Абстракстные типы данных. Стек, очередь, дек и очередь с приоритетом.

Абстракстный тип данных (АТД) - это математическая модель для типов данных, которая задаёт поведение этих типов, но не их внутреннею реализацию. Простейший пример АТД - это стек.

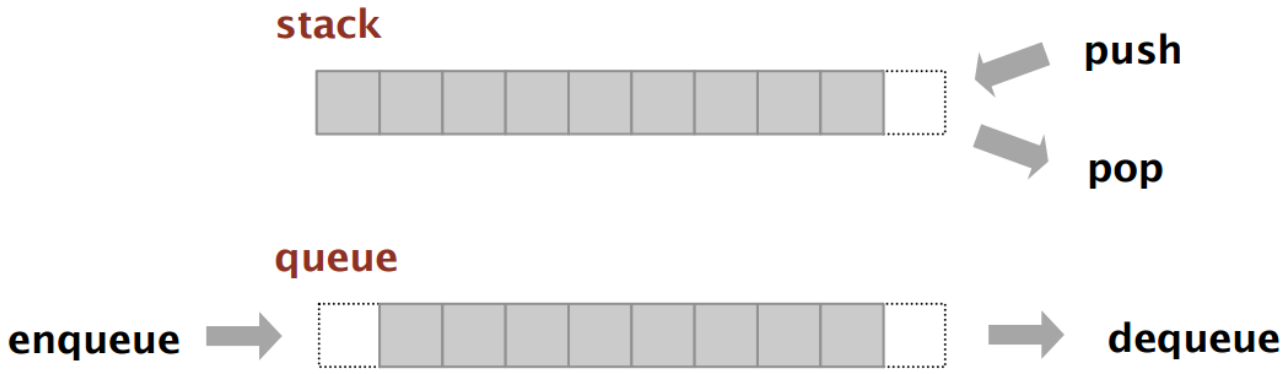
Стек (Stack) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- **push** - добавить элемент в стек.
- **pop** - извлечь из стека последний добавленный элемент.

Таким образом, поведение стека задаётся этими двумя операциями. Так как стек - это абстрактный тип данных, то его внутренняя реализация на языке программирования может быть самой разной. Стек можно сделать на основе статического массива, на основе динамического массива(`malloc`/`free`) или на основе связного списка. Внутренняя реализация не важна, важно только наличие операций **push** и **pop**.

Очередь (Queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- **enqueue** - добавить элемент в очередь.
- **dequeue** - извлечь из очереди первый добавленный элемент из оставшихся.



Дек (Deque = Double-ended queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью четырёх операций:

- **push_back** - добавить элемент в конец.
- **push_front** - добавить элемент в начало.
- **pop_back** - извлечь элемент с конца.
- **pop_front** - извлечь элемент с начала.

Очередь с приоритетом (Priority Queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- **insert** - добавить элемент.
- **extract_best** - извлечь из очереди элемент с наибольшим приоритетом.

То, что будет являться приоритетом может различаться. Это может быть как сам элемент, часть элемента (например, одно из полей структуры) или другие данные, подаваемые на вход операции **insert** вместе с элементом. В простейшем случае, приоритетом является сам элемент (тогда очередь с приоритетом просто возвращает максимальный элемент) или сам элемент со знаком минус (тогда очередь с приоритетом возвращает минимальный элемент).

Реализация очереди с приоритетом на основе массива.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 1000

struct priority_queue {
        int data[MAX_SIZE];
        int size;
};
typedef struct priority_queue PriorityQueue;

void init(PriorityQueue* pq) {
        pq->size = 0;
}
void insert(PriorityQueue* pq, int x) {
        if (pq->size == MAX_SIZE) {
                printf("Error! Can't insert into the priority queue\n");
                exit(1);
        }
        pq->data[pq->size] = x;
        pq->size++;
}
int extract(PriorityQueue* pq) {
        if (pq->size == 0) {
                printf("Error! Can't extract from the priority queue\n");
                exit(1);
        }
        // Находим минимальный элемент
        int min_index = 0;
        for (int i = 1; i < pq->size; i++)
                if (pq->data[i] < pq->data[min_index])
                        min_index = i;
        int result = pq->data[min_index];

        // Сдвигаем элементы на 1 влево
        for (int i = min_index + 1; i < pq->size; i++)
                pq->data[i-1] = pq->data[i];
        pq->size--;

        return result;
}
int main() {
        PriorityQueue a;
        init(&a);
        int numbers[] = {54, 32, 12, 16, 42, 53, 26, 91, 21, 43, 64, 75, 64, 37, 45};
        for (int i = 0; i < 15; i++)
                insert(&a, numbers[i]);

        for (int i = 0; i < 15; i++) {
                printf("%d ", extract(&a));
        }
}
```

Данная реализация очереди с приоритетом имеет один очень большой недостаток - операция извлечения **extract** очень неэффективна. Алгоритмическая сложность данной операции равна $O(N)$. Т.е. количество действий, необходимых чтобы совершить один **extract** зависит от N следующим образом: $K \approx cN$, где N - количество элементов в очереди, c - некоторая константа.