

# 1 Простые структуры данных

## Требования к структуре данных:

- От СД мы хотим обработку каких-то наших запросов.
- online-offline. Бывает, что мы знаем все запросы, бывает, что мы узнаем запрос только тогда, когда отвечаем на предыдущий
- При обсуждении времени работы отделяется время на препроцессинг и на последующие ответы на запросы (query).

## 1.1 Data structure / interface

*Структура данных* — это какой-то математический объект, который умеет отвечать на наши запросы конкретным способом. Красно-черное дерево — это структура данных.

*Интерфейс* — это объект, с которым может взаимодействовать пользователь, который каким-то образом умеет отвечать на наши запросы (пользователю все равно, как, его волнует только то, что интерфейс реализует, и за какое время(память) он это делает).  $std :: set$  — это интерфейс.

*Итератор* — это специальный объект, отвечающий непосредственно за ячейку в структуре данных. Для того, чтобы удалить элемент, мы должны иметь итератор на этот элемент. Т.е. удаление по ключу работает за  $O(erase)$ , а удаление по значению за  $O(find) + O(erase)$

**list** Списки бывают двусвязными, односвязными, циклическими. У каждого элемента есть ссылка на следующий (и иногда на предыдущий), а также есть отдельный глобальный указатель на начало списка.

**stack** Стек — структура данных, которая умеет делать добавление в конец, удаление из конца, взятие последнего элемента, за  $O(1)$ .

**queue** Очередь — структура данных, которая умеет делать добавление в конец, удаление из начала, взятие первого элемента, за  $O(1)$ .

**deque** Двусторонняя очередь — структура данных, которая умеет делать добавление, удаление и взятие элемента с любого конца последовательности, за  $O(1)$ .

**priority\_queue** Очередь с приоритетами ака куча — структура данных, которая умеет делать добавление, удаление, и быстрые операции с минимумом, представляющая из себя дерево с условием  $parent(u) = v \rightarrow value(v) \leq value(u)$ .

Все эти структуры реализуются как на массиве (храним последовательную память и указатели на начало/конец), так и на списках (что на самом деле является тем же самым, что и на массиве, просто ссылка вперед эквивалентна  $a_i \rightarrow a_{i+1}$  в терминологии массивов, *прим. автора*).

Структура данных на массиве кратно быстрее аналогичной на ссылках, потому что массив проходит по кэшу и не требует дополнительной памяти.

data structure	add	delete	pop	find	top	build	min	get by index
stack	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	-
dynamic array	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$
queue	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	-
deque	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	$O(1)$
linked list	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
sorted array	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(n \log n)$	$O(1)$	$O(1)$
priority_queue	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(n)$	$O(1)$	-	

## 1.2 Двоичная куча

Реализация двоичной кучи на массиве — создаем массив размера  $sz$ , и создаем ребра  $i \rightarrow 2 \cdot i$ ,  $i \rightarrow 2 \cdot i + 1$ .

От такой кучи мы хотим:

- `insert(x)`
- `get_min()`
- `extract_min()`
- `erase`
- `change = {decrease_key, increase_key}`

Для такой кучи мы реализуем *sift\_up(x)*, *sift\_down(x)* — просеивание вниз и вверх. Процедура должна устранить конфликты с элементом  $x$ . Остальные операции умеют реализовываться через нее.

$$insert = add\_leaf + sift\_up$$

$$extract\_min = swap(root, last) + last - 1 + sift\_down(root)$$

$$erase = decrease\_key(-\infty) + extract\_min$$

Отдельно отметим построение кучи за  $O(n)$  — *sift\_down* поочередно для всех элементов  $n, n - 1, \dots, 1$ .

```
void sift_up(int v) { // v >> 1 <=> v / 2
    if (key[v] < key[v >> 1]) {
        swap(key[v], key[v >> 1]);
        sift_up(v >> 1);
    }
}

void sift_down(int v, int size) { // indexes [1, size]
    if (2 * v > size) {
        return;
    }
    int left = 2 * v;
    int right = 2 * v + 1;
    int argmin = v;
```

```
    if (key[v] > key[left]) {  
        argmin = left;  
    }  
    if (right <= size && key[right] < key[argmin]) {  
        argmin = right;  
    }  
    if (argmin == v) {  
        return;  
    }  
    swap(key[v], key[argmin]);  
    sift_down(argmin, size);  
}
```