Стуктуры данных. Что такое и зачем?

> Структура данных - прослойка над memory model позволяет быстрее делать какие-то операции с данными (запросы поиска, изменения и т.д.).

Куча

Разберём простую СД - кучу.

Куча хранит мультимножество. Требуемый интерфейс:

```
insert(x) # A = A \cup {x}
get_min() # min(a)
remove_min() # A = A \ {min(A)} (removes only one min element, htere can be a lot of them)
```

v 0.1: Массив

```
void insert(int x):
    a[n++] = x; // O(1) write

void get_min(int x):
    return min(a); // O(n) read

void remove_min():
    swap(a[index_of_first_min(a)], a[--n]) // Read: O(n), Write: O(1)
```

v 0.1.1: Массив с сохранением индекса минимального элемента

```
std::vector<int> a;
int n;
int min_index = 0;


void insert(int x):
    a[n++] = x; // O(1) write
    if (a[n - 1] < a[min_index]) min_index = n - 1

void get_min(int x):
    return a[min_index]; // O(1) read

void remove_min():
    swap(a[min_index(a)], a[--n]); // Read: O(n), Write: O(1)
    min_index = index_of_first_min(a)
```

v 0.2: Массив, отсортированный по убыванию

```
void insert(int x):
    a[n++] = x; // O(1) write
    proper_index =

void get_min(int x):
    return a[n - 1]; // O(1) read

void remove_min():
    n--
```

v 0.3: Скрещиваем ужа с ежом (sqrt-декомпозиция)

Возьмём мнгого маленьких отсортированных структур данных размером $B \approx \sqrt{n}$

```
def insert():
    # O(B)
def remove_min():
    # O(n/B)
```

$$operations = kB + k\frac{n}{B} \leqslant \sqrt{n}$$

v 1.0: Двоичная куча

Построим некое бинарное дерево.

Высота (количество слоёв) дерева $\approx log(n)$.

Можно хранить дерево:

- Массив, когда структура заранее известна.
- Ноды, ходить по ссылкам

Сейчас будем хранить в массиве.

Перемещения между node-ами:

$$childs(i) = \{2i+1, 2i+2\} parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$$

Чтобы куча была валидной, childs должны быть $\geqslant$, чем родители. Тогда миимальный - корень.

```
def get_min():
    return a[0]
def insert(x):
    a.append(x)
    # Sifting up! While less than parent, swap.
```

```
    # Inequalities with other children now feel even better,
    # When changing upper with new, lower ones don't feel worse,
    # because their strict upper is subtree minima,
    # there can only be problems in inequality between «even upper» and the «new» ones
    while i > 0 and a[i] < a[(i - 1) / 2]:
        swap(a[i], a[(i - 1) / 2])
def remove_min():
    swap(a[0], a[n--]) # Special swap can be implemented,
                       # which will store each element's location in HashMap.

    # Sieve down
    i = 0
    while 2 * i + 1 < n:
        j = 2 * i + 1 # Set j to minimal existing child's index
        if 2i + 2 < n and a[2i + 2] < a[j]:
            j = 2i + 2
        if a[i] <= a[j]:
            break
        else:
            swap(a[i], a[j])
            i = j
```

Hint: Если боимся равных элементов, то можно использовать сортировку по id внутри “одинаковых”

Сортировка кучей (Heap sort)

Возьмём все элементы. Положим их в кучу. А потом вынем из кучи поочерёдно. Конец. Честный $n\log(n)$

Но попробуем не заводить лишний массив, а просто использовать тот же.

```
for i=0..n-1:
    sift_up(i)
for i=0..n-1:
    remove_min(n-1)
```

```
def heapify():
    for i=0..n-1:
        sift_up(i)
```

$$\sum_{i=0}^{n} log(n) \geqslant \frac{n}{2} log\left(\log\frac{n}{2}\right)$$

...

Будем просеивать элементы вниз, а не вверх, чтобы было больше малых перемещений вместо больших.

На очередном шагу свойства куча будут выполняться на префиксе.

```
for i =n-1..0:
    sift_down(i)
for i=0..n-1:
    remove_min(n-1)
```