**Очередь** — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, англ. first-in, first-out). Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть, словом, dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.

**Очередь** (англ. queue) — это структура данных, добавление и удаление элементов в которой происходит путём операций enqueue и dequeue соответственно. Притом первым из очереди удаляется элемент, который был помещен туда первым, то есть в очереди реализуется принцип «первым вошел — первым вышел» (англ. first-in, first-out — FIFO).

У очереди имеется голова (англ. head) и хвост (англ. tail). Когда элемент ставится в очередь, он занимает место в её хвосте. Из очереди всегда выводится элемент, который находится в ее голове.

Очередь поддерживает следующие **операции**:

* empty — проверка очереди на наличие в ней элементов,
* enqueue (запись в очередь) — операция вставки нового элемента,
* dequeue (снятие с очереди) — операция удаления нового элемента,
* peek – возвращает первый элемент, не удаляя его,
* rear – возвращает последний элемент, не удаляя его,
* size — операция получения количества элементов в очереди.

**Реализация**: Очереди могут быть реализованы с помощью массивов или связанных списков. При реализации на основе массивов используется массив фиксированного размера, и дополнительные элементы не могут быть добавлены после заполнения массива. Реализация на основе связных списков позволяет динамически изменять размер и поддерживает добавление элементов без ограничений.

**Циркулярная очередь**: Циркулярная очередь — это расширение обычной очереди, в которой последний элемент соединен с первым элементом, образуя круговую структуру. Это позволяет эффективно использовать память в реализации на основе массивов.

**Очередь с приоритетом**: В то время как обычная очередь следует принципу FIFO, приоритетная очередь присваивает значение приоритета каждому элементу и извлекает элементы на основе их приоритета. Элементы с более высоким приоритетом удаляются из очереди первыми.

**Очереди используются** в различных приложениях, включая:

* Планирование заданий: Управление заданиями или задачами в операционной системе.
* Очередь печати: Очередь запросов на печать.
* Веб-серверы: Обработка входящих запросов.
* Алгоритм BFS (Breadth-First Search): Поиск узлов в графе уровень за уровнем.
* Имитация и моделирование: Имитация реальных сценариев, в которых задействованы очереди.

**Сложность:**

* **Enqueue**: Амортизированная временная сложность O(1). Добавление элемента в конец списка обычно является операцией O(1), но иногда, когда базовый список должен быть изменен в размере, это может быть O(n) из-за операции копирования.
* **Dequeue**:

Временная сложность O(1). Вытаскивание элемента из передней части списка занимает постоянное время, поскольку не требует сдвига оставшихся элементов.

Если индекс удаляемого элемента известен, удаление его с помощью среза имеет временную сложность O(n), где n - количество элементов в массиве. Эта операция предполагает создание нового массива с нужными элементами, исключая удаляемый элемент. Это требует копирования элементов до и после удаляемого элемента в новый массив, что приводит к линейной временной операции.

* **Peek**, **Rear**: временная сложность O(1). Доступ к элементу, находящемуся в начале, конце списка, непосредственно с помощью индексации является операцией с постоянным временем.
* **Is Empty**: O(1) временная сложность. Проверка длины списка для определения его пустоты является операцией с постоянным временем.
* **Size**: O(1) временная сложность. Возврат длины списка является постоянной операцией по времени.

**Вариации добавления элементов в очередь:**

append() добавляет элемент в конец списка, сохраняя порядок существующих элементов, а insert() вставляет элемент по определенному индексу, сдвигая элементы, чтобы освободить место для нового элемента.

* Метод append():

**Синтаксис:** list.append(item)

**Функциональность:** добавляет заданный элемент в конец списка, увеличивая его длину на единицу. Исходные элементы остаются на своих местах, а новый элемент становится последним элементом списка.

**Временная сложность:** Амортизированная временная сложность O(1), поскольку обычно добавление элемента в конец списка занимает постоянное время. Однако иногда, когда необходимо изменить размер основного списка, это может быть O(n) из-за операции копирования.

* Метод insert():

**Синтаксис:** list.insert(index, item)

**Функциональность:** вставляет заданный элемент по указанному индексу в список. Все элементы от указанного индекса и далее сдвигаются вправо, увеличивая длину списка на единицу. Существующие элементы могут изменить свое положение в зависимости от индекса вставки.

**Временная сложность:** O(n), где n - длина списка. Поскольку все элементы от указанного индекса и далее должны быть сдвинуты вправо, временная сложность пропорциональна количеству перемещаемых элементов.

**Метод pop()** без аргументов (т.е. list.pop()) имеет временную сложность O(1). Он удаляет и возвращает последний элемент из списка за постоянное время. Поскольку список хранит свои элементы в непрерывном блоке памяти, удаление последнего элемента включает обновление внутренней длины списка и возврат значения в соответствующее место памяти.

Метод pop(index) имеет временную сложность O(n), где n - количество элементов в списке. Он удаляет и возвращает элемент по указанному индексу. Удаление элемента из середины или начала списка требует сдвига последующих элементов для заполнения пробела, что приводит к линейной временной операции.

**Функция len()** в Python имеет временную сложность O(1) для встроенных типов данных, таких как списки, кортежи и строки.

Когда вы вызываете функцию len() для списка, кортежа или строки, Python внутренне хранит информацию о длине этих типов данных. В результате доступ к длине этих типов данных является постоянной по времени операцией, независимо от количества элементов или символов, которые они содержат.

Эта постоянная временная сложность возможна потому, что Python хранит информацию о длине как часть внутренней структуры объекта. Поэтому, когда вы вызываете len(), Python может напрямую получить сохраненное значение длины без итераций по элементам или символам.

Временная сложность **доступа к элементу массива** в Python (или любом другом языке программирования с нулевой индексацией) составляет O(1).

В массиве каждый элемент хранится в определенной позиции индекса. Когда вы обращаетесь к элементу по его индексу, базовый адрес памяти этого элемента вычисляется на основе индекса и размера каждого элемента в массиве. Этот расчет включает в себя простую арифметическую операцию, которая определяет местоположение нужного элемента в памяти.

Поскольку вычисление адреса памяти не зависит от размера массива, доступ к элементу в массиве имеет постоянную временную сложность O(1). Доступ к элементу занимает одинаковое количество времени, независимо от размера массива или положения элемента в массиве.

Другими словами, время, необходимое для доступа к элементу массива, не зависит от длины массива, что делает эту операцию эффективной.