[Сортировки 5](#_Toc468703135)

[Пузырьковая 5](#_Toc1101513929)

[Общая информация 5](#_Toc1801979802)

[Алгоритм 5](#_Toc1315634022)

[Перемешиванием (Шейкерная) 6](#_Toc635723058)

[Общая информация 6](#_Toc1369133069)

[Алгоритм 6](#_Toc1125898167)

[Расческой 7](#_Toc1059961393)

[Общая информация 7](#_Toc2089018456)

[Алгоритм 7](#_Toc628175011)

[Вставками 8](#_Toc1656478042)

[Общая информация 8](#_Toc1131176229)

[Алгоритм 8](#_Toc1653377373)

[Выбором 9](#_Toc859484421)

[Общая информация 9](#_Toc1914544919)

[Алгоритм 9](#_Toc608413784)

[Быстрая 10](#_Toc756898537)

[Общая информация 10](#_Toc1734575198)

[Алгоритм 10](#_Toc1973594324)

[Слиянием 11](#_Toc149798315)

[Общая информация 11](#_Toc2038664370)

[Алгоритм 11](#_Toc1129566413)

[Пирамидальная 12](#_Toc184803526)

[Общая информация 12](#_Toc412776091)

[Алгоритм 12](#_Toc1424268980)

[Полезные ссылки 13](#_Toc1911759956)

[Структуры данных 14](#_Toc749241873)

[Списки 14](#_Toc137806862)

[Односвязный список 14](#_Toc42999170)

[Метод Add 14](#_Toc982906996)

[Метод Contains 14](#_Toc135497281)

[Метод Clear 14](#_Toc511702305)

[Метод Remove 15](#_Toc2084420925)

[Двусвязный список 16](#_Toc1937477084)

[Метод AddFirst 16](#_Toc1827336327)

[Метод AddLast 16](#_Toc572660336)

[Метод RemoveFirst 16](#_Toc1159126505)

[Метод RemoveLast 17](#_Toc805750846)

[Метод Remove 17](#_Toc1632621729)

[Стеки и очереди 18](#_Toc1100661313)

[Стэк (LIFO) на основе списка 18](#_Toc1433925857)

[Введение 18](#_Toc1141616124)

[Метод Push 18](#_Toc84353895)

[Метод Pop 18](#_Toc939819582)

[Метод Peek 18](#_Toc2001100545)

[Очередь (FIFO) на основе списка 19](#_Toc1998898814)

[Введение 19](#_Toc1548233367)

[Метод Enqueue 19](#_Toc610515434)

[Метод Dequeue 19](#_Toc1585990364)

[Метод Peek 19](#_Toc1374344043)

[Двусторонняя очередь (Deque) на основе списка 20](#_Toc760313750)

[Введение 20](#_Toc1477171087)

[Метод EnqueueFirst 20](#_Toc356426808)

[Метод EnqueueLast 20](#_Toc945117276)

[Метод DequeueFirst 20](#_Toc1889947178)

[Метод DequeueLast 20](#_Toc1780695788)

[Метод PeekFirst 21](#_Toc709393584)

[Метод PeekLast 21](#_Toc491705403)

[Двусторонняя очередь (Deque) на основе массива (Кольцевой буфер) 22](#_Toc1918502651)

[Хранение элементов в массиве 22](#_Toc752392754)

[Алгоритм роста 23](#_Toc1474612399)

[Метод EnqueueFirst 24](#_Toc2053999932)

[Метод EnqueueLast 25](#_Toc1264095060)

[Метод DequeueFirst 26](#_Toc1411549676)

[Метод DequeueLast 27](#_Toc1843993368)

[Метод PeekFirst 28](#_Toc943947739)

[Метод PeekLast 28](#_Toc1984210012)

[Графы 29](#_Toc855636226)

[Основные понятия 29](#_Toc1749698586)

[Классификация графов 29](#_Toc1469348094)

[Способы представления графов 31](#_Toc1956297539)

[1. Матричные структуры данных 31](#_Toc1036140795)

[2. Перечислительные структуры данных 32](#_Toc463480570)

[Алгоритмы обхода графов 34](#_Toc2040651434)

[Поиск в ширину 34](#_Toc1975960378)

[Поиск в глубину 37](#_Toc317097467)

[Деревья 41](#_Toc1892066601)

[Основные понятия 41](#_Toc1376710097)

[К -арное дерево 42](#_Toc927612902)

[Бинарное дерево поиска 43](#_Toc1330573317)

[Двоичная Куча 44](#_Toc603570492)

[Биноминальная Куча 45](#_Toc1687926652)

[Очередь с приоритетом 46](#_Toc660260756)

[Полезные ссылки 47](#_Toc959997301)

[Шахматные алгоритмы 48](#_Toc485560280)

[Вспомогательные функции и структуры 49](#_Toc402724286)

[Король 51](#_Toc593209441)

[Все возможные ходы 51](#_Toc1194953865)

[Поиск пути из т. a в т. b 51](#_Toc894429689)

[Встреча двух фигур 51](#_Toc364228444)

[Ферзь 52](#_Toc1947346619)

[Все возможные ходы 52](#_Toc221558440)

[Поиск пути из т. a в т. b 52](#_Toc270744729)

[Встреча двух фигур 52](#_Toc1063958031)

[Ладья 53](#_Toc1633108117)

[Все возможные ходы 53](#_Toc2114738097)

[Поиск пути из т. a в т. b 53](#_Toc2007905771)

[Встреча двух фигур 53](#_Toc1469834481)

[Слон 54](#_Toc822890675)

[Все возможные ходы 54](#_Toc1610120709)

[Поиск пути из т. a в т. b 54](#_Toc791698927)

[Встреча двух фигур 54](#_Toc631704567)

[Конь 55](#_Toc498777856)

[Все возможные ходы 55](#_Toc1255179497)

[Поиск пути из т. a в т. b 55](#_Toc524872353)

[Встреча двух фигур 55](#_Toc327254586)

[Пешка 56](#_Toc1572276965)

[Все возможные ходы 56](#_Toc269455306)

[Поиск пути из т. a в т. b 57](#_Toc1703964683)

[Встреча двух фигур 57](#_Toc352406219)

[(Бонус) Шашка 59](#_Toc1600028624)

[Все возможные расстановки 59](#_Toc160051528)

[Поиск пути из т. a в т. b 59](#_Toc2040332871)

[Встреча двух фигур 59](#_Toc112805732)

[Путь в дамки 59](#_Toc1120048829)

[Полезные ссылки 60](#_Toc378409503)

# Сортировки

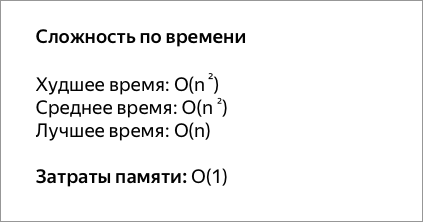
## Пузырьковая

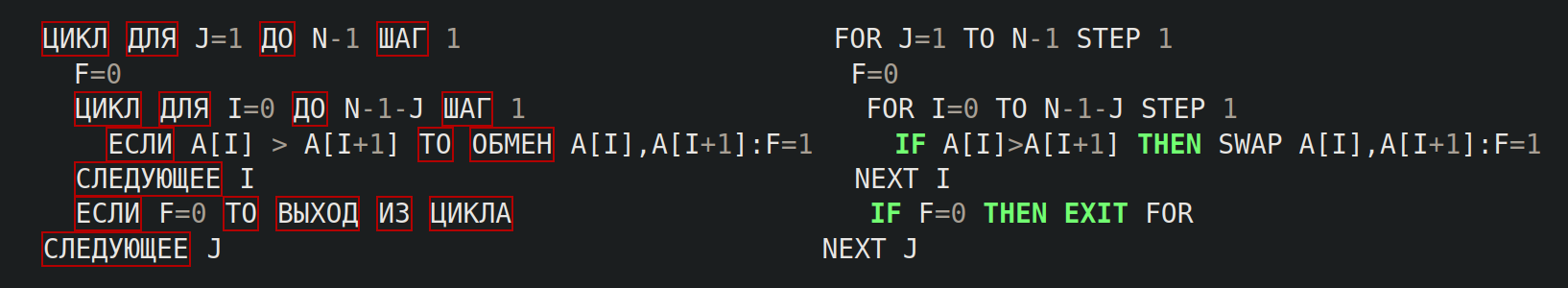
### Общая информация

Сортировка простыми обменами, сортировка пузырьком — простой [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8" \o "Алгоритм сортировки). Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов.

Алгоритм считается учебным и практически не применяется вне учебной литературы, вместо него на практике применяются более эффективные алгоритмы сортировки. В то же время метод сортировки обменами лежит в основе некоторых более совершенных алгоритмов, таких как [шейкерная сортировка](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D0%B9%D0%BA%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0" \o "Шейкерная сортировка), [пирамидальная сортировка](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%80%D0%B0%D0%BC%D0%B8%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0" \o "Пирамидальная сортировка) и [быстрая сортировка](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0" \o "Быстрая сортировка).

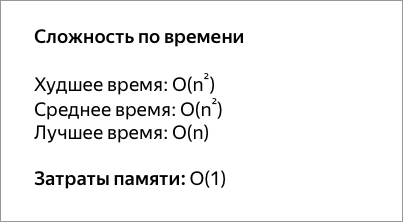
### Алгоритм

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется перестановка элементов. Проходы по массиву повторяются N − 1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма).



## Перемешиванием (Шейкерная)

### Общая информация

****Сортировка перемешиванием, или Шейкерная сортировка, или двунаправленная — разновидность [пузырьковой сортировки](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%BC" \o "Сортировка пузырьком). Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства.

Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения.

Во-вторых, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

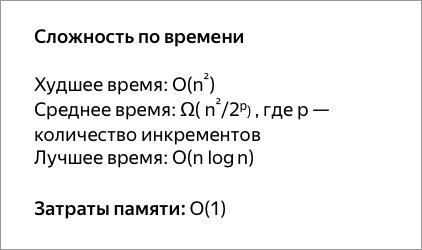
Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (то есть части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.

### Алгоритм



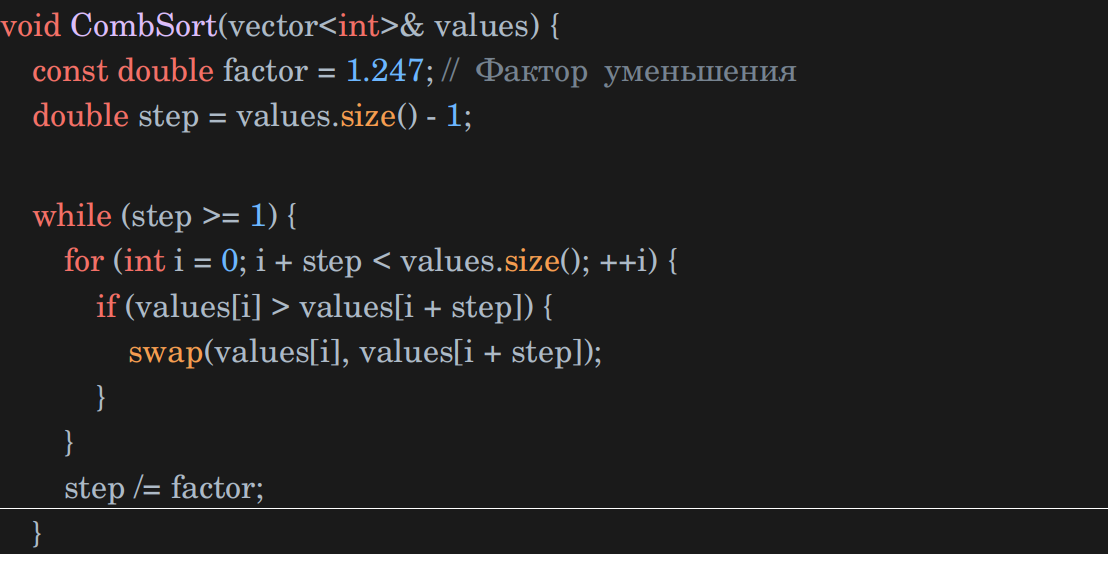
## Расческой

### Общая информация

В «[пузырьке](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%BC" \o "Сортировка пузырьком)», «[шейкере](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D1%88%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%D0%BC" \o "Сортировка перемешиванием)» и «[чёт-нечете](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D1%87%D1%91%D1%82-%D0%BD%D0%B5%D1%87%D0%B5%D1%82" \o "Сортировка чёт-нечет)» при переборе массива сравниваются соседние элементы. Основная идея «расчёски» в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы причёсываем массив, постепенно разглаживая на всё более аккуратные пряди. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения, оптимальное значение которой равно примерно 1,247. Сначала расстояние между элементами максимально, то есть равно размеру массива минус один. Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае сравниваются соседние элементы, как и в сортировке пузырьком, но такая итерация одна.

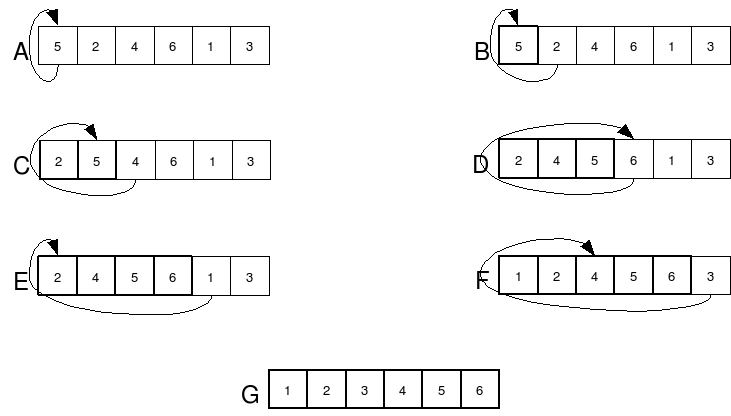
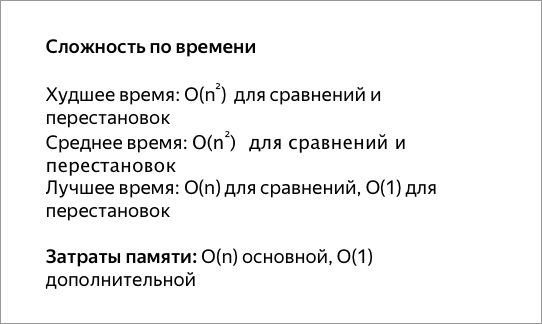
Оптимальное значение фактора уменьшения 1,247... = , где e — [основание натурального логарифма](https://ru.wikipedia.org/wiki/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BD%D0%B0%D1%82%D1%83%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BB%D0%BE%D0%B3%D0%B0%D1%80%D0%B8%D1%84%D0%BC%D0%B0" \o "Основание натурального логарифма), а ϕ — [золотое сечение](https://ru.wikipedia.org/wiki/%D0%97%D0%BE%D0%BB%D0%BE%D1%82%D0%BE%D0%B5_%D1%81%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5" \o "Золотое сечение).

### Алгоритм



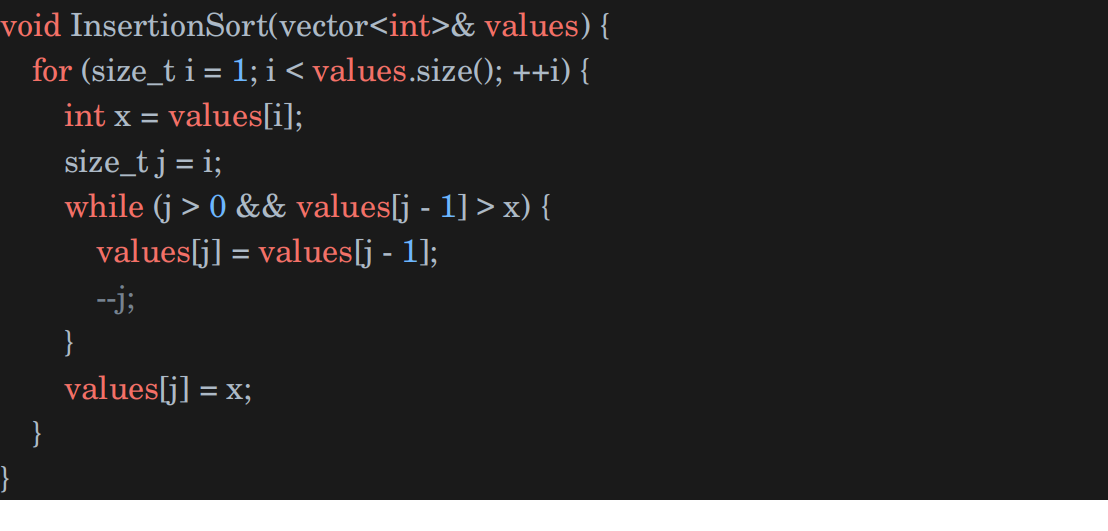
## Вставками

### Общая информация

В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элементы удовлетворяют требованиям к выходным данным алгоритма.

Данный алгоритм можно ускорить при помощи использования бинарного поиска для нахождения места текущему элементу в отсортированной части. Проблема с долгим сдвигом массива вправо решается при помощи смены указателей.

### Алгоритм



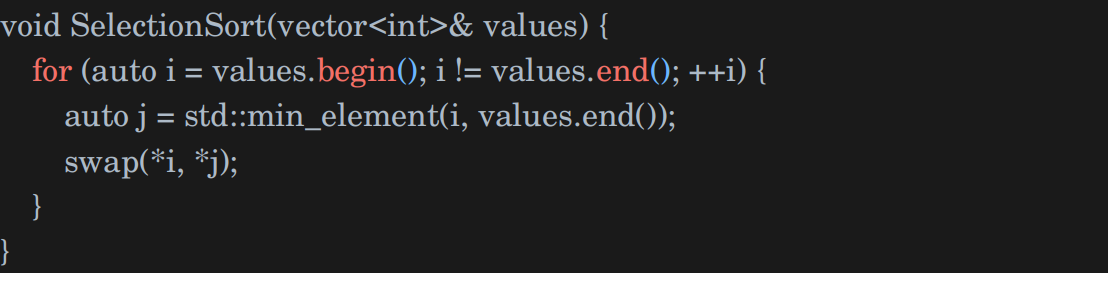
## Выбором

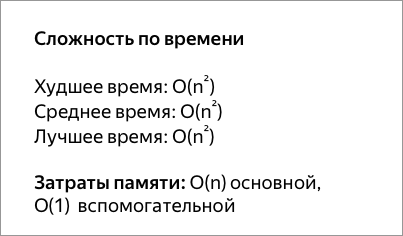
### Общая информация

### Алгоритм

Шаги алгоритма:

1. находим номер минимального значения в текущем списке
2. производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции)
3. теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы



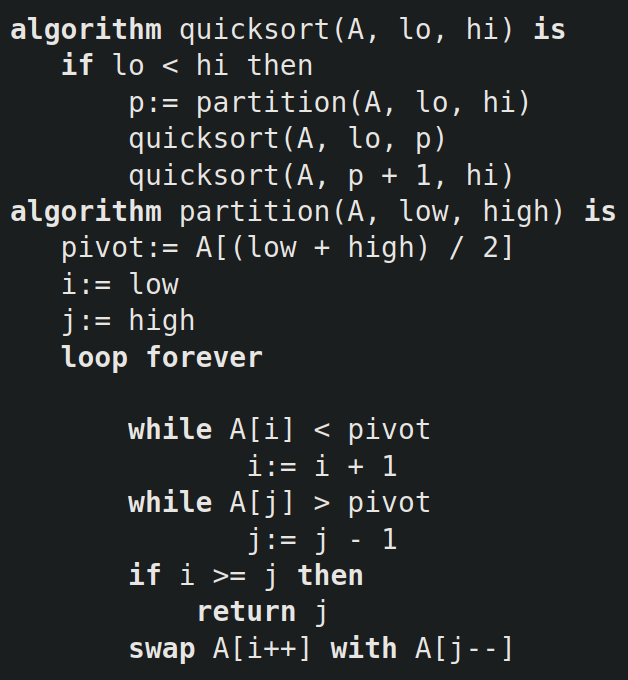
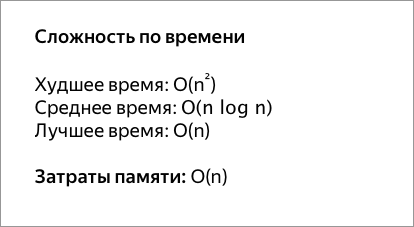


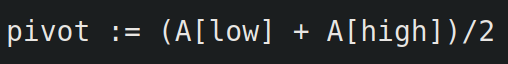
## Быстрая

### Общая информация

### Алгоритм

Разбиение Хоара:

Данная схема использует два индекса (один в начале массива, другой в конце), которые приближаются друг к другу, пока не найдётся пара элементов, где один больше опорного и расположен перед ним, а второй меньше и расположен после. Эти элементы меняются местами. Обмен происходит до тех пор, пока индексы не пересекутся. Алгоритм возвращает последний индекс. Схема Хоара эффективнее схемы Ломуто, так как происходит в среднем в три раза меньше обменов (swap) элементов, и разбиение эффективнее, даже когда все элементы равны. Подобно схеме Ломуто, данная схема также показывает эффективность в O(n2), когда входной массив уже отсортирован. Сортировка с использованием данной схемы нестабильна. Следует заметить, что конечная позиция опорного элемента необязательно совпадает с возвращённым индексом.

Упоминается, что такая реализация алгоритма имеет «много тонких моментов». Вот один из них: выбор в качестве опорного элемента уже существующего элемента в массиве может привести к переполнению стека вызовов из-за того, что номер элемента вычисляется как среднее, а это не целое число. Поэтому логичнее в данном алгоритме выбирать в качестве опорного элемента среднее значение между начальным и конечным элементом:

## Слиянием

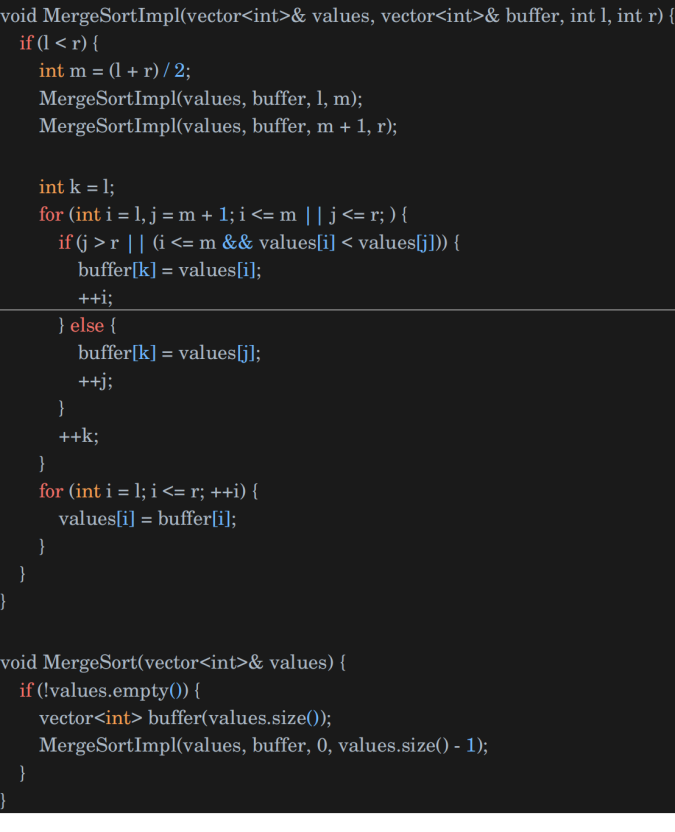
### Общая информация

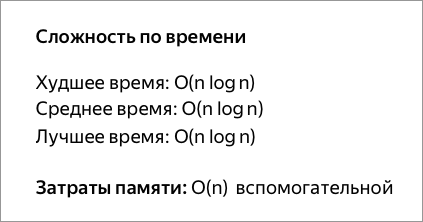
### Алгоритм

Для решения задачи сортировки эти три этапа выглядят так:

1. Сортируемый [массив](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)" \o "Массив (тип данных)) разбивается на две части примерно одинакового размера;
2. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.

1.1. — 2.1. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).



3.1. Соединение двух упорядоченных массивов в один.  
Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем два уже отсортированных по возрастанию подмассива. Тогда:  
3.2. Слияние двух подмассивов в третий результирующий массив.  
На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на 1.  
3.3. «Прицепление» остатка.  
Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.

## Пирамидальная

### Общая информация

### Алгоритм

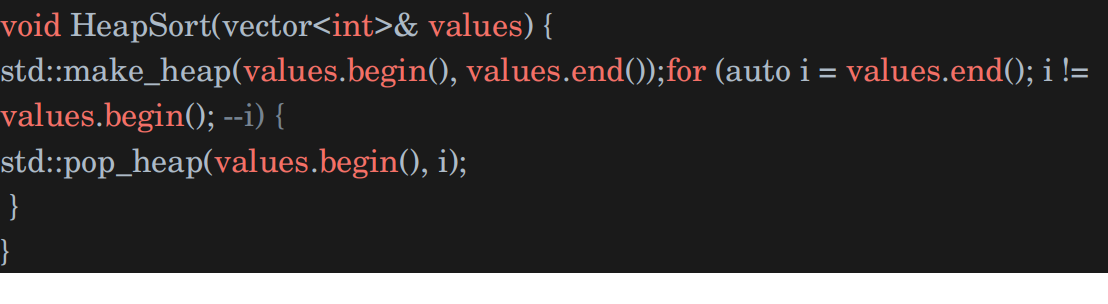
Сортировка пирамидой использует [бинарное сортирующее дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0" \o "Двоичная куча). Сортирующее дерево — это такое дерево, у которого выполнены условия:

1. Каждый лист имеет глубину либо dIMG_256, либо d − 1, d — максимальная глубина дерева.
2. Значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.

Удобная структура данных для сортирующего дерева — такой массив Array, что Array[0] — элемент в корне, а потомки элемента Array[i] являются Array[2i+1] и Array[2i+2].

Алгоритм сортировки будет состоять из двух основных шагов:

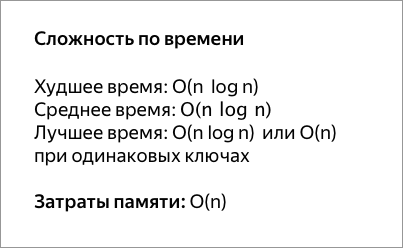
1. Выстраиваем элементы массива в виде сортирующего дерева:



Array [ i ] ≥ Array [ 2 i + 1 ]

Array [ i ] ≥ Array [ 2 i + 2 ]

при 0 ≤ i < n / 2

Этот шаг требует O ( n ) операций.

1. Будем удалять элементы из корня по одному за раз и перестраивать дерево. То есть на первом шаге обмениваем Array[0] и Array[n-1], преобразовываем Array[0], Array[1], … , Array[n-2] в сортирующее дерево. Затем переставляем Array[0] и Array[n-2], преобразовываем Array[0], Array[1], … , Array[n-3] в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда Array[0], Array[1], … , Array[n-1] — упорядоченная последовательность. Этот шаг требует O ( n logn ) операций.

## Полезные ссылки

1. **[Вышеизложенные сортировки](https://academy.yandex.ru/posts/osnovnye-vidy-sortirovok-i-primery-ikh-realizatsii)**

# Структуры данных

## Списки

### Односвязный список

#### Метод Add

● Поведение: Добавляет элемент в конец списка.

● Сложность: O(1) или O(n) если не храним tail.

Добавление элемента в связный список производится в три этапа:

1. Создать экземпляр класса LinkedListNode.
2. Найти последний узел списка.
3. Установить значение поля Next последнего узла списка так, чтобы оно указывало на созданный узел.

#### Метод Contains

● Поведение: Возвращает true или false в зависимости от того, присутствует ли искомый элемент в списке.

● Сложность: O(n)

#### Метод Clear

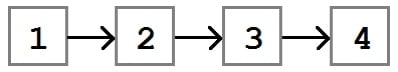
● Поведение: Удаляет все элементы из списка.

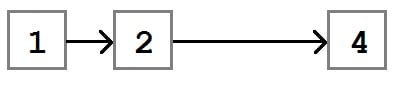
● Сложность: O(n)

#### Метод Remove

● Поведение: Удаляет первый элемент списка со значением, равным переданному. Возвращает true, если элемент был удален и false в противном случае.

● Сложность: O(n)

Основной алгоритм удаления элемента такой:

1. Найти узел, который необходимо удалить.
2. Изменить значение поля Next предыдущего узла так, чтобы оно указывало на узел, следующий за удаляемым.

Как всегда, основная проблема кроется в мелочах. Вот некоторые из случаев, которые необходимо предусмотреть:

● Список может быть пустым, или значение, которое мы передаем в метод может не присутствовать в списке. В этом случает список останется без изменений.

● Удаляемый узел может быть единственным в списке. В этом случае мы установим значения полей \_head и \_tail равными null.

● Удаляемый узел будет в начале списка. В этом случае мы записываем в \_head ссылку на следующий узел.

● Удаляемый узел будет в середине списка.

● Удаляемый узел будет в конце списка. В этом случае мы записываем в \_tail ссылку на предпоследний узел, а в его поле Next записываем null.

### Двусвязный список

#### Метод AddFirst

● Поведение: Добавляет переданный элемент в начало списка

● Сложность: O(1)

При добавлении элемента в начало списка последовательность действий примерно такая же, как и при добавлении элемента в односвязный список.

1. Установить значение поля Next в новом узле так, чтобы оно указывало на бывший первый узел.
2. Установить значение поля Previous в бывшем первом узле так, чтобы оно указывало на новый узел.
3. Обновить поле tail при необходимости и инкрементировать поле Count

#### Метод AddLast

● Поведение: Добавляет переданный элемент в конец списка.

● Сложность: O(1)

Добавление узла в конец списка легче, чем в начало. Мы просто создаем новый узел и обновляем поля head и tail, а затем инкрементируем поле кол-ва элементов списка.

#### Метод RemoveFirst

● Поведение: Удаляет первый элемент списка. Если список пуст, не делает ничего. Возвращает true, если элемент был удален и false в противном случае.

● Сложность: O(1)

RemoveFirst устанавливает ссылку head на второй узел списка и обнуляет поле Previous этого узла, удаляя таким образом все ссылки на предыдущий первый узел. Если список был пуст или содержал только один элемент, то поля head и tail становятся равны null.

### 

#### Метод RemoveLast

● Поведение: Удаляет последний элемент списка. Если список пуст, не делает ничего. Возвращает true, если элемент был удален и false в противном случае.

● Сложность: O(1)

RemoveLast устанавливает значение поля tail так, чтобы оно указывало на предпоследний элемент списка и, таким образом, удаляет последний элемент. Если список был пустым, или содержал только один элемент, то поля head и tail становятся равны null.

#### Метод Remove

● Поведение: Удаляет первый элемент списка со значением, равным переданному. Возвращает true, если элемент был удален и false в противном случае.

● Сложность: O(n)

Метод ICollection<T>.Remove() почти такой же, как и в односвязном списке. Единственное отличие — теперь нам необходимо поменять значение поля Previous при удалении узла. Для того, чтобы не повторять код, этот метод зовет RemoveFirst при удалении первого узла.

## Стеки и очереди

### Стэк (LIFO) на основе списка

#### Введение

Стек — коллекция, элементы которой получают по принципу Last-In-First-Out (LIFO.) Это значит, что мы будем иметь доступ только к последнему добавленному элементу.

В отличие от списков, мы не можем получить доступ к произвольному элементу стека. Мы можем только добавлять или удалять элементы с помощью специальных методов. У стека нет также метода Contains, как у списков.

#### Метод Push

● Поведение: Добавляет элемент на вершину стека.

● Сложность: O(1).

Поскольку мы используем связный список для хранения элементов, можно просто добавить новый в конец списка.

#### Метод Pop

● Поведение: Удаляет элемент с вершины стека и возвращает его. Если стек пустой, кидает InvalidOperationException.

● Сложность: O(1).

Push добавляет элементы в конец списка, поэтому забирать их будет также с конца. В случае, если список пуст, будет выбрасываться исключение.

#### Метод Peek

● Поведение: Возвращает верхний элемент стека, но не удаляет его. Если стек пустой, кидает InvalidOperationException.

● Сложность: O(1).

### Очередь (FIFO) на основе списка

#### Введение

Очереди очень похожи на стеки. Они также не дают доступа к произвольному элементу, но, в отличие от стека, элементы кладутся (enqueue) и забираются (dequeue) с разных концов. Такой метод называется «первый вошел, первый вышел» (First-In-First-Out или FIFO). То есть забирать элементы из очереди мы будем в том же порядке, что и клали. Как реальная очередь или конвейер.

Очереди часто используются в программах для реализации буфера, в который можно положить элемент для последующей обработки, сохраняя порядок поступления. Например, если база данных поддерживает только одно соединение, можно использовать очередь потоков, которые будут, как ни странно, ждать своей очереди на доступ к БД.

#### Метод Enqueue

● Поведение: Добавляет элемент в очередь.

● Сложность: O(1).

Новые элементы очереди можно добавлять как в начало списка, так и в конец. Важно только, чтобы элементы доставались с противоположного края. В данной реализации мы будем добавлять новые элементы в начало внутреннего списка.

#### Метод Dequeue

● Поведение: Удаляет первый помещенный элемент из очереди и возвращает его. Если очередь пустая, кидает InvalidOperationException.

● Сложность: O(1).

Поскольку мы вставляем элементы в начало списка, убирать мы их будем с конца. Если список пуст, кидается исключение.

#### Метод Peek

● Поведение: Возвращает элемент, который вернет следующий вызов метода Dequeue. Очередь остается без изменений. Если очередь пустая, кидает InvalidOperationException.

● Сложность: O(1).

### Двусторонняя очередь (Deque) на основе списка

#### Введение

Двусторонняя очередь (Double-ended queue), или дек (Deque), расширяет поведение очереди. В дек можно добавлять или удалять элементы как с начала, так и с конца очереди. Такое поведение полезно во многих задачах, например, планирование выполнения потоков или реализация других структур данных. Позже мы рассмотрим вариант реализации стека с помощью двусторонней очереди.

Класс Deque проще всего реализовать с помощью двусвязного списка. Он позволяет просматривать, удалять и добавлять элементы в начало и в конец списка. Основное отличие двусторонней очереди от обычной — методы Enqueue, Dequeue, и Peek разделены на пары для работы с обоими концами списка.

#### Метод EnqueueFirst

● Поведение: Добавляет элемент в начало очереди. Этот элемент будет взят из очереди следующим при вызове метода DequeueFirst.

● Сложность: O(1).

#### Метод EnqueueLast

● Поведение: Добавляет элемент в конец очереди. Этот элемент будет взят из очереди следующим при вызове метода DequeueLast.

● Сложность: O(1).

#### Метод DequeueFirst

● Поведение: Удаляет элемент из начала очереди и возвращает его. Если очередь пустая, кидает InvalidOperationException.

● Сложность: O(1).

#### Метод DequeueLast

● Поведение: Удаляет элемент с конца очереди и возвращает его. Если очередь пустая, кидает InvalidOperationException.

● Сложность: O(1).

#### Метод PeekFirst

● Поведение: Возвращает элемент из начала очереди, не изменяя ее. Если очередь пустая, кидает InvalidOperationException.

● Сложность: O(1).

#### Метод PeekLast

● Поведение: Возвращает элемент с конца очереди, не изменяя ее. Если очередь пустая, кидает InvalidOperationException.

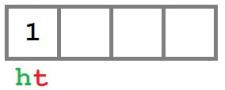
● Сложность: O(1).

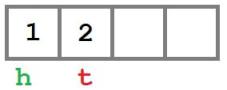
### Двусторонняя очередь (Deque) на основе массива (Кольцевой буфер)

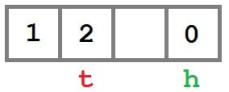
#### Хранение элементов в массиве

При создании очереди у нее внутри создается массив нулевой длины. Красные буквы «h» и «t» означают указатели \_head и \_tail соответственно.

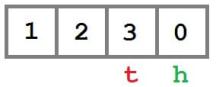
Deque deq = new Deque();

deq.EnqueueFirst(1);

deq.EnqueueLast(2);

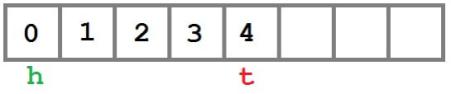
deq.EnqueueFirst(0);

Обратите внимание: индекс «головы» очереди перескочил в начало списка. Теперь первый элемент, который будет возвращен при вызове метода DequeueFirst — 0 (индекс 3).

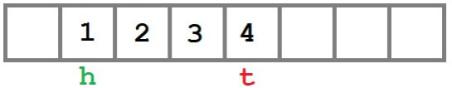
deq.EnqueueLast(3);

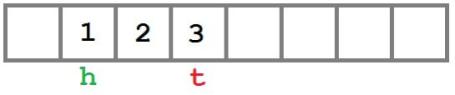
Массив заполнен, поэтому при добавлении элемента произойдет следующее:

1. Алгорим роста определит размер нового массива.
2. Элементы скопируются в новый массив с «головы» до «хвоста».
3. Добавится новый элемент.

deq.EnqueueLast(4);

Теперь посмотрим, что происходит при удалении элемента:

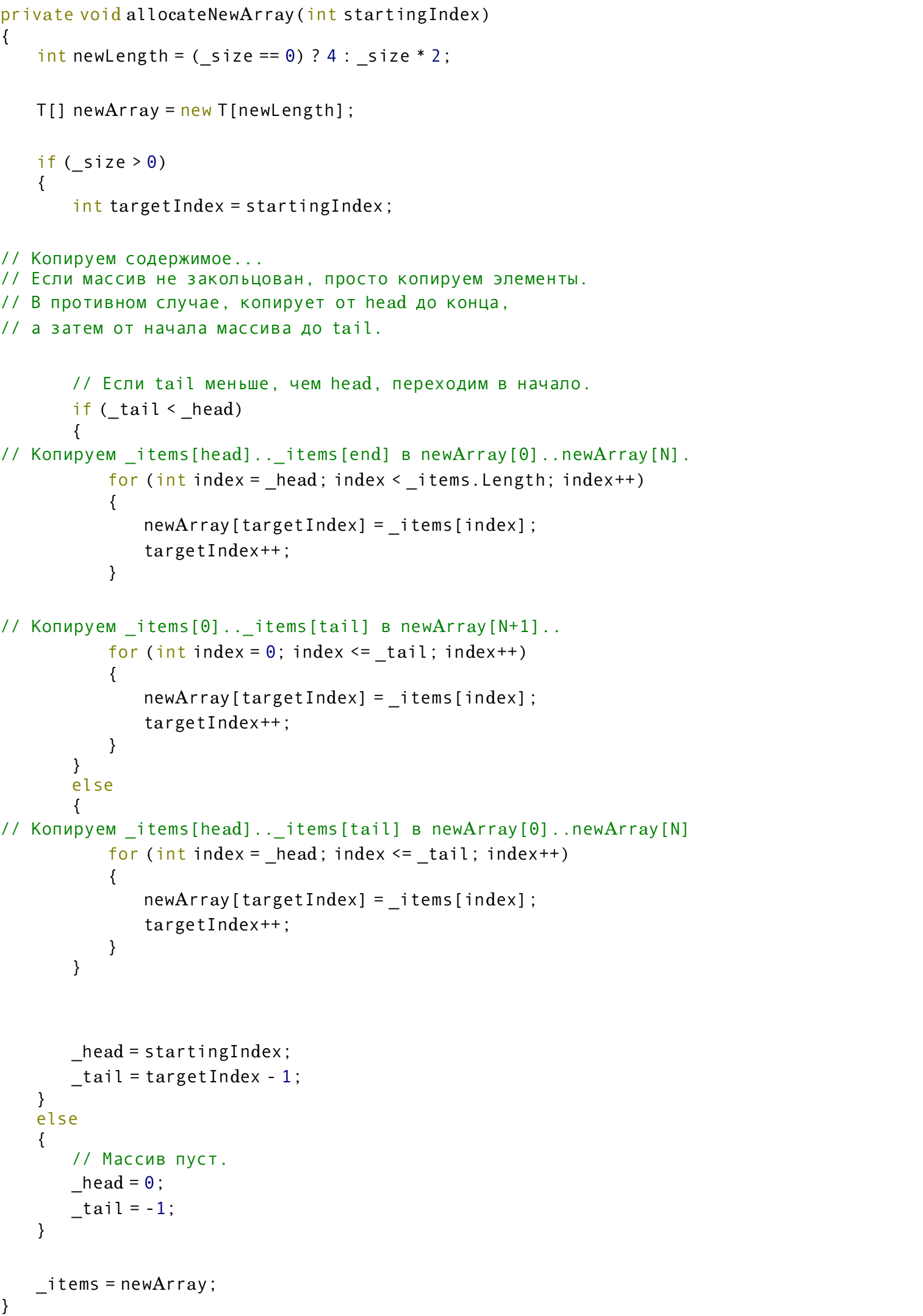
deq.DequeueFirst();

deq.DequeueLast();

Ключевой момент: вне зависимости от вместимости или заполненности внутреннего массива, логически, содержимое очереди — элементы от «головы» до «хвоста» с учетом «закольцованности». Такое поведение также называется «кольцевым буфером».

#### Алгоритм роста

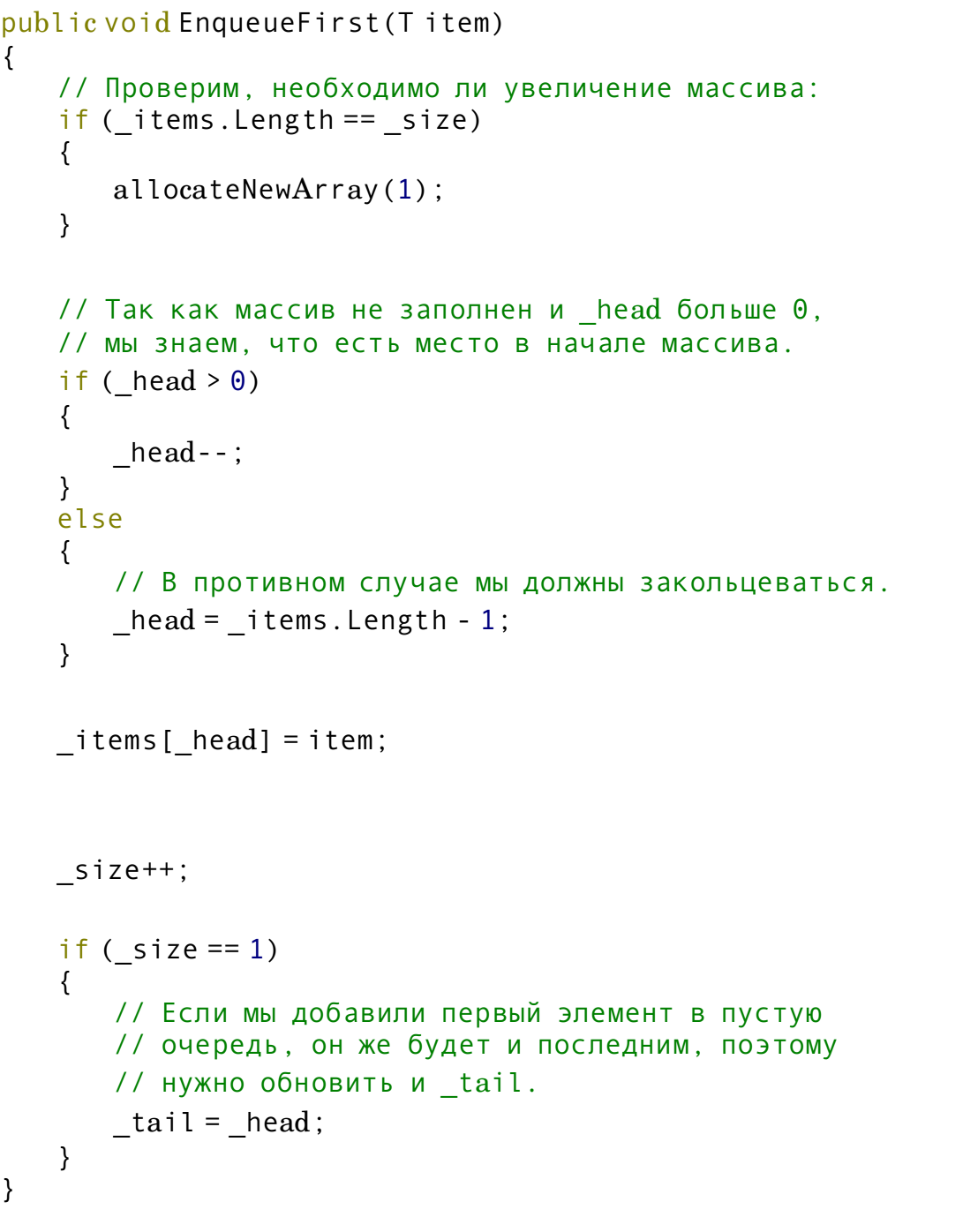
Когда свободное место во внутреннем массиве заканчивается, его необходимо увеличить, скопировать элементы и обновить указатели на «хвост» и «голову». Эта операция производится при необходимости во время добавления элемента. Параметр startingIndex используется, чтобы показать, сколько полей в начале необходимо оставить пустыми (в случае добавления в начало).



#### Метод EnqueueFirst

● Поведение: Добавляет элемент в начало очереди. Этот элемент будет взят из очереди следующим при вызове метода DequeueFirst.

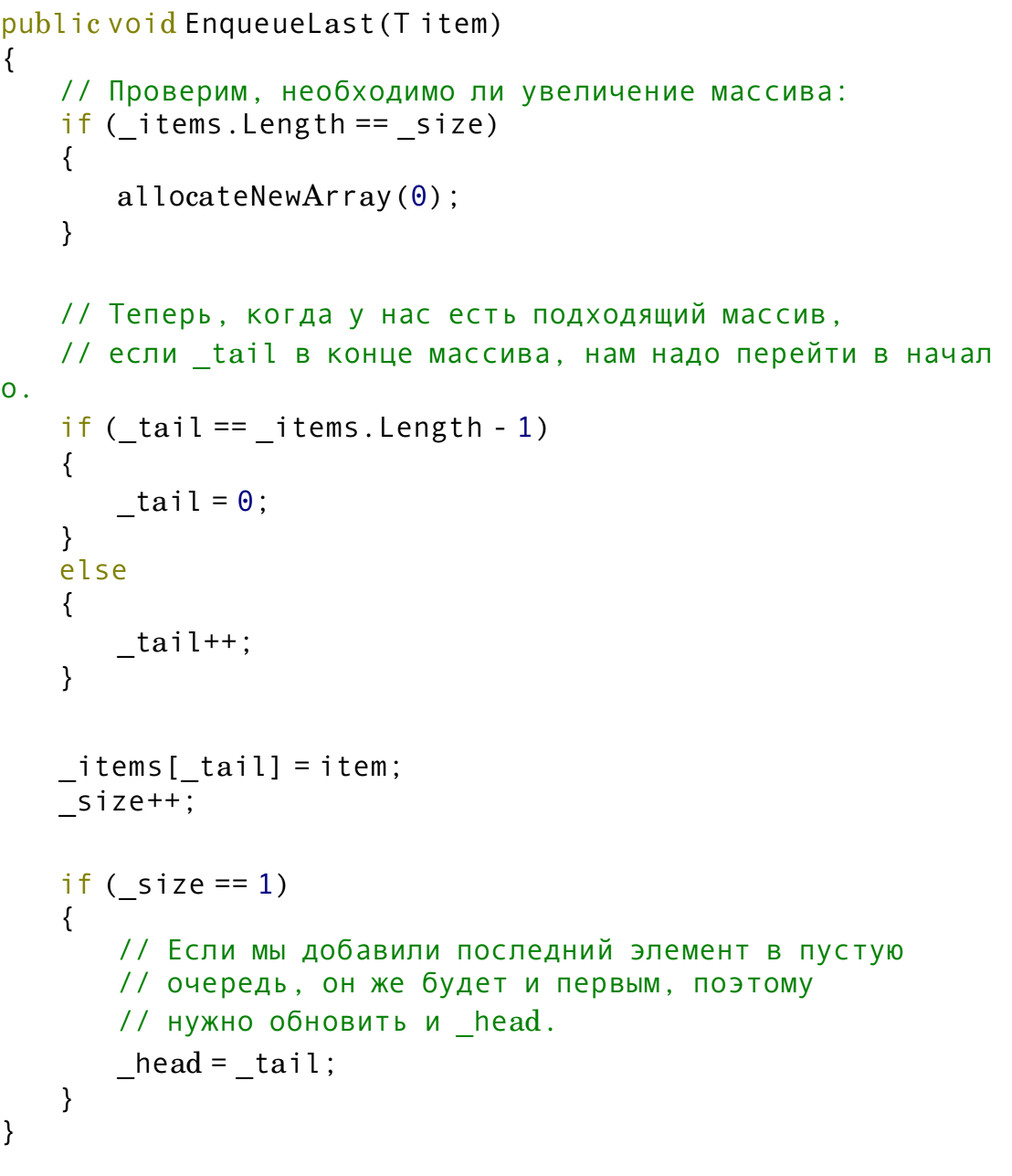
● Сложность: O(1) в большинстве случаев; O(n), когда нужно расширение массива.



#### Метод EnqueueLast

● Поведение: Добавляет элемент в конец очереди. Этот элемент будет взят из очереди следующим при вызове метода DequeueLast.

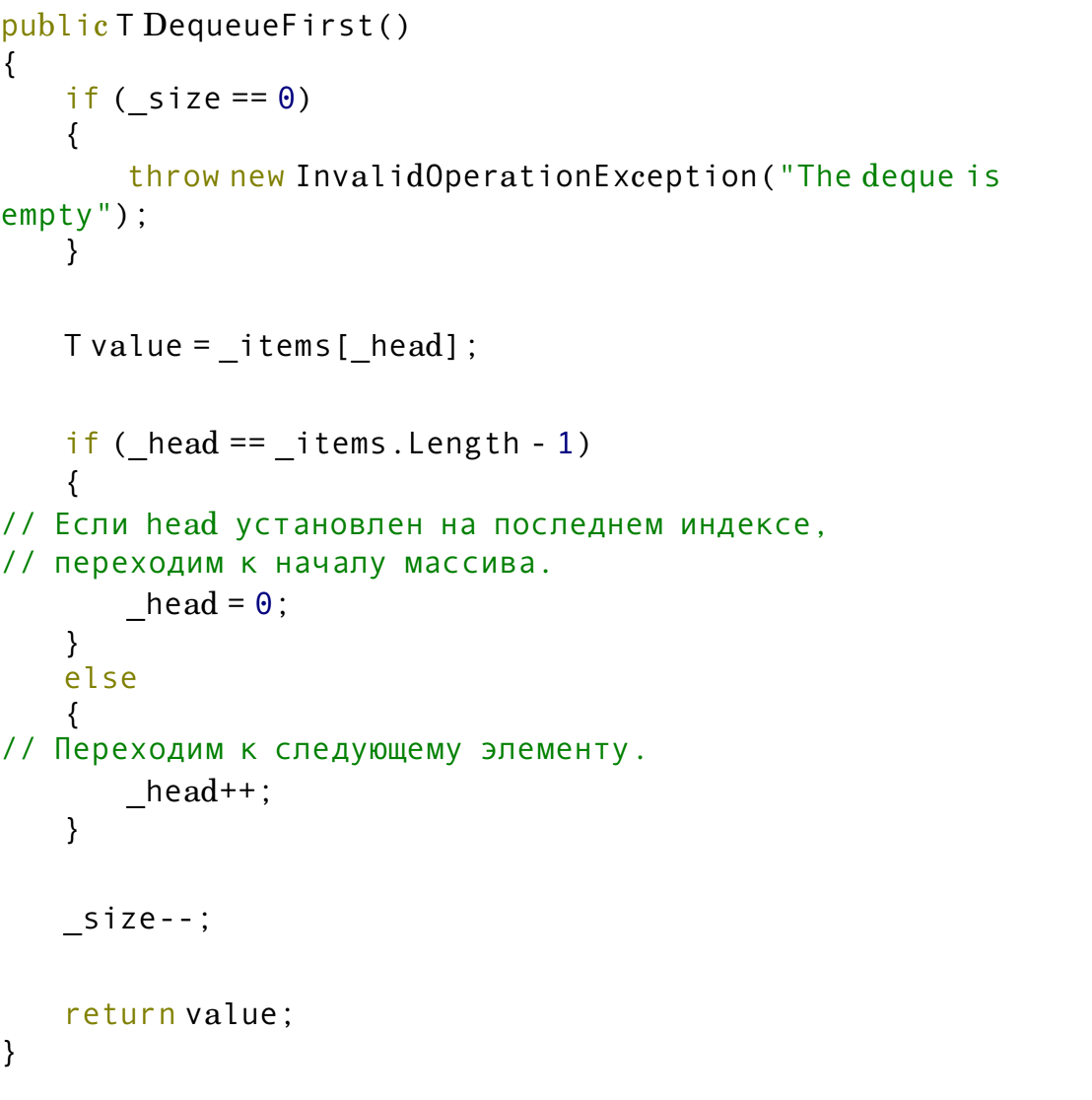
● Сложность: O(1) в большинстве случаев; O(n), когда нужно расширение массива.



#### Метод DequeueFirst

● Поведение: Удаляет элемент с начала очереди и возвращает его. Если очередь пустая, кидает InvalidOperationException.

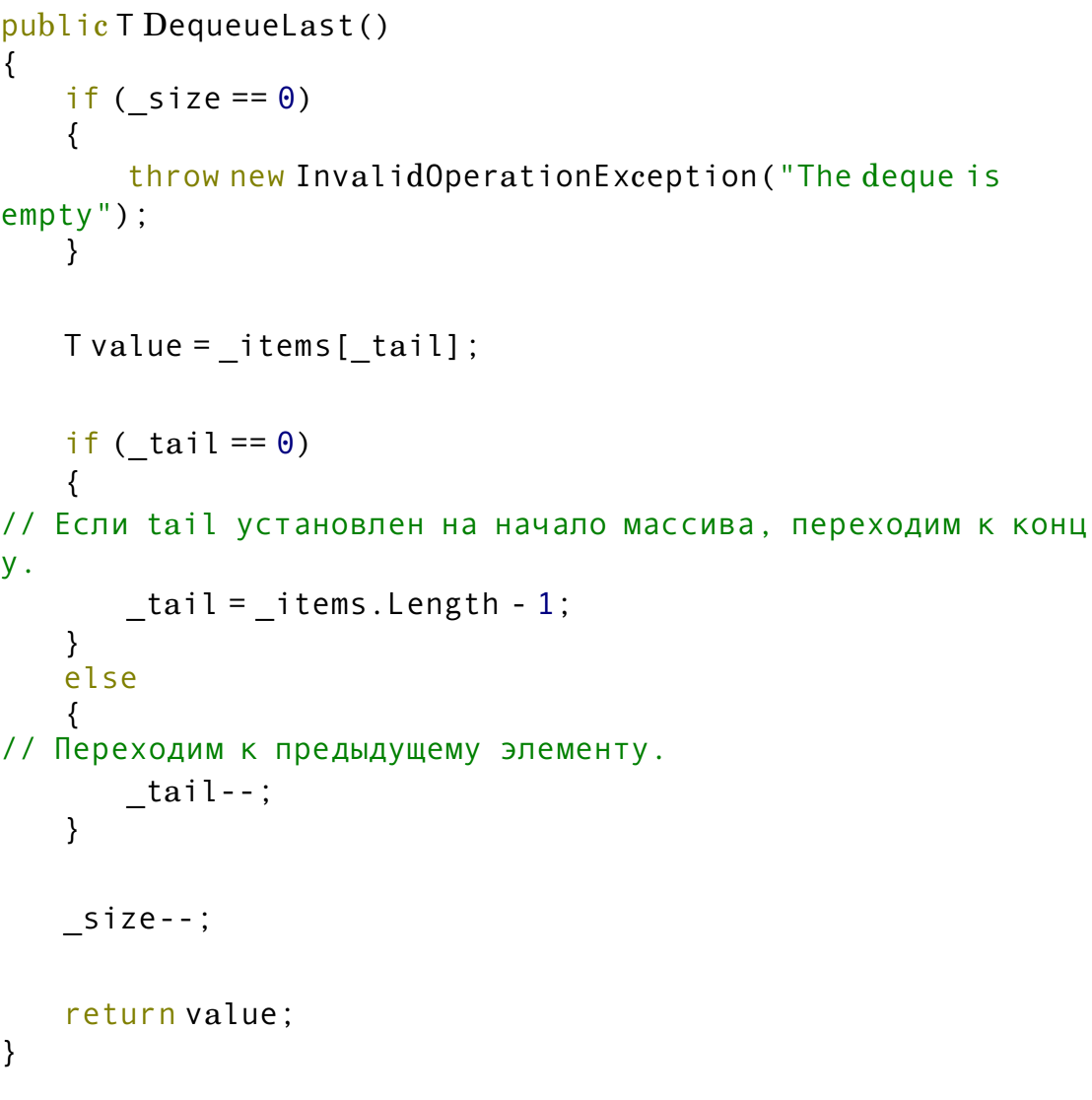
● Сложность: O(1).



#### Метод DequeueLast

● Поведение: Удаляет элемент с конца очереди и возвращает его. Если очередь пустая, кидает InvalidOperationException.

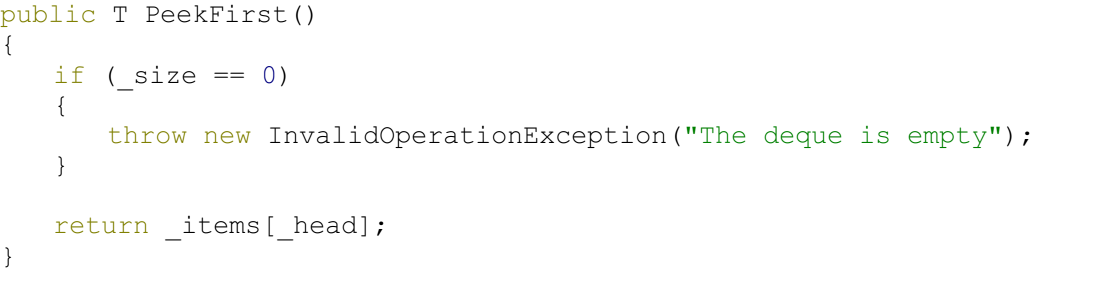
● Сложность: O(1).



#### Метод PeekFirst

● Поведение: Возвращает элемент с начала очереди, не изменяя ее. Если очередь пустая, кидает InvalidOperationException.

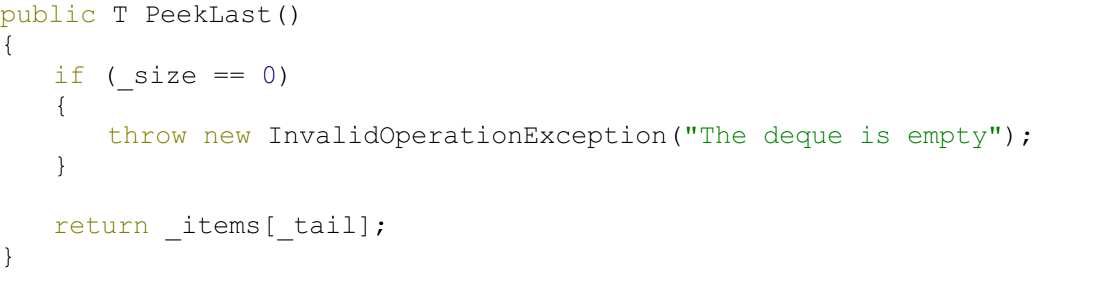
● Сложность: O(1).



#### Метод PeekLast

● Поведение: Возвращает элемент с конца очереди, не изменяя ее. Если очередь пустая, кидает InvalidOperationException.

● Сложность: O(1).



## Графы

### Основные понятия

● Граф – совокупность точек, соединенных линиями. Точки называются вершинами, или узлами, а линии – ребрами, или дугами.

● Степень входа вершины – количество входящих в нее ребер, степень выхода – количество исходящих ребер.

● Встречаются такие графы, ребрам которых поставлено в соответствие конкретное числовое значение, они называются взвешенными графами, а это значение – весом ребра.

● Когда у ребра оба конца совпадают, т.е. оно выходит из вершины и входит в нее, то такое ребро называется петлей.

● Граф, содержащий ребра между всеми парами вершин, является полным.

● Путь — последовательность рёбер, соединяющая разные (неповторяющиеся) вершины;

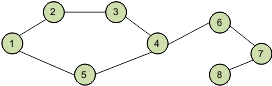
● Маршруты — это те же пути, только они не требуют последовательности разных вершин;

● Цикл — группа вершин, связанных вместе в замкнутую цепь.

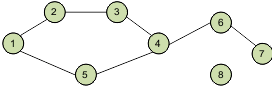
● Связный граф — граф, в котором между любой парой вершин имеется один путь;

● Дерево — связный граф, не содержащий цикла;

### Классификация графов

**Графы делятся на**

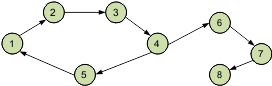
* *связные*

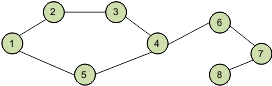


* *несвязные*

В связном графе между любой парой вершин существует как минимум один путь.  
  
В несвязном графе существует хотя бы одна вершина, не связанная с другими.

**Графы также подразделяются на**

* *ориентированные*

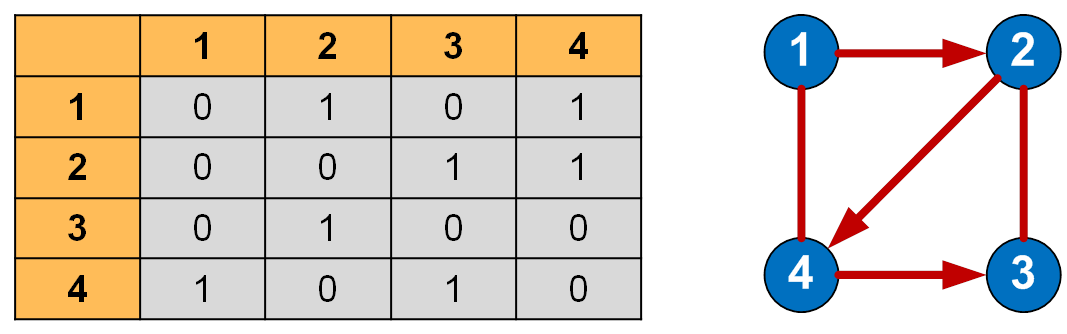


* *неориентированные*
* *смешанные*

В ориентированном графе ребра являются направленными, т.е. существует только одно доступное направление между двумя связными вершинами.  
  
В неориентированном графе по каждому из ребер можно осуществлять переход в обоих направлениях.  
  
Частный случай двух этих видов – смешанный граф. Он характерен наличием как ориентированных, так и неориентированных ребер.

### Способы представления графов

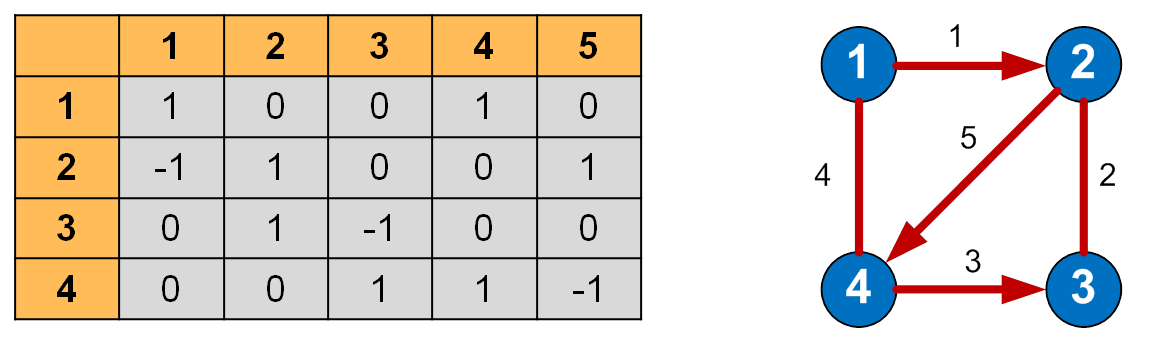
#### 1. Матричные структуры данных

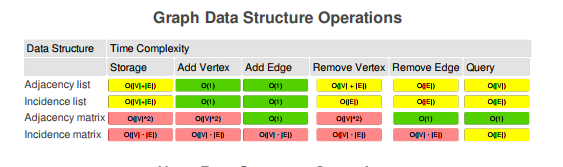
1.1 **Матрица смежности**. Это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1. Число строк матрицы смежности равно числу столбцов и соответствует количеству вершин графа.

0 – соответствует отсутствию ребра, 1 – соответствует наличию ребра.

Когда из одной вершины в другую проход свободен (имеется ребро), в ячейку заносится 1, иначе – 0. Все элементы на главной диагонали равны 0 если граф не имеет петель.

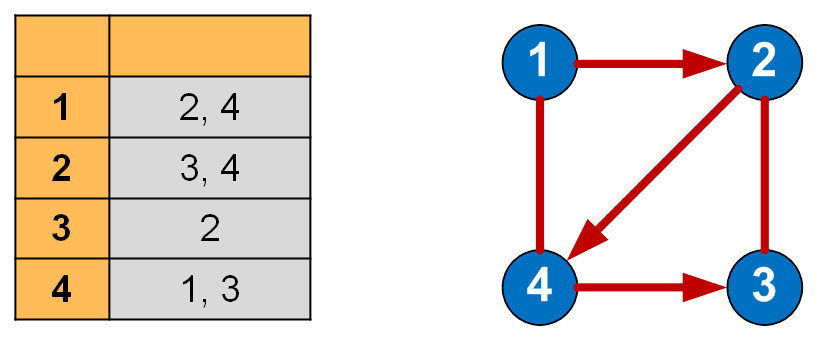
1.2 **Матрица инцидентности**. матрица, количество строк в которой соответствует числу вершин, а количество столбцов – числу рёбер. В ней указываются связи между инцидентными элементами графа (ребро(дуга) и вершина).  
В неориентированном графе если вершина инцидентна ребру то соответствующий элемент равен 1, в противном случае элемент равен 0.  
В ориентированном графе если ребро выходит из вершины, то соответствующий элемент равен 1, если ребро входит в вершину, то соответствующий элемент равен -1, если ребро отсутствует, то элемент равен 0.  
Матрица инцидентности для своего представления требует нумерации рёбер, что не всегда удобно.





#### 2. Перечислительные структуры данных

2.1 **Список смежности**. Если количество ребер графа по сравнению с количеством вершин невелико, то значения большинства элементов матрицы смежности будут равны 0. При этом использование данного метода нецелесообразно. Для подобных графов имеются более оптимальные способы их представления.  
По отношению к памяти списки смежности менее требовательны, чем матрицы смежности. Такой список можно представить в виде таблицы, столбцов в которой – 2, а строк — не больше, чем вершин в графе.  
В каждой строке в первом столбце указана вершина выхода, а во втором столбце – список вершин, в которые входят ребра из текущей вершины.

**Преимущества списка смежности:**

● Рациональное использование памяти.

● Позволяет быстро перебирать соседей вершины.

● Позволяет проверять наличие ребра и удалять его.

**Недостатки списка смежности:**

● При работе с насыщенными графами (с большим количеством рёбер) скорости может не хватать.

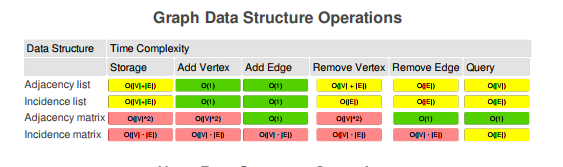
● Нет быстрого способа проверить, существует ли ребро между двумя вершинами.

● Количество вершин графа должно быть известно заранее.

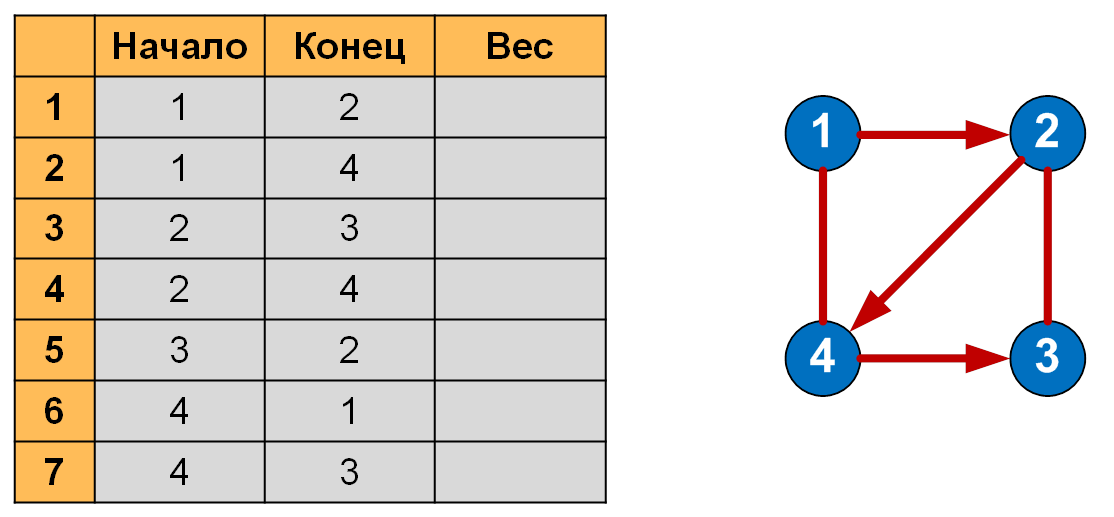
● Для взвешенных графов приходится хранить список, элементы которого должны содержать два значащих поля, что усложняет код:

● номер вершины, с которой соединяется текущая;

● вес ребра.



2.2 **Список ребер**. В списке рёбер в каждой строке записываются две смежные вершины и вес соединяющего их ребра (для взвешенного графа). Количество строк в списке ребер всегда должно быть равно величине, получающейся в результате сложения ориентированных рёбер с удвоенным количеством неориентированных рёбер.



### Алгоритмы обхода графов

#### [Поиск в ширину](https://prog-cpp.ru/data-graph/" \l "width)

Поиск в ширину подразумевает поуровневое исследование графа:

1. В начале посещается корень – произвольно выбранный узел,
2. Затем – все потомки данного узла,
3. После этого посещаются потомки потомков и т.д.

Вершины просматриваются в порядке возрастания их расстояния от корня. Алгоритм прекращает свою работу после обхода всех вершин графа, либо в случае выполнения требуемого условия (например, найти кратчайший путь из вершины 1 в вершину 6).

Применения алгоритма поиска в ширину :

1. Поиск кратчайшего пути в невзвешенном графе (ориентированном или неориентированном).
2. Поиск компонент связности.
3. Нахождения решения какой-либо задачи (игры) с наименьшим числом ходов.
4. Найти все рёбра, лежащие на каком-либо кратчайшем пути между заданной парой вершин.
5. Найти все вершины, лежащие на каком-либо кратчайшем пути между заданной парой вершин.
6. Алгоритм поиска в ширину работает как на ориентированных, так и на неориентированных графах.

**Реализация на C++ (с использованием очереди STL)**



Результат выполнения



**Задача поиска кратчайшего пути. Реализация на С++** 



#### [Поиск в г](https://prog-cpp.ru/data-graph/" \l "width)лубину

Поиск в ширину производится симметрично (вершины графа просматривались по уровням). Поиск в глубину предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность продвижения означает, что следующим шагом будет переход на последний, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершина).  
  
Отсутствие последнего свидетельствует об одной из двух возможных ситуаций:

● Все вершины графа уже просмотрены,

● Просмотрены вершины доступные из вершины, взятой в качестве начальной, но не все (несвязные и ориентированные графы допускают последний вариант).

Применения алгоритма поиска в глубину:

● Поиск любого пути в графе.

● Поиск лексикографически первого пути в графе.

● Проверка, является ли одна вершина дерева предком другой.

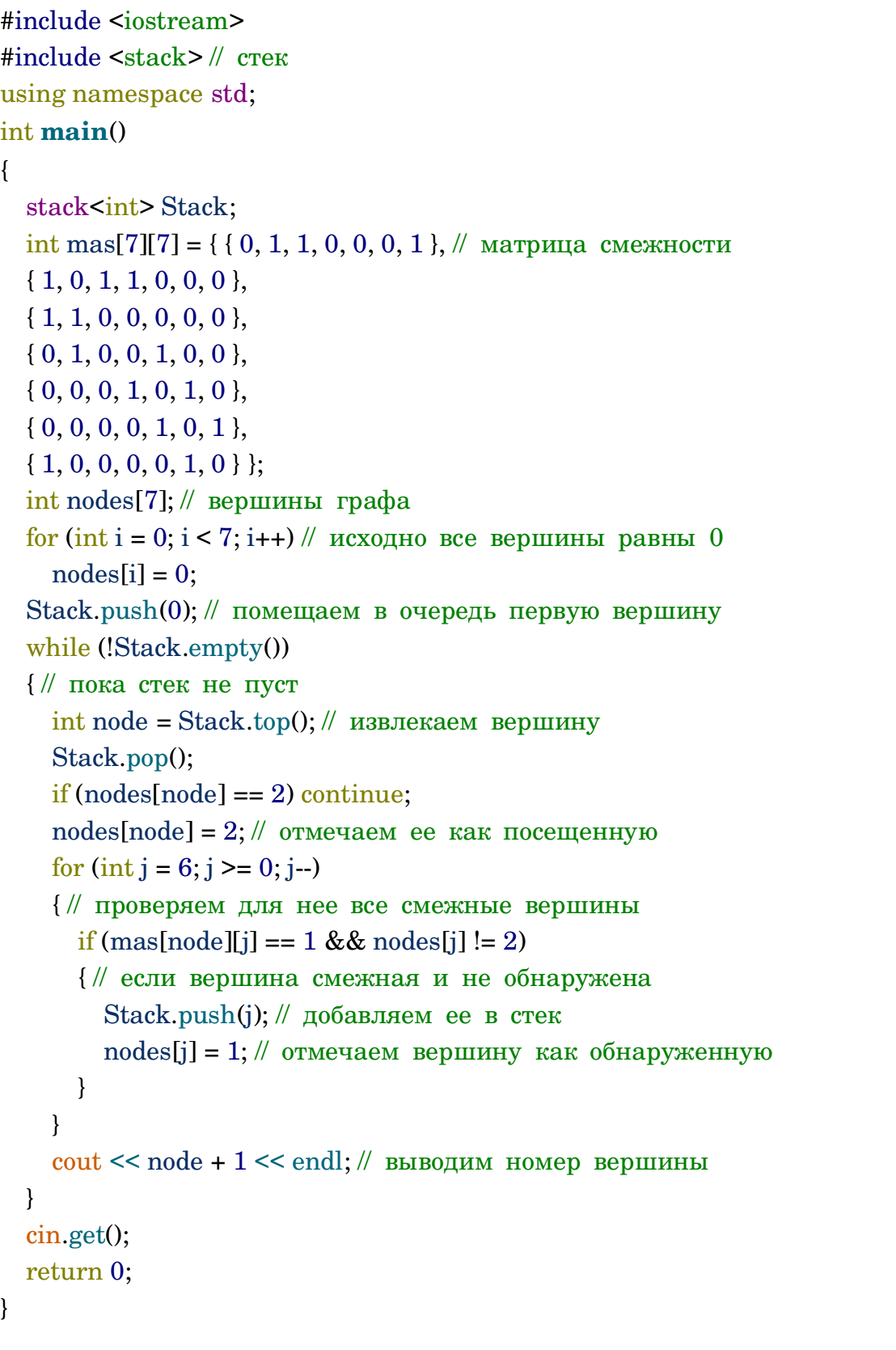
● Поиск наименьшего общего предка.

● Топологическая сортировка.

● Поиск компонент связности.

Алгоритм поиска в глубину работает как на ориентированных, так и на неориентированных графах. Применимость алгоритма зависит от конкретной задачи.  
Для реализации алгоритма удобно использовать стек или рекурсию.

**Реализация на C++ (с использованием стека STL)**

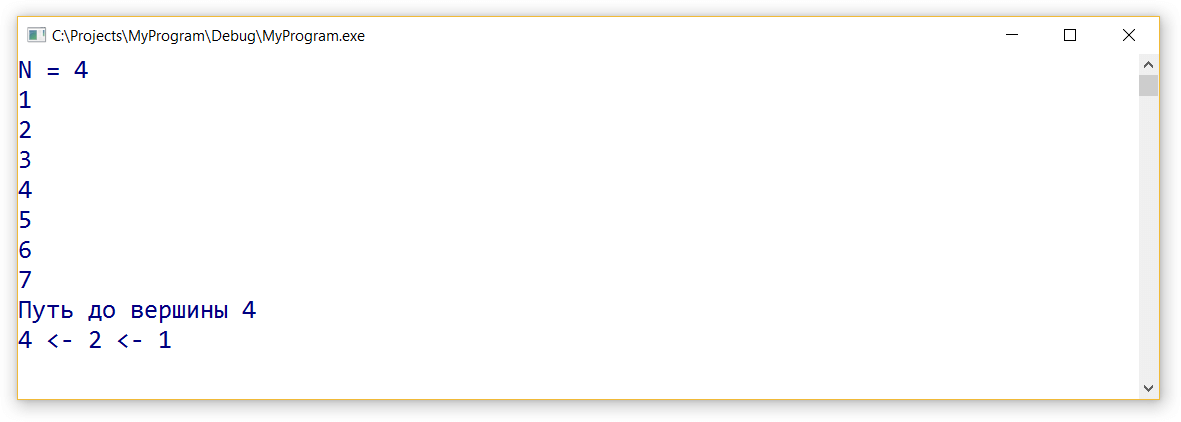
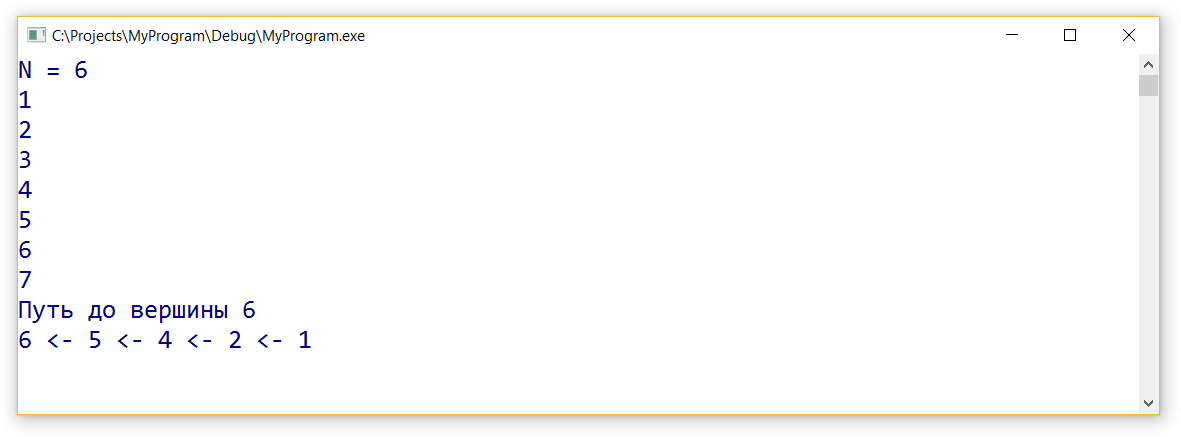


Результат выполнения



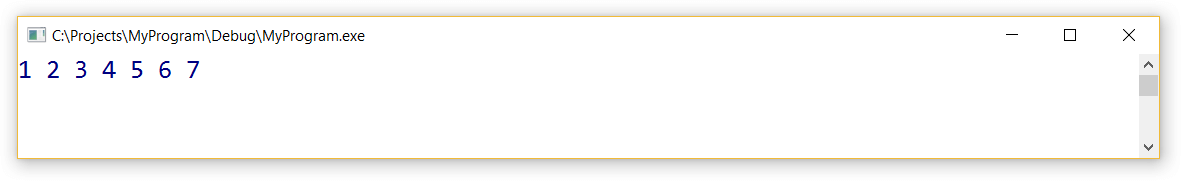
**Задача поиска лексикографически первого пути на графе. Реализация на C++**





**Поиск в глубину также может быть реализован с использованием рекурсивного алгоритма.**





### Деревья

#### Основные понятия

● Корневой узел — самый верхний узел дерева.

● Корень — одна из вершин, по желанию наблюдателя.

● Лист, листовой или терминальный узел — узел, не имеющий дочерних элементов.

● Внутренний узел — любой узел дерева, имеющий потомков, и таким образом, не являющийся листовым узлом.

● Дерево считается ориентированным, если в корень не заходит ни одно ребро.

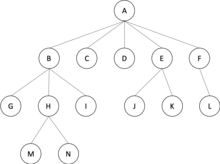
● Полный сцепленный ключ — идентификатор записи, который образуется путём конкатенации всех ключей экземпляров родительских записей (групп).

● M-арное дерево - число поддеревьев данного узла образует степень узла, максимальное значение m степени всех узлов дерева является степенью дерева. Дерево степени 2 называется бинарным деревом. Если в дереве на каждом уровне задан порядок следования вершин, то такое дерево называется упорядоченным

#### К -арное дерево

Сложность всех операций и памяти - O(n)

В теории графов , m -арное дерево (также известное как k -ary или k -way tree) - это корневое дерево , в котором каждый узел имеет не более m детей. Бинарное дерево - это особый случай, когда m = 2, а троичное дерево - это еще один случай с m = 3, который ограничивает количество его дочерних элементов тремя.

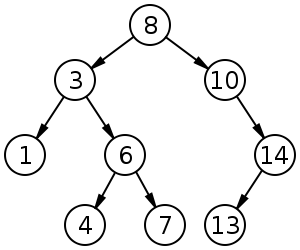


#### Бинарное дерево поиска

Поиск - O(log n); Вставка - O(log n); Удаление - O(log n);

Это двоичное дерево, для которого выполняются следующие условия:

1. У каждого узла не более двух детей(так как бинарное).
2. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
3. У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равно, нежели значение ключа данных самого узла X.



#### Двоичная Куча

Двоичная куча, пирамида, или сортирующее дерево — такое [двоичное дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE" \o "Двоичное дерево), для которого выполнены три условия:

1. Значение в любой вершине не меньше, чем значения её потомков.
2. Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой.
3. Последний слой заполняется слева направо без «дырок».

Над кучей можно выполнять следующие операции:

● Добавить элемент в кучу. Сложность O ( logn )

● Исключить максимальный элемент из кучи. Время работы O ( logn )

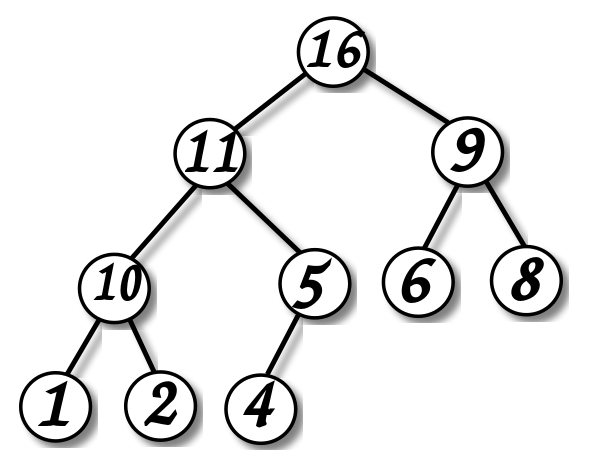
● Изменить значение любого элемента. Время работы O ( logn )

На основе этих операций можно выполнять следующие действия:

● Превратить неупорядоченный массив элементов в кучу. Сложность O ( n )

● Отсортировать массив путём превращения его в кучу, а кучу в отсортированный массив. Время работы O ( n logn )

Здесь nIMG_261 — количество элементов кучи. [Пространственная сложность](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C" \o "Пространственная сложность) — O ( 1 ) для всех вышеперечисленных операций и действий.



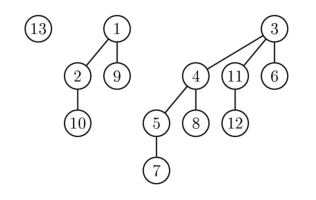
#### Биноминальная Куча

Биномиальная куча — [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Структура данных), реализующая [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Абстрактный тип данных) «[очередь с приоритетом](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_%D1%81_%D0%BF%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BE%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)" \o "Очередь с приоритетом (программирование))», которая представляет собой набор [биномиальных деревьев](https://ru.wikipedia.org/w/index.php?title=%D0%91%D0%B8%D0%BD%D0%BE%D0%BC%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE&action=edit&redlink=1" \o "Биномиальное дерево (страница отсутствует)) с двумя свойствами:

1. Ключ каждой вершины не меньше ключа её родителя;
2. Все биномиальные деревья имеют разный размер.

Из этих свойств вытекают два следствия. Во-первых, корень каждого из деревьев имеет наименьший ключ среди его вершин. Во-вторых, суммарное количество вершин в биномиальной куче однозначно определяет размеры входящих в него деревьев. Например, биномиальная куча с 13 = 2^3 + 2^2 + 2^0 вершинами состоит из трёх деревьев высотой 3, 2 и 0 и имеющих, соответственно, 8, 4 и 1 элементов (см. рис.)

Следующие операции выполняются за время O ( logn ), где n — число вершин:

● Вставка нового элемента (амортизированное O ( 1 ))

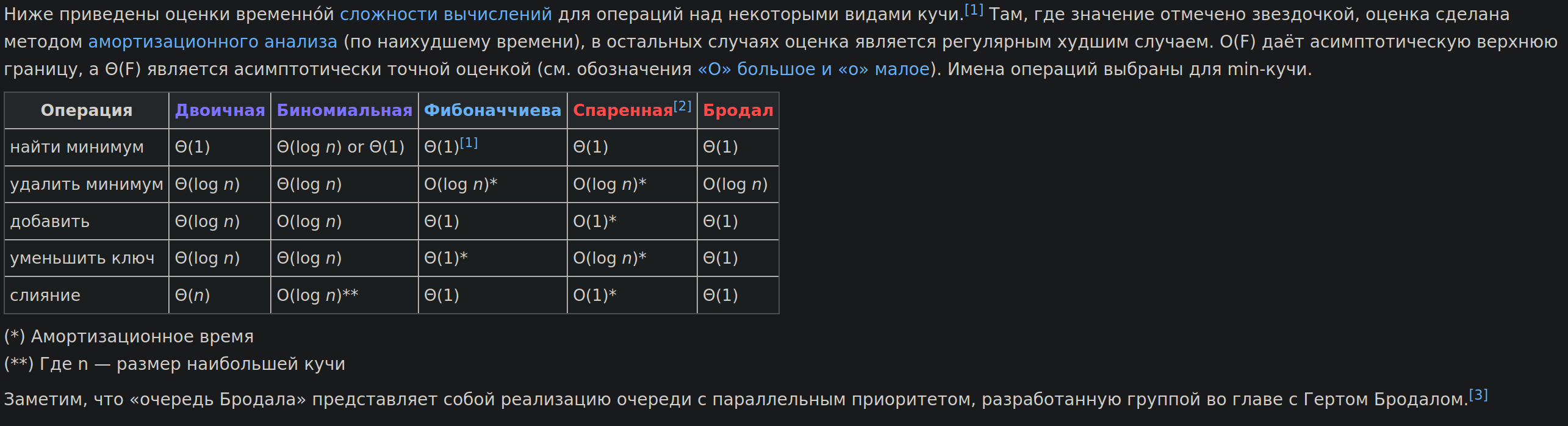
● Нахождение элемента с минимальным ключом

● Удаление элемента с минимальным ключом

● Уменьшение значения ключа данного элемента

● Удаление данного элемента

● Объединение двух куч.



#### Очередь с приоритетом

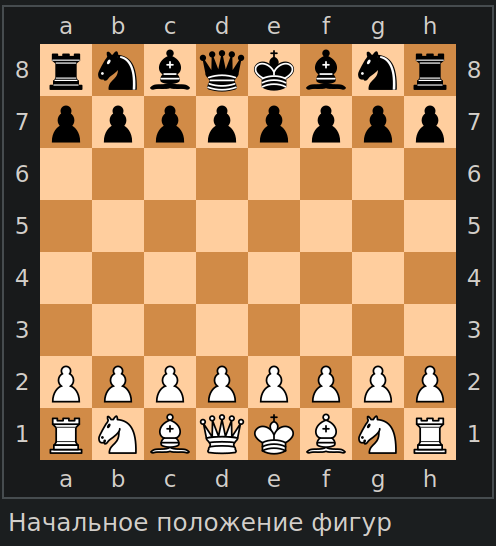
Сложность: O(log N).

Очередь с приоритетом — [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Абстрактный тип данных) в [программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5" \o "Программирование), поддерживающий две обязательные операции — добавить элемент и извлечь максимум (минимум). Предполагается, что для каждого элемента можно вычислить его приоритет — действительное число или в общем случае элемент [линейно упорядоченного множества](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D0%BE_%D1%83%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D1%87%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE" \o "Линейно упорядоченное множество).

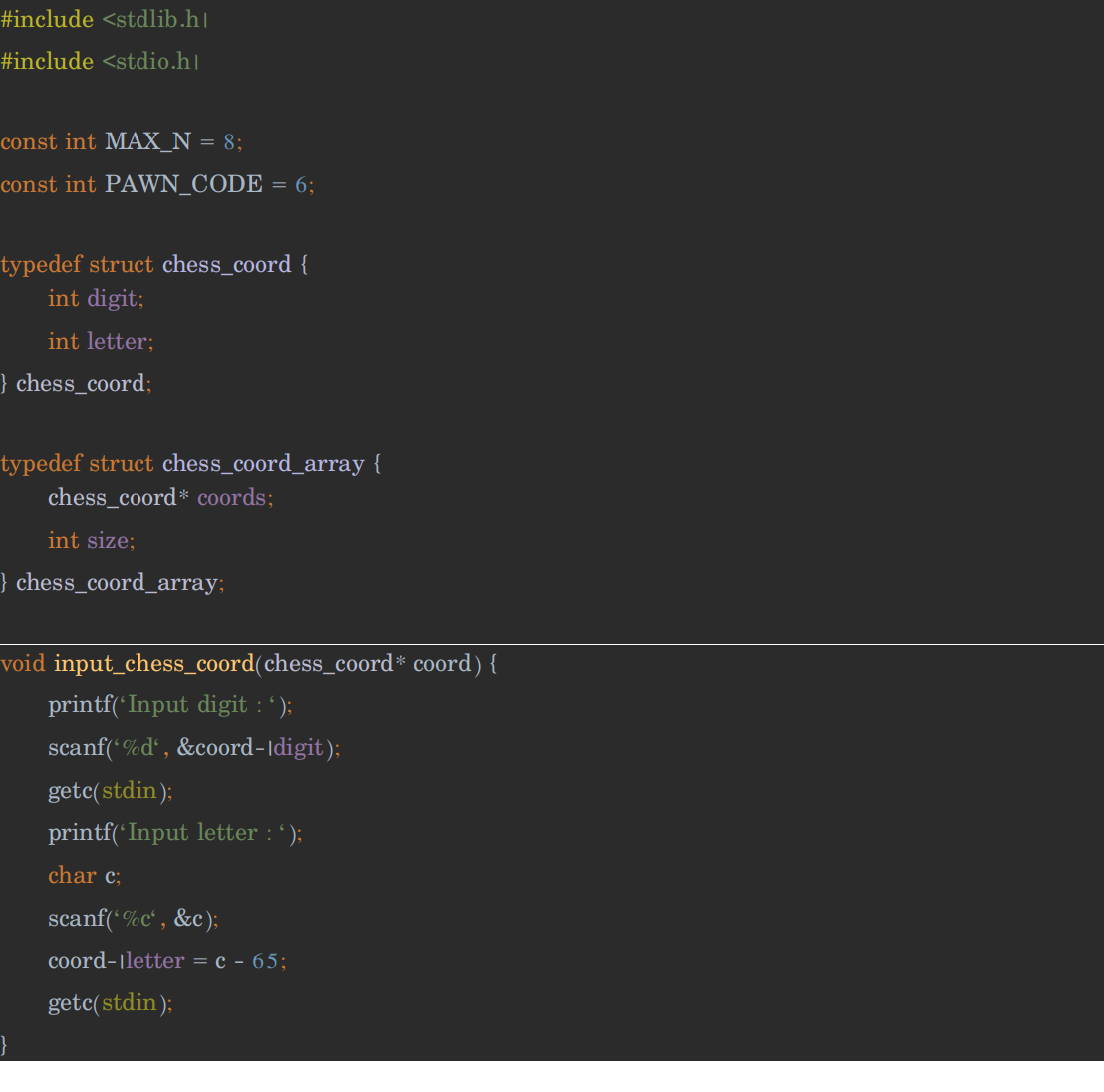
## Полезные ссылки

1. [Связные списки](https://tproger.ru/translations/linked-list-for-beginners/)
2. [Стеки и очереди](https://tproger.ru/translations/stacks-and-queues-for-beginners/)
3. [Способы представления графов](https://habr.com/ru/post/469967/)
4. [Способы представления графов + теория](https://prog-cpp.ru/data-graph/)

# Шахматные алгоритмы



## Вспомогательные функции и структуры





## Король

### Все возможные ходы

### Поиск пути из т. a в т. b

### Встреча двух фигур

## Ферзь

### Все возможные ходы

### Поиск пути из т. a в т. b

### Встреча двух фигур

## Ладья

### Все возможные ходы

### Поиск пути из т. a в т. b

### Встреча двух фигур

## Слон

### Все возможные ходы

### Поиск пути из т. a в т. b

### Встреча двух фигур

## Конь

### Все возможные ходы

1. x - 2; y + 1;
2. x - 1; y + 2;
3. x + 2; y + 1;
4. x + 1; y + 2;
5. x - 2; y - 1;
6. x - 1; y - 2;
7. x + 2; y - 1;
8. x + 1; y - 2;

!Проверка выхода за пределы шахматной доски

### Поиск пути из т. a в т. b

Прогон всех возможных ходов в for’е с коэфициентом ((i + 1) \* 2) вместо двойки и (i + 1) место единицы

!Проверка выхода за пределы шахматной доски

### Встреча двух фигур

1. Построение матрицы посещений для каждого коня

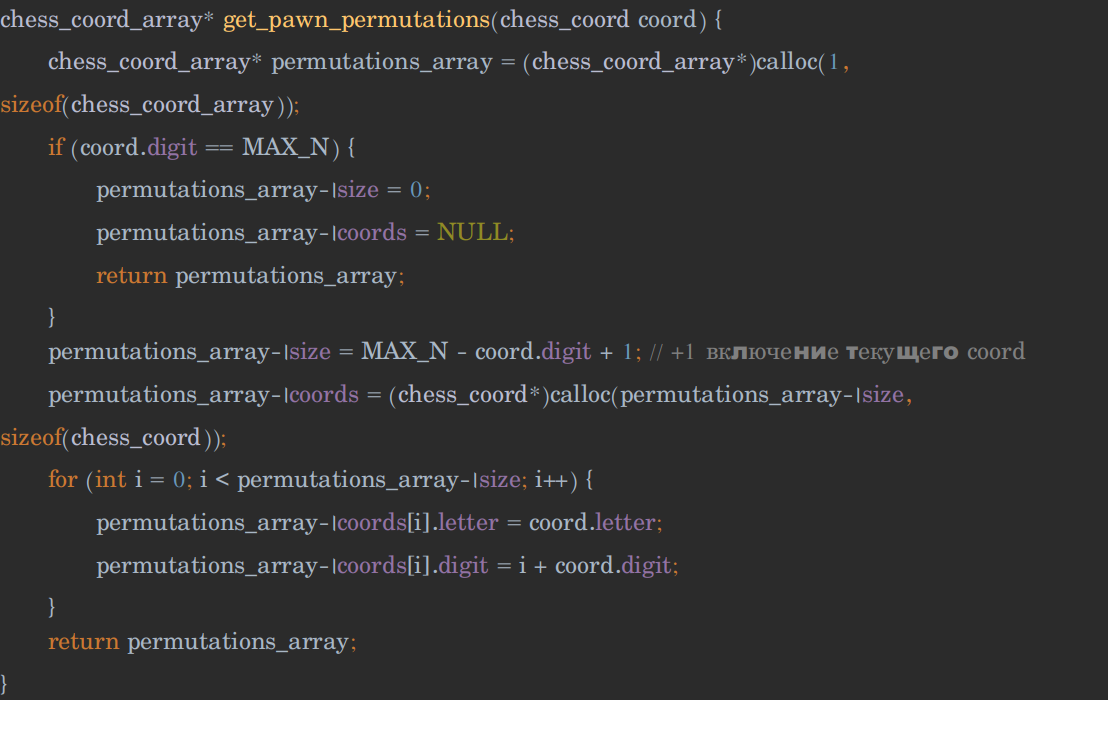
Матрица размера 8х8 в каждой ячейке которой лежит массив точек, которые обозначают путь к данной точке (последняя точка массива - данная точка), называется матрицей посещений

1. Рекурсивный обход всех ходов каждого коня (с проверкой выхода за границы и повторного посещения точки) и формирование матриц посещений
2. Сравнение двух матриц : если путь к одной и той же точке занял равное кол-во ходов (+-1 в силу условности), то помещаем данную точку в массив встречь

## Пешка

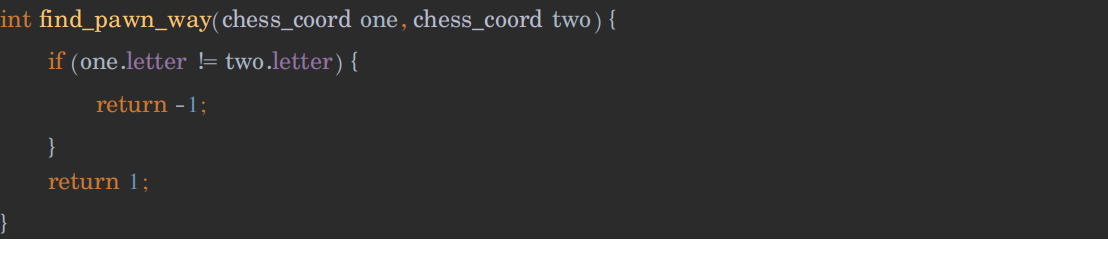
### Все возможные ходы

Все возможные расстановки - все позиции выше введенной



### Поиск пути из т. a в т. b

Если точка 1 выше точки два и находится на той же букве, то путь существует



### Встреча двух фигур

Две фигуры встретятся если они стоят на одной букве



## (Бонус) Шашка

### Все возможные расстановки

### Поиск пути из т. a в т. b

### Встреча двух фигур

### Путь в дамки

## Полезные ссылки