print('Структуры данных в Python:')
print('Начальный курс')

Структуры данных в Python: начальный курс

print('Дональд Р. Шихи')

Дональд Р. Шихи

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Дональд Р. Шихи

Структуры данных в Python: начальный курс

A First Course on Data Structures in Python

Donald R. Sheehy

Структуры данных в Python: начальный курс

Дональд Р. Шихи

ДМК ПРЕСС
издательство
Москва, 2022

УДК 004.438Python
ББК 32.973.22
Ш58

Дональд Р. Шихи
Ш58 Структуры данных в Python: начальный курс / пер. с англ. А. В. Снастина. – М.: ДМК Пресс, 2022. – 186 с.: ил.

ISBN 978-5-93700-110-8

В книге рассматриваются основополагающие вопросы, относящиеся к структурам данных в языке программирования Python. Теоретические концепции и абстрактные понятия подкрепляются простыми примерами. По мере изучения основ вводятся такие темы, как стратегии решения задач, продвинутое использование языка Python, принципы объектно-ориентированного проектирования и методологии тестирования. Подробно рассматриваются структуры данных, встроенные в язык Python, а также абстрактные типы данных (АТД): стеки, очереди, связные списки, деревья, графы и др.

Книга предназначена для всех, кто изучает язык программирования Python и предполагает активно использовать как встроенные структуры данных, так и собственные реализации АТД.

УДК 004.438Python
ББК 32.973.22

ISBN 978-5-93700-110-8

Оглавление

Глава 1

Предисловие

Эта книга предназначена для того, чтобы быстро рассмотреть множество основополагающих вопросов, не сокращая процесс обучения.

Что это означает? Это означает, что концепции обосновываются и подробно объясняются, и для начала приводятся простые примеры. Общие принципы сопровождаются задачами. Абстрактные понятия подтверждаются конкретными примерами.

Что это не означает? Эта книга не является ни исчерпывающим, охватывающим все темы руководством по структурам данных, ни полноценным введением во все подробности языка программирования Python. Представление минимально необходимых знаний для создания интересных программ и изучения полезных концепций – это не сокращение процесса обучения, это просто указание направления.

Существует множество книг, обучающих тонкостям программирования на Python, много других книг, в которых описываются способы решения задач, структуры данных и/или алгоритмы. Многие книги предназначены для изучения шаблонов проектирования, процесса тестирования и прочих важных практических аспектов программной инженерии. Цель этой книги – рассмотреть многие из этих тем как часть единого комплексного курса.

Организация процесса для достижения этой цели одновременно проста и сложна. Простая часть состоит в том, что общая последовательность основных тем определяется задачами обработки структур данных, как видно из заголовков глав. Сложная часть состоит в том, что многие другие концепции, включая стратегии решения задач, более продвинутое использование языка Python, принципы объектно-ориентированного проектирования и методологии тестирования вводятся постепенно по всему тексту в логическом, последовательном стиле.

В итоге книга не предназначена для использования в качестве справочника. Ее нужно проработать от начала до конца. Многие весьма близкие мне темы были опущены, чтобы можно было проработать всю книгу за один учебный семестр.

Глава 2

Основы языка программирования Python

Эта книга не является начальным курсом по программированию. Предполагается, что читатель обладает некоторым практическим опытом в программировании. Следовательно, также предполагается, что читатель уже знаком с некоторыми основными концепциями, самой основополагающей из которых является ментальная модель программирования, которую иногда называют «Последовательность, выбор и итерация».

2.1. Последовательность, выбор и итерация

Постоянно повторяющаяся тема в этом курсе – процесс перехода от размышлений о коде к написанию кода. Мы постараемся сформировать способ мышления о программах, способ написания программ и путь перехода между ними в обоих направлениях. То есть нам необходимо получить возможность как для непосредственной работы с исходным кодом, так и для высокоуровневого описания программ.

Превосходная модель для мышления об (императивном) программировании называется «Последовательность, выбор и итерация».

1. Последовательность (sequence) – выполнение операций по одной за один раз в заданном порядке.
2. Выбор (selection) – использование условных инструкций (операторов), таких как `if`, для выбора следующих операций для выполнения.
3. Итерация (iteration) – многократное повторение некоторых операций с использованием циклов или рекурсии.

В любом существующем языке программирования обычно имеется несколько механизмов для выбора и итерации, тогда как последовательность просто является поведением по умолчанию.

В действительности вы должны иметь в своем распоряжении специальные конструкции языка, чтобы сделать что-либо, отличающееся от выполнения заданных операций в заданном порядке.

2.2. Выражения и вычисление

Python может выполнять простые арифметические действия. Например, `2 + 2` – это простое арифметическое выражение (expression). Выражения вычисляются и в результате дают некоторое значение (value). Некоторые значения являются числовыми, как в примере `2 + 2`, но это не обязательно. Например, `5 > 7` – это выражение, результатом которого является логическое значение `False` (ложь). Выражения могут становиться более сложными при объединении нескольких (или даже многих) операций и функций. Например, выражение `5 * (3 + abs(-12) / 3)` содержит четыре различные функции. Круглые скобки и порядок записи операций определяют порядок вычисления этих функций. Напомню, что в программировании порядок выполнения операций также называется приоритетом операторов (operator precedence). В приведенном выше примере функции (операции) выполняются в следующем порядке: `abs`, `/`, `+`, `*`.

2.3. Переменные, типы и состояние

Предположим, что вы пытаетесь решить некоторую сложную математическую задачу без компьютера. Только при помощи бумаги (и карандаша). Вы записываете свое решение, чтобы можно было им воспользоваться в дальнейшем. Это тоже в некотором роде программирование. При каких-либо вычислениях зачастую случается так, что необходимо сохранить их ход на будущее, когда снова потребуется их использовать. Мы достаточно часто называем такую сохраненную (записанную) информацию состоянием (state).

Информация хранится в переменных (variables). В языке Python переменная создается инструкцией присваивания (assignment). Эта инструкция имеет следующую форму:

```
variable_name = some_value
```

Здесь знак равенства (`=`) выполняет некоторое действие (присваивание), а не описывает что-либо (равенство). Часть справа от знака равенства является выражением, которое вычисляется в первую очередь. Только после вычисления выражения в правой части выполняется присваивание. Если в левой части присваивания находится имя переменной, которая уже существует, то ее значение перезаписывается (замещается). Если переменная до этого не существовала, то она создается.

Порядок вычислений чрезвычайно важен. Необходимость вычисления в первую очередь правой части означает, что присваивания, подобные `x = x + 1`, имеют смысл, потому что значение `x` не изменяется до тех пор, пока не будет вычислено выражение `x + 1` (и только после этого вычисления выполняется присваивание). Кстати, существует более короткая запись этого вида обновления: `x += 1`. Также существуют аналогичные формы записи для операций `-=`, `*=` и `/=`.

Сама инструкция присваивания не является выражением. Она не имеет какого-либо значения. Это оказывается полезным свойством, позволяющим избежать широко распространенной ошибки из-за путаницы с присваиванием и проверкой на равенство (т. е. `x == y`). Но повторение инструкции присваива-

ния «в цепочку», например x = y = 1, не работает так, как ожидается, т. е. не присваивает 1 обеим переменным x и y.

Переменные – это всего лишь имена. Каждое имя связано с некоторым фрагментом данных, называемым объектом (object).

Имя – это строка символов, которая отображается в некоторый объект. Само по себе имя переменной интерпретируется как выражение, из которого вычисляется любой объект, с которым связано это имя. Такое отображение строк в объекты часто изображается графически в виде прямоугольников, представляющих объекты, и стрелок, показывающих отображение.

Каждый объект имеет тип (type). Тип часто определяет, что можно сделать с конкретной переменной. Так называемые элементарные (или атомные) типы (atomic types) в Python – это целые числа (integers), числа с плавающей точкой (floats) и логические значения (booleans), но любая полезная программа непременно будет также содержать переменные многих других типов. Тип переменной можно узнать с помощью встроенной функции `type()`. В Python термины тип (type) и класс (class) означают одно и то же (в большинстве случаев).

Различие между переменной и объектом, который она представляет, может быть потеряно, когда мы говорим о них в общем смысле, потому что переменная обычно действует как имя (name) объекта. В определенных случаях полезно четко различать переменную и объект, особенно при копировании объектов. Вероятно, вы захотите увидеть это различие на некоторых примерах копирования объектов из одной переменной в другую. Влияет ли изменение одного объекта на другой?

```
x = 5
y = 3.2
z = True
print("x has type", type(x))
print("y has type", type(y))
print("z has type", type(z))

x has type <class 'int'>
y has type <class 'float'>
z has type <class 'bool'>
```

Вы должны считать, что объект обладает тремя характеристиками: идентичностью (точнее, идентификатором) (identity), типом (type) и значением (value). Идентичность (идентификатор) объекта изменяться не может. Идентификатор можно использовать, чтобы узнать, являются ли два объекта в действительности одним и тем же объектом, с помощью ключевого слова `is`. Например, рассмотрим исходный код, приведенный ниже.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)
print(x is z)
print(x == z)

True
False
True
```

Объект не может изменить свой идентификатор. В Python вы также не можете изменить тип любого объекта. Можно выполнить переприсваивание (повторное присваивание) переменной, чтобы она указывала на другой объект другого типа, но это не одно и то же. Существует несколько функций, которые, как может показаться, изменяют типы объектов, но в действительности они просто создают новый объект из старого.

```
x = 2
print("x =", x)
print("float(x) =", float(x))
print("x still has type", type(x))

print("Overwriting x.")
x = float(x)
print("Now, x has type", type(x))
```

```
x = 2
float(x) = 2.0
x still has type <class 'int'>
Overwriting x.
Now, x has type <class 'float'>
```

Кроме того, можно делать еще более сложные вещи.

```
numstring = "3.1415926"
y = float(numstring)
print("y has type", type(y))

best_number = 73
x = str(best_number)
print("x has type", type(x))

thisworks = float("inf")
print("float(\'inf\') has type", type(thisworks))
infinity_plus_one = float('inf') + 1

y has type <class 'float'>
x has type <class 'str'>
float('inf') has type <class 'float'>
```

В приведенном выше примере представлен новый тип – строка (string). Строка – это последовательность символов. В Python нет специального класса для одного символа (как, например, в языке C). Если необходим один символ, то используется строка с длиной, равной единице.

Значение объекта может изменяться или не изменяться в зависимости от типа конкретного объекта. Если значение можно изменить, то такой объект называется изменяемым (mutable). Если изменение значения запрещено, то объект считается неизменяемым (immutable). Например, строки – это неизменяемые объекты. Если требуется перевести все символы в строке в нижний регистр, то придется создать новую строку.

2.4. Наборы данных

Пятью не менее важными типами данных в Python являются строки, списки, кортежи, словари и множества. Их называют наборами (или последовательностями) данных, потому что каждый из этих типов может использоваться для хранения некоторого набора сущностей. В этом курсе мы рассмотрим многие другие примеры наборов данных.

2.4.1. Строки (str)

Строки (strings) – это последовательности символов, которые можно использовать для хранения текста любого вида. Следует отметить, что можно объединять (concatenate – сцеплять) строки для создания новой строки, используя для этого знак «плюс». Также можно получать доступ к отдельным символам в строке, применяя квадратные скобки и числовой индекс (index). Имя класса для строк – `str`. В большинстве случаев можно преобразовать другие объекты в строки.

```
s = "Hello, "
t = "World."
u = s + t
print(type(u))
print(u)
print(u[9])
n = str(9876)
print(n[2])
```

```
<class 'str'>
Hello, World.
r
7
```

2.4.2. Списки (list)

Списки (lists) – это упорядоченные последовательности объектов. Объекты в списке не обязаны иметь один и тот же тип. Списки обозначаются квадратными скобками, внутри которых записываются элементы (elements или items) конкретного списка, разделенные запятыми. Можно добавить элемент в конец списка `L`, воспользовавшись командой `L.append(newitem)`. Также можно применять индекс в списке, как это было описано для строк.

```
L = [1,2,3,4,5,6]

print(type(L))
```

```
<class 'list'>
```

На рис. 2.1 показано визуальное представление списка.

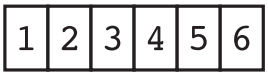

Рис. 2.1. Список

На рис. 2.2 изображен тот же список после выполнения команды `L.append(100)`.

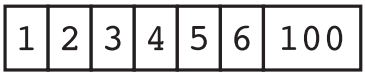

Рис. 2.2. Добавление элемента в конец списка

Элементы списка доступны по индексу (index). Нумерация индексов всегда начинается с 0. Отрицательные индексы позволяют начать отсчет с конца списка.

```
print("The first item is", L[0])
print("The second item is", L[1])
print("The last item is", L[-1])
print("The second to last item is", L[-2])

The first item is 1
The second item is 2
The last item is 100
The second to last item is 6
```

Кроме того, можно перезаписывать значения в списке, используя обычные инструкции присваивания.

```
L[2] = 'skip'
L[3] = 'a'
L[4] = 'few'
L[-2] = 99

from ds2.figs import drawlist

drawlist(L, 'list03')
```

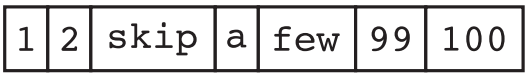

Рис. 2.3. Список после изменения его элементов

2.4.3. Кортежи (tuple)

Кортежи (tuples) – это также упорядоченные последовательности объектов, но, в отличие от списков, они неизменяемы. Можно получать доступ к элементам, но нельзя изменять их после создания кортежа. Например, попытка применения метода append для кортежа приводит к исключению.

```
t = (1, 2, "skip a few", 99, 100)

print(type(t))
print(t)
print(t[4])

<class 'tuple'>
(1, 2, 'skip a few', 99, 100)
100
```

Ниже показано, что происходит, если попытаться добавить элемент в кортеж.

```
t.append(101)

Traceback (most recent call last):
  File "bntsuj6iwl", line 3, in <module>
    t.append(101)
AttributeError: 'tuple' object has no attribute 'append'
               # объект 'tuple' не имеет атрибута 'append'
```

А вот что происходит при попытке присваивания значения одному из элементов кортежа.

```
t[4] = 99.5

Traceback (most recent call last):
  File "dar819jypk", line 3, in <module>
    t[4] = 99.5
TypeError: 'tuple' object does not support item assignment
          # объект 'tuple' не поддерживает присваивание значений элементам
```

Обратите внимание: точно таким же свойством обладают и строки.

```
s = 'ooooooooo'
s[4] = 'x'

Traceback (most recent call last):
  File "6qyosbli42", line 4, in <module>
    s[4] = 'x'
TypeError: 'str' object does not support item assignment
          # объект 'str' не поддерживает присваивание значений элементам
```

2.4.4. Словари (dict)

Словари (dictionaries) хранят пары ключ–значение (key-value). Таким образом, каждый элемент словаря содержит две части: ключ (key) и значение (value). Если у вас есть ключ, то можно получить соответствующее значение. Название происходит от основного принципа, по которому построен обычный словарь (книга): слово (ключ) позволяет найти его определение (значение).

Синтаксис операций доступа и присваивания значений для словарей точно такой же, как и для списков.

```
d = dict()
d[5] = 'five'
d[2] = 'two'
d['pi'] = 3.1415926

print(d)
print(d['pi'])

{5: 'five', 2: 'two', 'pi': 3.1415926}
3.1415926
```

Ключи могут иметь различные типы, но типы ключей обязательно должны быть неизменяемыми, т. е. элементарными (атомными) типами, кортежами или строками. Причина такого требования заключается в том, что место хранения любого значения строго определяется соответствующим ключом. Если ключ изменится, то при очередной попытке поиска по нему мы попадем совсем не туда, куда нужно.

Словари также называют отображениями (maps, mappings) или хеш-таблицами (hash tables). Немного позже в процессе изучения курса мы более подробно рассмотрим, как создаются словари. Элементы словаря не располагаются в каком-либо определенном порядке.

Если присваивается ключ, которого нет в словаре, то просто создается новый элемент. При попытке доступа к ключу, отсутствующему в словаре, возникает исключение `KeyError`.

```
D = {'a': 'one', 'b': 'two'}
D['c']
```

```
Traceback (most recent call last):
  File "egybsf1d93", line 2, in <module>
    D['c']
KeyError: 'c'
```

2.4.5. Множества (set)

Множества (sets) соответствуют математическому понятию. Это наборы объектов без дубликатов. Для обозначения множеств используются фигурные скобки, а для разделения элементов — запятые. Как и в словарях, в множествах нет какого-либо определенного порядка элементов. Поэтому мы говорим, что множества и словари – неупорядоченные наборы данных, т. е. они не являются последовательностями (nonsequential collections).

Следует особо отметить, что пустые фигурные скобки определяют пустой словарь, а не пустое множество. Ниже приведен пример создания нового множества. В него добавляются некоторые элементы. Обратите внимание: добавляемые дублирующиеся элементы игнорируются, что можно видеть при выводе значений множества.

```
s = {2,1}
print(type(s))
s.add(3)
s.add(2)
s.add(2)
s.add(2)
print(s)
```

```
<class 'set'>
{1, 2, 3}
```

Будьте осторожны, {} – это пустой словарь. Если необходимо создать пустое множество, то следует воспользоваться командой `set()`.

2.5. Некоторые общие правила работы с наборами данных

Существует несколько операций, которые можно выполнять с любыми классами наборов данных (разумеется, часто и с многими другими типами объектов).

Можно определить число элементов в наборе (длину набора – length) с помощью функции `len`.

```
a = "a string"
b = ["my", "second", "favorite", "list"]
c = (1, "tuple")
d = {'a': 'b', 'b': 2, 'c': False}
e = {1,2,3,4,4,4,4,2,2,2,1}

print(len(a), len(b), len(c), len(d), len(e))
```

```
8 4 2 3 4
```

Для типов, представляющих последовательности (списков, кортежей и строк), можно вырезать (slice) подпоследовательность индексов, используя квадратные скобки и двоеточие, как показано в приведенных ниже примерах. Диапазон индексов является полуоткрытым в том смысле, что вырезка начинается с первого заданного индекса, а заканчивается на втором заданном индексе, не включая его. Отрицательные индексы отсчитываются с конца последовательности. Если первый не указан, то вырезка начинается с индекса 0. При отсутствии второго индекса вырезка продолжается до конца последовательности.

Важное замечание: вырезка из последовательности создает новый объект. Это означает, что крупная вырезка требует большого объема копирования. Без учета этого факта легко написать неэффективный код.

```
a = "a string"
b = ["my", "second", "favorite", "list"]
c = (1, 2, 3, "tuple")
print(a[3:7])
print(a[1:-2])
print(b[1:])
print(c[:2])
```

```
trin
stri
['second', 'favorite', 'list']
(1, 2)
```

2.6. Итерации по наборам данных

Весьма часто необходимо организовать проход в цикле по набору данных. «Питоническим» (pythonic) способом выполнения итераций является цикл `for`.

Синтаксис цикла `for` показан в приведенных ниже примерах.

```
mylist = [1,3,5]
mytuple = (1, 2, 'skip a few', 99, 100)
myset = {'a', 'b', 'z'}
mystring = 'abracadabra'
mydict = {'a': 96, 'b': 97, 'c': 98}

for item in mylist:
    print(item)

for item in mytuple:
    print(item)

for element in myset:
    print(element)

for character in mystring:
    print(character)

for key in mydict:
    print(key)

for key, value in mydict.items():
    print(key, value)

for value in mydict.values():
    print(value)
```

Существует специальный класс `range` для представления последовательности чисел, которая ведет себя как набор данных. Он часто используется в циклах `for`, как показано ниже.

```
for i in range(10):
    j = 10 * i + 1
    print(j, end=' ')

1 11 21 31 41 51 61 71 81 91
```

2.7. Другие формы управления потоком выполнения

Управление потоком выполнения (control flow) использует специализированные команды любого языка программирования, которые влияют на порядок выполнения операций. Циклы `for` из предыдущего раздела являются классическими примерами управления потоком выполнения. Другими основными формами управления потоком выполнения являются инструкции `if`, циклы `while`, блоки `try`/`except` и вызовы функций. Мы кратко рассмотрим каждую из этих форм.

Инструкция `if` в ее простейшей форме вычисляет выражение и пытается интерпретировать его результат как логическое значение. Вычисляемое выражение интерпретируется как предикат (predicate; условное выражение). Если вычисление предиката дает результат `True`, то выполняется блок кода внутри инструкции `if`. Иначе этого не происходит. Это выбор последовательности, выбор («ветви» потока выполнения) и итерация. Ниже приведены примеры.

```
if 3 + 3 < 7:
```

```
    print("This should be printed.")

if 2 ** 8 != 256:
    print("This should not be printed.")

This should be printed.
```

Инструкция `if` также может содержать ветвь `else`. Это второй блок кода, который выполняется, если при вычислении предиката получен результат `False`.

```
if False:
    print("This is bad.")
else:
    print("This will print.")

This will print.
```

В цикле `while` также имеется предикат. Он вычисляется в самом начале блока кода. Если результатом вычисления является `True`, то блок в теле цикла выполняется, затем процесс повторяется до тех пор, пока при вычислении предиката не будет получено значение `False` или в теле цикла не встретится инструкция `break`.

```
x = 1
while x < 128:
    print(x, end=' ')
    x = x * 2

1 2 4 8 16 32 64
```

Блок `try` – это способ перехвата и восстановления при возникновении ошибок во время выполнения программы. Если имеется некоторый код, в котором могут возникать ошибки, но при этом весьма нежелательно аварийное завершение программы, то можно поместить такой код в блок `try`. После этого вы можете перехватить (catch) ошибку (также называемую исключением (exception)) и обработать ее. Простым примером может послужить случай, когда требуется преобразовать некоторое число в тип с плавающей точкой `float`. Объекты многих типов можно преобразовать в тип `float`, но не все. Если вы просто пытаетесь выполнить такое преобразование и оно работает, то все в порядке. Но если возникает ошибка (исключение) `ValueError`, то есть возможность выполнить некоторые другие действия (в блоке `except`).

```
x = "not a number"
try:
    f = float(x)
except ValueError:
    print("You can't do that!")

You can't do that!
```

Любая функция также вносит изменение в управление потоком выполнения. В Python функция определяется с помощью ключевого слова `def`. Оно создает объект для хранения блока кода. Параметры для функции указываются в виде списка в круглых скобках после имени функции. Инструк-

ция `return` возвращает управление потоком выполнения в то место, где была вызвана функция, а также определяет значение как результат вызова функции.

```
def foo(x, y):
    return 8 * x + y

print(foo(2, 1))
print(foo("Na", " batman"))
```

```
17
NaNaNaNaNaNaNaNa batman
```

Обратите внимание: здесь не требуется указание типов объектов, ожидаемых функцией в качестве аргументов. Это очень удобно, потому что можно использовать функцию для обработки различных типов объектов (как показано в приведенном выше примере). Если функция определяется дважды, даже если изменены ее параметры, то вторая функция полностью замещает первую (т. е. первая функция становится недоступной). Это в точности то же самое, что и двукратное присваивание значения переменной. Имя функции – это всего лишь имя, оно указывает (ссылается) на некоторый объект (функцию в данном случае). С функциями можно обращаться как с любыми другими объектами.

```
def foo(x):
    return x + 2

def bar(somefunction):
    return somefunction(4)

print(bar(foo))
somevariable = foo
print(bar(somevariable))
```

```
6
6
```

2.8. Модули и импортирование

Когда мы переходим к написанию более сложных программ, имеет смысл разместить исходный код в нескольких файлах. Отдельный файл с расширением *.py* называется модулем (module). Один модуль можно импортировать в другой, используя ключевое слово `import`. По умолчанию имя модуля совпадает с именем файла (без расширения *.py*). При импорте модуля его код становится выполняемым. Обычно содержимое модуля должно ограничиваться определениями некоторых функций и классов, но с технической точки зрения он может содержать все, что угодно. Кроме того, в модуле имеется пространство имен (namespace), в котором определены его функции и классы.

Например, предположим, что имеются следующие файлы.

```
# Файл: twofunctions.py

def f(x):
    return 2 * x + 3

def g(x):
    return x ** 2 - 1

# Файл: theimporter.py

import twofunctions

def f(x):
    return x - 1

print(twofunctions.f(1))        # Выводит 5.
print(f(1))                     # Выводит 0.
print(twofunctions.g(4))        # Выводит 15.
```

Инструкция `import` переносит имя модуля в текущее пространство имен. После этого можно использовать имя модуля для идентификации имен функций из него.

В инструкции импорта присутствует небольшая доля магии. В некотором смысле она всего лишь сообщает текущей программе о результатах работы другой программы. Поскольку результатом импорта (обычно) становится выполнение указанного модуля, правильным практическим приемом является изменение поведения скрипта, зависящее от того, запускается ли он напрямую или как импортируемая часть. Это можно проверить, рассматривая атрибут модуля `__name__`. Если модуль запускается напрямую (т. е. как автономный скрипт), то для переменной (атрибута) `__name__` автоматически устанавливается значение `__main__`. Если же модуль импортируется, то в атрибуте `__name__` по умолчанию остается его имя. Все это легко увидеть при проведении следующего эксперимента.

```
# Файл: mymodule.py
print("The name of this module is", __name__)

The name of this module is __main__

# Файл: theimporter.py

import mymodule
print("Notice that it will print something different when imported?")
```

Здесь показано, как можно использовать атрибут `__name__`, чтобы проверить, в каком режиме выполняется конкретная программа.

```
def somefunction():
    print("Real important stuff here.")

if __name__ == '__main__':
    somefunction()

Real important stuff here.
```

В приведенном выше коде сообщение выводится только в том случае, если модуль выполняется как скрипт. Сообщение не выводится (т. е. функция `somefunction` не вызывается), если модуль был импортирован. Это весьма часто используемая характерная особенность (идиома) языка Python.

Необходимо всегда помнить о том, что модули выполняются только один раз при первом импортировании. Например, если импортировать один и тот же модуль дважды, то он будет выполнен только один раз. После этого пространство имен такого модуля существует и доступно для второго варианта импорта. Это также позволяет избежать любых бесконечных циклов при возможных попытках создания двух модулей, каждый из которых импортирует другой.

Существует несколько других часто применяемых вариантов стандартной инструкции `import`.

1. Можно импортировать только конкретное имя или набор имен из модуля: `from <имя_модуля> import <нужное_имя>`. При этом новое имя `<нужное_имя>` переносится в текущее пространство имен. После этого не требуется префикс `<имя_модуля>` с точкой перед импортированным именем.
2. Можно импортировать все имена из модуля в текущее пространство имен: `from <имя_модуля> import *`. После этого каждое имя, определенное в указанном модуле, будет доступно в текущем пространстве имен, и также не потребуется префикс `<имя_модуля>` с точкой перед импортированными именами. Этот вариант легко и быстро пишется во многих случаях, но, как правило, такой подход не рекомендуется, потому что чаще всего точно не известно, какие именно имена импортируются.
3. Можно переименовать модуль при импортировании: `import numpy as np`. Это позволяет использовать другое (как правило, более короткое) имя для обращения к объектам импортируемого модуля. В приведенном выше примере можно написать `np.array` вместо `numpy.array`. Чаще всего причиной для переименования является получение более короткого имени. Реже применяется переименование для устранения конфликта имен.

Глава 3

Объектно-ориентированное программирование

Главная цель объектно-ориентированного программирования (object-oriented programming) – предоставление возможности написания исходного кода, наиболее близкого к образу мышления относительно тех сущностей, которые представляет этот код.

Класс (class) – это тип данных. В Python тип (type) и класс (в большинстве случаев) являются синонимами. Объект (object) – это экземпляр (instance) класса. Например, в Python есть класс списка `list`. Если я создаю список с именем `mylist`, то `mylist` – это объект типа `list`.

```
mylist = []
print(type(mylist))
print(isinstance(mylist, list))
print(isinstance(mylist, str))
```

```
<class 'list'>
True
False
```

В Python встроены все виды классов. Некоторые из них оказываются неожиданными для пользователя.

```
def foo():
    return 0

print(type(foo))
```

```
<class 'function'>
```

Для продвинутых студентов ниже приводится более экзотический пример под названием «генератор». В Python вместо `return` можно использовать инструкцию `yield`. В этом случае результатом будет некоторый объект, называемый генератором, а не функцией. Эта мощная идея часто встречается в Python, но мы не сможем осмыслить ее, пока не поймем, как в классы можно упаковать данные и код.

```
def mygenerator(n):
    for i in range(n):
        yield i

print(type(mygenerator))
print(type(mygenerator(5)))
```

```
<class 'function'>
<class 'generator'>
```

3.1. Простой пример

Одним из первых способов изучения методики объединения нескольких фрагментов информации в единый объект является обращение к математическому анализу или линейной алгебре, а именно к представлению векторов. Двумерный вектор можно мысленно представить как пару чисел. Если попытаться написать некоторый код, работающий с двумерными векторами, то можно было бы просто воспользоваться кортежами. Не составляет особого труда определить несколько простых функций, которые работают с векторами.

```
u = (3,4)
v = (3,6)

def add(a, b):
    return (a[0] + b[0], a[1] + b[1])

def subtract(a,b):
    return (a[0] - b[0], a[1] - b[1])

def dot(a, b):
    return (a[0] * b[0] + a[1] * b[1])

def norm(a):
    return (a[0] * a[0] + a[1] * a[1]) ** 0.5

def isvertical(a):
    return a[0] == 0

print(norm(u))
print(add(u,v))
print(u + v)
print(isvertical(subtract(v, u)))
```

```
5.0
(6, 10)
(3, 4, 3, 6)
True
```

Это было бы неплохим решением, если бы им ограничивались все наши требования, но при дальнейшем написании кода все становилось бы более запутанным. Например, предположим, что необходимо убедиться в том, что входными данными для этих функций действительно являются кортежи, содержащие два числа. Можно добавить соответствующий код в каждую функцию

для обнаружения подобных ошибок и восстановления при их возникновении, но это не самое лучшее решение, потому что в действительности требуются только операции с векторами. Вероятно, для этого потребовалось бы сделать дополнительную функцию намного более сложной или дать ей несколько более описательное имя, например `vectoradd`.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

u = Vector(3,4)

print(u.norm())
print(Vector(5,12).norm())

5.0
13.0
```

Функция, определенная в классе, называется методом (method). Существует стандартное соглашение использовать `self` в качестве имени первого параметра каждого метода. Этим параметром является объект, с которым должны выполняться операции в коде конкретного метода. При вызове метода не обязательно в явной форме передавать параметр `self`. Он формируется с помощью точечной записи (записи через точку). То есть `u.norm()` преобразуется в `Vector.norm(u)`.

Метод `__init__` называется инициализатором (initializer)[1]. Имена методов, подобных этому, начинаются и заканчиваются двумя символами подчеркивания, и иногда их называют «магическими методами» (magic methods) или «методами с двойным подчеркиванием» (dunder methods; dunder = **d**ouble **under**score – двойное подчеркивание). Не рекомендуется создавать собственные методы с именами, начинающимися и заканчивающимися двумя символами двойного подчеркивания, потому что именно таким способом Python отличает эти специализированные методы от прочих, и не следует вносить путаницу, называя в таком же стиле собственные методы. Кроме того, методы с двойным подчеркиванием обычно не вызываются явно, вместо этого предоставляются некоторые другие средства обращения к ним. В случае с инициализатором запись имени класса как функции вызывает инициализатор этого класса. Вы видели это и раньше, например в выражении `float("3.14159")`.

Для реализации операции сложения векторов мы используем еще один специальный магический метод.

[1] В объектно-ориентированном программировании чаще используется термин «конструктор» (constructor) для обозначения такого метода. – *Прим. перев.*

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def __add__(self, other):
        newx = self.x + other.x
        newy = self.y + other.y
        return Vector(newx, newy)

u = Vector(3,4)
v = Vector(3,6)

print(u + v)
```

```
<__main__.Vector object at 0x10db7bcd0>
```

Этот код выводит довольно странный результат. Он сообщает, что u + v – это объект вектора, размещенный по некоторому адресу памяти, но ничего не говорит о том, что на самом деле представляет собой этот вектор. Необходимо реализовать метод `__str__`, чтобы вывести сам вектор (его значение). Этот магический метод вызывается функцией `print` для преобразования его параметров в строку. Далеко не всегда понятно, как должна выводиться строка для конкретного класса. Мы должны сами определить способ вывода.

В приведенном ниже примере добавлен метод `__add__`, а также некоторый код проверки типа входных данных. Такой подход гарантирует, что результатом будет вектор, содержащий два числа с плавающей точкой как координаты.

```
class Vector:
    def __init__(self, x, y):
        try:
            self.x = float(x)
            self.y = float(y)
        except ValueError:
            self.x = 0.0
            self.y = 0.0

def norm(self):
    return (self.x ** 2 + self.y ** 2) ** 0.5

def __add__(self, other):
    newx = self.x + other.x
    newy = self.y + other.y
    return Vector(newx, newy)

def __str__(self):
    return "(%f, %f)" %(self.x, self.y)

u = Vector(3,4)
v = Vector(3,6)

print(u + v)
```

```
(6.000000, 10.000000)
```

3.2. Инкапсуляция и открытый (общедоступный) интерфейс класса

Термин «инкапсуляция» (encapsulation) имеет два различных, но взаимосвязанных значения. Во-первых, он обозначает объединение в одно целое данных и методов, которые работают с этими данными. В Python это выполняется через классы, как вы уже видели.

Второй смысл инкапсуляции особо выделяет границу между внутренним содержимым класса и всем, что находится вне его, и определяет, что должно быть видимым для пользователей класса. Часто это означает разделение атрибутов на открытые (общедоступные) (public) и закрытые (недоступные извне) (private). В Python не существует формального механизма защиты от доступа извне к атрибутам внутри класса. Так что в этом смысле все в классе является открытым. Но существует соглашение, которое уточняет, что в классе должно считаться закрытым. Любой атрибут, имя которого начинается с символа подчеркивания, считается закрытым. Воспринимайте это как аналогию с открытым (чужим) дневником. Вы можете читать его, но не должны этого делать.

```
class Diary:
    def __init__(self, title):
        self.title = title
        self._entries = []

    def addentry(self, entry):
        self._entries.append(entry)

    def _lastentry(self):
        return self._entries[-1]
```

В приведенном выше примере метод `addentry` является открытым. Каждый может добавить запись. Но метод `_lastentry` закрытый. Никто не должен вызывать его, находясь вне класса `Diary`. (Напомню еще раз: вы можете это сделать, но не должны так поступать.) Атрибут `title` также открытый, но список `_entries` закрытый. Набор всех открытых атрибутов (в данном примере это `addentry` и `title`) составляет открытый интерфейс (public interface) класса. Это та часть, с которой должен взаимодействовать пользователь данного класса. Например, любой пользователь может воспользоваться определенным выше классом следующим образом:

```
mydiary = Diary("Don't read this!!!")
mydiary.addentry("It was a good day.")
print("The diary is called ", mydiary.title)
```

```
The diary is called Don't read this!!!
```

Следует отметить, что в этом примере инкапсуляция класса не означает его защиту. Черт побери, если это мой дневник, то я должен иметь возможность читать его, не так ли? В действительности причина для соблюдения правила для закрытых атрибутов и использования только открытого интерфейса за-

ключается в том, чтобы помочь нам написать работающий код, который будет продолжать работать и в будущем. Исходный код изменяется со временем. Если вы изменяете класс, который используется во многих местах исходного кода, то должны быть осторожными и внимательными, чтобы не нарушить работу этого кода. Если открытый интерфейс и поведение класса не изменяется, то можно быть вполне уверенным в том, что изменения внутри класса не влияют на остальной код. Можно изменить имя закрытой переменной, например `_entries` на `_diaryentries`, и быть уверенным в том, что это изменение не нарушит работу в каком-то другом месте кода.

3.3. Наследование и отношение «является» (is a)

Когда мы говорим о типах объектов в повседневной жизни, можно подразумевать различные уровни обобщенности. Можно говорить о конкретном баскетболисте, например о Кайри Ирвинге (Kyrie Irving), или о профессиональных баскетболистах, или обо всех игроках в баскетбол, или о людях, или о живых существах. Можно сказать, что конкретный игрок, с которого мы начали обсуждение, принадлежит к любому из этих классов. Тот же принцип применим и к коду, который мы пишем.

Рассмотрим аналогичный пример из геометрической программы.

```
class Triangle:
    def __init__(self, points):
        self._sides = 3
        self._points = list(points)
        if len(self._points) != 3:
            raise ValueError("Wrong number of points.")

    def sides(self):
        return 3

    def __str__(self):
        return "I'm a triangle."

class Square:
    def __init__(self, points):
        self._sides = 4
        self._points = list(points)
        if len(self._points) != 4:
            raise ValueError("Wrong number of points.")

    def sides(self):
        return 4

    def __str__(self):
        return "I'm so square."
```

Очевидно, что это очень тесно связанные классы. Можно создать еще один класс, по отношению к которому оба эти класса будут являться подклассами (subclasses). Тогда все общие свойства обоих классов можно будет поместить в более объемлющий класс или суперкласс (superclass).

```
class Polygon:
    def __init__(self, sides, points):
        self._sides = sides
        self._points = list(points)
        if len(self._points) != self._sides:
            raise ValueError("Wrong number of points.")

    def sides(self):
        return self._sides

class Triangle(Polygon):
    def __init__(self, points):
        Polygon.__init__(self, 3, points)

    def __str__(self):
        return "I'm a triangle."

class Square(Polygon):
    def __init__(self, points):
        Polygon.__init__(self, 4, points)

    def __str__(self):
        return "I'm so square."
```

Обратите внимание: теперь в определения классов `Triangle` и `Square` включено имя класса `Polygon` в круглых скобках. Это называется наследованием (inheritance). Класс `Triangle` наследуется от (inherits from) класса `Polygon` (или расширяет (extends) этот класс). Таким образом, имеем суперкласс (superclass) `Polygon` и подклассы (subclasses) `Triangle` и `Square`. При вызове метода из объекта, если этот метод не определен в классе данного объекта, Python будет искать требуемый метод в суперклассе. Такой поиск корректной вызываемой функции называется порядком разрешения методов (method resolution order). Если метод суперкласса переопределен в подклассе, то при обращении к этому методу из экземпляра подкласса вызывается метод подкласса (а не суперкласса).

Инициализатор суперкласса не вызывается автоматически при создании нового экземпляра подкласса (если в подклассе не был определен метод `__init__`). В этом случае мы вручную вызываем метод `Polygon.__init__`. Это один из немногих случаев, когда допустимо в явной форме вызывать по имени специальный метод с подчеркиваниями.

При использовании наследования всегда нужно помнить самое важное его правило:

Наследование означает «является экземпляром» (is a).

Это значит, что если `ClassB` расширяет `ClassA`, то объект `ClassB` является экземпляром (is a) объекта `ClassA`. Это должно быть истинным на теоретическом концептуальном уровне. В рассмотренном выше примере с геометрическими фигурами мы соблюдаем это правило, поскольку треугольник является многоугольником (экземпляром многоугольника).

В общем случае это может выглядеть как увеличение объема исходного кода. Но при этом уменьшается дублирование. Дублирование – это весьма плохое

явление. Даже притом, что легко скопировать и вставить фрагмент кода, применяя возможности компьютера, дублирование является источником множества ошибок. Причина проста. Ошибки (bugs) везде. Если вы копируете и вставляете код с ошибкой, то получаете две. При поиске такой ошибки вы должны надеяться, что помните о необходимости ее исправления в двух местах. И всякий раз, когда приходится полагаться на собственную память, вы испытываете сильное беспокойство.

Инженеры программного обеспечения используют акроним DRY (читается: «драй»), означающий *Don't Repeat Yourself* («не повторяйся»; менее точно, но более правильно: «не повторяй своих ошибок»). Они даже применяют этот акроним как прилагательное, когда говорят: «Keep the code DRY» («Сохраняй код (как порох) сухим»). Процесс устранения дублирования посредством помещения общего кода в суперкласс называется вынесением в суперкласс (factoring out a superclass). Это самый распространенный способ, которым наследование вводится в кодовую базу. Иногда возможности для наследования определяются на этапе проектирования, т. е. до начала написания исходного кода.

3.4. Утиная типизация

Наследование считается одним из основных элементов объектно-ориентированного программирования, поэтому важно хорошо понимать этот механизм. И все же наследование не так полезно в Python, как в других языках программирования. Причина в том, что в Python встроен (параметрический) полиморфизм (polymorphism). Это означает, что мы можем передавать в функцию объект любого требуемого типа. Например, предположим, что имеется класс для хранения наборов многоугольников, показанный ниже.

```
class PolygonCollection:
    def __init__(self):
        self._triangles = []
        self._squares = []

    def add(self, polygon):
        if polygon.sides() == 3:
            self._triangles.append(polygon)
        if polygon.sides() == 4:
            self._squares.append(polygon)
```

Обратите внимание: метод `add` должен работать одинаково успешно с любой версией классов `Triangle` и `Square`, определенных ранее. В действительности мы можем передавать в метод `add` любой объект, в котором есть метод с именем `sides`. Нет никакой необходимости в наследовании для того, чтобы интерпретировать треугольники и квадраты как особые случаи некоторого объекта. Иногда наследование является правильным способом комбинирования классов с целью их интерпретации как единого класса, но в Python это не настолько необходимо, как в других языках.

Полиморфизм Python основан на концепции утиной типизации (duck typing). Это название происходит от старого выражения: если что-то ходит как

утка и крякает как утка, то это и есть утка[1]. В примере `PolygonCollection` если вызывается метод `add` с аргументом, содержащим метод `sides`, который возвращает что-либо, сравнимое с целым числом, то код будет выполняться без ошибок. Наличие корректных методов равнозначно признаку «плавает и крякает как утка». И хотя наследование всегда должно означать «является экземпляром» (is a), тем не менее не каждое отношение «is a» в вашем коде необходимо выражать с помощью наследования.

Эта идея, заключающаяся в том, что не только наследование означает «является экземпляром» (is a), весьма важна в Python и не менее важна на протяжении всего этого курса. Мы продолжим более глубокое обсуждение этой темы, когда будем рассматривать абстрактные типы данных.

Один из подобных примеров мы уже наблюдали ранее – это функция `str`. Объекты различных типов можно преобразовать в строки, в том числе объекты классов, которые мы сами определили. Если в нашем классе реализован метод `__str__`, то можно вызывать функцию `str` для любого экземпляра такого класса. Функция `str` вызывает соответствующий метод, например `str(t)` для объекта `Triangle t` вызывает метод `t.__str__()`, что равнозначно вызову `Triangle.__str__(t)`.

3.5. Композиция и отношения «содержит» (has a)

Во многих случаях необходимо, чтобы объекты различных типов совместно использовали некоторую функциональность. Иногда для обеспечения такого совместного использования применяется наследование, но чаще мы предпочитаем методику под названием «композиция» (composition). По этой методике некоторый класс хранит экземпляр другого класса. Это позволяет создавать все более и более сложные объекты. Самое важное правило композиции приведено ниже.

Композиция означает «содержит экземпляр» (has a).

Рассмотрим случай, когда необходим класс, который ведет себя как список (`list`). Например, потребуется возможность добавления элементов в этот список и получения доступа к элементам по индексу, но остальные свойства встроенного списка не нужны. В этом случае использование наследования было бы ошибочным вариантом. Вместо этого следует создать требуемый класс так, чтобы он содержал внутри себя список (это композиция). Далее, открытый интерфейс этого класса должен содержать методы, необходимые при обращениях к хранимому внутри экземпляру списка типа `list`, чтобы избежать дублирования реализации настоящего списка. Ниже приведен пример такой композиции.

[1] Уточнение: утиный тест придумал Джеймс Уиткомб Райли (James Whitcomb Riley): «Если это выглядит как утка, плавает как утка и крякает как утка, то это и есть утка». – *Прим. перев.*

```
class MyLimitedList:
    def __init__(self):
        self._L = []

    def append(self, item):
        self._L.append(item)

    def __getitem__(self, index):
        return self._L[index]
```

В этом примере магический метод `__getitem__` позволяет использовать синтаксис квадратных скобок в нашем классе. Мы не обращаемся к нему напрямую, как и к другим магическим методам. Ниже приведен пример использования класса `MyLimitedList`.

```
L = MyLimitedList()
L.append(1)
L.append(10)
L.append(100)
print(L[2])
```

```
100
```

Глава 4

Тестирование

Python – интерпретируемый язык. Это его свойство дает огромное преимущество в гибкости, например такое как утиная типизация. Но это же свойство может приводить к возникновению разнообразных типов часто встречающихся ошибок. Например, если передается значение типа `float` в функцию, которая в действительности должна принимать только значения типа `int`, то Python вас не остановит, но это может привести к непредсказуемому поведению. В общем случае нужно выполнить код, чтобы увидеть ошибку, но не все ошибки в исходном коде генерируют сообщения об ошибках. Стремясь к написанию корректного кода, мы используем тесты для ответа на два вопроса.

1. Это действительно работает? То есть выполняет ли написанный код именно то, что от него ожидается?
2. Будет ли это работать и в дальнейшем? Вы можете быть полностью уверенными в том, что вносимые изменения не приводят к нарушению нормальной работы другой части кода?

4.1. Написание тестов

Тестирование кода означает написание дополнительного кода, который проверяет, соответствует ли поведение программы (скрипта) вашим ожиданиям. Важное правило:

> Тестируйте поведение, а не реализацию.

У вас есть общее представление о том, что именно должен делать код. Вы запускаете этот код. Сделал ли он то, что вы ожидали? Как он ведет себя при вводе некоторых других данных? В самом простом случае можно просто добавить некоторый код в конец модуля.

```
class Doubler:
    def __init__(self, n):
        self._n = 2 * n

    def n(self):
        return self._n
```

```
if __name__ == '__main__':
    x = Doubler(5)
    assert(x.n() == 10)
    y = Doubler(-4)
    assert(y.n() == -8)
```

Инструкция `assert` генерирует ошибку, если предикат, следующий за ней, дает результат `False`. Иначе программа просто выполняется как обычно. Инструкции `assert` намного полезнее, чем простой вывод значений, потому что нет необходимости вручную сверять выведенные данные с ожидаемыми результатами. Кроме того, люди склонны удалять старые инструкции вывода, чтобы привести в порядок вывод результатов тестирования. Удаление тестов после того, как они успешно пройдены, – очень плохая идея. Исходный код будет изменяться, и вы должны знать, не нарушают ли нормальную работу программы новые вносимые изменения.

Строка `if __name__ == '__main__':` обеспечивает уверенность в том, что тесты не будут запускаться, если данный модуль импортирован из какой-либо другой программы.

У некоторых пользователей при обучении тестированию кода возникает мощный психологический барьер. Они считают, что тестирование кода выявит их недостатки как программистов. Если вы чувствуете хотя бы малейшую нерешительность при тестировании собственного кода, то вам следует применить на практике протокол OGAE – *Oh Good, An Error!* («О, отлично, – ошибка!»). При обнаружении каждой ошибки вы говорите эту фразу с искренним энтузиазмом. Компьютер только что оказал вам огромную услугу, обнаружив что-то некорректное, и сделал это тихо и незаметно в вашей комнате или офисе. Вы можете исправить эту ошибку, прежде чем их станет больше.

4.2. Модульное тестирование с использованием unittest

Простейший вид тестирования в модуле, описанный в предыдущем разделе, хорош для совсем маленьких программ, которые не потребуются снова в будущем, но для чего-то гораздо более серьезного необходимо правильно организованное модульное тестирование (unit tests). В этом словосочетании термин «модульное» (unit) означает единый неделимый вариант. Следовательно, при модульном тестировании предполагается тестирование конкретного поведения конкретной функции. Это означает создание многочисленных тестов, и при любом изменении исходного кода обязательно должны быть выполнены все эти тесты.

Чтобы немного упростить процесс тестирования, предлагается стандартный пакет `unittest`, предназначенный для написания модульных тестов в среде Python. Этот пакет обеспечивает стандартный способ написания тестов, возможность прогона всех их вместе, а также возможность вывода результатов в удобном формате. В современной программной инженерии тесты также выполняются автоматически как часть систем сборки и развертывания программного обеспечения.

Для использования `unittest` необходимо импортировать этот пакет в тестируемый файл. Затем импортировать код, который нужно тестировать. Реальные тесты будут методами в классе, который расширяет стандартный класс `unittest.TestCase`. Имя каждого тестирующего метода обязательно должно

начинаться со слова `test`. При отсутствии такого префикса тест не будет выполняться. Тесты запускаются при вызове метода `unittest.main`.

Ниже приведен пример тестирования конкретного поведения гипотетического класса `DayOfTheWeek`.

```
import unittest
from dayoftheweek import DayOfTheWeek

class TestDayOfTheWeek(unittest.TestCase):
    def testinitwithabbreviation(self):
        d = DayOfTheWeek('F')
        self.assertEquals(d.name(), 'Friday')

        d = DayOfTheWeek('Th')
        self.assertEquals(d.name(), 'Thursday')

unittest.main()
```

Обратите внимание: даже если мы никогда не видели исходный код класса `DayOfTheWeek`, тем не менее мы можем получить вполне правильное представление о его ожидаемом поведении, просто читая тесты. В данном случае мы видим, что можно создать экземпляр этого класса с указанием аббревиатуры `F`, а метод `name()` вернет значение `'Friday'`. Часто случается, что подобные модульные тесты предоставляют весьма точную спецификацию ожидаемого поведения обследуемой структуры данных. Кроме того, поскольку есть возможность выполнить эти тесты, можно быть уверенным в том, что данный класс действительно демонстрирует ожидаемое поведение. При наличии документации иногда обнаруживается, что изменения в коде в ней не отражены, но успешный прогон тестов не создает подобной проблемы.

4.3. Разработка через тестирование

Разработка через тестирование (Test-Driven Development – TDD) основана на простой идее: можно написать тесты до начала написания исходного кода. Но не провалится ли тест, если код еще не написан? Не провалится, если это хороший тест. А что, если тест пройдет? Тогда либо вы уже завершили работу (что маловероятно), либо с вашим тестом что-то не так.

Предварительное написание тестов вынуждает выполнить два действия.

1. Решить, как вы предполагаете использовать некоторую функцию. Какими должны быть параметры? Что она должна возвращать?
2. Написать только тот код, который действительно нужен. Если какой-либо код не поддерживает требуемое поведение при тестах, то нет необходимости писать его.

Для TDD существует мантра (заклинание): Red-Green-Refactor (красный–зеленый–рефакторинг). Она определяет три этапа процесса тестирования:

- красный (red): тесты не проходят. Так и должно быть. Ведь вы еще не написали код;
- зеленый (green): все тесты проходят при изменении кода;
- рефакторинг (refactor): чистка и улучшение кода, устранение дублирования.

Термины «красный» (red) и «зеленый» (green) обозначают множество тестовых рабочих сред (фреймворков), которые определяют непрошедшие тесты как красные, а прошедшие – как зеленые.

Рефакторинг (refactoring) – это процесс чистки кода, чаще всего обозначающий процедуру устранения дублирования. Дублирование в исходном коде, возникающее из-за применения копирования–вставки или просто как следствие повторения логики, может стать источником множества ошибок. Если вы дублируете код с ошибкой, то получаете две ошибки. Если вы нашли одну такую ошибку, то должны отыскать и вторую.

Ниже приведен пример рефакторинга кода.

Начальный исходный код с небольшим дублированием:

```
avg1 = sum(L1)/len(L1)
avg2 = sum(L2)/len(L2)
```

Затем становится очевидно, что здесь должно существовать некоторое поведение по умолчанию для пустых списков, поэтому (добавляется тест и) код обновляется, как показано ниже.

Обновленный код перед рефакторингом:

```
if len(L1) == 0:
    avg1 = 0
else:
    avg1 = sum(L1) / len(L1)

if len(L2) == 0:
    avg2 = 0
else:
    avg2 = sum(L2) / len(L2)
```

Код после рефакторинга:

```
def avg(L):
    if len(L) == 0:
        return 0
    else:
        return sum(L) / len(L)

avg1 = avg(L1)
avg2 = avg(L2)
```

В коде после рефакторинга подробности функции `avg` не дублируются. Если в дальнейшем потребуется изменение способа обработки пустых списков, то изменения нужно будет внести только в одном месте.

Кроме того, код после рефакторинга легче читать.

4.4. Что необходимо тестировать

Отойдите от компьютера. Подумайте о задаче, которую пытаетесь решить, и о методах, которые вы пишете. Спросите себя: что должно произойти, когда я выполняю этот код? И еще: как я предполагаю использовать этот код?

- Напишите тесты, использующие исходный код именно тем способом, которым он должен использоваться в реальной работе.

- Затем напишите тесты, использующие код некорректно, чтобы проверить варианты его критических отказов. Выдаются ли при этом понятные сообщения об ошибках?
- Протестируйте граничные случаи, т. е. коварные случаи, возникающие крайне редко. Попытайтесь нарушить нормальную работу своего кода.
- Внесите ошибки в тесты. Ошибка или некорректное поведение могут возникать снова после их исправления. Необходимо отлавливать их при появлении. Иногда вы замечаете ошибку, когда не проходит другой тест. Напишите специализированный тест для обнаружения такой ошибки и исправьте ее.
- Протестируйте открытый интерфейс. Обычно нет необходимости в тестировании закрытых методов класса. Вы должны интерпретировать тестовый код как пользователя тестируемого класса и не делать никаких предположений о закрытых атрибутах. Если закрытые атрибуты переименовываются или подвергаются рефакторингу, то при таком подходе не нужно будет изменять тесты.

4.5. Тестирование и объектно-ориентированное проектирование

При объектно-ориентированном проектировании исходный код разделяется на классы. Между этими классами существуют определенные отношения, иногда создаваемые наследованием или композицией. Классы содержат открытые методы. Мы называем их (открытым) интерфейсом класса.

В начале процесса проектирования мы рассматриваем задачу и определяем существительные (классы) и глаголы (методы). В описании задачи выражается, что должно произойти. Часто наши предположения выражаются на языке `if...then`, т. е. если (if) вызвать этот метод с такими параметрами, то (then) произойдет вот что. Модульное тестирование (unit testing) кодирует это предположение и позволяет проверить, что действительное поведение кода совпадает с ожидаемым.

При написании класса это помогает сконцентрировать внимание и уменьшить количество вещей, о которых нужно думать, если предположить, что каждый класс работает так, как должен. Мы пытаемся добиться этого, тестируя каждый класс отдельно. Затем, объединяя простые классы в более сложные, мы можем в большей степени быть уверенными в том, что любые обнаруженные ошибки находятся в новом классе, а не скрываются где-то в ранее написанных классах.

Поначалу тщательное тестирование кода может показаться пустой тратой времени. Однако любая небольшая экономия времени на раннем этапе за счет пропуска тестов очень быстро будет израсходована на бесконечные и мучительные часы отладки непротестированного кода. При огромном объеме непротестированного кода каждый раз, когда возникает непредвиденное поведение, ошибка может быть где угодно. Процесс отладки может практически прекратиться. Если вдруг это произойдет с вами: остановитесь. Выберите один фрагмент кода. Протестируйте его. Повторите. При внимательном и систематичном тестировании в целом вам потребуется значительно меньше времени. Тише едешь – дальше будешь.

Глава 5

Анализ во время выполнения

При программировании наша главная цель – написать корректный, эффективный и легко читаемый исходный код. При создании исходного кода для решения какой-либо задачи существует много способов написать корректный код. Если он некорректный, мы отлаживаем его. Если неэффективный, то мы можем не заметить этого при выполнении небольших тестов, потому что проблема становится видимой только при возрастании объема входных данных. Проблемы с эффективностью могут становиться весьма трудными для обнаружения.

Необходимо разработать специализированный словарь для описания эффективности кода. Высказывания типа «этот код быстрый» или «этот код медленный» почти ничего не сообщают нам. В какой степени код является быстрым или медленным? Кроме того, для выполнения одного и того же кода потребуется разное время и различный объем памяти в зависимости от размера входных данных. А на более мощном компьютере код будет выполняться быстрее. Поэтому наши цели более сложные:

> Необходимо дать подробное описание эффективности программы, адаптируемое к различным входным данным и разнообразным компьютерам.

Мы будем стремиться к достижению этих целей с использованием асимптотического анализа (asymptotic analysis). Для разработки этой теории начнем с измерения времени, затрачиваемого на выполнение некоторых программ. С помощью простых экспериментов мы можем наблюдать, как изменяется время выполнения при росте объема входных данных. Например, программа, принимающая список как входные данные, может выполняться медленнее, если этот список содержит миллион элементов, по сравнению со списком из десяти элементов, но, возможно, замедления не произойдет. Это зависит от самой программы.

Далее мы введем схему учета для вычисления стоимости (cost) программы. Это позволит присваивать значения стоимости различным операциям. Стоимость всей программы в целом будет равна сумме стоимостей всех операций, выполняемых в этой программе. Чаще всего стоимость является функцией от размера входных данных, а не фиксированным числом.

Наконец, мы введем некоторый словарь для классификации функций. Это и есть асимптотическая часть асимптотического анализа. Система записи (но-

тация) «О-большое» предоставляет весьма удобный способ группировки этих функций (времени их выполнения) по классам, которые можно с легкостью сравнивать. Таким образом, мы получаем возможность обсуждать и сравнивать эффективность различных алгоритмов и/или программ без необходимости проведения широкомасштабных экспериментов.

Размышления и анализ эффективности программ помогает нам выработать полезные привычки и чувство интуиции, способствующие принятию правильных проектных решений. Цель повышения эффективности применяемых структур данных будет основной мотивацией для внедрения новых структур и новых идей в процессе развития.

5.1. Измерение времени выполнения (тайминг) программ

Сначала рассмотрим некоторые различия во времени выполнения разных функций, которые делают одно и то же. Можно сказать, что эти функции обладают одинаковым поведением или одинаковой семантикой (смыслом).

Ниже приведен код функции, принимающей список как входные данные. Она возвращает `True`, если в списке имеются какие-либо дублирующиеся элементы, иначе возвращается `False`.

```
def duplicates1(L):
    n = len(L)
    for i in range(n):
        for j in range(n):
            if i != j and L[i] == L[j]:
                return True
    return False

assert(duplicates1([1,2,6,3,4,5,6,7,8]))
assert(not duplicates1([1,2,3,4]))
```

Главный вопрос, который мы задаем снова и снова: насколько быстрым является этот код?

Самый простой ответ на этот вопрос – простой запуск программы и измерение времени ее выполнения. Это можно сделать, как показано ниже.

```
import time

for i in range(5):
    n = 1000
    start = time.time()
    duplicates1(list(range(n)))
    timetaken = time.time() - start
    print("Time taken for n = ", n, ": ", timetaken)
```

```
Time taken for n = 1000 : 0.08032798767089844
Time taken for n = 1000 : 0.07732391357421875
Time taken for n = 1000 : 0.07471418380737305
Time taken for n = 1000 : 0.07387709617614746
Time taken for n = 1000 : 0.07915425300598145
```

Обратите внимание: в значениях времени выполнения наблюдаются небольшие отклонения. Причина этого – многочисленные факторы, но главным является то, что компьютер одновременно выполняет множество других задач: функции операционной системы и несколько других программ. Кроме того, при запуске на различных компьютерах можно ожидать совершенно другие результаты в зависимости от скорости процессора и прочих различий между компьютерами. Сейчас мы выполним этот код несколько раз и вычислим среднее значение времени его выполнения на конкретном компьютере. Воспользуемся как оберткой более обобщенной функцией, которая принимает как входные данные другую функцию и длину списка. Эта функция-обертка выполняет переданную функцию заданное число раз и вычисляет среднее время по всем сеансам выполнения.

```
import time

def timetrials(func, n, trials = 10):
    totaltime = 0
    #start = time.time()
    for i in range(trials):
        start = time.time() # Таймер должен запускаться здесь.
        func(list(range(n)))
        totaltime += time.time() - start
    print("average =%10.7f for n = %d" % (totaltime/trials, n))
```

Теперь можно наблюдать за изменением среднего времени выполнения по мере возрастания длины исходного списка входных данных. Неудивительно, что среднее время возрастает при увеличении длины списка `n`.

```
for n in [50, 100, 200, 400, 800, 1600, 3200]:
    timetrials(duplicates1, n)

average = 0.0001775 for n = 50
average = 0.0007093 for n = 100
average = 0.0032800 for n = 200
average = 0.0119008 for n = 400
average = 0.0460668 for n = 800
average = 0.1955777 for n = 1600
average = 0.7521353 for n = 3200
```

Попробуем сделать наш код более быстрым. Простым улучшением может быть устранение ситуаций, в которых выполняется ненужная или избыточная работа. В функции duplicates1 мы сравниваем каждую пару элементов дважды, потому что обе переменные `i` и `j` проходят по диапазону всех `n` индексов. Это можно исправить, применив стандартный прием, который позволяет ограничить диапазон `j` на текущей итерации только значением переменной `i`. Соответствующий код показан ниже.

```
def duplicates2(L):
    n = len(L)
    for i in range(1,n):
        for j in range(i):
            if L[i] == L[j]:
                return True
    return False
```

```
for n in [50, 100, 200, 400, 800, 1600, 3200]:
    timetrials(duplicates2, n)

average = 0.0000728 for n = 50
average = 0.0002818 for n = 100
average = 0.0010383 for n = 200
average = 0.0043344 for n = 400
average = 0.0175459 for n = 800
average = 0.0715153 for n = 1600
average = 0.2934786 for n = 3200
```

В Python существует возможность сокращенной записи того типа цикла, который использовался в функции `duplicates2`. Функция `any` принимает итерируемый набор логических значений и возвращает `True`, если каждое из этих логических значений истинно. Можно создать такой итерируемый набор тем же способом, который применяется для охвата всего диапазона значений. Этот прием может оказаться весьма полезным.

```
def duplicates3(L):
    n = len(L)
    return any(L[i] == L[j] for i in range(1,n) for j in range(i))

for n in [50, 100, 200, 400, 800, 1600, 3200]:
    timetrials(duplicates3, n)

average = 0.0001031 for n = 50
average = 0.0004371 for n = 100
average = 0.0017802 for n = 200
average = 0.0064625 for n = 400
average = 0.0268454 for n = 800
average = 0.1103599 for n = 1600
average = 0.4434901 for n = 3200
```

Последняя оптимизация сократила число строк кода и, возможно, оказалась полезной для улучшения его читабельности, но не увеличила скорость.

Если необходимо действительное улучшение, то для этого требуется существенно новая идея. Одной из таких возможностей является сортировка списка и обработка его смежных (соседних) элементов. После сортировки дублирующиеся элементы будут смежными (находящимися рядом друг с другом).

```
def duplicates4(L):
    n = len(L)
    L.sort()
    for i in range(n-1):
        if L[i] == L[i+1]:
            return True
    return False

def duplicates5(L):
    n = len(L)
    L.sort()
    return any(L[i] == L[i+1] for i in range(n-1))
```

```
def duplicates6(L):
    s = set()
    for e in L:
        if e in s:
            return True
        s.add(e)
        return False

def duplicates7(L):
    return len(L) != len(set(L))

def duplicates8(L):
    s = set()
    return any(e in s or s.add(e) for e in L)

for n in [50, 100, 200, 400, 800, 1600, 3200]:
    print("Quadratic: ", end="")
    timetrials(duplicates3, n)
    print("Sorting: ", end="")
    timetrials(duplicates5, n)
    print("Sets: ", end="")
    timetrials(duplicates7, n)
    print('--------------------------')
```

```
Quadratic: average = 0.0001001 for n = 50
Sorting: average = 0.0000055 for n = 50
Sets: average = 0.0000018 for n = 50
--------------------------
Quadratic: average = 0.0004027 for n = 100
Sorting: average = 0.0000100 for n = 100
Sets: average = 0.0000031 for n = 100
--------------------------
Quadratic: average = 0.0014382 for n = 200
Sorting: average = 0.0000191 for n = 200
Sets: average = 0.0000044 for n = 200
--------------------------
Quadratic: average = 0.0063662 for n = 400
Sorting: average = 0.0000450 for n = 400
Sets: average = 0.0000120 for n = 400
--------------------------
Quadratic: average = 0.0257344 for n = 800
Sorting: average = 0.0000934 for n = 800
Sets: average = 0.0000205 for n = 800
--------------------------
Quadratic: average = 0.1045543 for n = 1600
Sorting: average = 0.0002533 for n = 1600
Sets: average = 0.0000495 for n = 1600
--------------------------
Quadratic: average = 0.4281086 for n = 3200
Sorting: average = 0.0003976 for n = 3200
Sets: average = 0.0000858 for n = 3200
--------------------------
```

Некоторые основные выводы: выбор структуры данных действительно имеет большое значение. Скорость проверки на присутствие элемента в множестве или скорость создания множества дает существенное улучшение. Мы делали об элементах в списке предположения, являются ли они сравнимыми или хешируемыми. Это можно интерпретировать как предположения о типе элементов. Но это не то же самое, что класс. Это утиная типизация. Предположение заключается в том, что можно вызывать конкретные методы. Кроме того, приведенный выше пример дает представление о некоторых практических приемах использования выражений генераторов.

Основная и самая важная идея заключается в том, что время выполнения зависит от размера входных данных. Время выполнения возрастает при увеличении длины входных данных. Иногда разрыв (во времени выполнения) между двумя фрагментами кода увеличивается при росте размера входных данных.

5.2. Пример: сложение первых k чисел

Ниже приведена программа сложения первых k положительных целых чисел, которая возвращает сумму и время, затраченное на вычисления.

```
import time

def sumk(k):
    start = time.time()

    total = 0
    for i in range(k+1):
        total = total + i
    end = time.time()

    return total, end-start

for i in range(5):
    print("Sum: %d, time taken: %f" % sumk(10000))
```

```
Sum: 50005000, time taken: 0.000438
Sum: 50005000, time taken: 0.000430
Sum: 50005000, time taken: 0.000433
Sum: 50005000, time taken: 0.000431
Sum: 50005000, time taken: 0.000552
```

Обратите внимание: в значениях времени выполнения наблюдаются небольшие отклонения. Причина этого – многочисленные факторы, но главным является то, что компьютер одновременно выполняет множество других задач: функции операционной системы и несколько других программ. Кроме того, при запуске на различных компьютерах можно ожидать совершенно другие результаты в зависимости от скорости процессора и прочих различий между компьютерами. Сейчас мы выполним этот код несколько раз и вычислим среднее значение времени его выполнения на конкретном компьютере.

```
def timetrials(func, k, trials = 10):
    totaltime = 0
    for i in range(trials):
        totaltime += func(k)[1]
    print("average =%10.7f for k = %d" % (totaltime/trials, k))

timetrials(sumk, 10000)
timetrials(sumk, 100000)
timetrials(sumk, 1000000)
timetrials(sumk, 10000000)

average = 0.0004293 for k = 10000
average = 0.0047173 for k = 100000
average = 0.0466492 for k = 1000000
average = 0.4704423 for k = 10000000
```

Наблюдение за временем выполнения при различных значениях k позволяет обнаружить достаточно предсказуемую закономерность. Когда значение k увеличивается с коэффициентом 10, время, требуемое для выполнения sumk, также увеличивается с коэффициентом 10. Это вполне объяснимо, поскольку необходимо выполнить приблизительно k сложений и присваиваний. Другими словами, время выполнения пропорционально k. Чаще всего нас будет в большей степени интересовать определение того, чему пропорционально время выполнения, нежели определение самого точного времени выполнения. Желательно определить это отношение между временем выполнения и размером входных данных k вне зависимости от того, на каком компьютере выполняется код.

Показанный выше код выглядит корректным, но существует другой, более простой способ вычисления суммы чисел от 1 до k с использованием формулы, которая весьма важна для этого учебного курса. Мы будем пользоваться этой формулой многократно. Для доказательства того, что

$$\sum_{i=1}^{k} i = 1+2+3+\ldots+k = k(k+1)/2,$$

достаточно заметить, что можно суммировать числа парами, объединяя i с $k - i + 1$, начиная с 1 и k. Существует $k/2$ таких пар, и каждая пара суммируется $(k + 1)$ раз. Воспользуемся этой формулой, чтобы переписать функцию sumk и измерить время ее выполнения.

```
import time

def sumk2(k):
    start = time.time()
    total = (k*(k+1)//2)
    end = time.time()
    return total, end-start

timetrials(sumk2, 10000)
timetrials(sumk2, 100000)
timetrials(sumk2, 1000000)
timetrials(sumk2, 10000000)
timetrials(sumk2, 100000000)
```

```
average = 0.0000003 for k = 10000
average = 0.0000003 for k = 100000
average = 0.0000001 for k = 1000000
average = 0.0000002 for k = 10000000
average = 0.0000001 for k = 100000000
```

Эта версия выполняется гораздо быстрее. Даже если число k становится весьма большим, какого-либо заметного замедления не наблюдается.

5.3. Моделирование времени выполнения программы

В этом разделе будет представлена общая методика описания и вычисления итогового числа операций, требуемых для выполнения некоторого фрагмента кода, будь то одна строка, функция или вся программа в целом. По ходу дела мы разработаем словарь терминов для сравнения эффективности алгоритмов, и для этого не потребуется выполнение программ и измерение времени их выполнения.

Недостаточно подсчитать число строк исходного кода. Одна строка кода может выполнять много действий. Ниже показана однострочная функция, которая выполняет все требуемые действия. Она создает список из 200 элементов, суммирует все элементы для каждого значения i от 0 до $k - 1$ и возвращает список результатов.

```
def f001(k):
    return [sum([i, i + 1] * 100) for i in range(k)]

print(f001(9))
```

```
[100, 300, 500, 700, 900, 1100, 1300, 1500, 1700]
```

Вместо простого подсчета строк необходимо более тщательно считать выполняемые операции. Единицей измерения, которую мы будем использовать для описания времени выполнения алгоритма, является число элементарных (атомарных) операций. Это определенно не единица времени, но на некотором уровне (абстракции) все элементарные операции, которые мы будем описывать, могут быть выполнены за небольшое число тактовых циклов центрального процессора (ЦП), поэтому соответствуют реальному значению времени.

> **Примечание.** Нельзя использовать термин **runtime** (интервал времени, в который выполняется конкретная программа) как замену термина **running time** (время выполнения программы/алгоритма как мера ее/его эффективности). Это не одинаковые понятия.

К элементарным операциям относятся: арифметические и логические операции, присваивание значений переменным, доступ к значению переменной по ее имени, ветвление (переход к другой части кода в инструкциях `if`/`for`/`while`), вызов функции и возврат из функции.

В следующих подразделах приведены табл. 5.1, 5.2 и 5.3, содержащие асимптотическое (приблизительное) время выполнения для большинства наиболее

часто применяемых операций для стандартных классов наборов данных языка Python (списки, словари, множества). Вы должны познакомиться во всех подробностях с этими наборами данных. В частности, следует точно знать, какие операции в наборе данных создают его новую копию. Например, объединение (конкатенация) и вырезка создают новый набор данных, следовательно, время выполнения этих операций пропорционально длине этого нового создаваемого набора данных. Почти во всех случаях можно обнаружить причину именно такого времени выполнения, если понять, какую работу должны выполнять алгоритмы, а также как структура данных размещается в памяти.

5.3.1. Операции со списком

Таблица 5.1. Операции со списком

Название операции	Код	Стоимость
Доступ по индексу	`L[i]`	1
Присваивание по индексу	`L[i] = новое_значение`	1
Добавление	`L.append(новый_элемент)`	1
Извлечение (из конца списка)	`L.pop()`	1
Извлечение (по индексу `i`)	`L.pop(i)`	$n - i$
Вставка по индексу `i`	`insert(i, новый_элемент)`	$n - i$
Удаление элемента (по индексу `i`)	`del(элемент)`	$n - i$
Проверка присутствия в списке	`элемент in L`	n
Вырезка	`L[a:b]`	$b - a$
Объединение двух списков	`L1 + L2`	$n_1 + n_2$
Сортировка	`L.sort`	$n \log_2 n$

Следует отметить, что указанные в табл. 5.1 значения времени выполнения одинаковы для всех прочих последовательных наборов данных, `list`[1] и `str`, с предположением, что такие операции существуют для этих неизменяемых типов. Например, для них работают операции доступа по индексу, проверка присутствия в наборе, вырезка и объединение. Напомню, что вырезка и объединение (конкатенация) создают новые объекты и не изменяют исходные.

5.3.2. Операции со словарем

В отличие от операций со списком стоимости операций со словарем (`dict`) выглядят немного загадочными. Некоторые из них кажутся совершенно невоз-

[1] Вероятно, автор здесь имел в виду тип `tuple` (кортеж), который является неизменяемым, поскольку `list` (список) – изменяемый тип набора данных. – *Прим. перев.*

можными. Действительно ли требуется всего лишь одна элементарная операция для проверки того, находится ли данный элемент в наборе из миллиарда элементов? Здесь необходимо запомнить три описанных ниже факта.

1. Мы будем подробно рассматривать, как реализованы словари и как в них используется одна из самых замечательных и полезных идей информатики.
2. Это всего лишь модель, но полезная и точная.
3. Действительная стоимость – это в некотором роде средняя величина. Иногда операция может занять больше времени.

Таблица 5.2. Операции со словарем

Название операции	Код	Стоимость
Извлечение элемента (значения)	`D[ключ]`	1
Установка элемента (значения)	`D[ключ] = значение`	1
Проверка наличия ключа	`ключ in D`	1
Удаление элемента (пары) по ее ключу	`del D[ключ]`	1

5.3.3. Операции с множеством

Множество (`set`) очень похоже на словарь (`dict`), в котором есть ключи, но нет значений. Множества реализованы тем же способом, следовательно, значения времени выполнения для общих операций одинаковы. Некоторые операции, соответствующие математическим представлениям о множествах, не совпадают с операциями для словарей. Операции, создающие новое множество, сохраняют неизмененными исходные множества. В табл. 5.3 через n_A обозначен размер множества `A`, а через n_B – размер множества `B`.

Таблица 5.3. Операции с множеством

Название операции	Код	Стоимость
Добавление нового элемента	`A.add(новый_элемент)`	1
Удаление элемента	`A.delete(элемент)`	1
Объединение	`A | B`	$n_A + n_B$
Пересечение	`A & B`	min n_A,n_B
Выборка отличающихся элементов (разность множеств)	`A - B`	n_A
Симметричная разность	`A` $\wedge$ `B`	$n_A + n_B$

5.4. Асимптотический анализ и порядок роста

Основная цель заключается не в том, чтобы точно предсказать, сколько времени займет выполнение алгоритма, а, спрогнозировать порядок роста вре-

мени по мере увеличения размера входных данных. То есть если имеется алгоритм, работающий со списком длиной n, то время его выполнения может быть пропорционально n. В этом случае алгоритм будет выполняться в 100 раз дольше, если длина списка в 100 раз больше. Для второго алгоритма может потребоваться время выполнения, пропорциональное n^2 для входных данных длиной n. Тогда этот алгоритм потребует в 10 000 раз большего времени выполнения, если входной список будет в 100 раз длиннее. Следует отметить, что для этих фактов точный постоянный коэффициент пропорциональности не имеет важного значения.

Размер входных данных означает число битов, необходимое для кодирования этих данных. Поскольку мы не будем принимать во внимание постоянные множители, можно просто учитывать количество слов (слово равно 64 битам), необходимое для кодирования входных данных. В общем случае целое число и число с плавающей точкой хранится в одном слове. С технической точки зрения в Python можно хранить действительно большие числа, используя тип целого числа, для которого требуется намного больше слов. Но по соглашению мы будем предполагать, что целым числам (`int`) и числам с плавающей точкой (`float`) соответствует постоянное число битов. Это соглашение необходимо для предположения о том, что арифметические операции выполняются за постоянное время.

5.5. Сосредоточимся на самом худшем случае

Обычно для различных данных одинаковой длины может потребоваться разное время выполнения. В большинстве случаев мы будем использовать стандартное соглашение о рассмотрении наихудшего случая. Мы ищем верхние границы времени выполнения. Если алгоритм обладает временем выполнения, которое лучше (меньше), чем аналитический прогноз, то это очень хорошо.

5.6. O-большое

Почти всегда мы будем описывать время выполнения как функцию от размера входных данных. То есть время выполнения для входных данных размером n может быть описано как $5n^2 + 3n + 2$. Если записать это выражение как (математическую) функцию от n, то можно назвать эту функцию f и писать $f(n) = 5n^2 + 3n + 2$. В этом примере $5n^2$ – наиболее важный член.

Формальное математическое определение, которое позволяет не принимать во внимание постоянные множители (коэффициенты), называется нотацией (формой записи) «O-большое». Для разминки приведем простой пример, в котором можно сказать, что

$$f(n) = O(n^2),$$

если существует постоянная величина c, такая что

$$f(n) \leqslant cn^2 \text{ для всех достаточно больших } n.$$

Это верно для $f(n) = 5n^2 + 3n + 2$, потому что если принять $c = 6$, то можно видеть, что, пока $n > 4$, получаем:

$$f(n) = 5n^2 + 3n + 2 < 5n^2 + 4n < 5n^2 + n^2 \leqslant 6n^2.$$

Приведенные выше неравенства выводятся из повторного применения того факта, что $n > 4$.

Теперь можно привести формальное определение нотации О-большого: при заданных (неубывающих) функциях f и g мы говорим, что $f(n) = O(g(n))$, если существуют константы c и n_0, такие что для всех $n > n_0$ имеем $f(n) \leqslant cg(n)$.

5.7. Самые важные свойства использования О-большого

Нотация О-большого скрывает постоянные множители (коэффициенты). Любой член выражения, не зависящий от размера входных данных, считается константой и будет скрыт в нотации О-большого.

Нотация О-большого сообщает о том, какое время выполнения в итоге будет истинным, если размер входных данных достаточно велик.

Эти два свойства представлены в формальном определении О-большого. Постоянная величина c – это константа, которая замещает все прочие постоянные множители. Она также позволяет скрыть члены меньших порядков. Константа n_0 – это пороговое значение, после которого неравенство становится истинным.

5.8. Практическое использование О-большого и общие функции

Даже притом, что определение нотации О-большого позволяет сравнивать все типы функций, обычно мы будем использовать эту нотацию для упрощения функций, исключения лишних констант и понижения порядка членов. Поэтому, например, вы должны писать $O(n)$ вместо $O(3n)$ или $O(n^2)$ вместо $O(5n^2 + 3n + 2)$. Ниже перечислены некоторые функции, которые применяются так часто, что мы должны включить их в свой словарь.

- Постоянные функции $O(1)$.
- Логарифмические функции $O(\log n)$.
- Линейные функции $O(n)$.
- Функция $n \log n$: $O(n \log n)$.
- Квадратичные функции $O(n^2)$.
- Полиномиальные функции $O(n^k)$ для некоторой константы k.
- Экспоненциальные функции $O(2^n)$ (они отличаются от показательных функций $2^{O(n)}$).
- Факториальные функции $O(n!)$.

5.9. Основания логарифмов

Возможно, вы заметили, что в предыдущем разделе для логарифмов не были указаны основания. Причина в том, что внутри записи O-большого логарифмы с любым постоянным основанием одинаковы. То есть $\log_a(n) = O(\log_b(n))$, где a и b – две любые константы. Докажем это равенство. Пусть $c = 1 / \log_b(a)$ и $n_0 = 0$.

$$\log_a(n) = \log_b(n)/\log_b(a) \leqslant c \log_b(n) \text{ для всех } n > n_0.$$

5.10. Практические примеры

В каждом из приведенных ниже примеров предполагается, что входными данными является список длиной n.

```
def f002(L):
    newlist = []                # 2 - создание нового списка и присваивание переменной.
    for i in L:                 # Цикл выполняется n раз.
        if i % 2 == 0:          # 1
            newlist.append(i)   # 1 (в списках элемент добавляется за постоянное время).
    return newlist              # 1 возврат из функции.
```

Подсчитаем стоимость каждой строки этого кода. Стоимости указаны в комментариях. Таким образом, общая (суммарная) стоимость приблизительно равна $2n + 3$ в наихудшем случае (т. е. если все элементы списка четные). Полученный результат необходимо обозначить как $O(n)$ и называть этот код алгоритмом, выполняемым за линейное время, или просто линейным алгоритмом (linear algorithm).

```
def f003(L):
    x = 0               # 1
    for i in L:         # Цикл выполняется n раз.
        for j in L:     # Цикл выполняется n раз.
            x += i*j    # 3 - две арифметические операции и присваивание.
    return x            # 1
```

И для этой функции подсчитаем стоимость каждой строки этого кода. Стоимости указаны в комментариях. Стоимость внутреннего цикла равна $3n$, и он выполняется n раз, поэтому общая стоимость всего метода в целом равна $3n^2 + 2$. Полученный результат необходимо обозначить как $O(n^2)$ и называть этот код алгоритмом, выполняемым за квадратичное время, или просто квадратичным алгоритмом (quadratic algorithm).

Еще один пример, с которым мы уже встречались несколько раз.

```
def f004(L):
    x = 0                       # 1
    for j in range(1,len(L))    # Цикл выполняется n-1 раз.
        for i in range(j)       # Цикл выполняется j раз.
          x += L[i] * L[j]  # 5 - два доступа к списку, две арифметич.операции и присваивание.
    return x                    # 1
```

Здесь вычисления будут немного более сложными, потому что число итераций внутреннего цикла изменяется на каждой итерации внешнего цикла. В этом случае не составляет труда просуммировать стоимости каждой итерации внешнего цикла по одной за раз. Первая итерация стоит 5, вторая – 10 и т. д. до j-й итерации, которая стоит 5j. Общая стоимость (включая инициализацию x и возврат из функции) равна

$$2 + \sum_{i=1}^{n-1} 5i = 2 + 5\sum_{i=1}^{n-1} i = 2 + \frac{5n(n-1)}{2} = O(n^2).$$

В дальнейшем мы часто будем встречаться с этим типом суммирования, поэтому рекомендуется запомнить его и всегда распознавать (и в исходном коде, и как математическое выражение).

Глава 6

Стеки и очереди

6.1. Абстрактные типы данных

В этой книге мы постоянно будем использовать абстрактный тип данных (АТД) (abstract data types – ADT) как начальный пункт обсуждения любой конкретной структуры данных. Сам по себе АТД не является структурой данных, но он может многое сообщить о любых структурах данных. Способ использования термина АТД очень похож на применение термина «интерфейс». АТД отвечает на два главных вопроса:

1) что представляют собой данные, которые должны быть сохранены или представлены?
2) что мы можем сделать с этими данными?

Все это в совокупности описывает поведение или семантику (смысл) конкретной структуры данных. Когда мы рассматриваем АТД, то перечисляем имена методов, которые будут представлены, тип входных данных, которые они принимают, а также ожидаемый вывод. Кроме того, АТД описывает ситуации возникновения ошибок и что происходит при этом. Структура данных – это реализация АТД. Чтобы подчеркнуть это различие, иногда полезно применять термин «конкретная структура данных», хотя обычно мы не употребляем слово «конкретная».

АТД сообщает, какие методы структуры данных будут реализованы. Но АТД не дает никаких советов или предположений о том, как реализовать конкретную структуру данных. Это важно и как определение, и как основной направляющий принцип проектирования в объектно-ориентированном программировании, поэтому я еще раз напоминаю:

АТД должен быть независимым от всех предположений относительно его реализации.

Вероятно, вы обратили внимание на то, что два вопроса, на которые отвечает АТД, связаны с определением инкапсуляции, приводимым при обсуждении объектно-ориентированного программирования. Поэтому при реализации структур данных на языке Python мы будем размещать их в классах.

6.2. Абстрактный тип данных «стек»

- `push` – добавление нового элемента в стек.
- `pop` – удаление и возврат очередного элемента в порядке «последним вошел, первым вышел» (Last In First Out – LIFO).
- `peek` – возврат очередного элемента в порядке «последним вошел, первым вышел» (LIFO).
- `size` – возврат числа элементов в стеке (будет использоваться метод в стиле Python `__len__`).
- `isempty` – возврат `True`, если в стеке нет элементов, иначе возврат `False`.

Этот АТД можно относительно просто реализовать, используя список (`list`). Мы реализуем его с помощью класса с именем `ListStack`. Таким образом, в имени мы даем подсказку о реализации. Такой способ наиболее часто применяется при программировании на языке Java, но мы принимаем такое соглашение в этой книге, чтобы проще было различать реализации одного и того же АТД.

```
class ListStack:
    def __init__(self):
        self._L = []

    def push(self, item):
        self._L.append(item)

    def pop(self):
        return self._L.pop()

    def peek(self):
        return self._L[-1]

    def __len__(self):
        return len(self._L)

    def isempty(self):
        return len(self) == 0
```

Показанный выше класс `ListStack` демонстрирует объектно-ориентированную стратегию композиции (стек `ListStack` содержит список `list`). Он также представляет собой пример шаблона «обертка» (wrapper pattern). Встроенный в Python список `list` выполняет всю тяжелую работу, но пользователям ничего не известно, да и нет никакого дела до того, как реализованы методы этого класса. Но это утверждение не всегда верно. Пользователь должен начать беспокоиться, если производительность невысока. Не слишком трудно сделать этот код неэффективным. Например, можно реализовать стек, помещая элементы в начало списка. Здесь мы используем наследование, чтобы избежать переписывания методов, которые не должны измениться.

```
from ds2.stack import ListStack

class BadStack(ListStack):
    def push(self, item):
        self._L.insert(0, item)

    def pop(self):
        return self._L.pop(0)

    def peek(self):
        return self._L[0]
```

Простой асимптотический анализ показывает, почему эта реализация гораздо менее эффективна. Вставка нового элемента в список требует перемещения всех прочих элементов, чтобы освободить место для него. Если мы вставляем элемент в начало списка, то каждый существующий элемент должен быть скопирован в новую позицию. Таким образом, вызов операции вставки `insert` в методе `push` требует времени $O(n)$. И при извлечении элемента из начала списка каждый элемент, остающийся в списке, должен переместиться на одну позицию, чтобы заполнить образовавшуюся пустоту. Следовательно, вызов `list.pop` в методе `pop` класса стека также требует времени $O(n)$. То есть оба метода `push` и `pop` выполняются за линейное время в этой реализации. Линейный алгоритм действительно оправдывает свое название.

6.3. Абстрактный тип данных «очередь»

- `enqueue(item)` – добавление нового элемента в очередь.
- `dequeue()` – удаление и возврат очередного элемента в порядке «первым вошел, первым вышел» (First In First Out – FIFO).
- `peek()` – возврат (без удаления) очередного элемента в очереди в порядке FIFO.
- `__len()__` – возврат числа элементов в очереди.
- `isempty()` – возврат `True`, если в очереди нет элементов, иначе возврат `False`.

```
class ListQueueSimple:
    def __init__(self):
        self._L = []

    def enqueue(self, item):
        self._L.append(item)

    def dequeue(self):
        return self._L.pop(0)

    def peek(self):
        return self._L[0]

    def __len__(self):
        return len(self._L)

    def isempty(self):
        return len(self) == 0
```

Тут вы говорите: «Постойте. Я думал, что вызов `pop(0)` – это неправильное действие».

Да, это требует времени, пропорционального длине списка, но что мы можем сделать? Если удалить (с извлечением) конец списка, то мы должны будем пополнить очередь вставкой в начало списка. Это тоже плохой вариант.

Есть другая идея. Не будем по-настоящему удалять элементы из начала списка. Вместо этого игнорируем их, сохраняя текущий реальный индекс головы очереди.

```
class ListQueueFakeDelete:
    def __init__(self):
        self._head = 0
        self._L = []

    def enqueue(self, item):
        self._L.append(item)

    def peek(self):
        return self._L[self._head]

    def dequeue(self):
        item = self.peek()
        self._head += 1
        return item

    def __len__(self):
        return len(self._L) - self._head

    def isempty(self):
        return len(self) == 0
```

В этом коде есть одна небольшая странность: он никогда не избавляется от старых элементов после их удаления из очереди (dequeue). Даже если элементы удаляются, их позиция в списке сохраняется. Этот прием называется «ленивое обновление» (lazy update). Должны ли мы сами очищать очередь после таких «удалений»? Да, но не спешите. Здесь работает следующий принцип. Даже если список становится наполовину пустым, т. е. если значение `_head` превышает половину значения длины списка `_L`, то мы, стиснув зубы, начинаем действовать и заменяем `_L` вырезкой из этого списка. «Стиснув зубы» здесь – особенно удачное выражение, если рассматривать этот процесс как своего рода «ампутацию старого отмершего органа» из списка.

```
class ListQueue(ListQueueFakeDelete):
    def dequeue(self):
        item = self._L[self._head]
        self._head += 1
        if self._head > len(self._L)//2:
            self._L = self._L[self._head:]
            self._head = 0
        return item
```

В этом случае похоже, что мы потеряли все преимущества ленивого обновления, потому что теперь у нас есть метод `dequeue`, который иногда тре-

бует линейного времени выполнения. Но мы не выполняем его на каждом шаге. В какой степени он в действительности является слишком затратным (расточительным)? Если мы сначала выполняем все операции `enqueue`, после этого удаляем (dequeue) все элементы, то некоторые элементы перемещаются в конец списка (т. е. копируются в новую локацию памяти во время операции вырезки) много раз. Первая половина элементов вообще не перемещается. Следующая четверть списка (от 1/2 до 3/4) перемещается ровно один раз. Очередная восьмая часть списка сдвигается ровно два раза. При общем количестве n элементов существует $n/2^i$ элементов, которые были перемещены $i - 1$ раз. Таким образом, общее число перемещений для n элементов не превышает

$$n\sum_{i=1}^{\log n}\frac{i-1}{2^i} < n.$$

Поэтому «в среднем» стоимость для каждого элемента является постоянной.

Этот вид ленивого обновления чрезвычайно важен. В действительности это именно тот способ, которым Python обеспечивает быстрое выполнение операции `list.pop`. С технической точки зрения `pop()` также может требовать линейного времени для некоторых вызовов, но в среднем стоимость каждой операции постоянна. Тот же подход делает быстрым и метод `append`. В этом случае Python выделяет дополнительное пространство (памяти) для списка, и при каждой операции заполнения список копируется в область большего размера, т. е. размер приблизительно удваивается.

6.4. Обработка ошибок

Часто мы делаем предположения о том, как можно или необходимо использовать тот или иной класс. Это также существенная часть семантики конкретного класса, которая влияет на то, как мы программируем этот класс. Всегда желательно написать код без ошибок. Хотелось бы сделать так, чтобы код работал всегда независимо от того, используют его надлежащим образом или некорректно, но иногда сообщение об ошибке является правильным поведением.

В Python мы генерируем ошибку тем способом, которым бросают яйцо. Либо кто-то аккуратно ловит его, либо яйцо разбивается. В зависимости от ситуации любой вариант может быть правильным.

В случае со стеком не существует корректного варианта использования метода `pop` для пустого стека. Следовательно, есть смысл в том, что все, кто использует наш класс `Stack`, должны привести свою программу к аварийному завершению и прочесть сообщение об ошибке, если они пытаются вызвать `pop` при отсутствии элементов в стеке. В приведенной выше реализации со списком именно это и происходит:

```
s = ListStack()
s.push(5)
s.pop()
s.pop()
```

```
Traceback (most recent call last):
  File "liststackerror", line 24, in <module>
    s.pop()
  File "liststackerror", line 9, in pop
    return self._L.pop()
IndexError: pop from empty list
```

Если посмотреть на сообщение об ошибке, то оно выглядит достаточно понятным. В нем сообщается, что мы попытались «извлечь элемент из пустого списка» (`pop from empty list`). Но если заглянуть в исходный код, то вы, вероятно, спросите: «Из какого списка?» Мы знаем, как реализован класс `ListStack`, и даже его имя дает подсказку о реализации, поэтому можем догадаться о том, что происходит, но пользователю придется искать в трассировке стека строку собственного кода, в которой возникла эта проблема. Можно было бы перехватить это исключение в нашем методе `pop` и сгенерировать другую ошибку, чтобы источник проблемы стал более очевидным в пользовательском коде. В противном случае трассировка стека сообщает об ошибке в нашем коде. Затем пользователю, возможно, придется попытаться вникнуть в семантику нашего класса, чтобы вернуться назад и понять, что он сделал неправильно в своем коде. Вместо этого лучше сообщить пользователю об ошибке с точным объяснением, что произошло.

```
from ds2.stack import ListStack

class AnotherStack(ListStack):
    def pop(self):
        try:
            return self._L.pop()
        except IndexError:
            raise RuntimeError("pop from empty stack")

s = AnotherStack()
s.push(5)
s.pop()
s.pop()

Traceback (most recent call last):
  File "yjc5xp3sru", line 6, in pop
    return self._L.pop()
IndexError: pop from empty list

During handling of the above exception, another exception occurred:
# Во время обработки приведенного выше исключения возникло другое исключение:

Traceback (most recent call last):
  File "yjc5xp3sru", line 13, in <module>
    s.pop()
  File "yjc5xp3sru", line 8, in pop
    raise RuntimeError("pop from empty stack")
RuntimeError: pop from empty stack
```

Глава 7

Деки и связные списки

Дек (deque – double-ended queue) – это двусторонняя очередь (очередь с двусторонним доступом). Дек работает и как стек, и как очередь, сохраняя упорядоченный набор элементов с возможностью их добавления или удаления в начале и в конце дека. Это также абстрактный тип данных (АТД).

7.1. Абстрактный тип данных «дек»

- `addfirst(item)` – добавляет элемент `item` в начало дека.
- `addlast(item)` – добавляет элемент `item` в конец дека.
- `removefirst(item)` – удаляет и возвращает первый элемент в деке.
- `removelast(item)` – удаляет и возвращает последний элемент в деке.
- `len` – возвращает число элементов в деке.

Сначала рассмотрим, как может выглядеть реализация дека с использованием списка.

```
class ListDeque:
    def __init__(self):
        self._L = []

    def addfirst(self, item):
        self._L.insert(0, item)

    def addlast(self, item):
        self._L.append(item)

    def removefirst(self):
        return self._L.pop(0)

    def removelast(self):
        return self._L.pop()

    def __len__(self):
        return len(self._L)
```

Приведенный выше код достаточно прост, но некоторые операции начинают замедляться при увеличении размера дека. Вставка и извлечение по индексу 0 требует времени O(n). Это следует из способа последовательного хранения в памяти элементов списков. Невозможно изменить начало списка без сдвига всех прочих элементов, чтобы освободить место или заполнить пустоты. Для выполнения этой работы придется отказаться от идеи последовательного размещения элементов в памяти. Вместо этого будем хранить некоторую дополнительную информацию о каждом элементе, которую можно использовать для извлечения в определенном порядке.

7.2. Связные списки

Связные списки – это простая структура данных для хранения последовательного набора элементов. В отличие от стандартного списка Python связный список позволяет быстро вставлять элемент в начало. Основная идея заключается в хранении элементов в отдельных объектах, которые называются узлами (nodes). Особым узлом является голова списка. Каждый узел хранит ссылку на следующий узел в списке.

Мы воспользуемся обычным приемом, заглянув в наше интуитивное описание, приведенное выше, чтобы найти существительные, представляющие типы, которые нужно будет определить. В данном случае это сам список и узлы в нем. Сначала напишем класс для узла `ListNode`.

```
class ListNode:
    def __init__(self, data, link = None):
        self.data = data
        self.link = link
```

Теперь, чтобы начать создание связного списка `LinkedList`, необходимо сохранить его голову. Мы предоставляем два метода, `addfirst` и `removefirst`, которые изменяют начало списка. Их поведение приблизительно напоминает операции записи и извлечения элементов в стеке. Эта первая реализация скрывает узлы от пользователя. То есть, с точки зрения пользователя, можно создать связный список, добавлять и удалять узлы, но невозможно как-либо воздействовать на сами узлы (или даже узнать что-либо о них). Это и есть абстракция (скрытие подробностей реализации).

```
class LinkedList:
    def __init__(self):
        self._head = None

    def addfirst(self, item):
        self._head = ListNode(item, self._head)

    def removefirst(self):
        item = self._head.data
        self._head = self._head.link
        return item
```

7.3. Реализация очереди с помощью класса LinkedList

Напомню, что даже наша лучшая реализация очереди с помощью списка требовала линейного времени в наихудшем случае для операций dequeue (но в среднем – постоянного времени). Есть надежда улучшить этот результат с помощью связного списка. Но прямо сейчас у нас нет способа добавления или удаления элемента в конце списка. Ниже приведен простой и корректный, хотя и неэффективный способ решения этой задачи.

```
class LinkedList:
    def __init__(self):
        self._head = None

    def addfirst(self, item):
        self._head = ListNode(item, self._head)

    def addlast(self, item):
        if self._head is None:
            self.addfirst(item)
        else:
            currentnode = self._head
            while currentnode.link is not None:
                currentnode = currentnode.link
            currentnode.link = ListNode(item)

    def removefirst(self):
        item = self._head.data
        self._head = self._head.link
        return item

    def removelast(self):
        if self._head.link is None:
            return self.removefirst()
        else:
            currentnode = self._head
            while currentnode.link.link is not None:
                currentnode = currentnode.link
            item = currentnode.link.data
            currentnode.link = None
            return item
```

В новом методе addlast реализован шаблон, весьма часто применяющийся в связных списках. Метод начинает работу с головы связного списка и выполняет проход к концу, следуя по ссылкам link. При этом используется соглашение, по которому ссылкой link последнего узла является специальное значение None.

Подобным образом метод removelast проходит по списку, пока не достигнет второго с конца элемента.

Такой подход с последовательным проходом по списку не очень эффективен. Для списка длиной n потребуется время O(n), чтобы найти конец списка.

При другом подходе можно сохранить последний узел (хвост) списка, чтобы не нужно было тратить время на его поиск, когда возникнет такая необходимость. Это требует некоторых дополнительных накладных расходов для

полной уверенности в том, что мы всегда храним корректный конечный узел. Большинство особых случаев возникает, когда в списке имеется только один узел. У нас появляется возможность использовать это для некоторого улучшения метода `addlast`, поскольку можно сразу же перейти в конец списка без последовательного прохода. Кроме того, появляется возможность немного почистить код метода `removelast`, исключив проверку ссылки `link.link`, а вместо нее просто проверить достижение хвоста (последнего узла).

```
class LinkedList:
    def __init__(self):
        self._head = None
        self._tail = None

    def addfirst(self, item):
        self._head = ListNode(item, self._head)
        if self._tail is None: self._tail = self._head

    def addlast(self, item):
        if self._head is None:
            self.addfirst(item)
        else:
            self._tail.link = ListNode(item)
            self._tail = self._tail.link

    def removefirst(self):
        item = self._head.data
        self._head = self._head.link
        if self._head is None: self._tail = None
        return item

    def removelast(self):
        if self._head is self._tail:
            return self.removefirst()
        else:
            currentnode = self._head
            while currentnode.link is not self._tail:
                currentnode = currentnode.link
            item = self._tail.data
            self._tail = currentnode
            self._tail.link = None
            return item
```

Теперь можно реализовать АТД «очередь» с использованием связного списка. Это оказывается удивительно простым делом.

```
# linkedqueue.py
from ds2.deque import LinkedList

class LinkedQueue:
    def __init__(self):
        self._L = LinkedList()

    def enqueue(self, item):
        self._L.addlast(item)
```

```
    def dequeue(self):
        return self._L.removefirst()

    def peek(self):
        item = self._L.removefirst()
        self._L.addfirst(item)
        return item

    def __len__(self):
        return len(self._L)

    def isempty(self):
        return len(self) == 0
```

7.4. Хранение длины

В классе `ListQueue` мы реализовали метод `__len__` для получения числа элементов в очереди. Чтобы реализовать аналогичный метод в классе `LinkedQueue`, необходимо делегировать (delegate) вычисление длины классу `LinkedList`.

Добавим возможность получения длины связного списка. Это будет сделано посредством сохранения значения длины и его обновления при выполнении каждой операции.

```
class LinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
        self._length = 0

    def addfirst(self, item):
        self._head = ListNode(item, self._head)
        if self._tail is None: self._tail = self._head
        self._length += 1

    def addlast(self, item):
        if self._head is None:
            self.addfirst(item)
        else:
            self._tail.link = ListNode(item)
            self._tail = self._tail.link
            self._length += 1

    def removefirst(self):
        item = self._head.data
        self._head = self._head.link
        if self._head is None: self._tail = None
        self._length -= 1
        return item

    def removelast(self):
        if self._head is self._tail:
            return self.removefirst()
        else:
            currentnode = self._head
```

```
            while currentnode.link is not self._tail:
                currentnode = currentnode.link
            item = self._tail.data
            self._tail = currentnode
            self._tail.link = None
            self._length -= 1
            return item

    def __len__(self):
        return self._length
```

Здесь остается необходимость в последовательном проходе по всему списку для того, чтобы удалить элемент в его конце. Похоже, что избежать этого очень трудно. В результате метод `removelast` продолжает требовать линейного времени выполнения. Мы не сможем исправить такой недостаток до следующей главы, потому что для этого потребуется новая идея. Поэтому сейчас вместо тестирования методов АТД «дек» мы рассмотрим, как реализовать эффективную очередь с использованием связного списка.

7.5. Тестирование на основании АТД

Напомню, что АТД «очередь» определяет ожидаемое поведение структуры данных «очередь». АТД описывает открытый интерфейс конкретного класса. Теперь, когда мы можем воспользоваться классом `LinkedList` для реализации очереди с постоянным временем выполнения операций в наихудшем случае, у нас есть несколько различных реализаций АТД. Правильные тесты для этих структур данных предполагают реализацию только того, что предоставлено в ADT. В действительности нам, вероятнее всего, необходимо протестировать обе реализации с помощью одних и тех же тестов.

Если переименовать первые реализации очереди `ListQueue`, то можно получить показанные ниже тесты.

```
import unittest
from ds2.queue import ListQueue

class TestListQueue(unittest.TestCase):
    def testinit(self):
        q = ListQueue()

    def testaddandremoveoneitem(self):
        q = ListQueue()
        q.enqueue(3)
        self.assertEqual(q.dequeue(), 3)

    def testalternatingaddremove(self):
        q = ListQueue()
        for i in range(1000):
            q.enqueue(i)
            self.assertEqual(q.dequeue(), i)
```

```
    def testmanyoperations(self):
        q = ListQueue()
        for i in range(1000):
            q.enqueue(2 * i + 3)
        for i in range(1000):
            self.assertEqual(q.dequeue(), 2 * i + 3)

    def testlength(self):
        q = ListQueue()
        self.assertEqual(len(q), 0)
        for i in range(10):
            q.enqueue(i)
        self.assertEqual(len(q), 10)
        for i in range(10):
            q.enqueue(i)
        self.assertEqual(len(q), 20)
        for i in range(15):
            q.dequeue()
        self.assertEqual(len(q), 5)

if __name__ == '__main__':
    unittest.main()
```

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.003s

OK
```

Теперь, когда у нас есть другая реализация, возникает искушение просто скопировать старые тесты, изменив ссылки с `ListQueue` на `LinkedQueue`. Это достаточно большой объем дублируемого кода, а дублируемый код приводит к проблемам. Например, предположим, что в коде возникают проблемы, если попытаться выполнить операцию `dequeue` из пустой очереди. Если мы решаем, что это правильное поведение, то необходимо подкрепить его соответствующим тестом. Должны ли мы копировать и этот тест в другой тестовый файл? Что если в одной реализации эта проблема устранена, а в другой нет? При копировании и переносе кода мы неизбежно копируем и переносим ошибки. Была одна ошибка, стало две.

Это стандартная ситуация, в которой требуется наследование. Мы хотели скопировать набор методов, чтобы включить их в два различных класса (`TestListQueue` и `TestLinkedQueue`). Вместо этого необходимо совместное использование (share) этих методов. Поэтому выполняется рефакторинг исходного кода с вынесением его части в суперкласс. Этот новый класс будет называться `TestQueue`. Оба подкласса, `TestListQueue` и `TestLinkedQueue`, расширяют суперкласс `TestQueue`. Напомню, что расширение означает наследование от суперкласса.

```
# testqueue.py
class QueueTests:
    def testinit(self):
        q = self.Queue()

    def testaddandremoveoneitem(self):
        q = self.Queue()
        q.enqueue(3)
        self.assertEqual(q.dequeue(), 3)

    def testalternatingaddremove(self):
        q = self.Queue()
        for i in range(1000):
            q.enqueue(i)
            self.assertEqual(q.dequeue(), i)

    def testmanyoperations(self):
        q = self.Queue()
        for i in range(1000):
            q.enqueue(2 * i + 3)
        for i in range(1000):
            self.assertEqual(q.dequeue(), 2 * i + 3)

    def testlength(self):
        q = self.Queue()
        self.assertEqual(len(q), 0)
        for i in range(10):
            q.enqueue(i)
        self.assertEqual(len(q), 10)
        for i in range(10):
            q.enqueue(i)
        self.assertEqual(len(q), 20)
        for i in range(15):
            q.dequeue()
        self.assertEqual(len(q), 5)
```

Этот новый класс выглядит почти так же, как старый `TestListQueue` с несколькими небольшими изменениями. Вместо создания нового объекта `ListQueue` в каждом тесте создается объект `self.Queue`. Это может показаться странным по двум причинам: во-первых, у нас нет такого класса, во-вторых, создаваемый объект привязан к `self`, который является объектом `TestQueue`. При реализации специализированных тестов для каждого класса мы будем присваивать переменной `Queue` класс, соответствующий конкретной реализации, которую необходимо протестировать.

Существует еще одно важное различие между классами `TestQueue` и `TestListQueue` – класс `TestQueue` не расширяет базовый класс `unittest.TestCase`. Он просто определяет некоторые методы, которые будут включены в классы `TestListQueue` и `TestLinkedQueue`. Ниже приведено содержимое новых тестовых файлов.

```
# testlistqueue.py
import unittest
from ds2.test.testqueue import QueueTests
from ds2.queue import ListQueue
```

```
class TestListQueue(unittest.TestCase, QueueTests):
    Queue = ListQueue

if __name__ == '__main__':
    unittest.main()

.....
----------------------------------------------------------------------
Ran 5 tests in 0.003s

OK

# testlinkedqueue.py
import unittest
from ds2.test.testqueue import QueueTests
from ds2.queue import LinkedQueue

class TestListQueue(unittest.TestCase, QueueTests):
    Queue = LinkedQueue

if __name__ == '__main__':
    unittest.main()

.....
----------------------------------------------------------------------
Ran 5 tests in 0.005s

OK
```

Оба класса, `TestListQueue` и `TestLinkedQueue`, расширяют `unittest.TestCase` и `TestQueue`. Это называется множественным наследованием (multiple inheritance). В других языках, таких как C++, которые поддерживают множественное наследование, это считается плохим проектным решением. Но в Python вполне приемлемо применение этого вида включения (mix in). Единственное, что необходимо помнить при этом, – золотое правило наследования непременно должно соблюдаться: наследование означает «является экземпляром» (is a).

Если требуется добавить специализированные тесты в одну из реализаций, но не в другую, то можно сделать это в подклассе. Но если имеется несколько (или даже много) реализаций, которые должны проходить одни и те же тесты, то можно попробовать применить показанный ниже шаблон. Он создает тестовый класс в функции. При таком способе добавление тестов для новой реализации требует только одной строки исходного кода.

```
# testbothqueues.py
import unittest
from ds2.test.testqueue import QueueTests
from ds2.queue import ListQueue, LinkedQueue

def _test(queue_class):
    class QueueTestCase(unittest.TestCase, QueueTests):
        Queue = queue_class
    return QueueTestCase
```

```
TestLinkedQueue = _test(LinkedQueue)
TestListQueue = _test(ListQueue)
# TestYetAnotherQueue = _test(YetAnotherQueue)
# TestCrazyOtherQueue = _test(CrazyOtherQueue)

if __name__ == '__main__':
    unittest.main()

..........
----------------------------------------------------------------------
Ran 10 tests in 0.008s

OK
```

Обратите внимание: было выполнено десять тестов, по пять для каждой реализации. Этот шаблон постоянно используется для тестирования кода в данной книге. Рекомендуется внимательно изучить его исходный код на сайте GitHub, и в особенности тесты.

7.6. Основные уроки

- Используйте открытый интерфейс, который описан в АТД, для тестирования своего класса.
- Можно воспользоваться наследованием для совместного использования функциональности несколькими классами.
- Наследование означает «является экземпляром» (is a).

7.7. Шаблоны проектирования: шаблон «обертка»

В двух последних главах мы видели несколько различных реализаций АТД «очередь» (Queue). Основные реализации `LinkedQueue` и `ListQueue` были очень простыми. В обоих случаях применялась композиция – в класс включался объект другого класса, затем большинство операций делегировалось этому другому классу. Эти примеры демонстрировали прием, называемый шаблоном «оберткой» (Wrapper Pattern). В обоих случаях очередь являлась оберткой вокруг некоторой другой структуры данных.

Шаблоны проектирования (design patterns), также известные под названием «шаблоны объектно-ориентированного проектирования» (object-oriented design patterns), или просто «шаблоны» (patterns), – это способы организации классов при решении общих задач программирования. В случае шаблона «обертка» мы получаем класс, который уже «в определенной степени» делает то, что нужно, но содержит отличающиеся имена для операций и, возможно, множество других операций, поддержка которых нам не нужна. Поэтому мы создаем новый класс, который содержит (has an) экземпляр другого класса (`list` или `LinkedList` в рассматриваемых в этой главе примерах) и предоставляет методы, работающие с этим объектом (экземпляром). Пользователи, находящиеся снаружи этого класса, не должны ничего знать о классе, расположенном внутри обертки. Иногда такое разделение называется уровнем абстракции

(layer of abstraction). При использовании нашего класса внешний пользователь не должен знать ничего о его внутренней реализации.

Ниже перечислены некоторые полезные уроки, извлеченные из изложенного выше материала:

- везде, где возможно, используйте шаблоны проектирования для правильной организации исходного кода и улучшения его читабельности;
- шаблон «обертка» предоставляет способ создания альтернативного интерфейса к методам (или некоторому подмножеству методов) другого класса;
- композиция означает «содержит экземпляр» (has a).

Глава 8

Двусвязные списки

В предыдущей главе был представлен абстрактный тип данных (АТД) «дек» и приведены два варианта его реализации: с использованием обычного и связного списка. В реализации с использованием связного списка все основные операции выполнялись за постоянное время, за исключением операции удаления последнего элемента `removelast`. В этой главе будет представлена новая структура данных, позволяющая выполнять все операции дека за постоянное время. Здесь основной идеей будет хранение двух ссылок в каждом узле – в прямом и обратном направлении, – чтобы можно было выполнять проход по списку в любом направлении. В этом двусвязном списке (double-linked list) удаление элемента из конца будет симметрично удалению из начала, просто роли головы и хвоста меняются местами.

При создании нового класса узла `ListNode` можно определить узлы, расположенные перед текущим узлом и после него, чтобы установить соответствующие ссылки на них. Необходимо, чтобы всегда было истинным выражение `b == a.link`, если и только если `a = b.prev` для любых двух узлов `a` и `b`. Для обеспечения этого инварианта мы устанавливаем `self.prev.link = self` и `self.link.prev = self`, если только `prev` или `link` не содержат значение `None` соответственно.

```
class ListNode:
    def __init__(self, data, prev = None, link = None):
        self.data = data
        self.prev = prev
        self.link = link
        if prev is not None:
            self.prev.link = self
        if link is not None:
            self.link.prev = self
```

Сначала рассмотрим операции добавления элементов в двусвязный список `DoublyLinkedList`. Эти операции очень похожи на операцию `addfirst` в связном списке `LinkedList`. Здесь необходимо проделать некоторую дополнительную работу для обновления ссылки на предыдущий узел `prev`, которой не было в связном списке `LinkedList`.

```
class DoublyLinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
        self._length = 0

    def addfirst(self, item):
        if len(self) == 0:
            self._head = self._tail = ListNode(item, None, None)
        else:
            newnode = ListNode(item, None, self._head)
            self._head.prev = newnode
            self._head = newnode
            self._length += 1

    def addlast(self, item):
        if len(self) == 0:
            self._head = self._tail = ListNode(item, None, None)
        else:
            newnode = ListNode(item, self._tail, None)
            self._tail.link = newnode
            self._tail = newnode
        self._length += 1

    def __len__(self):
        return self._length
```

Приведенный выше код требует немедленного рефакторинга. Очевидно, что здесь имеется переизбыток логики, совместно используемой двумя методами. Мы должны воспользоваться этим как возможностью для упрощения кода. В данном случае можно рассмотреть более общую задачу добавления узла между двумя другими узлами. Необходимо разобрать только те частные случаи, в которых узел `before` или `after` или оба эти узла имеют значение `None`.

```
class DoublyLinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
        self._length = 0

    def __len__(self):
        return self._length

    def _addbetween(self, item, before, after):
        node = ListNode(item, before, after)
        if after is self._head:
            self._head = node
        if before is self._tail:
            self._tail = node
        self._length += 1

    def addfirst(self, item):
        self._addbetween(item, None, self._head)

    def addlast(self, item):
        self._addbetween(item, self._tail, None)
```

В коде этого класса симметрия также применима и для удаления элемента с любого конца списка. Как и для методов add, мы выносим в отдельный (закрытый) метод операцию удаления узла и возврата его данных. В этот метод включена логика, определяющая, изменится ли голова, или хвост, или оба этих узла при удалении.

```
    def _remove(self, node):
        before, after = node.prev, node.link
        if node is self._head:
            self._head = after
        else:
            before.link = after
        if node is self._tail:
            self._tail = before
        else:
            after.prev = before
        self._length -= 1
        return node.data

    def removefirst(self):
        return self._remove(self._head)

    def removelast(self):
        return self._remove(self._tail)
```

8.1. Объединение двусвязных списков

Существует несколько операций, которые выполняются очень быстро в двусвязных списках по сравнению с другими типами списков. Одной из наиболее полезных операций является возможность объединения (конкатенации) двух двусвязных списков. Напомню, что знак плюс можно использовать для конкатенации двух объектов списка list.

```
A = [1,2,3]
B = [4,5,6]
C = A + B
print(A)
print(B)
print(C)

[1, 2, 3]
[4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

Для списков операция конкатенации создает новый список. Эта операция требует времени, пропорционального длине нового создаваемого списка C, и не изменяет исходные списки A и B. Для двусвязных списков можно достигнуть такого же асимптотического времени выполнения при инкрементальном формировании нового списка. Но если разрешить изменять исходные списки, то конкатенацию можно выполнить так, чтобы хвост первого списка указывал на голову второго.

```
    def __iadd__(self, other):
        if other._head is not None:
            if self._head is None:
                self._head = other._head
            else:
                self._tail.link = other._head
                other._head.prev = self._tail
            self._tail = other._tail
            self._length = self._length + other._length

            # Очистка другого списка.
            other.__init__()
        return self

L = DoublyLinkedList()
[L.addlast(i) for i in range(11)]
B = DoublyLinkedList()
[B.addlast(i+11) for i in range(10)]

L += B

n = L._head
while n is not None:
    print(n.data, end = ' ')
    n = n.link

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Необходимо всегда помнить о различиях между продемонстрированной выше операцией объединения двусвязных списков и операцией объединения обычных списков. При объединении двусвязных списков конкатенация очищает второй список (делает его пустым). Это делается потому, что нам не нужны многочисленные двусвязные списки с одинаковыми узлами `ListNode`. Может возникнуть проблема, если мы пытаемся редактировать только один из них, потому что изменения будут отображаться и в другом списке, возможно, с катастрофическими последствиями.

Глава 9

Рекурсия

Рекурсия (recursion) является важной теоретической концепцией в информатике. При изучении структур данных и основных алгоритмов мы можем подумать о нескольких вариантах использования рекурсии. Самый простой вариант практического ее применения – функция вызывает саму себя.

Несмотря на истинность этого определения, оно в действительности не сообщает нам ничего о том, как и почему используется эта методика и что с ней делать, когда она обнаруживается в практической деятельности. В этой главе перед нами поставлены три взаимосвязанные цели:

1) понять, как реализуется рекурсия в компьютере, и преобразовать ее в модель, позволяющую интерпретировать ее как рекурсивные функции;
2) использовать рекурсию как методику решения задач, распознавая роли «подзадач» как главное обоснование применения рекурсии;
3) научиться анализировать время выполнения рекурсивных функций.

Ниже приведен пример рекурсивной функции.

```
def f(k):
    if k > 0:
        return f(k-1) + k
    return 0

print(f(5))
```

В действительности это просто хорошо знакомая нам функция `sumk`, только тщательно замаскированная. Как она работает? Для вычисления суммы чисел от 1 до k она просто вычисляет сумму чисел от 1 до k – 1, а затем прибавляет к ней k. Да, нам известен более эффективный способ вычисления этой суммы, но в показанном здесь подходе есть кое-что хорошее. Нам предложено решить эту задачу при некоторых заданных входных данных k, а мы сделали вид, что уже имеем решение для меньших чисел (в данном случае для k – 1). Затем используем это решение для формирования решения для k. Волшебство порождается тем фактом, что якобы «известное нам» решение в действительности являлось той же самой функцией. Я называю это методикой рекурсии,

выражаемой фразой «сделаем вид, что кто-то другой уже написал это решение». Это очень удобный способ как для написания, так и для анализа рекурсивных функций.

9.1. Рекурсия и индукция

Функция `f` из предыдущего раздела делает то же самое, что и функция `sumk`, которую мы видели ранее. Это можно проверить, воспользовавшись методом, который называется индукцией (induction). В действительности такой процесс проверки в точности означает «доказательство методом индукции» (proof by induction). При этом доказывается некоторый факт посредством проверки истинности базового (элементарного) случая. Затем вы доказываете истинность для случая k, предположив, что случай k – 1 является истинным. Для конкретного примера, проверяющего идентичность функции `f` другой функции `sumk`, которая просто возвращает k(k + 1) / 2, доказательство приведено ниже. Сначала проверяем истинность базового случая:

$$f(0) = 0 = \frac{0 \times (0+1)}{2}.$$

Затем рассматриваем

$$f(k) = f(k-1) + k = \frac{(k-1)(k-1+1)}{2} + k = \frac{(k-1)k + 2k}{2} = \frac{k(k+1)}{2}.$$

У нас не будет широкого диапазона возможностей для выполнения доказательств такого типа, но эту методику следует запомнить, потому что существует тесная связь между рекурсией и индукцией, и применение этой связи на практике может углубить ваше понимание обеих концепций.

9.2. Некоторые основные правила

Ниже приведены некоторые основные правила, которые вы должны по возможности соблюдать для уверенности в том, что ваши рекурсивные алгоритмы будут успешно завершаться.

Определить базовый (элементарный) случай. Если при каждом вызове функции выполняется очередной рекурсивный вызов, то процесс никогда не завершится.

Рекурсивные вызовы должны продвигаться к базовому случаю. Обычно это означает, что рекурсивные вызовы обращаются к «меньшим» подзадачам. Здесь слово «меньшим» может иметь различные значения.

Не составляет труда написать рекурсивную функцию, которая теоретически никогда не завершается. Но на практике так называемая бесконечная рекурсия (infinite recursion) достаточно быстро приводит к исключению `RecursionError`. В Python существует ограничение на глубину рекурсии. Обычно это ограничение приблизительно равно 1000.

9.3. Стек вызовов функций

Получить более четкое представление о рекурсии помогает наблюдение за тем, как она работает на реальном компьютере. Проще всего сначала понять, как работают все вызовы функций, после чего должно стать ясно, что с технической точки зрения не существует различия при вызове рекурсивной функции и при вызове любой другой (обычной) функции.

```
def f(k):
    var = k ** 2
    return g(k+1) + var

def g(k):
    var = k + 1
    return var + 1

print(f(3))
```

```
15
```

В приведенном ниже коде возникает исключение `RecursionError`, даже несмотря на то, что ни одна из функций не вызывает саму себя напрямую. В действительности эта ошибка сообщает о том, что лимит глубины стека вызовов функций исчерпан.

```
def a(k):
    if k == 0: return 0
    return b(k)

def b(k):
    return c(k)

def c(k):
    return a(k-1)

a(340)
```

Любопытный пример использования рекурсии (с ошибкой переполнения стека вызовов) можно сформировать, создавая два списка, каждый из которых содержит другой список.

```
A = [2]
B = [2]
A.append(A)
B.append(B)
A == B
```

```
Traceback (most recent call last):
  File "y451fhneyp", line 5, in <module>
    A == B
RecursionError: maximum recursion depth exceeded in comparison
              # при сравнении была превышена максимальная допустимая глубина рекурсии
```

В приведенном выше примере рекурсивной функцией становится метод списка `list.__eq__`, который сравнивает два заданных списка при использовании оператора ==. Метод сравнивает списки, проверяя равенство их элементов. Оба списка `A` и `B` имеют длину 2. Их первые элементы совпадают (равны). Вторым элементом каждого списка является другой список. Для их сравнения выполняется еще один вызов `list.__eq__`. Он абсолютно аналогичен первому вызову, и этот процесс повторяется до тех пор, пока не будет исчерпан лимит рекурсии.

9.4. Последовательность Фибоначчи

Рассмотрим классический пример рекурсивно определяемой функции. Его название происходит от имени Леонардо Фибоначчи (Leonardo Fibonacci), который изучал рекурсивный процесс в контексте прогнозирования роста численности семейства кроликов.

```
def fib(k):
    if k in [0,1]: return k
    return fib(k-1) + fib(k-2)

print([fib(i) for i in range(15)])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

Это работает, но начинает заметно замедляться уже для относительно небольших значений k. Например, я попытался выполнить эту функцию при `k = 40`, но пришлось отказаться от такой попытки.

Это случай, с которым предстоит неоднократно встречаться в этом учебном курсе. Если мы принимаем рекурсию как способ мышления о разделении задач на более мелкие части, то можем найти весьма короткие рекурсивные алгоритмы, но по той или иной причине, возможно, потребуется переписать их без использования рекурсии. Ниже приведена версия суммирования k первых чисел, которая вместо рекурсии использует цикл.

```
def fib(k):
    a, b = 0,1
    for i in range(k):
        a, b = b, a + b
    return a

print(fib(400))
```

```
176023680645013966468226945392411250770384383304492191886725992896575345044216019675
```

Это намного лучше. На первый взгляд может показаться, что единственным различием при выполнении этого вычисления являются дополнительные накладные расходы на вызовы функций вместо обновления локальных переменных. Это неверно. Исследуем более тщательно, что именно делают вызовы функций в рекурсивной реализации. Предположим, что мы вызываем `fib(6)`.

Это приводит к вызовам fib(5) и fib(4). После этого выполняются вызовы fib(4) и fib(3), а также fib(3) и fib(2). Каждая из этих четырех функций в свою очередь вызывает еще по две функции. Это можно изобразить в форме дерева. Следует отметить, что мы уже имеем несколько вызовов fib(4 и fib(3). Также можно спросить: сколько раз мы будем вызывать одну и ту же функцию с одним и тем же значением? При вычислении fib(k) ответ достаточно любопытным образом связан с самими числами Фибоначчи. Пусть T(k) обозначает число вызовов fib при вычислении fib(k). Легко заметить, что T(k) = T(k − 1) + T(k − 2) + 1. В этом случае результат почти точно соответствует числам Фибоначчи (T(1) = 1, T(2) = 2, T(3) = 4, T(4) = 7, ...). В каждом случае значение T на единицу меньше числа Фибоначчи. Следовательно, время выполнения функции fib будет возрастать, как ряд чисел Фибоначчи или, что равнозначно, как идеальное семейство кроликов, т. е. экспоненциально. k-е число Фибоначчи приблизительно равно ϕ^k, где $\phi = (1 + \sqrt{5})/2 \approx 1{,}618$ и известно как золотое сечение.

9.5. Алгоритм Евклида

Алгоритм Евклида является классическим в любом смысле этого слова. Входными данными является пара целых чисел a, b, а выводом (результатом) – их наибольший общий делитель, т. е. наибольшее целое число, на которое без остатка делятся a и b. Это очень простой рекурсивный алгоритм. Сначала рассмотрим его исходный код, затем попытаемся понять, что он делает, почему он работает и как можно его улучшить.

```
def gcd(a, b):
    if a == b:
        return a
    if a > b:
        a, b = b, a
    return gcd(a, b - a)
```

Как и во всех рекурсивных алгоритмах, здесь существует базовый случай. Здесь если a == b, то a (или равнозначно b) является ответом, и мы возвращаем это значение. Иначе мы устанавливаем порядок a < b (если это необходимо) и выполняем рекурсивный вызов gcd(a, b – a).

Если вручную выполнить несколько конкретных примеров, то может показаться, что алгоритм выполняет много рекурсивных вызовов, когда b намного больше a. В действительности легко заметить, что потребуется как минимум a // b рекурсивных вызовов, прежде чем мы поменяем местами a и b. Эти вызовы просто многократно выполняют вычитание из одного и того же числа, пока оно не станет достаточно малым. Такой процесс многократного вычитания известен ученикам начальной школы как деление. На самом деле это глубокая идея, над которой стоит задуматься. Деление – это итерационное вычитание в той же мере, в какой умножение – это итерационное сложение, возведение в степень – итерационное умножение, а логарифмы – итерационное деление. Теоретики в области информатики используют даже итерационные логарифмы (которые называются log* – log со звездочкой).

Мы можем просто выполнить деление напрямую вместо многократного вычитания меньшего числа из большего. В итоге немного изменится базовый случай, и операцию вычитания заменит операция взятия модуля (остатка от целочисленного деления). Ниже приведен измененный исходный код.

```
def gcd(a, b):
    if a > b:
        a, b = b, a
    if a == 0:
        return b
    return gcd(a, b % a)

print("GCD of 12 and 513 is", gcd(12, 513))
print("GCD of 19 and 513 is", gcd(19, 513))
print("GCD of 19 and 515 is", gcd(515 ,19))
```

```
GCD of 12 and 513 is 3
GCD of 19 and 513 is 19
GCD of 19 and 515 is 1
```

Между прочим, если разрешить задавать `a` и `b` как произвольные числа, то можно попытаться найти примеры, в которых алгоритм НОД (наибольший общий делитель) попадает в бесконечную рекурсию. Таким способом вы можете открыть иррациональные числа. Если `a` и `b` – рациональные числа, то алгоритм также непременно завершается.

Если требуется найти самый наихудший пример, то можно попытаться найти пару чисел `(a,b)`, такую что после одной итерации вы получаете пару `(b-a,a)` с тем же самым отношением чисел. Затем можно проверить, что этот процесс будет продолжаться, следовательно, вы никогда не приблизитесь к базовому случаю. Но возможно ли это? Существует ли пара чисел с таким свойством? Ответ: да. Один из возможных вариантов такой пары: $a = 1$ и $b = \phi$ (золотое сечение).

Глава 10

Динамическое программирование

Термин «динамическое программирование» означает методику написания алгоритмов, в которой задача решается с использованием решений этой же задачи на частных случаях меньшего размера. Напомню, что такое же интуитивное объяснение приводилось и для рекурсивных алгоритмов. Разумеется, существует множество частных случаев, с помощью которых можно получить алгоритм динамического программирования, начиная с рекурсивного алгоритма, который неэффективен, потому что многократно выполняет рекурсивные вызовы функции с абсолютно одинаковыми параметрами, напрасно расходуя время.

Мы начнем с задачи размена денег, где требуется дать сдачу, используя наименьшее количество монет. Предположим, что у вас есть список номиналов монет [1, 5, 10, 25] и сумма сдачи, которую вы должны набрать. Первой попыткой этого решения может стать набор сдачи жадным (greedy) способом. При таком подходе вы просто пытаетесь добавлять монету наибольшего номинала до тех пор, пока не соберете нужную сумму сдачи. Ниже приведена реализация жадного метода. (Обратите внимание на связь с алгоритмом Евклида о поиске НОД.)

10.1. Жадный алгоритм

```
def greedyMC(coinvalueList, change):
    coinvalueList.sort()
    coinvalueList.reverse()
    numcoins = 0
    for c in coinvalueList:
        # Добавить максимально возможное число монет очередного наибольшего номинала.
        numcoins += change // c
        # Обновить сумму сдачи, остающуюся до возврата.
        change = change % c
    return numcoins

print(greedyMC([1,5,10,25], 63))
print(greedyMC([1, 21, 25], 63))
print(greedyMC([1, 5, 21, 25], 63))
```

```
6
15
7
```

Очевидно, что этот подход не годится. Во втором примере возвращается 15 монет, а в третьем – 7. В обоих случаях правильным ответом должно быть число 3, который можно получить, набрав для сдачи три монеты в 21 цент. Проблема возникает не из-за ошибки в исходном коде, а в некорректном алгоритме. Здесь жадное решение не работает. Попробуем вместо него применить рекурсию.

10.2. Рекурсивный алгоритм

```
def recMC(coinValueList, change):
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(coinValueList,change-i)
            if numCoins < minCoins:
                minCoins = numCoins
    return minCoins

# print(recMC([1,5,10,25],63)) # Вполне серьезно: даже не пытайтесь выполнить этот вариант.
print(recMC([1, 21, 25],63))
print(recMC([1, 5, 21, 25],63))
```

```
3
3
```

Эта версия работает правильно, но очень медленно. Возникает естественная мысль: попробовать мемоизацию (memoize) решений, полученных из предыдущих рекурсивных вызовов. При таком подходе мы не будем повторять работу по вычислению количества монет для конкретной суммы сдачи более одного раза. Но при этом не следует забывать о передаче словаря уже известных результатов при выполнении рекурсивных вызовов.

10.3. Версия с мемоизацией

```
def memoMC(coinValueList, change, knownResults):
    minCoins = change
    if change in coinValueList:
        knownResults[change] = 1
        return 1
    elif change in knownResults:
        return knownResults[change]
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + memoMC(coinValueList, change-i, knownResults)
            if numCoins < minCoins:
                minCoins = numCoins
                knownResults[change] = minCoins
    return minCoins

print(memoMC([1,5,10,25],63, {}))
```

```
knownresults = {}
print(memoMC([1, 5, 10, 21, 25], 63, knownresults))
print(knownresults)

6
3
{25: 1, 26: 2, 21: 1, 10: 1, 11: 2, 5: 1, 6: 2, 1: 1, 12: 3, 7: 3, 13: 4, 8: 4, 14: 5, 9:
5, ...}
```

Эта версия намного быстрее и теперь может работать с гораздо более крупными экземплярами входных данных. Немного неудобной была необходимость передачи пустого словаря для того, чтобы начать работу, но это не так уж важно. Рассмотрим подробнее словарь `knownresults`, который заполняется алгоритмом. Он содержит почти все значения от 1 до 63. Возможно, вместо рекурсивных вызовов, которые начинают с изменений в полном объеме и продвигаются вниз (к базовому случаю), мы могли бы сформировать словарь значений, начиная с самых малых и продвигаясь вверх. Это и есть сущность динамического программирования.

Случается так, что здесь применение списка имеет смысл, потому что необходимо получить доступ к элементам по их целочисленным индексам, начиная с 1. На каждом шаге следующий наилучший ответ можно найти перебором каждой монеты, вычитая ее номинал из текущего значения и проверяя список с целью определить, сколько монет требуется, чтобы собрать оставшуюся часть сдачи, если будет использована эта монета.

10.4. Алгоритм динамического программирования

Наконец мы можем переписать нашу программу как алгоритм динамического программирования. Основной принцип заключается в явном заполнении таблицы всех значений с использованием предварительно вычисленных значений для определения новых.

```
def dpMakeChange(coinValueList, change):
    # Создание списка для хранения результатов решения подзадач.
    minCoins = [None]*(change + 1)

    # Для каждого значения от 0 до суммы сдачи вычислить минимальное число необходимых монет.
    for cents in range(change+1):
        # Сначала предположить, что используются только монеты номиналом 1.
        minCoins[cents] = cents
        # Проверить, не приводит ли какая-либо монета к решению, которое лучше,
        # чем текущее наилучшее.
        for c in coinValueList:
            if cents >= c:
                minCoins[cents] = min(minCoins[cents], minCoins[cents - c] + 1)
    # Возврат только того элемента таблицы, который соответствует требуемому значению.
    return minCoins[change]

print(dpMakeChange([1,5,10,21,25], 63))
print(dpMakeChange([1,5,10,21,25], 64))

3
4
```

Главное различие между методом динамического программирования и рекурсией с мемоизацией состоит в том, что динамическое программирование формирует результаты снизу вверх, тогда как рекурсивная версия работает сверху вниз, начиная с самой большой задачи и спускаясь к меньшим подзадачам по необходимости. Соответствует ли какая-либо из этих методик способу, которым вы решаете задачи программирования?

10.5. Еще один пример

Чтобы оценить динамическое программирование как обобщенную методику решения задач, необходимо рассмотреть еще один пример. Как и в случае применения рекурсии, здесь главным моментом является определение задач меньшего размера для решения. В действительности эта тема является актуальной на протяжении всего курса. Все задачи можно разделить на меньшие подзадачи. Если эти меньшие подзадачи являются частными случаями той задачи, с которой мы начали процесс решения, то, возможно, здесь применима рекурсия или динамическое программирование.

Следующая задача, которую мы решим методом динамического программирования, называется поиском наибольшей общей подпоследовательности (longest common subsequence – LCS). Подпоследовательность (subsequence) строки `s` – это строка `t`, такая что все символы `t` расположены в `s` в том же порядке. Например, `'abc'` является подпоследовательностью строки `'xxxxxaxxxbxxxcxxxx'`.

Входными данными для задачи поиска наибольшей общей подпоследовательности является пара строк, которые мы называем `X` и `Y`. Выводимым результатом будет самая длинная строка, являющаяся подпоследовательностью обеих строк `X` и `Y`.

Одним из подходов к решению этой задачи является необходимость поиска минимального набора символов, которые нужно удалить из `X` и `Y`, чтобы результаты были равны (строки одинаковы). Но мы не должны перебирать все возможные подпоследовательности. Для строки длиной n существует 2^n подпоследовательностей. Это слишком много. Наш код может никогда не завершиться даже при n = 100.

Существует прием, позволяющий обойти эту проблему. Если `X` и `Y` завершаются одинаковым символом, то он является самым последним символом в наибольшей общей подпоследовательности. Таким образом, если `X[-1] == Y[-1]`, то `LCS(X,Y)` – это `LCS(X[:-1], Y[:-1]) + X[-1]`. С другой стороны, если `X[-1] != Y[-1]`, то как минимум один из символов `X[-1]` или `Y[-1]` не входит в LCS. В этом случае `LCS(X,Y)` длиннее, чем `LCS(X[:-1],Y)` и `LCS(X,Y[:-1])`. Эти рассуждения можно превратить в весьма аккуратный рекурсивный алгоритм.

```
def reclcs(X,Y):
    if X == "" or Y == "":
        return ""
    if X[-1] == Y[-1]:
        return reclcs(X[:-1], Y[:-1]) + X[-1]
    else:
        return max([reclcs(X[:-1], Y), reclcs(X, Y[:-1])], key = len)
```

Но при запуске этого алгоритма с входными данными умеренного размера кажется, что он выполняется бесконечно долго. Легко заметить, что при наличии двух длинных строк, в которых не совпадает ни один символ, дерево рекурсивных вызовов должно быть полным двоичным деревом с глубиной n. Этот соответствует 2^n рекурсивным вызовам. Но на самом деле так много явных рекурсивных вызовов не выполняется. Каждый такой вызов представлен в форме `reclcs(X[:i], Y[:j])`, т. е. он берет `i` первых символов `X` и `j` первых символов `Y` при некоторых значениях `i` и `j`. Это означает, что должно выполняться только $O(n^2)$ явных рекурсивных вызовов. Поскольку значение n^2 намного меньше, чем 2^n, многие рекурсивные вызовы будут выполняться повторно. Это избыточные действия, которых можно избежать. Можно было бы добавить мемоизацию, как в предыдущем примере, но в действительности здесь требуется динамическое программирование. Мы будем сохранять решения подзадач в словаре `t`, так что `t[(i,j)] == LCS(X[:i], Y[:j])`. Таблица решений инициализируется пустыми строками `""` для подзадач при `i = 0` или `j = 0` (базовый случай). Ниже приведен исходный код поиска наибольшей общей подпоследовательности.

```
def lcs(X, Y):
    t = {}
    for i in range(len(X)+1): t[(i,0)] = ""
    for j in range(len(Y)+1): t[(0,j)] = ""

    for i, x in enumerate(X):
        for j, y in enumerate(Y):
            if x == y:
                t[(i+1,j+1)] = t[(i, j)] + x
            else:
                t[(i+1,j+1)] = max([t[(i, j+1)], t[(i+1, j)]], key = len)
    return t[(len(X), len(Y))]
```

Для инициализации требуется линейное время, а пара основных циклов выполняет по $O(n^2)$ итераций. Для внутреннего цикла необходимо время, пропорциональное LCS в наихудшем случае, потому что мы объединяем строки такой длины. Общее время выполнения равно $O(kn^2)$, где k – длина выходной строки.

Можно ли придумать способ уменьшения времени выполнения до $O(n^2)$? При этом потребуется другой вариант хранения решений подзадач.

Глава 11

Двоичный поиск

Двоичный поиск (binary search) – это классический алгоритм. В определенной степени его можно считать рекурсивным алгоритмом. При поиске элемента в отсортированном списке этот список разделяется на две половины, и процесс поиска продолжается в половине, предположительно содержащей искомый элемент, который можно найти посредством сравнения со средним элементом. Затем процедура поиска повторяется в уменьшенной наполовину части списка – это всего лишь один рекурсивный вызов.

```
def bs(L, item):
    if len(L) == 0: return False
    median = len(L) // 2
    if item == L[median]:
        return True
    elif item < L[median]:
        return bs(L[:median], item)
    else:
        return bs(L[median + 1:], item)
```

Хотя этот код корректен, он эффективен не настолько, как следовало бы. Для анализа этого кода воспользуемся методикой, применяемой для всех рекурсивных алгоритмов в этом учебном курсе. Будем подсчитывать все операции, выполняемые исследуемой функцией, за исключением рекурсивных вызовов. Затем построим дерево всех рекурсивных вызовов и добавим стоимость каждого вызова.

В рассматриваемом здесь примере время выполнения в наихудшем случае для алгоритма `bs` со списком длиной n равно n/2 плюс константа. Поэтому стоимость первого вызова равна n/2. Стоимость второго – n/4. Для третьего вызова стоимость составляет n/8. Суммирование этих стоимостей дает число, близкое к n. Поэтому принимаем линейное время для проверки на присутствие в списке. Можно было бы ускорить этот процесс с помощью однократного итерационного прохода по списку, используя функцию `in` (т. е. `list.__contains__`).

Если мы намерены ускорить поиск, то необходимо избавиться от всех операций вырезки. Попробуем лишь имитировать вырезку, но вместо нее будем передавать индексы, определяющие диапазон списка, в котором необходимо выполнить поиск. Ниже приводится первая попытка.

```
def bs(L, item, left = 0, right = None):
    if right is None: right = len(L)
    if right - left == 0: return False
    if right - left == 1: return L[left] == item
    median = (right + left) // 2
    if item < L[median]:
        return bs(L, item, left, median)
    else:
        return bs(L, item, median, right)
```

Следует отметить, что здесь пришлось выполнить небольшую дополнительную работу по установке параметров по умолчанию, чтобы оставалась возможность вызова этой функции в виде `bs(mylist, myitem)`. При этом подразумевается проверка переменной `right` на значение `None` и установка для нее значения длины списка, если это необходимо.

Как и ранее, при анализе этого рекурсивного алгоритма мы видим, что все операции выполняются за постоянное время, поэтому общее время выполнения будет пропорциональным общему числу рекурсивных вызовов. Дерево вызовов функции представляет собой одну цепочку длиной не более O(log n). (Почему именно log(n)? Это число возможных разделений n на половины, прежде чем оно уменьшится до 1.) Итак, мы видим, что асимптотическое время выполнения равно O(log n).

Выполняя этот анализ, мы заметили, что дерево вызовов функции представляет собой одну цепочку. Это называется линейной рекурсией (linear recursion). Здесь мы имеем особый случай, при котором функция напрямую возвращает результат рекурсивного вызова функции (самой себя). Это называется хвостовой рекурсией (tail recursion).

Вообще говоря, хвостовую рекурсию всегда можно заменить обычным циклом. Принцип заключается в простом обновлении переменных-параметров и замене рекурсивного вызова на инструкцию цикла. Вот как это выглядит в действии.

```
def bs(L, item):
    left, right = 0, len(L)
    while right - left > 1:
        median = (right + left) // 2
        if item < L[median]:
            right = median
        else:
            left = median
    return right > left and L[left] == item
```

Следует отметить, что это решение проще написать, чем то, которое было приведено в начале главы, а кроме того, оно, вероятно, немного быстрее. Требуется немного подумать, чтобы понять, почему оно корректно и почему оно выполняется за время O(log n).

11.1. Абстрактный тип данных «упорядоченный список»

- `add(item)` – добавляет элемент `item` в список.
- `remove(item)` – удаляет первый найденный экземпляр элемента `item` из списка. Если элемента `item` нет в списке, то генерируется исключение `ValueError`.
- `__getitem__(index)` – возвращает элемент с заданным индексом `index` из отсортированного списка. Эта операция также известна как выбор (selection).
- `__contains__(item)` – возвращает значение `True`, если в списке существует элемент, равный `item`.
- `__iter__` – возвращает итератор для упорядоченного списка, который передает элементы в отсортированном порядке.
- `__len__` – возвращает длину упорядоченного списка.

Ниже приведена весьма простая реализация абстрактного типа данных «упорядоченный список».

```
class OrderedListSimple:
    def __init__(self):
        self._L = []

    def add(self, item):
        self._L.append(item)
        self._L.sort()

    def remove(self, item):
        self._L.remove(item)

    def __getitem__(self, index):
        return self._L[index]

    def __contains__(self, item):
        return item in self._L

    def __len__(self):
        return len(self._L)

    def __iter__(self):
        return iter(self._L)
```

Это классический пример использования шаблона «обертка». Обычный список `list`, хранящий элементы, остается закрытым, поэтому можно постоянно поддерживать его в упорядоченном состоянии. Небольшой хитростью выглядит реализация метода `add` таким способом, но немного позже мы рассмотрим алгоритмы сортировки, которые могут обеспечить самую эффективную методику.

Единственным алгоритмом в этом наборе, который выглядит наиболее подходящим для немедленного улучшения, является метод `__contains__`. Несмотря на то что мы просто вызываем встроенный метод Python для проверки принадлежности к списку, есть некоторая надежда на повышение эффективности, поскольку нам известно, что список отсортирован.

Заменим встроенный метод алгоритмом двоичного поиска, который был реализован в начале главы.

```
from ds2.orderedlist import OrderedListSimple

class OrderedList(OrderedListSimple):
    def __contains__(self, item):
        left, right = 0, len(self._L)
        while right - left > 1:
            median = (right + left) // 2
            if item < self._L[median]:
                right = median
            else:
                left = median
        return right > left and self._L[left] == item
```

Также можно попытаться применить двоичный поиск для поиска индекса, по которому необходимо вставить новый узел (элемент), чтобы ускорить выполнение метода `add`. К сожалению, после нахождения индекса остается необходимость в линейном времени в наихудшем случае для вставки элемента в список по конкретному заданному индексу.

Глава 12

Сортировка

12.1. Алгоритмы сортировки, выполняемые за квадратичное время

Перед началом подробного рассмотрения конкретных алгоритмов сортировки попробуем ответить на более простой вопрос:

> Задан список L; определить, является ли L отсортированным, или нет.

После недолгих размышлений вы, вероятно, решите, что для ответа необходимо написать код, который мог бы выглядеть следующим образом.

```
def issorted(L):
    for i in range(len(L)-1):
        if L[i]>L[i+1]:
            return False
    return True

A = [1,2,3,4,5]
print(A, "is sorted:", issorted(A))

B = [1,4,5,7,2]
print(B, "is sorted:", issorted(B))
```

```
[1, 2, 3, 4, 5] is sorted: True
[1, 4, 5, 7, 2] is sorted: False
```

Этот код намного лучше, чем приведенный ниже исходный код в более явной форме.

```
def issorted_slow(L):
    for i in range(len(L)-1):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

В последнем фрагменте кода выполняется проверка правильного порядка каждой пары элементов. Первый фрагмент более интеллектуален в том, что проверяет только смежные элементы. Если список не отсортирован, то в нем непременно найдутся два смежных элемента, расположенные не в надлежащем порядке. Это утверждение истинно, так как отношения порядка являются транзитивными, т. е. если `a < b` и `b < c`, то `a < c`. Важно понимать, что сейчас мы используем это предположение, потому что в дальнейшем будем определять собственные способы упорядочения элементов, и нам потребуется уверенность в истинности того, что сортировка вообще имеет смысл (если `a < b < c < a`, то какой порядок сортировки является правильным?).

Используем метод `issorted` для написания алгоритма сортировки. Вместо возврата значения `False`, когда найдены два неупорядоченных элемента, мы просто изменим их порядок и продолжим выполнение. Назовем эту функцию сортировки (по причинам, которые скоро станут понятными) `dumbersort`.

```
def dumbersort(L):
    for i in range(len(L)-1):
        if L[i]>L[i+1]:
            L[i], L[i+1] = L[i+1], L[i]
```

Главная проблема этого кода заключается в том, что в действительности он не сортирует список.

```
L = [5,4,3,2,1]
dumbersort(L)
print(L)

[4, 3, 2, 1, 5]
```

Вероятно, этот код необходимо выполнить дважды или большее число раз. Можно было бы даже повторять выполнение этого алгоритма до тех пор, пока он не решит окончательно задачу сортировки.

```
def dumbsort(L):
    while (not issorted(L)):
        dumbersort(L)

L = [5,4,3,2,1]
dumbsort(L)
print(L)

[1, 2, 3, 4, 5]
```

Сможет ли этот алгоритм вообще отсортировать список?

Сколько раз должна выполняться функция `dumbersort` для конкретного списка?

После некоторых размышлений вы заметите, что если `n = len(L)`, то `n-1` выполнений цикла достаточно. Существует несколько способов доказать, что это верное утверждение. Можно видеть, что на каждой итерации цикла каждый элемент в списке перемещается как минимум на одну позицию ближе к своей конечной локации.

Другой способ убедиться в том, что этот алгоритм работает, – проверка того, что после однократного вызова функции `dumbersort(L)` самый большой элемент в любом случае переместится в конец списка и останется там при всех последующих вызовах. При втором вызове `dumbersort(L)` второй по величине элемент «всплывает, словно пузырек воздуха в воде» (bubble up), на второе с конца место в списке. Здесь мы обнаруживает то, что называется инвариантом (invariant), то, что является истинным каждый раз, когда мы достигаем определенного места в алгоритме. Для этого алгоритма инвариантом является то, что после `i` вызовов функции `dumbersort(L)` `i` последних элементов находятся в своих конечных (отсортированных) локациях в списке. Положительное свойство этого инварианта заключается в том, что если он сохраняется при `i = n`, то отсортирован весь список полностью, и мы можем сделать вывод, что алгоритм корректен.

В этот момент мы должны протестировать код и задуматься о его рефакторинге. Вспомним принцип DRY (*Don't Repeat Yourself* – «не повторяй своих ошибок»). Поскольку нам известно число итераций цикла, можно просто воспользоваться циклом `for`.

```
def bubblesort(L):
    for iteration in range(len(L)-1):
        for i in range(len(L)-1):
            if L[i]>L[i+1]:
                L[i], L[i+1] = L[i+1], L[i]

alist = [30, 100000,54,26,93,17,77,31,44,55,20]
bubblesort(alist)
print(alist)

[17, 20, 26, 30, 31, 44, 54, 55, 77, 93, 100000]
```

Теперь мы получили корректный алгоритм и можем достаточно легко определить границу времени его выполнения. Оно равно $O(n^2)$ – это алгоритм с квадратичным временем выполнения.

Мы кое-что упустили по сравнению с алгоритмом `dumbsort`, а именно: теперь мы не останавливаемся сразу, если список уже отсортирован. Вернем это полезное свойство. Воспользуемся флагом для проверки того, что все перестановки были сделаны.

```
def bubblesort(L):
    keepgoing = True
    while keepgoing:
        keepgoing = False
        for i in range(len(L)-1):
            if L[i]>L[i+1]:
                L[i], L[i+1] = L[i+1], L[i]
                keepgoing = True
```

Теперь, когда нам известен инвариант, приводящий к корректному алгоритму, возможно, мы могли бы пройти и в обратном направлении от инварианта к алгоритму. Напомню, что инвариантом является тот факт, что после `i` итераций

цикла i наибольших элементов находятся на своих конечных позициях. Можно написать алгоритм, который просто обеспечивает этот инвариант, выбирая самый большой среди первых n-i элементов и перемещая его на нужное место.

```
def selectionsort(L):
    n = len(L)
    for i in range(n-1):
        max_index=0
        for index in range(n - i):
            if L[index] > L[max_index]:
                max_index = index
        L[n-i-1], L[max_index] = L[max_index], L[n-i-1]
```

Есть еще один инвариант, который мы неявно учитывали ранее при работе с отсортированными списками. Напомню: там мы предположили, что у нас есть отсортированный список и добавлен один новый элемент, который должен быть перемещен в правильную позицию. Мы наблюдали несколько различных вариантов этого подхода. Можно превратить этот инвариант в еще один алгоритм сортировки. Для максимально возможного приближения к предыдущим попыткам будем говорить, что инвариантом является тот факт, что после i итераций i последних элементов в списке расположены в отсортированном порядке.

Вы видите различие между этим и предыдущим инвариантом?

Существует много способов, которыми можно было бы обеспечить этот инвариант. Сделаем это с помощью «пузырькового всплытия» элемента n-i в некоторую позицию на i-м шаге. Обратите внимание: это позиция не конечная, а та, которая соответствует инварианту.

```
def insertionsort(L):
    n = len(L)
    for i in range(n):
        for j in range(n-i-1, n-1):
            if L[j]>L[j+1]:
                L[j], L[j+1] = L[j+1], L[j]
```

Как и ранее, можно сделать этот алгоритм более быстрым, если список уже отсортирован (или почти уже отсортирован). Внутренний цикл останавливается сразу же после того, как элемент оказывается на правильном месте.

```
def insertionsort(L):
    n = len(L)
    for i in range(n):
        j = n - i - 1
        while j < n - 1 and L[j]>L[j+1]:
            L[j], L[j+1] = L[j+1], L[j]
            j+=1
```

Существует важный момент, о котором необходимо помнить в приведенном выше коде. Если j == n-1, то вычисление L[j] < L[j+1] должно привести к ошибке. Но в этом коде ошибка не возникает, потому что в выражении j < n-1

and L[j]>L[j+1] сначала вычисляется его первая часть. Как только вычисление j < n-1 дает результат False, исчезает необходимость в вычислении второй части оператора and. Вторая часть пропускается.

Вы часто будете видеть функцию insertionsort, написанную таким способом, который сохраняет отсортированную часть списка в его начале, а не в конце.

Сможете самостоятельно написать такую версию сортировки вставкой? Попробуйте.

12.2. Сортировка в Python

В Python есть две основные функции сортировки списка – sort() и sorted(). Различие между ними заключается в том, что первая сортирует сам исходный список, а вторая возвращает новый отсортированный список. Рассмотрим следующую простую демонстрацию работы этих функций.

```
X = [3,1,5]
Y = sorted(X)
print(X, Y)

X.sort()
print(X)
```

```
[3, 1, 5] [1, 3, 5] [1, 3, 5]
```

При работе с собственными классами может потребоваться сортировка элементов. Для обеспечения этой операции необходима возможность сравнения элементов. В приведенном ниже примере определяется класс Foo, который предоставляет собственный способ сравнения, определяя магический метод __lt__. В данном случае метод упорядочивает объекты Foo по их атрибуту b.

```
class Foo:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __lt__(self, other):
        return other.b < self.b

    def __str__(self):
        return "(%d, %d, %d)" % (self.a, self. b, self. c)

    def geta(self):
        return self.a
```

Теперь можно сгенерировать список случайных экземпляров Foo.

```
from random import randrange

L = [Foo(randrange(100),randrange(100), randrange(100)) for i in range(6)]
```

При сортировке этого списка, т. е. при вызове `L.sort()` операция сравнения использует метод `Foo.__lt__` для определения порядка элементов. Поэтому итоговый список должен быть отсортирован по атрибуту `b`.

```
L.sort()
for foo in L:
    print(foo)
```

Если ключевая функция `key` передается в метод `sort`, то в итоговом упорядоченном списке `x` будет находиться перед `y`, если `key(x) < key(y)`. В приведенном ниже примере мы используем `Foo.geta` как ключевую функцию, поэтому результат должен быть упорядочен по значению атрибута `a`. Следует отметить, что аргументом является сама функция (т. е. ее имя), а не результат вычисления (выполнения) этой функции.

```
L.sort(key = Foo.geta)
for foo in L:
    print(foo)
```

Если ключевая функция возвращает кортеж, то сортировка будет выполнена по его первому элементу, а при равенстве первых элементов – по последующим. Такой тип сортировки называется лексикографическим, потому что обеспечивает сортировку слов в алфавитном порядке.

Ниже приведен пример сортировки строк по их длине (от самой длинной до самой короткой) с использованием алфавитного порядка (регистр букв не учитывается), если длина одинакова.

```
strings = "here are Some sample strings to be sorted".split()

def mykey(x):
    return -len(x), x.upper()

print(sorted(strings, key=mykey))
```

```
['strings', 'sample', 'sorted', 'here', 'Some', 'are', 'be', 'to']
```

Глава 13

Сортировка методом «разделяй и властвуй»

В предыдущей главе мы рассматривали некоторые простые алгоритмы сортировки, выполняемые за квадратичное время, – `bubblesort`, `selectionsort` и `insertionsort`. Все эти алгоритмы рассматривались с точки зрения корректности. Необходимо было получить способы доказательства того, что результатом действительно является отсортированный список, и только после этого применялись некоторые методы оптимизации (такие как ранний останов). Мы использовали идею инварианта. В частности, это был инвариант цикла (loop invariant), являющийся истинным на каждой итерации конкретного цикла. Для начала был принят следующий инвариант: после `i` итераций `i` последних элементов расположены в отсортированном порядке.

Инварианты помогают обосновать исходный код тем же способом, который применялся для рекурсивных алгоритмов. После того как код написан, мы предполагаем, что алгоритм работает на небольших примерах и что инвариант цикла сохраняется с предыдущей итерации цикла.

Теперь наша цель – написать более быстрый алгоритм сортировки.

«Разделяй и властвуй» (divide and conquer) – это парадигма проектирования алгоритмов. Обычно она состоит из трех (плюс одной) частей. Первая часть – разделение (divide) задачи на две и более частей. Вторая часть – этап властвования (conquer), на котором задача решается по частям. Третья – этап объединения (combine), где решения отдельных частей объединяются в решение всей задачи в целом.

Описание этих частей практически напрямую приводит к рекурсивным алгоритмам, т. е. к использованию рекурсии в части властвования (conquer). Еще одна часть, которая присутствует во многих подобных алгоритмах, – это базовый случай (base case), который можно ожидать в рекурсивном алгоритме. Именно здесь вы доводите входные данные до столь малого размера, что их невозможно разделить.

13.1. Сортировка слиянием

Наиболее очевидным применением парадигмы «разделяй и властвуй» к задаче сортировки является алгоритм сортировки слиянием (merge sort). В этом алгоритме вся самая трудная работа выполняется на шаге слияния (объединения).

```
def mergesort(L):
    # Базовый случай.
    if len(L) < 2:
        return

    # Разделяй.
    mid = len(L) // 2
    A = L[:mid]
    B = L[mid:]

    # Властвуй.
    mergesort(A)
    mergesort(B)

    # Объединяй.
    merge(A, B, L)

def merge(A, B, L):
    i = 0 # Индекс в A.
    j = 0 # Индекс в B.
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            L[i+j] = A[i]
            i = i + 1
        else:
            L[i+j] = B[j]
            j = j + 1

# Добавление всех оставшихся элементов, когда один из списков является пустым.
L[i+j:] = A[i:] + B[j:]
```

Самая последняя строка может показаться немного странной. В правой части инструкции присваивания выполняется конкатенация (объединение) оставшихся элементов из двух списков (из которых один должен быть пустым). Затем полученный объединенный список присваивается вырезке. Благодаря своим чудесным свойствам Python позволяет выполнять присваивание вырезке точно так же, как выполнялось бы присваивание по индексу.

Также можно использовать некоторую дополнительную логику в цикле, чтобы исключить последний шаг, хотя я обнаружил, что студенты не согласны с тем, что такой подход проще.

```
def merge(A, B, L):
    i, j = 0, 0
    while i < len(A) or j < len(B):
        if j == len(B) or (i < len(A) and A[i] < B[j]):
            L[i+j] = A[i]
            i = i + 1
        else:
            L[i+j] = B[j]
            j = j + 1
```

В приведенном выше коде составная инструкция `if` в основном полагается на прием, который называется сокращенной (или укороченной) схемой вычисления (short-circuited evaluation) логических выражений. Если имеется логическая операция, например `or`, и первый операнд дает результат `True`, то нет необходимости вычислять второй операнд, чтобы определить, что итоговым результатом будет `True`. Используя этот факт, Python даже не вычисляет второй операнд. Аналогично если имеется выражение с логическим оператором `and` и первый операнд дает значение `False`, то второй никогда не вычисляется. Ключом к запоминанию этой схемы является тот факт, что здесь порядок имеет важное значение. Выражение `(i < len(A) and A[i] < B[j]) or j == len(B)` логически равнозначно приведенному выше в исходном коде, но если использовать его, то будет сгенерировано исключение `IndexError` при `j == len(B)`.

Сокращенная схема вычисления не является частью особых магических свойств языка Python, это стандартное функциональное свойство большинства языков программирования.

13.1.1. Анализ

Анализ алгоритма `mergesort` будет выполняться способом, обычным для рекурсивных алгоритмов. Мы подсчитаем элементарные операции в методах без учета рекурсивных вызовов. Затем построим дерево рекурсивных вызовов и запишем стоимость каждого вызова функции в этом дереве. Так как размер входных данных изменяется от вызова к вызову, стоимость также изменяется. После этого просуммируем все полученные стоимости для получения общего времени выполнения.

Функция `merge` для двух списков полной длины с суммированием до n требует времени выполнения O(n). Причина в том, что нам нужно выполнить только сравнение и некоторое присваивание для каждого элемента, добавляемого в итоговый список (имеется n таких элементов).

В дереве рекурсивных вызовов вершина (или корень) имеет стоимость O(n). Следующий уровень содержит два вызова со списками длиной n/2. Уровнем ниже имеются четыре вызова со списками длиной n/4. При спуске по этому дереву каждый уровень i содержит 2^i вызовов с соответствующими списками длиной $n/2^i$. Удобный прием суммирования всех стоимостей основан на следующем наблюдении: сумма стоимостей для всех узлов на каждом рассматриваемом уровне равна O(n). Существует приблизительно $\log_2 n$ уровней. (Сколько раз можно делить n на 2 до тех пор, пока не получим 1?)

Итак, мы имеем log n уровней со стоимостью каждого O(n), следовательно, общая стоимость равна O(n log n).

13.1.2. Итераторы слияния

Операция слияния – это еще один пример использования некоторой структуры для наших данных (это два списка, которые сортируются непосредственно) для быстрого выполнения некоторой операции (поиск минимального общего элемента за постоянное время). Но после того, как вы уже написали достаточно много кода на Python, возможно, вы начинаете чувствовать, что делаете что-то неправильно, если приходится работать с огромным количеством индексов. По возможности гораздо лучше использовать итераторы. Мы возьмем эту задачу как пример, стимулирующий к более глубокому изучению итераторов.

Напомню, что в Python объект является итерируемым (iterable), если он содержит метод с именем `__itrer__`, который возвращает итератор, а итератор (iterator) – это объект, содержащий методы `__iter__` и `__next__`. Эти магические методы вызываются извне как `iter(my_iterable)` и `next(my_iterator)`. Чаще всего они используются вместе с ключевым словом `for` либо в циклах `for`, либо в выражениях-генераторах (generator expressions) для полного охвата содержимого.

```
class SimpleIterator:
    def __init__(self):
        self._count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._count < 10:
            self._count += 1
            return self._count
        else:
            raise StopIteration

iterator1 = SimpleIterator()
for x in iterator1:
    print(x)

iterator2 = SimpleIterator()
L = [2 * x for x in iterator2]
```

```
1
2
3
4
5
6
7
8
9
10
```

Итераторы представляют собой фундаментальный шаблон в объектно-ориентированном программировании и присутствуют в других языках программирования с несколько другими методами. Например, в некоторых языках программирования имеются итераторы, поддерживающие метод `hashnext`, который может заранее сообщать, существует ли очередной элемент для итеративного перехода. В стандартном итераторе Python вы просто пытаетесь получить следующий элемент, и если такой элемент отсутствует, то генерируется исключение `StopIteration`. Это пример применения принципа Python «проще попросить прощения, чем разрешения» (*easier to ask forgiveness than permission* – EAFP). Предварительная проверка напоминает просьбу о разрешении. Но эта методика может оказаться удобной. При таком подходе, возможно, также потребуется метод `peek`, возвращающий следующий элемент без перехода к нему.

Ниже показано, как можно реализовать эти методы в Python. Класс `BufferedIterator` создается из любого итератора. Он опережает пользователя на один шаг итерации и сохраняет следующий элемент в буфере.

```
class BufferedIterator:
    def __init__(self, i):
        self._i = iter(i)
        self._hasnext = True
        self._buffer = None
        self._advance()

    def peek(self):
        return self._buffer

    def hasnext(self):
        return self._hasnext

    def _advance(self):
        try:
            self._buffer = next(self._i)
        except StopIteration:
            self._buffer = None
            self._hasnext = False

    def __iter__(self):
        return self

    def __next__(self):
        if self.hasnext():
            output = self.peek()
            self._advance()
            return output
        else:
            raise StopIteration
```

Можно использовать этот класс `BufferIterator` для реализации другого итератора, который принимает два итератора и объединяет их при помощи операции `merge` по алгоритму `mergesort`.

```
def merge(A, B):
    a = BufferedIterator(A)
    b = BufferedIterator(B)
    while a.hasnext() or b.hasnext():
        if not a.hasnext() or (b.hasnext() and b.peek() < a.peek()):
            yield next(b)
        else:
            yield next(a)
```

Этот итератор полностью отличается от показанного выше. Прежде всего он не является классом, а выглядит как обычный метод. Это хитроумный прием Python для определения простого итератора. В этом методе применяются инструкции `yield` вместо инструкций `return`. Python распознает их и создает объект генератора (generator object). Вызов `merge` возвращает объект типа `generator`. Объект типа `generator` – это итератор. Самый простой способ понимания генератора – интерпретация его как метода, который делает паузу каждый раз, когда доходит до инструкции `yield`, и возобновляет выполнение при запросе следующего элемента. Здесь магия объясняется пониманием того, что в действительности это упакованный в итератор объект, который возвращается из метода.

Показанный выше итератор `merge` можно использовать для написания новой версии алгоритма сортировки слиянием `mergesort`.

```
def mergesort(L):
    if len(L) > 1:
        m = len(L) // 2
        A, B = L[:m], L[m:]
        mergesort(A)
        mergesort(B)
        L[:] = merge(A, B)
```

13.2. Быстрая сортировка

В коде реализации алгоритма `mergesort` выполняется слишком много операций вырезки. Напомню, что операция вырезки создает копию. В конечном счете мы попытаемся получить версию, не злоупотребляющую вырезкой. Алгоритм сортировки, который просто переупорядочивает элементы в одном списке, называется алгоритмом сортировки на месте (in-place sorting algorithm). Для начала рассмотрим простейшую версию алгоритма быстрой сортировки `quicksort`. Одним из способов обоснования алгоритма quicksort является необходимость максимально возможного упрощения шага объединения. Напомню, что в алгоритме сортировки слиянием `mergesort` большая часть наших умственных усилий была направлена на выполнение объединения. В алгоритме быстрой сортировки `quicksort` самой трудной частью является разделение.

```
def quicksorted(L):
    # Базовый случай.
    if len(L) < 2:
        return L[:]

    # Разделяй.
    pivot = L[-1]
    LT = [e for e in L if e < pivot]
    ET = [e for e in L if e == pivot]
    GT = [e for e in L if e > pivot]

    # Властвуй.
    A = quicksorted(LT)
    B = quicksorted(GT)

    # Объединяй.
    return A + ET + B
```

Создадим версию сортировки на месте. Для этого необходимо исключить создание новых списков и объединение их в конце процесса.

```
def quicksort(L, left = 0, right = None):
    if right is None:
        right = len(L)

    if right - left > 1:
        # Разделяй.
        mid = partition(L, left, right)

        # Властвуй.
        quicksort(L, left, mid)
        quicksort(L, mid+1, right)

        # Объединяй.
        # Здесь ничего делать не нужно.

def partition(L, left, right):
    pivot = right - 1
    i = left          # Индекс в левой половине.
    j = pivot - 1     # Индекс в правой половине.

    while i < j:
        # Перемещение i в позицию элемента >= L[pivot].
        while L[i] < L[pivot]:
            i = i + 1

        # Перемещение j в позицию элемента < L[pivot].
        while i < j and L[j] >= L[pivot]:
            j = j - 1

        # Обмен местами элементов i и j, если i < j.
        if i < j:
            L[i], L[j] = L[j], L[i]
```

```
    # Размещение центрального элемента в надлежащем месте.
    if L[pivot] <= L[i]:
        L[pivot], L[i] = L[i], L[pivot]
        pivot = i

    # Возврат индекса центрального элемента.
    return pivot

# Простой тест, чтобы убедиться, что алгоритм работает.
L = [5,2,3,1,4]
quicksort(L)
print(L)

[1, 2, 3, 4, 5]
```

Ниже приведена вторая реализация. Она использует (закрытую) вспомогательную функцию вместо параметров по умолчанию для обработки первоначального вызова. Полезно рассмотреть эти две различные реализации одной и той же функции и сравнить разнообразные критерии, по которым мы могли бы выбрать один из двух вариантов.

Основное отличие исходного кода, приведенного ниже, заключается в том, что он использует случайно выбираемый центральный элемент вместо постоянного выбора самого последнего элемента. Следует отметить, что «случайный» (random) – это не то же самое, что «произвольный» (arbitrary). Мы используем генератор случайных чисел для выбора в списке элемента, который будет центральным.

Это не просто косметическое изменение. Обратите внимание: в приведенном ниже коде генерируется ошибка (исключение).

```
L = list(reversed(range(1000)))

quicksort(L)

  RecursionError: maximum recursion depth exceeded in comparison
                # При сравнении превышен предел максимальной глубины рекурсии
```

Ниже приведен новый исходный код.

```
from random import randrange

def quicksort(L):
    _quicksort(L, 0, len(L))

def _quicksort(L, left, right):
    if right - left > 1:
        mid = partition(L, left, right)
        _quicksort(L, left, mid)
        _quicksort(L, mid+1, right)
```

```
def partition(L, left, right):
    pivot = randrange(left, right)
    L[pivot], L[right -1] = L[right -1], L[pivot]
    i, j, pivot = left, right - 2, right - 1
    while i < j:
        while L[i] < L[pivot]:
            i += 1
        while i < j and L[j] >= L[pivot]:
            j -= 1
        if i < j:
            L[i], L[j] = L[j], L[i]
    if L[pivot] <= L[i]:
        L[pivot], L[i] = L[i], L[pivot]
        pivot = i
    return pivot
```

Глава 14

Выбор

Рассмотрим следующую задачу.

Задан список чисел, найти средний элемент.

Напомню, что средний элемент (median element) (медиана) – это элемент, который должен находиться в середине, если список отсортирован. Поэтому следующий код должен быть корректным.

```
def median(L):
    L.sort()
    return L[len(L) // 2]
```

Этот код переупорядочивает список, и мы должны решить, правильно ли он работает с конкретным заданным здесь списком. Если используется быстрый алгоритм сортировки, то можно ожидать, что он будет выполнен за время O(n log n).

Можно ли улучшить этот код? Как насчет алгоритма с линейным временем выполнения?

Сначала рассмотрим задачу, связанную с поставленной выше. Если наша цель – найти второй наибольший элемент в списке, то это можно сделать за линейное время, как показано ниже.

```
def secondsmallest(L):
    a, b = None, None
    for item in L:
        if a is None or item <= b:
            a, b = item, a
        elif b is None or item <= a:
            b = item
    return b
```

Мы просто предусмотрели переменную для хранения одного из двух наибольших значений, которые мы обнаружили до настоящего момента, и обновляем их, когда рассматриваем новые элементы. Вероятно, можно было бы сделать то же самое и для третьего наибольшего значения и т. д., но код стал бы

запутанным и медленным, особенно если требуется всего лишь найти средний элемент.

Для вычисления среднего элемента за линейное время потребуется другая методика. Попробуем стратегию «разделяй и властвуй». Чтобы сделать эту работу, воспользуемся методом, который часто применяется, когда мы имеем дело с рекурсией:

> Для решения задачи с помощью рекурсии иногда проще решить более трудную задачу.

Поэтому вместо решения задачи поиска среднего элемента напрямую мы будем решать задачу выбора (selection problem), определенную ниже.

> Задача выбора: задан список чисел и число k; найти k-е наименьшее число в этом списке.

14.1. Алгоритм quickselect

Для решения задачи выбора можно воспользоваться методикой «разделяй и властвуй», производной от алгоритма быстрой сортировки `quicksort`. Идея этой методики состоит в разделении списка на части и рекурсивного выбора в той или иной части. В отличие от `quicksort` нам нужно выполнить поиск только в одной половине списка (как при двоичном поиске). (Следует отметить, что наименьший элемент будет возвращен для k = 1, но не для k = 0, поэтому здесь существует несколько базовых случаев, в которых мы прибавляем единицу к индексу центрального элемента.)

```
from ds2.sorting.quicksort import partition

def quickselect(L, k):
    return _quickselect(L, k, 0, len(L))

def _quickselect(L, k, left, right):
    pivot = partition(L, left, right)
    if k <= pivot:
        return _quickselect(L, k, left, pivot)
    elif k == pivot + 1:
        return L[pivot]
    else:
        return _quickselect(L, k, pivot + 1, right)
```

Как и в алгоритме `quicksort`, мы используем случайный (рандомизированный) центральный элемент, поэтому можно ожидать исключение постоянной части списка на каждом новом шаге. Но, в отличие от `quicksort`, здесь мы не будем выполнять рекурсивный вызов на обеих сторонах. В результате увидим, что среднее время выполнения равно всего лишь O(n). Это означает, что выбор можно выполнить быстрее, чем сортировку, что не лишено смысла, но совсем не очевидно, как именно сделать это. Формальный анализ будет проведен в следующем разделе.

14.2. Анализ

Алгоритм `quickselect` представляет собой рандомизированный алгоритм (randomized algorithm).

В наихудшем случае время выполнения неприемлемое. Если мы пытаемся выбрать наибольший элемент и все центральные элементы попадают на наименьшие значения, то время выполнения будет квадратичным.

Вместо этого проанализируем ожидаемое время выполнения. Термин «ожидаемое» (expected) взят из теории вероятностей, и он означает среднее значение по всем случайным выборам, которые выполняет алгоритм.

Пусть задан список из n чисел, и функция разделения выбирает случайный элемент в качестве центрального (pivot). Мы говорим, что центральный элемент выбран удачно, если он в диапазоне индексов от n/4 до 3n/4. То есть при случайном выборе существует 50%-ная вероятность выбора удачного центрального элемента.

При каждом рекурсивном вызове мы получаем список меньшего размера. Пусть n_i – размер списка на i-м рекурсивном вызове, тогда $n = n_0 > n_1 > \ldots > n_k$, где k – (неизвестное) число рекурсивных вызовов. На каждом шаге существует вероятность 1/2 выбора удачного центрального элемента, поэтому ожидаемое значение n_i может быть ограничено, как показано ниже.

$$E\left[n_i\right] \le (1/2)\left(\frac{3E\left[n_{i-1}\right]}{4}\right) + (1/2)\left(E\left[n_{i-1}\right]\right) = \left(\frac{7}{8}\right)E\left[n_{i-1}\right].$$

Это ограничение объединяет два граничных условия, каждое из которых определяет как минимум половину времени: первое для случая удачного выбора центрального элемента, где $n_i \leqslant 3n_{i-1}/4$, второе для случая неудачного выбора центрального элемента, где как минимум $n_i \leqslant n_{i-1}$. Действительное ожидаемое значение будет меньшим, но эта верхняя граница вполне достаточна.

Повторное использование этого факта с многократным объединением его с самим собой (рекурсивно) для меньших значений дает следующую границу относительно n.

$$E\left[n_i\right] \le \left(\frac{7}{8}\right)n_{i-1} \le \left(\frac{7}{8}\right)^2 n_{i-2} \le \left(\frac{7}{8}\right)^3 n_{i-3} \le \ldots \le \left(\frac{7}{8}\right)^i n.$$

Каждый рекурсивный вызов требует линейного времени, поэтому общее время выполнения равно

$$T(n) = \sum_{i=0}^{k} O\left(n_i\right) = O\left(\sum_{i=0}^{k} n_i\right).$$

Необходимо ограничить эту сумму n_is, но нам требуется ограничение только ожидаемой (средней) суммы. Самым важным фактом в общей оценке вероятности является линейность ожидаемых значений. Это означает, что сумма двух ожидаемых значений равна ожидаемому значению их суммы. Воспользуемся этим утверждением для завершения анализа.

$$E\left[T(n)\right] = O\left(\sum_{i=0}^{k} E\left[n_i\right]\right) \leq O\left(\sum_{i=0}^{k} \left(\frac{7}{8}\right)^i n\right) = O(n).$$

На последнем шаге получено общеизвестное равенство для геометрических рядов. Оно должно быть вам знакомо из курса математики, но если вы его забыли, то не смущайтесь и загляните в справочник.

14.3. В последний раз без рекурсии

Алгоритм `quickselect` – это пример линейной рекурсии. При каждом вызове функции выполняется еще один. Это также является хвостовой рекурсией (tail recursion), потому что один рекурсивный вызов – это последняя операция перед возвратом. Поэтому, как мы уже неоднократно видели раньше, есть возможность исключить рекурсию и заменить ее одним циклом.

Исходный код этой версии приведен ниже, но вы, вероятно, захотите попытаться написать его самостоятельно, чтобы убедиться в полном понимании алгоритма и процесса исключения хвостовой рекурсии.

```
from ds2.sorting.quicksort import partition

def quickselect(L, k):
    left, right = 0, len(L)
    while left < right:
        pivot = partition(L, left, right)
        if k <= pivot:
            right = pivot
        elif k == pivot + 1:
            return L[pivot]
        else:
            left = pivot + 1
    return L[left]
```

Пока разность между левой и правой частью сокращается с постоянным множителем для каждой пары центральных элементов, мы можем быть уверены в отсечении достаточной части диапазона поиска, чтобы получить алгоритм с линейным временем.

14.4. Резюме стратегии «разделяй и властвуй»

Существует три основных класса алгоритмов и методов анализа «разделяй и властвуй», которые мы рассмотрели для списков.

Первый класс, вероятно, самый простой, как, например, двоичный поиск, где каждый рекурсивный шаг требует постоянного времени и выполняет один рекурсивный вызов для списка, который уменьшается постоянное число раз. Общее время выполнения в этих случаях пропорционально глубине рекурсии O(log n).

Второй класс рекурсивных алгоритмов «разделяй и властвуй» мы наблюдали при двоичной рекурсии, связанной с сортировкой. Здесь время выполнения

линейное плюс время на рекурсивные вызовы в укорачивающихся списках, общая длина которых равна O(n). В этих случаях, поскольку глубина рекурсии равна O(log n), общее время выполнения составляет O(n log n).

В этой главе рассматривался третий класс алгоритмов «разделяй и властвуй» – алгоритм выбора `quickselect`. Здесь, как и в алгоритмах сортировки, время выполнения линейное плюс стоимость рекурсивных вызовов. Но в отличие от двоичного поиска здесь выполняется только один рекурсивный вызов. В этих случаях, поскольку подзадачи сокращаются в размере с постоянным коэффициентом на каждом шаге, время выполнения равно O(n).

14.5. Замечание о дерандомизации

Алгоритм, не использующий фактор случайности, называется детерминированным (deterministic). До сих пор все рассматриваемые в этой книге алгоритмы были детерминированными, поэтому не было необходимости в определении этого различия. Задачу выбора можно решить за линейное время в наихудшем случае без использования фактора случайности.

Если вы помните, алгоритм `quickselect` зависит от взятия приблизительного среднего значения для того, чтобы сокращать длину списка с постоянным коэффициентом. Поэтому если у вас есть алгоритм поиска среднего значения за линейное время, то можно воспользоваться им для выбора центральных элементов в алгоритме `quickselect`, чтобы получить алгоритм, выполняемый за линейное время.

Это должно показаться немного неверным. Эта глава начиналась с утверждения о том, что можно решить задачу нахождения среднего значения, используя алгоритм выбора. Теперь мы намереваемся решить задачу выбора, используя алгоритм поиска среднего значения. Не попадаем ли мы в ситуацию с парадоксом о первичности курицы и яйца?

Этот алгоритм иногда называют алгоритмом медианы медиан (median of medians). Более подробную информацию о нем можно найти в интернете.

Глава 15

Отображения и хеш-таблицы

Отображение (mapping) – это связь (ассоциация) между двумя множествами сущностей (вещей, объектов и т. п.). Отображение связывает значение с ключом. Такие связанные пары называют парами ключ–значение (key-value pairs). Ключи обязательно должны быть неповторяющимися, поэтому с конкретным ключом может быть связано только одно значение.

В Python стандартным встроенным типом данных для отображений является словарь (`dict`). Этот тип отображений также используется внутри Python для связывания имен переменных (строк) с объектами. В соответствии с системой записи для словарей мы должны записывать выражения в следующей форме: `d[некоторый_ключ] = некоторое_значение`. Это создает новую пару ключ–значение, если `некоторый_ключ` отсутствует в словаре, или перезаписывает ранее существующую пару с этим ключом.

Не все языки программирования имеют встроенный тип данных для отображений. На короткое время сделаем вид, что в Python нет доступного для нас словаря, и подробно рассмотрим процесс самостоятельной реализации отображения. Это позволит решить одну из главных нерешенных загадок из предыдущей части нашего учебного курса:

> Почему операции доступа и присваивания значения в словаре требуют всего лишь постоянного времени?

15.1. Абстрактный тип данных «отображение»

Отображение (mapping) – это набор (множество) пар ключ–значение, таких что ключи являются неповторяющимися (т. е. ни одна из пар не содержит одинаковый ключ). Отображение поддерживает следующие методы:

- `get(k)` – возвращает значение, связанное с ключом `k`. Если заданного ключа нет в отображении, то обычно генерируется исключение `KeyError`;
- `put(k, v)` – добавляет пару ключ–значение `(k, v)` в отображение.

Это две основные операции. Именно они создают отображение и, вообще говоря, реализованы как `__getitem__` и `__setitem__` в Python для поддержки уже знакомой нам нотации с квадратными скобками. Пока не будем заниматься излишними подробностями. В некоторых последующих реализациях мы определим несколько другие условия для ключей.

15.2. Минимальная реализация

Ниже приведен весьма упрощенный метод использования списка (`list`) в качестве отображения. Начнем с небольшого класса для хранения пар ключ–значение, затем напишем два метода для реализации извлечения и записи пар в отображение.

```
# mapping/mapping.py

class Entry:
    def __init__(self, key, value):
        self.key = key
        self.value = value

    def __str__(self):
        return str(self.key) + " : " + str(self.value)

def mapput(L, key, value):
    for e in L:
        if e.key == key:
            e.value = value
            return
    L.append(Entry(key, value))

def mapget(L, key):
    for e in L:
        if e.key == key:
            return e.value
    raise KeyError

m = []
mapput(m, 4, 'five')
mapput(m, 1, 'one')
mapput(m, 13, 'thirteen')
mapput(m, 4, 'four')
assert(mapget(m, 1) == 'one')
assert(mapget(m, 4) == 'four')
```

В этот момент кажется, что единственным преимуществом структуры словаря `dict` является то, что он предоставляет удобный синтаксис для добавления и извлечения записей. Существуют и другие преимущества, но мы будем знакомиться с ними постепенно при попытках самостоятельного создания отображения.

Сначала поместим нашу новую структуру данных в класс. Это позволит инкапсулировать список более низкого уровня, чтобы пользователи случайно не испортили его, например добавляя в него элементы напрямую вместо того, чтобы использовать метод `put`. Мы должны защищать пользователей от самих себя, особенно если существуют свойства структуры, для которых необходима поддержка корректности. В данном случае требуется уверенность в том, что ключи остаются неповторяющимися.

```
from ds2.mapping import Entry

class ListMappingSimple:
    def __init__(self):
        self._entries = []

    def put(self, key, value):
        for e in self._entries:
            if e.key == key:
                e.value = value
                return
        self._entries.append(Entry(key, value))

    def get(self, key):
        for e in self._entries:
            if e.key == key:
                return e.value
        raise KeyError
```

Это неплохое начало. Класс поддерживает методы `get` и `put`, и этого достаточно, чтобы быть отображением, но желательно иметь более интересные методы для того, чтобы структура стала действительно полезной. Расширим этот АТД с помощью дополнительных функциональных свойств.

15.3. Расширенный абстрактный тип данных «отображение»

Как и для любого набора данных, возможно, потребуются такие методы, как `__len__`, `__contains__` и/или различные итераторы.

Стандартным поведением итераторов в словарях является итеративный проход по ключам. Альтернативные варианты итераторов предоставляются для прохода по значениям или по парам ключ–значение, как по кортежам. Для объекта типа `dict` это делается, как показано ниже.

```
d = {'key1': 'value1', 'key2': 'value2'}

for k in d:
    print(k)

for v in d.values():
    print(v)

for k, v in d.items():
    print(k, v)
```

```
key1
key2
value1
value2
key1 value1
key2 value2
```

Добавим аналогичную функциональность в наш АТД «отображение». Для этого расширенный АТД «отображение» должен содержать следующие методы (с переименованием методов `get` и `put` в магические методы Python):

- `__getitem__(k)` – возвращает значение, связанное с ключом `k`. Если заданного ключа нет в отображении, то обычно генерируется исключение `KeyError`;
- `__setitem__(k, v)` – добавляет пару ключ–значение `(k, v)` в отображение;
- `remove(k)` – удаляет запись с ключом `k`, если она существует;
- `__len__` – возвращает число ключей в отображении;
- `__contains__(k)` – возвращает `True`, если отображение содержит пару с ключом `k`;
- `__iter__` – возвращает итератор по ключам в отображении;
- `values` – возвращает итератор по значениям в отображении;
- `items` – возвращает итератор по парам ключ–значение (как по кортежам) в отображении;
- `__str__` – возвращает представление отображения в виде строки.

Очень важно вспомнить самое начало этого учебного курса, где было отмечено, что класс `dict` не является последовательным набором данных. То есть порядок элементов не имеет никакого значения; более того, вы не должны делать никаких предположений о порядке расположения пар. Не следует даже предполагать, что порядок будет сохраняться между двумя итерациями в одном словаре. Это же предупреждение актуально и для реализуемых нами отображений, и мы увидим, что в возможности изменения порядка хранения пар заключается секрет чудесно быстрого времени выполнения. Но в первой реализации элементы хранятся в фиксированном порядке, потому что для этого используется обычный список (`list`).

```
from ds2.mapping import Entry

class ListMapping:
    def __init__(self):
        self._entries = []

    def put(self, key, value):
        e = self._entry(key)
        if e is not None:
            e.value = value
        else:
            self._entries.append(Entry(key, value))

    def get(self, key):
        e = self._entry(key)
        if e is not None:
            return e.value
        else:
            raise KeyError
```

```
    def remove(self, key):
        e = self._entry(key)
        if e is not None:
            self._entries.remove(e)

    def _entry(self, key):
        for e in self._entries:
            if e.key == key:
                return e
        return None

    def __str__(self):
        return "{" + ", ".join(str(e) for e in self._entries) + "}"

    def __len__(self):
        return len(self._entries)

    def __contains__(self, key):
        if self._entry(key) is None:
            return False
        else:
            return True

    def __iter__(self):
        return (e.key for e in self._entries)

    def values(self):
        return (e.value for e in self._entries)

    def items(self):
        return ((e.key, e.value) for e in self._entries)

    __getitem__ = get
    __setitem__ = put
```

Обратите внимание: я воспользовался этой возможностью, чтобы исключить некоторое дублирование в методах `get` и `put`.

15.4. Это слишком медленно

Нашей целью является создание точно таких же операций, выполняемых за постоянное время, как и в классе словаря `dict`. Но сейчас мы очень далеки от этого. В настоящий момент требуется линейное время для выполнения операций `get`, `put` и проверки присутствия элемента в отображении. Для улучшения необходима новая идея. Класс `ListMapping` требует линейного времени, потому что он вынужден выполнять итеративный проход по списку. Эту процедуру можно было бы ускорить, если бы имелось множество коротких списков вместо одного длинного. Тогда нам бы потребовался только быстрый способ определения того короткого списка, в котором нужно выполнить поиск или обновление.

Попробуем хранить список объектов `ListMapping`. Для каждого ключа `k` необходимо вычислить индекс правильного объекта `ListMapping` с этим ключом `k`.

Такие объекты `ListMapping` часто называют контейнерами или корзинами (buckets). Этот термин возвращает нас к возможности быстрого группирования элементов в контейнерах. В дальнейшем при поиске чего-либо в контейнере можно проверить все элементы, разумно предполагая, что их не очень много.

Это означает, что требуется целое число, т. е. индекс в списке контейнеров. Хеш-функция (hash function) принимает ключ и возвращает целое число. В большинстве классов Python реализован метод `__hash__`, который делает именно это. Мы можем воспользоваться этим методом для реализации простой схемы отображения, которая улучшает работу `ListMapping`.

```
from ds2.mapping import ListMapping

class HashMappingSimple:
    def __init__(self):
        self._size = 100
        self._buckets = [ListMapping() for i in range(self._size)]

    def put(self, key, value):
        m = self._bucket(key)
        m[key] = value

    def get(self, key):
        m = self._bucket(key)
        return m[key]

    def _bucket(self, key):
        return self._buckets[hash(key) % self._size]
```

Рассмотрим этот код более подробно. Он выглядит относительно простым, но скрывает некоторые тайны.

Сначала инициализатор (конструктор) создает список из 100 объектов `ListMapping`. Здесь они называются контейнерами `buckets`. Если ключи равномерно распределяются между контейнерами, то все будет выполняться почти в 100 раз быстрее. Если два ключа помещены в один контейнер, это называется коллизией (collision).

Методы `__getitem__` и `__setitem__` вызывают метод `_bucket` для получения одного из контейнеров по заданному ключу, затем просто используют методы `get` и `put` объекта `ListMapping`. Идея проста: иметь несколько списков отображений вместо одного, но теперь нужен быстрый способ определения, какой список должен использоваться. Функция `hash` возвращает целое число, вычисленное на основе значения переданного ей ключа. Коллизии будут зависеть от этой хеш-функции.

15.4.1. Сколько контейнеров мы должны использовать?

Число 100 выбрано абсолютно произвольно. При наличии весьма большого количества записей можно было бы добиться 100-кратного ускорения по сравнению с простым списком `ListMapping`, но не больше. Задержимся на минуту, чтобы удивиться тому, насколько мы можем быть жадными: всего лишь 100-кратное ускорение?

При увеличении числа записей имеет смысл использовать больше контейнеров. Для этого будем отслеживать число записей в отображении. Это позволит реализовать метод __len__, а также наращивать число контейнеров при необходимости. При росте числа записей мы можем периодически увеличивать количество контейнеров. Ниже приведен соответствующий исходный код.

```
from ds2.mapping import Entry, ListMapping

class HashMapping:
    def __init__(self, size = 2):
        self._size = size
        self._buckets = [ListMapping() for i in range(self._size)]
        self._length = 0

    def put(self, key, value):
        m = self._bucket(key)
        if key not in m:
            self._length += 1
        m[key] = value

        # Проверить, нужны ли дополнительные контейнеры.
        if self._length > self._size:
            self._double()

    def get(self, key):
        m = self._bucket(key)
        return m[key]

    def remove(self, key):
        m = self._bucket(key)
        m.remove(key)

    def __contains__(self, key):
        m = self._bucket(key)
        return key in m

    def _bucket(self, key):
        return self._buckets[hash(key) % self._size]

    def _double(self):
        # Сохранить ссылку на старые контейнеры.
        oldbuckets = self._buckets
        # Удвоить размер.
        self._size *= 2
        # Создать новые контейнеры.
        self._buckets = [ListMapping() for i in range(self._size)]
        # Добавить во все контейнеры новые записи.
        for bucket in oldbuckets:
            for key, value in bucket.items():
                # Идентифицировать новый контейнер.
                m = self._bucket(key)
                m[key] = value
```

```
    def __len__(self):
        return self._length

    def __iter__(self):
        for b in self._buckets:
            for k in b:
                yield k

    def values(self):
        for b in self._buckets:
            for v in b.values():
                yield v

    def items(self):
        for b in self._buckets:
            for k, v in b.items():
                yield k, v

    def __str__(self):
        # Следующая строка опасна. Она предоставляет доступ к закрытому атрибуту.
        # К счастью, скоро мы ее исключим.
        itemlist = [str(e) for b in self._buckets for e in b._entries]
        return "{" + ", ".join(itemlist) + "}"

    __getitem__ = get
    __setitem__ = put
```

15.4.2. Двойное хеширование

Самой интересной частью приведенного выше исходного кода является метод `_double`, который увеличивает количество контейнеров. Недостаточно просто добавить дополнительные контейнеры в список, потому что метод `_bucket`, используемый для поиска подходящего контейнера, зависит от их количества. Когда количество контейнеров изменяется, мы должны повторно вставить в контейнеры все записи, хранящиеся в отображении, чтобы их можно было найти при следующем выполнении метода `get`.

15.5. Вынос общих частей в суперкласс

Мы получили две различные реализации одного абстрактного типа данных. Несколько методов, реализованных в классе `ListMapping`, потребуются также и в классе `HashMapping`. Имеет смысл избегать дублирования общих частей этих двух (конкретных) структур данных. Правильным способом для достижения этой цели является наследование.

Это самый широко распространенный способ, которым наследование применяется в исходном коде. Два класса должны совместно использовать некоторый код, поэтому такой код выносится в суперкласс, от которого выполняется наследование обоих классов с возможностью совместного использования внутреннего кода предка.

Некоторые методы суперкласса мы предполагаем реализовать в подклассе. Это можно сделать, поместив требуемые методы в подкласс, но при их вызове генерируется исключение. При таком подходе ошибка будет возникать, только если подкласс не замещает эти методы.

Ниже приведен исходный код суперкласса.

```
class Mapping:
    # Класс-потомок должен реализовать этот метод.
    def get(self, key):
        raise NotImplementedError

    # Класс-потомок должен реализовать этот метод.
    def put(self, key, value):
        raise NotImplementedError

    # Класс-потомок должен реализовать этот метод.
    def __len__(self):
        raise NotImplementedError

    # Класс-потомок должен реализовать этот метод.
    def _entryiter(self):
        raise NotImplementedError

    def __iter__(self):
        return (e.key for e in self._entryiter())

    def values(self):
        return (e.value for e in self._entryiter())

    def items(self):
        return ((e.key, e.value) for e in self._entryiter())

    def __contains__(self, key):
        try:
            self.get(key)
        except KeyError:
            return False
        return True

    def __getitem__(self, key):
        return self.get(key)

    def __setitem__(self, key, value):
        self.put(key, value)

    def __str__(self):
        return "{" + ", ".join(str(e) for e in self._entryiter()) + "}"
```

Здесь много кода, но следует отметить, что в действительности подкласс должен реализовать только четыре метода: `get`, `put`, `__len__` и метод с именем `_entryiter`, который обеспечивает итерационный проход по записям. Последний метод является закрытым, потому что пользователь этого класса не должен получать доступ к объектам `Entry`. Пользователям предоставлены методы отображения для доступа к данным. Именно поэтому класс `Entry` является внутренним классом (определенным внутри класса `Mapping`).

Теперь класс `ListMapping` можно переписать, как показано ниже.

```
from ds2.mapping import Mapping, Entry

class ListMapping(Mapping):
    def __init__(self):
        self._entries = []

    def put(self, key, value):
        e = self._entry(key)
        if e is not None:
            e.value = value
        else:
            self._entries.append(Entry(key, value))

    def get(self, key):
        e = self._entry(key)
        if e is not None:
            return e.value
        else:
            raise KeyError

    def _entry(self, key):
        for e in self._entries:
            if e.key == key:
                return e
        return None

    def _entryiter(self):
        return iter(self._entries)

    def __len__(self):
        return len(self._entries)
```

Все магические методы, а также открытые итераторы и метод преобразования в строку реализованы в суперклассе. Подкласс содержит только те части, которые являются специфическими для этой реализации.

Класс `HashMapping` также можно переписать, как показано ниже.

```
from ds2.mapping import Mapping, ListMapping

class HashMapping(Mapping):
    def __init__(self, size = 100):
        self._size = size
        self._buckets = [ListMapping() for i in range(self._size)]
        self._length = 0

    def _entryiter(self):
        return (e for bucket in self._buckets for e in bucket._entryiter())

    def get(self, key):
        bucket = self._bucket(key)
        return bucket[key]
```

```
    def put(self, key, value):
        bucket = self._bucket(key)
        if key not in bucket:
            self._length += 1
        bucket[key] = value

        # Проверить, нужны ли дополнительные контейнеры.
        if self._length > self._size:
            self._double()

    def __len__(self):
        return self._length

    def _bucket(self, key):
        return self._buckets[hash(key) % self._size]

    def _double(self):
        # Сохранить старые контейнеры.
        oldbuckets = self._buckets
        # Реинициализация с увеличенным количеством контейнеров.
        self.__init__(self._size * 2)
        for bucket in oldbuckets:
            for key, value in bucket.items():
                self[key] = value
```

Глава 16

Деревья

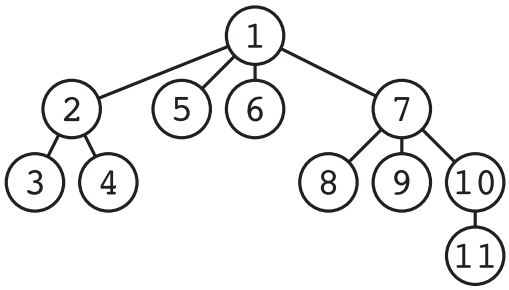

Рис. 16.1. Пример дерева

Дерево (tree) – это тип данных, идеально подходящий для представления иерархической структуры. Деревья состоят из узлов (nodes), которые имеют 0 или более прямых потомков (children) или узлов-потомков (child nodes). Узел называется родителем (parent) для своих потомков. Каждый узел имеет (не более) одного родителя. Если родители упорядочены некоторым образом, то мы получаем упорядоченное дерево (ordered tree). Мы будем рассматривать в основном деревья с корнями (rooted trees), т. е. имеющие один особый узел, называемый корнем (root) дерева. Корень – это единственный узел, который не имеет родителя. Узлы, не имеющие ни одного потомка, называются листьями (leaves) или узлами-листьями (leaf nodes).

Существует множество примеров иерархических (древовидных) структур:

- иерархии классов (предполагается одиночное наследование). Подкласс – это потомок, а суперкласс – родитель. В Python класс `object` является корнем;
- каталоги (папки) для хранения файлов на компьютере могут быть вложены в другие каталоги. Отношения родитель–потомок определяют вместимость;
- дерево рекурсивных вызовов функций при выполнении алгоритма типа «разделяй и властвуй». Здесь листья – это вызовы, в которых выполняются базовые случаи.

Несмотря на то что мы используем аналогию с семьей для многих терминов в деревьях, семейное (генеалогическое) дерево в действительности не является деревом, потому что в нем нарушено правило одного родителя.

При графическом изображении дерева принимается так называемое австралийское соглашение о размещении корня на вершине, а всех потомков – ниже. Это противоречит внешнему виду настоящих деревьев в природе, но полностью соответствует нашему общему представлению о большинстве других иерархий. К счастью, между вашими впечатлениями о деревьях в парке и представлением деревьев в этой книге будет очень мало смысловых совпадений, так что несоответствие аналогий не должно сбивать с толку.

16.1. Еще несколько определений

Путь (path) в дереве – это последовательность узлов, в которой каждый узел является потомком предыдущего. Мы говорим, что существует путь из x в y, если первым узлом (в последовательности) является x, а последним – y. Всегда существует путь из корня в каждый узел дерева. Длина пути (length of the path) – это число переходов или ребер (edges), которое на единицу меньше числа узлов в конкретном пути. Все потомки (descendants) узла x – это все узлы y, для которых существует путь из x в y. Если y – потомок x, то мы говорим, что x является предком (ancestor) y. Если задано дерево T и узел n в нем, то поддерево с корнем n (subtree rooted at n) – это дерево, корнем которого является узел n, и оно содержит всех потомков узла n.

Глубина (depth) узла – это длина пути к этому узлу из корня. Высота (height) дерева – максимальная глубина любого узла в дереве.

Все эти многочисленные определения необходимо усвоить сразу. Это поможет разобрать несколько наглядных примеров и определить, можно ли применить приведенные выше определения к примерам, т. е. идентифицировать родителя, прямых потомков, всех потомков и предков узла.

16.2. Деревья с точки зрения рекурсии

Дерево можно определить рекурсивно как корень с нулем или бóльшим числом прямых потомков, каждый из которых является деревом. Кроме того, корень хранит некоторые данные. Это будет удобно в начале и очень важно в дальнейшем, когда мы задумаемся о создании более полезных алгоритмов с применением деревьев, многие из которых проще всего описать рекурсивно. Однако имейте в виду, что наши объектно-ориентированные инстинкты сработают позже, когда мы разделим понятия деревьев и узлов. Наличие двух разных типов сущностей приведет нас к использованию двух разных классов для их представления (но не сейчас).

Можно воспользоваться списками для представления иерархической структуры, создавая списки списков. Чтобы сделать их немного более удобными (и полезными), мы также будем хранить некоторые данные в каждом узле дерева. При этом будет использоваться соглашение о том, что список является деревом, в котором первый элемент – данные, а все прочие элементы – прямые потомки (каждый должен быть списком).

Ниже приведен простой пример.

```
['a', ['p'], ['n'], ['t']]
```

Это дерево с символом `'a'`, хранящимся в корне. Корень имеет три прямых потомка, хранящих соответственно символы `'p'`, `'n'` и `'t'`. Ниже приведен немного более сложный пример, в котором предыдущий пример содержится как поддерево.

```
T = ['c', ['a', ['p'], ['n'], ['t']], ['o', ['n']]]
```

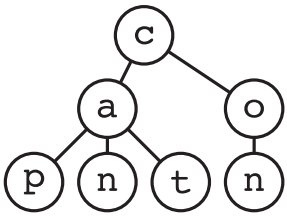

Рис. 16.2. Пример дерева, хранящего в узлах данные

Все узлы этого дерева можно вывести с помощью показанного ниже кода.

```
def printtree(T):
    print(T[0])
    for child in range(1, len(T)):
        printtree(T[child])

printtree(T)
```

```
c
a
p
n
t
o
n
```

Когда я вижу код, подобный приведенному выше, мне хочется заменить его на что-то такое, что может работать с любым итерируемым набором данных. При этом должно быть полностью исключено использование индексов, а вместо них применяется итератор. Ниже приведен равнозначный код с использованием итератора.

```
def printtree(T):
    iterator = iter(T)
    print(next(iterator))
    for child in iterator:
        printtree(child)

printtree(T)
```

```
c
a
p
n
t
o
n
```

Здесь итератор используется для извлечения первого элемента списка, затем выполняется проход в цикле по остальным потомкам. Поэтому, когда мы входим в цикл `for`, итератор уже получил первое значение ранее и начнет со второго, т. е. с первого потомка.

16.3. Абстрактный тип данных «дерево»

Вся информация в дереве представлена в структуре «список списков» (list of lists). Но может оказаться, что такую структуру неудобно читать, записывать и работать с ней. Мы упакуем дерево в класс, который позволяет писать код, максимально приближенный к нашему представлению о деревьях. Как всегда, начнем с абстрактного типа данных (АТД), который описывает наши предположения об этой структуре данных и ее использовании. Описание АТД «дерево» (tree) приведено ниже.

- `__init__(L)` – инициализирует новое дерево заданным списком списков. Используется то же соглашение: первый элемент списка – это данные, все последующие элементы (если они существуют) – его прямые потомки.
- `height()` – возвращает высоту дерева.
- `__str__()` – возвращает строку, представляющую все дерево в целом.
- `__eq__(other)` – возвращает `True`, если данное дерево равно другому дереву `other`. Это означает, что они оба содержат одинаковые данные и их потомки равны (и расположены в одинаковом порядке).
- `__contains__(k)` – возвращает `True`, если и только если дерево содержит данные `k` либо в корне, либо в одном из его потомков. Иначе возвращает `False`.
- `preorder()` – возвращает итератор по данным в дереве, который выдает значения в соответствии с прямым порядком обхода (preorder traversal) дерева.
- `postorder()` – возвращает итератор по данным в дереве, который выдает значения в соответствии с обратным порядком обхода (postorder traversal) дерева.
- `__iter__()` – псевдоним для метода `preorder`.
- `layerorder()` – возвращает итератор по данным в дереве, который выдает значения в соответствии с поуровневым порядком обхода (layer order traversal) дерева.

16.4. Реализация

```
class Tree:
    def __init__(self, L):
        iterator = iter(L)
        self.data = next(iterator)
        self.children = [Tree(c) for c in iterator]
```

Инициализатор (конструктор) принимает представление дерева в виде списка списков как входные данные. Объект `Tree` имеет два атрибута: `data` хранит данные, связанные с узлом, а `children` хранит список объектов `Tree`. Рекурсивное свойство этого дерева очевидно из способа, которым потомки генерируются как объекты `Tree`. Это определение не позволяет создать пустое дерево (т. е. дерево без узлов).

Перепишем функцию вывода так, чтобы она работала с экземпляром класса `Tree`.

```
def printtree(T):
    print(T.data)
    for child in T.children:
        printtree(child)

T = Tree(['a', ['b', ['c', ['d']]],['e',['f'], ['g']]])
printtree(T)
```

```
a
b
c
d
e
f
g
```

Это весьма часто применяемый шаблон для алгоритмов, работающих с деревьями. В нем содержатся две части: одна часть работает непосредственно с данными, другая применяет эту же функцию рекурсивно к потомкам.

Одним неприятным свойством этого кода является то, что, хотя он и выводит данные, но ничего не сообщает о структуре дерева. Было бы более удобно, если бы использовался сдвиг вправо для обозначения глубины узла, из которого выводятся данные. Оказывается, что сделать это не слишком трудно, поэтому мы реализуем такую возможность в методе `__str__`.

```
def __str__(self, level = 0):
    treestring = "  " * level + str(self.data)
    for child in self.children:
        treestring += "\n" + child.__str__(level + 1)
    return treestring

T = Tree(['a', ['b', ['c', ['d']]],['e',['f'], ['g']]])
print(str(T))
```

```
a
  b
    c
      d
  e
    f
    g
```

Чтобы «увидеть» структуру дерева в этом выводе, вы находите родителя узла, отыскав самую нижнюю строку над этим узлом, которая находится на другом (меньшем) уровне отступа. Для этого у вас есть некоторая подготовка: таким способом вы визуально анализируете содержимое файла с кодом Python, чтобы понять его блочную структуру.

Хотя приведенный выше код кажется работающим, на самом деле он делает нечто ужасное. Он формирует строку, итеративно добавляя дополнительные строки (т. е. применяет конкатенацию). При этом многократно повторяется копирование некоторой части строки для каждого узла дерева. Вместо этого следовало бы просто сохранить соответствующий список деревьев, а затем присоединить их к строке в одном заключительном действии перед возвратом. Для этого полезно использовать вспомогательный метод, который принимает уровень и текущий список деревьев как параметры. При каждом рекурсивном вызове мы прибавляем единицу к уровню и передаем соответствующий список, который должен быть добавлен.

```
    def _listwithlevels(self, level, trees):
        trees.append("  " * level + str(self.data))
        for child in self.children:
            child._listwithlevels(level + 1, trees)

    def __str__(self):
        trees = []
        self._listwithlevels(0, trees)
        return "\n".join(trees)

T = Tree(['a', ['b', ['c', ['d']]],['e',['f'], ['g']]])
print(str(T))
```

```
a
  b
    c
      d
  e
    f
    g
```

Шаблон, продемонстрированный здесь, называется обходом дерева (tree traversal), и немного позже мы рассмотрим его более подробно. Мы реализовали этот шаблон, потому что он помогает увидеть, что дерево действительно выглядит так, как и должно. Разумеется, необходимо написать правильные тесты, но, когда начнут возникать ошибки, мы сможем научиться очень многому, исследуя это дерево. Такой подход может подсказать, где искать ошибки.

Ниже приведен еще один пример похожего шаблона обхода дерева. Проверим, являются ли равными два дерева в том смысле, что они имеют одинаковую форму и данные. Воспользуемся методом `__eq__`, поскольку он вызывается, когда применяется оператор == для проверки равенства объектов `Tree`.

```
def __eq__(self, other):
    return self.data == other.data and self.children == other.children
```

Здесь использование рекурсии менее очевидно, но она есть, потому что `self.children` и `other.children` – это списки, а равенство списков определяется проверкой равенства их элементов. В данном случае элементами списков являются объекты `Tree`, поэтому метод `__eq__` будет вызываться для каждого такого объекта.

Ниже приведен еще один пример. Напишем функцию, которая вычисляет высоту дерева. Это можно сделать, вычисляя высоту всех поддеревьев и возвращая максимальное значение из полученных результатов. Если потомков нет, то высота равна нулю.

```
def height(self):
    if len(self.children) == 0:
        return 0
    else:
        return 1 + max(child.height() for child in self.children)
```

Надеюсь, вы уже вполне освоили эти выражения для создания генератора.

Напомню, что магический метод `__contains__` позволяет воспользоваться ключевым словом `in` для проверки присутствия элемента в наборе данных. Требуемое поведение этой функции, описанное в спецификации АТД, подсказывает нам, как реализовать ее.

```
def __contains__(self, k):
    return self.data == k or any(k in ch for ch in self.children)
```

Функция `any` принимает итерируемую последовательность логических значений и возвращает `True`, если хотя бы одно из них равно `True`. Эта функция применяет сокращенную схему вычислений, поэтому может остановиться сразу, как только найдет истинное значение. Если получен результат `False`, это приведет к итеративному проходу по всему дереву в целом. Точно такое поведение мы наблюдали при обычном обходе дерева.

16.5. Обход дерева

До настоящего момента все наборы, в которых хранились данные, были либо последовательными (т. е. `list`, `tuple` и `str`), либо непоследовательными (т. е. `dict` и `set`). Структура дерева выглядит находящейся где-то между этими двумя видами наборов данных. Некоторая структура существует, но она не является линейной. Мы можем придать этой структуре линейный (последовательный) вид, итеративно проходя по всем узлам дерева, но это не единственный способ

решения. Для деревьев процесс посещения всех узлов называется обходом дерева (tree traversal). Для упорядоченных деревьев существуют два стандартных обхода: обход в прямом порядке (preorder traversal) (обход в ширину) и в обратном (postorder traversal) (обход с отложенной выборкой). Вполне естественно, что оба обхода определяются рекурсивно.

При обходе в прямом порядке мы сначала посещаем конкретный узел, а затем выполняется обход всех его прямых потомков. При обходе в обратном порядке сначала выполняется обход всех потомков некоторого узла, а затем посещается сам узел. Посещение (visit) обозначает любое вычисление (действие), которое мы должны выполнить в узлах. Метод `printtree`, приведенный выше, является классическим примером обхода в прямом порядке. Но можно также выводить узлы при обходе в обратном порядке, как показано ниже.

```
def printpostorder(T):
    for child in self.children:
        printpostoder(child)
    print(T.data)
```

Всего лишь одно небольшое изменение в исходном коде, но в результате получается совсем другой вывод.

16.6. Если хотите немного развлечься...

Великим достижением в Python считалась возможность выполнить этот тип обхода с использованием генератора. Рекурсивные генераторы выглядят слегка таинственно, особенно при первом знакомстве с ними. Но если разобрать этот код на части и вручную пройти по нему, то такой способ поможет лучше понять, как работают генераторы.

```
def preorder(self):
    yield self.data
    for child in self.children:
        for data in child.preorder():
            yield data

# Определить __iter__ как псевдоним для метода preorder.
__iter__ = preorder
```

Это также можно сделать для итерационного прохода по деревьям (т. е. по их узлам). Я сделал этот метод закрытым, потому что пользователю дерева вряд ли потребуется доступ к отдельным узлам.

```
def _preorder(self):
    yield self
    for child in self.children:
        for descendant in child._preorder():
            yield descendant
```

16.6.1. Есть одно «но»

Этот рекурсивный генератор не выполняется за линейное время. Напомню, что каждое обращение к генератору создает объект. Этот объект не исчезает после выдачи значения, потому что может потребоваться выдача других значений. При вызове приведенного выше метода в дереве каждое выданное значение безоговорочно передается из узла в корень. Таким образом, общее время выполнения пропорционально сумме глубин всех узлов в дереве. Для вырожденного дерева (т. е. дерева с единственным путем) время выполнения равно $O(n^2)$. Для полностью сбалансированного двоичного дерева время выполнения равно $O(n \log n)$.

Использование рекурсии и стека вызовов существенно упрощает код обхода дерева по сравнению с отслеживанием процесса обхода вручную. Но недостаточно сохранить только стек узлов в пути из текущего узла в корень. Необходимо также отслеживать положение в итерационном обходе (прямых) потомков каждого из этих узлов. Следует помнить, что это задача объекта итератора – сохранять текущее положение в итерационном обходе. Следовательно, мы можем просто записать в стек еще и итераторы для потомков.

```
def _postorder(self):
    node, childiter = self, iter(self.children)
    stack = [(node, childiter)]
    while stack:
        node, childiter = stack[-1]
        try:
            child = next(childiter)
            stack.append((child, iter(child.children)))
        except StopIteration:
            yield node
            stack.pop()

def postorder(self):
    return (node.data for node in self._postorder())
```

В приведенном выше коде мне очень не нравится тот факт, что приходится заново присваивать значения переменным `node` и `childiter` на каждой итерации цикла. Вы сможете устранить этот недостаток, сохранив работоспособность кода?

16.6.2. Уровень за уровнем

Начав с нерекурсивного обхода в обратном порядке, можно изменить алгоритм для обхода дерева по уровням (уровень за уровнем). Позже мы подробнее рассмотрим этот тип обхода, когда будем изучать динамическую память (кучу – heap).

```
def _layerorder(self):
    node, childiter = self, iter(self.children)
    queue = Queue()
    queue.enqueue((node, childiter))
    while queue:
        node, childiter = queue.peek()
        try:
            child = next(childiter)
            queue.enqueue((child, iter(child.children)))
        except StopIteration:
            yield node
            queue.dequeue()

def layerorder(self):
    return (node.data for node in self._layerorder())
```

Глава 17

Деревья двоичного поиска

Начнем с абстрактного типа данных (АТД). Упорядоченное отображение (ordered mapping) – это тип данных «отображение», в котором ключи упорядочены. Упорядоченное отображение должно поддерживать все операции любого другого отображения, а также некоторые другие операции, аналогичные операциям упорядоченного списка `OrderedList`, например поиск предыдущего элемента.

17.1. Абстрактный тип данных «упорядоченное отображение»

Упорядоченное отображение хранит набор пар ключ–значение (со сравнимыми ключами) и поддерживает следующие операции.

- `get(k)` – возвращает значение, связанное с ключом `k`. Если заданный ключ отсутствует, то генерируется исключение `KeyError`.
- `put(k, v)` – добавляет пару ключ–значение `(k, v)` в отображение.
- `floor(k)` – возвращает кортеж `(k, v)`, соответствующий паре ключ–значение в отображении с наибольшим ключом, который меньше или равен `k`. Если такой кортеж не найден, то возвращается `(None, None)`.
- `remove(k)` – удаляет пару ключ–значение с ключом `k` из упорядоченного отображения. Если заданный ключ отсутствует, то генерируется исключение `KeyError`.

17.2. Определение и свойства дерева двоичного поиска

Дерево называется двоичным (binary tree), если каждый его узел имеет не более двух прямых потомков (children). Мы продолжаем предполагать, что работаем с упорядоченными деревьями, поэтому называем прямых потомков `left` (левый) и `right` (правый). Мы говорим, что двоичное дерево является деревом двоичного поиска (binary search tree), если для каждого узла `x` все ключи в поддереве `x.left` меньше, чем ключ в `x`, а все ключи в поддереве `x.right` больше, чем ключ в `x`. Это свойство упорядоченности, также известное как BST-свойство (или ДДП-свойство), отличает дерево двоичного поиска от любого другого типа двоичных деревьев.

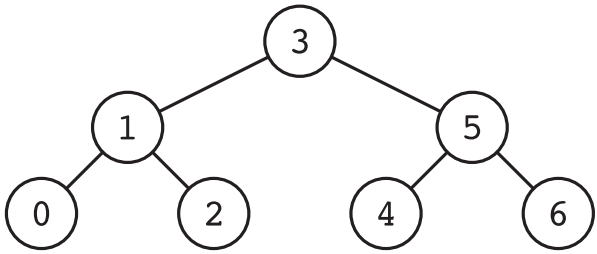

Рис. 17.1. Пример дерева двоичного поиска

Тот же набор узлов может быть размещен в другом дереве двоичного поиска.

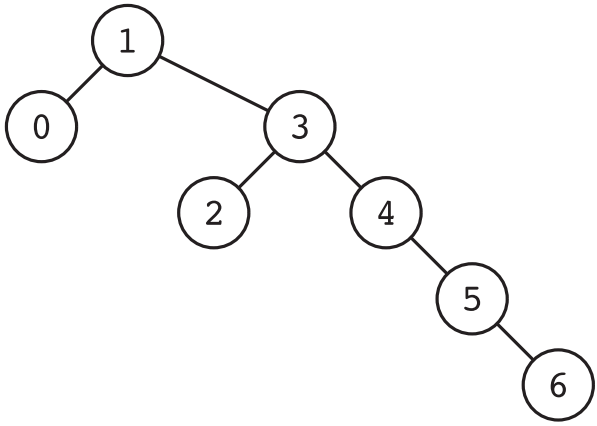

Рис. 17.2. Другое дерево двоичного поиска с тем же набором узлов

BST-свойство связано с новым типом обхода дерева, который невозможен в деревьях других типов. Ранее мы рассматривали обход деревьев в прямом и обратном порядке (preorder и postorder). При этих обходах посещались все узлы дерева. При обходе в прямом порядке посещался корень каждого поддерева до посещения любого из его потомков. При обходе в обратном порядке посещались все потомки до посещения соответствующего корня. Представляемый здесь новый тип обхода называется упорядоченным (симметричным) (inorder traversal) (или обходом с порядковой выборкой) и посещает все узлы в ветви левого прямого потомка до посещения корня, а затем – все узлы в ветви правого прямого потомка после посещения корня. Результатом является обход узлов в отсортированном порядке в соответствии с упорядоченностью ключей.

В итоге для различных деревьев двоичного поиска с одинаковыми узлами существует один и тот же упорядоченный обход.

17.3. Минимальная реализация

Мы реализуем АТД «упорядоченное отображение» с использованием дерева двоичного поиска. Поскольку у нас уже есть написанный абстрактный

класс для упаковки большой группы магических методов, которые предполагается использовать в любом отображении, начнем с наследования от этого класса.

Кроме того, в отличие от предыдущего раздела мы будем различать класс для дерева (`BSTMapping`) и класс для узлов (`BSTNode`). Мы будем придерживаться соглашения о том, что операции в классе `BSTNode` будут обрабатывать и возвращать другие объекты `BSTNode`, когда это необходимо, тогда как класс `BSTMapping` будет абстрагироваться от существования узлов, скрывая их от пользователя, и возвращать только ключи и значения.

Это всего лишь минимальные требования к отображению `Mapping`. Данная реализация выполняется сверху вниз, поэтому делегирует всю реальную работу классу `BSTNode`.

```
from ds2.mapping import Mapping

class BSTMapping(Mapping):
    def __init__(self):
        self._root = None

    def get(self, key):
        if self._root:
            return self._root.get(key).value
        raise KeyError

    def put(self, key, value):
        if self._root:
            self.root = self._root.put(key, value)
        else:
            self._root = BSTNode(key, value)

    def __len__(self):
        return len(self._root) if self._root is not None else 0

    def _entryiter(self):
        if self._root:
            yield from self._root

    def floor(self, key):
        if self._root:
            floornode = self._root.floor(key)
            if floornode is not None:
                return floornode.key, floornode.value
        return None, None

    def remove(self, key):
        if self._root:
            self._root = self._root.remove(key)
        else:
            raise KeyError

    def __delitem__(self, key):
        self.remove(key)
```

Приведенный выше код предоставляет нам все необходимое. Следует уделить внимание нескольким загадочным строкам. Одна из таких строк находится в методе `put` – она обновляет корень. Мы будем активно использовать это соглашение. Поскольку методы могут перестраивать структуру дерева, они возвращают узел, который непременно должен быть новым корнем поддерева. Этот шаблон применяется и в методе `remove`.

Здесь есть конструкция, которая раньше не встречалась, – операция `yield from` в итераторе. Она принимает итерируемый объект и выполняет итерационный проход по нему, выдавая каждый элемент. Поэтому `yield from self._root` – это то же самое, что `for item in iter(self._root): yield item`. Такая конструкция предполагает, что класс `BSTNode` обязательно должен быть итерируемым.

Рассмотрим подробнее, как реализованы эти методы. Начнем с инициализатора и нескольких других полезных методов.

```
class BSTNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
        self._length = 1

    def __len__(self):
        return self._length

    def __str__(self):
        return str(self.key) + " : " + str(self.value)
```

Метод `get` использует двоичный поиск для обнаружения требуемого ключа.

```
    def get(self, key):
        if key == self.key:
            return self
        elif key < self.key and self.left:
            return self.left.get(key)
        elif key > self.key and self.right:
            return self.right.get(key)
        else:
            raise KeyError
```

Обратите внимание: мы используем `self.left` и `self.right` как логические значения. Это работает, потому что `None` всегда вычисляется как `False`, а `BSTNode` всегда дает результат `True`. Для выполнения этой работы можно было бы реализовать метод `__bool__`, но в данном случае достаточно `__len__`. Объекты, содержащие метод `__len__`, являются истинными (`True`), если и только если их длина больше нуля. Это принятый по умолчанию способ проверки, не пуст ли контейнер. Поэтому, например, допустимо написать `while L: L.pop()`, и в этом цикле никогда не будет выполнена попытка извлечения элемента из пустого списка. В нашем случае можно написать `if self.left`, чтобы проверить существование левого прямого потомка, вместо записи `if self.left is not None`.

Далее следует реализация метода put. Сначала он выполняет двоичный поиск в дереве. Если метод находит ключ, уже существующий в дереве, то перезаписывает соответствующее значение (ключи в отображении не повторяются). Иначе он спускается в нижнюю часть дерева и добавляет новый узел.

```
def put(self, key, value):
    if key == self.key:
        self.value = value
    elif key < self.key:
        if self.left:
            self.left.put(key, value)
        else:
            self.left = BSTNode(key, value)
    elif key > self.key:
        if self.right:
            self.right.put(key, value)
        else:
            self.right = BSTNode(key, value)
    self._updatelength()

def _updatelength(self):
    len_left = len(self.left) if self.left else 0
    len_right = len(self.right) if self.right else 0
    self._length = 1 + len_left + len_right
```

Метод put также отслеживает длину, т. е. число записей в каждом поддереве.

17.3.1. Метод floor

Метод floor представляет собой просто усложненную версию метода get. Он также выполняет двоичный поиск, но демонстрирует другое поведение, если ключ не найден, в зависимости от того, выполнялся ли последний этап поиска влево или вправо. При начале с любого узла если выполняется поиск вправо и его результатом становится None, то возвращается сам начальный узел. Если выполняется поиск влево с результатом None, то возвращается значение None.

```
def floor(self, key):
    if key == self.key:
        return self
    elif key < self.key:
        if self.left is not None:
            return self.left.floor(key)
        else:
            return None
    elif key > self.key:
        if self.right is not None:
            floor = self.right.floor(key)
            return floor if floor is not None else self
        else:
            return self
```

17.3.2. Итерация

Как уже было отмечено выше, деревья двоичного поиска поддерживают упорядоченный обход. При упорядоченном обходе узлы выдаются в порядке их ключей.

Ниже приведен исходный код итератора упорядоченного обхода для дерева двоичного поиска, реализованного с использованием рекурсивных генераторов. Это вполне подходит в большинстве случаев. Существует немного более эффективный подход, но эта реализация, возможно, более удобна для чтения.

```
    def __iter__(self):
        if self.left is not None:
            yield from self.left
        yield self
        if self.right is not None:
            yield from self.right
```

17.4. Удаление

Для реализации операции удаления в дереве двоичного поиска воспользуемся стандартной методикой решения алгоритмических задач. Начнем с размышления о том, как обрабатывать простейшие возможные случаи. Затем мы увидим, как превратить каждый случай в простой.

Удаление начинается с поиска требуемого узла с использованием двоичного поиска в дереве. Затем, если узел является листом, его можно удалить без каких-либо затруднений. Так же просто удалить узел, у которого имеется только один прямой потомок, потому что можно удалить такой узел и передвинуть его потомка вверх без нарушения BST-свойства.

Более сложный случай возникает, когда удаляемый узел имеет левого и правого прямого потомка. В этом случае мы находим узел с наибольшим ключом в левом поддереве удаляемого узла (т. е. крайний правый узел). Мы меняем местами найденный узел с удаляемым и снова вызываем метод `remove` в левом поддереве. В следующий раз, когда мы приходим в этот узел, известно, что он имеет не более одного прямого потомка, потому что ранее находившийся на этом месте узел (перемещенный) не имел правого прямого потомка (иначе он не был бы крайним правым узлом перед перестановкой).

Внимание – важное замечание. Обмен местами двух узлов в дереве двоичного поиска нарушает BST-свойство. Но нарушается оно только в том плане, что удаляемый узел будет иметь ключ, который больше ключа перемещенного на его место узла. Поэтому удаление восстановит BST-свойство.

Ниже приведен исходный код для выполнения операции обмена местами и простой рекурсивный метод для поиска крайнего правого узла в поддереве.

```
    def _swapwith(self, other):
        self.key, other.key = other.key, self.key
        self.value, other.value = other.value, self.value

    def maxnode(self):
        return self.right.maxnode() if self.right else self
```

Теперь мы готовы к реализации метода `remove`. Как уже было отмечено выше, он выполняет рекурсивный двоичный поиск, чтобы найти заданный узел. При обнаружении требуемого ключа он меняет узлы местами и выполняет еще один рекурсивный вызов. Шаг обмена местами будет выполнен только один раз, поэтому общее время выполнения является линейным относительно высоты дерева.

```
    def remove(self, key):
        if key == self.key:
            if self.left is None: return self.right
            if self.right is None: return self.left
            self._swapwith(self.left.maxnode())
            self.left = self.left.remove(key)
        elif key < self.key and self.left:
            self.left = self.left.remove(key)
        elif key > self.key and self.right:
            self.right = self.right.remove(key)
        else: raise KeyError
        self._updatelength()
        return self

T = BSTMapping()
for i in [3,2,1,6,4,5,9,8,10]:
    T[i] = None
```

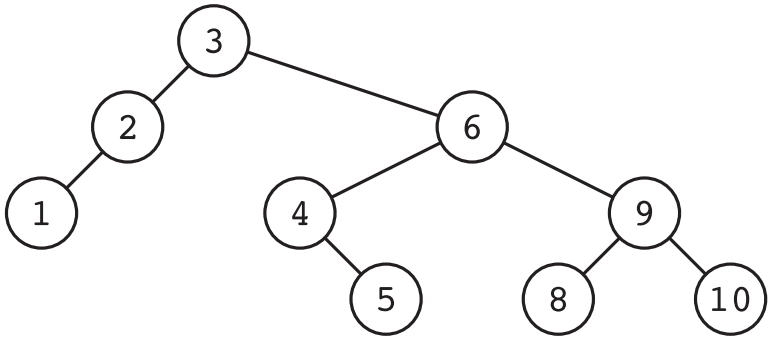

Рис. 17.3. Дерево двоичного поиска перед удалением узла

```
T.remove(6)
```

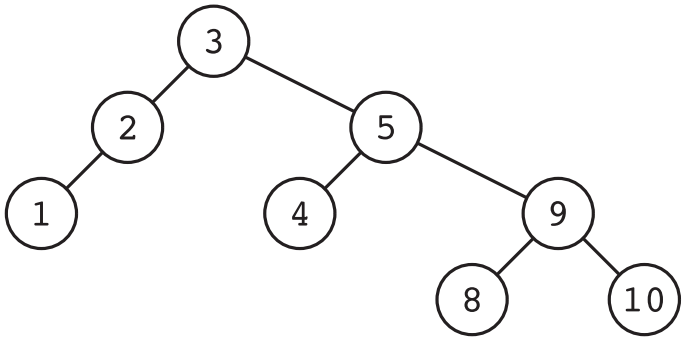

Рис. 17.4. Дерево двоичного поиска после удаления узла 6

```
T.remove(3)
```

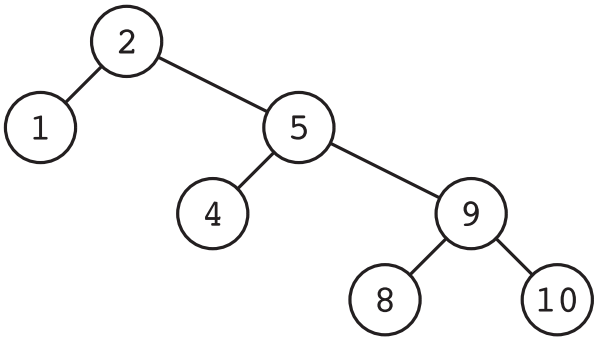

Рис. 17.5. Дерево двоичного поиска после удаления узла 3

Глава 18

Сбалансированные деревья двоичного поиска

В предыдущей главе мы рассматривали реализацию отображения с использованием дерева двоичного поиска. Но при анализе этого алгоритма стало понятно, что время выполнения всех основных операций зависит от высоты дерева.

Дерево двоичного поиска с n узлами имеет высоту n – 1. Легко создать такое дерево, просто вставляя в него ключи в отсортированном порядке. С другой стороны, всегда существует дерево двоичного поиска с высотой O(log n) для любого множества из n ключей. Это достигается первоначальной вставкой медианного значения ключа, затем вставляются медианные значения каждой половины и т. д. с применением рекурсии. Поэтому создается большой разрыв между наилучшим и наихудшим случаями. Наша цель – получить вариант, максимально близкий к наилучшему случаю с сохранением времени выполнения всех операций, пропорционального высоте дерева.

Мы будем говорить, что дерево двоичного поиска с n узлами является сбалансированным (balanced), если его высота не превышает некоторого постоянного значения, кратного log n. Чтобы сбалансировать конкретное дерево двоичного поиска, необходимо исключить перестроение всего дерева в целом. Вместо этого более предпочтительно вносить в дерево минимальные изменения, которые сохраняют BST-свойство.

Самой простой основной операцией такого рода является поворот дерева (tree rotation). Поворот выполняется в двух формах: вправо `rotateright` и влево `rotateleft`. Поворот узла вправо перемещает его так, что он становится правым прямым потомком своего левого прямого потомка, после чего соответствующим образом обновляются все прямые потомки этих узлов. Ниже приведен исходный код для выполнения операции поворота вправо.

```
def rotateright(self):
    newroot = self.left
    self.left = newroot.right
    newroot.right = self
    return newroot
```

Обратите внимание: функция `rotateright` возвращает новый корень (поддерева). Это весьма полезное соглашение при работе с деревьями двоичного поиска и выполнении операций поворота. Каждый метод, который может изменить структуру дерева, будет возвращать новый корень полученного в результате поддерева. Таким образом, вызов таких методов всегда должен быть включен в инструкцию присваивания. Например, если необходимо выполнить поворот левого прямого потомка узла `parent` вправо, то мы должны написать следующий код: `left = parent.left.rotateright()`.

18.1. Реализация класса BSTMapping

Ниже приведен исходный код для класса `BSTMapping`, в котором реализованы операции поворота. Мы также обеспечиваем обновление длин после каждой операции поворота. Основное отличие от приведенного выше кода состоит в том, что теперь все методы, которые могут изменить структуру дерева, включены в инструкции присваивания. При этом предполагается, что только методы `put` и `remove` будут перестраивать дерево, а методы `get` и `floor` оставят структуру дерева неизменной.

```
from ds2.orderedmapping import BSTMapping, BSTNode

class BalancedBSTNode(BSTNode):
    def newnode(self, key, value):
        return BalancedBSTNode(key, value)

    def put(self, key, value):
        if key == self.key:
            self.value = value
        elif key < self.key:
            if self.left:
                self.left = self.left.put(key, value)
            else:
                self.left = self.newnode(key, value)
        elif key > self.key:
            if self.right:
                self.right = self.right.put(key, value)
            else:
                self.right = self.newnode(key, value)
        self._updatelength()
        return self

    def rotateright(self):
        newroot = self.left
        self.left = newroot.right
        newroot.right = self
        self._updatelength()
        newroot._updatelength()
        return newroot
```

```
    def rotateleft(self):
        newroot = self.right
        self.right = newroot.left
        newroot.left = self
        self._updatelength()
        newroot._updatelength()
        return newroot

class BalancedBST(BSTMapping):
    Node = BalancedBSTNode

    def put(self, key, value):
        if self._root:
            self._root = self._root.put(key, value)
        else:
            self._root = self.Node(key, value)
```

18.1.1. Совместимость снизу вверх шаблонов «фабрика»

Мы в определенной степени оставляем пространство для будущего развития этого исходного кода. В частности, необходимо, чтобы этот класс был легко расширяемым. Для этого мы не создаем напрямую новые экземпляры узлов. В классе `BSTMapping` выполняется присваивание `Node = BSTNode`, а новые узлы создаются инструкцией `self.Node(key, value)` вместо `BSTNode(key, value)`. Таким образом, при расширении этого класса (а мы будем расширять его несколько раз) можно использовать другое значение для узла `Node`, чтобы получать различные типы узлов без необходимости переписывания всех методов из класса `BSTMapping`. Аналогичным образом мы используем метод `newnode` в классе `BSTNode`. Метод, который генерирует новые экземпляры некоторого класса, иногда называют фабрикой (factory).

С учетом совместимости снизу вверх (прямой совместимости, т. е. совместимости с будущими версиями) достаточно трудно придерживаться оптимального пути развития между правильным проектированием и предварительной (иногда преждевременной) оптимизацией. В нашем случае использование фабрики для генерации узлов стало возможным только после того, как была исключена необходимость в написании подклассов. Это должно быть всеобщим предупреждением для студентов при чтении кода в учебниках. Часто можно увидеть только то, как завершился код, но не то, с чего он начался. Полезно задавать вопрос: «Почему этот код написан тем или иным способом?», но ответ может заключаться в том, что в дальнейшем проще будет написать расширение для этого кода.

18.2. Взвешенные сбалансированные деревья

Теперь мы готовы к балансированию деревьев. Один из способов убедиться в том, что дерево сбалансировано, – наличие в каждом узле `x` ключа, являющегося медианным значением всех ключей в поддереве с корнем `x`. После этого ситуация может стать аналогичной двоичному поиску в отсортирован-

ном списке. При перемещении вниз по дереву у каждого узла в поддереве будет столько же узлов, сколько у его родителя. Таким образом, после log n шагов мы переходим к листу, поэтому дерево двоичного поиска должно быть сбалансированным.

Наличие медианных значений в каждом узле – идеальный вариант, который хотелось бы иметь, но его очень трудно получить, да и слишком строг инвариант для балансирования деревьев двоичного поиска. Напомню, что при анализе алгоритмов `quicksort` и `quickselect` мы рассматривали центральные элементы как удачно выбранные, если они находились в середине половины списка (т. е. элементы в диапазоне между n/4 и 3n/4). Если ключ в каждом узле является удачно выбранным центральным элементом среди ключей в своем поддереве, то мы можем также гарантировать, что общая высота имеет порядок $\log_{4/3}$n.

Мы будем называть узел x взвешенным сбалансированным (weight-balanced), если

```
len(x) + 1 < 4 * (min(len(x.left), len(x.right)) + 1).
```

Достаточно легко проверить это условие в нашей реализации `BSTMapping`, потому что в ней отслеживается длина каждого узла. Если при некоторых изменениях узел перестает быть взвешенным сбалансированным, то мы восстановим взвешенный баланс операциями поворота. В простейшем случае достаточно одного поворота. Если один поворот не достиг цели, то двух операций поворота будет достаточно. Мы рассмотрим несколько примеров и алгебраических формул, объясняющих, почему выполняются эти действия.

Метод `rebalance` проверяет условие сбалансированности и выполняет требуемые операции поворота. Этот метод вызывается после каждого изменения в дереве, поэтому можно предположить, что несбалансированный узел `x` всего лишь слегка разбалансирован. Без потери общего смысла предположим, что `x` имеет лишь несколько узлов в своем левом поддереве. Затем при вызове `x.rotateleft()` мы должны проверить, что узел `x` стал взвешенным сбалансированным и что его новый родитель `y` (бывший правый прямой потомок узла `x`) остался взвешенным сбалансированным. Очевидно, что `len(x) == len(x.rotateleft())`, так как оба содержат одни и те же узлы. Поэтому дисбаланс в `y` после поворота возникает, только если его правый прямой потомок имеет слишком малый вес. В этом случае мы выполняем поворот `y` вправо перед поворотом `x` влево. Это гарантирует, что все узлы в этом поддереве остаются взвешенными сбалансированными.

Этого описания недостаточно для того, чтобы полностью убедиться в корректности алгоритма, но достаточно, чтобы написать код. Сначала рассмотрим исходный код и понаблюдаем за работой метода `rebalance`. Затем вернемся к обсуждению алгоритма и выполним некоторые алгебраические действия, чтобы доказать его корректность.

```
from ds2.orderedmapping import BalancedBST, BalancedBSTNode

class WBTreeNode(BalancedBSTNode):
    def newnode(self, key, value):
        return WBTreeNode(key, value)

    def toolight(self, other):
        otherlength = len(other) if other else 0
        return len(self) + 1 >= 4 * (otherlength + 1)

    def rebalance(self):
        if self.toolight(self.left):
            if self.toolight(self.right.right):
                self.right = self.right.rotateright()
            newroot = self.rotateleft()
        elif self.toolight(self.right):
            if self.toolight(self.left.left):
                self.left = self.left.rotateleft()
            newroot = self.rotateright()
        else:
            return self
        return newroot

    def put(self, key, value):
        newroot = BalancedBSTNode.put(self, key, value)
        return newroot.rebalance()

    def remove(self, key):
        newroot = BalancedBSTNode.remove(self, key)
        return newroot.rebalance() if newroot else None

class WBTree(BalancedBST):
    Node = WBTreeNode
```

Метод `toolight` предназначен для проверки того, что поддерево содержит достаточно узлов, чтобы стать прямым потомком взвешенного сбалансированного узла. Мы используем его и для проверки того, что текущий узел является взвешенным сбалансированным, и для проверки, требуется ли одна операция поворота или две.

Методы `put` и `remove` вызывают соответствующие методы из суперкласса `BSTNode`, затем выполняют ребалансирование перед возвратом. Мы пытаемся повторно использовать весь возможный код из нашей предыдущей реализации.

18.3. Сбалансированные по высоте деревья (АВЛ-деревья)

Стимулом для балансирования деревьев двоичного поиска было сохранение их небольшой высоты. Кроме балансирования по весу можно также попытаться сохранить достаточно близкими значения высоты левого и правого поддерева. В действительности мы можем потребовать, чтобы значения их высоты различались не более чем на единицу. По аналогии со взвешенными сбалансиро-

ванными деревьями мы будем следить за тем, чтобы не нарушалось свойство сбалансированности по высоте, а при его нарушении восстанавливать баланс одним или двумя поворотами.

Такие сбалансированные по высоте деревья часто называют АВЛ-деревьями[1]. В этой реализации мы будем отслеживать высоту каждого поддерева и использовать эти значения для проверки сбалансированности. Часто АВЛ-деревья сохраняют только баланс в каждом узле, а не точную высоту, но вычисление значений высоты выполняется относительно безболезненно.

```
from ds2.orderedmapping import BalancedBST, BalancedBSTNode

def height(node):
    return node.height if node else -1

def update(node):
    if node:
        node._updatelength()
        node._updateheight()

class AVLTreeNode(BalancedBSTNode):
    def __init__(self, key, value):
        BalancedBSTNode.__init__(self, key, value)
        self._updateheight()

    def newnode(self, key, value):
        return AVLTreeNode(key, value)

    def _updateheight(self):
        self.height = 1 + max(height(self.left), height(self.right))

    def balance(self):
        return height(self.right) - height(self.left)

    def rebalance(self):
        bal = self.balance()
        if bal == -2:
            if self.left.balance() > 0:
                self.left = self.left.rotateleft()
            newroot = self.rotateright()
        elif bal == 2:
            if self.right.balance() < 0:
                self.right = self.right.rotateright()
            newroot = self.rotateleft()
        else:
            return self
        update(newroot.left)
        update(newroot.right)
        update(newroot)
        return newroot
```

[1] В названии этих деревьев аббревиатура АВЛ образована из первых букв фамилий их создателей: Георгия Адельсон-Вельского и Евгения Ландиса. – *Прим. перев.*

```
    def put(self, key, value):
        newroot = BalancedBSTNode.put(self, key, value)
        update(newroot)
        return newroot.rebalance()

    def remove(self, key):
        newroot = BalancedBSTNode.remove(self, key)
        update(newroot)
        return newroot.rebalance() if newroot else None

class AVLTree(BalancedBST):
    Node = AVLTreeNode
```

18.4. Косые деревья

Для удобства добавим еще один вид сбалансированного дерева двоичного поиска. В этом случае в действительности мы не будем обеспечивать сбалансированность полученного дерева, но придадим ему некоторые другие полезные свойства.

В косом дереве (splay tree) при каждом извлечении или записи элемента для соответствующего узла будет выполнен полный поворот к корню. Но вместо того, чтобы поворачивать его напрямую, мы рассматриваем два последовательно выполняемых шага. Метод `splayup` ищет требуемый ключ на два уровня вниз по дереву. Если ключ не найден ровно двумя уровнями ниже, метод ничего не делает. Иначе операция поворота выполняется дважды. Если требуются повороты в одном направлении, то сначала выполняется нижний, если в разных направлениях, то сначала выполняется верхний.

На самом высоком уровне можно выполнить только один поворот, чтобы полностью перевести узел в корень. Эта операция выполняется методом `splayup` в классе SplayTree.

Главное отличие от предыдущих реализаций теперь заключается в том, что мы будем изменять дерево при вызовах метода `get`. Поэтому придется переписать метод `get` вместо его наследования. Ранее `get` должен был возвращать требуемое значение. Но теперь нужно возвращать новый корень при каждой операции, которая может изменить дерево. Так что именно мы должны возвращать? Очевидно, что необходимо возвращать требуемое значение, но при этом не нарушать структуру дерева. К счастью, существует простое решение. Операция `splayup` удобно поворачивает найденный узел полностью к корню. Поэтому метод `SplayTreeNode.get` возвращает новый корень поддерева, а метод `SplayTree.get` возвращает значение в корне.

```
from ds2.orderedmapping import BalancedBST, BalancedBSTNode

class SplayTreeNode(BalancedBSTNode):
    def newnode(self, key, value):
        return SplayTreeNode(key, value)

    def splayup(self, key):
        newroot = self
        if key < self.key:
            if key < self.left.key:
                newroot = self.rotateright().rotateright()
            elif key > self.left.key:
                self.left = self.left.rotateleft()
                newroot = self.rotateright()
        elif key > self.key:
            if key > self.right.key:
                newroot = self.rotateleft().rotateleft()
            elif key < self.right.key:
                self.right = self.right.rotateright()
                newroot = self.rotateleft()
        return newroot

    def put(self, key, value):
        newroot = BalancedBSTNode.put(self, key, value)
        return newroot.splayup(key)

    def get(self, key):
        if key == self.key:
            return self
        elif key < self.key and self.left:
            self.left = self.left.get(key)
        elif key > self.key and self.right:
            self.right = self.right.get(key)
        else:
            raise KeyError
        return self.splayup(key)

class SplayTree(BalancedBST):
    Node = SplayTreeNode

    def splayup(self, key):
        if key < self._root.key:
            self._root = self._root.rotateright()
        if key > self._root.key:
            self._root = self._root.rotateleft()

    def get(self, key):
        if self._root is None: raise KeyError
        self._root = self._root.get(key)
        self.splayup(key)
        return self._root.value

    def put(self, key, value):
        BalancedBST.put(self, key, value)
        self.splayup(key)
```

Глава 19

Очереди с приоритетами

В этой главе будет рассматриваться универсальная фундаментальная структура данных – очередь с приоритетами (priority queue). Она похожа на обычную очередь, но все ее элементы имеют приоритет и упорядочиваются именно по этому приоритету, а не в порядке записи в очередь.

К этому моменту мы достаточно много знаем о разнообразных структурах данных, в том числе и о возможности реализации такой структуры данных несколькими различными способами. В итоге будет представлена структура данных под названием «куча» (heap), которая обладает хорошей производительностью во многих отношениях.

19.1. Абстрактный тип данных «очередь с приоритетами»

Абстрактный тип данных (АТД) «очередь с приоритетами» – это тип данных, позволяющий хранить набор элементов с приоритетами (которые не обязательно должны быть неповторяющимися) и поддерживающий описанные ниже операции.

- `insert(item, priority)` – добавляет элемент `item` с заданным приоритетом `priority`.
- `findmin()` – возвращает элемент с минимальным приоритетом. Если существует несколько элементов с минимальным приоритетом, то один из таких элементов выбирается произвольным образом.
- `removemin()` – удаляет и возвращает элемент с минимальным приоритетом. Если существует несколько элементов с минимальным приоритетом, то один из таких элементов выбирается произвольным образом.

19.2. Использование списка

Как для каждого абстрактного типа данных, представленного в этой книге, существует простая, но не всегда эффективная реализация с использованием списка. И здесь мы начнем со списка, в который поместим все записи (элементы). Для поиска минимального элемента выполняется итерационный проход по списку.

```
class SimpleListPQ:
    def __init__(self):
        self._L = []

    def insert(self, item, priority):
        self._L.append((item, priority))

    def findmin(self):
        return min(self._L, key = lambda x : x[1])[0]

    def removemin(self):
        item, priority = min(self._L, key = lambda x : x[1])
        self._L.remove((item, priority))
        return item
```

Этот код будет работать, но в нем имеются проектные решения, которые нельзя назвать удачными. Во-первых, то, что можно назвать магическими индексами. Поскольку данные хранятся в кортежах, нам придется использовать индекс для извлечения приоритета или самого элемента при необходимости. Это недокументированное соглашение. Если кто-то просматривает этот код, то ему придется читать и встроенный код, чтобы понять работу метода `findmin`. Кроме того, здесь имеет место дублирование кода в местах использования функции `min` с одинаковым лямбда-выражением для атрибута `key` в методах `removemin` и `findmin`. Эти недостатки могут стать стимулом для возврата к началу и очистки, т. е. к рефакторингу исходного кода.

Мы создадим класс, где хранятся записи, в каждой из которых содержатся атрибуты `item` и `priority`. Записи будут организованы с возможностью их сравнения с помощью реализации метода `__lt__`, таким образом, сравнение записей всегда будет просто сравнением их приоритетов.

```
class Entry:
    def __init__(self, item, priority):
        self.priority = priority
        self.item = item

    def __lt__(self, other):
        return self.priority < other.priority
```

Теперь можно переписать версию очереди с приоритетами на основе списка. Она почти такая же, как раньше, за исключением использования класса `Entry`, благодаря чему из кода стали более понятными типы объектов, и, как я надеюсь, его проще читать и понимать.

```
from ds2.priorityqueue import Entry

class UnsortedListPQ:
    def __init__(self):
        self._entries = []

    def insert(self, item, priority):
        self._entries.append(Entry(item, priority))
```

```
    def findmin(self):
        return min(self._entries).item

    def removemin(self):
        entry = min(self._entries)
        self._entries.remove(entry)
        return entry.item
```

Проанализируем время выполнения методов в этом классе. Метод `insert` явно требует только постоянного времени, потому что выполняет добавление одного списка. Для методов `findmin` и `removemin` необходимо время O(n) для очереди с приоритетами с n элементами, потому что функция `min` выполняет итерационный проход по всему списку, чтобы найти запись с наименьшим приоритетом.

Метод `findmin` должен выполняться намного быстрее, если бы список был отсортирован. Тогда он мог бы просто вернуть первый элемент такого списка. Еще лучшим подходом стала бы сортировка списка в обратном порядке. Тогда метод `removemin` также выполнялся бы за постоянное время, используя функцию `pop` с постоянным временем выполнения для такого списка.

```
from ds2.priorityqueue import Entry

class SortedListPQ:
    def __init__(self):
        self._entries = []

    def insert(self, item, priority):
        self._entries.append(Entry(item, priority))
        self._entries.sort(reverse = True)

    def findmin(self):
        return self._entries[-1].item

    def removemin(self):
        return self._entries.pop().item
```

Здесь я воспользовался превосходным методом Python `sort` с параметром `reverse`, который, как можно догадаться, позволяет выполнить сортировку в обратном порядке.

```
S = SortedListPQ()
S.insert("ham", 3)
S.insert("cheese", 1)
S.insert("bread", 2)
print([S.removemin() for i in [1,2,3]])
```

```
['cheese', 'bread', 'ham']
```

Асимптотическое время выполнения обоих методов `findmin` и `removemin` уменьшилось до постоянного. Метод `insert` теперь не выполняется за по-

стоянное время, потому что осуществляет сортировку. Это означает, что время выполнения будет зависеть от времени, необходимого для сортировки. В данном случае, предполагая, что никакие другие методы не переупорядочивают (закрытый) атрибут `_entries`, мы знаем, что существует только одна неупорядоченная запись – только что добавленная. При таком условии правильная реализация вставки с сортировкой должна выполняться за линейное время (вставляется только один элемент перед операцией сортировки всего списка элементов). Несмотря на то что для краткости кода вызывается встроенный метод сортировки Python `sort` (который может требовать время O(n log n)), в действительности это операция с линейным временем выполнения.

После создания этих двух реализаций АТД очереди с приоритетами мы пришли к компромиссу между эффективностью двух операций. В первом случае имеем быстрый метод `insert` и медленный метод `removemin`, во втором – медленный `insert` и быстрый `removemin`. Наша следующая цель – сделать все операции достаточно быстрыми. Невозможно обеспечить постоянное время выполнения для всех методов, но логарифмическое время вполне достижимо.

19.3. Кучи

Структура данных, которую мы будем использовать для эффективной реализации очереди с приоритетами, называется кучей (heap). Кучи почти всегда применяются для реализации очереди с приоритетами, поэтому если вы решите проверить другие источники, то можете не увидеть различий между этими двумя концепциями. Очередь с приоритетами – это абстрактный тип данных (АТД), куча – это конкретная структура данных.

Сложнее с двумя другими терминологическими проблемами. Во-первых, существует много разнообразных видов куч. Мы будем рассматривать только один пример – так называемую двоичную кучу (binary heap). Во-вторых, в куче вместо приоритета обычно используется термин «ключ» (key). Это может привести к некоторой путанице, так как, в отличие от ключей в отображениях, ключи как приоритеты могут быть повторяющимися. Тем не менее мы будем использовать термин «приоритет», чтобы сохранить различие этих двух концепций.

Двоичную кучу можно мысленно интерпретировать как двоичное дерево, организованное так, что меньшие приоритеты расположены выше бóльших приоритетов. Название подходящее, поскольку в правильно организованной куче любых объектов или предметов крупные объекты должны находиться внизу, а более мелкие – наверху. Про любое дерево с узлами, имеющими приоритеты, можно сказать, что это дерево пирамидально упорядоченное (heap-ordered tree), если для каждого узла приоритеты его прямых потомков по крайней мере не меньше приоритета самого этого узла. При этом естественным образом предполагается, что минимальный приоритет имеет корень этого дерева.

19.4. Хранение дерева в списке

Ранее мы хранили деревья как связные структуры данных. Это означает наличие класса узла, в котором хранятся ссылки на другие узлы, например на узлы прямых потомков. Мы будем использовать кучу как возможность увидеть другой способ представления дерева. Записи кучи мы поместим в список, при этом индексы списка кодируют отношения родитель–потомок в дереве.

Для узла с индексом `i` его левый прямой потомок будет иметь индекс `2 * i + 1`, а правый прямой потомок – `2 * i + 2`. Это означает, что родитель узла с индексом i имеет индекс `(i-1)//2`. При таком отображении узлов в индексы корень имеет индекс 0, а остальные узлы распределяются по уровням слева направо и сверху вниз. Используя это отображение, мы говорим, что список пирамидально упорядочен (heap-ordered), если соответствующее ему двоичное дерево является пирамидально упорядоченным. Можно равнозначно сказать, что для всех i > 0 приоритет записи с индексом i больше или равен приоритету записи с индексом (i–1)//2.

Мы будем поддерживать инвариант: после каждой операции список записей с приоритетами остается пирамидально упорядоченным. При вставке нового узла он добавляется в список, затем многократно выполняются операции обмена местами с родителями до тех пор, пока список не станет пирамидально упорядоченным. Этот шаг обновления списка назовем `_upheap` – он похож на внутренний цикл при сортировке вставкой. Но в отличие от сортировки вставкой `_upheap` выполняет не более O(log n) операций обмена местами (каждая такая операция уменьшает индекс наполовину).

Существует похожая операция `_downheap`, которая многократно меняет местами запись и ее прямого потомка до тех пор, пока список не станет пирамидально упорядоченным. Эта операция окажется полезной в следующем разделе для создания кучи с нуля.

Базовая операция, используемая в описанных выше методах для перестроения кучи, меняет местами два элемента, применяя метод `_swap`.

```
from ds2.priorityqueue import Entry

class HeapPQ:
    def __init__(self):
        self._entries = []

    def insert(self, item, priority):
        self._entries.append(Entry(item, priority))
        self._upheap(len(self._entries) - 1)

    def _parent(self, i):
        return (i - 1) // 2

    def _children(self, i):
        left = 2 * i + 1
        right = 2 * i + 2
        return range(left, min(len(self._entries), right + 1))
```

```
    def _swap(self, a, b):
        L = self._entries
        L[a], L[b] = L[b], L[a]

    def _upheap(self, i):
        L = self._entries
        parent = self._parent(i)
        if i > 0 and L[i] < L[parent]:
            self._swap(i, parent)
            self._upheap(parent)

    def findmin(self):
        return self._entries[0].item

    def removemin(self):
        L = self._entries
        item = L[0].item
        L[0] = L[-1]
        L.pop()
        self._downheap(0)
        return item

    def _downheap(self, i):
        L = self._entries
        children = self._children(i)
        if children:
            child = min(children, key = lambda x: L[x])
            if L[child] < L[i]:
                self._swap(i, child)
                self._downheap(child)

    def __len__(self):
        return len(self._entries)
```

19.5. Создание кучи с нуля, _heapify

Используя только лишь открытый интерфейс, можно создать объект HeapPQ из списка пар элемент–приоритет. Например, приведенный ниже код должен работать корректно.

```
pq = HeapPQ()
pairs = [(10, 10), (2, 2), (30, 30), (4,4)]
for item, priority in pairs:
    pq.insert(item, priority)
```

Для выполнения метода `insert` требуется время O(log n), поэтому общее время выполнения по этой методике равно O(n log n). Вы должны дважды проверить, прежде чем убедиться в том, что можно верить этому утверждению, несмотря на то что для каждой операции вставки имеется менее n записей.

Возможно, это покажется удивительным, но мы можем создать объект HeapPQ за линейное время. Назовем это превращением списка в кучу (heapifying a list). Для этого мы будем использовать метод _downheap, который уже написан ранее. Исходный код обманчиво прост.

```
    def _heapify(self):
        n = len(self._entries)
        for i in reversed(range(n)):
            self._downheap(i)
```

Внимательно посмотрите на различие между приведенным выше кодом `_heapify` и кодом метода `_heapify_slower`, показанным ниже. Первый метод работает с конца списка и спускает вниз (`_downheap`) каждую запись, а второй обрабатывает список с начала и поднимает вверх (`_upheap`) каждую запись. Оба метода корректны, но первый быстрее.

```
    def _heapify_slower(self):
        n = len(self._entries)
        for i in range(n):
            self._upheap(i)
```

Работа методов может показаться одинаковой, но это не так. Чтобы увидеть, почему она различна, необходимо немного подробнее рассмотреть время выполнения методов `_upheap` и `_downheap`. Рассмотрим список с точки зрения дерева. Для `_upheap` время выполнения зависит от глубины начального узла. Для `_downheap` время выполнения зависит от высоты поддерева с корнем в начальном узле. В полном двоичном дереве половина узлов является листьями, поэтому `_downheap` выполняется за постоянное время, а для выполнения `_upheap` требуется время O(log n). Таким образом, метод `_heapify_slower` потребует как минимум времени $(n/2)\ \log_2 n = O(n \log n)$.

С другой стороны, для анализа `_heapify` необходимо просуммировать высоты всех узлов в полном двоичном дереве. Это записывается в виде формулы $n\sum_{i=1}^{log_2 n} i / 2^i$. Существует остроумный прием для ограничения этой суммы. Просто нужно заметить, что если из каждого узла в дереве построить путь, который проходит влево на первом шаге и вправо на всех последующих шагах, то никакие два пути не перекрывают друг друга. Это означает, что сумма длин этих путей (которая также является и суммой высот) не превышает общее число ребер n – 1. Следовательно, метод `_heapify` выполняется за время O(n).

19.6. Значимость и изменение приоритетов

Иногда в очереди с приоритетами требуется сравнение элементов, а не их приоритетов. То есть приоритет перестает иметь значение. Для этого можно сделать приоритеты необязательными для методов, которые их используют. Кроме того, часто будет возникать необходимость в определении метода `key`, такого что приоритет элемента `item` должен вычисляться как `key(item)`. Например, именно так это работает для других методов в упорядоченных наборах данных, таких как `sort`, `min` и `max`. Мы примем к исполнению обе задачи одновременно. Для определения метода `key` добавим необязательный параметр в методе `__init__`. Значением по умолчанию будет функция, принимающая некоторый элемент и просто возвращающая тот же элемент. В этом случае если не задана ключевая функция и приоритет, то будет установлен приоритет, равный элементу.

В очередях с приоритетами существует еще одна стандартная операция – изменение приоритета `changepriority`. Она делает то, о чем говорит ее имя, – изменяет приоритет элемента. Это очень просто реализовать в нашем текущем коде, если известен индекс изменяемого элемента. Достаточно изменить приоритет элемента и вызвать методы `_upheap` и `_downheap` с этим индексом. Если необходимо переместить кучу вверх для восстановления инварианта ее порядка, то с этим справится метод `_upheap`. Если необходимо переместить кучу вниз, то метод `_upheap` оставит ее на своем месте, а метод `_downheap` будет сдвигать ее вниз до тех пор, пока упорядоченность кучи не будет восстановлена.

Но обычный способ изменения приоритета – указание элемента и его нового приоритета. При этом необходимо знать индекс элемента. Мы обеспечим такую возможность, с помощью словаря, выполняющего преобразование элементов в индексы. Потребуется обновление этого словаря при каждом изменении порядка элементов. Чтобы обеспечить корректность этой операции, добавим метод, меняющий местами элементы, и будем использовать его в каждой операции обмена.

Также добавим возможность инициализации кучи каким-либо набором данных, используя метод `_heapify` с установкой функции `key`, если это необходимо. По умолчанию параметром инициализации будет простой список элементов. Если вместо списка потребуется инициализировать кучу набором пар `(item, priority)`, то в качестве аргумента будет использовано ключевое слово `entries`. Эта полная версия очереди с приоритетами будет весьма полезной для работы с некоторыми алгоритмами на графах, которые рассматриваются в следующих главах.

```
from ds2.priorityqueue import Entry, HeapPQ

class PriorityQueue(HeapPQ):
    def __init__(self, items = (), entries = (), key = lambda x: x):
        self._key = key
        self._entries = [Entry(i, p) for i, p in entries]
        self._entries.extend([Entry(i, key(i)) for i in items])
        self._itemmap = {entry.item : index for index, entry in enumerate(self._entries)}
        self._heapify()

    def insert(self, item, priority = None):
        if priority is None:
            priority = self._key(item)
        index = len(self._entries)
        self._entries.append(Entry(item, priority))
        self._itemmap[item] = index
        self._upheap(index)

    def _swap(self, a, b):
        L = self._entries
        va = L[a].item
        vb = L[b].item
        self._itemmap[va] = b
        self._itemmap[vb] = a
        L[a], L[b] = L[b], L[a]
```

```
    def changepriority(self, item, priority = None):
        if priority is None:
            priority = self._key(item)
        i = self._itemmap[item]
        self._entries[i].priority = priority
        # Предположим, что дерево - упорядоченная куча, только для нее это будет действовать.
        self._upheap(i)
        self._downheap(i)

    def removemin(self):
        L = self._entries
        item = L[0].item
        self._swap(0, len(L) - 1)
        del self._itemmap[item]
        L.pop()
        self._downheap(0)
        return item
```

Мы можем использовать приведенный выше код для реализации класса `MaxHeap`, т. е. очереди с приоритетами, состоящей из элементов, упорядоченных по убывающим значениям.

```
maxheap = PriorityQueue(key = lambda x: -x)

n = 10
for i in range(n):
    maxheap.insert(i)   # Нет необходимости в определении приоритета.

# Содержимое должно быть выведено в убывающем порядке.
print([maxheap.removemin() for i in range(n)])
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

19.7. Итеративный проход по очереди с приоритетами

Для многих операций, использующих очередь с приоритетами, требуется получение ее элементов поочередно по одному, возможно, с добавлением новых элементов и последующим повторением до тех пор, пока элементов не останется. Это выглядит немного странным по сравнению с другими итераторами, с которыми мы имели дело ранее, потому что в данном случае изменяется структура данных. Фактически эта операция разрушает структуру. По этой причине не требуется создавать объект итератора для очереди с приоритетами. Нужно, чтобы очередь с приоритетами сама стала итератором.

Напомню: для того чтобы быть итератором, объект обязательно должен содержать методы `__iter__` и `__next__`. Метод `__iter__` предназначен для возврата итератора, которым в нашем случае является сама очередь с приоритетами.

```
    def __iter__(self):
        return self
```

Метод `__next__` возвращает следующий элемент, предполагая, что он существует. Если в очереди не осталось элементов, то метод должен сгенерировать

исключение StopIteration. Например, это исключение может стать сигналом для завершения цикла for.

```
    def __next__(self):
        if len(self) > 0:
            return self.removemin()
        else:
            raise StopIteration
```

19.8. Пирамидальная сортировка

Очередь с приоритетами можно использовать для сортировки списка. Мы просто помещаем все элементы в очередь с приоритетами, а затем извлекаем их, но уже в отсортированном порядке. Этот метод сортировки по традиции продолжают называть пирамидальной сортировкой heapsort, хотя он работает с любой очередью с приоритетами.

```
from ds2.priorityqueue import PriorityQueue

def heapsort(L):
    H = PriorityQueue(L)
    L[:] = [item for item in H]

L = [3,2,4,1, 6, 5]
print("before heapsort:", L)
heapsort(L)
print("after heapsort: ", L)

before heapsort: [3, 2, 4, 1, 6, 5]
after heapsort:  [1, 2, 3, 4, 5, 6]
```

Эффективен ли этот код? Инициализация очереди с приоритетами PriorityQueue требует лишь времени O(n) при использовании метода _heapify. Но на итеративный проход затрачивается время O(log n) для каждого элемента. Таким образом, общее время выполнения равно O(n log n), т. е. точно такое же, как для методов сортировки mergesort (сортировка слиянием) и quicksort (быстрая сортировка).

Глава 20

Графы

Граф (graph) – это один из основополагающих математических объектов в информатике. Графы используются для абстрактного представления сетей, отображений, потоков управления программами, вероятностных моделей и даже структур данных, а также многих других концепций информатики.

Для начала неплохой мысленной (ментальной) моделью может стать карта дорог страны. На ней изображаются города и дороги, соединяющие их. Графы идеальны для описания таких ситуаций, в которых существуют некоторые элементы и соединения между ними.

Формально граф представляет собой пару (V, E), где V – любое множество, а E – множество пар элементов V. Мы называем V множеством вершин (vertex set), а E – множеством ребер (edge set). Используя только определение графа и стандартные наборы данных языка Python, можно хранить граф, как показано ниже.

```
G = ({1,2,3,4}, {(1,2), (1,3), (1,4)})
```

Этот граф G содержит четыре вершины и три ребра. Часто подобный граф изображают с помощью четырех небольших кружков с соответствующими метками, представляющими вершины, и отрезками прямых между ними, представляющими ребра. Поскольку кортежи являются упорядоченными структурами, ребра могут изображаться стрелками. В этом случае представлен направленный (или ориентированный) граф (directed graph или digraph). Если порядок вершин в графе не имеет значения, то получаем ненаправленный граф (undirected graph). Между ненаправленным и направленным графом нет существенного различия, если направленный граф содержит пару (v, u) для каждой пары (u, v) в E.

Как правило, мы будем запрещать наличие петель (self-loops), т. е. ребер, которые начинаются и заканчиваются в одной и той же вершине, а также нескольких параллельных (или кратных) ребер между одной парой вершин. Графы без петель и параллельных ребер называются простыми (simple graphs).

Если две вершины соединены ребром, то мы говорим, что они смежные (adjacent). Все смежные вершины называются соседними (neighbors). Для ребра e = (u, v) мы говорим, что вершины u и v инцидентны (incident) ребру e. Степень вершины (degree of vertex) – это число ее соседей. Для направленных графов различаются степени входа (in-degree) и выхода (out-degree) – число соседей с входящими и исходящими ребрами соответственно.

20.1. Абстрактный тип данных «граф»

Сначала установим минимальный абстрактный тип данных (АТД) для простого ненаправленного графа. Мы должны иметь возможность использования любого (хешируемого) типа для вершин. Также необходима возможность создания графа из набора вершин и из набора упорядоченных пар вершин.

- `__init__(V, E)` – инициализация нового графа множеством вершин `V` и множеством ребер `E`.
- `vertices()` – возвращает итерируемый набор вершин.
- `edges()` – возвращает итерируемый набор ребер.
- `addvertex(v)` – добавляет новую вершину в граф. Эта новая вершина идентифицируется объектом `v`.
- `addedge(u, v)` – добавляет новое ребро в граф между вершинами с ключами `u` и `v`.
- `removeedge(u, v)` – удаляет ребро `u`, `v` из графа.
- `__contains__(v)` – возвращает `True`, если вершина `v` содержится в графе, иначе возвращает `False`.
- `hasedge(u, v)` – возвращает `True`, если ребро `(u,v)` содержится в графе, иначе возвращает `False`.
- `nbrs(v)` – возвращает итерируемый набор соседей (или исходящих соседей) вершины `v`, т. е. таких вершин `w`, для которых существует ребро `(v,w)`. (Для направленных графов возвращается набор исходящих соседей.)
- `__len__()` – возвращает число вершин в графе.

Необходимо рассмотреть несколько условий возникновения ошибок. Например, что происходит, если вершина добавляется более одного раза? Что, если добавляется ребро `(u,v)`, но `u` не является вершиной данного графа? Что, если выполняется вызов `nbrs(v)`, но `v` не является вершиной данного графа? Но мы займемся решением этих проблем позже. А сейчас просто обеспечим работу методов графа.

20.2. Реализация класса EdgeSetGraph

Изощренность ума будем проявлять позже. Сначала напишем класс графа, максимально приближенный к его математическому определению. Он будет хранить множество вершин и множество пар (кортежей из двух элементов) для ребер. Для получения соседей некоторой вершины будет выполняться итеративный проход по всем ребрам в графе.

```
class EdgeSetGraph:
    def __init__(self, V = (), E = ()):
        self._V = set()
        self._E = set()
        for v in V: self.addvertex(v)
        for u,v in E: self.addedge(u,v)

    def vertices(self):
        return iter(self._V)
```

```
    def edges(self):
        return iter(self._E)

    def addvertex(self, v):
        self._V.add(v)

    def addedge(self, u, v):
        self._E.add((u,v))

    def removeedge(self, u, v):
        self._E.remove((u,v))

    def __contains__(self, v):
        return v in self._V

    def hasedge(self, u, v):
        return (u,v) in self._E

    def nbrs(self, v):
        return (w for u,w in self._E if u == v)

    def __len__(self):
        return len(self._V)
```

Для создания ненаправленной версии этого класса можно было бы заменить используемые здесь кортежи на множества. Но возникает проблема: Python не допускает использование множеств как элементов множеств. Напомню, что изменяемые типы, такие как множества и списки, нельзя использовать таким способом. К счастью, Python предоставляет тип неизменяемого множества под названием `frozenset`. Это практически то же самое множество, за исключением того, что его нельзя изменять. Неизменяемым множеством можно воспользоваться для создания множества ребер в ненаправленном графе, как показано ниже.

```
class UndirectedEdgeSetGraph(EdgeSetGraph):
    def addedge(self, u, v):
        self._E.add(frozenset({u,v}))

    def removeedge(self, u, v):
        self._E.remove(frozenset({u,v}))

    def nbrs(self, v):
        for u, w in self._E:
            if u == v:
                yield w
            elif w == v:
                yield u
```

20.3. Реализация класса AdjacencySetGraph

В реализации из предыдущего раздела главная проблема заключалась в чрезвычайно медленном выполнении перечисления всех соседей некоторой заданной вершины. Нежелательно последовательно проходить по всем ребрам только для того, чтобы найти инцидентные вершины.

При другом подходе нужно хранить множество с каждой вершиной и сделать так, чтобы это множество содержало всех соседей этой вершины. Это позволит получить быстрый доступ к соседним вершинам. В действительности это означает, что ребра вообще не нужно хранить в явной форме.

```
class AdjacencySetGraph:
    def __init__(self, V = (), E = ()):
        self._V = set()
        self._nbrs = {}
        for v in V: self.addvertex(v)
        for e in E: self.addedge(*e)

    def vertices(self):
        return iter(self._V)

    def edges(self):
        for u in self._V:
            for v in self.nbrs(u):
                yield (u,v)

    def addvertex(self, v):
        self._V.add(v)
        self._nbrs[v] = set()

    def addedge(self, u, v):
        self._nbrs[u].add(v)

    def removeedge(self, u, v):
        self._nbrs[u].remove(v)

    def __contains__(self, v):
        return v in self._nbrs

    def nbrs(self, v):
        return iter(self._nbrs[v])

    def __len__(self):
        return len(self._nbrs)

G = AdjacencySetGraph({1,2,3}, {(1,2),(2,1),(1,3)})
print("neighbors of 1:", list(G.nbrs(1)))
print("neighbors of 2:", list(G.nbrs(2)))
print("neighbors of 3:", list(G.nbrs(3)))

neighbors of 1: [2, 3]
neighbors of 2: [1]
neighbors of 3: []
```

Выполним чрезвычайно простой пример, чтобы убедиться в том, что все работает, как предполагалось.

Проектным решением, которое может выглядеть достаточно странным, является то, что у нас нет классов для вершин и ребер. Начиная с этого момента, мы применяем полиморфизм (polymorphism) Python, который позволяет использовать любой (хешируемый) тип для множества вершин. Напомню, что

полиморфизм (в текущем контексте) – это возможность вызывать одну и ту же функцию для объектов различных типов. Это может оказаться весьма удобным, поскольку позволяет разместить структуру графа поверх любого множества.

Как было отмечено выше, нет явной причины использовать для хранения соседей множество вместо списка. Единственный случай, когда множество предпочтительнее, – если нужно проверить, содержит ли граф конкретное ребро. В приведенном выше классе `AdjacencySetGraph` этот метод можно реализовать следующим образом.

```
    def hasedge(self, u, v):
        return v in self._nbrs[u]
```

Если бы `self._nbrs[u]` был списком, то для выполнения этого метода потребовалось бы линейное время относительно степени вершины `u`.

Для создания версии для ненаправленных графов класса `AdjacencySetGraph` мы позволим ему вести себя как направленный граф, в котором каждое ребро имеет двойника в противоположном направлении. Такой подход повлияет на методы `addedge` и `removeedge`. Кроме того, потребуется уделить немного больше внимания методу `edges`, чтобы он не возвращал в два раза больше ребер, как ранее. Для устранения дублирования мы, как и в предыдущей версии, будем использовать неизменяемые множества (`frozenset`).

```
from ds2.graph import AdjacencySetGraph

class UndirectedAdjacencySetGraph(AdjacencySetGraph):
    def addedge(self, u, v):
        AdjacencySetGraph.addedge(self, u, v)
        AdjacencySetGraph.addedge(self, v, u)

    def removeedge(self, u, v):
        AdjacencySetGraph.removeedge(self, u, v)
        AdjacencySetGraph.removeedge(self, v, u)

    def edges(self):
        E = {frozenset(e) for e in AdjacencySetGraph.edges(self)}
        return iter(E)
```

20.4. Пути и связность

Путь (path) в графе $G = (V, E)$ – это последовательность вершин, соединенных ребрами. То есть не пустая последовательность вершин $(v_0, v_1, \ldots, v_k)$ является путем из v_0 в v_k, если и только если существуют ребра $(v_{i-1}, v_i) \in E$ для всех $i \in 1, \ldots, k$. Мы говорим, что путь является простым (simple path), если он не посещает повторно ни одну из вершин. Длина пути (length of path) – это число ребер. Одна вершина может рассматриваться как путь длиной ноль.

Цикл (cycle) – это путь длиной не менее единицы, который начинается и заканчивается в одной и той же вершине. Длина цикла (length of cycle) равна числу ребер в нем. Цикл называется простым (simple cycle), если он действительно является циклом и удаление последнего ребра превращает его в простой путь, т. е. в нем нет повторяющихся вершин, за исключением самой последней.

Для закрепления приведенных выше определений можно написать несколько методов, проверяющих их соблюдение.

```
    def ispath(self, V):
        """Return True if and only if the vertices V form a path."""
        # Возвращает True, если и только если вершины V формируют путь.
        return V and all(self.hasedge(V[i-1], V[i]) for i in range(1, len(V)))

    def issimplepath(self, V):
        """Return True if and only if the vertices V form a simple path."""
        # Возвращает True, если и только если вершины V формируют простой путь.
        return self.ispath(V) and len(V) == len(set(V))

    def iscycle(self, V):
        """Return True if and only if the vertices V form a cycle."""
        # Возвращает True, если и только если вершины V формируют цикл.
        return self.ispath(V) and V[0] == V[-1]

    def issimplecycle(self, V):
        """Return True if and only if the vertices V form a simple cycle."""
        # Возвращает True, если и только если вершины V формируют простой цикл.
        return self.iscycle(V) and self.issimplepath(V[:-1])

G = AdjacencySetGraph({1,2,3,4}, {(1,2),(3,1), (2,3), (3,4), (4,3)})
print("[1,2,3,1] is a path", G.ispath([1,2,3,1]))
print("[1,2,3,1] is a simple path", G.issimplepath([1,2,3,1]))
print("[1,2,3] is a simple path", G.issimplepath([1,2,3]))
print("[1,2,3] is a simple cycle:", G.issimplecycle([1,2,3]))
print("[1,2,3,1] is a simple cycle:", G.issimplecycle([1,2,3]))
print("[1,2,3,4] is a simple path:", G.issimplepath([1,2,3,4]))
print("[1,2,3,4] is a simple cycle:", G.issimplecycle([1,2,3,4]))
print("[1,2,3,4,3,1] is a cycle:", G.iscycle([1,2,3,4,3,1]))
print("[1,2,3,4,3,1] is a simple cycle:", G.issimplecycle([1,2,3,4,3,1]))
```

```
[1,2,3,1] is a path True
[1,2,3,1] is a simple path False
[1,2,3] is a simple path True
[1,2,3] is a simple cycle: False
[1,2,3,1] is a simple cycle: False
[1,2,3,4] is a simple path: True
[1,2,3,4] is a simple cycle: False
[1,2,3,4,3,1] is a cycle: True
[1,2,3,4,3,1] is a simple cycle: False
```

Мы говорим, что вершина u связана с вершиной v, если существует путь, начинающийся в u и заканчивающийся в v. Для ненаправленных графов, если u связана с v, то v связана с u. В таких графах можно разделить вершины на подмножества, называемые связными компонентами (connected components), в которых все вершины попарно связаны.

Для направленных графов две вершины u и v являются сильно связанными (strongly connected), если u связана с v, а также v связана с u.

Рассмотрим простой метод проверки связности двух вершин. Мы добавим его в класс `AdjacencySetGraph`. В качестве начального упражнения можно попытаться заставить этот метод работать рекурсивно. Идея проста: в базовом

случае проверяем, не пытаемся ли мы перейти от вершины к самой себе. В противном случае достаточно проверить, не соединены ли какие-либо из соседей первой вершины с последней.

```
    def connected(self, a, b):
        if a == b: return True
        return any(self.connected(nbr, b) for nbr in self.nbrs(a))
```

Сможете ли вы предсказать, что здесь пойдет не так? Ниже показан пример выполнения.

```
G = AdjacencySetGraph({1,2,3}, {(1,2), (2,3)})
assert(G.connected(1,2))
assert(G.connected(1,3))
assert(not G.connected(3,1))
print("First graph is okay.")

H = AdjacencySetGraph({1,2,3}, {(1,2), (2,1), (2,3)})
try:
    H.connected(1,3)
except RecursionError:
    print("There was too much recursion!")

First graph is okay.
There was too much recursion!
```

Здесь очевидно, что если граф содержит циклы, то предложенным выше способом мы не сможем проверить связность. Для обработки циклов можно сохранять множество посещенных вершин. Напомню, что такой подход называется мемоизацией.

```
    def connected(self, a, b):
        return self._connected(a, b, set())

    def _connected(self, a, b, visited):
        if a in visited: return False
        if a == b: return True
        visited.add(a)
        return any(self._connected(nbr, b, visited) for nbr in self.nbrs(a))
```

Теперь можно попытаться выполнить проверку еще раз и посмотреть, что происходит.

```
H = AdjacencySetGraph({1,2,3}, {(1,2), (2,1), (2,3)})
try:
    assert(H.connected(1,2))
    assert(H.connected(1,3))
except RecursionError:
    print('There was too much recursion!')
print('It works now!')
```

Глава 21

Поиск в графах

Теперь, когда у нас есть словарь терминов и определений для работы с графами, можно повторно рассмотреть некоторые предыдущие темы. Например, теперь дерево можно рассматривать как граф с ребрами, направленными от родителей к потомкам. Если задан граф, представляющий дерево с корнем, то мы могли бы переписать предыдущую версию обхода в прямом порядке, используя абстрактный тип данных (АТД) «граф». Предположим, что сначала нужно просто вывести все вершины.

```
def printall(G, v):
    print(v)
    for n in G.nbrs(v):
        printall(G, n)

G = Graph({1,2,3,4}, {(1,2), (1,3), (1,4)})
printall(G, 1)
```

```
1
2
3
4
```

Это вполне подходит для дерева, но сразу становится весьма неудачным вариантом, как только появляется цикл. В данном случае в коде нет никаких средств, защищающих от многократно повторяющихся проходов по циклу. В итоге получим исключение `RecursionError`. Применим способ Гензеля и Гретель (из сказки братьев Гримм): будем оставлять хлебные крошки в тех местах, где мы уже побывали. Самый простой способ сделать это – просто постоянно поддерживать (и обновлять) множество, хранящее вершины, которые уже были посещены в процессе поиска. Таким образом, метод `printall` приобретает следующий вид.

```
def printall(G, v, visited):
    visited.add(v)
    print(v)
    for n in G.nbrs(v):
        if n not in visited:
            printall(G, n, visited)
```

```
G = Graph({1,2,3,4}, {(1,2), (2,3), (3,4), (4,1)})
printall(G, 1, set())

1
2
3
4
```

Это самый прямой способ обобщения процедуры рекурсивного обхода дерева в некоторый метод, который также выполняет обход вершин графа. В отличие от деревьев (с корнем) мы должны определить начальную вершину, потому что в графе нет специально выделенной «корневой» вершины. Кроме того, приведенный выше код не всегда выводит все вершины в графе. В рассматриваемом здесь случае он будет выводить только те вершины, которые связаны с начальной. Этот шаблон необходимо доработать, чтобы он подходил для более общих случаев.

21.1. Поиск в глубину

Поиск в глубину (*depth-first search* – DFS) в графе G, начиная с вершины v, посещает все вершины, связанные с v. При этом приоритет всегда отдается «исходящему» перемещению в направлении новых вершин с минимально возможными возвратами. Приведенный выше метод `printall` выводит вершины в порядке, соответствующем порядку обхода при поиске в глубину. Ниже приведена обобщенная форма этого алгоритма.

```
def dfs(G, v):
    visited = {v}
    _dfs(G, v, visited)
    return visited

def _dfs(G, v, visited):
    for n in G.nbrs(v):
        if n not in visited:
            visited.add(n)
            _dfs(G, n, visited)

G = Graph({1,2,3,4}, {(1,2), (2,3), (3,4), (4,2)})
print('reachable from 1:', dfs(G, 1))
print('reachable from 2:', dfs(G, 2))

reachable from 1: {1, 2, 3, 4}
reachable from 2: {2, 3, 4}
```

С помощью этого кода легко проверить, являются ли две вершины связанными. Например, для этого можно написать следующий код.

```
def connected(G, u, v):
   return v in dfs(G, u)
```

Можно попробовать использовать этот код в небольшом примере, чтобы посмотреть, как он выполняется. Обратите внимание: код работает с направленным графом.

```
G = Graph({1,2,3,4}, {(1,2), (2,3), (3,4), (4,2)})

print("1 is connected to 4:", connected(G, 1, 4))
print("4 is connected to 3:", connected(G, 4, 3))
print("4 is connected to 1:", connected(G, 4, 1))

1 is connected to 4: True
4 is connected to 3: True
4 is connected to 1: False
```

Есть возможность изменить код метода `dfs`, чтобы обеспечить поддержку не только множества связанных вершин, но также путей, используемых в процессе поиска. Идея заключается в хранении словаря, который отображает вершины в предыдущие в пути от начальной вершины. Это действительно позволяет закодировать дерево как словарь, отображающий узлы в своих родителей. Корнем является начальная вершина.

Это отличается от деревьев, которые мы видели ранее, тем, что естественным образом выполняется подъем по дереву от листьев к корню, а не наоборот. Полученное дерево называется деревом поиска в глубину (depth-first search tree). Для его создания требуется лишь небольшое изменение. Мы используем соглашение, согласно которому начальная вершина отображается в `None`.

```
def dfs(G, v):
    tree = {v: None}
    _dfs(G, v, tree)
    return tree

def _dfs(G, v, tree):
    for n in G.nbrs(v):
        if n not in tree:
            tree[n] = v
            _dfs(G, n, tree)

G = Graph({1,2,3,4}, {(1,2), (2,3), (3,4), (4,2)})
print('dfs tree from 1:', dfs(G, 1))
print('dfs tree from 2:', dfs(G, 2))

dfs tree from 1: {1: None, 2: 1, 3: 2, 4: 3}
dfs tree from 2: {2: None, 3: 2, 4: 3}
```

21.2. Исключение рекурсии

Приведенный в предыдущем разделе код метода `dfs` использует рекурсию для отслеживания предыдущих вершин, поэтому есть возможность возвращаться (`return`) при достижении вершины, из которой невозможно продвинуться вперед. Для исключения рекурсии заменим стек вызовов функций на собственный стек.

Это не просто учебное упражнение. Исключая рекурсию, мы открываем структуру алгоритма таким способом, который позволяет обобщить его для других алгоритмов. Ниже приведен исходный код.

```
def dfs(self, v):
    tree = {}
    tovisit = [(None, v)]
    while tovisit:
        a,b = tovisit.pop()
        if b not in tree:
            tree[b] = a
            for n in self.nbrs(b):
                tovisit.append((b,n))
    return tree

G = Graph({1,2,3,4}, {(1,2), (2,3), (3,4), (4,2)})
print('dfs tree from 1:', dfs(G, 1))
print('dfs tree from 2:', dfs(G, 2))
```

```
dfs tree from 1: {1: None, 2: 1, 3: 2, 4: 3}
dfs tree from 2: {2: None, 3: 2, 4: 3, 1: 2}
```

21.3. Поиск в ширину

Можно получить другой важный способ обхода, если заменить стек очередью. В этом случае процесс поиска отдает приоритет ширине перед глубиной, в итоге мы получаем поиск в ширину (*breadth-first search* – BFS).

```
def bfs(G, v):
    tree = {}
    tovisit = Queue()
    tovisit.enqueue((None, v))
    while tovisit:
        a,b = tovisit.dequeue()
        if b not in tree:
            tree[b] = a
            for n in G.nbrs(b):
                tovisit.enqueue((b,n))
    return tree

G = Graph({1,2,3,4}, {(1,2), (2,3), (3,4), (4,2)})
print('bfs tree from 1:', bfs(G, 1))
print('bfs tree from 2:', bfs(G, 2))
```

```
bfs tree from 1: {1: None, 2: 1, 4: 2, 3: 2}
bfs tree from 2: {2: None, 1: 2, 4: 2, 3: 2}
```

Замечательным свойством поиска в ширину является то, что пути, закодированные в итоговом дереве поиска в ширину (breadth-first search tree), являются кратчайшими путями из каждой вершины в начальную. Таким образом, поиск в ширину можно использовать для нахождения кратчайшего пути, соединяющего пары вершин в графе.

```
def distance(G, u, v):
    tree = G.bfs(u)
    if v not in tree:
        return float('inf')
    edgecount = 0
    while v is not u:
        edgecount += 1
        v = tree[v]
    return edgecount

G = Graph({1,2,3,4,5}, {(1,2), (2,3), (3,4), (4,5)})
print("distance from 1 to 5:", distance(G, 1, 5))
print("distance from 2 to 5:", distance(G, 2, 5))
print("distance from 3 to 4:", distance(G, 3, 4))
```

21.4. Взвешенные графы и кратчайшие пути

В задаче о поиске кратчайшего пути из одной вершины графа во все остальные (*single source shortest path* – SSSP) целью является поиск кратчайшего пути из каждой вершины в заданную исходную вершину. Если предполагается, что ребра имеют одинаковую длину, то поиск в ширину решает эту задачу. Но в более общем смысле рассматривается вариант с взвешенными графами (weighted graphs), где (положительное) действительное число называется весом (weight), присвоенном каждому ребру. Мы усовершенствуем АТД «граф» для поддержки функции `wt(u,v)`, которая возвращает вес ребра. Тогда вес пути – это сумма весов ребер, составляющих этот путь. Ниже приведены простые примеры, позволяющие понять, что кратчайший путь может оказаться не тем, что мы получаем из дерева поиска в ширину.

```
from ds2.graph import AdjacencySetGraph
from ds2.priorityqueue import PriorityQueue

class Digraph(AdjacencySetGraph):
    def addedge(self, u, v, weight = 1):
        self._nbrs[u][v] = weight

    def removeedge(self, u, v):
        del self._nbrs[u][v]

    def addvertex(self, v):
        self._V.add(v)
        self._nbrs[v] = {}

    def wt(self, u, v):
        return self._nbrs[u][v]

from ds2.graph import Digraph

class Graph(Digraph):
    def addedge(self, u, v, weight = 1):
        Digraph.addedge(self, u, v, weight)
        Digraph.addedge(self, v, u, weight)
```

```
    def removeedge(self, u, v):
        Digraph.removeedge(self, u, v)
        Digraph.removeedge(self, v, u)

    def edges(self):
        E = {frozenset(e) for e in Digraph.edges(self)}
        return iter(E)
```

Эффективный алгоритм для решения задачи о поиске кратчайшего пути из одной вершины графа во все остальные (SSSP) называется алгоритмом Дейкстры (Dijkstra algorithm). Он во многом напоминает алгоритмы поиска в глубину и в ширину, за исключением того, что стек или очередь заменены на очередь с приоритетами. Вершины посещаются в порядке их расстояния от исходной вершины. Эти расстояния будут использоваться как приоритеты в очереди.

Мы рассмотрим две различные реализации. Первая, несмотря на то что менее эффективна, чрезвычайно близка к алгоритмам поиска в глубину и в ширину. Напомню, что в этих алгоритмах мы посещаем вершины, записывая пройденные ребра в словарь и добавляя все соседние вершины в стек или очередь для обхода в дальнейшем. Здесь мы будем делать то же самое, за исключением того, что для хранения искомых ребер будет использоваться очередь с приоритетами. Также поддерживается словарь расстояний от начальной вершины, обновляющийся при посещении каждой другой вершины. Приоритетом для ребра (u,v) является расстояние до u плюс вес ребра (u,v). Поэтому, если мы используем это ребро, то кратчайший путь до v будет проходить через u. При таком подходе дерево кодирует все кратчайшие пути из конкретной начальной вершины. Таким образом, результатом будут не только длины всех этих путей, но и эффективное кодирование (обозначение) самих кратчайших путей.

```
def dijkstra(G, v):
    tree = {}
    D = {v: 0}
    tovisit = PriorityQueue()
    tovisit.insert((None,v), 0)
    for a,b in tovisit:
        if b not in tree:
            tree[b] = a
            if a is not None:
                D[b] = D[a] + G.wt(a,b)
            for n in G.nbrs(b):
                tovisit.insert((b,n), D[b] + G.wt(b,n))
    return tree, D
```

Напишем небольшой фрагмент кода, чтобы посмотреть, как работает алгоритм Дейкстры. Добавим функцию для записи пути в дерево, затем выведем все кратчайшие пути, найденные этим алгоритмом.

```
def path(tree, v):
    path = []
    while v is not None:
        path.append(str(v))
        v = tree[v]
    return ' --> '.join(path)
```

```
def shortestpaths(G, v):
    tree, D = dijkstra(G, v)
    for v in G.vertices():
        print('Vertex', v, ':', path(tree, v), ", distance = ", D[v])

G = Graph({1,2,3}, {(1,2, 4.6), (2, 3, 9.2), (1, 3, 3.1)})
shortestpaths(G, 1)
print('------------------------')
# Добавление ребра создает сокращенный путь к вершине 2.
G.addedge(3, 2, 1.1)
shortestpaths(G, 1)

Vertex 1 : 1 , distance = 0
Vertex 2 : 2 --> 1 , distance = 4.6
Vertex 3 : 3 --> 1 , distance = 3.1
------------------------
Vertex 1 : 1 , distance = 0
Vertex 2 : 2 --> 3 --> 1 , distance = 4.2
Vertex 3 : 3 --> 1 , distance = 3.1
```

21.5. Алгоритм Прима для минимальных остовных деревьев

Напомню, что подграфом ненаправленного графа G = (V, E) является остовное дерево (spanning tree), если это дерево с множеством вершин V. Для взвешенного графа весом остовного дерева является сумма весов его ребер. Задача поиска минимального остовного дерева (*minimum spanning tree* – MST) заключается в поиске для исходного графа остовного дерева с минимальным весом. На практике эта задача встречается во многих контекстах.

При поиске алгоритма решения для этой задачи мы начинаем с попытки описать, какие ребра должны находиться в искомом минимальном остовном дереве. То есть мы должны решить, какой объект необходимо создать в первую очередь, и только потом подумать, как это сделать. Применим стратегию, что рассматривалась для алгоритмов сортировки. В данном случае мы сначала пытаемся написать функцию, которая должна проверять корректность вывода. Не будем делать это прямо сейчас, но зададим вопрос: «Как мы можем узнать, что получили именно минимальное остовное дерево?»

В отношении минимального остовного дерева определенно верно то, что, если бы мы удалили ребро (в результате получаются два дерева), мы не смогли бы найти ребро с меньшим весом, соединяющее эти два дерева. В противном случае это было бы остовное дерево с меньшим весом.

Истинным является даже еще несколько менее общее утверждение. Если мы разделяем вершины на любые два множества A и B, то ребро с наименьшим весом, один конец которого находится в A, а другой – в B, непременно должно содержаться в минимальном остовном дереве. Затем мы могли бы добавить это ребро и сформировать цикл, который должен содержать другое ребро, перекрывающее A и B. В дальнейшем это ребро можно было бы удалить, оставив остовное дерево с меньшим весом. Но это является противоречием, потому что мы предположили, что начинаем с минимальным остовным деревом.

Поэтому с учетом предыдущих алгоритмов для графов можно попытаться всегда добавлять ребро с минимальным весом от ранее посещенных вершин в некоторую не посещенную вершину. Это можно легко закодировать в очереди с приоритетами.

```
def prim(G):
    v = next(iter(G.vertices()))
    tree = {}
    tovisit = PriorityQueue()
    tovisit.insert((None, v), 0)
    for a, b in tovisit:
        if b not in tree:
            tree[b] = a
            for n in G.nbrs(b):
                tovisit.insert((b,n), G.wt(b,n))
            return tree
```

В приведенном ниже примере явно показано, что минимальное остовное дерево не совпадает с деревом кратчайшего пути.

```
G = Graph({1,2,3,4,5}, {(1, 2, 1),
                        (2, 3, 1),
                        (1, 3, 2),
                        (3, 4, 1),
                        (3, 5, 3),
                        (4, 5, 2),
                       })
mst = prim(G)
sp, D = dijkstra(G, 1)
print(mst)
print(sp)

{1: None, 2: 1, 3: 2, 4: 3, 5: 4}
{1: None, 2: 1, 3: 1, 4: 3, 5: 3}
```

21.6. Оптимизация поиска по первому наилучшему (приоритетному) совпадению

Приведенная в предыдущем разделе реализация, хотя корректна, с технической точки зрения не является алгоритмом Дейкстры, тем не менее близка к нему. Пытаясь улучшить асимптотическое время выполнения этой реализации, мы получим настоящий алгоритм Дейкстры. При размышлении над тем, как улучшить алгоритм, в первую очередь внимание привлекает проделанная впустую работа. В данном случае можно видеть, что многие ребра, добавляемые в очередь с приоритетами, впоследствии удаляются без какого-либо использования, потому что ведут к вершинам, которые ранее уже были посещены (более коротким путем).

Было бы лучше вообще не включать такие ребра в очередь с приоритетами, но, когда мы впервые видим ребро, неизвестно, будет ли оно обеспечивать кратчайший путь. Можно избежать добавление нового элемента в приоритет, а вместо этого изменить существующий элемент, который уже не является правильным.

Идея заключается в хранении вершин вместо ребер в очереди с приоритетами. Затем мы воспользуемся методом `changepriority` для обновления элемента при обнаружении нового более короткого пути к конкретной вершине. Несмотря на то что изначально не известны все расстояния, мы будем сохранять кратчайшее расстояние, найденное на текущий момент. Если обнаружен более короткий путь к рассматриваемой вершине, то его приоритет снижается, а очередь с приоритетами обновляется. Обновление после обнаружения более короткого пути называется ослаблением (релаксацией) ребра (edge relaxation). Это работает следующим образом. Расстояния до исходной вершины сохраняются в словаре D, который отображает вершины в соответствующие расстояния, основываясь на текущих результатах поиска. Если обнаружено, что `D[n] > D[u] + G.wt(u,n)`, то это должен быть более короткий путь в `n`, если взять кратчайший путь из исходной вершины в `u` и добавить ребро `(u,n)`. В таком случае мы устанавливаем `D[n] = D[u] + G.wt(u,n)` и обновляем очередь с приоритетами. Обратите внимание: именно этот алгоритм мы имели в виду, когда добавляли метод `changepriority` в реализацию АТД очередь с приоритетами.

Ниже приведен исходный код реализации этого алгоритма.

```
def dijkstra2(G, v):
    tree = {v: None}
    D = {u: float('inf') for u in G.vertices()}
    D[v] = 0
    tovisit = PriorityQueue(entries = [(u, D[u]) for u in G.vertices()])
    for u in tovisit:
        for n in G.nbrs(u):
            if D[u] + G.wt(u,n) < D[n]:
                D[n] = D[u] + G.wt(u,n)
                tree[n] = u
                tovisit.changepriority(n, D[n])
    return tree, D

from ds2.graph import Digraph

V = {1,2,3,4,5}
E = {(1,2,1),
     (2,3,2),
     (1,3,2),
     (3,4,2),
     (2,5,2)
}
G = Digraph(V, E)
tree, D = dijkstra2(G, 1)
print(tree, D)

{1: None, 2: 1, 3: 1, 5: 2, 4: 3} {1: 0, 2: 1, 3: 2, 4: 4, 5: 3}
```

Важное различие между исходным кодом реализации поиска в глубину и в ширину заключается в том, что теперь основная структура данных хранит вершины, а не ребра. Кроме того, не требуется проверка того, что мы уже посещали некоторую вершину, потому что если да, то мы не будем искать ослабляемое ребро (вершины посещаются в порядке их расстояний до исходной вершины).

Глава 22

(Непересекающиеся) множества

В главе об отображениях было описано, как мы можем рассчитывать на получение постоянного времени выполнения многих операций с множеством. Там также пояснялось, почему невозможно создать множество множеств в языке Python (по крайней мере, напрямую). Элементы множества обязательно должны быть хешируемыми, потому что они хранятся в хеш-таблице. Если требуется неизменяемое множество, то можно воспользоваться неизменяемой структурой данных `frozenset` для хранения множеств с сохранением многих преимуществ типа `set`, а также свойства хешируемости.

Но существует обобщенная задача структурирования данных, требующая хранения набора множеств, способных изменяться. Она называется задачей о непересекающихся множествах (disjoint sets). Здесь мы имеем набор объектов (сущностей), которые группируются в непересекающиеся множества. Желательно получить возможность быстро определять, находятся ли некоторые два объекта в одной группе. С другой стороны, иногда нам сообщается, что два объекта присутствуют в одной группе, и в этом случае, возможно, потребуется изменение основного принципа группировки – объединение двух групп в одну.

Структуру данных для хранения набора непересекающихся множеств часто называют структурой данных для объединения–поиска (union-find), потому что она поддерживает (как можно догадаться) операции объединения `union` и поиска find.

22.1. Абстрактный тип данных «непересекающиеся множества»

- `union(a,b)` – заменяет множества, содержащие элементы `a` и `b`, на одно множество, представляющее собой их объединение.
- `find(a,b)` – возвращает `True`, если `a` и `b` находятся в одном множестве. Иначе возвращает `False`.

22.2. Простая реализация

Чрезвычайно простым подходом является хранение множеств и отображение, связывающее каждый элемент с множеством, которое содержит его. Поскольку множества непересекающиеся, существует единственное в своем роде множество, содержащее каждый элемент, с соответствующим отображением. Множества являются значениями (не ключами) в таком отображении, поэтому нет необходимости в том, чтобы они были хешируемыми. Ниже приведен исходный код этой реализации.

```
class DisjointSetsMapping:
    def __init__(self, L):
        self._map = {item : {item} for item in L}

    def find(self, a, b):
        return a in self._map[b]

    def union(self, a, b):
        if not self.find(a,b):
            union = self._map[a] | self._map[b]
            for item in union:
                self._map[item] = union
```

Рассмотрим другой подход. Можно было бы просто пометить элементы. Если два элемента имеют одинаковую метку, то они содержатся в одном и том же множестве. Объединение просто изменяет метки элементов в одном из множеств. При таком подходе сами по себе множества становятся неявными.

```
class DisjointSetsLabels:
    def __init__(self, L):
        self._label = {item : item for item in L}

    def find(self, a, b):
        return self._label[a] is self._label[b]

    def union(self, a, b):
        if not self.find(a,b):
            for key, value in self._label.items():
                if value is self._label[b]:
                    self._label[key] = self._label[a]
```

Возможно, эта реализация хуже в некоторых отношениях. Приходится выполнять итеративный проход по всем меткам, а не по некоторому связанному их подмножеству. Но есть и преимущество – мы избавляемся от всех операций по объединению. Но опять же, возможно, это не улучшает алгоритм. Можно попытаться сделать операции объединения намного более быстрыми, изменяя меньшее число меток. В следующей реализации мы это сделаем. Вместо отображения элементов в метки мы отображаем их в элемент, который будем называть их родителем. Если каждый узел имеет единственного родителя и не существует циклов, то получаем лес (forest). Лес – это набор деревьев.

```
class DisjointSetsForest:
    def __init__(self, L):
        self._parent = {item : item for item in L}

    def _root(self, item):
        while item is not self._parent[item]:
            item = self._parent[item]
        return item

    def find(self, a, b):
        return self._root(a) is self._root(b)

    def union(self, a, b):
        if not self.find(a,b):
            self._parent[self._root(b)] = self._root(a)
```

Это намного лучше? Во многих случаях, вероятно, лучше, но не всегда. Легко написать пример, в котором этот код будет выполняться долго. Например, можно было бы принудительно сделать лес путем. После этого вызов метода `_root` может потребовать времени, пропорционального числу элементов. Если мы намерены улучшить время выполнения в наихудшем случае, то следует улучшить и наши примеры. Будем сохранять пути настолько короткими, насколько это возможно.

22.3. Сжатие пути

Если необходимо избежать весьма многократного обхода длинных путей, то можно просто укорачивать эти пути при каждом обходе. Простым способом сделать это является замена родителей на прародителей при проходе вверх по дереву. Для этого требуется всего лишь одна строка кода. Эффект состоит в том, что глубина каждого узла на пути, который мы проходим, сокращается вдвое (плюс один). Это означает, что самый длинный путь проходится только O(log n) раз, прежде чем он будет сжат до одного ребра.

```
# Методика сжатия пути позволяет сократить путь вдвое.
# Каждый узел в пути к корню обновляется так, чтобы создавалась ссылка на его прародителя.
class DisjointSetsPathCompression:
    def __init__(self, L):
        self._parent = {item : item for item in L}

    def _root(self, item):
        while item is not self._parent[item]:
            parent = self._parent[item]
            item, self._parent[item] = parent, self._parent[parent]
        return item

    def find(self, a, b):
        return self._root(a) is self._root(b)

    def union(self, a, b):
        if not self.find(a,b):
            self._parent[self._root(b)] = self._root(a)
```

Без особых усилий мы действительно можем сжимать пути к корню, выполняя второй проход. При первом проходе мы только лишь находим корень. При втором обновляются все родители так, чтобы они указывали на корень. Ниже приведен обновленный код.

```
# Сжатие пути с помощью двух проходов.
# Переопределение пути к корню: каждый узел в любом случае указывает выше на новый корень.
class DisjointSetsTwoPassPC:
    def __init__(self, L):
        self._parent = {item : item for item in L}

    def _root(self, item):
        root = item
        while root is not self._parent[root]:
            root = self._parent[root]
        self._compress(item, root)
        return root

    def _compress(self, item, newroot):
        while item is not newroot:
            item, self._parent[item] = self._parent[item], newroot

    def find(self, a, b):
        return self._root(a) is self._root(b)

    def union(self, a, b):
        if not self.find(a,b):
            self._parent[self._root(b)] = self._root(a)
```

Если вы действительно пытаетесь оптимизировать этот код, то можете добиться небольшого улучшения, удалив избыточность, связанную с вызовом `_root` дважды для каждого элемента в методе `union` (один раз в `find` и еще раз в операторе `if`).

22.4. Слияние по высоте

Еще один способ, которым можно попытаться сохранять пути короткими, – просто более внимательно следить за тем, какой узел становится новым корнем при выполнении операции объединения `union`. Более высокое дерево должно стать новым корнем. Затем высота не увеличивается до тех пор, если вы не объединяете два дерева с одинаковой высотой.

```
# Слияние по высоте.
class DisjointSetsMergeByHeight:
    def __init__(self, L):
        self._parent = {item : item for item in L}
        self._height = {item : 0 for item in L}

    def _root(self, item):
        while item is not self._parent[item]:
            item = self._parent[item]
        return item
```

```
    def find(self, a, b):
        return self._root(a) is self._root(b)

    def union(self, a, b):
        if not self.find(a,b):
            if self._height[a] < self._height[b]:
                a,b = b,a
            self._parent[self._root(b)] = self._root(a)
            self._height[a] = max(self._height[a], self._height[b] + 1)
```

22.5. Слияние по весу

Вместо оценки высот деревьев можно рассматривать число узлов в них. Если некоторое дерево содержит больше узлов, то, возможно, оно также имеет бóльшую высоту. Преимущество перед методом слияния по высоте заключается в том, что эта информация не изменяется под воздействием сжатия путей. Следовательно, мы можем (и скоро сделаем это) объединить эти приемы.

```
# Слияние по весу.
class DisjointSetsMergeByWeight:
    def __init__(self, L):
        self._parent = {item : item for item in L}
        self._weight = {item : 1 for item in L}

    def _root(self, item):
        while item is not self._parent[item]:
            item = self._parent[item]
        return item

    def find(self, a, b):
        return self._root(a) is self._root(b)

    def union(self, a, b):
        if not self.find(a,b):
            if self._weight[a] < self._weight[b]:
                a,b = b,a
            self._parent[self._root(b)] = self._root(a)
            self._weight[a] += self._weight[b]
```

22.6. Объединение эвристик

Как уже было отмечено ранее, мы можем использовать обе эвристики, объединив слияние по весу со сжатием путей. Оказывается, что это весьма эффективный способ как в теории, так и на практике. Время выполнения n операций (настолько точно, насколько это вообще можно предсказать) пропорционально n.

```
# Слияние по весу и сжатие путей.
class DisjointSets:
    def __init__(self, L):
        self._parent = {item : item for item in L}
        self._weight = {item : 1 for item in L}

    def _root(self, item):
        root = item
        while root is not self._parent[root]:
            root = self._parent[root]
        self._compress(item, root)
        return root

    def _compress(self, item, newroot):
        while item is not newroot:
            item, self._parent[item] = self._parent[item], newroot

    def find(self, a, b):
        return self._root(a) is self._root(b)

    def union(self, a, b):
        if not self.find(a,b):
            if self._weight[a] < self._weight[b]:
                a,b = b,a
            self._parent[self._root(b)] = self._root(a)
            self._weight[a] += self._weight[b]
```

22.7. Алгоритм Краскала

Вполне естественно воспринимать операции объединения `union` и поиска `find` в контексте графов. То есть вы можете считать операцию объединения добавлением ребра в граф, а операцию поиска – ответом на вопрос, связаны ли две вершины. Это удобная и полезная точка зрения, которая естественным образом приводит к алгоритму вычисления минимальных остовных деревьев. Идея такого алгоритма проста: сортировка ребер по весу. Затем выполняется попытка добавления ребер поочередно по одному, пока добавляемое ребро не образует цикл. В результате получим минимальное остовное дерево.

```
from ds2.disjointsets import DisjointSets

def kruskall(G):
    V = list(G.vertices())
    UF = DisjointSets(V)
    edges = sorted(G.edges(), key = lambda e :G.wt(*e))
    T = Graph(V, set())
    for u, v in edges:
        if not UF.find(u, v):
            UF.union(u, v)
            T.addedge(u, v)
    return T
```

```
from ds2.graph import Graph

G = Graph({1,2,3,4,5},
          {(1,2,1),
           (1,3,4),
           (2,3,2),
           (2,4,4),
           (5,3,1),
          })
MST = kruskall(G)
print(list(MST.edges()))
```

```
[frozenset({2, 4}), frozenset({2, 3}), frozenset({3, 5}), frozenset({1, 2})]
```

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Дональд Р. Шихи

Структуры данных в Python: начальный курс

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Снастин А. В.*
Корректор *Абросимова Л. А.*
Верстка *Луценко С. В.*
Дизайн обложки *Бурмистрова Е. А.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 15,11. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Структуры данных в Python: начальный курс

В книге освещаются основополагающие вопросы, относящиеся к структурам данных в Python. Теоретические концепции и абстрактные понятия подкреплены простыми примерами. Порядок следования глав соотносится с задачами обработки структур данных.

В числе рассматриваемых тем:

- основы языка Python;
- принципы объектно-ориентированного программирования;
- методологии тестирования;
- абстрактные типы данных (АТД);
- стратегии решения задач.

Издание предназначено для всех, кто изучает язык программирования Python и предполагает активно использовать как встроенные структуры данных, так и собственные реализации АТД.

Дональд Шихи – адъюнкт-профессор в области информационных технологий в университете Северной Каролины. Он занимается исследованием геометрических алгоритмов. Сфера наибольших интересов Дональда – пересечение геометрических алгоритмов и анализа топологических данных.

ISBN 978-5-93700-110-8