МИНИСТЕРСТВО СЕЛЬСКОГО ХОЗЯЙСТВА РОССИЙСКОЙ ФЕДЕРАЦИИ

РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ - МСХА имени К.А. ТИМИРЯЗЕВА

В.В. Демичев, Д.В. Быков, Д.Э. Храмов, А.Е. Ульянкин, А.Д. Титов

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Учебное пособие

Москва, 2024

УДК 004.42
ББК -018*32.973
А 45

Рецензент:
Сальников С.Г. – руководитель отдела информатизации агропромышленного комплекса Всероссийского института аграрных проблем и информатики им. А.А. Никонова (ВИАПИ) – филиала ФГБНУ ФНЦ ВНИИЭСХ, кандидат физико-математических наук

Демичев В.В., Быков Д.В., Храмов Д.Э., Ульянкин А.Е., Титов А.Д.

Алгоритмизация и программирование: учебное пособие / **А45** В.В. Демичев, Д.В. Быков, Д.Э. Храмов, А.Е. Ульянкин, А.Д. Титов. – М.: Издательство «Научный консультант», 2024. – 248 с.

ISBN 978-5-907933-15-6

Учебное пособие включает методические указания, теоретические положения, материалы для самостоятельной работы и контроля знаний студентов.

В учебном пособии изложены теоретические основы программирования на высокоуровневом языке Python, алгоритмов и структур данных, графов и деревьев.

Предназначено для студентов вузов направления подготовки 09.03.02 «Информационные системы и технологии». Рекомендовано к изданию учебно-методической комиссией института экономики и управления АПК ФГБОУ ВО «РГАУ-МСХА имени К.А. Тимирязева» (протокол № 1 от 30 августа 2024 г.).

УДК 004.42
ББК -018*32.973

ISBN 978-5-907933-15-6

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Учебное пособие подготовлено в соответствии с требованиями Федерального государственного образовательного стандарта высшего профессионального образования (ФГОС ВО) по направлению подготовки 09.03.02 «Информационные системы и технологии».

Учебное пособие охватывает ключевые разделы дисциплины «Алгоритмизация и программирование», включающей основные элементы и концепции теории программирования на высокоуровневым языке Python, базовые структуры данных, алгоритмы поиска и сортировки, а также основы графов и деревьев. На базе пособия организуются практические занятия, самостоятельная работа студентов и контроль усвоения ими знаний. Пособие рассчитано на выработку навыков самостоятельной работы, применению методов алгоритмизации и программирования в других дисциплинах, при курсовом, дипломном проектировании и научных исследованиях.

Содержание учебного пособия позволяет вузам вести подготовку специалистов по специальностям в сфере информатики и вычислительной техники на очном, очно-заочном и заочном отделениях. Оно может быть использовано также при подготовке в сельскохозяйственных вузах студентов и аспирантов специальностей, связанных с информационными технологиями.

При написании пособия учтен опыт преподавания дисциплины «Алгоритмизация и программирование» ФГБОУ ВО «РГАУ-МСХА имени К.А. Тимирязева» и других вузах России.

Практические задания охватывают все ключевые темы, связанные с теорией и практикой программирования на современных высокоуровневых языках, теорий алгоритмов и структур данных.

В каждом разделе приводятся задачи для самостоятельной работы, в ряде случаев повышенной сложности, что требует от студентов предварительного изучения теоретических вопросов на лекциях, по учебникам и учебным пособиям, рекомендованным кафедрами. Контрольные вопросы и задания используются студентами для самоконтроля, а преподавателями – при промежуточной проверке знаний студентов и проведении итоговых контрольных работ.

Учебное пособие подготовили на кафедре статистики и кибернетики: доцент, кандидат экономических наук В.В. Демичев, ассистенты Д.В. Быков, Д.Э. Храмов., Ульянкин А.Е., Титов А.Д.

Раздел 1
Основы языка Python

Практическое задание № 1
«Объекты. Типы объектов. Переменные. Операции в Python»

1. Объекты

Программный код на языке программирования Python располагается в файле с расширением «.py». Такой файл называют **сценарием** или **программой**.

При этом, обычно сценарий – файл, содержащий небольшое число строк кода, тогда как программа – один файл или множество файлов, представляющее собой относительно сложное приложение. Однако зачастую понятия «сценарий» и «программа» являются взаимозаменяемыми.

Пример программы и результат ее выполнения отражен в листинге 1.1.

Листинг 1.1 – Пример простой программы, выводящей в консоль два сообщения: «Hello, Tom», «15»

```
ntitled1.py × | AAP_#1.py × | untitled2.py × | untitled3.py × | untitled4.py × | untitled5.py ×
1  name = 'Tom'
2  print('Hello, ' + name)
3
4  a = 10
5  b = 5
6  print(a + b)
```

```
Hello, Tom
15
```

Сценарий (программа) Python состоит из объектов.

Объект (Object) в языке программирования Python является основным понятием, с помощью которого представляются все элементарные единицы программы (все данные).

Объект – порция памяти со значениями и наборами связанных операций [5].

Объект – абстракция для представления данных [16].

Каждый объект имеет:

- идентификатор (identity),
- тип (type),
- значение (value),
- счетчик ссылок (reference counter).

Идентификатор объекта остается неизменным с момента создания объекта. Идентификатор объекта можно определить как адрес объекта в

памяти компьютера. Получить идентификатор объекта можно с помощью специальной встроенной функции `id()` (пример использования см. в листинге 1.3).

Счетчик ссылок – используется для определения числа ссылок на данный объект. При создании объекта счетчик ссылок увеличивается на 1.

В Python реализован так называемый **сборщик мусора** для автоматического освобождения памяти.

Сборка мусора – заключается в следующем: если счетчик ссылок объекта становится равным 0, то объект автоматически уничтожается.

Пример объекта: конкретное целое число 10 (значение такого объекта: 10; набор операций, связанных с таким объектом: сложение, вычитание, умножение, возведение в степень, операции сравнения и др.).

Более точно разложить программу на составляющие можно с помощью следующей четырехуровневой иерархии, в которой объекты расположены на самом нижнем уровне:

1. Модули (Modules).
2. Инструкции (Statements).
3. Выражения (Expressions).
4. Объекты (Objects).

Модуль (Module) – программа (файл с расширением «.py»), которая может быть подключена к основной программе. При этом, при выполнении инструкций основной программы, Python выполнит все инструкций (весь код) в файле подключенного модуля.

Модуль – последовательность инструкций, сохраненных в файле для многократного применения.

Программа (Program) – последовательность инструкций, сохраненных в файле. Инструкции выполняются сверху вниз, т.е. сначала выполняется инструкция на самой первой строке программы, далее происходит переход на следующую инструкцию и т.д.

Инструкция (Statement) – совокупность операторов и операндов (литералов), используемая для выполнения определенного действия.

Выражение (Expression) – совокупность операторов и операндов (литералов), используемая для расчета, получения некоторого одного значения, которое может быть присвоено объекту.

Выражение – серия символов, в результате которой может быть рассчитано, получено конкретное значение [18].

Пример: выражение 10 + 5, в результате которого будет получено значение 15.

Любое выражение являться инструкцией (expression statement).

Однако нельзя сказать, что любая инструкция является выражением: та инструкция, в результате выполнения которой не происходит получения значения (т.е. нельзя создать объект в результате ее выполнения) не будет являться выражением.

Примеры выражений:

```
>>> 2 + 2
>>> 10
>>> 'My name is ' + name
>>> 'Result: ' + int(input_value) * 2.58
```

В результате выполнения подобных инструкций будет получено определенное значение.

Примеры инструкций (которые содержат выражения):

```
>>> v1 = 2 + 4
>>> if value > 10:
>>> while True:
```

В примере выше две последние инструкции называются if statement, while statement соответственно.

Примеры инструкций (которые не содержат выражения):

```
>>> pass
>>> break
```

Такие две инструкции называются pass statement и break statement соответственно.

Литерал – часть программного кода, представление, которое всегда обозначает одно и то же значение (константное значение). Имена переменных и функций не являются литералами [18].

Литерал – нотация (обозначение, запись) константных значений некоторых встроенных типов [20].

Например, в приведенной ниже инструкции три литерала: литерал `' is '` представляет текст « is » типа `str`, литерал `35` представляет число 35 типа `int`, литерал `' years old'` представляет соответствующий текст « years old»; имена переменных `v2`, `name` не будут являться литералами:

```
>>> v2 = name + ' is ' + str(35) + ' years old'
```

2. Типы объектов

Объекты в языке Python разделяются на типы. Все типы объектов можно разделить на 2 основные категории:

- **встроенные типы объектов**;
- **собственные новые типы объектов** (созданные программистом при написании программы).

Встроенные типы объектов описывают основные структуры данных, которых будет достаточно при написании небольших программ. При разработке более сложных программ может потребоваться описать особый тип объектов или особую структуру данных, для которой в Python не имеется

встроенного типа объектов; в таком случае прибегают к созданию **собственных типов объектов** на основе **классов** (см. практическое задание № 5).

Встроенные типы объектов можно разделить на группы: числа, последовательности, отображения, классы, экземпляры, исключения и др. В таблице 1.1 представлены типы объектов указанных групп.

Таблица 1.1 – Основные встроенные типы объектов в Python 3.13

Группа типов объектов	Тип объекта		Примеры
	Англоязычное название	Русскоязычное название	
numerics (числа)	`int`	целое число	`100, 587, 104103`
	`float`	вещественное число	`0.1, 100.07, 587.0`
	`complex`	комплексное число	`-4.5j, -1.23+4.5j, 1.23+0j`
sequences (последовательности)	`list`	список	`[1, 2, 3, 4], [10.05, 14, 'text']`
	`tuple`	кортеж	`(1, 2, 3, 4), (10.05, 14, 'text')`
	`range`	диапазон	`range(1, 10), range(2, 10, 2)`
	`str`	строка	`'text', 'hello, world!'`
set types (типы множеств)	`set`	множество	`{1, 2, 3, 4}, {10.05, 14, 'text'}`
	`frozenset`	зафиксированное множество	`{1, 2, 3, 4}, {10.05, 14, 'text'}`
mappings (отображения)	`dict`	словарь	`{'key0': 100, ' key1': 200}, {'Name': 'Tom', 'Age': 30}`
classes (классы)	`class / type`	класс / тип	`class Person():` `…`
instances (экземпляры)	`instance / <название_класса>`	экземпляр	`tom = Person()` `worker1 = Person()`
Прочие типы	`bool`	булевское значение	`True, False`
	`exception`	исключение	`class MyException(Exception):` `…`

Источник: составлено на основе [14, 5].

Проверить тип объекта можно с помощью встроенной функции `type()` (листинг 1.2).

Листинг 1.2 – Примеры использования функции `type()` для проверки типов объектов листинга 1.1

```
In [28]: type(print)
Out[28]: builtin_function_or_method

In [29]: type(name)
Out[29]: str

In [30]: type(a)
Out[30]: int

In [31]: type(b)
Out[31]: int

In [32]: type(print)
Out[32]: builtin_function_or_method

In [33]: type(type)
Out[33]: type
```

2.1. Динамическая типизация

Язык Python является **динамически типизируемым** языком, в отличие, например, от C, C++, Java – они являются **статически типизируемыми**.

Динамическая типизация означает, что типы определяются автоматически во время выполнения кода [5], т.е. отсутствует необходимость явно указывать тип объекта.

Например, в языке C++ при создании объектов перед именем переменной, при первом присваивании переменной значения (или при изменении типа), требуется указать тип создаваемого объекта (`int`, `float`, `std::string` и др.) (таблица 1.2).

При этом тип переменной в C++ не может быть изменен, тогда как в Python переменной может быть поставлено в соответствие, например, сначала целое число (`a = 11`), а потом некоторый текст (`a = "now here is the text"`) (таблица 1.2), т.е. можно сказать, что тип автоматически изменился с `int` на `str`, но в сущности переменная `a` просто перестала ссылаться на объект с типом `int` и стала ссылаться на иной объект с типом `str`.

Таблица 1.2 – Пример создания объектов на языке Python и на языке C++

Python	C++

`a = 10`
`b = 10.5`
`c = "text"`
`a = 11`
`a = "now here is the text"`
	`int a = 10;`
	`float b = 10.5;`
	`std::string c = "text";`
	`a = 11;`

2.2. Преобразование типов

При выполнении операций над объектами, имеющими разные типы, необходимо, чтобы все объекты были преобразованы к одинаковому типу.

Существует 2 вида преобразования типов:

- **Неявное преобразование типов.**
- **Явное преобразование типов.**

Неявное преобразование типов Python выполняет автоматически с некоторыми типами объектов, например числовыми (листинг 1.3).

Листинг 1.3 – Пример неявного преобразования типов

```
1 a = 10
2 b = 1.5
3 c = a + b
4 print(c)
```

```
11.5
```

Однако иногда такое автоматическое преобразование невозможно, в результате чего возникает ошибка (листинг 1.4).

Листинг 1.4 – Пример ошибки при неявном преобразовании типов

```
1 a = '10'
2 b = 1.5
3 c = a + b
4 print(c)
```

```
TypeError: can only concatenate str (not "float") to str
```

Явное преобразование типов подразумевает использование специальных встроенных функций, которые преобразуют исходный тип к необходимому типу

Листинг 1.5 – Пример явного преобразования типов

```
1 a = float('10')
2 b = 1.5
3 c = a + b
4 print(c)
```

```
11.5
```

Функции, использующиеся для преобразования типов, носят имена типов, на которые они преобразовывают исходный тип. Например, для преобразования к типу `float` используется одноименная функция `float()`. Примеры других функций: `int()`, `str()`, `list()` и т.д.

2.3. Структуры данных

Объекты в Python могут быть объединены в более сложный объект – коллекцию или последовательность объектов, которая имеет определенную структуру.

Структура данных (Data Structure) – способ (формат) хранения данных.

Структура данных в Python – способ (формат) хранения набора связанных объектов.

Для того, чтобы объекты в Python можно было объединить в рамках определенной структуры (например, массива), существуют специальные типы данных, которые представляют собой структуры данных. К основным таким типам относятся:

- `list` – список (аналог массива в Python),
- `tuple` – кортеж,
- `dict` – словарь,
- `set` – множество.

Такие типы данных еще называют **коллекциями**.

Коллекция (Collection) – набор объектов.

При этом в Python **строка** (`str`) представляется как коллекция, в которой каждый символ является элементом коллекции.

Таким образом, к коллекциям относятся следующие типы:

- `str` – строка,
- `list` – список,
- `tuple` – кортеж,
- `dict` – словарь,
- `set` – множество.

Частным случаем коллекции является **последовательность**.

Последовательность (Sequence) – позиционно упорядоченная коллекция объектов [5].

К последовательностям относятся:

- `str` – строка,
- `list` – список,
- `tuple` – кортеж.

Взаимосвязь понятий «структура данных», «коллекция» и «последовательность» представлена с помощью кругов Эйлера на рисунке 1.1.

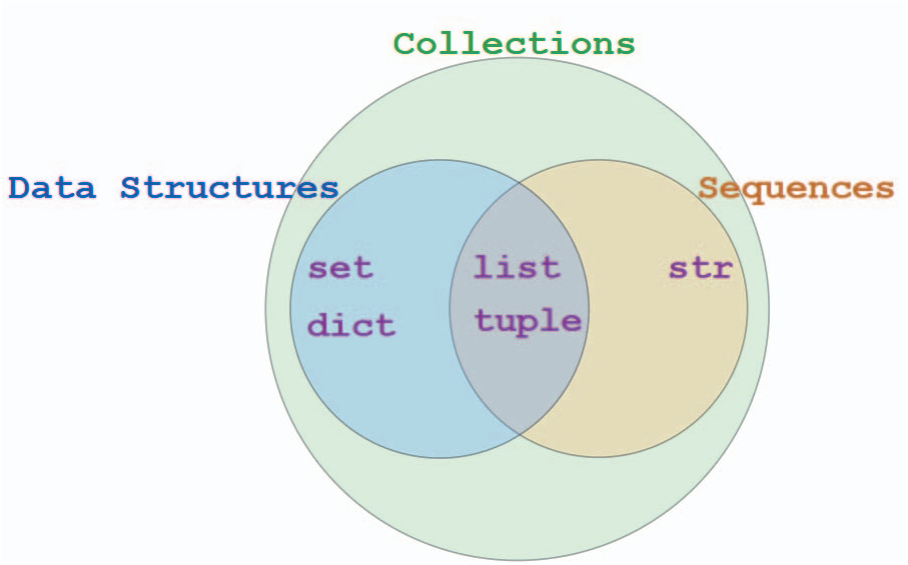


Рисунок 1.1 – Основные структуры данных, коллекции и последовательности в Python

3. Переменные

Переменная (также называемая в Python именем) – указатель на область памяти объекта [5]. Используется для доступа к объекту, ссылку на которую она (переменная) хранит.

Ссылка – связь от переменной к объекту.

Пример переменных из листинга 1.5: `a`, `b`, `c`.

3.1. Создание переменной

Переменная создается при первом **присваивании** ей значения. Для создания переменной необходимо указать имя переменной и присвоить ей определенной значение с помощью специального оператора присваивания `=` (знака равенства). В общем виде команда для создания переменной выглядит следующим образом:

```
имя_переменной = значение
```

Пример присваивания переменной `a` значения 10:

```
>>> a = 10
```

При этом язык Python поддерживает множественной присваивание.

Множественной присваивание подразумевает создание сразу нескольких переменных с присваиванием каждой из них определенного значения, но с использованием лишь одного оператора присваивания `=`. При таком присваивании переменные и их значения разделяются запятыми, а для их сопоставления учитывается порядок следования.

Пример присваивания переменным `a, b, c` значений 10, 15, 20 соответственно:

```
>>> a, b, c = 10, 15, 20
```

Подобное множественное присваивание эквивалентно трем простым присваиванием:

```
>>> a = 10
>>> b = 15
>>> c = 20
```

Переприсваивание значения переменной – случай, когда уже созданной переменной присваивается другое значение. В таком случае переменная будет ссылаться уже на другой объект.

Пример переприсваивания переменной `a` нового значения 15 (так как выполнение кода происходит сверху вниз, то сначала переменной `a` присваивается значение 10, а потом значение 15; в результате после выполнения двух команд переменная `a` останется со значением 15):

```
>>> a = 10
>>> a = 15
```

Пример: в таблице 1.2 на строке 1 присваивается значение 10, а на строке 4 переприсваивается новое значение – 11. Таким образом, ссылка на объект 10 уничтожается, так как переменная `a` после выполнения программы будет хранить ссылку на объект со значением 11.

Изменение ссылки или адреса объекта, на который ссылается переменная можно увидеть с помощью функции `id()` (листинг 1.6).

Листинг 1.6 – Примеры использования функции `id()` для получения адреса двух объектов через одну переменную `a`

```
In [44]: a = 10

In [45]: id(a)
Out[45]: 140718582764616

In [46]: a = 11

In [47]: id(a)
Out[47]: 140718582764648
```

В Python применяется **кэширование** для оптимизации работы с часто используемыми значениями, такими как небольшие целые числа. Например, для значения 15 не выделяется нового участка памяти (не создается новый объект) в отличие от более редких значений, таких как 15000 (листинг 1.7).

Листинг 1.7 – Пример получения адреса для различных объектов: для значения 15 (часто используемый объект) адрес всегда остается прежним, для значения 15000 происходит выделение нового участка памяти всякий раз, когда объект с таким значением создается.

```
In [49]: a = 15

In [50]: id(a)
Out[50]: 140716528122744

In [51]: a = 15

In [52]: id(a)
Out[52]: 140716528122744

In [53]: a = 15000

In [54]: id(a)
Out[54]: 2305035377136

In [55]: a = 15000

In [56]: id(a)
Out[56]: 2305035377264
```

3.2. Имена переменных

При выборе имени переменной необходимо учитывать следующее:

- Имя может состоять из букв, цифр, символов подчеркивания.
  Примеры: `name, Customer1, calculated_costs, x3`.
- Имя не должно начинаться с цифры.
- Имя переменной рекомендуется задавать отличное от ключевых слов (зарезервированных имен) языка Python, таких как имена встроенных функций (`print, type` и др.), названия типов (`int, str` и др.), инструкций (`in, if, for, while` и др.) и т.д. Обычно такие ключевые слова выделяются особым цветом шрифта в среде разработки, черный цвет шрифта означает, что имя не является зарезервированным.
- Имя чувствительно к регистру букв.
  Например, следующие переменные будут разными: `var1, Var1`.

4. Операции

Выражения в Python состоят из операндов и операторов.

Операнд – элемент, над которым выполняется операция (математическая операция) [18].

Пример: в выражении «10 + 5» операндами являются числа «10» и «5», над которыми выполняется операция «сложение» через соответствующий оператор (символ операции) «+».

4.1. Приоритет операций

В результате реализации выражения может быть выполнено несколько операций.

Очередность выполнения операций (старшинство) важна в случае, если выражение является смешанным, т.е. состоит из нескольких операций.

Пример простого выражения: `10 + 5`.

Пример смешанного выражения: `10 + 5 ** 3`.

В таблице 1.3 представлены основные операторы языка Python, в столбце 1 указана очередность выполнения операций, обратная их приоритету (т.е. операция с очередностью выполнения 1 имеет наивысший приоритет).

Таким образом, для расчета смешанного выражения последовательно выполняются все операции этого смешанного выражения в соответствии с их приоритетом (очередностью выполнения).

Например, в выражении `10 + 5 ** 3` *имеется 2 оператора: оператор сложения* `+` *и оператор возведения в степень* `**`. *В соответствии с таблицей 1.3 очередность выполнения* `**` *ниже, чем у* `+` *(приоритет у* `**` *выше, чем у* `+`*), поэтому сначала будет выполнена операция возведения в степень с помощью оператора* `**` *над операндами* `5, 3` *(очередность выполнения – 8), затем будет выполнена операция сложения с помощью оператора* `+` *(очередность выполнения – 15) над операндом* `10` *и результатом выражения* `5 ** 3`. *Т.е. результат будет рассчитан следующим образом:* `10 + 5 ** 3 = 10 + 125 = 135`.

Использование круглых скобок в смешанных выражениях позволяет изменить очередность выполнения операций: операция, заключенная в круглые скобки, выполняется в первую очередь.

Например, для изменения очередности выполнения операций в выражении `10 + 5 ** 3` *так, чтобы сначала было выполнено сложение, а затем возведение в степень, необходимо добавить круглые скобки:* `(10 + 5) ** 3`. *В таком случае результат будет рассчитан следующим образом:* `(10 + 5) ** 3 = 15 ** 3 = 3375`.

Таблица 1.3 – Основные операторы в Python и их очередность выполнения (ОВ) в смешанных выражениях

ОВ	Оператор	Операция	Описание	Пример
1	`{ }`	`{…}`	Словарь, множество, включения множеств и словарей	`{10, 13, 1} & {10}`
2	`[ ]`	`[…]`	Список, списковое включение	`[0] * 8`
3	`( )`	`(…)`	Кортеж, выражение, выражение генератора	`(0) * 8`
4	`.`	`x.атрибут`	Ссылка на атрибут	`a = 8` `a.__class__`

5	`…( )`	`x()`	Вызов (функции, метода, класса и др.)	`clear(),`
				`print('hello'),`
				`a = 8`
				`a.bit_length()`

ОВ	Оператор	Операция	Описание	Пример
6	…[: :]	x[i : j : k]	Нарезание	a = [10, 20, 30, 40, 50, 60 ,70] every_second_element = a[1 : 6 : 2]
7	…[ ]	x[i]	Индексация	a = [10, 20, 30] first_elemt = a[0]
8	**	x ** y	Возведение в степень	10 ** 5
9	~…	~x	Побитовое «НЕ» (инверсия)	~10
10	-…	-x	Противоположность	a = -10 b = -a
10	+…	+x	Идентичность	a = -10 b = +a
11	/	x / y	Деление обычное	10 // 5
11	//	x // y	Деление с округлением в меньшую сторону	10 // 9
12	%	x % y	Остаток от деления, формат	10 % 3
13	*	x * y	Умножение, повторение	10 * 5, 'repeat ' * 2
14	-	x - y	Вычитание, разность множеств	10 - 5
15	+	x + y	Сложение, конкатенация	10 + 5, 'My name is' + 'Tom'
16	<<	x << y	Сдвиг x влево на y битов	2 << 2
16	>>	x >> y	Сдвиг x право на y битов	2 >> 2
17	&	x & y	Побитовое «И», пересечение множеств	1 & 1
18	^	x ^ y	Побитовое «исключающее ИЛИ», симметричная разность множеств	11 ^ 10
19	|	x | y	Побитовое «ИЛИ», объединение множеств	1 | 0
20	==	x == y	Равенство значений	a = 10 b = 10 a == b
20	!=	x != y	Неравенство значений	10 != 15
21	<	x < y	Сравнение по абсолютной величине, проверка на подмножество и надмножество	10 < 15
21	>	x > y		'text1' > 'text'
21	<=	x <= y		10 <= 10
21	>=	x >= y		'text' >= 'text'
22	is (is not)	x is y, x is not y	Проверка идентичности объектов	a = 10 b = 10 a is b
23	in (not in)	x in y, x not in y	Членство для итерируемых объектов	10 in [1, 2, 10, 100], 50 not in [1, 2, 10, 100]
24	not	not x	Логическое «НЕ»	not (flower_type == 'virginica' and petal_width > 2.0)
25	and	x and y	Логическое «ИЛИ» (y оценивается только если x истинно)	flower_type == 'virginica' and petal_width > 2.0

ОВ	Оператор	Операция	Описание	Пример
26	`or`	`x or y`	Логическое «ИЛИ» (y оценивается только если x ложно)	`flower_type == 'virginica' or flower_type == 'setosa'`
27	`… if … else …`	`x if y else z`	Тернарный выбор (x оценивается только если y истинно)	`role = 'expert' if experience > 10 else role = 'specialist'`
28	`lambda`	`lambda аргументы: выражение`	Создание анонимной функции	`lambda x: x + 1`

29	`yield`	`yield x`	Протокол send функции генератора	`def get_generator(): for el in [10, 11, 12, 13, 14]: if el > 11: yield el`
				
				`gen1= get_generator()`
				`for el in gen1: print(el)`
				`for el in gen1: print(el)`

Источник: составлено на основе [5, 23].

4.2. Операции присваивания

Присваивание значения объекту осуществляется с помощью оператора присваивания `=`. При этом могут быть использованы расширенные операторы присваивания, реализующие дополнительную операцию, например сложение перед операцией присваивания (листинг 1.8).

Листинг 1.8 – Пример получения эквивалентных результатов при применении обычного оператора присваивания `=` вместе с оператором сложения `+` для увеличения исходного значения (строка 2) и расширенного оператора присваивания `+=`

```
1 a = 10
2 a = a + 10
3
4 b = 10
5 b += 10
6
7 print(a)
8 print(b)
```

```
20
20
```

Список основных расширенных операторов присваивания (augmented assignment aperators) представлен в таблице 1.4.

Таблица 1.4 – Операторы присваивания в Python: обычный оператор присваивания и расширенные операторы присваивания

Оператор	Операция	Описание	Пример
`=`	`x = y`	Присваивание	`a = 10`
`+=`	`x += y`	Сложение и присваивание	`a = 10` `a += 1`
`-=`	`x -= y`	Вычитание и присваивание	`a = 10` `a -= 10`
`*=`	`x *= y`	Умножение и присваивание	`a = 10` `a *= 2`
`/=`	`x /= y`	Деление нацело и присваивание	`a = 10` `a /= 5`
`//=`	`x //= y`	Деление нацело с округлением в большую сторону и присваивание	`a = 10` `a //= 9`
`%=`	`x %= y`	Вычисление остатка от деления и присваивание	`a = 10` `a %= 2`
`**=`	`x **= y`	Возведение в степень и присваивание	`a = 10` `a **= 2`

Источник: составлено на основе [23].

5. Встроенные функции

Одним из часто используемых элементов языка Python является **функция**.

Функция (Function) – элемент языка программирования для хранения инструкции (совокупности инструкций) с целью выполнения этой инструкции всякий раз, когда функция вызывается.

Функция – подпрограмма (часть программы), которая действует как математическая функция: при входном наборе значений аргументов возвращает уникальный результат [18].

Функция может:

- **принимать входные данные** в качестве аргументов, как например функция нахождения максимума `max(10, 7)`, принимающая в данном случае на вход числа 10 и 7;
- **не принимать входные данные**, как например консольная функция `clear()`;
- **возвращать выходные данные** (результаты выполнения инструкций), как например та же функция `max(10, 7)`, тогда для сохранения выходных значений (10 в данном случае) можно использовать дополнительную переменную: `res = max(10, 7)`;
- **не возвращать ничего**, как, например, функция `print()`.

Чтобы вызвать функцию, необходимо указать её имя и круглые скобки. В круглых скобках указываются входные данные для функции. Если входные данные не передаются, то скобки оставляют пустыми.

Встроенные функции Python выполняют полезные инструкции, облегчая процесс написания кода. Например, функция `print()` выводит в консоль сообщение, которое ей передается на вход в круглые скобки в качестве аргумента: `print('message')`. С помощью этой функции можно проверять значения объектов, отображать результаты выполнения кода, реализовать командный диалог с пользователем программы и т.д.

В таблице 1.5 представлены основные встроенные в Python функции.

Таблица 1.5 – Основные встроенные функции (built-in functions) в языке Python 3.13

Функция	Описание	Пример
`abs()`	Возвращает абсолютное значение числа (модуль числа).	`abs(-10)` `abs(0.5)`

`bin()`	Преобразует целое число в двоичное представление (объект типа str) (binary string) с префиксом '0b'.	`bin(1)`
		`bin(2)`
		`bin(10)`

`chr()`	Возвращает символ в кодировке Unicode на основе переданного на вход кода в виде значения типа `int`. Действует обратным образом по отношению к функции `ord()`.	`chr(8364)` `chr(65)`

`dir()`	Если вызывается без аргументов, то возвращает список используемых в программе имен (имен переменных, функций и др.).
	Если вызывается с аргументом (на вход принимается объект), то возвращает список атрибутов объекта.
		`dir()`
		`dir(v1)`
		`dir('text')`
		`dir(7)`

`divmod()`	Принимает два числа (некомлексных) a, b и возвращает пару чисел: целое при делении a на b, остаток при делении a на b.	`divmod(10, 5)` `divmod(10, 4)`
`eval()`	Принимает на вход выражение (например, в виде теста str) и возвращает результат выполнения такого выражения.	`eval('1 + 4 * 2')` `eval('abs(10 - 100)')`

`help()`	Если вызывается без аргумента, то запускает командный диалог через консоль для поиска справочной информации.
	Если вызывается с аргументом (объектом), то отображает справочную информацию об объекте в консоль.
		`help()`
		`help(int)`
		`help(7)`

`hex()`	Преобразует целое число в шестнадцатеричное представление (объект типа str) (hexadecimal string) с префиксом '0x'.	`hex(1)`
		`hex(9)`
		`hex(10)`
		`hex(15)`
		`hex(16)`

`id()`	Возвращает уникальный идентификатор передаваемого на вход	`id(v1)`
		`id(130)`
		`id('text')`

Функция	Описание	Пример
	объекта (адрес объекта в памяти компьютера).	`id(help)` `id(id)`

`input()`	Если вызывается без аргумента, то переводит консоль в режим ввода текста (после ввода текста необходимо нажать Enter) и возвращает введенный в консоль текст в виде объекта с типом str.
	Если вызывается с аргументом (текстом –объектом типа str), то в консоль предварительно выводится данный текст, затем консоль переводится в режим вводе текста.
		`input()`
		`input('Введите ваше имя:')`
		`v1 = input('Enter value')`
		`age = input(How old are you?')`

`isinstance()`	Проверяет, является ли объект экземпляром определенного класса (имеет ли объект определенный тип). Принимает два аргумента: object (проверяемый объект), classinfo (проверяемый класс, тип). Возвращает True или False.	`isinstance(7, int)`
		`isinstance('text', str)`
		`isinstance('5.01', float)`
		`isinstance(name, str)`

`len()`	Возвращает длину объекта (объект должен быть последовательностью или коллекцией).	`len('text')` `len(name)`
`max()`	Возвращает наибольший элемент из переданных на вход элементов или итерируемого объекта.	`max(1, 2, 3, 10)` `max('abca')`
`min()`	Возвращает наименьший элемент из переданных на вход элементов или итерируемого объекта.	`min(1, 2, 3, 10)` `min('abca')`

`oct()`	Преобразует целое число в восьмеричное представление (объект типа str) (octal string) с префиксом '0o'.	`oct(1)`
		`oct(7)`
		`oct(8)`
		`oct(9)`

`open()`	Открывает файл и возвращает соответствующий объект (file object)	`open('D:\info.txt')` `file = open(path)`
`ord()`	Возвращает код в виде значения типа `int` переданного на вход символа в кодировки Unicode. Действует обратным образом по отношению к функции `chr()`.	`ord('€')` `ord('A')`
`pow()`	Принимает на вход два аргумента: `base` (число-основание), `exp` (степень). Возвращает результат возведения `base` в степень `exp`.	`pow(10, 2)` `pow(4, 1/2)`

`print()`	Выводит передаваемые на вход объекты (значения объектов) в консоль в виде текста.	`print('Hello, World!')`
		`print(1, 2, 3)`
		`print(1, 'and', 2)`
		`print('my', name)`

`round()`	Принимает два аргумента: `number` (число), `ndigits` (число знаков после запятой). Возвращает число `number`,	`round(7.4)`
		`round(7.5)`
		`round(7.49)`

Функция	Описание	Пример

	округленное до `ndigits` знаков после запятой).
	Если `ndigits` не указывается, то возвращается число, округленное до целых.
		`round(8.67, 1)`
		`round(8.67, 1)`
		`round(10.000577, 4)`

`type()`	Возвращает тип переданного на вход объекта.	`type(7)`
		`type('4')`
		`type(10 + 0.8)`

Источник: составлено на основе [13].

Задачи

Задача № 1. Объекты. Типы объектов. Переменные. Арифметические и логические операции

1. Создайте объект с именем ***a*** и значением 10.
2. Создайте объект ***b*** со значением 15.
3. Выведете в консоль значения объектов ***a***, ***b***.
4. Определите тип объектов ***a***, ***b***.
5. Измените переменную ***a*** так, чтобы она ссылалась на объект со значением 8 (объект ***a*** должен быть со значением 8).
6. Увеличьте значение объекта ***a*** на 3.
7. Увеличьте значение объекта ***a*** в 5 раз.
8. Объект ***c*** определите как сумму ***a*** и ***b***.
9. Определите истинность высказывания: ***c*** равно ***a***.
10. Определите истинность высказывания: ***c*** не равно 0.
11. Определите, является ли ***c*** больше, чем ***a***.
12. Определите, истинно ли то, что ***b*** меньше или равно ***a***.
13. Определите истинность высказывания: объекты ***a***, ***b***, ***c*** больше 10.
14. Определите истинность высказывания: хотя бы один из объектов ***a***, ***b***, ***c*** больше 10.
15. Объект ***h*** определите как a^b.
16. Объект ***t*** определите как следующий текст «Python language».
17. Выведете в консоль значения объектов ***a***, ***b***, ***c***, ***h***, ***t***.
18. Определите тип объектов ***a***, ***b***, ***c***, ***h***, ***t***.
19. Измените тип объекта ***a*** на `float` путем явного преобразования типов.
20. Измените значение и тип объекта ***a*** на `float` путем неявного преобразования типов.
21. Объект ***b*** уменьшите на 3,5.
22. Выведете в консоль значения объектов ***a***, ***b***, ***c***, ***h***, ***t***.
23. Определите модуль ***b***.
24. Найдите целую часть от деления ***b*** на ***a***.
25. Найдите остаток от деления ***b*** на ***c*** и округлите до целых.

Задача № 2. Встроенные функции

1. Создайте объект ***var_in***, значение которого необходимо ввести в консоль.
2. Определите максимальное значение из объектов ***a***, ***b***, ***c***, ***h***, ***var_in***.
3. Определите минимальное значение из объектов ***a***, ***b***, ***c***, ***h***, ***var_in***.
4. Преобразуйте переменные в двоичное представление.
5. Преобразуйте переменные в восьмеричное представление.
6. Преобразуйте переменные в шестнадцатеричное представление.
7. Создайте объект ***r*** со значением 18,947301. Округлите значение до тысячных.
8. Создайте объект ***r_in***, значение которого формируется следующим образом. В консоль необходимо ввести два числа:
   - ***a1*** – исходное значение в виде вещественного числа
   - ***a2*** – значение в виде целого числа, отражающее количество знаков после запятой.

   В переменную ***r_in*** сохранятся результат округления числа ***a1*** до ***a2*** знаков после запятой.

   Пример:

   In: Введите два числа: 5,40917 2

   Out: r_in = 5,41

Задача № 3. Очередность выполнения операций

Условие: имеется некоторый алгоритм – последовательность простых (состоящих из одного оператора и двух операндов) математических выражений для расчета определенного числового значения. При этом на первом шаге алгоритма оба операнда являются исходными числами, на втором шаге – первый операнд является результатом выражения предыдущего (первого) шага и т.д.

Требуется: составить смешанное выражение на языке Python (состоящее из нескольких подвыражений) для расчета итогового числового значения с учетом последовательности выполнимых действий (математических операций). Ответ предоставьте в двух вариантах:

a. используйте как можно больше круглых скобок;
b. используйте как можно меньше круглых скобок.

Пример:

Имеется следующий алгоритм из пяти шагов:

1) 7 + 3
2) * 10
3) - 4
4) -90
5) ** 2

Описание принципа работы алгоритма:

На шаге 1 выражение примет вид 7 + 3, результат выражения = 10.

На шаге 2 выражение примет вид 10 * 10, результат = 100.

На шаге 3 выражение примет вид 100 - 4, результат = 96.
На шаге 4 выражение примет вид 96 - 90, результат = 6.
На шаге 5 выражение примет вид 6 ** 2, результат = 36.
Выражения на языке Python для правильного расчета результата:
a. ((((7 + 3) * 10) - 4) - 90) ** 2
b. (((7 + 3) * 10 - 4) - 90) ** 2

Решите задачу № 3 для следующих алгоритмов:

Алгоритм 1	**Алгоритм 2**	**Алгоритм 3**
1) 4 + 3 2) - 11 3) * 2	1) 2 * 3 2) + 10 3) * 2 4) ** 2	1) 1 / 10 2) + 4.1 3) - 8 4) - 2 5) ** 2 6) * 10

Задача № 4

1. Разработайте программный код, с помощью которого можно будет вводить номер товара в виде какого-то числа *a* и получать в консоли ответ в виде текстового сообщения «Товар номер *a* поступил на склад».
   Пример:
   In: Введите номер товара: 7
   Out: Товар номер 7 поступил на склад
2. Для п. 1 добавьте возможность вводить помимо номера товара вес этого товара в граммах.
   Пример:
   In: Введите номер товара и его вес в граммах (через пробел): 7 35.11
   Out: Товар номер 7 весом 35.11 г поступил на склад
3. Для п. 2 добавьте расчет веса в кг.
   Пример:
   In: Введите номер товара и его вес в граммах (через пробел): 7 35.11
   Out: Товар номер 7 весом 35.11 г (0.03511 кг) поступил на склад
4. Для п. 3 добавьте возможность вводить данные по двум товарам. В консоль отобразите информацию об итоговом количестве товаров, общем весе всех товаров.
   Пример:
   Out: Введите информацию по 2 товарам:
   In: Введите номер товара и его вес в граммах (через пробел): 7 35.11
   Out: Товар номер 7 весом 35.11 г (0.03511 кг) поступил на склад
   In: Введите номер товара и его вес в граммах (через пробел): 19 1307.84
   Out: Товар номер 19 весом 1307.84 г (1.30784 кг) поступил на склад

Out: На складе 2 товара (с номерами 7, 19) с общим весом 1342.95 г (1.34295 кг)

Таблица 1.6 – Представление 1 м^2 в других единицах измерения

Исходная единица измерения	Преобразованная единица измерения
1 м^2	10 000 см^2
1 м^2	0,01 сотки
1 м^2	0,0001 гектара (га)
1 м^2	0,000001 (1e-6 = $1 \cdot 10^{-6} = 1 \div 10^6$) км^2
1 м^2	1 550 дюймов^2
1 м^2	10,76 футов^2
1 м^2	0,000237 акров
1 м^2	3,86e-7 миль^2

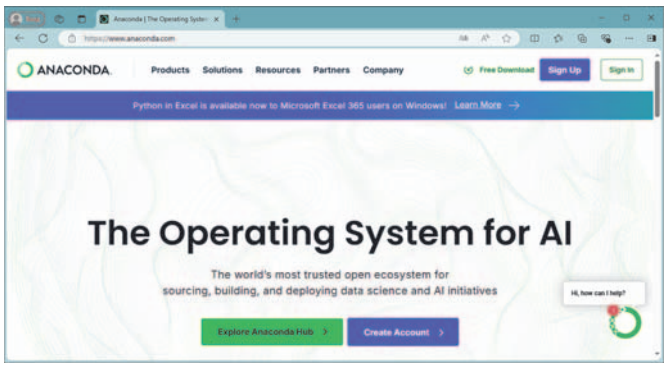
Задача № 5

1. Фермер владеет прямоугольным полем с длинами сторон 34,5 м^2, 11 м^2. Определите площадь и периметр такого поля.
2. Для п.1 определите площадь и периметр поля в м^2, сотках, гектарах и акрах.
3. Разработайте программный код, с помощью которого можно будет вводить длины сторон прямоугольного поля и получать в результате рассчитанные площадь и периметр поля.
4. Разработайте конвертер площади, переводящий м^2 в см^2, сотки, гектары, км^2, дюймы^2, футы^2, акры, мили^2.

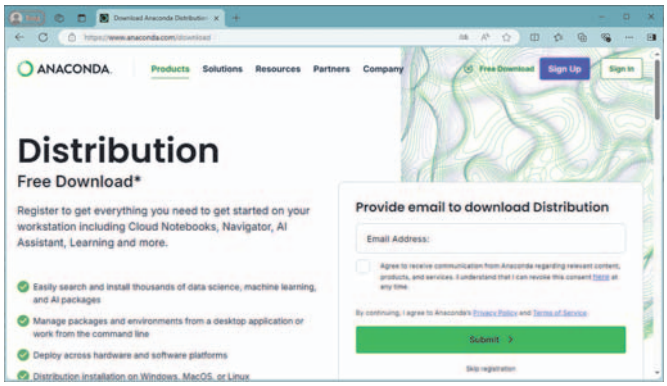
Методические указания по работе в среде разработки Spyder

Для выполнения задач необходимо скачать и установить на компьютер язык программирования Python и среду разработки, например Spyder. Именно данная среда разработки использовалась при написании и выполнении учебного программного кода, представленного в листингах и на рисунках пособия.

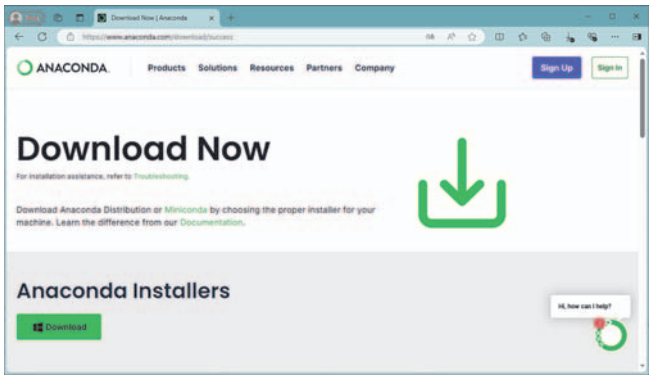
Установить Python, Spyder, а также другие дополнительные инструменты для языка можно с помощью свободно распространяющегося дистрибутива Anaconda, доступного для скачивания на официальном сайте: https://www.anaconda.com/. Нажмем на копку Free Download:



Далее нажмем на копку Skip registration:

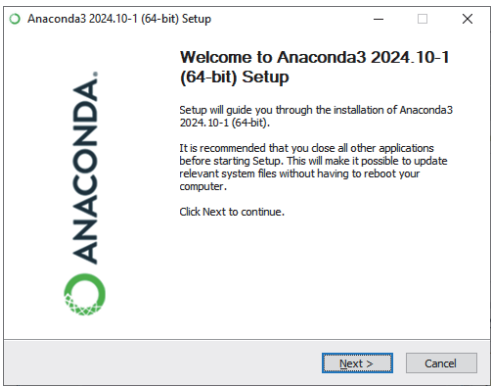


Нажмем на копку Download:

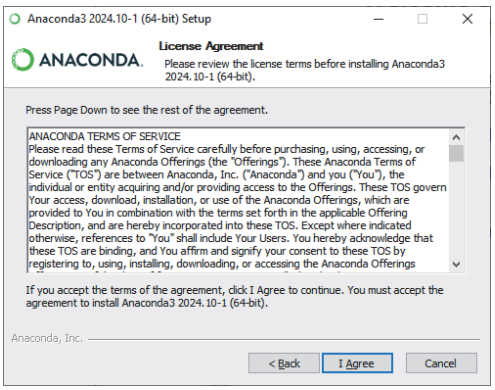


Установка является стандартной и состоит из нескольких шагов, переход на которые осуществляется через кнопку Next >.

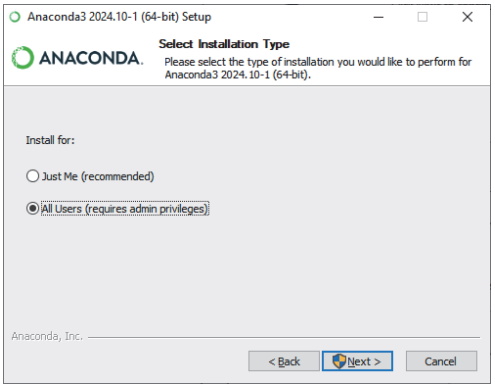
Шаг 1 Шаг 2

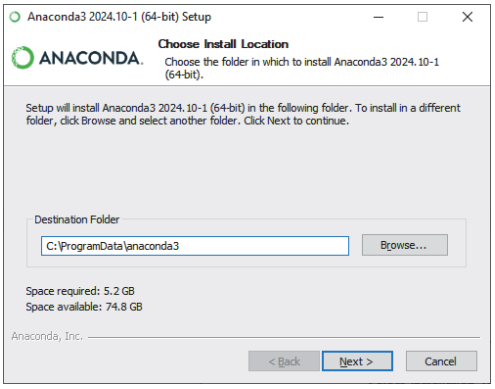


Шаг 3



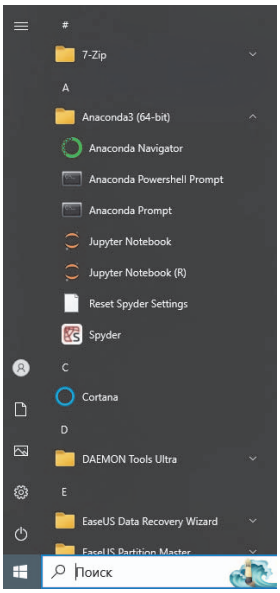
Шаг 4





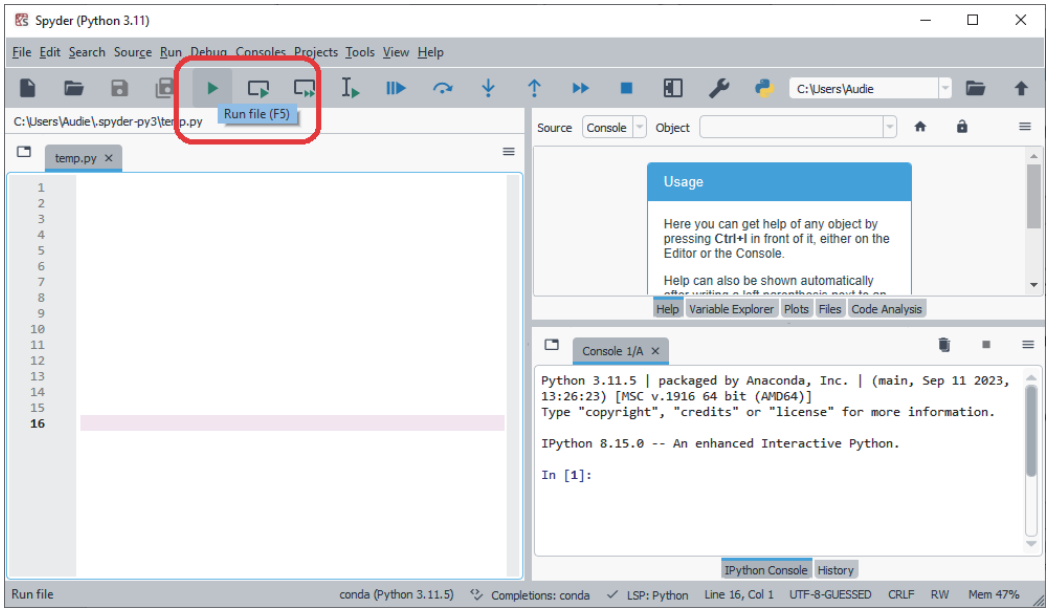
Можно оставить все параметры установки так, как они выбраны по умолчанию. В случае наличия прав администратора (домашний компьютер) удобнее будет установить дистрибутив для всех пользователей (шаг 3):

После успешной установки в меню Пуск системы Windows появится папка Anaconda3, в которой расположены ярлыки установленных программ, в том числе ярлыки среды разработки Spyder и командной строки Anaconda Prompt, с помощью которой можно устанавливать библиотеки для языка Python.



В среде разработки Spyder создадим новый файл (скрипт), нажав File – New File. Созданный файл сохраним, нажав File – Save As.

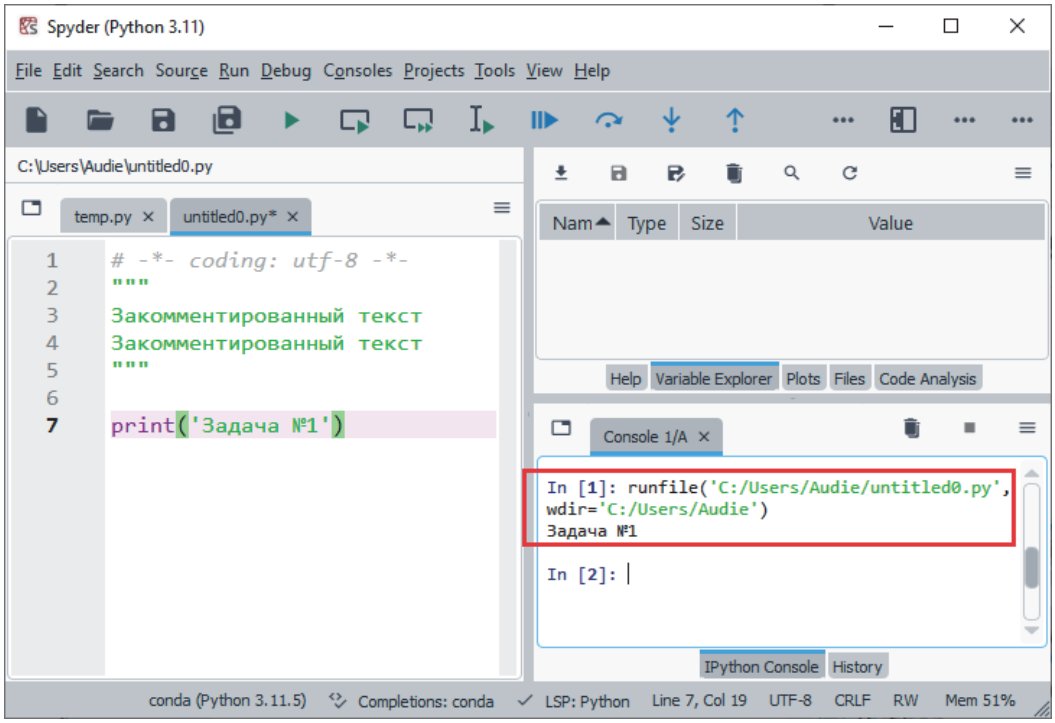
В файл мы можем вписывать команды. Чтобы они выполнились, необходимо запустить скрипт, нажав на кнопку «Run file» или клавишу F5:



Например, для вывода в консоль текста «Задача №1» можно воспользоваться встроенной функцией `print()` и вписать следующую команду:

```
print('Задача №1')
```

Текст отобразится в правой части среды разработки:



Для лучшего понимания написанной программы рекомендуется добавлять комментарии.

На последнем скриншоте в левой части были закомментированы строки 2-5, а также строка 1 разными способами. Комментирование текста бывает полезно для разработчиков и не влияет на выполнение программы, так как закомментированный текст не выполняется, даже если содержит какие-либо команды.

Комментирование текста осуществляется двумя способами.

Способ 1. С помощью символа «решетка» `#`. Весь текст по строке после символа «решетка» является комментариями.

Пример:

```
# текст по строке после символа "решетка" закомментирован
```

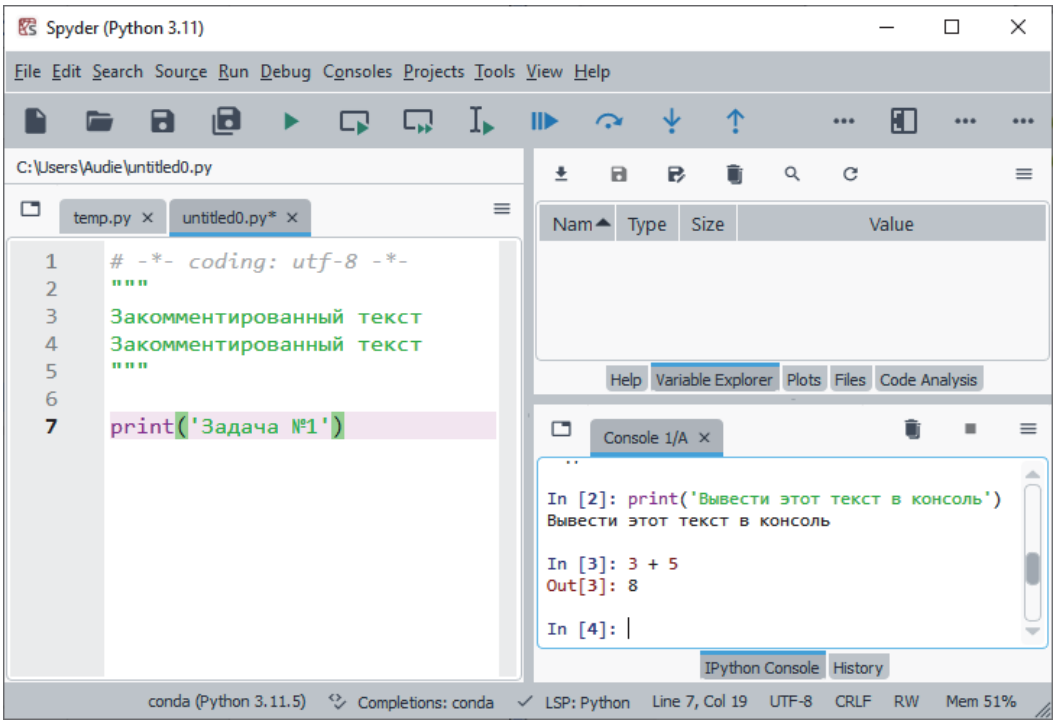
Способ 2. С помощью тройных кавычек `"""` или `'''`. Текст между ними является комментариями.

Пример:

```
""" текст закомментирован """
''' текст закомментирован '''
```

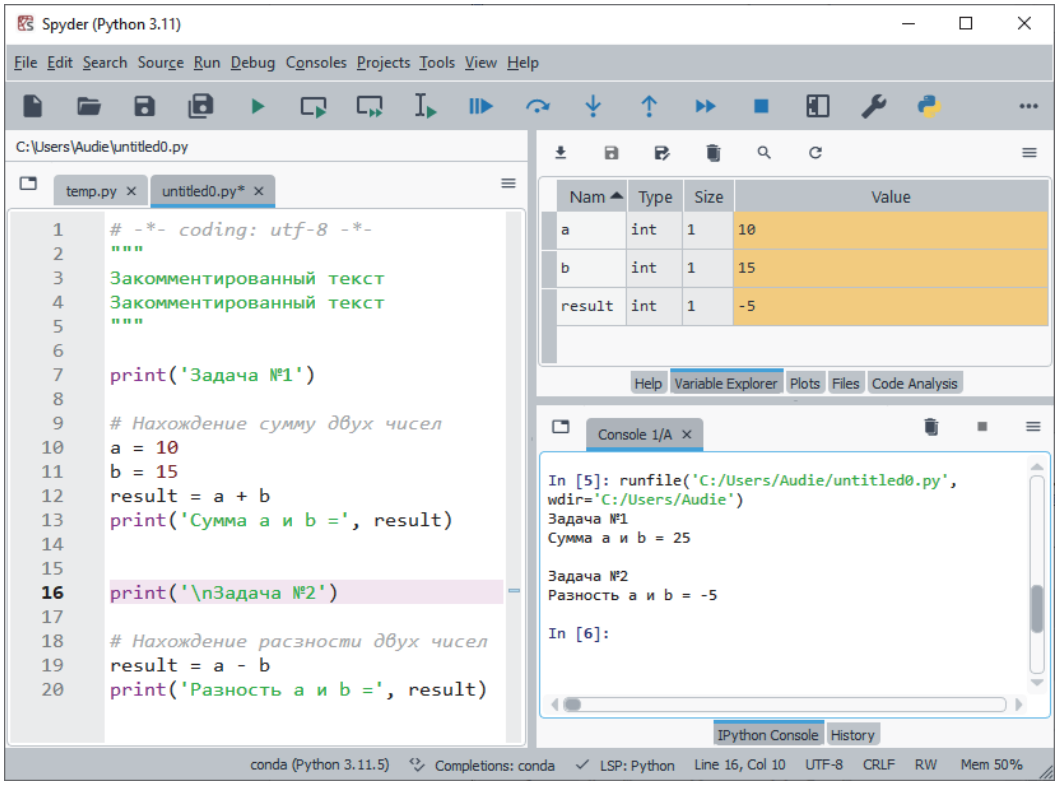
Команды можно писать либо в левой части среды разработки Spyder, либо в правой части (в консоли).

Для выполнения команды, введенной в консоль, необходимо нажать Enter. Пример:

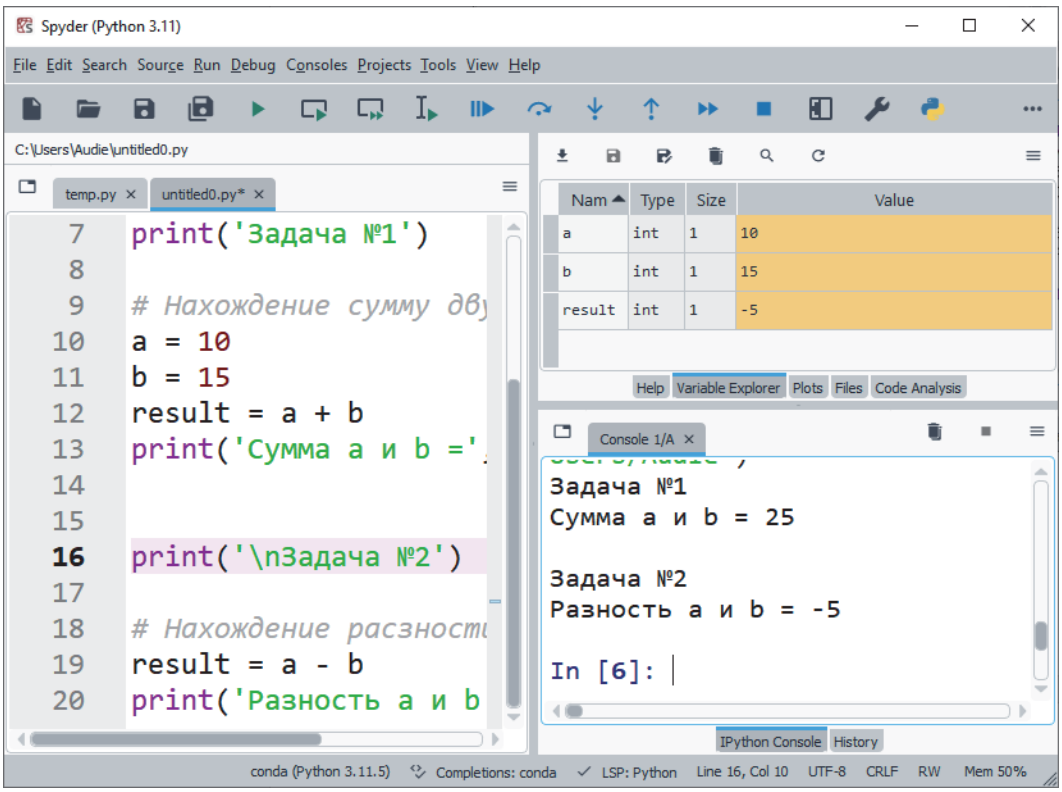


Здесь на строке `In [3]:` отображается введенная пользователем информация (введена команда `3 + 5`). На строке `Out [3]:` отображается результат выполнения команды.

При выполнении практических заданий рекомендуется писать код через скрипт и выводить результаты в консоль. Пример:



Для масштабирования текста (кода) скрипта можно воспользоваться комбинацией «Ctrl + Колесико мышки», для масштабирования текста консоли – «Ctrl +» или «Ctrl -», где «+» и «-» – кнопки, расположенные на цифровой клавиатуре (numpud):



Практическое задание № 2
«Строка. Подключение модулей в Python»

1. Строка

Для представления текста в Python используется тип объектов `str` (строка).

Строка (String) – набор символов, заключенный в кавычки (одинарные, двойные или тройные).

Строка – последовательность текста (text sequence).

Строка – неизменяемая последовательность кодовых точек (символов) Unicode [14].

Чтобы текст являлся строкой (строковым литералом), его необходимо разместить между кавычками. При этом могут быть использованы 4 типа кавычек

- одинарные кавычки: ' … '
- двойные кавычки: " … "
- тройные кавычки (три одинарных кавычки) ''' … '''
- тройные кавычки (три двойных кавычки) """… """

Пример использования четырех типов кавычек представлен в листинге 2.1.

Листинг 2.1 – Получение эквивалентных результатов при применении четырех типов кавычек для представления текста

```
t1 = 'text text 123'
t2 = "text text 123"
t3 = '''text text 123'''
t4 = """text text 123"""
print(t1, t2, t3, t4, sep='\n')
```

```
text text 123
text text 123
text text 123
text text 123
```

В отличие от двух других основных элементарных типов объектов `int` и `float`, объекты типа `str` являются последовательностями или наборами элементов (т.е. символов), для работы с которыми предоставляется дополнительный функционал в виде специальных символов экранирования, специальных методов строки, индексации элементов строки и т.д.

1.1. Управляющие последовательности. Префиксы

В строках могут быть использованы **управляющие последовательности**, которые являются не частью текста, а командами (инструкциями), изменяющими текст.

Управляющая последовательность (экранированная последовательности, escape sequences) состоит из:

- символа обратной косой черты \,
- специального буквенного символа (символов), например, символа n, означающего переход на новую строку (new line).

Т.е. управляющая последовательность для перевода текста на новую строку выглядит следующим образом: \n. Если такую управляющую последовательность включить в текст, то произойдет переход на новую строку в том месте, где она будет расположена (см. пример в листинг 2.1, строки 1, 2).

Перед текстом (т.е. перед значением объекта str, текстовым литералом, перед кавычками) могут быть расположены **префиксы**, которые также являются инструкциями для изменения текста.

Префикс строки (Prefix) – специальный буквенный символ, расположенный перед кавычками, выполняющий определенное действие по изменению текста в кавычках (строкового литерала).

Например, префикс r или R (от словосочетания row string literal) отключает выполнения управляющих последовательностей, т.е. представляет текст в кавычках таким образом, как будто там нет управляющих последовательностей (см. пример в листинге 2.2, строки 4, 5).

Листинг 2.2 – Пример использования управляющей последовательности \n и префикса строки r

```
1 t1 = 'text 1: Hello,\nWorld!'
2 print(t1)
3
4 t2 = r'text 2: Hello,\nWorld!'
5 print(t2)
```

```
text 1: Hello,
World!
text 2: Hello,\nWorld!
```

Так как текст в строке заключен в кавычки, то добавить в текст кавычки без помощи соответствующей управляющей последовательности \" или \', бывает затруднительно. Сделать это можно следующим образом (листинг 2.3).

Листинг 2.3 – Пример использования управляющей последовательности \" и префикса строки r

```
t1 = 'text 1: \"Hello, World!\"'
print(t1)

t2 = r'text 2: \"Hello, World!\"'
print(t2)
```

```
text 1: "Hello, World!"
text 2: \"Hello, World!\"
```

Список основных символов экранирования представлен в таблице 2.1.

Таблица 2.1 – Основные управляющие последовательности для строки (escape sequences) в Python 3.13

Управляющая последователь-ность	Описание	Примеры

\<newline>	Используется для переноса части строкового литерала на новую строку в редакторе кода, при этом в самом тексте перехода на новую строку не будет. Полезно для лучшего восприятия программного кода, если строковый литерал имеет слишком много символов и выходит вправо за пределы экрана.	`t1 = 'text 1: Hello, \`
		`World!'`
		`print(t1)`
		
		`t2 = 'text 1: Hello, \`
		`World! sample text \`
		`sample text \`
		`sample text \`
		`sample text'`
		`print(t2)`

\\	Добавление обратной косой черты в строковый литерал.	`t1 = 'file path: D:\\Downloads\\task.txt'` `print(t1)`
\'	Добавление одинарных кавычек в строковый литерал.	`t1 = 'text 1: \'Hello, World!\''` `print(t1)`
\"	Добавление двойных кавычек в строковый литерал.	`t1 = 'text 1: \"Hello, World!\"'` `print(t1)`
\a	Добавление звукового сигнала (bell) при отображении текста.	`t1 = '\aWarning!'` `print(t1)`
\b	Удаление предшествующего символа (backspace).	`t1 = 'text 1:\b\b\b\b\b\b\bHello, World!'` `print(t1)`
\f	Добавление «разрыва страницы» (formfeed) в текст.	`t1 = 'text 1:\fHello, World!'` `print(t1)`
\n	Перевод строки на новую строку (linefeed).	`t1 = 'text 1: Hello, \n\n \n World!'` `print(t1)`
\r	Возврат курсора в начало строки и перезапись первых *N* символов строки на *N* символов, стоящих после \r (carriage return).	`t1 = '        Hello, World!\rtext 1:'` `print(t1)`
\t	Добавление вертикальной табуляции в текст (vertical tab).	`t1 = 'text 1:\t\t\tHello, World!'` `t2 = 'text 12:\t\tsample`

Управляющая последователь-ность	Описание	Примеры

		`text'`
		`print(t1)`
		`print(t2)`

Источник: составлено на основе [20, 23].

1.2. f-строки

С помощью префикса `f` или `F` реализуются так называемые **f-строки** (f-strings, formatted string literal), с помощью которых можно удобно выводить на экран значения объектов, используя переменные.

F-строка может содержать области в фигурных скобках `{…}` (поля замены, replacement fields), где располагаются выражения, результаты выполнения которых преобразуются в строковый формат `str` и включаются в текст вместо соответствующих полей.

В общем виде f-строка выглядит следующим образом:

```
f'text text {expression} text text'
```

При отладке программы бывает полезно отобразить не только результат выражения, но и само выражение, или имя переменной перед значением объекта, на который это переменная ссылается. В таком случае после выражения (после имени переменной) ставится знак равенства `=` :

```
f'text text {expression = } text text'
```

Примеры использования f-строк представлены в листинге 2.4.

Листинг 2.4 – Примеры использования f-строк

```
t1 = f'Result 1: {100 * 5 - 30}'
print(t1)

t2 = f'Result 2: {100 * 5 - 30 = }'
print(t2)

a = 10 ** 5
b = a + 1
t3 = f'Result 3: {a=}, {b=}'
print(t3)

v1 = 'Tom'
t4 = f'My name is {v1}'
print(t4)
```

```
Result 1: 470
Result 2: 100 * 5 - 30 = 470
Result 3: a=100000, b=100001
My name is Tom
```

1.3. Операции над строками

Над строками могут быть выполнены следующие операции:

- конкатенация;
- повторение;
- проверка на вхождение;
- получение элементов строки по индексам;
- получение «среза», «нарезание»;
- использование специальных методов строки.

1.3.1 Конкатенация, повторение, проверка на вхождение

Конкатенация – слияние двух строк с помощью оператора `+`:

```
>>> 'text' + 'text'
'texttext'
```

Повторение строки – получение с использованием оператора * новой строки, состоящей из копий изначальной строки, между которыми реализована конкатенация:

```
>>> 'text' * 4
'texttexttexttext'
```

Проверка на вхождение – получение с использованием оператора `in` ответа (`True` или `False`) на вопрос «содержится ли указанная подстрока в строке?»:

```
>>> 't' in 'text'
True
```

Примеры реализации указанных операций представлены в листинге 2.5.

Листинг 2.5 – Примеры реализации операций над строками: конкатенация, повторение, проверка на вхождение

```
In [1]: 'My' + ' ' + 'name'
Out[1]: 'My name'

In [2]: '!' * 10
Out[2]: '!!!!!!!!!!'

In [3]: 'hello' in 'Hello, World!'
Out[3]: False

In [4]: 'Hello' in 'Hello, World!'
Out[4]: True
```

1.3.2 Индексация элементов строки. Срез

Строка является неизменяемой последовательностью символов. При этом к каждому символу можно обратиться по индексу (порядковому номеру). У каждого символа есть два индекса, т.е. два порядковых номера, связанных с прямым порядком и обратным порядком.

При этом прямой порядок номеров начинается с 0 (самый первый символ строки), а обратный порядок заканчивается номером -1 (самый последний символ строки) (рисунок 2.1).

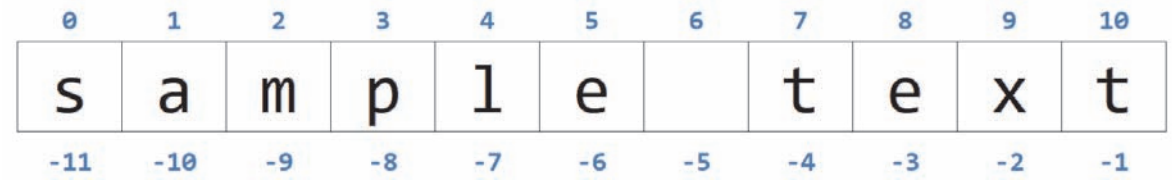


Рисунок 2.1 – Представление строки `'sample text'` в виде последовательности ячеек с расположенными в них символами

Обращение к элементу строки осуществляется с помощью квадратных скобок, в которых указывается индекс:

```
строка[индекс_элемента]
```

Индексация – предоставление числового смещения желаемого компонента путем указания индекса в квадратных скобках после последовательности (строки, списка и т.д.).

Индексация (`s[i]`) для строки `s` извлекает компоненты (элементы этой строки, символы) по смещениям (индексам).

Например, первый элемент находится по смещению 0 (т.е. имеет индекс 0). Если указать этот индекс в квадратных скобках после строки, то будет предоставлен первый элемент:

```
>>> 'abcdefg'[0]
a
```

Обычно в программе текст «сохраняется» в переменные. Доступ к элементам строки в таком случае происходит аналогичным образом:

```
>>> s = 'abcdefg'
>>> s[0]
a
```

Для получения последнего элемента можно указать отрицательный индекс, т.е. -1:

```
>>> s[-1]
g
```

Примеры обращения к элементам строки представлены в листинге 2.6.

Листинг 2.6 – Примеры обращения к элементам строки `'sample text'` по индексам

```
In [5]: 'sample text'[0]
Out[5]: 's'

In [6]: 'sample text'[-1]
Out[6]: 't'

In [7]: s = 'sample text'

In [8]: s[2]
Out[8]: 'm'

In [9]: s[-2]
Out[9]: 'x'

In [10]: s[6]
Out[10]: ' '
```

Для того, чтобы из последовательности отобрать элементы в определенном диапазоне (сделать срез), используют следующую запись:

```
последовательность[начальный_индекс : конечный_индекс : шаг]
```

При этом конечный индекс не включается в диапазон (верхняя граница является исключающей).

По умолчанию:

- начальный индекс = 0,
- конечный индекс = конечному индексу в массиве + 1,
- шаг = 1.

Ставить второе двоеточие и указывать шаг не обязательно. Значения всех трех параметров указывать не обязательно.

Нарезание (срез) (`s[i:j]`) извлекает непрерывный сегмент последовательности `s` начиная с элемента с индексом `i` и заканчивая элементом с индексом `j-1`.

Могут применяться следующие записи:

- `s[i:]` – извлекает непрерывный сегмент последовательности `s` начиная с элемента с индексом `i` и заканчивая самым последним элементом включительно:

```
>>> s = 'abcdefg'
>>> s[2:]
'cdefg'
```

- `s[:j]` – извлекает непрерывный сегмент последовательности `s` начиная с самого первого элемента (с индексом 0) и заканчивая элементом с индексом `j-1`:

```
>>> s = 'abcdefg'
>>> s[:4]
'abcd'
>>> s[:-1]
'abcdef'
```

- `s[:]` – извлекает непрерывный сегмент последовательности `s` начиная с самого первого элемента (с индексом 0) и заканчивая самым последним элементом включительно (извлекаются все элементы):

```
>>> s = 'abcdefg'
>>> s[:]
' abcdefg'
```

Расширенное нарезание (`s[i:j:k]`) включает шаг `k`, по умолчанию `k = 1`. Такая запись позволяет пропускать элементы на определенных позициях (например, пропускать элементы с нечетными индексами) и менять порядок на противоположный [5]:

```
>>> s = 'abcdefg'
>>> s[0:7:2]
'aceg'
>>> s[6::-1]
'gfedcba'
```

При этом, если требуется указать шаг, отличный от 1, но работать со всеми элементами строки, то первые два параметры `i` и `j` указывать не обязательно. Запись может выглядеть следующим образом:

```
>>> s = 'abcdefg'
>>> s[::2]
'aceg'
>>> s[::-1]
'gfedcba'
```

1.3.3 Методы строк

У объекта типа `str` имеются специальные методы, т.е. привязанные к этому объекту функции, к которым можно обращаться через оператор `.` (точка) следующим образом:

```
объект.метод_объекта()
```

Такие методы выполняют определенные действия по изменению объекта (строки). Например, метод `.upper()` изменяет все буквенные символы строки на заглавные (листинг 2.7).

Листинг 2.7 – Пример вызова метода `.upper()` у строкового объекта `t1` и «сохранение результата» в переменную `t2`

```
1 t1 = 'Sample text 123'
2 t2 = t1.upper()
3 print(t2)
```

```
SAMPLE TEXT 123
```

Основные методы строки представлены в таблице 2.2.

Таблица 2.2 – Основные методы строки (str) в языке Python 3.13

Функция	Описание	Примеры

`.capitalize()`	Возвращает копию строки, в которой первый символ, если это буква, является заглавной буквой.	`>>> s = 'hello, world!'`
		`>>> s.capitalize()`
		`Hello, world!`

`.count(sub)`
 	
`.count(sub[, start[, end]])`
	Возвращает количество подстрок `sub` (от слова substring – подстрока) в диапазоне [`start, end`]. Параметры start и end являются необязательными и интерпретируются как индексы среза.	`>>> s.count('h')`
		`1`
		`>>> s.count('H')`
		`0`
		`>>> s.count('o')`
		`2`
		`>>> s.count('o', 0, 5)`
		`1`

`.find(sub)`
 	
`.find(sub[, start[, end]])`
	Возвращает индекс первого элемента первой найденной подстроки `sub` в диапазоне [`start, end`]. Если подстроки `sub` нет в строке, то возвращает -1.	`>>> s.find('o')`
		`4`
		`>>> s.find('o', 5)`
		`8`
		`>>> s.find('o', 5, -1)`
		`8`
		`>>> s.find('world')`
		`7`
		`>>> s.find('World')`
		`-1`

`.index(sub)`
 	
`.index(sub[, start[, end]])`
	Действует как `.find()`, но вызывает исключение `ValueError`, если подстрока `sub` не найдена.	`>>> s.index('world')`
		`7`
		`>>> s.index('World')`
		`ValueError: substring not found`

`.isalnum()`	Возвращает `True`, если все символы в строке являются буквами или числами (alphanumeric characters) и в строке хотя-бы один символ. Иначе возвращает `False`.	`>>> s.isalnum()`
		`False`
		`>>> 'aaa123'.isalnum()`
		`True`
		`>>> 'hello,world!'.isalnum()`
		`False`
		`>>> 'helloworld'.isalnum()`
		`True`

Функция	Описание	Примеры

.isalpha()	Возвращает `True`, если все символы в строке являются буквами (alphabetic charactersc) и в строке хотя-бы один символ. Иначе возвращает `False`.	`>>> s.isalpha()`
		`False`
		`>>> 'helloworld'.isalpha()`
		`True`
		`>>> 'helloworld5'.isalpha()`
		`False`

.isdecimal()	Возвращает `True`, если все символы в строке являются десятичными числами (decimal characters) и в строке хотя-бы один символ. Иначе возвращает `False`.	`>>> '13456'.isdecimal()`
		`True`
		`>>> '13456a'.isdecimal()`
		`False`

.isdigit()	Возвращает `True`, если все символы в строке являются цифрами (digits) и в строке хотя-бы один символ. Иначе возвращает `False`.	`>>> '13456'.isdigit()`
		`True`
		`>>> '13456a'.isdigit()`
		`False`

.islower()	Возвращает `True`, если все символы в строке являются строчными (digits) и в строке хотя-бы один символ. Иначе возвращает `False`.	`>>> s.islower()`
		`True`
		`>>> 'aaa17'.islower()`
		`True`
		`>>> 'aAa17'.islower()`
		`False`

.isnumeric()	Возвращает `True`, если все символы в строке являются числовыми символами (numeric characters) и в строке хотя-бы один символ. Иначе возвращает `False`.	`>>> '13456'.isnumeric()`
		`True`
		`>>> '13456a'.isnumeric()`
		`False`

.join()
 	
.join(iterable)
	Возвращает строку, которая является результатом конкатенации основной строки и каждым элементом итерируемого объекта `iterable`.	`>>> s2 = 'text'`
		`>>> s2.join('123')`
		`'1text2text3'`
		`>>> ', '.join('abcdf')`
		`'a, b, c, d, f'`
		`>>> '+'.join('10 20 30')`
		`'1+0+ +2+0+ +3+0'`

.lower()	Возвращает копию строки, в которой все буквенные символы являются строчными буквами.	`>>> 'Hello, World'.lower()` `>>> 'hello, world'`

.lstrip()
 	
.lstrip([chars])
	Возвращает копию строки, в которой удалены лишние пробелы слева. Если указан аргумент `chars`, вместо пробелов	`>>> '   hello '.lstrip()`
		`'hello '`
		`>>> ' hello'.lstrip(' hl')`
		`'ello'`

Функция	Описание	Примеры
	удаляются символы chars.	

.replace(old, new)
 	
.replace(old, new, count=-1)
	Возвращает копию строки, в которой все подстроки old заменены на подстроки new. Если указан аргумент count, то количество замен = count.	>>> 'Hello'.replace('l', '?')
		'He??o'
		>>> 'Hello'.replace('l', '?', 1)
		'He?lo'

.rfind(sub)
 	
.rfind(sub[, start[, end]])
	Действует как .find(), но возвращает индекс первого элемента *последней* найденной подстроки sub в диапазоне [start, end]. Если подстроки sub нет в строке, то возвращает -1.	>>> 'Hello'.rfind('l')
		3
		>>> 'Hello'.find('l')
		2

.rsplit() .rsplit(sep=None, maxsplit=-1)	Действуйте схожим образом с методом .split(). Возвращает список подстрок, которые разделяются через разделитель sep. Если указан аргумент maxsplit, то максимальное число разбиений = maxsplit. В отличие от .split() разбиения начинаются справа.	>>> '1,2,3,4'.rsplit(',', 1) ['1,2,3', '4']

.rstrip()
 	
.rstrip([chars])
	Возвращает копию строки, в которой удалены лишние пробелы справа. Если указан аргумент chars, вместо пробелов удаляются символы chars.	>>> ' hello '.rstrip()
		' hello'
		>>> ' hello'.rstrip('lo')
		' he'

.split()
 	
.split(sep=None, maxsplit=-1)
	Возвращает список подстрок, которые разделяются через разделитель sep. Если указан аргумент maxsplit, то максимальное число разбиений = maxsplit.	>>> 'Hello'.split('l')
		['He', '', 'o']
		>>> '1,2,3,4'.split(',')
		['1', '2', '3', '4']
		>>> '1,2,3,4'.split(',', 1)
		['1', '2,3,4']

.strip()
 	
.strip([chars])
	Возвращает копию строки, в которой удалены лишние пробелы слева и справа. Если указан аргумент chars, вместо пробелов удаляются символы chars.	>>> ' hello '.strip()
		'hello'
		>>> ' hello '.strip('ho')
		'ell'

.title()	Возвращает версию строки, в которой во всех словах первые буквы	>>> 'hello, world!'.title() 'Hello, World!'

Функция	Описание	Примеры
	являются заглавными.	`>>> 'aa bbb,cd fff'.title() 'Aa Bbb,Cd Fff'`
`.upper()`	Возвращает копию строки, в которой все буквенные символы являются заглавными буквами.	`>>> 'Hello, World'.upper() 'HELLO, WORLD'`

Источник: составлено на основе [14].

2. Модули

Модуль – файл с расширением «.py», программа на языке Python, которая может быть подключена к основной программе для предоставления возможности пользоваться функционалом (элементами модуля: функциями, классами и т.д.) подключаемой программы в основной программе.

Как правило, такой функционал узко специализированный, имеет определенную направленность в соответствии с названием модуля. Например, модуль `math` предоставляет доступ к большому числу функций математического характера, таких как `cos()`, `log()`, `exp()`, а также к переменным, представляющим математические константы, такие как `pi`, `e`, `inf`.

К числовым и математическим модулям (Numeric and Mathematical Modules) относятся следующие:

- `numbers` – числовые абстрактные базовые классы;
- `math` – математические функции;
- `cmath` – математические функции для комплексных чисел;
- `decimal` – для работы с десятичными числами с фиксированной и плавающей точкой;
- `fractional` – для работы с рациональными числами;
- `random` – для генерации псевдослучайных чисел;
- `statistics` – функции математической статистики.

Список других модулей можно найти на сайте официальной документации Python [26].

Для подключения модуля к основной программе используется ключевое слово import, далее указывается название модуля (файла):

```
import название_модуля
```

Для доступа к элементам модуля (функциям, классам, переменным) используется оператор `.` (точка):

```
название_модуля.элемент_модуля
```

Обычно в среде разработке после указания оператора `.` появляется подсказка. В примере подключения модуля `math`, представленного на рисунке 2.1, появляется подсказка в виде окна со списком элементов модуля `math`.

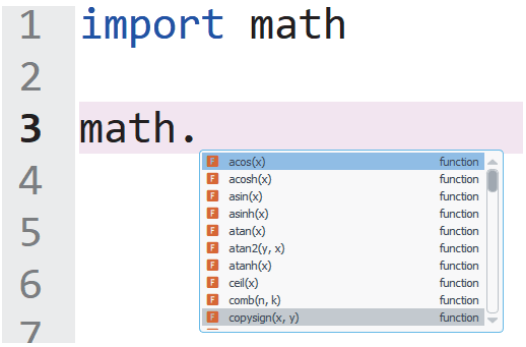


Рисунок 2.1 – Подключение модуля `math`

Список основных функций и констант модуля `math` представлен в таблице 2.3.

Таблица 2.3 – Некоторые функции и константы модуля `math` в языке Python 3.13

Функция / константа	Описание
Функции	
`atanh(x)`	Обратный гиперболический тангенс `x`.
`comb(n, k)`	Количество способов выбрать `k` элементов из `n` элементов без повторения и без учета порядка.
`cos(x)`	Косинус `x`.
`exp(x)`	Экспонента *e*, возведенная в степень `x`.
`fabs(x)`	Абсолютное значение числа `x`.
`factorial(n)`	Факториал числа `n`.
`floor(x)`	Наибольшее целое число, меньшее или равное `x`.
`fsum(iterable)`	Сумма значений во входном итерируемом объекте `iterable` (массиве).

`gcd(*integers)`	Наибольший общий делитель целых чисел.
	`*integers` означает, что на вход передается произвольное количество аргументов через запятую, например:
	`>>> gcd(10, 20, 15)`
	`5`

`lcm(*integers)`	Наименьшее общее кратное целых чисел.
`log(x, base)`	Логарифм `x` по основанию `base` (по умолчанию `base` = *e*)
`log2(x)`	Логарифм `x` по основанию 2.
`log10(x)`	Логарифм `x` по основанию 10.
`pow(x, y)`	Число `x` в степени `y`.
`sin(x)`	Синус `x`.
`sqrt(x)`	Квадратный корень из числа `x`.
`tan(x)`	Тангенс `x`.
`tanh(x)`	Гиперболический тангенс `x`.
Константы	
`pi`	π = 3,141592…
`e`	e = 2,718281…
`tau`	τ = 2 π = 6,283185…
`inf`	Положительная бесконечность (positive infinity)
`nan`	«Не число» («Not a number» – NaN)

Источник: составлено на основе [21, 14].

Пример применения некоторых функций и переменных, ссылающихся на математические константы модуля `math` представлен в таблице 2.3.

Листинг 2.8 – Пример использования функций и переменных модуля `math`

```
import math

print('pi: ', math.pi)

res1 = math.cos(1)
print('res1: ', res1)

res2 = math.factorial(5) + math.log(15)
print('res2: ', res2)
```

```
pi:  3.141592653589793
res1:  0.5403023058681398
res2:  122.70805020110221
```

Задачи

Задача № 1

Создайте объект ***name***: « My name is »

1. Уберите лишние пробелы.
2. Выведите в консоль текст: «My name is ФИО» (ФИО – ваше собственное).
3. Определите число символов в п.2.
4. Выведите в консоль текст:
   «My name is
   ФИО»
5. Выведите в консоль все слова п.2 через знак табуляции.
6. Для п.5 уберите все символы экранирования.
7. Повторите 8 раз фразу из п.2.
8. Выведите в консоль «MY NAME IS» и проверьте, все ли символы заглавные.
9. Выведите все символы в п.2 с заглавных букв.
10. Выведите в консоль «NAME» как часть строки из п.9.
11. Проверьте, есть ли символы «m», «i», «t» в п.2.
12. Замените «na» на «****» в п.2.
13. Найдите позицию символа «y» в п.2.

Задача № 2

Создайте объект ***t***: «a1 a2 a3».

1. Разделите строку на подстроки и добавьте их в список.
2. Разделите объект ***t*** на 2 объекта.
3. Соедините элементы списка из п.1 в одну строку через знак «,».

4. Разбейте объект в п.3 на 2 подстроки через знак «,», при этом поиск знака осуществите справа налево.

Задача № 3

Создайте переменную `x`, ссылающуюся на значение, соответствующее вашему варианту. Например, если вариант 1, то `x = 1`.

Рассчитайте результат применения следующих формул:

1. $x + 10!$
2. $(x - x^2 - x^3) + (x + x^2 + x^3)$
3. $\sqrt{|e^x + x! \cdot \pi|}$, ответ округлите в меньшую сторону;
4. $(\lg(x) + \sqrt{x^3})^{x-5}$, ответ округлите в большую сторону;
5. $x^e + \frac{\sin(x \cdot \pi)}{\tan x^{-1}}$

Задача № 4

Создайте переменную `x`, ссылающуюся на значение, соответствующее вашему варианту. Например, если вариант 1, то `x = 1`.

Вывести на печать, заменив «*» фактическими значениями:

1. «Наибольший общий делитель чисел x, 20, 50 равен *»
2. «Наибольший общий делитель чисел 10x, 20, 50, 100, 15000 равен *»
3. «Наименьшее общее кратное чисел x, 18, 50 равно *»
4. «cos(x) = *.***» *(округлить до 3 знаков после запятой)*
5. «$\sin^2$(x) + $\cos^2$(x) = *.** + *.** = 1» *(округлить до 2 знаков после запятой)*

Практическое задание № 3
«Список. Кортеж. Итерация по списку. Условный оператор»

1. Список

Списки Python могут напоминать **массивы** в других языках, но они, как правило, более мощные [5].

Список (List) – позиционно упорядоченная коллекция объектов произвольных типов, разделенных запятой, заключенная в квадратные скобки.

Список – массив ссылок на объекты. Формально список Python содержит ноль и более ссылок на другие объекты.

Списки – гибкий инструмент для представления произвольных коллекций, например:

- перечня файлов в каталоге:
  ['data.xlsx', 'info.txt', 'Report1.docx', 'Report2.docx']
- сотрудников в компании:
  ['Tom', 'R. Roy', 'Donald D.', 'Mr. Smith']
- сообщений в ящике электронной почты:
  ['message1', 'message2', 'message3']
- динамики численности товаров на складе за несколько дней:
  [10, 10, 8, 9, 5]
  почасовой температуры воздуха, получаемой с датчика:
- [13.2, 13.0, 11.4, 11.2, 11.0]

Свойства списка:

- не имеет фиксированного размера;
- изменяемый объект (в отличие от строки его можно модифицировать на месте);
- может содержать объекты любого вида: числа, строки и даже другие списки.

1.1. Операции над списками

Список достаточно сильно схож со строкой. Так же, как и строка, список состоит из элементов, к каждому из которых можно обратиться по индексу; список имеет специальные методы, а также над списками поддерживаются операции слияния (конкатенации), повторения, проверки на вхождения.

Таким образом, над списками могут быть выполнены те же операции, что и над строками:

- конкатенация;
- повторение;
- проверка на вхождение;
- получение элементов списка по индексам;

- получение «среза», «нарезание»;
- использование специальных методов списка.

1.1.1 Конкатенация, повторение, проверка на вхождение

Конкатенация (списков) – слияние двух списков с помощью оператора `+`:

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

Повторение (списка) – получение с использованием оператора `*` нового списка, состоящего из копий изначального списка, между которыми реализована конкатенация:

```
>>> [1, 2, 3] * 2
[1, 2, 3, 1, 2, 3]
```

Проверка на вхождение – получение с использованием оператора `in` ответа (`True` или `False`) на вопрос «содержится ли указанная подстрока в строке?»:

```
>>> 2 in [1, 2, 3]
True
```

Дополнительные примеры реализации указанных операций представлены в листинге 3.1.

Листинг 3.1 – Примеры реализации операций над списками: конкатенация, повторение, проверка на вхождение

```
In [1]: [1] + [2] + [3, 4, 5]
Out[1]: [1, 2, 3, 4, 5]

In [2]: [0] * 10
Out[2]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [3]: 1 in [10, 2, 'text', 100.07]
Out[3]: False

In [4]: 100.07 in [10, 2, 'text', 100.07]
Out[4]: True
```

1.1.2 Индексация элементов строки. Срез

Список является изменяемой последовательностью элементов. Это означает, что каждый элемент списка может быть изменен, а сам список может быть расширен или очищен. Данная особенность отличает список от

строки, которая является неизменяемой последовательностью элементов (символов).

В списке, так же, как и в строке, к каждому элементу можно обратиться по индексу (порядковому номеру). У каждого элемента есть два индекса, т.е. два порядковых номера, связанных с прямым порядком и обратным порядком.

Прямой порядок номеров начинается с 0 (самый первый элемент списка), а обратный порядок заканчивается номером -1 (самый последний элемент списка) (рисунок 3.1).

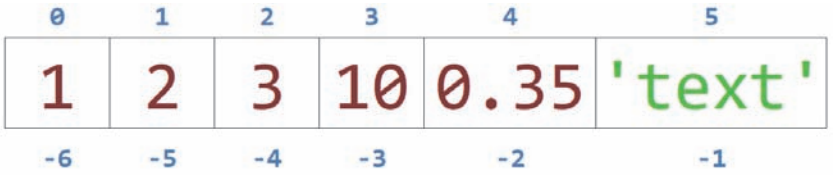


Рисунок 3.1 – Представление списка `[1, 2, 3, 10, 0.35, 'text']` в виде последовательности ячеек с расположенными в них элементами списка

Индексация и получение среза в списке осуществляются таким же образом, как и в случае строки.

Обращение к элементу списка реализуется с помощью квадратных скобок, в которых указывается индекс:

```
список[индекс_элемента]
```

Индексация (`L[i]`) для списка `L` извлекает компоненты (элементы этого списка) по смещениям (индексам).

Например, первый элемент находится по смещению 0 (т.е. имеет индекс 0). Если указать этот индекс в квадратных скобках после списка, то будет предоставлен первый элемент:

```
>>> [10, 20, 30, 40][0]
10
```

Доступ к элементам списка при использовании переменной происходит аналогичным образом:

```
>>> L = [10, 20, 30, 40]
>>> L[0]
10
```

Для получения последнего элемента можно указать отрицательный индекс, т.е. -1:

```
>>> s[-1]
g
```

Дополнительные примеры обращения к элементам списка представлены в листинге 3.2.

Листинг 3.2 – Примеры обращения к элементам списка `[1, 2, 3, 10, 0.35, 'text']` по индексам

```
In [5]: [1, 2, 3, 10, 0.35, 'text'][0]
Out[5]: 1

In [6]: [1, 2, 3, 10, 0.35, 'text'][-1]
Out[6]: 'text'

In [7]: L = [1, 2, 3, 10, 0.35, 'text']

In [8]: L[2]
Out[8]: 3

In [9]: L[-2]
Out[9]: 0.35

In [10]: L[-1][1]
Out[10]: 'e'
```

Для того, чтобы из последовательности (в том числе списка) отобрать элементы в определенном диапазоне (сделать срез), используют следующую запись:

```
последовательность[начальный_индекс : конечный_индекс : шаг]
```

При этом конечный индекс не включается в диапазон (верхняя граница является исключающей).

По умолчанию:

- начальный индекс = 0,
- конечный индекс = конечному индексу в массиве + 1,
- шаг = 1.

Ставить второе двоеточие и указывать шаг не обязательно. Значения всех трех параметров указывать не обязательно.

Нарезание (срез) (`L[i:j]`) извлекает непрерывный сегмент последовательности `L` начиная с элемента с индексом `i` и заканчивая элементом с индексом `j-1`.

Могут применяться следующие записи:

- `L[i:]` – извлекает непрерывный сегмент последовательности `s` начиная с элемента с индексом `i` и заканчивая самым последним элементом включительно:

```
>>> L = [10, 20, 30, 40, 150, 1000]
>>> L[2:]
[30, 40, 150, 1000]
```

- `L[:j]` – извлекает непрерывный сегмент последовательности `L` начиная с самого первого элемента (с индексом 0) и заканчивая элементом с индексом `j-1`:

```
>>> L = [10, 20, 30, 40, 150, 1000]
>>> L[:4]
[10, 20, 30, 40]
>>> L[:-1]
[10, 20, 30, 40, 150]
```

- `L[:]` – извлекает непрерывный сегмент последовательности `L` начиная с самого первого элемента (с индексом 0) и заканчивая самым последним элементом включительно (извлекаются все элементы):

```
>>> L = [10, 20, 30, 40, 150, 1000]
>>> L[:]
[10, 20, 30, 40, 150, 1000]
```

Расширенное нарезание (`s[i:j:k]`) включает шаг `k`, по умолчанию `k = 1`. Такая запись позволяет пропускать элементы на определенных позициях (например, пропускать элементы с нечетными индексами) и менять порядок на противоположный [5]:

```
>>> L = [10, 20, 30, 40, 150, 1000]
>>> L[0:6:2]
[10, 30, 150]
>>> L[5::-1]
[1000, 150, 40, 30, 20, 10]
```

При этом, если требуется указать шаг, отличный от 1, но работать со всеми элементами строки, то первые два параметры `i` и `j` указывать не обязательно. Запись может выглядеть следующим образом:

```
>>> L = [10, 20, 30, 40, 150, 1000]
>>> L[::2]
[10, 30, 150]
>>> L[::-1]
[1000, 150, 40, 30, 20, 10]
```

1.1.3 Методы списка

У объекта типа `list` имеются специальные методы – привязанные к этому объекту функции, к которым можно обращаться через оператор `.` (точка) следующим образом:

```
объект.метод_объекта()
```

Такие методы выполняют определенные действия по изменению объекта (списка).

Основные операции со списком:

1) `= []` – создание пустого списка.
2) `list()` – создание пустого списка.
3) `.append(x)` – добавление элемента `x` в список (в конец).
4) `.insert(i, x)` – добавление элемента `x` в список (на определенную позицию) (в качестве аргументов указывается позиция вставляемого элемента `i` и сам элемент).
5) `.pop()` – извлечение из списка (с конца). Также можно указывать индекс последнего элемента: `.pop(-1)`
6) `.pop(0)` – извлечение из списка (с начала) (в качестве параметра указывается индекс удаляемого элемента – первого элемента списка, то есть `0`).

Примеры создания, заполнения и извлечения элементов из списка представлены в таблице 3.1.

Таблица 3.1 – Примеры создания, заполнения и извлечения элементов из списка

Инструкция	Имя операции	Пример команды	Содержание списка
`= []`	Создание пустого списка	`q = []`	`[]`

`.append()`	Добавление элемента в список (в конец)	`q.append(1)`
		`q.append(2)`
		`q.append(3)`
			`[1]`
			`[1, 2]`
			`[1, 2, 3]`

`.pop()`	Извлечение из списка (с начала)	`q.pop(0)`
		`q.pop(0)`
		`q.pop(0)`
			`[2, 3]`
			`[3]`
			`[]`

`.insert()`	Добавление элемента в список (на определенную позицию)	`q.insert(0, 3)`
		`q.insert(0, 2)`
		`q.insert(0, 1)`
			`[3]`
			`[2, 3]`
			`[1, 2, 3]`

`.pop()`	Извлечение из списка (с конца)	`q.pop(-1)`
		`q.pop()`
		`q.pop()`
			`[1, 2]`
			`[1]`
			`[]`

Источник: составлено на основе: [5, 14].

Дополнительные операции:

- `len(q)` – определение размера списка `q`.
- `del q[i]` – удаление элемента списка `q` с индексом `i` (альтернатива `.pop(i)`).
- `del q[i:j]` – удаление секции списка `q` с индексами от `i` до `j`.
- `q[i] = a` – вставка списка `a` (подсписка) в список `q` на позицию `i`.
- `q[i:i] = a` – вставка списка `a` (секции) в список `q` на позицию `i`.
- `q.clear()` – очистка списка.

- `q.sort()` – сортировка по возрастанию.
- `q.reverse()` – сортировка по убыванию.
- `q.copy()` – копирование списка.
- `q.remove(x)` – удаление из списка `q` первого элемента `x`.
- `q.count(x)` – количество элементов `x` в списке `q`.
- `q.index(x)` – индекс первого элемента `x` в списке `q`.
- `q.extend(iter)` – добавление в конец списка `q` всех элементов итерируемого объекта `iter` (например, строки или другого списка).

Таблица 3.2 – Примеры использования дополнительных операций над списком

Инструкция	Имя операции	Пример команды	Содержание списка / результат выполнения
`len()`	Количество элементов в списке	`q = [3, 1, 2, 4]` `len(q)`	`[3, 1, 2, 4]` `4`
`.sort()`	Сортировка (по возрастанию)	`q.sort()`	`[1, 2, 3, 4]`
`.reverse()`	Сортировка (по убыванию)	`q.reverse()`	`[4, 3, 2, 1]`
`del q[i:j]`	Удаление секции	`del q[2:4]`	`[4, 3]`
`q[i] = a`	Вставка подсписка	`q[1] = [1, 1]`	`[4, [1, 1]]`
`del q[i]`	Удаление подсписка (как элемента списка)	`del q[1]`	`[4]`
`q[i:i] = a`	Вставка секции	`q[0:0] = [5, 6, 7]`	`[5, 6, 7, 4]`
`q.extend(iter)`	Расширение списка (вставка секции в конец)	`q.extend([1, 1])`	`[5, 6, 7, 4, 1, 1]`
`q.clear()`	Очистка списка	`q.clear()`	`[]`

`q.copy()`	Копирование списка
	Копирование списка с использованием метода `copy()` *вместо оператора присваивания* `=` *позволяет избежать изменение основного списка при изменении копии (например, при очистке)*
		`q = [3, 1, 2, 4]`
		`q1 = q`
		`q1.clear()`
		`q`
		`q1`
		
		`q = [3, 1, 2, 4]`
		`q2 = q.copy()`
		`q2.clear()`
		`q`
		`q2`
			`[3, 1, 2, 4]`
			`[3, 1, 2, 4]`
			
			`[]`
			`[]`
			
			`[3, 1, 2, 4]`
			`[3, 1, 2, 4]`
			
			`[3, 1, 2, 4]`
			`[]`

Источник: составлено на основе [5, 14].

1.1.4 Упаковка и распаковка элементов списка

В Python поддерживается **множественное присваивание**, которое основывается на распаковке элементов коллекции.

Распаковка (Unpacking) или **Деструктуризация** – разложение коллекции (например, списка или строки) на элементы [9].

Пример распаковки представлен в листинге 3.3, где четырем переменным `v1`, `v2`, `v3`, `v4` присваиваются элементы списка (строка кода 3) и элементы строки (строка кода 7).

Листинг 3.3 – Пример множественного присваивания переменным `v1`, `v2`, `v3`, `v4` при разложении списка `L` и строки `s`

```
1 L = [10, 11, 14, 15]
2
3 v1, v2, v3, v4 = L
4 print(v1, v2, v3, v4)
5
6 s = 'abcd'
7 v1, v2, v3, v4 = s
8 print(v1, v2, v3, v4)
```

```
10 11 14 15
a b c d
```

Упаковка (Packing) – объединение отдельных значений в одну коллекцию с помощью оператора *.

Например, в листинге 3.4 представлен пример упаковки значений

Листинг 3.4 – Пример упаковки объектов `v1`, `v2`, `v3`, в список `L2` (дополнительно необходимо указать запятую `,` поле имени коллекции для формирования списка)

```
1 L = [10, 11, 14, 15]
2
3 v1, v2, v3, v4 = L
4
5 *L2, = v1, v2, v3
6 print(L2)
```

```
[10, 11, 14]
```

Упаковка и распаковка часто применяются при создании собственных функций (см. практическое задание № 5).

2. Кортеж

Достаточно часто возникает потребность в **фиксации списка**, т.е. в наделении списка свойством **неизменяемости** для того, чтобы список нельзя было очистить, расширить, изменить определенные элементы случайно или преднамеренно с целью сохранения информации.

Для решения такой проблемы в Python используется тип данных **кортеж**.

Кортеж – неизменяемая последовательность объектов [5].

Кортеж можно определить как фиксированный (неизменяемый) список.

Кортеж имеет схожесть со строкой (он неизменяем) и со списком (является коллекцией или набором объектов).

Кортеж так же, как список и строка поддерживает основные операции над последовательностями:

- конкатенация,
- повторение,
- проверка на вхождение,
- индексация (срез, нарезания).

Кортеж создается с помощью функции `tuple()` или с помощью круглых скобок.

Примеры операций с кортежем представлены в листинге 3.5.

Листинг 3.5 – Примеры основных операций с кортежем

```
In [1]: tup = (1, 2, 3)

In [2]: tup + (4, 5)
Out[2]: (1, 2, 3, 4, 5)

In [3]: tup * 2
Out[3]: (1, 2, 3, 1, 2, 3)

In [4]: 1 in tup
Out[4]: True

In [5]: tup[0]
Out[5]: 1

In [6]: tup[:2]
Out[6]: (1, 2)

In [7]: tup[0] = 10
Traceback (most recent call last):

  Cell In[7], line 1
    tup[0] = 10

TypeError: 'tuple' object does not support item assignment
```

3. Итерация по списку через цикл `for`

Для перебора элементов в коллекции (списке) используется цикл `for` в связке с оператором `in` [5]. Также может использоваться функция `range()` [13].

Цикл `for` проходит по элементам в любой последовательности слева направо, выполняя один и более операторов для каждого элемента.

`in` используется для проверки членства для символов и подстрок. В записи типа `for i in [0, 1, 2, 3]` переменная `i` становится курсором, проходящим по элементам списка.

`range(stop)` или `range(start, stop, step=1)` производит последовательные целые числа в диапазоне от `start` до `stop` с шагом `step`. При этом по умолчанию `step=1`, а также:

- если указывается только один аргумент, то вызывается `range(stop)` (в этом случае подразумевается, что `start=0`, `step=1`),
- если указывается 2 или 3 аргумента, то вызывается `range(start, stop, step=1)`.

Часто используется в подобных записях: `for i in range(4)`, что эквивалентно `for i in range(0, 4, 1)`.

Примеры прохода по списку отражены в листинге 3.6.

Листинг 3.6 – Примеры итерации по элементам списка несколькими способами (в консоли отображаются элементы списка)

```
# Список из 5 элементов
list1 = [10, 15, 20, 25, 500]

# Проход по списку (способ №1)
for element in list1:
    print(element, end=' ')

# Проход по списку (способ №2). i - индекс
for i in range(5):
    print(list1[i], end=' ')

# Проход по списку (способ №2). i - индекс
for i in range(len(list1)):
    print(list1[i], end=' ')
```

```
10 15 20 25 500
10 15 20 25 500
10 15 20 25 500
```

Примечание: для того, чтобы элементы списка отобразились не в столбик, а в строку через, например, пробел или знак табуляции, необходимо указать значение для аргумента `end` при вызове `print()`, например `end=' '` или `end='\t'`.

4. Условный оператор `if`

Оператор `if` (от слова «if» – если) языка Python выбирает действия для выполнения по условию. Он принимает форму проверки `if`, за которой следует одна или большее количество проверок `elif` (сокращение от «else if» – иначе если) и финальный необязательный блок `else` (иначе, т.е. во всех остальных случаях) [5].

4.1. Форма записи конструкции «if-elif-else»

Общая форма такой конструкции «if-elif-else» выглядит следующим образом (рисунок 3.1):

```
if проверка_1:          # Проверка if
    блок_инструкций_1   # Связанный блок
elif проверка_2:        # Необязательный elif
    блок_инструкций_2
    ...
elif проверка_N-1:      # Необязательный elif
    блок_инструкций_N-1
else:                   # Необязательный else
    блок_инструкций_N
```

Рисунок 3.1 – Общая форма конструкции «if-elif-else»

Источник: составлено на основе [5].

Каждой части конструкции «if-elif-else» соответствует блок инструкций.

Первая проверка после части `if` является обязательной при использовании конструкции «if-elif-else». Остальные проверки и соответствующие им блоки кода с инструкциями являются необязательными. Т.е. в самом простом виде конструкция «if-elif-else» содержит только часть if (рисунок 3.2).

```
if проверка_1:          # Проверка if
    блок_инструкций_1   # Связанный блок
elif проверка_2:        # Необязательный elif
    блок_инструкций_2
    ...
elif проверка_N-1:      # Необязательный elif
    блок_инструкций_N-1
else:                   # Необязательный else
    блок_инструкций_N
```

Рисунок 3.2 – Форма конструкции «if-elif-else» в самом простом виде, включающем только обязательную часть `if`

Источник: составлено на основе [5].

Проверки начинаются сверху вниз. При этом применяется следующий алгоритм:

1. Проверка текущего условия *i*:
   - Если проверка *i* (условие *i*) дает **истинный результат**, то происходит переход в соответствующий блок инструкций (блок *i*) (шаг 2 алгоритма), а все остальные проверки пропускаются.
   - Если проверка *i* дает **ложный результат**, то выполняется одно из трех действий:
     - Если следующая часть конструкции «if-elif-else» содержит оператор `elif`, то происходит переход к

следующей по порядку проверке (следующему условию) (проверка $i + 1$) (в начало шага 1 алгоритма).

- Если следующая часть конструкции «if-elif-else» содержит оператор `else` (в этой части нет проверки), то происходит переход к блоку инструкций для части `else` (блок $i + 1$) (шаг 2 алгоритма).
- Если следующей части конструкции «if-elif-else» нет, то алгоритм завершается. Ни один из блоков инструкций выполнен не будет.

2. Выполнение инструкций в блоке *i*: происходит выполнение всех инструкций в блоке, соответствующая проверка которого дала истинный результат. Алгоритм завершается.

Таким образом, конструкция «if-elif-else» может содержать произвольное число блоков кода (инструкций), однако выполнен будет только один блок.

Также допускается краткая запись в одну строку, если блок инструкций является небольшим и содержит, как правило одну простую инструкцию (не являющуюся составной):

```
if проверка: инструкция
```

4.2. Примеры использования конструкции «if-elif-else»

Примеры использования конструкции «if-elif-else» отражены в листингах 3.4, 3.5, 3.6, 3.7.

Для применения конструкции «if-elif-else» в самом простом виде достаточно прописать часть `if`, используя соответствующий оператор, выражение проверки, после которого идет двоеточие, переход на новую строку и отступ (листинг 3.7). Так как результат проверки является истинным, то в результате выполнения программы в листинге 3.6 в консоль было выведено сообщение «Yes» (выполнился блок кода на строке 3).

Листинг 3.7 – Пример использования конструкции «if-elif-else», состоящей только из части `if`

```
1 x = 10
2 if x == 10:
3     print('Yes')
```

```
Yes
```

Если бы результат проверки оказался ложным, в консоль не было бы выведено никакого сообщения. Для отслеживания подобной ситуации можно добавить часть `else` с соответствующим блоком кода (листинг 3.8).

Листинг 3.8 – Пример использования конструкции «if-elif-else», состоящей из части `if` и части `else`

```
x = 20
if x == 10:
    print('Yes')
else:
    print('No')
```

```
No
```

Для реализации множественного ветвления, т.е. для отслеживания множества случаев, каждый из которых требует соответствующей проверки и выполнения соответствующего блока кода, в конструкцию «if-elif-else» добавляются дополнительные части `elif` (листинг 3.9).

Листинг 3.9 – Пример использования конструкции «if-elif-else», состоящей из всех необязательных частей, реализующее множественное ветвление. В примере представлены также множественные условия с использованием оператора `and`

```
x = 20
if (x > 0) and (x <= 10):
    print('0 < x <= 10')
elif (x > 10) and (x <= 20):
    print('10 < x <= 20')
elif (x > 20) and (x <= 30):
    print('20 < x <= 30')
else:
    print('Error')
```

```
10 < x <= 20
```

При проверки сложных (множественных) условий могут применятся логические операторы `and`, `or`, `not`. При проверке вхождения некоторого числа *x* в определенный интервал, например (0; 10] допускается подобная запись: `0 < x <= 10` (листинг 3.10).

Листинг 3.10 – Пример использования конструкции «if-elif-else», состоящей из всех необязательных частей. В примере представлены условия для проверки на вхождение числа в интервал в виде смешанного выражения с двумя операторами и тремя операндами

```
x = 20
if 0 < x <= 10:
    print('0 < x <= 10')
elif 10 < x <= 20:
    print('10 < x <= 20')
elif 20 < x <= 30:
    print('20 < x <= 30')
else:
    print('Error')
```

```
10 < x <= 20
```

4.3. Блок инструкций в Python

Блок инструкций – часть кода, которая связана с одним из специальных операторов языка Python, таких как `if`, `elif`, `else`, `for`, `while`, `def`, `class` и др. Для выделения такой части кода используется отступ (табуляция).

Python определяет границы блоков автоматически, по отступам строк – т.е. по пустому пространству слева от кода. После строки заголовка блока все инструкции с отступами на одинаковое расстояние вправо принадлежат тому же самому блоку кода. Другими словами, операторы внутри блока выровнены по вертикали как в столбце [5].

Например, имеется следующий код (листинг 3.11).

Листинг 3.11 – Пример некоторой программы, состоящей из нескольких блоков кода

```
x = 10
list1 = [x, 15, 20, 25, 500]
if len(list1) >= 5:
    for i in range(len(list1)):
        print(list1[i])
    for i in range(3):
        print(list1[i])
else:
    print('Error')
y = x ** 2
```

На рисунке 3.3 изображена структура блоков кода, представленного в листинге 3.11.

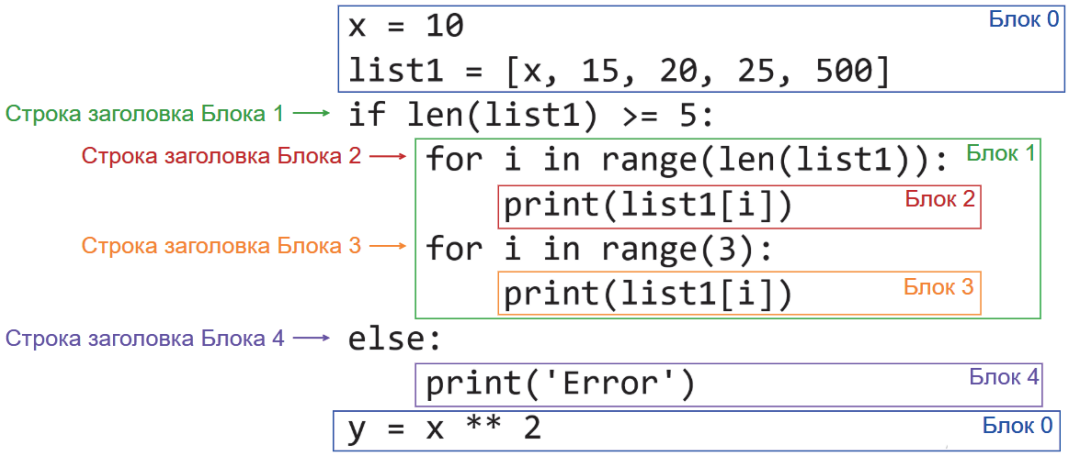


Рисунок 3.3 – Структура блоков программы из листинга 3.8

Источник: составлено на основе [5].

Вложенный блок начинается с оператора, смещенного дальше вправо, и заканчивается либо на операторе с меньшим отступом, либо в случае конца файла [5].

4.4. Проверка коллекции элементов на пустоту с помощью оператора `if`

Чтобы проверить, является ли коллекция (например, список или строка) пустой (не содержит ни одного элемента), можно использовать следующую короткую запись:

```
if имя_коллекции:
```

В случае списка полная запись может выглядеть следующим образом (листинг 3.12):

```
if имя_списка != []:
```

Листинг 3.12 – Пример проверки на пустоту двух списков `L1`, `L2`

```
L1 = [1, 2, 3]
L2 = []

if L1:
    print('L1 is not empty')
if L1 != []:
    print('L1 is not empty')
if L2:
    print('L2 is not empty')
if L2 != []:
    print('L2 is not empty')
```

```
L1 is not empty
L1 is not empty
```

В случае строки полная запись может выглядеть аналогично (листинг 3.13):

```
if имя_строки != '':
```

Листинг 3.13 – Пример проверки на пустоту двух строк `s1`, `s2`

```
1  s1 = 'abcd'
2  s2 = ''
3
4  if s1:
5      print('s1 is not empty')
6  if s1 != '':
7      print('s1 is not empty')
8  if s2:
9      print('s2 is not empty')
10 if s2 != '':
11     print('s2 is not empty')
```

```
s1 is not empty
s1 is not empty
```

Задачи

Задача № 1. Список

Создайте список ***list1***: 2, 3, 4, 2, 3, 2, 3, 4, 1, 1, 1, 5, 6, 6, 6.

1. Добавить в конец списка элементы 5, 7, 8.
2. Отсортировать список по возрастанию.
3. Выбрать элементы с 5 по 8 включительно.
4. Выбрать элементы с 3 по 10 не включительно.
5. Выбрать 3, 5 и 12 элементы.
6. Удалить элемент, стоящий на 4 месте в отсортированном списке.
7. На вторую позицию вставить значение 6.
8. Проверить есть ли значение 9 и 1 в списке.
9. Создать список ***list2*** как копию списка ***list1***.
10. Выстроить элементы списка ***list2*** в обратном порядке.
11. Удалить из ***list2*** последний элемент.
12. Добавить на 5 позицию значение 12 в список ***list2***.
13. Удалить 5 и 3 элементы из ***list2***.
14. Объединить списки ***list1*** и ***list2*** и вывести их 3 раза в консоль.
15. Определить, сколько раз в списке ***list2*** встречается значение 6.
16. Преобразовать список ***list2*** в строку, предусмотрев следующие разделители: «,», «_», «/».
17. Найти корень из каждого значения списка ***list2***.
18. Найти максимальное и минимальное значение списка ***list2***.
19. Создать список ***list3*** от 3 до 15.
20. Создать список ***list4*** от 20 до 3.

Задача № 2. Кортеж

1. Создайте кортеж ***c***: 4, 3, 11, 7, 2, 1, 100, 2, 4, 2.
2. Попробуйте добавить новый элемент в ***c***, удалить элемент из кортежа ***c***, изменить элемент с индексом 3.
3. Создайте кортеж, содержащий элементы от значения 10 до значения 555 с шагом 7.
4. Определите, сколько раз встречается «2» в кортеже ***c***, используя при этом только специальный метод.
5. Определите, сколько раз встречается «17» в кортеже ***c***, используя при этом только специальный метод.
6. Определите индекс элемента «100» в кортеже ***c***, используя при этом только специальный метод.
7. Создайте список и преобразуйте его в кортеж.
8. Преобразуйте кортеж ***c*** в список для его изменения, а затем обратно в кортеж. Используя функцию `list()`, добавьте новый элемент в ***c***, удалите элемент из кортежа ***c***, измените элемент с индексом 3. Объект ***c*** после выполнения пункта 7 должен быть кортежем.

Задача № 3. Условный оператор

1. Ввести любое число с клавиатуры. Составить программу, которая определяет принадлежность введенного числа к интервалу (-10, 10).
2. Ввести любые 3 числа с клавиатуры. Составить программу, которая определяет, какое из трех чисел наибольшее.
3. Ввести любое число с клавиатуры. Определить четное данное число или нечетное.
4. Ввести любые 2 числа с клавиатуры. Вычесть от большего меньшее и результат вывести на экран.

Задача № 4

Напишите программу, которая будет выводить все нечетные числа из диапазона от 39 до 248 и остановится, если встретится 139.

Задача № 5

Дан список ***list5*** = [11, 5, 8, 32, 15, 3, 20, 132, 21, 4, 555, 9, 20]. Необходимо вывести элементы, которые одновременно меньше 30 и делятся на 3 без остатка. Все остальные элементы списка необходимо просуммировать и вывести конечный результат.

Задача № 6

Выведите все числа от 0 до N, где

1. N = 66.
2. N = -31.

Задача № 7

Вывести на экран циклом пять строк из нулей, причем каждая строка должна быть пронумерована.

Задача № 8

Найти сумму ряда чисел от 1 до 100. Полученный результат вывести на экран.

Задача № 9

Дано семь чисел. Найти количество положительных чисел среди них:

1. 5964, -12, -68874, 101, -103, -741, 36985.
2. -713, -12563, -89, -45698, -898, -75632, -635.

Задача № 10

Даны три числа. Вывести на экран «**yes**», если среди них есть одинаковые, иначе вывести «**ERROR**»:

1. 956820, 956620, 936820.
2. 24930566, 24960566, 24930566.
3. 3496, 3496, 3496.

Задача № 11

Вывести на экран все чётные целые числа в диапазоне от 1 до 698.

Задача № 12

Посчитать сумму числового ряда:

1. от 0 до 14 *(пример: 0+1+2+3+…+14)*.
2. от 569 до 601.
3. от -65 до 12.

Задача № 13

Перемножить все нечётные целые числа в диапазоне от 0 до 84.

Задача № 14

Записать в массив все целые числа в диапазоне от 54 до 3945, кратные 5.

Задача № 15

Даны три числа. Вывести на экран «**yes**», если можно взять какие-то два из них и в сумме получить третье:

1. 9760, 3594, 6166.
2. 56783, 49998, 6784.

Задача № 16

Напишите программу, которая будет по номеру месяца выводить время года. Например, если введено 2, то следует вывести «Зима».

Задача № 17

Условие: дан текст «Алгоритм (от лат. написания арабского имени аль-Хорезми – Algorithmi), инструкция, точное описание способа действия с использованием простых, общепонятных элементов (например, операций). В математике понятие алгоритма сужается и уточняется следующим образом. Действие состоит в последовательности переходов от одного состояния вычисления (процесса работы алгоритма) к другому; состояния – это конструктивные объекты (например, слова в данном алфавите; в частности, целые числа в десятичной или двоичной записи). Алгоритм также является конструктивным объектом. Первое состояние называется исходным данным, последнее – результатом работы алгоритма. Фиксированный алгоритм можно применять к различным исходным данным; для некоторых он может не заканчивать работу. Тем самым алгоритм задаёт (возможно, не всюду определённую) функцию, вычисляемую этим алгоритмом. Такие функции называются вычислимыми. Понятия алгоритма и вычислимой функции относятся к исходным понятиям математики и через другие понятия не выражаются. Рассматриваются расширения понятия алгоритма, например вероятностные алгоритмы, т. н. алгоритмы с оракулом, алгоритмы взаимодействия с окружающей средой, параллельные алгоритмы. Часто алгоритм определяется с помощью абстрактной вычислительной машины, получающей на вход программу действия и исходное данное. До конца 19 в. алгоритм – общее понятие, относящееся к известным алгоритмам, таким как алгоритм выполнения арифметических операций в десятичной системе счисления, алгоритм дифференцирования функций, алгоритм Евклида нахождения общей меры отрезков или наибольшего общего делителя многочленов. В 1900–1910-х гг. были осознаны трудности в построении общего алгоритма решения некоторых массовых проблем. В 1930-е гг. предложены математические определения понятия вычислимой функции, исходящие из представлений о том, что может делать человек-вычислитель; среди них – понятие рекурсивной функции и понятие функции, вычислимой машиной Тьюринга. Тогда же была доказана эквивалентность различных понятий вычислимой функции и классов вычислимых функций, порождаемых этими понятиями; сформулирован т. н. тезис Чёрча, принятый в качестве естественно-научного факта: класс вычислимых функций совпадает с любым из упомянутых выше классов. Развитие компьютерных технологий не изменило представлений о классе функций, вычисляемых алгоритмами. Построение и анализ конкретных алгоритмов, предназначенных для выполнения компьютером, относится к программированию. Выделяются также классы вычислительных алгоритмов и обучающихся алгоритмов» [10].

Требуется:

1. Очистить представленный выше текст от знаков препинания.
2. Разбить текст на слова и создать список слов под именем L1 (допускается дублирование слов).
3. Найти индекс определенного слова.
4. Определить, сколько раз слово встречается в списке.
5. Добавить 5 слов в список (в конец).
6. Добавить 5 слов в список (в начало).
7. Сделать копию списка под именем L2.
8. Удалить из списка L2 каждое третье слово.
9. Удалить из списка L2 первое слово.
10. Удалить из списка L2 определенное слово.
11. Создать список L3 из списка L2, повторенного 4 раза.
12. Очистить список L2.
13. Добавить в список L2 несколько слов.
14. Удалить из списка L3 любые 10 слов, которые расположены подряд.

Задача № 18

1. Вывести каждое слово списка L1 в консоль.
2. Для п.1 добавить индекс слова, т.е. чтобы помимо слова в консоль выводился и его индекс.
3. Создать список A, который будет содержать размеры каждого слова в списке L1. Например, если первое слово списка L1 состоит из 10 символов, то первый элемент списка A будет иметь значение 10.
4. Определить размер четвертого слова в списке L1, используя при этом только список A.
5. Создать список B, который будет содержать только первые буквы каждого слова из списка L1. Например, если первое слово списка L1 «Монитор», то первый элемент списка B будет иметь значение «М».
6. Создать список C, как объединение пар элементов из двух списков A, B.
7. Объединить все элементы списка B в одну строку (один текст).

Задача № 19

1. Создать список L4 как копию списка L1. Удалить из списка L4 все слова, состоящие из 1 символа.
2. Удалить из списка L4 все слова, состоящие из 2 или 3 символов.
3. Создать список D, который будет содержать слова из списка L1, имеющие 5 или меньше символов.
4. Создать список E, который будет содержать слова из списка L1, имеющие 5 или больше символов, но не более 10 символов.
5. Создать список F, который будет содержать все остальные слова из списка L1, не вошедшие в список D или E.

Задача № 20*

Создать список предложений из текста. Определить число предложений в тексте.

Задача № 21*

Вывести последовательность Фибоначчи, где каждое следующее число равно сумме двух предыдущих (1, 1, 2, 3, 5, 8, 13, 21 ...), пока новое значение не превысит 444.

Задача № 22*

В первый день спортсмен пробежал 10 км, каждый следующий день он увеличивал дистанцию на 20% от дистанции предыдущего дня. Определите, после какого дня суммарный пробег за все дни превысит 500 км и выведите этот пробег.

Задача № 23*

Найдите все трёхзначные и четырёхзначные числа Армстронга. Числом Армстронга считается натуральное число, сумма цифр которого, возведенных в N-ную степень (N – количество цифр в числе) равна самому числу.

Например, $153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27$.

Практическое задание № 4
«Циклы while, for в Python»

Циклический оператор позволяет реализовать многократное повторение однотипных действий, таких как перебор элементов коллекции или последовательности.

В Python существует два основных циклических оператора:

- `while`
- `for`

Оператор `while` используется для написания универсальных циклов.

Оператор `for` используется для реализации цикла, использующегося, как правило, для прохода по определенным элементам в последовательности или в другом итерируемом объекте (например, строке, списке и т.д.) и выполнения блока кода для каждого взятого элемента [5].

1. Цикл `while`

Применение оператора `while` является наиболее универсальным способом реализовать цикл в языке Python.

1.1. Формат цикла `while`

Конструкция `while` состоит из:

- **строки заголовка** с выражением проверки, которая проверяется на каждой итерации цикла: если проверка дает истинный результат, происходит переход в тело цикла, после выполнения инструкций в теле цикла происходит переход опять на строку заголовка;
- **тела цикла**, представляющего собой отдельный блок инструкций, выполняющийся на каждой итерации цикла;
- необязательной части `else`, блок инструкций которой выполняется, если управление покидает цикл (проверка цикла вернула ложный результат) и оператор `break` не встретился (описание работы оператора `break` представлено в таблице 4.1).

Общий формат цикла `while` представлен на рисунке 4.1.

```
1 while проверка:          # - Проверка цикла
2     блок_инструкций_1    # - Тело цикла
3 else:                    # - Необязательная часть else
4     блок_инструкций_2    # - Выполняется, если не
5                          #   произведен выход из
6                          #   тела цикла с помощью
7                          #   оператора break
```

Рисунок 4.1 – Общая форма записи при реализации цикла `while`

Наиболее простая форма конструкции `while` включает две обязательные части: строку заголовка и тело цикла (рисунок 4.2).

```
1 while проверка:          # - Проверка цикла
2     блок_инструкций_1    # - Тело цикла
3 else:                    # - Необязательная часть else
4     блок_инструкций_2    # - Выполняется, если не
5                          #   произведен выход из
6                          #   тела цикла с помощью
7                          #   оператора break
```

Рисунок 4.2 – Наиболее простая форма записи при реализации цикла `while`

Оператор `while` выполняет инструкции в теле цикла многократно на каждой итерации, пока выполняется условие в строке заголовка. Как только условие перестанет выполняться, произойдет переход на следующий блок кода.

Например, в листинге 4.1 при переходе на строку заголовка (строка 2) проверка `a < 0` выполняется (т.к. `a = -5`), поэтому происходит переход в тело цикла и начинает выполнятся первая итерация цикла (строки 3, 4), где в том числе увеличивается значение переменной `a` на единицу. После этого происходит переход опять на строку заголовка, и проверка опять выполняется (т.к. `a = -4`), далее происходит переход опять к телу цикла и выполняется уже вторая итерация цикла. Так происходит до тех пор, пока значение переменной `a` не станет равняться `-5` – в таком случае проверка не будет выполнена и произойдет выход из тела цикла на строку 5.

Листинг 4.1 – Пример использования цикла `while`: было выполнено 5 итераций цикла, прежде чем проверка дала ложный результат и произошел выход из цикла

```
1 a = -5
2 while a < 0:
3     print(a, end=' ')
4     a += 1
5 print('\nКонец цикла while')
```

```
-5 -4 -3 -2 -1
Конец цикла while
```

Если условие в строке заголовка изначально не выполняется, то переход в тело цикла не будет произведен, т.е. не будет выполнена ни одна итерация цикла (листинг 4.2).

Листинг 4.2 – Пример использования цикла `while`: при первом переходе к строке заголовка цикла проверка дала ложный результат, поэтому ни одна итерация цикла не была выполнена (со строки 2 произошел переход сразу на строку 5)

```
1 a = -5
2 while a > 0:
3     print(a, end=' ')
4     a += 1
5 print('\nКонец цикла while')
```

```
-5 -4 -3 -2 -1
Конец цикла while
```

1.2. Операторы `break`, `continue`, `pass` и конструкция `else` цикла

В цикле `while`, как и в цикле `for`, могут быть использованы специальные операторы цикла и конструкция `else` цикла (таблица 4.1).

Таблица 4.1 – Операторы break, continue, pass и конструкция else цикла

Оператор / конструкция	**Описание**	**Примеры для цикла `while` (для цикла `for` операторы и конструкция работают аналогично)**

`break`	Переходит за пределы ближайшего заключающего цикла (после всего оператора цикла).	`a = 10`
		`while a > 0:`
		`  if a == 5:`
		`  break`
		`  print(a, end=' ')`
		`  a -= 1`
		
		`10 9 8 7 6`

`continue`	Переходит в начало ближайшего заключающего цикла (на строку заголовка цикла).	`a = 10`
		`while a > 0:`
		`  if a == 5:`
		`  a -= 1`
		`  continue`
		`  print(a, end=' ')`
		`  a -= 1`
		
		`10 9 8 7 6 4 3 2 1`

`pass`	Ничего не делает: является пустым оператором-заполнителем, которое обозначает отсутствие действий. Используется в ситуациях, когда синтаксис требует какой-то инструкции, например в теле цикла или условия, но отсутствует необходимость выполнять какую-либо инструкцию.	`while True:`
		`  pass`
		
		`a = 10`
		`while a > 0:`
		`  if a == 5:`
		`  pass`
		`  else:`
		`  break`

Блок `else`	Выполняется тогда и только	`a = 10`

Оператор / конструкция	Описание	Примеры для цикла `while` (для цикла `for` операторы и конструкция работают аналогично)

цикла	тогда, когда происходит нормальный выход из цикла т.е.:
	• проверка в строке заголовка цикла дала ложный результат
	ИЛИ
	• в теле цикла не был выполнен оператор `break`
		`while a > 100:`
		`  print(a, end=' ')`
		`  a -= 1`
		`else:`
		`  print('End')`
		
		`End`
		
		`a = 10`
		`while a > 0:`
		`  if a == 50:`
		`  break`
		`  print(a, end=' ')`
		`  a -= 1`
		`else:`
		`  print('End')`
		
		`10 9 8 7 6 5 4 3 2 1 End`

Источник: составлено на основе: [5, 14].

С учетом операторов `break` и `continue` общий формат цикла `while` выглядит следующим образом (рисунок 4.3).

```
while проверка:
    блок_инструкций_1
    if проверка: break    # Выход из цикла с пропуском else
    if проверка: continue # Переход на проверку в начало цикла
else:
    блок_инструкций_2     # Выполняется, если не было break
```

Рисунок 4.3 – Общая форма записи при реализации цикла `while` с учетом операторов `break` и `continue`

Операторы `break` и `continue` могут появляться в любом месте в теле цикла (`while` или `for`), но обычно их записывают в связке с оператором `if` для выхода из цикла или перехода на следующую итерацию при определенном условии.

1.3. Бесконечный цикл с помощью `while` и `break`

Оператор `while` способен порождать так называемый бесконечный цикл. Для этого необходимо, чтобы проверка выполнялась всегда.

Например, если модифицировать код листинга 4.1 так, чтобы не происходило изменение переменной `a`, то условие в заголовке цикла будет выполняться всегда и произойдет бесконечное зацикливание (листинг 4.3).

Примечание: для принудительного завершения цикла необходимо нажать комбинацию клавиш «Ctrl + C».

Листинг 4.3 – Пример использования цикла `while`: условие в строке заголовка выполняется всегда, что привело к бесконечному зацикливанию

```
a = -5
while a < 0:
    print(a, end=' ')
print('\nКонец цикла while')
```

```
-5 -5 -5 -5 -5 -5 -5 -5 -5
-5 -5 -5 -5 -5 -5 -5 -5 -5
-5 -5 -5 -5 -5 -5 -5 -5 -5
-5 -5 -5 -5 -5 -5 -5 -5 -5
            ...
```

Подобное бесконечное зацикливание, как правило, возникает непреднамеренно из-за ошибок в программном коде и опасно, т.к. оно может привести к зависанию компьютера, особенно, если в теле цикла содержаться инструкции, выполнение которых требует значительных вычислительных мощностей.

Однако иногда бесконечный цикл может быть полезен. Например, с помощью него можно реализовать диалоговый интерфейс с пользователем при помощи функции `input()` для ожидания ввода пользователем информации и оператора `break` для завершения диалога и выхода из цикла (листинг 4.4).

Листинг 4.4 – Пример использования цикла `while` для реализации командного диалога с пользователем: цикл выполняется бесконечно, пока пользователь не введет число 5

```
while True:
    a = int(input('Введите число: '))
    if a == 5:
        print('Выход из цикла')
        break
```

В листинге 4.4 проверка выполняется всегда, т.к. `True` является специальной версией целого числа 1 и всегда обозначает истинное значение [5].

2. Цикл for

Цикл `for` является универсальным итератором в Python – с помощью него можно реализовать проход по каждому элементу (или по определенным элементам) в любом итерируемом объекте, в тои числе в упорядоченной последовательности (например, строке, списке, кортеже и т.д.) [5].

Общая форма записи при реализации цикла `for` представлена на рисунке 4.4 и является идентичной форме цикла `while`, за исключением строки заголовка.

```
for цель in объект:     # Присваивает цели элементы объекта
    блок_инструкций_1   # Повторяемое тело цикла: использует цель
else:                   # Необязательная часть else
    блок_инструкций_2   # Выполняется, если не было break
```

Рисунок 4.4 – Общая форма записи при реализации цикла `for`

В строке заголовка цикла `for` после соответствующего ключевого слова `for` указывается цель (или цели) присваивания. Цель представляет собой переменную, которой присваивается значение элемента объекта (например, списка), по которому происходит проход. Далее указывает ключевое слово `in` и сам итерируемый объект.

После заголовка находится тело цикла с блоком инструкций, выполнение которых повторяется на каждой итерации. При этом обычно в теле цикла расположены инструкции для обработки цели (переменной), т.е. определенного элемента. В наиболее простой форме цикл `for` выглядит следующим образом (рисунок 4.5.)

```
for цель in объект:     # Присваивает цели элементы объекта
    блок_инструкций_1   # Повторяемое тело цикла: использует цель
else:                   # Необязательная часть else
    блок_инструкций_1   # Выполняется, если не было break
```

Рисунок 4.5 – Наиболее простая форма записи при реализации цикла `for`

Так же, как и в цикле while, в цикле for могут быть использованы операторы `break` и `continue` (рисунок 4.6), а также конструкция `else`, блок инструкций которой выполняется после прохода по всем элементам (или по определенным элементам) итерируемого объекта и в случае, если оператор `break` не был выполнен.

```
for цель in объект:       # Присваивает цели элементы объекта
    блок_инструкций_1     # Повторяемое тело цикла: использует цель
    if проверка: break    # Выход из цикла с пропуском else
    if проверка: continue # Переход на проверку в начало цикла
else:                     # Необязательная часть else
    блок_инструкций_2     # Выполняется, если не было break
```

Рисунок 4.6 – Общая форма записи при реализации цикла `for` с учетом операторов `break` и `continue`

В листинге 4.5 представлен пример использование цикла `for`, где в теле цикла происходит обработка каждого элемента списка.

Листинг 4.5 – Пример использования цикла `for` для прохода по элементам списка и умножения каждого элемента на 10

```
L1 = [1, 2, 3, 4, 5]

for el in L1:
    print(el * 10, end=' ')
else:
    print('End')
```

```
10 20 30 40 50 End
```

Цикл `for` часто используется в связке с функцией `range()`, которая возвращает диапазон в виде итерируемого объекта. Такой диапазон можно использовать для индексов элементов коллекции.

Например, для прохода по элементам строки с четными индексами можно в строке заголовка использовать выражение `range(0, 10, 2)` для создания диапазона от 0 до 10 с шагом 2, при этом переменной (например, с именем `i`) будут присваиваться элементы этого диапазона, т.е. 0, 2, 4, 6, 8 (10 не входит в диапазон) (листинг 4.6).

Листинг 4.6 – Пример использования цикла `for` и функции `range()` для прохода по элементам строки с четными индексами

```
s1 = 'AaBbCcDdEe'

for i in range(0, len(s1), 2):
    print(s1[i], end=' ')
else:
    print('End')
```

```
A B C D E End
```

Для получения элементов строки с нечетными индексами необходимо использовать диапазон от 1 до 10 с шагом 2 (листинг 4.7).

Листинг 4.7 – Пример использования цикла `for` и функции `range()` для прохода по элементам строки с нечетными индексами

```
s1 = 'AaBbCcDdEe'

for i in range(1, len(s1), 2):
    print(s1[i], end=' ')
else:
    print('End')
```

```
a b c d e End
```

Осуществить проход по элементам итерируемого объекта можно и с использованием цикла while. Однако такой подход менее удобен и требует большего числа строк кода (листинг 4.8).

Листинг 4.8 – Пример использования цикла `while` для прохода по элементам строки с нечетными индексами

```
s1 = 'AaBbCcDdEe'

i = 1
while i < len(s1):
    print(s1[i], end=' ')
    i += 2
else:
    print('End')
```

```
a b c d e End
```

Задачи

Задача № 1. Проход по элементам списка с помощью циклов `for`, `while`

Условие: имеется список L, состоящий из элементов 10, 15, 6, 13, 4, 2, 1, 2, 3.

Выполните следующие пункты задачи, предоставив ответ в двух вариантах:

1. Поочередно выведете все элементы списка L в консоль, используя:
   a. цикл `for`
   b. цикл `while`
2. Для каждого элемента списка прибавьте число 10 и выведете результат в консоль, используя:
   a. цикл `for`
   b. цикл `while`
3. Поочередно выводите все элементы списка L в консоль до тех пор, пока не встретится число 2, используя:
   a. цикл `for`
   b. цикл `while`
4. Поочередно выведете все элементы списка L, имеющие нечетные индексы в консоль, используя:
   a. цикл `for`
   b. цикл `while`

Задача № 2. Проход по элементам строки с помощью циклов `for`, `while`

Условие: имеется строка s, содержащая следующий текст: «Algorithmization and programming».

Выполните следующие пункты задачи, предоставив ответ в двух вариантах:

1. Поочередно выведете все элементы строки s в консоль, используя:
    a. цикл `for`
    b. цикл `while`
2. Поочередно выведете все буквы «а» строки s в консоль вместе с их индексами, используя:
    a. цикл `for`
    b. цикл `while`
3. Поочередно выведете все гласные буквы строки s в консоль вместе с их индексами, используя:
    a. цикл `for`
    b. цикл `while`

Задача № 3

Разработайте программу, которая в заданном списке имен находит самое длинное имя. Воспользуйтесь для этой цели циклом `for`.

Определите, как поведет себя алгоритм, предложенный вами в качестве решения, если «самых длинных» имен в списке будет несколько. В частности, как поведет себя ваш алгоритм, если все имена в списке будут одной длины?

Задача № 4

Модифицируйте программу из Задачи № 3 так, чтобы она находила самое длинное имя и самое короткое имя в заданном списке имен. Воспользуйтесь для этой цели так же циклом `for`.

Задача № 3

Разработайте программу, которая в заданном списке из пяти или более чисел находит 5 наименьших и 5 наибольших чисел.

Задача № 4

Используя только цикл `while` произведите подсчет элементов от значения *a* до значения *b* отсортированного списка целых чисел (произведите подсчет в диапазоне от *a* до *b*). При этом *a* и *b* могут не являться элементами списка.

Например, если имеется следующий список `L1` и значения *a* = 2, *b* = 30, программа должна дать ответ 5:

```
>>> L1 = [1, 2, 3, 10, 15, 20, 40, 100]
>>> a = 2
>>> b = 30
5
```

Задача № 5

Используя только цикл `for` решите Задачу № 4: произведите подсчет элементов от значения *a* до значения *b* отсортированного списка целых чисел (произведите подсчет в диапазоне от *a* до *b*). При этом *a* и *b* могут не являться элементами списка.

Практическое задание № 5
«Функции в Python»

1. Функции: общие сведения

Функция в Python – объект, который принимает аргументы и возвращает значения.

Функция является способом группирования набора операторов, позволяющим выполнять их более одного раза в программе

Функция – процедура, вызываемая по имени.

Функции способны вычислять результирующее значение и также дают возможность указывать параметры, которые служат входными данными функции и могут отличаться при каждом выполнении кода.

Функции позволяют разбивать сложные системы на поддающиеся управлению части. За счет реализации каждой части в виде функции мы делаем ее многократно применяемой и легкой для кодирования.

Причины использовать функции:

- **Доведение до максимума многократного использования кода и сведение к минимуму избыточности**
  Функции позволяют группировать и обобщать код для использования произвольно много раз в более позднее время. Позволяют сократить избыточность кода в программах и посредством этого уменьшить объем работ по сопровождению.
- **Процедурная декомпозиция**
  Функции предлагают инструмент для разбиения систем на части с хорошо определенными ролями, позволяют реализовать небольшие подзадачи обособленно, вместо реализации сразу полного процесса. В общем случае функции связаны с процедурой, которая описывает, как делать что-либо, а не то, в отношении чего делать [5].

2. Создание и вызов функций

2.1. Создание функции

Чтобы **создать функцию**, можно использовать следующую запись (рисунок 5.1).

```
def имя_функции(аргумент_1, аргумент_2, … , аргумент_n):
    тело_функции
    return возвращаемое_значение
```

Рисунок 5.1 – Общая форма записи при создании (определении) новой функции

В строке заголовка `def` указывается:

- **имя функции**, которому присваивается объект функции,

- список из нуля и более **аргументов** (также называемых параметрами) в круглых скобках. Именам аргументов в заголовке присваиваются объекты, передаваемые в круглых скобках в точке вызова [5].

Тело функции содержит блок инструкций, который выполняется каждый раз, когда функция вызывается.

Функция может не принимать ни одного аргумента и не возвращать ни одного значения (рисунок 5.2).

```
def имя_функции():
    тело_функции
```

Рисунок 5.2 – Форма записи при создании (определении) новой функции без аргументов и без возвращаемого значения

2.2. Вызов функции

Для **вызова функции** необходимо указать имя функции и круглые скобки, в которых через запятую перечислить все аргументы (значения для аргументов), принимаемые функцией:

```
имя_функции(аргумент_1 = значение_для_аргумента_1, …, аргумент_n = значение_для_аргумента_n,)
```

или

```
имя_функции(значение_для_аргумента_1, …, значение_для_аргумента_n)
```

Если таких аргументов нет, то скобки необходимо оставить пустыми:

```
имя_функции()
```

Пример создания функции представлен в листинге 5.1.

Листинг 5.1 – Пример определения и вызова функции суммирования двух чисел `get_sum()`

```
1 def get_sum(x, y):
2     res = x + y
3     return res
4
5 result = get_sum(x=10, y=15)
6 print(result)
```

```
25
```

В строках 1-3 происходит **определение функции**, в строке 5 происходит **вызов функции**, при этом указываются аргументы (значения 10,

15) в круглых скобках, подлежащие передаче (присваиванию) именам в **заголовке функции** (именам x, y) (строка 11).

2.3. Передача аргументов при вызове функции

Передача аргументов при вызове функции осуществляется по следующим трем основным правилам.

Правило № 1. Аргументы передаются по присваиванию (по ссылкам на объекты).

Передача аргументов на вход функции может осуществляться через оператор присваивания = (т.е. происходит создание ссылки на объект).

Например, в листинге 5.1 имени x в заголовке функции (строка 1) присваивается значение 10 (строка 5), имени y в заголовке функции (строка 1) присваивается значение переменной 15 (строка 5) и тело функции выполняется.

При этом присваивание значения имени в заголовке функции может происходить через дополнительную переменную (листинг 5.2).

Листинг 5.2 – Пример определения и вызова функции суммирования двух чисел get_sum(). При вызове именам из заголовка функции (x, y) присваиваются значения имен из вызывающего кода (v1, v2)

```
1 def get_sum(x, y):
2     res = x + y
3     return res
4 
5 v1 = 10
6 v2 = 15
7 
8 result = get_sum(x=v1, y=v2)
9 print(result)
```

```
25
```

В листинге 5.2 имени x в заголовке функции (строка 1) присваивается значение переменной v1 (строка 8), имени аргумента y в заголовке функции (строка 1) присваивается значение переменной v2 (строка 8) и тело функции выполняется.

Правило № 2. Аргументы передаются по позиции, если только не указано иначе.

Значения, которые передаются на вход функции при ее вызове, по умолчанию сопоставляются с именами аргументов в определении (строке заголовка) функции слева направо, при этом не обязательно использовать оператор присваивания (листинг 5.3).

Листинг 5.3 – Пример определения и вызова функции суммирования двух чисел `get_sum()`. При вызове именам из заголовка функции (`x`, `y`) присваиваются значения имен из вызывающего кода (`v1`, `v2`) без использования оператора `=`, с учетом порядка следования аргументов

```
def get_sum(x, y):
    res = x + y
    return res

v1 = 10
v2 = 15

result = get_sum(v1, v2)
print(result)
```

```
25
```

При вызове функций можно распаковывать произвольно много аргументов для отправки посредством добавления звездочек * или ** перед именами аргументов: `*позиционные_аргурменты` и `**ключевые_аргурменты` (** используются для работы со словарями) (листинг 5.4).

Листинг 5.4 – Пример определения и вызова функции суммирования двух чисел `get_sum()`. При вызове именам из заголовка функции (`x`, `y`) присваиваются значения элементов списка `var_list` (10, 15) с помощью процедуры распаковки на основе оператора *

```
def get_sum(x, y):
    res = x + y
    return res

var_list = [10, 15]

result = get_sum(*var_list)
print(result)
```

```
25
```

При определении функции для реализации сбора произвольного числа аргументов можно воспользоваться оператором * упаковки всех переданных значений в одну коллекцию (листинг 5.5).

Листинг 5.5 – Пример определения и вызова функции суммирования двух чисел `get_sum()`. При вызове функции имени `args` из заголовка функции присваивается коллекция из значений переменных v1, v2 (10, 15), упакованных с помощью оператора *

```
def get_sum(*args):
    res = args[0] + args[1]
    return res

v1 = 10
v2 = 15

result = get_sum(v1, v2)
print(result)
```

```
25 25
```

Для указания стандартных значений аргументов (значений по умолчанию) используется оператор присваивания `=` в строке заголовка функции. В таком случае, если при вызове функции не указывается значение для такого аргумента, то аргументу присваивается значение, заданное по умолчанию (листинг 5.6).

Листинг 5.6 – Пример определения и вызова функции суммирования двух чисел `get_sum()`. При вызове функции аргументу `x` присваивается значение имени из вызывающего кода (`v1`), аргументу `y` значение по умолчанию – `1000`

```
def get_sum(x=0.1, y=1000):
    res = x + y
    return res

v1 = 10
v2 = 15

result = get_sum(v1)
print(result)
```

```
1010
```

Правило № 3. Аргументы, возвращаемые значения и переменные не объявляются.

В функции можно передавать аргументы любого типа и возвращать объект любого типа. Поэтому вызов функции может применяться к разнообразным типам объектов, которые поддерживают совместимый

интерфейс (интерфейс означает набор методов и операций выражений, выполняемых кодом функции) независимо от их специфических типов [5]. Такая особенность функций называется **полиморфизмом**.

Полиморфизм функции – способность одной и той же функции по-разному обрабатывать объекты разных типов (менять свое поведение в зависимости от типа объекта).

Пример работы функции с разными типами представлен в листинге 5.7.

Листинг 5.7 – Пример определения и вызова функции `increase()`. При вызове функции и передаче в качестве аргумента `a` целого числа 2 (строка 5) выполняется операция умножения `a * b`, однако при передаче в качестве аргумента `a` списка `[2, 2, 2]` (строка 6) выполняется операция повторения – список `a` повторяется `b` раз

```
1 def increase(a, b):
2     res = a * b
3     return res
4
5 print(increase(2, 5))
6 print(increase([2, 2, 2], 5))
```

```
10
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

2.4. Приостановка кода при вызове функции

Вызов функции в коде приводит к его **приостановке** на время выполнения этой функции.

Например, в листинге 5.1 на строке 20 при вызове функции `sum_1()` происходит приостановка кода, т.е. код, который расположен ниже не будет выполняться, пока вызываемая функция `sum_1()` не закончит свою работу, т.е. не возвратит объект `result`. Только после этого произойдет возвращение управления в вызывающий код (на строку 20) и переход на следующую строку (строка 21).

3. Области видимости: глобальные и локальные переменные

С понятием «функция» связаны понятия «глобальная переменная» и «локальная переменная».

Локальная переменная (Local variable) – имя (переменная), которое видно только в коде функции (в строке заголовка и в теле функции) и которое существует только при вызове функции (во время выполнения кода функции). После выполнения кода функции, все локальные переменные этой функции перестают существовать.

Глобальная переменная (Global variable) – имя (переменная), которое видно во всем файле.

Например, в листинге 5.1 переменные `v1`, `v2`, `res` являются глобальными – к ним можно обратиться из любой части программы. Переменная `result` существует только при вызове функции `sum_1()`, поэтому к ней можно обратиться только в теле функции `sum_1()`.

Существует правило: по умолчанию все имена (переменные), которым выполнено присваивание внутри функции, ассоциируются с пространством имен данной функции и никаким другим [5].

Переменные могут присваиваться в трех местах:

1. **Локальная переменная** – переменная, которая присваивается внутри оператора `def`.
2. **Нелокальная переменная** – переменная, которая присваивается внутри объемлющего оператора `def` (объемлющей функции, т.е. включающей в себя создание другой функции – подфункции).
3. **Глобальная переменная** – переменная, которая присваивается за пределами всех операторов `def`.

Пример трех типов переменных представлен в листинге 5.8.

Листинг 5.8 – Пример трех типов переменных: глобальная переменная `a`, нелокальная переменная `b` и локальная переменная `c`

```
1 a = 5
2 def func1():
3     b = 50
4     def func2():
5         c = 500
```

Существование понятия **«нелокальная переменная»** связано с возможностью реализовать вложенность функций, т.е. в теле функции может быть применен оператор `def` для создания функции (строка 4 листинга 5.8). Тогда к переменным, объявленным в теле основной функции, можно обратиться из подфункции (например, в листинге 5.8 в теле функции `func2()` можно обратиться к переменной `b`, которая будет являться нелокальной по отношению к функции `func2()`).

4. Операторы, использующиеся при создании функций

Основные операторы, использующиеся при создании функций приведены в таблице 5.1.

Таблица 5.1 – Основные операторы при создании функций

Оператор	Описание	Пример команды

`def`	Создает объект функции и присваивает его имени (имени функции).	`def show_text(text):`
		`  print(f'Hello, {text}!')`
		
		`show_text('World')`

`return`	Возвращает некоторый результат (в виде объекта) коду, который вызывает функцию.	`def get_sqrt(x):`
		`  return x ** (1/2)`
		
		`a = get_sqrt(100)`

Оператор	Описание	Пример команды
	Свойства: • `return` может быть расположен в любом месте в теле функции; • `return` без значения просто возвращает управление в вызывающий код (и отправляет в таком случае результат `None`); • `return` является необязательным: если он отсутствует, то выход из функции происходит, когда поток управления достигает конца тела функции (когда весь код тела функции выполнен).	

`global`	Присваивает переменной внутри функции значение из включающего модуля (значение глобальной переменной). Другими словами, позволяет внутри функции выполнять присваивание глобальным переменным, т.к. по умолчанию при присваивании внутри функции переменная ассоциируется только с пространством имен внутри этой функции, даже если имя такой переменной совпадает с именем глобальной переменной.
	В примере в соседней колонке существует лишь одна глобальная переменная name. Если убрать оператор `gloabal`*, то будут существовать две переменные с одинаковым именем: глобальная переменная* `name` *и локальная переменная* `name`.
		`name = 'Tom'`
		`def change_name():`
		`  global name`
		`  name = 'Bob'`
		`change_name()`

`nonlocal`	Действует схожим образом с оператором `global`. Присваивает переменной внутри функции значение из объемлющей функции (значение нелокальной переменной).
	Позволяет объемлющим функциям служить местом сохранения состояния, т.е. информации, запоминаемой между вызовами функции, без потребности в применении глобальных переменных.
		`def change_info():`
		`  name = 'Tom'`
		
		`  def change_name():`
		`  nonlocal name`
		`  name = 'Bob'`

`lambda`	Выражение, которое позволяет встраивать определения функций в места, где оператор `def` синтаксически не допускается.
	В общем виде выражение `lambda` выглядит следующим образом:
	
	`lambda a1, …, an: выражение`
		`sum1 = lambda x, y: x + y`
		`print(sum1(10, 1))`
		
		`funcs = [lambda x: x ** 2,`
		`  lambda x: x ** 3]`
		`print(funcs[0](10))`
		`print(funcs[1](0.57))`

Оператор	Описание	Пример команды
	где a1, …, an – аргументы	

yield	В некоторой степени действует схожим образом с оператором return. Но при возвращении объекта вызывающему коду запоминает место, где он остановился и может продолжить выполнение кода. Используется в функциях-генераторах для оптимизации вычислений: позволяет получать промежуточные результаты (например, получение отдельного числа) и сразу их обрабатывать, вместо ожидания итоговых результатов (например, получения всего массива чисел).	def generate_5_values(n):
		  for i in range(n):
		  if i == 5:
		  return
		  else:
		  yield i
		
		for value in generate_5_values(10):
		  print(value, end=' ')
		
		0 1 2 3 4

pass	Используется для временного заполнения тела функции, где требуется какая-либо инструкция. Как правило, предполагается, что pass будет заменен на другие инструкции позже при написании кода. Вместо pass допускается использование троеточия …	def func1():
		  pass
		
		def func2():
		  …

Источник: составлено на основе [5].

Задачи

Задача № 1. Основы функций

1. Создайте функцию суммирования двух чисел.
2. Создайте функцию для конкатенации двух строк.
3. Создайте функцию перемножения двух чисел.
4. Создайте функцию для возведения числа в степень, не используя при этом оператор **.
5. Создайте функцию, реализующую 7 любых арифметических операций. Для выбора операции предусмотреть специальный аргумент. Пример: если `arg1=='+'`, то выполняется сложение.
6. Создайте функцию, которая будет выводить в консоль текст *n* раз, где текст и *n* задается пользователем.
7. Создайте функцию для определения числа элементов в списке, не используя встроенную функцию `len()`.
8. Создайте функцию, которая на вход может получать произвольное количество аргументов и возвращает сумму переданных ей на вход чисел.

   Пример:

```
In: func(2, 4, 1, 1)
Out: 8
In: func(4, 10)
Out: 14
```

9. Создайте глобальную переменную PATH, значение которой соответствует пути к некоторому каталогу «C:\Downloads\Projects\Data». Создайте функцию, которая на вход получает имя файла и изменяет значение глобальной переменной PATH, добавляя в конец пути обратную косую черту и имя файла.
   Пример:
   ```
   In: func('data1.txt')
   Out: 'C:\Downloads\Projects\Data\data1.txt'
   In: func('table.xlsx')
   Out: 'C:\Downloads\Projects\Data\table.txt'
   ```

Задача № 2. Лямбда-функции

1. Создайте лямбда-функцию для суммирования двух элементов.
2. Создайте лямбда-функцию для получения возведения в квадрат переданного на вход числа.
3. Создайте лямбда-функцию, которая на вход получает 4 числа и возвращает результат их перемножения.
4. Создайте лямбда-функцию, которая на вход получает 1 число и возвращает результаты возведения этого числа в степень 2, степень 3 и степень 4.
   Пример:
   ```
   In: func(2)
   Out: [4, 8, 16]
   ```
5. Создайте лямбда-функцию, которая на вход может получать произвольное число аргументов и возвращает число переданных ей на вход аргументов.
   Пример:
   ```
   In: func(1, 2, 10, 20)
   Out: 4
   ```

Задача № 3

Возьмите текст из условия к задаче № 16 практического задания № 3.

1. Создайте функцию для очистки текста от знаков препинания.
2. Создайте функцию для подсчета числа слов в тексте без знаков препинания.
3. Создайте функцию для подсчета частоты встречаемости каждого слова, которая будет возвращать список слов и список частот. Например, если первое слово встречается 2 раза, то во втором списке первый элемент будет равен 2. Функция должна в консоли выводить указанные два списка в виде таблицы.
4. Создайте функцию для поиска слова в тексте.
5. Создайте функцию для поиска самого часто встречающегося слова в тексте. Функция должна возвращать слово и его частоту.

Задача № 4

Создайте функцию для упорядочивания слов в тексте по их частоте встречаемости. Функция должна возвращать список слов (от наиболее часто встречающегося к наименее часто встречающемуся или наоборот) и соответствующий список частот. Также функция должна в консоль выводить указанные два списка в виде таблицы.

Задача № 5

Разработайте функцию, которая в заданном списке имен находит самое длинное и самое короткое имя.

Задача № 6

Разработайте функцию, которая в заданном списке из пяти или более чисел находит 5 наименьших и 5 наибольших чисел.

Задача № 7

Создайте функцию для решения задачи «произвести подсчет элементов от значения *a* до значения *b* отсортированного списка целых чисел». При этом *a* и *b* могут не являться элементами списка. Реализуйте возможность выбора при вызове функции, какой цикл использовать (`while` или `for`).

Например, если имеется следующий список `L1` и значения *a* = 2, *b* = 30, то функция должна дать ответ 5:

```
>>> L1 = [1, 2, 3, 10, 15, 20, 40, 100]
>>> a = 2
>>> b = 30
>>> func(L1, a, b, 'for')
5
```

Задача № 8

Для функции из Задачи № 7 дополнительно реализуйте следующие возможности:

a. корректная работа с неотсортированным списком (в этом случае список необходимо предварительно отсортировать);
b. опциональный (по выбору пользователя) вывод в консоль подсчитываемых элементов списка и их индексов (чтобы пользователь мог выбрать, какую информацию выводить в консоль: ничего не выводить, выводить только элементы списка, выводить элементы списка вместе с их индексами);
c. подсчет только четных элементов;
d. помимо подсчета элементов в диапазоне от *a* до *b*, также реализовать вычисление:
    - суммы элементов;
    - произведения элементов;
    - арифметической средней элементов.

Задача № 9

Создайте функцию для работы пользователя с функцией из Задачи № 8 через консоль. Необходимо реализовать возможность ввода пользователем списка элементов, диапазон, дополнительные параметры (необходим ли подсчет только четных элементов и т.д.).

При вызове функции должен запускаться бесконечный цикл, в котором реализуется диалог с пользователем через ввод определенных команд в консоль. В таком случае выход из цикла будет производится при вводе пользователем специальной команды, например слова `'end'`.

Задача № 10*

1. *Создайте функцию, которая определяет текущие дату и время в определенном городе. Функция должна принимать в качестве аргумента название города и возвращать текущие дату и время в этом городе. Функция должна работать как минимум с 10 городами.*

   Для определения даты и времени используется модуль datetime. Пример:

   ```
   >>> from datetime import datetime
   >>> str(datetime.now())
   '2024-11-23 20:15:49.236940'
   ```

   Документация: https://docs.python.org/3/library/datetime.html
2. *Создайте функцию для игры в орла и решку. Пользователь вводит «орел» или «решка», функция возвращает ответ «Вы угадали» или «Вы не угадали». Для загадывания «орла» и «решки» необходимо использовать модуль random и функцию randomint.*

   Документация:
   https://docs.python.org/3/library/random.html#random.randint
3. *Создайте функцию для игры в «Камень, ножницы, бумага».*

Практическое задание № 6
«Классы в Python»

1. Объектно-ориентированное программирование

Одной из наиболее распространенных парадигм программирования в настоящий момент является **объектно-ориентированное программирование**.

Основными понятиями в объектно-ориентированном программировании являются:

- Класс.
- Объект.
- Наследование.
- Инкапсуляция.
- Полиморфизм.

1.1 Класс. Объект

Как уже упоминалось ранее, типы объектов в языке Python можно разделить на 2 категории:

- **встроенные типы объектов**, такие как `int`, `float`, `str`, `list` и т.д.
- **собственные типы объектов**, т.е. новые, создаваемые программистом.

Создать собственный тип объектов можно с помощью **классов**.

Класс (Class) – некоторая абстрактная сущность, использующаяся для описания объектов.

Класс – абстракция совокупности реальных объектов, которые имеют общий набор свойств и обладают одинаковым поведением.

Класс – абстрактное описание множества однородных объектов, имеющих одинаковые:

- атрибуты,
- операции,
- отношения с объектами других классов.

Пример: класс «Человек» с атрибутами: ФИО, год рождения, рост, вес; с операциями: идти, работать, спать; связанный с классами «Дом», «Автомобиль», «Собака».

Объект (Object) – экземпляр соответствующего класса.

Объект – конкретное воплощение класса [8].

Пример: объект Иванов Иван Иванович, определенного возраста, роста, веса (конкретный человек) – экземпляр класса «Человек».

Таким образом, предметная область, которая моделируется или для которой разрабатывается программное обеспечение, может быть представлена как совокупность взаимосвязанных объектов, описываемых классами. Такой подход лежит в основе **объектно-ориентированного программирования**.

1.2 Наследование. Инкапсуляция. Полиморфизм

Наследование (Inheritance) – процесс, в результате которого объект нового типа B (наследник, потомок) определяется таким же, как объект старого (существующего) типа A (родитель, предок) с учетом возможных некоторых различий. Объекты второго типа B наследуют свойства объектов первого типа A [18].

Наследование – наделение объекта-потомка свойствами (состоянием и поведением) объекта-родителя.

Наследование предполагает, что при создании объекта-потомка, у него будут те же свойства, что и у объекта-предка, при этом:

- наследованные свойства объекта-потомка могут быть переопределены,
- у объекта-потомка могут быть собственные свойства, которых нет у объекта-родителя.

Инкапсуляция – сокрытие информации об объекте, предполагающая предотвращение получения и изменения информации об объекте.

Инкапсуляция – ограничение доступа к составляющим объекта компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса.

Полиморфизм – особенность операции, связанная с тем, что смысл операции (например, операции сложения) зависит от объектов, на которых она выполняется [5].

Полиморфизм – разное поведение одного и того же метода в разных классах.

Например, можно сложить два числа, и можно сложить две строки. При этом результат будет разный, так как числа и строки являются разными классами. Например, `1 + 1` и `'1' + '1'`. В первом случае результат будет `2`, а во втором случае `'11'`.

2. Определение класса и объектов класса в Python

Для создания (определения) класса используется ключевое слово `class`, после которого указывается имя класса (обычно с большой буквы), ставится двоеточие и на следующей строке начинается отдельный блок кода для описания функционала класса, который может содержать атрибуты класса (переменные) и методы класса (функции) (рисунок 6.1).

```
class Имя_класса:
    инструкция_1
    …
    инструкция_N
```

Рисунок 6.1 – Общая форма записи при создании (определении) нового класса

Каждый экземпляр класса (объект) может содержать:

- **атрибуты –** для поддержания его состояния;
- **методы –** для изменения его состояния [15].

Для определения класса без методов и атрибутов в теле класса необходимо указать оператор `pass`.

Для создания объекта класса требуется вызвать специальный метод класса «конструктор», который по умолчанию имеется во всех создаваемых классах и в вызывающем коде носит то же имя, что и класс. Объект класса создается следующим образом:

```
имя_объекта = имя_класса()
```

Например, если был определен класс `Student`, то для создания объекта этого класса необходимо вызвать метод (т.е. функцию) с тем же названием `Student()`, в результате чего будет возвращен новый объект типа `Student` (листинг 6.1).

Листинг 6.1 – Определение класса `Student` и трех объектов класса `Student`

```
# Определение класса Student
class Student:
    pass


# Определение объектов класса Student
std1 = Student()
std2 = Student()
std3 = Student()
```

3. Атрибуты и методы

3.1. Атрибуты класса

Для хранения данных (состояния) объектов класса используются **атрибуты**, представляющие собой переменные с локальной областью видимости внутри класса.

Доступ к атрибуту вне класса осуществляется с помощью оператора `.` (точка) в соответствии со следующей формой:

```
объект.атрибут
```

При этом **атрибуты** можно разделить на 2 категории:

- **атрибуты класса**
- **атрибуты объекта**

Атрибуты класса – атрибуты, которые характерны всем объектам класса.

Атрибуты объекта – атрибуты, которые характерны лишь определенным объектам класса.

Создание атрибута, так же, как и переменной, происходит при первом присваивании какого-либо значения.

Например, для описания состояния объекта класса Студент (`Student`) можно создать общие для всех объектов атрибуты «имя» (name), «возраст» (age), «курс обучения» (course) (листинг 6.2).

Листинг 6.2 – Определение класса `Student` с атрибутами класса

```
# Определение класса Student
class Student:

    # Атрибуты класса
    name = '...'
    age = 0
    course = 0


# Определение объектов класса Student
std1 = Student()
std2 = Student()

print(std1.name)
print(std2.course)
```

```
...
0
```

Вне класса для получения доступа к атрибуту используется оператор `.`. При этом необходимо указывать имя объекта. Например, чтобы получить значение атрибута `name` у объекта `std1` класса `Student` необходимо использовать команду `std1.name`. Таким образом, происходит обращение к атрибуту определенного объекта, т.е. к **атрибуту объекта**.

Если же дополнить команду оператором присваивания `=` (`std1.name =` …), то можно произвести переприсваивание значения, т.е. изменить значения атрибута объекта.

Например, вне класса мы можем изменить атрибуты у объекта класса Student следующим образом (листинг 6.3).

Листинг 6.3 – Изменение значений атрибутов `name`, `course` объекта `std1` класса `Student`

```
# Определение класса Student
class Student:

    # Атрибуты класса
    name = '...'
    age = 0
    course = 0


# Определение объектов класса Student
std1 = Student()
std2 = Student()

# Изменение атрибутов объекта
std1.name = 'Jack'
std1.course = 1

print(std1.name)
print(std1.age)
print(std1.course)
```

```
Jack
0
1
```

3.2. Методы класса

Для описания поведения объектов при определении класса используются **методы**, представляющие собой функции с локальной областью видимости внутри этого класса.

Для **определения метода** используют те же правила, что и при определении функции, за исключением ключевого слова `self` – его необходимо указывать первым аргументом при определении метода и не указывать при вызове аргумента.

Общая форма записи при определении метода выглядит следующим образом:

```
def имя_метода(self):
    тело_метода
```

или

```
def имя_метода(self, аргумент_1, …, аргумент_n)
    тело_метода
```

Для **доступа к методу**, так же, как и к атрибуту, используется оператор . (точка). Так как метод представляет собой функцию, для вызова метода необходимо указать круглые скобки после его имени, в которых при необходимости перечислить аргументы (параметры) метода (листинг 6.4):

```
объект.метод()
```

или

```
объект.метод(аргумент_1, …, аргумент_n)
```

Листинг 6.4 – Определение и вызов методов класса `Student`

```python
# Определение класса Student
class Student:

    # Метод посещения занятия студентом
    def start_studying(self):
        print('Студент приступил к занятиям')

    # Метод завершения занятия студентом
    def finish_studying(self):
        print('У студента закончились занятия')


# Определение объекта класса Student
std1 = Student()

# Вызов методов объекта
std1.start_studying()
std1.finish_studying()
```

```
Студент приступил к занятиям
У студента закончились занятия
```

Ключевое слово `self` в методах используется для ссылки на конкретный экземпляр класса (текущий объект). С помощью него при определении класса можно обращаться к атрибутам и методам объекта через оператор `.` (точка) [8]:

```
self.атрибут
self.метод()
```

Например, с помощью ключевого слова `self` при определении метода можно вызвать определенные ранее методы (листинг 6.5).

Листинг 6.5 – Использование ключевого слова `self` для вызова методов `start_studying()`, `finish_studying()` при определении метода `study()`

```
1  # Определение класса Student
2  class Student:
3
4      # Метод посещения занятия студентом
5      def start_studying(self):
6          print('Студент приступил к занятиям')
7
8      # Метод завершения занятия студентом
9      def finish_studying(self):
10         print('У студента закончились занятия')
11
12     # Метод описания процесса учебы
13     def study(self):
14         self.start_studying()  # Начать обучение
15         self.finish_studying() # Закончить обучение
16
17
18 std1 = Student()   # Определение объекта класса Student
19 std1.study()       # Вызов метода объекта
```

```
Студент приступил к занятиям
У студента закончились занятия
```

В случае, если метод имеет аргументы помимо ключевого слова `self`, то при вызове указываются значения для этих аргументов, ключевое слово `self` пропускается (строка 20 листинга 6.6).

Листинг 6.6 – Вызов метода `study()`, требующего значение для аргумента `student_name`

```
1  # Определение класса Student
2  class Student:
3
4      # Метод посещения занятия студентом
5      def start_studying(self):
6          print('Студент приступил к занятиям')
7
8      # Метод завершения занятия студентом
9      def finish_studying(self):
10         print('У студента закончились занятия')
11
12     # Метод описания процесса учебы
13     def study(self, student_name, student_course=-1):
14         self.name = student_name
15         self.course = student_course
16
17         print(f'Студент: {self.name}; Курс: {self.course}')
18         self.start_studying()  # Начать обучение
19         self.finish_studying() # Закончить обучение
20
21
22 std1 = Student()       # Определение объекта класса Student
23 std1 = Student()
24 std1.study(student_name='Jack')       # Вызов метода объекта
```

```
Студент: Jack; Курс: -1
Студент приступил к занятиям
У студента закончились занятия
```

3.3. Атрибуты объекта

Для создания атрибутов объекта необходимо использовать ключевое слово `self`, при этом использование его допускается только в теле метода. Поэтому атрибут объекта будет создан только при вызове такого метода, а значит и обращаться к атрибуту объекта возможно тоже только после вызова метода, в противном случае возникнет ошибка (листинг 6.7).

В листинге 6.7 происходит создание атрибутов `name`, `course` объекта с помощью метода `study()` путем присваивания переменным `name`, `course` (строки 8, 9) значений, которые при вызове передаются на вход методу `study()` в качестве аргументов `student_name`, `student_course` (строка 19 – для объекта `std1`, строка 20 – для объекта `std2`).

Листинг 6.7 – Создание атрибутов объекта `name`, `course` при вызове метода `study()`. Если метод `study()` не вызывается, то при попытке обращения к атрибуту `name` возникает ошибка об отсутствии такого атрибута у объекта `std3`: `'Student' object has no attribute 'name'` (закомментированная строка 26)

```
1  # Определение класса Student
2  class Student:
3
4      # Метод описания процесса учебы
5      def study(self, student_name, student_course):
6
7          # Атрибуты объекта
8          self.name = student_name
9          self.course = student_course
10
11         print(f'Студент: {self.name}; Курс: {self.course}')
12         print('Приступил к обучению')
13
14 # Определение объектов класса Student
15 std1 = Student()
16 std2 = Student()
17 std3 = Student()
18
19 std1.study('Jack', 1)
20 std2.study('Sophie', 3)
21
22 # Проверка атрибутов
23 print(std1.name, std1.course)
24 print(std2.name, std2.course)
25
26 #print(std3.name, std3.course) # Ошибка!
```

```
Студент: Jack; Курс: 1
Приступил к обучению
Студент: Sophie; Курс: 3
Приступил к обучению
Jack 1
Sophie 3
```

Специальный метод, который используется в том числе для создания **атрибутов объекта** называется **конструктором класса**.

Конструктор класса (инициализатор, initializer) – метод, использующийся для инициализации (подготовки, настройки) объекта и создании этого объекта.

Конструктор класса при определении класса (в коде класса) называется `__init__()` и выглядит следующим образом:

```
def __init__(self):
  тело_метода
```

При вызове конструктора, в результате которого происходит создание объекта класса, необходимо указывать имя класса:

```
имя_объекта = имя_класса()
```

По умолчанию в Python каждый создаваемый класс имеет конструктор, однако в строке заголовка метода-конструктора отсутствуют аргументы, кроме `self`, а в теле отсутствуют какие-либо инструкции. Конструктор, который изначально имеется у класса в явном виде можно прописать (переопределить) следующим образом (листинг 6.8).

Листинг 6.8 – Переопределение конструктора класса `Student`

```
# Определение класса Student
class Student:

    # Конструктор класса
    def __init__(self):
        pass


# Определение объектов класса Student
std1 = Student()
```

Если добавить в строку заголовка метода-конструктора аргументы, то при создании объекта необходимо будет указывать значения для этих аргументов. Блок инструкций в теле метода-конструктора будет выполняться при создании объекта (листинг 6.9).

Листинг 6.9 – Переопределение конструктора класса `Student`: добавление аргументов `name`, `age`, `course` в строку заголовка конструктора

```
# Определение класса Student
class Student:

    # Конструктор класса
    def __init__(self, name, age, course):
        print(name, age, course)


# Определение объектов класса Student
std1 = Student('Jack', 20, 1)
std2 = Student('Sophie', 22, 3)
```

```
Jack 20 1
Sophie 22 3
```

Передаваемые на вход метода-конструктора значения аргументов можно сохранить в атрибуты объекта. Для этого необходимо использовать ключевое слово `self` (листинг 6.10) – так как же, как и в листинге 6.7. Для обращение к таким атрибутам внутри конструктора тоже используется ключевое слово `self` (строка 11 листинга 6.10).

Листинг 6.10 – Переопределение конструктора класса `Student`: сохранение значений аргументов `name`, `age`, `course` в атрибуты объекта

```
# Определение класса Student
class Student:

    # Конструктор класса
    def __init__(self, name, age, course):

        # Атрибуты объекта
        self.name = name
        self.age = age
        self.course = course

        print(f'Создан объект: {self.name}, {self.age}, {self.course}')


# Определение объектов класса Student
std1 = Student('Jack', 20, 1)
std2 = Student('Sophie', 22, 3)
```

```
Создан объект: Jack, 20, 1
Создан объект: Sophie, 22, 3
```

Таким образом можно реализовать создание объектов сразу с определенными значениями атрибутов.

Изменение таких атрибутов в том блоке инструкций, где был создан объект, осуществляется так же через оператор присваивания `=`. При этом с

помощью данного оператора можно создавать новые атрибуты для конкретного объекта, которых не будет у других объектов (строка 18 листинга 6.11).

Листинг 6.11 – Пример изменения атрибута `name` и добавления нового атрибута `average_grade` для объекта `std1`. Так как атрибута `average_grade` у объекта `std2` нет, то при попытке обратиться к данному атрибуту возникнет ошибка (закомментированная строка 22)

```
1  # Определение класса Student
2  class Student:
3
4      # Конструктор класса
5      def __init__(self, name, age, course):
6
7          # Атрибуты объекта
8          self.name = name
9          self.age = age
10         self.course = course
11
12         print(f'Создан объект: {self.name}, {self.age}, {self.course}')
13
14
15 # Определение объектов класса Student
16 std1 = Student('Jack', 20, 1)
17 std2 = Student('Sophie', 22, 3)
18
19 std1.name = 'Jackie'        # Изменение атрибута
20 std1.average_grade = 5.0    # Добавление нового атрибута
21 print(f'Объект std1: {std1.name}, {std1.age}, {std1.course}, {std1.average_grade}')
22 #print(f'Объекм std2: {std1.name}, {std2.age}, {std2.course}, {std2.average_grade}')
```

```
Создан объект: Jack, 20, 1
Создан объект: Sophie, 22, 3
Объект std1: Jackie, 20, 1, 5.0
```

3.4. Подчеркивания в именах Python: _name, name_, __name, __name__

В именах переменных и функций Python (как правило, в теле класса или в модуле) используются подчеркивания четырех типов [22]:

- `_name` (одинарное подчеркивание в начале, single leading underscore) – является **слабым индикатором «внутреннего использования»** имени, т.е. такое подчеркивание показывает, что имя предназначается для использования внутри класса или модуля. При этом к имени все равно можно обратиться вне класса стандартным образом. В случае модуля при его импорте командой вида `from M import *` не импортируются объекты, имена которых начинаются с подчеркивания.
- `name_` (одинарное подчеркивание в конце, single trailing underscore) – общепринятое соглашение (не влияет на выполнения программного кода) **для использования зарезервированных имен** языка Python при создании новых объектов. Например, если объекту требуется присвоить зарезервированное имя `list`, то в таком случае следует добавить подчеркивание в конце: `list_ = [1, 2, 3]`.

- `__name` (двойное подчеркивание в начале, double leading underscore) – используется в теле класса как имя атрибута, к которому будет затруднительно обратиться вне класса из-за искажения имени (name mangling). Можно сказать, что это **сильный индикатор «внутреннего использования»**.
- `__name__` (двойное подчеркивание в начале и в конце, double leading and trailing underscore) – представляют собой **«магические» объекты или атрибуты** («magic» objects or attributes), которые создаются автоматически в языке Python (например, `__init__`, `__str__`, `__import__`, `__file__`). Их можно переопределять, но создавать новые имена, используя двойное подчеркивание в начале и в конце не рекомендуется.

4. Инкапсуляция: приватные атрибуты, методы-свойства

Инкапсуляция является фундаментальной концепцией объектно-ориентированного программирования.

Инкапсуляция предотвращает прямой доступ к атрибутам объекта из вызывающего кода.

В Python инкапсуляция реализуется через **приватные атрибуты**, доступ к которым предоставляется посредством специальных **методов-свойств** [8].

4.1. Приватные атрибуты

С точки зрения доступности можно выделить две основные категории атрибутов:

- **Публичные атрибуты**.
- **Приватные атрибуты**.

Публичные атрибуты – атрибуты, к которым можно получить доступ вне тела класса. По умолчанию в Python все атрибуты класса и объекта являются публичными.

Приватные атрибуты – атрибуты, к которым нельзя получить доступ вне тела класса (т.е. можно получить доступ только в теле класса).

В Python не существует приватных атрибутов, однако применяется **искажение имен переменных**, чтобы предотвратить случайный доступ к таким переменным [15].

Для создания подобного «приватного» атрибута (атрибута с искаженным именем, к которому будет затруднительно получить доступ вне класса) в теле класса перед именем атрибута необходимо добавить два подчеркивания следующим образом:

```
__атрибут
```

Получить доступ к такому атрибуту немного сложнее, чем к стандартному атрибуту (требуется добавить подчеркивание и указать имя класса):

```
объект._класс__атрибут
```

Одинарное нижнее подчеркивание в начале имени является общепринятым соглашением, показывающем, что атрибут условно является приватным:

```
_атрибут
```

Предполагается, что попыток получить доступ к такому атрибуту извне (вне тела класса) не будет. Тем не менее получение доступа осуществляется стандартным образом:

```
объект._атрибут
```

В листинге 6.12 происходит создание двух «приватных» атрибутов `__age`, `__course` (строки 9, 10).

Попытка обратиться к «приватному» атрибуту `__age` приведет к ошибке `'Student' object has no attribute '__age'` (закомментированная строка 19).

Попытка изменить значение атрибута `__age` приведет лишь к созданию одноименного публичного атрибута со значением 40 (строка 22), поэтому при обращении к такому атрибуту в консоль выводится значение 40.

Листинг 6.12 – Создание «приватных» атрибутов `__age`, `__course`

```
1  # Определение класса Student
2  class Student:
3
4      # Конструктор класса
5      def __init__(self, name, age, course):
6
7          # Атрибуты объекта
8          self.name = name        # Публичный атрибут
9          self.__age = age        # "Приватный" атрибут
10         self.__course = course # "Приватный" атрибут
11
12         print(f'Создан объект: {self.name}, {self.__age}, {self.__course}')
13
14
15 # Определение объектов класса Student
16 std1 = Student('Jack', 20, 1)
17 std2 = Student('Sophie', 22, 3)
18
19 #print(f'Объект std1: {std1.name}, {std1.__age}')
20
21 std1.age = 30       # Создание нового публичного атрибута age
22 std1.__age = 40     # Создание нового публичного атрибута __age
23
24 print(f'Объект std1: name = {std1.name}')
25 print(f'Объект std1: __age = {std1.__age}')
26 print(f'Объект std1: __age ("приватный") = {std1._Student__age}')
```

```
Создан объект: Jack, 20, 1
Создан объект: Sophie, 22, 3
Объект std1: name = Jack
Объект std1: __age = 40
Объект std1: __age ("приватный") = 20
```

Для работы с «приватными» атрибутами обычно создают специальные методы-свойства: **геттер**, **сеттер**, **делитер**.

4.2. Методы-свойства

Методы-свойства (или просто свойства) – стандартные методы класса, которые позволяют получать доступ к приватным атрибутам для их чтения (получения значения), записи (изменения значения), удаления (удаления атрибута).

В соответствии с типом операции выделяют три метода-свойства:

- **Геттер**.
- **Сеттер**.
- **Делитер**.

Геттер (Getter) – метод, который возвращает значение атрибута.

Сеттер (Setter) или Мютэйтор (Mutator) – метод, который изменяет значение атрибута.

Делитер (Deleter) – метод, который удаляет атрибут.

Пример создания **геттера** для получения значения «приватного» атрибута `__age` представлен в листинге 6.13.

После создания объекта `std1` получить значение атрибута `__age` можно с помощью геттера `get_age()`, который возвращает значение 20 (строка 24 листинга 6.13).

После попытки изменить значение «приватного» атрибута `__age` на 40 (строка 27) мы можем снова вызвать геттер и убедиться в том, что значение «приватного» атрибута не изменилось, т.к. геттер вновь возвратил число 20 (строка 30).

Листинг 6.13 – Создание геттера `get_age()` для получения значения «приватного» атрибута `__age`

```
1  # Определение класса Student
2  class Student:
3
4      # Конструктор класса
5      def __init__(self, name, age, course):
6
7          # Атрибуты объекта
8          self.name = name       # Публичный атрибут
9          self.__age = age       # Приватный атрибут
10         self.__course = course # Приватный атрибут
11
12         print(f'Создан объект: {self.name}, {self.__age}, {self.__course}')
13
14     # Метод получения возраста (Геттер) (Метод-свойство)
15     def get_age(self):
16         return self.__age
17
18
19 # Определение объектов класса Student
20 std1 = Student('Jack', 20, 1)
21 std2 = Student('Sophie', 22, 3)
22
23 print(f'Объект std1: {std1.name}')
24 print(f'Объект std1: вызов get_age(): {std1.get_age()}')
25
26 std1.age = 30       # Создание нового публичного атрибута age
27 std1.__age = 40     # Создание нового публичного атрибута __age
28
29 print(f'Объект std1: {std1.name}, {std1.__age}')
30 print(f'Объект std1: вызов get_age(): {std1.get_age()}')
```

```
Создан объект: Jack, 20, 1
Создан объект: Sophie, 22, 3
Объект std1: Jack
Объект std1: вызов get_age(): 20
Объект std1: Jack, 40
Объект std1: вызов get_age(): 20
```

Для изменения значения «приватного» атрибута `__age` необходимо создать **сеттер** (строки 19, 20 листинга 6.14). В листинге мы видим, что после вызова сеттера (строка 28) «приватному» атрибуту присвоилось новое значение – 40, что можно проверить путем вызова **геттера** (строка 29).

Листинг 6.14 – Создание сеттера `set_age()` для изменения значения «приватного» атрибута `__age`

```
# Определение класса Student
class Student:

    # Конструктор класса
    def __init__(self, name, age, course):

        # Атрибуты объекта
        self.name = name        # Публичный атрибут
        self.__age = age        # Приватный атрибут
        self.__course = course # Приватный атрибут

        print(f'Создан объект: {self.name}, {self.__age}, {self.__course}')

    # Метод получения возраста (Геттер) (Метод-свойство)
    def get_age(self):
        return self.__age

    # Метод измнения возраста (Аксессор) (Метод-свойство)
    def set_age(self, value):
        self.__age = value


# Определение объектов класса Student
std1 = Student('Tom', 20, 1)
print(f'Объект std1: вызов get_age(): {std1.get_age()}')

# Изменение значения приватного атрибута __age
std1.set_age(value=40)
print(f'Объект std1: вызов get_age(): {std1.get_age()}')
```

```
Создан объект: Tom, 20, 1
Объект std1: вызов get_age(): 20
Объект std1: вызов get_age(): 40
```

4.3. Аннотации свойств

В отдельных случаях, при определении сложных по функционалу классов, может потребоваться применение более сложного подхода для создания геттеров и сеттеров, основанного на **аннотации свойств**. Такой подход позволяет использовать одно и тоже имя для вызова разных функций.

Свойство – тип объекта, который присваивается атрибуту класса [6].

Для создания свойства используется встроенная функция `property()` следующим образом:

```
атрибут = property()
```

Функция `property()` принимает 4 аргумента (3 метода доступа и строку документации для этого свойства):

- `fget` – метод получения значения (геттер),
- `fset` – метод установки значения (сеттер),
- `fdel` – метод удаления значения (делитер),
- `doc` – строка документации.

Пример создания атрибута `age` в качестве свойства представлен в листинге 6.15.

Листинг 6.15 – Создание свойства `age` (строка 22), обращение к которому вызывает метод `get_age()` (строки 27, 31), либо `set_age()` при присваивании значения (строка 30) для изменения значения «приватного» атрибута `__age`

```
1  # Определение класса Student
2  class Student:
3
4      # Конструктор класса
5      def __init__(self, name, age, course):
6
7          # Атрибуты объекта
8          self.name = name        # Публичный атрибут
9          self.__age = age        # Приватный атрибут
10         self.__course = course # Приватный атрибут
11
12         print(f'Создан объект: {self.name}, {self.__age}, {self.__course}')
13
14     # Метод получения возраста (Геттер) (Метод-свойство)
15     def get_age(self):
16         return self.__age
17
18     # Метод измненения возраста (Аксессор) (Метод-свойство)
19     def set_age(self, value):
20         self.__age = value
21
22     age = property(fget = get_age, fset = set_age) # Свойтсво
23
24
25 # Определение объектов класса Student
26 std1 = Student('Jack', 20, 1)
27 print(f'Объект std1: обращение к геттеру age: {std1.age}')
28
29 # Изменение значения приватного атрибута __age
30 std1.age = 40
31 print(f'Объект std1: обращение к геттеру age: {std1.age}')
```

```
Создан объект: Jack, 20, 1
Объект std1: обращение к геттеру age: 20
Объект std1: обращение к геттеру age: 40
```

Для автоматизации подобной операции применяется специальный синтаксис с использованием символа `@`. Такая процедура называется **декорированием**.

Декорирование – способ указания дополнительного кода для функций и классов. Данный код автоматически запускается в конце операторов определения функций (`def`) и классов (`class`) [6].

Декораторы не являются обязательными элементами языка, их можно заменить вызовом вспомогательных функций.

Декорирование функции выглядит следующим образом:

```
@декоратор
def функция():
  тело_функции
```

Для того, чтобы указать, что метод необходимо рассматривать как свойство, используется декоратор `@property`. В таком случае метод будет являться геттером:

```
@property
def atr():
  …
print(atr)
```

Для определения сеттера необходимо использовать декоратор следующего вида:

```
@имя_геттера.setter
```

Если был определен геттер с именем `atr`, то определение сеттера будет выглядеть следующим образом:

```
@atr.setter
def atr():
  …
atr = …
```

Аналогичным образом определяется метод удаления:

```
@atr.deleter
def atr():
  …
del atr
```

Пример создания декорированных методов `age` (геттера, сеттера, делитера) представлен в листинге 6.16).

Листинг 6.16 – Использование декораторов для создания геттера, сеттера и делитера с одним и тем же именем `age`. После вызова делитера при попытке вызвать геттер возникает ошибка `AttributeError: 'Student' object has no attribute '_Student__age'` (закомментированная строка 40)

```
# Определение класса Student
class Student:

    # Конструктор класса
    def __init__(self, name, age, course):

        # Атрибуты объекта
        self.name = name       # Публичный атрибут
        self.__age = age       # Приватный атрибут
        self.__course = course # Приватный атрибут

        print(f'Создан объект: {self.name}, {self.__age}, {self.__course}')

    # Метод получения возраста (Геттер)
    @property
    def age(self):
        return self.__age

    # Метод измненения возраста (Аксессор)
    @age.setter
    def age(self, value):
        self.__age = value

    # Метод удаление атрибута __age (Делитер)
    @age.deleter
    def age(self):
        del self.__age


# Определение объектов класса Student
std1 = Student('Jack', 20, 1)
print(f'Объект std1: обращение к геттеру age: {std1.age}')

# Изменение значения приватного атрибута __age
std1.age = 40
print(f'Объект std1: обращение к геттеру age: {std1.age}')

# Удаление атрибута __age
del std1.age
#print(f'Объект std1: обращение к геттеру age: {std1.age}')
```

```
Создан объект: Jack, 20, 1
Объект std1: обращение к геттеру age: 20
Объект std1: обращение к геттеру age: 40
```

5. Наследование

Наследование (Inheritance) – процесс, в результате которого объект нового типа B (наследник, потомок) определяется таким же, как и объект старого (существующего) типа A (родитель, предок) с учетом возможных

некоторых различий. Объекты второго типа B наследуют свойства объектов первого типа A [18].

При **наследовании классов** участвуют классы двух категорий:

- **Базовый класс** (Base class), который еще называют как Суперкласс / Класс-родитель / Класс-предок.
- **Производный класс** (Derived class), который еще называют как Подкласс / Дочерний класс / Класс-потомок.

Наследование классов предполагает, что производный класс наследует свойства базового класса.

Механизм наследования классов в Python характеризуется следующими возможностями [15]:

- допускается несколько базовых классов (множественное наследование),
- производный класс может переопределять любые методы своего базового класса (базовых классов),
- метод производного класса может вызывать метод базового класса с тем же именем.

Объекты могут содержать произвольные объемы и типы данных. Как и модули, классы разделяют динамическую природу Python: они создаются во время выполнения и могут быть изменены в дальнейшем после создания [15].

5.1. Простое наследование

Синтаксис создания производного класса от базового класса выглядит следующим образом:

```
class Производный_класс(Базовый_класс):
    инструкция_1
    …
    инструкция_N
```

Рисунок 6.2 – Общая форма записи при создании (определении) производного класса

Наследование эффективно применять в случае, когда, например, у нескольких классов имеются одинаковые атрибуты и методы – тогда можно определить базовый класс с этими атрибутами и методами, которые будут унаследованы производными классами. Таким образом исключается лишнее дублирование идентичного кода.

Например, в листинге 6.17 производные классы `Student`, `Teacher` наследуют конструктор базового класса `User`. Это означает, что такой же метод `__init__()`, принимающий аргументы `name`, `age` фактически определен в теле класса `Student` и класса `Teacher` неявным образом, т.е. без наследования пришлось бы прописывать инструкции на строках 4-6 в теле каждого из классов `Student`, `Teacher`.

Листинг 6.17 – Определение двух производных классов `Student`, `Teacher`, которые наследуют конструктор базового класса `User`

```
1  # Базовый класс
2  class User():
3
4      def __init__(self, name, age):
5          self.name = name
6          self.age = age
7
8
9  # Производный класс
10 class Student(User):
11     pass
12
13 # Производный класс
14 class Teacher(User):
15     pass
16
17 # Определение объектов класса User
18 guest1 = User('guest', 30)
19 admin1 = User('admin', None)
20
21 # Определение объектов класса Student
22 std1 = Student('Jack', 20)
23 std2 = Student('Sophie', 22)
24
25 # Определение объекта класса Teacher
26 teacher1 = Teacher('Mr. James', 50)
```

В листинге 6.17 при создании экземпляров (объектов) классов `Student`, `Teacher` происходит создание атрибутов `name`, `age`, через унаследованный конструктор. Получить доступ к таким атрибутом в теле производного класса (как и вне тела производного класса) можно стандартным образом через оператор `.` (точка) (строки 13, 20 листинга 6.18).

Листинг 6.18 – Определение метода `print_atr()` в производном классе `Student`, получение доступа к атрибутам объекта, созданным через унаследованный конструктор

```
# Базовый класс
class User():

    def __init__(self, name, age):
        self.name = name
        self.age = age


# Производный класс
class Student(User):

    def print_atr(self):
        print(f'name: {self.name}, age: {self.age}')


# Определение объектов класса Student
std1 = Student('Jack', 20)
std2 = Student('Sophie', 22)

print(std1.name)

std1.print_atr()
std2.print_atr()
```

```
Jack
name: Jack, age: 20
name: Sophie, age: 22
```

В листинге 6.18 также представлено определение в производном классе `Student` собственного метода `print_atr()`, который отсутствует у базового класса `User`.

5.2. Множественное наследование

Множественное наследование предполагает, что производный класс наследует свойства нескольких базовых классов.

Синтаксис создания производного класса от нескольких базовых классов выглядит следующим образом (рисунок 6.3).

```
class Производный_класс(Базовый_класс_1, ..., Базовый_класс_N):
    инструкция_1
    …
    инструкция_N
```

Рисунок 6.3 – Общая форма записи при создании (определении) производного класса от нескольких базовых классов

Таким образом, у производного класса будет обобщенный функционал (атрибуты и методы) всех базовых классов. Если у базовых классов имеются атрибуты или методы с одним и тем же именем, то производный класс

унаследует эти атрибуты или методы от того базового класса, который расположен раньше в строке заголовка класса.

Например, в листинге 6.19 представлено множественное наследование, при котором класс `Teacher` наследует атрибуты `admin_password`, `guest_password` от двух базовых классов `Admin`, `Guest`. При этом, т.к. в строке заголовка класса `Teacher` первым указан базовый класс `Admin`, класс `Teacher` унаследует конструктор именно от класса `Admin`.

Листинг 6.19 – Определение производного класса `Teacher`, который наследует атрибуты и методы двух базовых классов `Admin`, `Guest`

```
# Базовый класс
class Admin:

    admin_password = 'admin11'

    def __init__(self):
        self.id = 'A001'
        print('Создан объект Администратор')

# Базовый класс
class Guest:

    guest_password = 'guest11'

    def __init__(self):
        self.id = 'G001'
        print('Создан объект Гость')


# Производный класс
class Teacher(Admin, Guest):

    def print_id(self):
        print(f'ID: {self.id}')


teacher1 = Teacher()
teacher1.print_id()
print(teacher1.admin_password)
print(teacher1.guest_password)
```

```
Создан объект Администратор
ID: A001
admin11
guest11
```

5.3. Переопределение унаследованного функционала. Обращение к базовому классу

Производный класс наследует все атрибуты и методы базового класса. Однако в теле производного класса эти атрибуты и методы могут быть переопределены.

Переопределение атрибута подразумевает использование того же имени при присваивании нового значения в теле класса. Другими словами, осуществляется переприсвавание значения (строка 13 листинга 6.20).

Переопределение метода подразумевает использование имени унаследованного метода при определении метода в теле производного класса (строки 16, 17 листинга 6.20).

Листинг 6.20 – Определение производного класса `Teacher`, который наследует атрибуты и методы двух базовых классов `Admin`, `Guest`

```
1  # Базовый класс
2  class User:
3
4      user_id = 'U001'
5
6      def print_id(self):
7          print(f'ID пользователя: {self.user_id}')
8
9
10 # Производный класс
11 class Student(User):
12
13     user_id = 'S001'  # Переприсваивание
14
15     # Переопределение метода print_id
16     def print_id(self):
17         print(f'ID студента: {self.user_id}')
18
19
20 # Определение объекта класса Student
21 std1 = Student()
22 std1.print_id()
```

```
S001
```

В случае, когда требуется не полностью переопределить атрибут или метод, а модифицировать его, то можно обратиться к функционалу базового класса с помощью специальной встроенной функции `super()`.

Функция `super()` возвращает прокси-объект, который делегирует вызовы методов (и обращения к атрибутам) базовому классу [13]. Обращение к атрибутам и методам происходит стандартным образом:

```
super().атрибут
super().метод()
```

Например, на строке 15 листинга 6.21 из тела производного класса `Student` происходит обращение к атрибуту `user_id` базового класса `User`, который используется как префикс (подстрока) при формировании одноименного атрибута со значением «User-S001». На строке 18 листинга 6.21 вызывается метод `print_text()` базового класса.

Листинг 6.21 – Обращение к функционалу базового класса `User` из тела производного класса `Student`: обращение к атрибуту `user_id`, вызов метода `print_text()`

```
# Базовый класс
class User:

    user_id = 'User-'

    def print_text(self):
        print('text1')
        print('text2')

# Производный класс
class Student(User):

    def print_text(self):

        prefix = super().user_id  # Обращение к user_id класса User
        user_id = prefix + 'S001'

        super().print_text()      # Вызов print_text() класса User
        print('text3')
        print('text4')
        print(f'ID: {user_id}')

# Определение объекта класса Student
std1 = Student()
std1.print_text()
```

```
text1
text2
text3
text4
ID: User-S001
```

Довольно часто может возникать необходимость использовать конструктор базового класса, как правило, в теле конструктора производного класса. Целью такой процедуры может являться, например, расширение конструктора для создания дополнительных атрибутов у производного класса (листинг 6.22).

Листинг 6.22 – Обращение к функционалу базового класса `User` из тела производного класса `Student`: вызов конструктора для создания атрибутов `name` и `age`, вызов метода `print_text()`

```python
# Базовый класс
class User:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def print_atr(self):
        print(f'name: {self.name}')
        print(f'age: {self.age}')


# Производный класс
class Student(User):

    def __init__(self, name, age, course):
        super().__init__(name, age)
        self.course = course

    # Эквивалентный конструктор:
    # def __init__(self, name, age, course):
    #     self.name = name
    #     self.age = age
    #     self.course = course

    def print_atr(self):
        super().print_atr()
        print(f'course: {self.course}')


# Определение объектов класса Student
user1 = User('user1', None)
std1 = Student('Jack', 20, 1)
std2 = Student('Sophie', 22, 3)

user1.print_atr()
std1.print_atr()
std2.print_atr()
```

```
name: user1
age: None
name: Jack
age: 20
course: 1
name: Sophie
age: 22
course: 3
```

Задачи

Задача № 1

Условие: заказчику требуется разработать программное обеспечение (ПО) в виде некоторой образовательной платформы. Вам необходимо продумать, как может выглядеть модуль такого программного обеспечения,

который отвечает за разграничение прав доступа к ресурсам ПО в зависимости от типа пользователя:

- User – стандартный тип пользователя.
- Student – тип пользователя с дополнительной информацией (о студенте).
- Teacher – тип пользователя с дополнительной информацией (о преподавателе) и с дополнительными возможностями.

Требуется:

1. Определите базовый класс `User`. В теле класса определите метод `show_info()`, который будет выводить в консоль информацию о пользователе. На данном этапе реализуйте вывод в консоль текста «Информация о пользователе:» при вызове `show_info()`.
2. Предусмотрите сохранение следующей информации о каждом новом пользователе:
    - ID пользователя формата «U001», который может использоваться как уникальный идентификатор пользователя.
    - Тип пользователя. Название типа пользователя соответствует имени класса, т.е. «User».
    - Логин.
    - Пароль.
3. Модифицируйте метод `show_info()` так, чтобы при его вызове в консоль выводилась вся информация о пользователе из п.2. (ID, Тип пользователя, Логин, Пароль).
4. Сделайте Логин и Пароль «приватными» атрибутами, доступ к которым предоставляется через соответствующие геттеры и сеттеры.
5. Определите производный класс `Student` (класс `User` будет являться базовым классом).
   Предусмотрите сохранение следующей информации о каждом новом пользователе типа «Student»:
    - ID пользователя формата «S001», который может использоваться как уникальный идентификатор пользователя.
    - Тип пользователя, т.е. «Student».
    - Логин.
    - Пароль.
    - Имя.
    - Курс.
6. Определите производный класс `Teacher` (класс `User` будет являться базовым классом).
   Предусмотрите сохранение следующей информации о каждом новом пользователе типа «Teacher»:
    - ID пользователя формата «T001», который может использоваться как уникальный идентификатор пользователя.

- Тип пользователя, т.е. «Teacher».
- Логин.
- Пароль.
- Имя.
- Стаж работы.

Определите следующие методы класса `Teacher`:

- метод, с помощью которого пользователь типа «Teacher» может добавлять новых студентов в систему.
- метод, с помощью которого пользователь типа «Teacher» может удалять студентов из системы.

7. Реализуйте следующие дополнительные возможности:
   a. Реализуйте проверку на допустимый пароль (пароль должен содержать не менее 7 символов, включать цифры, прописные и заглавные буквы).
   b. Реализуйте проверку на допустимый логин (логин должен быть уникальным для каждого пользователя).
   c. Реализуйте автоматическое инкрементирование числовой части идентификатора пользователя (последние 3 цифры). Например, первому пользователю присваивается ID «U001», второму пользователю – «U002» и т.д.

При решении Задачи № 1 допускается в случае необходимости определять дополнительные классы, атрибуты и методы, использовать глобальные переменные и т.д.

Задача № 2

Определите класс `MyList` для описания встроенного списка Python (тип данных `list`) с некоторыми дополнительными возможностями:

1. Определите метод `print_info()`, который выводит в консоль примерно следующую информацию о количестве элементов *n* в списке: «Данный список содержит *n* элемента(ов)».
   Пример:

```
>>> list1 = MyList([1, 2, 3])
>>> list1.print_info()
'Данный список содержит 3 элемента(ов)'
```

2. Определите метод `drop_value(value)`, который удаляет из списка все элементы, соответствующие `value`.
   Пример:

```
>>> list1 = MyList([1, 2, 3, 2, 3, 40, 50, 2, 2, 1, 0, 2])
>>> list1.drop_value(2)
[1, 3, 3, 40, 50, 1, 0]
```

3. Определите метод `add_indeces(value)`, который перед каждым элементом списка добавляет строковый элемент вида «id: *i*», где *i* – индекс элемента в исходном списке. Реализуйте возможность использовать метод `append()` так, чтобы список сохранял подобный вид.

   Пример:

```
>>> list1 = MyList([1, 3, 9, 80])
>>> list1.add_indeces()
['id-0', 1, 'id-1', 3, 'id-2', 9, 'id-3', 80]
>>> list1.append(10)
>>> list1
['id-0', 1, 'id-1', 3, 'id-2', 9, 'id-3', 80, 'id-4', 10])
```

Задача №3. Симуляция Пекарни

С помощью классов необходимо реализовать программу, которая описывает функционирование Пекарни.

Предусмотрите действующие лица:

- начальник,
- пекарь,
- кассир,
- клиент.

Предусмотрите следующие виды продукции:

- хлеб,
- булочка,
- готовая еда,
- напитки.

Задача №4. Симуляция Фермы

Реализуйте симуляцию функционирования Фермы. На ферме имеется несколько типов животных: курица, овца, корова. Выращивается несколько сельскохозяйственных культур: пшеница, кукуруза, картошка.

Фермер может выполнять следующие действия:

- покупать животных, зерна;
- кормить животных;
- собирать продукцию, урожай;
- улучшать ферму (приобретать дополнительные сельскохозяйственные поля, загоны для сельскохозяйственных животных).

Животные имеют стадии взросления, культуры – стадии созревания.

Начальные условия: 100 монет, 1 курица, 2 саженца пшеницы, 5 полей под посадку.

Время под посадку необходимо задать самостоятельно.

Реализуйте смену дня и ночи.

Работа, которую выполняет фермер выбирается с помошью консоли.

Цель: улучшить ферму на максимум и заработать 100 000 монет.

Контрольные вопросы к разделу «Основы Python»

1. Дайте определение понятию «объект» в контексте языка Python.
2. Объясните значение характеристики «динамически типизируемый» относительно языка программирования.
3. Перечислите названия нескольких встроенных типов объектов. Приведите примеры объектов таких типов.
4. Что из себя представляет преобразование типов?
5. Дайте определение понятию «переменная».
6. Каким образом происходит создание переменной?
7. Какие существуют правила при выборе имени переменной?
8. Дайте определение понятиям «оператор», «операнд», «операция».
9. Что подразумевает приоритет выполнения операций?
10. Каким образом можно изменить приоритет операций в выражении?
11. Перечислите названия нескольких математических операторов. Приведите примеры их использования.
12. Перечислите названия нескольких логических операторов. Приведите примеры их использования.
13. Что из себя представляет оператор присваивания?
14. Что из себя представляют расширенные операторы присваивания?
15. Что из себя представляет встроенные функции Python?
16. Перечислите названия нескольких встроенных функций Python. Приведите примеры их использования.
17. Дайте определение понятию «строка».
18. Что из себя представляют и для чего используются управляющие последовательности (последовательности экранирования) строки?
19. Что из себя представляют и для чего используются f-строки?
20. Что из себя представляет конкатенация строк?
21. Что из себя представляет операция «повторение» относительно строки?
22. Что из себя представляет операция «проверка на вхождение» относительно строки?
23. Что подразумевается под индексацией элементов строки?
24. Дайте определение понятиям «срез», «нарезание» относительно строки.
25. Перечислите несколько методов строк. Объясните принцип их функционирования.
26. Дайте определение понятию «модуль».
27. Каким образом происходит подключение модуля и обращение к элементам модуля?
28. Перечислите несколько элементов модуля `math`. Объясните их назначение.

29. Дайте определение понятию «список».
30. Каким образом можно создать пустой список?
31. Каким образом можно создать список из нескольких элементов?
32. Каким образом можно в пустой список добавить несколько элементов?
33. Что из себя представляет конкатенация списков?
34. Что из себя представляет операция «повторение» относительно списка?
35. Что из себя представляет операция «проверка на вхождение» относительно списка?
36. Чем похожи и чем различаются типы данных «строка» и «список»?
37. Перечислите несколько методов списка. Объясните принцип их функционирования.
38. Что подразумевает упаковка и распаковка элементов списка?
39. Каким образом можно осуществить проход по элементам списка?
40. Что из себя представляет условный оператор `if`?
41. Что из себя представляет конструкция «if-elif-else»?
42. Что из себя представляет блок инструкций в Python? Для чего используется табуляция в блоке инструкций?
43. Каким образом можно проверить, является ли список пустым, с помощью оператор `if`?
44. Дайте определение понятию «цикл».
45. Что из себя представляет цикл `while`?
46. Приведите простую форму цикла `while`, включающую только обязательные части.
47. Для чего используется конструкция `else` цикла?
48. Для чего используется оператор `break` в цикле?
49. Для чего используется оператор `continue` в цикле?
50. Для чего может использоваться оператор `pass` в цикле?
51. Приведите полную форму цикла `while`.
52. В каком случае цикл `while` будет выполняться «бесконечно»?
53. Что из себя представляет цикл `for`?
54. Приведите простую форму цикла `for`, включающую только обязательные части.
55. Приведите полную форму цикла `for`.
56. В каких случаях стоит использовать цикл `while`, а в каких – цикл `for`?
57. Дайте определение понятию «функция» относительно языка программирования.
58. Каким образом происходит создание функции?
59. Приведите общую форму записи для создания функции.
60. Каким образом происходит вызов функции?
61. Для чего используются аргументы функции?

62. Каким образом происходит передача аргументов при вызове функции?
63. Каким образом для аргумента указать значение по умолчанию?
64. Что подразумевается под приостановкой кода при вызове функции?
65. Что подразумевается под областью видимости переменных?
66. Дайте определение понятию «глобальная переменная».
67. Дайте определение понятию «локальная переменная».
68. Перечислите несколько операторов, которые используются при создании функции. Объясните их назначение.
69. Для чего используется оператор `def`?
70. Для чего используется оператор `return` при создании функции?
71. Для чего используется оператор `global` при создании функции?
72. Для чего используется оператор `nonlocal` при создании функции?
73. Для чего используется оператор `lambda`?
74. Для чего используется оператор `yield` при создании функции?
75. Для чего используется оператор `pass` при создании функции?
76. Что подразумевается под объектно-ориентированным программированием?
77. Дайте определение понятию «класс».
78. Дайте определение понятию «объект».
79. Дайте определение понятию «наследование».
80. Дайте определение понятию «инкапсуляция».
81. Дайте определение понятию «полиморфизм».
82. Каким образом происходит определение класса? Приведите общую форму записи при определении класса.
83. Каким образом происходит определение экземпляра (объекта) класса?
84. Что подразумевается под атрибутом класса?
85. Что подразумевается под методом класса?
86. Есть ли разница между атрибутом класса и атрибутом объекта?
87. Для чего используется одинарное подчеркивание в начале имени вида `_name` при определении класса?
88. Для чего используется двойное подчеркивание в начале имени вида `__name` при определении класса?
89. Что подразумевают под публичными и приватными атрибутами?
90. Что из себя представляет и для чего используются методы-свойства?
91. Что из себя представляет и для чего используется геттер (getter)?
92. Что из себя представляет и для чего используется сеттер (setter)?
93. Что из себя представляет и для чего используется делитер (deleter)?
94. Что подразумевают под аннотацией свойств (использованием деструкторов для функций)?
95. Дайте определение понятиям «базовый класс» и «производный класс».
96. Каким образом реализуется простое наследование?

97. Каким образом реализуется множественное наследование?
98. Каким образом происходит наследование метода при множественном наследовании в случае, если этот метод определен в нескольких базовых классах, назван одним и тем же именем, но содержит разные инструкции?
99. Что подразумевают под переопределением унаследованного функционала?
100. Каким образом можно обратиться к функционалу базового класса из тела производного класса?
101. Для чего используется ключевое слово `self`?
102. Для чего используется функция `super()`?

Раздел 2
Структуры данных. Алгоритмы поиска и сортировки

Практическое задание № 7
«Линейный и бинарный поиск. Сложность алгоритма»

1. Определение алгоритма

Алгоритм – последовательность действий.

Алгоритм – последовательность этапов, описывающая способ решения задачи.

Алгоритм – упорядоченная и конечная последовательность недвусмысленных, выполнимых действий для решения поставленной задачи.

Алгоритм – упорядоченный набор из недвусмысленных и выполнимых этапов, определяющий некоторый конечный процесс [2].

Характеристики (свойства) алгоритма:

1. Упорядоченность – этапы алгоритма выполняются последовательно, по порядку (этап 1, этап 2, этап 3, …).

2. Выполнимость – все этапы алгоритмы должны быть выполнимыми (пример выполнимого этапа: сложить числа 2 и 4; пример невыполнимого этапа: разделить 10 на 0).

3. Недвусмысленность (однозначность, детерминированность, определенность) – все этапы должны выполнятся однозначно (пример недвусмысленного этапа: сложить числа 2 и 4; пример двусмысленного, многозначного этапа: провести арифметическую операцию над числами 2 и 4).

4. Конечность – число этапов алгоритма должно быть конечным.

Дополнительные свойства:

*Дискретность (прерывность, разрывность) – алгоритм разделен на части, называемые этапами или шагами.

*Массовость – применимость алгоритма к решению таких же или похожих задач, но с другими исходными данными.

*Результативность – выполнимость алгоритма за конечное число шагов (2. Выполнимость + 4. Конечность).

*Формальность – исполнитель алгоритма последовательно выполняет этапы, не вдаваясь в их смысл (формальное исполнение этапов по инструкции).

Абстрактная природа алгоритма (алгоритм, процесс, программа)

Алгоритм является абстракцией. Алгоритм, как и всякую абстракцию, можно представлять различными способами. Представлением одного и того же алгоритма может быть:

- текст в виде списка этапов (инструкция),
- алгебраическая формула,
- блок-схема,
- программа.

Алгоритм, процесс и программа – это различные, но взаимосвязанные понятия [2].

Программа – представление алгоритма.

Процесс – деятельность, связанная с выполнением программы.

Процесс – деятельность по выполнению алгоритма.

2. Алгоритм линейного поиска

Алгоритм поиска – алгоритм, получающий на вход список элементов и искомый элемент, который необходимо найти в этом списке элементов. Если элемент присутствует в списке, то алгоритм поиска возвращает ту позицию, в которой он был найден (индекс искомого элемента). В противном случае алгоритм поиска возвращает `null` [4].

Линейный поиск (последовательный, простой поиск) (Linear search, Sequential search) – алгоритм поиска, в соответствии с которым поиск искомого элемента осуществляется путем перебора и сравнения каждого элемента с искомым элементом до тех пор, пока искомый элемент не будет найден.

3. Алгоритм бинарного поиска

Бинарный поиск (двоичный поиск / «метод деления пополам» / «дихотомия») (Binary search) – алгоритм поиска, в соответствии с которым поиск искомого элемента осуществляется следующим образом: в отсортированном списке элементов на каждом шаге происходит переход к центру списка, затем удаление половины списка (слева или справа от центрального элемента) до тех пор, пока элемент не будет найден.

Ограничение: может работать только с отсортированным списком элементов.

Визуализация алгоритма бинарного поиска представлена на рисунке 7.1.

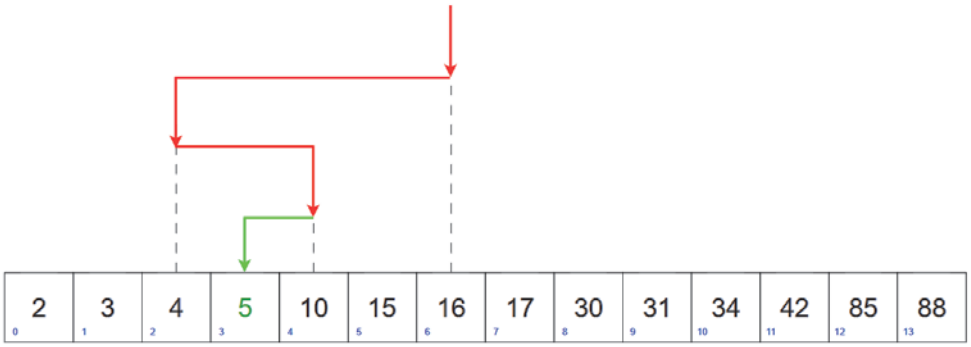


Рисунок 7.1 – Визуализация работы алгоритма бинарного поиска (число 5 – искомый элемент)

4. Нотация «О-большое»

Сложность алгоритма в худшем случае – максимальное число операций, которые необходимо выполнить алгоритму.

Сложность алгоритма поиска в худшем случае – максимальное число элементов, которые необходимо перебрать, чтобы найти нужный элемент.

Сложность линейного поиска в худшем случае: n.

Сложность бинарного поиска в худшем случае: $\log_2 n$.

Время выполнения алгоритма в худшем случае – темп роста времени выполнения алгоритма при увеличении числа операций.

Время выполнения алгоритма поиска в худшем случае – вид функции роста времени выполнения алгоритма при увеличении размера массива (время выполнения может быть: линейным, логарифмическим и др.).

Время выполнения алгоритма линейного поиска в худшем случае: линейное.

Время выполнения алгоритма бинарного поиска в худшем случае: логарифмическое.

О-большое (Big-O) – специальная нотация для обозначения скорости выполнения алгоритма (не в секундах). Показывает, на сколько быстро возрастает время выполнения алгоритма при увеличении числа операций, которые необходимо будет выполнить алгоритму. Применяется для сравнения скорости алгоритмов, а не для определения времени выполнения алгоритма в секундах [4].

О-большое записывается, в общем случае, следующем образом:

$$O(n),$$

где n – количество операций.

О-большое определяет время выполнения алгоритма в худшем случае.

О-большое используется для классификации алгоритмов в соответствии с тем, как их требования ко времени выполнения или пространству растут по мере увеличения количества входных данных (n).

Время выполнения алгоритма простого поиска в худшем случае: $O(n)$.

Время выполнения алгоритма бинарного поиска в худшем случае: $O(\log n)$.

Графики сложности алгоритмов в нотации **О-большое** представлены на рисунке 7.2.

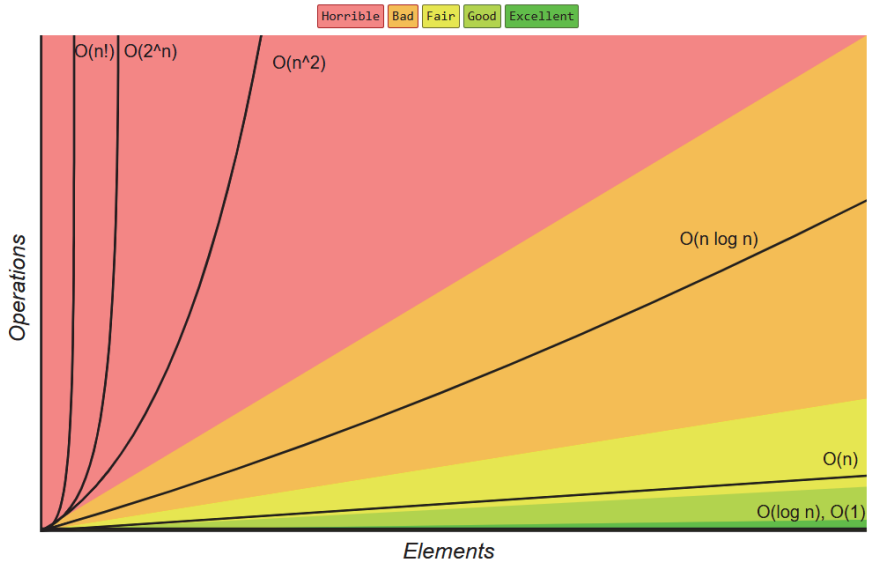


Рисунок 7.2 – Графики сложности алгоритмов в нотации Big-O (Big-O Complexity Chart)

Источник: [12].

Задачи

Задача № 1

Охарактеризуйте различия между процессом, алгоритмом и программой.

Задача № 2

Приведите примеры алгоритмов, с которыми вы знакомы. Действительно ли они являются алгоритмами в строгом смысле этого слова?

Задача № 3

Каким пунктам в определении алгоритма не соответствует приведенная ниже последовательность инструкций?

```
Этап 1. Возьмите монету из вашего кармана и положите ее на стол.
Этап 2. Возвратитесь к этапу 1.
```

Задача № 4

Приведите время выполнения «О-большое» для каждого из следующих сценариев:

a. Известна фамилия, нужно найти номер в телефонной книге.

b. Известен номер, нужно найти фамилию в телефонной книге (Подсказка: вам придется провести поиск по всей книге).

c. Нужно прочитать телефоны всех людей в телефонной книге.

Задача № 5

Для отсортированного списка из 2048 элементов за сколько шагов «в худшем случае» будет выполнен простой поиск и бинарный поиск. Предположим, что размер списка увеличился вдвое. Как изменится максимальное количество проверок?

Задача № 6

Имеется отсортированный список `a`, состоящий из нулей и единиц:

```
a = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

Необходимо найти место, где заканчиваются нули и начинаются единицы, используя бинарный поиск.

Разработайте программу, которая возвращает индекс первой единицы.

Дополнительно:

a. Отобразите промежуточные результаты работы алгоритма (см. методические указания к задаче № 6).

b. Решите задачу, используя линейный поиск.

Задача № 7

Решите с помощью бинарного поиска следующие задачи:

a. Найти первое число, равное X в отсортированном массиве, или вывести, что таких чисел нет.

b. Найти последнее число, равное *X* в отсортированном массиве, или вывести, что таких чисел нет.

c. Посчитать, сколько раз встречается число *X* в отсортированном массиве (в решении помогают два предыдущих пункта).

d. Дан массив чисел, первая часть состоит из нечетных чисел, а вторая - из четных. Найти индекс, начиная с которого все числа четные.

Необходимо придумать отсортированный массив размера n для решения задачи.

Задача № 8

Дано два массива *A* и *B*, элементами которых являются целые числа. Массив *A* отсортирован в порядке убывания.

Для каждого элемента массива *B* найти наиболее близкое число к данному в массиве *A*. Если таких несколько, то вывести оба.

Пример:

```
A = [65, 43, 23, 11, 7]
B = [3, 54, 23, 9, 65]
```

В результате получаем:

```
3 - 7
54 - 65 43
23 - 23
9 - 11 7
65 - 65
```

Необходимо придумать два массива. Они не должны повторяться. Алгоритм может быть реализован средствами любого языка программирования или в блок-схемах.

Методические указания

Пример решения Задачи № 6

Решение подобной задачи может быть осуществлено следующим образом.

На первом шаге возьмем элемент в центре массива. Если этот элемент – 0, то первую единицу необходимо искать в правой половине массива (т.к. слева будут только нули), если этот элемент – 1, то первую единицу необходимо искать в левой части массива (т.к. справа будут только единицы).

«Отбрасывание» половины массива на каждом шаге можно реализовать через две переменные, хранящие два индекса для рабочего диапазона массива, т.е. диапазона на каждом шаге бинарного поиска.

Определим две переменные для хранения индексов. Пусть переменная `left` всегда указывает на 0 в массиве, а переменная `right` всегда указывает на 1 в массиве. Т.к. изначально рабочий диапазон – это весь массив, то переменным `left` и `right` изначально можно присвоить значения 0 и $n-1$ соответственно. Однако `a[0]` может быть единицей, а `a[n-1]` может быть нулём. Чтобы избежать подобной проблемы, изначально возьмем выходящие за пределы индексы:

```
left = -1
right = len(a)
```

Через цикл реализуем поэтапное «смещение» переменных `left` и `right` друг к другу до тех пор, пока они не станут соседними индексами, показывающими, где заканчиваются нули (на индексе `left`) и начинаются единицы (на индексе `right`).

Первоначально `left` и `right` указывают на несуществующие индексы. Однако в конце алгоритма хотя бы один из индексов поменяет значение. Например, в массиве `[1, 1, 1, 1]` в конце алгоритма переменные будут иметь следующие значения: `left = -1`, `right = 0`.

Перейдем к написанию программного кода.

Объявим исходный массив и отобразим его в консоли. Объявим переменные `left` и `right`, каждая из которых будет содержать индекс одного элемента массива:

```
10  # Исходный массив
11  a = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1] # Массив
12  print('Исходный массив: ', a)        # Вывод массива в консоль
13
14  # Индексы, указывающие на 0 и 1
15  left = -1       # Переменная left = -1 (будет указывать на 0)
16  right = len(a)  # Переменная right = 14 (будет указывать на 1)
```

Через цикл `while` найдем индексы на границе нулей и единиц. Цикл `while` имеет следующую форму записи:

```
while условие:
  тело_цикла
```

Тело цикла выполняется до тех пор, пока выполняется условие.

Напишем цикл `while`:

```
18  # Поиск через цикл while
19  # Пока разница между индексами > 1
20  # (между индексами left и right есть хотя бы еще 1 индекс)
21  while right - left > 1:
22      middle = (left + right) // 2 # Средний индекс между left и right
23      # Если средний индекс = 1, то сдвиг right влево (к середине)
24      if a[middle] == 1:
25          right = middle # right всегда должна указывать на 1
26      # Иначе (если средний индекс = 0), то сдвиг left вправо (к середине)
27      else:
28          left = middle # left всегда должна указывать на 0
```

Здесь в цикле `while` проверяется условие: `right - left > 1`. Если разница больше 1, то выполняется тело цикла, в котором переменная `right` или переменная `left` изменяется (строки 25, 28). После выполнения тела цикла (строки 22-28), происходит возвращение на строку 21 и условие `right - left > 1` проверяется вновь, уже с новым значением переменной `right` или `left`.

Примечание: если бы мы не изменяли переменные right и left, то цикл while выполнялся бы бесконечно, что могло привести к зависанию компьютера. Поэтому рекомендуется сохранять все открытые файлы перед компиляцией программы с циклом while.

В теле цикла происходит поиск среднего индекса на строке 22 через оператор `//` – деление без остатка, затем происходит отбрасывание половины элементов массива (смещение индексов `left` и `right` к границе нулей и единиц).

Выведем результаты в консоль:

```
30  # Отображение результатов
31  print('Индексы на границе 0 и 1: ', left, right)
32  print('Элементы массива с этими индексами: ', a[left], a[right])
```

```
Задание №3

Исходный массив:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
Индексы на границе 0 и 1:  8 9
Элементы массива с этими индексами:  0 1
```

Задача решена для нулей и единиц, но решение может быть обобщено для абсолютно любой задачи, где есть какое-то **свойство, которое в начале массива не выполняется, а потом выполняется**.

Например, если необходимо найти, есть ли число *X* в отсортированном массиве, то мы просто представим, что 0 – это числа, меньшие *X*, а 1 – это числа, большие или равные *X*. Тогда достаточно найти первую «единицу» и проверить, равно ли это число *X*.

Примерный программный код для решения такой задачи может быть следующим:

```
a = [1, 3, 4, 10, 10, 10, 11, 80, 80, 81] # отсортированный массив
def bin_search(a, x):
    n = len(a)
    left = -1
    right = n
    while right - left > 1:
        middle = (left + right) // 2
        if a[middle] >= x: # практически единственная строка, которая меняется от задачи к задаче
            right = middle
        else:
            left = middle
    if right != n and a[right] == x: # ответ лежит в right
        return True
    else:
        return False

print (bin_search(a, 1))
print (bin_search(a, 10))
print (bin_search(a, 20))
print (bin_search(a, 79))
print (bin_search(a, 80))
```

Самостоятельно попробуйте преобразовать код к задаче 6 таким образом, чтобы выводилась подобная информация:

```
Задача №3

Исходный массив:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
left =  -1
right = 14

Шаг 0
Индекс среднего элемента: 6
Средний элемент: 0
left =  6
right = 14
Новый рабочий диапазон: [0, 0, 0, 1, 1, 1, 1, 1]
Шаг 1
Индекс среднего элемента: 10
Средний элемент: 1
left =  6
right = 10
Новый рабочий диапазон: [0, 0, 0, 1, 1]

Шаг 2
Индекс среднего элемента: 8
Средний элемент: 0
left =  8
right = 10
Новый рабочий диапазон: [0, 1, 1]
```

```
Шаг 3
Индекс среднего элемента: 9
Средний элемент: 1
left =  8
right = 9
Новый рабочий диапазон: [0, 1]

Результат найден за 4 шага (шагов)
Индексы на границе 0 и 1:  8 9
Элементы массива с этими индексами:  0 1
left =  8
right = 9
Индекс, начиная с которого идут единицы:  9
Элемент с этим индексом:  1
```

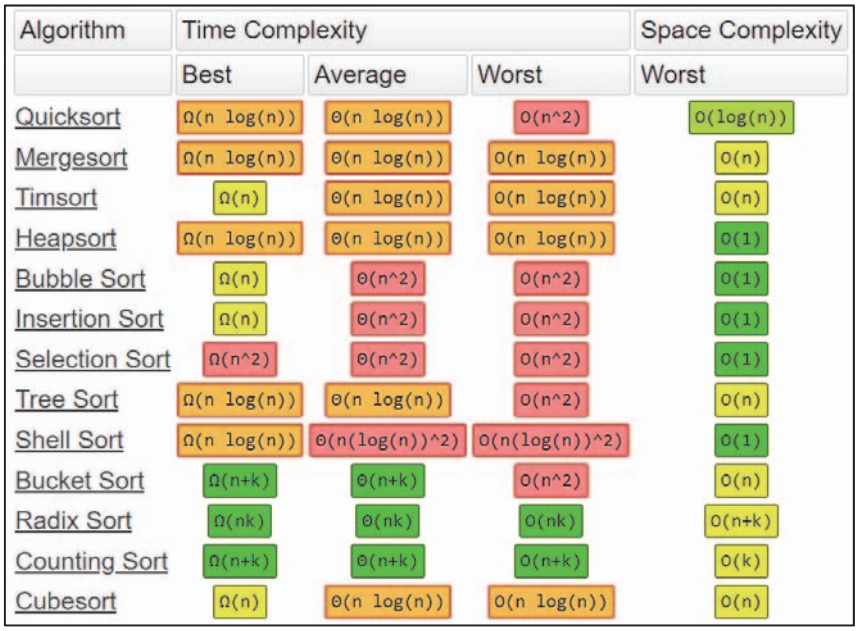
Практическое задание № 8
«Сортировка выбором и пузырьком. Оценка итоговой сложности программы»

1. Алгоритмы сортировки: сортировка выбором, сортировка пузырьком

Многие **алгоритмы поиска** работают только с отсортированными данными. Поэтому, чтобы применять такие алгоритмы поиска, необходимо сначала отсортировать данные с помощью **алгоритмов сортировки**.

Сортировка подразумевает изменение порядка элементов в массиве путем их перестановки.

Существует множество алгоритмов сортировки. Основные из них представлены на рисунке 8.1.



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Рисунок 8.1 – Сложность основных алгоритмов сортировки (Array Sorting Algorithms) в нотации Big-O

Источник: [12].

Ниже даны определения алгоритмам сортировки для случая сортировки по возрастанию. Для случая сортировки по убыванию в данных определениях изменится слово «минимальный» на «максимальный», а слово «максимальный» на «минимальный».

1.1. Сортировка выбором (простая сортировка) (Selection sort) *(реализация через 2 массива)* – повторяющиеся *n* раз поиск минимального значения в массиве и перемещение его в новый массив.

Сортировка выбором *(реализация через 1 массив)* – повторяющиеся *n* раз:

1) поиск минимального значения в массиве,

2) обмен значениями между двумя элементами: первым элементом в рабочем диапазоне и найденным элементом с минимальным значением,
3) исключение первого элемента из рабочего диапазона (сокращение рабочего диапазона вправо).

Время выполнения алгоритма сортировки выбором: $O(n^2)$.

1.2. Сортировка пузырьком (сортировка простыми обменами) (Bubble sort) – перемещение максимальных значений вправо путем попарного сравнения каждого элемента массива с соседним элементом справа.

Время выполнения алгоритма пузырьковой сортировки: $O(n^2)$.

2. Сложность операций со списком и кортежем

В Python имеются коллекции (структуры данных), операции над которыми имеют определенную сложность.

Большинство операций со списком/кортежем имеют сложность $O(n)$. (Таблица 8.1).

Таблица 8.1 – Асимптотические сложности для списка или кортежа

Операция	Сложность	Примечание
`len(lst)`	$O(1)$	
`lst.append(5)`	$O(1)$	
`lst.pop()`	$O(1)$	Аналогично `lst.pop(-1)`
`lst.clear()`	$O(1)$	Аналогично `lst = []`
`lst[a:b]`	$O(b-a)$	
`lst.extend(...)`	$O(len(\ldots))$	Зависит от длины аргумента
`list(...)`	$O(len(\ldots))$	Зависит от длины аргумента
`lst1 == lst2`	$O(n)$	
`lst[a:b] = ...`	$O(n)$	
`del lst[i]`	$O(n)$	
`lst.remove(...)`	$O(n)$	
`x in/not in lst`	$O(n)$	Поиск в списке
`lst.copy()`	$O(n)$	Аналогично `lst[:]`
`lst.pop(i)`	$O(n)$	
`min(lst)/max(lst)`	$O(n)$	
`lst.reverse()`	$O(n)$	
`for v in lst:`	$O(n)$	
`lst.sort()`	$O(n \log n)$	Направление сортировки не играет роли
`k * lst`	$O(k \cdot n)$	

Источник: [7].

3. Оценка сложности совокупности алгоритмов по О-нотации

Для оценки сложности совокупности операций используются законы сложения и умножения.

3.1. Закон сложения: итоговая сложность двух последовательных действий равна сумме их сложностей [7].

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$

Особенности:

1) Итоговая сложность алгоритма оценивается наихудшим из слагаемых:

$$O(n) + O(logn) = O(n + logn) = O(n)$$

2) В итоговой сложности константы отбрасываются:

$$O(N) + O(N) + O(N) = 3 \cdot O(N) = O(N)$$

3) При ветвлении берется наихудший вариант:

```
if test:     # O(t)
    block 1  # O(b1)
else:
    block 2  # O(b2)
```

$$O(N) = O(t) + max(O(b1), O(b2))$$

3.2. Закон умножения: итоговая сложность двух вложенных действий равна произведению их сложностей [7]:

$$O(f(n)) + O(g(n)) = O(f(n) \cdot g(n))$$

```
# Общая сложность:        O(N^2)
for i in range(N):      # O(N)
    for j in range(N):  # O(N)
```

Пример определения итоговой сложности функции представлен в листинге 8.1.

Листинг 8.1 – Пример определения итоговой сложности функции [7]

```python
def is_unique_1(data):
    """Вернуть 'True', если все значения 'data' различны.

    Алгоритм 1:
      Сделаем проход по списку с первого до последнего
      элемента и для каждого из них проверим, что такого
      элемента нет в оставшихся справа элементах.

    Сложность: O(N^2).
    """
    for i in range(len(data)):        # O(N)
        if data[i] in data[i+1:]:     # O(N) - срез + in: O(N) + O(N) = O(N)
            return False              # O(1) - в худшем случае не выполнится
    return True                       # O(1)
```

Задачи

Задача № 1

Определите итоговую сложность следующей программы:

```python
def showText(text):
    print(text)

text = 'Hello, World!'
showText(text = text)
```

Задача № 2

Определите итоговую сложность следующей программы:

```python
list1 = [10, 15, 20, 25, 500]
for element in list1:
    print(element, end='\t')
for i in range(5):
    print(list1[i])
```

Задача № 3

Определите итоговую сложность следующей программы:

```python
list1 = [10, 15, 20, 25, 500]
n = len(list1)
for i in range(n):
    el1 = [list1[i]]
    for j in range(n):
        el2 = [list1[j]]
        if el1 != el2:
            print(el1, '--', el2)
```

Задача № 4

Пусть имеется список: [6, 12, 4, 3, 8].

Необходимо отсортировать данный список в порядке возрастания, применив **алгоритм сортировки выбором**.

Решить задачу следующими способами:

1. Простое описание алгоритма сортировки.
2. Создание функции, внутри которой применяется только цикл `for`.
3. Создание функции, внутри которой применяется только цикл `while`.
4. Создание функции, реализующий **алгоритм сортировки выбором** через 2 массива.
5. Создание функции, реализующий **алгоритм сортировки выбором** через 1 массив.

*Выберете одну из созданных или создайте новую функцию, реализующую **алгоритм сортировки выбором**. Данную функцию необходимо будет использовать при решении Задачи № 6.*

Задача № 5

Необходимо отсортировать список из Задачи № 4 в порядке возрастания с помощью **алгоритма сортировки пузырьком**.

Решить задачу следующими способами:

1. Простое описание алгоритма сортировки.
2. Создание функции, внутри которой применяется только цикл `for`.
3. Создание функции, внутри которой применяется только цикл `while`.

*Выберете одну из созданных или создайте новую функцию, реализующую **алгоритм сортировки пузырьком**. Данную функцию необходимо будет использовать при решении Задачи № 6.*

Задача № 6

Модифицируйте программный код к двум основным созданным **функциям сортировки** из Задач № 4, 5 таким образом, чтобы в консоль опционально (по выбору через дополнительный аргумент, например `verbose = True / False`) выводилась следующая информация:

- исходный массив,
- шаг алгоритма,
- измененный массив на этом шаге.

После завершения выполнения алгоритма должна выводится информация о том, сколько шагов потребовалось для сортировки, также должен выводится отсортированный массив.

Задача № 7

Модифицируйте программный код к двум **функциям сортировки** из Задачи № 6 таким образом, чтобы функции реализовывали как сортировку по возрастанию, так и сортировку по убыванию.

Задача № 8

Протестируйте две созданные **функции сортировки** из задачи № 7 на массивах из:

a. 20 элементов,
b. 50 элементов.

Массив случайных целых чисел в определенном диапазоне можно получить с помощью библиотеки numpy.

Пример создания списка из 20 случайных целых чисел в диапазоне от 0 до 10:

```
import numpy as np
a1 = list(np.random.randint(low=0, high=10, size=20, dtype=int))
In [9]: a1
Out[9]: [2, 0, 4, 2, 7, 7, 0, 9, 3, 9, 8, 0, 2, 5, 7, 2, 4, 0, 5, 3]
```

Задача № 9

Имеется отсортированный массив целых чисел L1, состоящий из 50 элементов в диапазоне значений от 0 до 100.

Необходимо осуществить поиск элемента со значением 10.

Определить итоговую сложность предложенной программы.

Решить задачу, используя только рассмотренные алгоритмы поиска и сортировки (линейный поиск, бинарный поиск, сортировка выбором, сортировка пузырьком).

Предложить наиболее эффективное решение (итоговая сложность программы должна быть минимальной).

Задача № 10

Имеется неотсортированный массив целых чисел L1, состоящий из 50 элементов в диапазоне значений от 0 до 100.

Необходимо осуществить поиск элемента со значением 10.

Определить итоговую сложность предложенной программы.

Решить задачу, используя только рассмотренные алгоритмы поиска и сортировки (линейный поиск, бинарный поиск, сортировка выбором, сортировка пузырьком).

Предложить наиболее эффективное решение (итоговая сложность программы должна быть минимальной).

Задача № 11

Имеется неотсортированный массив целых чисел L1, состоящий из 50 элементов в диапазоне значений от 0 до 100.

Необходимо выполнить следующие операции:

a. вывести в консоль первый элемент.
b. найти минимальное значение.
c. найти максимальное значение.
d. осуществить поиск элемента со значением 10.

Определить итоговую сложность предложенной программы.

Решить задачу, используя только рассмотренные алгоритмы поиска и сортировки (линейный поиск, бинарный поиск, сортировка выбором, сортировка пузырьком).

Предложить наиболее эффективное решение (итоговая сложность программы должна быть минимальной).

Методические указания

Пример решение Задачи № 5

1. Простое описание алгоритма сортировки

Распишем все шаги на пути к отсортированному списку.

За 1-ю итерацию внешнего цикла число «12» переместится в конец. Это потребует 4 сравнения во внутреннем цикле:

6 > 12? Нет

12 > 4? Да. Меняем местами

12 > 3? Да. Меняем

12 > 8? Да. Меняем

Получим следующий результат: **[6, 4, 3, 8, 12].**

За 2-ю итерацию внешнего цикла число «6» сместиться на 3 место с конца, что потребует 3 сравнения:

6 > 4? Да. Меняем местами

6 > 3? Да. Меняем

6 > 8? Нет

Получим следующий результат: **[4, 3, 6, 8, 12].**

На 3-й итерации внешнего цикла меняются местами два первых элемента. Количество итераций внутреннего цикла равно 2:

4 > 3? Да. Меняем местами

4 > 6? Нет

Получим следующий результат: **[3, 4, 6, 8, 12].**

На 4-й итерации внешнего цикла осталось сравнить только первые два элемента, поэтому количество итераций внутреннего равно единице:

3 > 4? Нет

Получим следующий результат: **[3, 4, 6, 8, 12].**

Программная реализация алгоритмов сортировки

Ниже приведены примеры для реализации алгоритма сортировки выбором и алгоритма сортировки пузырьком.

Рекомендуется попытаться самостоятельно реализовать данные алгоритмы и лишь в случае затруднения или для самопроверки обратиться к примерам.

Сортировка выбором

Способ № 1

```
# Сортировка выбором. Способ № 1

# Переменные
a_old = a.copy()     # Копирование массива a
a_sort = []          # Отсортированный массив
n = len(a_old)       # Размер массива a

# Перемещение каждого мин. элемента в новый массив
for i in range(n):

    # Поиск очередного минимального элемента
    n = len(a_old)                # Получение нового размера массива a
    min_value = a_old[0]          # Предположение, что первый элемент - мин.
    min_value_idx = 0             # Индекс мин. элемента
    for j in range(1, n):         # Поиск мин. элемента
        if a_old[j] < min_value:
            min_value = a_old[j]
            min_value_idx = j

    # Изменение массивов
    a_sort.append(min_value) # Добавление в массив a_sort мин. элемент
    a_old.pop(min_value_idx)     # Удаление из массива a мин. элемент

print('\nСортировка выбором. Способ №1')
print(a)
print(a_old)
print(a_sort)
```

Способ № 2 (через функции)

```
# Сортировка выбором. Способ № 2 (через функции)

# Исходный массив
a = [6, 12, 4, 3, 8]    # Массив
n = len(a)              # Размер массива (n = 5)

# Функция нахождения индекса минимального элемента в массиве arr
def getIndexOfMin(arr):
    n = len(arr)            # Рамзер массива arr
    min_idx = 0             # Индекс мин. элемента
    min_val = arr[min_idx]  # Минимальный элемент
    # Поиск мин. элемента
    for i in range(n):
        if arr[i] < min_val:
            min_idx = i
    # Возвращаемое значение - индекс мин. элемента
    return min_idx
```

```
65  # Функция сортировки выбором
66  def selectionSort(arr):
67      n = len(arr)              # Размер массива arr
68      arr_sort = []             # Отсортированный массив
69      for i in range(n):
70          min_idx = getIndexOfMin(arr)      # Индекс мин. элемента
71          arr_sort.append(arr.pop(min_idx)) # Перемещение из arr в arr_sort
72      return arr_sort
73
74  print('\nСортировка выбором. Способ №2 (через функции)')
75  print(a)
76  print(selectionSort(a))
```

Сортировка пузырьком

Способ № 1 (через цикл for)

```
85  # Сортировка пузырьком. Способ № 1 (через цикл for)
86
87  # Исходный массив
88  a = [6, 12, 4, 3, 8]     # Массив
89  n = len(a)               # Размер массива (n = 5)
90
91  # Сортировка
92  # Каждый элемент смещается вправо (если больше следующего)
93  for i in range(n - 1):
94      # Внутренний цикл работает с конкретным элеменом на каждой итерации
95      # Уменьшающийся диапазон (последние элементы не рассматриваются)
96      for j in range(n - i - 1):
97          # Если текущий элемент > следующий элемент
98          if a[j] > a[j + 1]:
99              aj_old_value = a[j]      # Сохранение текущего значения a[j]
100             a[j] = a[j + 1]          # Перемещение a[j + 1] влево
101             a[j + 1] = aj_old_value  # Перемещение a[j] вправо
102
103 print('\nСортировка пузырьком. Способ № 1 (через цикл for)')
104 print(a)
```

Способ № 2 (через цикл while)

```
108 # Сортировка пузырьком. Способ № 2 (через цикл while)
109
110 # Исходный массив
111 a = [6, 12, 4, 3, 8]     # Массив
112 n = len(a)               # Размер массива (n = 5)
113
114 # Сортировка
115 i = 0
116 while i < (n - 1):
117     j = 0
118     while j < (n - 1 - i):
119         if a[j] > a[j + 1]:
120             a[j], a[j + 1] = a[j + 1], a [j] # Двойное присваивание (обмен, swap)
121         j += 1                               # Идентично j = j + 1
122     i += 1
123
124 print('\nСортировка пузырьком. Способ № 2 (через цикл while)')
125 print(a)
```

Практическое задание № 9
«Рекурсия. Стек»

1. Рекурсия

1.1. Основные понятия

Рекурсия (от лат. *recursio* «круговорот, возврат») – определение объекта с использованием самого этого объекта.

Пример рекурсии представлен на рисунке 9.1.

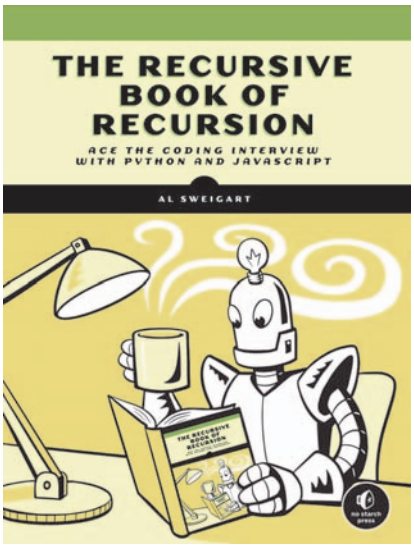


Рисунок 9.1 – Пример рекурсии

Источник: обложка книги [25].

Рекурсия – вызов функции самой себя.

Рекурсивная функция – функция, вызывающая сама себя.

Рекурсия является альтернативой циклам. Зачастую, код с циклами работает быстрее, чем код с рекурсией. Однако рекурсия в отдельных случаях облегчает написание программы и применяется во многих эффективных алгоритмах.

Знать о рекурсии полезно, т.к. она позволяет программам обходить структуры, которые имеют произвольные и непредсказуемые формы и глубины, например, при планировании маршрутов в путешествии, анализе языка и прохождении по ссылкам в веб-сети.

Рекурсивная функция состоит из 2 частей [4]:

- Базовый случай.
- Рекурсивный случай.

Базовый случай – случай, когда функция не вызывает сама себя, чтобы предотвратить зацикливание (бесконечный вызов).

Рекурсивный случай – случай, когда функция вызывает сама себя.

Пример функции суммирования, в которой применяется рекурсия представлен в листинге 9.1.

Листинг 9.1 – Пример функции суммирования, в которой применяется рекурсия

```
1  def recursive_sum(arr):
2
3      # Базовый случай
4      if not arr:
5          return 0
6
7      # Рекурсивный случай
8      else:
9          return arr[0] + recursive_sum(arr[1:])
10
11 a = [10, 5, 5, 20]
12 print(recursive_sum(a))
```

```
40
```

На каждом уровне функция `recursive_sum()` рекурсивно вызывает саму себя (строка 9 листинга 9.1), чтобы вычислить сумму остатка списка `arr[1:0]`, которая позже добавляется к первому элементу списка `arr[0]` (на каждом уровне список сокращается на один элемент слева). Когда список становится пустым (условие на строке 4 листинга 9.1 выполняется), рекурсивный цикл заканчивается и возвращается ноль.

В случае использования рекурсии такого рода каждый открытый уровень вызова функции имеет собственную копию локальной области видимости функции в **стеке вызовов** времени выполнения – здесь это означает, что переменная `arr` на каждом уровне разная [5].

Для лучшего понимания работы рекурсивной функции рекомендуется выводить в консоль промежуточные результаты на каждом уровне вызова с помощью функции `print()` (листинг 9.2.)

Листинг 9.2 – Пример рекурсивной функции суммирования, в которой происходит вывод в консоль списка элементов на каждом уровне вызова

```
1  def recursive_sum(arr):
2      print(arr)   # Трассировка уровней рекурсии
3      if not arr:
4          return 0
5      else:
6          return arr[0] + recursive_sum(arr[1:])
7
8  a = [10, 5, 5, 20]
9  print(recursive_sum(a))
```

```
[10, 5, 5, 20]
[5, 5, 20]
[5, 20]
[20]
[]
40
```

Самостоятельно попробуйте модифицировать программный код листинга 9.2 таким образом, чтобы в консоль выводилась примерно следующая информация на каждом уровне вызова:

```
arr[0] = 10;    arr: [10, 5, 5, 20]
arr[0] = 5;     arr: [5, 5, 20]
arr[0] = 5;     arr: [5, 20]
arr[0] = 20;    arr: [20]
Достигнут базовый случай. Возвращение 0
40
```

Таким образом, суммируемый список на каждом уровне рекурсии становится все меньше, пока окончательно не опустеет – конец рекурсивного цикла. Сумма вычисляется при раскручивании рекурсивных вызовов по возврату

Рекурсия может быть **прямой**, как было показано выше, или **косвенной**, как в следующем примере (функция, вызывающая другую функцию, которая снова вызывает первую функцию). Совокупный эффект оказывается таким же, хотя на каждом уровне существуют два вызова функций вместо одного [5] (листинг 9.3).

Листинг 9.3 – Пример косвенной рекурсии

```
1  def get_sum(arr):
2      if not arr:
3          return 0
4      else:
5          return indirect_recursive_sum(arr)
6
7  def indirect_recursive_sum(arr):
8      return arr[0] + get_sum(arr[1:]) # Косвенная рекурсия
9
10
11 a = [10, 5, 5, 20]
12 print(get_sum(a))
```

```
40
```

Рекурсия может быть заменена циклом. Например, рекурсивную функцию суммирования можно было заменить на обычную функцию, используя цикл `while` или цикл `for` (листинг 9.4).

Листинг 9.4 – Пример использования циклов `while`, `for` вместо рекурсии

```
# Сумммирование через цикл while
arr = [10, 5, 5, 20]
sum_ = 0
while arr:
    sum_ += arr[0]
    arr = arr[1:]
print(sum_)

# Сумммирование через цикл for
arr = [10, 5, 5, 20]
sum_ = 0
for el in arr:
    sum_ += el
print(sum_)
```

```
40
40
```

Циклы обеспечивают автоматическую итерацию, делая рекурсию во многих случаях излишней (и иногда менее эффективной в плане расхода памяти и времени выполнения).

Преимущества использования операторов цикла вместо рекурсии [5]:

- не требуется новая копия локальной области видимости в стеке вызовов для каждой итерации,
- исключаются затраты времени, связанные с вызовами рекурсивных функций.

1.2. Обработка произвольных структур

Преимущество рекурсии (или эквивалентных алгоритмов, основанных на стеке) перед операторами цикла заключается в способности **обхода структур произвольной формы**.

Например, если имеется список, включающий произвольно вложенные подсписки подобного вида:

```
[1, [2, [3, 5], 5], 6, [7, 8]]
```

То обработать такой список гораздо проще с помощью рекурсии. Т.к. подсписки могут быть вложенными на произвольную глубину и в произвольной форме – не существует способа узнать, сколько вложенных циклов необходимо написать для обработки всех случаев. Следующий программный код приспосабливается к такому универсальному вложению за счет применения рекурсии для посещения всех подсписков (листинг 9.5).

Листинг 9.5 – Пример использования рекурсии для обработки произвольно вложенных подсписков

```
1  def sumtree(L):
2      summa = 0
3      for x in L: # Для каждого элемента на этом уровне
4          if not isinstance(x, list):
5              summa += x                  # Сложение чисел напрямую
6          else:
7              summa +=sumtree(x)          # Рекурсия для подсписков
8      return summa
9
10 L = [1, [2, [3, 4], 5], 6, [7, 8]]      # Произвольное вложение
11 print(sumtree(L)) # Вывод: 36
12
13 # Патологические случаи
14 print(sumtree([1, [2, [3, [4, [5]]]]]))  # Вывод: 15 (перегруженное справа)
15 print(sumtree([[[[[1], 2], 3], 4], 5]))  # Вывод: 15 (перегруженное слева)
```

Дополнительное задание: отследите тестовые сценарии в конце кода (строки 10, 14, 15), чтобы посмотреть, как рекурсия обходит вложенные списки.

2. Стек вызовов

Стек (англ. *stack* – стопка) – структура данных, работающая по принципу LIFO (Last In First Out, последним вошел – первым вышел) или, другими словами, по принципу стопки тарелок (последняя положенная сверху тарелка будет взята первой).

Пример стека представлен на рисунке 9.2.

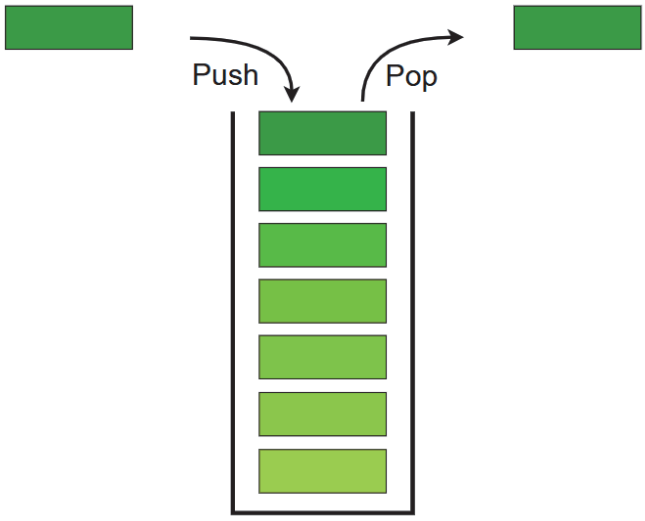


Рисунок 9.2 – Визуализация стека

Стек вызовов (от англ. *call stack*) – стек, использующийся во внутренней работе компьютера, в котором хранятся переменные вызываемых функций.

Стек вызовов – хранит все вызовы функций.

При вызове функции – в компьютере ей выделяется **блок памяти**. В блоке памяти хранятся значения всех переменных (атрибутов) для этого вызова.

Блоки памяти объединяются в стек:

- первым блоком идет блок памяти, выделяемый для **основной функции** (уровень 1),
- затем идет блок памяти, выделяемый для **вложенной функции** (уровень 2);
- если эта **вложенная функция имеет свою вложенную функцию**, то для нее выделяется блок памяти, который будет расположен на третьем месте в стеке (уровень 3).

При вызове вложенной функции происходит выделение памяти следующим образом:

1) Когда вложенная функция вызывается, выделяемый для нее блок памяти **добавляется в стек**.
2) В это время основная функция приостанавливается в частично завершенном состоянии.
3) По завершению выполнения кода вложенной функции, выделяемый для нее блок памяти **удаляется из стека**.
4) Происходит возвращение к основной функции. Основная функция продолжает выполняться с того места, где была прервана.

Стек играет важную роль в рекурсии. Рекурсивные функции также используют стек вызовов.

Хотя стек удобен, его использование может привести к большим затратам памяти в связи с хранением промежуточной информации. Существует два варианта решения данной проблемы:

- Использовать **цикл** вместо рекурсии.
- Использовать **хвостовую рекурсию**.

Хвостовая рекурсия – частный случай рекурсии, где последним вызовом (или конечным вызовом) является сама функция.

Рекусрия	Хвостовая рекурсия

`def recsum(x):`
`  if x == 1:`
`  return x`
`  else:`
`  return x + recsum(x - 1)`
	`def tailrecsum(x, running_total=0):`
	`  if x == 0:`
	`  return running_total`
	`  else:`
	`  return tailrecsum(x - 1, running_total + x)`

Один из способов приведения рекурсии к хвостово-рекурсивному виду заключается в том, что набор локальных данных, который нуждается в сохранении при рекурсивном вызове, переносится в параметры вызова функции.

3. Рекурсия и стек

Несмотря на искусственность примера с произвольно вложенными подсписками, он является типичным представителем более широкого класса программ. Например, рекурсия применяется для:

- тасования произвольных последовательностей;
- обхода цепочек импортирования модулей;
- обхода деревьев наследования классов;
- решении задач на подобие обратного отсчета и вычисления факториала.

Стандартный Python ограничивает глубину своего стека вызовов, критически важную для программ с рекурсивными вызовами, чтобы отлавливать ошибки бесконечной рекурсии. Для расширения стека используется модуль `sys` [5] (листинг 9.6).

Листинг 9.6 – Пример использования библиотеки `sys` для получения информации об установленной глубине стека вызовов (в примере – 3000 вызовов), изменении глубины стека вызовов (в примере – на 10000 вызовов), получения справочной информации с помощью функции `help`

```
In [14]: import sys

In [15]: sys.getrecursionlimit()
Out[15]: 3000

In [16]: sys.setrecursionlimit(10_000)

In [17]: sys.getrecursionlimit()
Out[17]: 10000

In [18]: help(sys.setrecursionlimit)
```

Задачи

Задача № 1

Определите, что делает следующая функция, и постройте для нее стек вызовов рекурсии.

```
def funct(n):
    if n < 10:
        return n
    else:
        return n % 10 + funct(n // 10)

print(funct(230))
```

Задача № 2

Определите, что делает следующая функция, и постройте для нее стек вызовов рекурсии.

```
def funct(k, n):
    if k < n:
        return str(k) + " " + str(funct(k + 1, n))
    else:
        return n

print(funct(4, 12))
```

Задача № 3

Реализуйте рекурсивную функцию, которая осуществит:

а) суммирование чисел во входящем списке.

б) суммирование положительных чисел во входящем списке, при обязательном наличии в списке отрицательных значений.

Задача № 4

Напишите функцию для подсчета нечетных элементов в массиве с помощью цикла. Напишите рекурсивную функцию для подсчета нечетных элементов в массиве. Распишите стек вызовов рекурсивной функции. Задача может быть реализована на любом языке – на псевдокоде или в блок-схемах.

Задача № 5

Рассмотрим последовательность, состоящую из круглых, квадратных и фигурных скобок:

а) `{}[()](())`

б) `()](()()[]`

Предложите алгоритм (программу), который позволит определить, является ли данная скобочная последовательность правильной (корректной). Правильная скобочная последовательность – такая последовательность скобок, в которой все скобки закрыты (каждой открывающейся скобке соответствует закрывающаяся скобка).

Задачу № 5 необходимо решить:

1. С помощью простого описания алгоритма.
2. С помощью языка Python.

Задача № 6

Дана строка, составленная из круглых скобок. Предложите алгоритм, позволяющий определять, какое наименьшее количество символов необходимо **удалить** из этой строки, чтобы оставшиеся символы образовывали правильную скобочную последовательность.

Предложите решение в итеративной и рекурсивной форме.

Задача № 7

Дана строка, составленная из круглых, квадратных и фигурных скобок. Предложите свой вариант решения задачи определения наименьшего количества скобок, которые необходимо **добавить** для этой строки, чтобы все представленные скобки образовывали правильную скобочную последовательность (также предложите решение с определением вида скобки и ее позиции в строке, чтобы превратить последовательность в корректную).

Задача № 8

Основываясь на ответе к задаче № 7, создайте собственную неправильную скобочную последовательность и преобразуйте ее в правильную путем добавления необходимых скобок.

Методические указания

Стек вызовов можно расписать следующим образом:

номер п/п	имя функции			возвращаемая команда		возвращаемое значение
	переменные	значения переменных				

Макет таблицы для стека вызовов из 3 элементов может быть следующим:

номер п/п	имя функции			возвращаемая команда		возвращаемое значение
	переменные	значения переменных				
3						
2						
1						

Пример стека вызовов для следующей рекурсивной функции:

```
def funct(a):
    if a < 100:
        return a
    else:
        return a + funct(a-1)

print(funct(101))
```

Номер п/п	имя функции: funct()			возвращаемая команда		возвращаемое значение
	переменные	значения переменных				
3	a	99		a		99
2	a	100		a + funct(a-1)		100 + funct(99)
1	a	101		a + funct(a-1)		101 + funct(100)

Результат работы функции: 99 + 100 + 101 = 300

Практическое задание № 10
«Быстрая сортировка. Сортировка слиянием»

1. «Разделяй и властвуй»

Алгоритмы «разделяй и властвуй» – рекурсивные алгоритмы, использующие стратегию «разделяй и властвуй» (рекурсивно реализованный алгоритм бинарного поиска, алгоритм быстрой сортировки, алгоритм сортировки слиянием и др.).

«Разделяй и властвуй» – подход к решению задачи.

Стратегия «разделяй и властвуй» включает 2 этапа:

1. Определить **базовый случай** как простейший случай из всех возможных.

2. Придумать способ, как свести задачу к базовому случаю (например, путем деления или сокращения).

Пример **базового случая**: при написании рекурсивной функции, в которой задействован массив, базовым случаем часто оказывается пустой массив или массив из одного элемента.

2. Быстрая сортировка

Быстрая сортировка (Quicksort) – рекурсивный алгоритм сортировки, в котором базовым случаем является массив размером 0 или 1 и который состоит из трех этапов:

1) выбор опорного элемента,

2) разделение массива на два подмассива,

3) рекурсивная сортировка подмассивов.

Время выполнения алгоритма в худшем случае: $O(n^2)$.

Время выполнения алгоритма в среднем случае: $O(n \log n)$.

Время выполнения алгоритма в лучшем случае: $O(n \log n)$.

Быстрая сортировка – один из самых быстрых существующих алгоритмов сортировки.

Особенность алгоритма быстрой сортировки в том, что добиться время выполнения в среднем случае и в лучшем случае достаточно легко:

- Если при выборе опорного элемента будет всегда выбираться **первый элемент (или последний)**, то алгоритм будет работать по худшему сценарию – $O(n^2)$.
- Если при выборе опорного элемента будет всегда выбираться **случайный элемент**, то алгоритм будет работать по среднему сценарию – $O(n \log n)$.
- Если при выборе опорного элемента будет всегда выбираться **средний элемент**, то алгоритм будет работать по лучшему сценарию – $O(n \log n)$.

Этапы алгоритма быстрой сортировки:

1) Выбрать **опорный** элемент в массиве.

2) Разделить массив на два подмассива: элементы, меньшие опорного, и элементы, большие опорного. В результате исходный массив разделяется на 3 части:
   - подмассив всех элементов, меньших опорного;
   - опорный элемент;
   - подмассив всех элементов, больших опорного.
3) Рекурсивно применить быструю сортировку к двум подмассивам.

Визуализация быстрой сортировки представлена на рисунке 10.1.

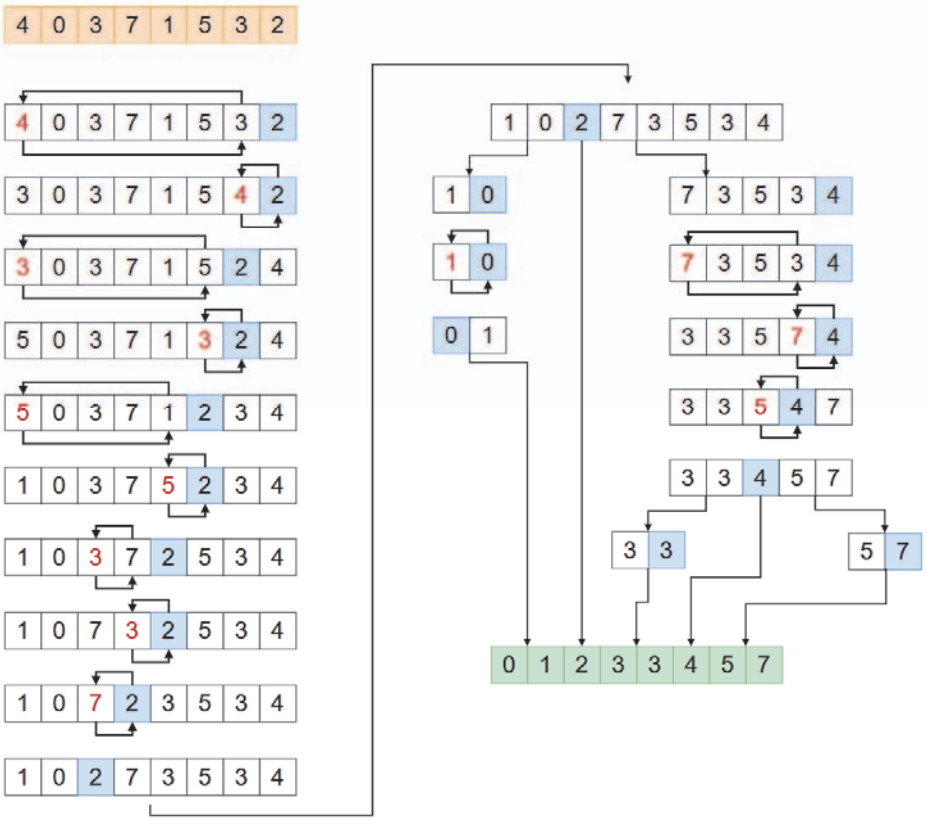


Рисунок 10.1 – Визуализация быстрой сортировки, при которой в качестве опорного элемента выбирается последний элемент (ячейка синего цвета)

3. Сортировка слиянием

Сортировка слиянием (Merge sort) – разделение массива на подмассивы (вплоть до единичных), слияние соседних подмассивов с одновременной сортировкой.

Время выполнения алгоритма (все случаи): $O(n \log n)$.

Этапы алгоритма сортировки слиянием:

1) Разбиение массива на две части примерно одинакового размера.
2) Сортировка каждой части массива (применение рекурсии).
3) Соединение отсортированных подмассивов в один массив, подразумевающее:

a. поэтапное и попарное сравнивание элементов из двух подмассивов, включение в результирующий массив наименьшего элемента из этой пары элементов (для сортировки по возрастанию).
b. когда один из подмассивов остается пустым – простое добавление второго подмассива в конец результирующего массива.

Визуализация сортировки слиянием представлена на рисунке 10.2.

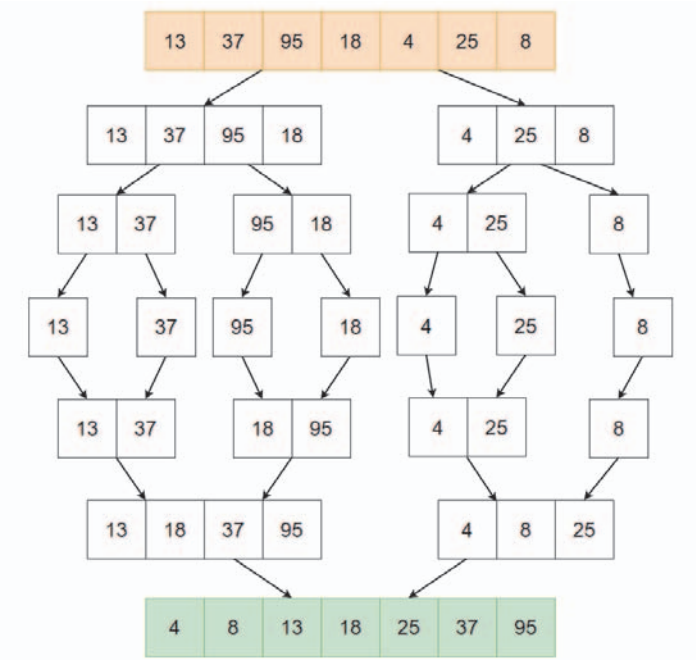


Рисунок 10.2 – Визуализация сортировки слиянием

4. Константы в нотации «О-большое»

Время выполнения алгоритма по нотации «О-большое» может содержать константу, которая обозначается буквой *c*. Пример:

$$O(c \cdot n)$$

Константа *c* может быть равна произвольному числу.

Если для двух алгоритмов время выполнения разное, например, $O(n)$ и $O(n^2)$, то константа несущественно влияет на время выполнения (предполагается, что n – большое число), поэтому она не учитывается и опускается.

Если для двух алгоритмов время выполнения одинаковое, например, $O(n)$, то константа может иметь значение.

У быстрой сортировки константа меньше, чем у сортировки слиянием, поэтому несмотря на то, что оба алгоритма характеризуются временем $O(n \log n)$, быстрая сортировка работает быстрее. А на практике быстрая сортировка работает быстрее, потому что средний случай встречается намного чаще худшего [4].

Исходные данные:

Варианты для задач 3-5:

1. [45, 67, 79, 40, 40, 23, 15, 29, 47]
2. [40, 63, 9, 56, 70, 17, 38, 97, 19]
3. [44, 45, 74, 56, 66, 44, 50, 31, 96]
4. [80, 72, 48, 67, 58, 43, 39, 54, 64]
5. [61, 34, 93, 52, 38, 33, 11, 45, 72]
6. [100, 16, 77, 58, 2, 98, 50, 36, 47]
7. [49, 91, 22, 81, 58, 87, 60, 56, 42]
8. [5, 73, 24, 44, 86, 12, 42, 47, 67]
9. [23, 16, 99, 82, 68, 91, 83, 48, 41]
10. [96, 17, 36, 39, 16, 9, 99, 97, 9]
11. [72, 82, 92, 3, 21, 39, 90, 41, 15]
12. [78, 34, 38, 56, 50, 48, 69, 27, 94]
13. [48, 32, 92, 2, 42, 40, 21, 61, 73]
14. [6, 85, 14, 35, 48, 29, 25, 91, 51]
15. [42, 24, 32, 77, 73, 50, 47, 72, 86]
16. [89, 34, 27, 40, 31, 37, 45, 93, 58]
17. [24, 53, 8, 58, 16, 46, 10, 27, 18]
18. [100, 35, 76, 83, 51, 2, 44, 84, 69]
19. [16, 44, 42, 19, 45, 15, 90, 50, 17]
20. [36, 26, 42, 35, 79, 93, 71, 2, 83]
21. [27, 70, 47, 22, 48, 10, 42, 92, 53]
22. [32, 22, 15, 19, 94, 26, 50, 84, 71]
23. [22, 73, 68, 87, 7, 77, 87, 30, 10]
24. [50, 53, 54, 94, 20, 37, 79, 37, 5]
25. [61, 86, 48, 7, 100, 92, 17, 6, 99]
26. [93, 92, 18, 37, 68, 99, 7, 45, 91]
27. [19, 80, 27, 37, 97, 99, 24, 3, 78]
28. [18, 60, 50, 77, 48, 73, 47, 21, 87]
29. [37, 45, 87, 51, 84, 95, 39, 1, 87]
30. [62, 28, 50, 45, 94, 63, 94, 15, 79]
31. [85, 77, 67, 44, 17, 63, 1, 10, 3]
32. [93, 41, 60, 49, 5, 72, 55, 68, 62]
33. [75, 75, 77, 96, 76, 57, 99, 80, 15]
34. [21, 49, 42, 1, 40, 12, 82, 27, 10]
35. [31, 24, 91, 34, 68, 25, 92, 28, 13]

Задачи

Задача № 1

Определите О-большое для каждой из следующих операций:

1. Вывод значения каждого элемента массива.
2. Удвоение значения каждого элемента массива.
3. Удвоение значения только первого элемента массива.
4. Создание таблицы умножения для всех элементов массива. Например, если массив состоит из элементов [4, 10, 8, 2, 5, 2], сначала каждый элемент умножается на 4, потом каждый элемент умножается на 10, затем на 8 и так далее.

Задача № 2

Напишите рекурсивную функцию для подсчета элементов в списке (на Python, псевдокоде или средствами иного языка программирования). Содержание списка придумайте самостоятельно.

Задача № 3

Напишите рекурсивную функцию для поиска наибольшего числа в списке.

Задача № 4

Графически изобразить процесс сортировки массива целых чисел для быстрой сортировки и сортировки слиянием. Представить соответствующий стек вызова рекурсии для каждого алгоритма.

Задача № 5

Реализовать **быструю сортировку** и **сортировку слиянием** на Python через создание двух соответствующих функций. Предусмотреть следующие возможности:

a. визуализация (через консоль) всех шагов алгоритма, включая массив и (или) подмассивы, с которыми работает алгоритм на текущем шаге;
b. для **быстрой сортировки** – выбор опорного элемента (первый, последний, случайный);
c. выбор типа сортировки (по возрастанию, по убыванию);

Проверить созданные функции на следующих массивах:

- массив из исходных данных;
- массив 10 случайных целых чисел (без повторов) в диапазоне [0; 100];
- массив 20 случайных целых чисел (с повторами) в диапазоне [0; 100];
- массив 30 случайных целых чисел (с повторами) в диапазоне [-100; 100];
- массив 50 случайных целых чисел (с повторами) в диапазоне [-1000; 1000].

Задача № 6

Дан отсортированный массив, который содержит положительные и отрицательные элементы. Необходимо отсортировать этот массив по модулю. Предложите наиболее оптимальный вариант реализации.

Практическое задание № 11
«Очередь»

1. Очередь: общие сведения

Очередь (Queue) – структура данных, работающая по принципу FIFO (First In First Out, первым вошел – первым вышел) или, другими словами, по принципу трубки с шариками (первый вошедший в трубку шарик выйдет также первым).

Очередь, так же, как и стек, не поддерживает обращение к произвольному элементу.

Операции очереди. Очередь поддерживает 2 операции:

- Enqueue – постановка в очередь.
- Dequeue – извлечение из очереди.

Визуализация очереди представлена на рисунке 11.1.

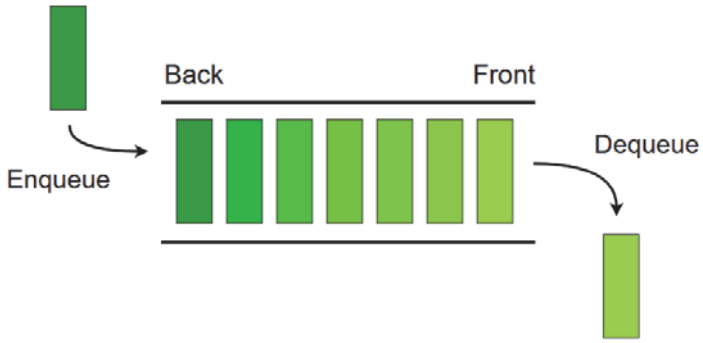


Рисунок 11.1 – Визуализация очереди

Дек (Deque, Double-ended queue) – особый тип очереди, двусторонняя очередь.

Дек отличается от очереди тем, что операции постановки и извлечения элемента могут происходить с двух сторон этой очереди.

Операции дека. Дек поддерживает 4 операции:

- Enqueue – постановка в очередь.
- Dequeue – извлечение из очереди.
- Front – постановка в очередь (в начало).
- Rear – извлечение из очереди (с конца).

Визуализация очереди, в сравнении с деком, представлена на рисунке 11.2.

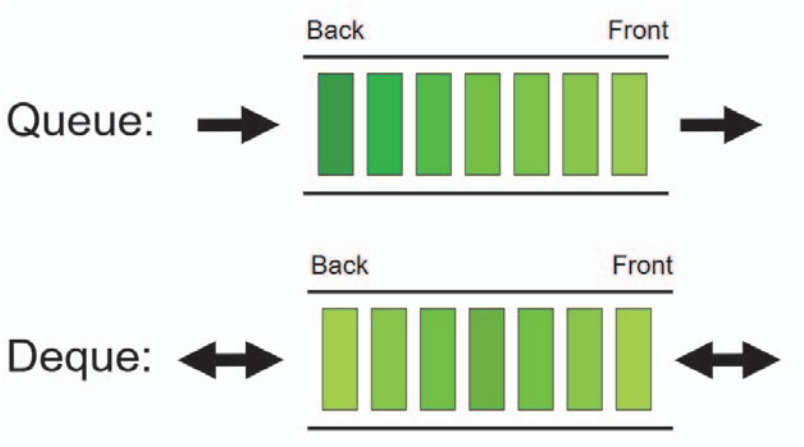


Рисунок 11.2 – Визуализация очереди и дека

2. Реализация очереди в Python

Очередь может быть реализована на Python следующими способами:

1. Встроенный список (list).
2. Класс collections.deque.
3. Класс queue.Queue.
4. Класс multiprocessing.Queue.

2.1. Встроенный список (list)

Используются следующие операции:

1) `= []` – создание пустого списка.
2) `.append(x)` – постановка в очередь.
3) `.pop(0)` – извлечение из очереди (в качестве параметра указывается индекс удаляемого элемента – первого элемента очереди, то есть 0).
4) `.insert(0, x)` – постановка в начало очереди (в качестве параметров указывается позиция вставляемого элемента, то есть 0, и сам элемент).
5) `.pop()` – извлечение из конца очереди. Также может указывать индекс последнего элемента: `.pop(-1)`

Примеры:

Имя операции	Команда	Пример команды	Содержание очереди
Создание пустой очереди	`= []`	`q = []`	`[]`

Постановка в очередь	`.append()`	`q.append(1)`
		`q.append(2)`
		`q.append(3)`
			`[1]`
			`[1, 2]`
			`[1, 2, 3]`

Извлечение из очереди	`.pop()`	`q.pop(0)`
		`q.pop(0)`
		`q.pop(0)`
			`[2, 3]`
			`[3]`
			`[]`

Постановка в очередь (в начало)	`.insert()`	`q.insert(0, 3)`
		`q.insert(0, 2)`
		`q.insert(0, 1)`
			`[3]`
			`[2, 3]`
			`[1, 2, 3]`

Извлечение из очереди (с конца)	`.pop()`	`q.pop(-1)`
		`q.pop()`
		`q.pop()`
			`[1, 2]`
			`[1]`
			`[]`

Дополнительные операции:

- `len(q)` – определение размера.
- `q.clear()` – очистка.

Недостаток встроенного списка при реализации очереди: низкая производительность из-за того, что удаление первого элемента или вставка на первую позицию требует перестановки всех остальных элементов (на это уходит $O(n)$ времени).

В связи с этим, не рекомендуется использовать встроенный список для реализации очереди с большим числом элементов.

2.2. Класс collections.deque

Класс deque встроенной библиотеки collections позволяет создавать объекты типа deque для реализации стека и очереди.

Объекты deque поддерживают эффективно использующие память операции добавления и извлечения с любой стороны двухсторонней очереди с примерно одинаковой производительностью: $O(1)$.

Для подключения библиотеки используется команда:

```
from collections import deque
```

Пример:

Имя операции	Команда	Пример команды	Содержание очереди
Создание пустой очереди	`deque()`	`q = deque()`	`deque([])`

Постановка в очередь	`.append()`	`q.append(1)`
		`q.append(2)`
		`q.append(3)`
			`[1]`
			`[1, 2]`
			`[1, 2, 3]`

Извлечение из очереди	`.popleft()`	`q.popleft()`
		`q.popleft()`
		`q.popleft()`
			`[2, 3]`
			`[3]`
			`[]`

Постановка в очередь (в начало)	`.appendleft()`	`q.appendleft(3)`
		`q.appendleft(2)`
		`q.appendleft(1)`
			`[3]`
			`[2, 3]`
			`[1, 2, 3]`

Извлечение из очереди (с конца)	`.pop()`	`q.pop()`
		`q.pop()`
		`q.pop()`
			`[1, 2]`
			`[1]`
			`[]`

Дополнительные операции:

- `len(q)` – определение размера.
- `q.clear()` – очистка.

Дополнительную информацию см. в документации: https://docs.python.org/3/library/collections.html#collections.deque

2.3. Класс queue.Queue

Встроенная библиотека queue позволяет реализовывать стеки и очереди, особенно полезные в многопоточном программировании (параллельные вычисления), когда необходимо безопасно обмениваться информацией между несколькими потоками.

Для подключения библиотеки используется команда:

```
from queue import Queue
```

Пример:

Имя операции	Команда	Пример команды	Содержание очереди
Создание пустой очереди	`Queue()`	`q = Queue()`	`[]`

Постановка в очередь	`.put()`	`q.put(1)`
		`q.put(2)`
		`q.put(3)`
			`[1]`
			`[1, 2]`
			`[1, 2, 3]`

Извлечение из очереди	`.get()`	`q.get()`
		`q.get()`
		`q.get()`
			`[2, 3]`
			`[3]`
			`[]`

Постановка в очередь (в начало)	-	-	-
Извлечение из очереди (с конца)	-	-	-

Стек можно реализовать через класс queue.LifoQueue.

Дополнительные операции:

- `q.qsize()` – определение размера.
- `q.empty()` – проверка на пустоту.

Метод get() помимо извлечения из очереди, также возвращает элемент.

Просмотреть содержание всей очереди Queue не представляется возможным. Однако можно преобразовать данную очередь в объект класса collections.deque с помощью команды:

```
q1 = q.queue
```

Данный подход является небезопасным, так как при изменении объекта q1 объект q также будет изменяться.

Дополнительную информацию см. в документации: https://docs.python.org/3/library/queue.html#queue-objects

Задачи

При выполнении задач не допускается обращение к произвольному элементу стека или очереди, необходимо учитывать особенности указанных структур данных [1].

Задача № 1

Дано число N (> 0) и набор из N чисел. Создать стек, содержащий исходные числа (последнее число будет вершиной стека). Извлечь из стека все элементы и вывести их значения. Вывести также количество извлеченных элементов N.

Стек необходимо реализовать, используя:

a. Встроенный список (list)
b. Класс collections.deque
c. Класс queue.LifoQueue

Задача № 2

Даны два непустых стека. Переместить все элементы из первого стека во второй (в результате элементы первого стека будут располагаться во втором стеке в порядке, обратном исходному).

Задача № 3

Дан один непустой стек. Создать два новых стека, переместив в первый из них все элементы исходного стека с четными значениями, а во второй – с нечетными (элементы в новых стеках будут располагаться в порядке, обратном исходному; один из этих стеков может оказаться пустым).

Стек необходимо реализовать, используя:

a. Класс collections.deque
b. Класс queue.LifoQueue

Задача № 4

Дан набор из N чисел (N > 10). Создать очередь, содержащую данные числа в указанном порядке (первое число будет размещаться в начале очереди, последнее — в конце). Извлечь из очереди все элементы и вывести их значения. Вывести также количество извлеченных элементов N.

Очередь необходимо реализовать, используя:

c. Встроенный список (list)
d. Класс collections.deque
e. Класс queue.Queue

Задача № 5

Дан набор из 10 чисел. Создать две очереди: первая должна содержать числа из исходного набора с нечетными номерами (индексами), а вторая – с четными; порядок чисел в каждой очереди должен совпадать с порядком чисел в исходном наборе.

Задача № 6

Дан набор из 10 чисел. Создать две очереди: первая должна содержать все нечетные, а вторая – все четные числа из исходного набора (порядок

чисел в каждой очереди должен совпадать с порядком чисел в исходном наборе).

Задача № 7

Дано число D и очередь, содержащая не менее 10 элементов. Добавить элемент со значением D в конец очереди и извлечь из очереди первый (начальный) элемент. Вывести значение извлеченного элемента.

Задача № 8

Дано число N (> 0) и непустая очередь. Создать функцию для извлечения из очереди N начальных элементов и отображения их значения (если очередь содержит менее N элементов, то извлечь все ее элементы).

Задача № 9

Дана непустая очередь. Извлекать из очереди элементы, пока значение начального элемента очереди не станет четным, и выводить значения извлеченных элементов (если очередь не содержит элементов с четными значениями, то извлечь все ее элементы).

Задача № 10

Даны две очереди. Переместить все элементы первой очереди (в порядке от начала к концу) в конец второй очереди.

Задача № 11

Дано число N (> 0) и две непустые очереди. Создать функцию для перемещения N начальных элементов первой очереди в конец второй очереди. Если первая очередь содержит менее N элементов, то переместить из первой очереди во вторую все элементы.

Очередь необходимо реализовать, используя:

a. Класс collections.deque
b. Класс queue.Queue

Задача № 12

Даны две непустые очереди. Очереди содержат одинаковое количество элементов. Объединить очереди в одну, в которой элементы исходных очередей чередуются (начиная с первого элемента первой очереди).

Задача № 13

Даны две непустые очереди. Элементы каждой из очередей упорядочены по возрастанию (в направлении от начала очереди к концу). Объединить очереди в одну с сохранением упорядоченности элементов.

Очередь необходимо реализовать, используя:

c. Класс collections.deque
d. Класс queue.Queue

Практическое задание № 12
«Связный список»

1. Структуры данных: массив, связный список

1.1. Память компьютера

При сохранении какого-то отдельного значения в памяти компьютера используется адрес (адрес ячейки памяти).

Сохранение нескольких элементов (значений) осуществляется двумя способами, в основе которых лежит одна из двух **структур данных**:

1. Массив.
2. Связный список.

Структура данных (Data Structure) – способ (формат) хранения данных для их эффективного поиска и извлечения.

Структура данных характеризуется типом доступа.

Тип доступа – способ чтения элементов из структуры данных. Тип доступа бывает последовательный и произвольный.

Последовательный – элементы читаются по одному, начиная с первого.

Произвольный – элементы читаются в произвольном порядке.

1.2. Массив

Массив (Array) – структура данных, при которой все элементы хранятся в памяти непрерывно (рядом друг с другом).

Тип доступа: произвольный.

Так как все элементы хранятся непрерывно, то при выделении памяти для массива, необходимо знать, сколько элементов содержит массив (или будет содержать в будущем). Если соседние ячейки памяти могут быть заняты, то добавление нового элемента в массив (расширение массива), потребует перемещение всего массива в новую область памяти, что ресурсозатратно.

В массиве все элементы пронумерованы, причем нумерация начинается с 0. То есть первый элемент имеет номер 0 (индекс 0).

Индекс – позиция, номер элемента в массиве.

Недостатки массива:

- Неэффективный расход памяти в случае выделения для массива ячеек памяти «про запас», которые могут так и не понадобиться.
- Ресурсозатратное перемещение всего массива в новую область памяти, если для массива понадобятся дополнительные ячейки памяти.

Применение массива:

- когда данные читаются в произвольном порядке (элемент 1, элемент 3, элемент 104, элемент 56, …).
- много операций чтения (поиск элементов).

1.3. Связный список

Связный список (Linked List) – структура данных, при которой все элементы хранятся в памяти разрозненно (не рядом друг с другом).

Связный список – совокупность элементов (узлов), каждый из которых связан только с соседними элементами (узлами).

Тип доступа: последовательный.

В каждом элементе хранится адрес следующего элемента списка (для однонаправленного связного списка). Таким образом, набор произвольных адресов памяти объединяется в цепочку [4].

При добавлении нового элемента в связный список, выделяется любая свободная ячейка памяти, а ее адрес сохраняется в предыдущий элемент.

Недостатки связного списка:

- Осложнено получение конкретного элемента связного списка, особенно последнего, так как для определения его адреса, необходимо пройти по всему связному списку.

Применение связного списка:

- когда данные должны читаться последовательно (элемент 1, элемент 2, элемент 3, элемент 4, …).
- когда данные записываются в середину списка.
- много операций вставки/удаления.

Визуализация массива и связного списка представлена на рисунке 12.1.

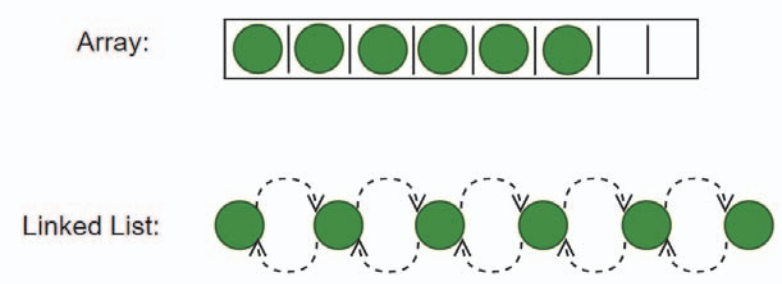


Рисунок 12.1 – Визуализация массива и связного списка

Типы связных списков:

1. Однонаправленный (Single Linked List):

2. Двунаправленный (Doubly Linked List):

3. Циклический (круговой) (Circular Linked List):

Визуализация трех типов связного списка представлена на рисунке 12.2.

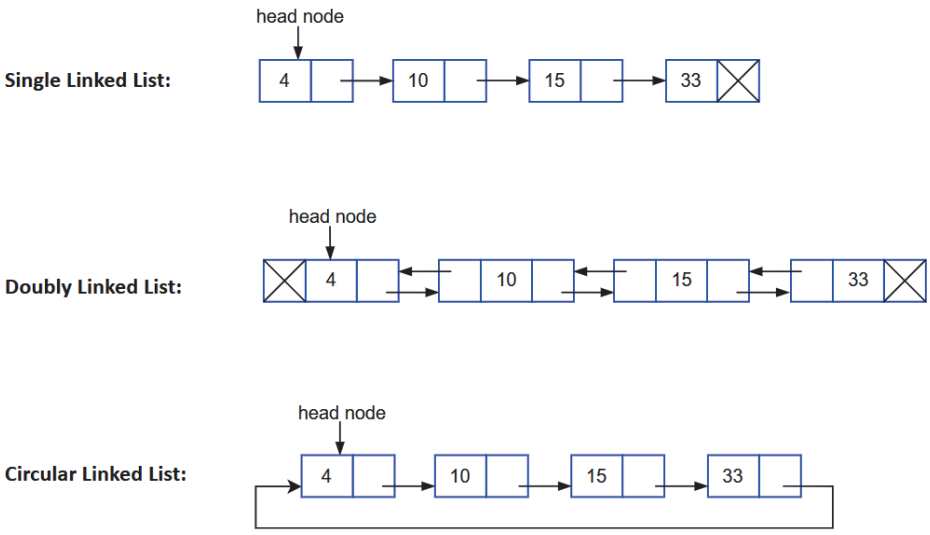


Рисунок 12.2 – Визуализация типов связного списка

1.4. Операции со структурами данных

Над структурами данных выполняют следующие основные операции:

- Чтение.
- Вставка.
- Удаление.

Время выполнения каждой операции для массива и связного списка приведено в таблице 12.1.

Таблица 12.1 – Сложность основных операций над массивом и связным списком

Структура данных	Операция		
	Чтение	Вставка	Удаление
Массив	$O(1)$	$O(n)$	$O(n)$
Связный список	$O(n)$	$O(1)$	$O(1)$

Примечание: для связного списка вставка и удаление выполняются за $O(1)$ *только в том случае, если можно получить доступ к необходимому элементу мгновенно (за рамки выносится операция поиска (чтения) этого элемента).*

Список основных структур данных и сложность операций над ними представлены на рисунке 12.3.

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Рисунок 12.3 – Сложность основных операций со структурами данных (Common Data Structure Operations) в нотации Big-O

Источник: [12].

2. Реализация однонаправленного связного списка на Python

Создадим класс `Node`, каждый экземпляр которого будет представлять собой узел или элемент связного списка.

Каждый узел хранит значение (атрибут `value`) и ссылку на следующий узел (атрибут `link_next`). При создании первого узла ссылка на следующий узел будет отсутствовать, то есть равняться `None`.

```
# Определения класса узла (Node)
class Node:

    # Метод, вызывающийся при создании экзмепляра
    def __init__(self, value):
        '''
        value - значение, которое будет сохранено в узле
        '''
        self.value = value      # Сохранение value как атрибута
        self.link_next = None  # Ссылка на следующий элемент св. списка
```

Для добавления нового узла в конец связного списка создадим метод `appendToTail()`:

```
    # Метод добавления элемента в конец св. списка
    def appendToTail(self, value):
        node_end = Node(value = value)   # Узел в конце св. списка
        link_current = self              # Ссылка на первый узел св. списка

        # Прохождение по списку в конец до предпоследнего узла
        while(link_current.link_next is not None):
            link_current = link_current.link_next   # Переход на след. узел
        link_current.link_next = node_end           # Присваивание атрибуту link_next экземпляр Node
```

Внутри метода новый узел, который необходимо добавить в конец связного списка, сохраняется в переменную `node_end`.

Необходимо, чтобы значение атрибута `link_next` самого последнего узла связного списка было равно не `None`, а переменной `node_end`. Для этого происходит переход по ссылкам (по экземплярам класса `Node`) до последнего узла (строки 18, 19). Атрибуту `link_next` последнего узла присваивается новое значение – значение переменной `node_end` (строка 20).

Создадим узел со значением 10 и последовательно свяжем его с узлами со значениями 11 и 12, получив таким образом связный список:

```
23  llist = Node(value = 10)
24  llist.appendToTail(value = 11)
25  llist.appendToTail(value = 12)
```

Переходя по ссылкам, получая значения атрибутов, мы можем извлечь все значения связного списка:

```
In [53]: llist
Out[53]: <__main__.Node at 0x19912130>

In [54]: llist.value
Out[54]: 10

In [55]: llist.link_next
Out[55]: <__main__.Node at 0x184febe0>

In [56]: llist.link_next.value
Out[56]: 11

In [57]: llist.link_next.link_next.value
Out[57]: 12
```

Через цикл извлечь все элементы связного списка можно, например, следующим образом:

```
28  # Отображение всех элементов св. списка
29  node = llist
30  print(node.value)
31  while (node.link_next):
32      node = node.link_next
33      print(node.value)
```

Здесь условие на строке 31 означает то же, что и:

```
while (node.link_next is not None):
```

Задачи

1. Реализуйте **однонаправленный связный список** путем создания класса в Python. При этом в виде методов класса для однонаправленного связного списка реализовать операции:

- Чтение всех элементов.
- Вставка элемента в конец связного списка.
- Вставка элемента в начало связного списка.
- Удаление элемента с конца связного списка.
- Удаление элемента с начала связного списка.
- Поиск элемента в связном списке.
- Вставка элемента после указанного элемента.
- Удаление указанного элемента.

2. Реализуйте **двунаправленный связный список** путем создания класса в Python. При этом в виде методов класса для двунаправленного связного списка реализовать операции:

- Чтение всех элементов.
- Вставка элемента в конец связного списка.
- Удаление элемента с конца связного списка.

3. Реализуйте **циклический связный список** путем создания класса в Python. При этом в виде методов класса для циклического связного списка реализовать операции:

- Чтение всех элементов.
- Вставка элемента в конец связного списка.
- Удаление элемента с конца связного списка.

Практическое задание № 13
«Хеш-таблица»

1. Хеш-таблица: общие сведения

Хеш-таблица (Hash Table) – структура данных, основанная на массиве и хеш-функции.

Хеш-таблица – массив, формируемый в определенном порядке хеш-функцией.

Хеш-таблица также известна как:

- Ассоциативный массив.
- Словарь.
- Отображение.
- Хеш-карта.
- Хеш.

Хеш-функция (Hash Function) – функция, получающая на вход строку (любые данные – последовательность байтов) и возвращающая число (либо другую строку, хеш-код).

Хеш-функция отображает строки на числа (либо на другие строки).

Хеширование (Hashing) – преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины (хеш-код).

Такие преобразования также называются хеш-функциями или функциями свертки, а их результаты называют хешем, хеш-кодом, хеш-таблицей или дайджестом сообщения (message digest) [4].

Требования к хеш-функции:

- Последовательность – при передаче на вход одной и той же строки хеш-функция должна возвращать одно и то же число.
- Многосторонность – при передаче на вход разных строк хеш-функция должна возвращать разные числа.

Таким образом, с помощью хеш-функции можно получить число для соответствующей строки. Если это число будет индексом элемента в массиве, то мы можем осуществлять быстрый поиск за время $O(1)$: передавать в хеш-функцию искомый элемент и получать индекс этого элемента в массиве.

Массив и связанный список – напрямую отображаются на адреса памяти.

Хеш-таблица – определяет место хранения с помощью хеш-функции.

Использование хеш-таблицы: хеши эффективно применяется в случаях:

- Создания **связи, отображающей один объект на другой** (примеры: студент и его оценка, товар и его стоимость, имя контакта и его номер, избиратель и факт голосования (True / False) , символическое имя адреса и его IP-адрес, URL-адрес страницы и данные страницы (кэширование) и т.д.).

- Поиска значения в списке.
- Устранения дубликатов.

Примеры использования хеширования в повседневной жизни: распределение книг в библиотеке по тематическим каталогам, упорядочивание в словарях по первым буквам слов, шифрование специальностей в вузах и т.д. [4].

Время выполнения каждой операции для хеш-таблицы приведено в таблице 13.1.

Таблица 13.1 – Быстродействие хеш-таблицы, массива и связанного списка

Структура данных	Операция		
	Чтение	Вставка	Удаление
Хеш-таблица (средний случай)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Хеш-таблица (худший случай)	$O(n)$	$O(n)$	$O(n)$
Массив	$O(1)$	$O(n)$	$O(n)$
Связный список	$O(n)$	$O(1)$	$O(1)$

В среднем каждая операция в хеш-таблице выполняется за постоянное время $O(1)$, то есть время выполнения не зависит от размера хеш-таблицы.

Избежать худшего случая при работе с хеш-таблицей можно минимизировав коллизии.

2. Коллизии

Коллизия – случай, когда несколько ключей отображаются на один элемент хеш-таблицы.

2.1 Методы разрешения коллизий (стратегии обработки коллизий)

1. Метод цепочек (внешнее или открытое хеширование) – если элементам множества соответствует одно и то же значение хеш-функции, то они связываются в цепочку-список (связный список).

Другими словами, если в ячейку массива (в соответствии с результатом работы хеш-функции) необходимо сохранить несколько элементов, то в этой ячейке создается связный список.

Либо изначально создается массив, каждая ячейка которого представляет собой связный список.

2. Метод открытой адресации (закрытое хеширование / линейное пробирование) – если элементам множества соответствует одно и то же значение хеш-функции, то первый элемент сохраняется в ячейку, соответствующую значению хеш-функции, а остальные элементы сохраняются в последующие ближайшие свободные ячейки.

В отличие от хеширования с цепочками, при открытой адресации никаких списков нет, а все записи хранятся в самой хеш-таблице. Каждая ячейка таблицы содержит либо элемент динамического множества, либо NULL.

В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку до тех пор, пока не будет найдена пустая позиция в таблице.

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более 3 – 4 сравнений, то эффективность использования такой хеш-таблицы падает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента.

Удаление элементов в такой схеме несколько затруднено. Обычно поступают так: заводят логический флаг для каждой ячейки, помечающий, удален ли элемент в ней или нет. Тогда удаление элемента состоит в установке этого флага для соответствующей ячейки хеш-таблицы, но при этом необходимо модифицировать процедуру поиска существующего элемента так, чтобы она считала удаленные ячейки занятыми, а процедуру добавления – чтобы она их считала свободными и сбрасывала значение флага при добавлении.

Для минимизации или предотвращения коллизий необходимо создавать эффективные («хорошие») хеш-функции.

2.2. Коэффициент заполнения

Коэффициент заполнения (Load Factor) – число заполненных элементов в хеш-таблице.

С помощью коэффициента заполнения можно определить число пустых ячеек в хеш-таблице.

Так как хеш-таблица для хранения данных использует массив, то мы можем посчитать число заполненных элементов этого массива по формуле:

$$Load\ Factor = \frac{n}{N},$$

где n – число элементов в хеш-таблице,
N – общее число элементов.

Если $Load\ Factor = 1$, то хеш-таблица заполнена полностью, необходимо ее расширить. Для добавления нового элемента необходимо создавать новый массив (обычно вдвое большего размера), а затем все элементы вставить в новую хеш-таблицу.

Чем меньше $Load\ Factor$, тем меньше коллизий, тем быстрее работает хеш-таблица.

Существует правило, в соответствии с которым увеличивать размер хеш-таблицы следует, если $Load\ Factor > 0{,}7$.

2.3. Эффективная («хорошая») хеш-функция

Эффективная хеш-функция – обеспечивает равномерное распределение значений в массиве.

Неэффективная хеш-функция – не обеспечивает равномерное распределение значений в массиве, создает скопления, порождает множество коллизий [4].

3. Алгоритмы хеширования

Существует множество алгоритмов хеширования или видов хеш-функций.

Данные алгоритмы получают на вход строку и возвращают хеш-код этой строки.

Алгоритмы семейства SHA (Secure Hash Algorithm): SHA-0, SHA-1, SHA-2, SHA-3. Семейство алгоритмов SHA-2 в свою очередь включает алгоритмы: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/256 и SHA-512/224.

Алгоритмы MD: MD2, MD4, MD5.

Примеры использования алгоритмов хеширования:

1. Сравнение файлов: допустим, на компьютерах А и В хранятся определенные файлы большого размера. Необходимо выяснить, присутствует ли на компьютере В один из этих больших файлов, который присутствует на компьютере А. Вместо ресурсозатратного копирования этого файла на компьютер В с последующим сравнением, можно вычислить хеш-код этого файла на компьютере А. Затем вычислить хеш-коды файлов на компьютере В. Если хеш-код файла с компьютера А присутствует в списке хеш-кодов файлов с компьютера В, то значит и сам этот файл присутствует на компьютере В.

2. Проверка паролей: алгоритм хеширования может использоваться для сравнения исходных строк при отсутствии информации о самих исходных строках. Если в определенной таблице хранить не сами пароли, а только хеш-коды этих паролей, то при авторизации пользователя будет сначала вычисляться хеш-код введенного им пароля, а затем осуществляться поиск этого хеш-кода в таблице. В случае взлома этой таблицы злоумышленниками, они получат хеш-коды, а не пароли, причем восстановить пароли по хеш-кодам если и возможно, то, как минимум, затруднительно (зависит от слабости алгоритмов хеширования).

3.1. «Соление» паролей

Соль (также модификатор входа хэш-функции) – строка данных, которая передаётся хеш-функции вместе с входным массивом данных (прообразом) для вычисления хеш-кода (образа).

Используется для усложнения определения прообраза хэш-функции **методом перебора по словарю** возможных входных значений (прообразов), включая атаки с использованием **радужных таблиц**. Позволяет скрыть факт использования одинаковых прообразов при использовании для них разной соли.

Различают статическую соль (одна и та же для всех входных значений) и динамическую (генерируется для каждого входного значения персонально).

4. Реализация хеш-таблицы в Python

4.1. Хеш-таблица и словарь

В языках программирования имеются готовые реализации хеш-таблиц. В Python хеш-таблица реализована в виде типа данных «словарь» (dictionary),

при этом обеспечивается ее производительность в среднем случае: за постоянное время $O(1)$.

Словарь – неупорядоченный набор пар «ключ : значение». Ключ и значение разделяются двоеточием. При этом ключи должны быть уникальными в рамках одного словаря.

Доступ к элементу словаря осуществляется по ключу (индексация по ключу).

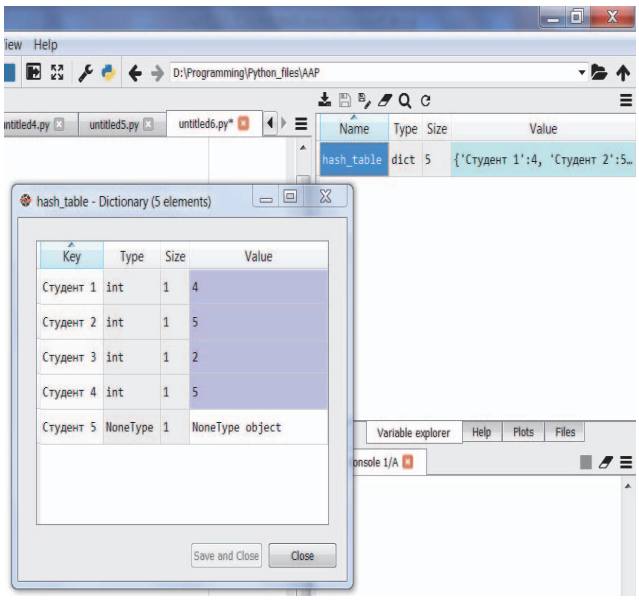
Для создания словаря (хеш-таблицы) используются фигурные скобки `{}`, либо функция `dict()`:

```
hash_table = {}       # Создание пустой хеш-таблицы

hash_table = dict()   # Создание пустой хеш-таблицы
```

Заполнить пустой словарь можно через обращение по ключу с помощью квадратных скобок. Если такого ключа в словаре нет, то он добавляется в словарь, поэтому нужно указывать чему будет равен новый элемент словаря (указать значение соответствующее ключу):

```
5  # Добавление элементов в пустую хеш-таблицу
6  hash_table['Студент 1'] = 4
7  hash_table['Студент 2'] = 5
8  hash_table['Студент 3'] = 2
9  hash_table['Студент 4'] = 5
10 hash_table['Студент 5'] = None
```

Посмотреть содержание словаря можно с помощью проводника переменных (Variable explorer) в правой части среды разработки Spyder:



Создать заполненный словарь можно следующим образом:

```
12  # Очистка хеш-таблицы
13  hash_table.clear()
14
15  # Создание заполненной хеш-таблицы
16  hash_table = {'Студент 1' : 4,
17                'Студент 2' : 5,
18                'Студент 3' : 2,
19                'Студент 4' : 5,
20                'Студент 5' : None}
```

По ключу мы можем получить соответствующее значение:

```
22  print('\nhash_table:\n', hash_table)
23  print('\nОценка студента №4:', hash_table['Студент 4'])
```

```
hash_table:
 {'Студент 1': 4, 'Студент 2': 5, 'Студент 3': 2, 'Студент 4': 5, 'Студент 5': None}

Оценка студента №4: 5
```

Узнать о наличии ключа в словаре можно с помощью метода `get()`, который возвращает значение ключа, если он содержится в словаре, либо `None` в противном случае:

```
In [26]: hash_table.get('Студент 1')
Out[26]: 4

In [27]: hash_table.get('Студент 4')
Out[27]: 5

In [28]: hash_table.get('Студент 5')

In [29]: hash_table.get('Студент 6')

In [30]: type(hash_table.get('Студент 6'))
Out[30]: NoneType
```

Дополнительную информацию см. в документации: [17].

4.2. Хеш-таблица на основе списка

Без использования структуры данных Python «словарь», реализовать хеш-таблицу можно через структуру данных «список», то есть массив. Данный массив будет являться двумерным: каждая ячейка массива будет содержать массив из двух ячеек: ключ, значение.

Двумерный массив (список списков) из 5 элементов можно создать следующим образом (первые 2 одинаковые команды записаны разными способами, 3 команда аналогична первым двум):

```
hash_table = [[], [], [], [], []]

hash_table = [[],
              [],
              [],
              [],
              []]

hash_table = [[],] * 5
```

Создадим хеш-таблицу с некоторыми заполненными ячейками:

```
hash_table = [[],
              ['Студент 1', 3],
              ['Студент 2', 5],
              [],
              []]
```

Доступ к элементу данной хеш-таблицы осуществляется через индексацию:

```
In [14]: hash_table[0]
Out[14]: []

In [15]: hash_table[1]
Out[15]: ['Студент 1', 3]
```

Так как каждый элемент данной хеш-таблицы – массив, то для доступа к ключу или значению необходимо так же указывать индекс в квадратных скобках:

```
In [16]: hash_table[1][0]
Out[16]: 'Студент 1'

In [17]: hash_table[1][1]
Out[17]: 3
```

Добавление записи в хеш-таблицу:

```
hash_table[3] = ['Студент 3', 2]
print(hash_table)
```

```
[[], ['Студент 1', 3], ['Студент 2', 5], ['Студент 3', 2], []]
```

Индекс 3 в действительности должен быть получен с помощью некоторой хеш-функции (`hash_function`):

```
string = ['Студент 3', 2]
index = hash_function(string) # = 3
hash_table[index] = string
print(hash_table)
```

Задачи

Задача № 1

Разработайте функцию, с помощью которой будет производиться сохранение в хеш-таблицу нового элемента, если ключа такого элемента нет в хеш-таблице.

Задача № 2

Разработайте функцию, которая на вход получает путь к папке, а на выходе возвращает сгруппированные названия одинаковых по содержанию файлов.

Проверить функцию на разных файлах (.txt, .docx, .pdf, .xlsx). Файлы необходимо разместить в папке «Директория_1».

Задача № 3

Ориентируясь на ответ к задаче № 2, разработайте функцию для проверки наличия дубликата определенного файла. При этом необходимо учесть следующие условия, выполняющиеся одновременно:

- Файл может находиться в любой папке.
- Дубликат файла может находиться только в папке «Директория_1».
- Имя файла и имя его дубликата могут не совпадать.
- Имена файлов могут совпасть, но по содержанию файлы все равно могут быть разные.

Задача № 4

Необходимо хешировать пароли для последующего сохранения в базе данных. Необходимо «посолить» пароль с помощью `salt`. `Salt` является случайной последовательностью, добавленной к строке пароля перед использованием хеш-функции. `Salt` используется для предотвращения перебора по словарю (**dictionary attack**).

Реализовать статическое и динамическое «соление» паролей.

Задача № 5

Ориентируясь на ответ к задаче № 4, реализуйте регистрацию и авторизацию пользователей с помощью функций и хеш-таблицы. При этом хеш-таблица будет хранить логин и хеш-код пароля пользователя.

Зарегистрируйте как минимум 5 пользователей. Сымитируйте авторизацию пользователей, проверьте правильность процесса авторизации.

Задача № 6

Реализуйте хеш-таблицу с помощью массива и собственной хеш-функции (не используйте при этом словарь `dict` Python).

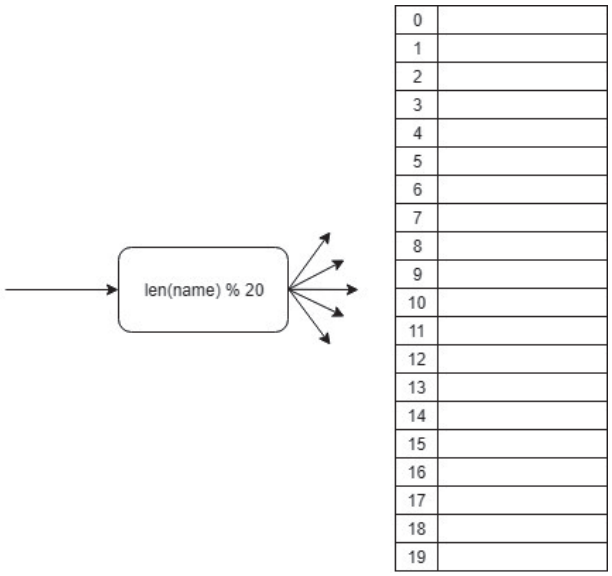
В хеш-таблицу необходимо сохранить название продукта (ключ) и его стоимость (значение).

Хеш-функция работает следующим образом: вычисляет длину получаемой строки (название продукта) и делит ее на 20 – получившийся остаток от деления является хеш-кодом (индексом ячейки массива, в которую необходимо сохранить строку).

Для разрешения коллизий необходимо использовать метод открытой адресации (линейное пробирование).

a. Заполните хеш-таблицу для следующего набора данных и рассчитать коэффициент заполнения (load factor):

'Водоросли': 280
'Картофель': 260
'Лук-порей': 59
'Манго': 291
'Орехи грецкие': 266
'Салями': 225
'Специи': 283
'Сыр сливочный': 152
'Творог': 215
'Тофу': 142
'Хек': 248
'Чай черный': 118
'Чернила каракатицы': 95
'Шампиньоны': 101
'Финик': 104



Примечание: хеш-таблица изначально должна содержать 20 пустых ячеек.

b. После добавления всего набора данных выполните удаление из хэш-таблицы следующих данных, в том порядке, в котором они представлены и рассчитать коэффициент заполнения (load factor):

'Орехи грецкие': 266
'Водоросли': 280
'Специи': 283
'Манго': 291

Практическое задание № 14
«Словарь в Python. Сортировка подсчетом»

1. Словарь в Python

Словарь (Dictionary) – неупорядоченная структура данных, каждый элемент которой является парой «ключ – значение».

1.1. Создание словаря

Создание словаря осуществляется через фигурные скобки `{…}` или функцию `dict()`.

Например, создать словарь из первых пяти букв разных алфавитов с их порядковым номером можно следующим образом:

```
dictionary = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5,
              'а': 1, 'б': 2, 'в': 3, 'г': 4, 'д': 5}
```

Данный словарь содержит 10 элементов, в нем используются строки в качестве **ключей**.

Ключом может являться любой неизменяемый тип данных, такой как, например, строка, число.

Значением ключа может быть любой тип данных.

1.2. Обращение к элементам словаря

Для получения значения по ключу используются квадратные скобки следующем образом:

```
словарь[ключ]
```

Например, если требуется узнать порядковый номер буквы «d» в алфавите (в объявленном ранее словаре `dictionary`), то для необходимо ввести команду:

```
In [3]: dictionary['d']
Out[3]: 4
```

При попытке получить значение по несуществующему ключу возникает ошибка:

```
In [4]: dictionary['dd']
Traceback (most recent call last):

  Cell In[4], line 1
    dictionary['dd']

KeyError: 'dd'
```

1.3. Добавление элемента

Добавление элемента в словарь происходит через обращение по ключу с помощью квадратных скобок. Если такого ключа в словаре нет, то он добавляется в словарь, поэтому нужно указывать чему будет равен новый элемент словаря (указать значение, соответствующее ключу):

```
словарь[новый_ключ] = значение
```

Например, чтобы добавить элемент с ключом 'ff' и значением 7 необходимо ввести следующую команду:

```
dictionary['ff'] = 7
```

```
In [5]: dictionary['ff'] = 7

In [6]: dictionary
Out[6]:
{'a': 1,
 'b': 2,
 'c': 3,
 'd': 4,
 'e': 5,
 'а': 1,
 'б': 2,
 'в': 3,
 'г': 4,
 'д': 5,
 'ff': 7}
```

1.4. Обновление элемента

Обновить значение по ключу можно аналогичным образом путем переприсваивания значения:

```
dictionary['ff'] = 6
```

```
In [7]: dictionary['ff'] = 6

In [8]: dictionary
Out[8]:
{'a': 1,
 'b': 2,
 'c': 3,
 'd': 4,
 'e': 5,
 'а': 1,
 'б': 2,
 'в': 3,
 'г': 4,
 'д': 5,
 'ff': 6}
```

1.5. Удаление элемента

Для удаления элемента из используется оператор `del`:

```
del dictionary['ff']
```

```
In [9]: del dictionary['ff']

In [10]: dictionary
Out[10]:
{'a': 1,
 'b': 2,
 'c': 3,
 'd': 4,
 'e': 5,
 'а': 1,
 'б': 2,
 'в': 3,
 'г': 4,
 'д': 5}
```

1.6. Методы словаря

Словарь имеет множество методов. Одними из наиболее часто использующихся являются следующие методы.

`update()`

Добавляет в словарь другой словарь, который ему передается на вход в качестве аргумента:

```
dictionary.update({'f' : 6, 'ж' : 8})
{'a': 1,
 'b': 2,
 'c': 3,
 'd': 4,
 'e': 5,
 'а': 1,
 'б': 2,
 'в': 3,
 'г': 4,
 'д': 5,
 'f': 6,
 'ж': 8}
```

`get()`

Возвращает значение ключа, если он содержится в словаре, либо `None` в противном случае:

```
In [28]: dictionary.get('b')
Out[28]: 2

In [29]: dictionary.get('bb')

In [30]:
```

pop()

Удаляет элемент по ключу и возвращает значение удаленного элемента.

```
In [30]: dictionary.pop('b')
Out[30]: 2

In [31]: dictionary
Out[31]:
{'a': 1,
 'c': 3,
 'd': 4,
 'e': 5,
 'a': 1,
 'б': 2,
 'в': 3,
 'г': 4,
 'д': 5,
 'f': 6,
 'ж': 8}
```

keys()

Возвращает коллекцию ключей в словаре.

```
In [32]: dictionary.keys()
Out[32]: dict_keys(['a', 'c', 'd', 'e',
'a', 'б', 'в', 'г', 'д', 'f', 'ж'])
```

values()

Возвращает коллекцию значений в словаре.

```
In [33]: dictionary.values()
Out[33]: dict_values([1, 3, 4, 5, 1, 2, 3,
4, 5, 6, 8])
```

items()

Возвращает пары «ключ – значение».

```
In [34]: dictionary.items()
Out[34]: dict_items([('a', 1), ('c', 3),
('d', 4), ('e', 5), ('a', 1), ('б', 2),
('в', 3), ('г', 4), ('д', 5), ('f', 6),
('ж', 8)])
```

1.7. Проход по элементам словаря

Для прохода по элементам словаря используется цикл `for`.

Следующие два эквивалентных варианта записи позволяют пройтись по ключам словаря:

```
dict1 = {'a' : 1, 'b' : 2, 'c' : 3}

for key in dict1:
    print(key)

for key in dict1.keys():
    print(key)
```

```
a
b
c
a
b
c
```

Для получения и ключа и значения на каждой итерации можно воспользоваться методом `items()`:

```
dict1 = {'a' : 1, 'b' : 2, 'c' : 3}

for key, value in dict1.items():
    print(key, value)
```

```
a 1
b 2
c 3
```

2. Сортировка подсчетом

Сортировка подсчетом (Counting Sort) – сортировка на основе вспомогательного массива (или словаря), содержащего частоты для каждого числа из исходного массива.

Сортировку подсчетом эффективно применять в случае небольшого диапазона значений, например для сортировки массива целых чисел от 0 до 100, содержащего миллион элементов.

В простом случае сортировка подсчетом может быть реализована в соответствии со следующими шагами:

1. Определение минимального q и максимального элемента p в исходном массиве.

2. Создание вспомогательного словаря частот с ключами от q до p. Заполнение словаря нулями.

3. Проход по исходному массиву и обновление частот в словаре частот (увеличение на 1 того элемента словаря, чей ключ соответствует числу в исходном массиве).

4. Формирование результирующего массива: проход по словарю частот и добавление в массив элементов от q до p в соответствии с их частотами.

Пример:
Имеется исходный массив

1	9	0	3	0	2	1	1	5	7	1	4	9

1. Определение минимального q и максимального элемента p в исходном массиве:

min = 0
max = 9

2. Создание вспомогательного словаря частот с ключами от q до p. Заполнение словаря нулями.

ключ	0	1	2	3	4	5	6	7	8	9
значение	0	0	0	0	0	0	0	0	0	0

3. Проход по исходному массиву и обновление частот в словаре частот (увеличение на 1 того элемента словаря, чей ключ соответствует числу в исходном массиве).

0	1	2	3	4	5	6	7	8	9
2	4	1	1	1	1	0	1	0	2

4. Формирование результирующего массива: проход по словарю частот и добавление в массив элементов от q до p в соответствии с их частотами.

0	0	1	1	1	1	2	3	4	5	7	9	9

Сложность алгоритма сортировки подсчетом:

$$O(n + k),$$

где n – количество элементов в исходном массиве, k – количество элементов в вспомогательном массиве (словаре) частот.

При малом диапазоне значений относительно количества элементов скорость алгоритма может быть высокой:

- при n = 1024, k = 10 скорость алгоритма составит:

$$O(n+k) = O(n+\log n) = O(n)$$

При большом диапазоне значений относительно количества элементов скорость алгоритма может быть крайне низкой:

- при n = 100, k = 10000 скорость алгоритма составит:

$$O(n+k) = O(n+n^2) = O(n^2)$$

- при n = 10, k = 3628800 скорость алгоритма составит:

$$O(n+k) = O(n+n!) = O(n!)$$

Задачи

Задача № 1

1. Создать словарь ***d1*** (не используя функцию `dict()`):

a - 100
b - 500
c - 60

2. Вывести только обозначения букв.
3. Вывести на печать только цифры.
4. Создать словарь ***d2*** (через функцию `dict()`):

d - 400
e - 789
f - 350

5. Объединить 2 словаря, обновив при этом ***d1***.
6. Проверить, есть ли в словаре ***d1*** ключи «b», «p», «b».
7. Определить количество ключей в словаре.
8. Удалить запись с ключом «f».
9. Поменять значение с ключом «a» на 500.
10. Добавить запись g - 300.
11. Создать копию словаря (***d3***).
12. Очистить содержимое словаря ***d3***.

Задача № 2

Напишите функцию `to_dict(lst)`, которая принимает аргумент в виде списка и возвращает словарь, в котором каждый элемент списка является и ключом, и значением. Предполагается, что элементы списка будут соответствовать правилам задания ключей в словарях.

Задача № 3

Дана строка в виде случайной последовательности чисел от 0 до 9.

Требуется создать словарь, который в качестве ключей будет принимать данные числа (т.е. ключи будут иметь тип `int`), а в качестве значений – количество этих чисел в имеющейся последовательности.

Для построения словаря создайте функцию `count_it(sequence)`, принимающую строку из цифр. Функция должна возвратить словарь из 3-х самых часто встречаемых чисел.

Задача № 4

Реализуйте алгоритм сортировки подсчетом.

Задача № 5

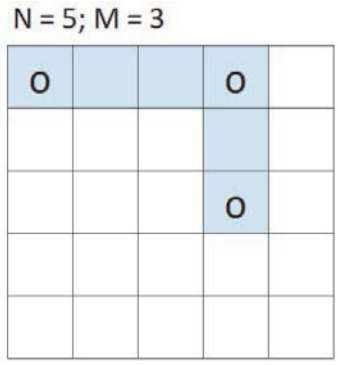
С помощью промежуточного этапа сортировки подсчетом решите задачу:

Дано два числа X и Y без ведущих нулей. Необходимо проверить, можно ли получить одно число из другого перестановкой цифр.

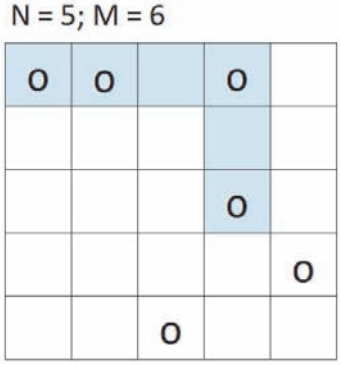
Задача № 6

На шахматной доске NxN находятся M ладей (ладья бьет клетки по той же вертикали или горизонтали до ближайшей занятой). Определите сколько пар ладей бьют друг друга. Ладьи задаются парой чисел I и J (индексы), обозначающие координаты клетки. Разработайте функцию для решения данной задачи.

Например, здесь представлена доска 5x5, на которой расположены 3 ладьи, при этом будет 2 пары ладей, которые бьют друг друга:



Здесь представлена доска 5x5, на которой расположены 6 ладей, при этом будет 3 пары ладей, которые бьют друг друга:



Практическое задание № 15
«Множество»

1. Множество. Общие сведения

Множество (Set) в Python –представляет собой неупорядоченный набор элементов без дубликатов.

Множество в Python – неупорядоченная коллекция уникальных и неизменяемых объектов, которая поддерживает операции, соответствующие математической теории множеств.

Элемент встречается во множестве только один раз независимо от того сколько раз он добавлялся. Таким образом, с множествами связаны разнообразные применения, особенно при работе, ориентированной на числа и базы данных [5].

Дополнительную информацию см. в документации: [17].

Общие свойства множества соответствуют свойствам других коллекций, таких как список и словарь:

- итерируемо;
- может увеличиваться и уменьшаться по требованию;
- может содержать объекты разнообразных типов.

Уникальные свойства множества:

- не упорядочено позиционно, и потому оно не является последовательностью – порядок множества произволен и может варьироваться от выпуска к выпуску Python;
- содержит уникальные элементы (без повторов);
- содержит элементы неизменяемых типов (хешируемые объекты).

Множество действует во многом подобно ключам в **словаре** без значений (но поддерживает дополнительные операции) – поскольку элементы множества неупорядочены, уникальны и неизменяемы, они ведут себя очень похоже на ключи словаря.

Хотя операции над множествами (Объединение, Пересечение, Разность, Симметрическая разность и др.) можно кодировать вручную с другими типами (структурами данных) вроде списков и словарей, встроенные множества Python применяют эффективные алгоритмы и методики реализации для обеспечения быстрой работы.

2. Инициализация множества

Способ 1: конструкция `set(iterable)`.

Способ 2: фигурные скобки `{…}`, где вместо … перечисляются элементы множества. Применение пустых фигурных скобок `{}` приведет к созданию словаря.

Примеры создания множеств представлены в листинге 15.1.

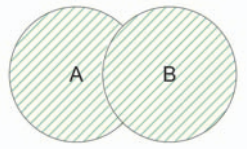
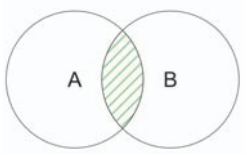
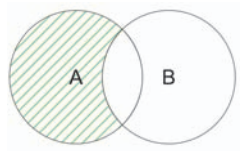
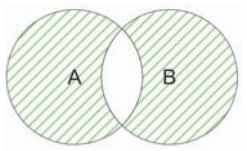
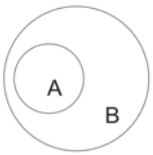
Листинг 15.1 – Пример создания множеств

```
a1 = set([1, 2, 3, 3, 3, 2, 4])

a2 = {2, 2, 1, 3}

a1:
{1, 2, 3, 4}
a2:
{1, 2, 3}
```

3. Математические операции над множеством

Множества в Python поддерживают распространенные математические операции над множествами через **операции выражений** (таблица 15.1):

Таблица 15.1 – Основные операции над множествами

Название операции	Обозначение	Изображение кругами Эйлера	Определение
Объединение (Union)	Математическое: A ∪ B В Python: `A | B`		**Объединение** множеств A, B – это множество, состоящее из всех элементов, принадлежащих *хотя бы* одному из множеств A или B.
Пересечение (произведение) (Intersection)	Математическое: A ∩ B В Python: `A & B`		**Пересечение** множества A, B – это множество, состоящее из всех элементов, принадлежащих *одновременно* и множеству A, и множеству B.
Разность (Difference)	Математическое: A \ B В Python: `A - B`		**Разность** множеств A, B – это множество, состоящее из всех элементов множества A, не принадлежащих множеству B.
Симметрическая разность (исключающее ИЛИ) (Symmetric Difference)	Математическое: A ∆ B В Python: `A ^ B`		**Симметрическая разность** множеств A, B – это множество, состоящее из всех элементов, принадлежащих *только одному* множеству A или множеству B.
Надмножество (Superset), **Подмножество** (Subset)	Математическое: A ⊂ B В Python: `A < B`		Множество A – **подмножество** множества B, если каждый элемент множества A является элементом множества B.

Указанные в таблице 15.1 операции нельзя выполнять на простых последовательностях наподобие строк, списков и кортежей – для применения таких инструментов необходимо создавать множества, передавая последовательности на вход функции `set()`.

Примеры работы с множествами представлены в листинге 15.2.

Листинг 15.2 – Примеры реализации основных операций над множествами

```
In [2]: x = set([1, 2, 3, 4, 'text', 't', 'e'])

In [3]: y = set([3, 10, 15, 'hello', 't', 'h'])

In [4]: x | y
Out[4]: {1, 10, 15, 2, 3, 4, 'e', 'h', 'hello', 't', 'text'}

In [5]: x & y
Out[5]: {3, 't'}

In [6]: x - y
Out[6]: {1, 2, 4, 'e', 'text'}

In [7]: y - x
Out[7]: {10, 15, 'h', 'hello'}

In [8]: x ^ y
Out[8]: {1, 10, 15, 2, 4, 'e', 'h', 'hello', 'text'}

In [9]: x < y
Out[9]: False

In [10]: x > y
Out[10]: False

In [11]: set([1, 2]) < x
Out[11]: True
```

4. Методы множества

В дополнение к выражениям, объект множества предоставляет методы, которые соответствуют этим и другим операциям и поддерживают изменения множеств:

- `add()` – добавление одного элемента во множество.
- `update()` – добавление подмножества во множество (слияние, объединение на месте).
- `remove()` – удаление элемента по значению из множества.
- `union()` – объединение множеств.
- `intersection()` – пересечение множеств.
- `difference()` – разность множеств.
- `symmetric_difference()` – симметрическая разность множеств.
- `issubset()` – определение, является ли множество подмножеством.
- `issuperset()` – определение, является ли множество надмножеством.

Вызовите dir() для любого экземпляра множества либо для имени типа set(), чтобы увидеть все доступные методы.

Выражения с множествами обычно требуют двух множеств. Однако специальные методы, в отличие от операторов, часто позволяют работать также и с любым итерируемым типом, таким как список, строка, диапазон (`range`) и т.п. (листинг 15.3).

Листинг 15.3 – Пример использования оператора `<` и метода `issubset()`

```
In [12]: set(['t', 'e']) < 'text'
Traceback (most recent call last):

  Cell In[12], line 1
    set(['t', 'e']) < 'text'

TypeError: '<' not supported between instances of 'set' and 'str'


In [13]: set(['t', 'e']).issubset('text')
Out[13]: True
```

5. Проход по множеству

Как итерируемые контейнеры, множества могут также использоваться в операциях вроде `len()`, циклах `for` и списковых включениях [5]:

Листинг 15.4 – Пример прохода по множеству через цикл `for`

```
1 set1 = {1, 2, 3, 2, 2, 3, 10}
2 print('set1:', set1)
3 for el in set1:
4     print(el)
```

```
set1: {3, 1, 10, 2}
3
1
10
2
```

Тем не менее, из-за того, что множества неупорядочены, они не поддерживают операции над последовательностями, подобные индексации и нарезанию [5].

6. Ограничение неизменяемости и фиксированные множества

Множества являются мощными и гибкими объектами, однако имеют одно важное ограничение – множества способны содержать только объекты неизменяемых типов (**хешируемые объекты**).

Хешируемый объект (Hashable object) – объект:

- для которого существует хеш-значение (хеш-код), которое не меняется на протяжении существования объекта (для определения хеш-значения используется метод `__hash__()`) и
- который можно сравнить с другими объектами (для этого используется метод `__eq__()`).

Хешируемые объекты, которые при сравнении оказываются равными (идентичными) имеют одинаковое хеш-значение [19].

Неизменяемые типы в Python: `int, float, complex, bool, str, tuple, range, frozenset, bytes`.

Изменяемые типы в Python: `list, dict, set, bytearray, user-defined classes`.

Проверить, является ли объект хешируемым, можно с помощью функции `hash()`, которая возвращает хеш-значение объекта, если он является хешируемым (листинг 15.5).

Листинг 15.5 – Пример применения функции `hash()`

```
In [1]: hash(14)
Out[1]: 14

In [2]: hash('qqq')
Out[2]: 4239919775372059942

In [3]: hash('qq')
Out[3]: -2880092063754836397

In [4]: hash([1, 2])
Traceback (most recent call last):

  Cell In[4], line 1
    hash([1, 2])

TypeError: unhashable type: 'list'
```

С помощью функции `dir()` можно получить список методов для класса или объекта. Наличие методов `__hash__()`, `__eq__()` свидетельствует о том, что объект данного класса (данный объект) является хешируемым:

Листинг 15.6 – Пример применения функции `dir()` и `hash()` для определения того, является ли кортеж (`tuple`) хешируемым

```
In [5]: dir(tuple)
Out[5]:
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'count',
 'index']

In [6]: (1, 3).__hash__()
Out[6]: -1440771752368011620

In [7]: hash((1, 3))
Out[7]: -1440771752368011620
In [9]: a = (1, 2)

In [10]: b = (1, 2)

In [11]: a.__eq__(b)
Out[11]: True
```

Например, списки и словари не могут встраиваться во множества, а кортежи могут, если необходимо хранить составные значения.

Кортежи во множестве могут использоваться, например, для представления дат, записей, IP-адресов и т.п.

Сами множества изменяемы и, соответственно, не могут вкладываться в другие множества напрямую.

Если необходимо хранить множество внутри другого множества, то применяется встроенная функция `frozenset()`, которая работает так же, как `set()`, но создает **неизменяемое множество** (фиксированное множество).

7. Применение множества

На практике множества могут применяться для решения как математических задач, так и для задач иного характера, например:

- Фильтрация дубликатов.
- Выделение различий в итерируемых объектах.

- Проверка на равенство, нейтральное к порядку [5].

7.1. Фильтрация дубликатов

Фильтрация дубликатов подразумевает удаление всех повторяющихся элементов из коллекции (например, списка), в результате чего в коллекции остаются только уникальные элементы.

Например, чтобы исходного списка целых чисел с повторами составить список уникальных целых чисел, можно реализовать преобразование списка во множество с помощью функции `set()`, а замет реализовать обратное преобразование этого множества в список с помощью функции `list()`. При этом порядок следования элементов в итоговом списке может изменится, что связано с особенностью множества в Python (листинг 15.7).

Листинг 15.7 – Пример применения множества для удаления дубликатов из списка

```
In [1]: a = [1, 2, 10, 200, 3, 1, 2, 3, 200, 5]

In [2]: a = list(set(a))

In [3]: a
Out[3]: [1, 2, 3, 5, 200, 10]
```

7.2. Выделение различий в итерируемых объектах

Поиск различий в итерируемых объектах подразумевает нахождение той части объекта A, которая отсутствует в объекте B. Такими объектами могут быть, например, тексты, документы, списки сотрудников и т.д.

Для решения такой задачи части объекта из предметной области необходимо представить как элементы множества, а затем найти разность между двумя множествами (листинг 15.8).

Листинг 15.8 – Пример применения множества для поиска различий в двух текстах

```
In [4]: text1 = set(['Hello', ',', 'World', '!'])

In [5]: text2 = set(['Hello', ',', 'Python', '!'])

In [6]: text2 - text1
Out[6]: {'Python'}
```

7.3. Проверка на равенство, нейтральное к порядку

Проверка на равенство, нейтральное к порядку подразумевает сравнение двух объектов, не учитывая порядок следования элементов (частей) этих объектов.

Листинг 15.9 – Пример применения множества для проверки на равенство двух текстов без учета порядка следования слов (проверка по содержанию)

```
In [13]: text1 = set(['Abstract', 'Introduction', 'Methods', 'Results',
'Conclusion', 'References'])

In [14]: text2 = set(['Introduction', 'Abstract', 'Methods', 'Results',
'Conclusion', 'References'])

In [15]: text1 == text2
Out[15]: True

In [16]: text1 = ['Abstract', 'Introduction', 'Methods', 'Results',
'Conclusion', 'References']

In [17]: text2 = ['Introduction', 'Abstract', 'Methods', 'Results',
'Conclusion', 'References']

In [18]: text1 == text2
Out[18]: False
```

Например, такая проверка может применяться для сравнения текстовых документов, таких как техническая документация, сборники статей, препринты книг, где имеются некоторые разделы, но они могут быть расположены в разном порядке (листинг 15.9).

Для решения такой задачи с помощью, например, списков, требуется предварительная сортировка двух списков с помощью функции `sorted()`, что зачастую может быть менее эффективно по сравнению с подходом на основе множеств.

Задачи

Задача № 1

1. Создать список ***x***: 5, 3, 6, 8, 3, 4, 6, 2, 1, 4, 5, 6, 7, 7.
2. Создать список ***y***: 12, 1, 6, 7, 3, 22, 6, 2, 17, 4, 5, 67, 3, 9.
3. Удалить из списка ***x*** повторяющиеся элементы.
4. Преобразовать списки во множества.
5. Определить число элементов множества ***y.***
6. Проверить, входит ли во множество ***x***: 5, 8, 12, 9.
7. Добавить 9 во множество ***x***.
8. Удалить из множества ***x*** значения 8 и 12 (разными способами).
9. Объединить множества в ***w.***
10. Определить общие элементы множеств ***x*** и ***y.***
11. Найти элементы ***x***, которые не входят в ***y.***
12. Найти элементы, которые входят в ***x*** или ***y***, но не в оба одновременно.
13. Проверить, является ли ***x*** подмножеством ***w.***
14. Проверить, является ли ***y*** подмножеством ***x.***
15. Проверить, равны ли множества ***x*** и ***y.***
16. Определить, ***x*** больше ***y*** или нет**.**

17. Заморозить множество ***x*** (метод `frozenset()`) и попробовать добавить значение 19 в новое замороженное множество.

Задача № 2

Имеется ряд словарей с пересекающимися ключами (значения – положительные числа). Напишите 2 функции, которые производят с массивом словарей следующие операции:

- функция `max_dct(*dicts)` формирует новый словарь по правилу:
  - если в исходных словарях имеются **повторяющиеся ключи**, среди их значений выбирается максимальное и присваивается этому ключу (например, в `словаре_1` есть ключ `"a"` со значением 5, и в `словаре_2` есть ключ `"a"`, но со значением 9. Выбирается максимальное значение, т.е. 9, и присваивается ключу `"a"` в уже новом словаре).
  - если **ключ не повторяется**, то он просто переносится со своим значением в новый словарь (например, ключ `"c"` встретился только у одного словаря, а у других его нет. Следовательно, переносим в новый словарь этот ключ вместе с его значением). Сформированный словарь возвращаем.
- функция `sum_dct(*dicts)` суммирует значения повторяющихся ключей. Значения остальных ключей остаются исходными. (Проводятся операции по аналогу первой функции, но берутся не максимумы, а суммы значений одноименных ключей). Функция возвращает сформированный словарь.

Задача № 3

Создайте 2 текстовых файла и заполните их разным текстом.
С помощью множеств определите:

1. Уникальные слова в файле 1 (без дубликатов). Уникальные слова в файле 2.
2. Уникальные слова в двух файлах (если бы текст в двух файлах необходимо было объединить).
3. Уникальные слова, которые встречаются только в файле 1 и отсутствуют в файле 2.
4. Уникальные слова, которые встречаются лишь в одном файле из двух (то есть слова, которые находятся только в файле 1 и отсутствуют в файле 2, а также слова, которые находятся только в файле 2 и отсутствуют в файле 1).
5. Уникальные слова, которые встречаются одновременно в обоих файлах.
6. Создайте подмножество слов из множества всех слов в файле 1. Создайте подмножество слов из множества всех слов в файле 2. Определите, равны ли два подмножества.

Задача № 4

Дана последовательность положительных чисел длинной N и число X. Создать множество из последовательности чисел.

Нужно найти два разных числа A и B из множества, таких что A + B = X или вернуть пару 0, 0, если такой пары нет.

Пример:

Последовательность чисел: 1, 10, 2, 4, 16, 7, 1, 2, 10, 7, 3.

N = 11, X = 6.

Множество чисел: 1, 10, 2, 4, 16, 7, 3.

Вывод: 2, 4.

Задача № 5*

Реализуйте собственное множество.

Условие:

Элемент в множество добавляется по последней цифре числа. Ячеек в множестве 10 (0-9).

Реализовать функции:

- *добавления элементов*
- *поиск и вывод всех элементов ячейки*
- *поиск конкретного числа в множестве*
- *удаление элемента из множества.*

Задача № 6*

Решите Задачу № 4, используя собственное множество из задачи № 5.

Контрольные вопросы к разделу «Структуры данных. Алгоритмы поиска и сортировки»

1. Перечислите известные вам алгоритмы сортировки. Классифицируйте их по сложности.
2. Как работает пузырьковая сортировка? Почему ее редко используют на практике?
3. Опишите алгоритм быстрой сортировки (QuickSort). Как выбирается опорный элемент?
4. В чем разница между сортировкой вставками и сортировкой выбором?
5. Объясните принцип работы пирамидальной сортировки (Heap Sort).
6. Как работает сортировка слиянием (Merge Sort)? Почему она считается эффективной?
7. В чем особенность сортировки подсчетом (Counting Sort)? В каких ситуациях она применяется?
8. Сравните временную сложность различных алгоритмов сортировки в среднем и худшем случае.
9. Что такое устойчивая сортировка? Приведите пример устойчивого алгоритма.
10. Объясните, как работает сортировка Шелла (Shell Sort). Почему она может быть быстрее других методов?
11. Назовите линейный алгоритм поиска. В каких случаях он эффективен?
12. Опишите алгоритм бинарного поиска. Какие условия должны быть выполнены для его применения?
13. Что такое массив? Какие основные операции можно выполнять с массивами?
14. Что такое связный список? Каковы его преимущества?
15. Объясните разницу между однонаправленным и двунаправленным списками.
16. Что такое стек и очередь? Приведите примеры их использования.
17. Определите понятие хеш-таблицы. Какие методы разрешения коллизий существуют?
18. Что такое множество в Python? Приведите примеры использования множества.

Раздел 3
Графы и деревья

Практическое задание № 16
«Бинарное дерево поиска»

1. Граф

Граф – структура, модель, описывающая объекты и связи между ними.

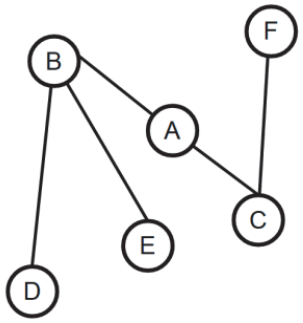
Граф – набор узлов и ребер, через которые связаны узлы.

Граф состоит из:

- **Узлов** (вершин).
- **Ребер**.

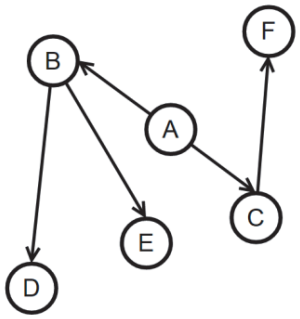
Неориентированный граф (неограф) – граф, ни одному ребру которого не присвоено направление.

Пример неориентированного графа с 6 узлами и 5 ребрами:



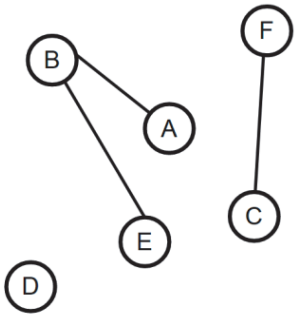
Ориентированный граф (орграф) – граф, ребрам которого присвоено направление. Направленные ребра именуются также дугами.

Пример ориентированного графа:



Связный граф – граф, в котором существует путь между любыми двумя вершинами.

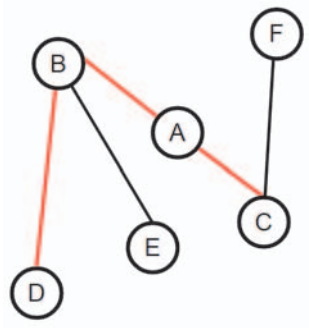
Пример несвязного графа:



Путь – последовательность вершин $x_1, x_2, x_3, \dots, x_k$, в которой каждая пара вершин (x_i, x_{i+1}), $i = 1,2, \dots, k-1$ является ребром, причем все эти ребра различны. Путь соединяет вершины x_1 и x_k.

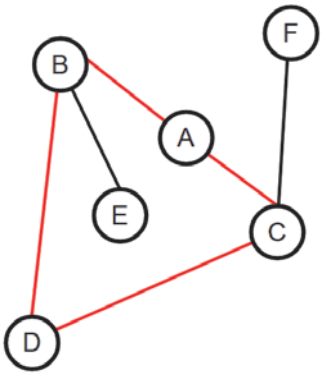
Длина пути – число ребер в пути, т.е. $k-1$.

Например, путь из вершины D в вершину C следующий: D, B, A, C. Длина пути равна $k-1 = 4-1 = 3$:



Цикл – путь, у которого $x_1 = x_k$.

Например, в следующем графе путь D, B, A, C, D является циклом:



2. Дерево

Дерево – связный неориентированный граф без циклов (ациклический), содержащий более 2 вершин.

Дерево – подкатегория графов. Любое дерево – граф. Не любой граф – дерево.

Корень – узел, находящийся на самом верхнем уровне (не являющийся чьим-либо потомком).

Корень – узел с минимальным порядковым номером.

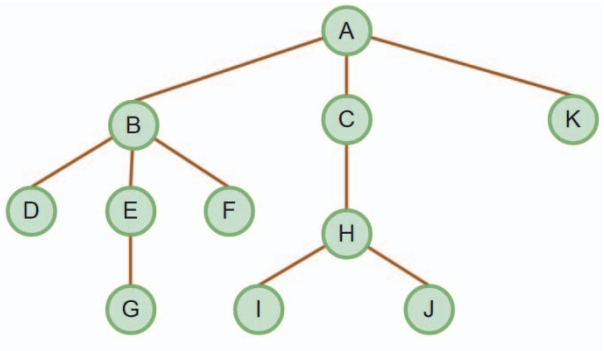
Листья (висячие вершины) – узлы, не имеющие потомков (оба потомка которых равны NULL).

Листья – вершины, которым инцидентно лишь одно ребро.

Если в графе есть ребро (a, b), то говорят, что вершины a и b в нем **смежны**. Говорят, что ребро $e = (a, b)$ **инцидентно** каждой из вершин a и b, а каждая из этих вершин **инцидентна** ребру.

Корневое дерево – дерево, в котором задан корень (дерево изображено сверху вниз и в нем есть самая верхняя вершина).

Пример дерева:



Здесь корень – узел A, листья – узлы D, G, F, I, J, K.

3. Бинарное дерево

Бинарное дерево (Двоичное дерево) – это древовидная структура данных, в которой каждый узел имеет не более двух дочерних узлов, которые называются **левый потомок** и **правый потомок**.

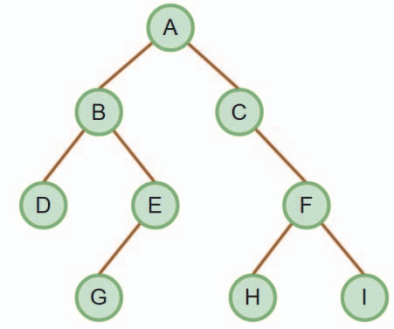
Бинарное дерево – иерархическая структура данных, в которой каждый узел имеет значение и ссылки на левого и правого потомка.

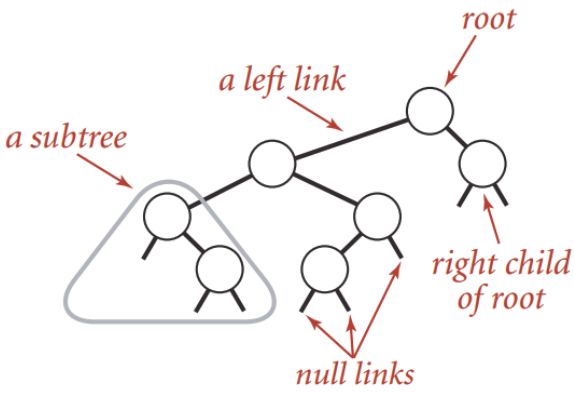
Бинарное дерево – набор узлов, разбитый на три непересекающиеся части (если бинарное дерево не является пустым):

- корень,
- левое поддерево,
- правое поддерево.

Таким образом, бинарное дерево – это **рекурсивная** структура данных [11].

Пример бинарного дерева:



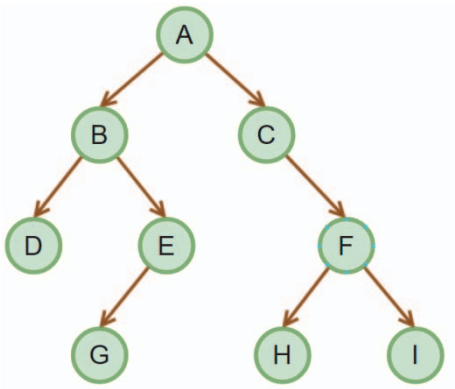


Anatomy of a binary tree

Источник: [24].

Ориентированное бинарное дерево – бинарное дерево, ребрам которого присвоено направление. Направленные ребра именуются также дугами.

Пример ориентированного бинарного дерева *T*:



Уровень вершины (узла) – длина пути от корня дерева до данной вершины. Корень имеет уровень, равный нулю.

Для примера выше:

- узел A имеет уровень 0,
- узлы B, C – уровень 1,

- узлы D, E, F – уровень 2,
- узлы G, H, I – уровень 3.

Высота дерева – наибольшее число ребер от корня до самого дальнего листового узла.

Высота дерева – величина, равная максимальному из уровней вершин.

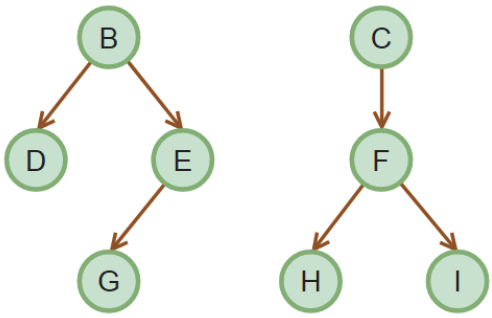
Для примера выше: высота дерева равна 3.

Глубина дерева (количество уровней) – число вершин, которые лежат на максимальной ветви от корня к листьям.

Для примера выше: глубина дерева равна 4.

Поддерево – дерево, содержащее часть вершин главного дерева, причем корнем поддерева может быть любая вершина главного дерева.

Левое поддерево *T1* и правое поддерево *T2* узла A дерева *T*:



4. Бинарное дерево поиска

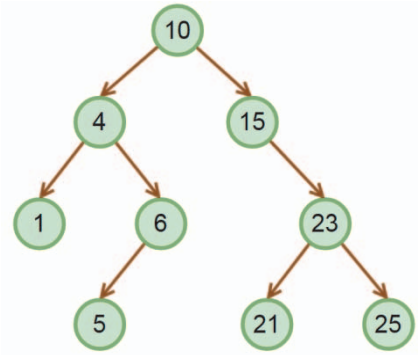
Бинарное дерево поиска (binary search tree, BST) – бинарное дерево, обладающее дополнительными свойствами для каждого узла дерева:

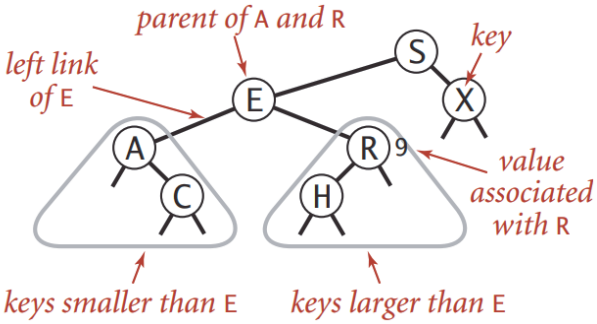
- все узлы левого поддерева содержат значения, меньшие, чем значение в данном узле;
- все узлы правого поддерева содержат значения, большие, чем значение в данном узле.

Бинарное дерево поиска – бинарное дерево, в котором:

- каждый узел имеет сопоставимый ключ (comparable key) и связанное с ним значение;
- ключ в любом узле больше, чем ключи во всех узлах левого поддерева и меньше, чем ключи во всех узлах правого поддерева.

Пример бинарного дерева поиска:





Anatomy of a binary search tree

Источник: [24].

Чем меньшей высоты бинарное дерево поиска, тем быстрее осуществляются сам поиск.

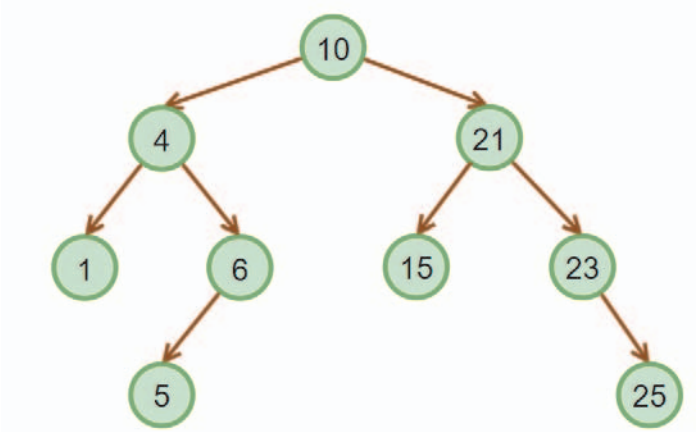
Эффективность работы бинарного дерева поиска зависит от его **сбалансированности**: несбалансированное дерево поиска неэффективно, сбалансированное дерево поиска – эффективно.

4.1 Виды сбалансированных бинарных деревьев поиска:

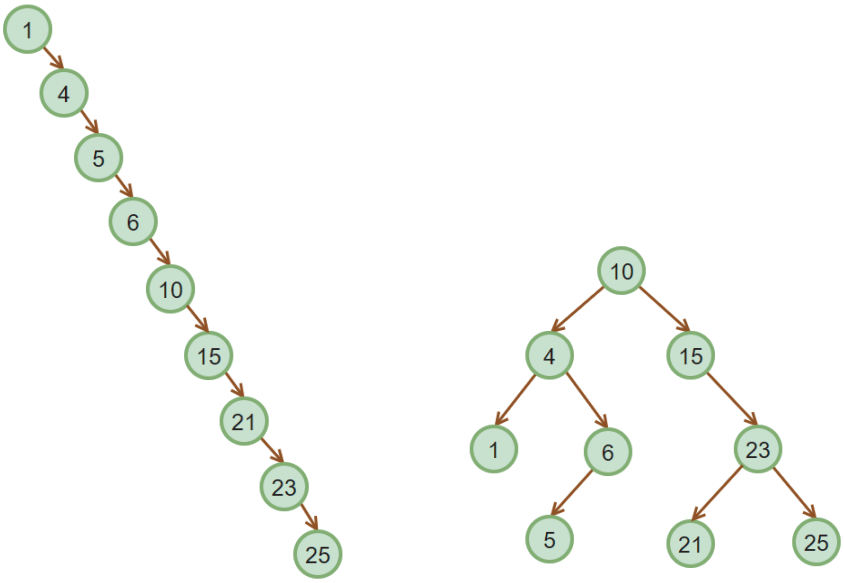
- Идеально сбалансированное бинарное дерево поиска.
- АВЛ-деревья (сокращения от фамилий Г.М. Адельсон-Вельский и Е.М. Ландис).
- Красно-черные деревья.
- Самоперестраивающиеся деревья (splay-деревья) [11].

Идеально сбалансированное бинарное дерево поиска – дерево, у которого для каждого узла выполняется требование: число узлов в левом и правом поддеревьях различается не более чем на 1 [11].

Пример идеально сбалансированного бинарного дерева поиска:



Примеры двух несбалансированных бинарных деревьев поиска, построенных по одному и тому же набору ключей:



При вставке в дерево нового узла или при удалении узла **балансировка** может нарушиться и ее необходимо будет восстанавливать. Т.к. идеальную балансировку поддерживать достаточно сложно, на практике зачастую применяют иные виды сбалансированных деревьев.

4.2. Вставка узла в бинарное дерево поиска

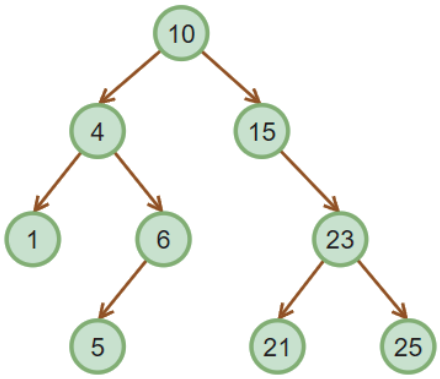
Вставка узла в бинарное дерево поиска происходит следующим образом.

Предшественник узла A – узел, который имеет предыдущее значение относительно значению узла A.

Последователь узла A – узел, который имеет следующее значение после узла A.

Существует правило: если правое поддерево узла A не пустое, то **последователем** узла A будет самый левый узел в правом поддереве узла A (т.е. узел с минимальным значением в правом поддереве).

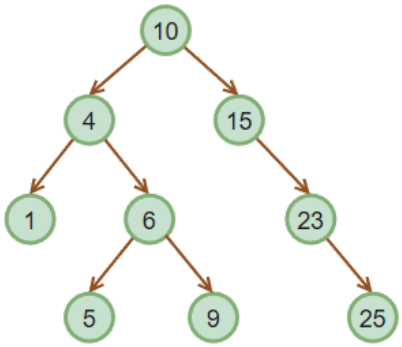
Например, в следующем бинарном дереве для узла 10 последователем является узел со значением 15, для узла 4 последователем является узел 5:



Если правое поддерево узла A пустое, то для поиска **последователя** узла A необходимо идти вверх по дереву, пока значение очередного узла не будет больше значения узла A (т.е. пока не будет найден узел, имеющий левого потомка).

Для примера выше последователем узла 5 будет узел 6 (переход вверх по дереву на 1 уровень).

Для следующего дерева последователем узла 9 будет узел 10 (переход вверх по дереву на 3 уровня):

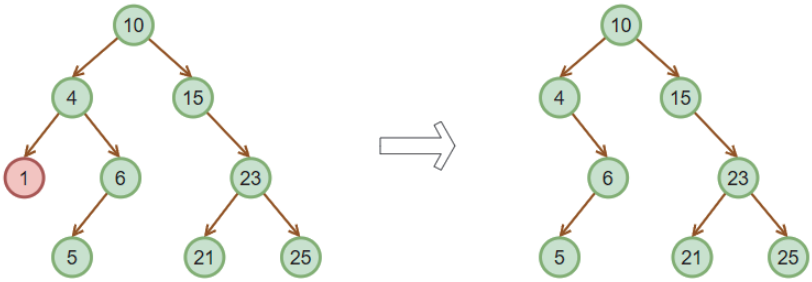


Для вставки нового узла A в бинарное дерево поиска необходимо идти с корня дерева к наиболее близкому по значению узлу (предшественнику или последователю узла A), на каждом уровне сравнивая значения текущего узла и значение добавляемого узла A.

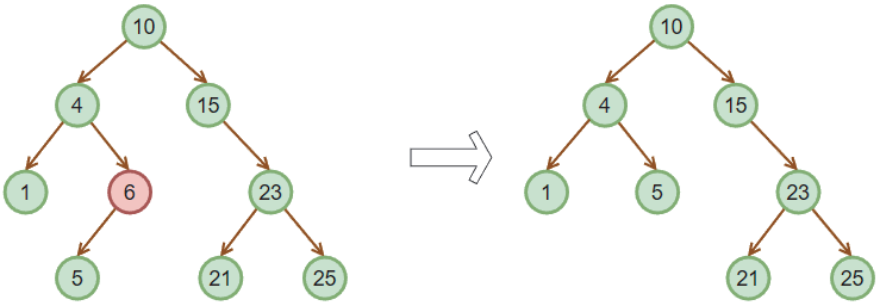
4.3. Удаление узла из бинарного дерева поиска

При удалении узла из двоичного дерева поиска возможно 3 разные ситуации [11]:

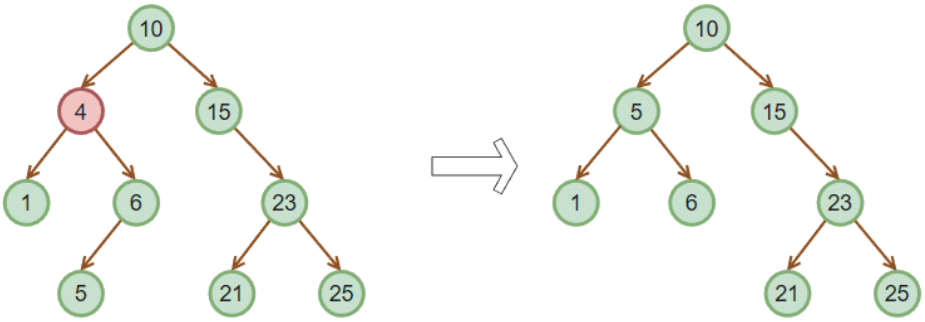
1) Узел листовой: узел удаляется, поддерево его предка становится пустым:



2) Узел имеет 1 потомка: узел удаляется, его потомок переходит к его предку:



3) Узел имеет 2 потомков: узел удаляется, его место занимает его последователь:



Время выполнения каждой операции для бинарного дерева поиска приведено в таблице 16.1

Таблица 16.1 – Быстродействие массива, связного списка и бинарного дерева поиска

Структура данных	Средний случай			Худший случай		
	Чтение	Вставка	Удаление	Чтение	Вставка	Удаление
Массив	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$
Связный список	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(1)$	$O(1)$
Бинарное дерево поиска	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(n)$	$O(n)$	$O(n)$

5. Реализация бинарного дерева поиска в Python

Узел бинарного дерева можно представить в виде структуры данных, состоящей из следующих полей [11]:

- данные, обладающие ключом, по которому их можно идентифицировать;
- указатель на левое поддерево;
- указатель на правое поддерево;
- указатель на родителя (необязательное поле).

Определим класс бинарного дерево поиска:

```
5   # Определние класса бинарного дерева поиска
6   class BTree:
7
8       # Конструктор
9       def __init__(self, value = None, link_left = None,
10                   link_right = None, link_parent = None):
11          self.value = value              # Значение узла
12          self.link_left = link_left      # Ссылка на левое поддерево
13          self.link_right = link_right    # Ссылка на правое подерево
14          self.link_parent = link_parent  # Ссылка на родителя
```

Создадим экземпляр класса под именем `bt` и установим корневой узел со значением 10:

```
104 bt = BTree(10)
```

Добавим узел слева (значение 7) и узел справа (значение 12) путем создания новых экземпляров класса:

```
105 bt.link_left = BTree(7, link_parent = bt)
106 bt.link_right = BTree(12, link_parent = bt)
```

Переход по узлам осуществляется через обращение к атрибутам экземпляра класса:

```
In [81]: bt.value
Out[81]: 10

In [82]: bt.link_left.value
Out[82]: 7

In [83]: bt.link_right.value
Out[83]: 12

In [84]: bt.link_right.link_parent.value
Out[84]: 10
```

Создадим метод поиска узла по значению. Суть метода заключается в переходе по узлам на уровень ниже до тех пор, пока не будет достигнут листовой узел или не найдено искомое значение:

```
    # Метод поиска узла по значению
    def search(self, value_search):

        node = self # Текущий узел (корень)

        # Спуск по ветвям дерева
        while node != None and value_search != node.value:
            if value_search < node.value:
                node = node.link_left  # Переход на уровень ниже влево
            else:
                node = node.link_right # Переход на уровень ниже вправо

        return node
```

При этом если искомого значения нет в дереве, то метод возвратит `None`, если искомое значение есть в дереве, то метод возвратит узел с этим значением:

```
In [85]: bt.search(10)
Out[85]: <__main__.BTree at 0x1b538f70>

In [86]: bt.search(10).value
Out[86]: 10

In [87]: bt.search(99)

In [88]: type(bt.search(99))
Out[88]: NoneType
```

Задачи

Исходные данные к задачам см. в разделе «Исходные данные к задачам по вариантам».

Задача № 1

1. Для наборов чисел (Добавить: …) построить бинарное дерево поиска, поочередно добавляя элементы в дерево в том порядке, в котором они даны.

2. Удалить из дерева указанные элементы (Удалить: …) в том порядке, в котором они даны. Пункт 2 выполняется после завершения пункта 1.

Задачу № 1 можно выполнить, например, средствами сайта diagrams.net, либо с помощью Microsoft Visio.

Задача № 2

Реализовать **бинарное дерево поиска** путем создания класса в Python. При этом в виде методов класса реализовать следующие операции:

- Поиск узла по значению.
- Поиск узла с минимальным значением.
- Поиск узла с максимальным значением.
- Вставка (добавление узла).
- Поиск последователя для узла бинарного дерева поиска.
- Удаление узла (при этом предусмотреть 3 случая: удаляемый узел не имеет потомков, имеет 1 потомка, имеет 2 потомков).
- *Дополнительно: отображение всего дерева в консоли.*

Исходные данные к задачам по вариантам

Вариант 1:
Добавить: 27, 34, 17, 20, 10, 5, 15, 11, 14, 12, 16, 40, 33, 37
Удалить: 33, 15, 14

Вариант 2:
Добавить: 33, 40, 23, 26, 18, 22, 46, 39, 43, 16, 11, 21, 17, 20
Удалить: 39, 21, 20

Вариант 3:
Добавить: 31, 38, 21, 24, 14, 9, 19, 15, 18, 16, 20, 44, 37, 41
Удалить: 37, 19, 18

Вариант 4:
Добавить: 41, 48, 31, 34, 24, 19, 29, 25, 28, 26, 30, 54, 47, 51
Удалить: 47, 29, 28

Вариант 5:
Добавить: 32, 39, 22, 25, 15, 10, 20, 16, 19, 17, 21, 45, 38, 42
Удалить: 38, 20, 19

Вариант 6:
Добавить: 39, 46, 29, 32, 22, 17, 27, 23, 26, 24, 28, 52, 45, 49
Удалить: 45, 27, 26

Вариант 7:
Добавить: 40, 47, 30, 33, 23, 18, 28, 24, 27, 25, 29, 53, 46, 50
Удалить: 46, 28, 27

Вариант 8:
Добавить: 34, 41, 24, 27, 17, 12, 22, 18, 21, 19, 23, 47, 40, 44
Удалить: 40, 22, 21

Вариант 9:
Добавить: 29, 36, 19, 22, 12, 7, 17, 13, 16, 14, 18, 42, 35, 39
Удалить: 35, 17, 16

Вариант 10:
Добавить: 46, 53, 36, 39, 29, 24, 34, 30, 33, 31, 35, 59, 52, 56
Удалить: 52, 34, 33

Вариант 11:
Добавить: 45, 51, 36, 39, 28, 24, 34, 30, 85, 31, 35, 59, 44, 56
Удалить: 36, 28, 44

Вариант 12:
Добавить: 40, 33, 37, 27, 34, 17, 20, 10, 5, 15, 11, 14, 12, 16,
Удалить: 33, 15, 14

Вариант 13:
Добавить: 21, 17, 20, 18, 22, 46, 39, 43, 33, 40, 23, 26, 16, 11,
Удалить: 39, 21, 20

Вариант 14:
Добавить: 29, 25, 28, 26, 30, 54, 47, 51, 41, 48, 31, 34, 24, 19
Удалить: 47, 29, 28

Вариант 15:
Добавить: 23, 18, 28, 24, 27, 25, 29, 53, 46, 50, 40, 47, 30, 33
Удалить: 46, 28, 27

Практическое задание № 17
«АВЛ-дерево»

Виды сбалансированных бинарных деревьев поиска:

- Идеально сбалансированное бинарное дерево поиска.
- АВЛ-деревья (сокращения от фамилий Г.М. Адельсон-Вельский и Е.М. Ландис).
- Красно-черные деревья.
- Самоперестраивающиеся деревья (splay-деревья) [11].

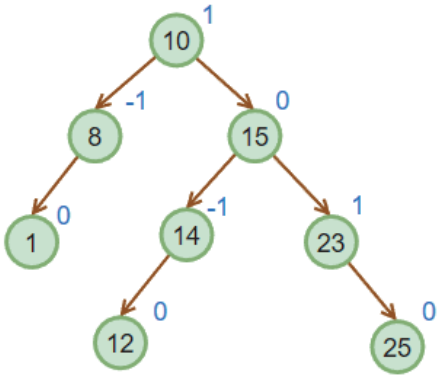
АВЛ-дерево

АВЛ-дерево (AVL tree) (сбалансированное по высоте бинарное дерево поиска) – дерево, для каждой вершины которого выполняется требование: высота левого и правого поддеревьев различаются не более, чем на 1.

Показатель баланса – разность высот правого и левого поддеревьев АВЛ-дерева.

Высота дерева – наибольшее число ребер от корня до самого дальнего листового узла.

В АВЛ-дереве показатель баланса для каждого узла, включая корень, по модулю не превосходит 1:



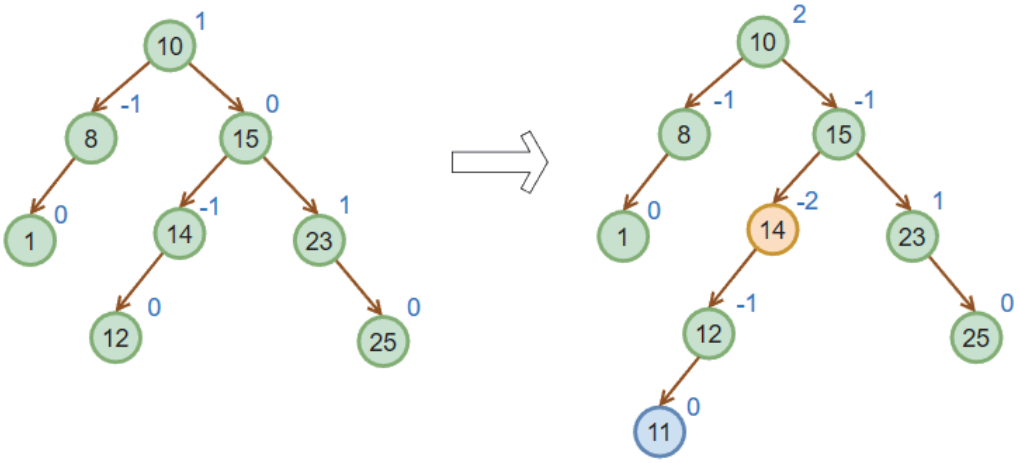
Вставка узла в АВЛ-дерево

При добавлении нового узла может произойти разбалансировка сразу в нескольких узлах, но все они будут лежать на пути от этого добавленного узла к корню.

Существует правило: чтобы найти корень поддерева, которое понадобится перестраивать, надо подниматься вверх по дереву от вновь добавленного узла до тех пор, пока не найдется первый узел, в котором нарушена сбалансированность (**опорный узел**).

Опорный узел – первый узел, в котором нарушена сбалансированность при переходе от добавленного узла к корню АВЛ-дерева [11].

Например, при добавлении узла 11 балансировка нарушилась в двух узлах: 10, 14. При этом опорным узлом будет узел 14:

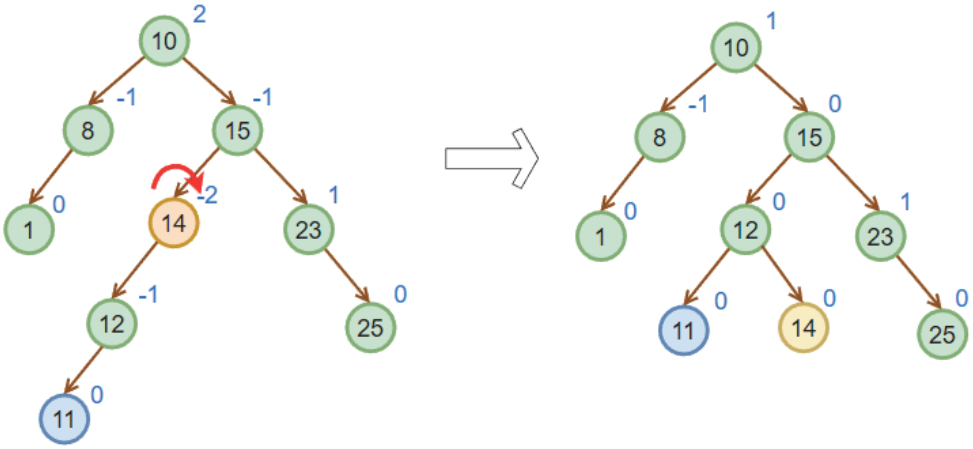


Балансировка АВЛ-дерева – возвращение баланса дереву путем вращения или, другими словами, поворотов (одного или двух поворотов).

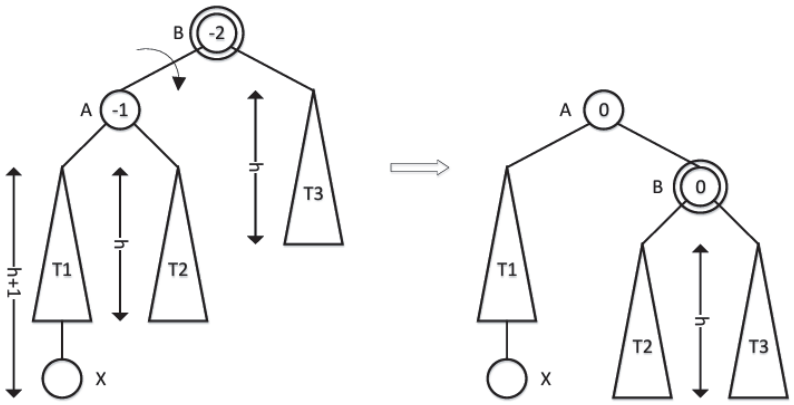
Существует 4 случая, в зависимости от того, в какое поддерево опорного узла был добавлен новый узел.

1. В левое поддерево левого потомка опорного узла (ЛЛ).

Производится правый поворот (R): опорный узел поворачивается вправо относительно своего левого потомка [11]:



Общий вид:



Источник: [11].

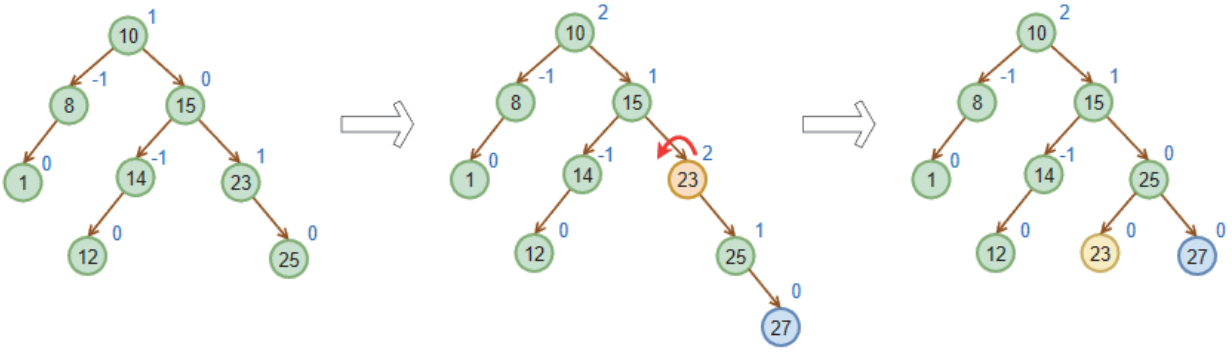
На рисунке выше: B – опорный узел, X – новый узел.

2. В правое поддерево правого потомка опорного узла (ПП).

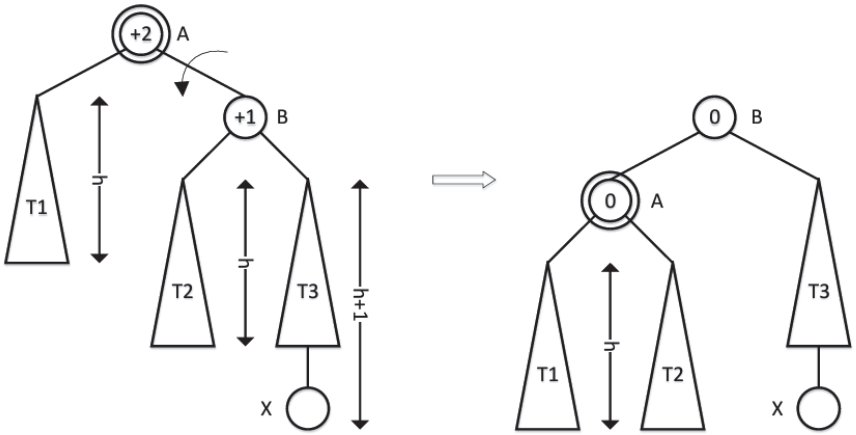
(Случай симметричен случаю 1).

Производится левый поворот (L): опорный узел поворачивается влево относительно своего правого потомка [11].

Например, при добавлении узла 27 балансировка нарушилась в двух узлах: 10, 23. При этом опорным узлом будет узел 23:



Общий вид:



Источник: [11].

3. В правое поддерево левого потомка опорного узла (ПЛ).

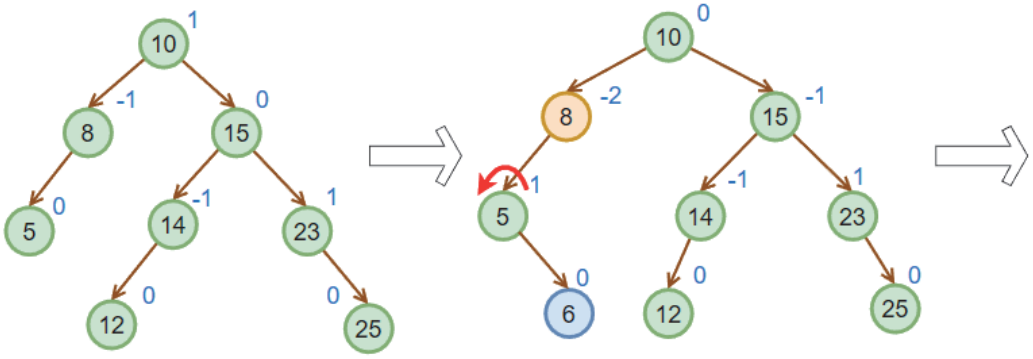
Производится двойной поворот – сначала налево, потом направо (LR):

1) сначала левый потомок опорного узла поворачивается налево относительно своего правого потомка,

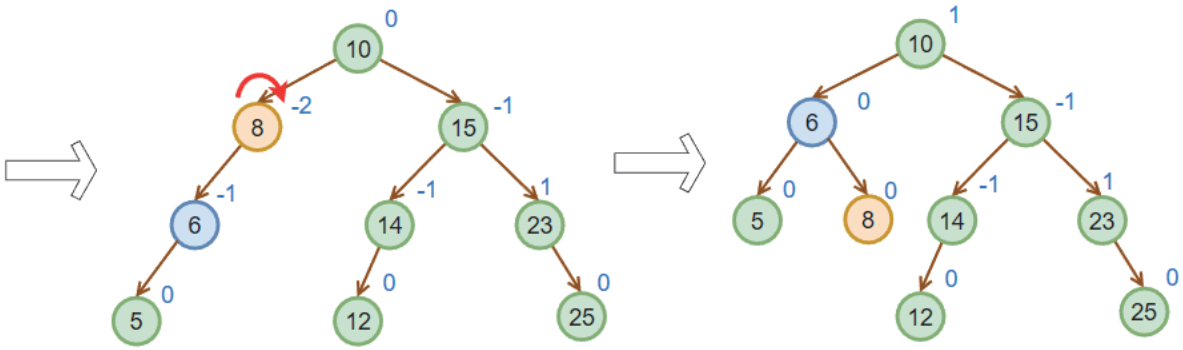
2) затем опорный узел поворачивается направо относительно своего нового левого потомка [11].

Например, при добавлении узла 6 балансировка нарушилась в узле 8. Данный узел будет являться опорным.

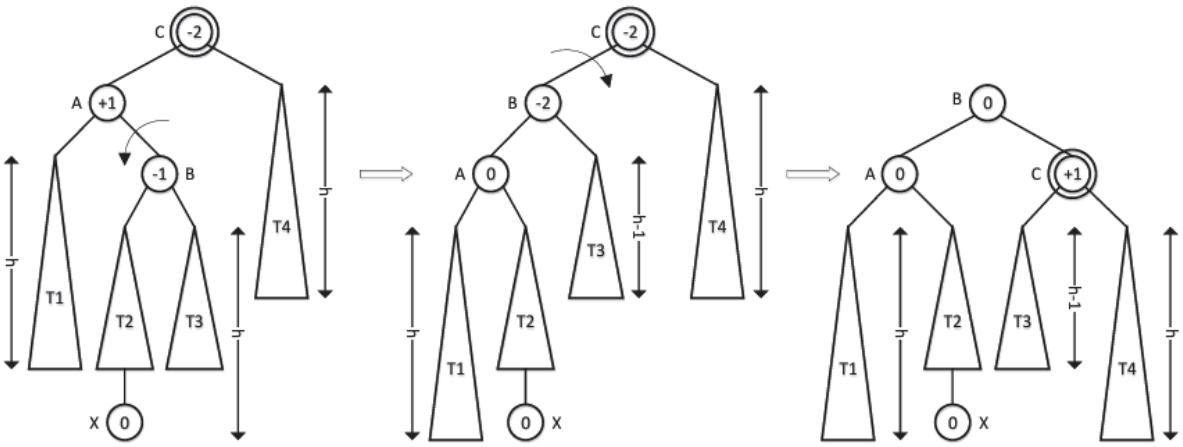
Произведем поворот налево:



Затем поворот направо:



Общий вид:



Источник: [11]

4. В левое поддерево правого потомка опорного узла (ЛП).

(Случай симметричен случаю 3).

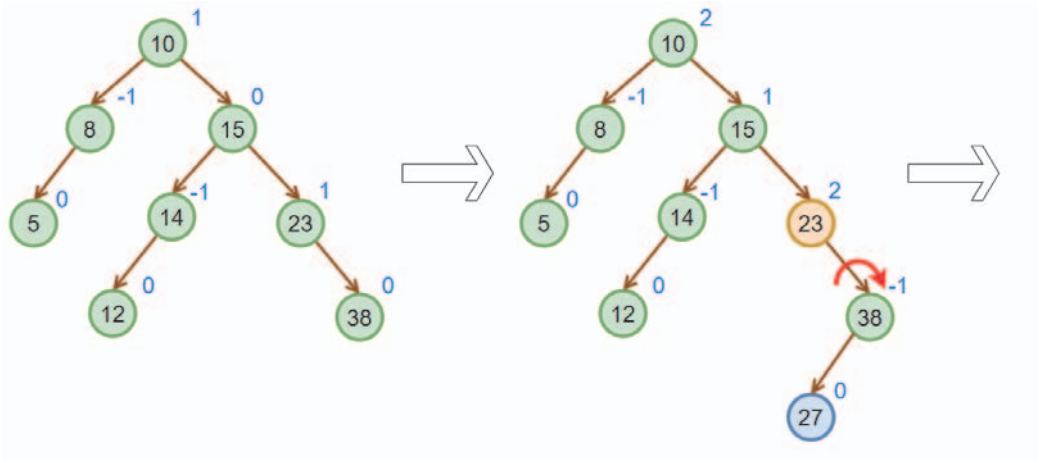
Производится двойной поворот – сначала направо, потом налево (RL):

1) сначала правый потомок опорного узла поворачивается направо относительно своего левого потомка,

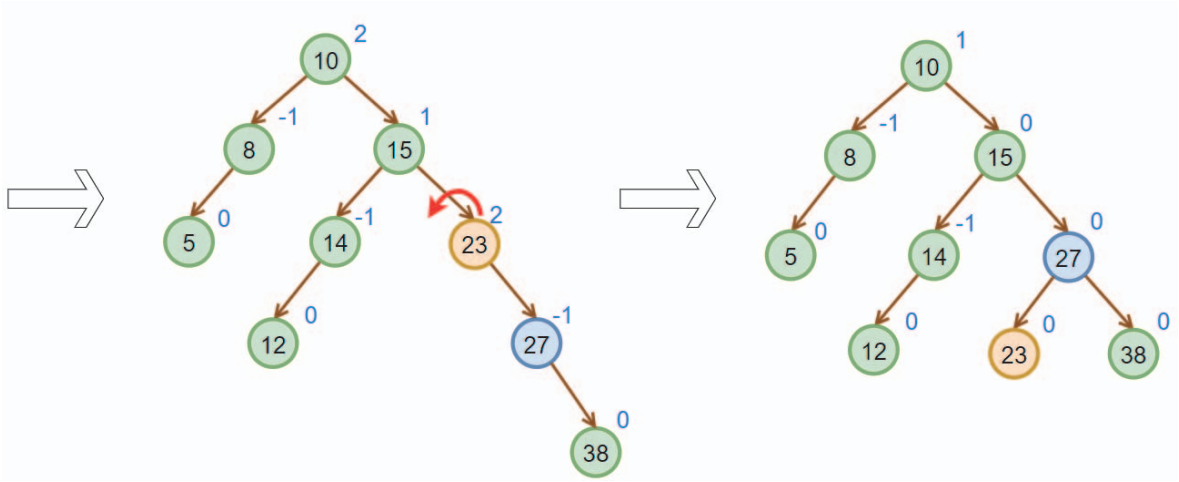
2) затем опорный узел поворачивается налево относительно своего нового правого потомка [11].

Например, при добавлении в следующее дерево узла 27 балансировка нарушилась в двух узлах: 10, 23. Узел 23 будет являться опорным.

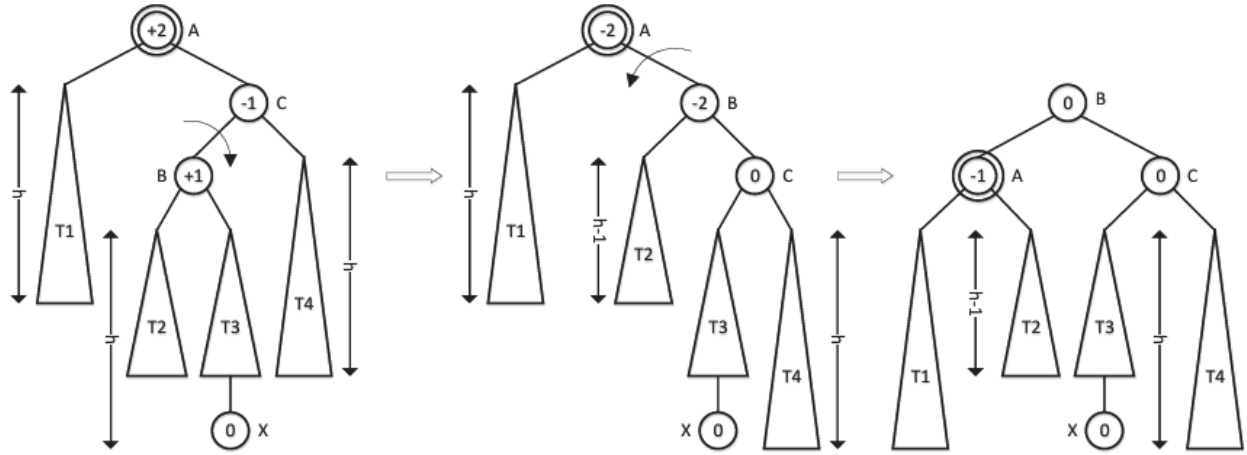
Произведем поворот направо:



Затем поворот налево:



Общий вид:



Источник: [11]

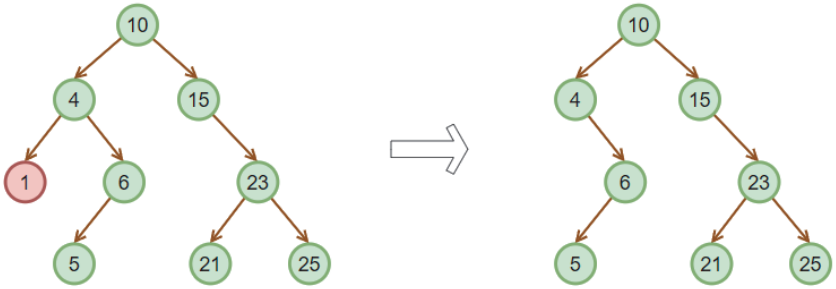
Удаление узла из АВЛ-дерева

1. Этап удаления

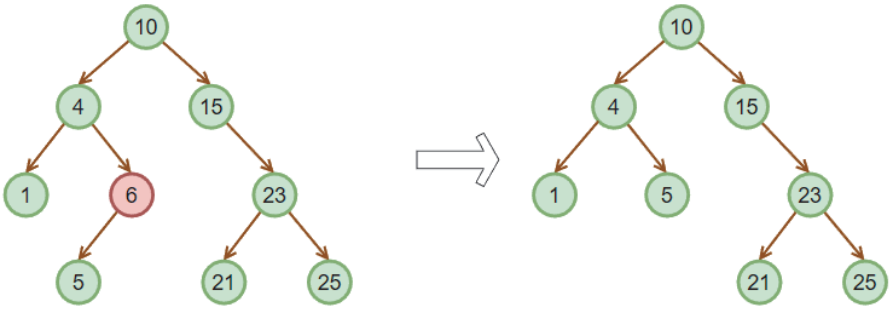
Удаление узла из АВЛ-дерева происходит так же, как и удаление узла из обычного двоичного дерева поиска: если у узла менее двух сыновей, то удаляется сам узел, а если два сына, то удаляемым узлом становится его последователь, информация (ключ) из которого предварительно переписывается в удаляемый узел [11].

При удалении узла из двоичного дерева поиска возможно 3 разные ситуации:

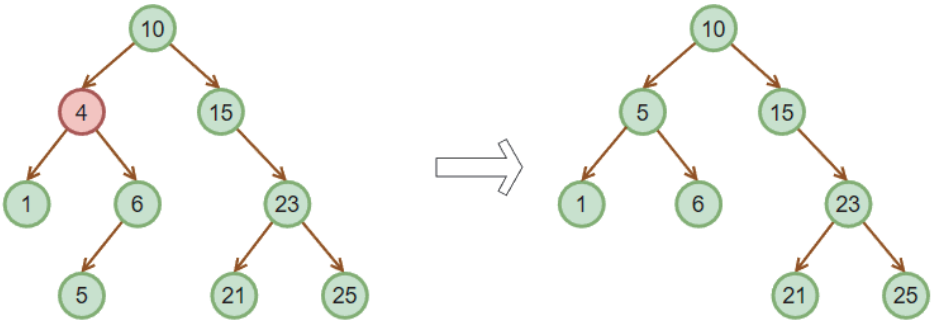
1) Узел листовой: узел удаляется, поддерево его предка становится пустым:



2) Узел имеет 1 потомка: узел удаляется, его потомок переходит к его предку.



3) Узел имеет 2 потомков: узел удаляется, его место занимает его последователь:



Чтобы после удаления сохранились свойства АВЛ-дерева, возможно, понадобится выполнить балансировку.

2. Этап балансировки

После удаления узла необходимо подниматься вверх по пути от удаленного узла к корню и проверять в этих узлах баланс. Если в узле баланс нарушен, то надо выполнить соответствующий поворот – одинарный или двойной.

Опорный узел – первый узел, в котором нарушена сбалансированность при переходе от удаленного узла к корню АВЛ-дерева.

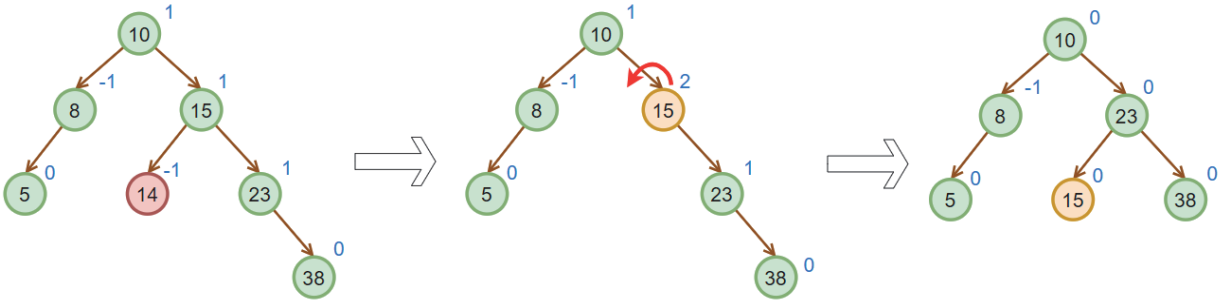
При балансировке после удаления используются те же виды поворотов, что и после вставки узла в дерево. Рассмотрим основные правила.

1. Левое поддерево стало ниже правого на 2 уровня.

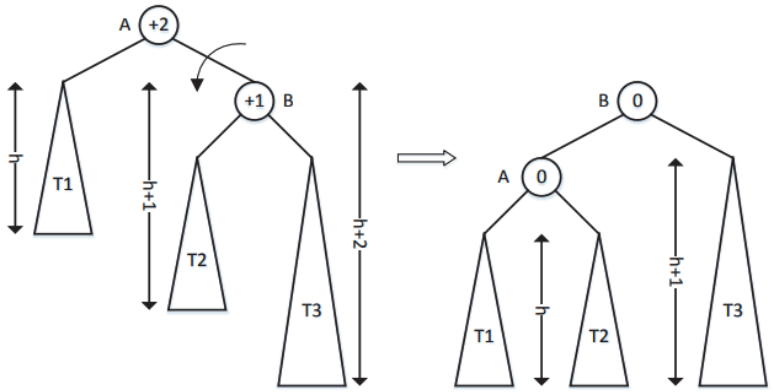
а. У правого сына (B) высота правого поддерева больше, либо равна высоте левого поддерева (height(T3) ≥ height(T2))

Необходимо произвести левый поворот (L): опорный узел (A) поворачивается налево относительно своего правого сына (B) [11]..

Пример: У правого сына (B) опорного узла высота правого поддерева больше высоты левого поддерева (height(T3) > height(T2):

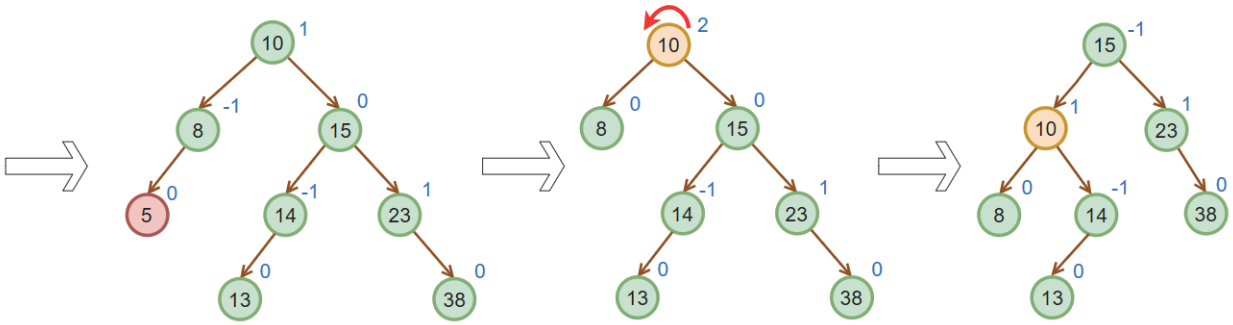


Общий вид:

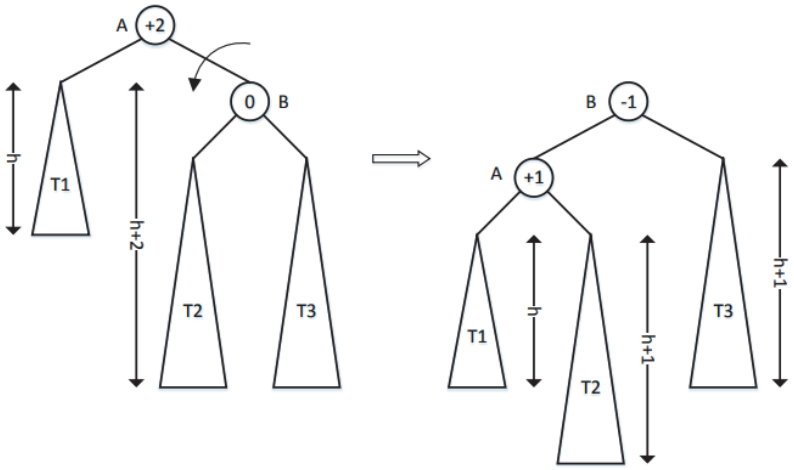


Источник: [11].

Пример: У правого сына (B) опорного узла высота правого поддерева равна высоте левого поддерева (height(T3) = height(T2):



Общий вид:



Источник: [11].

b. У правого сына (C) опорного узла высота правого поддерева меньше высоты левого поддерева (height(T4) < height(B))

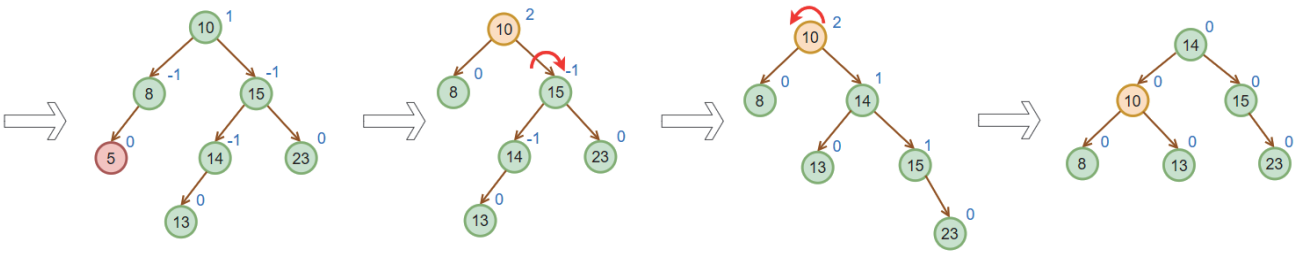
Необходимо произвести двойной поворот — направо, потом налево (RL):

1) сначала правый сын опорного узла (C) поворачивается направо относительно своего левого сына (B),

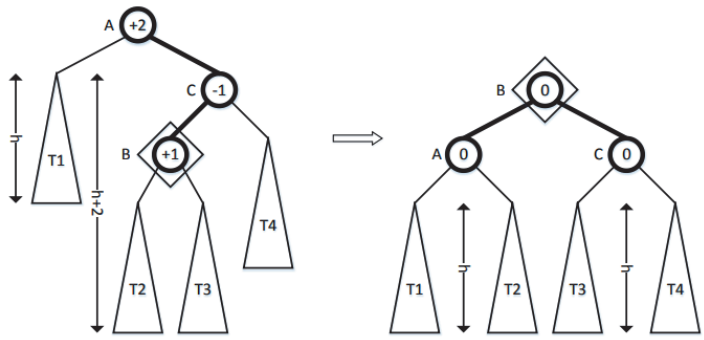
2) затем опорный узел (A) поворачивается налево относительно своего нового правого сына (B).

Поддерево с корнем в B должно иметь высоту h+1. При этом высоты поддеревьев T2 и T3 могут быть равны h, а могут различаться на 1: либо h и h-1, либо h-1 и h [11].

Пример:



Общий вид:



Источник: [11].

2. Правое поддерево стало ниже левого на 2 уровня

(Случай, симметричный случаю 1)

a. У левого сына высота левого поддерева больше, либо равна высоте правого поддерева.

Случай, симметричный случаю 1a.

b. У левого сына высота левого поддерева меньше высоты правого поддерева.

Случай, симметричный случаю 1b.

Время выполнения каждой операции для бинарного дерева поиска, сбалансированного по АВЛ, приведено ниже:

Таблица 17.1 – Быстродействие АВЛ-дерева

Случай	Операция		
	Чтение	Вставка	Удаление
Средний случай	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Худший случай	$O(\log n)$	$O(\log n)$	$O(\log n)$

Реализация АВЛ-дерева

Узел АВЛ-дерева можно представить в виде структуры данных, состоящей из следующих полей:

- данные, обладающие ключом, по которому их можно идентифицировать;
- указатель на левое поддерево;
- указатель на правое поддерево;
- указатель на родителя (необязательное поле);
- показатель баланса [11].

Задачи

Исходные данные к задачам см. в разделе «Исходные данные к задачам по вариантам» Практического задания № 16.

Задача № 1

1. Для наборов чисел (Добавить: …) построить АВЛ-дерево, поочередно добавляя элементы в дерево в том порядке, в котором они даны. При необходимости производить балансировку.

Задачу № 1 можно выполнить, например, средствами сайта diagrams.net, либо с помощью Microsoft Visio.

Задача № 2

Взять за основу построенное АВЛ-дерево из ПЗ 16 (задача 1).

Удалить такие 4 узла, чтобы выполнялись случаи 1a, 1b, 2a, 2b. При необходимости добавить дополнительные узлы в АВЛ-дерево.

Задачу № 1 можно выполнить, например, средствами сайта diagrams.net, либо с помощью Microsoft Visio, Paint и т.д.

Задача № 3

Реализовать **АВЛ-дерево** путем создания класса (классов) в Python. При этом в виде методов класса реализовать следующие операции:

- Поиск узла по значению.
- Вставка узла (при этом предусмотреть балансировку АВЛ-дерева).
- Удаление узла (при этом предусмотреть балансировку АВЛ-дерева).
- *Дополнительно: отображение всего дерева в консоли.*

Практическое задание № 18
«2-3 дерево. Левостороннее красно-черное дерево»

АВЛ-деревья исторически были первым примером использования сбалансированных деревьев поиска. В настоящее время более популярны красно-черные деревья (**КЧ-деревья**).

Логика КЧ-дерева основана на так называемом **2-3 дереве** (B-дереве 3 порядка) [11].

B-дерево

B-дерево (B-tree) – сбалансированное, ветвистое дерево поиска.

Основные отличия B-дерева от бинарного дерева поиска:

- узел может иметь несколько значений,
- узел может иметь более двух потомков.

Ветвистость – свойство каждого узла дерева ссылаться на большое число узлов-потомков.

B-дерево обобщает, расширяет бинарное дерево поиска, допуская узлы с более чем двумя дочерними элементами.

Порядок B-дерева – максимально возможное число потомков у любого узла дерева.

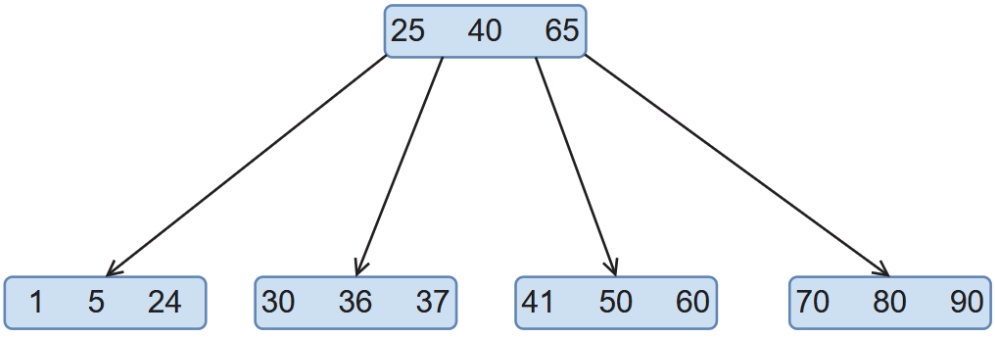
Свойства B-дерева порядка m:

1. Каждый узел имеет не более m потомков.
2. Каждый внутренний узел (не лист и не корень) имеет не менее $m/2$ потомков.
3. Каждый нелистовой узел имеет не менее 2 потомков.
4. Все листья находятся на одном уровне.
5. Нелистовой узел с k потомками имеет k-1 значений (полей, ключей).

Значения каждого узла B-дерева являются, своего рода, разделителями значений для его поддеревьев.

Например, если узел имеет 2 значения a и b, то он может иметь 3 потомка (3 поддерева). Значения узлов его левого поддерева будут меньше a, значения узлов среднего поддерева буду находится в диапазоне от a до b, значения узлов правого поддерева будут больше b.

Пример B-дерева порядка 4:



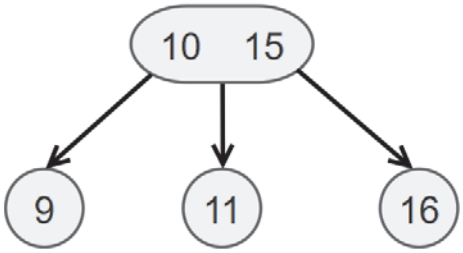
2-3 дерево

2-3 дерево (2-3 tree) – B-дерево, каждый узел которого имеет:

- либо 2 потомка и 1 значение (1 поле данных),
- либо 3 потомка и 2 значения (2 поля данных).

2-3 дерево – это B-дерево порядка 3.

Пример 2-3 дерева:



Балансировка 2-3 дерева

Рассмотрим этапы формирования и балансировки 2-3 дерева.

1) Создание 2-3 дерева с корнем 10.

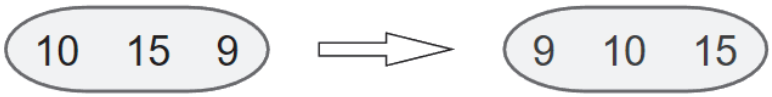
При формировании 2-3 дерева первый и единственный узел становится корнем:

2) Добавление значения 15.

Новое значение добавляется в корень и происходит сортировка двух значений:

3) Добавление значения 9.

Новое значение добавляется в корень и происходит сортировка трех значений:



Однако узел может содержать только 1 или 2 значения, поэтому необходимо провести **балансировку**.

Балансировка необходима, когда узел имеет 3 значения.

В результате балансировки все узлы 2-3 дерева должны иметь 1 или 2 значения, при этом сохранить свойства отсортированного дерева поиска.

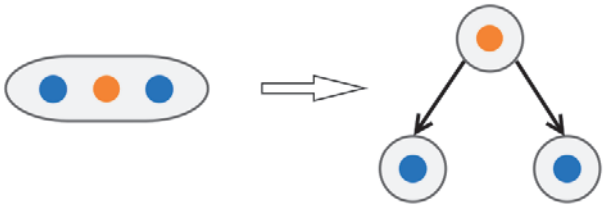
Существует 3 возможных случая, когда узел имеет 3 значения.

1. Узел с 3 значениями является корнем.
2. Предок узла с 3 значениями имеет 1 значение.
3. Предок узла с 3 значениями имеет 2 значения.

Рассмотрим все случаи.

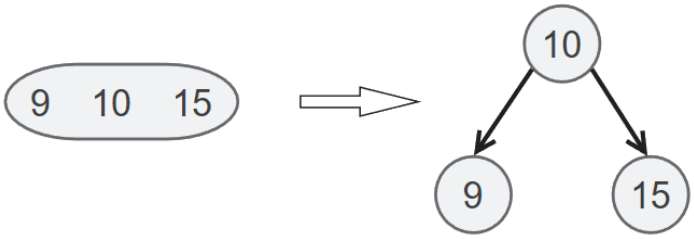
1. Узел с 3 значениями является корнем

В этом случае необходимо представить левое и правое значения (1-е и 3-е) в виде узлов-потомков узла со средним значением. При этом узел со средним значением становится корнем:



Другим словами, происходит проталкивание вверх среднего значения.

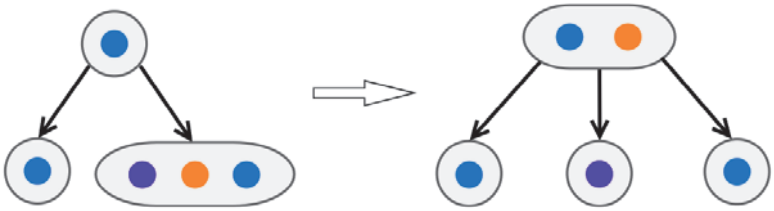
Дерево примет следующий вид:



2. Предок узла с 3 значениями имеет 1 значение

В этом случае:

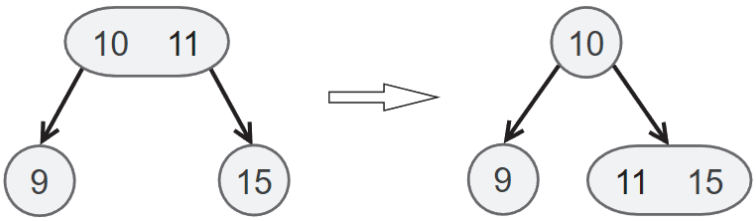
- среднее значение переходит предку,
- крайнее значение представляется как отдельный узел-брат, то есть имеющий того же предка (у предка должно быть 2 значения и 3 потомка).



Продолжим заполнять 2-3 дерево.

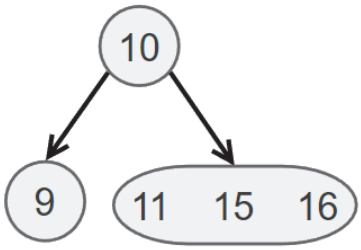
4) Добавим значение 11 в дерево.

Новое значение добавляется в корень и происходит сортировка 2 значений. Однако нелистовой узел с 2 потомками должен иметь 1 значение (свойство 5 B-дерева), то есть корень уже не может содержать 2 значения. Поэтому 11 должно перейти правому потомку, где впоследствии произойдет сортировка 2 значений:



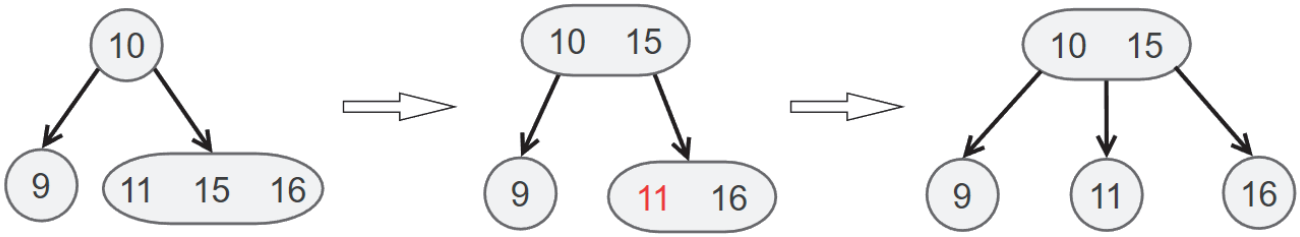
5) Добавим значение 16.

Аналогичным образом оно перейдет правому потомку:



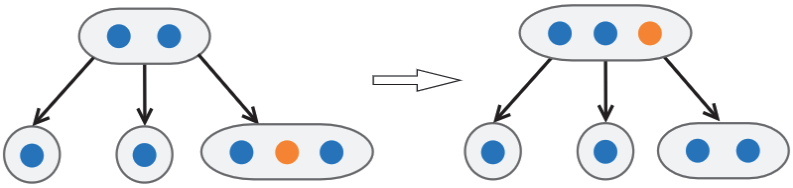
Необходима балансировка для узла с 3 значениями.

Передадим среднее значение 15 предку, однако оставшееся значение 11 уже не может находится в правом поддереве узла 10-15, так как оно не больше 15. Это значение находится в диапазоне от 10 до 15, поэтому его можно представить в виде отдельного узла-потомка по отношению к узлу 10-15:

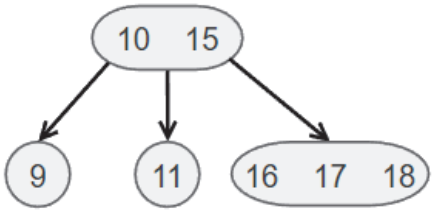


3. Предок узла с 3 значениями имеет 2 значения

В этом случае происходит проталкивание среднего значения вверх до тех пор, пока не возникнет случай 1 или 2:



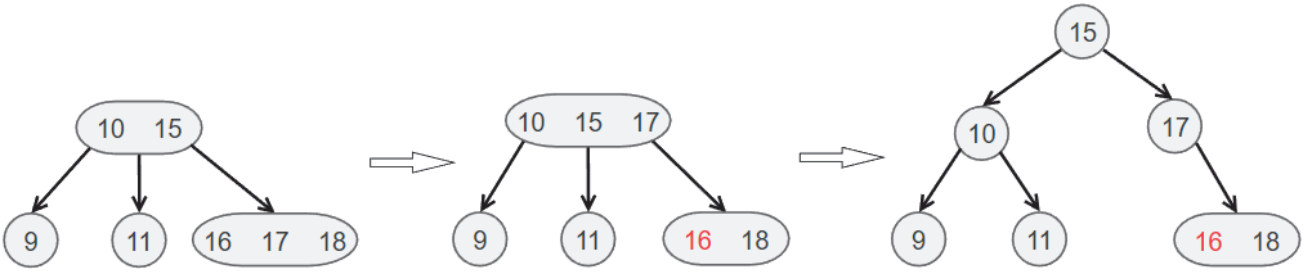
6) Добавим значения 17, 18 в дерево:



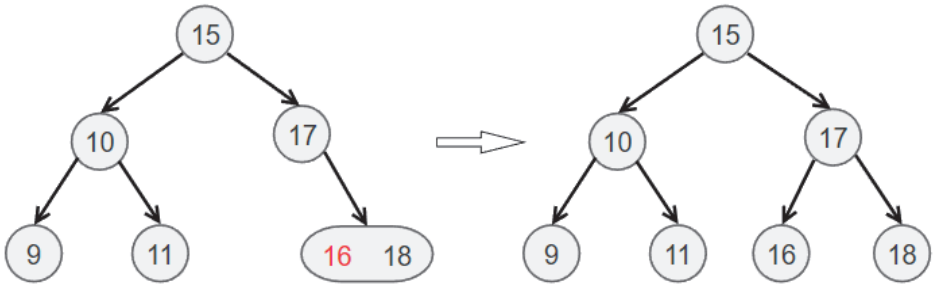
Необходима балансировка для узла с 3 значениями.

Передадим среднее значение 17 предку, однако оставшееся значение 16 уже не может находится в правом поддереве узла 10-15-17. Выделим его красным цветом для наглядности. Узел 10-15-17 мы можем сбалансировать

по правилу для случая 1. Узел 11 при этом перейдет левому потомку 10 (правому потомку 17 он перейти не может по правилам отсортированного дерева поиска):



Наконец, узел 16 не может находится в правом поддереве узла 17, поэтому он переходит в левое поддерево:



Красно-черное дерево

Недостатком 2-3 дерева является сложность реализации. Красно-черное дерево применяется для представления 2-3 дерева в таком формате, чтобы оно было просто в реализации.

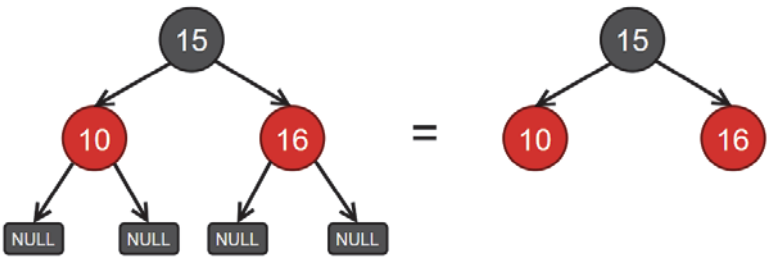
КЧ-дерево (Красно-черное дерево) (Red-Black tree, RB tree) – сбалансированное бинарное дерево поиска, в котором:

- каждый узел хранит дополнительное поле color – цвет узла (красный или черный).
- все узлы внутренние, т.е. нелистовые (листовые узлы определяются как фиктивные листовые узлы) [11].

Фиктивный листовой узел (фиктивный лист, NULL-узел) – узел КЧ-дерева, ссылка на которого равна NULL.

Т.е. листы обычного бинарного дерева в КЧ-дереве не являются листами: ссылки на левого потомка и (или) правого потомка у такого листа равны NULL, при этом считается, что эти потомки и есть фиктивные узлы (NULL-узлы).

Обычно NULL-узлы не отображаются:



Черная высота узла – количество черных узлов на пути от этого узла к узлу, у которого оба сына – фиктивные листья. Сам узел не включается в это число.

Черная высота дерева – черная высота его корня.

Черная высота дерева на рисунке выше равна 0.

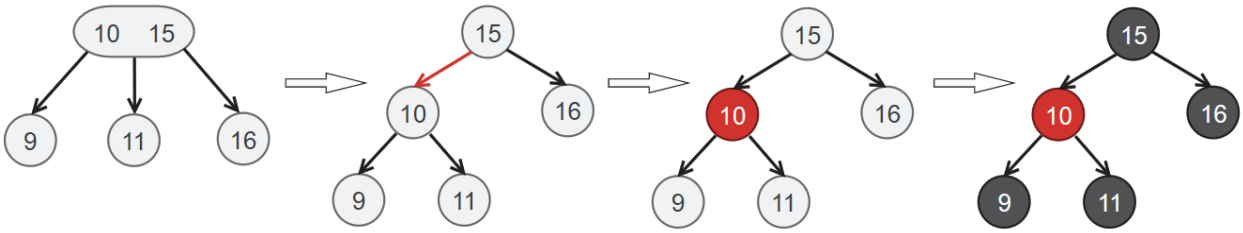
Свойства КЧ-дерева:

1. Каждый узел имеет цвет: красный или черный.
2. Корень дерева – черный.
3. Каждый фиктивный листовой узел – черный.
4. Потомки красного узла – черные. При этом потомками черных узлов могут быть как черные, так и красные узлы.
5. Все пути, идущие от корня к любому фиктивному листу, содержат одинаковое количество черных узлов [11].

Преобразование 2-3 дерева в КЧ-дерево

2-3 дерево можно представить в виде КЧ-дерева. При этом необходимо выделить узлы, являющиеся одним узлом в 2-3 дереве. Графически можно было бы выделить связь между этими узлами красным цветом, однако реализовать это программно не представляется возможным.

Чтобы выделить узел из 2-3 дерева с двумя значениями (например, узел 10-15), в КЧ-дереве красным цветом выделяют узел, который хранит одно из значений узла с двумя значениями из 2-3 дерева (например, выделяют узел 10):



Если выделяется левое значение, то такое КЧ-дерево называется **левосторонним КЧ-деревом (ЛКЧ-деревом)**, если правое, то – **правосторонним КЧ-деревом (ПКЧ-деревом)**.

Левосторонне или правостороннее КЧ-дерево более просто в реализации.

Формирование ЛКЧ-дерева

Существуют правила:

1. Добавляемый в дерево узел окрашивается в красный цвет. Если это первый узел, то он перекрашивается в черный цвет и становится корнем.

2. Для левостороннего КЧ-дерева красные узлы могут лежать только слева, иначе требуется **балансировка**.

После добавления каждого узла происходит проверка выполнимости свойств КЧ-дерева. Если они нарушаются, необходимо проводить **балансировку**.

Балансировка левостороннего КЧ-дерева

Для балансировки КЧ-дерева применяются 3 операции:

1. Левый поворот.
2. Правый поворот.
3. Перекраска («свап» цвета).

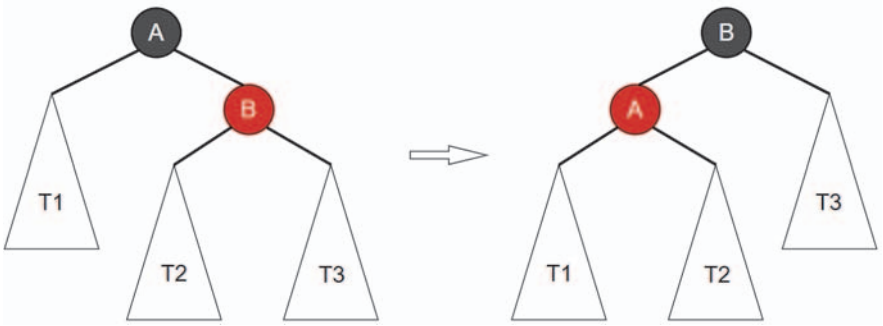
Левый и правый повороты являются стандартными поворотами, использующимися для балансировки АВЛ-деревьев.

1. Левый поворот

Левый поворот узла A применяется в случае, если правый потомок этого узла (узел B) – красный узел.

Во время поворота происходит изменение цвета: узел A становится красным, узел B – черным.

Общая форма:

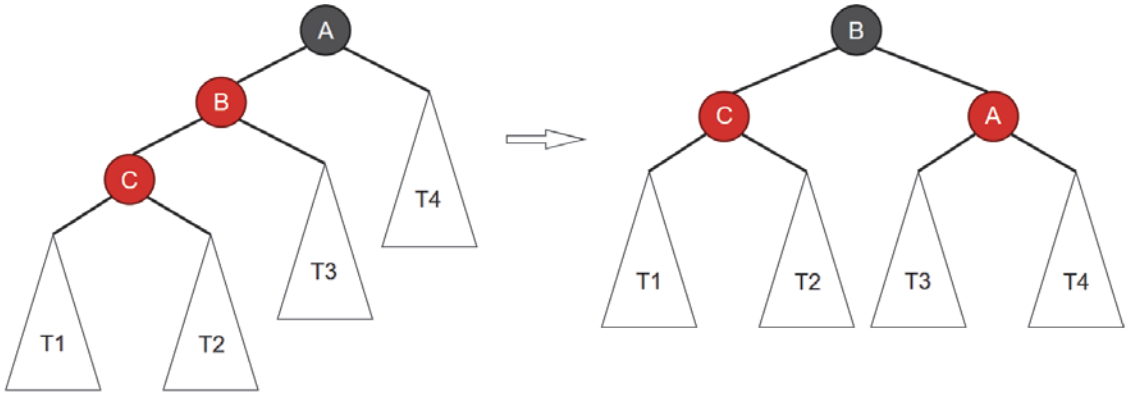


2. Правый поворот

Правый поворот узла A применяется в случае, если левый потомок этого узла (узел B) – красный, и левый потомок узла B (узел C) – так же красный (два красных узла расположены подряд).

Во время поворота происходит изменение цвета: узел A становится красным, узел B – черным.

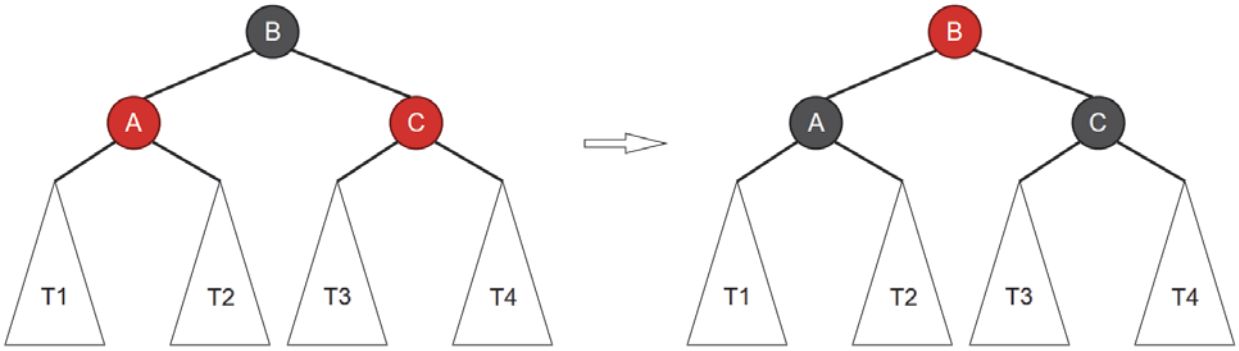
Общая форма:



3. Перекраска

Перекраска (обмен цветами) трех узлов A, B, C происходит в случае, если у черного узла B имеется два красных потомка (A, C).

В результате перекраски происходит изменение цвета: узел B становится красным, два потомка этого узла (A, C) становятся черными.

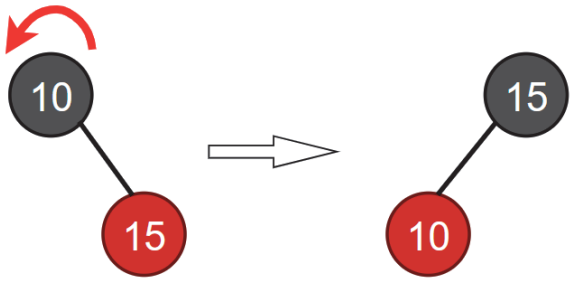


Пример формирования ЛКЧ-дерева

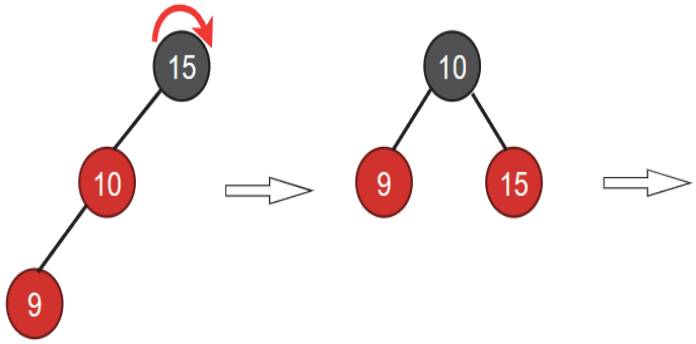
Поэтапно построим левосторонне КЧ-дерево для того же массива значений, который использовался при построении 2-3 дерева: 10, 15, 9, 11, 16, 17, 18.

1) Добавим значение 10. Так как это первый узел, он становится корнем и окрашивается в черный цвет:

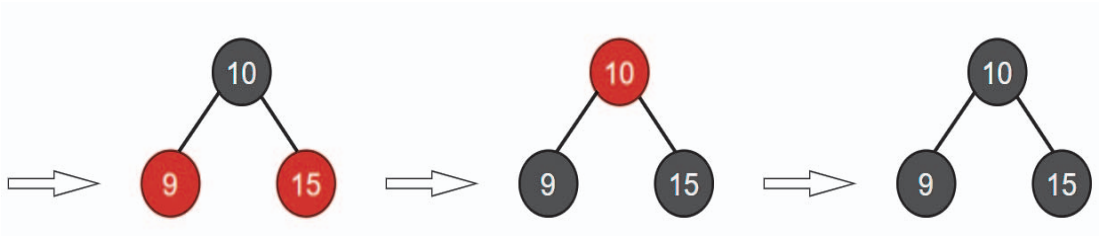
2) Добавим значение 15. Добавленный узел окрашивается в красный цвет. Но так как в ЛКЧ-дереве красные узлы могут быть только левыми потомками, то необходимо применить **левый поворот**, при котором узлы меняют цвет:



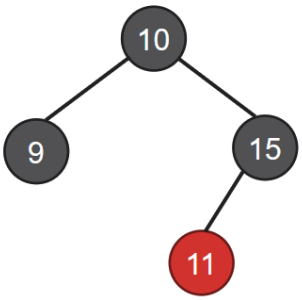
3) Добавим значение 9. Добавленный узел окрашивается в красный цвет. Но нарушается правило 4 для КЧ-дерева: потомки красного узла – черные. Необходимо применить **правый поворот**, при котором узлы 15 и 10 меняют цвет:



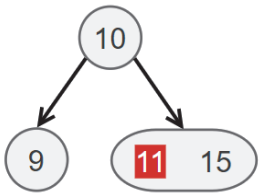
Так как в ЛКЧ-дереве для черного узла левый и правый потомок красные, необходимо применить **перекраску**. Но так как узел 10 является корнем, то он не может быть красным, поэтому он вновь становится черным:



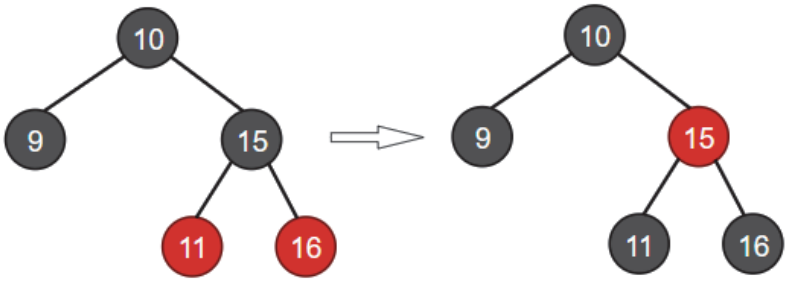
4) Добавим значение 11. Правила ЛКЧ-дерева не нарушились:



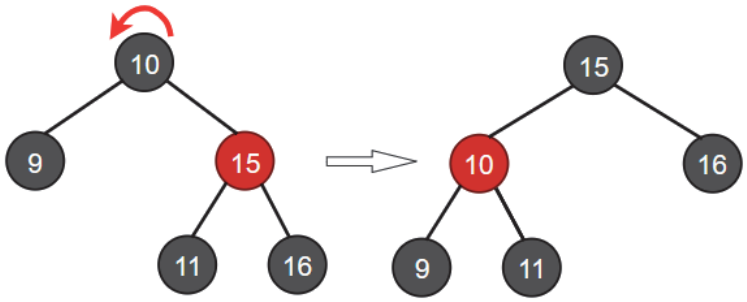
Если сравнить такое ЛКЧ-дерево с полученным ранее 2-3 деревом, то мы увидим, что красный узел 11 и его предок 15 соответствуют узлу 11-15 в 2-3 дереве:



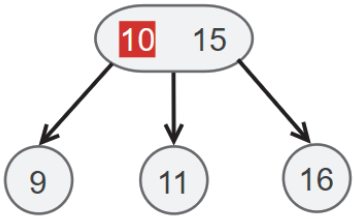
5) Добавим значение 16. Применим **перекраску**:



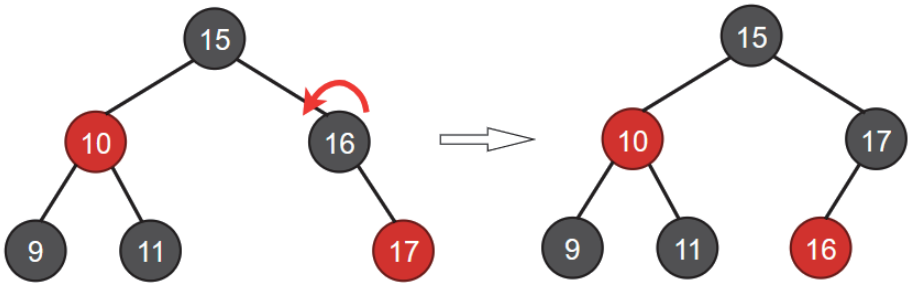
Применим **левый поворот**:



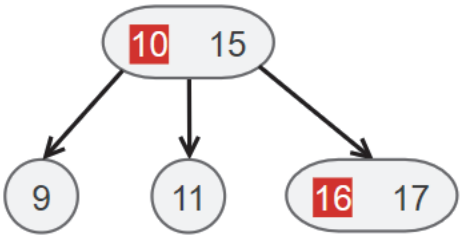
ЛКЧ-дерево соответствует полученному ранее 2-3 дереву:



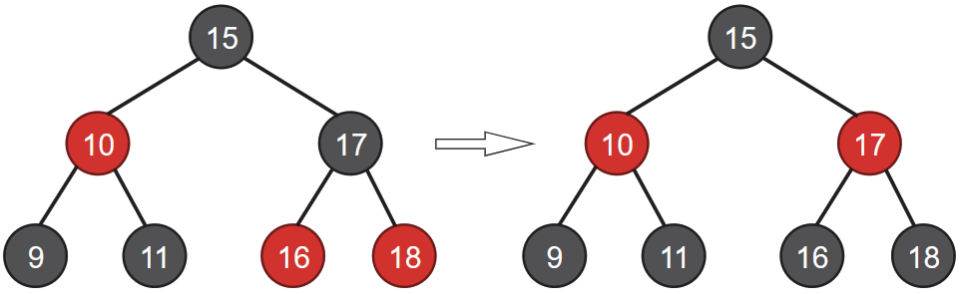
6) Добавим значение 17. Применим **левый поворот**:



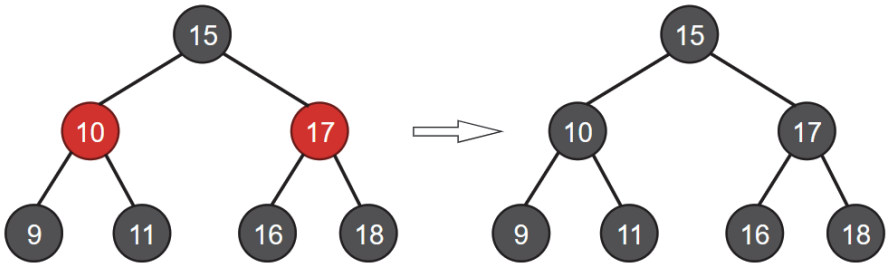
ЛКЧ-дерево соответствует полученному ранее 2-3 дереву:



7) Добавим значение 18. Применим **перекраску**:



Применим второй раз **перекраску**:



В нашем случае итоговое ЛКЧ-дерево приобрело вид обычного бинарного дерева поиска, в котором все узлы имеют один и тот же цвет. Это соответствует итоговому 2-3 дереву, полученному ранее, в котором не было узлов с двумя значениями:

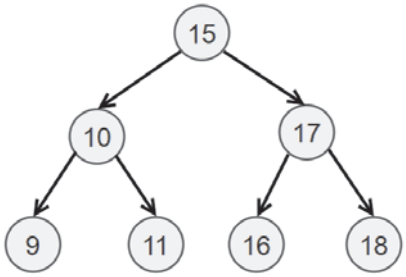


Таблица 18.1 – Быстродействие B-дерева, 2-3 дерева, КЧ-дерева для худшего и среднего случаев

Тип дерева (случай худший и средний)	Операция		
	Чтение	Вставка	Удаление
B-дерево	$O(\log n)$	$O(\log n)$	$O(\log n)$
2-3 дерево	$O(\log n)$	$O(\log n)$	$O(\log n)$
КЧ-дерево	$O(\log n)$	$O(\log n)$	$O(\log n)$

Реализация КЧ-дерева

Узел КЧ-дерева можно представить в виде структуры данных, состоящей из следующих полей:

- данные, обладающие ключом, по которому их можно идентифицировать;
- указатель на левое поддерево;
- указатель на правое поддерево;
- указатель на родителя (необязательное поле);
- цвет узла (красный или черный) [11].

Для самопроверки визуализацию работы 2-3 дерева и КЧ-дерева см. по ссылкам:

- Визуализация структур данных: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
- 2-3 дерево: https://www.cs.usfca.edu/~galles/visualization/BTree.html
- КЧ-дерево: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Задачи

Исходные данные к задачам см. в разделе «Исходные данные к задачам по вариантам» Практического задания № 16.

Задача № 1

Для наборов чисел (Добавить: …) построить:

1. 2-3 дерево.
2. Левостороннее КЧ-дерево.

Необходимо поочередно добавлять элементы в дерево в том порядке, в котором они даны. При необходимости выполнять балансировку дерева.

Задачу № 1 можно выполнить, например, средствами сайта diagrams.net, либо с помощью Microsoft Visio, Paint.

Задача № 2

Реализовать **левостороннее КЧ-дерево** путем создания класса (классов) в Python. При этом в виде методов класса реализовать следующие операции:

- Поиск узла по значению.
- *Вставка узла (при этом предусмотреть балансировку левостороннего КЧ-дерева).*
- *Дополнительно: отображение всего дерева в консоли.*

Практическое задание № 19
«2-3-4 дерево. Красно-черное дерево»

2-3-4 дерево

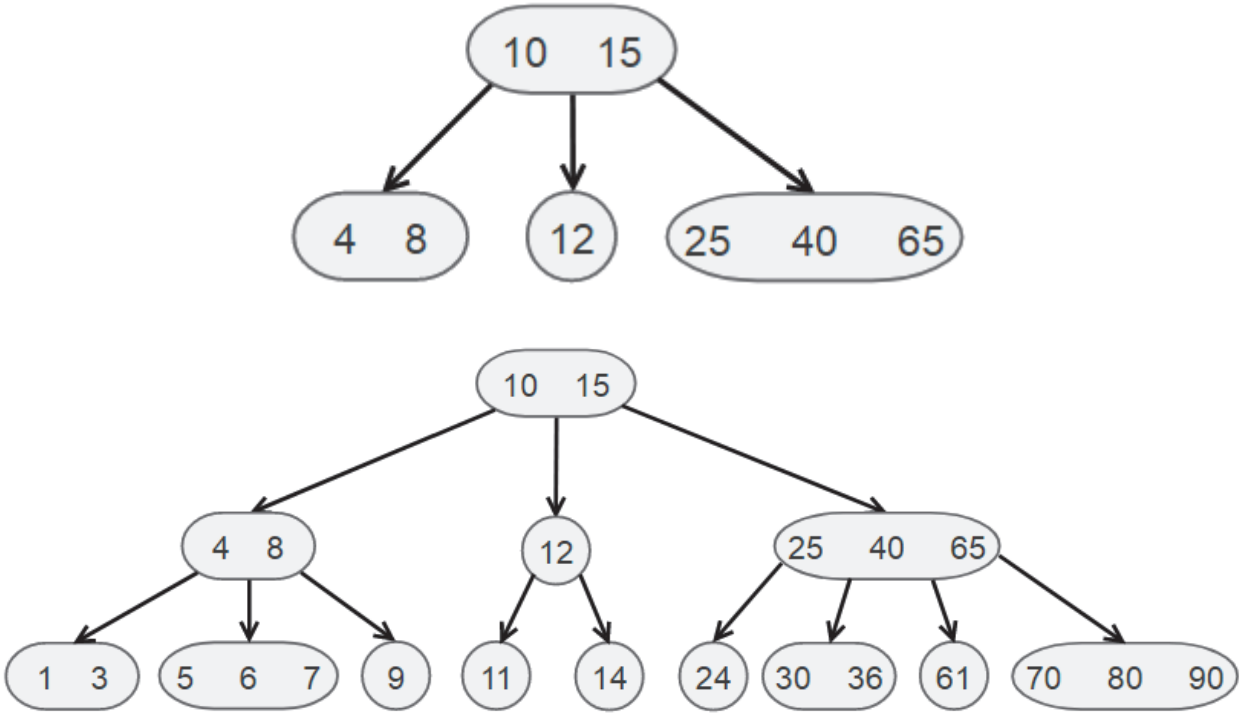
2-3-4 дерево (2-3-4 tree) – дерево поиска, каждый узел которого может иметь до 3-х ключей (значений) и до 4-х потомков.

2-3-4 дерево (2-3-4 tree) – B-дерево, каждый узел которого имеет:

- либо 1 значение (1 поле данных) и 2 потомка (или 0 потомков),
- либо 2 значения (2 поля данных) и 3 потомка (или 0 потомков),
- либо 3 значения (3 поля данных) и 4 потомка (или 0 потомков).

2-3-4 дерево – это B-дерево порядка 4.

Примеры 2-3-4 деревьев:



2-3 дерево можно представить, как левостороннее или правостороннее КЧ-дерево:

- В **левостороннем КЧ-дереве** красный узел может быть только левым потомком черного узла.
- В **правостороннем КЧ-дереве** красный узел может быть только правым потомком черного узла.

2-3-4 дерево можно представить, как обычное КЧ-дерево. При этом каждый узел с 2 значениями и с 3 значениями будет представлен в КЧ-дерева как 2 узла и 3 узла соответственно.

Красными узлами в таком КЧ-дереве будут:

- Для узла с 3 значениями: 1-е и 3-е значения (тогда 2-е значение будет является их предком).

- Для узла с 2 значениями:
  - 1-е значение (тогда 2-е значение будет являться его предком) или
  - 2-е значение (тогда 1-е значение будет являться его предком).

Пример формирования 2-3-4-дерева

Поэтапно построим 2-3-4-дерево для того же массива значений, который использовался при построении 2-3 дерева: 10, 15, 9, 11, 16, 17, 18.

1) Создание 2-3-4 дерева с корнем 10.

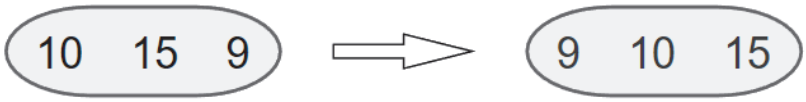
При формировании 2-3-4 дерева первый и единственный узел становится корнем:

2) Добавление значения 15.

Новое значение добавляется в корень и происходит сортировка двух значений:

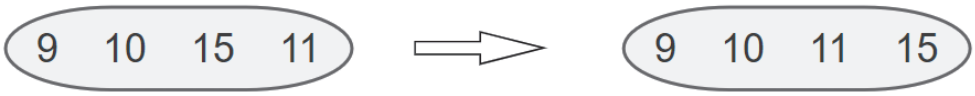
3) Добавление значения 9.

Новое значение добавляется в корень и происходит сортировка трех значений:

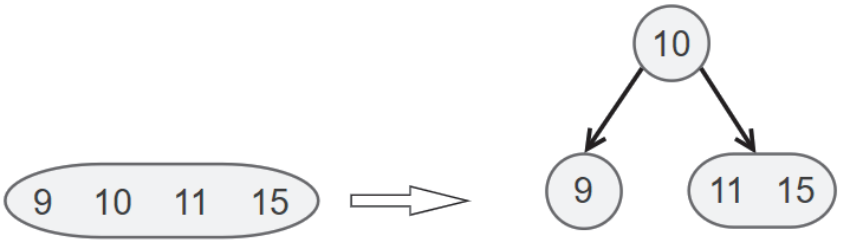


4) Добавление значения 11.

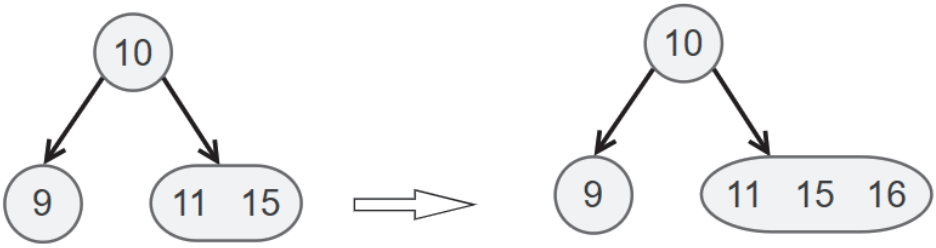
Новое значение добавляется в корень и происходит сортировка четырех значений:



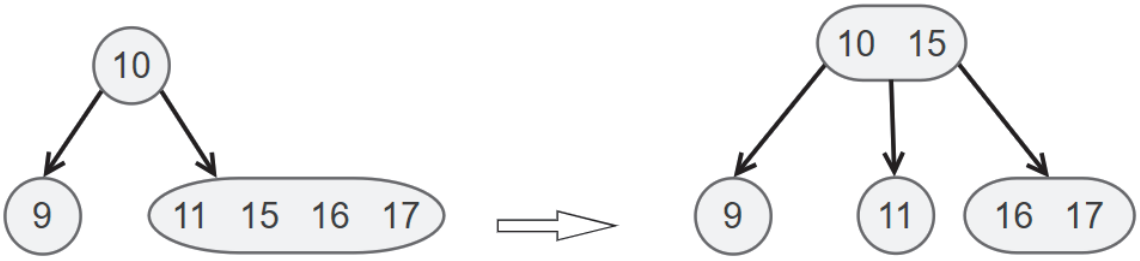
Однако узел может содержать только 1, 2 или 3 значения, поэтому необходимо провести **балансировку**.



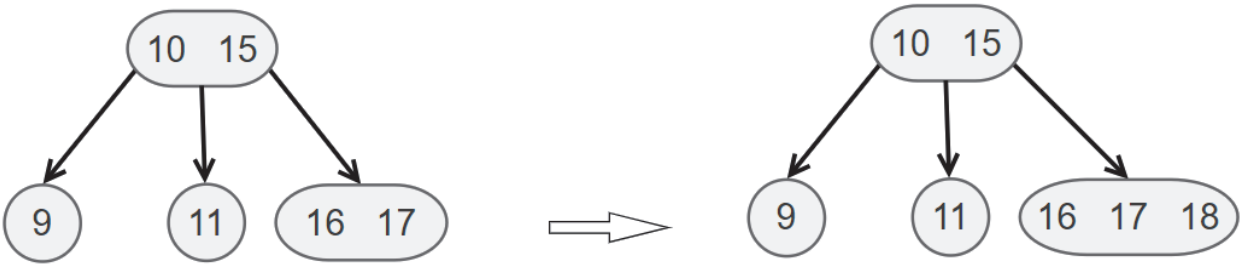
5) Добавление значения 16.



6) Добавление значения 17.



7) Добавление значения 18.



Переход от 2-3-4 дерева к КЧ-дереву

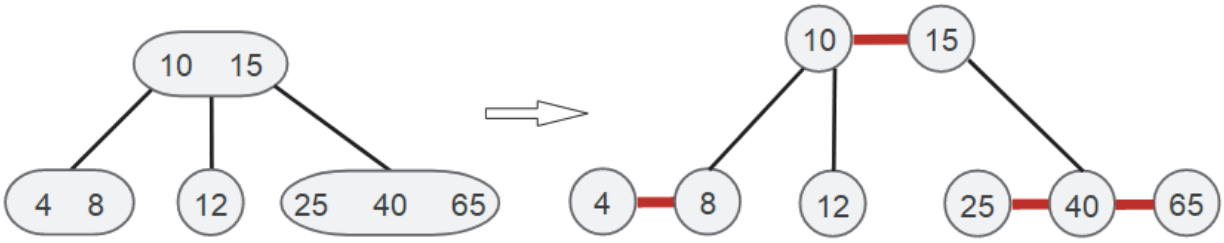
Пример:

Для простоты построим ненаправленное 2-3-4 дерево:

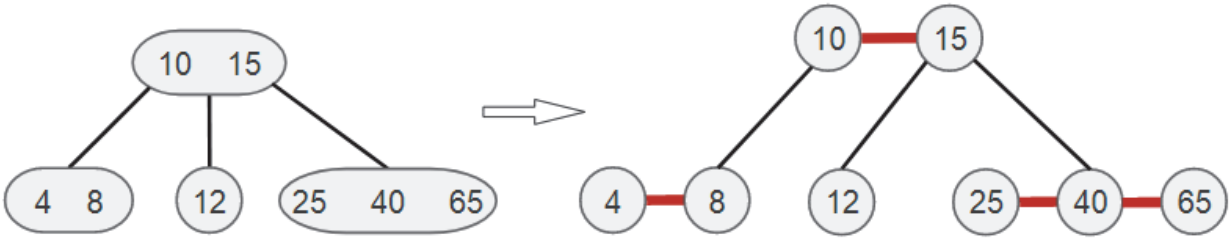
Разобьем узлы с 2 и с 3 значениями на отдельные узлы и свяжем их красной линией для наглядности. Также для простоты понимания мы можем изобразить связь между этими узлами горизонтально.

Представление данного 2-3-4 дерева в виде дерева с узлами, имеющими лишь 1 значение можно реализовать в 2 вариантах.

Вариант 1 – узел 12 может быть потомком узла 10:

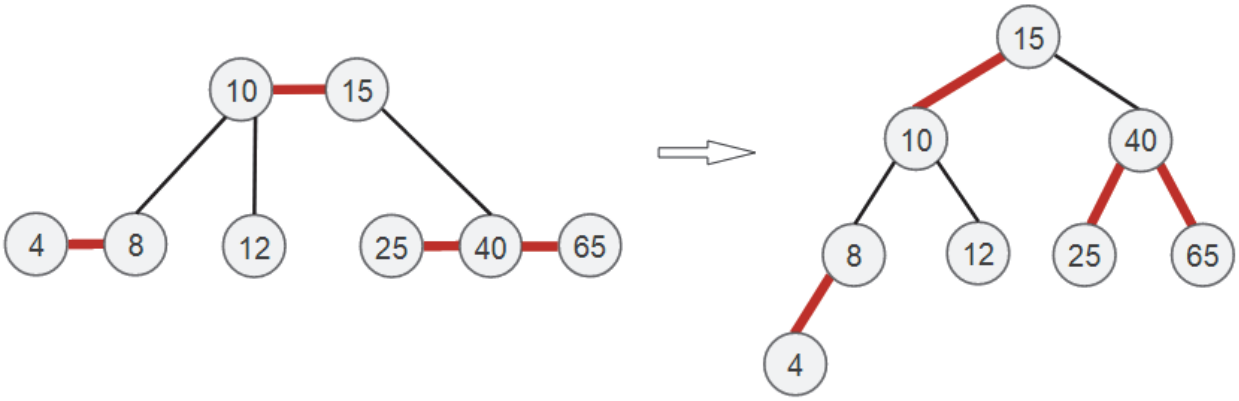


Вариант 2 – узел 12 может быть потомком узла 15:

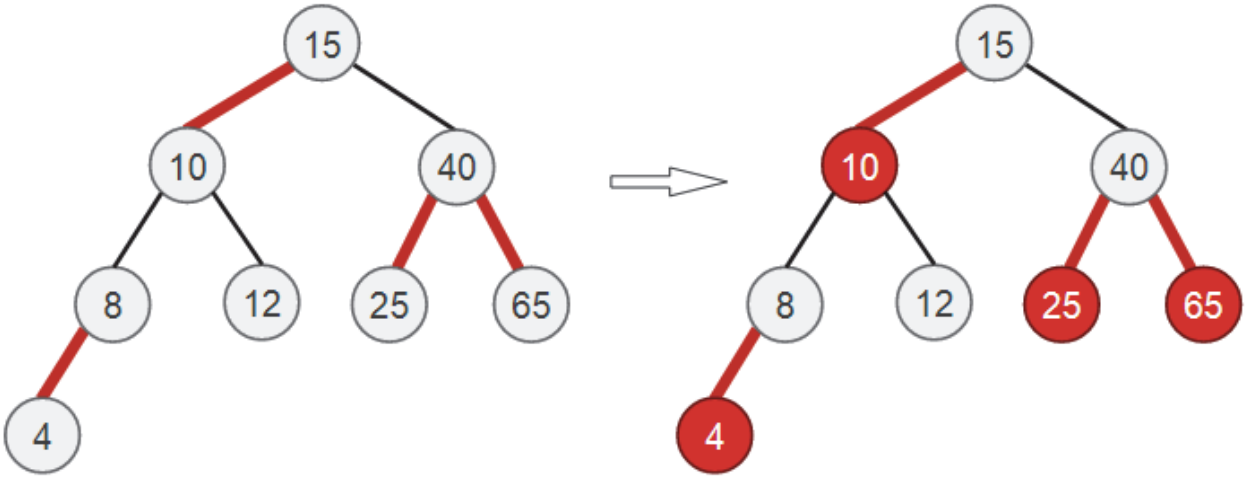


Остановимся на варианте 1.

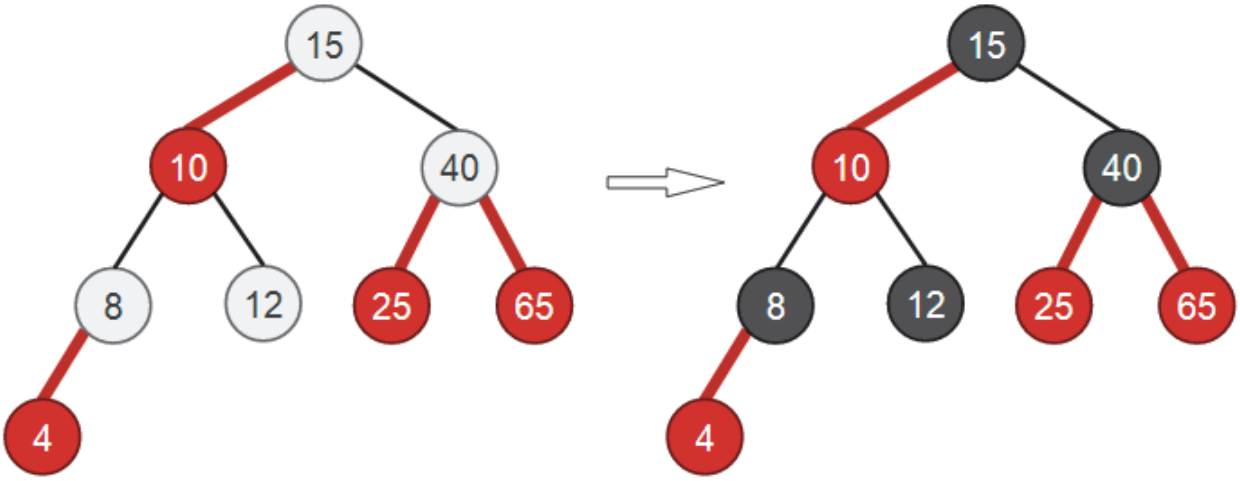
Далее представим горизонтальные связи в стандартном для дерева поиска виде:



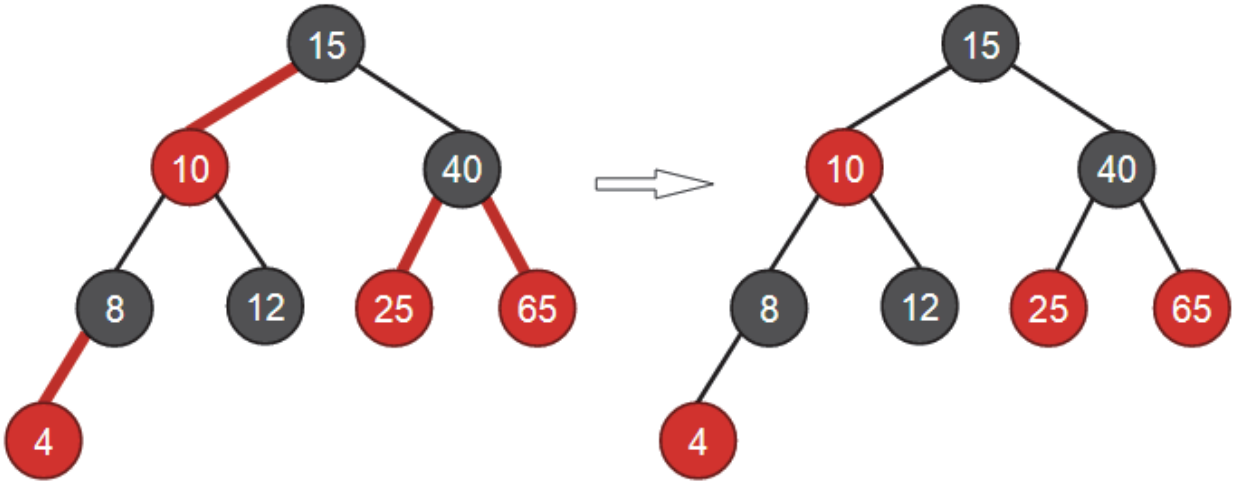
Рассмотрим только узлы с красной связью и раскрасим в красный цвет только те из них, которые являются потомками:



Остальные узлы раскрасим в черный цвет:



Представим красные связи стандартным образом:



Таким образом, из 2-3-4 дерева мы получили КЧ-дерево.

Свойства КЧ-дерева

1. Каждый узел имеет цвет: красный или черный.
2. Корень дерева – черный.
3. Каждый фиктивный листовой узел – черный.
4. Потомки красного узла – черные. При этом потомками черных узлов могут быть как черные, так и красные узлы.
5. Все пути, идущие от корня к любому фиктивному листу, содержат одинаковое количество черных узлов.

Формирование КЧ-дерева

Существуют правила:

1. Добавляемый в дерево узел окрашивается в красный цвет. Если это первый узел, то он перекрашивается в черный цвет и становится корнем.

После добавления каждого узла происходит проверка выполнимости свойств КЧ-дерева. Если они нарушаются, необходимо проводить балансировку.

Вставка узла в КЧ-дерево

Случаи при вставке узла X в КЧ-дерево:

1. Узел X – единственный

Действия: узел становится корнем и перекрашивается в черный цвет.

2. Предок узла X – красный

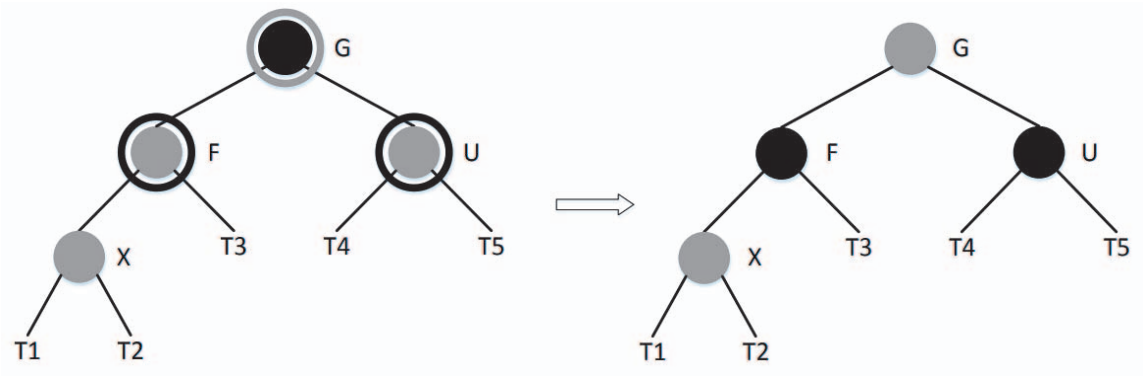
(нарушение свойства 4. Потомки красного узла – черные, называемое также «красно-красным нарушением»)

Здесь выделяется 3 дополнительный случая (условие, что предок узла X – красный выполняется).

Обозначим добавленный узел за X, его предка (отца) за F, его деда за G, а второго сына деда – дядю – за U. Поддеревья обозначим буквами T_i.

2.1. Дядя U добавляемого узла X красный

Действия: перекраска F и U в черный цвет, а G в красный (свап цвета).



Источник: [11].

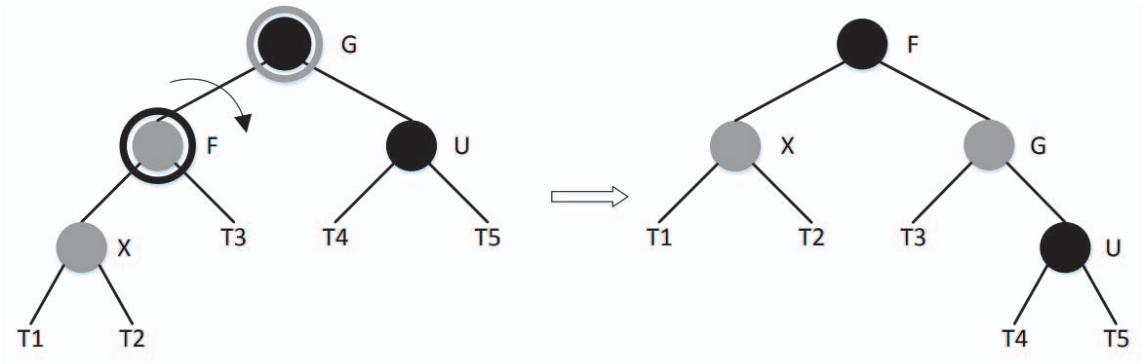
Если G – корень, то он просто перекрашивается в черный цвет.

При этом все 5 свойств КЧ-деревьев оказываются выполненными.

Если отец узла G тоже красный, то появится новое красно-красное нарушение, и понадобится дальнейшее восстановление свойств КЧ-дерева, только в роли узла X теперь будет выступать узел G [11].

2.2. Дядя U добавляемого узла X черный и цепочка узлов X-F-G образует прямую линию

Действия: перекраска F и G и одинарный поворот.

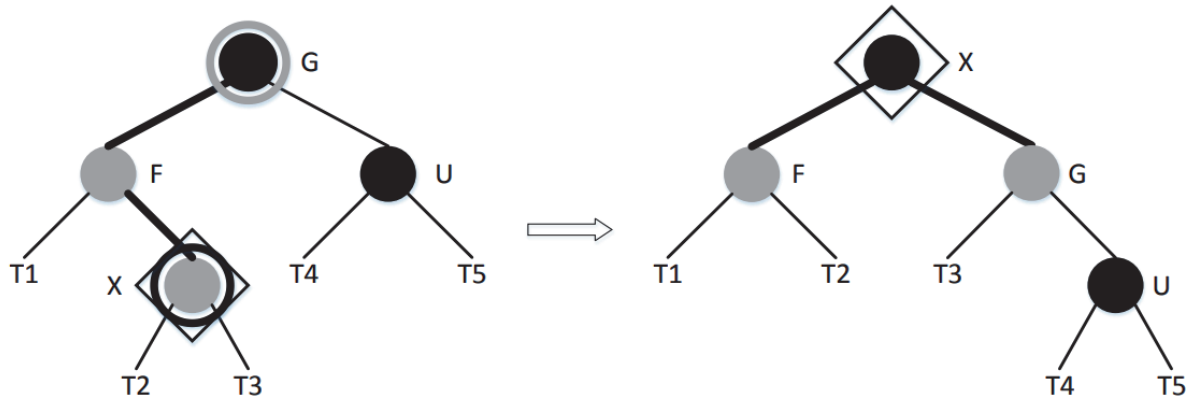


Источник: [11].

Только перекраски отца F узла X в черный цвет, а деда G в красный будет недостаточно для восстановления свойств КЧ-дерева, потому что количество черных узлов в путях, проходящих через G направо, сократится на 1. Поэтому потребуется одинарный поворот деда (G) относительно отца, в данном случае направо. Тогда количество черных узлов в путях, идущих от F, который теперь стал корнем поддерева, налево и направо будет равно первоначальному количеству [11].

2.3. Дядя U добавляемого узла X черный и цепочка узлов X-F-G образует угол

Действия: перекраска X и G и двойной поворот.



Источник: [11].

Перекрасив узел X в черный цвет, а его деда G – в красный, получим новое красно-красное нарушение F-G между отцом и дедом X. Ситуацию разрешает двойной поворот комбинации X-F-G, когда нижний узел (X) оказывается наверху комбинации.

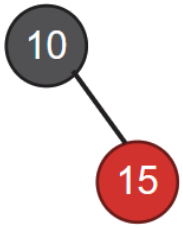
В результате изменилось только количество красных узлов – слева уменьшилось на 1, а справа увеличилось на 1. Однако количество красных узлов в путях ни на что не влияет, все свойства полученного КЧ-дерева выполняются [11].

Пример формирования КЧ-дерева

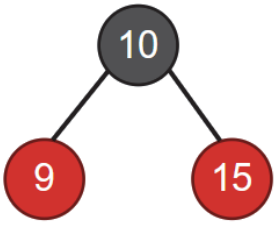
Поэтапно построим КЧ-дерево для того же массива значений, который использовался при построении 2-3-4 дерева: 10, 15, 9, 11, 16, 17, 18.

1) Добавим значение 10. Так как это первый узел, он становится корнем и окрашивается в черный цвет:

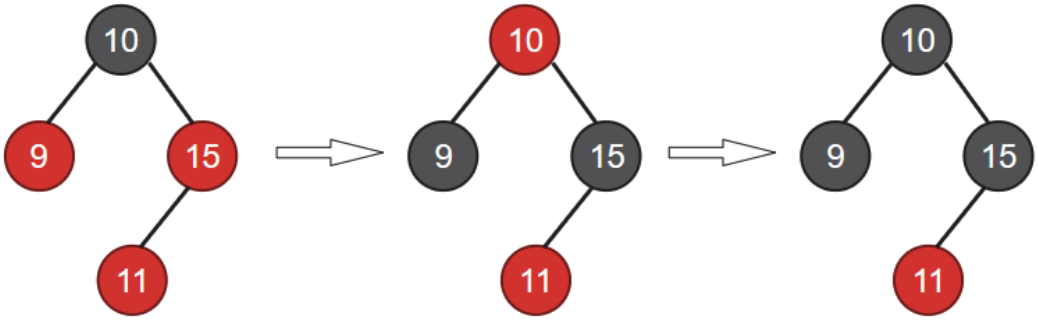
2) Добавим значение 15. Добавленный узел окрашивается в красный цвет:



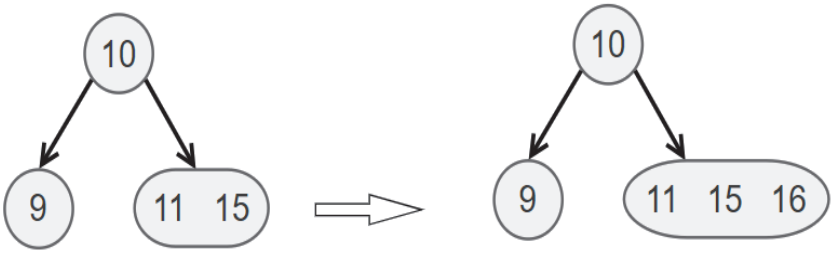
3) Добавим значение 9. Добавленный узел окрашивается в красный цвет:



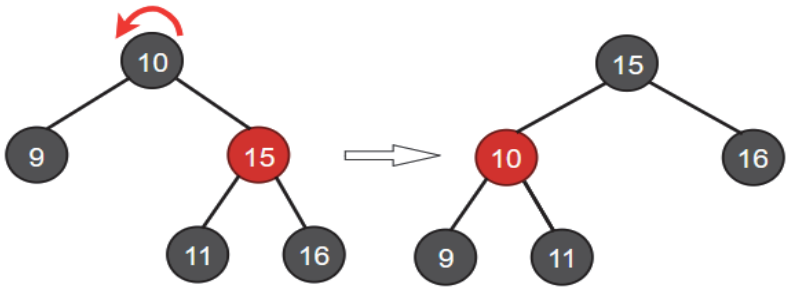
4) Добавим значение 11. Правила ЛКЧ-дерева не нарушились:



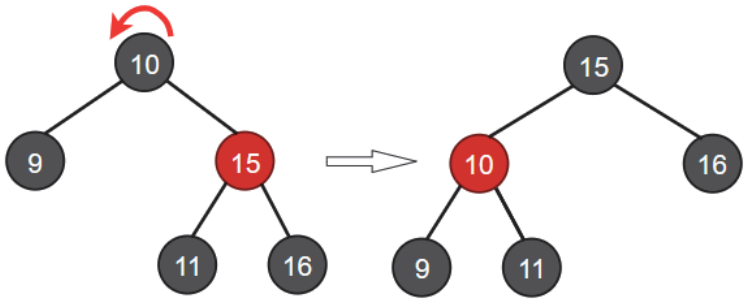
Если сравнить такое ЛКЧ-дерево с полученным ранее 2-3 деревом, то мы увидим, что красный узел 11 и его предок 15 соответствуют узлу 11-15 в 2-3 дереве:



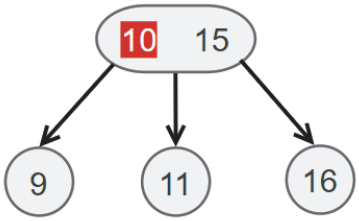
5) Добавим значение 16. Применим **перекраску**:



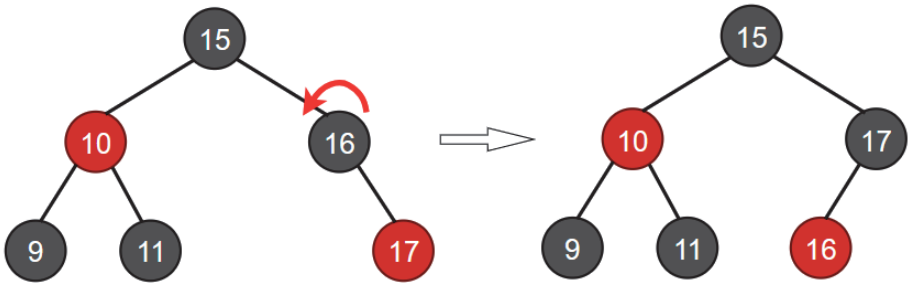
Применим **левый поворот**:



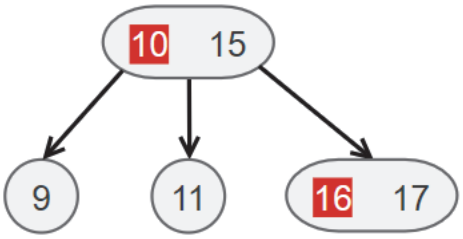
ЛКЧ-дерево соответствует полученному ранее 2-3 дереву:



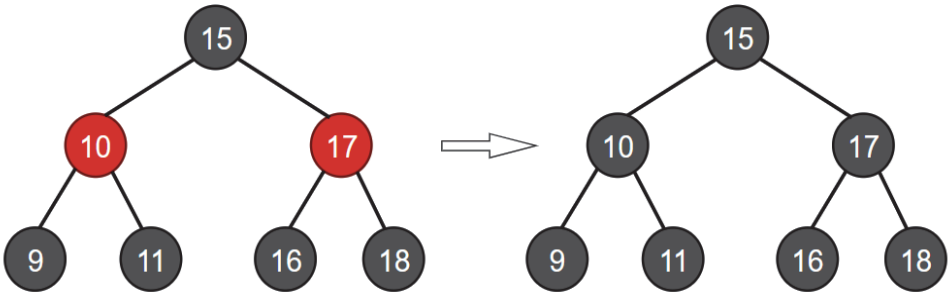
6) Добавим значение 17. Применим **левый поворот**:



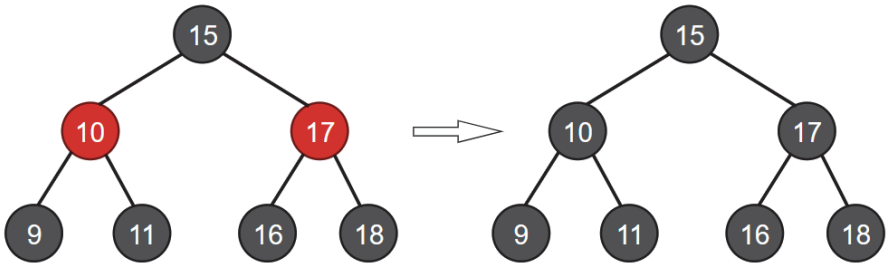
ЛКЧ-дерево соответствует полученному ранее 2-3 дереву:



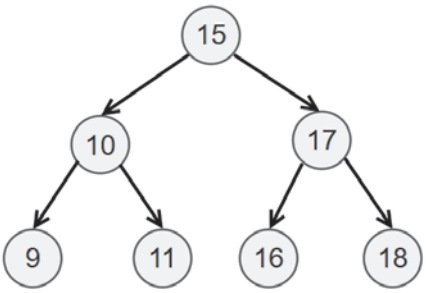
7) Добавим значение 18. Применим **перекраску**:



Применим второй раз **перекраску**:



В нашем случае итоговое ЛКЧ-дерево приобрело вид обычного бинарного дерева поиска, в котором все узлы имеют один и тот же цвет. Это соответствует итоговому 2-3 дереву, полученному ранее, в котором не было узлов с двумя значениями:



Задачи

Исходные данные к задачам см. в разделе «Исходные данные к задачам по вариантам» Практического задания № 16.

Задача № 1

Для наборов чисел (Добавить: …) построить:

1. 2-3-4 дерево.
2. КЧ-дерево.

Необходимо поочередно добавлять элементы в дерево в том порядке, в котором они даны. При необходимости выполнять балансировку дерева.

Дополнительно:

Задача № 2

Реализовать КЧ-дерево путем создания класса (классов) в Python. При этом в виде методов класса реализовать следующие операции:

- *Поиск узла по значению.*
- *Вставка узла (при этом предусмотреть балансировку КЧ-дерева).*
- *Определение черной высоты дерева.*
- *Отображение всего дерева в консоли.*

Практическое задание № 20
«Граф»

1. Граф

Граф – абстрактный математический объект, представляющий собой множество вершин (vertices) и ребер (edges), то есть соединений между парами вершин.

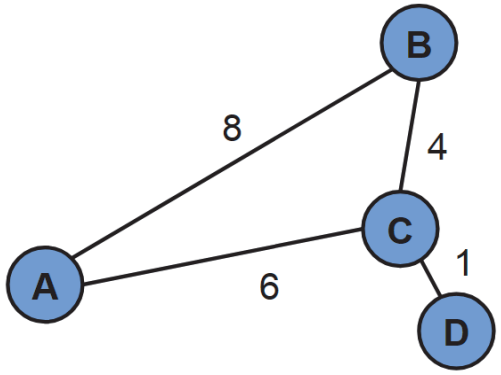
В программе граф можно представить связью (откуда, куда, длина пути) или (куда, длина пути).

Реализация графа на Python

Реализовать граф на Python можно несколькими способами. Один из них – создание класса.

Другой способ связан с использованием словаря, где ключами будут вершины, а в качестве значений будут выступать списки пар (вершина, длина пути).

Пример: реализуем следующий граф в виде словаря.



Вершина A связана с вершиной B (длина 8) и вершиной C (длина 6). Тогда в словаре ключу A будет соответствовать список из двух пар: (B, 8) и (C, 6) (листинг 20.1).

Листинг 20.1 – Пример создания графа на основе словаря

```
graph1 = {'A' : [('B', 8), ('C', 6)],
          'B' : [('A', 8), ('C', 4)],
          'C' : [('A', 6), ('B', 4), ('D', 1)],
          'D' : [('C', 1)]}

print('Граф:', graph1)
print('Вершина A:', graph1['A'])
```

```
Граф: {'A': [('B', 8), ('C', 6)], 'B': [('A', 8), ('C', 4)],
'C': [('A', 6), ('B', 4), ('D', 1)], 'D': [('C', 1)]}
Вершина A: [('B', 8), ('C', 6)]
```

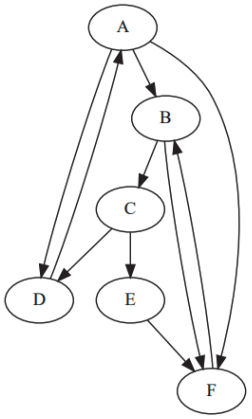
Визуализировать граф можно с помощью библиотеки `graphviz`. Для ее установки необходимо выполнить следующие команды:

```
conda install graphviz
pip install graphviz
```

Пример визуализации графа представлен в листинге 20.2.

Листинг 20.2 – Пример визуализации графа с помощью библиотеки `graphviz`

```
graph = {
    "A": ["B", "D", "F"],
    "B": ["C", "F"],
    "C": ["D", "E"],
    "D": "A",
    "E": "F",
    "F": "B",
}


from graphviz import Digraph

dot = Digraph()
for k in graph.keys():
    dot.node(k, k)
edges = []
for k, v in graph.items():
    edges += [f"{k}{to}" for to in v]
dot.edges(edges)
dot.render(view=True)
```



2. Обход графа (поиск в графе)

К классическим алгоритмам на графах относятся **алгоритмы обхода графов**.

Обход графа – процесс последовательного прохода по смежным вершинам графа.

Обход графа – перечисление вершин графа [3].

Существуют две основные стратегии обхода графа:

- **Обход в ширину**.
- **Обход в глубину**.

2.1. Обход в ширину

Обход в ширину подразумевает следующий рекурсивный алгоритм:

1. Начиная со стартовой вершины, посетить все смежные с ней вершины.

2. Для каждой смежной вершины повторить шаг 1, заходя только в непосещенные вершины.

Алгоритм заканчивается, если:

- посещены все вершины,

ИЛИ

- нет непосещенных вершин, смежных с уже посещенными [3].

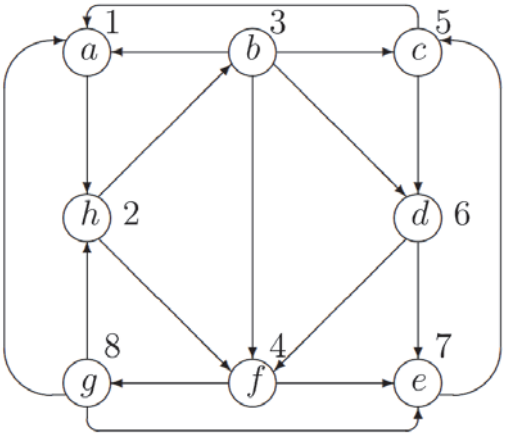


Рисунок 20.1 – Пример обхода в ширину. Порядок обхода имеет вид: *ahbfcdeg*

Источник: [3].

2.2. Обход в глубину

Обход в глубину подразумевает следующий рекурсивный алгоритм:

1. Начиная со стартовой вершины, посетить смежную с ней вершину.

2. Из этой смежной вершины посетить смежную с ней непосещенную вершину, и повторить шаг 2. При этом:

- Если нет непосещенных вершин, смежных с текущей, вернуться на уровень назад и повторить шаг 2.

Алгоритм заканчивается, если:

- посещены все вершины,

ИЛИ

- произошло возвращение в стартовую вершину и не осталось смежных с ней непосещенных вершин [3].

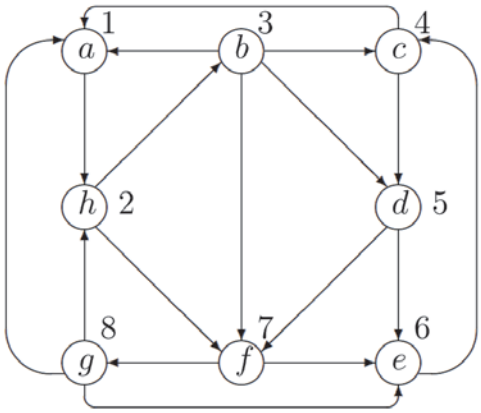


Рисунок 20.2 – Пример обхода в глубину. Порядок обхода имеет вид: *ahbcdefg*

Источник: [3].

Алгоритм Дейкстры – алгоритм на графах, находящий кратчайшее расстояние от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

Эйлеров путь (**эйлерова цепь**) в графе – это путь, проходящий по всем рёбрам графа и притом только по одному разу.

Эйлеров цикл – эйлеров путь, являющийся циклом, то есть замкнутый путь, проходящий через каждое **ребро** графа ровно по одному разу.

Гамильтонов цикл – цикл (замкнутый путь), который проходит через каждую **вершину** данного графа ровно по одному разу; то есть простой цикл, в который входят все вершины графа.

Задачи

Задача № 1. Реализуйте граф (например, с помощью словаря).
Задача № 2. Реализуйте поиск в ширину (обход графа в ширину).
Задача № 3. Реализуйте поиск в глубину (обход графа в глубину).
Задача № 4. Реализуйте алгоритм Дейкстры.
Задача № 5. Реализуйте поиск эйлерова пути (цикла) в графе.
Задача № 6. Реализуйте поиск гамильтонова цикла в графе.

Контрольные вопросы к разделу «Графы и деревья»

1. Дайте определение понятию «граф».
2. Дайте определение понятию «дерево».
3. Дайте определение понятию «бинарное дерево».
4. Дайте определение понятию «бинарное дерево поиска».
5. Каким образом происходит добавление узла в бинарное дерево поиска?
6. Каким образом происходит удаление узла из бинарного дерева поиска?
7. Перечислите виды сбалансированных деревьев поиска.
8. Дайте определение понятию «идеально сбалансированное бинарное дерево поиска».
9. Дайте определение понятию «АВЛ-дерево».
10. Каким образом происходит добавление узла в АВЛ-дерево?
11. Каким образом происходит удаление узла из АВЛ-дерева?
12. Дайте определение понятию «2-3 дерево».
13. Каким образом происходит добавление узла в 2-3 дерево?
14. Каким образом происходит удаление узла из 2-3 дерева?
15. Дайте определение понятию «Левостороннее красно-черное дерево».
16. Каким образом происходит переход от 2-3 дерева к Левостороннему красно-черному дереву?
17. Каким образом происходит добавление узла в Левостороннее красно-черное дерево?
18. Каким образом происходит удаление узла из Левостороннего красно-черного дерева?
19. Дайте определение понятию «2-3-4 дерево».
20. Каким образом происходит добавление узла в 2-3-4 дерево?
21. Каким образом происходит удаление узла из 2-3-4 дерева?
22. Дайте определение понятию «Красно-черное дерево».
23. Каким образом происходит переход от 2-3-4 дерева к Красно-черному дереву?
24. Каким образом происходит добавление узла в Красно-черное дерево?
25. Каким образом происходит удаление узла из Красно-черного дерева?
26. Каким образом происходит обход графа в ширину?
27. Каким образом происходит обход графа в глубину?
28. Что из себя представляет алгоритм Дейкстра?
29. Что из себя представляет эйлеров цикл?
30. Что из себя представляет гамильтонов цикл?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Абрамян М.Э. Programming Taskbook: Электронный задачник по программированию, 2005. – 139 с.
2. Брукшир Дж. Г. Компьютерные науки. Базовый курс, 13-е изд.: Пер. с англ. // Брукшир Дж. Г., Брилов Д. – СПб.: ООО «Диалектика», 2019. – 992 с.
3. Буркатовская Ю.Б. Теория графов. Часть 1: учебное пособие / Ю.Б.Буркатовская; Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2014. – 200 с.
4. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб.: Питер, 2017. – 288 с.
5. Лутц М. Изучаем Python, том 1, 5-е изд.: Пер. с англ. – СПб.: ООО «Диалектика», 2019. – 832 с.
6. Лутц М. Изучаем Python, том 2, 5-е изд.: Пер. с англ. – СПб.: ООО «Диалектика», 2020. – 720 с.
7. Петров Ю. Программирование на языке высокого уровня (Python) // yuripetrov.ru/edu. – сайт о программировании. – URL: https://www.yuripetrov.ru/edu/python/index.html. – Дата публикации: 22.10.2024
8. Попов Е. Объектно-ориентированное программирование // metanit.com. – сайт о программировании. – URL: https://metanit.com/python/tutorial/7.1.php. – Дата публикации: 03.02.2024
9. Попов Е. Упаковка и распаковка // metanit.com. – сайт о программировании. – URL: https://metanit.com/python/tutorial/3.7.php. – Дата публикации: 01.02.2023
10. Семёнов А. Л. Алгоритм // Большая российская энциклопедия: научно-образовательный портал – URL: https://bigenc.ru/c/algoritm-ee5b00/?v=4523149. – Дата публикации: 10.08.2022
11. Сенюкова О.В. Сбалансированные деревья поиска: Учебно-методическое пособие. – М.: Издательский отдел факультета ВМиК МГУ имени М.В. Ломоносова; МАКС Пресс, 2014. – 68 с.
12. Big-O Cheat Sheet // bigocheatsheet.com – сводная информация про сложность алгоритмов. – URL: https://www.bigocheatsheet.com/
13. Built-in Functions // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/library/functions.html#
14. Built-in Types // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex
15. Classes // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/tutorial/classes.html

16. Data model // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/reference/datamodel.html
17. Data Structures // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/tutorial/datastructures.html#
18. Downing D. Dictionary of Computer and Internet terms / Douglas A. Downing, Michael A. Covington, Melody Mauldin Covington. – 10th ed. – Barron's Educational Series, Inc., 2009. – 554 pp.
19. Glossary // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/glossary.html
20. Lexical analysis // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/reference/lexical_analysis.html
21. math – Mathematical functions // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/library/math.html
22. PEP 8 – Style Guide for Python Code // peps.python.org – Python Enhancement Proposals – Предложения по улучшению Python (документация и руководства по использованию). URL: https://peps.python.org/pep-0008/
23. Ramalho L. Fluent Python. – O'Reilly Media, Inc., Sebastopol, 2022. – 980 pp.
24. Sedgewick R. Algorithms / R. Sedgewick, K. Wayne, 2011. – 976 pp.
25. Sweigart A. The Recursive Book of Recursion. – San Francisco: No Starch Press, 2022. – 303 pp.
26. The Python Standard Library // docs.python.org – документация к языку программирования Python. – URL: https://docs.python.org/3/library/index.html

Учебное издание

Демичев Вадим Владимирович
Быков Денис Витальевич
Храмов Дмитрий Эдуардович
Ульянкин Александр Евгеньевич
Титов Артем Денисович

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Учебное пособие

Сдано в набор 03.12.2024. Подп. в печ. 10.12.2024.
Формат 60×88/16. Бумага офсетная.
Усл.печ.л. 15,5 Тираж 500 экз.

Издательство «Научный консультант» предлагает авторам:
издание рецензируемых сборников трудов научных
конференций; печать монографий, методической и иной литературы

ISBN 978-5-907933-15-6

Издательство Научный консультант
123007, г. Москва, Хорошевское ш., 35к2, офис 508.
Тел.: +7 (926) 609-32-93, +7 (499) 195-60-77 www.n-ko.ru keyneslab@gmail.com