# Методичка по алгоритмам C++ (v0.2)

R314 & TL

12 августа 2018 г.

# Оглавление

# Глава 1

# Структуры данных

## 1.1 Простейшие структуры

### 1.1.1 Массив

Массив — это несколько переменных, объединённых под одним именем и расположенных подряд в памяти. Обращение к элементам массива происходит через *индекс* — порядковый номер элемента в массиве, если считать с **нуля**.
Пример применения массива: для последовательности из $n$ чисел вывести попарные произведения всех элементов.

```
1 int n, a[]
2 read n
3 for(i : 1 .. n)
4     read a[i]
5     for(j : 1 .. i-1)
6         print a[i]*a[j]
```

Благодаря удобству адресации массив стал основой многих структур данных и методов решения задач.

### 1.1.2 Массив подсчёта

В самом базовом случае массив подсчёта для последовательности $a$ для каждого своего индекса $i$ хранит **количество вхождений числа $i$ в последовательность**. Эти значения достигаются проходом по последовательности $a$, и для каждого элемента последовательности $a[j]$ увеличением элемента $p[a[j]]$ на 1 (до прохода в $p$ должны быть нули).
Полученный массив подсчёта можно использовать для проверки факта наличия того или иного значения в последовательности, а также для вывода последовательности в отсортированном виде. Последнее имеет смысл, когда последовательность длинная, а разница между минимальным и максимальным элементом маленькая, т.е. мало различных элементов; такая сортировка называется *сортировкой подсчётом*.

Очевидно, что массив подсчёта также можно использовать в задачах, где важно поддерживать инфомацию о количестве тех или иных значений.
Пусть для некоторой строки $s$ из маленьких латинских букв требуется сказать, можно ли переставить в ней буквы таким образом, чтобы получился палиндром (слово, которое читается одинаково слева направо и справа налево).
Рассуждения должны быть такими: в палиндроме есть левая и правая половины, которые по сути повторяют друг друга. Это значит, что если из набора символов можно составить палиндром, количество каждого из символов в нём должно быть чётно. Если мы рассматриваем палиндром нечётной длины, помимо левой и правой частей у него есть ещё один центральный элемент, у которого нет пары. Следовательно, если длина исходной строки чётная, для составления палиндрома каждый символ должен встречаться в ней чётное количество раз, а если длина нечётная, то должен быть ровно один символ, который встречается нечётное количество раз.
Благодаря особенностям кодирования символов можно использовать их числовые значения в качестве индексов массива подсчёта. Можно сделать использование массива подсчёта более удобным, особенно для работы с отрицательными числами, если для элемента $a[i]$ количество его вхождений будет храниться не в $p[a[i]]$, а в $p[a[i] - \alpha]$, где $\alpha$ — это некоторый целочисленный сдвиг, в идеале равный минимальному элементу последовательности, чтобы его индекс в массиве подсчёта был равным нулю.
Пример:

$$a = \boxed{b}\boxed{e}\boxed{b}\boxed{a}\boxed{e}\boxed{a}\boxed{e}\ ,\ p = \boxed{2}\boxed{2}\boxed{0}\boxed{0}\boxed{3}$$

Все, кроме одного элемента, встречаются чётное количество раз — палиндром составить можно (например, *abeeeba*).

$$a = \boxed{b}\boxed{e}\boxed{a}\boxed{a}\boxed{e}\boxed{a}\boxed{e}\ ,\ p = \boxed{3}\boxed{1}\boxed{0}\boxed{0}\boxed{3}$$

Больше одного элемента встречаются нечётное количество раз — палиндром составить нельзя.

### 1.1.3 Префиксные суммы

В задаче требуется быстро считать на отрезке последовательности сумму. Количество элементов не позволяет для каждого запроса пробегаться по последовательности, так как это займёт слишком много времени: $O(q \times n)$, где $q$ — количество запросов, а $n$ — количество элементов последовательности.
В таком случае можно применить *предподсчёт* — предварительный подсчёт значений, который значительно сократит время вычислений основного блока программы.
В данной задаче потребуется построить массив *префиксной суммы*. Суть массива будет заключаться в следующем: каждый элемент массива будет хранить **сумму всех чисел последовательности с индексами от первого до текущего**.
Пусть исходная последовательность хранится в массиве $a$, и нумерация элементов начинается с 0. Тогда префиксная сумма $p$ строится по следующей формуле:

$$p[i] = \begin{cases} a[i], & i = 0; \\ p[i-1] + a[i], & i > 0. \end{cases}$$

Теперь мы можем за время $O(1)$ считать сумму на отрезке $[0, r]$, где $r$ — заданная правая граница отрезка. Такой отрезок называется *префиксом длины r*.
Что же делать, когда надо посчитать сумму на отрезке $[l, r]$ с заданной границей $l$? Рассмотрим пример. Пусть $a = \{3, 4, 2, 5, 1\}$. Тогда $p = \{3, 7, 9, 14, 15\}$. Пусть требуется посчитать сумму на отрезке $[2, 4]$.
Нам известна сумма $p[4] = 15 = 3 + 4 + 2 + 5 + 1$, но для ответа она содержит в себе лишние первые два слагаемых $3 + 4$. Но ведь первые два слагаемых — это $p[1]$!
Получается, сумма на отрезке $[2, 4] = p[4] - p[1]$. В общем случае:

$$\sum_{[l,r]} a = p[r] - p[l-1]$$

С помощью этой формулы подсчёт суммы на отрезке выполняется за одну операцию. Таким образом, алгоритм теперь использует $O(n + q)$ времени: $O(n)$ на предподсчёт $n$ префиксных сумм и $O(q)$ на выполнение $q$ запросов. От "квадрата" мы перешли к "линии".
Обратите внимание: в случае, если придётся всё же посчитать сумму на отрезке $[0, r]$, указанная выше формула не сработает, потому что мы попытаемся обратиться к элементу $p[0 - 1] = p[-1]$. Чтобы этого избежать можно либо добавить проверку на этот случай:

$$\sum_{[l,r]} a = \begin{cases} p[r], & l = 0; \\ p[r] - p[l-1], & l > 0 \end{cases}$$

Либо нумеровать массивы с единицы, и в таком случае можно присвоить элементу $p[0]$ значение 0.

### 1.1.4 Дельта-кодирование

В задаче требуется быстро прибавлять значение на отрезках последовательности, и затем вывести результат. Так же, как в предыдущей задаче, количество элементов не позволяет пробегаться по последовательности и прибавлять значения к элементам для каждого запроса (асимптотика $O(q \times n)$, где $q$ — количество запросов, а $n$ — количество элементов последовательности).
Для начала упростим условие задачи: пусть требуется прибавлять значение на отрезке от данной левой границы $l$ до конца поледовательности. Такой отрезок называют *суффиксом*.
В этом случае мы можем не прибавлять значение к каждому числу последовательности, а **запомнить, что значение должно быть прибавлено после определённой позиции**.
Рассмотрим пример: пусть последовательность $a =$ | 2 | 4 | 5 | 1 | 3 |, и есть запрос на прибавление числа 10 на суффиксе, начинающемся в позиции 2 (индексация с нуля). В результате прибавления итоговую последовательность можно представить как | $2+0$ | $4+0$ | $5+10$ | $1+10$ | $3+10$ |, как будто изначально, идя по массиву слева направо, мы прибавляли к его элементам гипотетическое значение $\Delta = 0$, но на позиции 2

оно увеличилось на 10. Если мы также хотим прибавить значение 3 на суффиксе от позиции 1, последовательность примет вид $\boxed{2+0 \mid 4+3 \mid 5+10+3 \mid 1+10+3 \mid 3+10+3}$, то есть к $\Delta$ сначала прибавляется 3 на позиции 1, а затем **ещё** 10 на позиции 2.
Так как при выводе массива действительно приходится обходить его "слева направо", можно сделать это гипотетическое значение реальным.
Создадим ещё один массив $d$, изначально хранящий нули. Этот массив будет хранить **значение, которое нужно будет прибавить к $\Delta$ на текущей позиции в последовательности**. Значение $\Delta$ — это разница между исходными значениями массива и теми, которые должны получаться в результате.
Если требуется прибавить значение $x$ на суффиксе от индекса $l$, нужно просто **прибавить это значение к элементу** массива $d$ на позиции $l$: $d[l] = d[l] + x$. При изменении массива $d$ необходимо не присваивать элементу новое значение, а именно прибавлять его к уже имеющемуся значению, так как при присваивании сотрётся информация о предыдущем прибавлении на этом суффиксе.
В случае примера, приведённого выше:

| $a =$ | 2 | 4 | 5 | 1 | 3 |
|---|---|---|---|---|---|
| $d =$ | 0 | 3 | 10 | 0 | 0 |
| $\Delta =$ | 0 | 3 | 13 | 13 | 13 |
| $a[i] + \Delta$ | 2 | 7 | 18 | 14 | 16 |

Во время вывода последовательности её итоговые значения высчитываются следующим образом:

```
delta = 0
for(i : 0 .. n-1)
    delta = delta + d[i]
    print a[i] + delta
```

Но что же делать с изначальным условием задачи, когда требуется прибавлять на отрезке, а не на суффиксе? Всё очень просто: чтобы прибавить значение $x$ на отрезке $[l, r]$, нужно прибавить число $x$ на суффиксе от $l$ и прибавить число $-x$ на суффиксе от $r + 1$, тем самым компенсируя предыдущее прибавление:

| $d =$ | 0 | 10 | 0 | 0 | $-10$ |
|---|---|---|---|---|---|
| $\Delta =$ | 0 | 10 | 10 | 10 | 0 |

### 1.1.5 Стек

В задаче требуется выполнять запросы трёх типов:

1. Добавить элемент в множество;
2. Удалить последний добавленный элемент из множества;
3. Среди всех элементов непустого множества найти и вывести минимальный.

### Реализация на массиве

Остановимся сначала на первых двух запросах. Множество, в которое можно добавлять элементы, а извлекать можно только самые поздние из добавленных, называется *стеком*. Пример стека в реальной жизни — стопка: можно складывать книги в стопку, а снять книгу из стопки без рисков обвала можно, только если она лежит сверху. Такая структура данных называется ещё *LIFO (Last In, First Out)* — последним зашёл, первым вышел.
Стек можно реализовать на массиве, поддерживая длину стека $n$. Если $n = 0$, стек пуст. При добавлении элемента в стек значение записывается в массив по индексу $n$, а затем значение $n$ увеличивается на 1. При удалении элемента из стека $n$ просто уменьшается на 1. Чтобы получить последний добавленный элемент, надо обратиться к массиву по индексу $n - 1$.

### Минимум на стеке

Вернёмся к задаче.
Идея решения состоит в поддержании минимума на стеке: вместо того, чтобы по запросу искать минимальный элемент за линейное время, минимумы считаются на ходу при добавлении новых элементов.
Заведём два стека: $st$ будет хранить элементы множества, а $min$ — минимумы на множестве после добавления очередного элемента. Изначально стеки пусты. Пусть первые несколько запросов будут на добавление в множество чисел 4, 2, 5 и 1.
При добавлении элемента в пустое множество элемент просто добавляется в оба стека: минимум на множестве из одного числа равно этому самому числу.

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | 4 | | | | |
| $min =$ | 4 | | | | |

Далее в множество $st$ добавляется новый элемент 2, и требуется пересчитать минимум. Если крайний элемент стека минимумов $min.top() \leq st.top()$, в стек $min$ добавляется значение $min.top()$, так как минимум не изменился; в противном случае в $min$ добавляется $st.top()$.
Так как старый минимум 4 больше нового элемента 2, новый минимум становится равен 2.

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | 4 | 2 | | | |
| $min =$ | 4 | 2 | | | |

При добавлении 5 минимум не обновляется, так как $2 \leq 5$:

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | 4 | 2 | 5 | | |
| $min =$ | 4 | 2 | 2 | | |

Но $2 > 1$:

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | 4 | 2 | 5 | 1 | |
| $min =$ | 4 | 2 | 2 | 1 | |

Теперь, чтобы ответить за запрос о взятии минимума на множестве, мы может ответить крайним элементом множества минимумов $min.top()$.
При удалении элемента из множества мы также удалим соответствующий минимум из стека минимумов:

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | 4 | 2 | 5 | ~~1~~ | |
| $min =$ | 4 | 2 | 2 | ~~1~~ | |

На самом деле, хранить сами элементы множества в стеке $st$ для данной задачи необязательно, так как требуется знать только минимумы, а новый элемент можно хранить в одной переменной.

### 1.1.6 Очередь

В задаче требуется выполнять запросы трёх типов:

1. Добавить элемент в множество;
2. Удалить самый первый добавленный элемент из множества;
3. Среди всех элементов непустого множества найти и вывести минимальный.

#### Реализация на массиве

Остановимся сначала на первых двух запросах. Множество, в которое можно добавлять элементы, а извлекать можно только самые первые из добавленных, называется *очередью*. Пример очереди в реальной жизни — собственно, очередь: ты встаёшь в конец (*хвост*) очереди, а покинуть её ты можешь, только дойдя до её начала (*головы*). Такая структура данных называется ещё *FIFO (First In, First Out)* — первым зашёл, первым вышел.
Очередь можно реализовать на массиве, храня дополнительно два значения: индекс начала очереди $head$ и индекс элемента **после** последнего элемента очереди $tail$. Если $head = tail$, очередь пуста. При добавлении элемента в очередь, как в стеке, значение записывается в массив по индексу $tail$, а затем значение $tail$ увеличивается на 1. При удалении элемента из очереди значение $head$ увеличивается на 1. Чтобы получить первый добавленный элемент, нужно обратиться к массиву по индексу $head$.
Чтобы не происходило переполнения массива, очередь можно закольцевать на нём: если значение $tail$ превышает допустимую границу, его меняют на 0. Пока хвост очереди не дорастёт до головы, в такую очередт можно добавлять новые элементы. С значением $head$ можно делать то же самое.
Также очередь можно реализовать на двух стеках! Рассмотрим эту реализацию на примере решения данной выше задачи.

#### Минимум на очереди. Реализация на стеках

Идея решения так же состоит в поддержании минимума на стеке, как в пункте выше.

Заведём три стека: $st$ будет хранить элементы множества, $minIn$ и $minOut$ — минимумы на множестве после добавления очередного элемента. Изначально стеки пусты. Пусть первые несколько запросов будут на добавление в множество чисел 3, 1, 4 и 5.
До тех пор, пока требуется только добавлять элементы, мы добавляем их в стек $st$ и $minIn$, как будто мы ищем минимум на стеке:

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | 3 | 1 | 4 | 5 | |
| $minIn =$ | 3 | 1 | 1 | 1 | |
| $minOut =$ | | | | | |

Когда требуется удалить элемент из множества или взять минимум, мы будет производить эти операции со стеком минимумов $minOut$. Если стек пуст, как сейчас в примере, его требуется сформировать следующим образом: элементы из стека $st$ по одному добавляются в множество, опять же, как если бы это было решение задачи о минимуме на стеке. Единственное отличие: элементы идут в обратном порядке.
Одновременно с тем, как элементы удаляются из $st$, соответствующие минимумы удаляются из $minIn$.
При добавлении первого элемента в $minOut$ он записывается в него без изменений, потому что минимум из одного числа — это само число.

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | 3 | 1 | 4 | ~~5~~ | |
| $minIn =$ | 3 | 1 | 1 | ~~1~~ | |
| $minOut =$ | 5 | | | | |

При добавлении 4 и 1 минимум обновляется, так как $5 > 4$, а $4 > 1$:

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | 3 | ~~1~~ | ~~4~~ | | |
| $minIn =$ | 3 | ~~1~~ | ~~1~~ | | |
| $minOut =$ | 5 | 4 | 1 | | |

Но $1 \leq 3$:

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | ~~3~~ | | | | |
| $minIn =$ | ~~3~~ | | | | |
| $minOut =$ | 5 | 4 | 1 | 1 | |

Теперь, чтобы ответить за запрос о взятии минимума на множестве, мы можем ответить крайним элементом множества минимумов $minOut.top()$. Действительно, минимум из множества $\{5, 4, 1, 3\} = 1$. **Но это верно только в случае, когда стеки $st$ и $minIn$ пусты!**
При удалении элемента из множества мы удалим соответствующий минимум из стека минимумов $minOut$:

| | | | | | |
|---|---|---|---|---|---|
| $st =$ | | | | | |
| $minIn =$ | | | | | |
| $minOut =$ | 5 | 4 | 1 | ~~1~~ | |

Если требуется взять минимум на множестве, когда стеки $minOut$ и $minIn$ непусты одновременно, ответом станет минимум из крайних элементов стеков $minIn.top()$ и $minOut.top()$.

### 1.1.7 Дек

Структура данных deque (*Double-Ended QUEue* - очередь о дух концах) выполняет те же функции, что и обычная очередь, но симметрично с двух сторон, если провести аналогию — это очередь, которая может менять направление на обратное.

#### Реализация на массивах

Реализация основана на очереди с добавленными методами, позволяющими добавить элемент в начало очереди и удалить элемент из конца.

#### Минимум на деке

Задачу про минимум на очереди можно также решать, используя дек. Этот способ менее затратен по памяти.
Заведём дек $deq$ и очередь $q$, которая будет иметь такой же функционал, как стек $st$ в задаче о минимуме на стеке: наглядность и больше ничего.
Дек изначально пуст. Дек будет играть роль очереди с дополнительной операцией удаления с конца. Добавлять элементы будем с "правой" стороны дека, а получать минимум с "левой".
Когда требуется добавить элемент в множество, надо сначала проверить уже имеющиеся в нём элементы. Если крайний справа элемент меньше того, что требуется добавить — он добавляется в дек. Что происходит иначе, описано ниже.
Пусть первые несколько запросов будут на добавление в множество чисел 1, 2, 4 и 5.

| | | | | | |
|---|---|---|---|---|---|
| $q =$ | 1 | 2 | 4 | 5 | |
| $deq =$ | 1 | 2 | 4 | 5 | |

Когда требуется получить минимум на множестве, ответ хранится слева. Действительно, минимум из множества $1, 2, 4, 5 = 1$. При удалении крайнего элемента из очереди минимум продолжает поддерживаться: минимум из $2, 4, 5 = 2$

| | | | | | |
|---|---|---|---|---|---|
| $q =$ | ~~1~~ | 2 | 4 | 5 | |
| $deq =$ | ~~1~~ | 2 | 4 | 5 | |

| | | | | | |
|---|---|---|---|---|---|
| $q =$ | 2 | 4 | 5 | | |
| $deq =$ | 2 | 4 | 5 | | |

Таким образом, в деке **всегда поддерживается неубывающая** последовательность чисел.
Если при добавлении элемента во множество крайний элемент оказывается больше него, производится следующая операция: для всех элементов в деке, которые больше добавляемого значения, их значение заменяется на добавляемое. Это делается потому, что числа с худшим ответом на текущем множестве больше не нужны.

Например, в дек добавляют элемент 3:

| | | | | | |
|---|---|---|---|---|---|
| $q =$ | 2 | 4 | 5 | 3 | |
| $deq =$ | 2 | ④ | ⑤ | 3 | |

Обведённые числа имеют значение большее, чем добавленное.

| | | | | | |
|---|---|---|---|---|---|
| $q =$ | 2 | 4 | 5 | 3 | |
| $deq =$ | 2 | 3 | 3 | 3 | |

При удалении чисел из множества минимум на деке продолжает поддерживаться: минимум из $5, 3 = 3$.

| | | | | | |
|---|---|---|---|---|---|
| $q =$ | ~~2~~ | ~~4~~ | 5 | 3 | |
| $deq =$ | ~~2~~ | ~~3~~ | 3 | 3 | |

Для большей оптимизации времени работы алгоритма и используемой памяти в деке можно хранить не просто значения, а пары *значение* - *количество*, которые означают. что в очереди на самом деле идут подряд *количество* одинаковых *значений*. Вместо замены значений элементов можно удалять крайние, а их *количества* прибавлять к будущему *количеству* пары с добавленным значением. Когда требуется удалить элемент из такого множества, у крайней левой пары в деке *количество* уменьшается на 1. Элемент справа удаляется только тогда, когда его *количество* стало равно нулю.