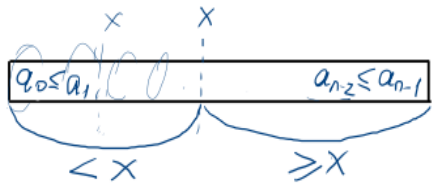
Бинарный поиск



Инвариант: индексы l и r, т.ч. a[l]<x и a[r] >=x

```
BinarySearch(A,x)
l=-1
r=n
while (r>l+1)
    m=⌊(l+r)/2⌋
    if (a[m]<x)
        l=m
    else
        r=m
if (r<n && a[r]==x)
    return r
else return -1
```

O(log n)

Пример

0	1	2	3	4	5	6	7
1	2	10	14	25	30	89	90

10?

l	r	m
-1	8	3
-1	3	1
1	3	2
1	2	

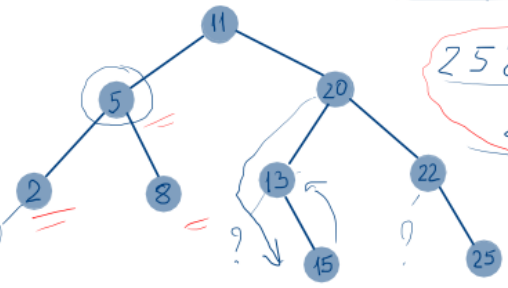
log(n)

Обходы двоичного дерева поиска

Вывод элементов в отсортированном порядке с помощью **центрированного** обхода

левый, корень, правый

```
Inorder_Tree_Walk(x)
    if x != Nil
        Inorder_Tree_Walk(x.left)
        print x.key
        Inorder_Tree_Walk(x.right)
```



2 5 8 11 13 15 20 22 25

Прямой обход: корень, левый, правый

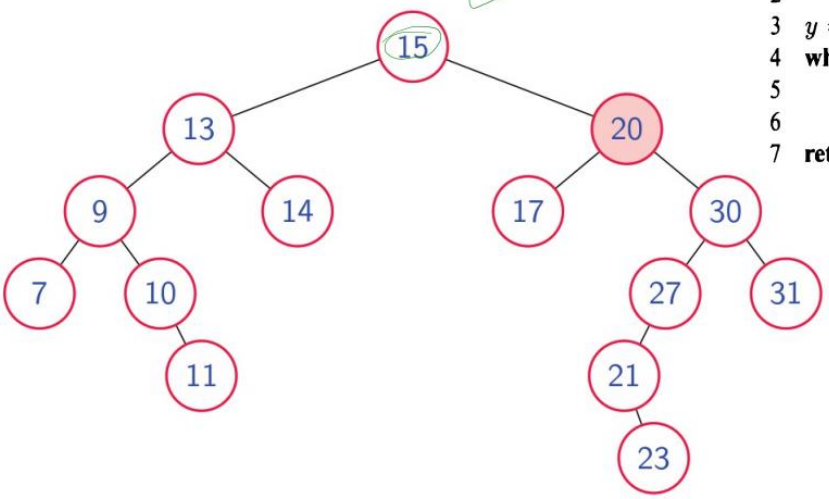
Обратный обход: левый, правый, корень

Нужны для описания дерева поиска: центрированный + прямой/обратный

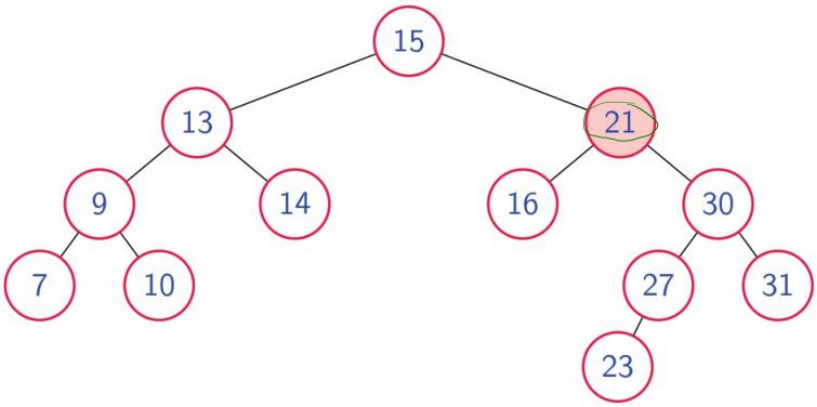
Только прямого и обратного не достаточно для однозначного описания дерева

- Найти последующий за 20

можно делать без сравнения ключей




```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL и x == y.right
5      x = y
6      y = y.p
7  return y
```



3 случая, когда удаляемая вершина:

1) лист - убрать из дерева (убрать у родителя указатель на неё)

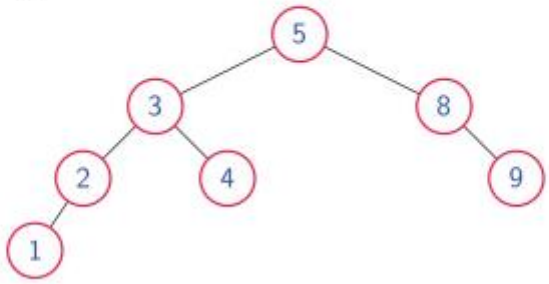
2) имеет 1 ребенка - переподвесить ребенка на родителя этой вершины

3) имеет 2 ребенка - найти последующий (у него точно нет левого ребенка) и записать его значение на место удаляемого, а сам последующий удалить, как в варианте 1 или 2.

1/1

Способ 1. Поддержание инвариантов сбалансированности

- Пример: Высота левого поддерева отличается от высоты правого поддерева не больше, чем на единицу
- Реализация: АВЛ-дерево
  - АВЛ: Адельсон-Вельский и Ландис



- Высота дерева: $h = O(\log n)$

```
Node *AVL::rotate(Node *a) {
  int d = a->diff;
  if (d == -2) {
    Node *b = a->r;
    if (b->diff == 1) {
      Node *c = b->l;
      if (c->diff == 1)
        a->diff = 0, b->diff = -1, c->diff = 0;
      else if (c->diff == -1)
        a->diff = 1, b->diff = 0, c->diff = 0;
      else
        a->diff = 0, b->diff = 0, c->diff = 0;
      a = big_rotate_left(a);
    }
    else {
      if (b->diff == -1)
        b->diff = 0, a->diff = 0;
      else
        b->diff = 1, a->diff = -1;
      a = rotate_left(a);
    }
  } else if (d == 2) {
    Node *b = a->l;
    if (b->diff == -1) {
      Node *c = b->r;
      if (c->diff == -1)
        a->diff = 0, b->diff = 1, c->diff = 0;
      else if (c->diff == 1)
        a->diff = -1, b->diff = 0, c->diff = 0;
      else
        a->diff = 0, b->diff = 0, c->diff = 0;
      a = big_rotate_right(a);
    }
    else {
      if (b->diff == 0)
        b->diff = -1, a->diff = 1;
      else
        b->diff = 0, a->diff = 0;
      a = rotate_right(a);
    }
  }
  return a;
}

Node *AVL::rotate_left(Node *a) {
  Node *b = a->r;
  a->r = b->l;
  b->l = a;
  return b;
}

Node *AVL::rotate_right(Node *a) {
  Node *b = a->l;
  a->l = b->r;
  b->r = a;
  return b;
}

Node *AVL::big_rotate_left(Node *a) {
  a->r = rotate_right(a->r);
  a = rotate_left(a);
  return a;
}
```

```
Node *AVL::big_rotate_right(Node *a) {
  a->l = rotate_left(a->l);
  a = rotate_right(a);
  return a;
}
```

Добавление вершины

Пусть нам надо добавить ключ t. Будем спускаться по дереву, как при поиске ключа t. Если мы стоим в вершине a и нам надо идти в поддерево, которого нет, то делаем ключ t листом, а вершину a его корнем. Дальше поднимаемся вверх по пути поиска и пересчитываем баланс у вершин. Если мы поднялись в вершину i из левого поддерева, то $diff[i]$ увеличивается на единицу, если из правого, то уменьшается на единицу. **Если пришли в вершину и её баланс стал равным нулю, то это значит высота поддерева не изменилась и подъём останавливается. Если пришли в вершину и её баланс стал равным 1 или -1, то это значит высота поддерева изменилась и подъём продолжается. Если пришли в вершину и её баланс стал равным 2 или -2, то делаем одно из четырёх вращений и, если после вращения баланс стал равным нулю, то останавливаемся, иначе продолжаем подъём.**

Удаление вершины

Для простоты опишем рекурсивный алгоритм удаления. Если вершина — лист, то удалим её, иначе найдём самую близкую по значению вершину a, переместим её на место удаляемой вершины и удалим вершину a. От удалённой вершины будем подниматься вверх к корню и пересчитывать баланс у вершин. Если мы поднялись в вершину i из левого поддерева, то $diff[i]$ уменьшается на единицу, если из правого, то увеличивается на единицу. **Если пришли в вершину и её баланс стал равным 1 или -1, то это значит, что высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины стал равным нулю, то высота поддерева уменьшилась и подъём нужно продолжить. Если баланс стал равным 2 или -2, следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.**