The Pragmatic Programmers

Второе издание

прикладные

структуры данных и алгоритмы

прокачиваем навыки

Джей Венгроу

под редакцией Брайана Макдоналда

КРОК

Джей Венгроу

Прикладные структуры данных и алгоритмы. Прокачиваем навыки

Перевел с английского С. Черников

Научный редактор Анна Белых, старший инженер-разработчик компании КРОК

ББК 32.973.2-018
УДК 004.422.63+004.421

Венгроу Джей

В29 Прикладные структуры данных и алгоритмы. Прокачиваем навыки. — СПб.: Питер, 2024. — 512 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2068-0

Структуры данных и алгоритмы — это не абстрактные концепции, а турбина, способная превратить ваш софт в болид «Формулы-1». Научитесь использовать нотацию «О большое», выбирайте наиболее подходящие структуры данных, такие как хеш-таблицы, деревья и графы, чтобы повысить эффективность и быстродействие кода, что критически важно для современных мобильных и веб-приложений.

Книга полна реальных прикладных примеров на популярных языках программирования (Python, JavaScript и Ruby), которые помогут освоить структуры данных и алгоритмы и начать применять их в повседневной работе. Вы даже найдете слово, которое может существенно ускорить ваш код. Практикуйте новые навыки, выполняя упражнения и изучая подробные решения, которые приводятся в книге.

Начните использовать эти методы уже сейчас, чтобы сделать свой код более производительным и масштабируемым.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1680507225 англ.
ISBN 978-5-4461-2068-0

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 05.07.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 1500. Заказ 0000.

Краткое содержание

ГЛАВА 1

О важности структур данных

Когда человек только учится программировать, его основная цель — обеспечение правильной работы кода, которая оценивается с помощью одного простого критерия — фактической работоспособности.

Но с опытом к разработчикам ПО приходит понимание дополнительных нюансов, влияющих на *качество* кода. Они узнают, что два разных фрагмента кода могут решать одну задачу, но при этом один из них может быть *лучше* другого.

Есть много показателей качества кода, но один из важнейших — его сопровождаемость, которая охватывает такие аспекты, как читабельность, структурированность и модульность.

Еще одна отличительная черта качественного кода — его *эффективность*. Например, у вас может быть два фрагмента кода, решающих одну задачу, но один из них *может работать быстрее, чем другой*.

Взгляните на следующие две функции, каждая из которых выводит на экран все четные числа от 2 до 100:

```
def print_numbers_version_one():
  number = 2

  while number <= 100:
    # Если число четное, вывести его на экран:
    if number % 2 == 0:
      print(number)

    number += 1

def print_numbers_version_two():
  number = 2

  while number <= 100:
    print(number)
```

```
# Увеличить число на 2, чтобы получить следующее четное число:
number += 2
```

Как вы думаете, какая из функций работает быстрее?

Если вы выбрали версию 2, то вы правы. Дело в том, что в первой версии цикл выполняется 100 раз, а во второй — только 50. Получается, что версия 1 требует вдвое больше шагов, чем 2.

В этой книге мы будем говорить о написании *эффективного* кода. Умение писать код, который работает быстро, — один из важнейших навыков хорошего разработчика ПО.

Чтобы развить это умение, сначала нужно разобраться в том, что такое структуры данных и как они влияют на скорость работы создаваемого кода. Итак, начнем.

Структуры данных

Поговорим о данных.

Данные — это обширный термин, охватывающий все типы информации, вплоть до простейших чисел и строк. В классической программе «Hello World!» строка `"Hello World!"` будет фрагментом данных. По сути, даже самые сложные фрагменты данных обычно состоят из чисел и строк.

Структура данных — это то, как они *организованы*. Позже вы узнаете, что одни и те же данные могут быть организованы по-разному.

Рассмотрим следующий фрагмент кода:

```
x = "Hello! "
y = "How are you "
z = "today?"

print x + y + z
```

Эта простая программа работает с тремя фрагментами данных и выводит три строки для составления одного связного сообщения. Если бы нам потребовалось описать структуру данных в этой программе, мы бы сказали, что у нас есть три независимые строки, каждая из которых содержится в одной переменной.

Но эти же данные могут храниться и в массиве:

```
array = ["Hello! ", "How are you ", "today?"]

print array[0] + array[1] + array[2]
```

В этой книге вы узнаете, что организация данных важна не просто сама по себе, — она может значительно повлиять на *скорость выполнения вашего кода*. В зависимости от выбранного способа организации данных ваша программа может работать быстрее или медленнее, причем разница может быть в несколько порядков. А если вы создаете программу, которой предстоит обрабатывать большие объемы данных, или веб-приложение, используемое одновременно тысячами людей, то от выбранных структур данных может зависеть, будет ваше ПО нормально работать или сбоить из-за чрезмерной нагрузки.

Когда вы разберетесь с влиянием структур данных на производительность программного обеспечения, у вас появятся ключи к написанию быстрого и качественного кода, а вы сами перейдете на совершенно новый уровень как специалист.

В этой главе мы проанализируем две структуры данных: массивы и множества. Несмотря на их похожесть, они по-разному влияют на производительность программы. И проанализировать это влияние нам помогут описанные далее инструменты.

Массив: базовая структура данных

Массив — это одна из простейших структур данных в информатике. Надеюсь, вы уже работали с массивами и знаете, что это — списки элементов. Массив — универсальный инструмент, который может быть полезен во многих ситуациях. Рассмотрим небольшой пример.

В исходном коде приложения, позволяющего пользователям создавать и использовать списки покупок, вам может встретиться такой фрагмент:

```
array = ["apples", "bananas", "cucumbers", "dates", "elderberries"]
```

Этот массив включает пять строк, каждая из которых содержит наименование продукта, который я могу купить в супермаркете (вы *просто обязаны* попробовать ягоды бузины (elderberries)).

При работе с массивами применяется особый технический жаргон.

Под *размером* массива понимается количество содержащихся в нем элементов. В нашем примере размер массива равен 5, так как он содержит пять значений.

Индекс массива — это число, определяющее позицию элемента в массиве.

В большинстве языков программирования нумерация индексов начинается с 0. Итак, в нашем примере индекс элемента `"apples"` — 0, а `"elderberries"` — 4:

Операции над структурами данных

Чтобы оценить производительность структуры данных, например массива, нужно проанализировать способы взаимодействия кода с ней.

Есть четыре основных способа использования структур данных, которые мы называем *операциями:*

- *Чтение* — получение элемента из определенного места структуры данных. В случае с массивом это означает поиск значения с определенным индексом. Например, поиск названия продукта с индексом 2 — это *чтение* массива.
- *Поиск* — нахождение определенного значения в структуре данных. В случае с массивом это означает выяснение того, есть ли конкретное значение в массиве, и если да, то какой у него индекс. Пример: *поиск* элемента `"dates"` в нашем списке продуктов.
- *Вставка* — добавление нового значения в структуру данных, в нашем случае — в массив. Если бы мы решили добавить инжир (`"figs"`) в список покупок, это было бы примером *вставки* нового значения в массив.
- *Удаление* — исключение значения из структуры данных. В случае с массивом это означает удаление из него одного из элементов. Например, если мы решим исключить бананы (`"bananas"`) из списка покупок, это значение будет *удалено* из массива.

В этой главе мы попробуем разобраться в том, насколько быстро выполняется каждая из этих операций.

Измерение скорости

Итак, как же измерить скорость выполнения операции?

Если вы вынесете из этой книги только один урок, пусть он будет таким: при измерении скорости выполнения операции мы учитываем не количество *времени*, а число *шагов*, которое для этого требуется.

Мы уже сталкивались с этим в примере функции, выводящей на экран четные числа от 2 до 100. Вторая версия работала быстрее, потому что требовала вдвое меньше шагов, чем первая.

Почему мы измеряем скорость выполнения кода числом шагов?

Дело в том, что мы никогда не можем однозначно сказать, что какая-то операция занимает, скажем, пять секунд. Выполнение одного и того же фрагмента кода может занять пять секунд на более новом компьютере и гораздо больше времени на старом оборудовании. А, допустим, на суперкомпьютерах будущего этот же код сможет работать еще быстрее. Измерять скорость операции в секундах или минутах ненадежно, поскольку длительность зависит от оборудования, на котором она выполняется.

Но мы *можем* измерить скорость операции в количестве вычислительных *шагов*, необходимых для ее выполнения. Если операция А требует 5 шагов, а Б — 500, мы можем предположить, что первая всегда будет выполняться быстрее, чем вторая, на любых компьютерах. Поэтому определение числа шагов — ключевой этап анализа скорости выполнения операции.

Измерение скорости выполнения операции по-другому еще называют измерением ее *временной сложности*. В книге я буду использовать термины *скорость*, *временная сложность*, *эффективность*, *производительность* и *время выполнения* как синонимы. Все они относятся к количеству шагов, необходимых для выполнения операции.

Теперь рассмотрим четыре операции над массивом и определим, сколько шагов нужно для выполнения каждой из них.

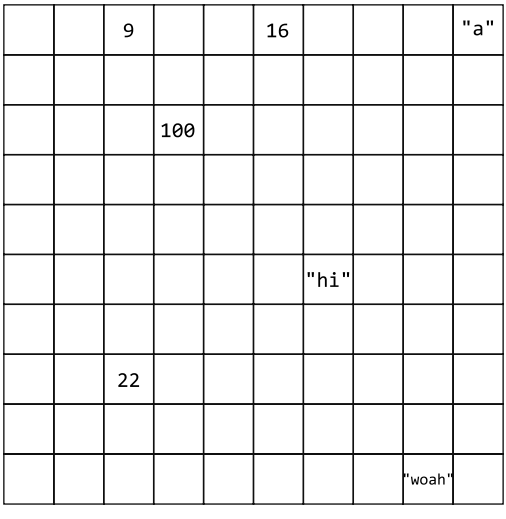
Чтение

Начнем с *чтения* — с определения значения элемента массива с конкретным индексом.

Компьютер может выполнить эту операцию всего за один шаг, потому что может обратиться к любому элементу массива с необходимым индексом и считать его значение. Если бы мы решили найти элемент с индексом 2 в массиве `["apples", "bananas", "cucumbers", "dates", "elderberries"]`, то компьютер перешел бы прямо к нему и сообщил, что в этой позиции находится значение `"cucumbers"`.

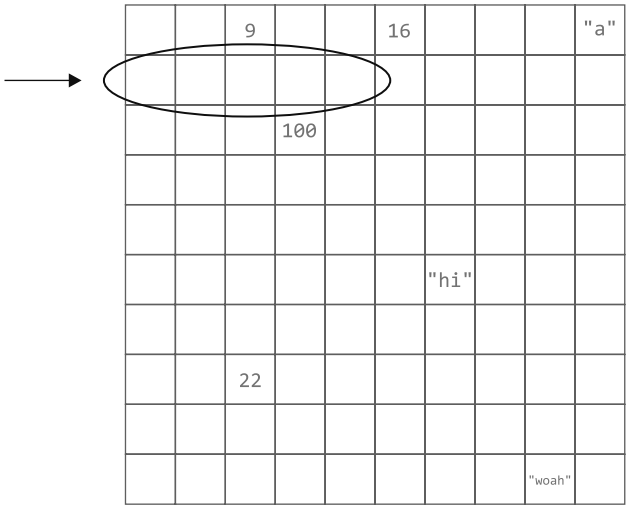
Как компьютеру удается отыскать нужный элемент всего за шаг? Давайте разберемся.

Память компьютера похожа на гигантский набор ячеек. Ниже показана сетка, где некоторые ячейки пустые, а некоторые содержат биты данных.



Хотя это очень упрощенная модель устройства памяти компьютера, она хорошо передает основную идею.

При объявлении массива программа выделяет для него непрерывный блок памяти из пустых ячеек. Так, если вы создаете массив для хранения пяти элементов, ваш компьютер находит группу из пяти пустых ячеек подряд и выделяет ее для использования в качестве массива:



У каждой ячейки есть определенный адрес. От почтового, где содержатся названия улиц и городов, его отличает лишь то, что он представлен числом. Значение адреса каждой последующей ячейки памяти на одну единицу превышает значение предыдущей:

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
1010	1011	1012	1013	1014	1015	1016	1017	1018	1019
1020	1021	1022	1023	1024	1025	1026	1027	1028	1029
1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
1040	1041	1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057	1058	1059
1060	1061	1062	1063	1064	1065	1066	1067	1068	1069
1070	1071	1072	1073	1074	1075	1076	1077	1078	1079
1080	1081	1082	1083	1084	1085	1086	1087	1088	1089
1090	1091	1092	1093	1094	1095	1096	1097	1098	1099

На следующей схеме показан массив из нашего примера со списком покупок с индексами и адресами ячеек памяти.

	"apples"	"bananas"	"cucumbers"	"dates"	"elderberries"
Адрес ячейки памяти:	1010	1011	1012	1013	1014
Индекс:	0	1	2	3	4

Когда компьютер считывает значение по определенному индексу массива, он может сразу обратиться к нужному элементу в силу следующих факторов.

1. Компьютер может перейти к любому *адресу памяти* за один шаг. Например, если вы попросите его проверить значение в ячейке памяти с адресом 1063, он сможет получить к нему доступ без выполнения поиска. Точно так же, если я попрошу вас поднять мизинец правой руки, вам не придется перебирать все пальцы, чтобы определить нужный.

2. Выделяя блок памяти под массив, компьютер отмечает, с какого адреса он *начинается*. Так, если мы попросим компьютер найти первый элемент массива, он мгновенно обратится к соответствующему адресу памяти.

Теперь мы знаем, как именно компьютер может найти первое значение массива за один шаг. Впрочем, он может найти значение с *любым* индексом, выполнив простое сложение. Если бы мы попросили компьютер найти значение массива с индексом 3, он просто взял бы адрес памяти, соответствующий индексу 0, и прибавил к нему 3 (в конце концов, у адресов памяти последовательная нумерация).

Применим это к нашему массиву списка продуктов. Он начинается с адреса памяти 1010. Итак, если бы мы велели компьютеру считать значение элемента с индексом 3, он выполнил бы следующую логическую операцию.

1. Массив начинается с индекса 0, который соответствует адресу памяти 1010.
2. Индекс 3 находится ровно через три ячейки после индекса 0.
3. Логично предположить, что элемент с индексом 3 находится в ячейке памяти с адресом 1013, поскольку 1010 + 3 = 1013.

Как только компьютер выяснит, что индекс 3 соответствует адресу памяти 1013, он обратится к соответствующей ячейке и узнает, что в ней содержится значение `"dates"`.

Чтение из массива — довольно эффективная операция, ведь компьютер может обратиться к любому адресу памяти за один шаг. Хотя я разбил мыслительный процесс компьютера на три этапа, сейчас нас интересует его обращение к адресу памяти (в следующих главах мы подробнее поговорим о том, на каких шагах стоит сосредоточиться).

Конечно же, операция, которая выполняется всего за шаг, самая быстрая. Выходит, что массив — это не только основная, но и очень мощная структура данных, потому что мы можем считывать значения в нем с огромной скоростью.

А что, если бы вместо выяснения значения в позиции с индексом 3 мы попросили бы компьютер узнать индекс элемента `"dates"`? В этом случае речь пойдет об операции поиска, которую мы рассмотрим далее.

Поиск

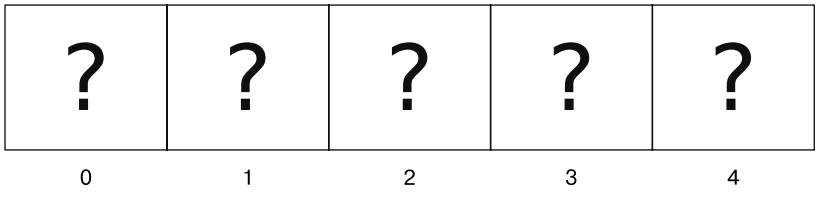
Как я уже говорил, при *поиске* в массиве компьютер проверяет, есть ли в нем конкретное значение, и если да, то в позиции с каким индексом.

В каком-то смысле эта операция обратна чтению, при котором мы предоставляем компьютеру *индекс* и просим возвратить соответствующее ему значение. При поиске же мы даем компьютеру *значение* и просим возвратить его индекс.

Эти две операции кажутся похожими, но в плане эффективности они сильно разнятся. Чтение выполняется быстро, так как компьютер может сразу обратиться к любому индексу и выяснить значение соответствующего элемента. Поиск требует времени, ведь у компьютера нет возможности сразу перейти к нужному значению.

Важный факт: компьютер может мгновенно получить доступ к любому адресу памяти, но не имеет ни малейшего представления о том, какие *значения* содержатся в каждой из ячеек.

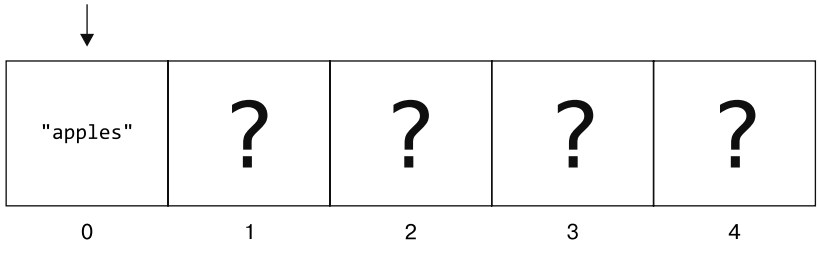
Вернемся к нашему примеру со списком продуктов. Компьютер не может сразу узнать содержимое каждой ячейки памяти. Для него массив выглядит примерно так:



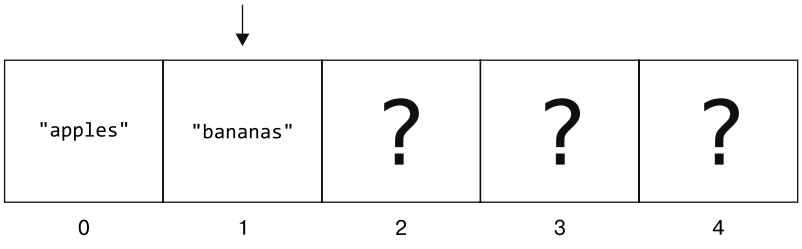
Чтобы найти нужное значение в массиве, он вынужден проверять все ячейки по очереди.

На следующих изображениях показан процесс, который компьютер будет использовать для поиска значения `"dates"` в нашем массиве.

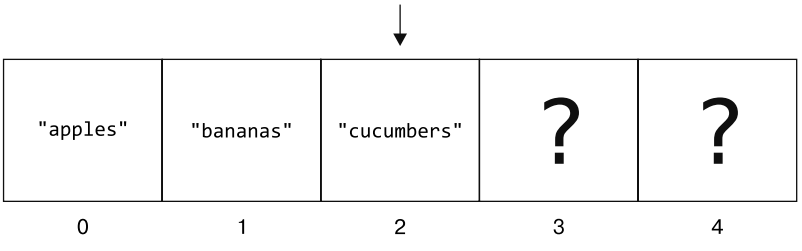
Сначала компьютер проверяет элемент с индексом 0:



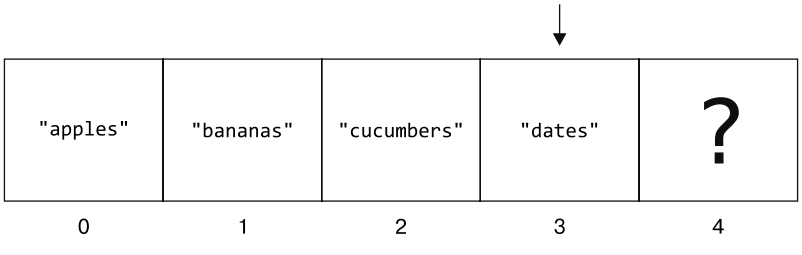
Поскольку его значение `"apples"`, а не `"dates"`, компьютер переходит к следующему индексу:



В позиции с индексом 1 тоже нет искомого значения `"dates"`, поэтому компьютер переходит к индексу 2:



И снова нам не повезло, и компьютер переходит к следующей ячейке:



Ура! Мы нашли нужное значение и теперь знаем, что его индекс — 3. Компьютеру больше не нужно переходить к следующей ячейке массива, так как он уже нашел то, что мы просили.

В этом примере поиск был выполнен за четыре шага, поскольку компьютеру пришлось проверить четыре ячейки, чтобы обнаружить значение `"dates"`.

В главе 2 вы узнаете о другом способе поиска в массиве, но эта базовая операция поиска, при которой компьютер проверяет каждую ячейку по одной, называется *линейным поиском*.

Какое же *максимальное* количество шагов может выполнить компьютер, чтобы провести линейный поиск в массиве?

Если искомое значение окажется в последней ячейке (например, `"elderberries"`), компьютеру придется перебрать *все* элементы массива, чтобы его отыскать.

Кроме того, если этого значения и вовсе нет в массиве, все равно придется проверить каждую ячейку, чтобы в этом убедиться.

Итак, получается, что при линейном поиске в массиве из пяти элементов максимальное число шагов равно пяти, а в массиве из 500 элементов — 500.

Проще говоря, линейный поиск в массиве из *N* элементов потребует максимум *N* шагов. Здесь *N* — переменная, которую можно заменить любым числом.

В любом случае понятно, что поиск менее эффективен, чем чтение, поскольку может требовать множества шагов, тогда как чтение всегда предполагает только один, вне зависимости от размера массива.

Теперь разберем операцию вставки.

Вставка

Эффективность операции вставки нового фрагмента данных в массив зависит от того, *куда* именно вы его вставляете.

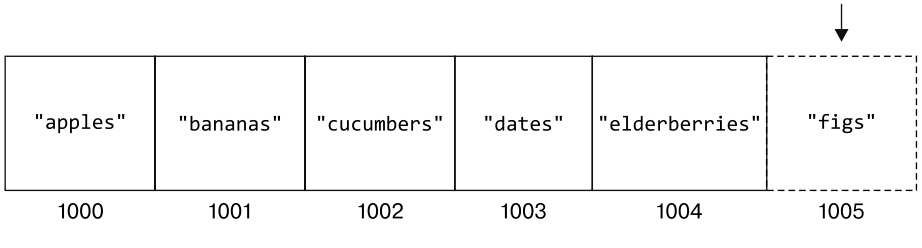
Допустим, мы хотим добавить значение `"figs"` в конец нашего списка покупок. На такую вставку у нас уйдет один шаг.

Это обусловлено еще одним фактором: при выделении блока памяти под массив компьютер всегда отслеживает его размер.

Если мы прибавим к этому тот факт, что компьютер знает, с какого адреса памяти массив начинается, то вычислить местонахождение его последнего элемента не составит труда: если массив начинается с адреса памяти 1010, а его размер 5, то его последний элемент хранится в ячейке 1014. Выходит, вставка элемента после него означала бы его добавление к *следующему* адресу памяти, равному 1015.

После вычисления адреса ячейки, в которую нужно поместить новое значение, компьютер сможет сделать это за один шаг.

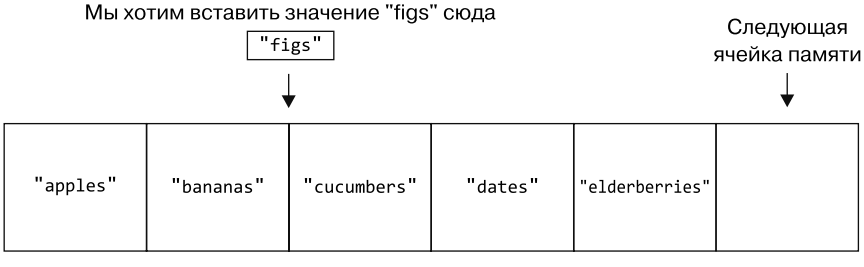
Так выглядит вставка значения `"figs"` в конец массива:



Но есть одна проблема. Поскольку изначально компьютер выделил под массив только пять ячеек памяти, при добавлении шестого элемента ему придется найти дополнительные ячейки. Как правило, это делается автоматически, но каждый язык программирования обрабатывает этот процесс по-своему, поэтому я не буду вдаваться в детали.

Мы рассмотрели процесс вставки нового элемента в конец массива, но вставка в *начало* или в *середину* — это совсем другая история. В этих случаях нам приходится *сдвигать* фрагменты данных, чтобы освободить место для нового значения, а это требует дополнительных шагов.

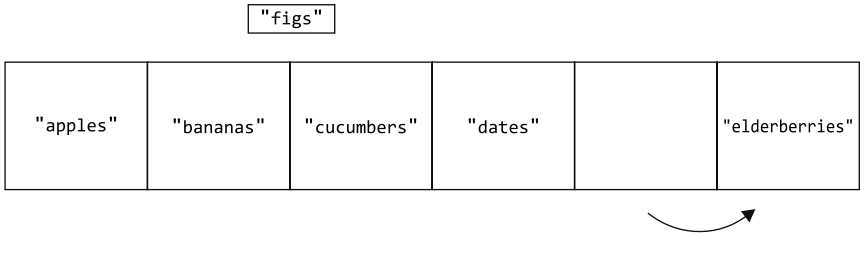
Допустим, мы хотим добавить значение `"figs"` в позицию массива с индексом 2:



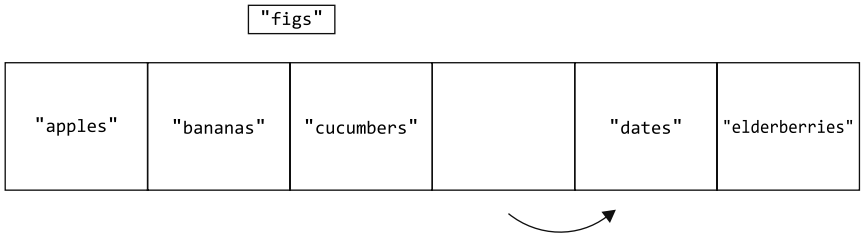
Чтобы освободить место для нового значения, нам нужно сдвинуть `"cucumbers"`, `"dates"` и `"elderberries"` вправо. На это у нас уйдет несколько шагов, так как нам нужно сначала сдвинуть значение `"elderberries"` на одну позицию вправо, чтобы освободить место для перемещения `"dates"`, а затем сдвинуть значение `"dates"`, чтобы освободить место для `"cucumbers"`.

Рассмотрим этот процесс подробнее.

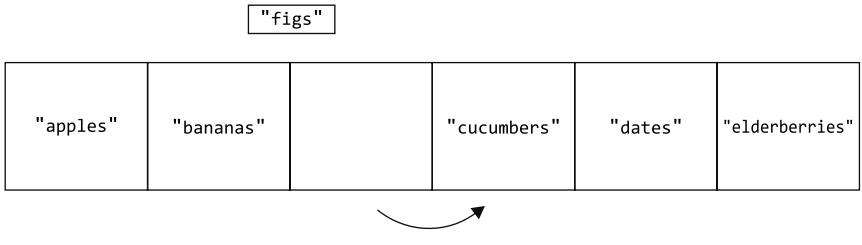
Шаг 1: сдвигаем вправо значение `"elderberries"`:



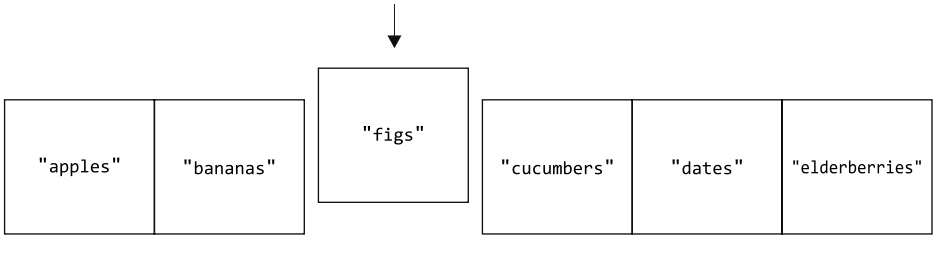
Шаг 2: сдвигаем вправо значение `"dates"`:



Шаг 3: сдвигаем вправо значение `"cucumbers"`:



Шаг 4: вставляем значение `"figs"` в позицию с индексом 2:



Обратите внимание, что в этом примере на операцию вставки ушло четыре шага: три из них заняло передвижение элементов вправо и только один ушел на фактическую вставку нового значения.

Худший сценарий при вставке в массив — тот, на который уйдет больше всего шагов, — вставка значения в *начало*, ведь нам придется сдвинуть вправо *все* остальные значения.

Проще говоря, в худшем случае вставка значения в массив из N элементов может потребовать $N + 1$ *шагов,* ведь перед вставкой нового значения нам нужно переместить все N элементов.

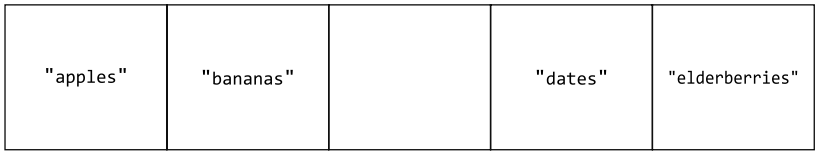
Теперь рассмотрим последнюю операцию над массивом — удаление элемента.

Удаление

Удаление элемента массива сводится к исключению значения в позиции с определенным индексом.

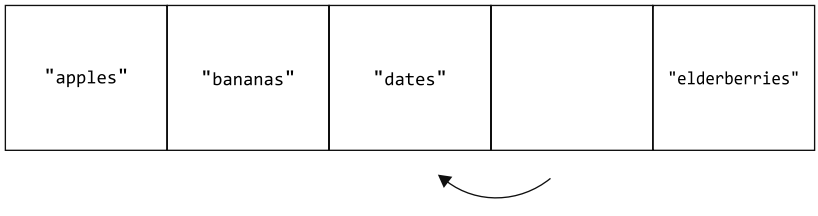
Давайте удалим из нашего массива значение с индексом 2 — `"cucumbers"`.

Шаг 1: удаляем значение `"cucumbers"` из массива:

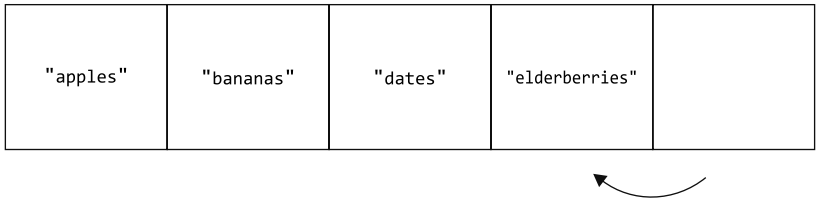


Хотя на удаление значения ушел всего шаг, у нас возникла пустая ячейка в середине массива. Сложно работать с массивом, когда в нем есть пробелы, поэтому для решения этой проблемы нам нужно сдвинуть влево значения `"dates"` и `"elderberries"`. Это означает, что процесс удаления требует дополнительных шагов.

Шаг 2: сдвигаем влево значение `"dates"`:



Шаг 3: сдвигаем влево значение `"elderberries"`:



Итак, на операцию удаления у нас ушло в общей сложности три шага: первый сводится к фактическому удалению значения, а два других — к сдвигу остальных значений для избавления от пробела.

Как и в случае вставки, худший сценарий при удалении — избавление от первого элемента массива, потому что для избавления от пробела в позиции с индексом 0 нам пришлось бы сдвинуть влево все оставшиеся элементы.

В случае с массивом из пяти элементов удаление первого потребовало бы одного шага, а сдвиг четырех оставшихся — четырех. В случае с массивом из 500 элементов удаление первого потребовало бы одного шага, а сдвиг оставшихся — 499. Следовательно, операция удаления значения из массива, состоящего из N элементов, потребует максимум N шагов.

Поздравляю! Вы проанализировали временну́ю сложность первой структуры данных. Теперь пришло время узнать о том, что у разных структур данных разная эффективность. Это важный аспект, так как выбор структуры данных для использования в коде может сильно повлиять на производительность вашего ПО.

Следующая структура данных — *множество* — на первый взгляд очень похожа на массив. Но позже вы увидите, что эффективность операций над массивами и множествами разная.

Множества: как одно правило может повлиять на эффективность

Теперь поговорим о *множестве*. Множество — это структура данных без повторяющихся значений.

Есть разные типы множеств, но мы будем говорить о *множестве на базе массива*. Это, как и массив, простой список значений. Единственная разница между ним и классическим массивом в том, что множество не допускает вставку повторяющихся значений.

Например, если вы попытаетесь добавить еще одно значение `"b"` в множество `["a", "b", "c"]`, компьютер просто не позволит этого сделать — ведь `"b"` уже есть в этом множестве.

Множества полезны, когда вам нужно гарантировать отсутствие дубликатов.

Например, если вы создаете телефонный онлайн-справочник, вам не нужно, чтобы один и тот же номер встречался в нем более одного раза. На самом деле, я сам столкнулся с такой проблемой: в местной телефонной книге наш домашний номер дублируется — по какой-то ошибке он указан не только рядом с моей фамилией, но и рядом с фамилией Зиркинд (да, это реальная история). Вы даже не представляете, как мне надоело получать телефонные звонки и сообщения

от людей, пытающихся дозвониться до Зиркиндов. Я уверен, что и Зиркинды недоумевают, почему им никто не звонит. А когда я звоню Зиркиндам, чтобы сообщить об ошибке, трубку берет моя жена, потому что я набираю свой номер (ладно, последнюю часть я придумал). Если бы только в этой программе использовалось множество...

В любом случае, множество на базе массива — это массив с одним дополнительным ограничением: запретом дубликатов. Хотя этот запрет полезен, он влияет на *эффективность* одной из четырех основных операций.

Рассмотрим проведение четырех основных операций на базе массива.

Чтение из множества выполняется точно так же, как и из массива — чтобы выяснить значение в позиции с определенным индексом, компьютеру нужен всего один шаг. Как вы уже знаете, это связано с тем, что компьютер может перейти к любому индексу множества, потому что способен с легкостью вычислить соответствующий адрес памяти и обратиться к нему.

Поиск в множестве тоже ничем не отличается от поиска в массиве и требует до N шагов. То же самое касается удаления элемента из множества: чтобы удалить значение и сдвинуть данные влево, нужно до N шагов.

Но, когда дело доходит до вставки, между массивами и множествами появляется различие. Сначала рассмотрим процесс вставки значения в *конец* множества (лучший сценарий для массива). Как мы уже убедились, компьютер может вставить значение в конец массива за один шаг.

В случае с множеством компьютеру сначала нужно убедиться, что вставляемого значения в нем еще нет, потому что множества используются именно для предотвращения вставки дубликатов.

Как же удостовериться в том, что множество еще не содержит данные, которые мы пытаемся в него добавить? Как вы помните, компьютер не знает, какие значения содержатся в ячейках массива или множества, поэтому ему сначала нужно выполнить *поиск* и убедиться, что значения, которое мы хотим вставить, нет. Только тогда компьютер разрешит вставку.

Итак, в случае с множеством каждая операция вставки требует *предварительного выполнения поиска*.

Рассмотрим этот процесс на примере. Представьте, что наш прошлый список продуктов сохранен в виде множества. Это было бы верным решением, ведь мы не хотим дважды покупать один и тот же продукт. Если мы захотим вставить в множество `["apples", "bananas", "cucumbers", "dates", "elderberries"]`