Алгоритмы: сложность, структуры данных | Я.Шпора

Импортируемые структуры данных: массив

Массив — это набор однородных элементов, в числе которых не может быть структур данных.

Модуль `array` в Python даёт возможность создавать массивы, где элементами служат примитивные типов данных — целые числа, вещественные числа и другие. Массив обычно использует память более эффективно, чем список, и предоставляет быстрый доступ к своим элементам.

Массив `array` объявляется так:

```
from array import array

new_array = array('тип_данных_элементов', [элемент_1, элемент_2, элемен
т_3, ...])
```

В классе `array` можно указать один из тринадцати доступных типов данных элементов.

- `b` — целые числа в диапазоне от -128 до 127.
- `f`, `d` — числа с плавающей точкой разной степени точности.
- `u` — символы Unicode: строки. Синтаксис отличается: строка не делится запятыми на элементы и не замыкается в квадратные скобки: `text = array('u', 'Я тоже массив!')`
- Остальные девять типов описывают разные диапазоны целых чисел.

Посмотреть описания доступных в `array` типов элементов можно в документации.

Доступ к элементам массива — по индексам:

```
from array import array

a = array('b', [1, 2, 3, 4, 5])
```

```
print(a[4])
# 5
```

В `array` можно добавлять элементы и удалять их: у `array` есть методы `append()`, `extend()`, `insert()`, `pop()`, `remove()`.

Время доступа к элементам в массивах — постоянное

: `O(1)`. Вставка и удаление имеют сложность `O(n)`.

Связные списки

Связный список — это структура данных, в которой элементы линейно упорядочены, но порядок определяется не номерами элементов (как в массивах), а указателями на следующий элемент списка.

Сложность основных операций в связном списке:

	Время в среднем	Время в худшем случае
Обращение по индексу	`O(n)`	`O(n)`
Добавление элемента на следующую позицию за текущим элементом	`O(1)`	`O(1)`
Добавление элемента на позицию перед текущим элементом	`O(n)`	`O(n)`
Удаление первого элемента	`O(1)`	`O(1)`
Удаление текущего элемента	`O(n)`	`O(n)`
Поиск по значению	`O(n)`	`O(n)`
Определение длины списка	`O(n)`	`O(n)`

Существует «продвинутая» модификация связного списка: **двусвязный список**. Его узлы устроены так же, как в односвязном, но хранят две ссылки: на следующий и на предыдущий элемент.

Стек

Стек — это массив, добавлять или считывать элементы которого можно только «с одной стороны», с **вершины стека**.

Стек основан на принципе LIFO (англ. *last in, first out* — «последний вошёл — первым вышел»). Первым извлекают элемент, который добавлен позже всех.

Официальная документация рекомендует реализовать стек с помощью списка. Методы списка `append()`, `pop()` и `len()` реализуют минимально необходимый набор методов для стека: добавление элемента на вершину стека, получение элемента и определение размера стека. Вершиной стека в этой ситуации будет конец списка.

Набор методов стека:

- `push(item)` — добавляет элемент на вершину стека;
- `pop()` — возвращает элемент с вершины стека, при этом элемент удаляется из стека;
- `size()` — возвращает количество элементов в стеке.

Иногда стек реализует дополнительные операции:

- `peek()` или `top()` — возвращает элемент с вершины стека, но не удаляет его;
- `is_empty()` — определяет, пуст ли стек.

Реализация стека через класс:

```
class Stack:

    def __init__(self):
        # Для хранения элементов в списке используем приватный атрибут.
        # На его приватность указывают два подчёркивания в имени.
        self.__items = []

    def push(self, item):
        """Добавить элемент в стек."""
        self.__items.append(item)

    def pop(self):
        """Извлечь элемент из стека."""
        return self.__items.pop()

    def peek(self):
        """Получить последний элемент без изъятия."""
        return self.__items[-1]
```

```
    def size(self):
        """Вернуть размер стека."""
        return len(self.__items)
```

Сложность операций стека:

	Время в среднем	Время в худшем случае
Добавление элемента в стек	O(1)	O(n)
Извлечение элемента	O(1)	O(n)
Определение размера стека (реализация на массиве)	O(1)	O(1)

Очередь

Очередь основана на принципе FIFO (англ. *first in, first out* — «первым вошёл — первым вышел»). Первым извлекают элемент, который добавили раньше всех.

Методы очереди:

- `push(item)` — добавляет элемент в конец очереди;
- `pop()` — возвращает элемент из начала очереди и удаляет его из очереди;
- `peek()` — возвращает элемент из начала очереди без удаления;
- `size()` — возвращает количество элементов в очереди.

Реализация очереди через класс:

```
class Queue:

    def __init__(self):
        # Для хранения элементов очереди применяем список.
        self.__items = []

    def push(self, item):
        """Добавить элемент в очередь."""
        # Добавляем элемент
```

```
        # в начало списка - на место элемента с индексом 0.
        self.__items.insert(0, item)

    def pop(self):
        """Извлечь элемент из очереди."""
        return self.__items.pop()

    def peek(self):
        """Получить элемент, но не удалять его из очереди."""
        return self.__items[-1]

    def size(self):
        """Вернуть размер очереди."""
        return len(self.__items)
```

Реализация очереди на основе списка имеет недостаток — вставка элемента в начало списка выполняется за время `O(n)` — ведь при каждой вставке придётся сдвигать все элементы массива так, чтобы они по-прежнему размещались в последовательно расположенных ячейках памяти.

Дек

Дек (англ. *deque*) — интерфейс, позволяющий извлекать и добавлять элементы с двух концов массива.

Методы дека:

- `push_back(item)` — вставка нового элемента в конец дека;
- `pop_back()` — возврат последнего элемента и удаление его из дека;
- `push_front(item)` — вставка нового элемента в начало;
- `pop_front()` — возврат первого элемента и удаление его из дека;
- `size()` — количество элементов в деке.

Предполагается, что каждый из этих методов должен работать за константное время `O(1)`, как и в очереди.

В Python есть готовая реализация дека, которую можно импортировать из модуля `collections`.

```
from collections import deque
```

У класса `deque` есть те же методы, что и у списка:

- `append()` — добавить новый элемент в конец дека,
- `extend()` — добавить к деку другой массив,
- `pop()` — получить последний элемент и удалить его из дека,
- …и все прочие.

Есть и специальные методы, присущие именно деку, например:

- `appendleft()` — добавить новый элемент в начало дека,
- `popleft()` — получить первый элемент и удалить его из дека.

Эти операции работают за константное время `O(1)`.

У встроенного класса `deque` есть полезная особенность: для него можно установить максимальную длину. Тогда при заполнении очереди всякий новый элемент будет выталкивать, удалять «лишние» элементы с противоположной стороны очереди.

```
from collections import deque

data = deque(maxlen=10)

for item in range(15):
    data.appendleft(item)

print(data)
```

Хеш-таблицы

Хеш-таблица — это коллекция элементов. Каждый элемент состоит из ключа и значения, которые доступны разработчику:

- при создании элемента разработчик задаёт пару «ключ-значение»,
- при поиске нужного элемента разработчик указывает ключ.

У хеш-таблиц есть строгие ограничения:

- Ключи элементов хеш-таблицы должны быть уникальными: двух одинаковых ключей в таблице быть не может.
- Ключи должны относиться к неизменяемым типам данных.

Особенности хеш-таблицы:

- Быстрый поиск элемента по ключу: переданный для поиска ключ преобразуется (как и при создании нового элемента) в хеш, а хеш — в индекс. Временная сложность извлечения элемента по индексу — `O(1)`.
- Добавление нового элемента в большинстве случаев выполняется за время, близкое к `O(1)`: смещения элементов не требуется, ведь из хешей создаются уникальные индексы, и новые элементы встают «на незанятые места» в массиве. Однако при увеличении длины массива потребуется реаллокация, и в этом случае добавление элемента пройдёт за время, примерно равное `O(n)`.
- Удаление элемента происходит за время `O(1)`: ключ удаляемого элемента преобразуется в хеш, а хеш — в индекс; элемент по индексу ищется за время `O(1)`.

Море знаний исследует мудрый, и
шпаргалка у него — как компас.

Риф Эгегейский, мореплаватель