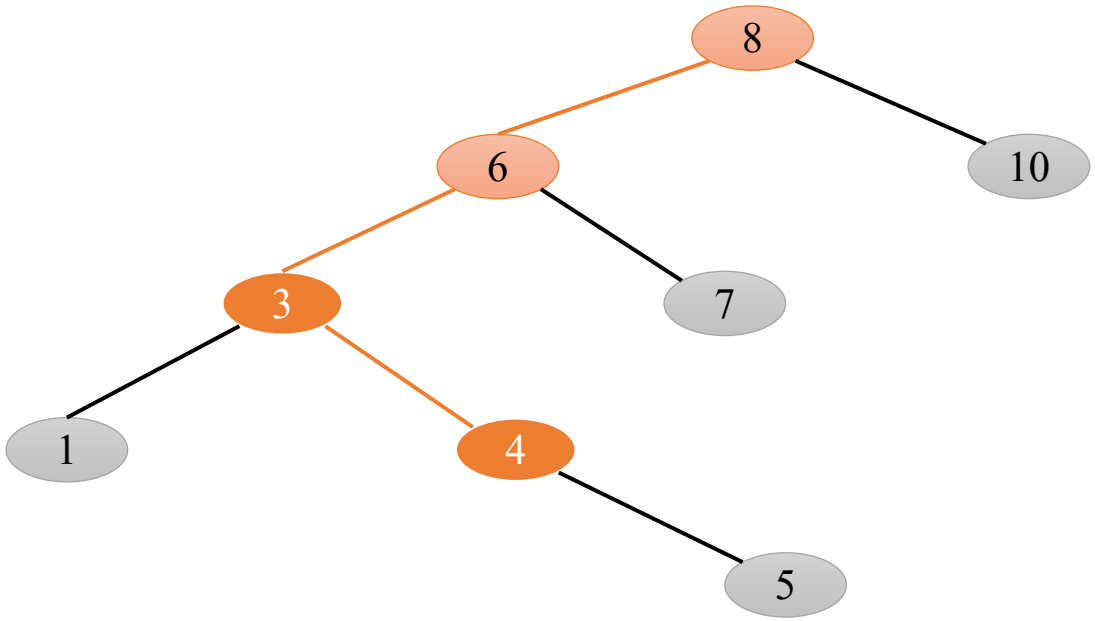


```
// root =  3, z = 3
Node delete(Node root, T z):
    if root == null          // ложь
        return root
    if z < root.key
        root.left = delete(root.left, z)
    else if z > root.key
        root.right = delete(root.right, z)
    else if root.left != null and root.right != null
        root.key = min(root.right).key
        root.right = delete(root.right, root.key)
    else
        if root.left != null
            root = root.left
        else if root.right != null
            root = root.right
                    else
            root = null
    return root
```



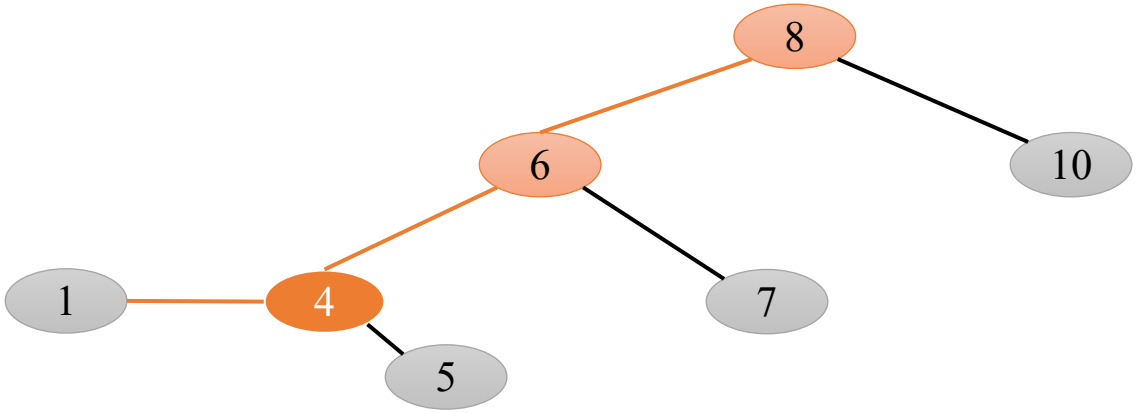


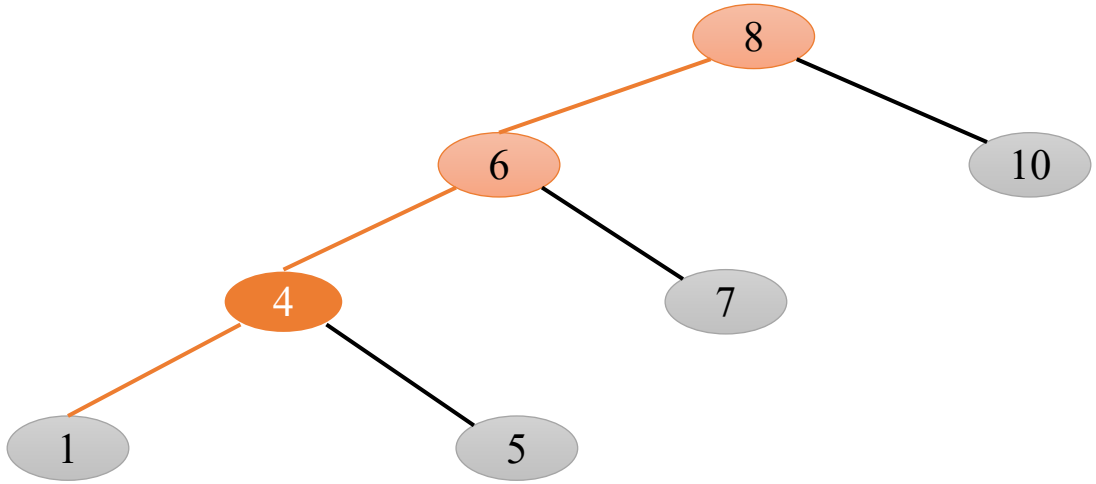
```
// root =  4, z = 3
Node delete(Node root, T z):
    if root == null          // ложь
        return root
    if z < root.key
        root.left = delete(root.left, z)
    else if z > root.key
        root.right = delete(root.right, z)
    else if root.left != null and root.right != null
        root.key = min(root.right).key
        root.right = delete(root.right, root.key)
    else
        if root.left != null
            root = root.left
        else if root.right != null
            root = root.right
                    else
            root = null
    return root
```



// root = ?, z = 3 – рекурсия завершается, раннее мы поменяли ключ ребенка вершины 6 – теперь он равен 4 – переподвесили

```
Node delete(Node root, T z):
    if root == null          // истина
        return root
    if z < root.key
        root.left = delete(root.left, z)
    else if z > root.key
        root.right = delete(root.right, z)
    else if root.left != null and root.right != null
        root.key = min(root.right).key
        root.right = delete(root.right, root.key)
    else
        if root.left != null
            root = root.left
        else if root.right != null
            root = root.right
                    else
            root = null
    return root
```



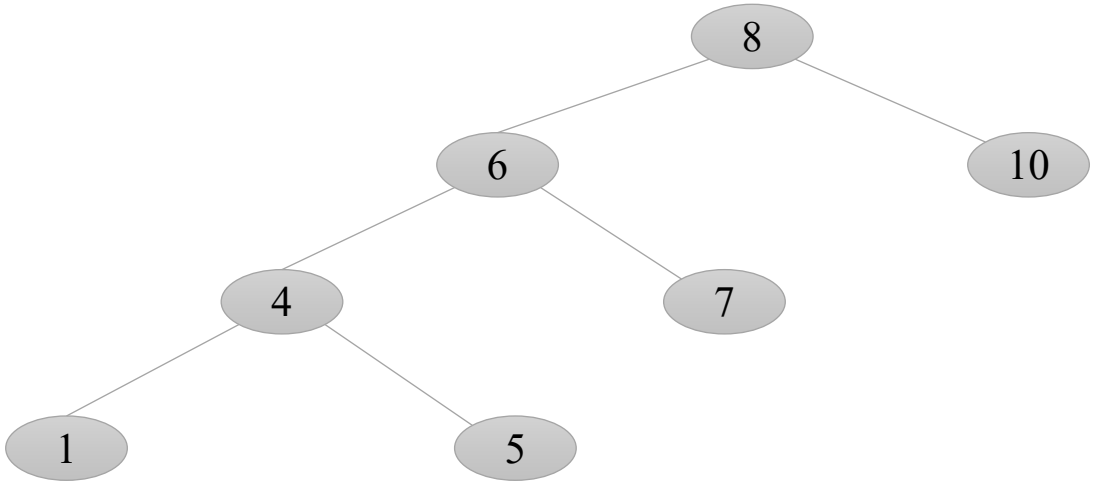


Пример удаления

Дано: бинарное дерево поиска

Надо: удалить из него элемент с ключом ***3***

Вывод:



Вычислительная сложность

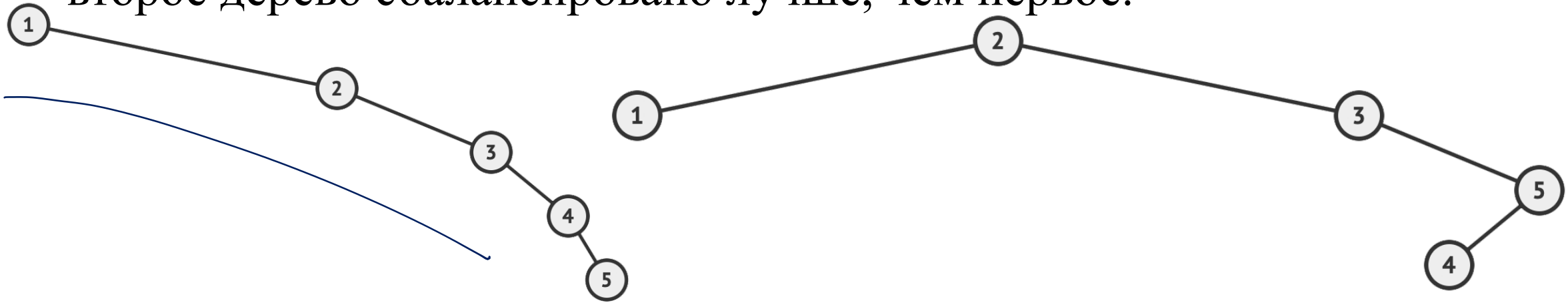
- Поиск
  Для поиска элемента 1 мы должны пройти все элементы (в порядке 3, 2, 1). Следовательно, поиск в двоичном дереве поиска имеет наихудшую сложность O(n).
- Вставка
  Для вставки элемента 0 он должен быть вставлен как левый дочерний элемент 1. Следовательно, нам нужно пройти по всем элементам (в порядке 3, 2, 1), чтобы вставить 0, который имеет наихудшую сложность O(n).
- Удаление
  Для удаления элемента 1 мы должны обойти все элементы, чтобы найти 1 (в порядке 3, 2, 1). Следовательно, удаление в двоичном дереве имеет наихудшую сложность O(n).

Применение

- Хранение иерархических данных, таких как структура папок, организационная структура, данные XML/HTML.
- Бинарное дерево поиска - это дерево, которое позволяет быстро выполнять поиск, вставку, удаление по отсортированным данным. Это также позволяет найти ближайший элемент.

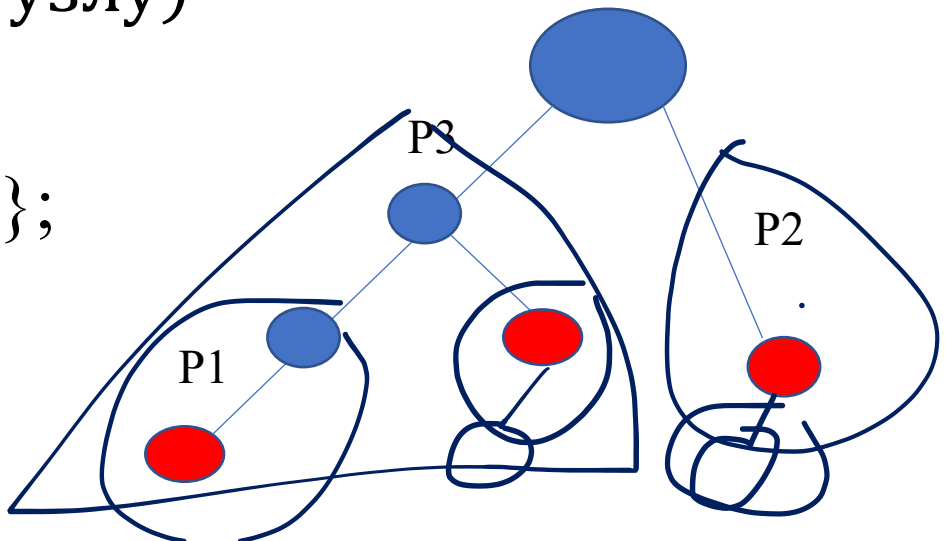
Сбалансированные деревья

Как уже было отмечено, время, затрачиваемое на исполнение операций с бинарными деревьями поиска, напрямую зависит от высоты этих деревьев. При этом в худшем случае высота будет равна количеству элементов, как видно из уже рассмотренных примеров. Во избежание такой разницы в асимптотике существуют **сбалансированные деревья**. В данном случае считается, что второе дерево сбалансировано лучше, чем первое.



Сбалансированные деревья

- Будем называть дерево сбалансированным, если для каждой его вершины высота её двух поддеревьев различается не более чем на 1.
- В некоторых случая возможно ослабление данного условия
- $|P2 - P1| \leq 1 \; \forall P1, P2 \; -$
  пути от корня к листу (или внешнему узлу)
- Внешний узел – это узел без элемента
- Node {Item item, Node * left, Node * right};
  Внешний узел – это 0.



Один из вариантов балансировки

$partR(\text{node}, k)$ – помещение k-го наименьшего элемента в корень дерева node

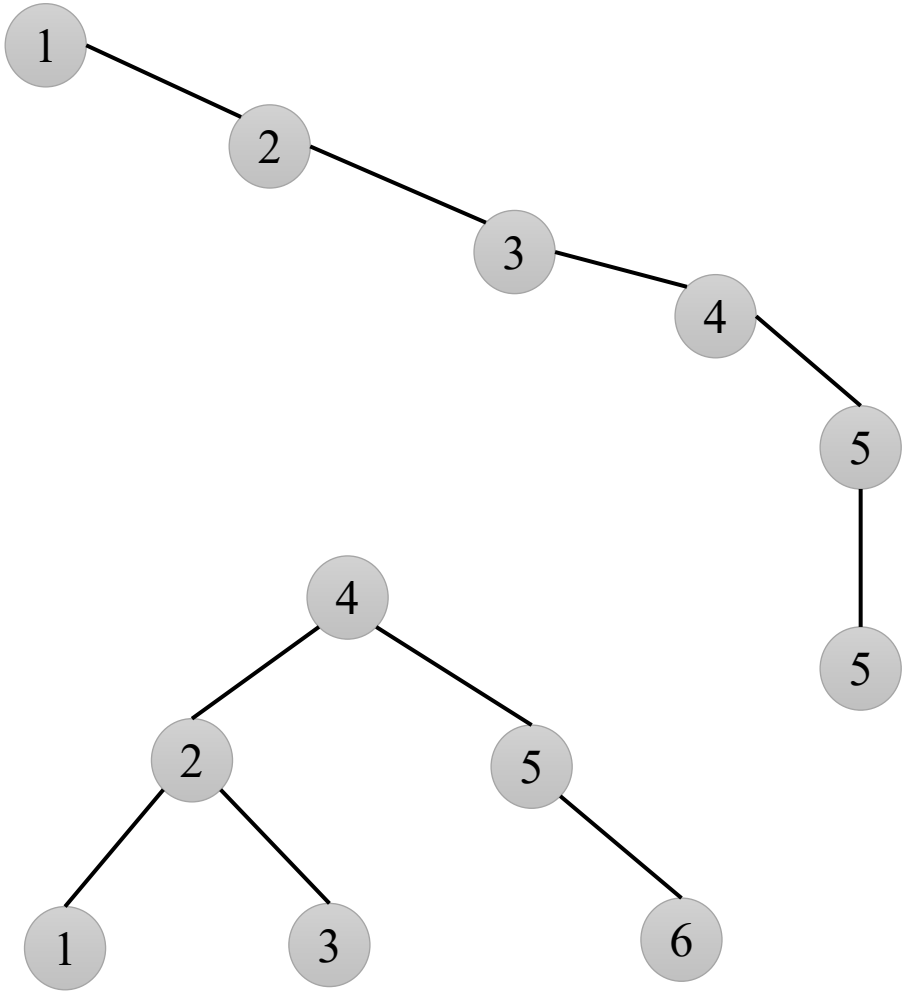
Программа балансировки:

```
void balance(Node * node) {
    If (node == 0) || (node->size == 1) { return; }
    partR(node, [node->size/2]);
    balance(node->left);
    balance(node->right);
}
```

Сбалансированные деревья

Проблема: как построить дерево поиска минимальной высоты?

Если набор ключей известен заранее, то его надо упорядочить. Корнем поддерева становится узел с ключом, значение которого – медиана этого набора. Для упорядоченного набора, содержащего нечетное количество ключей, – это ключ, находящийся ровно в середине набора (для четного числа можно брать любое из средних значений).



Сбалансированные деревья

Проблема: полный набор ключей не всегда известен заранее

Если ключи поступают по очереди, то построение дерева поиска будет зависеть от порядка их поступления. Поэтому при добавлении очередного узла, возможно, дерево понадобится перестраивать, чтобы уменьшить его высоту, сохраняя тот же набор узлов. Идеальную балансировку поддерживать сложно. Если при добавлении очередного узла количество узлов в левом и правом поддеревьях какого-либо узла дерева станет различаться более, чем на 1, то дерево не будет являться идеально сбалансированным, и его надо будет перестраивать, чтобы восстановить свойства идеально сбалансированного дерева поиска. Поэтому обычно требования к сбалансированности дерева менее строгие.

Виды сбалансированных деревьев

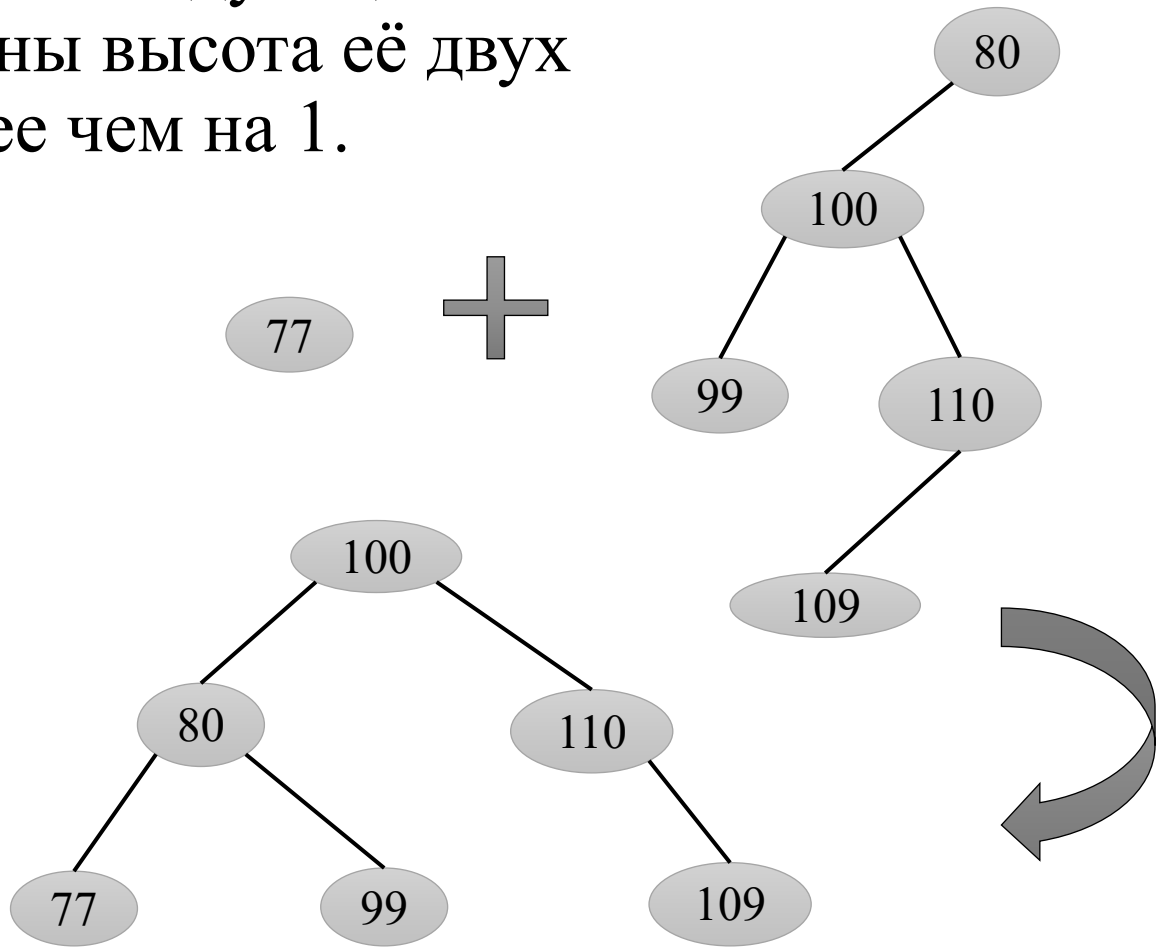
Основные виды сбалансированных деревьев поиска:

- АВЛ-деревья
- красно-черные деревья
- самоперестраивающиеся деревья (splay-деревья)

АВЛ-деревья. Определение

АВЛ-дерево – сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Данное условие проверяется после каждого добавления или удаления узла, также определен минимальный набор операций перестройки дерева, который приводит к восстановлению свойства сбалансированности, если оно оказалось нарушено.



Вычислительная сложность

- Поиск
  Для поиска элемента 1 мы должны пройти элементы (в порядке 5, 4, 1) = 3 = log n. Следовательно, поиск в дереве AVL имеет наихудшую сложность O(log n).
- Вставка
  Для вставки элемента 12 он должен быть вставлен как правый дочерний элемент 9. Следовательно, нам нужно пройти элементы (в порядке 5, 7, 9), чтобы вставить 12, сложность которого в наихудшем случае равна O(log n).
- Удаление
  Для удаления элемента 9 мы должны обойти элементы, чтобы найти 9 (в порядке 5, 7, 9). Следовательно, удаление в двоичном дереве имеет наихудшую сложность O(log n).

Красно-черные деревья. Определение

Красно-чёрное дерево – двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" и "чёрный". При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными (для экономии памяти фиктивные листья можно сделать одним общим фиктивным листом).

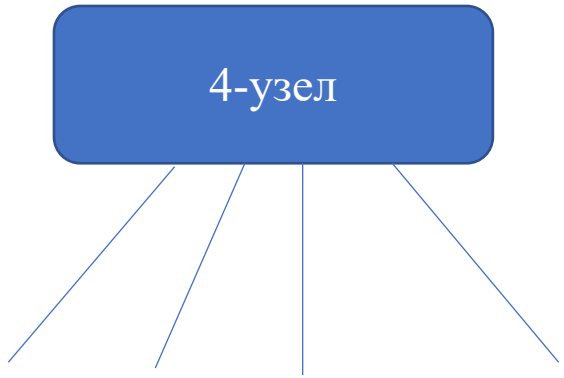
Для каждого узла должны выполняться *свойства*:

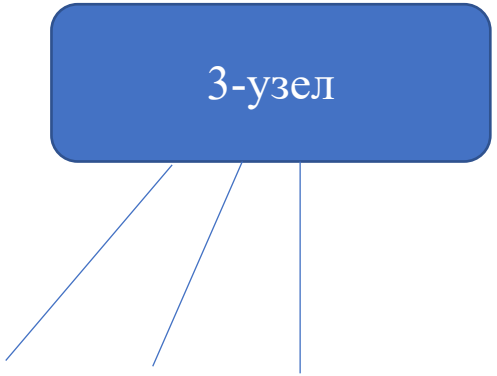
- Каждый узел промаркирован красным или чёрным цветом
- Корень и конечные узлы (листья) дерева — чёрные
- У красного узла родительский узел — чёрный
- Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
- Чёрный узел может иметь чёрного родителя

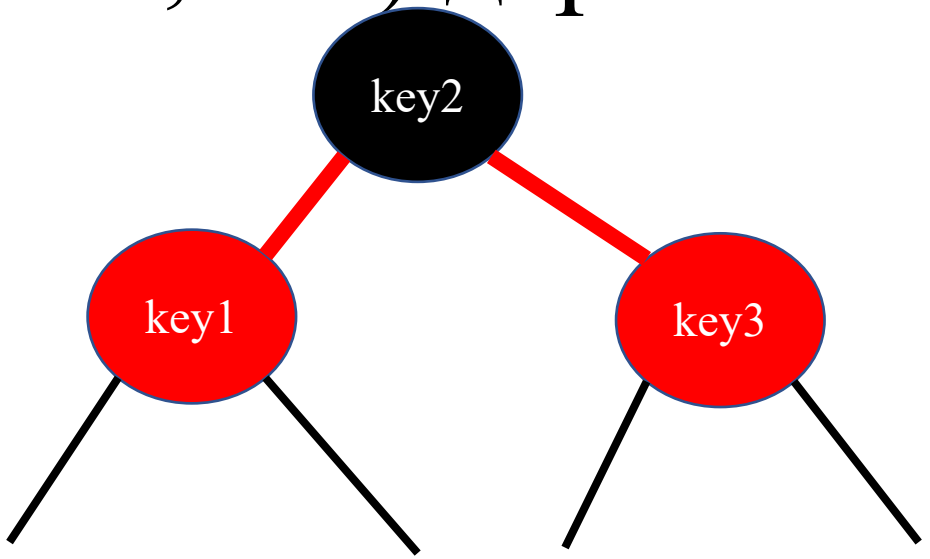
Красно-черные (Red-Black, RB) деревья

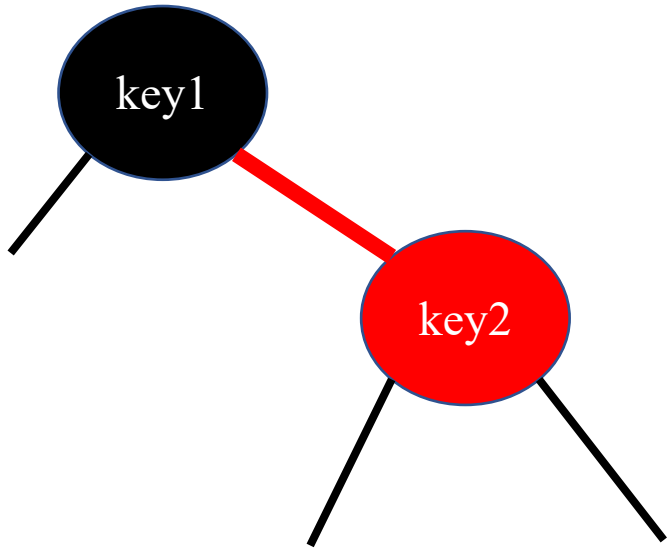
- Это способ упрощенной реализации 2-3-4 дерева с помощью бинарных деревьев: добавляется специальный разряд для кодирования узлов
- Типы связей:
  - Красная (red) связь объединяет бинарные деревья, образующие 3-узлы и 4-узлы
  - Черная (black) связь объединяет 2-3-4 дерево
- Каждый узел определяется одной связью – окраска узлов и связей имеет одно значение

Красно-черные (Red-Black, RB) деревья









Красно-черные (Red-Black, RB) деревья

- Работает стандартный поиск
- Балансировка дерева осуществляется по алгоритму 2-3-4 дерева при вставке автоматически
- Лемма: поиск в RB-дереве с N узлами требуется менее $2lgN + 2$ сравнений
- Лемма: поиск в RB-дереве с N случайными узлами требует в среднем $1.002\ lgN$ сравнений

Красно-черные (Red-Black, RB) деревья

- Бинарное RB дерево – это бинарное дерево, где каждый узел помечен как красный или черный, где два красных узла не могут быть соседями
- Сбалансированное Бинарное RB дерево – это бинарное RB-дерево, где каждый путь от корня к листу содержит одинаковое число черных узлов.

Красно-черные (Red-Black, RB) деревья

```
struct Node {
    Item item;
    Node * left;
    Node * right;
    Int depth;
    bool isRed;
};

void Insert(Item & item);
```

Вычислительная сложность

- Вставка, поиск, удаление – O(log n)

Применение

- RB-деревья обеспечивают сопоставимые с различными вычислениями, идеальное время вычислений для операций вставки, удаления и поиска.
- Этот факт позволяет использовать их в приложениях с точки зрения времени расчета, например, в непрерывных приложениях.
- Несмотря на это, благодаря их качествам, мы также можем использовать RB-деревья в качестве важнейших структурных блоков в информационных структурах, фундаментальных для различных областей:
  - Деревья AVL
  - Функциональное программирование
  - Вычислительная геометрия
  - Ядро Linux
  - Машинное обучение
  - Движки баз данных

Преимущества и недостатки

+ Красно-черные деревья уравновешивают уровень параллельного дерева.

+ Красно-черное дерево получает некоторый запас для структурирования дерева, восстанавливая уровень параллельного дерева.

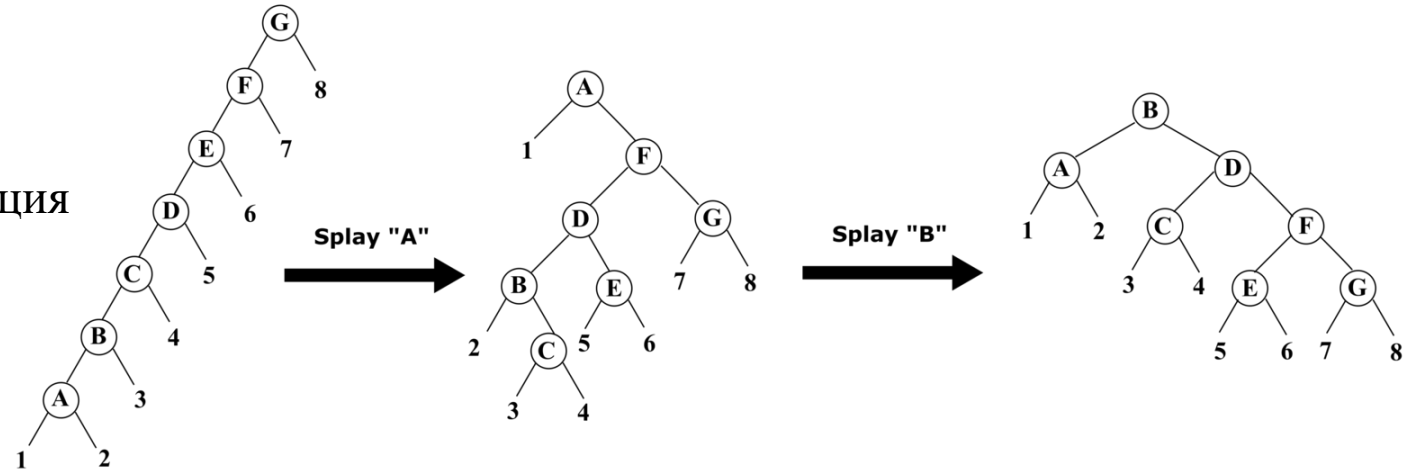
+ Хорошая вычислительная сложность.

- Сложность в реализации из-за сложных граничных случаев.
- Поскольку B-деревья могут иметь переменное количество дочерних элементов, их регулярно предпочитают красно-черным деревьям для упорядочения и размещения большого количества данных на табличках, поскольку они могут быть несколько мелкими, чтобы ограничить круговые задачи.
- Блокировка красно-черных деревьев неэффективна при одновременном доступе по сравнению с блокировкой skip-list, которые (1) очень быстры даже при одновременном доступе; (2) часто менее сложны в выполнении; и (3) предлагают в основном все преимущества фиксации красно-темных деревьев.

Splay деревья. Определение

Splay-дерево – это двоичное дерево поиска, которое позволяет находить быстрее те данные, которые использовались недавно.

Идея основана на принципе перемещения найденного узла в корень дерева. Эта операция называется splay(T, k), где k – это ключ, а T – двоичное дерево поиска. После выполнения операции splay(T, k) дерево T перестраивается, оставаясь при этом деревом поиска, так, что:



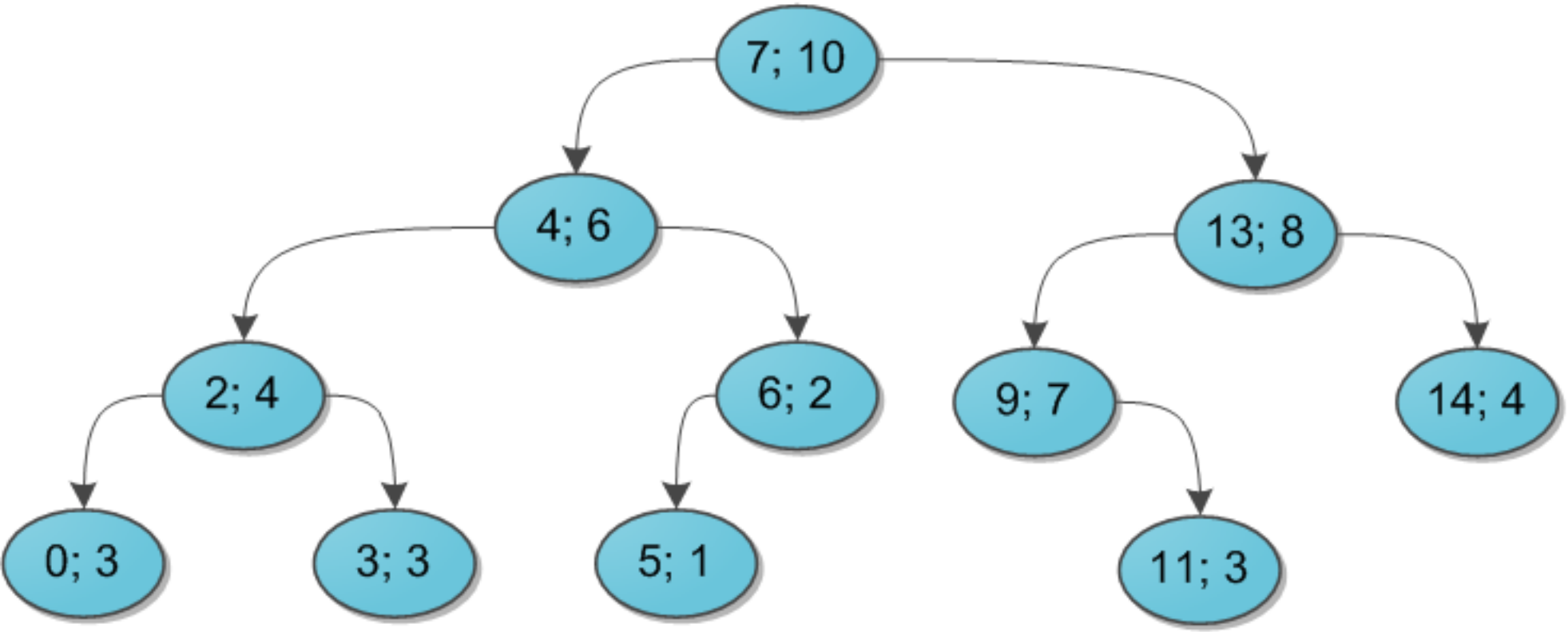
- если узел с ключом k есть в дереве, то он становится корнем;
- если узла с ключом k нет в дереве, то корнем становится его предшественник или последователь.

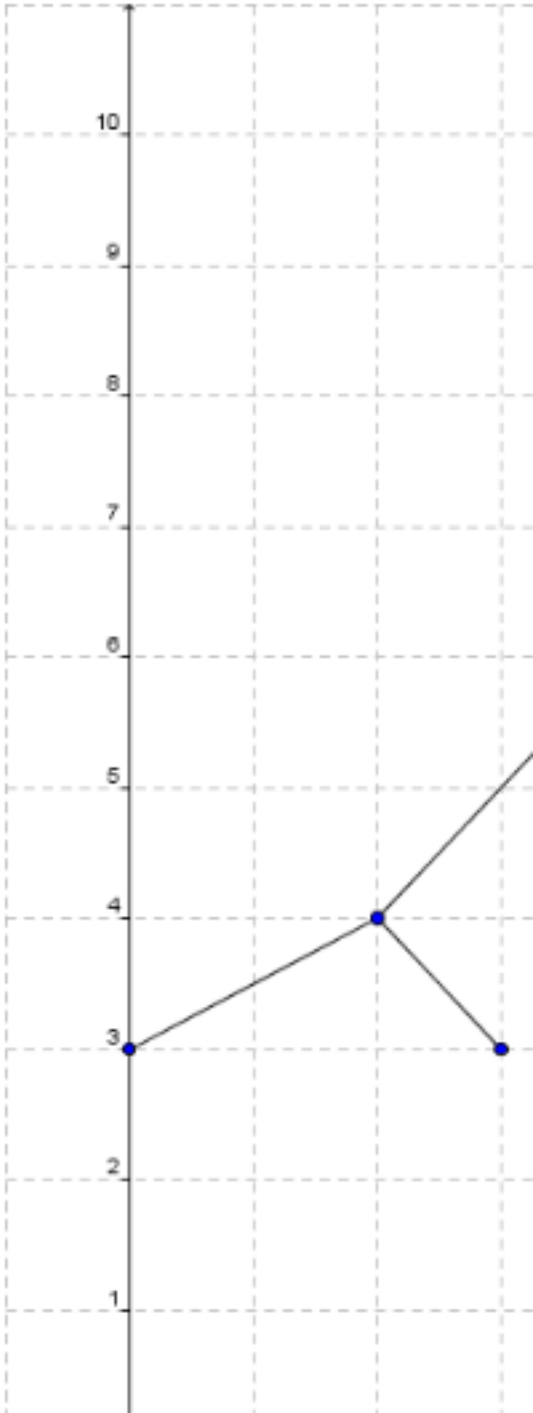
Таким образом, поиск узла в самоперестраивающемся дереве фактически сводится к выполнению операции splay.

Декартово дерево

Декартово дерево – структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу. Она хранит пары (x, y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска по x и бинарной пирамидой по y.

* *Бинарная куча/пирамида* – такое двоичное дерево, для которого выполнены три условия: значение в любой вершине не меньше, чем значения её потомков; глубина всех листьев отличается не более чем на 1 слой; последний слой заполняется слева направо без «дырок».


7; 10
4; 6
13; 8
2; 4
6; 2
9; 7
14; 4
0; 3
3; 3
5; 1
11; 3


10
9
8
7
6
5
4
3
2
1

Операции

Декартово дерево предоставляет следующие операции:

- **вставка(x, y)** – в среднем *O(log N)*
- **поиск(x)** – в среднем *O(log N)*
- **удаление(x, y)** – в среднем *O(log N)*
- **построение(x1…xn)** – в среднем *O(N)*
- **объединение(T1, T2)** – в среднем *O(M log (N/M))*
- **пересечение(x, y)** – в среднем *O(M log (N/M))*

Кроме того, за счёт того, что декартово дерево является и бинарным деревом поиска по своим значениям, к нему применимы такие операции, как нахождение K-го по величине элемента, и, наоборот, определение номера элемента.

Представление

С точки зрения реализации, каждый элемент содержит в себе X, Y и указатели на левого L и правого R сына.

Например,

```
struct Node:
    T x
    T y
    Node left
    Node right
```

Вспомогательные операции

Для поддержки описанных ранее операций с деревом необходимо реализовать такие вспомогательные операции, как ***Split*** и ***Merge***.

- **Split(T, X)** разделяет дерево T на два дерева L и R (которые являются возвращаемым значением) таким образом, что L содержит все элементы, меньшие по ключу X, а R содержит все элементы, большие X. Эта операция выполняется за *O(log N)*.
- **Merge (T1, T2)** объединяет два поддерева T1 и T2 и возвращает новое дерево. Эта операция также реализуется за *O(log N)*. Она работает в предположении, что T1 и T2 обладают соответствующим порядком (все значения X в первом меньше значений X во втором). Таким образом, нам нужно объединить их так, чтобы не нарушить порядок по приоритетам Y. Для этого просто выбираем в качестве корня то дерево, у которого Y в корне больше, и рекурсивно вызываем себя от другого дерева и соответствующего сына выбранного дерева.

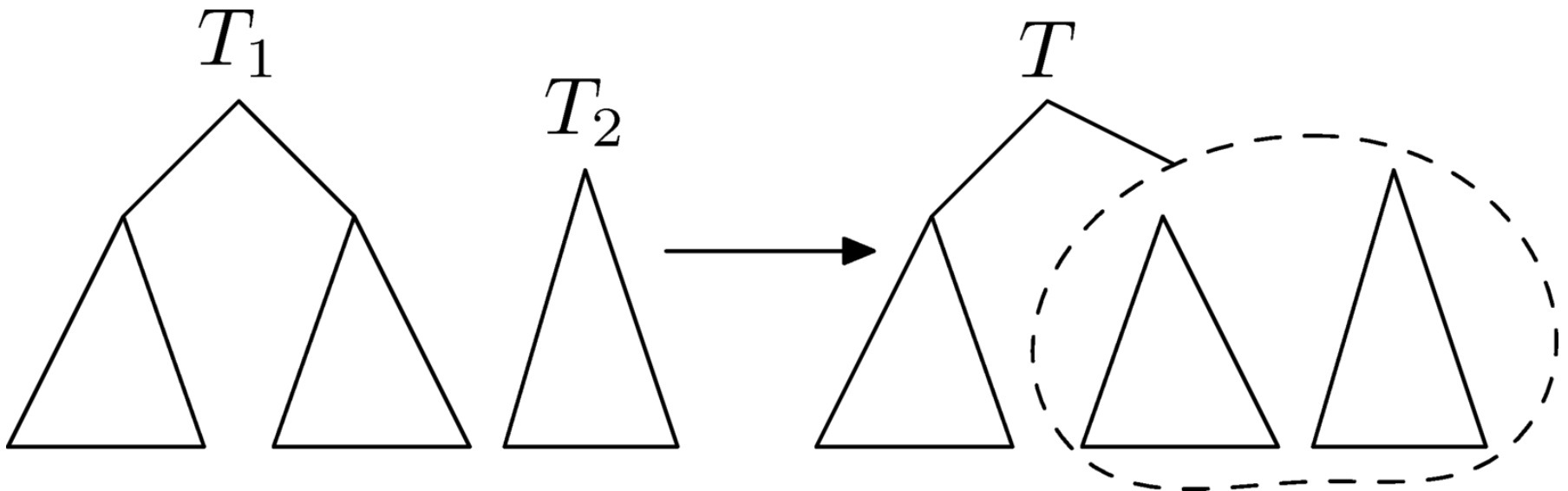
Merge

Пусть у нас существуют два декартовых дерева – левое и правое. Причем все ключи *x* левого меньше всех ключей правого. Необходимо реализовать операцию для слияния этих деревьев в одно дерево T. ***Вопрос***: что будет являться корнем итогового дерева? – Это должен быть узел с наибольшим значением y. Для этого есть два кандидата – корни левого и правого поддеревьев.

Merge

Рассмотрим случай, когда у корня T1 больше у корня T2. Случай наоборот будет симметричен данному.

В данном случае корнем нового дерева T будет корень T1. Тогда левое поддерево T совпадет с левым поддеревом T1. Справа же нужно подвесить объединение правого поддерева T1 и дерева T2.



Merge

```
// treap – сокращенное название декартова дерева
Treap Merge(Treap t1, Treap t2):
    if t2 == ∅
        return t1
    if t1 == ∅
        return t2
    if t1.y > t2.y
        t1.right  = merge(t1.right, t2)
        return t1
    else
        t2.left = merge(t1, t2.left)
        return t2
```

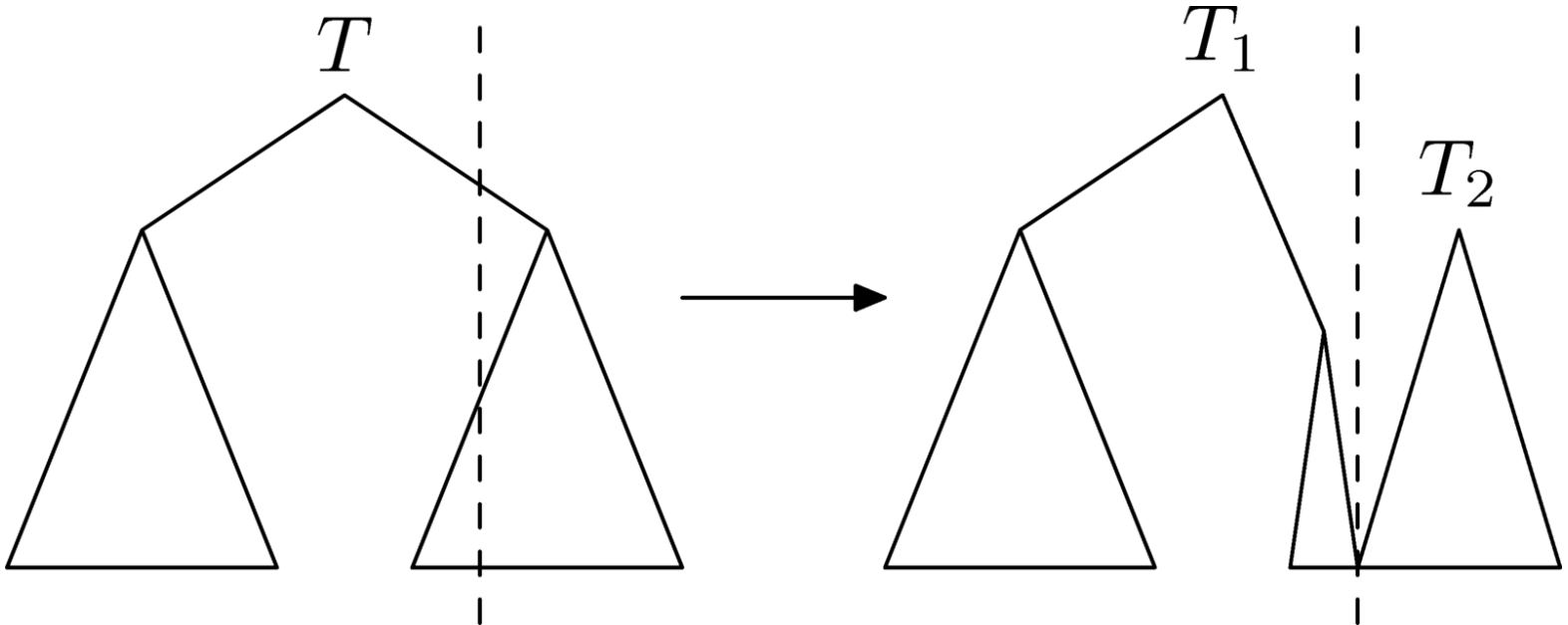
Split

Данная операция позволяет разрезать исходное дерево T по ключу k. Возвращать она будет такую пару деревьев (T1, T2), что в дереве T1 ключи меньше *k*, а в T2 – все остальные.

Split

Рассмотрим случай, в котором требуется разрезать дерево по ключу, большему ключа корня. Посмотрим, как будут устроены результирующие деревья:

- T1: левое поддерево T1 совпадет с левым поддеревом T. Для нахождения правого поддерева T1 нужно разрезать правое поддерево T на T1(R) и T2(R) по ключу *k* и взять T1(R)
- T2: совпадет с T2(R)



Split

```
<Treap, Treap> Split(Treap t, int k):
    if t == ∅
        return <∅, ∅>
    if k > t.x
        t1, t2 = split(t.right, k)
        t.right = t1
        return <t, t2>
    else
        t1, t2 = split(t.left, k)
        t.left = t2
        return <t1, t>
```

Insert

Операция **insert(T, k)** добавляет в дерево T элемент k, где $k.x$ – ключ, а $k.y$ – приоритет.

Будем считать, что k – декартово дерево из одного элемента, и для того чтобы его добавить в наше декартово дерево T нам нужно их слить. ***Проблема***: T может содержать ключи как меньше, так и больше, чем $k.x$, поэтому сначала нужно разрезать T по данному ключу.

Insert

```
void insert(Treap t, Treap k):
    if t == ∅
        t = k
    if k.y > t.y                    // спускаемся по дереву и останавливаемся на первом элементе, где
                                        приоритет меньше добавляемого
        t1, t2 = split(t, k.x)      // разбиваем поддерево по нужному ключу
        k.left = t1
        k.right = t2
        t = k                       //  перезаписываем результат
    else
        insert(k.x < t.x ? t.left : t.right, k)
```

Remove

Операция **remove(T, k)** удаляет из дерева T элемент с ключом *k*.

```
void remove(Treap t, int k):
    if t.x == k                                  // спускаемся по дереву в поисках удаляемого элемента
        t1 = merge(t.left, t.right)              // найдя элемент, вызываем слияние левого и правого сыновей
        t = t1                                   // результат ставим на место удаляемого элемента
    else
        remove(k < t.x ? t.left : t.right, k)
```

Принцип построения дерева

Рассмотрим случай, когда нужно построить дерево на основе существующих пар (x1, y1) … (xn, yn), при этом ключи упорядочены по возрастанию (x1 < … < xn).

Идея построения за линию: будем строить слева направо, начиная с (x1, y1), при этом помнить последнюю добавленную *k*-ую пару. Этот элемент будет самым правым, так как у него будет максимальный ключ, а по ключам декартово дерево представляет собой двоичное дерево поиска. При добавлении нового элемента пытаемся сделать его правым сыном предыдущего (если $yk > y(k+1)$), иначе делаем шаг к предку последнего элемента и смотрим его приоритет. Поднимаемся до тех пор, пока приоритет в рассматриваемом элементе меньше приоритета в добавляемом, после чего делаем *(k+1)-й* элемент его правым сыном, а предыдущий *k-й* – левым сыном *(k+1)-го*.

Заметим, что каждую вершину мы посетим максимум дважды: при непосредственном добавлении и, поднимаясь вверх (ведь после этого вершина будет лежать в чьём-то левом поддереве, а мы поднимаемся только по правому). Из этого следует, что построение происходит за линию.

Рандомизированное бинарное дерево

- Операция вставки:
  - Пусть в дереве N узлов
  - $N + 1$ узел будем добавлять в корень с вероятностью $1/(N + 1)$ иначе стандартным образом добавляем в дерево. Данная операция повторяется рекурсивно.
- Лемма 1.

  Построение рандомизированного дерева эквивалентно построению обычного дерева когда его ключи переставлены в произвольном порядке.

  Для создания такого дерева из N элементов используется порядка $\sim 2NlnN$ шагов

  Для выполнения поиска требуется порядка $\sim 2lnN$ сравнений
- Лемма 2.

  Вероятность того, что затраты на создание рандомизированного дерева превышают усредненные затраты в α раз, меньше $e^{-\alpha}$
- Следствие:
  - Затраты на поиск близки к усредненным

Рандомизированное дерево. Добавление.

```
void insertR(nodeLink & n, Item a) {
    if (n == 0) { n = new node(a); return; }
    if (rand() < RAND_MAX / (n->size() + 1)) {
        insertRoot(n, item);
    }
    else if (key(a) < key(n->item)) {
        insertR(n->left, a);
    } else {
        insertR(n->right, a);
    }
}

Key key(Item a) {
    return a.key;
}
```

Рандомизированное дерево. Объединение.

```
// a - что добавляем
// b - куда добавляем
node * joinR(node * a, node * b) {
    if (a == 0) { return b; }
    if (b == 0) { return a; }
    insertR(b, a->Item);
    b->left = joinR(a->left, b->left);
    b->right = joinR(a->right, b->right);
    delete a;
}
```

```
void join(BTree & tree) {
    if (rand()/(RAND_MAX/(head->size + tree.head->size)+1) < head->size) {
        head = joinR(head, tree->head);
    } else {
        head = joinR(tree->head, head);
    }
}
```

Рандомизированное дерево. Удаление.

- Удаление элемента то же, но для объединения двух деревьев

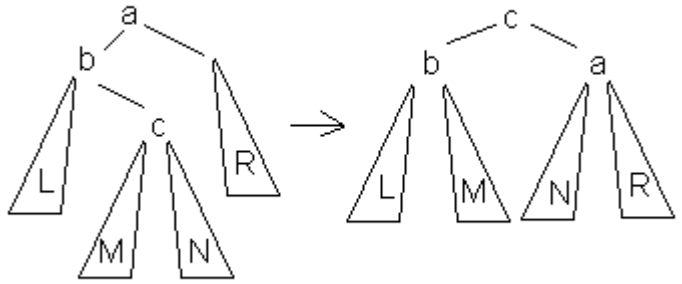
```
node * joinLR(node * a, node * b) {
    if (a == 0) { return b; }
    if (b == 0) { return a; }
    if (rand()/(RAND_MAX/(a->size + b->size)+1) < a->size) {
        a->r = joinLR(a->r, b); return a;
         Fix(a);
    } else {
        b->l = joinLR(a, b->l); return b;
         Fix(b);
    }
}
```

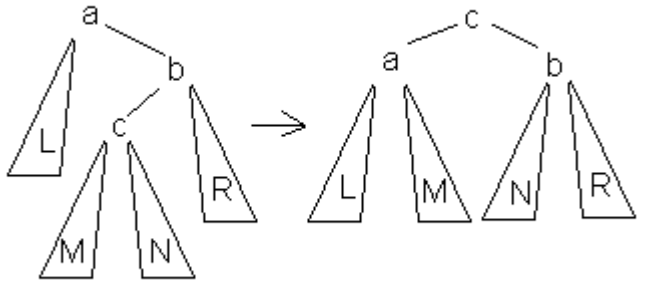
Временная сложность

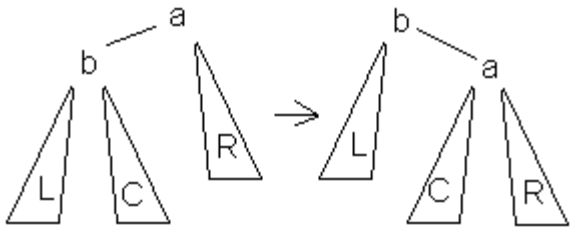
- Вставка, поиск, удаление – O(log N)

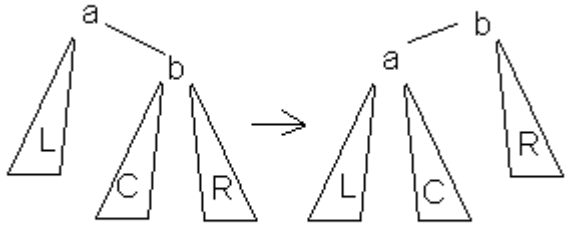
Расширенные бинарные деревья

- Используется двойная ротация дерева при вставке элемента
- Такое расширение позволяет исключить худший случай добавления новых элементов в дерево – квадратичный.









Расширенные бинарные деревья

```
typedef node * link;

void insert(Item & item) {
      splay(head, item);
   }

void splay(link & h, Item & item) {
      if (h == 0) { h = new node(item); return;}
      if (key(item) < key(h->item)) {
         link & hl = h->left;
         if (hl == 0) { h = new node(item, 0, h); return; }
         if (key(item) < key(hl->item)) { splay(hl->left, item); rotateR(h); }
         else { splay(hl->right, item); rotateL(hl); }
         rotateR(h);
      } else {
            link &hr = h->right;
            if (hr == 0) { h = new node(item, h, 0); return; }
            if (key(hr->item) < key(item)) { splay(hr->right, item); rotateL(h); }
            else { splay(hr->right, item); rotateR(hr); }
            rotateL(h);
         }
   }
```

Расширенные бинарные деревья

- Лемма: N вставок -> $O(NlgN)$ сравнений
- Лемма (Слеатор, Тарьян, 1985): M вставок или операций поиска в дерево глубины N требует $O((N+M)\lg(N+M))$ сравнений

- struct Node {Item item; link left; link right; int size; }

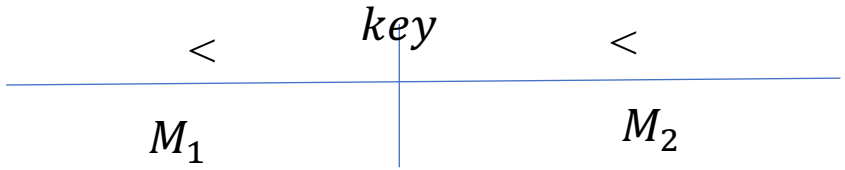
Применение

- Расширенные бинарные деревья полезны для представления в алгебраических выражениях.
- Они позволяют вычислить взвешенную длину пути: используется для вычисления общей длины пути в случае взвешенного дерева.

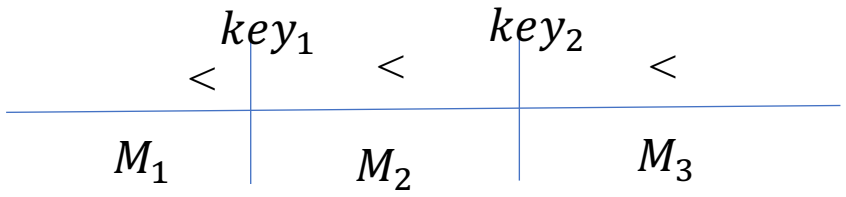
Нисходящие 2-3-4 деревья

- Узел может содержать более одного ключа:
  - 2-узел содержит один ключ и имеет 2 связи
  - 3-узел содержит 2 ключа и имеет 3 связи на каждый интервал, определяемый ключами
  - 4-узел содержит 3 ключа и имеет 4 связи на каждый интервал, определяемый ключами
- 2-3-4 дерево это дерево с узлами типа 2-узел, 3-узел и 4-узел
- Сбалансированное 2-3-4 дерево – это 2-3-4 дерево, где все пустые деревья расположены на одинаковой глубине

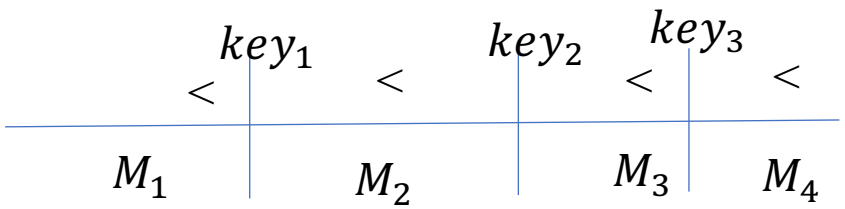
Нисходящие 2-3-4 деревья



$2-$ узел



$3-$ узел



$4-$ узел

Нисходящие 2-3-4 деревья. Вставка

- Вставка: 2-узел превращается в 3-узел, 3-узел превращается в 4-узел.
- Не допускается на нижнем уровне хранения 4-узла за счёт:
  - если встречается комбинация 2-узел – 4-узел, то она заменяется на 3-узел соединенный с двумя 2-узлами
  - если встречается комбинация 3-узел – 4-узел, то такая комбинация заменяется на 4-узел, соединенный с двумя 2-узлами
  - Корень дерева 4-узел -> разделяем на три 2 узла – только эта операция увеличивает высоту дерева

Нисходящие 2-3-4 деревья

- Лемма: При поиске в N-узловых деревьях посещается максимум lgN + 1 узел
- Лемма: Для вставки в N-узловое деревое требуется разделение менее lgN + 1, а в среднем с вероятностью 1 требуется разделение менее 1 узла

Нисходящие 2-3-4 деревья

- struct Node {
  - Item * item1; Item * item2; Item * item3;
  - int nodeSize();
  - Link link1; Link link2; Link link3; Link link4;
- };
- struct Node {
  - Item * item;
  - int nodeSize();
  - Node ** link;
- };

Временная сложность

- Поиск, вставка и удаление - O(log N).
- В худшем случае в 2-3-4-деревьях высота равна log N, а в лучшем случае высота равна 1/2 * log N (это условие, когда все узлы равны 4 узлам).