Задания — pull request, означает, что вы хотите сдать очередное задание.

— результаты в таблице (ссылка в письме).

Двоичные деревья поиска.

— структура данных

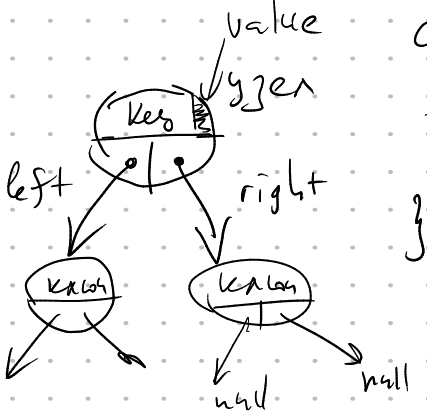
1) хранит ключи - значения
2) быстрый поиск значения по ключу / наличие ключа.
3) добавлять / удалять ключи со значениями.

хэш таблицы (ассоциативные массивы в Python: dict, в Java: Map)

---

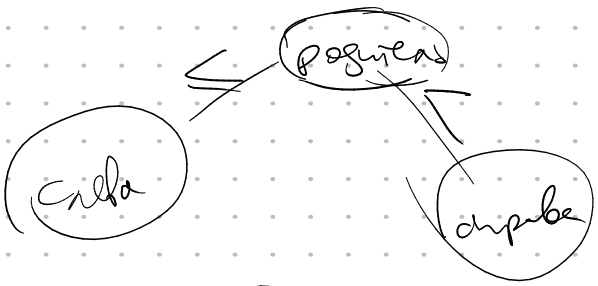
4) перебор ключей по порядку.

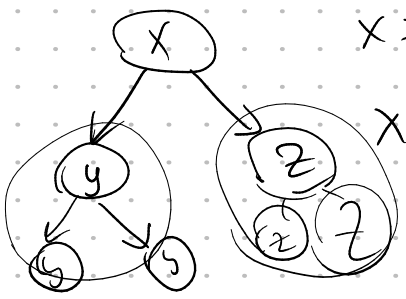
Как хранить в памяти:



class Node {
key: int
value: V
left: Node
right: Node
}

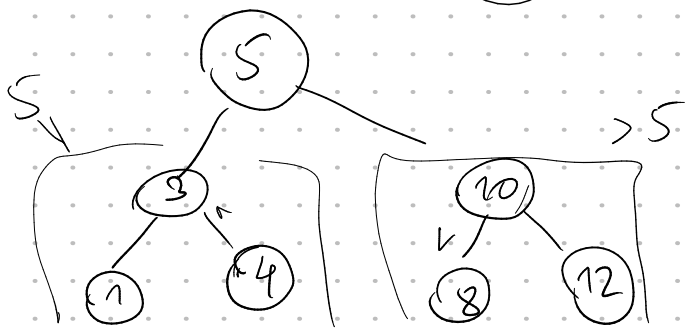
Правило дерева поиска:





$x \geq y$ - в левом

$x < z$ - в правом



Пустое дерево : ⇒ нет узлов.

добавить узел с ключом

удалить узел по ключу

найти ключ (проверка)

перебор по порядку ⟷ следующий по порядку ключ.

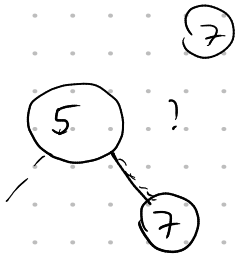
Наивная реализация.

Берём дерево, при добавлении ключа идем в подходящее место.

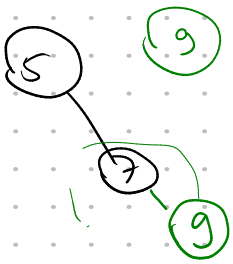
пусто

+5

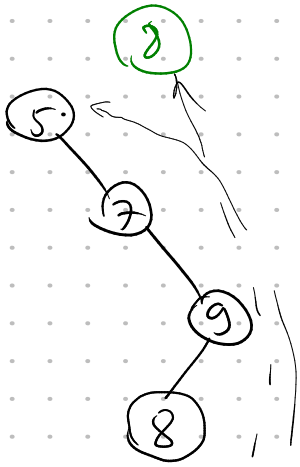
+7



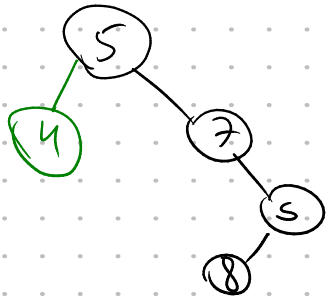
+9



+8



+4



алгоритм

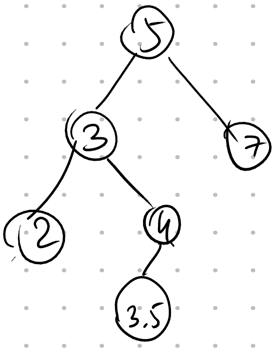
```
add(key, node)
  if key < node.key
    if node.left=null
      node.left=new Node(key)
    else
      add(key, node.left)
  else
    if node.right=null
      node.right=new Node(key)
    else
      add(key, node.right)
```

Хранение дерева

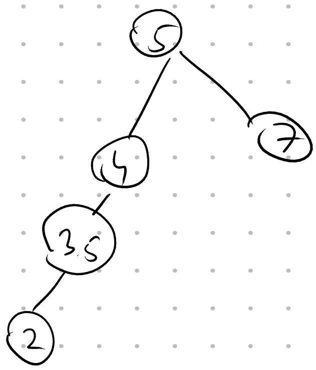
```
class Tree {
    root: Node
    add(key) → add(key, root)
}
```

удаление — нет в задачах

delete(3) ....



можно разбирать случаи

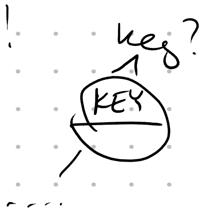


поиск. search(key) — проверка, есть ли ключ.

```
search (key, node)
    if node=null return False
    if node.key = key
        return True        ← помни!!
    if node.key < key
        return search (node.left)
    else
        return search (node.right)
```

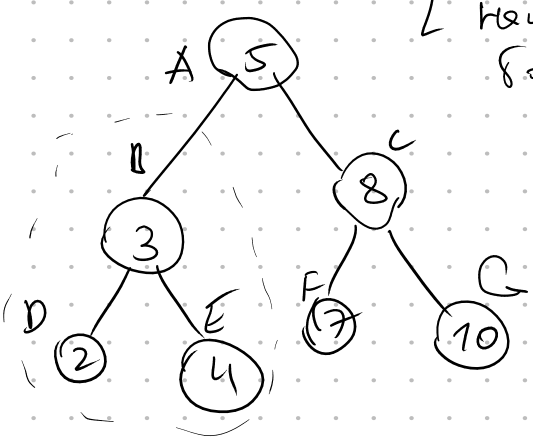
для search_next:

```
if node.left=null
    return key
else
    search_next(
        key, node.left)
```



Дополнительно. поиск следующего элемента.

search_next(key) — найти ключ, следующий после key [найти min ключ из всех, что больше key].



search_next(6, A) ⟶ search_next(6, C) ⟶

5<6 поэтому идём справа

search_next(6, F) → 8>6, идем налево, м.б. 8 - ответ

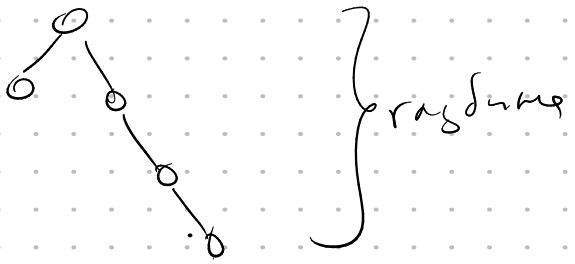
search_next(6, F.left) → 7>6 ага. 8 точно не ответ, может 7??

=> 7 ответ.

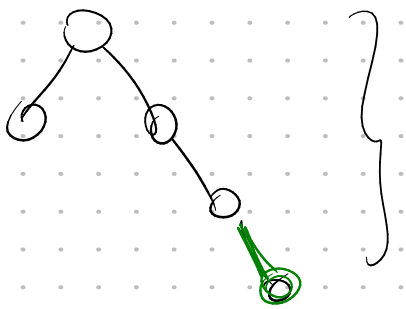
search_next очень похож на search.

(только, если ключ удал. > искомого, но слева пусто)

сложность всех действий: кол-во действий ~ глубине дерева

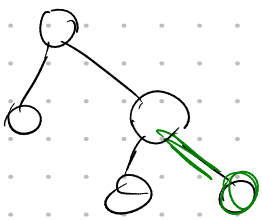


хорошо, когда в дереве мал. высот. Обычно вместо наивной реализации делают сбалансир. реализ.



ручной спуск

глубина справа > слева



→ АВЛ.

сложно балансировать, АВЛ, красно-черное — много случаев.

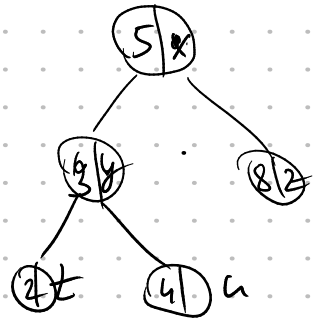
В сбалансированных деревьях глубина дерева из N узлов ~ $\log_2 N$. ($\log_2 10^6 \approx 30$)

Сбалансированные деревья - рекомендую Декартово дерево.

Ключ двойной

ключ + ключ 2

просто в реализации.



$x > y$

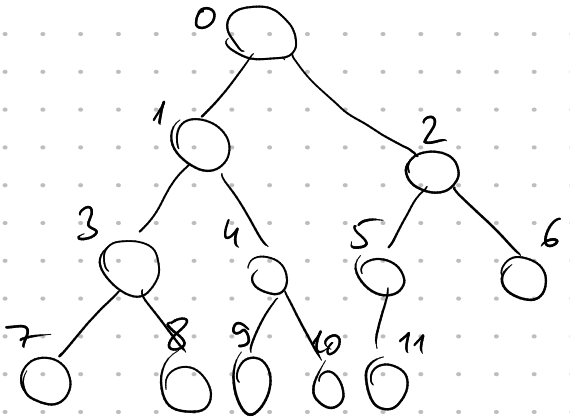
$x > z$

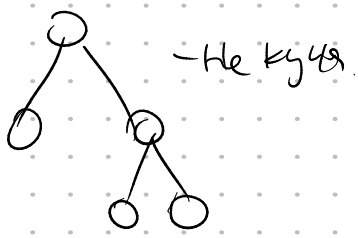
$y > t$

$y > u$

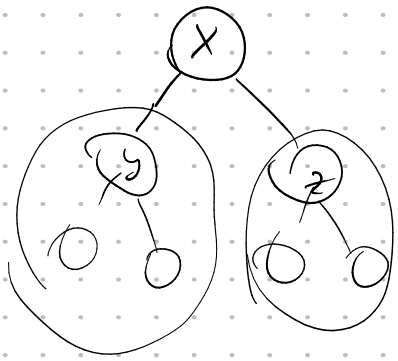
Куча. Двоичное дерево особой формы.

(заполняется по слоям)





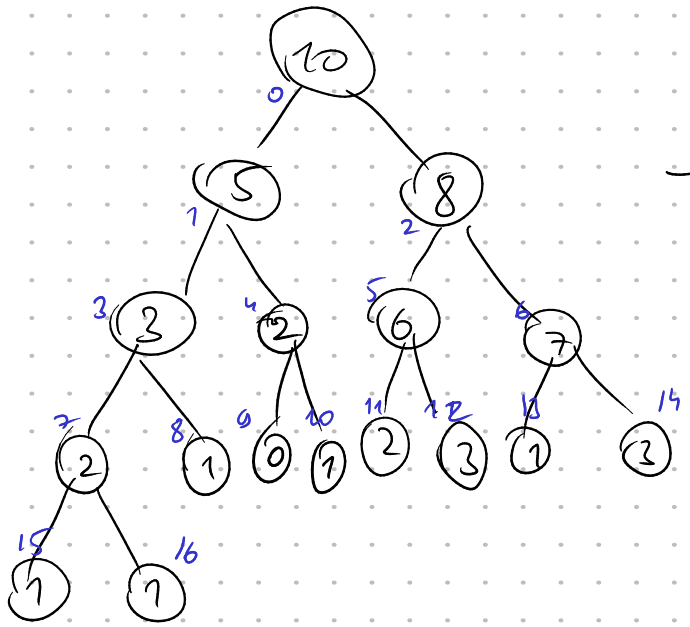
Хранит ключи в узлах



$x > y$

$x > z$

Корень - всегда max.



- куча

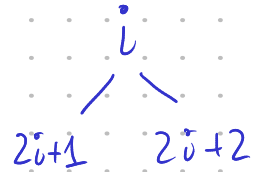
10, 5, 8, 3, 2, 6, 7, 2, 1, 0, 1, 2, 3, 1, 1, 1

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ← ind

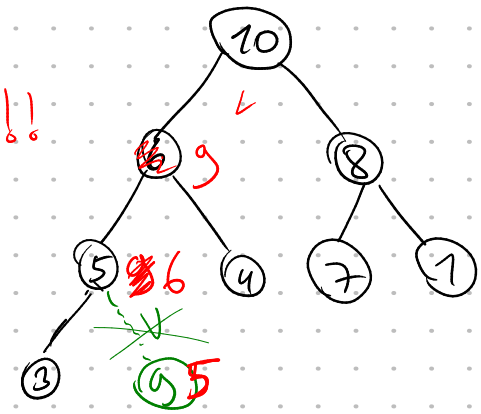
умеет: – добавлять ключ + (? значение)

– выбирать max ключ / удалять его

1. Куча хранит массивом со вида



2. добавить элемент: Пример.



добавим 9

1) – добавляем его очередным эл-ом в конец дерева.

2) он наверняка нарушает условие...

– поднимать кучу (поднимаем вверх)

переставим 5 и 9.

Пока новый элемент больше родителя, меняем его с родителем

Сложность добавления $\leq$ высота дерева $\leq \log_2 N$

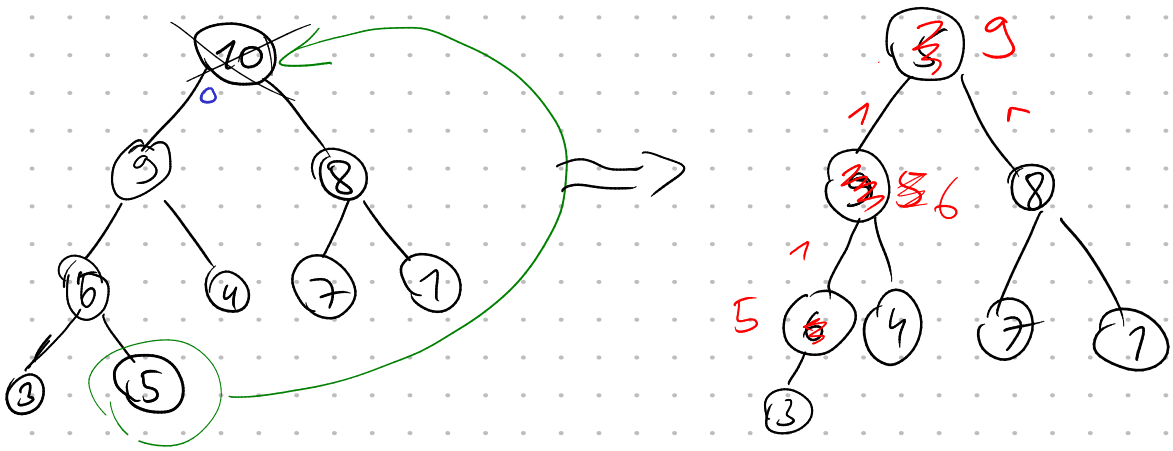
N – кол-во элементов

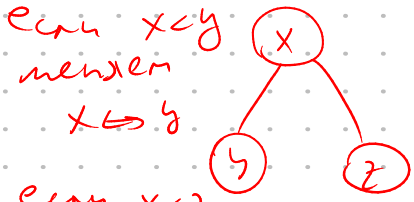
3. Взять max ключ и удалить его.

– max эл-т, очевидно, в корне arr[0]

Забираем корень

– последний элемент – в начало





если x>y, z
то у нас
куча

Сложность $\sim \log_2 N$

Зачем куча? **Очередь с приоритетами.**

– добавляем элемент в очередь, каждому назначен приоритет. Следующий тот, у кого max приоритет.