ТЕОРЕТИЧЕСКИЕ ВОПРОСЫ ДЛЯ ПОДГОТОВКИ К ЭКЗАМЕНУ ПО АИСД (ВТОРОЙ СЕМЕСТР)

Готовил Литяев Матвей ПМ24-6 для “Домашки ПМ-ки” :)

1. Концепция класса и объекта. Принципы и механизмы ООП.

1. Класс и объект

- **Класс** — это абстрактный шаблон, описывающий структуру и поведение будущих объектов. Он определяет:
  - **атрибуты** (данные, характеризующие состояние);
  - **методы** (действия, которые может выполнять объект).
- **Объект (экземпляр класса)** — конкретная сущность, созданная на основе класса, обладающая уникальным состоянием (значениями атрибутов).

2. Основные принципы ООП

1. **Инкапсуляция**
   a. Принцип, объединяющий данные и методы работы с ними в единую структуру (класс), и ограничивающий прямой доступ к внутреннему состоянию объекта.
   b. **Цель**: защита данных от некорректного использования, упрощение взаимодействия через строго определённые интерфейсы (например, геттеры/сеттеры).
2. **Наследование**
   a. Механизм создания нового класса на основе существующего (родительского) с возможностью:
      i. Заимствования атрибутов и методов родителя;
      ii. Расширения или изменения унаследованного поведения.
   b. **Цель**: повторное использование кода и построение иерархий.
3. **Полиморфизм**
   a. Способность объектов с одинаковым интерфейсом (например, методами одного имени) выполнять разные действия в зависимости от их типа.
   b. **Виды**:
      i. **Ad-hoc (перегрузка операторов)** — разные действия для одного оператора (+ для чисел и строк);
      ii. **Параметрический (истинный)** — один интерфейс для разных типов (метод `draw()` для круга и квадрата).
4. **Абстракция**
   a. Выделение ключевых характеристик объекта, игнорируя несущественные детали.
   b. **Реализация**: через абстрактные классы (задают шаблон без реализации) и интерфейсы.

2. Объявление класса, конструктор, создание объектов и одиночное наследование в Python. Управление доступом к атрибутам класса в Python.

Объявление класса

Класс — это шаблон для создания объектов, который объявляется с помощью ключевого слова `class`.

Конструктор (`__init__`)

- **Конструктор** — специальный метод `__init__()`, который автоматически вызывается при создании объекта.
- Используется для инициализации атрибутов.
- Первый параметр — `self` (ссылка на экземпляр класса).

Одиночное наследование

- **Наследование** позволяет создать дочерний класс на основе родительского.
- Дочерний класс наследует все атрибуты и методы родителя.

Управление доступом к атрибутам

Python использует соглашения для условной инкапсуляции:

- **Публичные атрибуты** — доступны везде (например, `self.title`).
- **Защищённые атрибуты** — одно подчёркивание (`_protected`), сигнализируют, что атрибут не следует использовать вне класса.
- **Приватные атрибуты** — два подчёркивания (`__private`), Python искажает имя (`_ИмяКласса__private`).

```
class BankAccount:

        def init(self, balance):

                self.__balance = balance # Приватный атрибут

        def get_balance(self):  # Геттер
                return self.__balance
```

3. Полиморфизм и утиная типизация, проверка принадлежности объекта к классу в языке Python.

Полиморфизм в Python

Полиморфизм — это возможность использовать объекты разных классов через единый интерфейс. В Python реализуется двумя способами:

1. **Ad-hoc полиморфизм (перегрузка операторов)**
   a. Разные объекты могут по-разному реагировать на одни и те же операции.
   b. Пример: оператор + работает по-разному для чисел (`3 + 5` → `8`) и строк (`"a" + "b"` → `"ab"`).
2. **Параметрический полиморфизм (истинный полиморфизм)**
   a. Объекты разных классов могут иметь методы с одинаковыми именами, но разной реализацией.
   b. Пример: метод `draw()` для классов `Circle` и `Square` рисует разные фигуры.

Пример (без кода):
Если классы `Cat` и `Dog` имеют метод `speak()`, то при вызове `animal.speak()` для объектов этих классов будет выполнена соответствующая реализация.

Утиная типизация (Duck Typing)

- **Принцип**: *"Если что-то выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, утка"*.
- В Python тип объекта определяется не его классом, а **поведением** (наличием нужных методов и атрибутов).
- Проверка типов выполняется во время выполнения программы.

Пример (без кода):
Функция, принимающая объект с методом `quack()`, будет работать с любым объектом, у которого есть этот метод (даже если он не принадлежит классу `Duck`).

3. Проверка принадлежности объекта к классу

Python предоставляет несколько способов проверки:

`isinstance(obj, Class)`

Проверяет, является ли объект `obj` экземпляром класса `Class` или его подкласса.

`issubclass(Child, Parent)`

Проверяет, является ли класс `Child` подклассом `Parent`.

`type(obj) is Class`

Проверяет точное совпадение типа (игнорирует наследование).

4. Методы классов и статические переменные и методы в Python. Специальные методы для использования пользовательских классов со стандартными операторами и функциями.

Методы классов и статические переменные

Статические переменные (атрибуты класса) — общие для всех экземпляров.
Методы класса (`@classmethod`) — работают с классом, а не экземпляром.
Статические методы (`@staticmethod`) — не зависят ни от класса, ни от экземпляра.

```python
class Car:
    total_cars = 0  # Статическая переменная (атрибут класса)

    def __init__(self, brand):
        self.brand = brand
        Car.total_cars += 1  # Увеличиваем счетчик при создании экземпляра

    @classmethod
    def get_total_cars(cls):  # cls - ссылка на класс, а не на экземпляр
        return cls.total_cars

    @staticmethod
    def info():  # Не принимает ни self, ни cls
        return "Это класс Car"

# Использование
car1 = Car("Toyota")
car2 = Car("BMW")

print(Car.get_total_cars())  # 2 (вызов через класс)
print(car1.get_total_cars()) # 2 (можно вызвать и через экземпляр)
print(Car.info())            # "Это класс Car"
```

Специальные методы (магические методы)

Позволяют переопределить поведение объектов при использовании стандартных операторов (+, ==, `len()` и др.).

Основные магические методы:

Метод	Описание	Пример использования
`__init__`	Конструктор	`obj = Class()`
`__str__`	Строковое представление	`print(obj)`
`__len__`	Длина объекта	`len(obj)`
`__add__`	Сложение (+)	`obj1 + obj2`
`__eq__`	Сравнение (==)	`obj1 == obj2`
`__getitem__`	Доступ по индексу	`obj[0]`

5. Основные возможности, поддерживаемые функциональными языками программирования. Поддержка элементов функционального программирования в Python.

Основные концепции ФП

1. **Чистые функции** — не изменяют состояние, результат зависит только от входных данных.
2. **Функции высшего порядка** — принимают/возвращают другие функции.
3. **Неизменяемость** — данные не меняются после создания.
4. **Рекурсия** — заменяет циклы.
5. **Ленивые вычисления** — обработка данных по требованию.

Поддержка в Python

Чистые функции

def add(a, b): return a + b # Без побочных эффектов

Функции высшего порядка

def apply(func, x): return func(x)

apply(lambda x: x*2, 5) # 10

Неизменяемые типы

`str`, `tuple`, `frozenset` — нельзя изменить после создания.

Рекурсия

def fact(n): return 1 if n == 0 else n * fact(n-1)

Ленивые вычисления (с помощью yield, например)

```python
def square_numbers(numbers):
    for num in numbers:
        print(f"Вычисляю квадрат числа {num}")  # Демонстрация "ленивости"
        yield num ** 2

# Создаем генератор (пока ничего не вычисляется!)
squares = square_numbers([1, 2, 3, 4, 5])

print("Генератор создан, но вычислений еще не было\n")

# Запрашиваем значения по одному
print("Первое число:", next(squares))  # Вычисляет только первый квадрат
print("Второе число:", next(squares))  # Вычисляет только второй квадрат

# Можно продолжить позже
print("\n...прошло время...\n")
print("Третье число:", next(squares))
```

6. Концепция «функции — граждане первого класса» в языке программирования, поддержка этой концепции в Python. Специфика лямбда-функций в Python их возможности и ограничения. Типичные сценарии пользования лямбда-функций в Python.

Функции — граждане первого класса в Python

Концепция означает, что функции в Python являются объектами, которые можно:

1. **Присваивать** переменным
2. **Передавать** как аргументы других функций
3. **Возвращать** из других функций
4. **Хранить** в структурах данных

```python
def greet(name):
    return f"Hello, {name}!"

# 1. Присваивание переменной
func = greet
print(func("Alice"))  # Hello, Alice!

# 2. Передача как аргумента
def call_func(f, x):
    return f(x)

print(call_func(greet, "Bob"))  # Hello, Bob!

# 3. Возврат из функции
def create_greeter(prefix):
    def greeter(name):
        return f"{prefix}, {name}!"
    return greeter

morning_greet = create_greeter("Good morning")
print(morning_greet("Kate"))  # Good morning, Kate!
```

Лямбда-функции в Python

Специфика:

- Анонимные функции, записанные в одну строку: `lambda args: expression`
- Могут содержать **только одно выражение** (нельзя использовать многострочные `if-else`, циклы, но можно тернарный оператор)
- Не поддерживают аннотации типов и многострочные операции

Ограничения:

1. Невозможно добавить docstring (описание функции)
2. Не могут содержать сложную логику
3. Усложняют чтение кода при злоупотреблении

Типичные сценарии использования:

Сортировка с ключом:

users = [{"name": "Alice", "age": 25}, {"name": "Bob", "age": 30}]

users.sort(key=lambda x: x["age"]) # Сортировка по возрасту

Обработка данных в `map()`/`filter()`:

numbers = [1, 2, 3] squared = list(map(lambda x: x**2, numbers)) # [1, 4, 9]

Тернарные операции:

is_even = lambda x: True if x % 2 == 0 else False

7. Глобальные и локальные переменные в функциях на примере Python. Побочные эффекты вызова функций и их последствия.

1. Локальные переменные

- **Определение**: Переменные, объявленные внутри функции.
- **Область видимости**: Только внутри функции.

```python
def my_func():
    local_var = 10  # Локальная переменная
    print(local_var)

my_func()  # Выведет: 10
print(local_var)  # Ошибка! Переменная не определена
```

2. Глобальные переменные

- **Определение**: Переменные, объявленные вне функций.
- **Область видимости**: Вся программа.
- **Как использовать внутри функции**:
  - Чтение: можно без объявления.
  - Запись: требуется ключевое слово `global`.

```python
global_var = 20  # Глобальная переменная

def my_func():
    print(global_var)  # Чтение — работает

def modify_global():
    global global_var  # Явное объявление
    global_var = 30    # Изменение значения

my_func()         # Выведет: 20
modify_global()
print(global_var)  # Выведет: 30
```

Что такое побочный эффект функции?

Любое изменение состояния программы, кроме возврата значения:

- Изменение глобальных переменных.
- Модификация переданных изменяемых объектов (списков, словарей).
- Ввод/вывод данных (например, запись в файл).

Последствия побочных эффектов

- **Неожиданные изменения**: Трудно отследить, где и как изменились данные.
- **Сложность тестирования**: Функции зависят от внешнего состояния.
- **Проблемы многопоточности**: Конкуренция за ресурсы.

8. Вложенные функции и замыкания, специфика реализации в Python.

1. Вложенные функции (Nested Functions)

Определение: Функции, объявленные внутри других функций.
Особенности:

- Видны только внутри родительской функции.
- Могут обращаться к переменным внешней функции (но не изменять их без `nonlocal`).

```python
def outer():
    x = 10

    def inner():  # Вложенная функция
        print(f"Внутренняя функция: x = {x}")

    inner()  # Вызов внутри outer()

outer()  # Выведет: "Внутренняя функция: x = 10"
# inner()  # Ошибка! Не видна снаружи.
```

2. Замыкания (Closures)

Определение: Вложенная функция, которая запоминает значения переменных из внешней области видимости, даже после завершения работы внешней функции.

Условия создания замыкания:

1. Есть вложенная функция.
2. Вложенная функция ссылается на переменную из внешней функции.
3. Внешняя функция возвращает вложенную функцию.

```python
def make_counter():
    count = 0

    def counter():  # Замыкание
        nonlocal count  # Разрешаем изменение count
        count += 1
        return count

    return counter  # Возвращаем функцию, а не её результат

my_counter = make_counter()
print(my_counter())  # 1
print(my_counter())  # 2 (сохраняет состояние count)
```

3. Специфика реализации в Python

1. **Доступ к переменным**:
    a. Чтение переменных внешней функции — работает "из коробки".
    b. Изменение требует `nonlocal` (для неизменяемых типов) или использования изменяемых объектов (например, списков).
2. **Клетки (Closure Cells)**:
   Python автоматически создает специальные объекты — *клетки*, чтобы сохранять значения переменных для замыканий.
3. **Просмотр замыканий**:
   Атрибут `__closure__` хранит кортеж клеток:

print(my_counter.__**closure**__[0].cell_contents) # Последнее значение count

9. Функции высшего порядка и декораторы в Python.

1. Функции высшего порядка (Higher-Order Functions, HOF)

Определение: Функции, которые:

- Принимают другие функции как аргументы, **или**
- Возвращают функции как результат.

```
def apply(func, x):
    return func(x)

def square(n):
    return n ** 2

print(apply(square, 5))  # 25
```

2. Декораторы (Decorators)

Определение: Функции, которые модифицируют поведение других функций.

Синтаксис: Используется символ @.

10. Концепция map/filter/reduce. Реализация map/filter/reduce в Python и пример их использования.

1. Общая концепция

Функции map, filter и reduce — это базовые инструменты функционального программирования для обработки коллекций:

`map – преобразует каждый элемент коллекции.`

`filter – выбирает элементы по условию.`

`reduce – агрегирует элементы в одно значение.`

2. Реализация в Python

map(func, iterable)

Применяет функцию func к каждому элементу iterable (списку, кортежу и т.д.). Пример: Удвоение чисел в списке.

```
numbers = [1, 2, 3]
doubled = map(lambda x: x * 2, numbers)
print(list(doubled))  # [2, 4, 6]
```

filter(func, iterable)

Оставляет только те элементы, для которых func возвращает True. Пример: Выбор чётных чисел.

```
numbers = [1, 2, 3, 4]
even = filter(lambda x: x % 2 == 0, numbers)
print(list(even))  # [2, 4]
```

reduce(func, iterable)

Последовательно применяет func к элементам, сводя коллекцию к одному значению. Требует импорта из functools. Пример: Сумма чисел.

```
from functools import reduce
numbers = [1, 2, 3, 4]
sum_all = reduce(lambda a, b: a + b, numbers)
print(sum_all)  # 10
```

11. Итераторы в Python: встроенные итераторы, создание собственных итераторов, типичные способы обхода итераторов и принцип их работы. Встроенные функции для работы с итераторами и возможности модуля itertools.

Фото 12

12. Функции генераторы и выражения генераторы: создание и применение в Python.

13. Специфика массивов, как структур данных. Динамические массивы — специфика работы, сложность операций. Специфика работы с array в Python.

1. Специфика массивов

Массив — упорядоченная коллекция элементов одного типа, хранящихся в непрерывной области памяти.

Ключевые свойства:

- **Фиксированный тип** элементов (в классических массивах)
- **Индексный доступ** за O(1)
- **Непрерывность памяти** — обеспечивает кэш-эффективность

Ограничения:

- Статический размер (в базовой реализации)
- Неудобство вставки/удаления (требуют сдвига элементов)

2. Динамические массивы

Принцип работы:

1. Изначально выделяется небольшой буфер (напр., на 4 элемента)
2. При заполнении массив увеличивается в **N раз** (в Python — ~1.125)
3. Старые элементы копируются в новую область памяти

14. Абстрактная структура данных стек и очередь: базовые и расширенные операции, их сложность.

1. Стек (LIFO - Last In First Out)

Принцип работы: Последний добавленный элемент извлекается первым (как стопка тарелок).

Базовые операции:

Операция	Описание	Сложность	Реализация в Python
push(x)	Добавление элемента на вершину	O(1)	`stack.append(x)`
pop()	Удаление элемента с вершины	O(1)	`stack.pop()`
peek()	Просмотр вершины стека	O(1)	`stack[-1]`
isEmpty()	Проверка на пустоту	O(1)	`not stack`

2. Очередь (FIFO - First In First Out)

Принцип работы: Первый добавленный элемент извлекается первым (как очередь в магазине).

Базовые операции:

Операция	Описание	Сложность	Реализация в Python
enqueue()	Добавление элемента в конец	O(1)	`queue.append(x)`
dequeue()	Удаление элемента из начала	O(1)	`queue.popleft()`
peek()	Просмотр первого элемента	O(1)	`queue.peek()`
Is_empty()	Проверка на пустоту	O(1)	`queue.is_empty()`

15. Специфика реализации и скорости основных операций в очереди на базе массива и связанного списка.

16. Связанные списки: однонаправленные и двунаправленные — принцип реализации. Сравнение скорости выполнения основных операций в связанных списках и в динамическом массиве.

17. Алгоритм обменной сортировки, сложность сортировки и возможности по ее улучшению.

Базовый алгоритм (Bubble Sort)

Принцип:

- Последовательно сравниваются пары соседних элементов.
- Если порядок неправильный, элементы меняются местами.
- Процесс повторяется, пока массив не будет отсортирован.

Хотя сортировка пузырьком проста в реализации и легко поддаётся визуализации, она обладает низкой эффективностью при работе с большими объёмами данных. **Её худшая и средняя временная сложность составляет *O*(*n*^2) (O(n) в лучшем случае)**, что делает её непрактичной в реальных проектах при значительных объёмах информации.

Однако в случаях, когда массив почти отсортирован, данный алгоритм может завершиться быстрее — особенно в модифицированных версиях, где предусмотрена проверка наличия обменов.

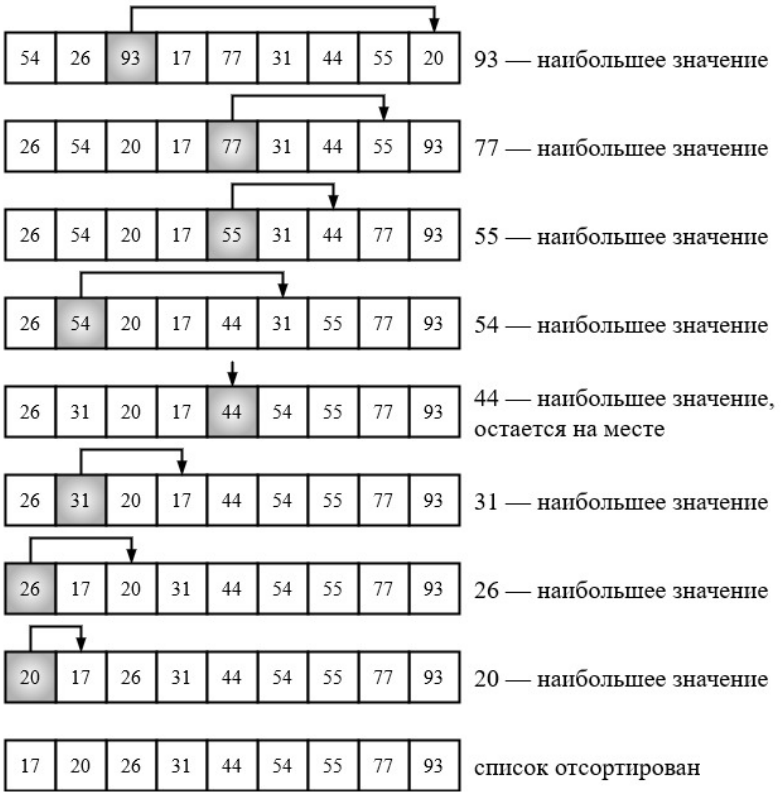
Пример стандартного алгоритма сортировки:

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

Шейкерная сортировка, также называемая *двунаправленной пузырьковой сортировкой* или *cocktail sort*, представляет собой модификацию классического пузырькового алгоритма, призванную повысить его эффективность. Алгоритм реализует попеременные проходы слева направо и справа налево.

Сортировка расчёской (*Comb Sort*) — усовершенствованный вариант пузырьковой сортировки, призванный устранить неэффективность при обработке удалённых друг от друга элементов, находящихся в неправильном порядке. На первых этапах используются крупные интервалы между сравниваемыми элементами, что позволяет быстро устранить грубые нарушения порядка и переместить большие значения ближе к концу массива. По мере выполнения алгоритма величина промежутка постепенно уменьшается до минимального значения, равного единице. На завершающих этапах происходит окончательная доупорядоченность массива, аналогичная последним проходам пузырьковой сортировки.

18. Алгоритм сортировки выбором, сложность сортировки и возможности по ее улучшению.



19. Алгоритм сортировки вставками, его сложность. Алгоритм быстрого поиска в отсортированном массиве. Сложность поиска в отсортированном и не отсортированном массиве.

20. Алгоритм сортировки Шелла, сложность сортировки и возможности по ее улучшению.

21. Алгоритм быстрой сортировки, сложность сортировки и возможности по ее улучшению.

22. Алгоритм сортировки слиянием, сложность сортировки.

23. Реализация двоичных деревьев в виде связанных объектов. Различные реализации рекурсивного обхода двоичных деревьев.

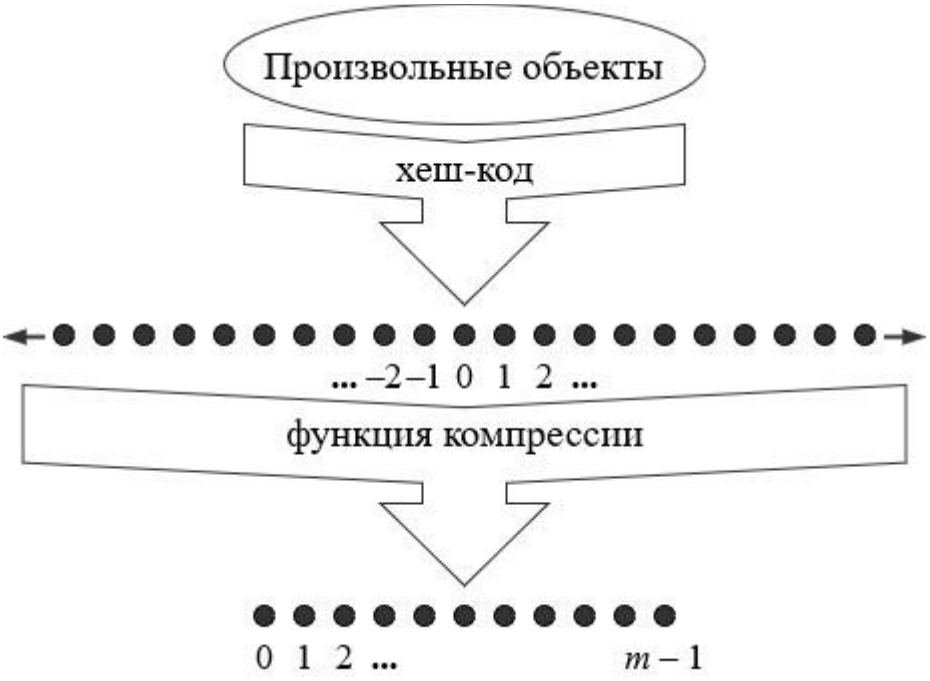
24. Двоичное дерево поиска — принципы реализации и логика реализации основных операций.

25. Двоичная куча — принципы реализации и логика реализации основных операций.

26. Абстрактный тип данных — ассоциативный массив и принцип его реализации на основе хэш-таблиц и хэш-функций.

27. Общая схема построения хэш-функции и возможная роль в этой схеме хэш-функции multiply-add-and-divide. Принцип работы хэш-функции multiply-add-and-divide.

Хеш-функция строится как композиция двух этапов. Эта общая схема представлена на рис. 3.



1 этап: преобразование ключа в целое число. Основное требование к хеш-коду — согласованность: для одного и того же ключа всегда должен возвращаться один и тот же хеш-код.

При работе со строками, графами или другими объектами, хеш-код вычисляется, извлекая числовое представление из структурированных данных.

2 этап: хеш-код преобразуется в значение из диапазона [0, m−1], где m — размер хеш-таблицы (хеш-значение можно использовать как индекс массива). Функция компрессии

должна быть детерминированной и равномерно распределять значения по диапазону, чтобы минимизировать вероятность коллизий.

$$h(k) = k \bmod m,$$ - метод деления

Метод MAD (Multiply–Add–and–Divide) представляет собой параметрическую хеш-функцию, задаваемую формулой

$$h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m,$$

где k — ключ, p — большое простое число, превышающее возможные значения ключей, m — это число ячеек в хеш-таблице, а параметры $a \in \{1, 2, \ldots, p-1\}$ и $b \in \{0, 1, \ldots, p-1\}$ выбираются случайным образом.

Преимущества: возможность использовать не только простые значения m (лучше метода деления, т. к. там выбор модуля существенно влияет на равномерность распределения); число различных хеш-функций в семействе составляет (p−1)· p.

Метод MAD широко применяется в контексте универсального хеширования — подхода, при котором хеш-функция выбирается случайным образом из заданного семейства H. Такое семейство считается универсальным, если для любых двух различных ключей x≠y вероятность коллизии не превышает 1/m.

Теоретически доказано, что если параметры a и b в методе MAD выбираются равномерно и независимо, то семейство хеш-функций будет универсальным. Это делает MAD надёжным и теоретически обоснованным выбором при построении хеш-таблиц с гарантированными свойствами распределения.