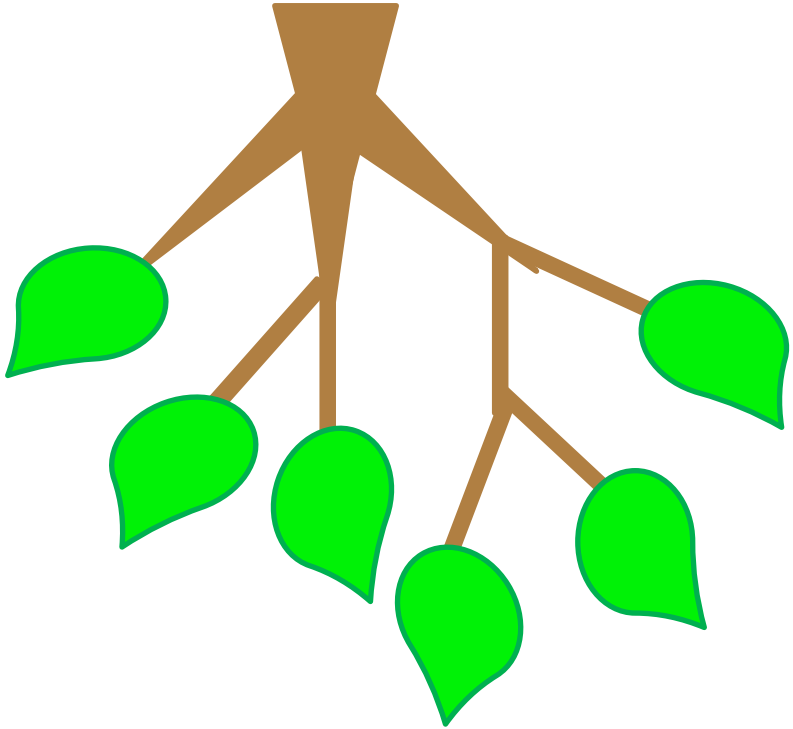
АиСД

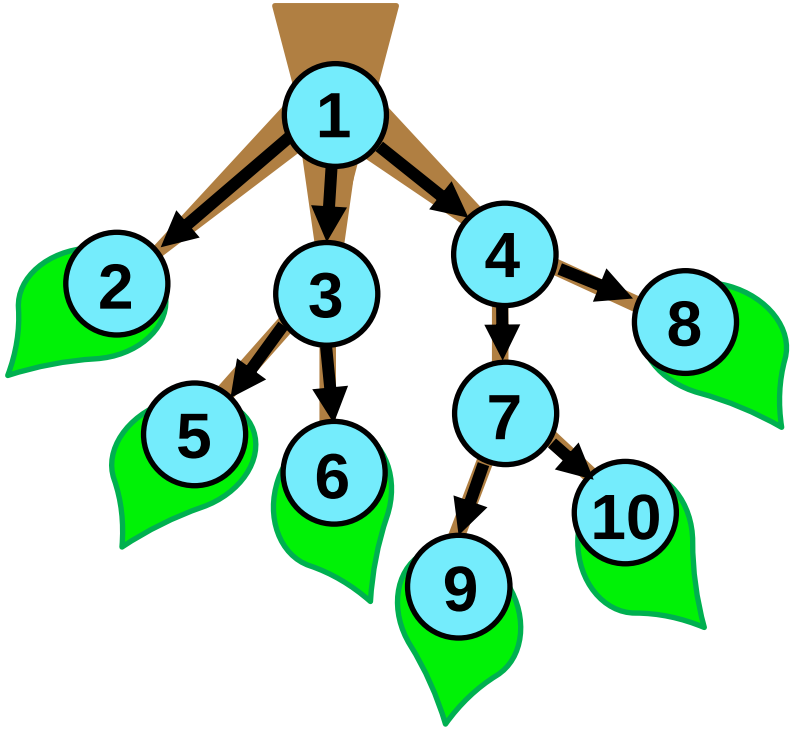
Бинарные деревья поиска

Вспомним, что такое дерево



Вспомним, что такое дерево

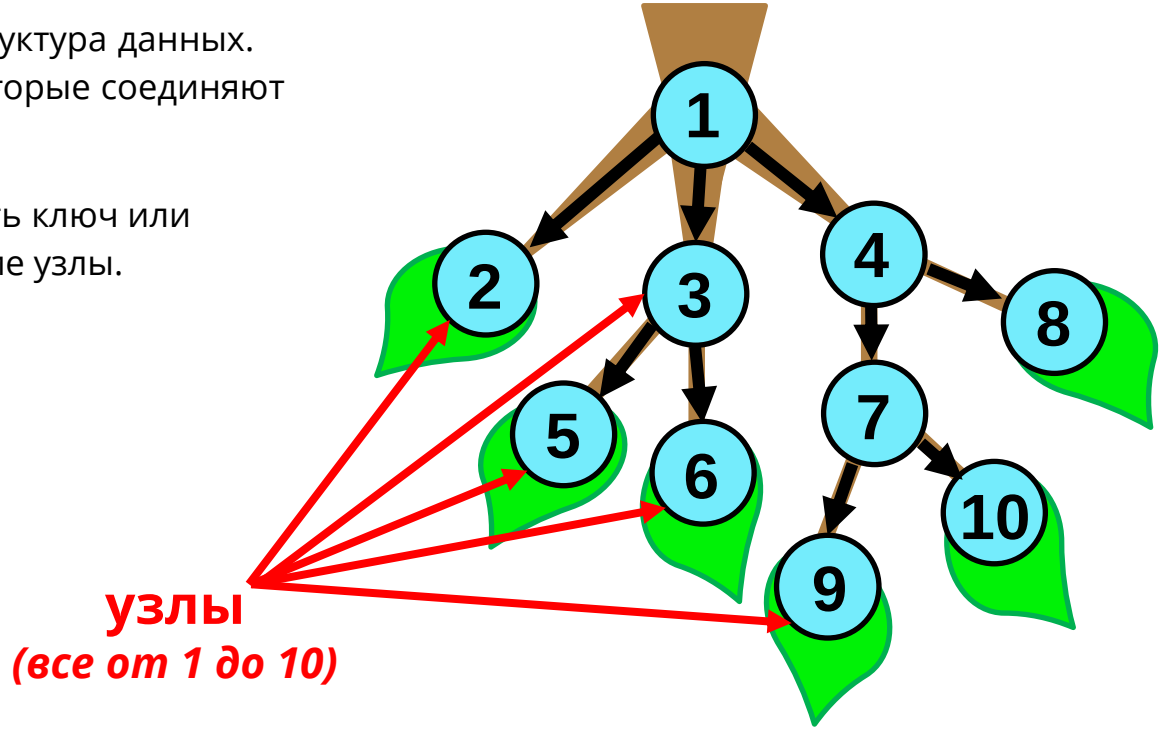
Дерево — это иерархическая структура данных. Она состоит из узлов и ребер, которые соединяют узлы.



Вспомним, что такое дерево

Дерево — это иерархическая структура данных. Она состоит из узлов и ребер, которые соединяют узлы.

Узел — это объект, в котором есть ключ или значение и указатели на дочерние узлы.

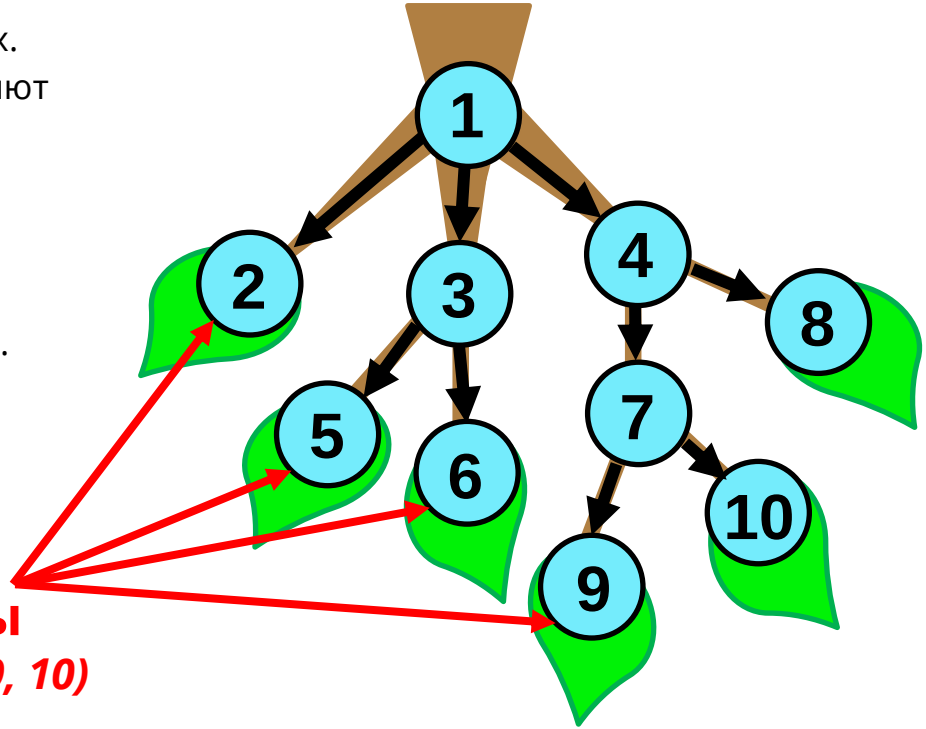


Вспомним, что такое дерево

Дерево — это иерархическая структура данных. Она состоит из узлов и ребер, которые соединяют узлы.

Узел — это объект, в котором есть ключ или значение и указатели на дочерние узлы.

Лист — это узел, у которых нет дочерних узлов.



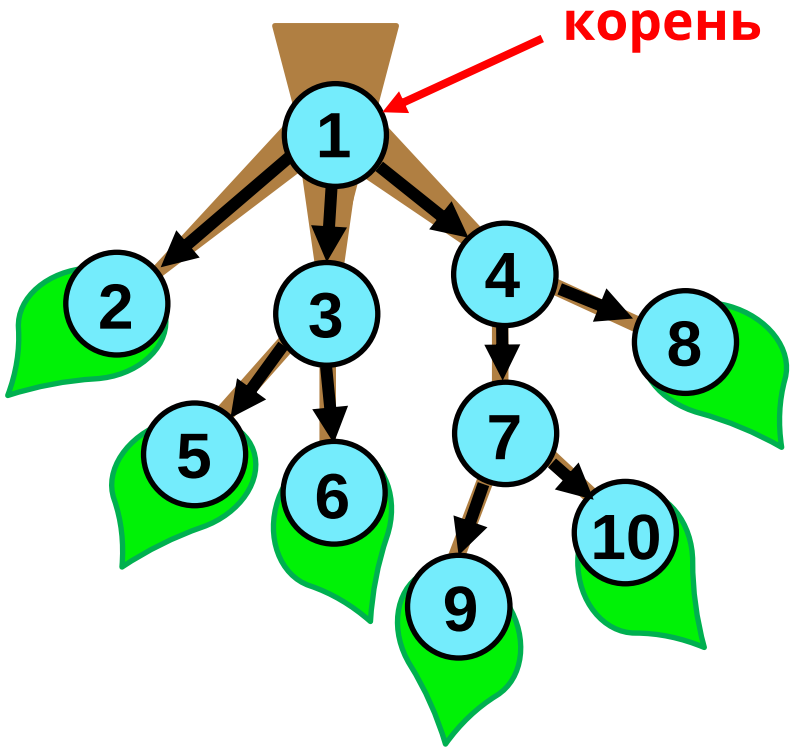
Вспомним, что такое дерево

Дерево — это иерархическая структура данных. Она состоит из узлов и ребер, которые соединяют узлы.

Узел — это объект, в котором есть ключ или значение и указатели на дочерние узлы.

Лист — это узел, у которых нет дочерних узлов.

Корень — это самый верхний узел дерева. Его ещё иногда называют корневым узлом.



Вспомним, что такое дерево

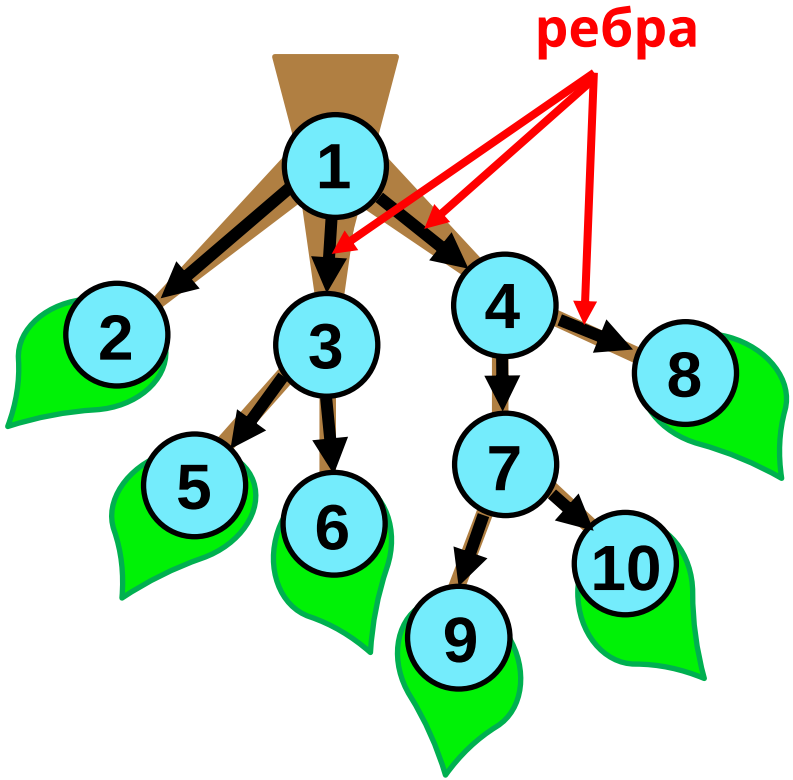
Дерево — это иерархическая структура данных. Она состоит из узлов и ребер, которые соединяют узлы.

Узел — это объект, в котором есть ключ или значение и указатели на дочерние узлы.

Лист — это узел, у которых нет дочерних узлов.

Корень — это самый верхний узел дерева. Его ещё иногда называют корневым узлом.

Ребро — это указатель, который связывает два узла



Высота узла

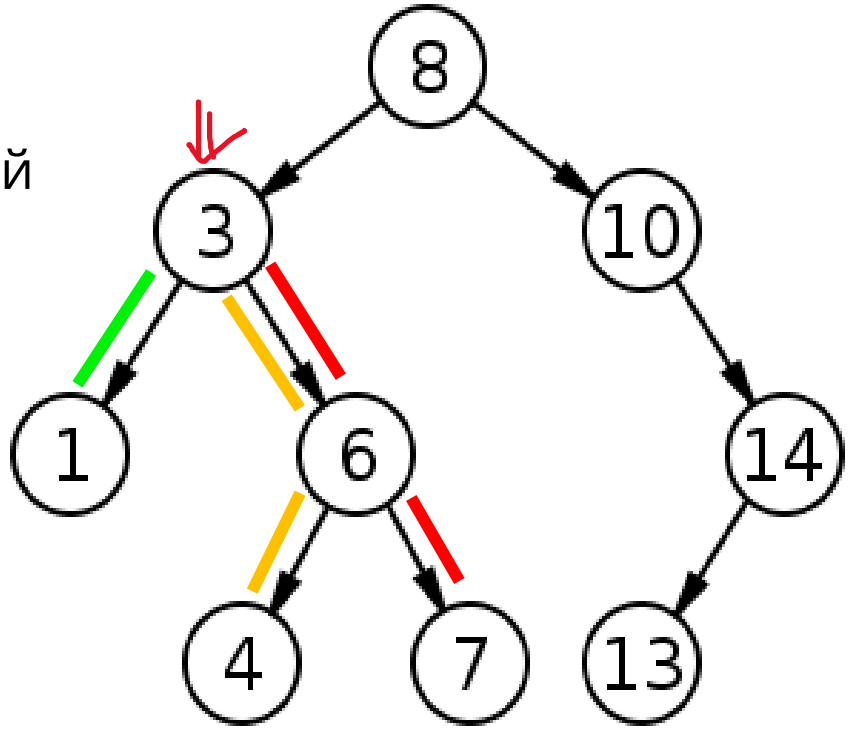
Высота узла (h) – это максимальное из всех расстояний от узла до его листьев

УЗЕЛ 3

Листья:	1	4	7	
Расстояния:	2	3	3	

Максимальное расстояние = **3**

h = 3



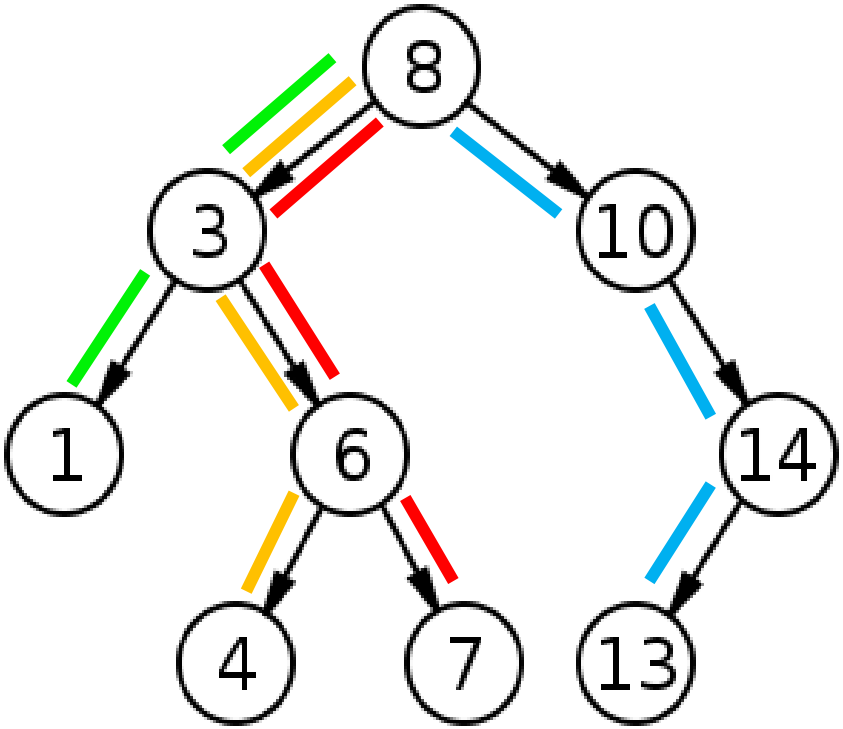
Высота дерева

Высота дерева (h) – это максимальное из всех расстояний от корня до листьев

Листья:	1	4	7	13
Расстояния:	2	3	3	3

Максимальное расстояние = **3**

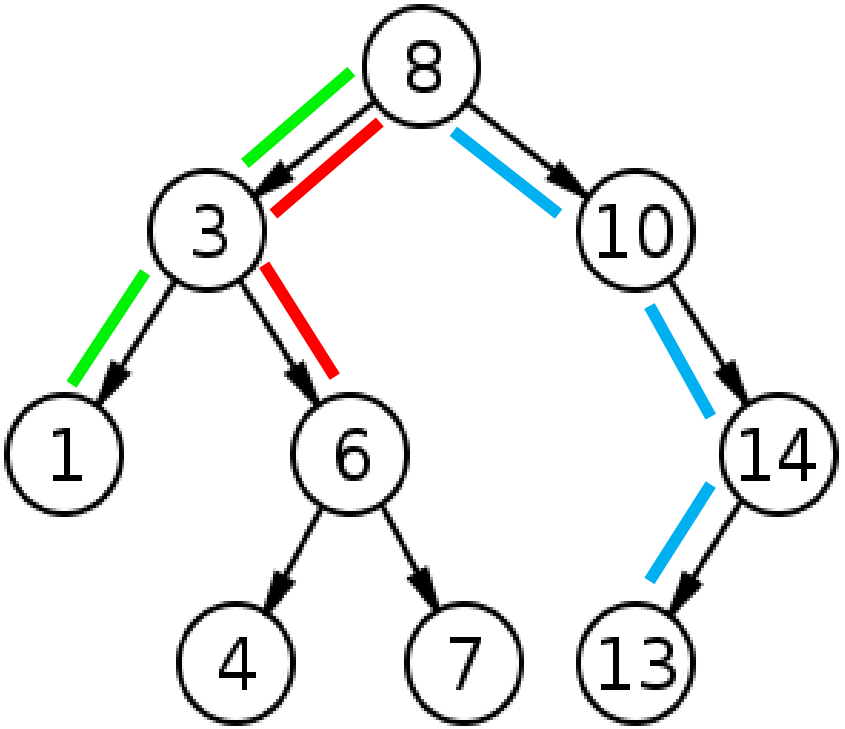
h = 3



Глубина узла

Глубина узла – это расстояние от корня до этого узла

- для узла **6** глубина = **2**
- для узла **13** (лист) глубина совпадает с высотой дерева и = **3**
- для узла **1** (лист) глубина = **2**



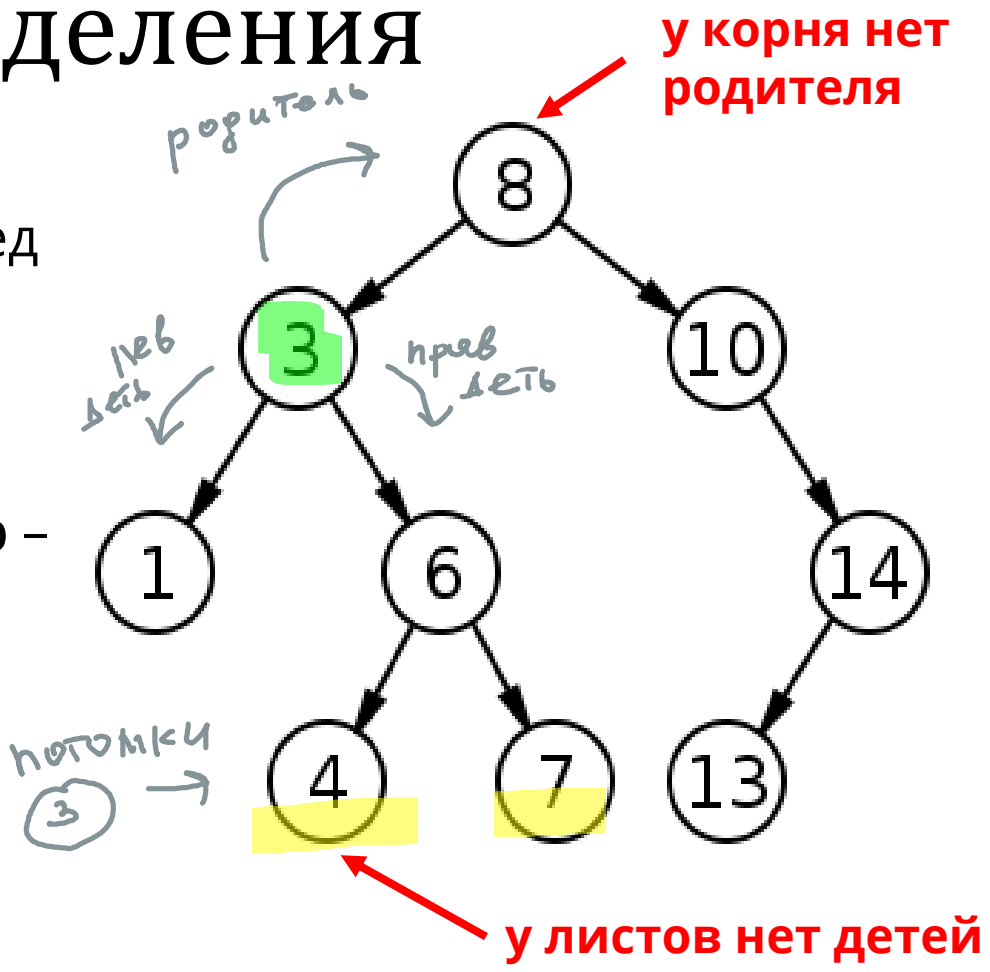
Некоторые определения

Родитель некоторого узла **v** – это узел **p**, который стоит перед **v** на пути от корня до **v**

По аналогии **дети** узла **p** – это все такие узлы **v**, что для них **p** – родитель

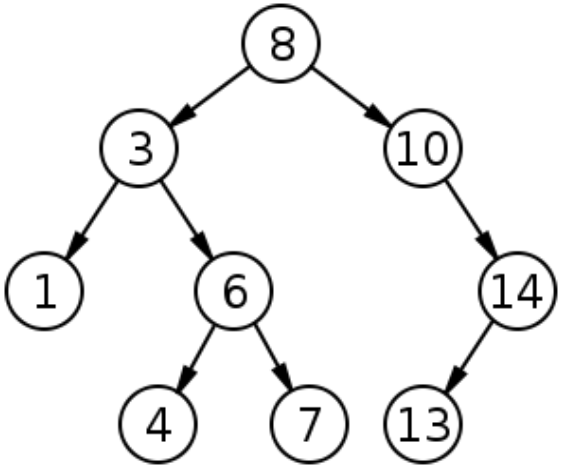
Пример: для узла **3**

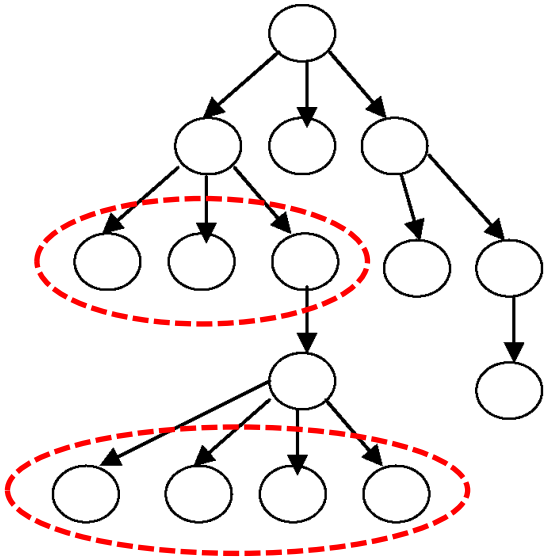
- родитель: узел **8**
- дети: узлы **1** и **6**



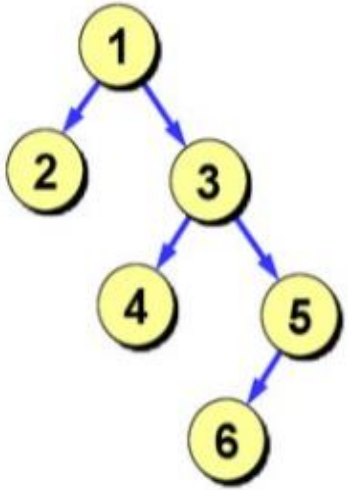
Двоичное дерево. Определение

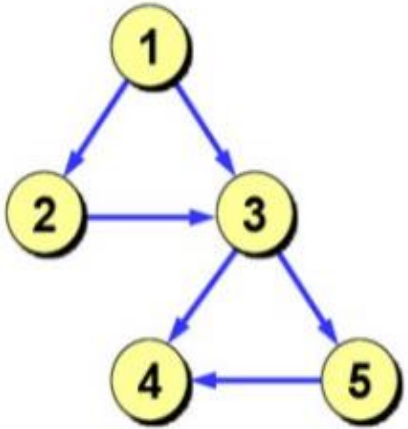
Двоичное дерево — древовидная структура данных, в которой у родительских узлов может быть не больше **двух** детей.

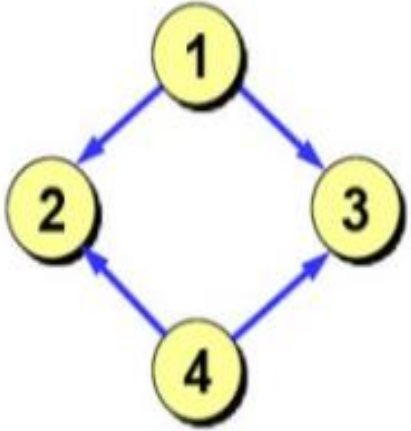




Какие структуры не деревья?

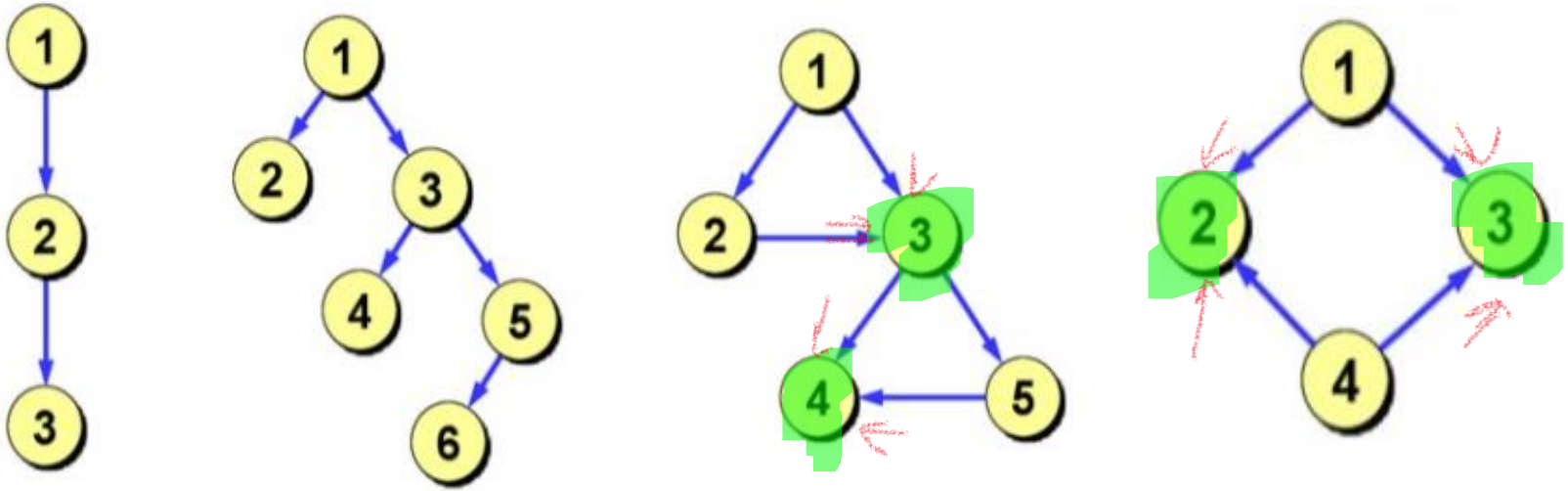






у узлов должно быть не более 1 родителя

Какие структуры не деревья?

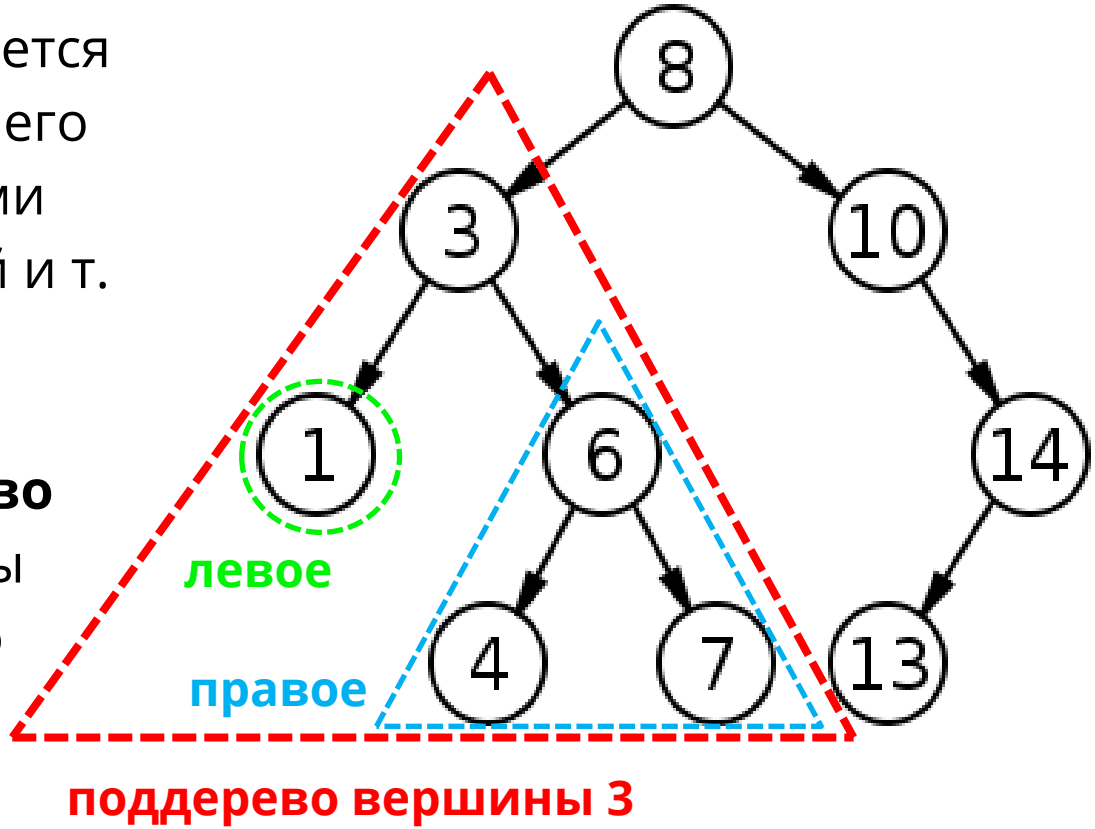


у узлов должно быть не более 1 родителя

Некоторые определения

Поддеревом узла называется сам узел, вместе со всеми его потомками (детьми, детьми детей, детьми детей детей и т. д. до листов)

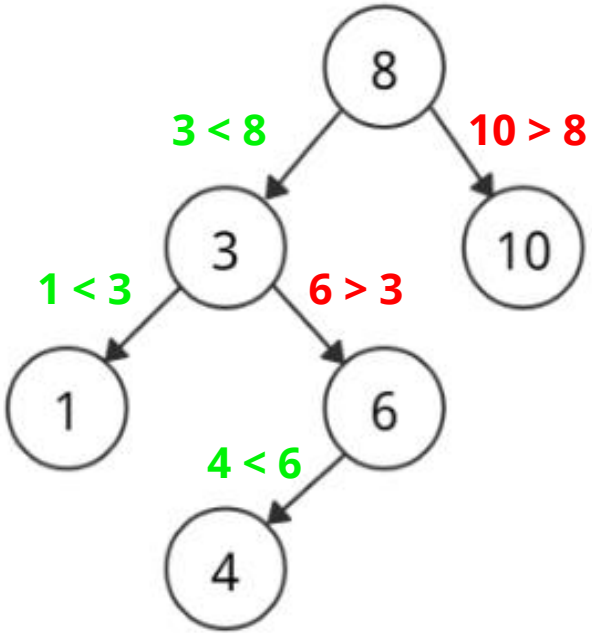
Левое / правое поддерево узла – это только вершины слева / справа (без самого узла)



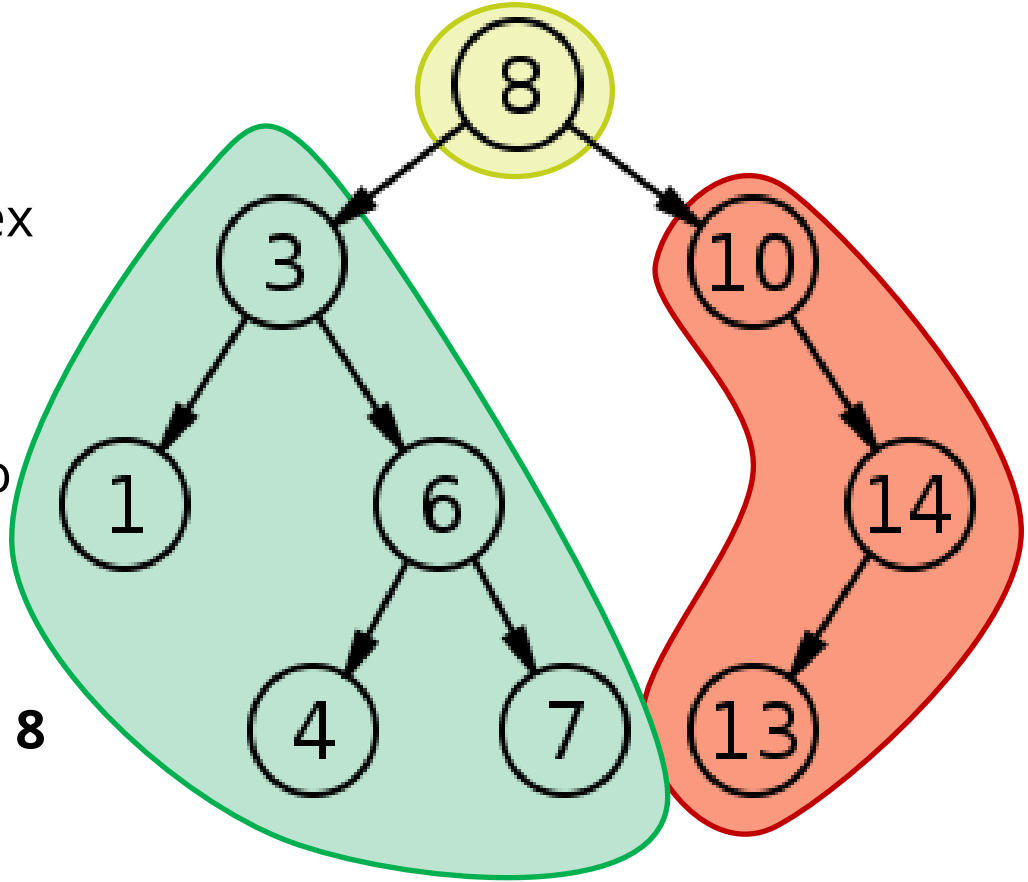
Двоичное дерево **поиска**

Двоичное дерево поиска - двоичное дерево, для которого выполняются следующие условия (свойства дерева):

- у всех узлов левого поддерева некоторого узла **X** значения ключей данных меньше (или равны) значения узла **X**
- у всех узлов правого поддерева некоторого узла **X** значения ключей данных больше значения узла **X**
- оба поддерева (левое и правое) некоторого узла **X** - двоичные деревья поиска

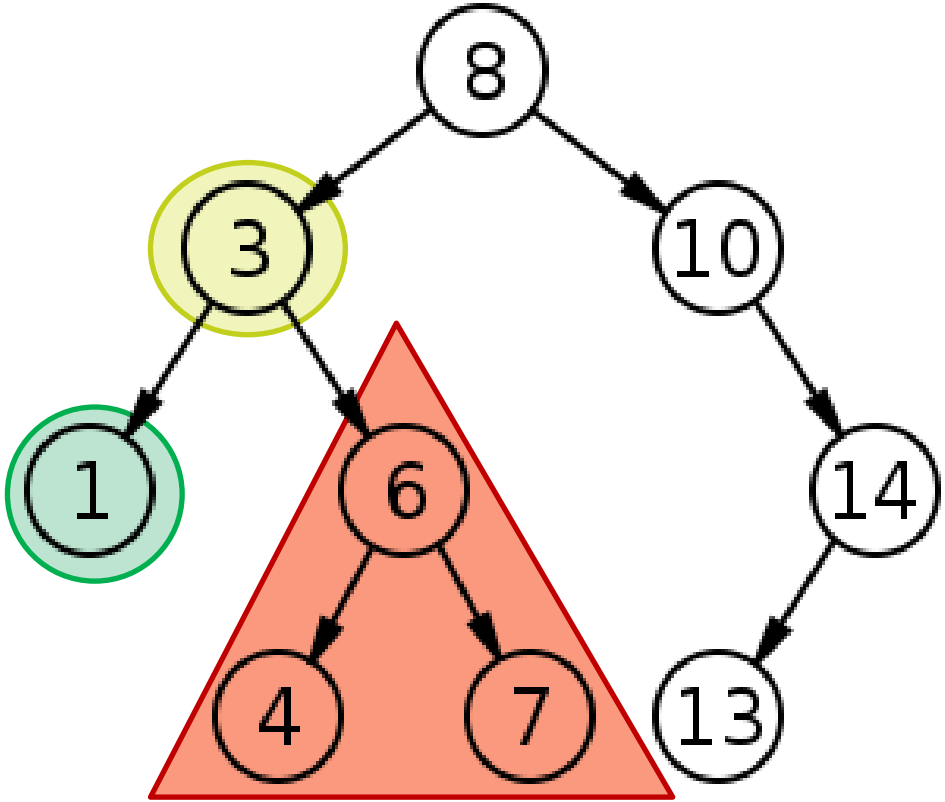


значения всех узлов левого поддерева вершины меньше либо равны, чем значение в узле
1, 3, 4, 6, 7 <= 8



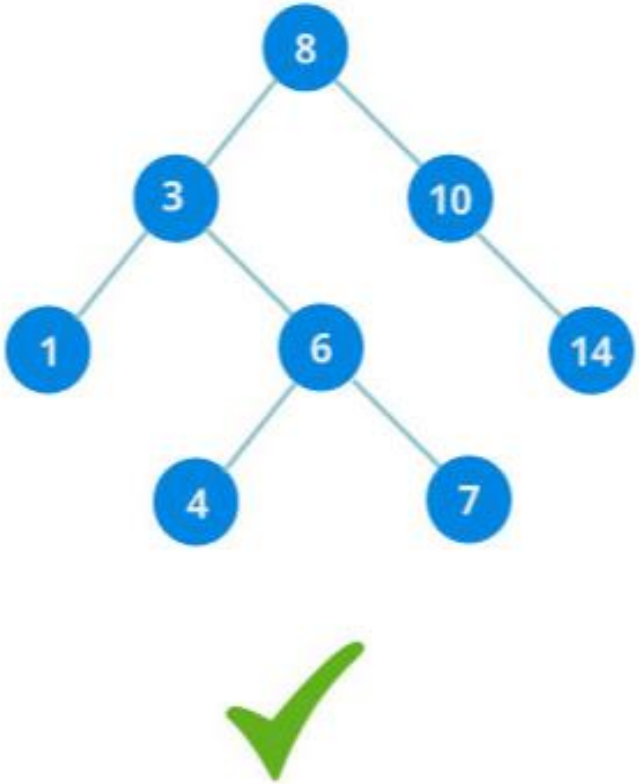
значения всех узлов правого поддерева вершины больше, чем значение в узле
10, 13, 14 > 8

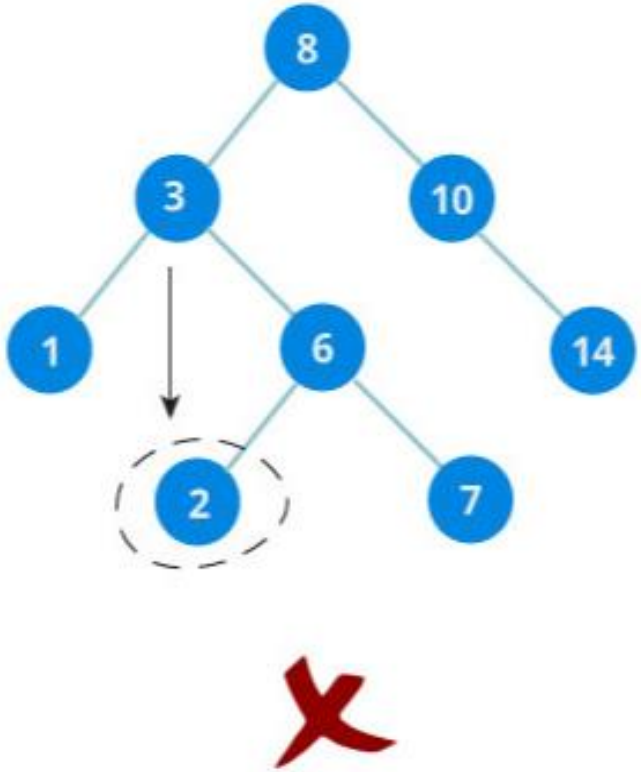
значения всех узлов левого поддерева вершины меньше либо равны, чем значение в узле
1 <= 3



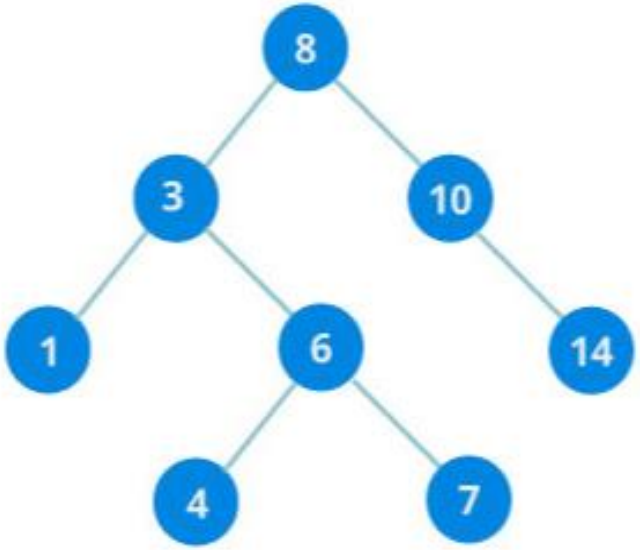
значения всех узлов правого поддерева вершины больше, чем значение в узле
4, 6, 7 > 3

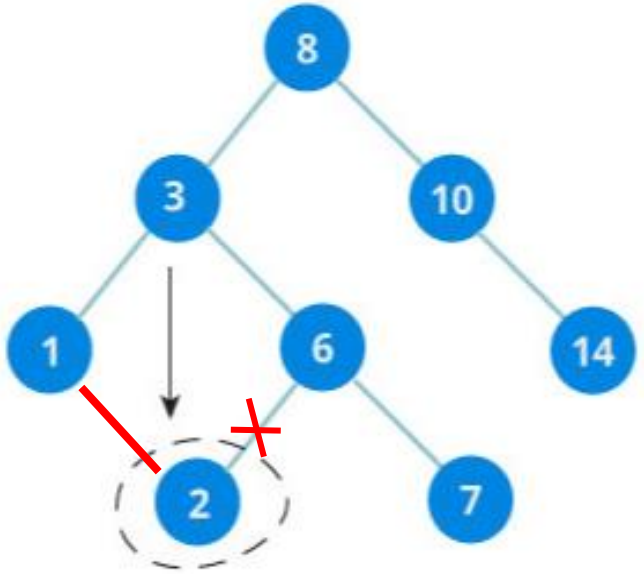
Двоичное дерево поиска





Двоичное дерево поиска





2 <= 3, а значит, должна находиться в правом поддереве узла 3

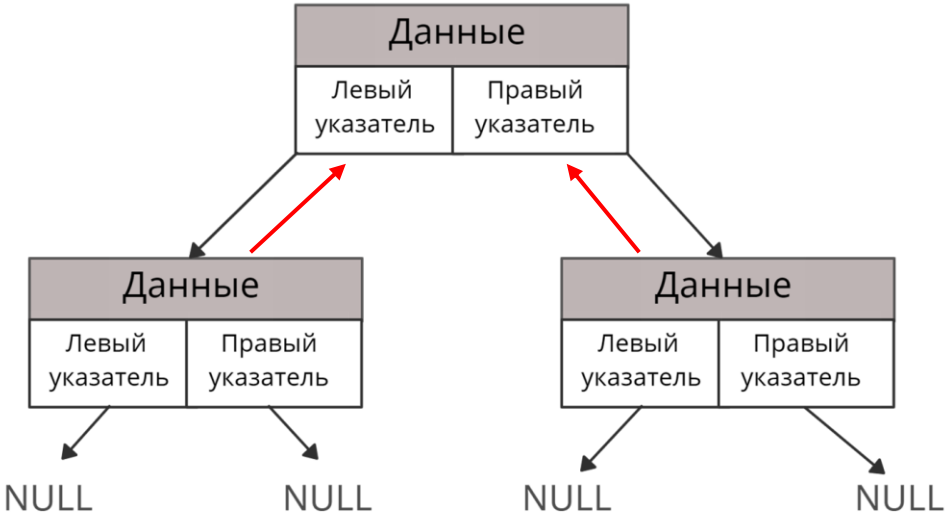
Представление двоичного дерева

Узел двоичного дерева можем представить как структуру, содержащую **данные** (значение) и **два указателя** на другие узлы (дочерние):

```
struct Node {

        int data;
        struct Node *left;
        struct Node *right;

}
```

Также можно хранить информацию о родителе:

```
struct Node {

        int data;
        struct Node *left;
        struct Node *right;
        struct Node *parent;
}
```



Использование

Предыдущие структуры *(массив, список, стек, очередь)*, которые мы уже знаем, являются **линейными** - данные хранятся последовательно, с увеличением размера растет временная сложность некоторых операций.

Дерево же структура нелинейная, что позволяет быстрее и легче получать доступ к данным. Отсюда следует их применение:

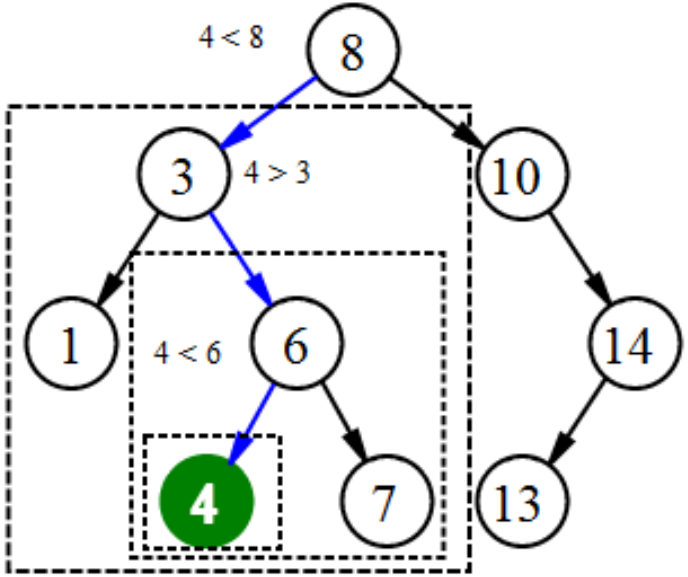
- **двоичное дерево поиска**
- куча (сортировка кучей)
- различные виды деревьев используются в базах данных, маршрутизаторах, более сложных системах

Использование

Высокая эффективность реализации алгоритмов поиска и сортировки.

Двоичное дерево **поиска** применяется для построения более абстрактных структур:

- множества;
- мультимножества;
- ассоциативные массивы;



Пример: поиск числа **4**
Зная, что для каждого узла значения узлов в левом поддереве меньше, а в правом - больше, можем достаточно быстро **(за O(logN))** найти нужный узел.

Операции с двоичным деревом поиска

- Обход дерева поиска
- Поиск элемента
- Поиск минимума и максимума
- Поиск следующего и предыдущего элемента
- Вставка элемента
- Удаление элемента

Поиск элемента

1. Для каждого узла функция сравнивает значение его ключа с искомым ключом.
2. Если ключи одинаковы:
   a. то функция возвращает текущий узел,
   b. в противном случае функция вызывается рекурсивно для левого или правого поддерева.

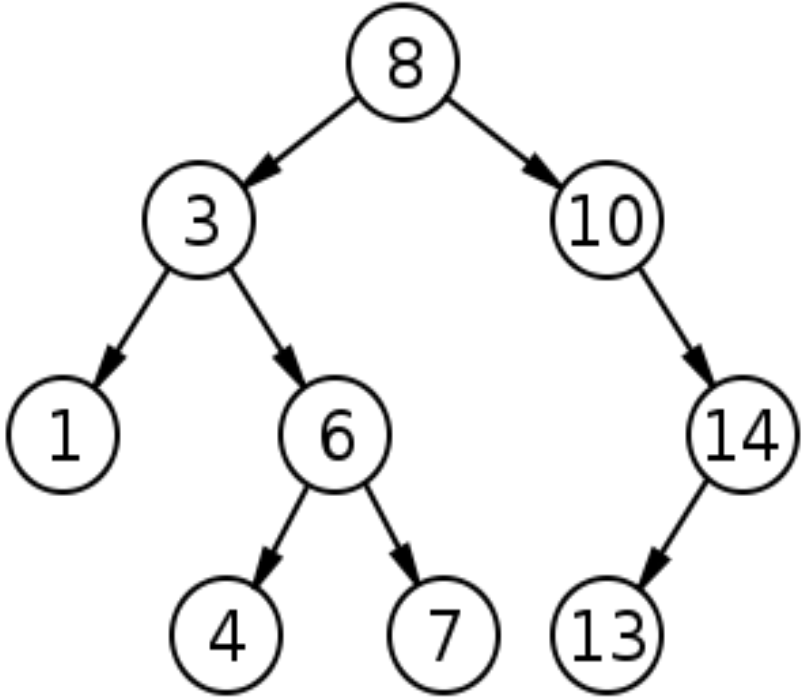
```
Node search(x : Node, k : Data):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

Узлы, которые посещает функция образуют нисходящий путь от корня, так что время ее работы **O(h)**, где **h** - высота дерева.
= **O(log N)**

Поиск элемента

Требуется найти **4**

```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

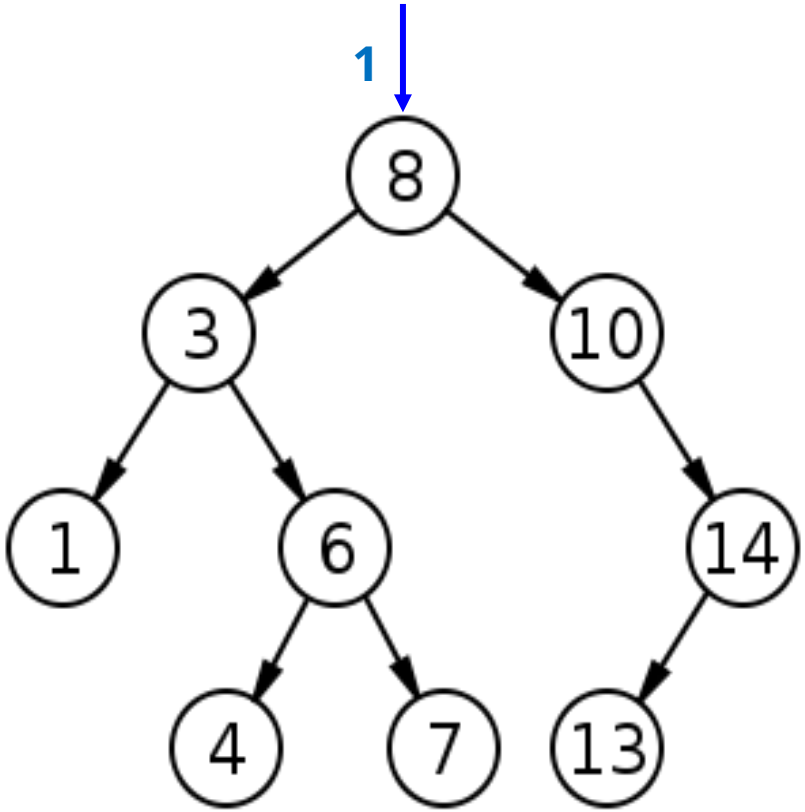


Поиск элемента

Требуется найти **4**

1. 8 == 4?

```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

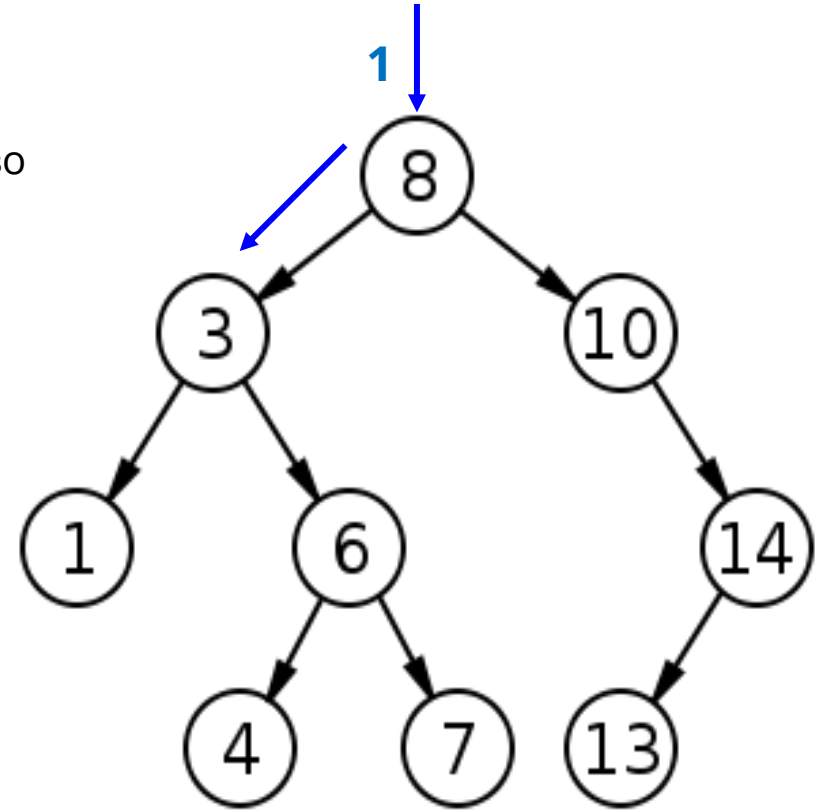


Поиск элемента

Требуется найти **4**

1. 8 == 4? Нет, 4 < 8 => идем в левое поддерево

```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

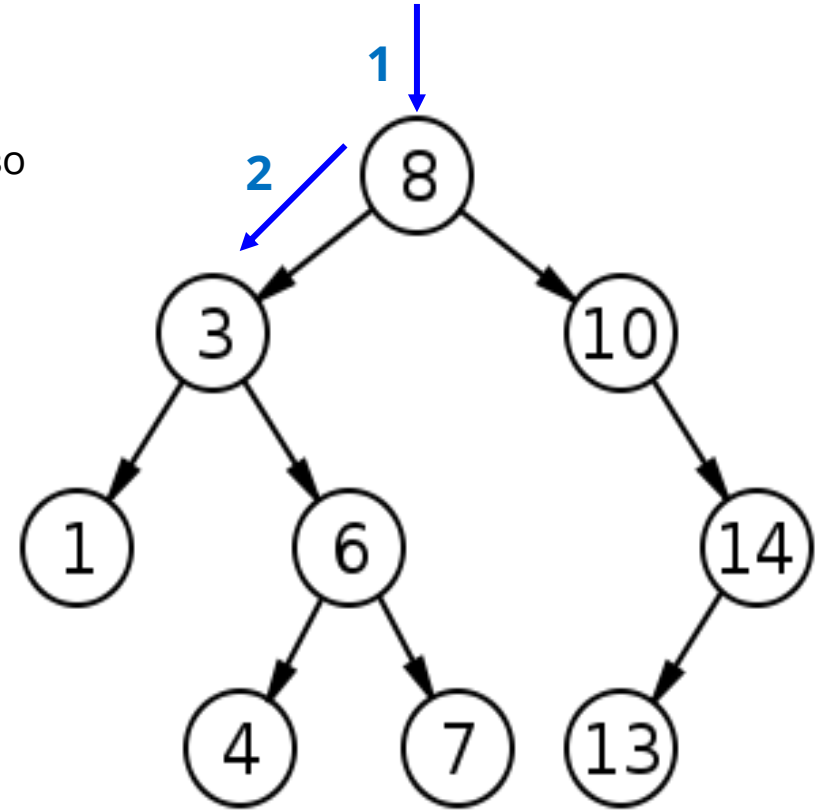


Поиск элемента

Требуется найти **4**

1. 8 == 4? Нет, 4 < 8 => идем в левое поддерево
2. 3 == 4?

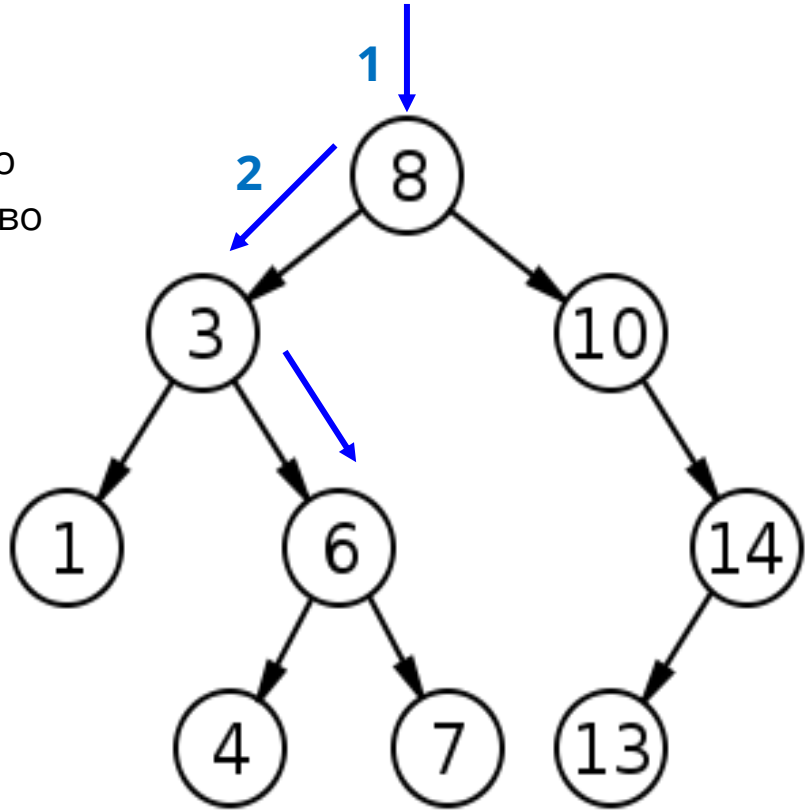
```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```



Поиск элемента

Требуется найти **4**

1. 8 == 4? Нет, 4 < 8 => идем в левое поддерево
2. 3 == 4? Нет, 4 > 3 => идем в правое поддерево



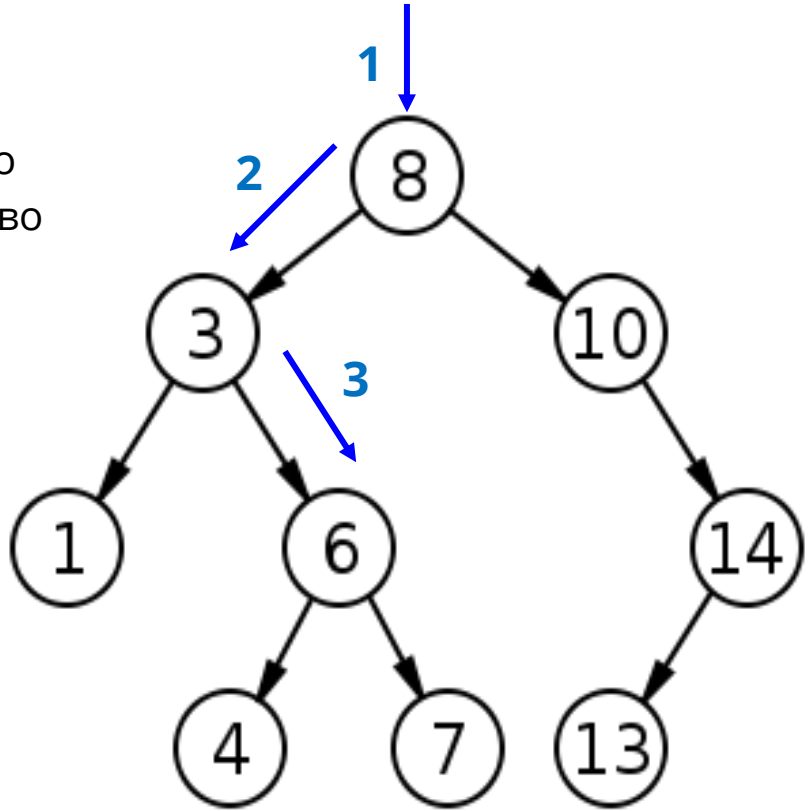

```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

Поиск элемента

Требуется найти **4**

1. 8 == 4? Нет, 4 < 8 => идем в левое поддерево
2. 3 == 4? Нет, 4 > 3 => идем в правое поддерево
3. 6 == 4?

```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

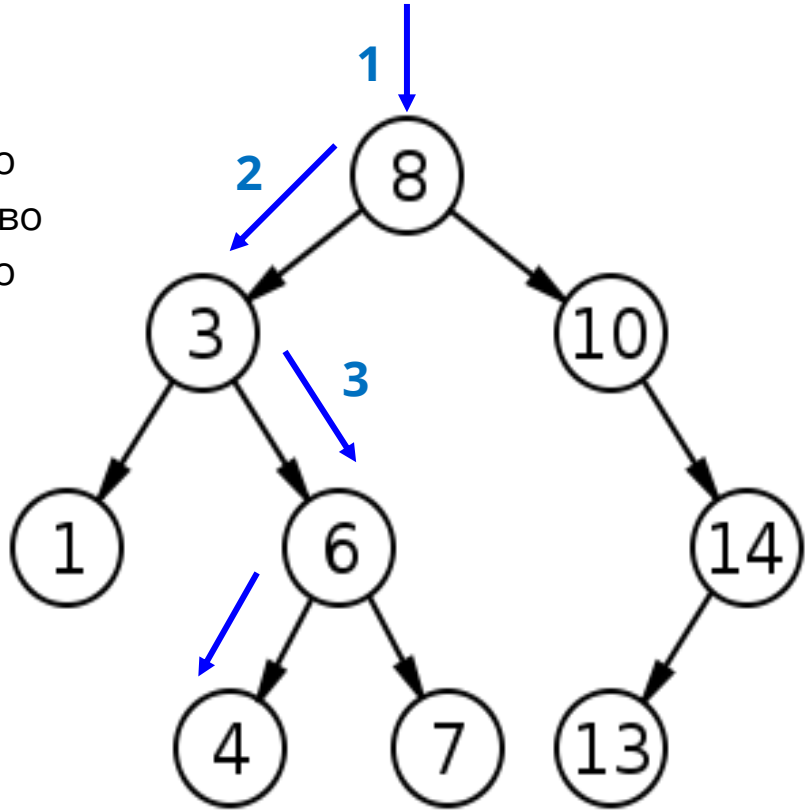


Поиск элемента

Требуется найти **4**

1. 8 == 4? Нет, 4 < 8 => идем в левое поддерево
2. 3 == 4? Нет, 4 > 3 => идем в правое поддерево
3. 6 == 4? Нет, 4 < 6 => идем в левое поддерево

```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

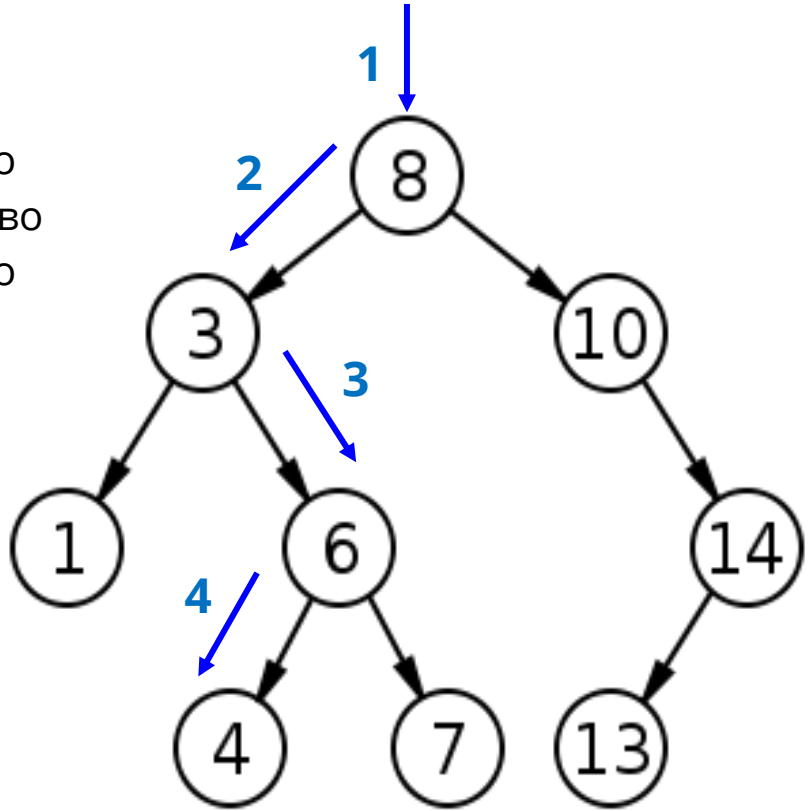


Поиск элемента

Требуется найти **4**

1. 8 == 4? Нет, 4 < 8 => идем в левое поддерево
2. 3 == 4? Нет, 4 > 3 => идем в правое поддерево
3. 6 == 4? Нет, 4 < 6 => идем в левое поддерево
4. 4 == 4?

```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

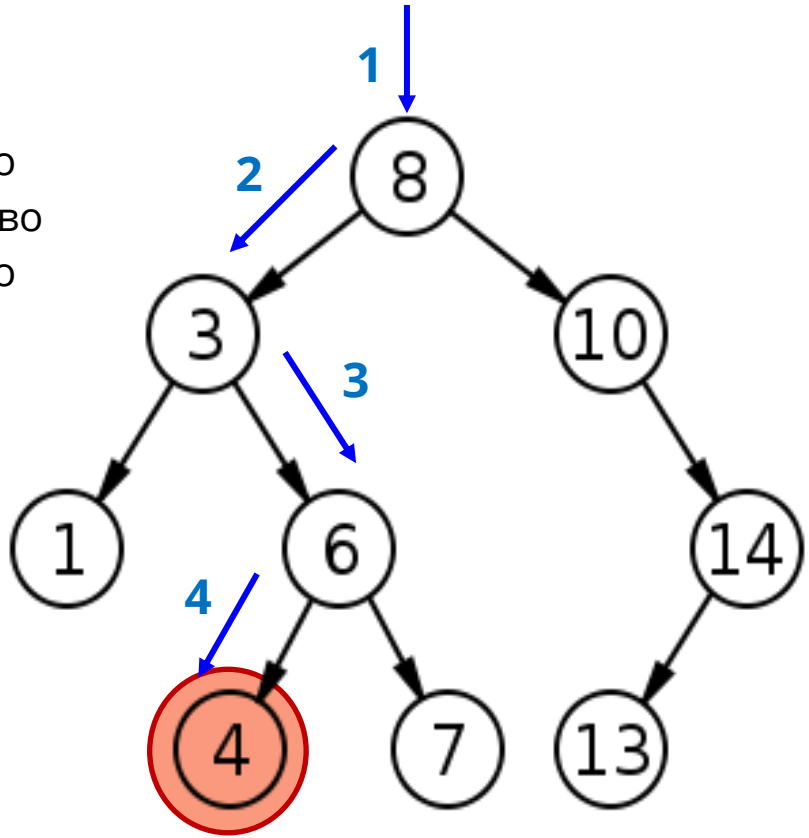


Поиск элемента

Требуется найти **4**

1. 8 == 4? Нет, 4 < 8 => идем в левое поддерево
2. 3 == 4? Нет, 4 > 3 => идем в правое поддерево
3. 6 == 4? Нет, 4 < 6 => идем в левое поддерево
4. 4 == 4? Да, элемент найден!

```
Node search(x : Node, k : T):
    if x == null or k == x.key
        return x
    if k < x.key
        return search(x.left, k)
    else
        return search(x.right, k)
```

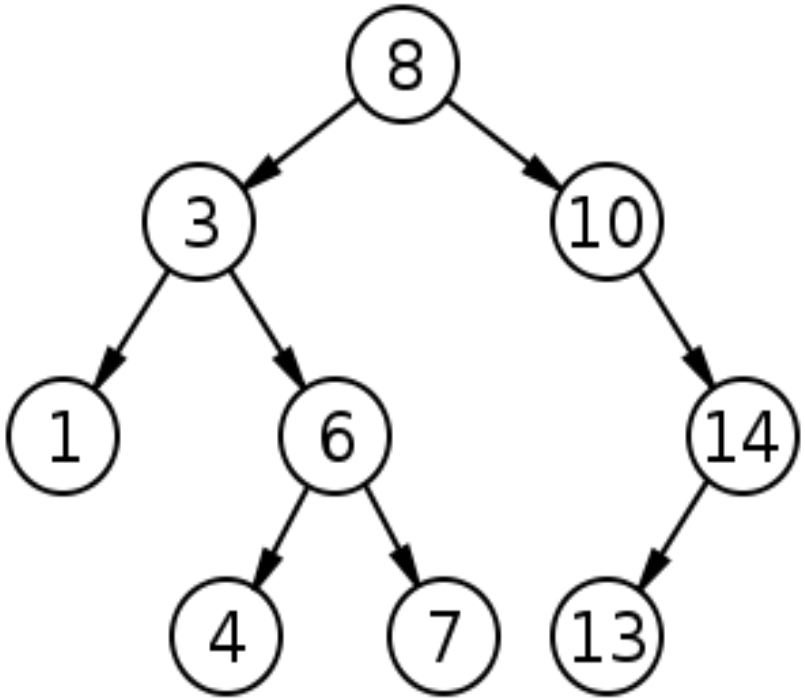


Вставка элемента

Нужно найти куда вставить

- Найти место вставки
- Создать новый элемент (ноду)
- Связать его с деревом

Задача: вставить 9

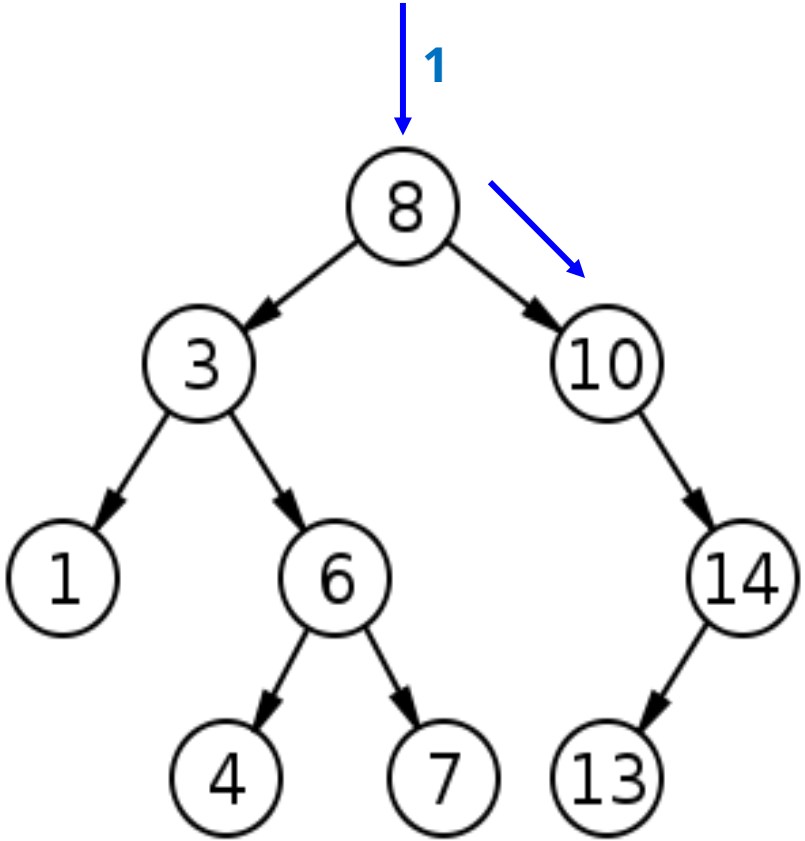


Вставка элемента

Задача: вставить 9

1. Узел со значением **8 != null** и **9 > 8**

   => запускаемся от правого поддерева



Вставка элемента

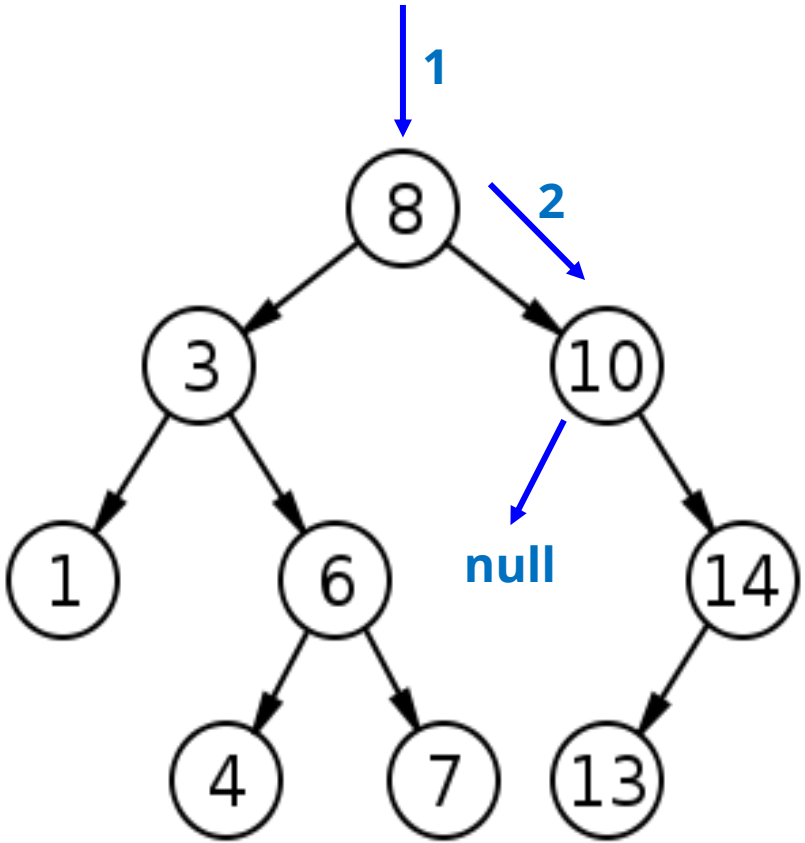
Задача: вставить 9

1. Узел со значением **8 != null** и **9 > 8**

   => запускаемся от правого поддерева

2. Узел со значением **10 != null** и **9 < 10**

   => запускаемся от левого поддерева



Вставка элемента

Задача: вставить 9

1. Узел со значением **8 != null** и **9 > 8**

   => запускаемся от правого поддерева

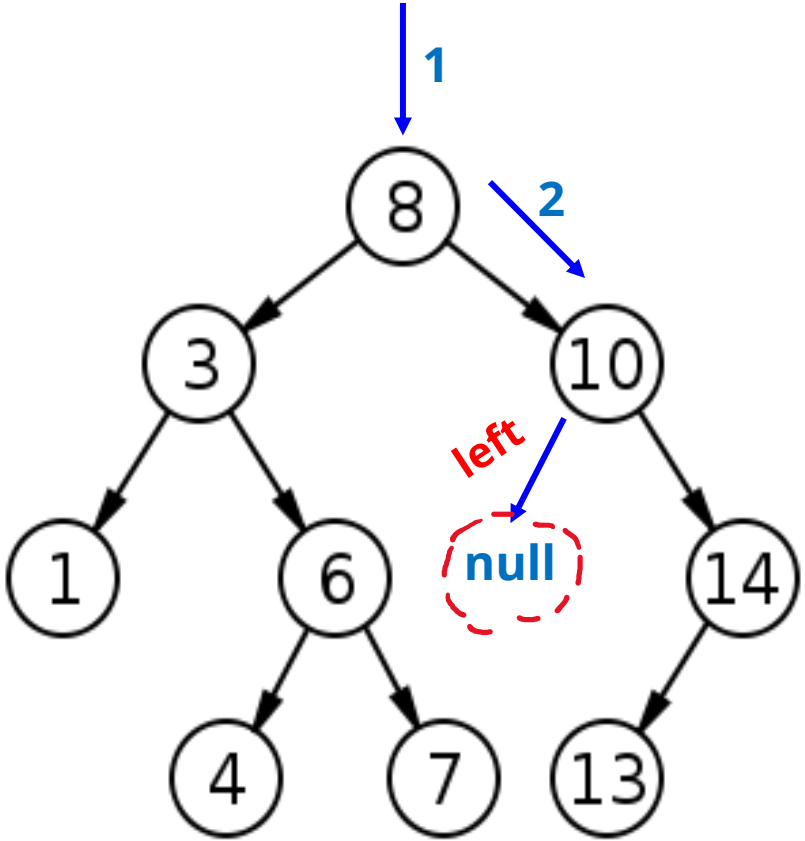
2. Узел со значением **10 != null** и **9 < 10**

   => запускаемся от левого поддерева

3. Так как **10 – > left** указывает на **null**

(информируя об отсутствии левого ребенка),

попадаем в первое условие **if x == null** (поиска)



Вставка элемента

Задача: вставить 9

1. Узел со значением **8 != null** и **9 > 8**

   => запускаемся от правого поддерева

2. Узел со значением **10 != null** и **9 < 10**

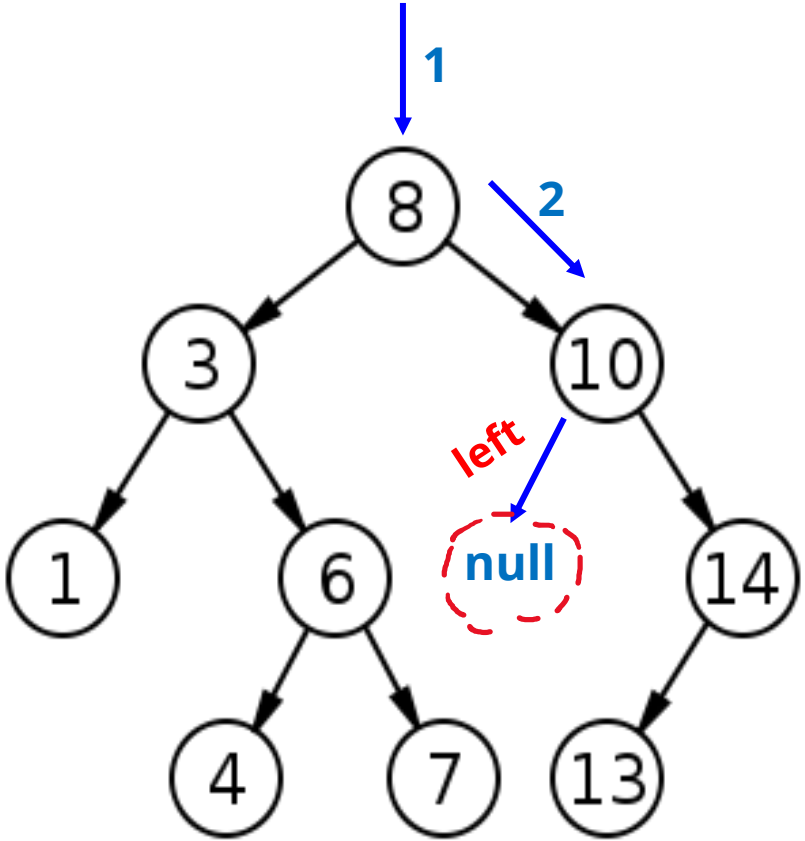
   => запускаемся от левого поддерева

3. Так как **10 – > left** указывает на **null**

(информируя об отсутствии левого ребенка),

попадаем в первое условие **if x == null** (поиска)

=> мы **нашли место** для вставки нашего элемента



Вставка элемента

Задача: вставить 9

1. Узел со значением **8 != null** и **9 > 8**

   => запускаемся от правого поддерева

2. Узел со значением **10 != null** и **9 < 10**

   => запускаемся от левого поддерева

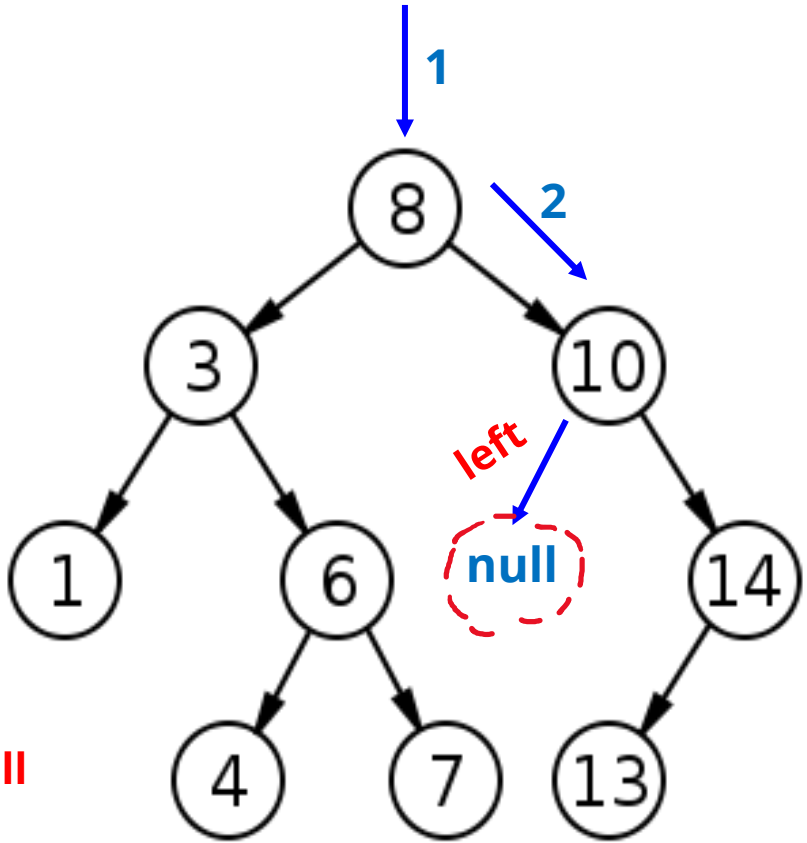
3. Так как **10 – > left** указывает на **null**

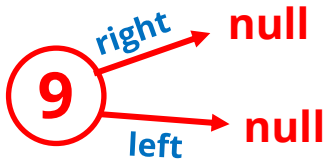
(информируя об отсутствии левого ребенка),

попадаем в первое условие **if x == null** (поиска)

⇒ мы **нашли место** для вставки нашего элемента

⇒ Создадим НОВЫЙ узел





Вставка элемента

Задача: вставить 9

1. Узел со значением **8 != null** и **9 > 8**

   => запускаемся от правого поддерева
2. Узел со значением **10 != null** и **9 < 10**

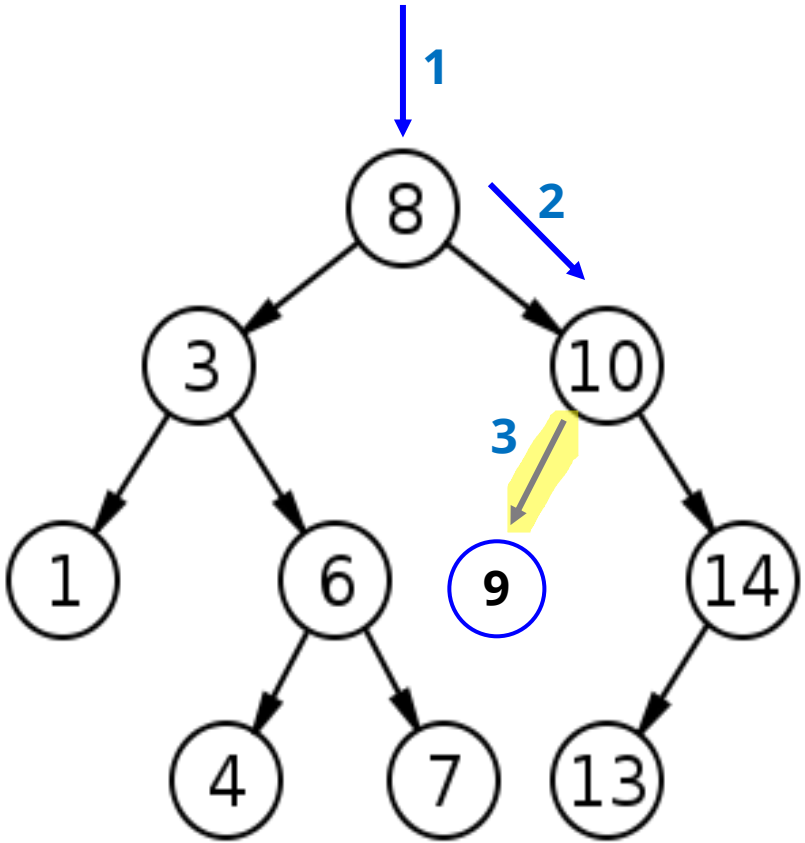
   => запускаемся от левого поддерева
3. Так как **10 – > left** указывает на **null**

(информируя об отсутствии левого ребенка),

попадаем в первое условие **if x == null** (поиска)

⇒ мы **нашли место** для вставки нашего элемента

⇒ Подвесим **10.left = новый узел (9)**



Вставка элемента

Операция вставки работает аналогично поиску элемента, только при обнаружении у элемента отсутствия ребенка нужно подвесить на него вставляемый элемент.

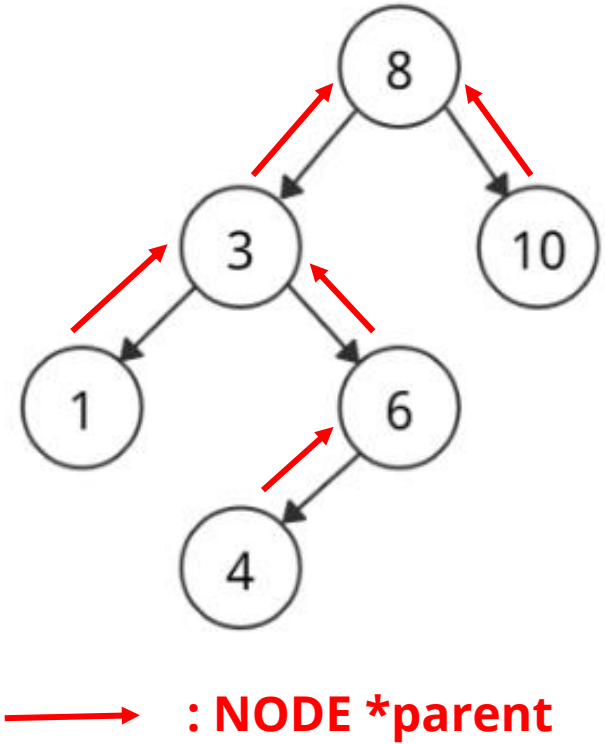
Реализация без использования информации о родителе:

```
// x — корень поддерева, z — вставляемый ключ
Node insert(x : Node, z : T):
   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
```

Вставка элемента

Вставка элемента в двоичное дерево поиска, если структура хранения с использованием информации о родителе.

Для реализации необходимо запоминать путь, по которому мы спускаемся в процессе нахождения места для вставки элемента, для того чтобы связать узлы между собой.



Вставка элемента

Вставка элемента в двоичное дерево поиска, если структура хранения с использованием информации о родителе.

Для реализации необходимо запоминать путь, по которому мы спускаемся в процессе нахождения места для вставки элемента, для того чтобы связать узлы между собой.

```
// x — корень поддерева, z — вставляемый элемент
func insert(x : Node, z : Node):
    while x != null
        if z.key > x.key
            if x.right != null
                x = x.right
            else
                z.parent = x
                x.right = z
                break
        else if z.key < x.key
            if x.left != null
                x = x.left
            else
                z.parent = x
                x.left = z
                break
```

Поиск минимума

Чтобы **найти минимальный элемент** в бинарном дереве поиска, необходимо просто следовать указателям **left** от корня дерева, пока не встретится значение **null**.

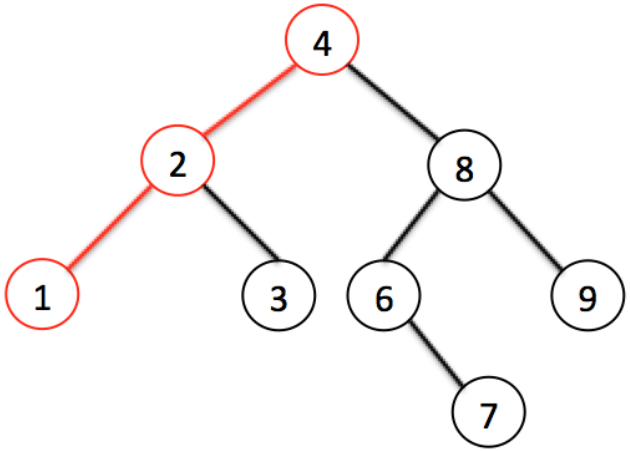
Если у вершины есть левое поддерево, то по свойству бинарного дерева поиска в нем хранятся все элементы с меньшим ключом. Если его нет, значит эта вершина и есть минимальная.

Поиск минимума

```
Node minimum(x : Node):
  if x.left == null
     return x
  return minimum(x.left)
```

Чтобы **найти минимальный элемент** в бинарном дереве поиска, необходимо просто следовать указателям **left** от корня дерева, пока не встретится значение **null**.

Если у вершины есть левое поддерево, то по свойству бинарного дерева поиска в нем хранятся все элементы с меньшим ключом. Если его нет, значит эта вершина и есть минимальная.



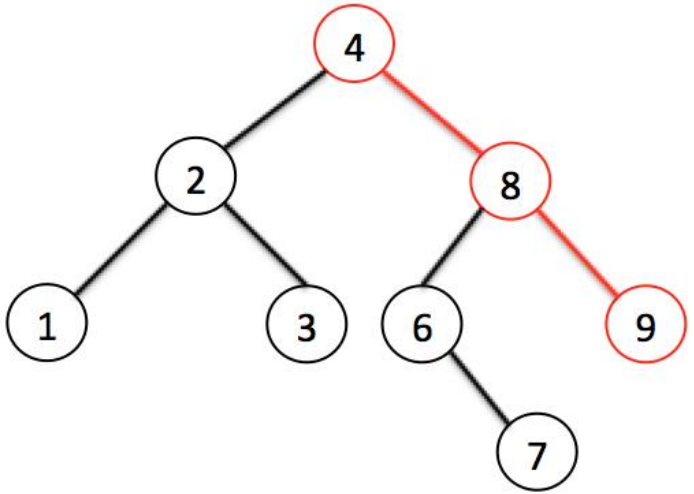
Данная функция принимает корень дерева и возвращает минимальный элемент в поддереве. Процедура работает за **O(h)**, **h** - высота дерева

Поиск максимума

```
Node maximum(x : Node):
  if x.right == null
     return x
  return maximum(x.right)
```

Чтобы **найти максимальный элемент** в бинарном дереве поиска, необходимо просто следовать указателям **right** от корня дерева, пока не встретится значение **null**.

Если у вершины есть правое поддерево, то по свойству бинарного дерева поиска в нем хранятся все элементы с большим ключом. Если его нет, значит эта вершина и есть максимальная.



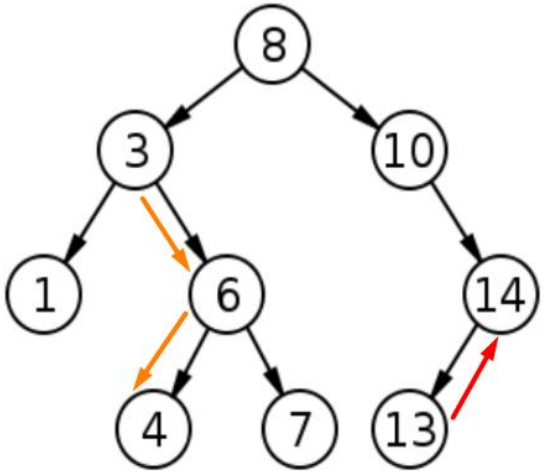
Данная функция принимает корень дерева и возвращает максимальный элемент в поддереве. Процедура работает за **O(h)**, **h** - высота дерева

Поиск следующего элемента

Следующий элемент некоторого элемента k - элемент, ключ которого минимален и больше ключа элемента k.

Следующий элемент элемента **3** - элемент **4**

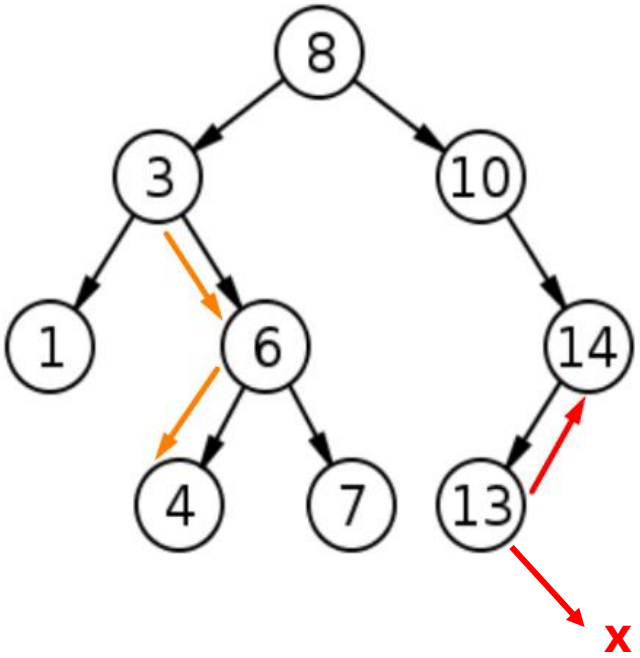
Следующий элемент элемента **13** - элемент **14**



Поиск следующего элемента

Реализация с использованием информации о родителе:

- Если у узла **есть правое поддерево**:
  - следующий за ним элемент будет минимальным элементом в этом поддереве.
- Если у него **нет правого поддерева**:
  - нужно следовать вверх, пока не встретим узел, который является левым дочерним узлом своего родителя



Поиск следующего элемента

Реализация с использованием информации о родителе:

- Если у узла **есть правое поддерево**:
  - следующий за ним элемент будет минимальным элементом в этом поддереве.
- Если у него **нет правого поддерева**:
  - нужно следовать вверх, пока не встретим узел, который является левым дочерним узлом своего родителя

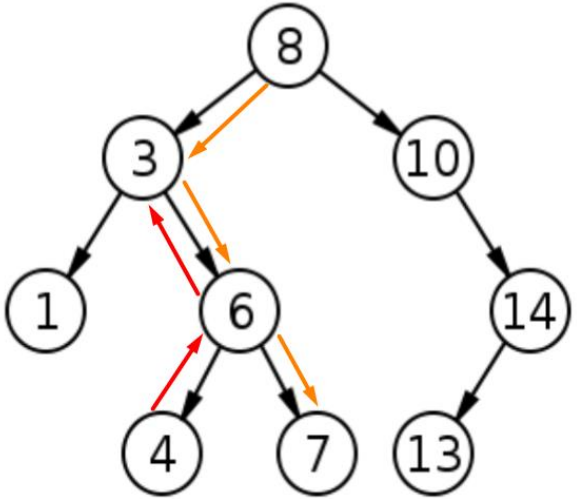
```
Node next(x : Node):
    if x.right != null
        return minimum(x.right)
    y = x.parent
    while y != null and x == y.right
        x = y
        y = y.parent
    return y
```

Поиск предыдущего элемента

Предыдущий элемент некоторого элемента k - элемент, ключ которого максимален и меньше ключа элемента k.

Предыдущий элемент элемента **8** - элемент **7**

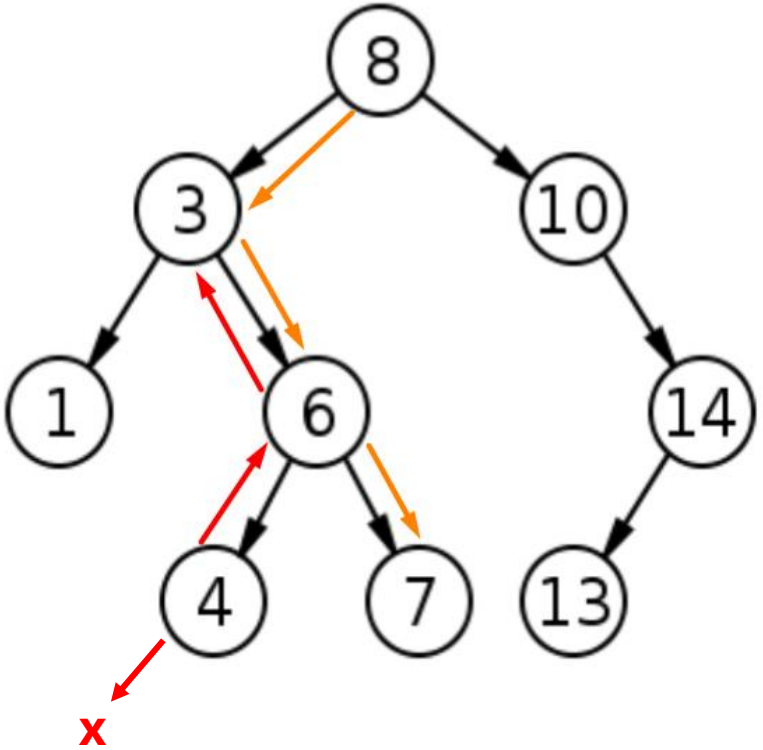
Предыдущий элемент элемента **4** - элемент **3**



Поиск предыдущего элемента

Реализация с использованием информации о родителе:

- Если у узла **есть левое поддерево**:
  - предыдущий ему элемент будет максимальным элементом в этом поддереве.
- Если у него **нет левого поддерева**:
  - нужно следовать вверх, пока не встретим узел, который является правым дочерним узлом своего родителя



Поиск предыдущего элемента

Реализация с использованием информации о родителе:

- Если у узла **есть левое поддерево**:
  - предыдущий ему элемент будет максимальным элементом в этом поддереве.
- Если у него **нет левого поддерева**:
  - нужно следовать вверх, пока не встретим узел, который является правым дочерним узлом своего родителя

```
Node prev(x : Node):
    if x.left != null
        return maximum(x.left)
    y = x.parent
    while y != null and x == y.left
        x = y
        y = y.parent
    return y
```

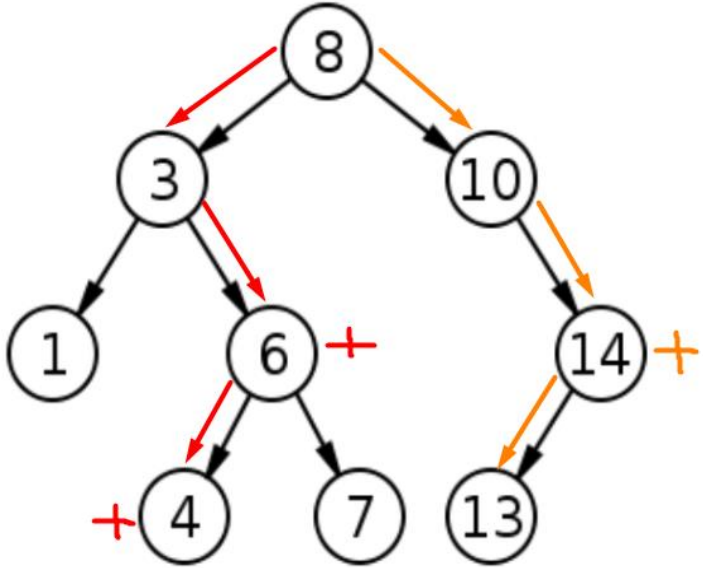
Поиск следующего элемента

Реализация без использования информации о родителе:

Следующий элемент элемента **3** - элемент **4**

Следующий элемент элемента **13** - элемент **14**

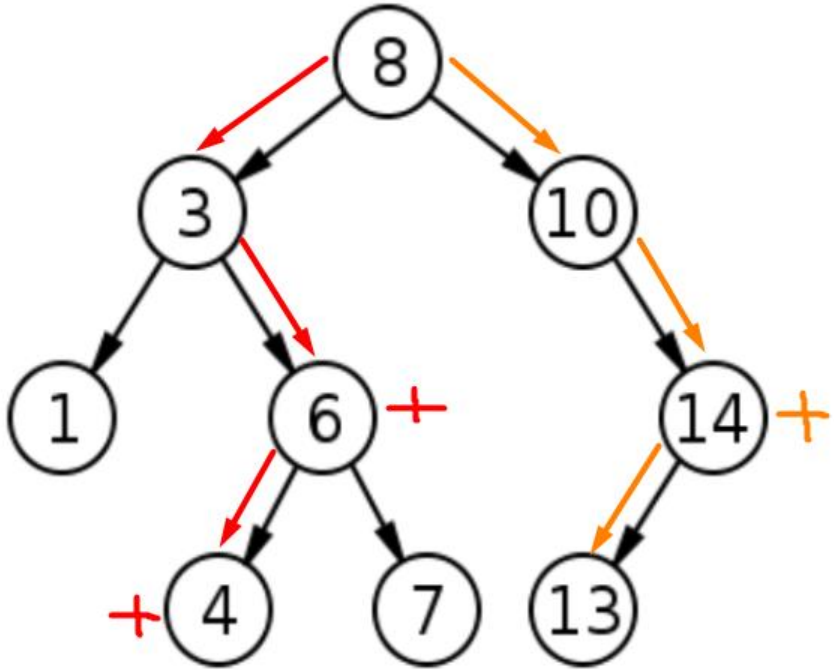
```
Node next(x : T):
// root — корень дерева
   Node current = root, successor = null
   while current != null
      if current.key > x
         successor = current
         current = current.left
      else
         current = current.right
   return successor
```



Поиск следующего элемента

Рассмотрим поиск следующего элемента для некоторого ключа **x**.

- Поиск будем начинать с корня дерева, храня текущий узел **current** и узел **successor**, последний посещенный узел, ключ которого больше **x**.
- Если **current.key⩽x**, значит следующий за x узел находится в правом поддереве (в левом поддереве все ключи меньше **current.key**).
- Если же **x<current.key**, то **x<next(x)⩽current.key**, поэтому **current** может быть следующим для ключа **x**, либо следующий узел содержится в левом поддереве **current**.



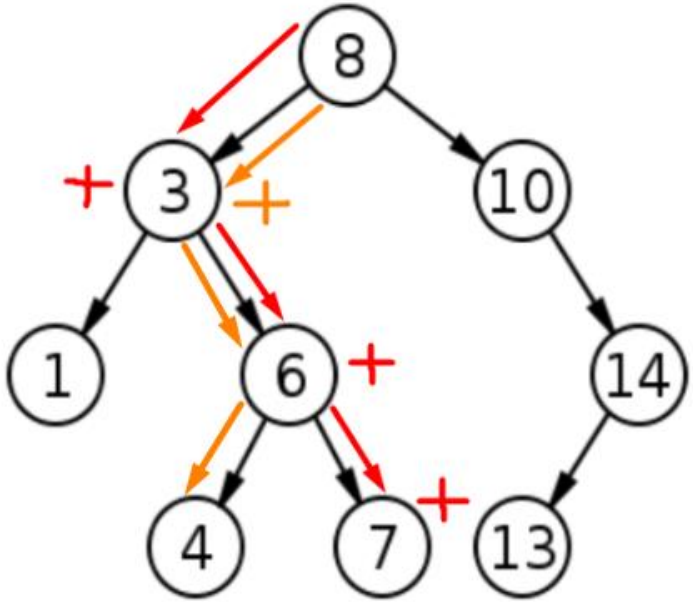
Поиск предыдущего элемента

Реализация без использования информации о родителе:

Предыдущий элемент элемента **8** - элемент **7**

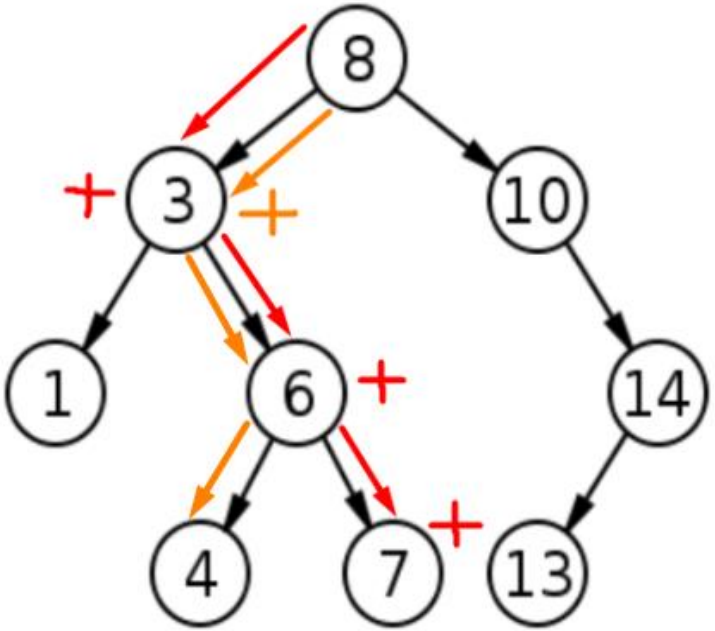
Предыдущий элемент элемента **4** - элемент **3**

```
Node prev(x : T):
// root — корень дерева
   Node current = root, successor = null
   while current != null
      if current.key >= x
         current = current.left
      else
         successor = current
         current = current.right
   return successor
```

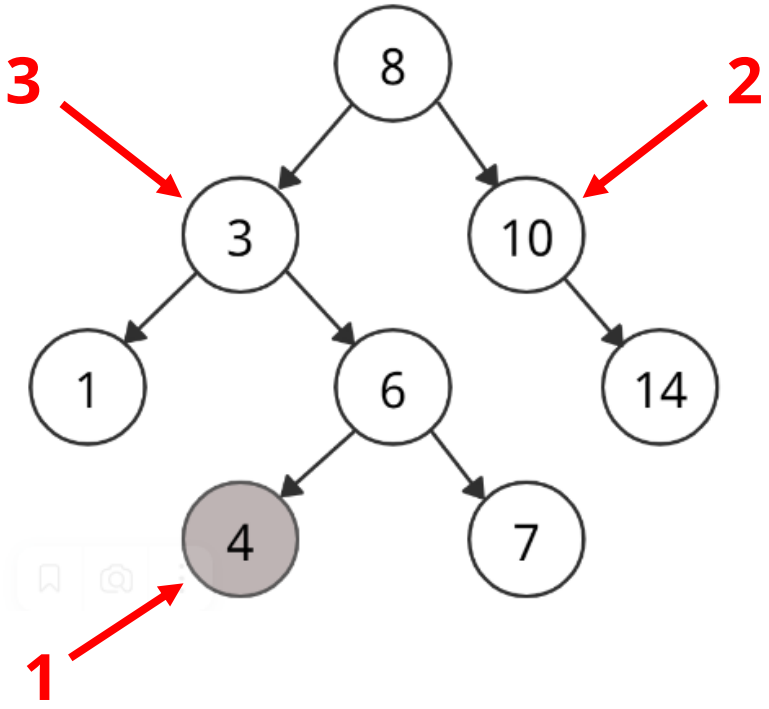


Поиск предыдущего элемента

- Если ее ключ больше или равен ключа **x**, значит, предыдущий точно в левом поддереве, т.к. он еще меньше, чем **x**.
- Как только находим такое значение, сохраняем его и дальше ищем в правом для него поддереве, т.к. в его левом будут значения еще меньше, а в правом, возможно, найдутся более близкие к **x**.



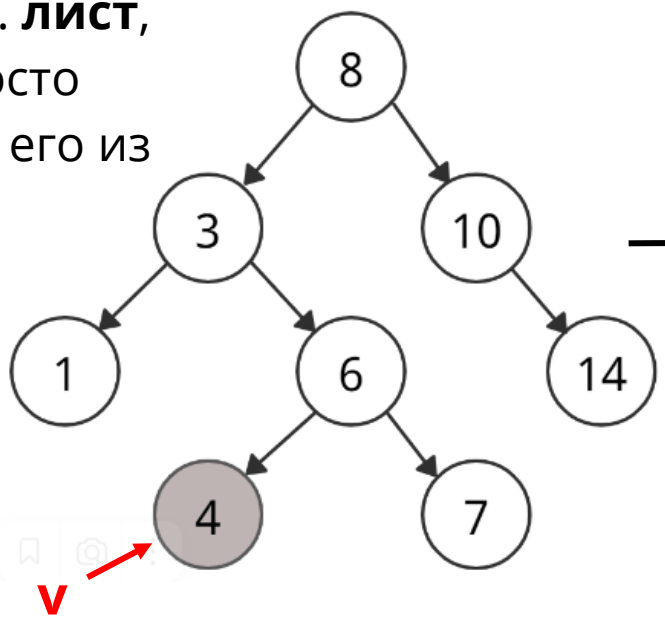
Удаление элемента. 3 случая

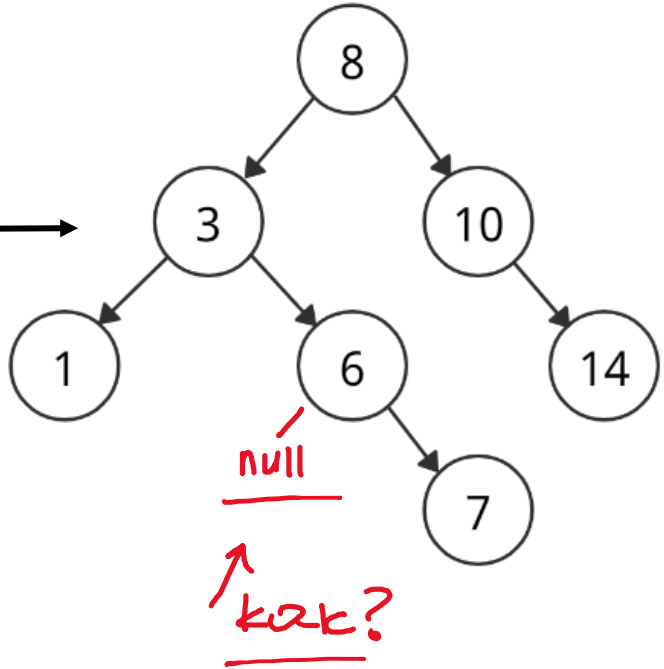


Удаление элемента. 1 случай

Удаляемый узел - узел без дочерних узлов, т.е. **лист**, тогда просто **удаляем** его из дерева

Найдем (4) = (5)

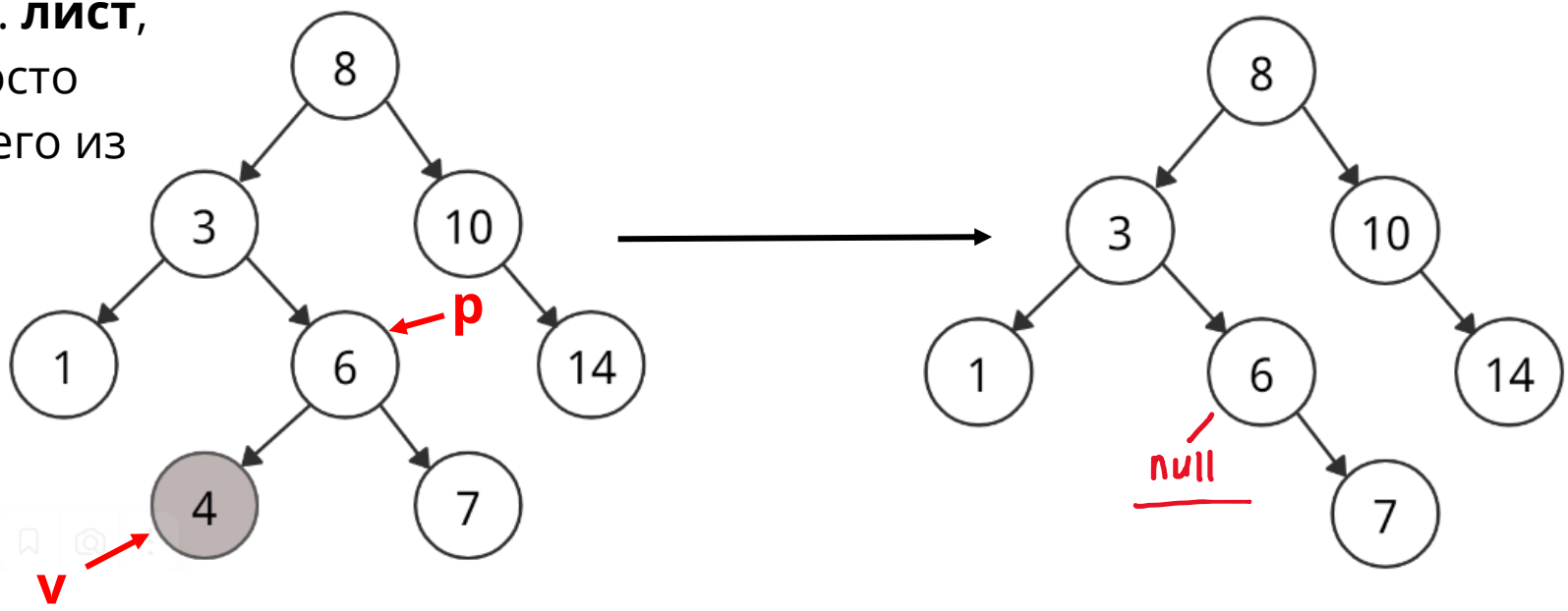




Удаление элемента. 1 случай

Удаляемый узел - узел без дочерних узлов, т.е. **лист**, тогда просто удаляем его из дерева

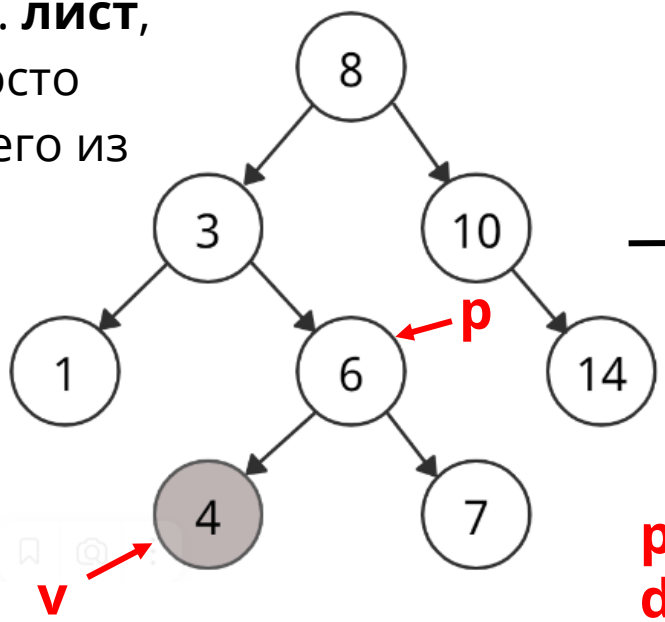
```
p = v.parent
```

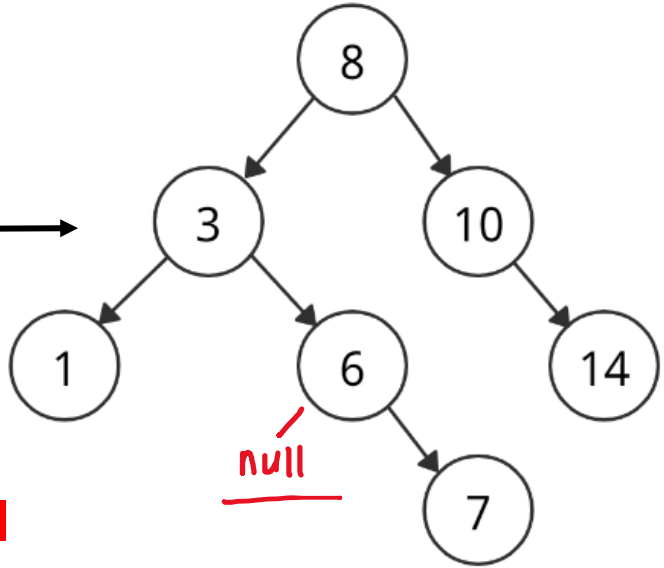


Удаление элемента. 1 случай

Удаляемый узел - узел без дочерних узлов, т.е. **лист**, тогда просто удаляем его из дерева

```
p = v.parent
```



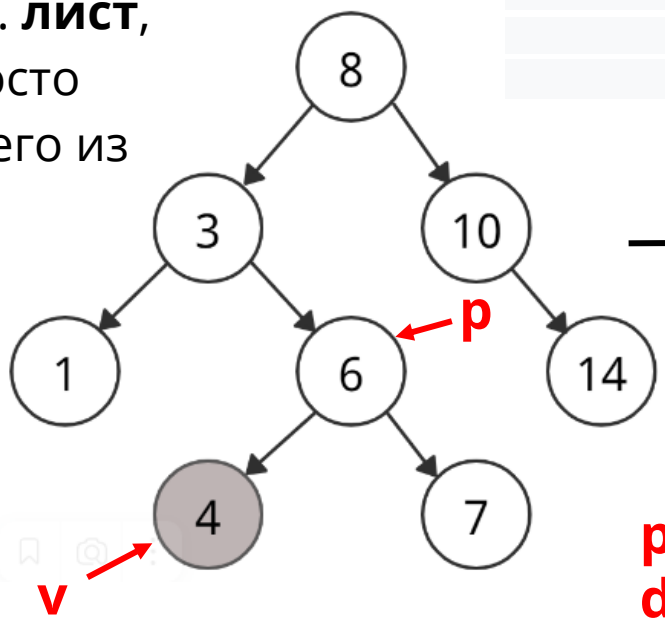


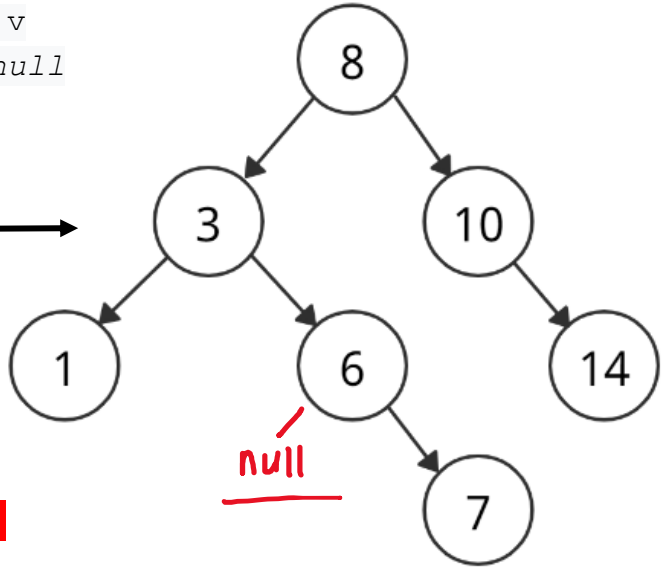
p.left = null
dispose(v)

Удаление элемента. 1 случай

Удаляемый узел - узел без дочерних узлов, т.е. **лист**, тогда просто удаляем его из дерева

```
p = v.parent
if v.left == null and v.right == null
        if p.left == v
            p.left = null
        if p.right == v
            p.right = null
```



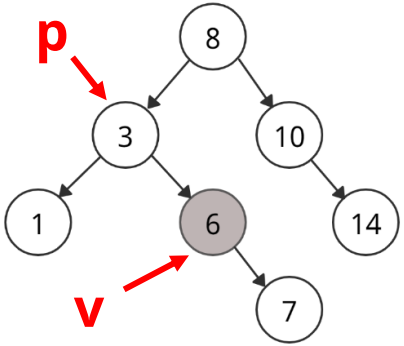


p.left = null
dispose(v)

Удаление элемента. 2 случай

Удаляемый узел - узел с **одним** дочерним узлом. В этом случае сначала заменяем удаляемый на дочерний, после чего удаляем нужный

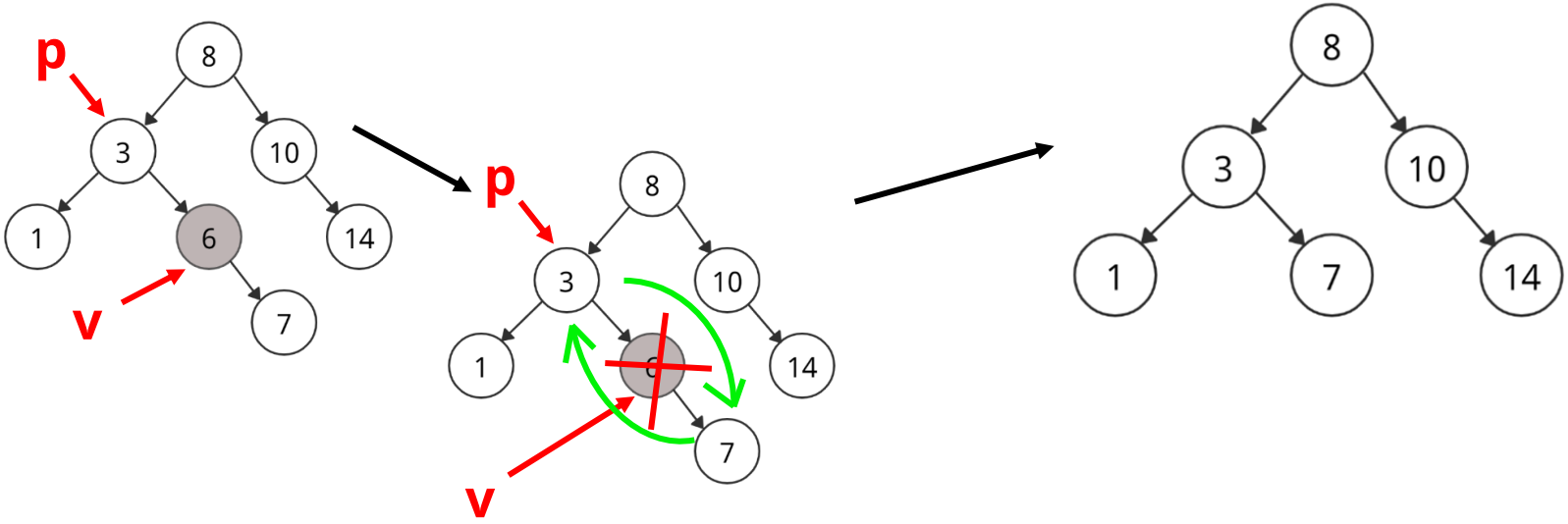
```
p = v.parent
```



Удаление элемента. 2 случай

Удаляемый узел - узел с **одним** дочерним узлом. В этом случае сначала заменяем удаляемый на дочерний, после чего удаляем нужный

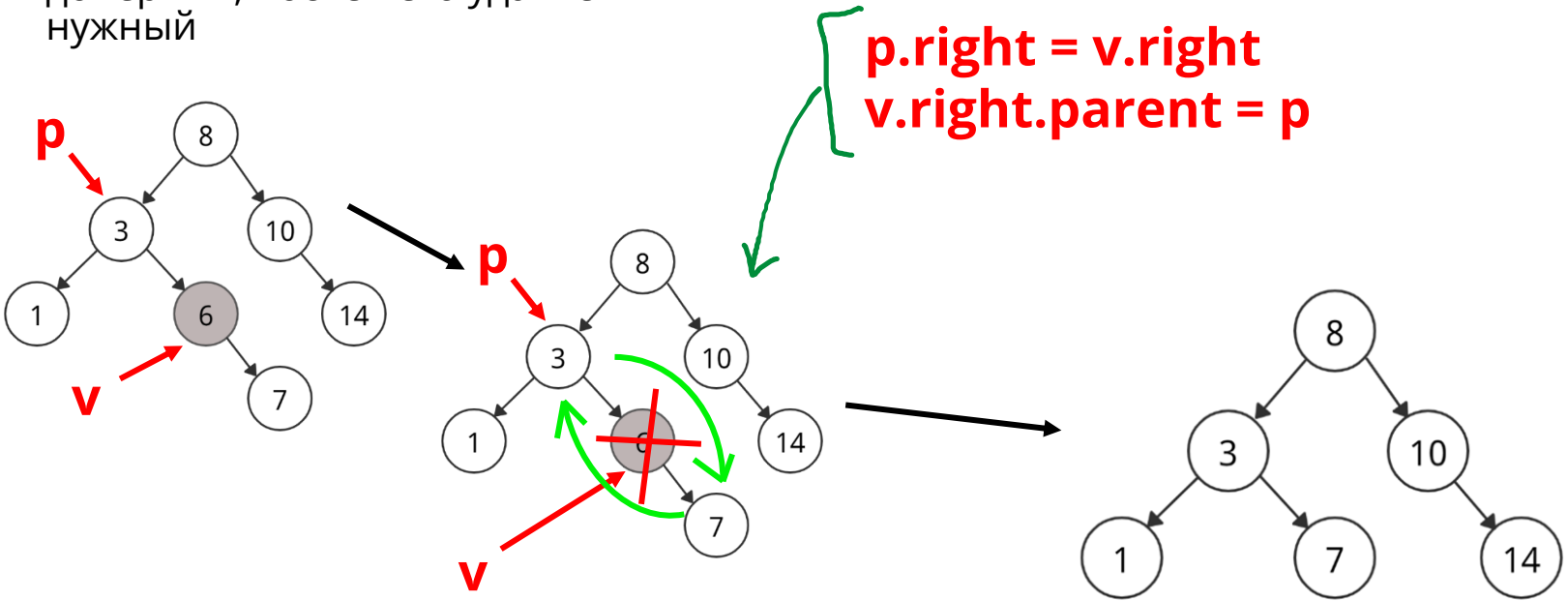
```
p = v.parent
```



Удаление элемента. 2 случай

Удаляемый узел - узел с **одним** дочерним узлом. В этом случае сначала заменяем удаляемый на дочерний, после чего удаляем нужный

```
p = v.parent
```



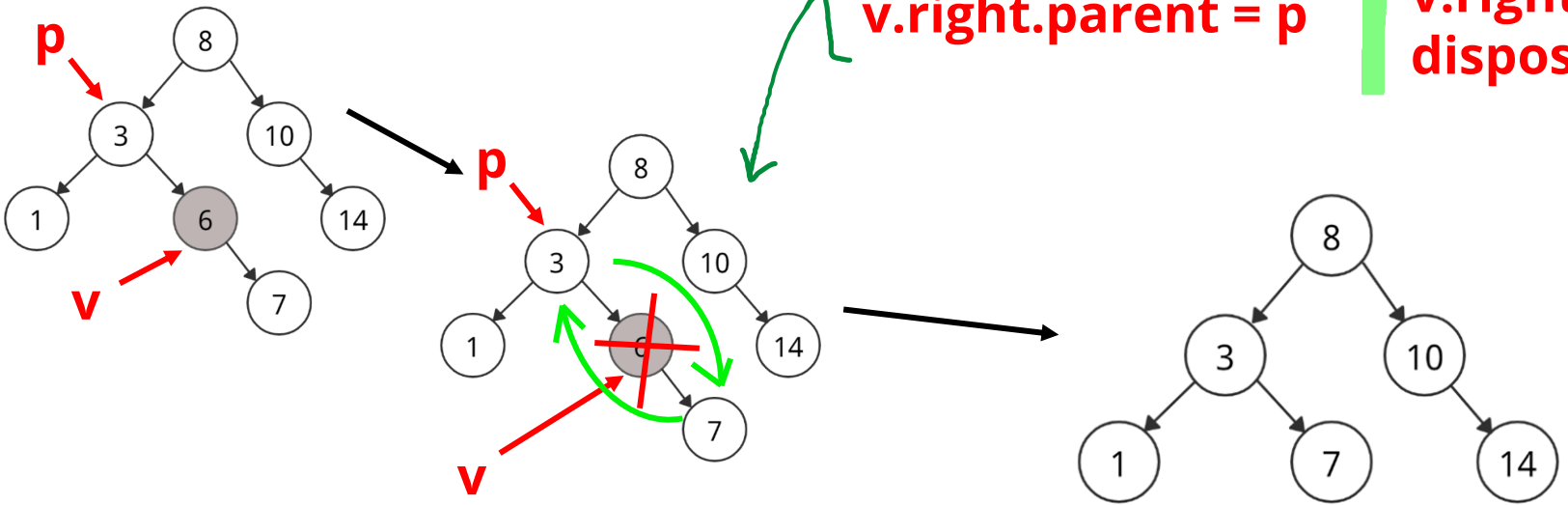
Удаление элемента. 2 случай

Удаляемый узел - узел с **одним** дочерним узлом. В этом случае сначала заменяем удаляемый на дочерний, после чего удаляем нужный

```
p = v.parent
```

p.right = v.right
v.right.parent = p

v.parent = null
v.right = null
dispose(v)

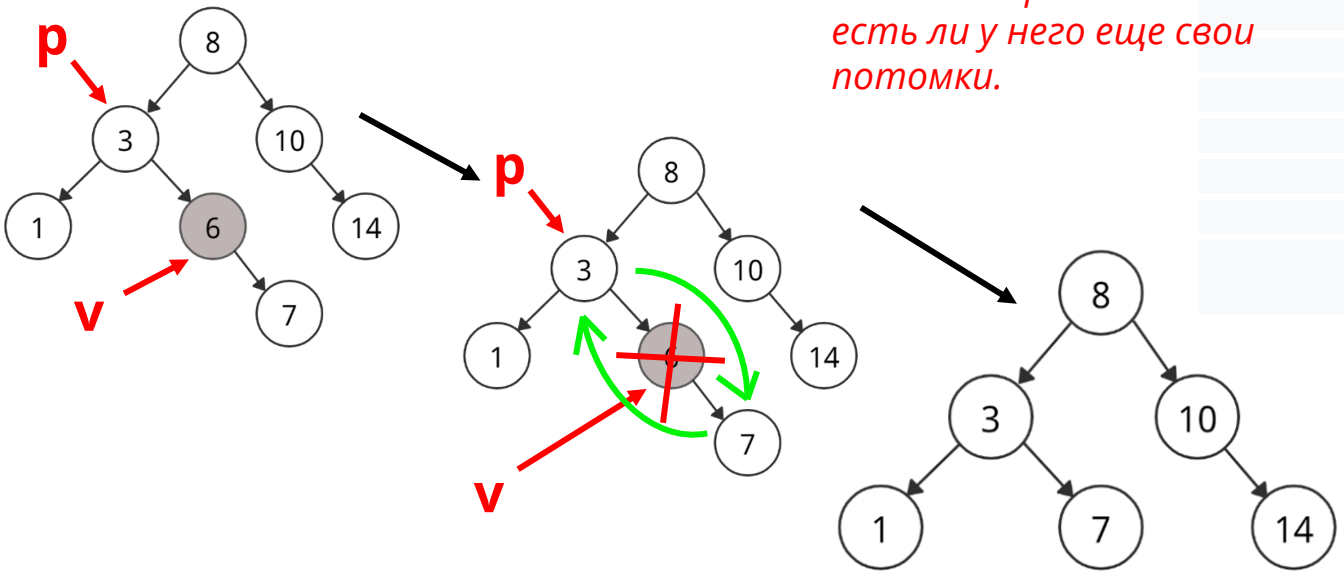


Удаление элемента. 2 случай

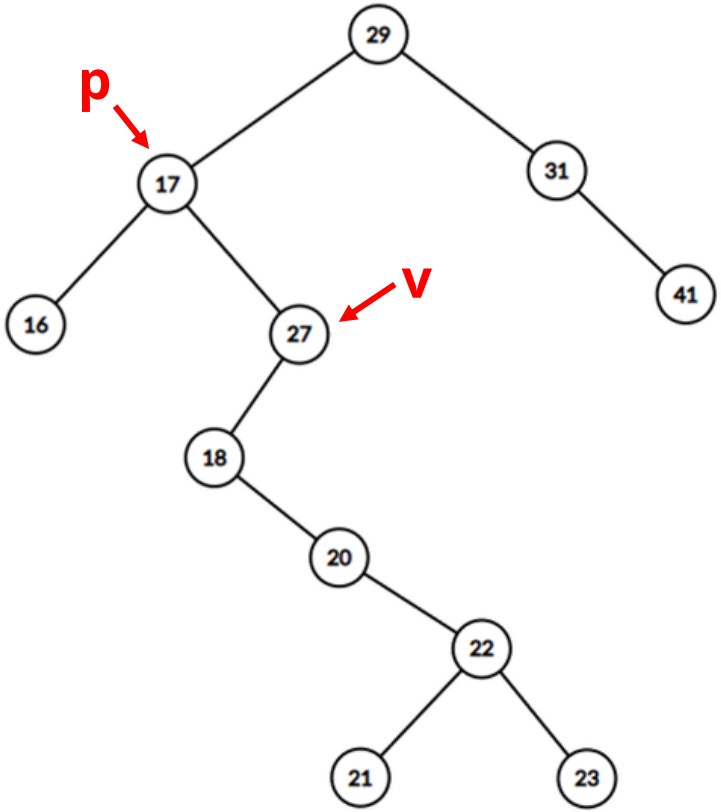
Удаляемый узел - узел с **одним** дочерним узлом. В этом случае сначала заменяем удаляемый на дочерний, после чего удаляем нужный

Нам не важно, является ли этот дочерний узел правым или левым ребенком и есть ли у него еще свои потомки.

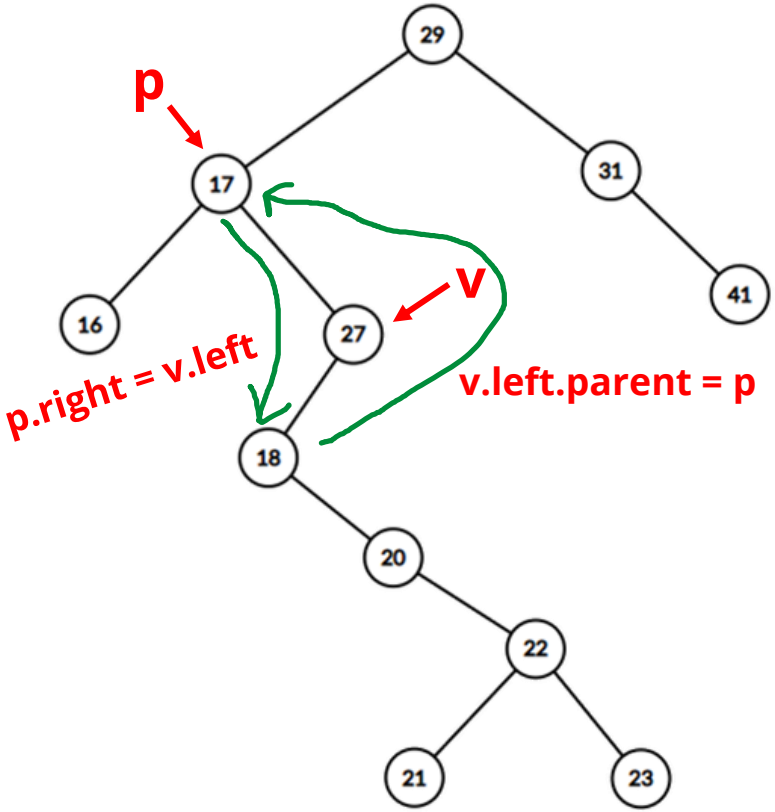
```
p = v.parent
if v.left == null or v.right == null
            if v.left == null
                if p.left == v
                        p.left = v.right
                else
                        p.right = v.right
                v.right.parent = p
            else
                if p.left == v
                        p.left = v.left
                else
                        p.right = v.left
                v.left.parent = p
```



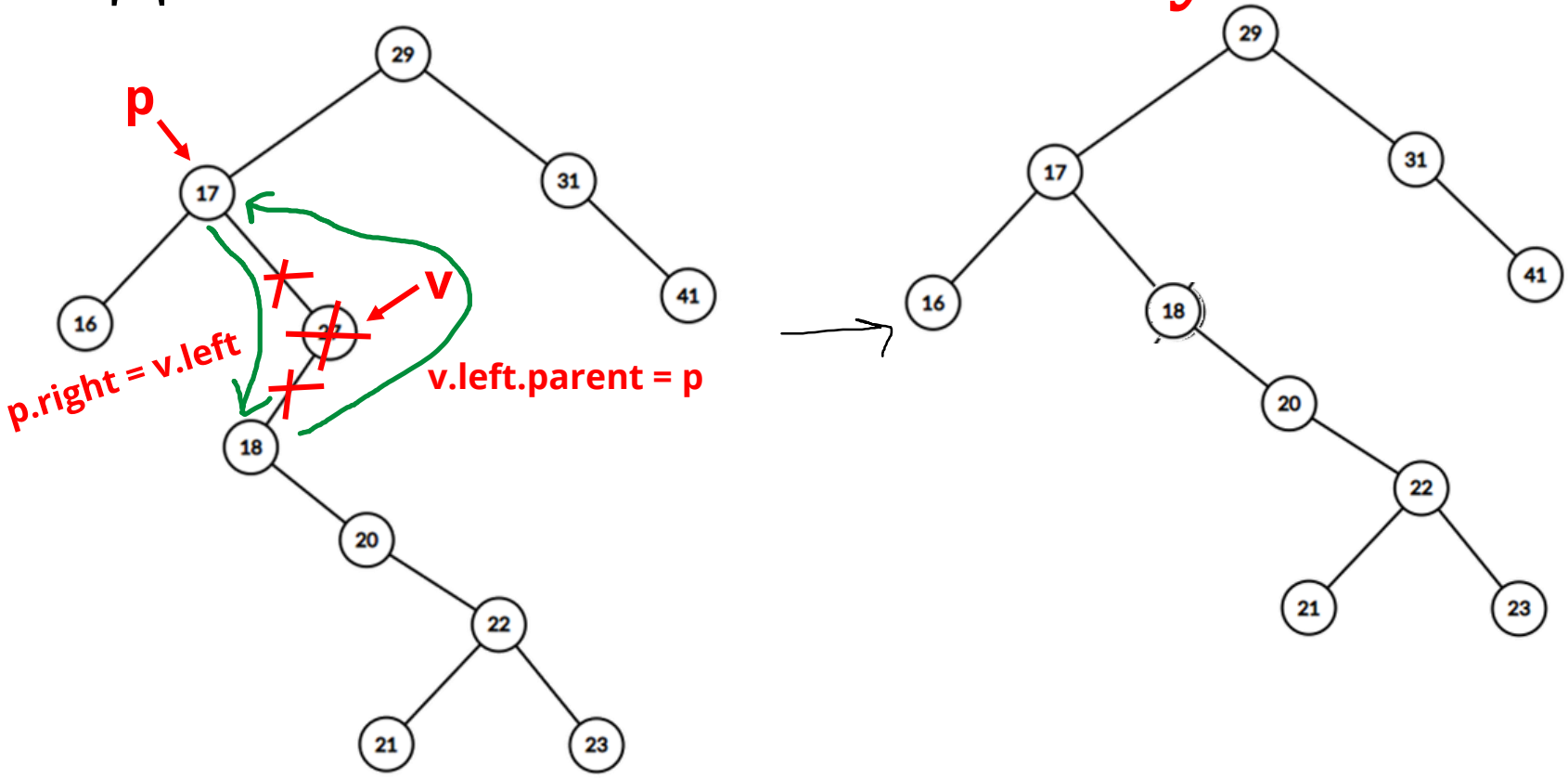
Удаление элемента. 2 случай



Удаление элемента. 2 случай

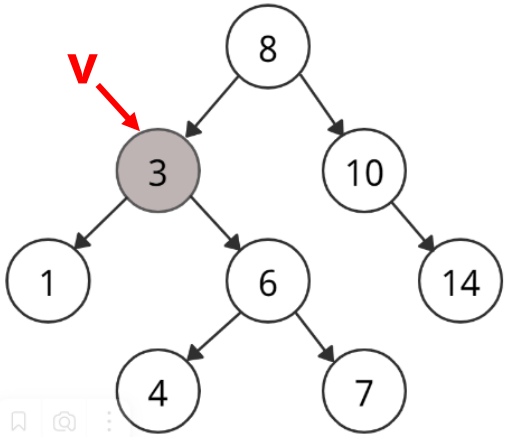


Удаление элемента. 2 случай



Удаление элемента. 3 случай

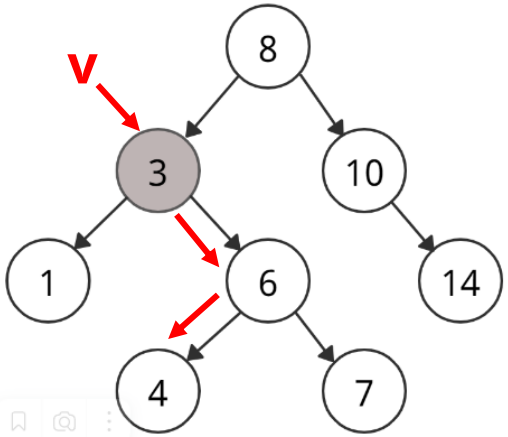
Удаляемый узел - узел с **двумя** дочерними узлами.



Гарантировано есть левое и правое поддерево

Удаление элемента. 3 случай

Удаляемый узел - узел с **двумя** дочерними узлами. В этом случае сначала находим следующий для удаляемого узла элемент и вставляем его на место удаляемого



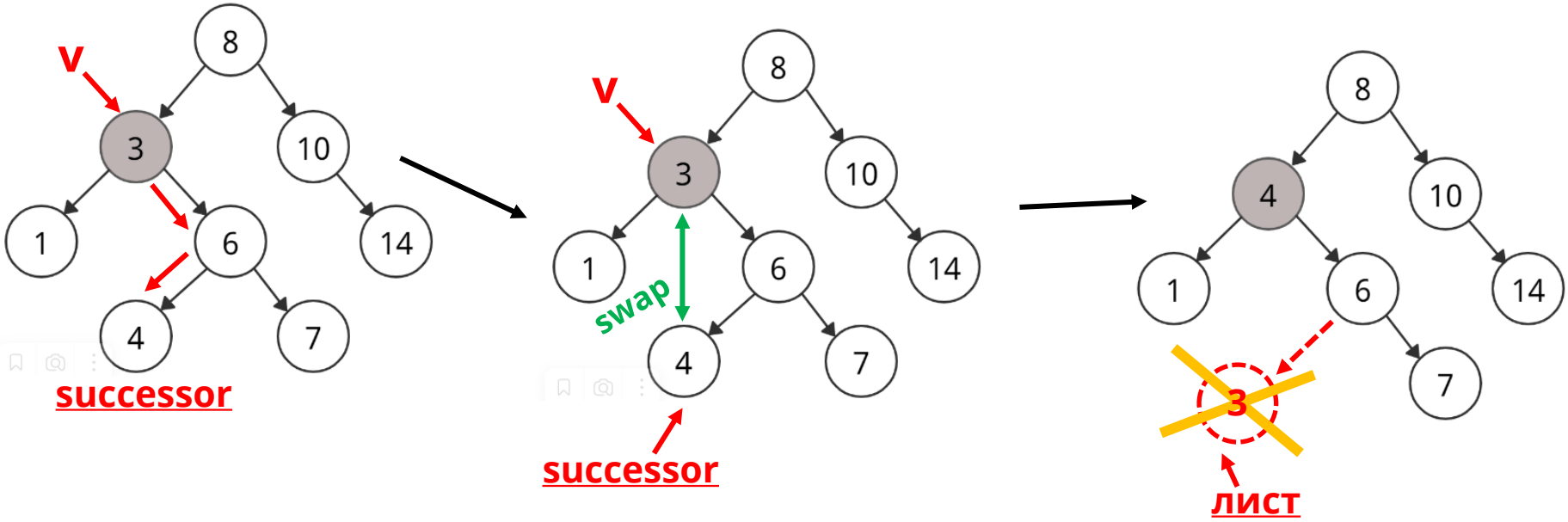
successor

Сведем к удалению **1** или **2** случая для **successor**

Удаление элемента. 3 случай

Удаляемый узел - узел с **двумя** дочерними узлами. В этом случае сначала находим следующий для удаляемого узла элемент и вставляем его на место удаляемого

swap(v.key, succ.key)

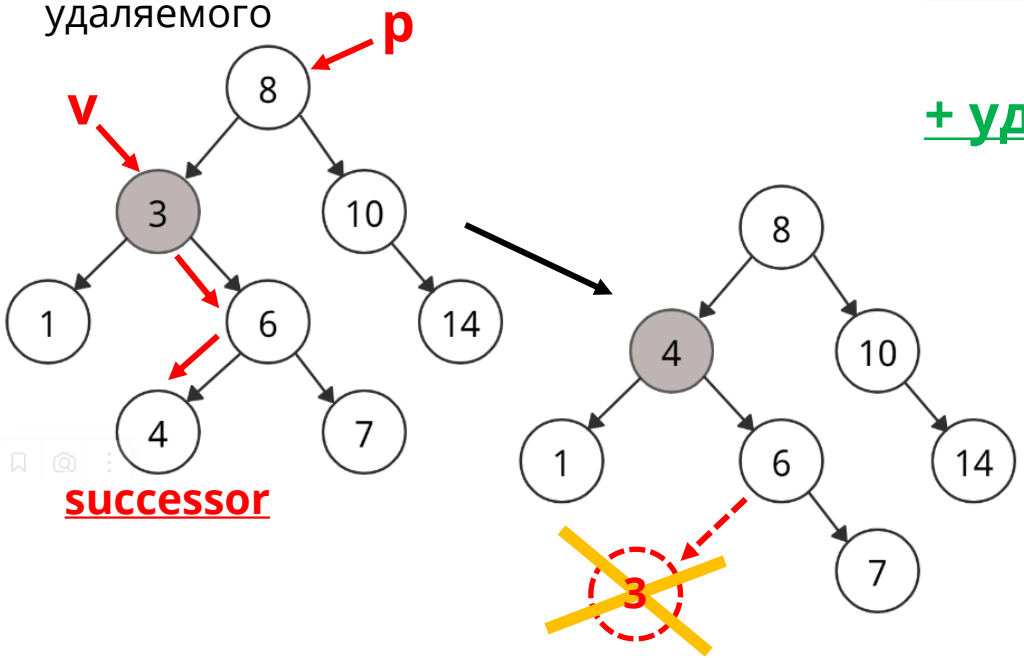


Удаление элемента. 3 случай

Удаляемый узел - узел с **двумя** дочерними узлами. В этом случае сначала находим следующий для удаляемого узла элемент и вставляем его на место удаляемого

```
p = v.parent
successor = next(v, t)
v.key = successor.key
```

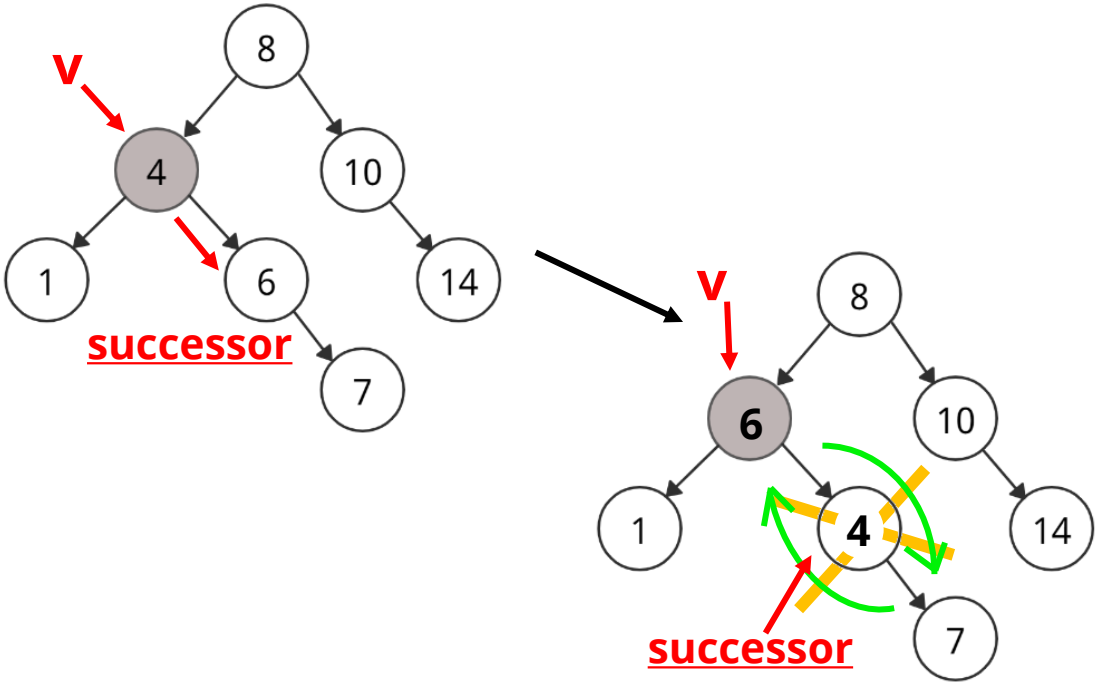
+ удаление листа 1 случай



Удаление элемента. 3 случай

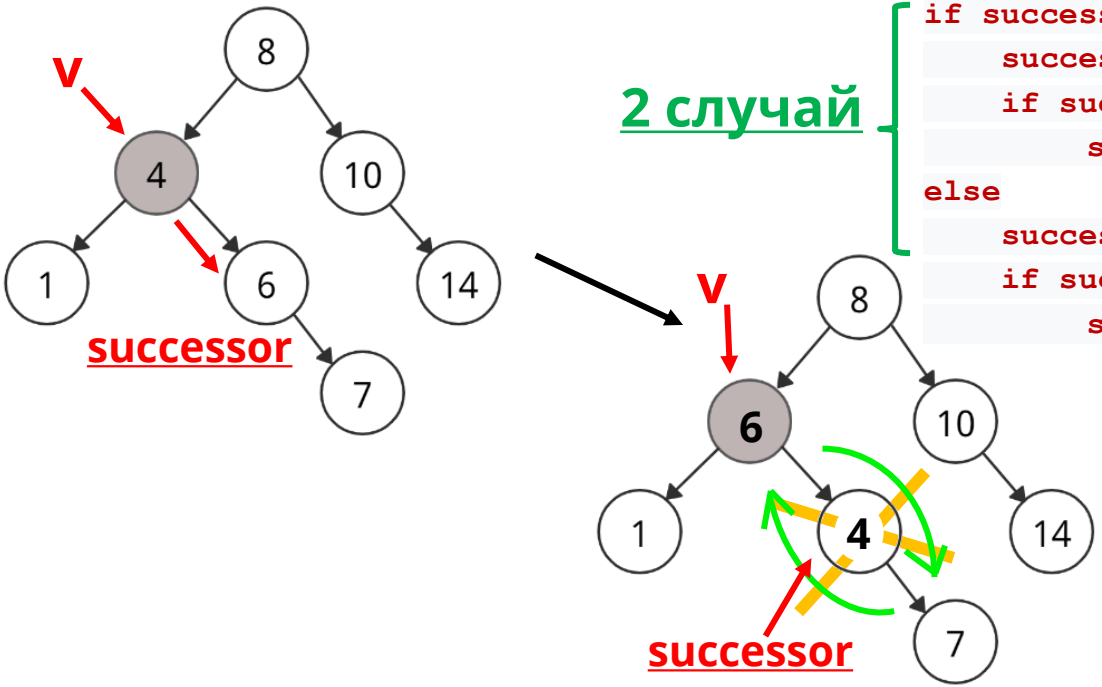
Если у следующего есть дети?

```
p = v.parent
successor = next(v, t)
v.key = successor.key
```



Удаление элемента. 3 случай

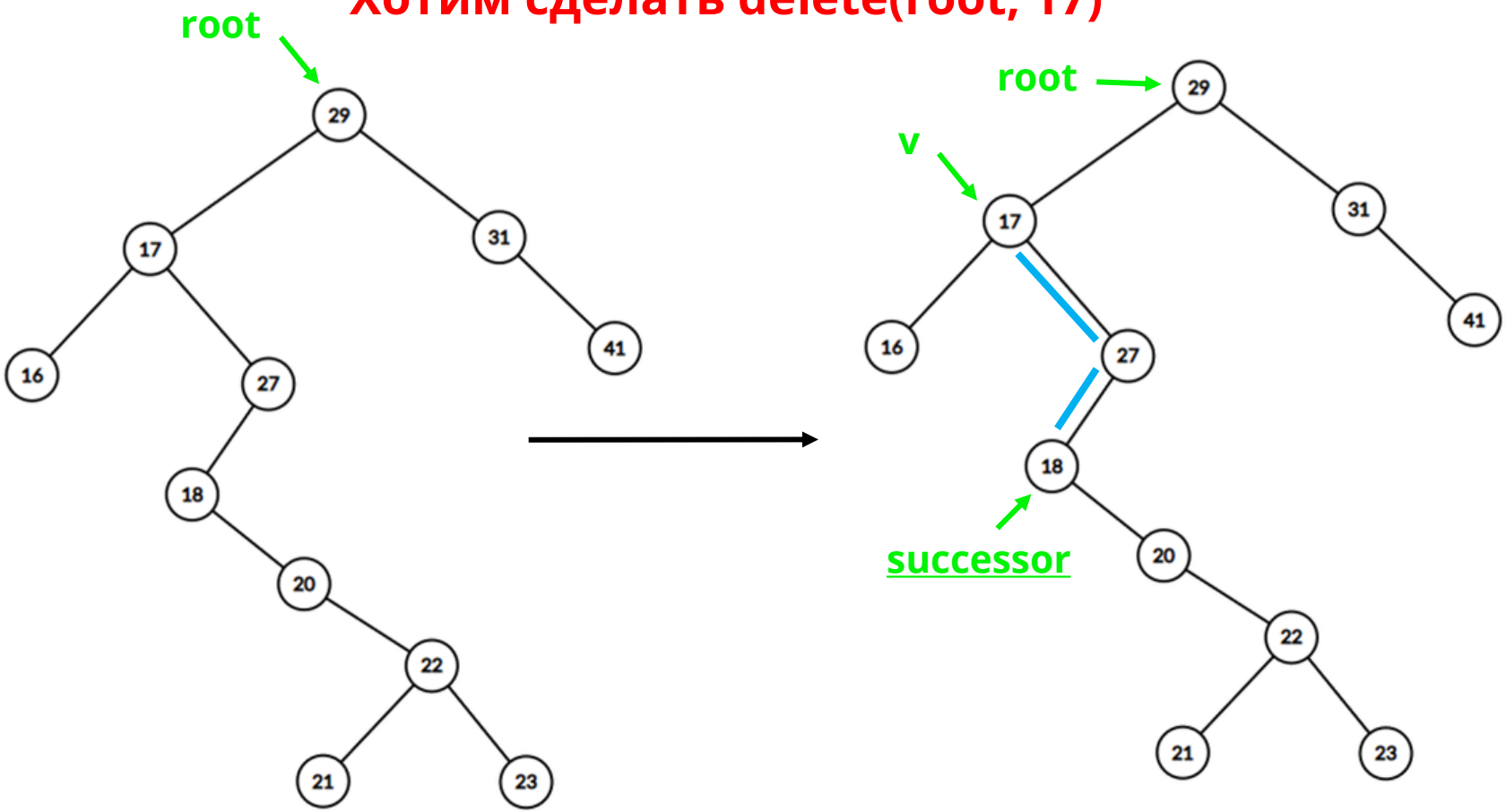
Если у следующего есть дети?

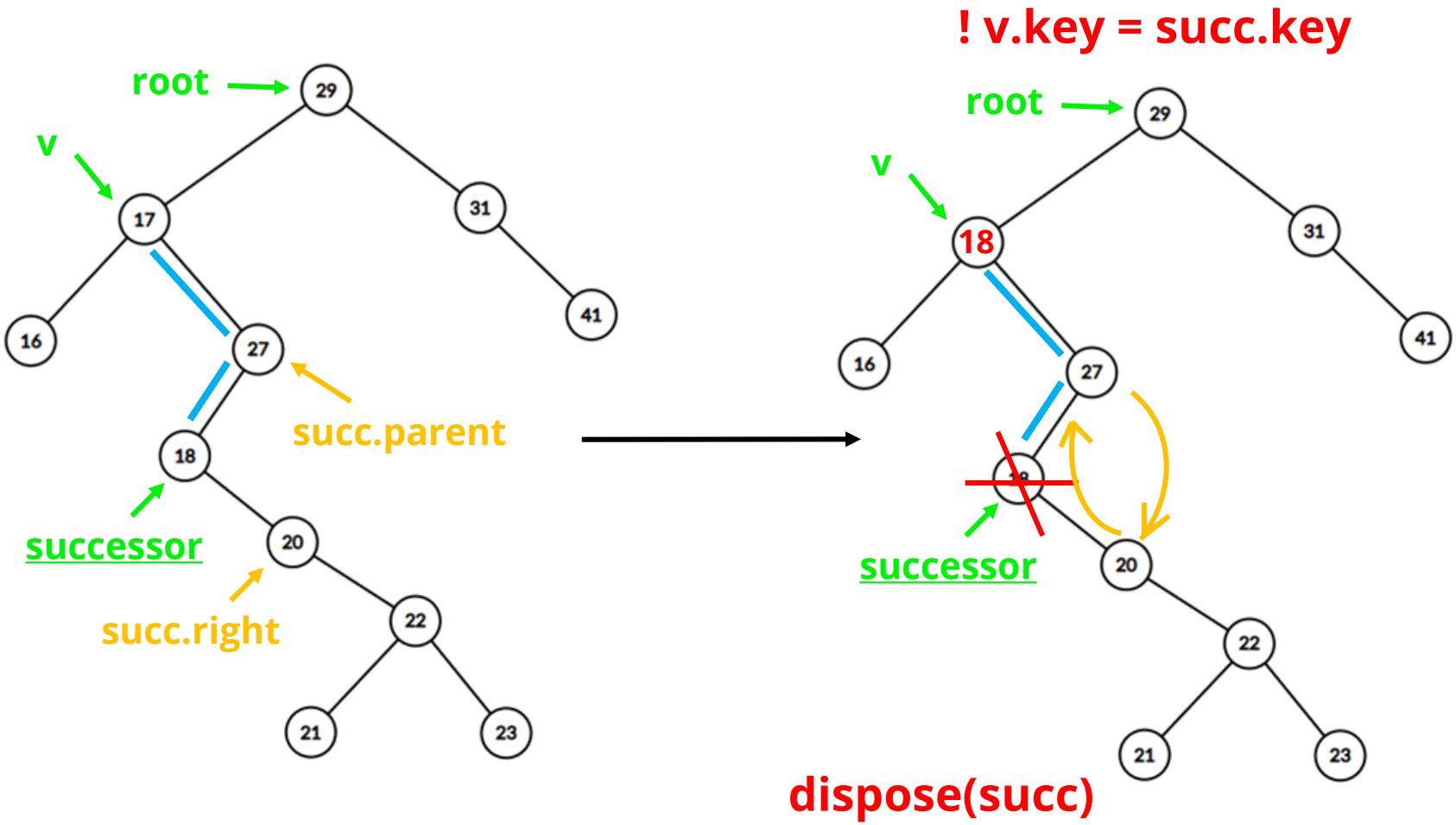


2 случай

```
p = v.parent
successor = next(v, t)
v.key = successor.key
if successor.parent.left == successor
    successor.parent.left = successor.right
    if successor.right != null
        successor.right.parent = successor.parent
else
    successor.parent.right = successor.right
    if successor.right != null
        successor.right.parent = successor.parent
```

Хотим сделать delete(root, 17)




! v.key = succ.key
root
v
29
17
31
16
27
41
18
succ.parent
successor
20
succ.right
22
21
23
root
v
29
18
31
16
27
41
successor
20
22
21
23
dispose(succ)

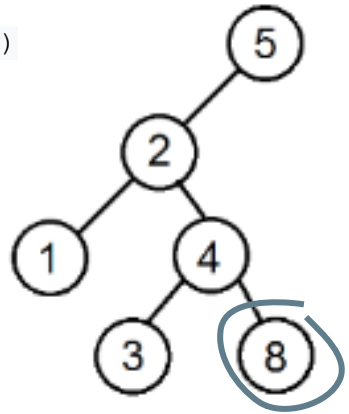
```
func delete(t : Node, v : Node):
p = v.parent
if v.left == null and v.right == null     // 1 случай, когда удаляем лист
            if p.left == v
               p.left = null
            if p.right == v
               p.right = null
else if v.left == null or v.right == null    // 2 случай, когда у удаляемого узла один ребенок
            if v.left == null
               if p.left == v
                           p.left = v.right
               else
                           p.right = v.right
               v.right.parent = p
            else
               if p.left == v
                           p.left = v.left
               else
                           p.right = v.left
               v.left.parent = p
else                              // 3 случай, когда у удаляемого узла два ребенка
            successor = next(v, t)
            v.key = successor.key
            if successor.parent.left == successor
               successor.parent.left = successor.right
               if successor.right != null
                           successor.right.parent = successor.parent
            else
               successor.parent.right = successor.right
               if successor.right != null
                           successor.right.parent = successor.parent
```

Время работы - **O(h)**

Является ли деревом поиска?

Для проверки того, является ли заданное двоичное дерево деревом поиска, надо убедиться, существует ли **хотя бы одна** вершина, нарушающая свойство дерева.

```
bool isBinarySearchTree(root: Node): // root — корень дерева

    bool check(v : Node, min: T, max: T): // min и max — мин. и макс. допустимые значения в вершинах поддерева
            if v == null
                    return true
            if v.key <= min or max <= v.key
                    return false
            return check(v.left, min, v.key) && check(v.right, v.key, max)

        return check(root, −∞, ∞)
```



Время работы - **O(n)**, где n - число вершин

Обход дерева поиска

Есть три операции обхода узлов дерева, отличающиеся порядком обхода узлов:

- ***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.

Данные алгоритмы выполняют обход за время **O(n)**, поскольку процедура вызывается для каждого узла дерева.

Обход дерева поиска

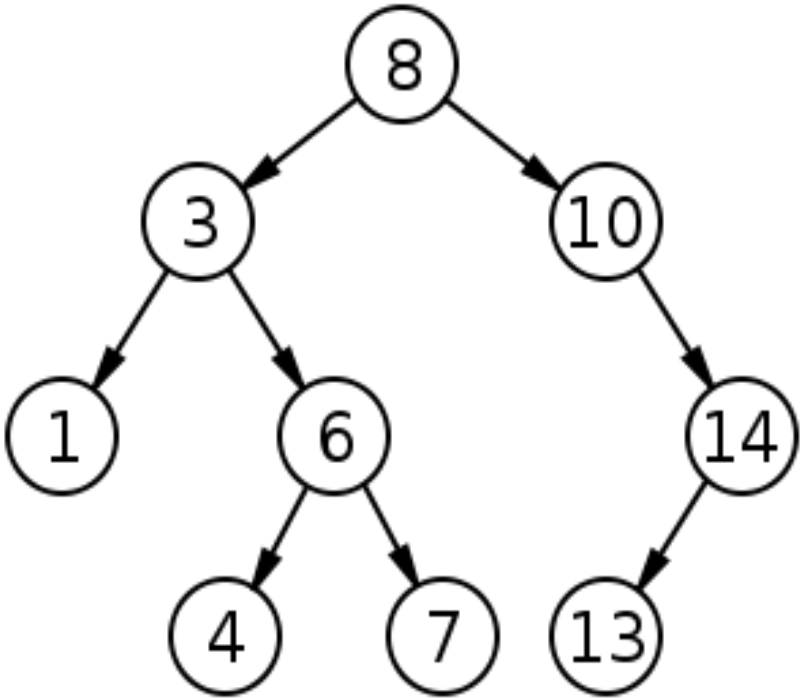
inorderTraversal

В результате данного обхода мы получаем ***отсортированную последовательность ключей дерева***.

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

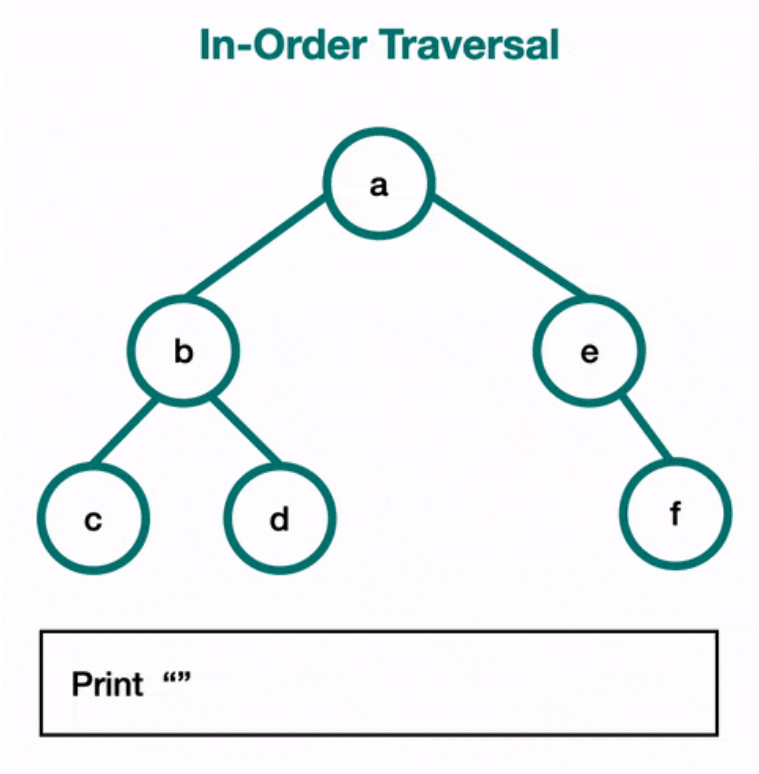
При выполнении данного обхода вершины будут выведены в следующем порядке:

- ***1 3 4 6 7 8 10 13 14***



Обход дерева поиска

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



Обход дерева поиска

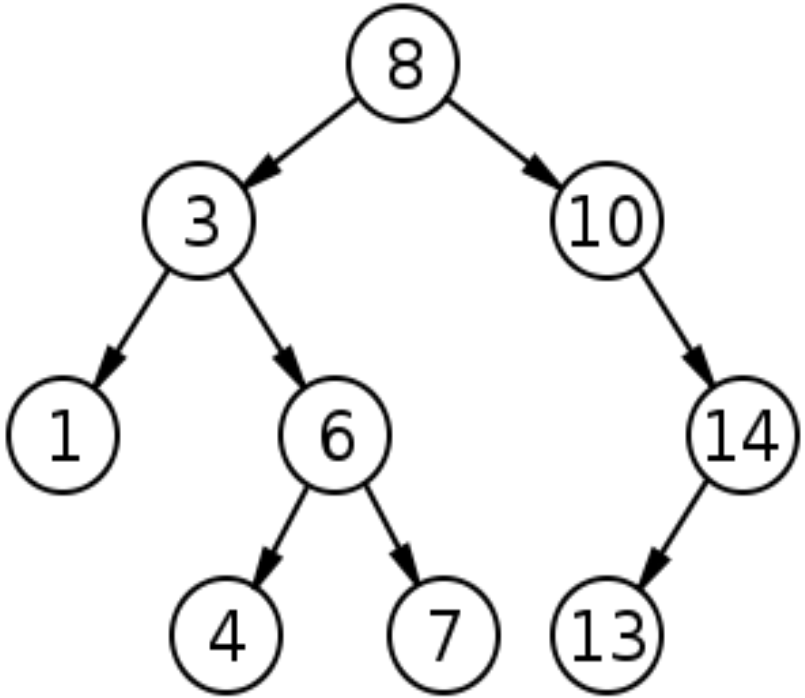
preorderTraversal

Позволяет ***восстановить дерево*** по последовательности, выведенной после выполнения данной процедуры

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

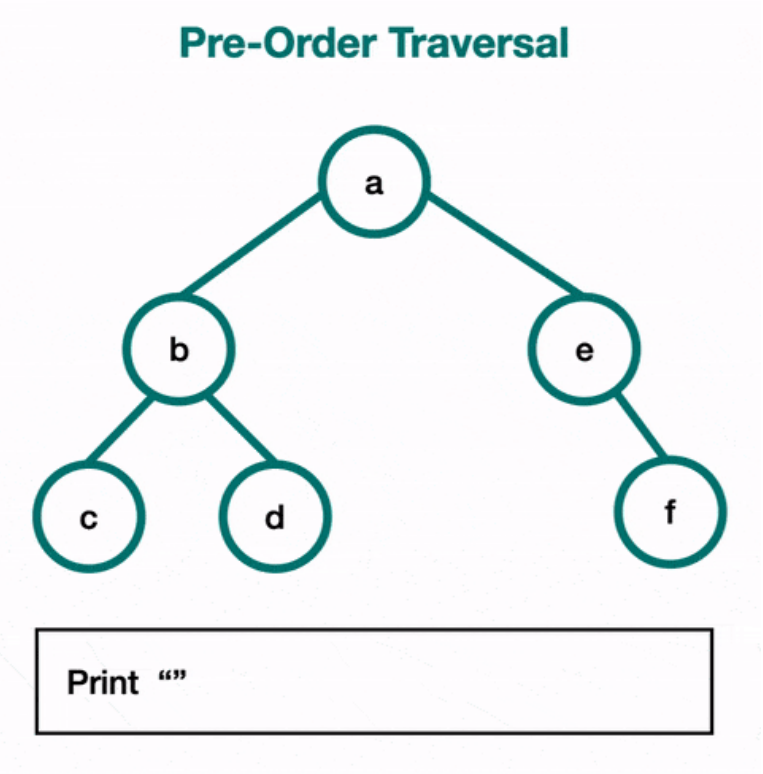
При выполнении данного обхода вершины будут выведены в следующем порядке:

- ***8 3 1 6 4 7 10 14 13***



Обход дерева поиска

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



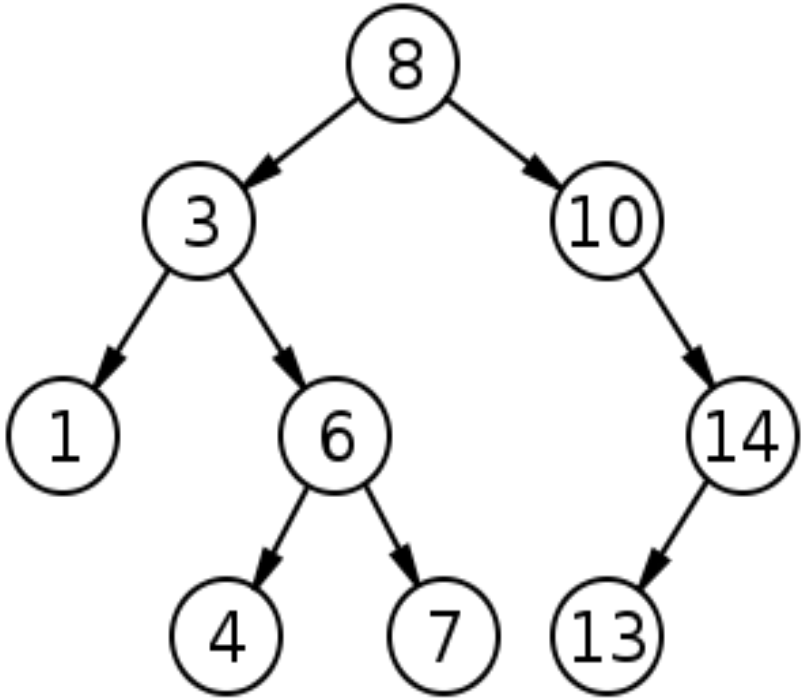
Обход дерева поиска

postorderTraversal - используется для удаления дерева.

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

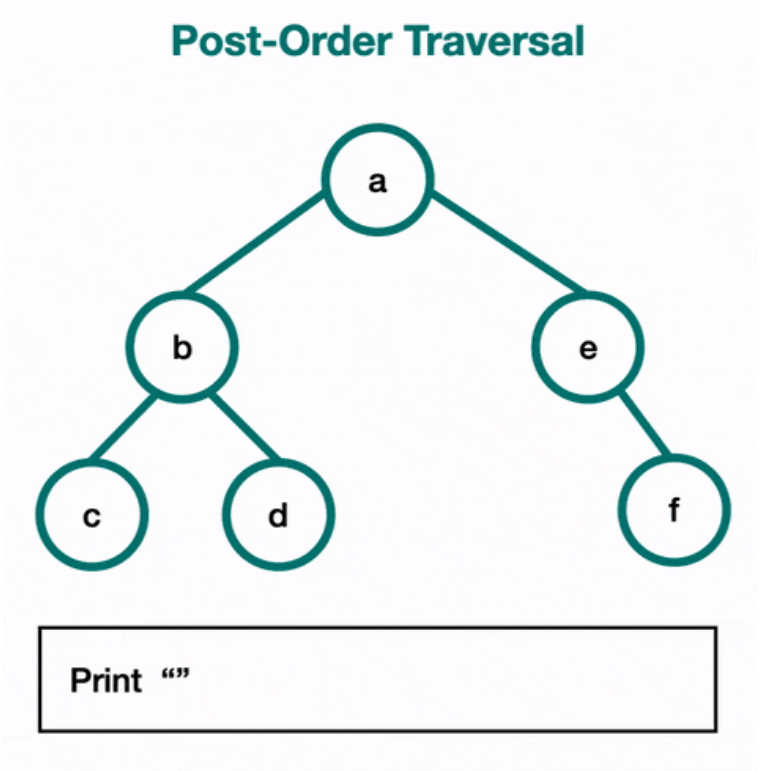
При выполнении данного обхода вершины будут выведены в следующем порядке:

- ***1 4 7 6 3 13 14 10 8***



Обход дерева поиска

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



Восстановление дерева по результату обхода preorderTraversal

- Можно однозначно определить расположение всех узлов поддерева;
- Первая вершина всегда будет корнем;
- Каждая последующая вершина становится левым сыном предыдущей;
- Если убывающая последовательность нарушается, то процедура обхода обратится к кому-то из правых поддеревьев;
- Зайдя в правое поддерево, процедура обхода снова начнет двигаться влево до упора.

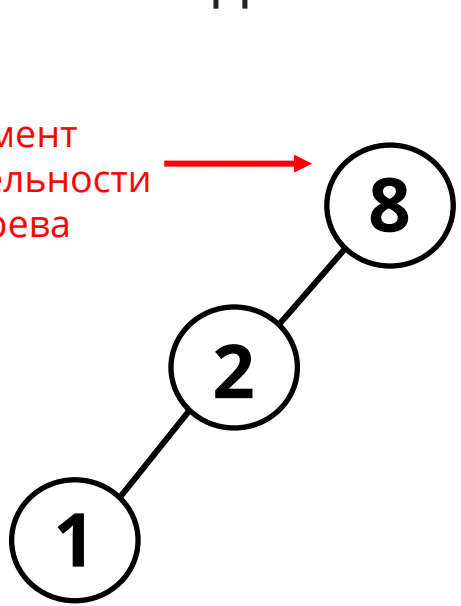
Вывод: 8 2 1 4 3 5

Разберём алгоритм на примере последовательности: 8 2 1 4 3 5

8 2 1 4 3 5	Делаем вершину корнем
8 ***2*** 1 4 3 5	Каждую вершину подвешиваем к последней из взятых ранее в качестве левого сына.
8 ***2*** ***1*** 4 3 5	
8 ***2*** 1 **4** 3 5	Ищем максимальное значение, меньшее 1. В данном случае оно равно 2.
8 2 1 ***4*** ***3*** 5	Находим убывающую подпоследовательность.
8 2 1 ***4*** 3 **5**	Для этой вершины ищем максимальное значение, меньшее его.

Вывод: 8 2 1 4 3 5

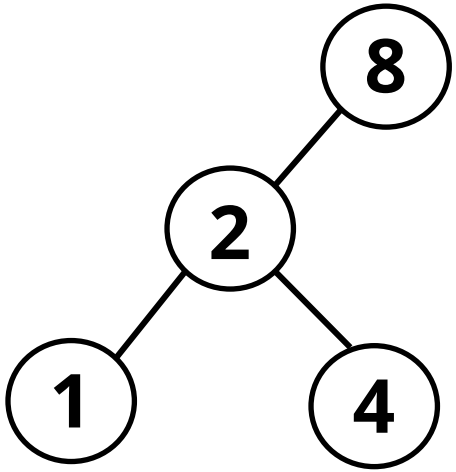
Разберём алгоритм на примере последовательности: **8 2 1 4 3 5**



Пока последовательность **убывает** (каждый элемент меньше предыдущего), будем вставлять элемент в **левое** поддерево предыдущего

Разберём алгоритм на примере последовательности: **8 2 1 4 3 5**

1, 2 < 4, из них **2** – максимальный



Очередной элемент – **4**

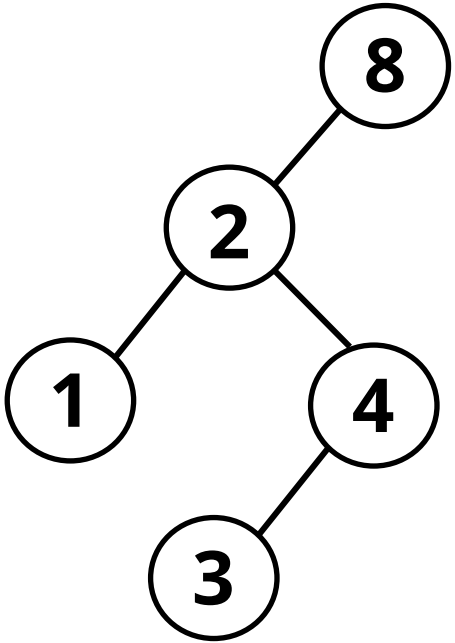
4 > 1 -> последовательность перестала убывать

Тогда найдем максимальный элемент последовательности слева от **4** такой, что он **< 4**

Таким элементом является **2**. Запишем **4** в **правое** поддерево **2**

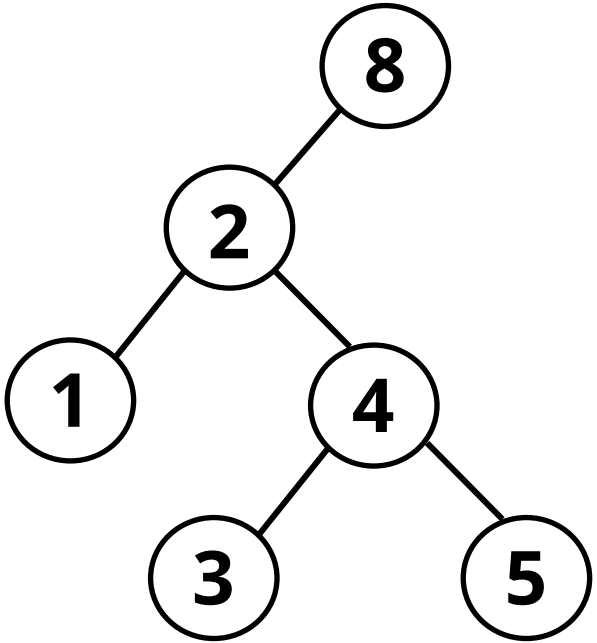
Разберём алгоритм на примере последовательности: **8 2 1 4 3 5**

убывает



3 < 4 -> записываем **3** в **<u>левое</u>** поддерево **4**

Разберём алгоритм на примере последовательности: **8 2 1 4 3 5**



1, 2, 3, 4 < 5, из них **4** – максимальный

Очередной элемент – **5**

5 > 3 -> последовательность перестала убывать

Найдем максимальный меньший, это **4**. Запишем **5** в **правое** поддерево **4**